Contents

1 Course 1: Neural Networks and Deep Learning 1

1.1 Module 1 - Introduction to Deep Learning . . . 2

1.1.1 Neural Networks . . . 2

1.1.1.1 House Price Prediction . . . 2

1.1.1.2 Neural Network Implementation of House Price Prediction . . . 2

1.1.2 Supervised Learning . . . 2

1.1.3 Why is Deep Learning Taking Off? . . . 3

1.1.3.1 Scale Drives Deep Learning Progress . . . 3

1.2 Module 2 - Neural Networks Basics . . . 5

1.2.1 Logistic Regression . . . 5

1.2.1.1 Notation . . . 5

1.2.1.2 Cost Function . . . 6

1.2.1.3 Explanation of Cost Function . . . 6

1.2.1.4 Gradient Descent . . . 7

1.2.1.5 Forward Propagation . . . 7

1.2.1.6 Backpropagation . . . 7

1.2.2 Vectorizing Logistic Regression . . . 8

1.2.2.1 Forward Propagation . . . 8

1.2.2.2 Backpropagation . . . 9

1.2.2.3 Gradient Descent . . . 9

1.2.3 Broadcasting in `NumPy` . . . 9

1.3 Module 3 - Shallow Neural Network . . . 11
1.3.1 Forward Propagation . . . 11
1.3.1.1 Single Training Example . . . 11
1.3.1.2 Vectorization Across Multiple Training Examples . . . 12
1.3.2 Activation Function . . . 14
1.3.2.1 Why Non-Linear Activation Function? . . . 15
1.3.3 Backpropagation . . . 15
1.3.3.1 Vectorization Across Multiple Training Examples . . . 15
1.3.4 Random Initialization . . . 16
1.4 Module 4 - Deep Neural Networks . . . 18
1.4.1 Forward Propagation . . . 18
1.4.1.1 Single Training Example . . . 18
1.4.1.2 Vectorization Across Multiple Training Examples . . . 19
1.4.2 Why Deep Representations? . . . 20
1.4.2.1 Intuition About Deep Representation . . . 20
1.4.2.2 Circuit Theory and Deep Learning . . . 20
1.4.3 Backpropagation . . . 21
1.4.3.1 Single Training Example . . . 21
1.4.3.2 Vectorization Across Multiple Training Examples . . . 22
1.4.4 Gradient Descent . . . 24
1.4.5 Parameters vs Hyperparameters . . . 24

2 Course 2: Improving Deep Neural Networks 25

2.1 Module 1 - Practical Aspects of Deep Learning . . . 26
2.1.1 Setting Up Machine Learning Application . . . 26
2.1.1.1 Train/Dev/Test Sets . . . 26
2.1.1.2 Bias/Variance . . . 27
2.1.2 Regularizing Neural Network . . . 28

2.1.2.1 L_2 Regularization 28
2.1.2.2 Dropout Regularization 30
2.1.2.3 Other Regularization Methods 31
2.1.3 Setting Up Optimization Problem 32
2.1.3.1 Inputs Normalization 32
2.1.3.2 Vanishing/Exploding Gradients 33
2.1.3.3 Weight Initialization 33
2.1.3.4 Gradient Checking 34
2.2 Module 2 - Optimization Algorithms 36
2.2.1 Mini-batch Gradient Descent 36
2.2.2 Momentum 38
2.2.2.1 Exponentially Weighted Averages 38
2.2.2.2 Gradient Descent with Momentum 39
2.2.3 RMSprop 40
2.2.4 Adam 40
2.2.5 Learning Rate Decay 41
2.2.6 Problem of Local Optima 42
2.3 Module 3 - Hyperparameter Tuning, Batch Normalization and Programming Frameworks 43
2.3.1 Hyperparameter Tuning 43
2.3.1.1 How to Organize Hyperparameter Search Process 45
2.3.2 Batch Normalization 45
2.3.2.1 Deep NN Mini-Batch Gradient Descent with Batch Normalization 46
2.3.2.2 Why Batch Norm Works? 46
2.3.2.3 Batch Norm at Test Time 47
2.3.3 Multi-class Classification 47
2.3.3.1 Softmax vs Sigmoid Function 48
2.3.3.2 Softmax Regression 49
2.3.3.3 L-layer Neural Networks for Multi-Class Classification 49

2.3.4 Introduction to Programming Frameworks 52

3 Course 3: Structuring Machine Learning Projects 55

3.1 Module 1 - ML Strategy 56

3.1.1 Introduction to ML Strategy 56

3.1.1.1 Why ML Strategy? 56

3.1.1.2 Orthogonalization 56

3.1.2 Setting Up Goals 57

3.1.2.1 Single Number Evaluation Metric 57

3.1.2.2 Satisficing and Optimizing Metric 58

3.1.2.3 Train/Dev/Test Distributions 59

3.1.2.4 Size of Dev and Test Sets 59

3.1.2.5 When to Change Dev/Test Sets and Metrics? 60

3.1.3 Comparing to Human-level Performance 61

3.1.3.1 Why Human-level Performance? 61

3.1.3.2 Avoidable Bias 61

3.1.3.3 Understanding Human-level Performance 62

3.1.3.4 Surpassing Human-level Performance 62

3.1.3.5 Improving Model Performance 63

3.1.4 Case Study: Bird Recognition in the City of Peacetopia 64

3.1.4.1 Question 1 64

3.1.4.2 Question 2 64

3.1.4.3 Question 3 65

3.1.4.4 Question 4 65

3.1.4.5 Question 5 65

3.1.4.6 Question 6 66

3.1.4.7 Question 7 66

3.1.4.8 Question 8 67

3.1.4.9 Question 9 67

3.1.4.10 Question 10 . . . 68
3.1.4.11 Question 11 . . . 68
3.1.4.12 Question 12 . . . 68
3.1.4.13 Question 13 . . . 69
3.1.4.14 Question 14 . . . 69
3.1.4.15 Question 15 . . . 70

3.2 Module 2 - ML Strategy . . . 71

3.2.1 Error Analysis . . . 71
3.2.1.1 Carrying Out Error analysis . . . 71
3.2.1.2 Cleaning Up Incorrectly Labeled Data . . . 71
3.2.1.3 Build First System Quickly, Then Iterate . . . 73

3.2.2 Mismatch Training and Dev/Test Set . . . 73
3.2.2.1 Training and Testing on Different Distributions . . . 73
3.2.2.2 Bias and Variance with Mismatched Data Distributions . . . 74
3.2.2.3 Addressing Data Mismatch . . . 75

3.2.3 Learning from Multiple Tasks . . . 76
3.2.3.1 Transfer Learning . . . 76
3.2.3.2 Multi-task Learning . . . 76

3.2.4 End-to-end Deep Learning . . . 77

4 Course 4: Convolutional Neural Networks 79

4.1 Module 1 - Foundations of Convolutional Neural Networks . . . 80

4.1.1 Convolutional Neural Networks (CNN/ConvNet) . . . 80
4.1.1.1 Computer Vision . . . 80
4.1.1.2 Convolution . . . 80
4.1.1.3 Padding . . . 80
4.1.1.4 Strided . . . 81
4.1.1.5 Convolutions Over Volume . . . 81
4.1.1.6 One Convolution Layer . . . 82

4.1.1.7 Pooling Layers . . . 83
4.1.1.8 CNN Example . . . 84
4.1.1.9 Why Convolutions? . . . 85

4.2 Module 2 - Deep Convolutional Models: Cases Studies . . . 86
4.2.1 Case Studies . . . 86
4.2.1.1 Classic Networks . . . 86
4.2.1.2 ResNet . . . 88
4.2.1.3 Why ResNets Work . . . 90
4.2.1.4 Inception Network Motivation . . . 91
4.2.1.5 Inception Network . . . 91
4.2.1.6 MobileNet . . . 94
4.2.1.7 MobileNet Architecture . . . 94
4.2.1.8 EfficientNet . . . 94
4.2.2 Practical Advice for Using ConvNets . . . 94
4.2.2.1 Using Open-Source Implementation . . . 94
4.2.2.2 Transfer Learning . . . 95
4.2.2.3 Data Augmentation . . . 95
4.2.2.4 State of Computer Vision . . . 95

4.3 Module 3 - Object Detection . . . 96
4.3.1 Detection Algorithms . . . 96
4.3.1.1 Object Localization . . . 96
4.3.1.2 Landmark Detection . . . 96
4.3.1.3 Object Detection . . . 96
4.3.1.4 Convolutional Implementation of Sliding Windows . . . 96
4.3.1.5 Bounding Box Predictions . . . 96
4.3.1.6 Intersection Over Union . . . 96
4.3.1.7 Non-max Suppression . . . 96
4.3.1.8 Anchor Boxes . . . 96
4.3.1.9 YOLO Algorithm . . . 96

4.3.1.10 Semantic Segmentation with U-Net . . . 96
4.3.1.11 Transpose Convolutions . . . 96
4.3.1.12 U-Net Architecture Intuition . . . 96
4.3.1.13 U-Net Architecture . . . 96

4.4 Module 4 - Special Applications: Face Recognition & Neural Style Transfer . . . 97

4.4.1 Face Recognition . . . 97
4.4.1.1 What is Face Recognition? . . . 97
4.4.1.2 One Shot Learning . . . 97
4.4.1.3 Siamese Network . . . 97
4.4.1.4 Triplet Loss . . . 97
4.4.1.5 Face Verification and Binary Classification . . . 97

4.4.2 Neural Style Transfer . . . 97
4.4.2.1 What is Neural Style Transfer? . . . 97
4.4.2.2 What are deep ConvNets learning? . . . 97
4.4.2.3 Cost Function . . . 97
4.4.2.4 Content Cost Function . . . 97
4.4.2.5 Style Cost Function . . . 97
4.4.2.6 1D and 3D Generalizations . . . 97

5 Course 5: Sequence Models 99

5.1 Module 1 - Recurrent Neural Networks (RNNs) . . . 100

5.1.1 Why Sequence Models? . . . 100

5.1.2 Notation . . . 100

5.1.3 RNN Model . . . 101
5.1.3.1 Forward Propagation . . . 102
5.1.3.2 Backpropagation Through Time . . . 102
5.1.3.3 Different Types of RNNs . . . 102

5.1.4 Language Model . . . 105
5.1.4.1 Language Modeling with RNNs . . . 106

5.1.4.2 Sampling Novel Sequences . . . 108
5.1.4.3 Types of Language Models . . . 108
5.1.5 Vanishing Gradients with RNNs . . . 108
5.1.5.1 Gated Recurrent Unit (GRU) . . . 109
5.1.5.2 Long Short Term Memory (LSTM) . . . 110
5.1.6 Bidirectional RNN (BRNN) . . . 112
5.1.7 Deep RNN . . . 114
5.2 Module 2 - Natural Language Processing & Word Embeddings . . . 115
5.2.1 Introduction to Word Embeddings . . . 115
5.2.1.1 Word Representation . . . 115
5.2.1.2 Using Word Embeddings . . . 115
5.2.1.3 Properties Word Embeddings . . . 116
5.2.1.4 Embedding Matrix . . . 117
5.2.2 Learning Word Embeddings: Word2vec & GloVe . . . 117
5.2.2.1 Learning Word Embeddings . . . 118
5.2.2.2 Word2vec . . . 118
5.2.2.3 Negative Sampling . . . 118
5.2.2.4 GloVe Word Vectors . . . 118
5.2.3 Applications Using Word Embeddings . . . 118
5.2.3.1 Sentiment Classification . . . 118
5.2.3.2 Debiasing Word Embeddings . . . 118
5.3 Module 3 - Sequence Models & Attention Mechanism . . . 120
5.3.1 Various Sequence To Sequence Architectures . . . 120
5.3.1.1 Basic Models . . . 120
5.3.1.2 Beam Search . . . 123
5.3.1.3 Attention Model . . . 125
5.3.2 Speech Recognition - Audio Data . . . 126
5.3.2.1 Speech Recognition . . . 126

5.4 Module 4 - Transformer Network 128
5.4.1 Transformer Network Intuition 128
5.4.2 Self-Attention 128
5.4.3 Multi-Head Attention 129
5.4.4 Transformer Network 130

6 Supplementary Information **135**

6.1 Material 136

Chapter 1

Course 1: Neural Networks and Deep Learning

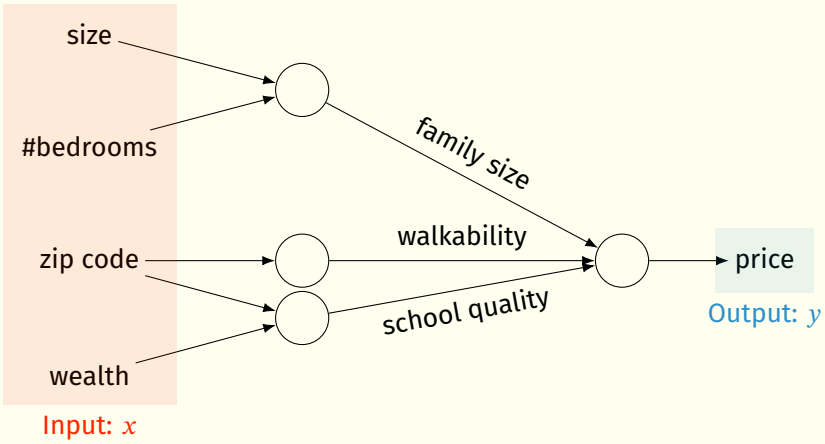
This chapter explains how to implement a neural network.

1.1 Module 1 - Introduction to Deep Learning

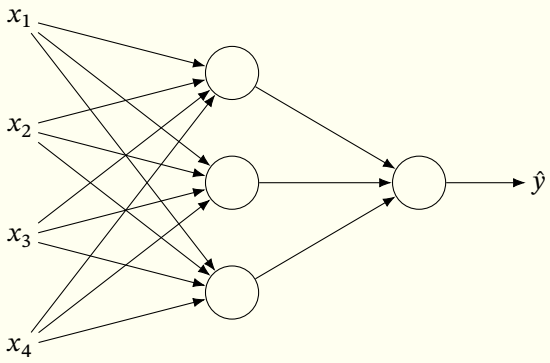
AI is the new electricity.

1.1.1 Neural Networks

1.1.1.1 House Price Prediction



1.1.1.2 Neural Network Implementation of House Price Prediction



1.1.2 Supervised Learning

Supervised learning is the process of learning a function that maps input data x to corresponding output y.

Neural networks have transformed supervised learning by automatically learning hidden layers from training data, using only the input x and output y for each example.

Most of the economic value by far generated by neural networks comes from supervised learning.

Types of neural networks:

- Standard NN
- Convolutional NN (CNN): for image
- Recurrent NN (RNN): for sequence data such as audio, language

Types of data:

- Structured data: each feature is well-defined
- Unstructured data: audio, image, text

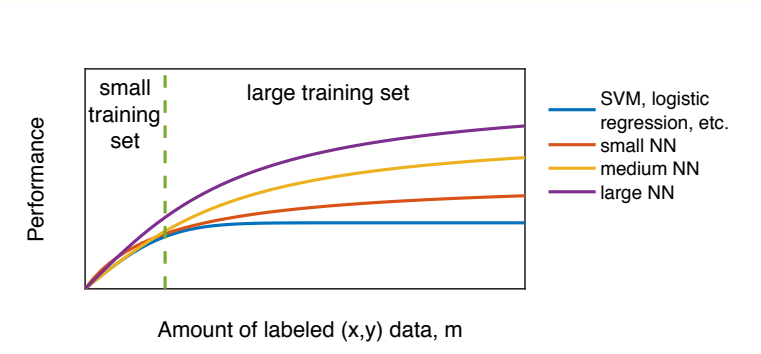
Historically, it has been much harder for computers to interpret unstructured data compared to structured data. However, with neural networks, computers are now much better at understanding unstructured data.

A lot of short term economic value generated by neural networks still comes from applications on structured data. Many of techniques in this course will apply to both structured and unstructured data.

1.1.3 Why is Deep Learning Taking Off?

1.1.3.1 Scale Drives Deep Learning Progress

Achieving high performance requires training a sufficiently large neural networks on a huge amount of data. Consequently, scale (network size and dataset size) has been a key driver of deep learning progress.



In the regime of small training sets, performance is less well-defined, and traditional methods such as SVMs can sometimes perform better than neural networks.

Training a neural network is highly iterative and usually involves repeating the following cylce many times:

- Propose a neural network architecture.

- Implement the idea in code.
- Run experiments and evaluate the network's performance.

Three main factors have driven progress in deep learning:

- Data: increase the amount and quality of training data.
- Computation: advances in CPU and GPU hardware enable larger models and faster training.

  Training a neural network is highly iterative. Faster computation speeds up training in each iteration, which significantly reduces the time required for each cycle.
- Algorithms: more efficient algorithms speed up computation.

  For example, replacing the sigmoid activations with a ReLU can significantly accelerate gradient descent, since sigmoid gradients can become very small, making parameter updates become very small and learning slows down considerably.

1.2 Module 2 - Neural Networks Basics

1.2.1 Logistic Regression

Logistic regression is an algorithm for binary classification.

The logistic regression can be regarded as a neural network with one layer (no hidden layer) and a single value output.

1.2.1.1 Notation

- A single training example (x, y), with

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_x} \end{bmatrix} \in \mathbb{R}^{n_x \times 1}, \quad y \in \{0, 1\}.$$

- A set of m training examples $\left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m} = \left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}$

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_x}^{(1)} & x_{n_x}^{(2)} & \cdots & x_{n_x}^{(m)} \end{bmatrix} \in \mathbb{R}^{n_x \times m}$$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}$$

  - m_{train} (or m): number of training examples
  - m_{test}: number of test examples

- w and b: parameters (model to be learned), with

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix} \in \mathbb{R}^{n_x \times 1} \quad b \in \mathbb{R}$$

Let $\hat{y}$ be the predicted probability of $y = 1$ given x, i.e., $P(y = 1 \mid x)$. $\hat{y}$ is defined as

$$\hat{y} = P(y = 1 \mid x) = \sigma(w^T x + b)$$
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Here, sigmoid function $\sigma(z)$ is applied to ensure that the probability is valid, i.e., $0 \leq \hat{y} \leq 1$.

1.2.1.2 Cost Function

Given a set of m training examples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, the goal of a logistic regression model is to find optimal parameters w and b such that the predicted probabilities $\hat{y}^{(i)}$ closely match the true labels $y^{(i)}$ for all $i = 1, \ldots, m$. To quantify the prediction error, the cost function $J(w, b)$ is defined. The optimal parameters w and b are then obtained by minimizing $J(w, b)$.

Given a single training example (x, y), the loss (error) function is defined as

$$\mathcal{L}(\hat{y}, y) = -y \ln \hat{y} - (1 - y) \ln(1 - \hat{y})$$

This loss function measures how close the predicted probability $\hat{y}$ is to the true label y for a single example, i.e., $\hat{y} \approx y$. One might consider using the squared error $\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$. However, this choice is not good for logistic regression because it leads to a non-convex optimization problem, which may result in multiple local optima and prevent gradient descent from finding the global optimum.

The loss function measures the error for a single example, while the cost function is the average loss over the entire training set. Given a set of m training examples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, the cost function is defined as,

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \ln \hat{y}^{(i)} + (1 - y^{(i)}) \ln (1 - \hat{y}^{(i)}) \right)$$

1.2.1.3 Explanation of Cost Function

Define

$$p(y \mid x) = \hat{y}^{y} \cdot (1 - \hat{y})^{1-y}$$

Thus,

$$p(y = 1 \mid x) = \hat{y}$$
$$p(y = 0 \mid x) = 1 - \hat{y}$$

The loss function becomes

$$\mathcal{L}(\hat{y}, y) = -y \ln \hat{y} - (1 - y) \cdot \ln(1 - \hat{y}) = -\ln p(y \mid x)$$

The cost function becomes

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\big(\hat{y}^{(i)}, y^{(i)}\big) = -\frac{1}{m}\sum_{i=1}^{m}\ln p\big(y^{(i)} \mid x^{(i)}\big) = -\frac{1}{m}\ln\left(\prod_{i=1}^{m} p\big(y^{(i)} \mid x^{(i)}\big)\right)$$

1.2.1.4 Gradient Descent

The gradient descent algorithm for logistic regression is

$$\begin{aligned} w &:= w - \alpha\,\frac{\partial J(w,b)}{\partial w} = w - \alpha\,dw \\ b &:= b - \alpha\,\frac{\partial J(w,b)}{\partial b} = b - \alpha\,db \end{aligned}$$

Here, α is called the learning rate.

1.2.1.5 Forward Propagation

The forward propagation step is used to calculate the output.

- Single example:

$$\begin{aligned} z &= w^T x + b = \sum_{j=1}^{n_x} w_j\,x_j + b \\ \hat{y} &= \sigma(z) = \frac{1}{1+e^{-z}} \\ \mathcal{L}(\hat{y}, y) &= -y\ln\hat{y} - (1-y)\ln\big(1-\hat{y}\big) \end{aligned}$$

- m examples:

$$\begin{aligned} z^{(i)} &= w^T x^{(i)} + b \\ \hat{y}^{(i)} &= \sigma\big(z^{(i)}\big) \\ \mathcal{L}\big(\hat{y}^{(i)}, y^{(i)}\big) &= -y\ln\hat{y}^{(i)} - \big(1-y^{(i)}\big)\ln\big(1-\hat{y}^{(i)}\big) \\ J(w,b) &= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\big(\hat{y}^{(i)}, y^{(i)}\big) \end{aligned}$$

1.2.1.6 Backpropagation

The backpropagation step is used to calculate the gradients (dw and db), using the chain rule to propagate derivatives through the computational graph.

According to definition of $\mathcal{L}(\hat{y}, y)$,

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$$
$$\frac{d\hat{y}}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) = \hat{y}\left(1-\hat{y}\right)$$
$$\frac{\partial z}{\partial w_i} = x_j$$

By chain rule,

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{d\hat{y}}{dz} = \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}\right) \cdot \hat{y}\left(1-\hat{y}\right) = \hat{y}\left(1-y\right) - y\left(1-\hat{y}\right) = \hat{y} - y$$

By chain rule again, the gradient of the loss function is

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial w_i} = (\hat{y} - y)\ x_j, \quad j = 1, 2, \dots, n_x$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial b} = \hat{y} - y$$

Thus, the gradients of the cost function $J(w, b)$ is

$$dw_i = \frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}\left(a^{(i)}, y^{(i)}\right)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right) x_j^{(i)}$$
$$db = \frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}\left(a^{(i)}, y^{(i)}\right)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)$$

1.2.2 Vectorizing Logistic Regression

Avoid explicit for-loops whenever possible.

1.2.2.1 Forward Propagation

Denote

$$Z = \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} \in \mathbb{R}^{1\times m}, \quad \sigma(Z) = \begin{bmatrix} \sigma(z^{(1)}) & \sigma(z^{(2)}) \cdots & \sigma(z^{(m)}) \end{bmatrix} \in \mathbb{R}^{1\times m}$$
$$A = \hat{Y} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix} \in \mathbb{R}^{1\times m}, \quad lnA = \begin{bmatrix} \ln \hat{y}^{(1)} & \ln \hat{y}^{(2)} & \cdots & \ln \hat{y}^{(m)} \end{bmatrix} \in \mathbb{R}^{1\times m}$$
$$\ln\left(1_{1\times m} - A\right) = \begin{bmatrix} \ln\left(1-\hat{y}^{(1)}\right) & \ln\left(1-\hat{y}^{(e)}\right) & \cdots & \ln\left(1-\hat{y}^{(m)}\right) \end{bmatrix} \in \mathbb{R}^{1\times m}$$
$$b_{1\times m} = \begin{bmatrix} b & b & \cdots & b \end{bmatrix} \in \mathbb{R}^{1\times m}, \quad 1_{1\times m} = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^{1\times m}$$

Thus,

$$
\begin{aligned}
Z &= w^T X + b_{1\times m} \\
A &= \sigma(Z) \\
J(w,b) &= -\frac{1}{m}\left(\ln A \cdot Y^T + \ln\left(1_{1\times m} - A\right)\cdot\left(1_{1\times m} - Y\right)^T\right)
\end{aligned}
$$

1.2.2.2 Backpropagation

Denote

$$
dw = \begin{bmatrix} dw_1 \\ dw_2 \\ \vdots \\ dw_{n_x} \end{bmatrix} = \frac{\partial J(w,b)}{\partial w} \in \mathbb{R}^{n_x\times 1}, \quad db = \frac{\partial J(w,b)}{\partial b}, \quad 1_{m\times 1} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^{m\times 1}
$$

Thus,

$$
dw = \begin{bmatrix} \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)x_1^{(i)} \\ \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)x_2^{(i)} \\ \vdots \\ \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)x_{n_x}^{(i)} \end{bmatrix} = \frac{1}{m}\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_x}^{(1)} & x_{n_x}^{(2)} & \cdots & x_{n_x}^{(m)} \end{bmatrix}\cdot\begin{bmatrix} \hat{y}^{(1)} - y^{(1)} \\ \hat{y}^{(2)} - y^{(2)} \\ \vdots \\ \hat{y}^{(m)} - y^{(m)} \end{bmatrix} = \frac{1}{m}X\cdot(A-Y)^T
$$

$$
db = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right) = \frac{1}{m}\begin{bmatrix} \hat{y}^{(1)} - y^{(1)} & \hat{y}^{(2)} - y^{(2)} & \cdots & \hat{y}^{(m)} - y^{(m)} \end{bmatrix}\cdot\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \frac{1}{m}(A-Y)\cdot 1_{m\times 1}
$$

1.2.2.3 Gradient Descent

The gradient descent for logistic regression becomes

$$
\begin{aligned}
w &:= w - \alpha\, dw = w - \frac{\alpha}{m}X\cdot\left(A-Y\right)^T \\
b &:= b - \alpha\, db = b - \frac{\alpha}{m}\left(A-Y\right)\cdot 1_{m\times 1}
\end{aligned}
$$

1.2.3 Broadcasting in `NumPy`

In addition to vectorization (e.g., matrix multiplication `np.dot()` or `@`), broadcasting in `NumPy` allows element-wise operations on arrays of different shapes without explicit loops, which often makes com-

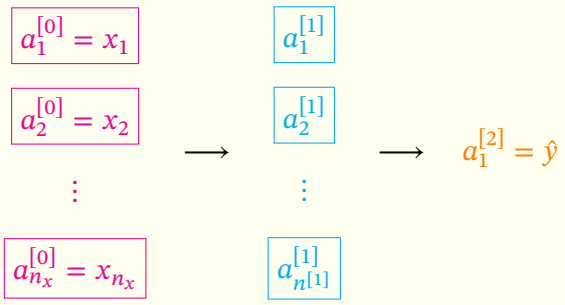
putations much faster.

According to broadcasting rules in `NumPy`, an array with shape (m, n) can be added to, subtracted from, or divided by an array with shape $(n,)$, $(1, n)$, $(m, 1)$, or by a scalar. Note that $(n,)$ works, whereas $(m,)$ does not work in general. This is because `NumPy` aligns array dimensions starting from the rightmost side when applying broadcasting.

Key points about broadcasting in `NumPy`:

- It automatically "stretches" dimensions of size 1 to match the other array, without copying any data.
- It applies only to element-wise operations. By contrast, `np.multiply()` performs strict element-wise multiplication without broadcasting and requires exactly matching shapes).
- By adjusting internal strides rather than duplicating arrays in memory, broadcasting avoids memory-intensive operations such as `np.tile`.

1.3 Module 3 - Shallow Neural Network



This module focuses on 1-hidden-layer NN with a scaler output (i.e., 0 or 1 for binary classification).

1.3.1 Forward Propagation

1.3.1.1 Single Training Example

Based on the flowchart above,

$$
\begin{aligned}
&a_1^{[0]} = x_1 \\
&a_2^{[0]} = x_2 \\
&\dots \\
&a_{n_x}^{[0]} = x_{n_x}
\end{aligned}
$$

$$
\begin{aligned}
&z_1^{[1]} = w_1^{[1]T} \cdot x + b_1^{[1]}, \quad a_1^{[1]} = g^{[1]}(z_1^{[1]}) \\
&z_2^{[1]} = w_2^{[1]T} \cdot x + b_2^{[1]}, \quad a_2^{[1]} = g^{[1]}(z_2^{[1]}) \\
&\dots \\
&z_{n^{[1]}}^{[1]} = w_{n^{[1]}}^{[1]T} \cdot x + b_{n^{[1]}}^{[1]}, \quad a_{n^{[1]}}^{[1]} = g^{[1]}(z_{n^{[1]}}^{[1]})
\end{aligned}
$$

$$
z_1^{[2]} = w_1^{[2]T} \cdot a^{[1]} + b_1^{[2]}, \quad a_1^{[2]} = g^{[2]}(z_1^{[2]}) = \hat{y}
$$

Here,

- $w_i^{[1]} \in \mathbb{R}^{n_x \times 1}$ for $i = 1, 2, \dots, n^{[1]}$, and $w_1^{[2]} \in \mathbb{R}^{n^{[1]} \times 1}$.
- $w_i^{[1]T}$ denotes $\left(w_i^{[1]}\right)^T$ for $i = 1, 2, \dots, n^{[1]}$, and $w_1^{[2]T}$ denotes $\left(w_1^{[2]}\right)^T$.
- Both $g^{[1]}(z)$ and $g^{[2]}(z)$ are called the activation function.

Denote

$$W^{[1]} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ \vdots \\ w_{n^{[1]}}^{[1]T} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times n_x}, \quad W^{[2]} = w_1^{[2]T} \in \mathbb{R}^{1 \times n^{[1]}}, \quad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_{n^{[1]}}^{[1]} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times 1}, \quad b^{[2]} = b_1^{[2]} \in \mathbb{R}$$

$$z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ \vdots \\ z_{n^{[1]}}^{[1]} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times 1}, \quad z^{[2]} = z_1^{[2]} \in \mathbb{R}, \quad g^{[1]}\big(z^{[1]}\big) = \begin{bmatrix} g^{[1]}\big(z_1^{[1]}\big) \\ g^{[1]}\big(z_2^{[1]}\big) \\ \vdots \\ g^{[1]}\big(z_{n^{[1]}}^{[1]}\big) \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times 1}, \quad g^{[2]}\big(z^{[2]}\big) = g^{[2]}\big(z_1^{[2]}\big) \in \mathbb{R}$$

$$a^{[0]} = \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ \vdots \\ a_{n_x}^{[0]} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_x} \end{bmatrix} = x \in \mathbb{R}^{n_x \times 1}, \quad a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_{n^{[1]}}^{[1]} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times 1}, \quad a^{[2]} = a_1^{[2]} = \hat{y} \in \mathbb{R}$$

Thus,

$$\begin{aligned} a^{[0]} &= x \\ z^{[1]} &= W^{[1]} \cdot a^{[0]} + b^{[1]}, \quad a^{[1]} = g^{[1]}\big(z^{[1]}\big) \\ z^{[2]} &= W^{[2]} \cdot a^{[1]} + b^{[2]}, \quad a^{[2]} = g^{[2]}\big(z^{[2]}\big) = \hat{y} \end{aligned}$$

1.3.1.2 Vectorization Across Multiple Training Examples

Given a set of m training examples $\big\{\big(x^{(i)}, y^{(i)}\big)\big\}_{i=1}^{m}$, the matrix is organized such that training examples are arranged horizontally and hidden units are arranged vertically.

$$\begin{aligned} a^{[0](i)} &= x^{(i)} \\ z^{[1](i)} &= W^{[1]} \cdot a^{[0](i)} + b^{[1]}, \quad a^{[1](i)} = g^{[1]}\big(z^{[1](i)}\big) \\ z^{[2](i)} &= W^{[2]} \cdot a^{[1](i)} + b^{[2]}, \quad a^{[2](i)} = g^{[2]}\big(z^{[2](i)}\big) = \hat{y}^{(i)} \end{aligned}$$

Denote

$$Z^{[1]} = \begin{bmatrix} z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \end{bmatrix} = \begin{bmatrix} z_1^{[1](1)} & z_1^{[1](2)} & \dots & z_1^{[1](m)} \\ z_2^{[1](1)} & z_2^{[1](2)} & \dots & z_2^{[1](m)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n^{[1]}}^{[1](1)} & z_{n^{[1]}}^{[1](2)} & \dots & z_{n^{[1]}}^{[1](m)} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}$$

$$Z^{[2]} = \begin{bmatrix} z^{[2](1)} & z^{[2](2)} & \dots & z^{[2](m)} \end{bmatrix} = \begin{bmatrix} z_1^{[2](1)} & z_1^{[2](2)} & \dots & z_1^{[2](m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}$$

$$g^{[1]}(Z^{[1]}) = \begin{bmatrix} g^{[1]}(z^{[1](1)}) & g^{[1]}(z^{[1](2)}) & \dots & g^{[1]}(z^{[1](m)}) \end{bmatrix}$$

$$= \begin{bmatrix} g^{[1]}(z_1^{[1](1)}) & g^{[1]}(z_1^{[1](2)}) & \dots & g^{[1]}(z_1^{[1](m)}) \\ g^{[1]}(z_2^{[1](1)}) & g^{[1]}(z_2^{[1](2)}) & \dots & g^{[1]}(z_2^{[1](m)}) \\ \vdots & \vdots & \ddots & \vdots \\ g^{[1]}(z_{n^{[1]}}^{[1](1)}) & g^{[1]}(z_{n^{[1]}}^{[1](2)}) & \dots & g^{[1]}(z_{n^{[1]}}^{[1](m)}) \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}$$

$$g^{[2]}(Z^{[2]}) = \begin{bmatrix} g^{[2]}(z^{[2](1)}) & g^{[2]}(z^{[2](2)}) & \dots & g^{[2]}(z^{[2](m)}) \end{bmatrix}$$

$$= \begin{bmatrix} g^{[2]}(z_1^{[2](1)}) & g^{[2]}(z_1^{[2](2)}) & \dots & g^{[2]}(z_1^{[2](m)}) \end{bmatrix} \in \mathbb{R}^{1 \times m}$$

$$A^{[0]} = \begin{bmatrix} a^{[0](1)} & a^{[0](2)} & \dots & a^{[0](m)} \end{bmatrix} = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix} = X \in \mathbb{R}^{n_x \times m}$$

$$A^{[1]} = \begin{bmatrix} a^{[1](1)} & a^{[1](2)} & \dots, a^{[1](m)} \end{bmatrix} = \begin{bmatrix} a_1^{[1](1)} & a_1^{[1](2)} & \dots & a_1^{[1](m)} \\ a_2^{[1](1)} & a_2^{[1](2)} & \dots & a_2^{[1](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^{[1]}}^{[1](1)} & a_{n^{[1]}}^{[1](2)} & \dots & a_{n^{[1]}}^{[1](m)} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}$$

$$A^{[2]} = \begin{bmatrix} a^{[2](1)} & a^{[2](2)} & \dots & a^{[2](m)} \end{bmatrix} = \begin{bmatrix} a_1^{[2](1)} & a_1^{[2](2)} & \dots & a_1^{[2](m)} \end{bmatrix}$$

$$= \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \dots & \hat{y}^{(m)} \end{bmatrix} = \hat{Y} \in \mathbb{R}^{1 \times m}$$

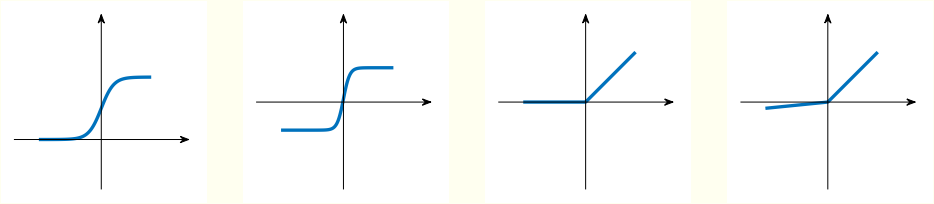
$$b_{1 \times m}^{[1]} = \begin{bmatrix} b^{[1]} & b^{[1]} & \dots & b^{[1]} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}$$

$$b_{1 \times m}^{[2]} = \begin{bmatrix} b^{[2]} & b^{[2]} & \dots & b^{[2]} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}$$

The forward propagation becomes

$$\begin{aligned} A^{[0]} &= X \\ Z^{[1]} &= W^{[1]} \cdot A^{[0]} + b_{1 \times m}^{[1]}, \quad A^{[1]} = g^{[1]}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]} \cdot A^{[1]} + b_{1 \times m}^{[2]}, \quad A^{[2]} = g^{[2]}(Z^{[2]}) = \hat{Y} \end{aligned}$$

1.3.2 Activation Function



Types of activation functions:

- Sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}, \quad g'(z) = \frac{dg(z)}{dz} = g(z)\,(1-g(z))$$

- tanh function

$$g(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}, \quad g'(z) = 1-g^{2}(z)$$

- ReLU (Rectified Linear Unit) function

$$g(z) = \max(z,0), \quad g'(z) = \begin{cases} 0, & z<0 \\ 1, & z\geq 0 \end{cases}$$

- Leaky ReLU function

$$g(z) = \max(0.01z, z), \quad g'(z) = \begin{cases} 0.01, & z<0 \\ 1, & z\geq 0 \end{cases}$$

The sigmoid function is one possible choice for an activation function, but other functions often perform better. For hidden layers, the tanh function is usually preferred over the sigmoid function because it outputs values between -1 and 1, which makes the activations more centered around zero and can help gradient descent converge faster. Mathematically, the tanh function is the shift version of the sigmoid function.

However, in the output layer for binary classification, the sigmoid function is typically used because the predicted output $\hat{y}$ should represent a probability between 0 and 1.

A drawback of both the sigmoid and tanh functions is that when z is very large or very small, their gradients become close to zero. This is known as the vanishing gradient problem, which can slow down gradient descent.

The ReLU activation function helps alleviate this issue because its gradient does not vanish for positive values of z. In practice, ReLU is commonly used for hidden layers. A variation called leaky ReLU can

sometimes perform slightly better by allowing a small nonzero gradient when $z < 0$.

In summary,

- Use the sigmoid function in the output layer for binary classification.
- The tanh function generally performs better than the sigmoid function for hidden layers.
- ReLU is the most commonly used activation function for hidden layers.
- Leaky ReLU is a useful alternative that can sometimes perform better than ReLU.

1.3.2.1 Why Non-Linear Activation Function?

For hidden layers of a NN, if a linear activation function $g(z) = z$ is used (or no activation function is used), the network simply outputs a linear function of the input, regardless of how many hidden layers it has. This is because the composition of linear functions is still a linear function. Therefore, stacking multiple layers with linear activation functions (or no activation) is equivalent to a single linear model (e.g., logistic regression for binary classification).

One situation where a linear activation function may be used is in the output layer for regression problems, where the target value can take any real number.

In other words, without nonlinear activation functions, adding hidden layers does not increase the model's expressive power.

1.3.3 Backpropagation

1.3.3.1 Vectorization Across Multiple Training Examples

Rewrite the cost function as,

$$J\big(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\big) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\big(\hat{y}^{(i)}, y^{(i)}\big) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\big(a^{[2](i)}, y^{(i)}\big)$$

Denote

$$dW^{[2]} = \frac{\partial J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right)}{\partial W^{[2]}} \in \mathbb{R}^{1\times n^{[1]}}, \quad db^{[2]} = \frac{\partial J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right)}{\partial b^{[2]}} \in \mathbb{R}$$

$$dW^{[1]} = \frac{\partial J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right)}{\partial W^{[1]}} \in \mathbb{R}^{n^{[1]}\times n_x}, \quad db^{[1]} = \frac{\partial J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right)}{\partial b^{[1]}} \in \mathbb{R}^{n^{[1]}\times 1}$$

$$dZ^{[2]} = \begin{bmatrix} \frac{\partial \mathcal{L}\left(a^{[2](1)}, y^{(1)}\right)}{\partial z^{[2](1)}} & \frac{\partial \mathcal{L}\left(a^{[2](2)}, y^{(2)}\right)}{\partial z^{[2](2)}} & \cdots & \frac{\partial \mathcal{L}\left(a^{[2](m)}, y^{(m)}\right)}{\partial z^{[2](m)}} \end{bmatrix} \in \mathbb{R}^{1\times m}$$

$$dZ^{[1]} = \begin{bmatrix} \frac{\partial \mathcal{L}\left(a^{[1](1)}, y^{(1)}\right)}{\partial z^{[1](1)}} & \frac{\partial \mathcal{L}\left(a^{[1](2)}, y^{(2)}\right)}{\partial z^{[1](2)}} & \cdots & \frac{\partial \mathcal{L}\left(a^{[1](m)}, y^{(m)}\right)}{\partial z^{[1](m)}} \end{bmatrix} \in \mathbb{R}^{n^{[1]}\times m}$$

$$g^{[1]'}(Z^{[1]}) = \begin{bmatrix} g^{[1]'}\left(z^{[1](1)}\right) & g^{[1]'}\left(z^{[1](2)}\right) & \cdots & g^{[1]'}\left(z^{[1](m)}\right) \end{bmatrix}$$

$$= \begin{bmatrix} g^{[1]'}\left(z_1^{[1](1)}\right) & g^{[1]'}\left(z_1^{[1](2)}\right) & \cdots & g^{[1]'}\left(z_1^{[1](m)}\right) \\ g^{[1]'}\left(z_2^{[1](1)}\right) & g^{[1]'}\left(z_2^{[1](2)}\right) & \cdots & g^{[1]'}\left(z_2^{[1](m)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ g^{[1]'}\left(z_{n^{[1]}}^{[1](1)}\right) & g^{[1]'}\left(z_{n^{[1]}}^{[1](2)}\right) & \cdots & g^{[1]'}\left(z_{n^{[1]}}^{[1](m)}\right) \end{bmatrix} \in \mathbb{R}^{n^{[1]}\times m}$$

When $g^{[2]}(z) = \sigma(z)$, the backpropagation is

$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} \cdot A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} dZ^{[2]} \cdot 1_{m\times 1}$$

$$dZ^{[1]} = W^{[2]T} \cdot dZ^{[2]} \odot g^{[1]'}\left(Z^{[1]}\right)$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} \cdot X^T$$
$$db^{[1]} = \frac{1}{m} dZ^{[1]} \cdot 1_{m\times 1}$$

1.3.4 Random Initialization

It's important to initialize the weights randomly rather than zero when training NNs.

- For logistic regression, initializing weights $w = 0$ is fine.
- For a NN, initializing all weights $W^{[l]} = 0$ prevents proper learning with gradient descent.
  - Explanation: If $W^{[1]} = 0$ and $b^{[1]} = 0$, then for the first hidden layer

$$a_1^{[1]} = a_2^{[1]} = \cdots = a_{n^{[1]}}^{[1]}$$
$$dz_1^{[1]} = dz_2^{[1]} = \cdots = dz_{n^{[1]}}^{[1]}$$

All hidden units in this layer are symmetric and compute the same function during every

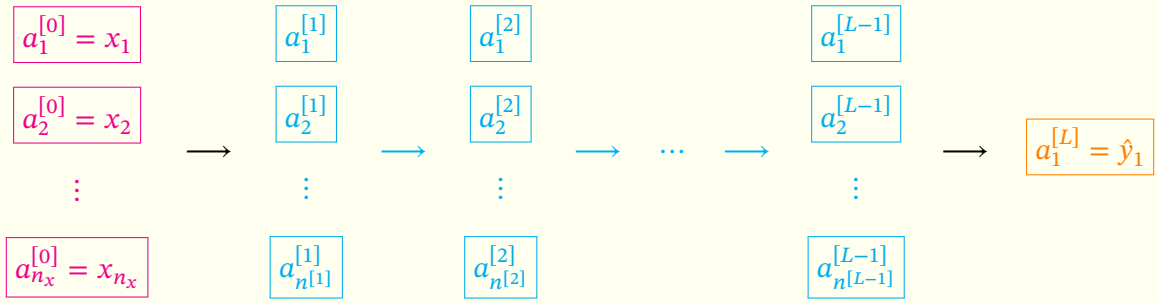
iteration. Gradient descent cannot break this symmetry, so the network cannot learn distinct features.

Thus, the weights $W^{[l]}$ must be randomly initialized, while it is to initialize biases $b^{[l]}$ to zero. Specifically,

```
W = np.random.randn(m, n) * 0.01
b = np.zeros((m, n))
```

Multiplying by a small constant (scaling factor, e.g., 0.01) ensures that weights start small, which helps prevent saturation of the sigmoid or tanh activations. Saturation results in very small gradients, which slows down learning. A scaling factor of 0.01 usually works well for shallow NNs. For very deep NNs, a different scaling factor may be needed.

1.4 Module 4 - Deep Neural Networks



This module focuses on $(L-1)$-hidden-layer NN (or L-layer NN) with a scaler output (i.e., $n^{[L]} = 1$).

1.4.1 Forward Propagation

Notation:

- L: Number of layers (hidden/output layers)
- $n^{[l]}$: number of unites in layer l
- $a^{[l]} = g^{[l]}(z^{[l]})$: activation in layer l
- $w^{[l]}$: weights for computing $z^{[l]}$
- $b^{[l]}$: bias for computing $z^{[l]}$
- $a^{[0]} = x$ and $a^{[L]} = \hat{y}$

Here, $l = 1, 2, \dots, L$.

1.4.1.1 Single Training Example

For $l = 1, 2, \dots, L$, denote

$$W^{[l]} = \begin{bmatrix} w_1^{[l]T} \\ w_2^{[l]T} \\ \vdots \\ w_{n^{[l]}}^{[l]T} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[l]} & w_{1,2}^{[l]} & \cdots & w_{1,n^{[l-1]}}^{[l]} \\ w_{2,1}^{[l]} & w_{2,2}^{[l]} & \cdots & w_{2,n^{[l-1]}}^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[l]},1}^{[l]} & w_{n^{[l]},2}^{[l]} & \cdots & w_{n^{[l]},n^{[l-1]}}^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}, \quad b^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times 1}$$

$$z^{[l]} = \begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{n^{[l]}}^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times 1}, \quad g^{[l]}(z^{[l]}) = \begin{bmatrix} g^{[l]}(z_1^{[l]}) \\ g^{[l]}(z_2^{[l]}) \\ \vdots \\ g^{[l]}(z_{n^{[l]}}^{[l]}) \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times 1}, \quad a^{[l]} = \begin{bmatrix} a_1^{[l]} \\ a_2^{[l]} \\ \vdots \\ a_{n^{[l]}}^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times 1}$$

Thus,

$$
\begin{aligned}
&a^{[0]} = x \\
&z^{[1]} = W^{[1]} \cdot a^{[0]} + b^{[1]}, \quad a^{[1]} = g^{[1]}\big(z^{[1]}\big) \\
&z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]}, \quad a^{[2]} = g^{[2]}\big(z^{[2]}\big) \\
&\vdots \\
&z^{[L]} = W^{[L]} \cdot a^{[L-1]} + b^{[L]}, \quad a^{[L]} = g^{[L]}\big(z^{[L]}\big) = \hat{y}
\end{aligned}
$$

1.4.1.2 Vectorization Across Multiple Training Examples

Denote

$$
Z^{[l]} = \begin{bmatrix} z^{[l](1)} & z^{[l](2)} & \dots & z^{[l](m)} \end{bmatrix} = \begin{bmatrix} z_1^{[l](1)} & z_1^{[l](2)} & \cdots & z_1^{[l](m)} \\ z_2^{[l](1)} & z_2^{[l](2)} & \cdots & z_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n^{[l]}}^{[l](1)} & z_{n^{[l]}}^{[l](2)} & \cdots & z_{n^{[l]}}^{[l](m)} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times m}, \quad l = 1, 2, \dots, L-1
$$

$$
Z^{[L]} = \begin{bmatrix} z^{[L](1)} & z^{[L](2)} & \dots & z^{[L](m)} \end{bmatrix} = \begin{bmatrix} z_1^{[L](1)} & z_1^{[L](2)} & \dots & z_1^{[L](m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}
$$

$$
\begin{aligned}
g^{[l]}\big(Z^{[l]}\big) &= \begin{bmatrix} g^{[l]}\big(z^{[l](1)}\big) & g^{[l]}\big(z^{[l](2)}\big) & \dots & g^{[l]}\big(z^{[1](m)}\big) \end{bmatrix} \\
&= \begin{bmatrix} g^{[l]}\big(z_1^{[l](1)}\big) & g^{[l]}\big(z_1^{[l](2)}\big) & \cdots & g^{[1]}\big(z_1^{[l](m)}\big) \\ g^{[l]}\big(z_2^{[l](1)}\big) & g^{[l]}\big(z_2^{[l](2)}\big) & \cdots & g^{[1]}\big(z_2^{[l](m)}\big) \\ \vdots & \vdots & \ddots & \vdots \\ g^{[l]}\big(z_{n^{[l]}}^{[l](1)}\big) & g^{[1]}\big(z_{n^{[l]}}^{[l](2)}\big) & \cdots & g^{[1]}\big(z_{n^{[l]}}^{[l](m)}\big) \end{bmatrix} \in \mathbb{R}^{n_x \times m}, \quad l = 1, 2, \dots, L-1
\end{aligned}
$$

$$
\begin{aligned}
g^{[L]}\big(Z^{[L]}\big) &= \begin{bmatrix} g^{[L]}\big(z^{[L](1)}\big) & g^{[L]}\big(z^{[L](2)}\big) & \dots & g^{[L]}\big(z^{[L](m)}\big) \end{bmatrix} \\
&= \begin{bmatrix} g^{[L]}\big(z_1^{[L](1)}\big) & g^{[L]}\big(z_1^{[L](2)}\big) & \dots & g^{[L]}\big(z_1^{[L](m)}\big) \end{bmatrix} \in \mathbb{R}^{1 \times m}
\end{aligned}
$$

$$
A^{[0]} = \begin{bmatrix} a^{[0](1)} & a^{[0](2)} & \cdots & a^{[0](m)} \end{bmatrix} = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix} = X \in \mathbb{R}^{n_x \times m}
$$

$$
A^{[l]} = \begin{bmatrix} a^{[l](1)} & a^{[l](2)} & \dots, a^{[l](m)} \end{bmatrix} = \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \cdots & a_1^{[l](m)} \\ a_2^{[l](1)} & a_2^{[l](2)} & \cdots & a_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^{[l]}}^{[l](1)} & a_{n^{[l]}}^{[l](2)} & \cdots & a_{n^{[l]}}^{[l](m)} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}, \quad l = 1, 2, \dots, L-1
$$

$$
\begin{aligned}
A^{[L]} &= \begin{bmatrix} a^{[L](1)} & a^{[L](2)} & \dots, a^{[L](m)} \end{bmatrix} = \begin{bmatrix} a_1^{[L](1)} & a_1^{[L](2)} & \cdots & a_1^{[L](m)} \end{bmatrix} \\
&= \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix} = \hat{Y} \in \mathbb{R}^{1 \times m}
\end{aligned}
$$

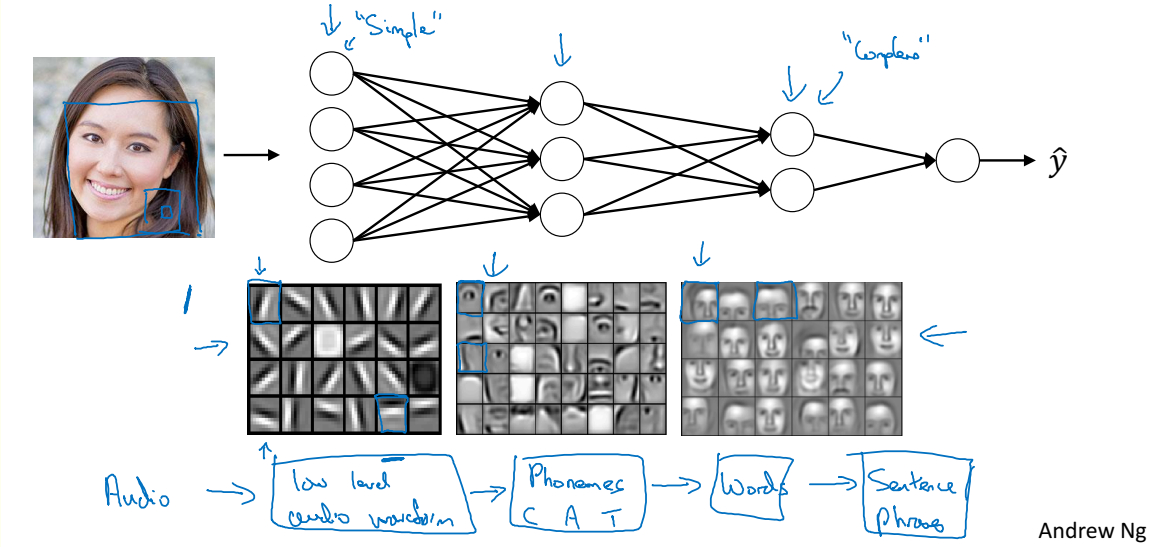
$$
b_{1 \times m}^{[l]} = \begin{bmatrix} b^{[l]} & b^{[l]} & \dots & b^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times m}
$$

The forward propagation becomes

$$
\begin{aligned}
&A^{[0]} = X \\
&Z^{[1]} = W^{[1]} \cdot A^{[0]} + b^{[1]}_{1\times m}, \quad A^{[1]} = g^{[1]}\left(Z^{[1]}\right) \\
&Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]}_{1\times m}, \quad A^{[2]} = g^{[2]}\left(Z^{[2]}\right) \\
&\vdots \\
&Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + b^{[L]}_{1\times m}, \quad A^{[L]} = g^{[L]}\left(Z^{[L]}\right) = \hat{Y}
\end{aligned}
$$

1.4.2 Why Deep Representations?

1.4.2.1 Intuition About Deep Representation



1.4.2.2 Circuit Theory and Deep Learning

Informally: There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

1.4.3 Backpropagation

1.4.3.1 Single Training Example

Rewrite the cost function as,

$$J\big(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}\big) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\big(\hat{y}^{(i)}, y^{(i)}\big) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\big(a^{[L](i)}, y^{(i)}\big)$$

For $l = 1, 2, \dots, L$, denote

$$dW^{[l]} = \frac{\partial J\big(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}\big)}{\partial W^{[l]}} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$$
$$db^{[l]} = \frac{\partial J\big(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}\big)}{\partial b^{[l]}} \in \mathbb{R}^{n^{[l]} \times 1}$$
$$da^{[l]} = \frac{\partial \mathcal{L}\big(a^{[l]}, y\big)}{\partial a^{[l]}} \in \mathbb{R}^{n^{[l]} \times 1}$$
$$dz^{[l]} = \frac{\partial \mathcal{L}\big(a^{[l]}, y\big)}{\partial z^{[l]}} \in \mathbb{R}^{n^{[l]} \times 1}$$

Thus,

$$dz^{[L]} = da^{[L]} \cdot g^{[L]'}\big(z^{[L]}\big)$$
$$dW^{[L]} = dz^{[L]} \cdot a^{[L-1]}$$
$$db^{[L]} = dz^{[L]}$$
$$da^{[L-1]} = W^{[L]T} \cdot dz^{[L]}$$

$$dz^{[L-1]} = da^{[L-1]} \cdot g^{[L-1]'}\big(z^{[L-1]}\big)$$
$$dW^{[L-1]} = dz^{[L-1]} \cdot a^{[L-2]}$$
$$db^{[L-1]} = dz^{[L-1]}$$
$$da^{[L-2]} = W^{[L-1]T} \cdot dz^{[L-1]}$$

$$\vdots$$

$$dz^{[1]} = da^{[1]} \cdot g^{[1]'}\big(z^{[1]}\big)$$
$$dW^{[1]} = dz^{[1]} \cdot a^{[0]} = dz^{[1]} \cdot x$$
$$db^{[1]} = dz^{[1]}$$

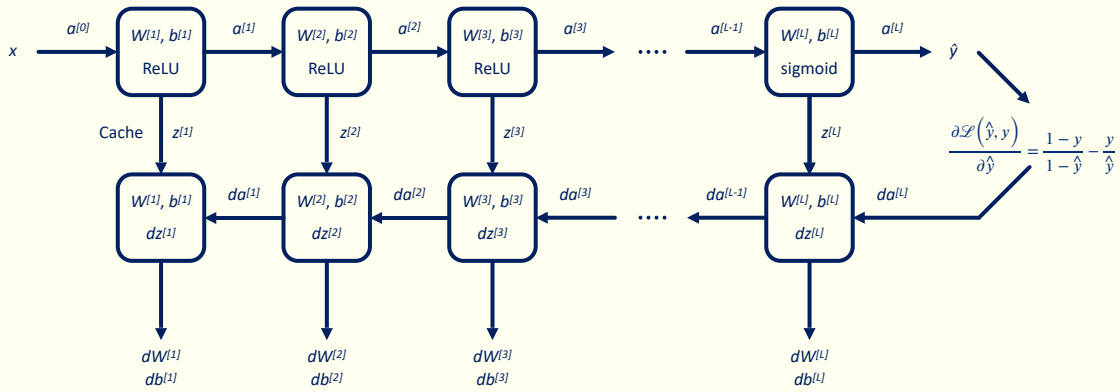
When activation of the output layer L is the sigmoid function, i.e., $g^{[L]}(z) = \sigma(z)$, by calculation,

$$g^{[L]'}(z^{[L]}) = g^{[L]}(z^{[L]}) \cdot \left(1 - g^{[L]}(z^{[L]})\right) = a^{[L]} \cdot \left(1 - a^{[L]}\right)$$
$$da^{[L]} = \frac{\partial \mathcal{L}(a^{[L]}, y)}{\partial z^{[L]}} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}} = \frac{1-y}{1-a^{[L]}} - \frac{y}{a^{[L]}}$$

Thus, by chain rule,

$$\begin{aligned} dz^{[L]} &= da^{[L]} \cdot g^{[L]'}\left(z^{[L]}\right) \\ &= \left(\frac{1-y}{1-a^{[L]}} - \frac{y}{a^{[L]}}\right) \cdot a^{[L]} \cdot \left(1 - a^{[L]}\right) \\ &= a^{[L]} - y \end{aligned}$$

A flowchart of the whole process:



1.4.3.2 Vectorization Across Multiple Training Examples

For $l = 1, 2, \cdots, L$, denote

$$dA^{[l]} = \left[da^{[l](1)}, da^{[l](2)} \quad \ldots \quad da^{[l](m)}\right] \in \mathbb{R}^{n^{[l]} \times m}$$
$$dZ^{[l]} = \left[dz^{[l](1)} \quad dz^{[l](2)} \quad \ldots \quad dz^{[l](m)}\right] \in \mathbb{R}^{n^{[l]} \times m}$$
$$g^{[l]'}(Z^{[l]}) = \left[g^{[l]'}(z^{[l](1)}) \quad g^{[l]'}(z^{[l](2)}) \quad \ldots \quad g^{[l]'}(z^{[l](m)})\right] \in \mathbb{R}^{n^{[l]} \times m}$$

Thus

$$
\begin{aligned}
dZ^{[L]} &= dA^{[L]} \odot g^{[L]'}\left(Z^{[L]}\right) \\
dW^{[L]} &= \frac{1}{m}\, dZ^{[L]} \cdot A^{[L-1]T} \\
db^{[L]} &= \frac{1}{m}\, dZ^{[L]} \cdot 1_{m\times 1} \\
dA^{[L-1]} &= W^{[L]T} \cdot dZ^{[L]}
\end{aligned}
$$

$$
\begin{aligned}
dZ^{[L-1]} &= dA^{[L-1]} \odot g^{[L-1]'}\left(Z^{[L-1]}\right) \\
dW^{[L-1]} &= \frac{1}{m}\, dZ^{[L-1]} \cdot A^{[L-2]T} \\
db^{[L-1]} &= \frac{1}{m}\, dZ^{[L-1]} \cdot 1_{m\times 1} \\
dA^{[L-2]} &= W^{[L-2]T} \cdot dZ^{[L-2]}
\end{aligned}
$$

$$
\vdots
$$

$$
\begin{aligned}
dZ^{[1]} &= dA^{[1]} \odot g^{[1]'}\left(Z^{[1]}\right) \\
dW^{[1]} &= \frac{1}{m}\, dZ^{[1]} \cdot A^{[0]T} = \frac{1}{m}\, dZ^{[1]} \cdot X^{T} \\
db^{[1]} &= \frac{1}{m}\, dZ^{[1]} \cdot 1_{m\times 1}
\end{aligned}
$$

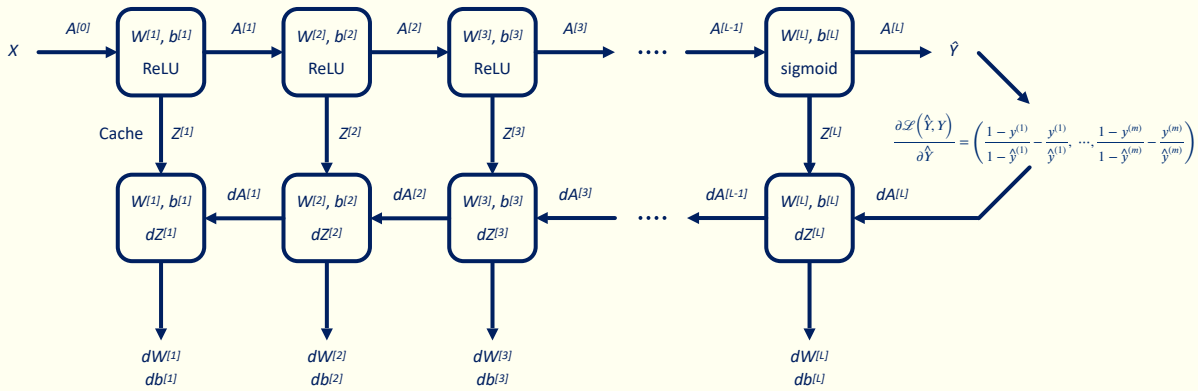
Similarly, when $g^{[L]}(z) = \sigma(z)$, by calculation,

$$
\begin{aligned}
g^{[L]'}(Z^{[L]}) &= \left[g^{[L]'}\left(z^{[L](1)}\right) \quad g^{[L]'}\left(z^{[L](2)}\right) \quad \cdots \quad g^{[L]'}\left(z^{[L](m)}\right)\right] = A^{[L]} \odot \left(1_{1\times m} - A^{[L]}\right) \\
dA^{[L]} &= \left[\frac{1-y^{(1)}}{1-a^{[L](1)}} - \frac{y^{(1)}}{a^{[L](1)}} \quad \frac{1-y^{(2)}}{1-a^{[L](2)}} - \frac{y^{(2)}}{a^{[L](2)}} \quad \cdots \quad \frac{1-y^{(m)}}{1-a^{[L](m)}} - \frac{y^{(m)}}{a^{[L](m)}}\right] = \frac{1_{1\times m} - Y}{1_{1\times m} - A^{[L]}} - \frac{Y}{A^{[L]}}
\end{aligned}
$$

Thus, by chain rule

$$
\begin{aligned}
dZ^{[L]} &= dA^{[L]} \odot g^{[L]'}(Z^{[L]}) \\
&= \left(\frac{1_{1\times m} - Y}{1_{1\times m} - A^{[L]}} - \frac{Y}{A^{[L]}}\right) \odot A^{[L]} \odot \left(1_{1\times m} - A^{[L]}\right) \\
&= A^{[L]} - Y
\end{aligned}
$$

A flowchart of the whole process:



1.4.4 Gradient Descent

The gradient descent for the L-layer NN becomes,

$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

with $l = 1, 2, \ldots, L$.

1.4.5 Parameters vs Hyperparameters

- Parameters: $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$, ..., $W^{[L]}$, $b^{[L]}$
- Hyperparameters: parameters that determine the final values of the parameters
  - Learning rate α
  - Number of iterations in gradient decent algorithm
  - Number of layers L
  - Number of hidden units in each layer, $n^{[1]}, n^{[2]}, \ldots, n^{[L]}$
  - Choice of activation function
  - Momentum, mini-batch size, regularizations, etc.

Applied deep learning is a very empirical process.

Chapter 2

Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

This chapter introduces practical aspects of training and improving neural networks.

2.1 Module 1 - Practical Aspects of Deep Learning

2.1.1 Setting Up Machine Learning Application

2.1.1.1 Train/Dev/Test Sets

As mentioned earlier, applied machine learning is a highly iterative process. It is almost impossible to guess the best hyperparameters on the first attempt. Instead, one needs to go through this cycle many times to find a good network for the application. Thus, how efficiently one can complete each cycle becomes an important factor. Making good choices when setting up training, development, and test sets can make a huge difference in quickly finding a high-performance neural network, thereby making each cycle much more efficient.

- No hyperparameters to tune:
  - `train/test` split: train (80%) + test (20%)
- Hyperparameters need tuning: A `train/test` split does not work, because reusing the test set for development can make the model overfit to the test set.
  - Train/dev/test split: train for learning model parameters, dev (or development, hold-out cross validation) for tuning hyperparameters, test for final evaluation
    - Smaller dataset: train (60%) + dev (20%) + test (20%)
    - Very large dataset (e.g., 1,000,000): train (98%) + dev (1%) + test (1%), or train (99.5%) + dev (0.5%) + test (0.1%)
  - k-fold cross-validation: Use when the dataset is too small for a separate dev set. Build dev set from train set, while keep test set untouched.
    - train (80%) + test (20%)
    - Split training data into k folds
    - For each iteration, train on $k-1$ folds, validate on the remaining fold
    - Repeat for all folds, select hyperparameters with the best average validation score
    - Train the final model on the full train set and evaluate once on the test set

Notes:

- Rule: Use the test set only once, for final evaluation.
- It may be fine without a test set.
- Ensure that the test and dev set come from the same distribution (as the train set).

2.1.1.2 Bias/Variance

Assume that

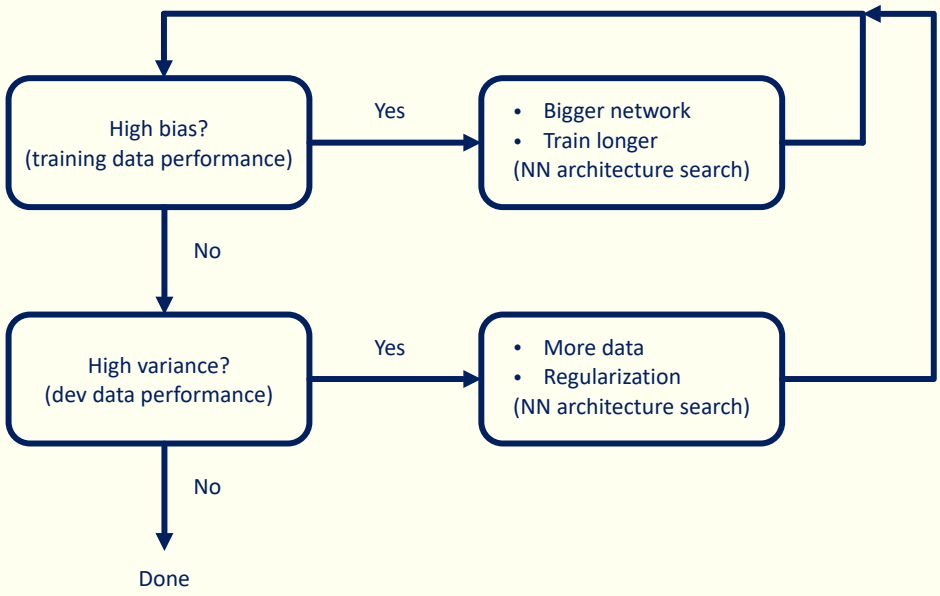
- Optimal (Bayes) error is $\approx 0\%$ (i.e., human error $\approx 0\%$).
- The train and dev sets are drawn from the same distribution.

	Case 1	**Case 2**	**Case 3**	**Case 4**
training error	1%	15%	15%	0.5%
dev error	12%	16%	30%	1%
Diagnosis	high variance	high bias	high bias & high variance	low bias & low variance

By looking at the training error, one can get a sense of how well the model fits the data (i.e., whether there is a bias problem). By comparing the training error with the dev error, one can assess the severity of the variance problem.

- High variance (overfitting): $0 \sim$ training error $\ll$ dev error (e.g., 1% vs. 11%).
- High bias (underfiting): $0 \ll$ training error $\approx$ dev error (e.g., 15% vs. 16%).
- High bias & high variance: $0 \ll$ training error $\ll$ dev error (e.g., 15% vs. 30%).
- Low bias & low variance: training error $\approx$ dev error ≈ 0 (e.g., 0.5% vs. 1%).

Basic recipe for machine learning



A bias–variance trade-off means that lowering bias often raises variance, and lowering variance often raises bias. In earlier machine learning, this trade-off was emphasized because there were fewer ways to improve one without hurting the other. However, in modern deep learning, this trade-off is often less restrictive. For example, using a bigger network almost always reduces bias without necessarily increasing variance (if regularize appropriately). Similarly, adding more training data usually reduces variance with little effect on bias.

2.1.2 Regularizing Neural Network

Regularization is usually the first method to try to solve overfitting (high variance). Another possible approach is to collect more training data. However, this is not always practical because obtaining additional data can sometimes be difficult.

2.1.2.1 L_2 Regularization

The regularization is defined by adding a regularization term to the cost function. Specifically, in logistic regression, the cost function becomes

- L_1 regularization:

$$J(W,b) + \frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j| = J(W,b) + \frac{\lambda}{2m}\, w^T \cdot 1_{n_x\times 1} = J(W,b) + \frac{\lambda}{2m}\|w\|_2^1$$

- L_2 regularization (also called weight decay):

$$J(W,b) + \frac{\lambda}{2m}\sum_{j=1}^{n_x} w_j^2 = J(W,b) + \frac{\lambda}{2m}\, w^T \cdot w = J(W,b) + \frac{\lambda}{2m}\|w\|_2^2$$

Here, λ is called the regularization parameter, and

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix} \in \mathbb{R}^{n_x\times 1}$$

In fact, L_2 regularization is used much more often then L_1 regularization. In practice, L_1 regularization only slightly helps make the model sparse.

In L-layer NN, when applying L_2 regularization, the cost function becomes

$$\begin{aligned}
&J\big(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}\big) \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|W^{[l]}\|_2^2 \\
&= -\frac{1}{m}\sum_{i=1}^{m}\big(y^{(i)}\ln\hat{y}^{(i)} + \big(1-y^{(i)}\big)\ln\big(1-\hat{y}^{(i)}\big)\big) + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}\big(w_{i,j}^{[l]}\big)^2
\end{aligned}$$

For $A = \big(a_{ij}\big) \in \mathbb{R}^{m\times n}$, the Frobenius norm is defined as $\|A\|_{\mathrm{F}} = \|A\|_2 = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$.

The backpropagation becomes

$$\begin{aligned}
dZ^{[L]} &= dA^{[L]} \odot g^{[L]'}\big(Z^{[L]}\big) \\
dW^{[L]} &= \frac{1}{m}\, dZ^{[L]} \cdot A^{[L-1]T} + \frac{\lambda}{m} W^{[L]} \\
db^{[L]} &= \frac{1}{m}\, dZ^{[L]} \cdot 1_{m\times 1} \\
dA^{[L-1]} &= W^{[L]T} \cdot dZ^{[L]}
\end{aligned}$$

$$\begin{aligned}
dZ^{[L-1]} &= dA^{[L-1]} \odot g^{[L-1]'}\big(Z^{[L-1]}\big) \\
dW^{[L-1]} &= \frac{1}{m}\, dZ^{[L-1]} \cdot A^{[L-2]T} + \frac{\lambda}{m} W^{[L-1]} \\
db^{[L-1]} &= \frac{1}{m}\, dZ^{[L-1]} \cdot 1_{m\times 1} \\
dA^{[L-2]} &= W^{[L-2]T} \cdot dZ^{[L-2]}
\end{aligned}$$

$$\vdots$$

$$\begin{aligned}
dZ^{[1]} &= dA^{[1]} \odot g^{[1]'}\big(Z^{[1]}\big) \\
dW^{[1]} &= \frac{1}{m}\, dZ^{[1]} \cdot A^{[0]T} + \frac{\lambda}{m} W^{[1]} = \frac{1}{m}\, dZ^{[1]} \cdot X^{T} + \frac{\lambda}{m} W^{[1]} \\
db^{[1]} &= \frac{1}{m}\, dZ^{[1]} \cdot 1_{m\times 1}
\end{aligned}$$

The gradient descent algorithm becomes

$$\begin{aligned}
W^{[l]} &:= W^{[l]} - \alpha\, dW^{[l]} \\
b^{[l]} &:= b^{[l]} - \alpha\, db^{[l]}
\end{aligned}$$

with $l = 1, 2, \dots, L$.

By calculation,

$$\begin{aligned} W^{[l]} - \alpha\, dW^{[l]}_{\text{with reg}} &= W^{[l]} - \alpha\left(W^{[l]}_{\text{without reg}} + \frac{\lambda}{m} W^{[l]}\right) \\ &= \left(1 - \frac{\alpha\lambda}{m}\right) W^{[l]} - \alpha\, W^{[l]}_{\text{without reg}} \end{aligned}$$

This is why L_2 regularization is also called weight decay, because the weights are multiplied by the factor $1 - \frac{\alpha\lambda}{m}$, which gradually shrinks them during training.

Regularization works by adding an extra term that penalizes large weights. That is, if the regularization parameter λ is large, the weights $W^{[l]}$ tend to become small (i.e., $W^{[l]} \approx 0$).

Two intuitions for why regularization reduces overfitting:

- If λ is large, then $W^{[l]} \approx 0$. This makes the NN simpler, as each hidden unit has a smaller effect (similar to having fewer hidden units), which helps reduce overfitting.
- If λ is large and tanh is used as the activation, then $W^{[l]} \approx 0$. Consequently, $z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]} \approx 0$. When z is small, $\tanh(z)$ approximates to a linear function. As mentioned before, when activation of each layer is linear, the NN couldn't model the non-linear boundaries, which reduces overfitting.

2.1.2.2 Dropout Regularization

Dropout is another very effective regularization technique. The basic idea is to randomly "drop" (ignore) each unit in a NN layer with a certain probability during training. When a unit is dropped, all its incoming and outgoing connections are also ignored, ending up with a simpler network.

To implement the inverted dropout:

```
# layer l
keep_prob = 0.8
dl = np.random.rand(al.shape[0], al.shape[1]) < keep_prob
al = np.multiply(al, dl)
al /= keep_prob # inverted dropout technique
```

Here, `keep_prob` is the probability that a given hidden unit will be kept.

The inverted dropout technique is used to ensure that the expected value of $a^{[l]}$ remains same. Without it, dropping $1 -$ `keep_prob` of hidden units would reduce value of $a^{[l]}$ by `keep_prob`. Inverted dropout makes test time easier, since no extra scaling of activations is needed.

Dropout has a regularizing effect because it ends up with a smaller NN by randomly dropping units. Another intuition to understand dropout is from the perspective of a single unit. For a unit with many inputs and one output, it cannot rely on any single input, because each input might be randomly dropped. This encourages the algorithm to spread weights across all inputs, rather than putting too

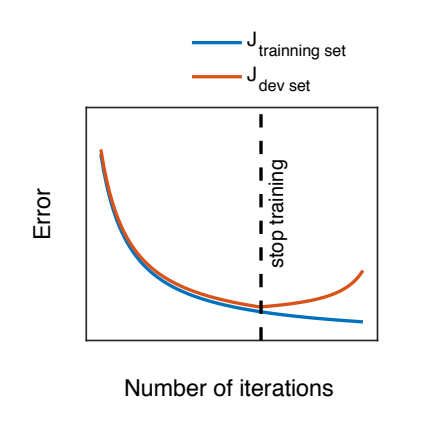
much weight on any one or a few inputs. Spreading the weights reduces the squared norm of the weights (i.e., the weights shrink), which has a similar effect to L_2 regularization.

Dropout is widely used in computer vision, where input sizes are large and overfitting is common due to limited data.

- Different hidden units are dropped for different training examples. For multiple passes through the same training set, each pass randomly drops different hidden units, even for the same example.
- It is common to use `keep_prob` $= 1$ for the input layer. It is possible to vary `keep_prob` by layer, e.g., smaller values for layers more prone to overfitting. The downside is that this introduces more hyperparameters to tune. An alternative is to use a single `keep_prob` value and apply dropout only to selected layers, resulting in a single hyperparameter for dropout.
- Using a larger mini-batch size reduces the noise introduced by dropout and therefore decreases its regularization effect.
- Do not use dropout at test time, because dropping units would add noise to the predictions. In theory, run multiple predictions with different units randomly dropped and average the results works, but this is computationally expensive and gives roughly the same outcome as without dropout (due to inverted dropout).
- Dropout is a regularization technique used to prevent overfitting. Therefore, there is no need to use dropout unless the algorithm is overfitting. One downside of dropout is that the cost function J is no longer well-defined. As a result, it is difficult to check the performance of gradient descent by verifying whether the cost function decreases at every iteration. One solution is temporarily turning off dropout by setting $keep\_prob = 1$ and then check whether the cost function decreases monotonically at every iteration.

2.1.2.3 Other Regularization Methods

- Augment training set by flipping the image horizontally, take random crops of the image, taking random distortions, rotations and translations of the image.
- Early stopping (fewer training iterations)



- The weights w are initialized to small random values at the beginning of training and tend to grow larger as the number of iterations increases. Therefore, stopping the training early may result in mid-sized weights, which helps reduce overfitting.
- However, early stopping has one downside.

  Machine learning can be viewed as consisting of two steps: (1) optimizing the cost function using algorithm such as gradient descent, momentum, RMSprop, Adam, etc., and (2) reducing overfitting using techniques such as regularization, getting more data, etc. Ideally, these two steps should be separated so that each can be handled independently. This principle is called orthogonalization. Early stopping performs both steps at the same time, making them no longer independent and therefore making the process more complicated.
- Compared with early stopping, L_2 regularization allows the network to be trained for many iterations. This makes it easier to decompose and search the hyperparameter space. However, a downside is that multiple values of the hyperparameter λ may need to be tried, which increases the computational cost.

2.1.3 Setting Up Optimization Problem

2.1.3.1 Inputs Normalization

Inputs normalization is one technique that will speed up the training.

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \quad x^{(i)} := x^{(i)} - \mu$$
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)}\right)^2, \quad x^{(i)} := \frac{x^{(i)}}{\sigma}$$

Use the same μ and σ to normalize the test set.

Why normalize inputs?

- If input features are on very different scales, the cost function becomes elongated or "squished" in some directions. Gradient descent on such a cost function may oscillate and take many steps to reach the minimum, often requiring a very small learning rate.
- By normalizing the input features, the cost function becomes more symmetric. Gradient descent can then take larger steps and move directly toward the minimum without excessive oscillation.

2.1.3.2 Vanishing/Exploding Gradients

Vanishing and exploding gradients occur when training very deep networks with gradient descent, i.e., the derivatives can become extremely small or extremely large, which makes training difficult.

A simple example is used to illustrate this. Assume the activation function of a L-layer NN is $g^{[l]}(z) = z$, and $b^{[l]} = 0$,

$$\hat{y} = W^{[L]}W^{[L-1]}W^{[L-2]} \dots W^{[1]}\, x$$

Thus,

$$W^{[l]} < I \rightarrow \hat{y} \rightarrow 0$$
$$W^{[l]} > I \rightarrow \hat{y} \rightarrow \infty$$

Here, I is identity matrix.

Examples of such $W^{[l]}$ are,

$$W^{[l]} = \begin{bmatrix} 0.5 & 0 & \cdots & 0 \\ 0. & 0.5 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0.5 \end{bmatrix} < I, \quad W^{[l]} = \begin{bmatrix} 1.5 & 0 & \cdots & 0 \\ 0. & 1.5 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1.5 \end{bmatrix} > I$$

2.1.3.3 Weight Initialization

A careful choice of the random weight initialization can significantly help (partially solve but not entirely) reduce vanishing and exploding gradients problems.

Random weight initialization .

- ReLU activation:

```
#W_l: W^[l]
#n_l: n^[l]
#n_lm1: n^[l-1]
```

```
W_l = np.random.randn(n_l, n_lm1]) * np.sqrt(2/n_lm1)
```

Here, the weights $W^{[l]}$ have mean 0 and variance $\frac{2}{n}$.

- tanh activation:

```
W_l = np.random.randn(n_l, n_lm1) * np.sqrt(1/n_lm1)
# or
W_l = np.random.randn(n_l, n_lm1) * np.sqrt(2/(n_lm1 + n_l))
```

Here, the weights $W^{[l]}$ have mean 0 and variance approximately $\frac{1}{n}$.

2.1.3.4 Gradient Checking

Gradient checking approximates the derivative of the cost function numerically, which helps debug backpropagation by verifying that the implementation is correct.

The derivative can be estimated in two ways, given a small $\epsilon > 0$,

- Central difference (more accurate):

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \sim \mathcal{O}(\epsilon^2)$$

- Forward difference (less accurate):

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \sim \mathcal{O}(\epsilon)$$

The central difference method is used for gradient checking because it provides a more accurate approximation.

Gradient checking for a NN:

- Take $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}$ and reshape them into a vector θ,

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}) = J(\theta)$$

- Take $dW^{[1]}, db^{[1]}, dW^{[2]}, db^{[2]}, \dots, dW^{[L]}, db^{[L]}$ and reshape them into a vector $d\theta$. $d\theta$ has the same shape as θ, since $dW^{[l]}$ has the same shape as $W^{[l]}$ and $db^{[l]}$ has the same shape as $b^{[l]}$.
- For each i, calculate

$$\begin{aligned} d\theta_i &= \frac{\partial J}{\partial \theta_i} \\ d\theta_i^{\text{approx}} &= \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \cdots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \cdots)}{2\epsilon}. \end{aligned}$$

Here, $\epsilon = 10^{-7}$.

- Check the difference

$$\frac{\|d\theta_i^{\text{approx}} - d\theta_i\|_2}{\|d\theta_i^{\text{approx}}\|_2 + \|d\theta_i\|_2}$$

  - If the value is very small (i.e., 10^{-7}), then $d\theta_i \approx d\theta_i^{\text{approx}}$, and the implementation of back-propagation is likely correct.
  - If it is relatively large (i.e., 10^{-3}), then $d\theta_i^{\text{approx}}$ is far from $d\theta_i$, meaning a possible error in the implementation.

Gradient checking implementation notes:

- Do not use during training, only used for debugging.
- If the algorithm fails gradient checking, check components to try to identify bug.
- Remember to include regularization for computing the cost and gradients.
- Gradient checking doesn't work with dropout. Turn off dropout by setting `keep_dropout` $= 1$ during gradient checking, then turn it on).
- Run gradient checking at random initialization, perhaps again after some training. (This is used under the rare case where implementation of backpropagation is correct only when w and b are close to 0.)

2.2 Module 2 - Optimization Algorithms

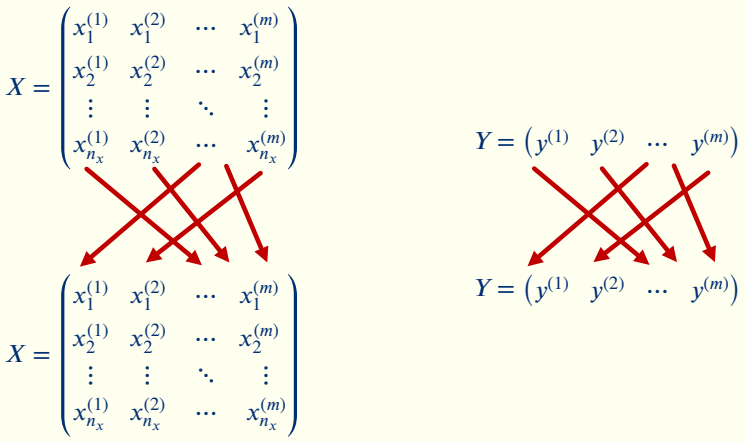
2.2.1 Mini-batch Gradient Descent

Applying machine learning is a highly empirical and iterative process, so it is important to train models quickly. Training NNs on large datasets can be slow, so efficient optimization algorithms are essential.

- Batch gradient descent: Gradients $dW^{[l]}$ and $db^{[l]}$ are calculated using all m training examples, followed by one parameter update per iteration.
  - Requires two for-loops:
    - Outer loop: over epochs (each is vectorized)
    - Inner loop: over L layers (during forward/back propagation)
  - Works well for small training sets. However, for very large datasets, each iteration becomes slow because all training examples must be processesed, which is a major disadvantage.
- Stochastic Gradient Descent (SGD): Gradients $dW^{[l]}$ and $db^{[l]}$ are calculated using only one training example, followed by m parameter updates per iteration.
  - Requires three for-loops:
    - Outer loop: over epochs
    - Middle loop: over m training examples
    - Inner loop: over L layers (during forward/back propagation)
  - Much faster for large datasets, since parameters are updated with one example at a time. However, loses the computational speed benefits of vectorization, which is a key drawback.
- Mini-batch Gradient Descent: Gradients $dW^{[l]}$ and $db^{[l]}$ are calculated using an intermediate number of examples (a mini-batch), followed by one parameter update per mini-batch. Specifically, the training set is divided into several subsets called mini-batch, each of size `mini_batch_size`.
  - Requires three for-loops:
    - Outer loop: over epochs (one epoch = a single pass through the training set)
    - Middle loop: over (vectorized) mini-batches (in a epoch)
    - Inner loop: over L layers (during forward/back propagation)
  - Allows vectorization and enables progress without processing the entire training set, making it much faster than batch gradient descent.

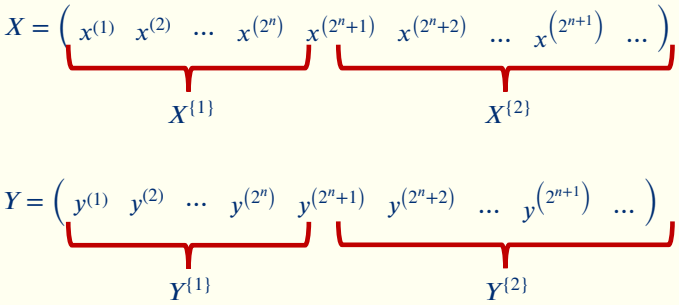
Steps of building mini-batches from the training set (X, Y):

- Shuffle: Create a shuffled version of the training set (X, Y).



Shuffling ensures that examples are split randomly into different mini-batches.

- Partition: Partite the shuffled (X, Y) into mini-batches of size `mini_batch_size`.



The last mini-batch might be smaller since the total number of examples is not always divisible by `mini_batch_size`.

In summary,

- In SGD, parameters tend to oscillate toward the minimum rather than converge smoothly. This noise can be reduced by using a smaller learning rate α.
- `mini_batch_size` is a hyperparameter:
  - `mini_batch_size` $= m$: batch gradient descent
  - `mini_batch_size` $= 1$: SGD
  - $1 <$ `mini_batch_size` $< m$: mini-batch gradient descent
- In batch gradient descent, one epoch allows only one gradient update. In mini-batch gradient descent, one epoch allows multiple gradient updates (one update per mini-batch).
- Mini-batch gradient descent is the most widely used in deep learning.
- Which to choose?
  - Small training set($m \leq 2000$): use batch gradient descent

- Big training set($m \leq 2000$): use mini-batch gradient descent with mini-batch size commonly 64, 128, 256, or 512. Ensure that each mini-batch fit in CPU/GPU memory.

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)

# Gradient descent
for i in range(num_epochs):
    A, caches = forward_propagation(X, parameters) # forward propagation
    cost = compute_cost(A, Y) # compute cost
    grads = backward_propagation(A, caches, parameters) # backpropagation
    parameters = update_parameters(parameters, grads) # update parameters

# SGD
for i in range(num_epochs):
    for j in range(0, m):
        a, caches = forward_propagation(X[:,j], parameters)
        cost = compute_cost(a, Y[:,j])
        grads = backward_propagation(a, caches, parameters)
        parameters = update_parameters(parameters, grads)

# Mini-batch gradient descent
for i in range(num_epochs): # loop over epochs
    for t in range(0, num_minibatches):  # loop over mini-batches
        X_batch = X[:, t*mini_batch_size:(t+1)*mini_batch_size]
        Y_batch = Y[:, t*mini_batch_size:(t+1)*mini_batch_size]
        a, caches = forward_propagation(X_batch, parameters)
        cost = compute_cost(A, Y_batch)
        grads = backward_propagation(A, caches, parameters)
        parameters = update_parameters(parameters, grads) # update parameters per mini-batch
```

2.2.2 Momentum

2.2.2.1 Exponentially Weighted Averages

Let θ_t be the temperature on day t. An exponentially weighted (moving) averages is defined as,

$$v_t = \beta\, v_{t-1} + (1-\beta)\,\theta_t, \quad v_0 = 0$$

Thus,

$$\begin{aligned} v_t &= (1-\beta)\left(\theta_t + \beta\,\theta_{t-1} + \cdots + \beta^{t-2}\,\theta_2 + \beta^{t-1}\,\theta_1\right) \\ &= (1-\beta)\sum_{k=0}^{t-1}\beta^k\theta_{t-k} \end{aligned}$$

Since

$$\lim_{\beta\to 1}\beta^{\frac{1}{1-\beta}} = \lim_{\epsilon\to 0}(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e} \approx 0.37$$

It means that the weight (β^k) of an observation θ_{t-k} decays to about $\frac{1}{e}$ after roughly $\frac{1}{1-\beta}$ steps. A larger β results in a larger averaging window, giving more weight to past observations. Here, v_t can be regarded as approximately averaging the temperature over the last $\frac{1}{1-\beta}$ days.

Bias correction in exponentially weighted averages is a technique used to make the estimate more accurate, especially during the initial phase when v_t is biased toward zero. The bias-corrected estimate is defined as

$$v_t := \frac{v_t}{1-\beta^t}$$

2.2.2.2 Gradient Descent with Momentum

Gradient descent with momentum is an optimization algorithm that speeds up standard gradient descent by dampening oscillations in the optimization path toward the minimum, especially when using mini-batch gradient descent.

The basic idea is to update parameters W and b using an exponentially weighted average of the gradients. Specifically,

- First, initialize

$$v_{dW} = 0, \quad v_{db} = 0$$

- Then at iteration t, compute gradients dW and db on current mini-batch, then

$$\begin{aligned} v_{dW} &= \beta\, v_{dW} + (1-\beta)\, dW \\ v_{db} &= \beta\, v_{db} + (1-\beta)\, db \\ W &:= W - \alpha\, v_{dW} \\ b &:= b - \alpha\, v_{db} \end{aligned}$$

  Here, α is the learning rate and β is the momentum hyperparameter. Typically, $\beta = 0.9$.

  In practice, bias correction is usually not required when implementing gradient descent with momentum. After just a few iterations (e.g., ten), the moving averages have warmed up, and any initial bias from starting at zero becomes negligible.

An analogy for a ball rolling down a bowl:

- Friction: α

- Velocity: v_{dW} and v_{db}
- Acceleration: dW and $db^{[l]}$

2.2.3 RMSprop

RMSprop (root mean square propagation) is an optimization algorithm that accelerates standard gradient descent by dampening oscillations in the path toward the minimum in mini-batch gradient descent. Specifically,

- First, initialize

$$s_{dW} = 0, \quad s_{db} = 0$$

- Then at iteration t, compute gradients dW and db on current mini-batch, then

$$\begin{aligned} s_{dW} &= \beta_2 s_{dW} + (1 - \beta_2) dW \odot dW \\ s_{db} &= \beta_2 s_{db} + (1 - \beta_2) db \odot db \\ W &:= W - \alpha \frac{dW}{\sqrt{s_{dW} + \epsilon}} \\ b &:= b - \alpha \frac{db}{\sqrt{s_{db} + \epsilon}} \end{aligned}$$

  ϵ is a very small number to avoid dividing by zero, usually $\epsilon = 10^{-8}$.

By adapting the learning rate for each parameter, RMSprop often allows a larger effective learning rate α, thereby speeding up convergence.

2.2.4 Adam

Adam (adaptive moment estimation) is an optimization algorithm that combines momentum and RMSprop, resulting in an even better optimization algorithm.

- First, initialize

$$\begin{aligned} v_{dW} &= 0, \quad s_{dW} = 0 \\ v_{db} &= 0, \quad s_{db} = 0 \end{aligned}$$

- Then at iteration t, compute gradients dW and db on current mini-batch, then

$$
\begin{aligned}
&v_{dW} = \beta_1 \cdot v_{dW} + (1-\beta_1) \cdot dW, \quad v_{db} = \beta_1 \cdot v_{db} + (1-\beta_1) \cdot db \\
&s_{dW} = \beta_2 \cdot s_{dW} + (1-\beta_2) \cdot dW \odot dW, \quad s_{db} = \beta_2 \cdot s_{db} + (1-\beta_2) \cdot db \odot db \\
&\tilde{v}_{dW} = \frac{v_{dW}}{1-\beta_1^t}, \quad \tilde{v}_{db} = \frac{v_{db}}{1-\beta_1^t} \\
&\tilde{s}_{dW} = \frac{s_{dW}}{1-\beta_2^t}, \quad \tilde{s}_{db} = \frac{s_{db}}{1-\beta_2^t} \\
&W := W - \alpha \cdot \frac{\tilde{v}_{dW}}{\sqrt{\tilde{s}_{dW}} + \epsilon} \\
&b := b - \alpha \cdot \frac{\tilde{v}_{db}}{\sqrt{\tilde{s}_{db}} + \epsilon}
\end{aligned}
$$

  $\tilde{v}_{dW}$, $\tilde{v}_{db}$, $\tilde{s}_{dW}$, $\tilde{s}_{db}$ denote the bias-corrected version of v_{dW}, v_{db}, s_{dW}, s_{db}, respectively. α, β_1, β_2 and ϵ are all hyperparameters. α needs to be tuned. Usually, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. $\epsilon = 10^{-8}$.

2.2.5 Learning Rate Decay

One technique that can help speed up a learning algorithm is to gradually decrease the learning rate over time, which is known as learning rate decay.

The intuition behind slowly reducing Alpha is that during the initial steps of learning, the algorithm can take larger steps to make rapid progress. As the algorithm approaches convergence, a smaller learning rate allows it to take smaller and more precise steps toward the minimum.

Ways of implementing learning rate decay:

- $$\alpha = \frac{1}{1 + \texttt{decay\_rate} \cdot \texttt{epoch\_num}} \alpha_0$$

  `decay_rate` is a hyperparamer that needs to be tuned.

- $$\alpha = 0.95^{\texttt{epoch\_num}} \alpha_0$$

- $$\alpha = \frac{k}{\sqrt{\texttt{epoch\_num}}} \alpha_0$$

  k is a hyperparameter.

- $$\alpha = \frac{k}{\sqrt{t}}\,\alpha_0$$

  t represents the mini-batch number. k is a hyperparameter.

- α can also be a discrete staircase function, or manual decay.

2.2.6 Problem of Local Optima

- One lesson from deep learning is that intuitions from low-dimensional spaces often do not transfer to high-dimensional spaces. E.g., most zero-gradient points in NNs are saddle points rather than local minima, so getting stuck in bad local minima is unlikely.
- Instead, plateaus, where gradients is close to zero for a long time, can significantly slow down learning.

2.3 Module 3 - Hyperparameter Tuning, Batch Normalization and Programming Frameworks

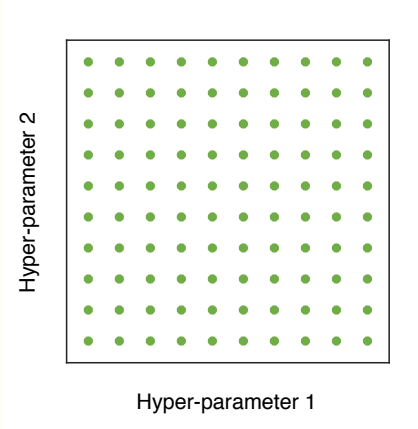
2.3.1 Hyperparameter Tuning

A list of hyperparameters,

- Most important:
  - Learning rate α
- Second most important:
  - In momentum: β (usually $\beta = 0.9$)
  - Mini-batch size `mini_batch_size`
  - Number of hidden units in each layer, $n^{[1]}$, $n^{[2]}$, ... , $n^{[L]}$
- Less important:
  - Number of layers L
  - Learning rate decay
  - In Adam: β_1, β_2, ϵ (usually $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$)
- Other:
  - Regularization strength λ
  - In Dropout: `keep_prob`
  - Number of iterations in gradient decent
  - Choice of activation function

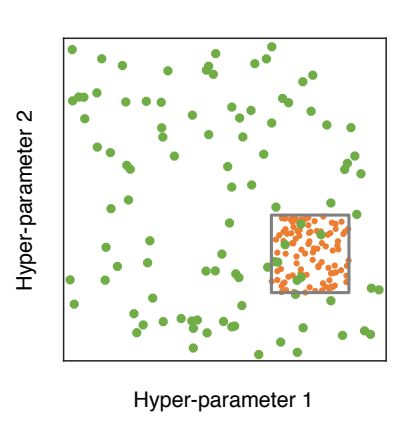
Common practices for assigning assigning values hyperparameters:

- Random sampling: Instead of systematically exploring a grid, assign values by randomly sampling points in the hyperparameter space.
  - In early machine learning, when the number of hyperparameters was small, grid search worked reasonably well.



  - In deep learning, where many hyperparameters exist and their relative importance is unknown, random sampling is preferred because it allows important hyperparameters explore a wider range of values.

- Coarse-to-fine sampling: First sample hyperparameters broadly (coarse search) to find promising regions, then focus on these regions and sample more densely (fine search).



Sampling at random does not necessarily mean uniform sampling. It is important to choose the appropriate scale on which to explore hyperparameters. For example,

- $n^{[l]} \in \{50, 60, \dots, 100\}$
- $L \in \{2, 3, 4\}$
- $\alpha \in \{0.0001, 0.001, 0.01, 0.1, 1\}$ (log-scale), i.e., $\alpha = 10^r$ with $r \in [-4, 0]$

```
r = -4 * np.random.rand(k)
alpha = 10**r
```

- $\beta \in \{0.9, 0.99, 0.999\}$ (log-scale), i.e., $\beta = 1 - 10^r$ with $r \in [-3, -1]$

```
r = np.random.randint(low=-3, high=-1, size=k)
beta= 1-10**r
```

Since $\frac{1}{1-\beta}$ is very sensitive to small changes in β when β is close to 1, sample more densely in this region.

2.3.1.1 How to Organize Hyperparameter Search Process

- Panda: babysitting one model
- Caviar: training many models in parallel

Choice of approach depends on computational resources: use the caviar approach (many random trials) if parallel computation is available, the panda approach if training is costly (i.e., large datasets or models), or a combination of both.

2.3.2 Batch Normalization

In deep learning, applying normalization to the intermediate units of a hidden layer is known as batch normalization. This technique makes hyperparameter tuning (hyperparameter search problem) easier.

Although there is some debate about whether normalization should be applied before or after the activation function (i.e., on $z^{[l]}$ or $a^{[l]}$), in practice batch normalization is most commonly applied to $z^{[l]}$. For $z^{[l](i)}$ in a layer of l of a NN with $i = 1, 2, \dots, m$ and $l = 1, 2, \dots, L$, the batch normalization is defined as,

$$\mu^{[l]} = \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)}, \quad \left(\sigma^{[l]}\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z^{[l](i)} - \mu^{[l]}\right)^2$$
$$z^{[l](i)}_{\text{norm}} = \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\left(\sigma^{[l]}\right)^2 + \epsilon}}, \quad \tilde{z}^{[l](i)} = \gamma^{[l]} z^{[l](i)}_{\text{norm}} + \beta^{[l]}$$

$\gamma^{[l]}$ and $\beta^{[l]}$ control the mean and variance of $\tilde{z}^{[l](i)}$, allowing it to take an appropriate range of values. In particular, if $\gamma^{[l]} = \sqrt{\left(\sigma^{[l]}\right)^2 + \epsilon}$ and $\beta^{[l]} = \mu^{[l]}$, then $\tilde{z}^{[l](i)} = z^{[l](i)}$. Unlike input normalization, $\tilde{z}^{[l](i)}$ is not constrained to have mean 0 and variance 1.

$\left\{W^{[l]}, b^{[l]}, \gamma^{[l]}, \beta^{[l]}\right\}_{l=1}^{L}$ are learnable parameters. However, $b^{[l]}$ can be removed when applying the batch normalization,

$$\begin{cases} z^{[l](i)} = W^{[l]}, a^{[l-1](i)} + b^{[l]} \\ \tilde{z}^{[l](i)} = \gamma^{[l]} \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\left(\sigma^{[l]}\right)^2 + \epsilon}} + \beta^{[l]} \end{cases} \iff \begin{cases} z^{[l](i)} = W^{[l]}, a^{[l-1](i)} \\ \tilde{z}^{[l](i)} = \gamma^{[l]} \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\left(\sigma^{[l]}\right)^2 + \epsilon}} + \beta^{[l]} \end{cases}$$

2.3.2.1 Deep NN Mini-Batch Gradient Descent with Batch Normalization

- For iteration t, select a mini-batch of size `mini_batch_size`:

$$X^{\{t\}} = X[:, t \cdot \texttt{mini\_batch\_size} : (t+1) \cdot \texttt{mini\_batch\_size}]$$

- Compute forward propagation with batch normalization (loop over all layers). In each hidden layer, replace $z^{[l]}$ with $\tilde{z}^{[l]}$ via batch normalization. Each mini-batch is normalized using its own mean mean $\mu^{\{t\}[l]}$ and variance $\left(\sigma^{\{t\}[l]}\right)^2$.
- Perform backward propagation to compute gradients $dW^{[l]}$, $d\gamma^{[l]}$, and $d\beta^{[l]}$ (loop over all layers).
- Update parameters $W^{[l]}$, $\gamma^{[l]}$, and $\beta^{[l]}$, for $l = 1, 2, \dots, L$:

$$\begin{aligned} W^{[l]} &:= W^{[l]} - \alpha\, dW^{[l]} \\ \gamma^{[l]} &:= \gamma^{[l]} - \alpha\, d\gamma^{[l]} \\ \beta^{[l]} &:= \beta^{[l]} - \alpha\, d\beta^{[l]} \end{aligned}$$

Given mini-batch $X^{\{t\}}$,

$$X^{\{t\}} : x^{(i)} = a^{[0](i)} \xrightarrow[z^{[1](i)} = W^{[1]} a^{[0](i)}]{W^{[1]}} z^{[1](i)} \xrightarrow[\tilde{z}^{[1](i)} = \gamma^{[1]} \frac{z^{[1](i)} - \mu^{\{t\}[1]}}{\sqrt{\left(\sigma^{\{t\}[1]}\right)^2 + \epsilon}} + \beta^{[1]}]{\text{BN: } \gamma^{[1]}, \beta^{[1]}} \tilde{z}^{[1](i)} \xrightarrow[a^{[1](i)} = g^{[1]}\left(\tilde{z}^{[1](i)}\right)]{\text{Activation}}$$

$$a^{[1](i)} \xrightarrow[z^{[2](i)} = W^{[2]} a^{[1](i)}]{W^{[2]}} z^{[2](i)} \xrightarrow[\tilde{z}^{[2](i)} = \gamma^{[2]} \frac{z^{[2](i)} - \mu^{\{t\}[2]}}{\sqrt{\left(\sigma^{\{t\}[2]}\right)^2 + \epsilon}} + \beta^{[2]}]{\text{BN: } \gamma^{[2]}, \beta^{[2]}} \tilde{z}^{[2](i)} \xrightarrow[a^{[2](i)} = g^{[2]}\left(\tilde{z}^{[2](i)}\right)]{\text{Activation}} \dots$$

Here,

$$\begin{aligned} &W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}, \quad b^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}, \quad z^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}, \quad \tilde{z}^{[l]} \in \mathbb{R}^{n^{[l]} \times 1} \\ &\gamma^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}, \quad \beta^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}, \quad a^{[l]} \in \mathbb{R}^{n^{[l]} \times 1} \end{aligned}$$

Batch normalization also works with gradient descent with momentum, RMSprop, or Adam.

2.3.2.2 Why Batch Norm Works?

Normalizing input features to have mean 0 and variance 1 makes all features on a similar scale, which helps speed up learning. Similarly, batch normalizing normalizes hidden unit values in a NN, which also helps speed up learning.

For example, a cat detection classifier trained on a dataset containing only black cat images might not work well on a dataset containing colored cat images, due to the data distribution has changed, which

is called covariate shift. The general idea behind covariate shift is that if a model learns a mapping from X to Y, and the distribution of X changes, the model may need retraining, even if the true function mapping from X to Y remains the same. The problem becomes even worse if the true function itself also changes. In deep NN, the distribution of inputs to later layers keeps changing as earlier layers update their weights. This forces later layers to constantly adapt to new input distributions, which slows down learning. Batch normalization reduces this problem by keeping the mean and variance of hidden units unchanged (controlled by the γ and β). This makes the inputs to each layer more consistent (i.e., weakens the dependency between layers), so each layer can learn more independently without constantly adjusting to changes from previous layers, which speeds up learning.

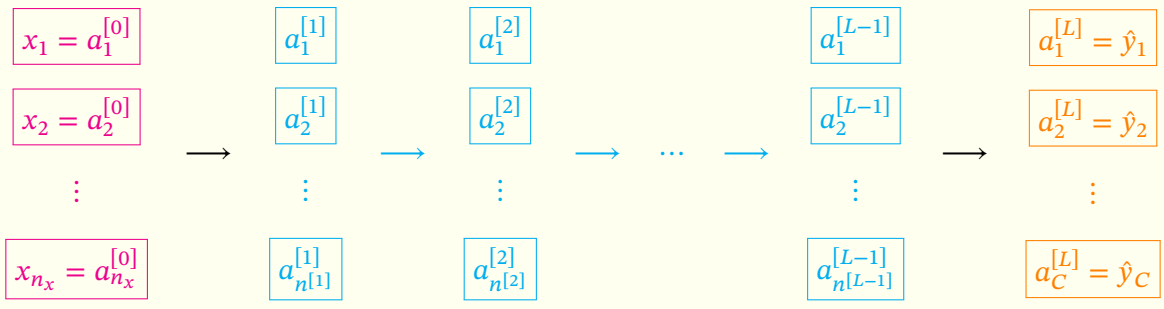
In addition, batch normalization also has a slight regularization effect. The mean and variance in batch normalization are computed from each mini-batch, which is only a small sample of the dataset, and thus adds some noise to each hidden layers. This noise prevents downstream hidden units from relying too much on any single hidden unit, producing a small regularization effect similar to dropout. However, the noise is relatively small, so the regularization effect is weak. For stronger regularization, batch normalization is often used together with dropout. Similar with the dropout, a larger mini-batch size can reduce the noise and therefore also reduce the regularization effect. Although the batch normalization has a regularization effect, it is not really used as regularization.

2.3.2.3 Batch Norm at Test Time

During training, batch normalization processes data one mini-batch at a time, computing the mean and variance for each mini-batch. During testing, the model needs to process one example at a time, with a single mean $\lambda^{[l]}$ and variance $\left(\sigma^{[l]}\right)^2$ for all test examples for each single layer l.

- In theory, $\lambda^{[l]}$ and $\left(\sigma^{[l]}\right)^2$ can be obtained by running the entire training set through the final trained NN.
- In practice, $\lambda^{[l]}$ and $\left(\sigma^{[l]}\right)^2$ are calculated using an exponentially weighted average of the mini-batch means $\lambda^{\{t\}[l]}$ and mini-batch variances $\left(\sigma^{\{t\}[l]}\right)^2$ computed during training.

2.3.3 Multi-class Classification



We have explored $L-1$-layer NN with a scaler for binary classification. This section focuses on $L-1$-hidden-layer NN with a vectorized output (for multi-class classification, with C classes).

2.3.3.1 Softmax vs Sigmoid Function

- Softmax function:
  - Maps a vector of real-valued inputs (logits) into a probability distribution (components summing to 1).

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

  - Commonly used in
    - Multi-class classification (only one correct answer): converts model outputs (logits, vector) into class probabilities (vector).

    E.g., softmax classifier, activation function in the final (output) layer of a neural network for multi-class classification tasks.
- Sigmoid Function:
  - Maps any real-valued input to the interval $(0, 1)$, making it suitable for probabilistic outputs.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

  - When defining $z = z_1 - z_2$, the sigmoid function is equivalent to the softmax function with two logits z_1 and z_2.

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(z_1-z_2)}} = \frac{e^{z_1}}{e^{z_1}+e^{z_2}}$$
$$1 - \frac{e^{z_1}}{e^{z_1}+e^{z_2}} = \frac{e^{z_2}}{e^{z_1}+e^{z_2}}$$

  - Commonly used in:
    - Binary classification: converts a single model output (logit, scaler) into a probability
    - Multi-label classification (multiple correct answers): converts model outputs (logits, vector) into independent class probabilities (vector).

    E.g., logistic regression, activation function in the final (output) layer of a neural network for binary or multi-label classification tasks.

2.3.3.2 Softmax Regression

Softmax regression is a generalization of logistic regression. Logistic regression is designed for binary classification, while softmax regression extends the same idea to multi-class classification.

The softmax function converts scores (logits) into class probabilities by normalizing exponentiated scores. When there are only two classes, the two probabilities must sum to 1, so only one is independent. In this case, the softmax function reduces to a single ratio, which is equivalent to the sigmoid function.

Consequently, for the two-class case, softmax regression is equivalent to logistic regression.

2.3.3.3 L-layer Neural Networks for Multi-Class Classification

Softmax regression can be viewed as a single-layer (no hidden layer) NN with a vectorized output. It is limited to multi-class classification problems with linear decision boundaries between classes. By contrast, deep NNs with hidden layers can learn nonlinear decision boundaries, allowing them to handle more complex classification tasks.

A binary classifier and a multi-class classifier are very similar, except for the output layer activation:

- Binary classification: sigmoid
- Multi-class classification: softmax

Given a single training example (x, y), denote

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_C \end{bmatrix} \in \mathbb{R}^{C\times 1}, \quad \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_C \end{bmatrix} \in \mathbb{R}^{C\times 1}, \quad \ln \hat{y} = \begin{bmatrix} \ln \hat{y}_1 \\ \ln \hat{y}_2 \\ \vdots \\ \ln \hat{y}_C \end{bmatrix} \in \mathbb{R}^{C\times 1}$$

Here, y is represented as a one-hot vector.

The loss function is defined as,

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j \ln \hat{y}_j$$

Particularly, if $y_k = 1$, the loss function becomes

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j \ln \hat{y}_j = -\ln \hat{y}_k$$

Given a set of m training examples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$. Denote

$$Z^{[l]} = \begin{bmatrix} z^{[l](1)} & z^{[l](2)} & \dots & z^{[l](m)} \end{bmatrix} = \begin{bmatrix} z_1^{[l](1)} & z_1^{[l](2)} & \cdots & z_1^{[l](m)} \\ z_2^{[l](1)} & z_2^{[l](2)} & \cdots & z_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n^{[l]}}^{[l](1)} & z_{n^{[l]}}^{[l](2)} & \cdots & z_{n^{[l]}}^{[l](m)} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times m}, \quad l = 1, 2, \dots, L-1$$

$$Z^{[L]} = \begin{bmatrix} z^{[L](1)} & z^{[L](2)} & \dots & z^{[L](m)} \end{bmatrix} = \begin{bmatrix} z_1^{[l](1)} & z_1^{[l](2)} & \cdots & z_1^{[l](m)} \\ z_2^{[l](1)} & z_2^{[l](2)} & \cdots & z_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ z_C^{[l(1)} & z_C^{[l](2)} & \cdots & z_C^{[l](m)} \end{bmatrix} \in \mathbb{R}^{C \times m}$$

$$g^{[l]}(Z^{[l]}) = \begin{bmatrix} g^{[l]}(z^{[l](1)}) & g^{[l]}(z^{[l](2)}) & \dots & g^{[l]}(z^{[1](m)}) \end{bmatrix}$$

$$= \begin{bmatrix} g^{[l]}(z_1^{[l](1)}) & g^{[l]}(z_1^{[l](2)}) & \cdots & g^{[1]}(z_1^{[l](m)}) \\ g^{[l]}(z_2^{[l](1)}) & g^{[l]}(z_2^{[l](2)}) & \cdots & g^{[1]}(z_2^{[l](m)}) \\ \vdots & \vdots & \ddots & \vdots \\ g^{[l]}(z_{n^{[l]}}^{[l](1)}) & g^{[1]}(z_{n^{[l]}}^{[l](2)}) & \cdots & g^{[1]}(z_{n^{[l]}}^{[l](m)}) \end{bmatrix} \in \mathbb{R}^{n_x \times m}, \quad l = 1, 2, \dots, L-1$$

$$g^{[L]}(Z^{[L]}) = \begin{bmatrix} g^{[L]}(z^{[L](1)}) & g^{[L]}(z^{[L](2)}) & \dots & g^{[L]}(z^{[L](m)}) \end{bmatrix}$$

$$= \begin{bmatrix} g^{[l]}(z_1^{[l](1)}) & g^{[l]}(z_1^{[l](2)}) & \cdots & g^{[1]}(z_1^{[l](m)}) \\ g^{[l]}(z_2^{[l](1)}) & g^{[l]}(z_2^{[l](2)}) & \cdots & g^{[1]}(z_2^{[l](m)}) \\ \vdots & \vdots & \ddots & \vdots \\ g^{[l]}(z_C^{[l](1)}) & g^{[1]}(z_C^{[l](2)}) & \cdots & g^{[1]}(z_C^{[l](m)}) \end{bmatrix} \in \mathbb{R}^{C \times m}$$

$$A^{[0]} = \begin{bmatrix} a^{[0](1)} & a^{[0](2)} & \dots & a^{[0](m)} \end{bmatrix} = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix} = X \in \mathbb{R}^{n_x \times m}$$

$$A^{[l]} = \begin{bmatrix} a^{[l](1)} & a^{[l](2)} & \dots, a^{[l](m)} \end{bmatrix} = \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \cdots & a_1^{[l](m)} \\ a_2^{[l](1)} & a_2^{[l](2)} & \cdots & a_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^{[l]}}^{[l](1)} & a_{n^{[l]}}^{[l](2)} & \cdots & a_{n^{[l]}}^{[l](m)} \end{bmatrix} \in \mathbb{R}^{n^{[1]} \times m}, \quad l = 1, 2, \dots, L-1$$

$$A^{[L]} = \begin{bmatrix} a^{[L](1)} & a^{[L](2)} & \dots, a^{[L](m)} \end{bmatrix} = \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \cdots & a_1^{[l](m)} \\ a_2^{[l](1)} & a_2^{[l](2)} & \cdots & a_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_C^{[l](1)} & a_C^{[l](2)} & \cdots & a_C^{[l](m)} \end{bmatrix}$$

$$= \begin{bmatrix} \hat{y}_1^{(1)} & \hat{y}_1^{(2)} & \cdots & \hat{y}_1^{(m)} \\ \hat{y}_2^{(1)} & \hat{y}_2^{(2)} & \cdots & \hat{y}_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{y}_C^{(1)} & \hat{y}_C^{(2)} & \cdots & \hat{y}_C^{(m)} \end{bmatrix} = \hat{Y} \in \mathbb{R}^{C \times m}$$

$$b_{1 \times m}^{[l]} = \begin{bmatrix} b^{[l]} & b^{[l]} & \dots & b^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times m}$$

The forward propagation becomes

$$
\begin{aligned}
&A^{[0]} = X \\
&Z^{[1]} = W^{[1]} \cdot A^{[0]} + b^{[1]}_{1\times m}, \quad A^{[1]} = g^{[1]}\big(Z^{[1]}\big) \\
&Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]}_{1\times m}, \quad A^{[2]} = g^{[2]}\big(Z^{[2]}\big) \\
&\vdots \\
&Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + b^{[L]}_{1\times m}, \quad A^{[L]} = g^{[L]}\big(Z^{[L]}\big) = \hat{Y}
\end{aligned}
$$

For $l = 1, 2, \cdots, L$, denote

$$
\begin{aligned}
&dA^{[l]} = \Big[da^{[l](1)}, da^{[l](2)} \quad \dots \quad da^{[l](m)}\Big] \in \mathbb{R}^{n^{[l]}\times m} \\
&dZ^{[l]} = \Big[dz^{[l](1)} \quad dz^{[l](2)} \quad \dots \quad dz^{[l](m)}\Big] \in \mathbb{R}^{n^{[l]}\times m} \\
&g^{[l]'}\big(Z^{[l]}\big) = \Big[g^{[l]'}\big(z^{[l](1)}\big) \quad g^{[l]'}\big(z^{[l](2)}\big) \quad \dots \quad g^{[l]'}\big(z^{[l](m)}\big)\Big] \in \mathbb{R}^{n^{[l]}\times m}
\end{aligned}
$$

Thus, the backpropagation becomes

$$
\begin{aligned}
&dZ^{[L]} = \hat{Y} - Y \\
&dW^{[L]} = \frac{1}{m}\, dZ^{[L]} \cdot A^{[L-1]T} \\
&db^{[L]} = \frac{1}{m}\, dZ^{[L]} \cdot 1_{m\times 1} \\
&dA^{[L-1]} = W^{[L]T} \cdot dZ^{[L]}
\end{aligned}
$$

$$
\begin{aligned}
&dZ^{[L-1]} = dA^{[L-1]} \odot g^{[L-1]'}\big(Z^{[L-1]}\big) \\
&dW^{[L-1]} = \frac{1}{m}\, dZ^{[L-1]} \cdot A^{[L-2]T} \\
&db^{[L-1]} = \frac{1}{m}\, dZ^{[L-1]} \cdot 1_{m\times 1} \\
&dA^{[L-2]} = W^{[L-2]T} \cdot dZ^{[L-2]}
\end{aligned}
$$

$$
\vdots
$$

$$
\begin{aligned}
&dZ^{[1]} = dA^{[1]} \odot g^{[1]'}\big(Z^{[1]}\big) \\
&dW^{[1]} = \frac{1}{m}\, dZ^{[1]} \cdot A^{[0]T} = \frac{1}{m}\, dZ^{[1]} \cdot X^{T} \\
&db^{[1]} = \frac{1}{m}\, dZ^{[1]} \cdot 1_{m\times 1}
\end{aligned}
$$

According to

$$\mathcal{L}\left(a^{[L]}, y\right) = -\sum_{j=1}^{C} y_j \ln a_j^{[L]}$$

$$g^{[L]}\left(z^{[L]}\right) = \begin{bmatrix} \frac{e^{z_1^{[L]}}}{\sum_{j=1}^{C} e^{z_j^{[L]}}} \\ \frac{e^{z_2^{[L]}}}{\sum_{j=1}^{C} e^{z_j^{[L]}}} \\ \vdots \\ \frac{e^{z_C^{[L]}}}{\sum_{j=1}^{C} e^{z_j^{[L]}}} \end{bmatrix}$$

By calculation

$$dA^{[L]} = \begin{bmatrix} -\frac{y^{(1)}}{a^{[L]}} & -\frac{y^{(2)}}{a^{[L]}} & \cdots & -\frac{y^{(m)}}{a^{[L]}} \end{bmatrix} = -\frac{Y}{A^{[L]}}$$

$$g^{[L]'}\left(z^{[L]}\right) = \frac{1}{\sum_{j=1}^{C} e^{z_j^{[L]}}} \begin{bmatrix} e^{z_1^{[L]}} \\ e^{z_2^{[L]}} \\ \vdots \\ e^{z_C^{[L]}} \end{bmatrix} - \frac{1}{\left(\sum_{j=1}^{C} e^{z_j^{[L]}}\right)^2} \begin{bmatrix} e^{2z_1^{[L]}} \\ e^{2z_2^{[L]}} \\ \vdots \\ e^{2z_C^{[L]}} \end{bmatrix} = g^{[L]}\left(z^{[L]}\right) - \left(g^{[L]}(z^{[L]})\right)^2 = a^{[L]} - a^{[L]} \odot a^{[L]}$$

That is,

$$dA^{[L]} = \begin{bmatrix} -\frac{y^{(1)}}{a^{[L]}} & -\frac{y^{(2)}}{a^{[L]}} & \cdots & -\frac{y^{(m)}}{a^{[L]}} \end{bmatrix} = -\frac{Y}{A^{[L]}}$$
$$g^{[L]'}\left(Z^{[L]}\right) = A^{[L]} - A^{[L]} \odot A^{[L]}$$

Thus, by chain rule,

$$dZ^{[L]} = dA^{[L]} \odot g^{[L]'}\left(Z^{[L]}\right) = -\frac{Y}{A^{[L]}} \odot \left(A^{[L]} - A^{[L]} \odot A^{[L]}\right) = A^{[L]} - Y$$

2.3.4 Introduction to Programming Frameworks

A list of deep learning frameworks:

- Caffe/Caffe2
- CNTK
- DL4J
- Keras

- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Criteria of choosing deep learning frameworks:

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)

Chapter 3

Course 3: Structuring Machine Learning Projects

3.1 Module 1 - ML Strategy

3.1.1 Introduction to ML Strategy

3.1.1.1 Why ML Strategy?

- Examples of machine learning strategies:
  - Collect more data
  - Collect more diverse training set
  - Train algorithm longer with gradient descent
  - Try Adam instead of gradient descent
  - Try bigger network
  - Try smaller network
  - Try dropout
  - Add L_2 regularization
  - Network architecture (activation functions, number of layers, number of hidden units, etc.)
- Why machine learning strategy?

  A good machine learning strategy helps identify the most promising direction when solving a problem. In the era of deep learning, this process has changed, as deep learning algorithms are reshaping how problems are approached.

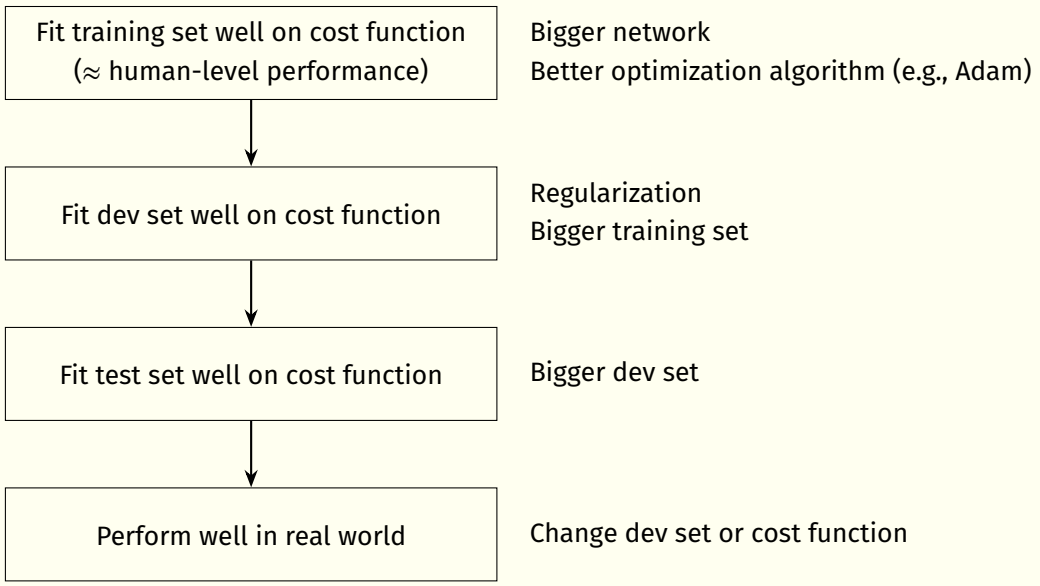
3.1.1.2 Orthogonalization

Effective machine learning focuses on clearly knowing what to tune for a desired effect.

Imagine a car with two knobs, each affecting both steering and speed (e.g., one controls 0.3 steering and 0.8 speed, the other 0.2 steering and 0.9 speed). In theory, any steering angle and speed can be achieved by adjusting these two knobs. However, it is much easier to control the car with two separate knobs (one controlling steering angle and the other controlling speed) than with knobs that affect both. This idea is called orthogonalization, means designing systems so each component can be adjusted independently, making tuning simpler and more effective.

Early stopping is a useful and commonly used technique in training neural networks. However, it can be harder to reason about because it affects both how well the model fits the training data and how well it performs on the validation set. Since it influences both at the same time, it is less orthogonal. Having more independent controls makes model tuning easier.

A diagram of orthogonal tuning in supervised learning:



3.1.2 Setting Up Goals

3.1.2.1 Single Number Evaluation Metric

When starting on a machine learning problem, one recommendation is to define a single number evaluation metric (or a single real number evaluation metric). This allows one to quickly determine whether a new idea improves performance compared to previous approaches.

Examples:

- The F_1 score is used to evaluate a classification model, especially when the data is imbalanced. It captures the trade-off between precision and recall by combining them into a single number using their harmonic mean.

  The F_1 score is defined as the harmonic mean of precision P and recall R, i.e.,

  $$F_1 = \frac{2PR}{P+R}$$

  with

  $$P = \frac{N_{\text{TP}}}{N_{\text{predicted positive}}} = \frac{N_{\text{TP}}}{N_{\text{TP}} + N_{\text{FP}}}$$
  $$R = \frac{N_{\text{TP}}}{N_{\text{actual positive}}} = \frac{N_{\text{TP}}}{N_{\text{TP}} + N_{\text{FN}}}$$

Classifier	Precision	Recall	F_1 Score
A	95%	90%	≈ 0.924
B	98%	85%	≈ 0.910

In practice, it is useful to have a well-defined dev set (used to measure P and R) and a single number evaluation metric. This makes it easier to compare different models and quickly determine whether a new approach improves performance, thereby speeding up the iterative process of improving a machine learning algorithm.

- Average performance is another reasonable single number evaluation metric.

Algorithm	US	China	India	Other	Mean
A	3%	7%	5%	9%	6.00%
B	5%	6%	5%	10%	6.50%
C	2%	3%	4%	5%	3.50%
D	5%	8%	7%	2%	5.50%
E	4%	5%	2%	4%	3.75%
F	7%	11%	8%	12%	9.50%

3.1.2.2 Satisficing and Optimizing Metric

Using a single number evaluation metric is often useful, but it can be difficult to capture everything cared about in one number. In such cases, it is better to use two types of metrics:

- an optimizing metric
- one or more satisficing metrics

If there are N metrics, choose one as the optimizing metric and treat the remaining $N-1$ as satisficing metrics. The goal is to improve the optimizing metric as much as possible, while ensuring that the other metrics stay above their minimum acceptable thresholds. There is no need to improve the satisficing metrics beyond these thresholds.

Example: Wake word detection

- For a voice assistant that uses wake words (e.g., “Alexa” or “Hey Siri”) to activate the device, two metrics are important:
  - Accuracy: how often the system correctly activates when the wake word is spoken
  - False positives: how often the system mistakenly activates when it shouldn’t (no wake word is spoken)

A good approach to evaluate this system is to maximize accuracy (optimizing metric) while ensuring at most one false positive per 24 hours (satisficing metric). The goal is to make the system as accurate as possible while keeping false activations within an acceptable limit.

3.1.2.3 Train/Dev/Test Distributions

Evaluation metrics are computed on the training set, dev set, or test set. The way these sets are constructed can significantly impact how efficiently a machine learning project progresses. Therefore, it is important to set them up properly.

E.g., in a classification task, a typical workflow involves

- training different models (i.e., classifiers) on the training set
- evaluating models and selecting the best one using the dev set
- finally assessing performance on the test set

Setting up the dev set and the evaluation metric is like defining the target for the model to aim at. If the dev and test sets come from different distributions, it is as if the target changes at test time (when evaluating the model on the test set). Therefore, it is important that the dev and test sets come from the same distribution. One way to ensure this is to randomly shuffle the data and then split it into dev and test sets so that both are drawn from the same distribution. A guideline is to choose dev and test sets that reflect the data expected in future applications, while also ensuring they come from the same distribution.

The way the training set is used determines how well the model is able to hit the target.

3.1.2.4 Size of Dev and Test Sets

In addition to their distribution, the size of the dev and test sets can also influence the performance of deep learning.

The guideline of how to set up the dev and the test sets has changed in the deep learning era.

- In earlier machine learning settings with relatively small datasets (e.g., 10^2-10^4 examples), it was common to use splits such as training/test with $(70\%, 30\%)$ or training/dev/test with $(60\%, 20\%, 20\%)$.
- However, in the modern era with much larger datasets (e.g., 10^6 or more), it is often more practical to use splits such as training/dev/test with $(98\%, 1\%, 1\%)$, with most data for training while keeping smaller dev and test sets.

Rules:

- Set the dev set to be big enough to detect differences in various models.
- Set the test set to be big enough to give high confidence in the overall performance of the system.

- In some cases, it might be okay without a test set.

3.1.2.5 When to Change Dev/Test Sets and Metrics?

A well-defined metric and a representative dev set enable faster model comparison and more efficient iteration. Even if they are imperfect, it is better to establish them early and refine them over time.

In general, the evaluation metric and dataset should align (as closely as possible) with the true objective.

As a practical guideline,

- If good performance on the current metric and dev/test sets does not reflect real-world performance, then the metric and/or the dev/test sets should be revised. E.g., if the dev/test sets contain only high-quality images but the application must handle low-quality images, the data should be updated to better reflect actual usage conditions.
- Similarly, if the evaluation metric no longer ranks models in a way that matches practical preferences, it indicates that the metric (or the dev/test sets) should be adjusted. E.g., consider an anti-porn cat classifier.
  - The standard error metric is

    $$\epsilon = \frac{1}{m_{\text{dev}}} \sum_{i=1}^{m_{\text{dev}}} \mathcal{I}\{y^{(i)}_{\text{predict}} \neq y^{(i)}\}$$

    Here, $\mathcal{I}\{y_{\text{predict}} \neq y\}$ represntes the indicator function. It is defined as

    $$\mathcal{I}\{y_{\text{predict}} \neq y\} = \begin{cases} 1, & y_{\text{predict}} \neq y \\ 0, & y_{\text{predict}} = y \end{cases}$$

  - However, if some errors (e.g., misclassifying porn images) are more critical, a weighted error metric can be used:

    $$\epsilon = \frac{1}{\sum_{i=1}^{m_{\text{dev}}} w^{(i)}} \sum_{i=1}^{m_{\text{dev}}} w^{(i)} \cdot \mathcal{I}\{y^{(i)}_{\text{predict}} \neq y^{(i)}\}, \quad w^{(i)} = \begin{cases} 1, & \text{if } y^{(i)} \text{ is un-porn} \\ 10, & \text{if } y^{(i)} \text{ is porn} \end{cases}$$

There exists an orthogonalization between defining a metric and changing the metric:

- How to define a metric to evaluate classifiers (place the target).
- How to do well on this metric (aim accurately).

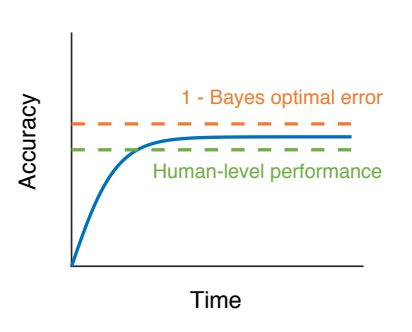
3.1.3 Comparing to Human-level Performance

3.1.3.1 Why Human-level Performance?

In recent years, many machine learning teams have focused on comparing systems to human-level performance, for two main reasons:

- Advances in deep learning have enabled models to approach or even match human abilities in many tasks.
- It is often more efficient to develop and improve models on tasks that humans can also perform, making human performance a natural benchmark.

Here is a figure showing the progress of exploring the high accuracy.



Progress is often fast until human-level performance is reached, after which it slows down for two main reasons:

- Human-level performance is often close to the Bayes (optimal) error (the minimum possible error, which may be greater than 0%), leaving limited room for further improvement.
- There are tools to improve performance while the model is still below human-level performance,
  - Get labeled data from humans.
  - Gain insight from manual error analysis: Why did a person get this right?
  - Better analysis of bias/variance.

  However, these tools become less useful once human-level performance is surpassed.

3.1.3.2 Avoidable Bias

A learning algorithm should perform well on the training set (i.e. achieve low training error), but its achievable performance is limited by the Bayes error.

Human-level performance provides a useful reference for how well the model needs to perform on the training set.

- Avoidable bias measures how much the model can still improve its performance on the training set. It is defined as the difference between the training error and the Bayes error,

$$\text{avoidable bias} = \text{training error} - \text{Bayes error}$$

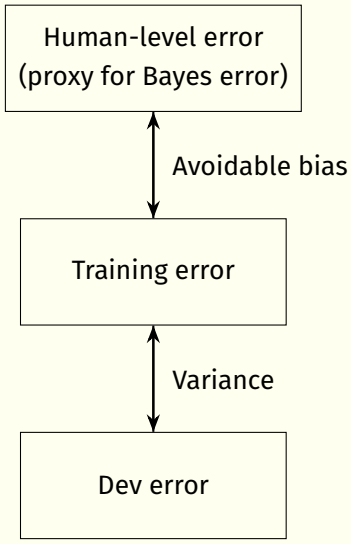
- In practice, since the Bayes error is unknown, human-level performance (or human-level error) is often used as an approximation. In this case,

$$\text{avoidable bias} \approx \text{training error} - \text{human-level performance}$$

E.g.,

Human-level error (≈ Bayes optimal error)	1%	7.5%
Training error	8%	8%
Dev error	10%	10%
Focus	avoidable bias	variance

3.1.3.3 Understanding Human-level Performance



3.1.3.4 Surpassing Human-level Performance

Once the training error surpasses human-level performance, the ways to further improve the machine learning system become less clear.

Fields where machine learning significantly surpasses the human-level performance:

- Online advertising
- Product recommendations
- Logistics (predicting transit time)
- Loan approvals

All of these examples involve structured data rather than natural perception tasks.

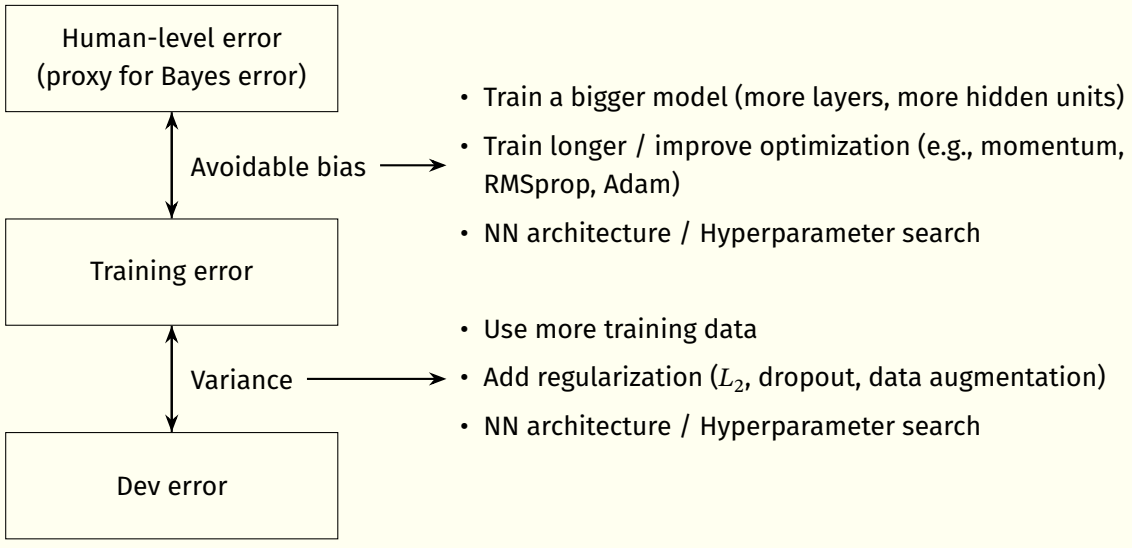
Natural perception tasks (e.g., computer vision, speech recognition, and natural language processing) are areas where humans typically perform very well, making it harder for machines to surpass human-level performance. However, in some specific tasks (e.g., speech recognition, certain image recognition tasks, and medical applications), machine learning systems have significantly exceeded human performance.

3.1.3.5 Improving Model Performance

Getting a supervised learning algorithm to work well essentially means assuming it can do two things:

- Fit the training set well, achieving low avoidable bias, by using larger neural networks or training longer.
- Generalize from the training set to the dev/test sets, meaning low variance, which can be improved through regularization or by collecting more training data.

To summarize,



3.1.4 Case Study: Bird Recognition in the City of Peacetopia

You are a renowned researcher in the City of Peacetopia. The residents of Peacetopia share a unique characteristic: they are afraid of birds. To protect them, you are tasked with developing an algorithm that will detect any bird flying over Peacetopia and alert the population.

The City Council provides you with a dataset of 10,000,000 images of the sky above Peacetopia, captured by the city's security cameras. They are labeled:

- $y = 0$: There is no bird on the image
- $y = 1$: There is a bird on the image

Your goal is to create an algorithm capable of classifying new images taken by security cameras in Peacetopia. You have several decisions to make regarding the evaluation metric and how to structure your data into train/dev/test sets.

3.1.4.1 Question 1

The City Council specifies that they want an algorithm that:

- Has high accuracy.
- Operates quickly and takes only a short time to classify a new image.
- Requires minimal memory, allowing it to run on a small processor attached to various security cameras.

You discuss with them the need for a singular evaluation metric to guide development.

True or False: Choosing one primary metric simplifies decision-making and enhances development speed, allowing for clearer comparisons between different models.

✓ True

✗ False

3.1.4.2 Question 2

After further discussions, the city narrows down its criteria to:

- We need an algorithm that can let us know a bird is flying over Peacetopia as accurately as possible.
- We want the trained model to take no more than 10 seconds to classify a new image.
- We want the model to fit in 10MB of memory.
- We require a minimum of 98% test accuracy.

If you had the three following models, which one would you choose?

- ✓ Test Accuracy: 98%; Runtime: 9 sec; Memory size: 9MB
- ✗ Test Accuracy: 99%; Runtime: 13 sec; Memory size: 9MB
- ✗ Test Accuracy: 97%; Runtime: 1 sec; Memory size: 3MB
- ✗ Test Accuracy: 97%; Runtime: 3 sec; Memory size: 2MB

3.1.4.3 Question 3

Based on the context of a city's data analysis project, which of the following statements is true regarding the metrics used?

- ✓ Accuracy is an optimizing metric; running time and memory size are satisficing metrics.
- ✗ Accuracy, running time, and memory size are all optimizing metrics because you want to do well on all three.
- ✗ Accuracy, running time, and memory size are all satisficing metrics because you have to do sufficiently well on all three for your system to be acceptable.
- ✗ Accuracy is a satisficing metric; running time and memory size are an optimizing metric.

3.1.4.4 Question 4

Before implementing your algorithm, you need to split your data into train/dev/test sets. You have a dataset of 10,000,000 examples.

Which of these do you think is the best choice?

- ✓ Train: 9,500,000; Dev: 250,000; Test: 250,000 (With a large dataset, smaller dev and test sets are sufficient for evaluating bias and variance.)
- ✗ Train: 3,333,334; Dev: 3,333,333; Test: 3,333,333
- ✗ Train: 6,000,000; Dev: 1,000,000; Test: 3,000,000
- ✗ Train: 6,000,000; Dev: 3,000,000; Test: 1,000,000

3.1.4.5 Question 5

Now that you've set up your train/dev/test sets, the City Council comes across another 1,000,000 images from social media and offers them to you. These images have a different distribution from the images the City Council originally provided, but you think they could help your algorithm.

Should you add this data to the training set?

✓ Yes (This will cause the training and dev/test set distributions to become different. However, as long as the dev/test distributions are the same, you are aiming at the same target.)

✗ No

Here,

- The training set is used to help the model learn useful patterns from as much data as possible, so it can learn better features. For this reason, the training set can come from a different distribution than the dev and test sets, and in many cases the model benefits from having more diverse training examples.
- In contrast, the dev and test sets should come from the same distribution, since they define the target for model evaluation and selection. They need to be consistent with each other to be able to measure generalization performance.

3.1.4.6 Question 6

One member of the City Council wants to add 1,000,000 citizen data images to the development (dev) set. Your original data is from security cameras.

You object to adding the citizen data because: (Choose all that apply)

✓ The dev set no longer reflects the distribution of data (security cameras) you most care about. (The performance of the model should be evaluated on the same distribution of images it will see in production.)

✓ This would cause the dev and test set distributions to become different. This is a bad idea because you're not aiming where you want to hit. (Adding a different distribution to the dev set will skew bias.)

✗ The 1,000,000 citizen data images do not have a consistent input-output relationship as the security camera data.

✗ A bigger test set will slow down the speed of iterating because of the computational expense of evaluating models on the test set.

3.1.4.7 Question 7

You train a system, and the train/dev set errors are 3.5% and 4.0% respectively. You decide to try regularization to close the train/dev error gap.

Do you agree?

✓ No, because you do not know what the human performance level is. (You need to know what the human performance level is to estimate avoidable bias.)

- ✗ No, because this shows your variance is higher than your bias.
- ✗ Yes, because this shows your bias is higher than your variance.
- ✗ Yes, because having a 4.0% training error shows you have a high bias.

Need to first focus on reducing avoidable bias (the gap between training error and human performance), and then focus on reducing variance (gap between dev and training error)

3.1.4.8 Question 8

You ask a few people to label a bird species dataset to determine human-level performance. The following error rates were recorded:

- Bird watching expert #1: 0.3% error
- Bird watching expert #2: 0.5% error
- Normal person #1 (not a bird watching expert): 1.0% error
- Normal person #2 (not a bird watching expert): 1.2% error

If your goal is to use "human-level performance" as an estimate for Bayes error, how would you define "human-level performance" in this scenario?

- ✓ 0.3% (The lowest error rate achieved by an expert) (The best performance of an expert is the closest practical estimate of Bayes error.)
- ✗ 0.75% (Average of all four error rates)
- ✗ 0.0% (Perfect accuracy, representing an unattainable ideal)
- ✗ 0.4% (Average of the two experts' error rates)

3.1.4.9 Question 9

Which of the following statements do you agree with?

- ✓ A learning algorithm's performance can be better than human-level performance but it can never be better than Bayes error. (This statement accurately reflects the relationship between the learning algorithm, human-level performance, and Bayes error.)
- ✗ A learning algorithm's performance can never be better than human-level performance nor better than Bayes error.
- ✗ A learning algorithm's performance can never be better than human-level performance but it can be better than Bayes error.
- ✗ A learning algorithm's performance can be better than human-level performance and better than Bayes error.

3.1.4.10 Question 10

Given the following performance metrics

- Human-level performance: 0.1%
- Training set error: 2.0%
- Dev set error: 2.1%

Which of the following best describes the most effective next step in your project?

- ✓ Prioritize actions to decrease bias by increasing model complexity, as the training error significantly exceeds human-level performance. (Addressing the largest performance gap (between human-level and training error) is the most efficient strategy.)
- ✗ Evaluate the test set to determine the variance.
- ✗ Continue tuning until the training set error matches human-level performance, focusing solely on the optimizing metric.
- ✗ Deploy the model to target devices to evaluate against satisficing metrics.

3.1.4.11 Question 11

You've now also run your model on the test set and find that the error rate is 7.0% compared to a 2.1% error rate for the dev set.

What should you do? (Choose all that apply)

- ✓ Increase the size of the dev set. (A larger dev set can help provide a more accurate estimate of model performance and reduce overfitting.)
- ✓ Try increasing regularization to reduce overfitting to the dev set. (Increasing regularization can help reduce overfitting to the dev set.)
- ✗ Get a bigger test set to increase its accuracy.
- ✗ Try decreasing regularization for better generalization with the dev set.

3.1.4.12 Question 12

After working on this project for a year, you finally achieve:

- Human-level performance: 0.1%
- Training set error: 0.05%
- Dev set error: 0.05%

What can you conclude? (Check all that apply.)

- ✓ If the test set is big enough for the 0.05% error estimate to be accurate, this implies Bayes error is ≤ 0.05. (Bayes error is the theoretical minimum error, and if the test error is accurate, it implies the Bayes error is at or below that level.)
- ✗ It is now harder to measure avoidable bias, thus progress will be slower going forward.
- ✗ With only 0.05% further progress to make, you should quickly be able to close the remaining gap to 0%.
- ✗ It is highly unlikely this result is purely a statistical anomaly, but statistical noise may still contribute to the error. (While rare, surpassing human-level performance is possible, so it's unlikely to be purely a statistical anomaly.)

3.1.4.13 Question 13

Your system is now very accurate but has a higher false negative rate than the City Council of Peacetopia would like.

What is your best next step?

- ✓ Reset your "arget" (metric) for the team and tune to it. (The target has shifted so an updated metric is required.)
- ✗ Look at all the models you've developed during the development process and find the one with the lowest false negative error rate.
- ✗ Expand your model size to account for more corner cases.
- ✗ Pick false negative rate as the new metric, and use this new metric to drive all further development.

3.1.4.14 Question 14

You've handily beaten your competitor, and your system is now deployed in Peacetopia and is protecting the citizens from birds! But over the last few months, a new species of bird has been slowly migrating into the area, so the performance of your model is being tested on a new type of data.

There are only 1,000 images of the new species. The city expects a better system from you within the next 3 months.

Which of these should you do first?

- ✓ Augment your data to increase the number of images of the new bird species. (Generating a sufficient number of images of the new species is crucial for your model to learn its features effectively.)
- ✗ Add hidden layers to further refine feature development.

✗ Add the new images and split them among train/dev/test.

✗ Put the 1,000 images into the dev set to evaluate the bias and re-tune.

3.1.4.15 Question 15

The City Council thinks that having more cats in the city would help scare off birds. They are so happy with your work on the Bird detector that they also hire you to build a Cat detector.

Because of years of working on Cat detectors, you have such a huge dataset of 100,000,000 cat images that training on this data takes about two weeks.

Which of the statements do you agree with? (Check all that agree.)

✓ Buying faster computers could speed up your team's iteration speed and thus your team's productivity. (Enhanced computational resources can reduce training time and improve productivity.)

✓ If 100,000,000 examples is enough to build a good enough Cat detector, you might be better off training with just 10,000,000 examples to gain a $\sim 10\times$ improvement in how quickly you can run experiments, even if each model performs a bit worse because it's trained on less data. (A smaller dataset can expedite training and allow for more iterations, potentially leading to a satisfactory model faster.)

✓ Needing two weeks to train will limit the speed at which you can iterate. (The long training time constrains how quickly you can test and refine models.)

✗ Having built a good Bird detector, you should be able to take the same model and hyperparameters and just apply it to the Cat dataset, so there is no need to iterate.

3.2 Module 2 - ML Strategy

3.2.1 Error Analysis

3.2.1.1 Carrying Out Error analysis

Error analysis is the process of manually examining a machine learning model's mistakes when it has not yet reached human-level performance. It helps identify the most promising improvements and supports better prioritization.

Process of error analysis:

- Collect examples from the dev set that are misclassified by the algorithm (i.e., false positives and false negatives).

  For example, images incorrectly classified as containing a cat in a cat detection task.

- Categorize the errors for each example (this can be done while evaluating multiple ideas in parallel).

  For example, in a cat detection task, errors can be labeled using categories in a table such as:

Example Label	Dog	Great Cat	Blurry	Instagram Filter	Comments
1	Y/N	Y/N	Y/N	Y/N	
2	Y/N	Y/N	Y/N	Y/N	
3	Y/N	Y/N	Y/N	Y/N	
⋮	⋮	⋮	⋮	⋮	
% of total	%	%	%	%	

- Compute the percentage of examples in each category to determine which types of errors are worth addressing.

3.2.1.2 Cleaning Up Incorrectly Labeled Data

- Training set:
  - For incorrectly labeled (i.e., ground-truth label is wrong) examples in training set, deep learning algorithms are quite robust to random errors in the training set. If the errors are reasonably random, it is usually fine to leave them as they are.

  - However, deep learning algorithms are less robust to systematic errors. E.g., if a cat detection model consistently mislabels white dogs as cats, the model will learn this incorrect pattern and misclassify all white colored dogs as cats.

- Dev/test set:
  - For incorrectly labeled examples in the dev/test set, add an extra column during error analysis for justifying the label (i.e., indicate whether the label y is correct).

Example Index	**Dog**	**Great Cat**	**Blurry**	**Incorrectly Labeled**	**Comments**
1	Y/N	Y/N	Y/N	Y/N	
2	Y/N	Y/N	Y/N	Y/N	
3	Y/N	Y/N	Y/N	Y/N	
⋮	⋮	⋮	⋮	⋮	
% of total	%	%	%	%	

    By comparing the overall dev error with errors caused by incorrect labels, one can decide whether fixing labels is worthwhile. If correcting labels significantly improves dev set performance, it is worthwhile; otherwise, it may not be the best use of time.

  - Guidelines for manually reviewing and fix labels in the dev/test sets:
    * Apply the same label correction process to both the dev and test sets to ensure they remain from the same distribution.
    * Consider reviewing both examples that the model got wrong and examples that the model got right (the latter is less commonly done), that is,
      · Manually review and fix labels for examples that the model got wrong.

        However, Only fixing the model's mistakes is easier, but it can bias the evaluation, since it tends to correct only examples that hurt the model's performance and may therefore give the model an unfair advantage.

      · Examples that the model got right may also be mislabeled and should be corrected.

        However, this is often not done because it can be very time-consuming. When model accuracy is high (e.g., 98%), there are far more correctly classified examples than incorrectly classified ones, making it very time-consuming to inspect all correctly classified examples and correct any mislabeled data.

In practice, labels are often corrected only in the dev and test sets, since fixing a large training set is time-consuming. This is acceptable because learning algorithms are robust to training distribution differences, but the dev and test sets must remain from the same distribution.

Deep learning may seem like just feeding data into a model, but in practice, building good systems still requires manual error analysis and human insight. Some engineers hesitate to manually inspect data, but reviewing even a small number of examples can reveal important patterns in the model's mistakes. Spending a short amount of time on this can provide valuable guidance on what to improve next and is a highly effective use of time when prioritizing ideas.

3.2.1.3 Build First System Quickly, Then Iterate

For most machine learning problems, there are many possible directions to improve the system, but the key challenge is deciding which direction to focus on.

A good strategy when starting a new machine learning application is to build an initial system quickly and then iterate, i.e.,

- Set up dev/test sets and the metric
- Build initial system quickly
- Use bias/variance analysis and error analysis to prioritize next steps

3.2.2 Mismatch Training and Dev/Test Set

3.2.2.1 Training and Testing on Different Distributions

In modern deep learning, it is common to train models on data from a different distribution than the dev and test sets. Handling this setup requires careful design and best practices.

For example,

- Consider a cat classifier with:
  - 200,000 images from webpages
  - 10,000 images from a mobile app (which are blurrier)
- One possible data split is:
  - Training set: 200,000 webpage images + 5,000 mobile images
  - Dev set: 2,500 mobile images
  - Test set: 2,500 mobile images
- Advantage and disadvantage:
  - Advantage: The dev and test sets reflect the target (mobile app) distribution, so evaluation is realistic.
  - Disadvantage: The training data comes from a different distribution than the dev and test sets.

3.2.2.2 Bias and Variance with Mismatched Data Distributions

Estimating bias and variance helps determine what to improve next. However, this analysis becomes more complex when the training set comes from a different distribution than the dev and test sets.

For example, consider a cat classifier with $\approx 0\%$ human-level error, 1% training error, and 10% dev error. The large gap between training and dev error could be due to:

- High variance: the model does not generalize well from the training set to unseen data in the dev set.
- Data mismatch: the dev set comes from a different distribution from the training set

To distinguish between these two effects, a training-dev set is introduced by randomly sampling a subset from the training set. This ensures that the training-dev set has the same distribution as the training set, and is not used for training. Error analysis can then be performed on the training set, training-dev set, and dev set, to identify whether the issue is high variance or data mismatch.

- Example 1:

Human	0%	0%	0%	0%
Training	1%	1%	10%	10%
Training-dev	9%	1.5%	11%	11%
Dev	10%	10%	12%	20%
	Variance	Data mismatch	Avoidable bias	Avoidable bias & data mismatch

- Example 2:

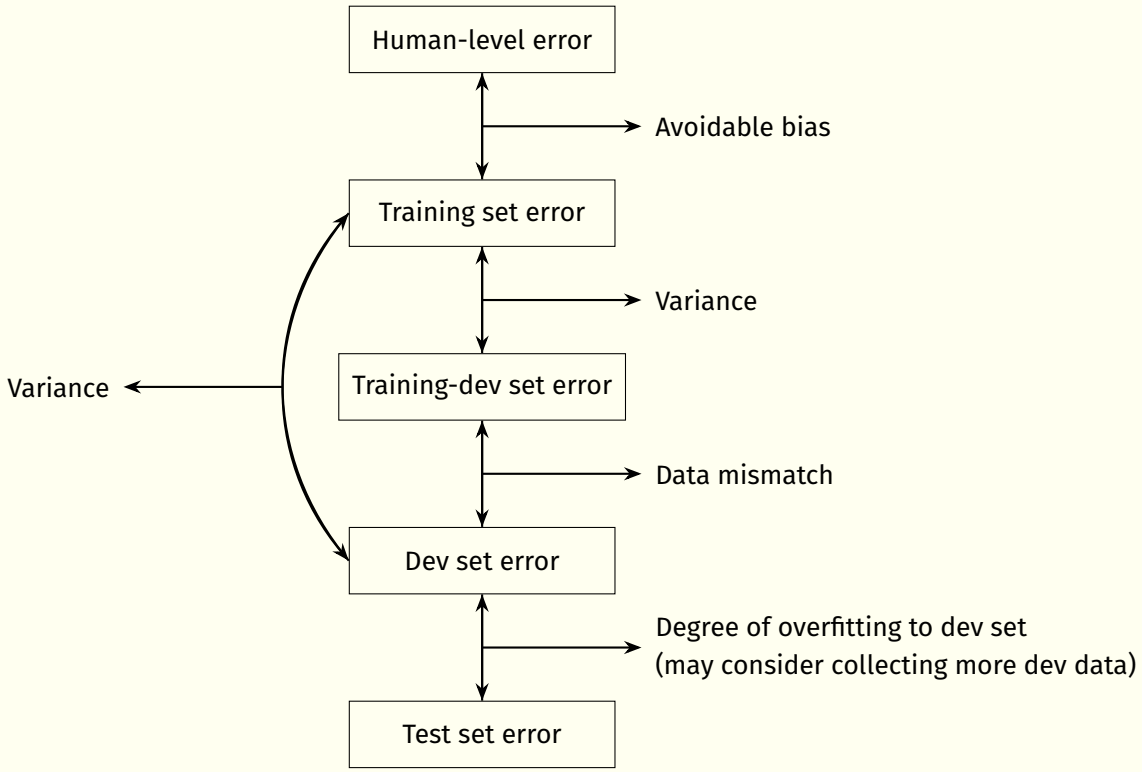
Human	4%	4%
Training	7%	7%
Training-dev	10%	10%
Dev	12%	6%
Test	12%	6%

  Error values often increase as moving down the table (e.g., in the first column), but this is not always the case (e.g., in the second column). This can happen when the training data is harder than the dev/test data, causing training error to be higher while dev/test errors are lower.

- Example 3 (speech recognition, rearview mirror speech data):

	General speech recognition	**Rearview mirror speech data**
Human-level error	Human-level error 4%	Human-level error 6%
Error on examples trained on	Training error 7%	Training error 6%
Error on examples not trained on	Training-dev error 10%	Dev/test error 6%

More general,



3.2.2.3 Addressing Data Mismatch

Training and testing on different distributions can lead to a data mismatch problem during error analysis.

There is no completely systematic way to address data mismatch. Here is a rough guideline (no guarantee) of things that can be tried:

- Carry out manual error analysis to try to understand difference between training and dev/test sets (e.g., noisy)
- Make the training data more similar to dev/test sets, or collect more data similar to dev/test sets (e.g., through artificial data synthesis). When using artificial data synthesis, be careful not to generate data from only a small subset of the space of all possible examples.

3.2.3 Learning from Multiple Tasks

3.2.3.1 Transfer Learning

Transfer learning refers to using knowledge of a neural network learned from one task to solve a different task.

- For example, a model trained for image recognition can be fully or partially reused for radiology diagnosis. Specifically,
  - Data:
    - Image recognition: $(x, y) = (\text{image}, \text{Y/N})$
    - Radiology diagnosis: $(x, y) = (\text{X-ray image}, \text{diagnosis})$
  - Remove the original output layer (i.e., all units of the output layer and the weights feeding into the output layer) and replace it with a new output layer with randomly initialized weights.
  - Retrain only the new output layer (i.e., weights of output layer) while keeping the rest of layers (i.e., parameters of earlier layers) fixed.

  This works because early layers learn general features (e.g., edges, curves, shapes) from large image datasets, which can also be useful for medical images.
- Definitions:
  - Training on the original task is called
  - Training on the new task is called fine-tuning
- Rule of thumb:
  - Small dataset $\rightarrow$ retrain only the last layer (or last few layers).
  - Large dataset $\rightarrow$ retrain the entire network (all parameters).
- When transfer learning (from task A to task B) makes sense:
  - Task A and task B have the same type of input x.
  - There is much more data for task A than task B.
  - Low level features from task A are helpful for learning task B.

3.2.3.2 Multi-task Learning

- Comparison of transfer learning and multi-task learning:
  - Transfer learning is a sequential approach: a model is first trained on one task (Task A), and the learned knowledge is then transferred to another task (Task B).
  - In contrast, multi-task learning trains a single neural network on multiple tasks simulta-

neously, rather than training separate models for each task, allowing shared learning to improve performance across all tasks.

- In practice, transfer learning is used more often than multi-task learning.

- Example:
  - Multi-object detection in autonomous driving involves recognizing different objects (e.g., pedestrians, cars, stop signs, and traffic lights). If some of the early features in a neural network are shared across these different object types, then training a single neural network to perform all tasks (identify each object) can achieve better performance than training multiple separate neural networks. This shared representation allows the model to learn more efficiently and generalize better across tasks, illustrating the power of multi-task learning.
  - In addition, multi-task learning still works when some images have only a subset of labels (with missing labels indicated as question marks), or when some images contain extra unrelated labels. In such cases, only the available labels are used in the loss computation during training, and the missing labels are ignored.
- When multi-task learning makes sense:
  - Training on a set of tasks that could benefit from having shared lower-level features.
  - Usually (less of a hard and fast rule): Amount of data you have for each task is quite similar.
  - Can train a big enough neural network to do well on all the tasks.

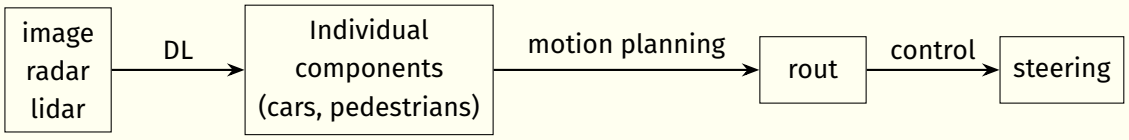
    Researchers have found that multi-task learning may perform worse than training separate neural networks only when the neural network is not large enough to handle all the tasks effectively.

3.2.4 End-to-end Deep Learning

One of the most exciting recent developments in deep learning is end-to-end learning.

- Traditional systems are often built as multiple processing stages, while end-to-end learning uses a single neural network to map inputs directly to outputs, e.g, speech recognition: audio → transcript.
- However, end-to-end methods usually require large datasets. When data is limited, In practice, breaking the problem into smaller sub-problems can work better than solving everything in one step.
  - Example 1: Face recognition (image → identity)
    - Step 1: Detect the face region in the image (image → cropped face image)
    - Step 2: Identify the person using a neural network by comparing this face image with known image (face image → identity).

  - Example 2: Bone age estimation (X-ray image → child age)
    * Step 1: Detect specific bone regions (X-ray image → bone segments)
    * Step 2: Measure bone lengths, then estimate age by comparing with known statistics tables (bone segments → age)
  - Two main reasons why this decomposed approach can outperform a pure end-to-end model:
    * Simpler tasks: Each sub-problem is easier to learn than the full end-to-end problem
    * More data: There is often more data available for each sub-task than for the full problem
- Pros and cons of end-to-end deep learning:
  - Pros:
    * Let the data speak
    * Less hand-designing of components needed
  - Cons:
    * May need large amount of data
    * Excludes potentially useful hand-designed components (hand-designed components is less important when there is a ton of data)
- Guidelines for deciding whether an end-to-end approach is suitable: the key question is whether you have sufficient data to learn a function complex enough to map x to y?



Chapter 4

Course 4: Convolutional Neural Networks

4.1 Module 1 - Foundations of Convolutional Neural Networks

4.1.1 Convolutional Neural Networks (CNN/ConvNet)

4.1.1.1 Computer Vision

Computer vision problems,

- Image classification
- Object detection: identify multiple objects in an image, and figure out their positions by drawing boxes around them.
- Neural style transfer: a content image and a style image → repaint the content image with the style of the style image, to create new type of artwork

One of the challenges in computer vision is that the input image can be very large (raw pixels $\times$ 3 color channels), which makes them computationally expensive to process. To address this, the convolution operation is used, which is a fundamental building block of convolutional neural networks.

4.1.1.2 Convolution

In mathematics, convolution involves double flipping (both horizontally and vertically) the filter and then performing a cross-correlation. In deep learning, the flipping step is usually omitted, and only the cross-correlation is performed. By convention, this operation is still referred to as convolution.

Edge detection example:

- Convolution can be used in edge detection, i.e., detecting edges in an image, such as vertical, horizontal, or angled ones. It works by taking an image (a matrix) and applying a small filter (or kernel), which is an $f \times f$ matrix (where f is usually an odd number, e.g., $f = 3$, to provide a central position). The filter slides across the image, and computations at each position produce a new output image.
- Different filters can be used to detect different types of edges. Instead of manually choosing the filter values, the filter elements can be treated as learnable parameters in a machine learning model, which enable the model to learn the most effective filters directly from data.

4.1.1.3 Padding

Downsides of convolution:

- Output size shrinks. When an $n \times n$ image is convolved with an $f \times f$ filter, the output size becomes $(n - f + 1) \times (n - f + 1)$. Repeating this process multiple times can make the image shrink significantly.

- A lot of information from the corners and edges of the image is lost. Pixels at the edges and corners are used less often than those in the center. For example, a corner pixel is included in only one filter region, while a pixel in the center is included in many overlapping regions. As a result, edge and corner information is not captured as well as information from the middle of the image.

These issues can be solved or reduced by adding a border of p zero-valued pixels around the image before applying the filter, which is called padding. Here, p is called the padding amounts.

For an $n \times n$ image, after adding padding of size p on all sides and applying an $f \times f$ filter, the output size becomes $(n + 2p - f + 1) \times (n + 2p - f + 1)$.

Types of convolutions:

- Valid convolution: no padding is used,

$$\boxed{n \times n \text{ image}} \star \boxed{f \times f \text{ filter}} \xrightarrow{\text{no padding } (p = 0)} \boxed{(n - f + 1) \times (n - f + 1)}$$

- Same convolution: padding is added so that the output has the same size as the input,

$$\boxed{n \times n \text{ image}} \star \boxed{f \times f \text{ filter}} \xrightarrow{\text{padding } p = \frac{f-1}{2}} \boxed{n \times n}$$

4.1.1.4 Strided

In addition to padding, stride is another basic building block of the convolution in CNN.

Stride s is defined as the number of steps moving forward when applying filter to image.

$$\boxed{n \times n \text{ image}} \star \boxed{f \times f \text{ filter}} \xrightarrow[\text{stride } s]{\text{padding } p} \boxed{\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor}$$

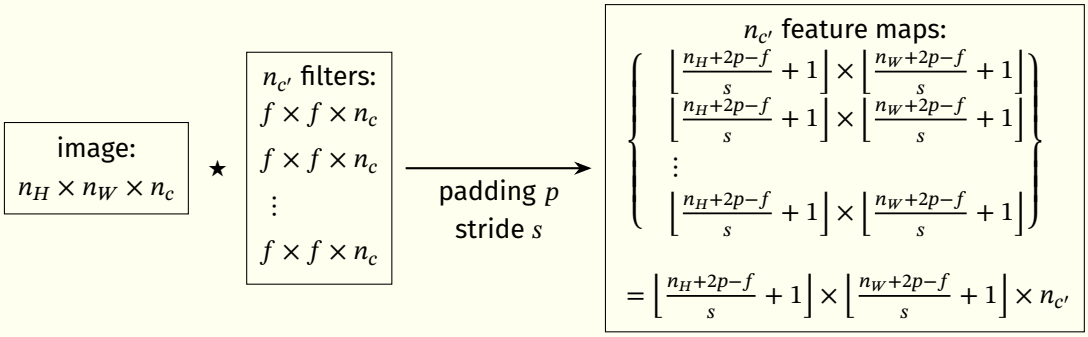
4.1.1.5 Convolutions Over Volume

In addition to convolution over a 2D data (e.g., a matrix $n_H \times n_W$ representing a grayscale image), convolution can be extended to 3D data (e.g., $n_H \times n_W \times n_c$ representing a color image with n_c being the number of channels (or depth) of an image) or higher-dimensional volumes.

- For a single filter (used to detect a specific type of edge in a given channel),

$$\boxed{\begin{array}{c}\text{image:} \\ n_H \times n_W \times n_c\end{array}} \star \boxed{\begin{array}{c}\text{filter:} \\ f \times f \times n_c\end{array}} \xrightarrow[\text{stride } s]{\text{padding } p} \boxed{\begin{array}{c}\text{feature map:} \\ \left\lfloor \frac{n_H+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W+2p-f}{s} + 1 \right\rfloor\end{array}}$$

- For multiple filters (used to detect different types of edges across different channels),



4.1.1.6 One Convolution Layer

Notation:

- $f^{[l]}$: filter size of layer l
- $p^{[l]}$: padding of layer l
- $s^{[l]}$: stride of layer l
- $n_c^{[l]}$: number of filters in layer l

Thus, for convolution layer l (for a single example),

- The activations have shape $a^{[l]} \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}}$.
- Each filter has shape $\in \mathbb{R}^{f^{[l]} \times f^{[l]} \times n_c^{[l-1]}}$.
- The weights (i.e., a set of filters) have shape $W^{[l]} \in \mathbb{R}^{f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}}$.

  Filters in CNN plays the same role as the weights $W^{[l]}$ in standard non CNN.
- The bias has shape $b^{[l]} \in \mathbb{R}^{n_c^{[l]}}$ (or equivalently $b^{[l]} \in \mathbb{R}^{1 \times 1 \times n_c^{[l]}}$).

  Here, each filter (or output channel) has its own bias term.
- The input has shape $\left(n_H^{[l-1]}, n_W^{[l-1]}, n_c^{[l-1]}\right)$.
- The output has shape $\left(n_H^{[l]}, n_W^{[l]}, n_c^{[l]}\right)$, with

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor, \quad n_W^{[l]} = \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

- The total number of parameters is $\left(f^{[l]} \cdot (f^{[l]} \cdot n_c^{[l-1]} + 1) \cdot n_c^{[l]}\right.$, which is independent of the input size.

  As a result, convolutional layers typically require significantly fewer parameters than fully connected layers, which helps reduce overfitting.

$$\boxed{\begin{array}{c}\text{input:}\\ a^{[l-1]} \in \mathbb{R}^{n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}}\end{array}} \star \boxed{\begin{array}{c}n_c^{[l]} \text{ filters:}\\ f^{[l]} \times f^{[l]} \times n_c^{[l-1]}\\ f^{[l]} \times f^{[l]} \times n_c^{[l-1]}\\ \vdots\\ f^{[l]} \times f^{[l]} \times n_c^{[l-1]}\end{array}}$$

$$\xrightarrow[\text{stride } s]{\text{padding } p} \boxed{\begin{array}{c}n_c^{[l]} \text{ feature maps:}\\ \left\{\begin{array}{l} a^{[l-1]} \star \text{filter}_1 \in \mathbb{R}^{\left\lfloor \frac{n_H^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}}+1 \right\rfloor \times \left\lfloor \frac{n_W^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}}+1 \right\rfloor} = \mathbb{R}^{n_H^{[l]} \times n_W^{[l]}} \\ a^{[l-1]} \star \text{filter}_2 \in \mathbb{R}^{\left\lfloor \frac{n_H^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}}+1 \right\rfloor \times \left\lfloor \frac{n_W^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}}+1 \right\rfloor} = \mathbb{R}^{n_H^{[l]} \times n_W^{[l]}} \\ \vdots \\ a^{[l-1]} \star \text{filter}_{n_c^{[l]}} \in \mathbb{R}^{\left\lfloor \frac{n_H^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}}+1 \right\rfloor \times \left\lfloor \frac{n_W^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}}+1 \right\rfloor} = \mathbb{R}^{n_H^{[l]} \times n_W^{[l]}} \end{array}\right\} \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}}\end{array}}$$

$$\xrightarrow{\text{bias } b^{[l]}} \boxed{\begin{array}{c}\text{output:}\\ \left\{\begin{array}{l} \text{Relu}\left(a^{[l-1]} \star \text{filter}_1 + b_1^{[l]}\right) \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]}} \\ \text{Relu}\left(a^{[l-1]} \star \text{filter}_2 + b_2^{[l]}\right) \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]}} \\ \vdots \\ \text{Relu}\left(a^{[l-1]} \star \text{filter}_{n_c^{[l]}} + b_{n_c^{[l]}}^{[l]}\right) \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]}} \end{array}\right\} = a^{[l]} \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}}\end{array}}$$

Types of layers in a ConvNet,

- Convolution (CONV)
- Pooling (POOL)
- Fully connected (FC)

4.1.1.7 Pooling Layers

In addition to convolutional layers, pooling layers provide another way to reduce the size of input and speed up computation. They also make feature detectors more robust, i.e., more invariant to their position in the input.

- Types of pooling:
  - Max pooling: Slides an $f \times f$ window over the input with stride s, taking the maximum value in each window.
  - Average pooling: Slides an $f \times f$ window over the input with stride s, taking the average value in each window.

Max pooling is generally used more often than average pooling in neural networks. However, in

very deep layers, average pooling sometimes is used to reduce (or collapse) the representation.

Pooling is computed independently on each channel. The operation can be viewed as applying an $f \times f$ window (similar to an $f \times f$ filter in convolution) to each channel.

$$\boxed{\begin{array}{c}\text{input:}\\ n_H \times n_W\end{array}} \xrightarrow[\substack{f \times f \text{ window}\\ \text{stride } s}]{\text{pooling}} \boxed{\begin{array}{c}\text{output:}\\ \left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor\end{array}}$$

- Hyperparameters of pooling:
  - filter size f
  - stride s
  - choice of pooling type (e.g., max pooling or average pooling)

  Padding is very rarely used in pooling layers (i.e., $p = 0$), especially in max pooling. Therefore, padding is typically not considered a hyperparameter for pooling.

  Common choices for f and s include $f = 2$ and $s = 2$, or $f = 3$ and $s = 2$.

A key property of pooling (both max pooling and average pooling) is that it has hyperparameters but no learnable parameters.

4.1.1.8 CNN Example

- One common CNN pattern:
  - After a series of (e.g., L) convolutional and pooling layers, the output has shape $\mathbb{R}^{n_H^{[L]} \times n_W^{[L]} \times n_c^{[L]}}$.
  - The output is then flattened from a volume of shape $\mathbb{R}^{n_H^{[L]} \times n_W^{[L]} \times n_c^{[L]}}$ into a vector of shape $\mathbb{R}^{n_H^{[L]} \cdot n_W^{[L]} \cdot n_c^{[L]} \times 1}$.
  - This vector is then passed into one (or more) fully connected layers, i.e., standard neural network layers, often followed by a softmax output layer.
- Another common CNN pattern is to stack several convolutional layers followed by a pooling layer, repeat this block multiple times, and then end with a few fully connected layers and a softmax output layer.

Note:

- By convention, a convolutional layer together with a pooling layer is often referred to as a single layer in a CNN.
- Usually, the spatial dimensions $n_H^{[l]}$ and $n_W^{[l]}$ decrease as the network goes deeper, while the depth $n_c^{[l]}$ increases. However, the reduction of $n_H^{[L]}$ and $n_W^{[L]}$ should be gradual. Otherwise, the CNN may not perform well if the dimensions shrink too quickly.
- Max pooling layers have no learnable parameters. Convolutional layers typically have relatively

few parameters compared to fully connected layers.

- A significant part of CNN design involves selecting hyperparameters (e.g. $f^{[l]}$, $p^{[l]}$, $s^{[l]}$, and $n_c^{[l]}$). A common guideline is to refer to the literature and use architectures that have worked well for others, rather than inventing hyperparameter settings from scratch. These proven architectures often transfer well to new applications.

4.1.1.9 Why Convolutions?

The main advantage of convolutional layers over fully connected layers is that they use far fewer parameters, for two main reasons:

- Parameter sharing: A feature detector (e.g., a vertical edge detector) that is useful in one part of the image is likely to be useful in other parts as well. Therefore, a single filter (i.e., one set of parameters) is applied across different spatial locations of the input image. This significantly reduces the number of parameters.

  This parameter sharing (especially in early layers) helps convolutional neural networks learn features that are more robust to small shifts in the input and contributes to capturing translation invariance. Specifically, if an image of a cat is shifted slightly (e.g., a few pixels to the right), it is still recognized as a cat because the same filter is applied across all spatial locations, allowing features detected in one region to also be detected in other regions.
- Sparsity of connections: In each layer, each output value depends only on a small local region of the input rather than the entire input. This local connectivity significantly reduces the number of parameters.

4.2 Module 2 - Deep Convolutional Models: Cases Studies

4.2.1 Case Studies

4.2.1.1 Classic Networks

It is not trivial and requires quite a bit of insight in terms of how to put the basic building blocks, i.e., convolutional layer, pooling layer, and fully connected layer, together to build an effective CNN. One of the best ways of getting intuition is learning from a number of concrete examples of how others have done it. And it turns out that a neural network architecture that works well on one computer vision task often works well on other tasks as well.

LeNet-5

The original goal of LeNet-5 is to recognize handwritten digits.

$$32 \times 32 \times 1 \xrightarrow[f=5,s=1,c_n=6]{CONV} 28 \times 28 \times 6 \xrightarrow[f=2,s=2]{AVE\text{-}POOL} 14 \times 14 \times 6$$

$$\xrightarrow[f=5,s=1,c_n=16]{CONV} 10 \times 10 \times 16 \xrightarrow[f=2,s=2]{AVE\text{-}POOL} 5 \times 5 \times 16$$

$$\xrightarrow{FC} 120 \xrightarrow{FC} 84 \rightarrow Softmax$$

LeNet-5 has about 6000 parameters.

AlexNet

$$227 \times 227 \times 3 \xrightarrow[f=11,s=4,c_n=96]{CONV} 55 \times 55 \times 96 \xrightarrow[f=3,s=2]{MAX\text{-}POOL} 27 \times 27 \times 96$$

$$\xrightarrow[f=5,s=1,p=2,c_n=256]{SAME\text{-}CONV} 27 \times 27 \times 256 \xrightarrow[f=3,s=2]{MAX\text{-}POOL} 13 \times 13 \times 256$$

$$\xrightarrow[f=3,s=1,p=1,c_n=384]{SAME\text{-}CONV} 13 \times 13 \times 384$$

$$\xrightarrow[f=3,s=1,p=1,c_n=384]{SAME\text{-}CONV} 13 \times 13 \times 384$$

$$\xrightarrow[f=3,s=1,p=1,c_n=256]{SAME\text{-}CONV} 13 \times 13 \times 256 \xrightarrow[f=3,s=2]{MAX\text{-}POOL} 6 \times 6 \times 256$$

$$\xrightarrow{FC} 4096 \xrightarrow{FC} 4096 \rightarrow Softmax$$

AlexNet has similar building blocks with LeNet-5, but it is much bigger than LeNet-5. AlexNet has about 60 million parameters.

VGG-16 VGG-16 simplifies the neural network architectures.

$$224 \times 224 \times 3 \xrightarrow[f=3,s=1,p=1,c_n=64]{SAME\text{-}CONV\times 2} 224 \times 224 \times 64 \xrightarrow[f=2,s=2]{MAX\text{-}POOL} 112 \times 112 \times 64$$

$$\xrightarrow[f=3,s=1,p=1,c_n=128]{SAME\text{-}CONV\times 2} 112 \times 112 \times 128 \xrightarrow[f=2,s=2]{MAX\text{-}POOL} 56 \times 56 \times 128$$

$$\xrightarrow[f=3,s=1,p=1,c_n=256]{SAME\text{-}CONV\times 3} 56 \times 56 \times 256 \xrightarrow[f=2,s=2]{MAX\text{-}POOL} 28 \times 28 \times 256$$

$$\xrightarrow[f=3,s=1,p=1,c_n=512]{SAME\text{-}CONV\times 3} 28 \times 28 \times 512 \xrightarrow[f=2,s=2]{MAX\text{-}POOL} 14 \times 14 \times 512$$

$$\xrightarrow[f=3,s=1,p=1,c_n=512]{SAME\text{-}CONV\times 3} 14 \times 14 \times 512 \xrightarrow[f=2,s=2]{MAX\text{-}POOL} 7 \times 7 \times 512$$

$$\xrightarrow{FC} 4096 \xrightarrow{FC} 4096 \rightarrow Softmax$$

VGG-16 has about 138 million parameters. VGG-19 is a bigger version of VGG-16. But VGG-16 does almost as well as VGG-19. Thus, a lot of people will use VGG-16.

4.2.1.2 ResNet

Residual block is the basic building block of ResNet (Residual Network). Residual blocks in ResNet allows one to train very deep neural networks. The main idea of a residual block is using a technique called skip connection that allows the activation from one layer feed into a nonsuccessive deeper layer. Here is an illustrated explanation of the skip connection. Usually, a standard neutral network is as follows,

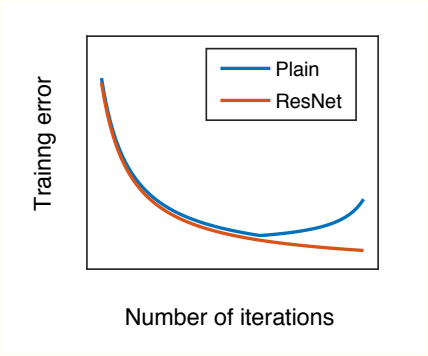
$$
\begin{aligned}
x = a^{[0]} &\xrightarrow[z^{[1]}=W^{[1]}a^{[0]}+b^{[1]}]{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[a^{[1]}=g^{[1]}(z^{[1]})]{\text{Activation: } g^{[1]}} \\
&\cdots \\
a^{[l-1]} &\xrightarrow[z^{[l]}=W^{[l]}a^{[l-1]}+b^{[l]}]{W^{[l]},b^{[l]}} z^{[l]} \xrightarrow[a^{[l]}=g^{[l]}(z^{[l]})]{\text{Activation: } g^{[l]}} \\
a^{[l]} &\xrightarrow[z^{[l+1]}=W^{[l+1]}a^{[l]}+b^{[l+1]}]{W^{[l+1]},b^{[l+1]}} z^{[l+1]} \xrightarrow[a^{[l+1]}=g^{[l+1]}(z^{[l+1]})]{\text{Activation: } g^{[l+1]}} \\
a^{[l+1]} &\xrightarrow[z^{[l+2]}=W^{[l+2]}a^{[l+1]}+b^{[l+2]}]{W^{[l+2]},b^{[l+2]}} z^{[l+2]} \xrightarrow[a^{[l+2]}=g^{[l+2]}(z^{[l+2]})]{\text{Activation: } g^{[l+2]}} \\
a^{[l+2]} &\xrightarrow[z^{[l+2]}=W^{[l+2]}a^{[l+1]}+b^{[l+2]}]{W^{[l+2]},b^{[l+2]}} z^{[l+2]} \xrightarrow[a^{[l+2]}=g^{[l+2]}(z^{[l+2]})]{\text{Activation: } g^{[l+2]}} \\
&\cdots \\
a^{[L-1]} &\xrightarrow[z^{[L]}=W^{[L]}a^{[L-1]}+b^{[l]}]{W^{[L]},b^{[L]}} z^{[L]} \xrightarrow[a^{[L]}=g^{[L]}(z^{[L]})]{\text{Activation: } g^{[L]}} a^{[L]}.
\end{aligned}
$$

Here, the process in blue color is called main pain. Replace the process in blue color with the following process in red color, which called short cut (or skip connection).

$$
\begin{aligned}
x = a^{[0]} &\xrightarrow[z^{[1]}=W^{[1]}a^{[0]}+b^{[1]}]{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[a^{[1]}=g^{[1]}(z^{[1]})]{\text{Activation: } g^{[1]}} \\
&\dots \\
a^{[l-1]} &\xrightarrow[z^{[l]}=W^{[l]}a^{[l-1]}+b^{[l]}]{W^{[l]},b^{[l]}} z^{[l]} \xrightarrow[a^{[l]}=g^{[l]}(z^{[l]})]{\text{Activation: } g^{[l]}} \\
a^{[l]} &\xrightarrow[z^{[l+1]}=W^{[l+1]}a^{[l]}+b^{[l+1]}]{W^{[l+1]},b^{[l+1]}} z^{[l+1]} \xrightarrow[a^{[l+1]}=g^{[l+1]}(z^{[l+1]})]{\text{Activation: } g^{[l+1]}} \\
a^{[l+1]} &\xrightarrow[z^{[l+2]}=W^{[l+2]}a^{[l+1]}+b^{[l+2]}]{W^{[l+2]},b^{[l+2]}} z^{[l+2]} \xrightarrow[a^{[l+2]}=g^{[l+2]}(z^{[l+2]}+a^{[l]})]{\text{Activation: } g^{[l+2]}} \\
a^{[l+2]} &\xrightarrow[z^{[l+2]}=W^{[l+2]}a^{[l+1]}+b^{[l+2]}]{W^{[l+2]},b^{[l+2]}} z^{[l+2]} \xrightarrow[a^{[l+2]}=g^{[l+2]}(z^{[l+2]})]{\text{Activation: } g^{[l+2]}} \\
&\dots \\
a^{[L-1]} &\xrightarrow[z^{[L]}=W^{[L]}a^{[L-1]}+b^{[l]}]{W^{[L]},b^{[L]}} z^{[L]} \xrightarrow[a^{[L]}=g^{[L]}(z^{[L]})]{\text{Activation: } g^{[L]}} a^{[L]}.
\end{aligned}
$$

Notice that $a^{[l]}$ is being injected after the linear part but before the activation part.

Here is a figures showing the dependance of the training error on the number of layers, for both plain neural network and ResNet.



Theoretically, the performance of a plain neural network on the training set will be better and better as the neural network becomes deeper and deeper. However, empirically, as the increase of the number of layers, the training error tends to decrease for a while and then increase. Thus, the training error gets worse a too deep plain neural network is used. Because having a plain network that is very deep means that the optimization algorithm has a much harder time training. But what happened to ResNet is that the training error keep going down as the number of layers increases. The skip connection in ResNet helps with the vanishing and exploding gradient problems and thus allows one train much deeper neural networks without really appreciable loss in performance. Notice that maybe at some point, the training error will reach a plateau. This means that a much deeper ResNet will not help to increase the performance.

4.2.1.3 Why ResNets Work

As it known that doing well on the training set is usually a prerequisite to doing well on the test set. Here are some intuitions about why ResNets work so well, in the sense of how the ResNet can be very deep without hurting its performance on the training set.

Take the following deep plain neutral network and deep ResNet with large l as examples,

$$\text{Plain NN: } x = a^{[0]} \xrightarrow[z^{[1]}=W^{[1]}a^{[0]}+b^{[1]}]{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[a^{[1]}=g^{[1]}(z^{[1]})]{\text{Activation: } g^{[1]}} a^{[1]} \cdots a^{[l-1]} \xrightarrow[z^{[l]}=W^{[l]}a^{[l-1]}+b^{[l]}]{W^{[l]},b^{[l]}} z^{[l]} \xrightarrow[a^{[l]}=g^{[l]}(z^{[l]})]{\text{Activation: } g^{[l]}} a^{[l]},$$

$$\begin{aligned}\text{ResNet: } x = a^{[0]} &\xrightarrow[z^{[1]}=W^{[1]}a^{[0]}+b^{[1]}]{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[a^{[1]}=g^{[1]}(z^{[1]})]{\text{Activation: } g^{[1]}} a^{[1]} \cdots a^{[l-1]} \xrightarrow[z^{[l]}=W^{[l]}a^{[l-1]}+b^{[l]}]{W^{[l]},b^{[l]}} z^{[l]} \xrightarrow[a^{[l]}=g^{[l]}(z^{[l]})]{\text{Activation: } g^{[l]}} a^{[l]} \\ &\xrightarrow[z^{[l+1]}=W^{[l+1]}a^{[l]}+b^{[l+1]}]{W^{[l+1]},b^{[l+1]}} z^{[l+1]} \xrightarrow[a^{[l+1]}=g^{[l+1]}(z^{[l+1]})]{\text{Activation: } g^{[l+1]}} a^{[l+1]} \\ &\xrightarrow[z^{[l+2]}=W^{[l+2]}a^{[l+1]}+b^{[l+2]}]{W^{[l+2]},b^{[l+2]}} z^{[l+2]} \xrightarrow[a^{[l+2]}=g^{[l+2]}(z^{[l+2]}+a^{[l]})]{\text{Activation: } g^{[l+2]}} a^{[l+2]}.\end{aligned}$$

When the activation function is ReLU function, and $W^{[l+2]} = 0$, $b^{[l+2]} = 0$,

$$\begin{aligned}a^{[l+2]} &= g^{[l+2]}(z^{[l+2]} + a^{[l]}) \\ &= g^{[l+2]}(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}) \\ &= g^{[l+2]}(a^{[l]}) \\ &= a^{[l]}.\end{aligned}$$

The identity function, i.e., ReLU, is easy for residual block to learn. It's easy to get $a^{[l+2]} = a^{[l]}$ because of skip connection. Thus, the ResNet just copy $a^{[l]}$ to $a^{[l+2]}$ despite the addition of these two layers. This means that adding these two layers to the neural network doesn't really hurt the performance of neural network compared with plain network without these two layers. Moreover, adding this two layers not only doesn't hurt the performance, but also helps improve the performance, because one could expect that something useful might be learned by these extra layers.

Notice that here assume that $z^{[l+2]}$ and $a^{[l]}$ have the same dimension, thus $z^{[l+2]} + a^{[l]}$ makes sense. Thus, same convolutions, which can preserve dimensions, is usually used in ResNet. In case that the input and output activation have different dimensions, for example, after a mx pooling layer. A extra matrix is added to convert the dimension of input activation. Here,

$$a^{[l+2]} = g^{[l+2]}(z^{[l+2]} + W_s a^{[l]}).$$

And the matrix W_s could be would be a matrix of parameters to be learned, or it could be a fixed matrix that just implements zero paddings for $a^{[l]}$.

4.2.1.4 Inception Network Motivation

4.2.1.5 Inception Network

For a image with one-channel, what 1×1 convolution does is multiplying by a number for each position of the image. While, for a image with multi-channel, what 1×1 convolution does makes more sense. Assume n_c is the number of channels of the image. 1×1 convolution puts n_c weights to the same position but different channels of the image, then sums them all and applies to a ReLU function. 1×1 convolution is also called network in network.

The pooling layer could shrink the height and width of the volume, while 1×1 convolution could shrink the number of channels of the volume, by using a number of 1×1 filters. This helps to save on computation in some neural networks. If the number of 1×1 filters used is the same as the channel of the input image, it also has an effect of adding non-linearity, which allows one learn more complex function of a neural network. 1×1 convolution is very useful for building the Inception network.

Here is an example of how 1×1 convolution can reduce the computation cost. For the following process,

$$28 \times 28 \times 192 \xrightarrow[f=5,s=1,p=2,c_n=32]{SAME\text{-}CONV} 28 \times 28 \times 32.$$

The computation cost is $(28 \cdot 28 \cdot 32) \cdot (5 \cdot 5 \cdot 192) = 120422400 \approx 120$ million. By adding a 1×1 convolution, as follows,

$$28 \times 28 \times 192 \xrightarrow[f=1,s=1,c_n=16]{SAME\text{-}CONV} 28 \times 28 \times 16 \xrightarrow[f=5,s=1,p=2,c_n=32]{SAME\text{-}CONV} 28 \times 28 \times 32.$$

The computation cost is $(28 \cdot 28 \cdot 16) \cdot (1 \cdot 1 \cdot 192) + (28 \cdot 28 \cdot 32) \cdot (5 \cdot 5 \cdot 16) = 12443648 \approx 12.4$ million. In fact, 1×1 convolution can be used as a bottleneck layer to reduce the computational costs by about a factor of 10. And as long as the bottleneck layer is used reasonably, it can reduce the computation cost significantly without hurting the performance of the neural network.

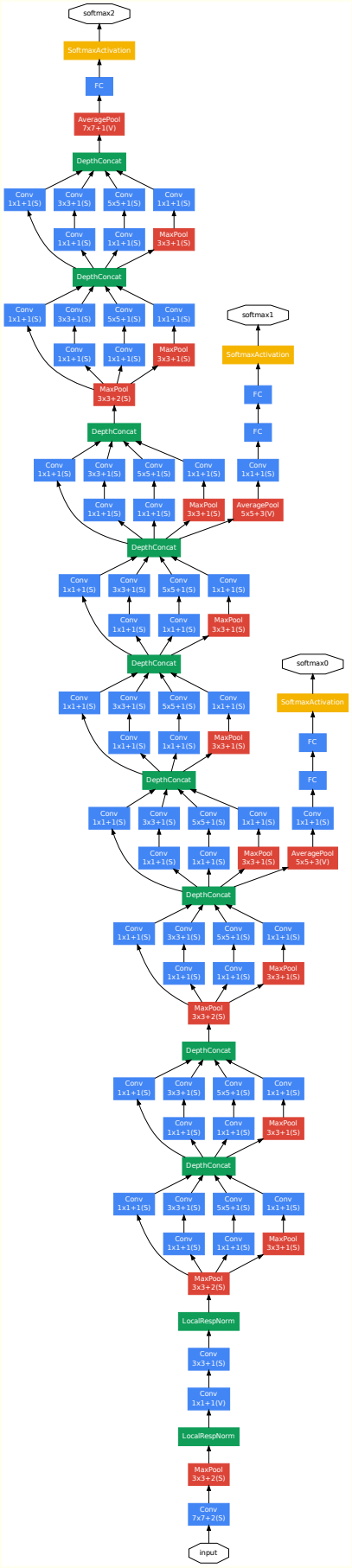
There is a list of consideration when designing a layer for a ConvNet, such as, what size of the filter (1×1, 3×3, or 5×5) to use, whether to include a pooling layer or not, etc. Instead of choosing one or some from the list, the Inception networks do them all and learn whatever parameters should be by itself. This makes the Inception network architecture more complicated but works remarkably well. For example, an Inception layer could stack up a 1×1 same convolution, a 3×3 same convolution, a 5×5 same convolution, a max pooling layer. Same convolution is used here to make the dimensions match when stacking up. And padding is needed for max pooling to make all dimensions match.

Here is an example of one Inception module,

$$28 \times 28 \times 192 \rightarrow \left\{ \begin{array}{l} \xrightarrow[f=1,s=1,c_n=64]{SAME\text{-}CONV} 28 \times 28 \times 64 \\ \xrightarrow[f=1,s=1,c_n=96]{SAME\text{-}CONV} 28 \times 28 \times 96 \xrightarrow[f=3,s=1,p=1,c_n=128]{SAME\text{-}CONV} 28 \times 28 \times 128 \\ \xrightarrow[f=1,s=1,c_n=16]{SAME\text{-}CONV} 28 \times 28 \times 16 \xrightarrow[f=5,s=1,p=2,c_n=32]{SAME\text{-}CONV} 28 \times 28 \times 32 \\ \xrightarrow[f=3,s=1,p=1,c_n=32]{SAME\text{-}MAX\text{-}POOL} 28 \times 28 \times 192 \xrightarrow[f=1,s=1,c_n=32]{SAME\text{-}CONV} 28 \times 28 \times 32 \end{array} \right\}$$

$$\xrightarrow{\text{Channel Concat}} 28 \times 28 \times 256.$$

Inception network is built by putting a bounch of Inception modules together. Notice that Inception network might also have some additional side-branches, which takes some hidden layer and pass it through a few fully connected layers and then a softmax to output a prediction. This helps to ensure that the features computed even at intermediate layers are not too bad for predicting the output, which appears to have a regularizing effect on the inception network and helps prevent this network from overfitting. Here is an example of an Inception network, GoogLeNet developed by Google,


softmax2
SoftmaxActivation
FC
AveragePool 7x7+1(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
softmax1
SoftmaxActivation
FC
FC
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
softmax0
SoftmaxActivation
FC
FC
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
LocalRespNorm
Conv 3x3+1(S)
Conv 1x1+1(V)
LocalRespNorm
MaxPool 3x3+2(S)
Conv 7x7+2(S)
input

The combination of Inception network and ResNet sometimes works even better.

4.2.1.6 MobileNet

4.2.1.7 MobileNet Architecture

4.2.1.8 EfficientNet

4.2.2 Practical Advice for Using ConvNets

4.2.2.1 Using Open-Source Implementation

Here are some practical advices on how to build a convolutional neural network for computer vision system.

- Using open source implementations.

  A lot of convolutional neural networks as introduced before are difficult or finicky to replicate because a lot of details about tuning of the hyperparameters such as learning decay and other things that make some difference to the performance. Fortunately, a lot of deep learning researchers routinely open source their work on the Internet, such as on GitHub. Thus, it is recommended to look online for an open source implementation when you see a research paper whose results you would like to build on top of. Because if you can get the author's implementation, you can usually get going much faster than if you would try to reimplement it from scratch.

  ```
  git clone http://...
  ```

- Transfer learning

  When building a computer vision application, you often make much faster progress if you transfer someone else's already trained network architecture as pre-training very good initialization to your own task rather than training a neural network from scratch and random initialization.

  The computer vision research community has been pretty good at posting lots of datasets online. Take the classifier with a small training set as an example. Download an open open source implementation of a neural network, both the code and the weights. Replace the original softmax layer with you own one, because your classifier might have different number of classes. And freeze all other layers as well as the parameters in all of these layers. Then only train parameters associated with softmax layer. By using someone else's free pre-trained weights, you might probably get pretty good performance even with a small data set. In fact, there are some fixed function that doesn't change because of the frozen layers. One trick to speed up the training is pre-compute activations of those layers for all examples in training set. When you have a larger training set, just freeze first fewer layers, and train other later layers with original parameters as initialization, or built your own new later layers and train them. The more data you have, the less number of frozen layers. If you have a lot of data, one thing you might do is take this open

source network and ways and use the whole thing just as initialization and train the whole network. In all the different disciplines, in all the different applications of deep learning, I think that computer vision is one where transfer learning is something that you should almost always do unless, you have an exceptionally large data set to train everything else from scratch yourself. But transfer learning is just very worth seriously considering unless you have an exceptionally large data set and a very large computation budget to train everything from scratch by yourself.

- Data augmentation

  Data augmentation is one of the techniques that is often used to improve the performance of computer vision systems. For the majority of computer vision problems, we feel like we just can't get enough data. And this is not true for all applications of machine learning, but it does feel like it's true for computer vision.

  Common augmentation methods:

  - Mirroring
  - Random cropping (perfect)
  - Color shifting: RGB channel PCA color augmentation, AlexNet
  - Rotation, shearing, local warping (less used)

4.2.2.2 Transfer Learning

4.2.2.3 Data Augmentation

4.2.2.4 State of Computer Vision

4.3 Module 3 - Object Detection

4.3.1 Detection Algorithms

4.3.1.1 Object Localization

4.3.1.2 Landmark Detection

4.3.1.3 Object Detection

4.3.1.4 Convolutional Implementation of Sliding Windows

4.3.1.5 Bounding Box Predictions

4.3.1.6 Intersection Over Union

4.3.1.7 Non-max Suppression

4.3.1.8 Anchor Boxes

4.3.1.9 YOLO Algorithm

4.3.1.10 Semantic Segmentation with U-Net

4.3.1.11 Transpose Convolutions

4.3.1.12 U-Net Architecture Intuition

4.3.1.13 U-Net Architecture

4.4 Module 4 - Special Applications: Face Recognition & Neural Style Transfer

4.4.1 Face Recognition

4.4.1.1 What is Face Recognition?

4.4.1.2 One Shot Learning

4.4.1.3 Siamese Network

4.4.1.4 Triplet Loss

4.4.1.5 Face Verification and Binary Classification

4.4.2 Neural Style Transfer

4.4.2.1 What is Neural Style Transfer?

4.4.2.2 What are deep ConvNets learning?

4.4.2.3 Cost Function

4.4.2.4 Content Cost Function

4.4.2.5 Style Cost Function

4.4.2.6 1D and 3D Generalizations

Chapter 5

Course 5: Sequence Models

5.1 Module 1 - Recurrent Neural Networks (RNNs)

5.1.1 Why Sequence Models?

Sequence data is often addressed using sequence models, where the input, the output, or both take the form of sequences. When both the input and output are sequences, their lengths do not necessarily have to be the same.

Examples of Sequence Data:

- Text input
  - Machine translation (text → text)
  - Sentiment classification (comment text → review rating)
  - Named entity recognition (text → person names, organizations, countries, locations, and dates)
- Text input
  - DNA sequence analysis (DNA sequence → info)
- Audio input
  - Speech recognition (audio → text)
- Video input
  - Activity recognition (video → activity type), similar to object detection (image → object type)
- No explicit input
  - Music generation (metadata → music)

5.1.2 Notation

- Input: $x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \dots, x^{\langle t \rangle}, \dots, x^{\langle T_x \rangle}$, or $x^{(i)\langle 1 \rangle}, x^{(i)\langle 2 \rangle}, \dots, x^{(i)\langle t \rangle}, \dots, x^{(i)\langle T_x^{(i)} \rangle}$
- Output: $y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \dots, y^{\langle t \rangle}, \dots, y^{\langle T_y \rangle}$, or $y^{(i)\langle 1 \rangle}, y^{(i)\langle 2 \rangle}, \dots, y^{(i)\langle t \rangle}, \dots, y^{(i)\langle T_y^{(i)} \rangle}$

Each $x^{\langle t \rangle}$ is a one-hot vector (i.e., a sparse vector exactly one entry equals 1 and all others equal 0),

with the length equaling the vocabulary (or dictionary) size.

$$x^{\langle t \rangle} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

This is known as a one-hot representation.

One-hot encoding represents each token (word/character) in a sequence in NLP, where $x^{\langle t \rangle}$ denotes the token at time step t. It has two main limitations:

- High dimensionality: The vector length equals the vocabulary size, which can be very large and inefficient in memory and computation.
- No semantic information (no feature sharing across tokens): One-hot vectors treat all tokens as independent and unrelated, so similar words like “cat” and “dog” are no closer than “cat” and “car”.

One challenge in sequence modeling is variable sequence lengths. Input and output sequences may have different lengths across examples (i.e., both $T_x^{(i)}$ and $T_y^{(i)}$ can vary for different i), making it challenging for standard fixed-size models.

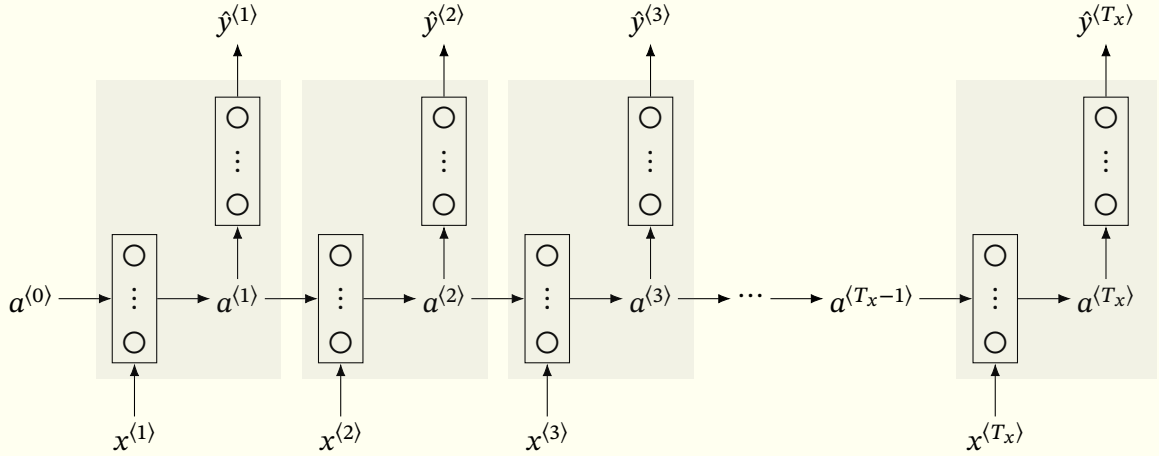
One advantage of RNNs is feature sharing across positions (model design advantage). RNNs share weights across all time steps, allowing patterns learned at one position to generalize to other positions. Without weight sharing, a model might recognize the word “Alice” as a name only at a specific position. This is similar to CNNs, where features learned in one region of an image can generalize to other regions.

5.1.3 RNN Model

At time step t, an RNN forms its prediction by conditioning on the entire history $x^{\langle 1 \rangle}, \ldots, x^{\langle t \rangle}$, rather than on $x^{\langle t \rangle}$ alone.

Consider the following named entity recognition example (see Sections 5.1.3.1 and 5.1.3.2): “Harry Potter and Hermione Granger invented a new spell.” In this case, the input and output sequences have the same length ($T_x = T_y$), and $y^{\langle t \rangle} \in \{0, 1\}$ indicates whether the t-th word is a named entity.

5.1.3.1 Forward Propagation



$$a^{\langle 0\rangle} = \mathbf{0}$$

$$a^{\langle t\rangle} = g\big(W_{aa}a^{\langle t-1\rangle} + W_{ax}x^{\langle t\rangle} + b_a\big) = g\Big(\begin{bmatrix} W_{aa} & W_{ax}\end{bmatrix}\begin{bmatrix} a^{\langle t-1\rangle} \\ x^{\langle t\rangle}\end{bmatrix} + b_a\Big) = g\big(W_a\big[a^{\langle t-1\rangle}, x^{\langle t\rangle}\big] + b_a\big)$$

$$\hat{y}^{\langle t\rangle} = g_o\big(W_{ya}a^{\langle t\rangle} + b_y\big) = g_o\big(W_y a^{\langle t\rangle} + b_y\big)$$

where g is typically tanh (or ReLU), and g_o is chosen based on the task (e.g., sigmoid or softmax).

5.1.3.2 Backpropagation Through Time

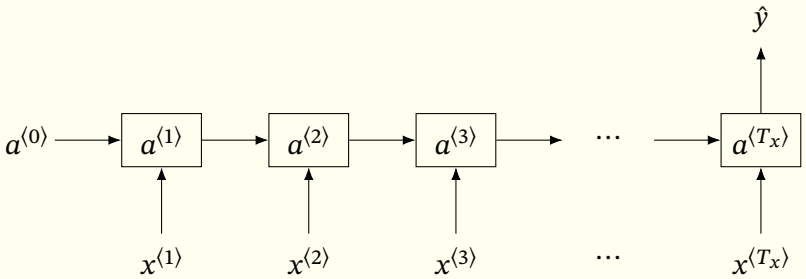
The loss (cross-entropy) is defined as,

$$\mathcal{L}\big(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}\big) = -y^{\langle t\rangle}\log\hat{y}^{\langle t\rangle} - \big(1 - y^{\langle t\rangle}\big)\log\big(1 - \hat{y}^{\langle t\rangle}\big)$$

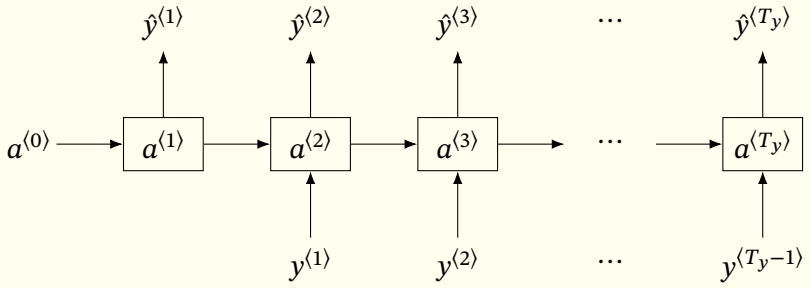
$$\mathcal{L}\big(\hat{y}, y\big) = \sum_{t=1}^{T_y}\mathcal{L}\big(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}\big)$$

5.1.3.3 Different Types of RNNs

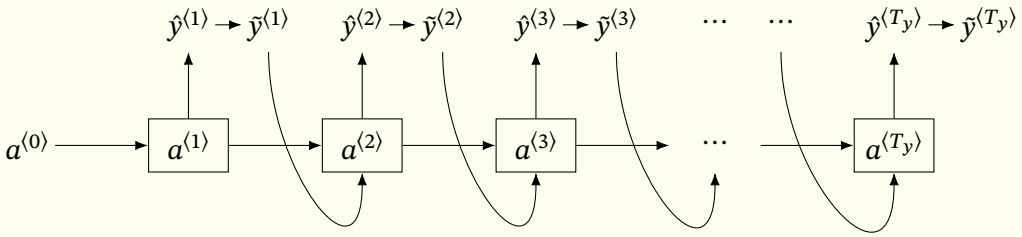
- Many-to-one (sentiment classification, action prediction: video → action class):



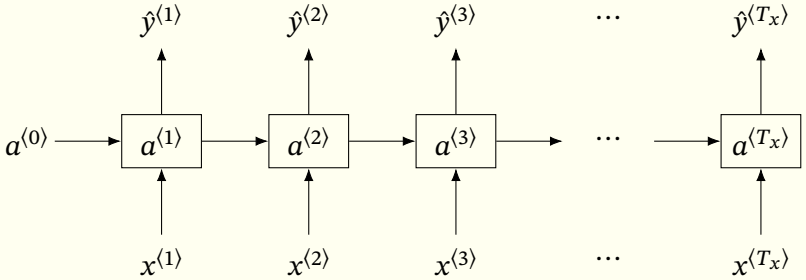
- One-to-many (music generation: metadata (e.g., genre or style) / $\varnothing \rightarrow$ music, image captioning: image $\rightarrow$ caption):
  - During training:



  - During inference/generation (when the ground-truth outputs $y^{\langle t\rangle}$ are not availablee):

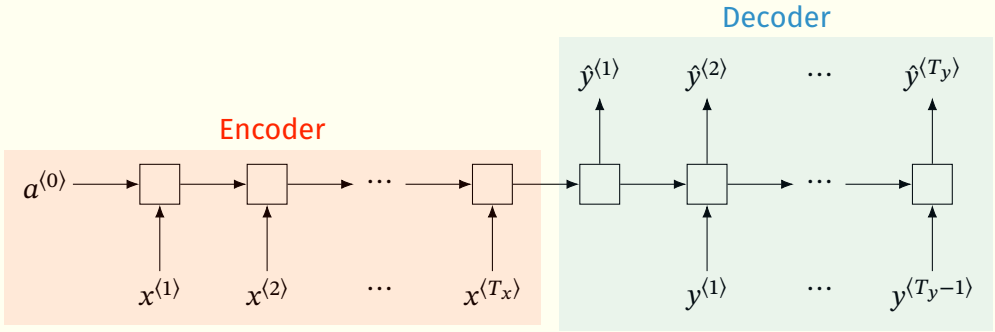


  Music generation can also be regarded as conditional many-to-many with $T_x = T_y$.
- Many-to-many with $T_x = T_y$ (named entity recognition, speech recognition, video classification on frame level):

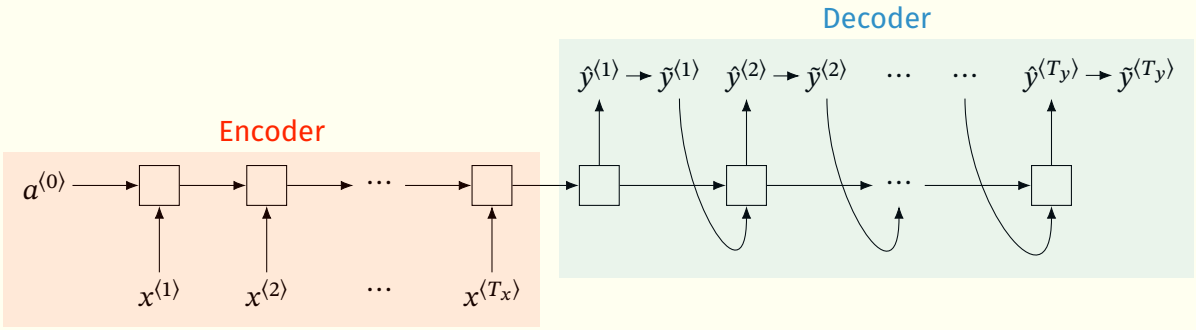


Language models are many-to-many with $T_x = T_y$.

- Many-to-many with $T_x \neq T_y$ (machine translation, video captioning: video → caption):
  - During training:



  - During inference:



- Attention based architectures: Attention lets a model focus on the most important parts of the input when making a prediction. This is especially useful for tasks (e.g., translation, summarization, speech recognition) where standard RNNs or LSTMs struggle with long input sequences. Instead of trying to store everything in one final hidden state, the model looks at all input positions and picks what matters most using a weighted sum, where higher weights indicate more important inputs for the current output.

Standard vanilla RNN (without attention) update rule:

- Many-to-one (or many-to-many with $T_x = T_y$, or encoder part of many-to-many with $T_x \neq T_y$) sequences,

$$a^{\langle t \rangle} = g\big(W_a\big[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\big] + b_a\big)$$

  Hidden state $a^{\langle t \rangle}$ depends on:
  - Previous hidden state $a^{\langle t-1 \rangle}$, which carries information from earlier time steps.
  - Current input $x^{\langle t \rangle}$, which provides the new information at time step t.

- One-to-many (or decoder part of many-to-many with $T_x \neq T_y$) sequences,
  - During training (teacher forcing):

$$a^{\langle t \rangle} = g\big(W_a\big[a^{\langle t-1 \rangle}, y^{\langle t-1 \rangle}\big] + b_a\big)$$

  - During inference:

$$a^{\langle t \rangle} = g\big(W_a\big[a^{\langle t-1 \rangle}, \tilde{y}^{\langle t-1 \rangle}\big] + b_a\big)$$

  Hidden state $a^{\langle t \rangle}$ depends on:
  - Previous hidden state $a^{\langle t-1 \rangle}$, which carries information from earlier time steps.
  - Previous output $y^{\langle t-1 \rangle}$ during training (or $\tilde{y}^{\langle t-1 \rangle}$ during inference), which provides the model's most recent prediction and used as input for the time step t.

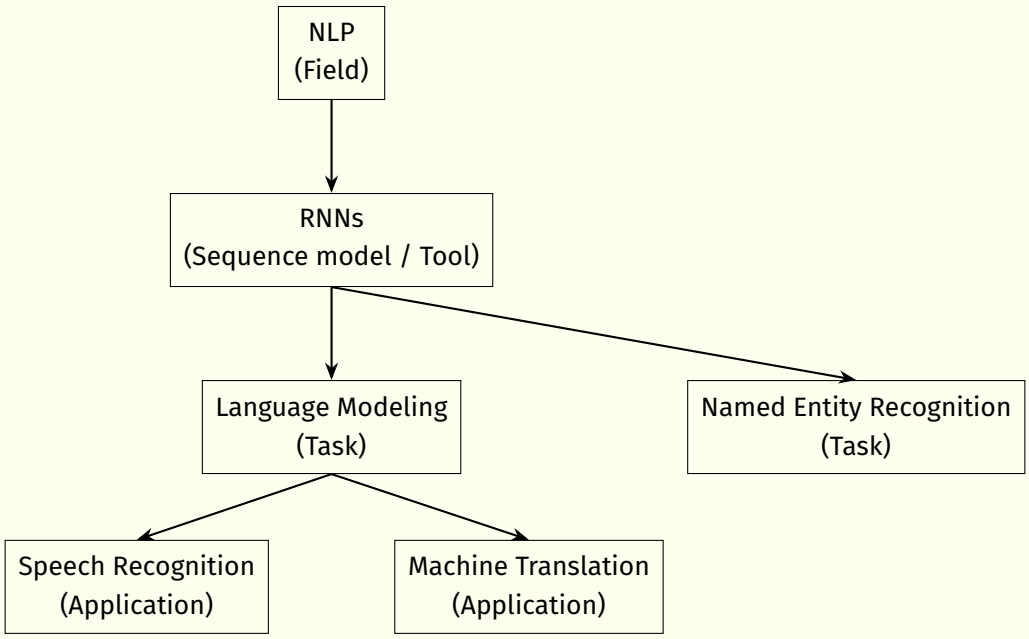
5.1.4 Language Model

Language modeling is a core natural language processing (NLP) task, important for applications like speech recognition and machine translation. It also provides representations useful for downstream tasks such as named entity recognition.

Language modeling is the task of predicting the next token given the previous tokens and, equivalently, estimate the joint probability of an entire sequence $y^{\langle 1 \rangle}, \dots, y^{\langle T_y \rangle}$, i.e.,

$$p\big(y^{\langle 1 \rangle}, \dots, y^{\langle T_y \rangle}\big) = \prod_{t=1}^{T_y} p\big(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}\big)$$

RNNs provide a natural framework for implementing language modeling.



- Field:
  - NLP: Process and understand natural language.
- Task (specific problem or objective):
  - Language modeling (core NLP task): Predict the next token in a sequence.
  - Named entity recognition (NLP task): Identify entities (e.g., names, organizations) in text.
- Application (real-world system using one or more tasks):
  - Speech recognition: Convert audio to text, rely on language modeling to predict likely words.
  - Machine translation: Translate text between languages, rely on language modeling for fluent output.
- Sequence model / Tool:
  - RNNs: Used in NLP for modeling sequences of tokens.

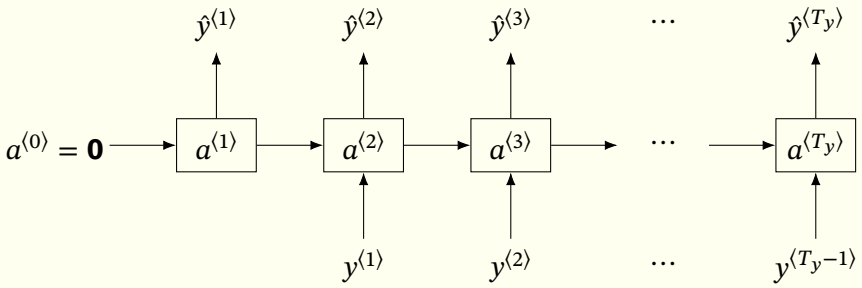
5.1.4.1 Language Modeling with RNNs

An RNN language model (language modeling using RNNs) for speech recognition:

- Training set: A large text corpus (i.e., many sentences).
- Tokenization: Split each sentence into tokens (e.g., words) and encode them as one-hot vectors. Include an end-of-sentence token `EOS` at the end; punctuation may also be treated as tokens. Out-of-vocabulary words are mapped to `UNK`.

$$y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \dots, y^{\langle T_y \rangle}$$

- Model structure:



- Loss function (cross-entropy):

$$\mathcal{L}\big(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}\big) = -\sum_i y_i^{\langle t \rangle} \log \hat{y}_i^{\langle t \rangle},$$
$$\mathcal{L}\big(\hat{y}, y\big) = \sum_t \mathcal{L}\big(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}\big).$$

An RNN is trained to predict the next token at each time step t by using the previous true tokens $y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}$. At each step, it outputs a (discrete) probability distribution over the entire vocabulary for the t-th token (via a softmax), conditioned on previous true tokens $y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}$, i.e.,

$$\hat{y}^{\langle t \rangle} = p\big(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}\big)$$

For any word w in the vocabulary, the corresponding entry of $\hat{y}^{\langle t \rangle}$ represents the probability

$$\mathsf{Pr}\big(y^{\langle t \rangle} = w \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}\big)$$

The model is trained by minimizing the loss between the true token $y^{\langle t \rangle}$ and the predicted distribution $\hat{y}^{\langle t \rangle}$ across all time steps.

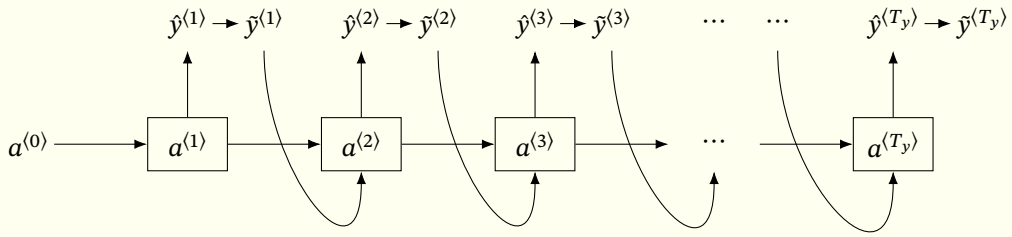
RNN are very effective for NLP and other sequence tasks because they have “memory”. They can read inputs tokens $y^{\langle t \rangle}$ one at a time, and remember some contextual information through the hidden layer activations that get passed from one time step to the next. This allows a unidirectional (one-way) RNN to take information from the past to process later inputs.

An RNN language model is an example of self-supervised learning, since it is trained on unlabeled text data to predict the next word in a sequence.

5.1.4.2 Sampling Novel Sequences

Sampling novel sequences from a trained RNN:

- Given an initial token $\tilde{y}^{\langle 1 \rangle}$, predict the next token distribution $\hat{y}^{\langle 2 \rangle} = p\big(y^{\langle 2 \rangle} \mid \tilde{y}^{\langle 1 \rangle}\big)$, then sample the next token $\tilde{y}^{\langle 2 \rangle} \sim \hat{y}^{\langle 2 \rangle}$
- Predict the next token distribution $\hat{y}^{\langle 3 \rangle} = p\big(y^{\langle 2 \rangle} \mid \tilde{y}^{\langle 1 \rangle}, \tilde{y}^{\langle 2 \rangle}\big)$, then sample the next token $\tilde{y}^{\langle 3 \rangle} \sim \hat{y}^{\langle 3 \rangle}$
- Repeat this process iteratively until:
  - an end-of-sequence token `EOS` is generated (indicating the sentence has ended), if `EOS` is included in the vocabulary
  - or a predefined maximum sequence length is reached, if `EOS` is not used included in the vocabulary



5.1.4.3 Types of Language Models

- Word-level language models: still widely used in practice.
- Character-level language models:
  - Advantage: avoid the need for an unknown-word token such as `UNK`.
  - Disadvantage: operate on much longer sequences, which can be more computationally expensive to train and can make it harder to capture long-range dependencies (how the earlier parts of the sentence affect the later part).

5.1.5 Vanishing Gradients with RNNs

Training very deep neural networks can lead to vanishing or exploding gradient problems, where gradients shrink or grow exponentially as the number of layers increases.

In RNNs,

- Exploding gradients can be solved with gradient clipping: if the gradient (vector) norm exceeds a threshold, it is rescaled to stay within a fixed bound.

- Vanishing gradients are harder to address, and often prevent RNNs from learning long-range dependencies.dependencies.

A example of a long-range dependency in language is subject-verb agreement,

- The **cat**, which already ate a bunch of food that ..., **was** full.
- The **cats**, which already ate a bunch of food that ..., **were** full.

Here, the correct verb depends on a subject that appears much earlier in the sentence.

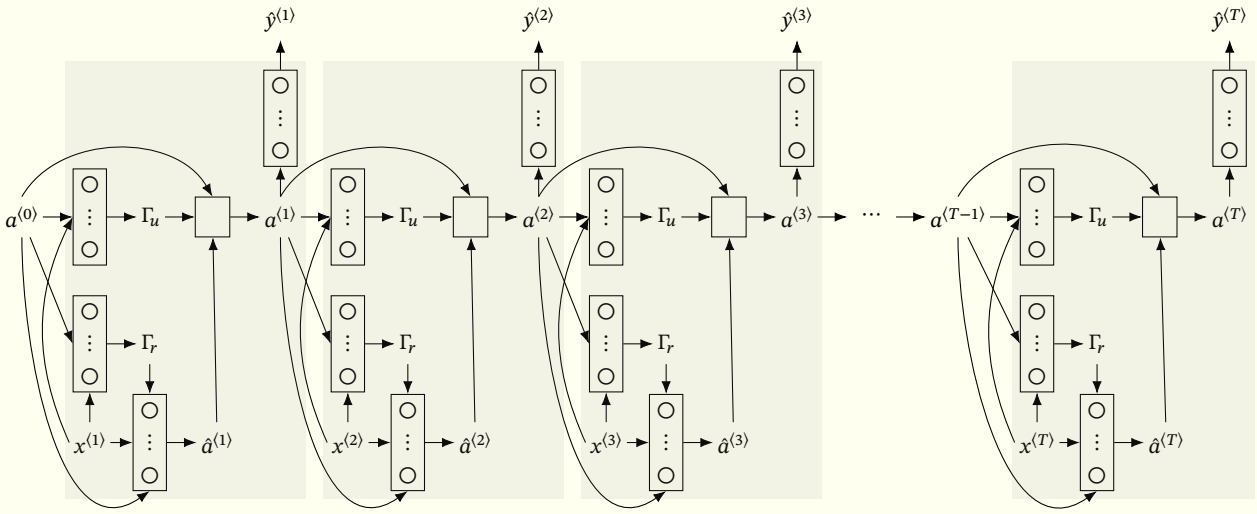
RNNs struggle with this because of the vanishing gradient problem. During backpropagation through many time steps, gradients shrink toward zero. As a result, later errors have little effect on earlier hidden states. This makes it difficult for the RNN to learn what information to keep over long sequences.

5.1.5.1 Gated Recurrent Unit (GRU)

GRU (Gated Recurrent Unit) is a modification of the standard RNN hidden layer that helps capture long-range dependencies and reduce vanishing gradient problem by using gating mechanisms to control information flow across time steps.

A GRU maintains a hidden state $a^{\langle t \rangle}$, which acts as the model's memory at time step t. This hidden state carries information across time steps and is selectively updated using gates.

- RNN architecture with GRU:



- Equations:

$$\Gamma_r = \sigma\big(W_r\left[a^{\langle t-1\rangle}, x^{\langle t\rangle}\right] + b_r\big), \quad \hat{a}^{\langle t\rangle} = \tanh\big(W_a\left[\Gamma_r \odot a^{\langle t-1\rangle}, x^{\langle t\rangle}\right] + b_a\big)$$
$$\Gamma_u = \sigma\big(W_u\left[a^{\langle t-1\rangle}, x^{\langle t\rangle}\right] + b_u\big), \quad a^{\langle t\rangle} = \Gamma_u \odot \hat{a}^{\langle t\rangle} + \big(1 - \Gamma_u\big) \odot a^{\langle t-1\rangle}$$
$$\hat{y}^{\langle t\rangle} = g_o\big(W_y\, a^{\langle t\rangle} + b_y\big)$$

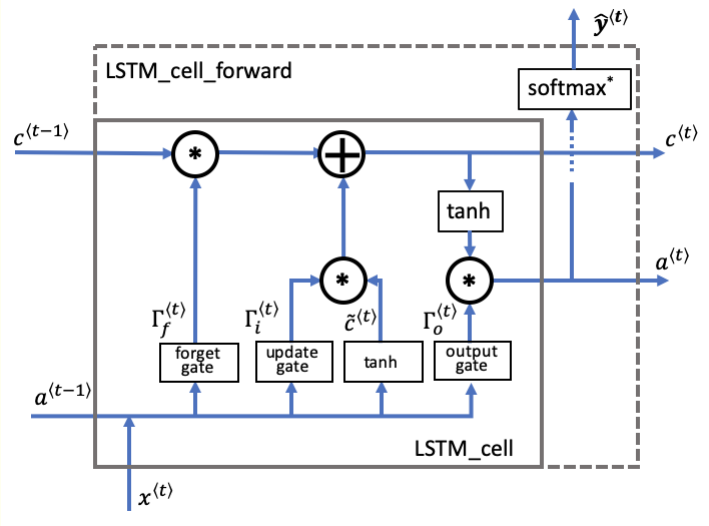
  Here, σ is the sigmoid function. $\odot$ denotes element-wise multiplication.

- Components:
  - $a^{\langle t\rangle}$: hidden state at time step t, which represents the memory of the GRU at time t. It is passed forward to the next time step and carries information across time.
  - $\hat{a}^{\langle t\rangle}$: candidate state at time step t, represents new candidate information that could be added to the hidden state $a^{\langle t\rangle}$.
- Gates:
  - Γ_r: relevance/reset gate, controls how much of the previous hidden state $a^{\langle t-1\rangle}$ is used when computing the candidate state $\hat{a}^{\langle t\rangle}$
    - If $\Gamma_r \to 0$, the model ignores most past information.
    - If $\Gamma_r \to 1$, the model uses most of the past.
  - Γ_u: update gate, controls how much of the new candidate state $\hat{a}^{\langle t\rangle}$ is incorporated versus how much of the old hidden state $a^{\langle t-1\rangle}$ is preserved, during updating into the new hidden state $a^{\langle t\rangle}$
    - If $\Gamma_u \to 0$, mostly past information is retained.
    - If $\Gamma_u \to 1$, mostly new information is used.
- Properties:
  - All states and gates can be vectors (same dimensionality). Element-wise multiplication $\odot$ allows each dimension to be gated independently, i.e., when Γ_u is a vector, each of its entries acts as a gate for the corresponding dimension of the hidden state.

5.1.5.2 Long Short Term Memory (LSTM)

LSTM (Long Short-Term Memory) is a type of recurrent architecture designed to learn long-range dependencies in sequence data. It is even more powerful than the GRU because it uses a more complex gating mechanism that allows finer control over memory.

- Architecture of LSTM:



- Equations:

$$\hat{c}^{\langle t \rangle} = \tanh\left(W_c\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_c\right)$$
$$\Gamma_u = \sigma\left(W_u\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_u\right), \quad \Gamma_f = \sigma\left(W_f\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_f\right), \quad c^{\langle t \rangle} = \Gamma_u \odot \hat{c}^{\langle t \rangle} + \left(1 - \Gamma_f\right) \odot c^{\langle t-1 \rangle}$$
$$\Gamma_o = \sigma\left(W_o\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_o\right), \quad a^{\langle t \rangle} = \Gamma_o \odot \tanh\left(c^{\langle t \rangle}\right)$$
$$\hat{y}^{\langle t \rangle} = g_o\left(W_y a^{\langle t \rangle} + b_y\right)$$

- Components:
  - $a^{\langle t \rangle}$: activation (hidden state) at time step t, output of the LSTM cell at time step t, input of the LSTM cell at next time step $t+1$ (short-term memory).
  - $c^{\langle t \rangle}$: cell state (memory) at time step t, carries information across time steps (long-term memory).
  - $\hat{c}^{\langle t \rangle}$: candidate cell state at time step t, represents new candidate information that could be added to the cell state $c^{\langle t \rangle}$.
- Gates:
  - Γ_u: update gate, controls how much of the candidate state $\hat{c}^{\langle t \rangle}$ is incorporated into the next cell state $c^{\langle t \rangle}$.
  - Γ_f: forget gate, controls how much of the previous cell state $c^{\langle t-1 \rangle}$ is remained to get the next cell state $c^{\langle t \rangle}$.
  - Γ_o: output gate, controls how much of the cell state contributes to the activation $a^{\langle t \rangle}$.
- In a standard LSTM, each gate (Γ_u, Γ_f, Γ_o) is computed from $a^{\langle t-1 \rangle}$ and $x^{\langle t \rangle}$. A peephole connection is a variation of the LSTM in which each gate is computed not only from $a^{\langle t-1 \rangle}$ and $x^{\langle t \rangle}$, but also from the previous cell state $c^{\langle t-1 \rangle}$.

Comparison of LSTM and GRU:

- LSTM was introduced earlier; GRU is a newer, simplified variant.
- GRUs have two gates (reset, update) and a single activation state. LSTMs have three gates (update, forget, output) and separate cell and activation states. LSTMs are typically more flexible due to an additional gate. GRUs are faster and easier to scale for larger models.
- Both architectures address the vanishing gradient problem and enable learning of long-range dependencies in sequences.
- LSTMs have been the more common default choice. In recent years, GRUs have gained popularity as well, since their simpler structure often performs comparable performance while being easier to scale to larger models.

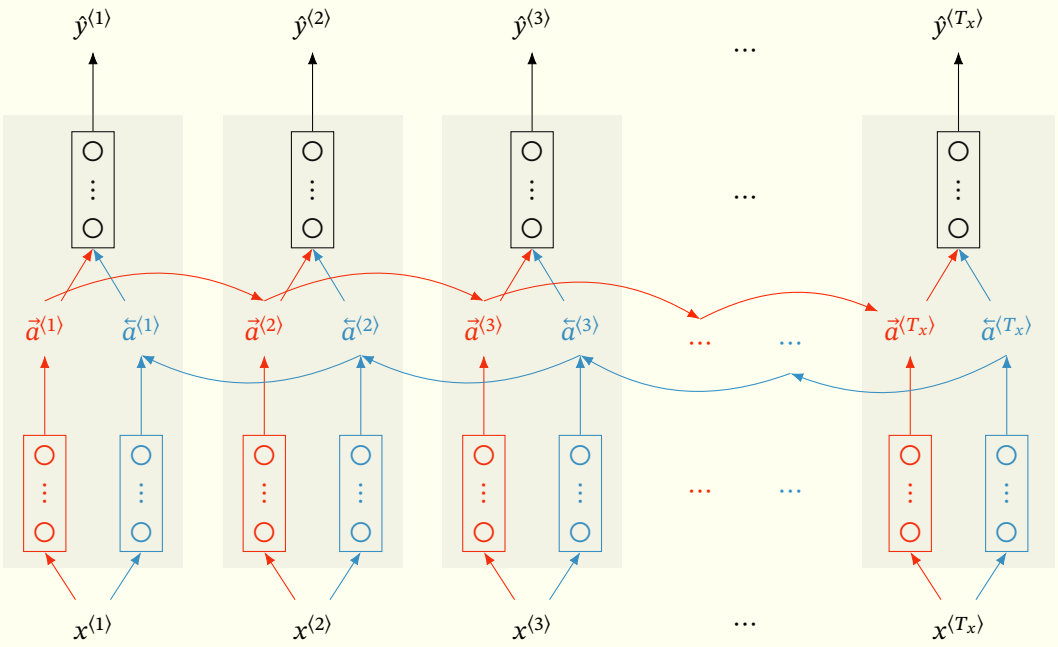
5.1.6 Bidirectional RNN (BRNN)

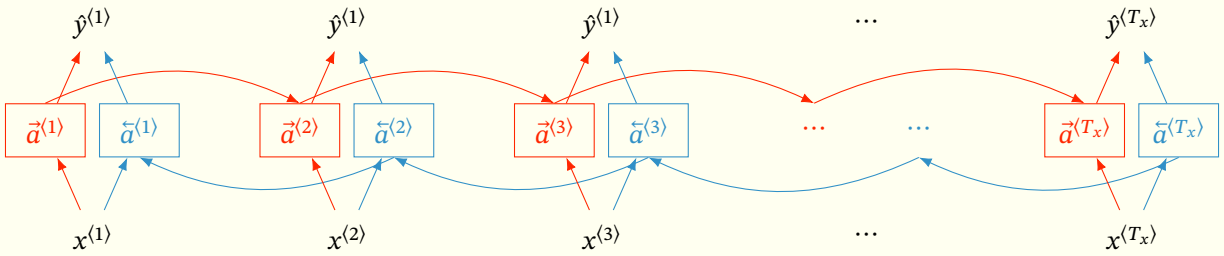
A limitation of unidirectional RNNs is that it cannot use future context. Bidirectional RNNs (BRNNs) address this by processing the sequence in both the forward and backward directions.

An example of named entity recognition:

- He said, "**Teddy** bears are on sale!"
- He said, "**Teddy** Roosevelt was a great president!"

To determine whether the word "Teddy" is part of a person's name, the two words before "Teddy" are not enough, and information from later in the sentence is also needed. A bidirectional RNN can use the words that come after "Teddy" to make a better prediction.





Here,

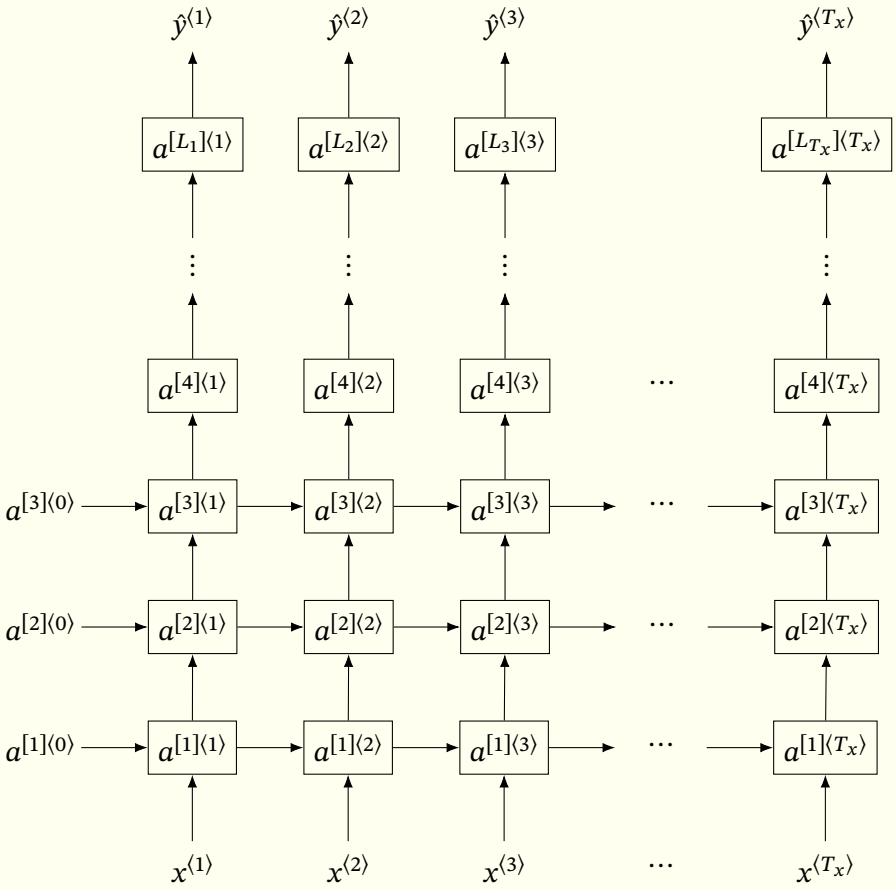
$$\hat{y}^{\langle t\rangle} = g_o\left(W_y\left[\vec{a}^{\langle t\rangle}, \overleftarrow{a}^{\langle t\rangle}\right] + b_y\right)$$

BRNNs can be built using standard RNN, GRU, or LSTM units. In practice, BRNNs with LSTM units are widely used in NLP.

One disadvantage of BRNNs is that they require the entire input sequence before making predictions. For example, in speech recognition, a BRNN can use both past and future audio context, but it may need to wait until the speaker finishes before producing a transcript. For real-time speech recognition, other models are often used instead.

However, when the full sentence is available (as in many NLP tasks), standard BRNNs work very well. For example, in named entity recognition, a BRNN with LSTM units is often a reasonable first model to try because it can use both left and right context to make predictions at any position in the sequence.

5.1.7 Deep RNN



For $t = 1, 2, \cdots$

$$a^{[1]\langle t\rangle} = g\big(W_a^{[1]}\big[a^{[1]\langle t-1\rangle}, x^{\langle t\rangle}\big] + b_a^{[1]}\big)$$
$$a^{[l]\langle t\rangle} = g\big(W_a^{[l]}\big[a^{[l]\langle t-1\rangle}, a^{[l-1]\langle t\rangle}\big] + b_a^{[l]}\big), \quad l = 2, 3, \cdots$$

5.2 Module 2 - Natural Language Processing & Word Embeddings

5.2.1 Introduction to Word Embeddings

5.2.1.1 Word Representation

A word can be represented as a one-hot vector over a vocabulary. However, a key limitation of this representation is that each word is treated as completely independent. It does not capture relationships between words, since the inner product (similarity) between any two different one-hot vectors is zero.

Word embeddings (a featurized representations of words) represent words as dense vectors in a continuous space (e.g., 300 dimensions). They provide a more effective representation than one-hot vectors because they capture semantic relationships between words, placing similar words closer together in the vector space.

t-SNE: an algorithm (introduced by Laurens van der Maaten and Geoffrey Hinton) that reduces high-dimensional data (e.g., 300-dimensional word embeddings) to 2D so it can be visualized. It preserves the local structure of the data, meaning that words close to each other in the high-dimensional space stay close in the 2D visualization.

5.2.1.2 Using Word Embeddings

An example of named entity recognition (NER):

- Sally Johnson is an orange farmer.
- Robert Lin is an apple farmer.
- Robert Lin is an durian cultivator.

An NER model includes:

- Word embeddings as the input representation (feature representation)
- Additional layers that process those embeddings to make predictions (e.g., identifying names)

Word embeddings can be used for transfer learning in tasks (e.g., NER). After learning patterns from large amounts of unlabeled text (e.g., downloaded from the internet), they can be applied to an NER model, even when only a small labeled dataset is available for training.

In general, transfer learning is most effective when there is a large amount of data available for one task (task A) and a relatively small amount of data for another task (task B). In such cases, knowledge learned from task A can be transferred to improve performance on task B. This is common in many NLP applications (e.g., named entity recognition, text summarization, co-reference resolution, and parsing), where labeled data may be limited. However, it is less useful for tasks like language modeling or

machine translation, where large amounts of labeled data are already available.

For example, given a small training set that contains "Sally Johnson is an orange farmer." where "Sally Johnson' is labeled as a person, an NER model can learn to recognize "Robert Lin" as a person in "Robert Lin is an apple farmer." since words "orange" and "apple" are similar. Furthermore, if a word embedding model pre-trained on large unlabeled text has learned that words like "orange" and "durian" are similar, and that "farmer" and "cultivator" are similar, the NER model can generalize and recognize that "Robert Lin" is also a person in "Robert Lin is a durian cultivator."

Transfer learning and word embeddings:

- Learn word embeddings from a large text corpus (1–100B words), or use downloaded pre-trained embeddings.
- Transfer the embeddings to a new task with a smaller labeled dataset (e.g., 100k words).

  This allows using a 300-dimensional dense vector instead of a 10000-dimensional one-hot vector.
- Optionally, continue to fine-tune the word embeddings using the new task-specific data.

Word embeddings have an interesting relationship to face encoding methods from computer vision. In face recognition, a Siamese network is used to learn a fixed-length representation (e.g., a 128-dimensional vector) for each face. These representations can then be compared to determine whether two images belong to the same person. Similarly, in NLP, word embeddings represent words as vectors, allowing comparison between words based on their meanings. The terms "encoding" and "embedding" are often used interchangeably, as both refer to representing data as vectors in a continuous space. The key difference is not in the terminology, but in how they are used:

- In face recognition, the system must generalize to an unlimited number of new faces.
- In NLP, models typically work with a fixed vocabulary, and unseen words are often treated as unknown tokens.

5.2.1.3 Properties Word Embeddings

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

Question: What word w satisfies the analogy: Man is to Woman as King is to w?

- Find w such that

$$\arg\max_{w} \text{sim}(e_w, e_{\text{king}} - e_{\text{man}} + e_{\text{woman}})$$

Here, $\text{sim}(\cdot, \cdot)$ denotes the cosine similarity, defined as

$$\text{sim}(u, v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$$

- Notice that

$$e_{\text{man}} - e_{\text{women}} = \begin{bmatrix} -1 \\ 0.01 \\ 0.03 \\ 0.09 \end{bmatrix} - \begin{bmatrix} 1 \\ 0.02 \\ 0.02 \\ 0.01 \end{bmatrix} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$e_{\text{king}} - e_{\text{queen}} = \begin{bmatrix} -0.95 \\ 0.93 \\ 0.70 \\ 0.02 \end{bmatrix} - \begin{bmatrix} 0.97 \\ 0.95 \\ 0.69 \\ 0.01 \end{bmatrix} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

i.e.,

$$e_{\text{man}} - e_{\text{women}} \approx e_{\text{king}} - e_{\text{queen}}$$

5.2.1.4 Embedding Matrix

Implementing an algorithm to learn word embeddings ends up with an embedding matrix E.

Given embedding matrix E, and one-hot vector of word j (i.e., o_j), the embedding for word j (i.e., e_j) is given by

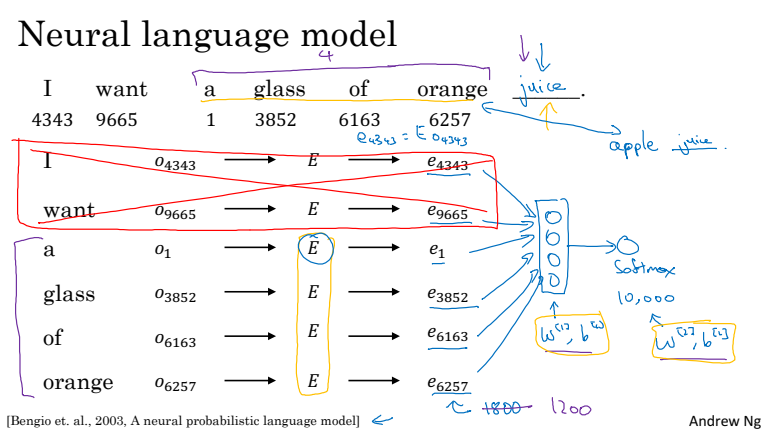
$$e_j = E\, o_j$$

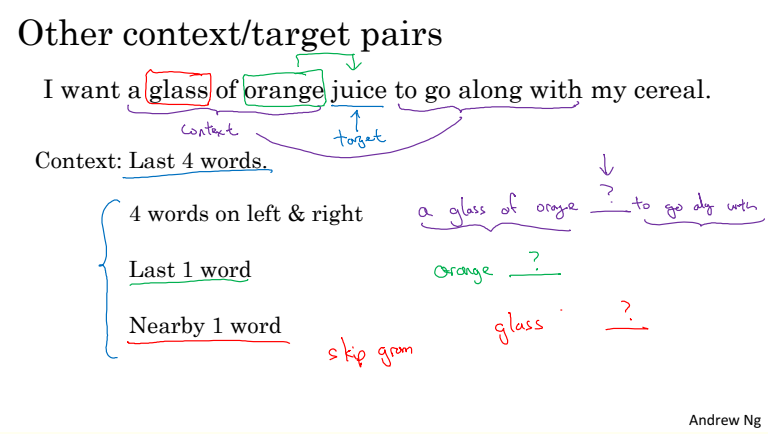
which is the j-th column of E.

5.2.2 Learning Word Embeddings: Word2vec & GloVe

Needs to be rewatch for details.

5.2.2.1 Learning Word Embeddings





5.2.2.2 Word2vec

5.2.2.3 Negative Sampling

5.2.2.4 GloVe Word Vectors

5.2.3 Applications Using Word Embeddings

5.2.3.1 Sentiment Classification

5.2.3.2 Debiasing Word Embeddings

Bias in word embeddings: Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. E.g.,

- Man is to Woman as King is to Queen
- Bias: Man is to Computer Programmer as Woman is to **Homemaker**

- Bias: Father is to Doctor as Mother is to **Nurse**

Addressing bias in word embeddings (Bolukbasi et. al., 2016. Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings):

- Identify bias direction (direction in the embedding space that captures the bias):

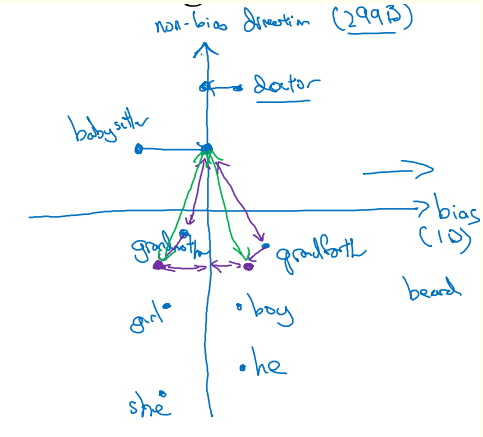
  Use word pairs (e.g., $(\text{he}, \text{she}), (\text{man}, \text{woman})$, etc.), compute their differences, and average them to get the bias direction **b**.

- Neutralize (remove unwanted bias from neutral words):

  For neutral words (e.g., "doctor"), remove their projection on **b**.

- Equalize pairs:

  For word pairs that should differ only by the bias (e.g., "he" vs "she"), make them symmetric along **b** so they are equally distant from the neutral subspace.

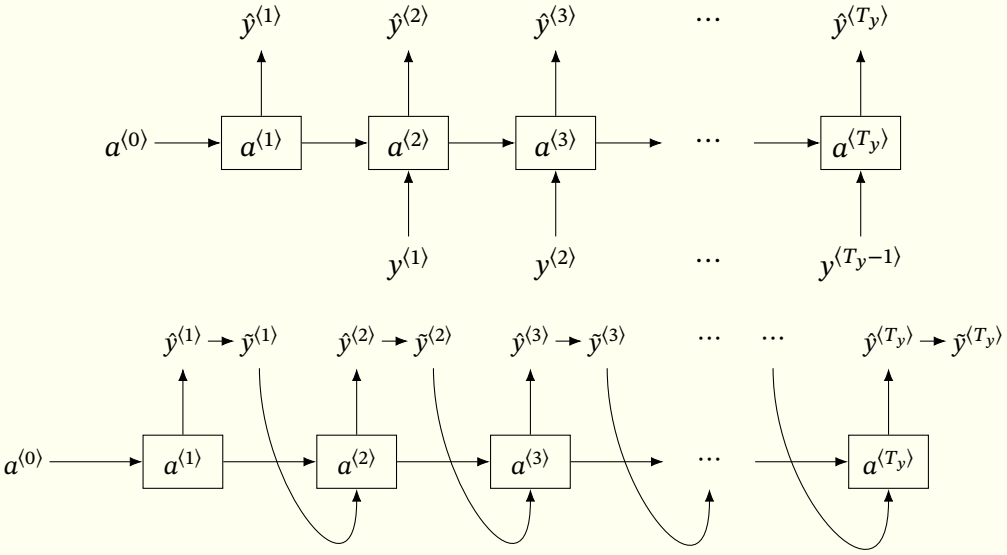


5.3 Module 3 - Sequence Models & Attention Mechanism

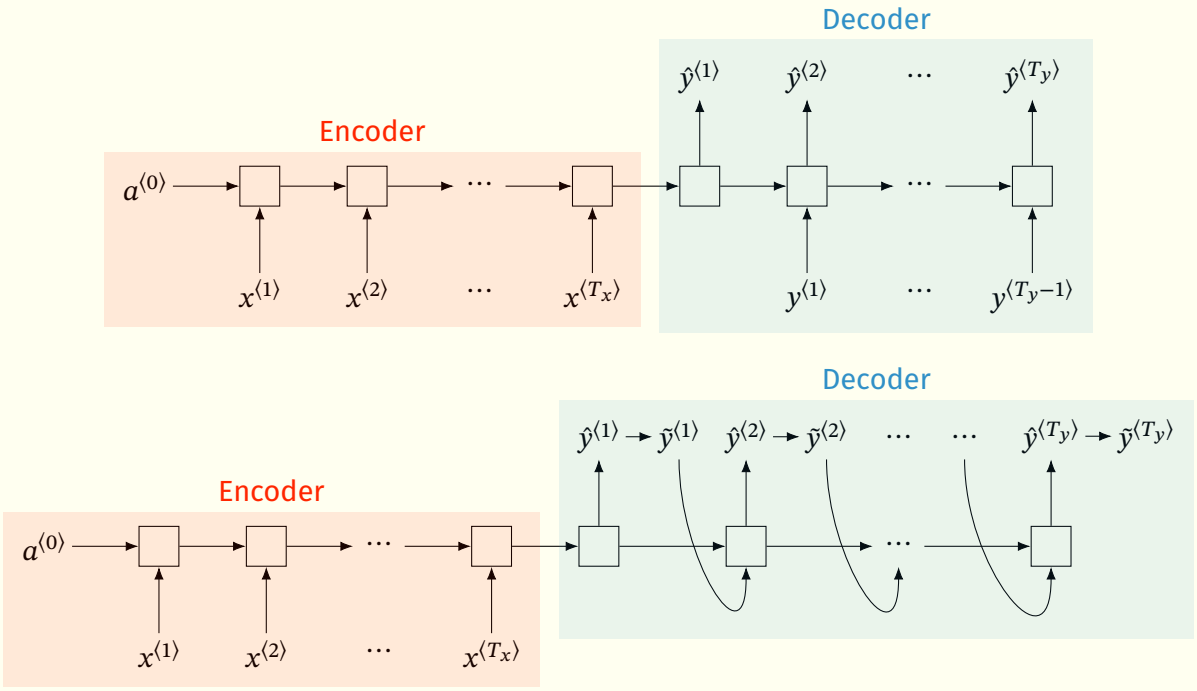
5.3.1 Various Sequence To Sequence Architectures

5.3.1.1 Basic Models

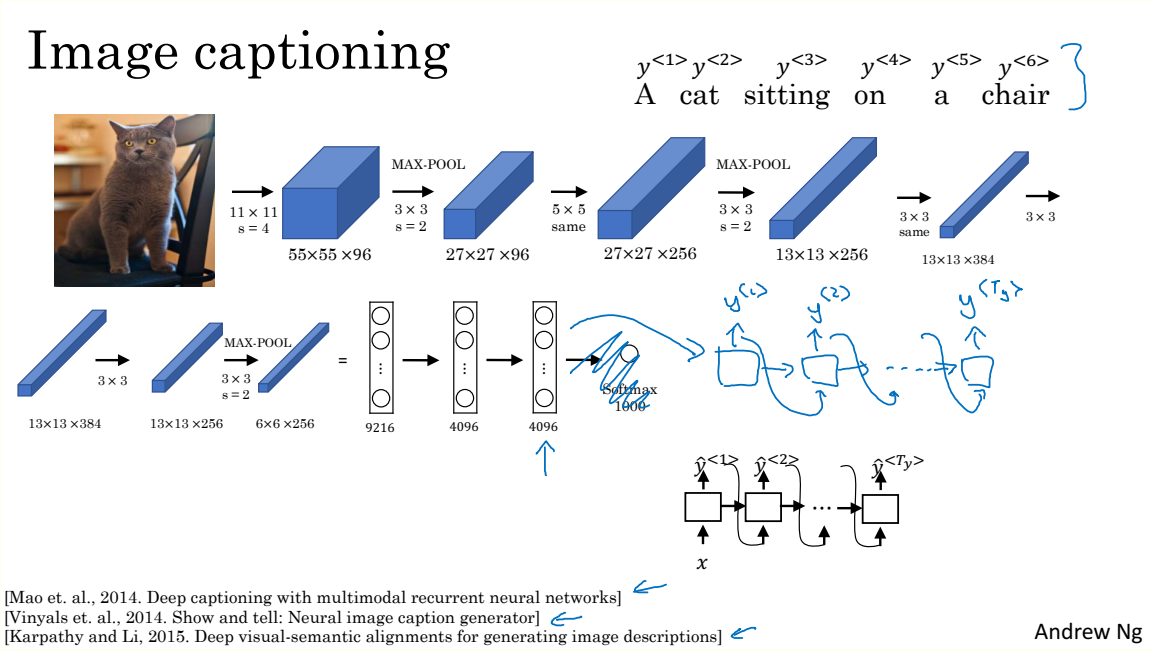
- Language model:



- Encoder–decoder:



- Comparison:
  - A language model is used for generating a sequence without a separate input sequence.
  - An encoder–decoder architecture is used for tasks where a sequence input (e.g., text, audio, or image) is mapped to an output sequence. E.g.,
    * Machine translation: word embeddings + RNN
    * Image captioning: CNN (feature extractor) + RNN



Here, a pre-trained AlexNet (with the final softmax layer removed) is used, which takes an input image ($224 \times 224 \times 3$), and outputs a 4096-dimensional feature vector representing the image. This vector is then fed into an RNN to generate a caption.

  - The decoder in a machine translation model works is similar to language model, but with one key difference in how it begins:
    * Language model: typically starts with no prior information (blank or zero input).
    * Machine translation model: uses an encoder to first process the input sentence to obtain a meaningful representation, which is then used by an decoder to generate the output translation.
  - A language model defines a (discrete) joint probability distribution over sequences, factor-

ized into conditional probabilities using the chain rule,

$$\begin{aligned} p(y) &= p(y^{\langle 1 \rangle}, \dots, y^{\langle T_y \rangle}) \\ &= \prod_{t=1}^{T_y} p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}) \\ &= \prod_{t=1}^{T_y} \hat{y}^{\langle t \rangle} \end{aligned}$$

It is often used for open-ended text generation, where each word $\tilde{y}^{\langle t \rangle}$ is generated by randomly sampling from the predicted distribution $\hat{y}^{\langle t \rangle}$,

$$\tilde{y}^{\langle t \rangle} \sim \hat{y}^{\langle t \rangle}$$

- Machine translation can be regarded as a conditional language model,

$$\begin{aligned} p(y \mid x) &= p(y^{\langle 1 \rangle}, \dots, y^{\langle T_y \rangle} \mid x^{\langle 1 \rangle}, \dots, x^{\langle T_x \rangle}) \\ &= \prod_{t=1}^{T_y} p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x^{\langle 1 \rangle}, \dots, x^{\langle T_x \rangle}) \\ &= \prod_{t=1}^{T_y} p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x) \end{aligned}$$

The goal is to find the most probable full sentence,

$$\tilde{y} = \arg\max_{y} p(y \mid x) = \arg\max_{y} \prod_{t=1}^{T_y} p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x)$$

Unlike language models, tasks like machine translation, image captioning and speech recognition aim to generate the most probable sequence,

  - In machine translation, the best possible translation is chosen rather than a random one.
  - In image captioning, the most accurate and likely caption is selected.

- Generation strategies of machine translation:
  - Random sampling:

$$\tilde{y}^{\langle t \rangle} \sim p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x)$$

    This can lead to low-quality or incorrect translations.

* Greedy search:

$$\tilde{y}^{\langle t \rangle} = \arg\max_{y^{\langle t \rangle}} p\big(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x\big)$$

This does not guarantee the best overall sentence, since it chooses the best word at each step.

* Exhaustive search: try all possible sentences

The number of possible sentences is extremely large, so exhaustive search is infeasible.

Instead, search algorithms like beam search are used to find a good approximation.

5.3.1.2 Beam Search

- Basic:
  - Beam search is an approximate (heuristic) search algorithm used in sequence generation (e.g., machine translation) to find a high-probability output without evaluating all possible sequences. It generalizes greedy search by keeping multiple candidate sequences at each time step instead of just one. Greedy search is beam search with beam width $B = 1$.
  - Computational complexity:
    * Exhaustive search: $O\big(V^{T_y}\big)$, with V being the vocabulary size.
    * Beam search: $O\big(B\, T_y\, V\big)$
  - In practice, try different beam widths (e.g., 3, 10, 100, 1000, 3000) to find what works best for a given task. However, very large beam widths often yield diminishing returns, as improvements become smaller as B increases.
    * large B: better result, slower
    * small B: worse result, faster
- Refinements:
  - Multiplying many small probabilities can produce extremely tiny values, leading to numerical underflow. Use log probabilities to avoid this.

$$\tilde{y} = \arg\max_{y} \sum_{t=1}^{T_y} \log p\big(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x\big)$$

  - Longer sentences tend to have lower probabilities because they involve more terms, which biases the model toward shorter outputs. Use Length normalization (normalization by sen-

tence length) to fix this, which significantly reduces the penalty for longer translations.

$$\tilde{y} = \arg\max_{y} \frac{1}{T_y} \sum_{t=1}^{T_y} \log p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x)$$

- Instead of full normalization, a softer normalization is often used:

$$\tilde{y} = \arg\max_{y} \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log p(y^{\langle t \rangle} \mid y^{\langle 1 \rangle}, \dots, y^{\langle t-1 \rangle}, x)$$

  Here, α is hyperparameter.

  * If $\alpha = 1$: full normalization
  * If $\alpha = 0$: no normalization
  * If $0 < \alpha < 1$: partial normalization

- Error analysis:
  - Example:
    * x: Jane visite l'Afrique en septembre.
    * $\tilde{y}$ (algorithm): Jane visited Africa last September.
    * y^* (human): Jane visits Africa in September.
  - Rule: Use RNN (encoder + decoder) computes $p(\tilde{y} \mid x)$ and $p(y^* \mid x)$,
    * $p(y^* \mid x) > p(\tilde{y} \mid x)$: y^* attains higher $p(y \mid x)$ than $\tilde{y}$, with $\tilde{y}$ chosen by beam search, thus beam search is at fault.
    * $p(y^* \mid x) \leq p(\tilde{y} \mid x)$: RNN predicted that $\tilde{y}$ attains higher $p(y \mid x)$ than $\tilde{y}$, under assume that y^* is a better translation than $\tilde{y}$, thus RNN model is at fault.
  - Process:

Human, y^*	**Algorithm,** $\tilde{y}$	$p(y^* \mid x)$	$p(\tilde{y} \mid x)$	**At fault?**
Jane visits Africa in September.	Jane visited Africa last September.	2×10^{-10}	1×10^{-10}	Beam search
⋮	⋮	⋮	⋮	⋮

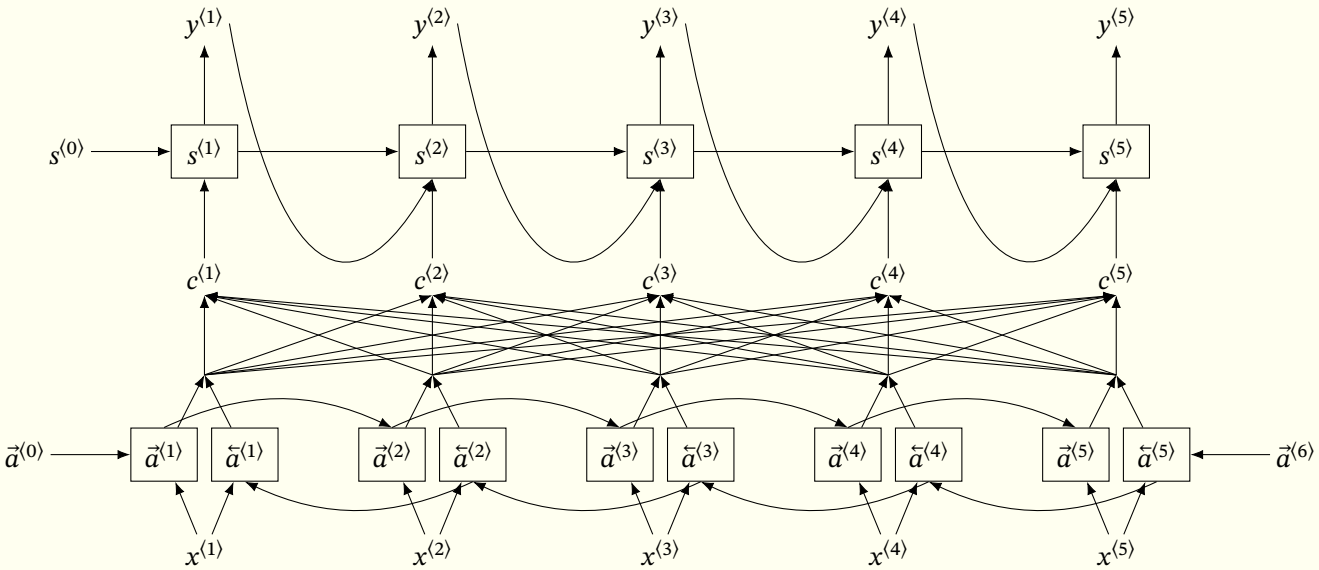
    * If beam search is responsible for a lot of errors, increase the beam width.
    * If the RNN model causes many errors, focus on improving the model (e.g., add regularization, use more data, or try a different architecture, etc.).
  - Summary: This error analysis process is useful when using approximate optimization algo-

rithms (like beam search) to optimize an objective or cost function learned by a learning model (e.g., sequence-to-sequence models or RNNs).

5.3.1.3 Attention Model

The attention model is a modification of the encoder–decoder architecture that improves machine translation. Although it was originally developed for machine translation, it has since been used in many other applications.

In a standard encoder–decoder model, the system reads the entire input sentence, compresses it into a single representation, and then generates the translation. This works well for short sentences, but performance drops for long sentences (e.g., $30 - 40$ words or more). In contrast, a human translator works step by step, i.e., reading part of a sentence, translating it, then continuing. The attention model behaves similarly by using attention weights to focus on different parts of the input at each step of generating the output.



Notations:

- $x^{\langle t'\rangle}$: Embedding vector (feature vector) of the t'-th word (input position t') (obtained from a one-hot representation via the embedding layer)
- $a^{\langle t'\rangle}$: Encoder (hidden) state (or encoder activation / annotation) at position t'.

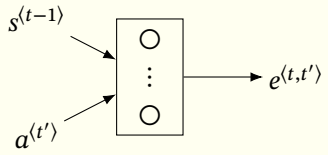
  For a bidirectional encoder,

$$a^{\langle t'\rangle} = \begin{bmatrix} \vec{a}^{\langle t'\rangle} \\ \overleftarrow{a}^{\langle t'\rangle} \end{bmatrix}$$

- $s^{\langle t\rangle}$: Decoder (hidden) state at time step t.

- $e^{\langle t,t' \rangle}$: Attention score (also energies) that measures how well the encoder state at position t' (i.e., $a^{\langle t' \rangle}$) and the decoder state at previous time step $t-1$ (i.e., $s^{\langle t-1 \rangle}$) match when generating output at time step t (i.e., $y^{\langle t \rangle}$).

  The attention (attention weight and attention score) at time step t depends on two things: the previous hidden state (since the current hidden state is not yet available) and the encoder activations. Specifically, the model uses the hidden state from the previous time step $s^{\langle t-1 \rangle}$ and the encoder activation $a^{\langle t' \rangle}$. Since the exact function is unknown, a small NN (typically with at least one hidden layer) is used to learn how to compute the attention scores,



  that is,

$$e^{\langle t,t' \rangle} = f_{\text{NN}}\left(s^{\langle t-1 \rangle}, a^{\langle t' \rangle}\right)$$

- $\alpha^{\langle t,t' \rangle}$: Attention weight that denotes how much attention the output at time step t (i.e., $y^{\langle t \rangle}$) should pay to the the encoder state at position t' (i.e., $a^{\langle t' \rangle}$), i.e., the normalized importance (a discrete probability distribution summing to 1, $\sum_{t'=1}^{T_x} \alpha^{\langle t,t' \rangle} = 1$) of $a^{\langle t' \rangle}$ when generating $y^{\langle t \rangle}$.

$$\alpha^{\langle t,t' \rangle} = \frac{\exp\left(e^{\langle t,t' \rangle}\right)}{\sum_{k=1}^{T_x} \exp\left(e^{\langle t,k \rangle}\right)}$$

- $c^{\langle t \rangle}$: Context vector for the output at time step t, computed as the weighted sum of encoder states,

$$c^{\langle t \rangle} = \sum_{t'=1}^{T_x} \alpha^{\langle t,t' \rangle} a^{\langle t' \rangle}$$

5.3.2 Speech Recognition - Audio Data

5.3.2.1 Speech Recognition

- Speech recognition: audio clip $x \rightarrow$ text transcript y

  Speech recognition is one of the most important applications of sequence-to-sequence models.

  To apply sequence-to-sequence models to audio data, a common preprocessing step is to convert raw audio into a spectrogram, which represents the audio signal as features over time.

  In academic settings, speech datasets typically contain a few thousand hours of labeled audio

(e.g., 300–3000 hours). In contrast, commercial systems are trained on much larger datasets, often over 10,000 or even 100,000 hours of audio.

Another effective method for speech recognition is the CTC (Connectionist Temporal Classification) loss. The idea is to use a neural network where:

- the input is a long sequence of audio features
- the output is a shorter sequence of text

In practice:

- the number of input time steps is much larger than the number of output steps
- models are usually bidirectional RNNs (with LSTM or GRU) and often deeper networks

- Trigger word detection: e.g., Amazon Echo, Apple Siri

5.4 Module 4 - Transformer Network

5.4.1 Transformer Network Intuition

Transformer networks (or transformers) are an architecture that many of the best NLP models today are based on.

As sequence tasks become more complex, the complexity of model increases, e.g., RNNs → GRU → LSTM. However, these are all sequential models that processes input one token at a time, with each step depending on the previous, with each step creating a bottleneck in the flow of information. In contrast, transformers process the whole sequence at once in parallel by using attention-based representations and a CNN-style (treat tokens as pixels) of parallel processing.

Two key ideas in transformer :

- Self-attention: compute representations for all tokens in a sequence in parallel, where each token's representation can attend to and incorporate information from other tokens.
- Multi-headed attention: apply self-attention multiple times in parallel (instead of once) to produce richer representations (multiple versions of the representation), where each head can learn different types of patterns or relationships.

5.4.2 Self-Attention

Section 5.3.1.3 explains how attention $\alpha^{\langle t,t' \rangle}$ is used to compute the context vector $c^{\langle t \rangle}$ in RNNs. For transformer attention, the idea is similar, where self-attention for each token i in the input sequence (sentence) is computed, i.e., $A\left(q^{\langle i \rangle}, K, V\right)$ or $A^{\langle i \rangle}$, where q denotes the query, K denotes the keys, and V denotes the values.

The key advantage of this representation for each token i is that it is not a fixed token embedding. Instead, it allows the self-attention mechanism to capture context and produce a richer and more meaningful representation of the token.

Specifically,

- For RNN attention, the context vector at time step t is

$$
\begin{aligned}
c^{\langle t \rangle} &= \sum_{t'=1}^{T_x} \alpha^{\langle t,t' \rangle} a^{\langle t' \rangle} \\
\alpha^{\langle t,t' \rangle} &= \frac{\exp\left(e^{\langle t,t' \rangle}\right)}{\sum_{k=1}^{T_x} \exp\left(e^{\langle t,k \rangle}\right)} \\
e^{\langle t,t' \rangle} &= f_{\text{NN}}\left(s^{\langle t-1 \rangle}, a^{\langle t' \rangle}\right)
\end{aligned}
$$

  - $\alpha^{\langle t,t' \rangle}$ is the attention weight.
  - $e^{\langle t,t' \rangle}$ is the attention score.

- For transformer self-attention, the attention-based (vector) representation of token i is

$$
\begin{aligned}
A^{\langle i \rangle} &= \sum_{r=1}^{T_x} \frac{\exp\left(q^{\langle i \rangle} \cdot k^{\langle r \rangle}\right)}{\sum_{s=1}^{T_x} \exp\left(q^{\langle i \rangle} \cdot k^{\langle s \rangle}\right)} v^{\langle r \rangle} \\
q^{\langle i \rangle} &= W^Q x^{\langle i \rangle} \\
k^{\langle i \rangle} &= W^K x^{\langle i \rangle} \\
v^{\langle i \rangle} &= W^V x^{\langle i \rangle}
\end{aligned}
$$

  - Each token i is associated with a query $q^{\langle i \rangle}$, a key $k^{\langle i \rangle}$, and a value $v^{\langle i \rangle}$.
    * The query $q^{\langle i \rangle}$ can be interpreted as a question posed by token i.
    * The keys $k^{\langle s \rangle}$ of all tokens are compared with this query using the dot product $q^{\langle i \rangle} \cdot k^{\langle s \rangle}$ (which corresponds to the attention score), which gives similarity scores measuring how relevant token s is as an answer to the question posed by token i.
    * The values $v^{\langle r \rangle}$ of all tokens are summed with weights $\frac{\exp\left(q^{\langle i \rangle} \cdot k^{\langle r \rangle}\right)}{\sum_{s=1}^{T_x} \exp\left(q^{\langle i \rangle} \cdot k^{\langle s \rangle}\right)}$ (which corresponds to the attention weights) to form $A^{\langle i \rangle}$, which aggregates information from all tokens.
  - Let

$$
Q = \begin{bmatrix} q^{\langle 1 \rangle} \\ q^{\langle 2 \rangle} \\ \vdots \\ q^{\langle T_x \rangle} \end{bmatrix}, \quad K = \begin{bmatrix} k^{\langle 1 \rangle} \\ k^{\langle 2 \rangle} \\ \vdots \\ k^{\langle T_x \rangle} \end{bmatrix}, \quad V = \begin{bmatrix} v^{\langle 1 \rangle} \\ v^{\langle 2 \rangle} \\ \vdots \\ v^{\langle T_x \rangle} \end{bmatrix}
$$

    A matrix form of the attention-based representations for all tokens

$$
\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}}\right) V
$$

    The term in the denominator $\sqrt{d_k}$ is used to scale the dot product to avoid exploding values. Another name for this type of attention is scaled dot-product attention.

5.4.3 Multi-Head Attention

A head is a single self-attention operation that uses learned projection matrices W^Q, W^K and W^V to compute queries $q^{\langle i \rangle}$, keys $k^{\langle i \rangle}$, and values $v^{\langle i \rangle}$ for all tokens, and then applies $\text{Attention}(Q, K, V)$. Multi-

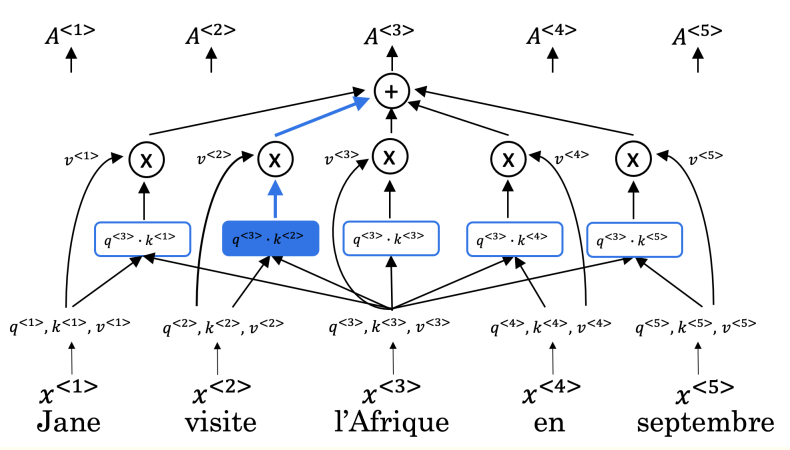
head attention refers to running multiple such self-attention heads in parallel and then concatenating and linearly projecting their outputs.

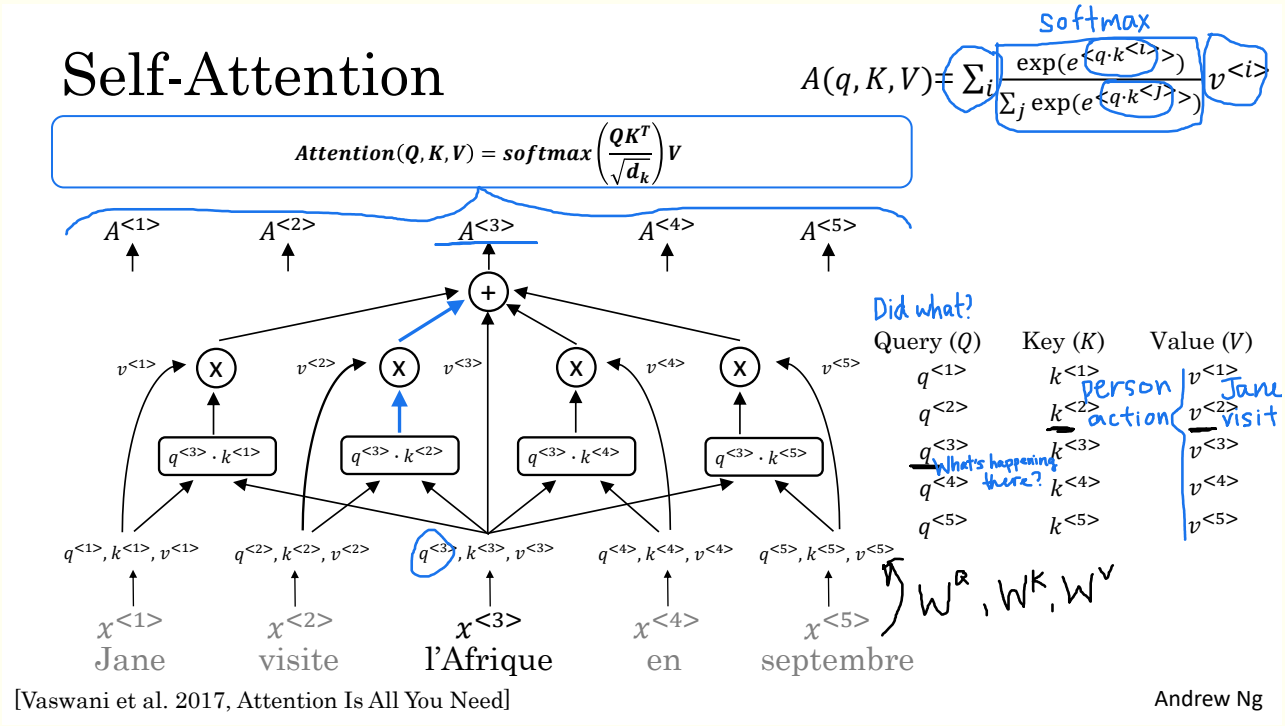
$$
\begin{aligned}
&\text{MultiHead}(Q, K, V) = \text{concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h)\, W_o \\
&\text{head}_1 = \text{Attention}(Q_1, K_1, V_1) = \text{Attention}(W_1^Q X, W_1^K X, W_1^V X) \\
&\text{head}_2 = \text{Attention}(Q_2, K_2, V_2) = \text{Attention}(W_2^Q X, W_2^K X, W_2^V X) \\
&\vdots \\
&\text{head}_h = \text{Attention}(Q_h, K_h, V_h) = \text{Attention}(W_h^Q X, W_h^K X, W_h^V X)
\end{aligned}
$$

Here, different heads head_i can be computed in parallel.

h is the total number of heads, which is a hyperparameter.

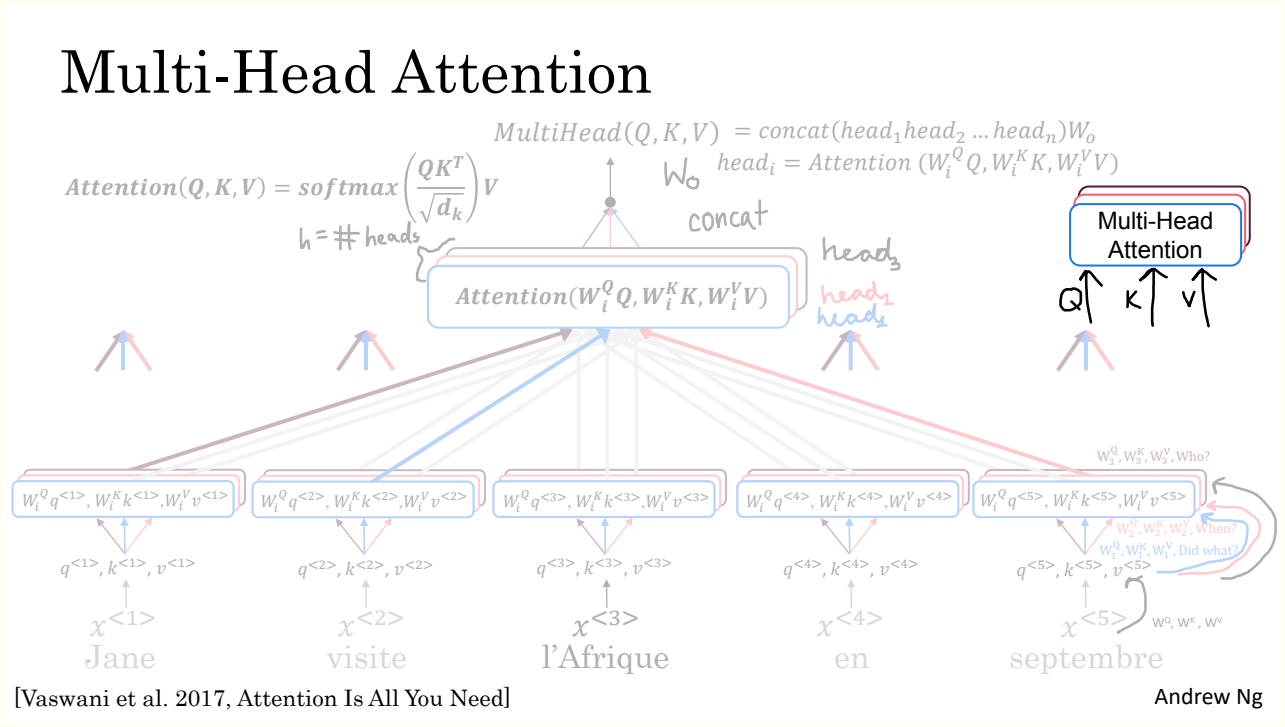
5.4.4 Transformer Network





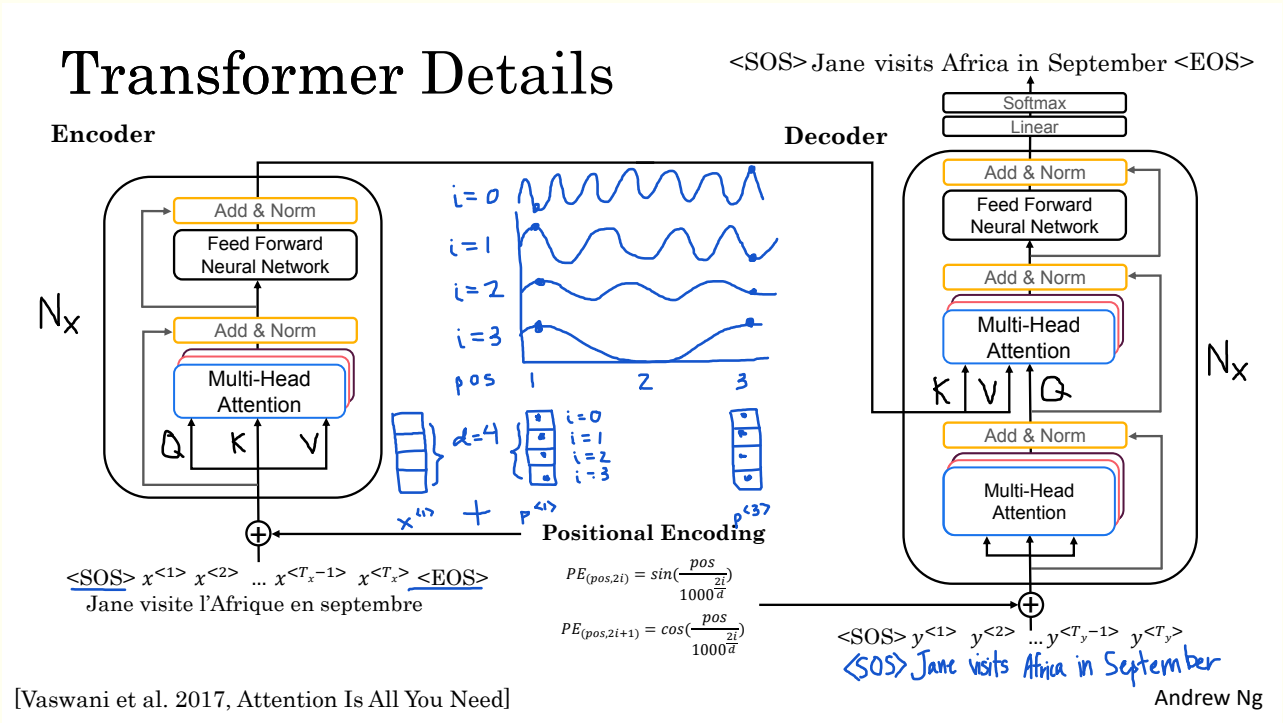
[Vaswani et al. 2017, Attention Is All You Need]

Andrew Ng



[Vaswani et al. 2017, Attention Is All You Need]

Andrew Ng



In sequence-to-sequence tasks, the relative order of tokens is very important for meaning. In RNNs, tokens (words) are processed one by one, so the model naturally receives information about sequence order. However, in Transformers using multi-head attention, all tokens are processed at the same time using multi-head attention. This makes training faster but removes built-in information about token positions.

To address this, positional encoding is introduced. Positional encoding can be regarded as a feature that encodes the absolute positions of tokens, which allows the model to infer relative positional relationships.

The positional encoding is defined as

- k-based form:

$$\mathrm{PE}(\mathrm{pos}, k) = \begin{cases} \sin\left(\dfrac{\mathrm{pos}}{10000^{\frac{k}{d}}}\right), & \text{if } k \text{ is even} \\ \cos\left(\dfrac{\mathrm{pos}}{10000^{\frac{k-1}{d}}}\right), & \text{if } k \text{ is odd} \end{cases}$$

with $k = 0, 1, \ldots, d-1$ and $\mathrm{pos} = 0, 1, \ldots, T_x - 1$. Here, d is the embedding dimension.

- Pair-based form (which is equivalent to k-based form):

$$\text{PE}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{d}}}\right) = \sin(\theta(\text{pos}, i))$$
$$\text{PE}(\text{pos}, 2i+1) = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d}}}\right) = \cos(\theta(\text{pos}, i))$$

  with $i = 0, 1, \ldots, \frac{d}{2} - 1$. Here, each pair $(2i, 2i+1)$ shares the same frequency $\frac{1}{10000^{\frac{2i}{d}}}$ through the same angle $\theta(\text{pos}, i) = \frac{\text{pos}}{10000^{\frac{2i}{d}}}$.

Chapter 6

Supplementary Information

6.1 Material

Course Platform (Deep Learning Specialization):

- Course 1: Neural Networks and Deep Learning
  - Slides
  - Course Forum: DeepLearning.AI Forum
  - Feedforward Neural Networks in Depth (Optional Reading)
- Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization
  - Slides
- Course 3: Structuring Machine Learning Projects
  - Slides
- Course 4: Convolutional Neural Networks
  - Slides
- Course 5: Sequence Models
  - Slides
  - Understanding LSTM Networks (Blog)