# CS 230: Deep Learning

Hargen Zheng

December 31, 2023

# Contents

# Part I

# Neural Networks and Deep Learning

# Chapter 1

# Introduction

The objectives of this chapter are

- Discuss the major trends driving the rise of deep learning.
- Explain how deep learning is applied to supervised learning.
- List the major categories of models (CNNs, RNNs, etc.), and when they should be applied
- Assess appropriate use cases for deep learning

Why it matters?

- AI is the new Electricity.
- Electricity had once transformed countless industries: transportation, manufacturing, healthcare, communications, and more.
- AI will now bring about an equally big transformation.

Courses in this sequence (Specialization):

1. Neural Networks and Deep Learning
2. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization
3. Structuring your Machine Learning project
4. Convolutional Neural Networks (CNNs)
5. Natural Language Processing: Building sequence models (RNNs, LSTM)

## 1.1 What is a Neural Network?

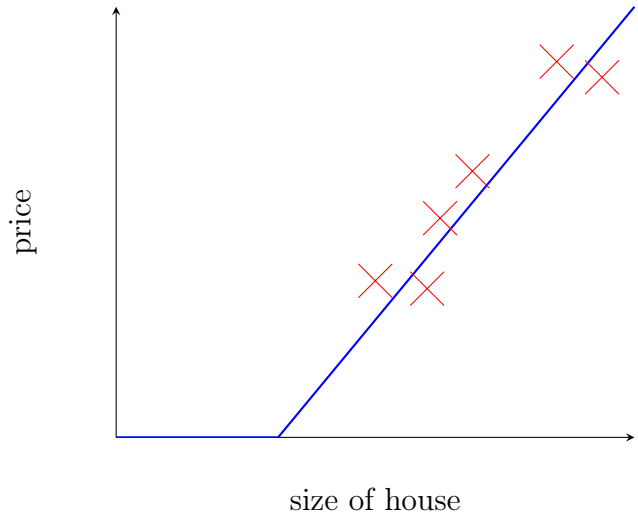



Figure 1.1: Housing Price Prediction

In the neural network literature, this function appears by a lot. This function is called a ReLU function, which stands for rectified linear units.

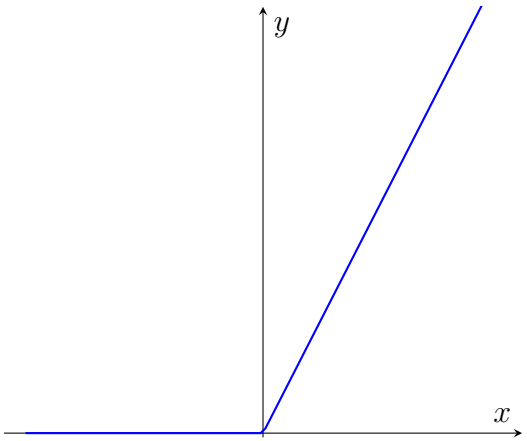



Figure 1.2: ReLU function: $\max\{0, x\}$

With the size of houses in square feet or square meters and the price of the house, we can fit a function to predict the price of a house as a function of its size.

This is the simplest neural network. We have the size of a house as input $x$, which goes into a node (a single "neuron"), and outputs the price $y$.

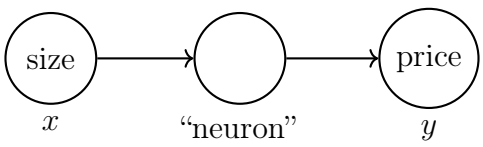



Figure 1.3: Simple Neural Network of Housing Example

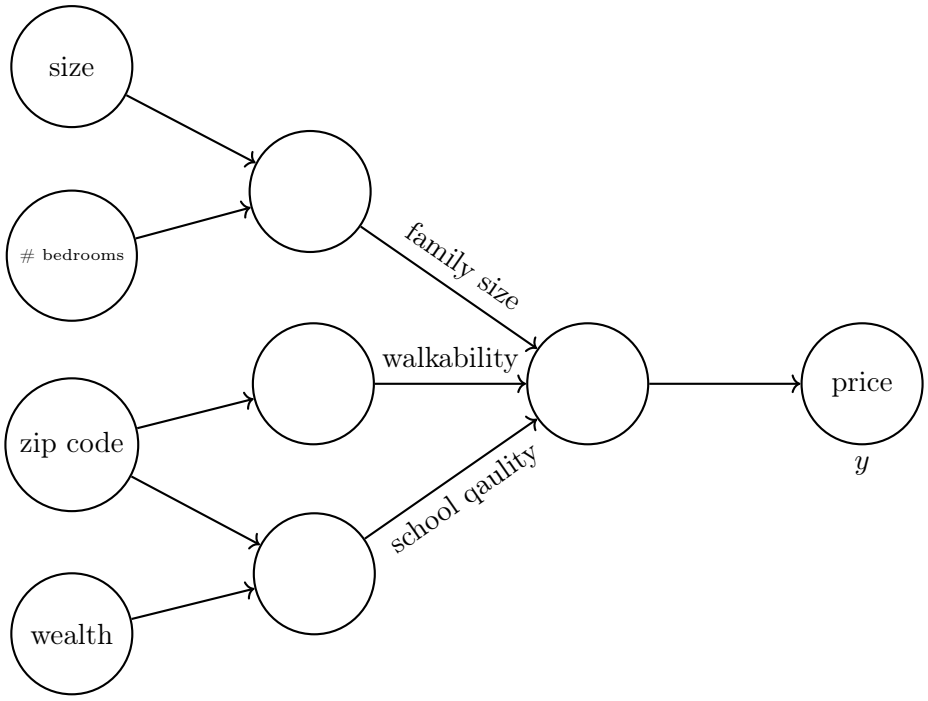



Figure 1.4: Slightly Larger Neural Network

Suppose that, instead of predicting the price of a house just from the size, we also have other features, such as the number of bedrooms, zip code, and wealth. The number of bedrooms determines whether or not a house can fit one's family size; Zip code tells walkability; and zip code and wealth tells how good the school quality is. Each of the circle could be ReLU or other nonlinear function.

We need to give the neural network the input $x$ and the output $y$ for a number of examples in the training set and, for all the things in the middle, neural network will figure out by itself.

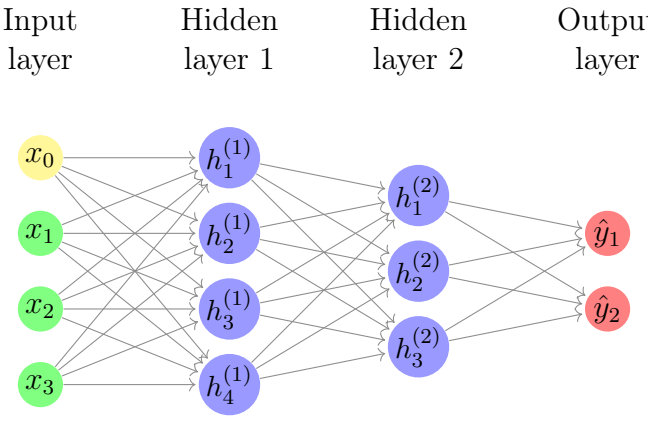



Figure 1.5: General Picture of Neural Network

## 1.2 Supervised Learning with Neural Networks

Supervised Learning Examples.

| Input($x$) | Output($y$) | Application | Type of NN |
|---|---|---|---|
| Home features | Price | Real Estate | Standard |
| Ad, user info | Click on ad? (0/1) | Online Advertising | Standard |
| Image | Object $(1, \ldots, 1000)$ | Photo tagging | CNN |
| Audio | Text transcript | Speech recognition | RNN |
| English | Chinese | Machine translation | RNN |
| Image, Radar info | Position of other cars | Autonomous driving | Custom |

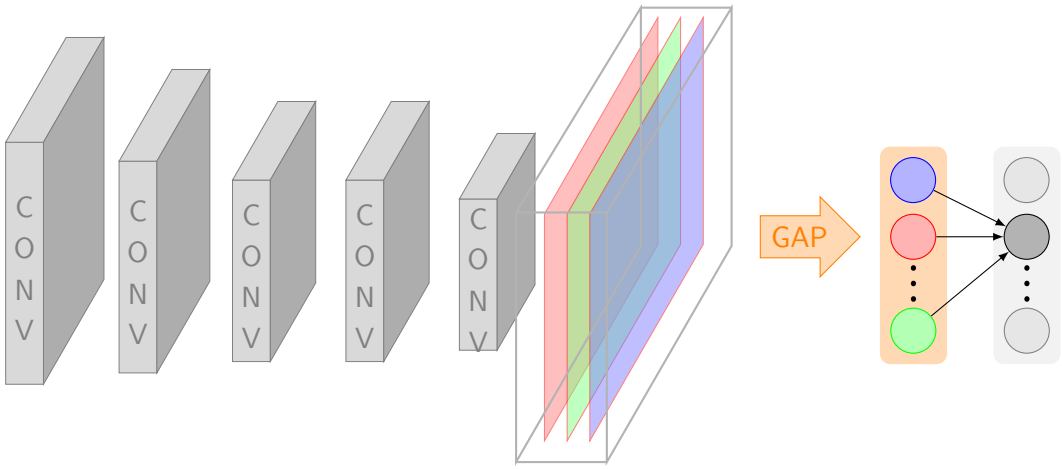



Figure 1.6: Convolutional Neural Network

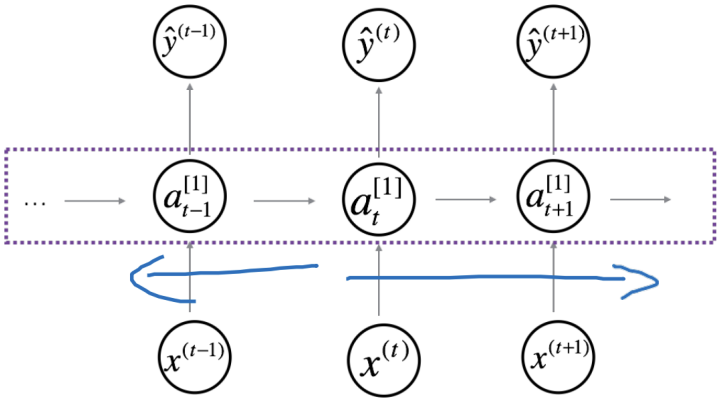



Figure 1.7: Recurrent Neural Network

There are two types of data: Structured Data and Unstructured Data. We could have the following examples for each.

| Size | #bedrooms | $\cdots$ | Price(1000$s) |
|---|---|---|---|
| 2104 | 3 | | 400 |
| 1600 | 3 | | 330 |
| 2400 | 3 | | 369 |
| $\vdots$ | $\vdots$ | | $\vdots$ |
| 3000 | 4 | | 540 |

| User Age | Ad ID | $\cdots$ | Click |
|---|---|---|---|
| 41 | 93242 | | 1 |
| 80 | 93287 | | 0 |
| 18 | 87312 | | 1 |
| $\vdots$ | $\vdots$ | | $\vdots$ |
| 27 | 71244 | | 1 |

Table 1.1: Structured Data

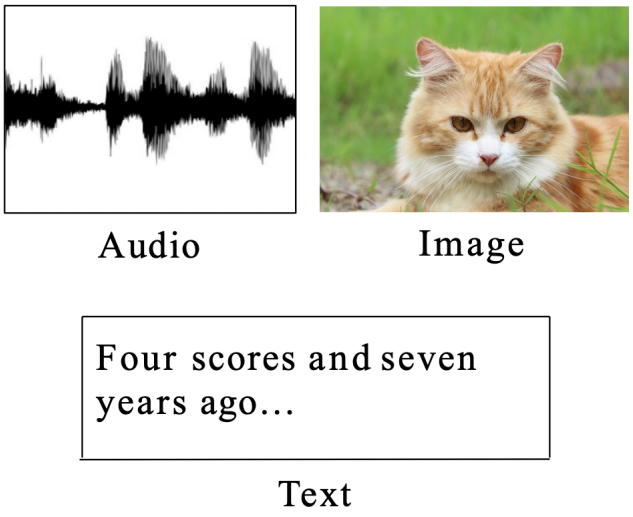



Figure 1.8: Unstructured Data

Historically, it has been much harder for computers to make sense of unstructured data compared to structured data. Thanks to the neural networks, computers are better at interpreting unstructured data as well, compared to just a few years ago.

## 1.3 Why is Deep Learning Taking off?

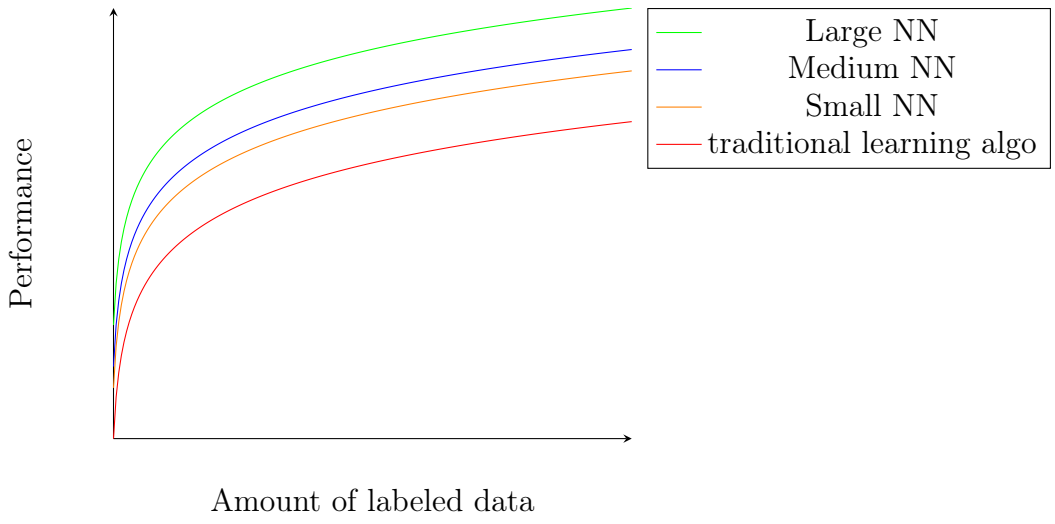



Figure 1.9: Scale drives deep learning progress

In the regime of small number of training sets, the relative ordering of the algorithms is not very well defined. If you don't have a lot of training data, it is often up to your skill at hand engineering features that determines performance. When we have large training sets – large labeled data regime in the right, we more consistently see large neural networks dominating the other approaches.

# Chapter 2

# Neural Networks Basics

The objectives of this chapter are

- Build a logistic regression model structured as a shallow neural network
- Build the general architecture of a learning algorithm, including parameter initialization, cost function and gradient calculation, and optimization implemetation (gradient descent)
- Implement computationally efficient and highly vectorized versions of models
- Compute derivatives for logistic regression, using a backpropagation mindset
- Use Numpy functions and Numpy matrix/vector operations
- Work with iPython Notebooks
- Implement vectorization across multiple training examples
- Explain the concept of broadcasting

## 2.1 Logistic Regression as a Neural Network

When implementing a neural network, you usually want to process entire training set without using an explicit for-loop. Also, when organizing the computation of a neural network, usually you have what's called a forward pass or forward propagation step, followed by a backward pass or backward propagation step.

### 2.1.1 Notation

Each training set will be comprised of $m$ training examples:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}.$$

We denote $m_{train}$ to be the number of training examples in the training set, and $m_{test}$ to be the number of test examples. The $i$th training example is represented

by a pair $(x^{(i)}, y^{(i)})$, where $x^{(i)} \in \mathbb{R}^{n_x}$ and $y^{(i)} \in \{0, 1\}$. To put all of the training examples in a more compact notation, we define matrix $X$ as

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix}$$

where $X \in \mathbb{R}^{n_x \times m}$. Similarly, we define $Y$ as

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

where $Y \in \mathbb{R}^{1 \times m}$.

### 2.1.2 Binary Classification


Figure 2.1: Cat Image

You might have an input of an image and want to output a label to recognize this image as either being a cat, in which case you output 1, or non-cat, in which case you output 0. We use $y$ to denote the output label.

To store an image, computer stores three separate matrices corresponding to the red, green, and blue color channels of this image. If the input image is 64 pixels by 64 pixels, then you would have 3 $64 \times 64$ matrices corresponding to the red, green, and blue pixel intensity values for the image. We could define a feature vector $x$ as follows:

$$x = \begin{bmatrix} | & | & | \\ red & green & blue \\ | & | & | \end{bmatrix}.$$

Suppose each matrix has dimension $64 \times 64$, then

$$n_x = 64 \times 64 \times 3 = 12288.$$

### 2.1.3 Logistic Regression

Given an input feature $x \in \mathbb{R}^{n_x}$, we want to find a prediction $\hat{y}$, where we want to interpret $\hat{y}$ as the probability of $y = 1$ for a given set of input features $x$.

$$\hat{y} = P(y = 1|x).$$

Define the weights $w \in \mathbb{R}^{n_x}$ and bias term $b \in \mathbb{R}$ in the logistic regression. It turns out, when implementing the neural networks, it is better to keep parameters $w$ and $b$ separate, instead of putting them together as $\theta \in \mathbb{R}^{n_x+1}$.

In linear regression, we would use $\hat{y} = w^T x + b$. However, this is not good for binary classification because the prediction could be much bigger than 1 or even negative, which does not make sense for probability. Therefore, we apply the sigmoid function, where

$$\hat{y} = \sigma(w^T x + b) = \sigma(z).$$

The formula of the sigmoid function is given as follows:

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

If $z$ is very large, then $e^{-z}$ will be close to zero, so $\sigma(z) \approx 1$. Conversely, if $z$ is very small or a very large negative number, then $\sigma(z) \approx 0$.

The graph of the sigmoid function is given below:

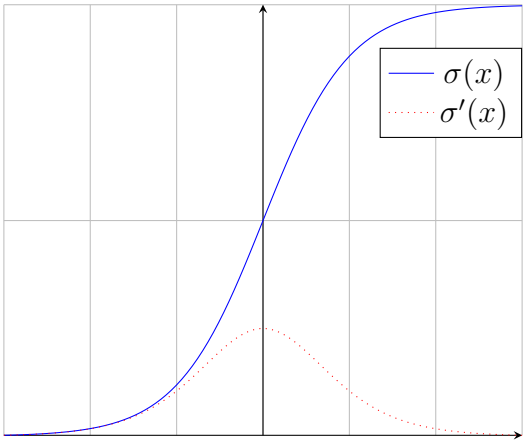



Figure 2.2: Sigmoid Function

### 2.1.4 Logistic Regression Cost Function

Logistic regression is a supervised learning algorithm, where given training examples $\{(x^{(1)}, y^{(1)}, \ldots, (x^{(m)}, y^{(m)})\}$, we want to find parameters $w$ and $b$ such that $\hat{y}^{(i)} \approx y^{(i)}$.

MSE is not used as the loss function because when learning parameters, the optimization problem becomes non-convex, where we end up with multiple local optima, so gradient descent may not find a global optimum. The loss function is defined to measure how good our output $\hat{y}$ is when the true label is $y$. Instead, the loss (error) function for logistic regression could be represented by

$$\mathcal{L}(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})).$$

The intuition behind is that when considering the squared error cost function, we want the squared error to be as small as possible. Similarly, we want to loss function for logistic regression to be as small as possible.

If $y = 1$, then the loss function ends up with

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

In this case, we want $\log(\hat{y})$ to be large $\Leftrightarrow$ want $\hat{y}$ to be large.
If $y = 0$, then

$$\mathcal{L}(\hat{y}, 0) = -\log(1 - \hat{y}).$$

In this case, we want $\log(1 - \hat{y})$ to be large $\Leftrightarrow$ want $\hat{y}$ to be small.

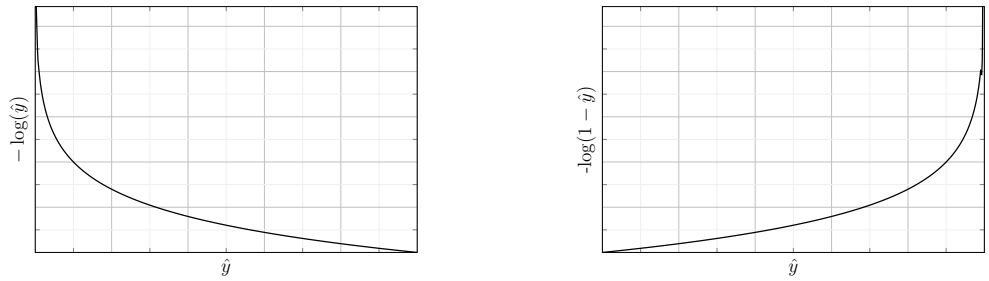



Figure 2.3: A binary classifier's loss function based on if $y = 1$ or $y = 0$.

The loss function is defined with respect to a single training example. The cost function measures how well the model is doing on the entire training set, which is given as

$$\begin{aligned} J(w, b) &= \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})). \end{aligned}$$

**Interpretation of the Cost Function.** We interpret $\hat{y}$ to be the probability of $y = 1$ given a set of input features $x$. This means that

- If $y = 1$, $P(y|x) = \hat{y}$
- If $y = 0$, $P(y|x) = 1 - \hat{y}$

We can summarize the two equations as follows:

$$P(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}.$$

Since the log function is a strictly monotonically increasing function, maximizing $\log(P(y|x))$ would give a similar result as maximizing $P(y|x)$.

$$\begin{aligned} \log(P(y|x)) &= \log(\hat{y}^y (1 - \hat{y})^{1-y}) \\ &= y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \\ &= -\mathcal{L}(\hat{y}, y). \end{aligned}$$

Therefore, minimizing the loss means maximizing the probability.
The above was for one training example. For $m$ examples, we can express the probability as follows:

$$\begin{aligned} \log(P(\text{labels in training set})) &= \log(\prod_{i=1}^{m} P(y^{(i)}|x^{(i)})) \\ &= \sum_{i=1}^{m} \log(P(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \end{aligned}$$

The cost is defined as

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

We get rid of the negative sign as now we want to minimize the cost. $\frac{1}{m}$ is added in the front to make sure our quantities are at a better scale. To minimization of the cost function carries out maximum likelihood estimation with the logistic regression model, under the assumption that our training examples are i.i.d (independent and identically distributed).

### 2.1.5 Gradient Descent

We want to find $w, b$ that minimize $J(w,b)$. Our cost function $J(w,b)$ is a single big bowl, which is a convex function, which is one of the huge reasons why we use this particular cost function $J$ for logistic regression.

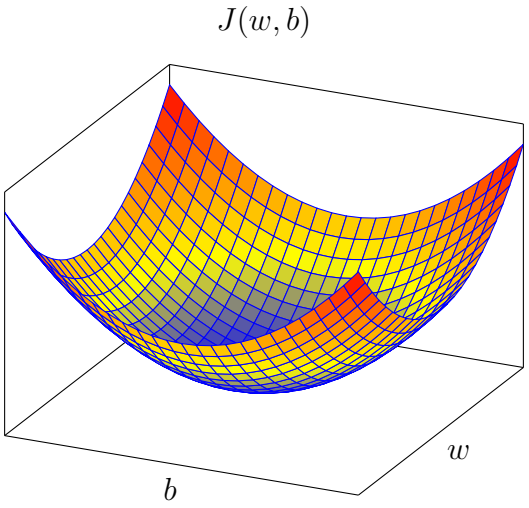



Figure 2.4: Convex Cost Function

For logistic regression, almost any initialization method works. Though random initialization works, people don't usually do that for logistic regression. Instead, we generally initialize the weights to 0. Gradient descent takes a step in the steepest downhill direction. After iterations, we converge to the global optimum or get close to the global optimum. In one dimension, for the parameter $w$, we can define the gradient descent algorithm as follows

$$w := w - \alpha\frac{\partial J(w,b)}{\partial w}$$
$$b := b - \alpha\frac{\partial J(w,b)}{\partial b}$$

where $\alpha$ is the learning rate that control the size of steps and $\frac{\partial J(w)}{\partial w}, \frac{\partial J(w,b)}{\partial b}$ are the changes we make for the parameter $w, b$.

### 2.1.6 Computation Graph

The computations of neural network are organized in terms of a forward pass step, in which we compute the output of the neural network, followed by a backward propagation step, which we use to compute the gradients. The computation graph explains why it is organized this way.

Suppose we want the compute the function

$$J(a, b, c) = 3(a + bc).$$

We can break down the computation into three steps:

1. $u = bc$
2. $v = a + u$
3. $J = 3v$

This could be represented by the computation graph below:

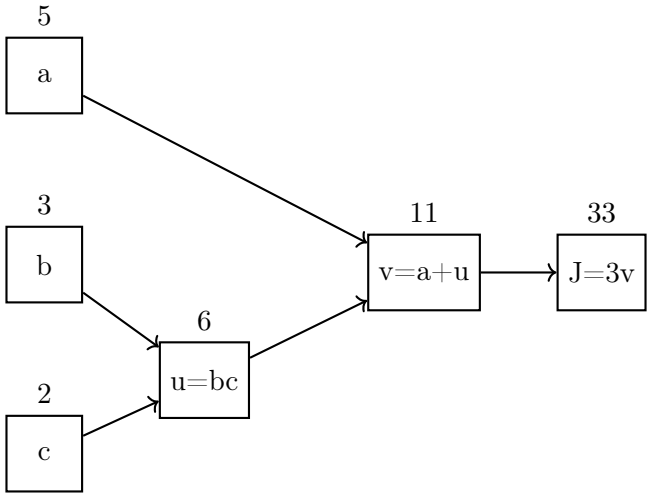



Figure 2.5: Computation Graph for $J(a, b, c) = 3(a + bc)$

Suppose $a = 5, b = 3, c = 2$, then $J(a, b, c) = 3(a + bc) = 3(5 + 3 \times 2) = 33$.

### 2.1.7 Derivatives with a Computation Graph

Consider the computation graph 2.5. Suppose we want to compute the derivative of $J$ with respect to $v$. Since we know $J = 3v$, we can take the derivative as follows:

$$\frac{\partial J}{\partial v} = \frac{\partial}{\partial v}(3v) = 3.$$

Suppose then we want to compute the derivative of $J$ with respect to $a$. We can apply the chain rule as follows:

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a} = 3 \cdot \frac{\partial}{\partial a}(a + u) = 3 \cdot 1 = 3.$$

For brevity, we could use "$da$" to represent $\frac{\partial J}{\partial a}$. Similarly, we can use "$dv$" to represent $\frac{\partial J}{\partial v}$.

To find the derivative of $J$ with respect to $u$, we could do similar computation as $\frac{\partial J}{\partial a}$ as follows:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u} = 3 \cdot 1 = 3.$$

$\frac{\partial J}{\partial b}$ could be compute as follows:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial b} = 3 \cdot c = 3 \cdot 2 = 6.$$

Similarly, we can compute

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial c} = 3 \cdot b = 3 \cdot 3 = 9.$$

The most efficient way to compute all these derivatives is through a right-to-left computation following the direction of the arrows below:

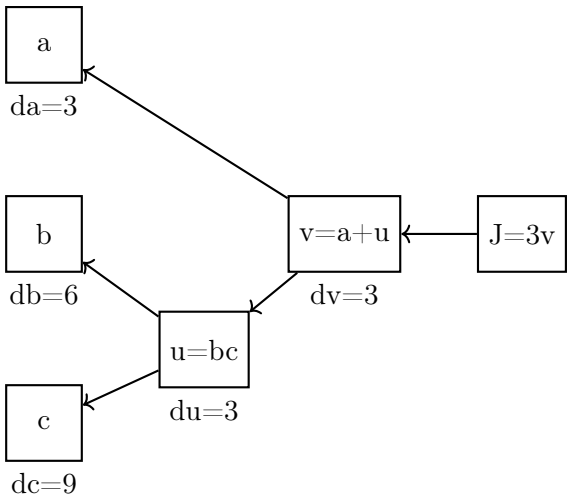



Figure 2.6: Backward Propagation for $J(a, b, c) = 3(a + bc)$

### 2.1.8 Logistic Regression Gradient Descent

This section will cover how to compute derivatives to implement gradient descent for logistic regression.

To recap, logistic regression is set up as follows:

$$\begin{aligned} z &= w^T x + b \\ \hat{y} &= a = \sigma(z) \\ \mathcal{L}(a, y) &= -(y \log(a) + (1 - y) \log(1 - a)) \end{aligned}$$

This can be represented by the computation graph below:

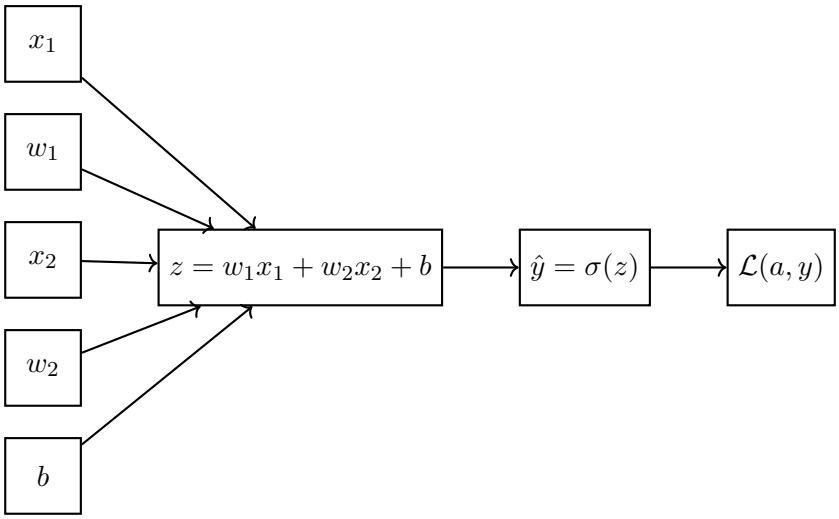



Figure 2.7: Computation Graph for Logistic Regression

In this case, we can compute "$da$" as follows:

$$da = \frac{\partial \mathcal{L}(a,y)}{\partial a} = -y \cdot \frac{1}{a} - (1-y) \cdot \frac{1}{1-a} \cdot (-1) = -\frac{y}{a} + \frac{1-y}{1-a}.$$

We can give both terms the same denominator and clean up as follows:

$$\frac{\partial \mathcal{L}(a,y)}{\partial a} = \frac{-y(1-a)}{a(1-a)} + \frac{a(1-y)}{a(1-a)} = \frac{-y + ay + a - ay}{a(1-a)} = \frac{a-y}{a(1-a)}.$$

Then, we can go backwards and compute $\frac{\partial \mathcal{L}(a,y)}{\partial z}$. We know that the derivative of a sigmoid function has the form $\frac{\partial}{\partial z}\sigma(z) = \sigma(z)(1-\sigma(z))$. In this case $\sigma(z) = a$, as we have defined, so $\frac{\partial a}{\partial z} = a(1-a)$. Therefore,

$$dz = \frac{\partial \mathcal{L}(a,y)}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} = \frac{a-y}{a(1-a)} \times a(1-a) = a - y.$$

Finally, we do the backward propagation for the input layer as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}(a,y)}{\partial w_1} &= x_1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = x_1 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial w_2} &= x_2 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = x_2 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial x_1} &= w_1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = w_1 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial x_2} &= w_2 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = w_2 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial b} &= 1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = (a-y)
\end{aligned}$$

### 2.1.9 Gradient Descent on $m$ examples

Recall that the cost function for $m$ training examples is given by

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y),$$

where $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$.

The cost function is the average of loss values for each training example. It turns out that derivative works similarly and we can compute the derivative of $J(w, b)$ with respect to $w_1$ as follows:

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_i} \mathcal{L}(a^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} x_1 \cdot (a^{(i)} - y^{(i)}).$$

Now, let's formalize the algorithm for the logistic regression gradient descent.

**Algorithm 1** Gradient Descent for Logistic Regression

**Initialize** $J = 0; dw_1 = 0; dw_2 = 0; db = 0$
**for** $i = 1$ to $m$ **do**
  $z^{(i)} \leftarrow w^T x^{(i)} + b$
  $a^{(i)} \leftarrow \sigma(z^{(i)})$
  $J \leftarrow J + [-(y^{(i)} \log(a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))]$
  $dz^{(i)} \leftarrow a^{(i)} - y^{(i)}$ ▷ Superscript refers to one training example
  $dw_1 \leftarrow dw_1 + x_1^{(i)} dz^{(i)}$ ▷ Here we assume $n_x = 2$
  $dw_2 \leftarrow dw_2 + x_2^{(i)} dz^{(i)}$ ▷ No superscript i as it's accumulative
  $db \leftarrow db + dz^{(i)}$
**end for**
$J \leftarrow J/m$
$dw_1 \leftarrow dw_1/m; dw_2 \leftarrow dw_2/m; db \leftarrow db/m$

To implement logistic regression this way, we have two for-loops. The first for-loop is to iterate through all training examples. The second for-loop is to iterate through all the features. In the example above, we only have features $w^{(1)}, w^{(2)}$. If we have $n$ features instead, then we need to iterate through all of them.

However, when implementing deep learning algorithms, the explicit for-loop makes the algorithms less efficient. The vectorization technique allows us to get rid of the explicit for-loops, which allows us to scale to larger datasets.

## 2.2 Python and Vectorization

### 2.2.1 Vectorization

The aim of vectorization is to remove explicit for-loops in code. In the deep learning era, we often train on relatively large dataset because that's when deep learning algorithms tend to shine. In deep learning, the ability to perform vectorization has become a key skills.

Recall that in logistic regression, we compute $z = w^T x + b$, where $w, x \in \mathbb{R}^{n_x}$. For non-vectorized implementation, we compute $z$ as

$$z = \sum_{i=1}^{n_x} w[i] \times x[i], ; z+ = b$$

The vectorized implementation is very fast:

$$z = np.dot(w, x) + b.$$

SIMD (single instruction multiple data), in both GPU and CPU, enables built-in functions, such as $np.dot()$, to take much better advantage of parallelism to do computations much faster.

**Neural network programming guideline:** whenever possible, avoid explicit for-loops.

If we want to compute $u = Av$, then by the definition of matrix multiplication,

$$u_i = \sum_j A_{ij} v_j.$$

The vectorized implementation would be

$$u = np.dot(A, v).$$

Suppose we want to apply the exponential operation on every element of a vector

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

We can apply the vectorization technique as follows:

$$u = np.exp(v).$$

Similarly, we have $np.log(v), np.abs(v), np.maximum(v, 0), v**2$, etc. to apply the vectorization technique. We can revise the original gradient descent algorithm and use the vectorization technique as follows

**Algorithm 2** Gradient Descent for Logistic Regression with Vectorization

**Initialize** $J = 0; dw = np.zeros((n_x, 1)); db = 0$
**for** $i = 1$ to $m$ **do**
  $z^{(i)} \leftarrow w^T x^{(i)} + b$
  $a^{(i)} \leftarrow \sigma(z^{(i)})$
  $J \leftarrow J + [-(y^{(i)} \log(a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))]$
  $dz^{(i)} \leftarrow a^{(i)} - y^{(i)}$
  $dw \leftarrow dw + x^{(i)} dz^{(i)}$ ▷ Get rid of the for-loop
  $db \leftarrow db + dz^{(i)}$
**end for**
$J \leftarrow J/m$
$dw \leftarrow dw/m; db \leftarrow db/m$

### 2.2.2 Vectorizing Logistic Regression

For logistic regression, we need to compute $a^{(i)} = \sigma(z^{(i)})$ $m$ times, where $z^{(i)} = w^T x^{(i)} + b$. There is a way to compute without using explicit for-loops. Recall that we define

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix},$$

where $X \in \mathbb{R}^{n_x \times m}$.

We want to compute all $z^{(i)}$'s at the same time.

From math, we know that

$$\begin{aligned} Z &= \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} \\ &= w^T X + \begin{bmatrix} b & b & \cdots & b \end{bmatrix} \\ &= \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & \cdots & w^T x^{(m)} + b \end{bmatrix} \end{aligned}$$

In order to implement this in NumPy, we can do

$$Z = np.dot(w.T, X) + b,$$

where $b \in \mathbb{R}$. Python will expand this real number to a $1 \times m$ vector by **broadcasting**.

Similarly, we can stack lower case $a^{(i)}$'s horizontally to obtain capital $A$ as follows:

$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \sigma(Z).$$

With $Z$ being a $1 \times m$ vector, we can apply the sigmoid function to each element of $Z$ by

$$A = \frac{1}{1 + np.exp(-Z)}.$$

### 2.2.3 Vectorizing Logistic Regression's Gradient Computation

Recall that we compute individual $dz$'s as $dz^{(i)} = a^{(i)} - y^{(i)}$ for the $i$th training example. Now, let's define

$$dZ = \begin{bmatrix} dz^{(1)} & dz^{(2)} & \cdots & dz^{(m)} \end{bmatrix},$$

where $dz^{(i)}$'s are stacked horizontally and $dZ \in \mathbb{R}^{1 \times m}$.

Since we have

$$A = \begin{bmatrix} a^{(1)} & \cdots & a^{(m)} \end{bmatrix}, \; Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(m)} \end{bmatrix}$$

we can compute $dZ$ as

$$dZ = A - Y = \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & \cdots & a^{(m)} - y^{(m)} \end{bmatrix}.$$

We can compute $db$ with vectorization as

$$\begin{aligned} db &= \frac{1}{m} \sum_{i=1}^{m} dz^{(i)} \\ &= \frac{1}{m} \times np.sum(dZ). \end{aligned}$$

Similarly, we can compute $dw$ with vectorization as

$$\begin{aligned} dw &= \frac{1}{m} X dZ^T \\ &= \frac{1}{m} \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix} \\ &= \frac{1}{m} \begin{bmatrix} x^{(1)} dz^{(1)} + \cdots + x^{(m)} z^{(m)} \end{bmatrix}, \end{aligned}$$

where $dw \in \mathbb{R}^{n_x \times 1}$.

### 2.2.4 Broadcasting in Python

The following matrix shows the calories from Carbs, Proteins, Fats in 100g of different foods:

$$\begin{array}{r} \\ \text{Carb} \\ \text{Protein} \\ \text{Fat} \end{array} \begin{array}{c} \begin{array}{cccc} \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \end{array} \\ \begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix} \end{array}$$

Suppose we want to calculate the percentage of calories from Carb, Protein, and Fat for each of the four foods. For example, for apples, the percentage of calories for Carb is

$$\frac{56.0}{59.0} \approx 94.9\%.$$

We could do this without explicit for-loops. Suppose the matrix is denoted by $A$, where $A \in \mathbb{R}^{3\times 4}$.

To sum each column vertically, we can use

$$cal = A.sum(axis = 0).$$

$axis = 0$ refers to the vertical columns inside the matrix, whereas $axis = 1$ will sum horizontally. The $cal = cal.reshape(1, 4)$ operation would convert $s$ to a one by four vector, which is very cheap to call as it's runtime is $\mathcal{O}(1)$.

Then, we can compute percentages by

$$percentage = 100 * A/(cal.reshape(1, 4)),$$

where we divide a $(3, 4)$ matrix by a $(1, 4)$ vector. This will divide each of the three elements in a column by the value in the corresponding column of $cal.reshape(1, 4)$.

A few more examples follows are provided below. Suppose we want to add 100 to every element of the vector $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$. We can do

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100.$$

Python will convert the number 100 into the vector

$$\begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}$$

and so

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \Rightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}.$$

If we add a $(m, n)$ matrix by a $(1, n)$ vector, Python will copy the vector $m$ times to convert it into an $(m, n)$ matrix, and then add two matrices together.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}.$$

Now, suppose we add a $(m, n)$ matrix by a $(m, 1)$ vector. Then, Python will copy the vector $n$ times horizontally and form a $(m, n)$ matrix.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}.$$

### 2.2.5 A Note on Python/NumPy Vectors

Let's set

$$a = np.random.randn(5),$$

which creates five random Gaussian variables stored in array $a$. It turns out if we do $print(a.shape)$, we get $(5, )$. This is called a **rank one array**, which is neither a row vector nor a column vector. In this case, $a.T$ and $a$ look the same. When we call $np.dot(a, a^T)$, we will get a single number, instead of a matrix generated by the outer product.

When doing neural network computations, it is recommended not to use rank one array. Instead, we can create a $(5, 1)$ vector as follows:

$$a = np.random.randn(5, 1),$$

which generates a column vector. Now, if we call $a.T$, we will get a row vector. This way, the behaviors of the vector are easier to understand.

When dealing with a matrix with unknown dimensions, we can use an assertion statement to make sure the shape is as desired:

$$assert(a.shape == (5, 1)).$$

Also, if we ultimately encounter the rank one array, we can always reshape it to the column vector as follows:

$$a = a.reshape((5, 1)),$$

so it behaves more consistently as a column vector, or a row vector.

# Chapter 3

# Shallow Neural Networks

The objectives of this chapter are

- Describe hidden units and hidden layers
- Use units with a non-linear activation function, such as tanh
- Implement forward and backward propagation
- Apply random initialization to your neural network
- Increase fluency in Deep Learning notations and Neural Network Representations
- Implement a 2-class classification neural network with a single hidden layer
- Compute the cross entropy loss

## 3.1 Neural Networks Overview

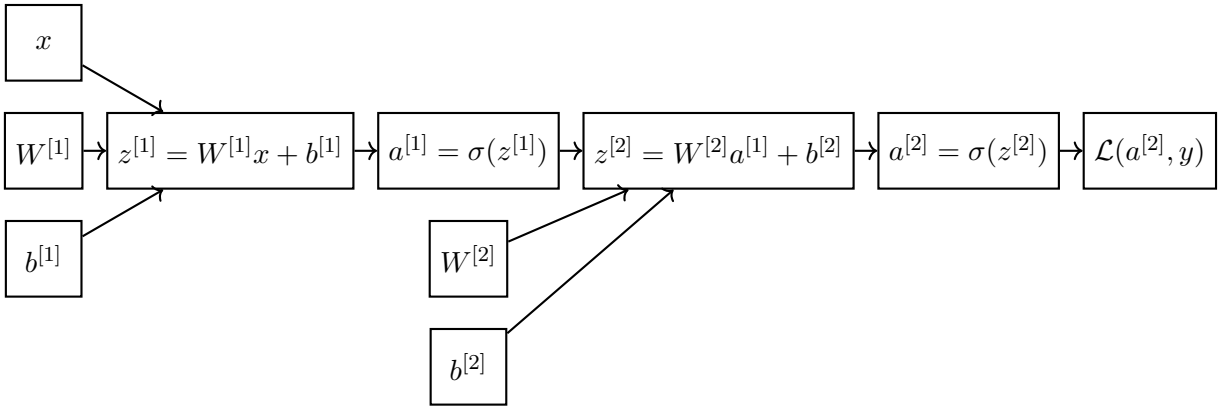



Figure 3.1: Shallow Neural Network

After forward pass, we do backward propagation to compute the derivatives

$$da^{[2]}, dz^{[2]}, dW^{[2]}, db^{[2]}, da^{[1]}, dz^{[1]}, dW^{[1]}, db^{[1]}, dx.$$

This is basically taking logistic regression and repeating it twice.

## 3.2 Neural Network Representation

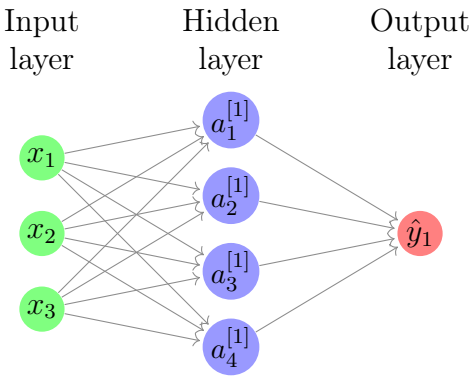



Figure 3.2: Neural Network Representation

In a neural network that we train with supervised learning, the training set contains values of the inputs $x$ as well as the target outputs $y$. The term "Hidden layer" refers to the fact that in the training set, the true values for those nodes in the middle are not observed.

Alternatively, we can use $a^{[0]}$, which stands for the activation, to represent the input layer $X$. Then, $a^{[1]}$ can be used to represent the hidden layer, where

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}.$$

Finally, the output layer could be denoted by $a^{[2]}$.

The figure 3.2 is called 2 layer Neural Network. The reason is that when we count layers in neural networks, we do not count the input layer, so hidden layer is layer 1 and output layer is layer 2. Input layer is referred to as layer 0. For the hidden layer, there are parameters $w^{[1]}, b^{[1]}$ associated with it, where $w^{[1]} \in \mathbb{R}^{4\times 3}$ and $b^{[1]} \in \mathbb{R}^{4\times 1}$. Similarly, the output layer has parameters $w^{[2]}, b^{[2]}$ associated with it, where $w^{[2]} \in \mathbb{R}^{1\times 4}$ and $b^{[2]} \in \mathbb{R}^{1\times 1}$.

## 3.3 Computing a Neural Network's Output

Each node in the hidden layer involves two-step computation:

- $z_j^{[i]} = w_j^{[i]T} x + b_j^{[i]}$, where $i$ represent the index of layer and $j$ represent the node index in that layer.
- $a_j^{[i]} = \sigma(z_j^{[i]})$.

If we refer to 3.2, then we can represent the hidden units with the following

equations:

$$
\begin{aligned}
z_1^{[1]} &= w_1^{[1]T}x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]}) \\
z_2^{[1]} &= w_2^{[1]T}x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]}) \\
z_3^{[1]} &= w_3^{[1]T}x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]}) \\
z_4^{[1]} &= w_4^{[1]T}x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})
\end{aligned}
$$

Using for-loop to compute these equations would be pretty inefficient. Therefore, we implement vectorization technique.

As each hidden unit has a corresponding parameter vector $w$, we can stack these parameter vectors together to form a matrix. Multiplying the resulting matrix with vector $x$ and adding the whole multiplication result with vector $b$ would give us the following:

$$
z^{[1]} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}.
$$

The matrix that contains the parameter vector could be referred to as $W^{[1]}$ and the bias vector could be referred to as $b^{[1]}$ to brevity, so $z^{[1]} = W^{[1]}x + b^{[1]}$.

Similarly, we could use vectorization to compute $a^{[1]}$ as follows:

$$
a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]}).
$$

In summary, given input $x$, or $a^{[0]}$, we perform the following computations:

- $z^{[1]} = W^{[1]}x + b^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$, where $z^{[1]}$ has shape $(4, 1)$, $W^{[1]}$ has shape $(4, 3)$, $a^{[0]}$ has shape $(3, 1)$, and $b^{[1]}$ has shape $(4, 1)$.
- $a^{[1]} = \sigma(z^{[1]})$, where $a^{[1]}$ has shape $(4, 1)$ and $z^{[1]}$ has shape $(4, 1)$.
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$, where $z^{[2]}$ has shape $(1, 1)$, $W^{[2]}$ has shape $(1, 4)$, $a^{[1]}$ has shape $(4, 1)$ and $b^{[2]}$ has shape $(1, 1)$.
- $\hat{y} = a^{[2]} = \sigma(z^{[12]})$, where both $a^{[2]}$ and $z^{[2]}$ have shape $(1, 1)$.

## 3.4 Vectorizing Across Multiple Examples

Previously, we have only computed the value of $\hat{y}$ for one single input $x$. For $m$ training examples, we would need to run a for-loop to compute $a^{[2](i)}$, where $(i)$ represents the $i$th training example and [2] represents layer 2, for each and every training example. This is achieved as follows:

We would like to vectorize the whole computation to get rid of the for-loop.

**Algorithm 3** Forward Propagation with $m$ Training Examples

**for** $i = 1$ to $m$ **do**
$z^{[1](i)} = W^{[1]}a^{[0](i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$
**end for**

Recall that we can stack training examples together for form a matrix as follows:

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix},$$

where $X$ has shape $(n_x, m)$.

It turns out we need to do the following computations with Python broadcasting technique:

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} = W^{[1]}A^{[0]} + b^{[1]} \\ A^{[1]} &= \sigma(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(Z^{[2]}) \end{aligned}$$

Similar to how matrix $X$ was formed, we can stack individual $z^{[1](i)}$'s and $a^{[1](i)}$ horizontally to form matrix $Z^{[1]}$ and $A^{[1]}$ as follows:

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix}, \; A^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ | & | & & | \end{bmatrix}.$$

Note that horizontal indices correspond to different training examples and vertical indices correspond to the activation of different hidden units in the neural network.

## 3.5 Activation Functions

In previous illustrations, we use **sigmoid function**

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

as an activation function, where $\sigma(z) \in (0, 1)$.

The sigmoid function can be graphed as follows:

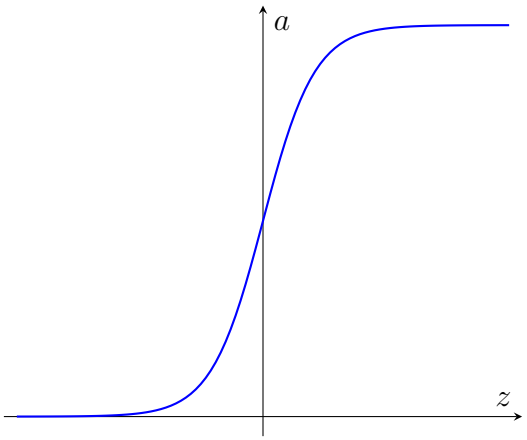



Figure 3.3: Sigmoid Function

More generally, we can use a different activation function $g(z)$ in place of $\sigma(z)$, where $g(z)$ could be a nonlinear function.

Similar to the sigmoid function, we can use **tanh, or hyperbolic tangent**, function

$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

where $a \in (-1, 1)$.

The tanh function can be graphed as follows:

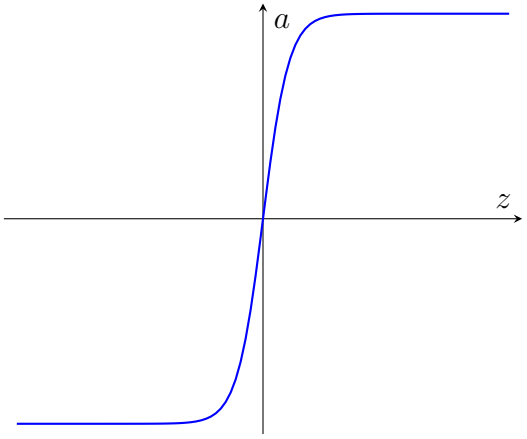



Figure 3.4: Hyperbolic Tangent Function

It turns out that for hidden units, if we let $g(z) = \tanh(z)$, this almost always works better than the sigmoid function. With the values between $+1$ and $-1$, the mean of the activations that come out of the hidden layer are closer to zero. The tanh function is almost always superior than the sigmoid function. One exception is for the output layer because if $y$ is either zero or one, then it makes sense for $\hat{y}$ to have

$$0 \leq \hat{y} \leq 1.$$

One of the downsides of both tanh and sigmoid functions is that if $z$ is either very large or very small, then the gradient of the function ends up being close to zero, so this can slow down gradient descent. One other choice of activation function, which

is very popular in machine learning, is the rectified linear unit (ReLU) function

$$a = \max\{0, z\}.$$

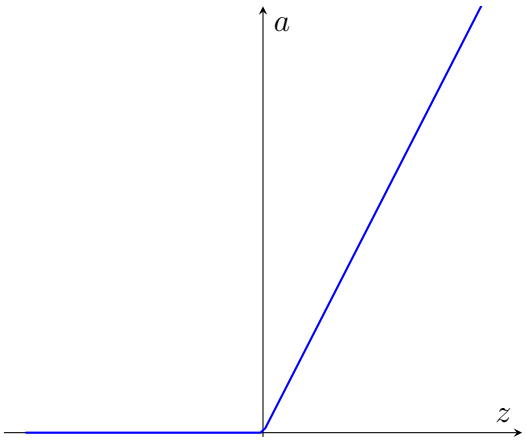



Figure 3.5: ReLU Function

In this case, the derivative is 1 so long as $z$ is positive and the derivative is 0 when $z$ is negative. Technically, when $z = 0$, the derivative is not well defined. However, when we implement this in the computer, the odds that we get exactly $z = 0.00000000000$ is very small. In practice, we can pretend the derivative, when $z = 0$, is 1 or 0 and the code would work fine.

Rule of thumb for choosing the activation function:

- If the output is zero-one value, when doing binary classification, then the sigmoid activation function is a natural choice for the output layer.

- For other outputs, ReLU (Rectified Linear Unit) is increasingly the default choice of activation function, though sometimes people also use the tanh activation function.

One disadvantage of ReLU is that the derivative is equal to zero when $z < 0$. In practice, this works just fine. However, there is another version of ReLU called the **Leaky ReLU** , which usually works better than the ReLU activation function, but it's not used as much in practice.

The formula is given by

$$a = \max\{0.01 \times z, z\}$$

and it's graphed as follows:

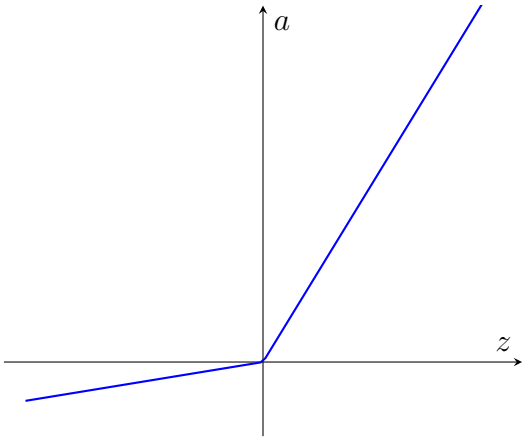



Figure 3.6: Leaky ReLU Function

The advantage for both ReLU and Leaky ReLU is that for a lot of the space of $z$, the derivative of the activation function is very different from zero. In practice, by using the ReLU activation function, the neural network will often learn much faster than when using the tanh or the sigmoid activation function. The main reason is that there is less of the effect of the derivative of the function going to zero, which slows down learning.

Even though half of $z$ would have derivative equal to $z$ for ReLU, in practice, enough of hidden units will have $z > 0$, so learning can still be quite fast for most training examples.

Although ReLU is widely used and often times the default choice, it is very difficult to know in advance exactly which activation function will work the best for the idiosyncrasies problems. When unsure about which activation function works the best, try them all and evaluate on a holdout validation set or a development set. Then, see which one works better and then go with that activation function.

## 3.6 Why Non-Linear Activation Functions?

It turns out that for neural network to compute interesting functions, we do need to pick a non-linear activation function.

Suppose given $x$, we use linear activation function, or in this case, identity activation function, as follows:

- $z^{[1]} = W^{[1]}x + b^{[1]}, a^{[1]} = z^{[1]}$
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, a^{[2]} = z^{[2]}$

If we do this, then the model is just computing $\hat{y}$ as a linear function of the input feature $x$. We would have

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

In this case, the neural network is just outputting a linear function of the input. For deep neural networks, if we use a linear activation function, or, alternatively, if we do not have an activation function, then no matter how many layers the neural network has, all it's doing is just computing a linear activation function, so we might as well not have any hidden layers.

Suppose we have the following neural network:

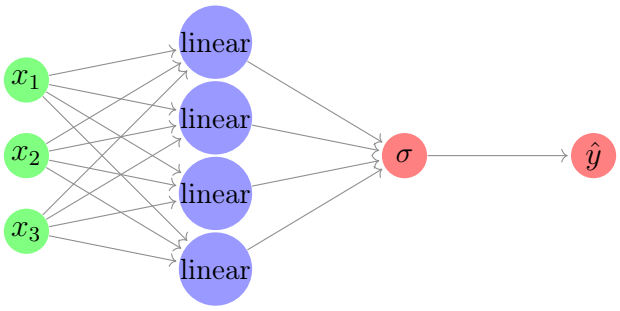



Figure 3.7: Neural Network with Linear Layers

This model is no more expressive than standard logistic regression without any hidden layer. A linear hidden layer is more or less useless because the composition of two linear functions is itself a linear function. Unless we throw a non-linearity, we are not computing more interesting functions even as we go deeper in the network.

There is one player where we might use a linear activation function $g(z) = z$. That's when we are doing machine learning on the regression problem. For example, if we are trying to predict housing prices, where a price $y \in \mathbb{R}$. It might be okay to use a linear activation for the output layer so that $\hat{y}$ is also a real number. However, the hidden units should not use linear activation functions. Other than this, using a linear activation function in the hidden layer, except for some very special circumstances relating to compression, is extremely rare.

## 3.7 Derivatives of Activation Functions

**Sigmoid activation function** is given by

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

We can compute its derivative as follows:

$$\begin{aligned} g'(z) = \frac{\partial}{\partial z} g(z) &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \\ &= \frac{1+e^{-z}-1}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \\ &= \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right) \cdot \frac{1}{1+e^{-z}} \\ &= (1 - \frac{1}{1+e^{-z}}) \cdot \frac{1}{1+e^{-z}} \\ &= g(z) \cdot (1 - g(z)). \end{aligned}$$

Let's sanity check that this expression makes sense. When $z$ is very large. Say, $z = 10$. Then, $g(z) \approx 1$ and

$$\frac{\partial}{\partial z} g(z) \approx 1 \cdot (1 - 1) = 0.$$

Conversely, when $z = -10, g(z) \approx 0$. From the derivative formula above, we have

$$\frac{\partial}{\partial z} g(z) \approx 0 \cdot (1 - 0) = 0.$$

When $z = 0, g(z) = 0.5$. In this case,

$$\frac{\partial}{\partial z} g(z) \approx \frac{1}{2} \cdot (1 - \frac{1}{2}) = \frac{1}{4}.$$

In the neural network setting, we use $a$ to denote $g(z)$. Therefore, sometimes we see

$$g'(z) = a(1 - a).$$

If we have the value of $a$, then we can quickly compute the value for $g'(z)$. **tanh activation function** is given by

$$g(z) = \tanh(z).$$

We can compute the derivative as follows:

$$\begin{aligned} \frac{\partial}{\partial} \tanh(z) &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\ &= 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\ &= 1 - \tanh^2(z). \end{aligned}$$

Similar to sigmoid function, we may use $a$ to denote $g(z)$. In this case,

$$g'(z) = 1 - a^2.$$

**ReLU activation function** is given by

$$g(z) = \max\{0, z\}.$$

The derivative is given by

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$$

Technically, if we define $g'(z) = 1$ when $z = 0$, then $g'(z)$ becomes a sub-gradient of the activation function $g(z)$, which is why gradient descent still works. Therefore, in practice, we use the following derivative:

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

**Leaky ReLU activation function** is given by

$$g(z) = \max\{0.01z, z\}.$$

The derivative, in practice, can be expressed as follows:

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

## 3.8 Gradient Descent for Neural Networks

In this section, we show how to implement gradient descent for neural network with one hidden layer.

The neural network with a single hidden layer has parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$, where we have $n^{[0]}$ input features, $n^{[1]}$ hidden units, and $n^{[2]} = 1$ output units. Therefore, $W^{[1]}$ has shape $(n^{[1]}, n^{[0]})$, $b^{[1]}$ has shape $(n^{[1]}, 1)$, $W^{[2]}$ has shape $(n^{[2]}, n^{[1]})$, and $b^{[2]}$ has shape $(n^{[2]}, 1)$.

Suppose we are doing binary classification, so the cost function is as follows:

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

where $\hat{y} = a^{[2]}$.

In this case, our loss function is given by

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

The gradient descent algorithm for neural networks could be given as follows:

**Algorithm 4** Gradient Descent for Neural Networks

Initialize parameters **randomly** rather than to all zeros.
**repeat**
  Compute predictions $\hat{y}^{(i)}$ for $i = 1, \ldots, m$.
  Compute $dW^{[1]} = \frac{\partial J}{\partial db^{[1]}}, dW^{[1]} = \frac{\partial J}{\partial db^{[1]}}, dW^{[2]} = \frac{\partial J}{\partial dW^{[2]}}, db^{[2]} = \frac{\partial J}{\partial db^{[2]}}$.
  Update $W^{[1]} = W^{[1]} - \alpha dW^{[1]}$
  Update $b^{[1]} = b^{[1]} - \alpha db^{[1]}$
  Update $W^{[2]} = W^{[2]} - \alpha dW^{[2]}$
  Update $b^{[2]} = b^{[2]} - \alpha db^{[2]}$
**until** the cost function $J$ converges

In summary, the following are the equations for forward propagation (in vectorized form), assuming we are doing binary classification:

$$\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) = \sigma(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}X + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})
\end{aligned}$$

In backward propagation step, the derivatives are given as follows:

$$\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\
db^{[2]} &= \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} \times g^{[1]'}(Z^{[1]}) \text{ element-wise} \\
dW^{[2]} &= \frac{1}{m} dZ^{[1]} X^T \\
db^{[1]} &= \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)
\end{aligned}$$

Recall the 2 layer neural network we had:

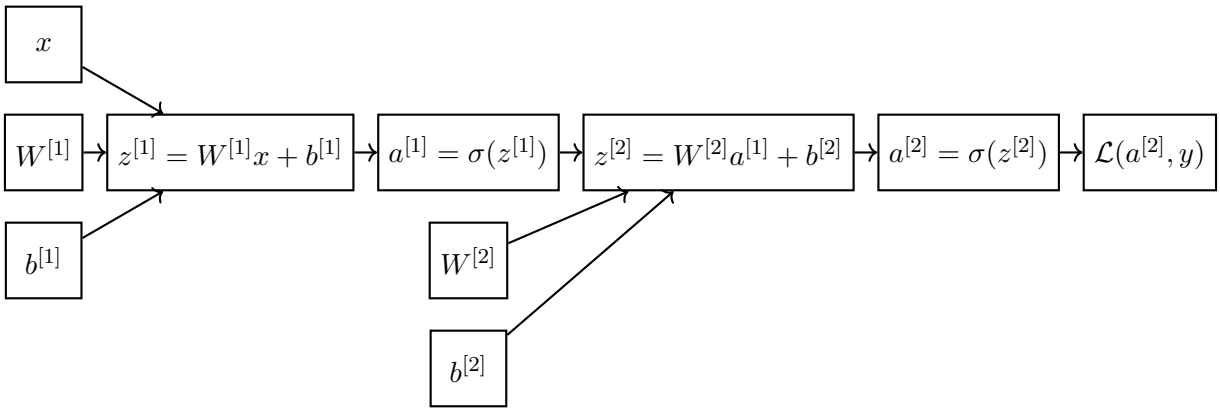



Figure 3.8: 2 Layer Neural Network

Now, we can compute derivatives as follows:

$$\begin{aligned}
dZ^{[2]} &= a^{[2]} - y \\
dW^{[2]} &= dZ^{[2]}a^{[1]T} \\
db^{[2]} &= dZ^{[2]} \\
dZ^{[1]} &= W^{[2]T}dZ^{[2]} \times g^{[1]'}(Z^{[1]}) \text{ element-wise} \\
dW^{[1]} &= dZ^{[1]}x^T \\
db^{[1]} &= dZ^{[1]}
\end{aligned}$$

The vectorized form can be represented as follows:

$$\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]})
\end{aligned}$$

where

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix}$$

The vectorized implementation is as follows:

$$\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m}dZ^{[2]}A^{[1]T} \\
db^{[2]} &= \frac{1}{m}np.sum(dZ^{[2]}, axis = 1, keepdims = True) \\
dZ^{[1]} &= W^{[2]T}dZ^{[2]} \times g^{[1]'}(Z^{[1]}) \text{ element-wise} \\
dW^{[1]} &= \frac{1}{m}dZ^{[1]}X^T \\
db^{[1]} &= \frac{1}{m}np.sum(dZ^{[1]}, axis = 1, keepdims = True)
\end{aligned}$$

## 3.9 Random Initialization

For logistic regression, it was okay to initialize the weights to zero. However, for a neural network, if we initialize the weights to parameters to all zero and apply gradient descent, it will not work.

Suppose we have the following shallow neural network:

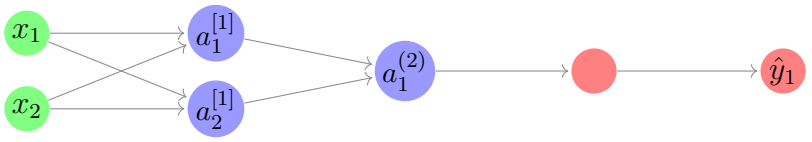



Figure 3.9: Shallow Neural Network Example

In this case, we have two input features and so $n^{[0]} = 2$. Also, we have two hidden units in the first hidden layer, so $n^{[1]} = 2$. Thus, $W^{[1]} \in \mathbb{R}^{2\times 2}$ and $b^{[1]} \in \mathbb{R}^{2\times 1}$. Initializing $b^{[1]}$ to all zeros is okay, but not for $W^{[1]}$.

Suppose we have

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, W^{[2]} = \begin{bmatrix} 0 & 0 \end{bmatrix},$$

then $a_1^{[1]} = a_2^{[1]}$ and $dZ_1^{[1]} = dZ_2^{[1]}$. If two neurons in the same layer are equal, then they are going to have the same rate of change and update from the gradient. We can proof by induction to show that after $t$ iterations, two hidden units would be computing exactly the same function, which means there is no point to have more than one hidden unit. With the same changes after each iteration, the two neurons will act as if they are the same, thus the additional neuron does not add more information. We call this error the **symmetry breaking problem**.

The solution to this is to initialize parameters randomly as follows:

$$W^{[1]} = np.random.randn((2,2)) * 0.01.$$

We multiply the random generated number by 0.01 to make sure that the weights are initialized to very small random values. If we are using hyperbolic tangent or sigmoid function, then if the weights are too large, $Z^{[1]} = W^{[1]}X + b^{[1]}$ would be very large and thus we are more likely to end up at flat parts of the activation function, where the gradient is very small. In this case, gradient descent will be very slow, thus the learning. On the other hand, if we do not have sigmoid or tanh activation functions throughout the neural network, then this is less of an issue.

Sometimes, there can be better constants than 0.01. When training a shallow neural network, say it has only one hidden layer, set constant to 0.01 would probably work okay. But when we are training on a very deep neural network, then we might to choose a different constant, which will be discussed in later sections.

It turns out that our bias term $b^{[1]}$ does not have the symmetry breaking problem, so it is okay to initialize $b^{[1]}$ to just zeros as follows:

$$b^{[1]} = np.zeros((2,1)).$$

We can initialize $W^{[2]}, b^{[2]}$ in a similar way.

Note that so long as $W^{[1]}$ is initialized randomly, we start off with different hidden units computing different functions and thus we would not have the symmetry breaking problem.

# Chapter 4

# Deep Neural Networks

The objectives of this chapter are

- Describe the successive block structure of a deep neural network
- Build a deep L-layer neural network
- Analyze matrix and vector dimensions to check neural network implementations
- Use a cache to pass information from forward to back propagation
- Explain the role of hyperparameters in deep learning
- Build a 2-layer neural network

## 4.1 Deep L-layer Neural Network

We say that logistic regression is a "shallow" model, where is does not have any hidden layers.

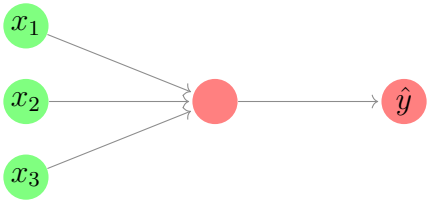



Figure 4.1: Logistic Regression

On the other hand, a neural network, shown below, with 5 hidden layers is a much deeper model.

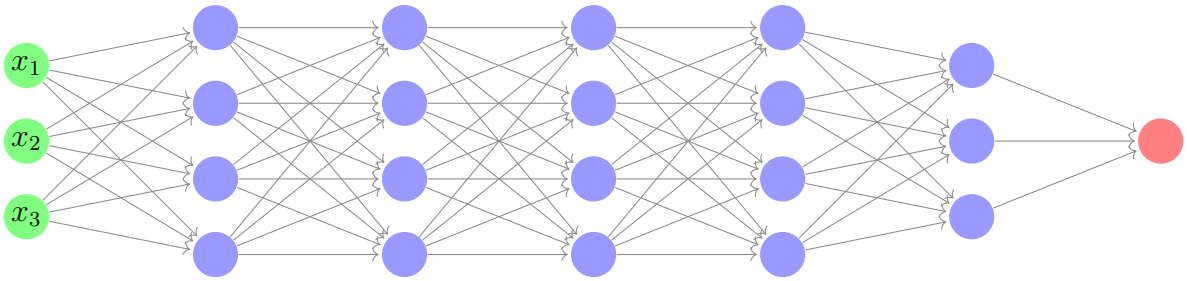



Figure 4.2: Neural Network with 5 Hidden Layers

A neural network with 1 hidden layer is called a "2 layer Neural Network."

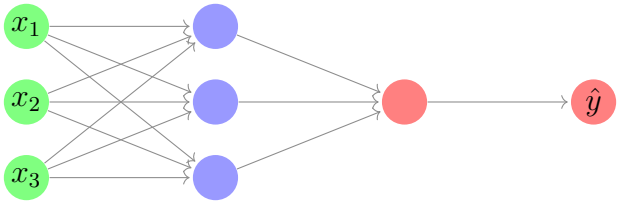



Figure 4.3: Neural Network with 1 Hidden Layers

To clarify the notations about deep neural network, let's use the following 4 layer neural network as an example:

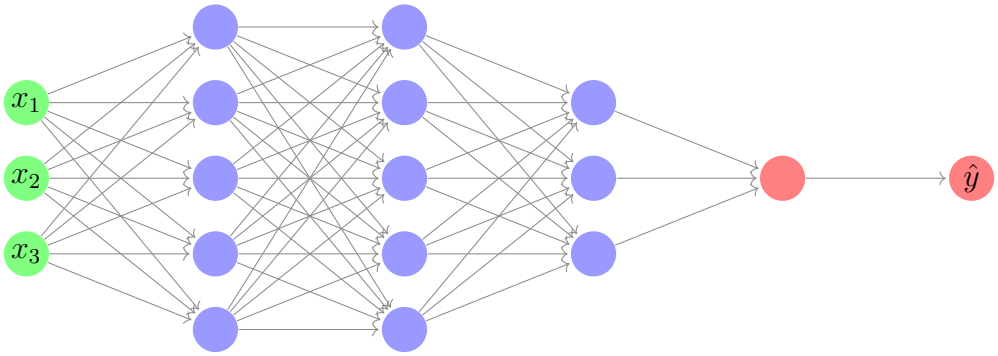



Figure 4.4: 4 Layer Neural Network

There are 3 hidden layers. The number of units in these hidden layers are $5, 5, 3$ and there is 1 output unit.

- We use capital $L$ to denote the number of layers. In this case, $L = 4$.
- We use $n^{[l]}$ to denote the number of nodes/units in layer $l$. For example, $n^{[0]} = n_x = 3, n^{[1]} = n^{[2]} = 5, n^{[3]} = 3, n^{[4]} = n^{[L]} = 1$.
- For each layer $l$, we use $a^{[l]}$ to denote the activations in layer $l$. For example, $a^{[l]} = g^{[l]}(z^{[l]})$. In any case, $x = a^{[0]}$ and $\hat{y} = a^{[L]}$.
- We use $w^{[l]}$ to denote the weights for $z^{[l]}$.
- We use $b^{[l]}$ to denote the biases for $z^{[l]}$.

## 4.2 Forward Propagation in a Deep Network

Given a training example $x$, we can compute the activations of the first layer as follows:

$$z^{[1]} = w^{[1]}x + b^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$

$$z^{[3]} = w^{[3]}a^{[2]} + b^{[3]}$$
$$a^{[3]} = g^{[3]}(z^{[3]})$$

$$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$$
$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$$

The general rule is that $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$ and $a^{[l]} = g^{[l]}(z^{[l]})$.

The vectorized form would be the following:

$$Z^{[1]} = W^{[1]}X + b^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$

$$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$$
$$A^{[3]} = g^{[3]}(Z^{[3]})$$

$$\hat{Y} = g^{[4]}(Z^{[4]}) = A^{[4]}$$

In this implementation of vectorization, there is a for-loop where we compute values of $l = 1, \ldots, 4$. In this case, there is no way to implement without an explicit for-loop. Therefore, it's okay to have a for-loop for forward propagation.

## 4.3 Getting Matrix Dimensions Right

One way to do this is to pull a piece of paper and work through the dimensions of matrices we are working with.

Suppose we have the following neural network:

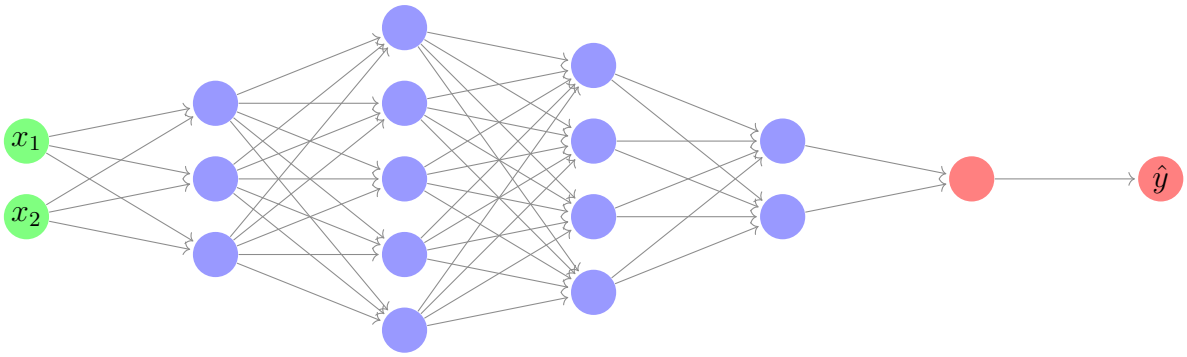



Figure 4.5: 5-Layer Neural Network

For this neural network, we have

$$n^{[0]} = 2, n^{[1]} = 3, n^{[2]} = 5, n^{[3]} = 4, n^{[4]} = 2, n^{[5]} = 1.$$

When implementing forward propagation, we have

$$z^{[1]} = w^{[1]}x + b^{[1]},$$

which is the activations for the first hidden layer. Therefore, the dimensions of $z^{[1]}$ is $(n^{[1]}, 1)$, or $(3, 1)$. $x$ has dimensions $(2, 1)$, or more generally $(n^{[0]}, 1)$. By the rule of matrix multiplication, we can infer that $W^{[1]}$ has dimensions $(3, 2)$, or more generally, $(n^{[1]}, n^{[0]})$. $b^{[1]}$ has dimension $(3, 1)$, or more generally, $(n^{[1]}, 1)$.

Similarly, $w^{[2]}$ has dimensions $(5, 3)$. $z^{[2]}$ has dimensions $(5, 1)$. $a^{[1]}$ has dimensions $(3, 1)$. $w^{[3]}$ has dimensions $(4, 5)$. $w^{[4]}$ has dimensions $(2, 4)$. $w^{[5]}$ has dimensions $(1, 2)$.

Since $a^{[l]}$ is the obtained from element-wise operation of $g^{[l]}$ on $z^{[l]}$, $a^{[l]}$ and $z^{[l]}$ should have the same dimensions.

When implementing backward propagation, $dw^{[l]}$ should have the same dimensions as $w^{[l]}$ and $db^{[l]}$ should have the same dimensions as $b^{[l]}$.

The general formula to check dimensions is given as follows:

$$\begin{aligned} w^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ b^{[l]} &: (n^{[l]}, 1) \\ dw^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ db^{[l]} &: (n^{[l]}, 1) \end{aligned}$$

For vectorized implementation, the dimensions are a little different. For the forward propagation, we have

$$Z^{[1]} = W^{[1]}X + b^{[1]}.$$

In this case, $Z^{[1]}$ has dimensions $(n^{[1]}, m)$. The dimensions of $w^{[1]}$ stays the same, which are $(n^{[1]}, n^{[0]})$. Now, $X$ has dimensions $n^{[0]}, m)$. $b^{[1]}$ is still $(n^{[1]}, 1)$, but broadcasting gives us $(n^{[1]}, m)$. We can summarize dimensions as follows:

$$Z^{[l]}, A^{[l]}, dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$$

## 4.4 Why Deep Representation?

What is a deep neural network computing? Suppose we are building a system for face recognition, or face detection. Then, the first layer of neural network could be an edge detector for the input picture. It can then group edges together to form parts of a face in the next hidden layer. Finally, by putting together different parts of a face like an eye, an ear, or a nose, the next layer, it can then try to recognize or detect different types of faces.

This type of simple to complex hierarchical representation, or compositional representation, applies to other types of data than images and face recognition. For example, if we are trying to build a speech recognition system, it's hard to visualize speech. However, if we input an audio clip, then the first layer of a neural network might learn to detect low level audio wave form features, such as if the tone is going up. Then, by composing low level wave forms, the neural network might learn to detect basic units of sounds, which is called phonemes in Linguistics. By composing these together, it learns words. Then, neural network learns to recognize sentences or phrases.

Whereas early layers of a deep neural network compute relatively simple functions of the input, such as where the edge is, by the time we get deep into the network, we can do surprisingly complex things.

**Circuit theory** pertains thinking about what types of functions we can compute with different AND gates, OR gates, NOT gates, and other logic gates. Informally, there are functions we can compute with a "small" (small number of hidden units) L-layer deep neural network that shallower networks require exponentially more hidden units to compute. For example, if we want to compute $y = x_1 \, XOR \, x_2 \, XOR \, \ldots, XOR \, x_n$. It would take $\mathcal{O}(\log(n))$ layers to build the XOR tree, each layer with small number of hidden units. However, if we restrict the number of layers to 1, then we would need $\mathcal{O}(2^n)$ hidden units, which is exponentially large.

## 4.5 Building Blocks of Deep Neural Networks

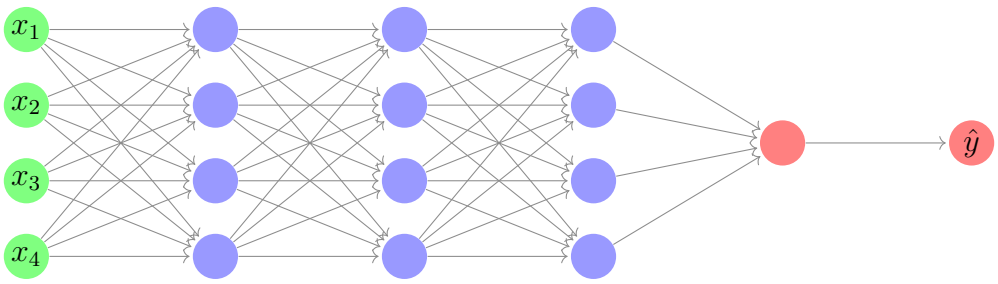



Figure 4.6: 4-Layer Neural Network Example

Suppose we have the above neural network. Let's look into the computations focusing on just the layer $l$. For layer $l$, we have $w^{[l]}, b^{[l]}$.

For the forward propagation, it has input $a^{[l-1]}$ and outputs $a^{[l]}$ as follows:

$$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

Meanwhile, we cache the result $z^{[l]}$, and potentially $w^{[l]}, b^{[l]}$, as it would be useful for backward propagation step later.

Then, for the backward propagation, we have input $da^{[l]}, cache(z^{[l]})$ and want output $da^{[l-1]}, dw^{[l]}, db^{[l]}$.

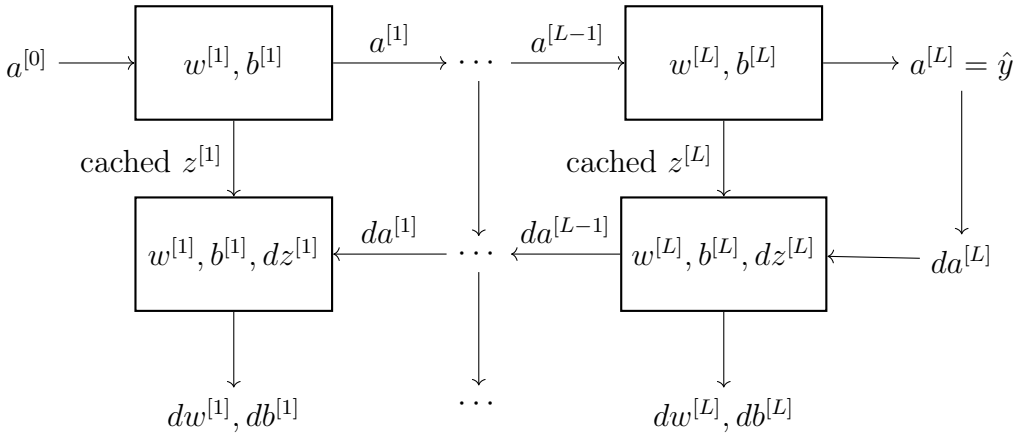



Figure 4.7: One Full Iteration of Gradient Descent for Neural Network

## 4.6 Forward and Backward Propagation

In this section, we discuss the implementation of forward and backward propagation.

The vectorized implementation of forward propagation is given below:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

where we initialize $A^{[0]} = X$.

For backward propagation, given input $da^{[l]}$, we output

$$da^{[l-1]}, dW^{[l]}, db^{[l]}.$$

The implementation is given as follows:

$$dz^{[l]} = da^{[l]} \times g^{[l]'}(z^{[l]}) \text{ element-wise}$$
$$dW^{[l]} = dz^{[l]}a^{[l-1]T}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = W^{[l]T}dz^{[l]}$$

If we take the definition of $da$ and plug into the definition of $dz$, we get

$$dz^{[l]} = W^{[l+1]T}dz^{[l+1]} \times g^{[l]'}(z^{[l]}),$$

which is consistent with previous presentation.

The vectorized version is given below:

$$\begin{aligned} dZ^{[l]} &= dA^{[l]} \times g^{[l]'}(Z^{[l]}) \text{ element-wise} \\ dW^{[l]} &= \frac{1}{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True) \\ dA^{[l-1]} &= W^{[l]T} dZ^{[l]} \end{aligned}$$

In the context of binary classification, we can derive that

$$da^{[l]} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} = \frac{-y}{a} + \frac{(1-y)}{(1-a)},$$

which is used for initialization of backward recursion.
In the vectorized implementation, we initialize as follows:

$$dA^{[l]} = \left( \frac{-y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{(1-a^{(1)})} \right) + \cdots + \left( \frac{-y^{(m)}}{a^{(m)}} + \frac{(1-y^{(m)})}{(1-a^{(m)})} \right)$$

## 4.7 Parameters vs Hyperparameters

Parameters in the learning algorithm are weights $W^{[l]}$'s and biases $b^{[l]}$'s. Hyperparameters determine the real parameters and they are

- learning rate $\alpha$
- number of iterations
- number of hidden layers $L$
- number of hidden units $n^{[1]}, n^{[2]}, \ldots$
- Choice of activation function

In later sections, we will see our hyperparameters in deep learning, such as momentum term, the mini-batch size, various forms of regularization parameters, and so on.

Applied deep learning is a very empirical process where we might have an idea about the learning rate $\alpha = 0.01$. Then, we implement it and see how it works. Then, we adjust the learning rate accordingly. We need to try out many things and see which work.

Also, as we make progress on a problem, it is quite possible that the best values for hyperparameters might change – this might be caused by the computer infrastructure. The rule of thumb is that for every few months when we are working on a problem for an extended period of time, just try a few values of hyperparameters and check if there are better values for them.

## 4.8 What does this have to do with the brain?

People often make analogy between deep learning and the human brain. When doing neural network, we implement forward and backward propagation. A single biological neurons in the brain receives electric signals from other neurons, does a simple thresholding computation, and then if this neuron fires, it sends a pulse of electricity down the axon to other neurons. Therefore, there is a loose analogy between a single neuron in a neural network and biological neuron.

However, even today, the neuroscientist have almost no idea what even a single neuron is doing. A single neuron appears to be more complex than we are able to characterize with neuroscience. It is completely unclear today whether the human brain uses an algorithm does anything like backward propagation or gradient descent, or there are some fundamentally different learning principle that the human brain uses.

# Part II

# Improving Deep Neural Network: Hyperparameter Tuning, Regularization, and Optimization

# Chapter 5

# Practical Aspects of Deep Learning

The objectives of this chapter are

- Give examples of how different types of initializations can lead to different results
- Examine the importance of initialization in complex neural networks
- Explain the difference between train/dev/test sets
- Diagnose the bias and variance issues in your model
- Assess the right time and place for using regularization methods such as dropout or L2 regularization
- Explain Vanishing and Exploding gradients and how to deal with them
- Use gradient checking to verify the accuracy of your backpropagation implementation
- Apply zeros initialization, random initialization, and He initialization
- Apply regularization to a deep learning model

## 5.1 Setting up Machine Learning Application

Making good choices of setting up training, development, and test sets can make a huge difference in helping us quickly find a good high-performance neural network. When starting off on a new problem, it is almost impossible to correctly guess the right values for hyperparameter choices. In practice, applied machine learning is a highly iterative process We often start with an idea. Then we code it up and run an experiment to get back a result that tells how well one particular network works. Based off the outcome, we then refine ideas and change the choices. We keep iterating this process and find a better and better neural network.

### 5.1.1 Train/Dev/Test Sets

Setting up dataset well, in terms of train, development, and test sets, can make us more efficient at going around the iterative process.

For a given training data, traditionally, we will carve off some portion of it to be the training set, some to be hold-out cross validation, or development, set, and some final portion to be the test set. We keep on training our algorithm on training set and use hold-out cross validation set to see which model performs the best on the “dev" set. After having done this long enough, when we have a final model that we want to evaluate, we can take the best model and evaluate it on the test set in order to get an unbiased estimate of how well our algorithm is doing.

In the previous era of machine learning, it was a common practice to take all data and split it according to **70%/30%** train-test split if we do not have explicit “dev" set. If we have “dev" set, then people do a **60%/20%/20%** split. Several years ago, this was widely considered the best practice in machine learning.

However, in the modern big data era, where we might have a million examples in total, the trend is that our “dev" and test sets become a much smaller percentage of the total. The reason is that the goal of the dev set, or development set, is that we are testing different algorithms on it and see which ones works better, so the dev set just has to be big enough for us to evaluate different algorithm choices and quickly decide which ones are better. For example, if we have a million training examples, we might decide to have 10K examples in the dev set, which is more than enough. In a similar vein, the main goal of the test set is, given the final algorithm, to give us a pretty confident estimate of how well it is doing. Again if we have a million training examples, we might decide to have 10K examples to evaluate the algorithm. In this case, we are splitting with **98%/1%/1%** ratios. If we have more than a million examples, then we might end up having **99.5%/0.25%/0.25%**, or **99.5%/0.4%/0.1%** split.

Furthermore, more and more people are training on **mismatched train and test** distributions. Suppose we are building an app that lets users upload a lot of pictures and the goal is to find pictures of cats in order to show the users. The training set could come from cat pictures downloaded off the Internet, but our dev and test sets might comprise of cat pictures from users using the app. It turns out that a lot of webpages have very high resolution, very professional, nicely framed pictures of cats. But the users may be uploading blurrier, lower resolution images taken with a cell phone camera in a more casual condition. Therefore, the distributions of data may be different. The rule of thumb in this case is to make sure the **dev and test sets come from the same distribution**.

Finally, it might be okay to not have a test set if we don’t need an unbiased estimate to evaluate our final model. In this case, we go over the iterative process by training on the training set, trying different model architectures, and evaluating models on the dev set.

## 5.2 Bias/Variance

Bias and Variance is one of those concepts that’s easily learned but difficult to master. In the Deep Learning era, there has been less discussion of Bias/Variance trade-off.

Suppose we have a dataset with two labels of 2D data points. If we fit a line to the data with high bias, then we say this is underfitting the data. On the other hand, if we fit a very complex classifier, then we have a high variance, in which case we overfit the data. With the medium level of model complexity, then we are fitting "just right."

With high dimensional data, we cannot visualize the decision boundary. Instead, we will look at different metrics to try to understand bias and variance. For the cat classification problem, we will look at **train set error** and **dev set error**.

Let's say the train set error is 1% and dev set error is 11%. In this case, we are doing pretty well on the training set, but relatively poorly on the development set. We might have overfitted the training set, so the model does not generalize well to the hold-out cross validation set. This has **high variance**.

Suppose, instead, the train set error is 15% and dev set error is 16%. Assume humans can achieve roughly 0% error, then it looks like the algorithm is not even doing very well on the training set. In this case, we are underfitting the data and thus it has **high bias**.

Now, suppose the train set error is 15% and dev set error is 30%. Then the algorithm has **high bias and high variance**. This could happen because we choose a very simple model, yet overfit a few training examples. On the other hand, suppose train set error is 0.5% and dev set error is 1%. This model has **low bias and low variance**. These categorizations all rely on the assumption that the optimal error, sometimes called **Bayes Error**, is nearly 0%. We would have had a different analysis if we have a much higher Bayes Error.

The trade-off is summarized in the figure below:

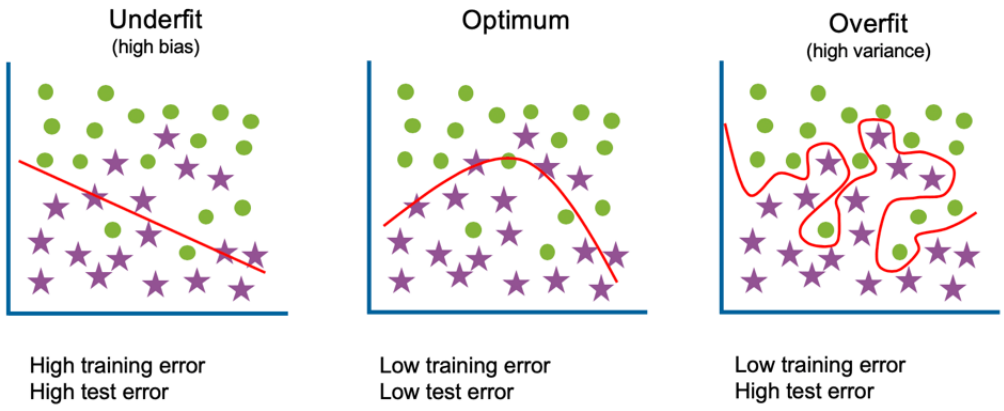



Figure 5.1: Bias/Variance Trade-off

## 5.3 Basic Recipe for Machine Learning

In this section, we present the ways to improve algorithm more systematically, depending on whether it has high bias or high variance issues.

After training the algorithm, we check the following:

1. **If the algorithm have high bias** by looking at the training set performance. If it does have high bias – not fitting training set well – we can try

- bigger network, such as more hidden layers or more hidden units.
- train algorithm longer.
- try some more advanced optimization algorithms.
- try a different neural network architecture that is better suited for the problem.

2. Once we have reduced bias to an acceptable value, we ask **if we have variance problem** by looking at dev set performance – are we able to generalize from a pretty good training set performance to having a pretty good dev set performance? If we have a high variance problem, then we can

   - get more data.
   - regularization.
   - try a more appropriate neural network architecture.

3. Then we go back to check **if we have high bias again** and hopefully we find something with both low bias and low variance.

In pre deep learning era, we reducing bias would often lead to an increased variance, or vice versa. However, with deep learning technique, we can almost always reducing one without hurting the other. It almost never hurts to try a deeper neural network, as long as we regularize well.

## 5.4 Regularizing Neural Network

### 5.4.1 Regularization

If we have a high variance problem, then one of the first things to try is regularization. The other way to get rid of high variance is to get more training data, which is also quite reliable. However, we cannot always get more training data, as it could be expensive. Adding regularization will often help prevent overfitting, or reduce variance in the network.

Let's develop the idea using logistic regression. Recall that for logistic regression, the cost function is given by

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

where $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$.

With regularization term, we use the following cost function instead:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2,$$

where $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$. This is called **L2 regularization**, which is the most common type of regularization. In practice, we could also add a regularization term for $b$, but it is usually omitted. $w$ is usually pretty high dimensional parameter vector, so most likely high variance means we are not fitting these parameters well,

whereas $b$ is just a single number. Almost all the parameters are in $w$, rather than in $b$. Therefore, adding a regularization for $b$ will not make much difference.

We also have **L1 regularization** given by

$$\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w| = \frac{\lambda}{2m}\|w\|_1.$$

If we use L1 regularization, then $w$ will end up being sparse, meaning the $w$ vector will have a lot of zeros in it. Some people say this can help compressing the model – as a set of parameters are zero, we need less memory to store the model, although, in practice, this helps only a little bit.

In either case, $\lambda$ is called the **regularization parameter** and usually we set this using the development set by trying a variety of values and see which one does the best.

For the neural network, the cost function is given by

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2,$$

where $\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(w_{i,j}^{[l]})^2$. This matrix norm is called the **Frobenius norm**, or $L2$ norm, of the matrix. Previously, to implement gradient descent, we would compute $dw^{[l]}$ by backprop and we update $w^{[l]}$ with $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$. Now, with the regularization term, we add

$$\frac{\lambda}{m}w^{[l]}$$

to the gradient.

In this case, we multiply $w^{[l]}$ by $1 - \frac{\alpha\lambda}{m}$ and then subtract learning rate times the original gradient to do the update. Since we multiply $w^{[l]}$ by a number smaller than 1 every time we update the parameter, L2 norm is also called “weight decay."

### 5.4.2 Why Regularization Reduces Overfitting?

If we set $\lambda$ to be really big, then we will incentivize the model to set the weight matrix to be reasonably close to zero, meaning $w^{[l]} \approx 0$. This zeroes out impacts of many hidden units. Then, the simplified neural network becomes a much smaller neural network. It gets closer and closer to as if we are just using logistic regression. This moves high variance problem towards high bias issue, but hopefully there will be an intermediate value of $\lambda$ that results in the “just right" case.

Here is another way to gain intuition. Suppose we use tanh activation function, meaning $g(z) = \tanh(z)$. If $\lambda$ is really large, then the set of parameters in $w^{[l]}$ will be quite small, resulting in relatively small $z^{[l]}$ because $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$. This puts $g(z)$ to the linear regime of the tanh function., where $g(z)$ is roughly linear. Therefore, every layer would be roughly linear, as if it is just linear regression. We showed before that if every hidden layers are linear, then the whole network is just a linear network. As a result, the deep neural network cannot fit very complicated non-linear decision boundaries that overfit the dataset.

### 5.4.3 Dropout Regularization

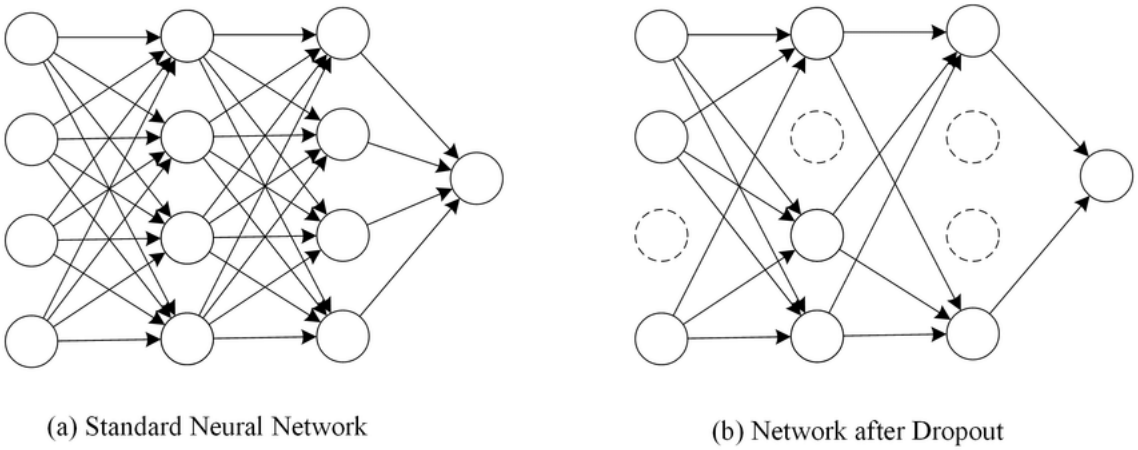



Figure 5.2: Neural Network with Dropout

With dropout, we go through each layer of the network and set some probability of eliminating a node in the neural network. Suppose for each layer, we have 50% chance of keeping each node and 50% chance of removing each node. After eliminating nodes, we remove all the outgoing edges from that node, so we end up with a much diminished network. Then, we do backprop, training one example on this much diminished network. On different examples, we toss a set of coins again and keep a different set of nodes and then dropout different set of nodes. For each training example, we train it using one of the reduced networks. As we train individual examples on much smaller networks, this gives a reason why we end up being able to regularize the network.

Now, we discuss a implementation of dropout, called **inverted dropout technique**. We illustrate with a single layer, layer $l = 3$, where we use $d3$ to represent the dropout vector for layer 3 and keep_prob to represent the probability of keeping a node.

$$
\begin{aligned}
d3 &= np.random.rand(a3.shape[0], a3.shape[1]) < kepp\_prob \\
a3 &= np.multiply(a3, d3) \\
a &= a/keep\_prob
\end{aligned}
$$

Suppose $keep\_prob = 0.8$ and we have 50 hidden units in the third hidden layer. Now, we have 80% chance of keeping the node and 20% chance of removing the node for each individual node. On average, we have 10 units shut off, so $a^{[3]}$ is expected to be reduced by 20%. In order to maintain the expected value of $z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$, we need to divide $a^{[3]}$ by 0.8. This ensures that no matter what we set keep_prob to, the expected value of $a^{[3]}$ remains the same, and thus $z^{[4]}$. It turns out that at test time, the inverted dropout technique makes test time easier because we have less of a scaling problem.

At test time, we are given some $a^{[0]} = x$ and we want to make a prediction. We do not use dropout at test time, as we do not want the output to be random. If we implement dropout at test time, this just adds noise to the predictions.

### 5.4.4 Understanding Dropout

Many of the first successful implementations of dropouts were in computer vision, where input sizes are really big. We almost always don't have enough data, so we are almost always overfitting. Dropout randomly knocks out units in the network, so it's as if on every iteration, we are working with a smaller neural network. Using smaller neural network seems like it should have a regularizing effect. In this section, we provide another intuition.

With dropout, some input units are eliminated, so a hidden unit cannot rely on any one feature because any one feature could go away at random. Consequently, the unit will be more motivated to spread out weights and give a little bit of weight to each of its inputs. By spreading out the weights, this will tend to have an effect of shrinking the squared norm of the weights. Similar to L2 regularization, it helps prevent overfitting. It actually turns out that dropout can formally be shown to be an adaptive form of L2 regularization, but L2 penalty on different weights are different depending on the size of the activation being multiplied.

It turns out we can use different values of keep_prob for different layers. To reduce overfitting of the matrix with largest parameter size, we can have a low keep_prob for that layer with large dimension of parameters. We can also drop out some of the inputs, though, in practice, we usually don't do that, so $keep\_prob = 1.0$ is quite common for the input layer.

Actually, unless the algorithm is overfitting, we would not bother to use dropout. One big downside of dropout is that the cost function $J$ is no longer well-defined. Therefore, it's hard to plot the value of cost function against the number of iterations, so we lose a debugging tool to find bugs. In this case, we can turn off dropout by setting keep_prob to 1 and run the code to make sure that the cost value is monotonically decreasing. Then, we turn on the dropout and hope that we did not introduce bugs in the code.

### 5.4.5 Other Regularization Methods

Suppose we are fitting a cat classifier. If we are overfitting, more training data can help, but getting more training data can be expensive. We can augment our data by, for example, flipping it horizontally and adding it to the training set. However, since the training set is now a bit redundant, this isn't as good as if we had collected an additional set of new independent examples. Also, we can random crops, distortions, or translations of an image. This is called **data augmentation** and this is an inexpensive way to give our algorithm more data and reduce overfitting. For optical character recognition, we can also augment dataset by imposing random rotations and distortions to a given digit.

One other technique that is often used is **early stopping**. As we run gradient descent, we plot training error, or cost $J$, against the number of iterations, which should decrease monotonically. We also plot dev set error. We usually find that dev set error decreases for a while and then it will increase. In this case, we will stop training on the neural network when dev set error starts to increase. Why does this work? When we just start the gradient descent iteration, the parameters $w$ will be close to zero. As we iterate, $w$ will get bigger than bigger. By stopping halfway, we only have a mid-side $\|w\|_F^2$. Similar to L2 regularization, by picking a neural network with smaller norm for parameters $w$, hopefully the neural network

is overfitting less.

Machine learning process is comprised of several different steps.

- We want the algorithm to find a set of parameters $w, b$ to optimize cost function $J$. We use tools like gradient descent, Adam optimizer, etc. to do it.
- Not overfit by using regularization, getting more data, etc..

In machine learning, we already have many hyperparameters to tune. Therefore, it is easier to think about when we have one set of tools for optimizing cost function $J$, where all we care about is finding $w, b$ such that $J(w, b)$ is as small as possible. Then, reduce overfit is another completely separate task. This principle is called **orthogonalization**. The idea is that we want to think about only one task at a time. This introduces the downside of early stopping as it couples two tasks together. We can no longer work on two problems independently. By stopping early, we are breaking the optimization of cost function $J$ and, simultaneously, trying to not overfit. In this case, rather than using early stopping, we can just use L2 regularization to train as long as possible. The downside of this, though, is that we need to try a lot of values of hyperparameters, which makes it more computationally expensive.

## 5.5 Setting Up Optimization Problem

### 5.5.1 Normalizing Inputs

By normalizing inputs, we can speed up the training for neural network. We can do so by subtracting the mean as follows:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
$$x = x - \mu$$

The second step is to normalize the variances as follows:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} * *2$$
$$x = x/\sigma$$

If we use this to scale our training data, we use the same $\mu, \sigma$ to normalize the test set.

Why do we bother normalizing inputs? Recall that the cost function is defined as

$$J(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

It turns out if we use unnormalized input features, it is more likely that the cost function would look like a squished out bar. If the features are in very different scales, then parameters will take very different values, causing cost function to be elongated. Therefore, we would need a small learning rate for the cost function to

oscillate back and forth until it reaches the minimum. If we normalize the features, then the cost function will be more symmetric. In this case, no matter where we start, gradient descent can take much larger steps and go straight to the minimum. The comparison could be summarized by the following figure:

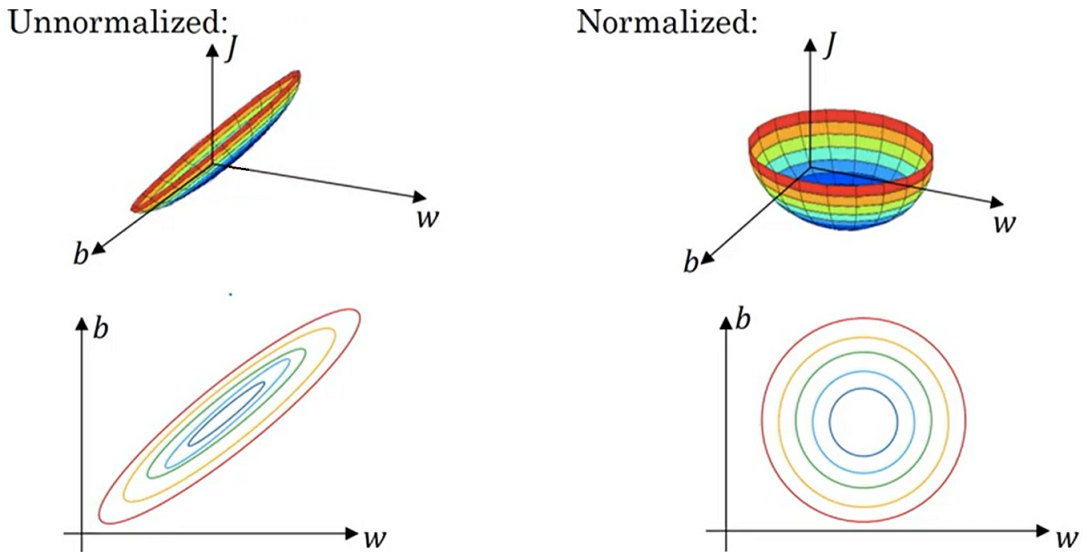



Figure 5.3: Normalizing Inputs

### 5.5.2 Vanishing/Exploding Gradients

One big problem of training neural networks, especially deep neural networks, is data vanishing and exploding gradients. The derivatives can sometimes get either very big or very small, which makes training difficult.

Suppose we are training a very deep neural network with parameters $w^{[1]}, \ldots, w^{[l]}$, $b^{[l]} = 0$, and activation function $g(z) = z$ for the sake of simplicity. Now, the output would be

$$\hat{y} = \left( \prod_{l=1}^{L} w^{[l]} \right) x.$$

Suppose each feature matrix $w^{[l]}$ is

$$w^{[l]} = \begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix}.$$

Then, $\hat{y}$ can be rewritten as

$$\hat{y} = w^{[L]} \begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix}^{L-1} x.$$

Effectively, $\hat{y} = \gamma^{L-1} x$. If $\gamma > 1$ and $L$ is very large for deep network, then $\hat{y}$ will be very large. In fact, this grows exponentially. Conversely, if $\gamma < 1$, then activation values will decrease exponentially as a function of the number of layers $L$ of the network. A similar argument could be used to show that gradients will also increase or decrease exponentially as a function of the number of layers.

With some of the modern neural networks, $L = 150$. Microsoft got great results with 152 layer neural network. With such a deep neural network, if the activations or gradients increase or decrease exponentially as a function of $L$, then the values could be really big or really small, which makes training difficult.

### 5.5.3 Weight Initialization for Deep Networks

A partial solution to the vanishing/exploding gradients would be a better or more careful choice of random initialization for the neural network. Suppose we have the following single neuron network:

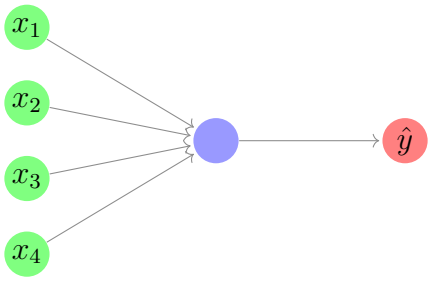



Figure 5.4: Single Neuron Example

In this case, by setting $b = 0$, we have

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n.$$

The larger the $n$ is, the smaller we want $w_i$ to be. One reasonable thing to do is to set the variance of $w$ to be

$$Var(w_l) = \frac{1}{n^{[l-1]}},$$

where $n$ is the number of input features that is going into the neuron.
In practice, we can do the following:

$$w^{[l]} = np.random.randn(shape) * np.sqrt(1/n^{[l-1]}).$$

It turns out if the activation function is ReLU, then setting variance of

$$Var(w_l) = \frac{2}{n^{[l-1]}}$$

works a little bit better.

By doing this, the values $w^{[l]}$ are not too much bigger or smaller than 1, so it does not explode or vanish too quickly. For other variants, if we use tanh activation function, then we use

$$Var(w_l) = \frac{1}{n^{[l-1]}}.$$

This is called **Xavier initialization**. Another version is given by

$$Var(w_l) = \frac{2}{n^{[l-1]} + n^{[l]}}.$$

If we want, this variance could also be a hyperparameter that we try to tune, but this is less important relative to other hyperparameters.

### 5.5.4 Numetical Approximation of Gradients

When implementing backprop, there is a technique call **gradient checking** that helps make sure that the implementation is correct. To build up to gradient checking, we will first discuss how to numerically approximate computations of gradients.

Recall that the formall definition of derivative is given by

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}.$$

For $f(x) = x^3$ and $x = \theta$. We nudge $\theta$ to both left and right to $\theta - \epsilon$ and $\theta + \epsilon$ for some small number $\epsilon > 0$. Therefore,

$$g(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

gives a much better approximation to the derivative at $\theta$ than using $\theta + \epsilon$ alone. We can show that the approximation error is on the order of $\mathcal{O}(\epsilon^2)$, where the Big-O constant happens to be 1. When $\theta = 1, \epsilon = 0.01$, our approximation is

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001,$$

and the real derivative is $g(\theta) = 3\theta^2 = 3$. The approximation error is thus 0.0001, whereas using $\theta + \epsilon$ alone gives us 0.301 approximation error. If we use one-sided approximation, then the approximation error is on the order of $\mathcal{O}(\epsilon)$. Since $\epsilon$ is a small number, the one-sided approximation is less accurate than the two-sided approximation.

However, it turns out that in practice, the two-sided approximation runs twice as slow as the one-sided approximation, but it's worth it to use two-sided method, as it's more accurate.

### 5.5.5 Gradient Checking

Gradient Checking (Grad Check) is a technique that helps find bugs in the implementations of backprop and verify that the implementation is correct.

Suppose our neural network has parameters $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$. The first thing we need to do for gradient checking is to take all these parameters and reshape into a giant vector $\theta$. Now the cost function becomes a function of $\theta$:

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta) = J(\theta_1, \theta_2, \ldots).$$

Similarly, we take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape them into a giant vector $d\theta$, with the same dimension as $\theta$. Is $d\theta$ the gradient of $J$?

### 5.5.6 Gradient Checking Implementation Notes

In this section, we cover some practical tips about how to implement grad check correctly to debug our code. The tips are as follows.

- Firstly, we do not use grad check in training. It's only used to debug. Computing $d\theta_{approx}[i]$ is a very slow computation.

**Algorithm 5** Gradient Checking

**for** each i **do**

Compute

$$d\theta_{approx}[i] = \frac{J(\theta_1, \ldots, \theta_i + \epsilon, \ldots) - J(\theta_1, \ldots, \theta_i - \epsilon, \ldots)}{2\epsilon}.$$

Compute $d\theta[i] = \frac{\partial J}{\partial \theta_i}$.

**end for**

Then, we check if $d\theta_{approx}$ and $d\theta$ are approximately equal. We could check Euclidean distance and then normalize it as follows:

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}.$$

In practice, we use $\epsilon = 10^{-7}$. If the formula gives a value of $10^{-7}$ or smaller, then the derivative approximation is likely to be correct. If it's $10^{-5}$, maybe it's okay, but we need to take a careful look. If the formula gives value of $10^{-3}$, then most likely there is a bug. We should look at the vector to find if there is a specific value of $i$, which $d\theta_{approx}[i]$ is very different from $d\theta[i]$.

- Secondly, if an algorithm fails grad check, we look at individual components to try to identify the bug. For example, if $db^{[l]}$'s are pretty far off, but $dW^{[l]}$'s are quite close, then the bug might be the way we compute $db$, and vice versa.
- We need to remember the regularization term when computing the gradient.
- Grad check does not work with dropout. There isn't an easy-to-compute cost function $J$ that dropout is doing gradient descent on. Therefore, we need to implement grad check without dropout, or we can fix the pattern of nodes dropped and use grad check to check grads for that pattern are correct, though in practice we do not usually do this.
- Maybe our implementation of backprop is only correct when $w, b$ are close to 0 and it gets more inaccurate when $w, b$ become larger. Though not done often in practice, we can run grad check at random initialization and then run grad check again after some number of iterations.

# Chapter 6

# Optimization Algorithms

The objectives of this chapter are

- Apply optimization methods such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Use random minibatches to accelerate convergence and improve optimization
- Describe the benefits of learning rate decay and apply it to your optimization

In this chapter, we present optimization algorithms that would enable us to train neural networks much faster. Deep learning tends to work the best in the regime of big data, when we train our neural network on a huge dataset. However, this is just slow. Having a good optimization algorithm can spped up the efficiency of training.

## 6.1 Mini-batch Gradient Descent

Previously, we have shown that vectorization allows us to efficiently compute on m training examples without an explicit for-loop. However, if m is very large – say, 5 million – then the process can still be slow.

With the implementation of gradient descent, we need to process the entire training set before we take one little step of gradient descent. We can split training sets into little baby training sets, which are called **mini-batches**, each having 1000 training examples. We can denote the set of training examples $x^{(1)}, \ldots, x^{(1000)}$ as $X^{\{1\}}$ and all the way until $X^{\{5000\}}$. Similarly, we split $Y$ accordingly into $Y^{\{1\}}, \ldots, Y^{\{5000\}}$. Therefore, mini-batch $t$ is comprised of $X^{\{t\}}, Y^{\{t\}}$, where $X^{\{t\}}$ has dimensions $(n_x, 1000)$ and $Y^{\{t\}}$ has dimensions $(1, 1000)$.

Batch gradient descent processes the entire batch of training examples all at the same time, whereas mini-batch gradient descent processes a single mini-batch $X^{\{t\}}, Y^{\{t\}}$ at the same time.

When we have a large training set, mini-batch gradient descent runs much faster than batch gradient descent. Pretty much every one in deep learning uses it.

## 6.2 Understanding Mini-batch Gradient Descent

In this section, we show more details of how to implement mini-batch gradient descent and gain a better understanding of what it's doing and why it works.

**Algorithm 6** Mini-batch Gradient Descent

**for** each i **do**

Perform forwardprop on $X^{\{t\}}$ for $l = 1, \ldots, L$:

$$Z^{[l]} = W^{[l]}X^{\{t\}} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]}$$

Compute cost $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l}\mathcal{L}(\hat{y}^{(i)}, y^{(i)} + \frac{\lambda}{2\cdot 1000}\sum_{l}\|w^{[l]}\|_F^2$.
Backprop to compute gradients w.r.t $J^{\{t\}}$.
Update weights with $W^{[l]} := W^{[l]} - \alpha dW^{[l]}, b^{[l]} := b^{[l]} - \alpha db^{[l]}$

**end for**

With batch gradient descent, we expect the cost to go down as we train for more iterations. On mini-batch gradient descent, if cost might not decrease on every additional iteration, but it shows a decreasing trend overtime. The reason why mini-batch cost values are more noisy is because $X^{\{t\}}, y^{\{t\}}$ might be a easy mini-batch while $X^{\{t+1\}}, y^{\{t+1\}}$ is a harder mini-batch. If there are mislabeled examples inside, then the cost will be higher. The comparison is shown in the figure below:

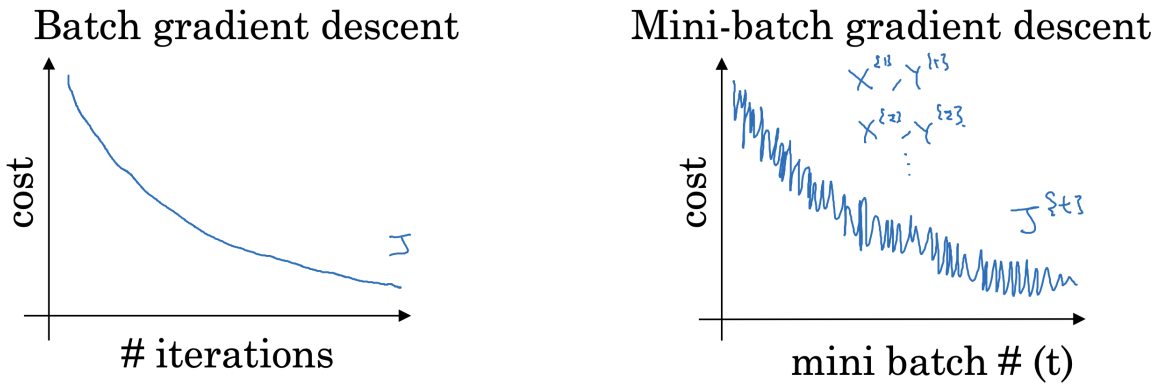



Figure 6.1: Batch vs Mini-batch Gradient Descent

As we use mini-batch gradient descent, we would need to choose mini-batch size. If mini-batch size is m, then we are doing batch gradient descent where

$$(X^{\{t\}}, y^{\{t\}} = (X, Y).$$

On the other hand, if we choose mini-batch size to be 1, then we are doing **stochatic gradient descent** where every training example is its own mini-batch, where

$$(X^{\{t\}}, y^{\{t\}} = (x^{(1)}, y^{(1)}), \ldots, (x^{(n_x)}, y^{(n_x)})$$

In practice, we use mini-batch size in between 1 and m. If we use batch gradient descent, where mini-batch size is m, then we are processing huge training set at every iteration, so it takes a long time to run each iterations. On the other hand, if we use stochastic gradient descent, even though the noise can be reduced by using smaller learning rate, we lose all the speed-ups from vectorization. By choosing a size in between 1 and m, we have the fastest learning, where we do get a lot of vectorization and make progress without processing the entire training set.

The convergence is shown below, where purple represents stochastic gradient descent, blue represents batch gradient descent, and green represents mini-batch size in between 1 and m:

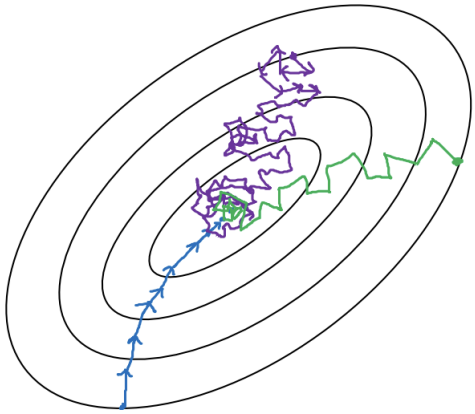


Figure 6.2: Convergence Comparison among Different Batch Sizes

Here is the general guidelines of choosing the mini-batch size.

- If we have small training set, then just use batch gradient descent. By small, it means $m \leq 2000$.
- Typical mini-batch size would be from 64 to 512. Because of the way computer memory is laid out and accessed, sometimes the code runs faster if the mini-batch size is a power of 2. Therefore, we choose mini-batch size from $\{64, 128, 256, 512\}$.
- Make sure mini-batch $X^{\{t\}}, y^{\{t\}}$ fit into CPU/GPU memory.

At the end, the mini-batch size is also a hyperparameter we can tune. We can run gradient descent with mini-batch size of some power of 2 and see which one does the best job.

## 6.3 Exponentially Weighted Averages

In order to understand optimization algorithms that run faster than gradient descent, we need to use **exponentially weighted averages**, or exponentially weighted moving averages in Statistics.

For example, temperature data could be pretty noisy. If we want to compute the trends, or the moving average of the temperature, we can do the following:

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= 0.9v_0 + 0.1\theta_1 \\
v_2 &= 0.9v_1 + 0.1\theta_2 \\
v_3 &= 0.9v_2 + 0.1\theta_3 \\
&\vdots \\
v_t &= 0.9v_{t-1} + 0.1\theta_t
\end{aligned}
$$

where $\theta_i$ represents the temperature on day $i$. This is the exponentially weighted average of the daily temperature. More generally, we can represent the moving average as

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t.$$

## 6.4 Understanding Exponentially Weighted Averages

Suppose $t = 100$ and $\beta = 0.9$. By expanding iterative equations $v_{100} = 0.9v_{99} + 0.1\theta_{100}$, we get the following:

$$\begin{aligned} v_{100} &= 0.1 + 0.9v_{99} \\ &= 0.1 + 0.9(0.1 + 0.9v_{98}) \\ &= 0.1 + 0.9(0.1 + 0.9(0.1 + 0.9v_{97})) \\ &= 0.1\theta_{100} + 0.1(0.9)\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + 0.1(0.9)^4\theta_{96} + \cdots \end{aligned}$$

All the coefficients add up to 1 or close to 1, up to a detail called **bias correction**. In this example,

$$0.9^{10} \approx 0.35 \approx \frac{1}{e}.$$

One fact we know is that

$$(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}.$$

Therefore, it takes 10 days for the weight to decay to less than about a third of the weight of the current day. If $\beta = 0.98$, then

$$0.98^{50} \approx \frac{1}{e},$$

as $\epsilon = 0.02$.

In general, we are averaging over

$$\frac{1}{1-\beta}$$

days, where $\epsilon = 1 - \beta$.

We can implement this as follows:

$v_\theta = 0$
*repeat*{
    get next $\theta_t$
    $v_\theta := \beta v_\theta + (1-\beta)\theta_t$
    }

This takes very little memory since we keep on overwriting one variable with the formula. Also, this just takes one line of code, which is very efficient, especially when we try to compute averages for multiple variables.

## 6.5 Bias Correction in Exponentially Weighted Averages

Bias correction can make our computation of exponentially weighted averages more accurate.

Suppose $\beta = 0.98$. Then, from the formula we showed,

$$v_1 = 0.98v_0 + 0.02\theta_1 = 0.02\theta_1.$$

Similarly,

$$\begin{aligned} v_2 &= 0.98v_1 + 0.02\theta_2 \\ &= 0.98(0.02)\theta_1 + 0.02\theta_2 \\ &= 0.0196\theta_1 + 0.02\theta_2. \end{aligned}$$

Neither $v_1$ nor $v_2$ would be a good estimate of the temperature. The way to modify the estimate to make it much more accurate, especially during the initial phase of the estimate, we can instead take

$$\frac{v_t}{1-\beta^t}.$$

For example, suppose $t = 2$, then

$$1 - \beta^t = 1 - (0.98)^2 = 0.0396.$$

Therefore, our estimate of the temperature on day 2 becomes

$$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

and this becomes a weighted average of $\theta_1$ and $\theta_2$ and removes the bias.

Also, as $t$ becomes large, $\beta^t$ approaches 0 and thus the bias correction makes almost no difference. When $t$ is small, when we are warming up the estimations, the bias correction could help us make a better estimate of the temperature.

## 6.6 Gradient Descent with Momentum

Gradient descent with momentum almost always works faster than the standard gradient descent algorithm. The basic idea is that we compute an exponentially weighted average of gradients, and then use that gradient to update weights instead.

Suppose the contour plot of the cost function is given as follows:

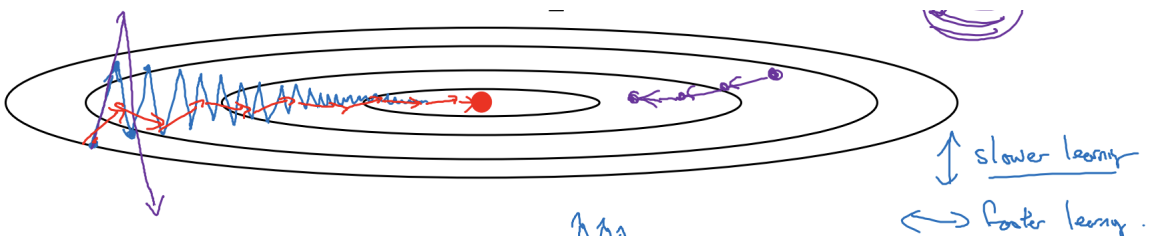



Figure 6.3: Gradient Descent with Momentum Contour Plot

For the standard gradient descent, shown in blue line, gradient descent takes a lot of steps and slowly oscillates towards the minimum. This up and down oscillations slow down gradient descent and prevent us from using a much larger learning rate. If we use a much larger learning rate, then we could overshoot like the purple line in 6.3.

On the vertical axis, we want the learning to be a bit slower because we do not want to overshoot oscillations. However, on the horizontal axis, we want faster learning because we want to aggressively move from left to right towards the red dot.

Gradient descent with momentum smooths out the steps of gradient descent. For gradients, the oscillations on the vertical direction tend to average out to close to zero. This will average out positive and negative numbers and so the average will be close to zero. On the horizontal direction, all the directions are pointing to the right, so the average in the horizontal direction will still be pretty big. Therefore, gradient descent with momentum will take steps with much smaller oscillations in the vertical direction and more directed to moving quickly in the horizontal direction, as shown by the red line in 6.3. The algorithm is given below:

**Algorithm 7** Gradient with Momentum

$v_{dW} = 0, v_{db} = 0$.
**for** each iteration t **do**
Compute $dW, db$ on the current mini-batch (omitted superscripts).
Compute $v_{dW} := \beta v_{dW} + (1 - \beta)dw$.
Compute $v_{db} := \beta v_{db} + (1 - \beta)db$.
Update parameters as

$$W := W - \alpha v_{dW}$$
$$b := b - \alpha v_{db}$$

**end for**

Imagine we are optimizing a bowl-shaped function. Then, the gradient terms are providing acceleration to a ball rolling down the hill. The momentum terms represent the velocity. Therefore, we can accelerate down the bowl and gain momentum.

For this algorithm, we have hyperparameters $\alpha$, learning rate, and $\beta$, which controls the exponentially weighted average. In practice, $\beta = 0.9$, which averages last 10 gradients, works very well. It appears to be a pretty robust value. Also, in practice, people do not do bias correction because after just 10 iterations, the moving average will have warmed up and is no longer a bias estimate.

In some literature, we see that $v_{dW}, v_{db}$ are represented as

$$v_{dW} := \beta v_{dW} + dw$$
$$v_{db} := \beta v_{db} + db$$

where $(1 - \beta)$ is omitted. The net effect is that $v_{dW}$ and $v_{db}$ end up being scaled by a factor of $1 - \beta$. Therefore, when we perform the gradient descent updates, $\alpha$ has to be changed by a corresponding value of

$$\frac{1}{1 - \beta}.$$

this requires us to tune the learning rate $\alpha$ differently.

## 6.7 RMSprop

RMSprop (root mean squared prop) can similarly speed up gradient descent. In this section, we show how it works.

The algorithm is presented as follows:

**Algorithm 8** RMSprop

$v_{dW} = 0, v_{db} = 0$.
**for** each iteration t **do**
Compute $dW, db$ on the current mini-batch (omitted superscripts).
Compute $S_{dW} := \beta_2 v_{dW} + (1 - \beta_2) dw^2$ (element-wise).
Compute $S_{db} := \beta_2 v_{db} + (1 - \beta_2) db^2$ (element-wise).
Update parameters as

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \epsilon}$$
$$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$$

**end for**

Small $\epsilon > 0$ is added to ensure numerical stability in case $S_{dW}$ or $S_{db}$ equals to zero. $\epsilon = 10^{-8}$ is a good default. With these terms, we hope $S_{dW}$ will be relatively small and $S_{db}$ will be relatively large, so we can slow down the updates on the vertical direction and speed up the updates on the horizontal direction. As $db$ will be relatively large and $dW$ will be relatively small, dividing $db$ by $\sqrt{S_{db}}$ will damp out the oscillations. The net effect can be shown in the figure below, with the green line:

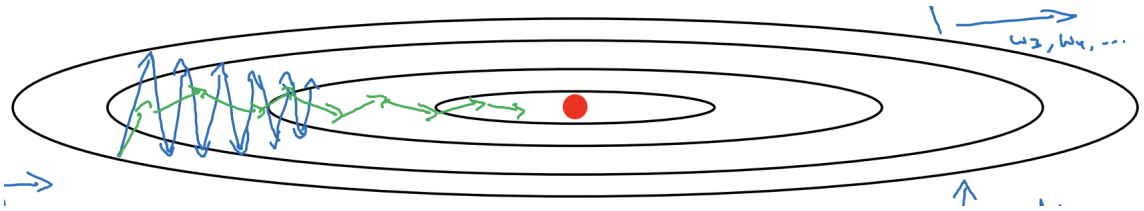


Figure 6.4: RMSprop Convergence

In this case, we can use larger $\alpha$ without diverging in the vertical direction. In practice, the vertical/horizontal direction could be sets of parameters in high-dimensional parameter vector.

This idea was not first proposed in a academic paper, but in a Coursera course by Geoffrey E. Hinton.

## 6.8 Adam Optimization Algorithm

RMSprop and Adam optimizers have shown to work well on a wide range of deep learning architectures. Adam optimizer basically takes momentum and RMSprop

and puts them together.

The algorithm is presented as follows:

**Algorithm 9** Adam Optimization Algorithm

$v_{dW} := 0, S_{dW} := 0, v_{db} := 0, S_{db} := 0.$

**for** each iteration t **do**

Compute $dW, db$ on the current mini-batch (omitted superscripts).

Compute $v_{dW} := \beta_1 v_{dW} + (1-\beta_1)dW, v_{db} := \beta_1 v_{db} + (1-\beta_1)db$

Compute $S_{dW} := \beta_2 v_{dW} + (1-\beta_2)dw^2; S_{db} := \beta_2 v_{db} + (1-\beta_2)db^2$ (element-wise).

Bias correction on $v$ as $v_{dW}^{corrected} := v_{dW}/(1-\beta_1^t); v_{db}^{corrected} := v_{db}/(1-\beta_1^t)$

Bias correction on S as $S_{dW}^{corrected} := S_{dW}/(1-\beta_1^t); S_{db}^{corrected} := S_{db}/(1-\beta_1^t)$

Update parameters as

$$W := W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$$

$$b := b - \alpha \frac{v_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

**end for**

This is a common used algorithm that's proven to be very effective for many neural network architectures. In this algorithm, there are many hyperparameters.

- Learning rate $\alpha$. This ususally needs to be tuned.
- Moving average of $dW$, $\beta_1$, has a default value 0.9.
- $\beta_2$ has a recommended value of 0.999 by the author of Adam algorithm paper.
- The choice of $\epsilon$ does not matter much, but the author of the Adam paper recommends $10^{-8}$.

Adam stands for **Adaptive moment estimation**, where $\beta_1$ is computing the mean of the derivatives – the first moment – and $\beta_2$ is computing exponentially weighted average of the squares – the second moment.

## 6.9 Learning Rate Decay

If we slowly decrease the learning rate $\alpha$, then we can wander around a tight region close to the optimal point. In the initial steps of learning, we can afford to take much bigger steps, but as learning approaches convergence, then having a smaller learning rate allows us to take smaller steps.

Recall that one epoch is one pass through the dataset. We can set learning rate to

$$\alpha = \frac{1}{1 + \text{decayRate} \times \text{epochNumber}} \alpha_0,$$

where $\alpha_0$ is some initial learning rate. Here, the decayRate becomes another hyperparameter we might need to tune.

Suppose $\alpha_0 = 0.2$ and $decayRate = 1$. We can construct the following example to better illustrate the idea:

| Epoch | $\alpha$ |
|---|---|
| 1 | 0.1 |
| 2 | 0.067 |
| 3 | 0.05 |
| 4 | 0.04 |

There are other learning rate decay methods.

- Exponentially decay
$$\alpha = \beta^{\text{epochNumber}}\alpha_0$$
where $0 \leq \beta < 1$.
- $$\alpha = \frac{k}{\sqrt{\text{epochNumber}}}\alpha_0 \quad or \quad \alpha = \frac{k}{\sqrt{t}}\alpha_0$$
where $k$ is some constant and $t$ is the mini-batch number.
- Discrete staircase decrease of learning rate $\alpha$ where we decay after some number of steps.
- Manual decay. If we are training one model and the model is taking hours or days to train, we can tune $\alpha$ hour-by-hour or day-by-day. This only works when we are training a small number of models.

## 6.10 The Problem of Local Optima

In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima. It turns out if we train a neural network, most points of zero gradients are not local optima, but saddle points. In 20000 dimensional space, for a point to be a local minimum, we would need to have gradient concaving down in all directions, which has a very small chance. It's more likely the case where one direction is curving down and another is curving up, giving us a saddle point. It is very unlikely to get stuck in a bad local optima, so long as we train on a reasonably large neural network. This tells us a lot of intuitions we have for low-dimensional spaces really don't transfer to high-dimensional spaces, where our learning algorithms are operating over.

It turns out **plateaus** can really slow down learning. A plateau is a region where the derivatives is close to zero for a long time. As the plateau is nearly flat, it takes a very long time to slowly find our way off the plateau, as shown below:

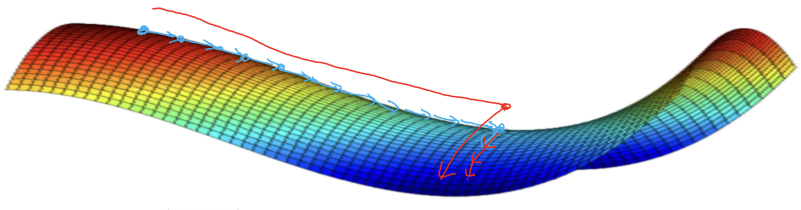


Figure 6.5: Plateau Example

# Chapter 7

# Hyperparameter Tuning and More

The objectives of this chapter are

- Master the process of hyperparameter tuning
- Describe softmax classification for multiple classes
- Apply batch normalization to make your neural network more robust
- Build a neural network in TensorFlow and train it on a TensorFlow dataset
- Describe the purpose and operation of GradientTape
- Use tf.Variable to modify the state of a variable
- Apply TensorFlow decorators to speed up code
- Explain the difference between a variable and a constant

## 7.1 Hyperparameter Tuning

One of the painful things about training a deep neural network is the sheer number of hyperparameters we have to deal with. Out of all sorts of hyperparameters, the learning rate $\alpha$ is the most important one to tune. Then, we would prioritize momentum term $\beta$, though its default is 0.9, mini-batch size, and number of hidden units. The third importance would be learning rate decay. When using Adam optimizer, we almost never tune $\beta_1, \beta_2, \epsilon$, which are, by default, $0.9, 0.999, 10^{-8}$. But we can tune these if we wish.

To tune the hyperparameters, how do we set values to explore? In earlier generations of machine learning algorithms, if we have two hyperparameters, it was a common practice to sample the points in a grid and systematically explore these values like the following figure:

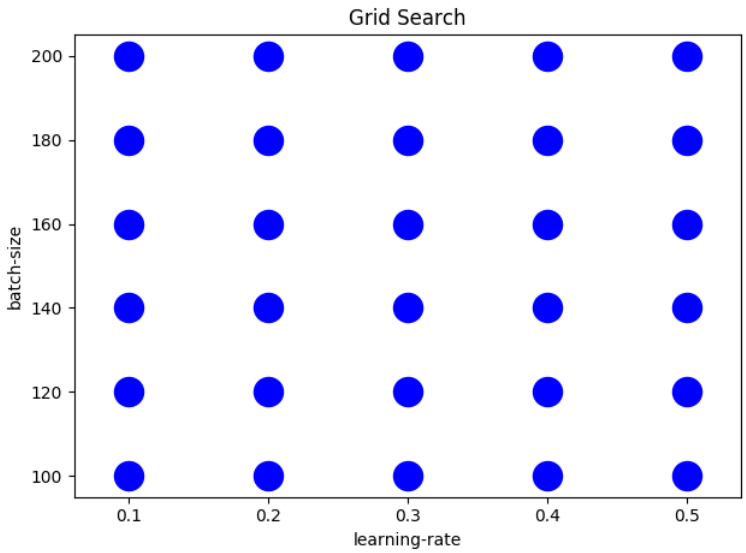



Figure 7.1: Grid Search

This works okay if the number of hyperparameters is relatively small. In deep learning, we tend to choose points at random and try out hyperparameters on the randomly chosen set of points. It's difficult to know in advance which hyperparameters are going to be the most important for the problem we work on. If we do grid search on one important hyperparameter and one unimportant hyperparameter, then we are trying out less values of the important hyperparameter. If we were to sample at random, then we will have tried out more distinct values of the more important hyperparameter and find out which value works better. The random search could be visualized as follows:

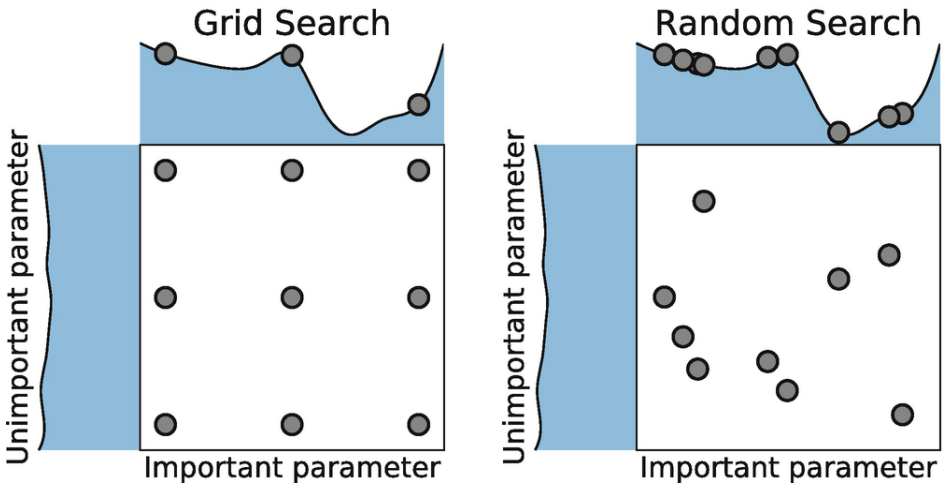



Figure 7.2: Random Search

Another common practice for sampling hyperparameters is to use a **course to fine** sampling scheme. We zoom in into a smaller region and sample more densely within the space at random for the region that generally does a good job. We can visualize as follows:

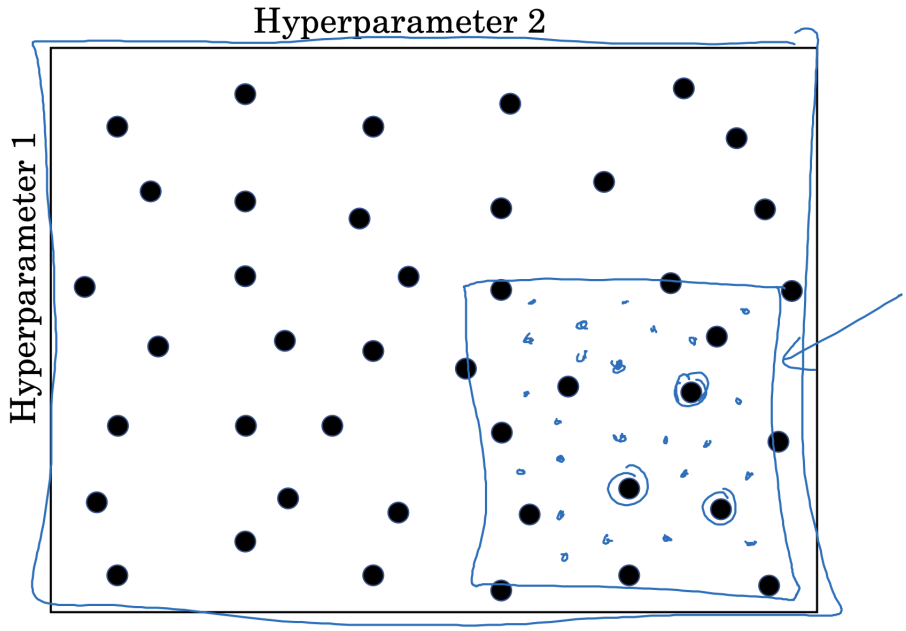



Figure 7.3: Course to Fine Search

By trying out these hyperparameters, we can pick values that allow us to do the best on the training set objective, or do the best on the development set.

### 7.1.1 Using an Appropriate Scale to Pick Hyperparameter

Sampling at random does not mean sampling uniformly at random over the range of valid values. It is important to pick the appropriate scales to explore hyperparameters.

Suppose we try to choose the number of hidden units and we believe a good range of values are $n^{[l]} = 50, \ldots, 100$. Then we can randomly pick values within this range. If, instead, we try to decide on the number of layers in the neural network $L$ and we think it should be somewhere between 2 to 4. Then, sampling uniformly within $2, 3, 4$ might be reasonable, or even using a grid search, where we explicitly evaluate the values $2, 3, 4$. These two examples are reasonable things to do. However, this is not true for all hyperparameters.

Suppose we are searching for the hyperparameter $\alpha$ and we suspect that 0.0001 might be on the low end and it could be as high as 1. If we draw a number line from 0.0001 to 1 and sample values uniformly at random over the number line, then about 90% of the values would be between 0.1 and 1. We only use 10% of resources to search for values between 0.0001 and 0.1. Instead, it seems more reasonable to search for hyperparameters on log scale, where we have $0, 0001, 0.001, 0.01, 0.1, 1$. We then search uniformly at random on the log scale, which we dedicate more resources to search between 0.0001 and 0.1. The way to implement this is as follows:

$$r = -4 * np.random.rand()$$
$$\alpha = 10^r$$

where $r \in [-4, 0]$ and $\alpha \in [10^{-4}, 0]$. More generally, if the end points of possible values are $10^a$ and $10^b$, we can sample $r$ uniformly at random between a and b.

One other tricky thing is sampling hyperparameter $\beta$ for exponentially weighted averages. Suppose we suspect $0.9 \leq \beta \leq 0.999$. Recall that using $\beta = 0.9$ is like averaging over the last 10 values, whereas using $\beta = 0.999$ is like averaging over last 1000 values. It also does not make sense to search samples on the linear scale. A good way to thing about this is that we want to explore $0.001 \leq 1 - \beta \leq 0.1$. Therefore, we can use the example we figured out before to do the search, where we sample $r \in [-3, -1]$ and we set $1 - r = 10^r \Rightarrow \beta = 1 - 10^r$. When $\beta$ is close to 1, the sensitivity of the results we get changes. If $\beta$ goes from 0.9 to 0.90005, this is hardly any change in our results, as we are still averaging roughly 10 values. However, if $\beta$ goes from 0.999 to 0.9995, this will have a huge impact on what the algorithm is doing, as we are averaging from 1000 examples to 2000 examples. This causes us to sample more densely in the regime where $\beta$ is close to 1, or $1 - \beta$ is close to 0.

### 7.1.2 Hyperparameters Tuning in Practice

There are two major schools of thoughts where people go about hyperparameter searching. One way is we **babysit one model**. We usually do this if we have a huge dataset but not a lot of computational resources, i.e. CPU, GPU., so we can only afford to train only one model or a very small number of models at a time. For example, at day 0, we might initialize parameters at random and start training. At the end of day 1, if we see the algorithm is learning quite well, we can probably increase the learning rate and see how it does. After day 2, if it still does well, then we can fill the momentum term a bit or decrease the learning rate. Every day, we look at the algorithm performance and nudge up and down hyperparameters.

The other approach would be if we **train many models in parallel**. We have a set of hyperparameters and let the algorithm run for days while keeping track of a metric. We plot the curve of metric at the end. A second model would generate another curve for the metric. At the end, we can quickly pick one set of hyperparameter values that works the best.

## 7.2 Batch Normalization

This technique can make our neural network much more robust to the choice of hyperparameters. It does not work for all neural networks, but when it does, it can make the hyperparameter search much easier and also make training go much faster. This also enable us to much more easily train even very deep networks.

### 7.2.1 Normalizing Activations in a Network

Batch Normalization was created by two researchers, Sergey Loffe and Christian Szegedy.

We saw before that data normalization could turn the contour of learning problem from very elongated shape to more round, which enables algorithms like gradient descent to optimize easier. This works by normalizing input features. How about a deeper network? If we want to train on parameters $w^{[3]}, b^{[3]}$, it would be nice if we can normalize $a^{[2]}$. Technically, batch normalization, or batch norm in short, would normalize $z^{[2]}$. There are debates about whether we should normalize before the

activation function, $z^{[2]}$, or after applying the activation function, $a^{[2]}$. In practice, normalizing on $a^{[2]}$ is done much more often.

Given some intermediate values in our neural network, $z^{[l](1)}, \dots, z^{[l](m)}$. We will compute the following for layer $l$:

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

where $\epsilon > 0$ is added for numerical stability and $\gamma, \beta$ are learnable parameters of the model. $\gamma, \beta$ allow us to set the mean and variance of $\tilde{z}$ to the values we want. In fact, if $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$, then $\gamma, \beta$ will invert the normalization equation and $\tilde{z}^{[l](i)} = z^{[l](i)}$. After normalization, we will use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$ for later computations of the neural network.

### 7.2.2 Fitting Batch Norm into a Neural Network

Suppose we have the following neural network:

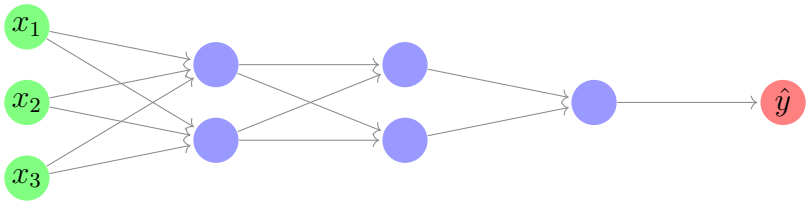



Figure 7.4: Batch Norm Example

With batch norm, we would compute the first two layers as follows:

$$x \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm(BN)}]{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \xrightarrow[\text{BatchNorm(BN)}]{\beta^{[2]},\gamma^{[2]}} \tilde{z}^{[2]} \to a^{[2]}.$$

In either case, we use normalized value $\tilde{z}$. The parameters are $w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}$ and $\beta^{[1]}, \gamma^{[1]}, \dots, \beta^{[L]}, \gamma^{[L]}$. These $\beta$'s have nothing to do with the momentum term. We might compute $d\beta^{[l]}$ and update $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$. We can also use Adam or RMSprop to update the parameters $\beta, \gamma$, not just gradient descent. In deep learning frameworks, we usually do not need to implement batch norm by ourselves.

In practice, batch norm is applied along with mini-batches of training set. We would do batch normalization on one batch at a time as follows:

$$X^{\{1\}} \xrightarrow{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm(BN)}]{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \to \dots$$

Previously, we showed that the parameters for each layer $l$ are $w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$, all of which have dimensions $(n^{[l]}, 1)$. Notice that $z^{[l]}$ is computed as

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}.$$

During the batch norm step, we compute the means of $z^{[l]}$'s and subtract out the mean. Thus, adding any constant to all examples in the mini-batch does not change anything, as the constant will be cancelled out by the mean subtraction step. Therefore, when applying batch norm, we can eliminate $b^{[l]}$'s. We end up using $\beta^{[l]}$ to decide the mean of $\tilde{z}^{[l]}$.

**Algorithm 10** Gradient Descent for Batch Normalization

**for** t=1,...,numMiniBatches **do**

Compute forward prop on $X^{\{t\}}$, where we use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$.

Use backprop to compute $dw^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$ (as $b^{[l]}$ goes away).

Update parameters as

$$w^{[l]} := w^{[l]} - \alpha dw^{[l]}; \beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}; \gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$$

**end for**

This also works with gradient descent with momentum, RMSprop, or Adam.

### 7.2.3 Why does Batch Norm Work?

Batch normalization makes weights deeper in the neural network more robust to changes to weights in earlier layers of the neural network.

In case of **covariate shift**, we would not expect our learning algorithm to do well on finding the decision boundary for binary classification task. The idea is that if we learn some $x$ to $y$ mapping, then if the distribution of $x$ changes, we might need to retrain the learning algorithm. This is true even if the mapping function remains unchanged. How does this relate to neural network? Suppose we have the following deep neural network:

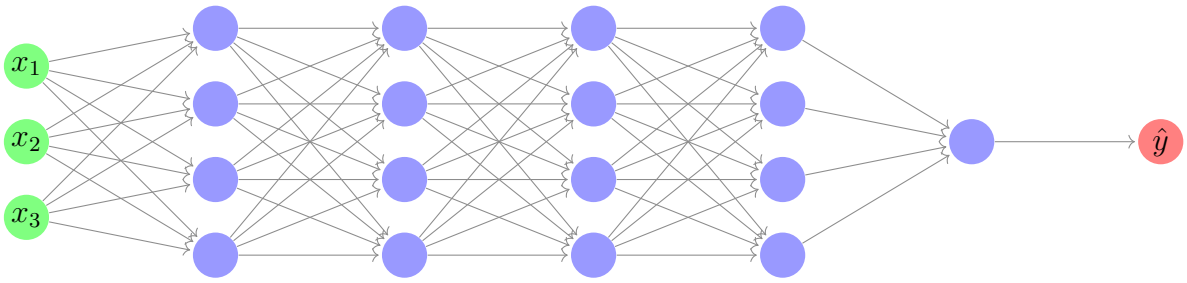



Figure 7.5: Deep Neural Network Example

Consider the learning process of the third hidden layer. The network has learned parameters $w^{[3]}, b^{[3]}$. From the perspective of the third hidden layer, it gets some values $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$. The third hidden layer takes in these values and find a way to map them into $\hat{y}$. As $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ change, the values of $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$ will also change. Therefore, the third hidden layers suffers from the covariate shift. Batch norm reduces shifting of distribution of previous hidden units. The values of $z^{[1]}, z^{[2]}, z^{[3]}, z^{[4]}$ will change, but batch norm ensures that the mean and variance of these values remain the same. It limits the amount to which updating parameters in the earlier layers can affect the distribution of values that the third hidden layer takes, thus making input values more stable.

It turns out batch norm also has some regularization effect.

- Each mini-batch $X^{\{t\}}$ has values $z^{[l]}$ scaled by the mean and variance computed on just that mini-batch, as opposed to computed on the entire dataset. In this case, the mean and variance have a bit noise in them.

- This adds some noise to the values $z^{[l]}$ to $\tilde{z}^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations. This forces the downstream hidden units not to rely too much on any one previous hidden units.

- This has a slight regularization effect – as the noise is quite small.

- By using larger minibatch size, we are reducing the noise and therefore also reducing the regularization effect.

We do not turn batch norm as a regularization. We use it as a way to normalize hidden unit activations and therefore speed up learning. The regularization is an almost unintended side effect.

### 7.2.4 Batch Norm at Test Time

Batch norm handles data for one mini-batch at a time. At test time, when we try to evaluate the network, we might not have a mini-batch of examples. We might be processing one single example at a time. Therefore, we need to do something slightly different to ensure our predictions make sense.

In order to apply neural network at test time, we need separate estimates of $\mu, \sigma^2$, which is usually done using exponentially weighted average across mini-batches. Suppose we go through mini-batches $X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$, where we get

$$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}, \ldots.$$

The exponentially weighted average of $\mu^{\{i\}[l]}$'s becomes $\mu$ we use. Similarly, by computing the exponentially weighted average of

$$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]}, \ldots,$$

we get $\sigma^2$. Then, at test time, we compute $z^{(i)}_{norm}$ as follows:

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

using the $\mu, \sigma$ obtained above. Then, we compute $\tilde{z}^{(i)}$ as normal, using the $\beta, \gamma$ parameters we have learned during the neural network training process. In practice, this process is pretty robust to the exact way we used to estimate $\mu, \sigma^2$. Any way of reasonably estimating $\mu, \sigma^2$ would work fine at test time.

## 7.3 Multi-class Classification

### 7.3.1 Softmax Regression

Softmax regression is the generalization of logistic regression, where we can make predictions on one of multiple classes, rather than just two classes.

Suppose we want to recognize cats (1), dogs (2), baby chicks (3), and others (0) as shown below:

3 1 2 0 3 2 0 1

Figure 7.6: Multi-class Classification Example

In this case, there are $C = 4$ classes and the indices are $(0, 1, 2, 3)$. In this case, we want the neural network output to have four units, where $n^{[L]} = 4$, like below:

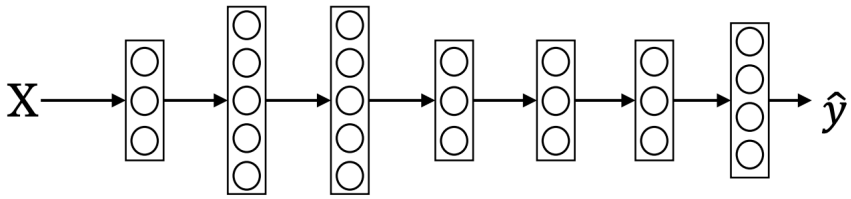



Figure 7.7: Neural Network for Multi-class Classification

We want the number in the output layer would be a $(4, 1)$ vector that tells us the probability of each of four classes in this example.

- The first node would tell us $P(other|x)$.
- The second node would tell us $P(cat|x)$.
- The third node would tell us $P(dog|x)$.
- The fourth node would tell us $P(babyChick|x)$.

Since the probability should sum to 1, the four numbers in the output layer should also sum to 1. The standard model for the neural network to do this uses **softmax layer**.

In the output layer, we are going to compute the linear part of the layer as follows:

$$z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}.$$

Now, we need to apply the softmax activation function as follows:

$$t = e^{z^{[L]}} \text{ (element-wise)}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_j}.$$

where $t$ is a $(4,1)$ vector and $a_j^{[L]} = \frac{t_j}{\sum_{j=1}^4 t_j}$.

Suppose $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$. Then, we have

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}, \quad \sum_{j=1}^{4} t_j = 176.3.$$

Therefore,

$$a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix},$$

which represents the chance of being each of the four classes.

More softmax examples are shown below, where decision boundaries are quite linear:

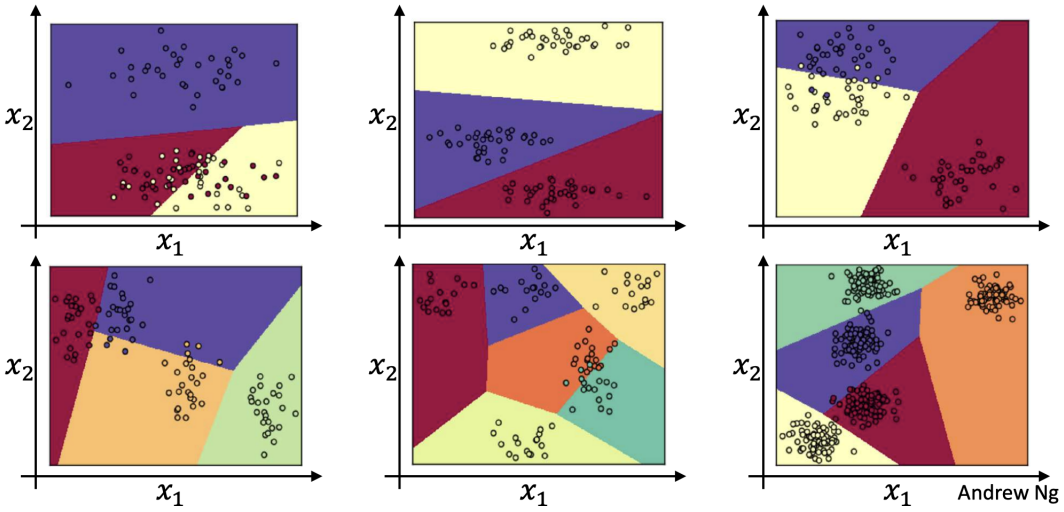



Figure 7.8: Softmax Classification Decision Boundaries without Hidden Layers

### 7.3.2 Training a Softmax Classifier

The name softmax comes from contrasting to **hardmax**, where it maps $z^{[L]}$ to $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ in our previous example, where it puts 1 at the position of the biggest element and 0's everywhere else.

Now, let's see how we train the neural network with softmax output layer. Suppose the target output is $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$. This means that the input image is a cat image.

Further suppose that $a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$, which is not doing well. In this case, the loss we use is

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log(\hat{y})_j.$$

In our example, $y_1 = y_3 = y_4 = 0, y_2 = 1$. Therefore, the only term left in the loss function is

$$-y_2 \log(\hat{y}_2) = -\log(\hat{y}_2).$$

The way to make loss small is to make $-\log(\hat{y}_2)$ small, which is equivalent of making $\hat{y}_2$ as big as possible, but no bigger than 1. This is in a form of maximum likelihood estimation.

Then, the cost function is defined as follows:

$$J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

When we do backprop, we have gradient of $dz^{[L]}$ as $dz^{[L]} = \hat{y} - y$. The programming framework would take care of gradients for us in this case.

## 7.4 Introduction to Programming Frameworks

### 7.4.1 Deep Learning Frameworks

As we implement more complex neural network models, such as convolutional neural networks or recurrent nerual networks, or we start to build very large models, it is increasingly unpractical to implement everything from scratch. There are many deep learning software frameworks that help us implement these.

The leading deep learning frameworks are outlined below:

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Criteria to choose frameworks:

- Ease of programming (development and deployment) for neural networks.
- Running speed.
- Truly open (open source with good governance) for a long time.

### 7.4.2 TensorFlow

As a motivating problem, let's say we want to minimize a cost function

$$J(w) = w^2 - 10w + 25 = (w - 5)^2.$$

The value of $w$ that minimizes the cost function is $w = 5$. Suppose we do not know the value of optimal $w$, which is often the case when we have multiple parameters, then we can find the optimal parameters using TensorFlow.

First, we need to import packages as follow:

```
import numpy as np
import tensorflow as tf
```

The good thing about TensorFlow is that we only need to implement forward-prop. TensorFlow will figure out how to do backprop, or the gradient computations by Gradient Tape.

We can find the optimal value of $w$ as follows:

```
w = tf.Variable(0, dtype=tf.float32)
optimizer = tf.keras.optimizers.legacy.Adam(learning_rate=0.1)

def train_step():
    with tf.GradientTape() as tape:
        cost = w ** 2 - 10 * w + 25
    trainable_variables = [w]
    grads = tape.gradient(cost, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))
```

After running for 1000 iterations, we obtain $w = 5.000001$ as

$$< tf.Variable'Variable : 0'shape = ()dtype = float32, numpy = 5.000001 > .$$

Suppose the cost function not only depends on parameters but also depends on the training set. We have another version of implementation as follows:

```
w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optimizers.legacy.Adam(learning_rate=0.1)

def training(x, w, optimizer):
    def cost_fn():
        return x[0] * w ** 2 + x[1] * w + x[2]
    for i in range(1000):
        optimizer.minimize(cost_fn, [w])

    return w
```

Similarly, after 1000 iterations, we obtain $w = 5.000001$.

# Part III

# Structuring Machine Learning Projects

# Chapter 8

# ML Strategy

The objective of this chapter are

- Explain why Machine Learning strategy is important
- Apply satisficing and optimizing metrics to set up your goal for ML projects
- Choose a correct train/dev/test split of your dataset
- Define human-level performance
- Use human-level performance to define key priorities in ML projects
- Take the correct ML Strategic decision based on observations of performances and dataset
- Describe multi-task learning and transfer learning
- Recognize bias, variance and data-mismatch by looking at the performances of your algorithm on train/dev/test sets

## 8.1 Introduction to ML Strategy

### 8.1.1 Why ML Strategy

Let's do a motivating example. Suppose we are working on cat classifier. After some time of training, we've gotten our system to have 90% accuracy, but this is not good enough for our application. We might have the following ideas to improve the system:

- Collect more data
- Collect more diverse training set
- Train algorithm longer with gradient descent
- Try Adam instead of grdient descent
- Try bigger network
- Try smaller netwokr

- Try dropout
- Add L2 regularization
- Network architecture (Activation functions, Number of hidden units, ...)

We discuss a number of strategies to analyze a machine learning problem that point us in the direction of the most promising things to try.

### 8.1.2 Orthogonalization

We need to have a clear eye on what to tune in order to achieve what specific effect. This is the process that we call orthogonalization.

For supervised machine learning to do wellm, we usually need to tune hyperparameters to make sure that four things hold true (chain of assumptions in ML):

1. Fit training set well on cost function (close to human-level performance)
    - Train a bigger network
    - Switch to a better optimization algorithm like the Adam
2. Fit dev set well on cost function
    - Regularization
    - Get a bigger training dataset
3. Fit test set well on cost function
    - Get a bigger dev set
4. Perform well in real world (e.g. happy cat picture app user)
    - Go back and change either the dev set or the cost function

Though a lot of people use early stopping, it is difficult to think about because it simultaneously affects how well we underfit the training set and improve the dev set performance, so it is less orthonalized. The other orthogonalization techniques would make the process of tuning network much easier.

## 8.2 Setting Up Goal

### 8.2.1 Single Number Evaluation Metric

We will find that our progress will be much faster, for tuning hyperparameters or trying out different ideas for learning algorithms, if we have a single real number evaluation metric that lets us quickly tell if the new thing we just tried is working better or worse than the last idea.

- **Precision**: of examples recognized as cat, what percentage are actually cats?
- **Recall**: of all images that are cats, what percentage were correctly recognized by the algorithm?

It turns out that there is often a trade-off between precision and recall, but we care about both. The problem of using precision and recall as the evaluation metric is that if classifier A does better on recall and classifier B does better on precision, we are unsure about which classifier is better. Therefore, rather than using precision and recall, we combine them together and use F1 score (average of precision $P$ and recall $R$):

$$\frac{2}{\frac{1}{P}+\frac{1}{R}}.$$

In Mathematics, this is called the **harmonic mean** of precision and recall. This would give us the comparison of two classifiers as follows:

| Classifier | Precision | Recall | F1 Score |
|---|---|---|---|
| A | 95% | 90% | 92.4% |
| B | 98% | 85% | 91.0% |

In this case, classifier A has a better F1 score. Assuming F1 score is a reasonable way to combine precision and recall, we can quickly select classifier A over B.

Having a well-defined dev set and a single (row) number evaluation metric help us quickly pick the best classifier. This tends to speed up the iterative process of improving our machine learning algorithm.

Suppose we are building a cat app for cat lovers in four major geographies and two classifiers achieve different errors in different geographies as follows:

| Algorithm | United States | China | India | Other |
|---|---|---|---|---|
| A | 3% | 7% | 5% | 9% |
| B | 5% | 6% | 5% | 10% |

However, keeping track of four numbers makes it very difficult for quick decision making of which classifier is better, especially if we have even more classifiers. We could also compute the average and assume average performance is a reasonable single row evaluation metric.

| Algorithm | United States | China | India | Other | Average |
|---|---|---|---|---|---|
| A | 3% | 7% | 5% | 9% | 6% |
| B | 5% | 6% | 5% | 10% | 6.5% |
| C | 2% | 3% | 4% | 5% | **3.5%** |
| D | 5% | 8% | 7% | 2% | 5.5% |
| E | 4% | 5% | 2% | 4% | 3.75% |
| F | 7% | 11% | 8% | 12% | 5.5% |

### 8.2.2 Satisficing and Optimizing Metric

It is not always easy to set up a single row evaluation metric. Sometimes, it is useful to set up satisficing as well as optimizing metrics.

Let's say we care about the classification accuracy and also the running time of our cat classifier. Suppose we have the following results:

| Classifier | Accuracy | Running time |
|---|---|---|
| A | 90% | 80ms |
| B | 92% | 95ms |
| C | 95% | 1,500ms |

We could combine accuracy and running time into an overall evaluation metric by linear weighted sum like the following:

$$cost = accuracy - 0.5 \times runningTime.$$

We could also choose the classifier that maximizes the accuracy but subject to the constraint that running time is less than 100ms. In this case, accuracy is the **optimizing metric** and running time is the **satisficing metric** – it just has to be good enough, which we do not care too much.

More generally, if we have N metrics that we care about, it is reasonable to pick one of them to be the optimizing metric and $N - 1$ to be the satisficing metrics.

Let's do another example. Suppose we are building a system to detect wake words, or trigger words. This refers to the voice control devices like the Amazon Echo, which we wake up by saying Alexa, or some Google devices, which we wake up by saying OK Google, or some Apple devices, which we wake up by saying Hey Sari, or some Baidu devices, which we wake up by saying nihaobaidu. In this case, we might care about the accuracy of trigger word detection system. We also care about the number of false positive – when nobody says the trigger words, how often does the device randomly wake up? We can reasonably combine these two evaluation metrics by maximizing accuracy, subject to the constraint that we have at most 1 false positive for every 24 hours. In this case, accuracy is the optimizing metric and false positive is the satisficing metric.

### 8.2.3 Train/Dev/Test Distributions

The way we set up training/dev/test sets have a huge impact on how rapidly we can build machine learning applications. We want to set them up to maximize efficiency.

Dev set is also called the development set, or hold out cross validation set. Suppose we are building a cat classifier over regions US, UK, Other Europe, South America, India, China, Other Asia, and Australia. We would want dev and test sets to come from the same distribution. Otherwise, months of optimization on dev set might not give us a good performance on test set. To do this, we can take all the data and randomly shuffle data into the dev and test sets, so both dev and test sets have data form all eight regions.

### 8.2.4 Size of the Dev and Test Sets

The guidelines of setting up dev and test sets are changing in the Deep Learning era. In a previous section, we've showed the ways to split data.

We want to set test set to be big enough to give high confidence in the overall performance of the system. Unless we need to have a very accurate measure of how well our final system is performing, we do not need millions of examples in the test set. If we think 10,000 or 100,000 examples in the test set would give us enough confidence, then the size would suffice. For some applications, maybe we do not even need a test set. If we have a very large dev set and we think we won't overfit the dev set too badly, then train/dev split would be sufficient.

### 8.2.5 When to Change Dev/Test Sets and Metrics?

If we put the target in a wrong place, we should move the target while training.

Suppose we build a cat classifier to try to find lots of pictures of cats to show to cat-loving users and the evaluation metric we use is classification error. Assume algorithm A has 3% error and algorithm B has 5% error. In this case, the misclassification error metric can be written as

$$J = \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} \mathbb{1}\{y_{pred}^{(i)} \neq y^{(i)}\}.$$

It seems algorithm A is doing better. However, if, for some reason, algorithm A is letting through a lot of pornographic images, even users will see more cat images, it also shows the users some pornographic images, which is totally unacceptable both for the company and the users. In contrast, algorithm B misclassifies more images, but it does not have pornographic images. From company and user acceptance points of view, algorithm B is a much better algorithm. In this case, metric and dev set prefer algorithm A, but we do prefer algorithm B in reality.

When our evaluation metric is no longer correctly rank preference orderings between algorithms, then it's a sign that we should change our evaluation metric, or dev/test set. For the above evluation metric, we can add a weight term $w^{(i)}$ and change the normalization constant as follows:

$$J = \frac{1}{\sum_i w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathbb{1}\{y_{pred}^{(i)} \neq y^{(i)}\},$$

where

$$\begin{cases} 1 & \text{if } x^{(i)} \text{ is non-porn} \\ 10 & \text{if } x^{(i)} \text{ is porn} \end{cases}$$

The idea of orthogonalization applies here, as we can define a metric to evaluate classifiers first (place the target) and then think about how to do well on this metric (shoot at the target) using similar cost function $J$ as above.

Also, if doing well on metric and de/test set does not correspond to doing well on the application/deployment, then we need to change metric and/or dev/test set, so our data better reflects the type of data we actually need to do well on.

## 8.3 Comparing to Human-level Performance

### 8.3.1 Why Human-level Performance?

There are two main reasons. First, because of the advances in deep learning, machine learning algorithms are suddenly working much better and so it becomes more feasible in a lot of application areas for machine learning algorithm to actually become competitive with human-level performance. Second, it turns out that the workflow of building machine learning systems is much more efficient when we are doing something that humans can do.

For many machine learning tasks, as we work on a problem over time, progress tends to be relatively rapid as we approach human-level performance. After a while, the algorithm surpasses the human-level performance. Then, the progress often slows down even though it keeps getting better. We hope to achieve an optimal level of performance. As we train on bigger models or on more and more data, the

performance would approach but never approach a theoretical limit, which is called the **Bayes optimal error**. This is the best possible error that there is no way to surpass that level of accuracy.

For example, in speech recognition, suppose $x$'s are audio clips. Some audios are just too noisy that it is impossible to tell what is the correct transcription. Or for cat recognition, some images might be pretty blurry that it is impossible for anyone to tell whether or not there is a cat in the picture, so the perfect level of accuracy might not be 100%.

The rise of performance can be graphed as follows:

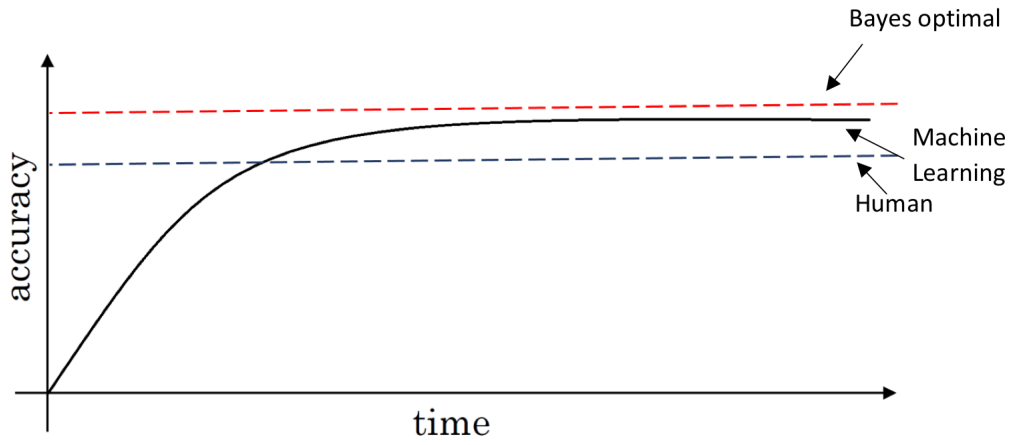



Figure 8.1: Comparing to Human-level Performance

It turns out the progress is pretty fast until we surpass human-level performance and we slow down after surpassing human level performance. There are two main reasons for that. Firstly, human-level performance, for many tasks, is not far from Bayes optimal error. Therefore, by the time our algorithm surpasses human-level performance, maybe there is not much edge room to still improve. Secondly, so long as our performance is lower than the human-level performance, there are certain tools we could use to improve performance. These tools are harder to use when we surpass human-level performance. So long as ML is worse than huamns, we can

- Get labeled data from humans.
- Gain insight from manual error analysis: Why did a person get this right?
- Better analysis of bias/variance.

### 8.3.2 Avoidable Bias

Knowing what human-level performance is can tell us exactly how well, but not too well, we want our algorithm to do on the training set.

Back to the cat classification example. Suppose human-level performance is nearly perfect where we have 1% error. In this case, if our learning algorithm achieves 8% training error and 10% dev error. We might want it to do better on the training set. The gap shows that our algorithm is not even fitting the training set well. We would need to focus on reducing bias by, for example, train on a bigger neural network or run gradient descent longer.

Suppose that the training and dev errors stay the same, but human-level error is now 7.5% for a different application, maybe because images are too blurry. In this case, the algorithm might do just fine on the training set, so we want to focus on reducing the variance of the learning algorithm by, for example, regularization.

For the cat classification example, we think of **human-level error as an estimate for Bayes optimal error**. For computer vision tasks, this is a pretty reasonable proxy because humans are very good at computer vision. What humans can do might not be too far from Bayes optimal error in this case.

We call the different between the approximate of Bayes optimal error and training error as **avoidable bias**. We want to keep improving training performance to get down to Bayes optimal error, but we cannot do better than Bayes optimal error, unless we are overfitting.

### 8.3.3 Understanding Human-level Performance

We know that human-level error is a proxy for Bayes optimal error. Let's take medical image classification as an example.

Suppose we want to look at radiology image and make diagnosis decision based on the image.

- Typical human achieves 3% error.
- Typical doctor achieves 1% error.
- Experienced doctor achieves 0.7% error.
- Team of experienced doctors achieves 0.5% error.

In this case, how do we define human-level error? It turns out if we want a proxy of Bayes optimal error, then we know from this case that Bayes optimal error is less than or equal to 0.5%. Therefore, we would use 0.5% as the estimate of Bayes optimal error and thus human-level performance. For the purpose of publishing a paper or deploying a system, there might be a different definition of human-level error that we can use – so long as we surpass the performance of a typical doctor, then the system might be good enough to deploy in some context, which is 1% in the example.

To see why this matters, here is an error analysis example. Suppose for the medical imaging diagnosis example, the training error is 5% and dev error is 6%. We think of human-level performance as proxy of Bayes optimal error, which could be 1%,0.7%,0.5% depending on our definition. Then the avoidable bias would be roughly 4% and variance is 1%. In this case, the definition of human-level performance does not matter much as avoidable bias is a bigger problem than variance problem anyways, so we would focus on bias reduction techniques.

Suppose instead that training error is 1% and dev error is 5%. Then the definition of human-level performance still does not matter too much as we would focus on variance problem over avoidable bias issue.

However, if training error is 0.7% and dev error is 0.8%. In this case, the definition of human-level performance really matters and we would go with 0.5% error, as the avoidable error would be twice as big as variance. Maybe both avoidable bias and variance are problems, but avoidable bias is a bigger problem. This also gives

a sense of why making progress in a machine learning algorithm gets harder as we approach human-level performance. Once we approach 0.7% error, unless we are very careful with estimating the Bayes optimal error, we do not know how far away we are from Bayes error and how much we should try to reduce avoidable error.

### 8.3.4 Surpassing Human-level Performance

Suppose we have a problem where a team of humans achieves 0.5% error, one human achieves 1% error, training achieves 0.6% error, and dev achieves 0.8% error. In this case, we estimate avoidable bias to be at least 0.1% and variance is 0.2%. Therefore, there might be more do to to reduce variance than avoidable bias.

A harder example would be that a team of humans achieves 0.5% error, one human achieves 1% error, training achieves 0.3% error, and dev achieves 0.4% error. Now, it is much harder to answer the question - what is avoidable bias? Does the example tell us we have overfitted by 0.2% or is Bayes optimal error actually lower? We don't really know. We do not have enough information to tell if we should focus on reducing avoidable bias or variance, which slows down our efficiency. If our error is better than a team of humans, then it is harder to rely on human intuition to tell the algorithm about ways to improve, so the ways of improving the machine learning algorithm is less clear.

Here a problems where ML significantly surpasses human-level performance:

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

All the above four examples are learning from structured data and these are not natural perception problems (computer vision, speech recognition, or natural language processing task), which humans are good at. The best systems for all four of above applications most likely have looked at far more data than any human could possibly look at. This makes it relatively easy for computer to surpass human-level performance.

Today, there are speech recognition systems, some image recognition tasks, medical tasks (e.g. reading ECG, diagnosing skin cancer, and some other radiology tasks) that surpass single human level performances.

### 8.3.5 Improving Model Performance

In this section, we show a set of guidelines to improve the performance of our learning algorithm.

There are two fundamental assumptions of supervised learning:

1. We can fit the training set pretty well, where we can achieve low avoidable bias.
2. The training set performance generalizes pretty well to the dev/test set, where we have low variance.

If we were to reduce avoidable bias (the difference between the proxy of Bayes optimal bias and training error), we should apply tactics like

- Train bigger bigger model.
- Train longer or use better optimization algorithms (momentum, RMSprop, Adam).
- Find a better Neural Netowrk architecture (change activation function, chang number of layers, chang number of hidden units, try other models, etc.) or search for better hyperparameters.

If we were to reduce variance (the difference between dev error and training error), we should apply tactics like

- Try to get more data.
- Regularization (L2, dropout, data augmentation)
- Neural Network architecture or hyperparameter search.

## 8.4 Error Analysis

### 8.4.1 Carrying Out Error Analysis

If our learning algorithm is not yet at the performance of a human, then manually examining mistakes that our algorithm is making can give us insights into what to do next. This process is called **error analysis**.

Suppose we are working with cat classifier and we achieve 10% error on the dev set, which is much worse than what we hope to do. As we look into the the misclassified examples, we realize the classify dogs into cats, for those dogs that look like cats. Should we try to make our cat classifier do better on dogs? Rather than wasting months working on this only to risk finding out at the end that it wasn't that helpful, we can use error analysis procedure that let us quickly tell whether or not this could be worth the effort.

Error analysis:

- Get $\tilde{1}00$ mislabeled dev set examples.
- Examine them manually to count up how many are dogs.

Suppose 5% of our 100 mislabeled dev set examples are pictures of dogs, which is 5 out of 100. This means of a typical set of 100 examples we are getting wrong, even if we completely solve the dog problem, we only get 5 out of 100 more correct. The error might only go down from 10% to 9.5%, which might not be the best use of our time. We call this the **ceiling on performance**.

Suppose, instead, that we fine 50 out of 100 of the first 100 mislabeled dev set examples are dog pictures. Now, we could be much more optimistic about spending time on the dog problem, as our error will go down to potentially 5%.

Sometimes, we can also evaluate multiple ideas in parallel doing error analysis. For example, let's say we have several ideas on improving our cat detector:

- Fix picture of dogs being recognized as cats.
- Fix great cats (lions, panthers, etc...) being misrecognized.
- Improve performance on blurry images.

We could create the following table, as an example, using spreadsheet, but an ordinary text file would also work.

| Image | Dog | Great Cats | Blurry | Comments |
|---|---|---|---|---|
| 1 | ✓ | | | Pitball |
| 2 | | | ✓ | |
| 3 | | ✓ | ✓ | rainy day at zoo |
| ⋮ | | | | |
| % of total | 8% | 43% | 61% | |

The conclusion of this process gives us an estimate of how worthwhile it might be to work on each of these category of errors. the outcome is not that we must work on blurry images. This does not give us a rigid mathematical formula of what to do, but it gives a sense of the best option to pursue.

### 8.4.2 Cleaning Up Incorrectly Labeled Data

Sometimes we notice that some of the examples in dev set are mislabeled. Is it worth to go in to fix up some of these labels? We use the term **incorrectly labeled example** to refer to the examples where assigned label is incorrect.

First, let's consider the training set. It turns out that **deep learning algorithms are quite robust to random errors** in the training set. So long as our errors, or our incorrectly labeled examples, are not too far from random, or our training example size is large enough that the actual percentage of errors is not too high, then it's probably okay to leave the errors as they are. There is one caveat to this – **deep learning algorithms are less robust to systematic errors**. If our labels consistently label white dots as cats, then that is a problem because our classifier will learn to classify all white colored dogs as cats.

For dev/test sets, we would need to do error analysis with one extra column **incorrectly labeled**. For example, we could have the following table

| Image | Dog | Great Cats | Blurry | Incorrectly labeled | Comments |
|---|---|---|---|---|---|
| ... | | | | | |
| 98 | | | | ✓ | Cat in background |
| 99 | | ✓ | | | |
| 100 | | | | ✓ | Drawing of a cat |
| % of total | 8% | 43% | 61% | 6% | |

Is it worthwhile to go in and try to fix this 6% of incorrectly labeled examples? If it makes a significant difference on our ability to evaluate algorithms on our dev set, then we can spend time to fix incorrect labels. If it does not make a significant difference, then it might not be the best use of our time.

It's recommended to look at **overall dev set error**, say 10%, **errors due to incorrect labels**, which in the example is 0.6%, and **errors due to other causes**,

say 9.4%. There is 9.4% worth of error that we could focus on fixing, whereas the errors due to incorrect labels in just a small fraction of the errors, so it is not the most important thing to do for now. Suppose, instead, that we have made significant progress on learning algorithm and our overall dev set error is 2%, with 0.6% errors due to incorrect labels and 1.4% errors due to other causes. In this case, we have 30% of all errors that are caused by incorrect labels. It seems much more worthwhile to fix up the incorrect labels in the dev set. Remember, the main purpose of dev set is to help us select between classifiers A and B. If one has 2.1% error and the other one has 1.9%, we do not trust these two numbers anymore as we have 0.6% of errors due to incorrect labels.

Here are some additional guidelines:

- Apply same process to our dev and test sets to make sure they continue to come from the same distribution.
- Consider examining examples your algorithm got right as well as ones it got wrong. The examples that the algorithm got right might also need to be fixed.
- Train and dev/test data may now come from slightly different distributions.

### 8.4.3 Build our First System Quickly, then Iterate

If we are working on a brand new machine learning application, we should build the first system quickly, and then iterate.

If we are thinking of building a new speech recognition system, there are a lot of directions we can go in:

- Noisy background (cafe noise, car noise)
- Accented speech
- Far from microphone (far-field)
- Young children's speech
- Stuttering, or use nonsensical phrases like oh, ah, um, ...

With many directions to do, how do we pick the one to focus on? It is recommended that we first quickly set up a dev/test set and metric. Then, we build a initial machine learning system quickly and see how we are doing with the current setup. After that, we use bias/variance analysis and error analysis to prioritize next steps.

The guideline applies less strongly if we are working on an application area in which we have significant prior experience, or if there is a significant body of academic literature that we can draw on for the exact same problem. For example, there are a lot of academic literature on face recognition and if we are trying to build a face recognizer, it might be okay to build a more complex system by building on the large body of academic literature.

## 8.5 Mismatched Training and Dev/Test Set

### 8.5.1 Training and Testing on Different Distributions

In deep learning era, more and more teams are training on data that comes from a different distribution than dev and test sets.

Suppose we are building a mobile app for cat detector. The data from mobile app tends to be less professionally shot, less well framed, and more blurry. The other source of data come from crawling the web, which has higher resolution and more professional. Suppose we have 10,000 images from the mobile app and 200,000 pictures of cat downloaded from the Internet. We care about the application doing well on the mobile app images. Now we have a dilemma as we have a relatively small number of examples from that distribution.

One option is that we put both dataset together and then take 210,000 images and randomly shuffle them into train/dev/test sets, where dev/test sets could be 2,500 each and train set could have 205,000 images. The advantage is that now train/dev/test sets all come from the same distribution. A huge disadvantage is that a lot 2,500 dev set examples would come from the Internet distribution of images, rather than mobile app images, which we care about. Since 200K out of 210K examples come from the Internet, only 119 examples in the dev set will come from mobile app images.

Instead of the above option, we can put all 200,000 images from the Internet to the training set and add in 5,000 images from the mobile app to it. Then dev and test sets would be all mobile app images. The advantage of this splitting method is that we are aiming at the target where we want it to be. The disadvantage is that our training distribution is different from dev and test sets distribution. But it turns out this split will get us better performance over the long term.

Another example. Suppose we are building a speech activated rearview mirror for a car. Where does data come from? Maybe we have worked on speech recognition for a long time and we have a lot of data from other speech recognition applications. We can take all these data like purchased data, smart speaker control, voice keyboard, etc. in the training set, where we might have 500,000 utterances from all these sources. For the dev/test set, we put maybe 20,000 speech activated rearview mirror data. Alternatively, if we think we don't need all 20,000 examples in the dev/test sets, then we can put 10,000 (half) of these into the training set, so training set could have 510,000 utterances and dev/test sets have 5,000 utterances each.

### 8.5.2 Bias and Variance with Mismatched Data Distribution

The way we analyze bias and variance changes when our training set comes from a different distribution than dev/test sets.

Let's keep using the cat classification example. Suppose humans get nearly perfect performance on this, which means 0% error, training error is 1%, and dev error is 10%. We used to conclude that we have a large variance problem, but now we cannot safely draw that conclusion. Maybe it's doing fine in the dev set because training set was really easy and dev set is much harder – much more difficult to classify correctly. Two things have changes: 1) the algorithm only sees data in the training set, but not in the dev set and 2) the distribution of data in the dev set is

different. Since we are changing two things at once, it is difficult to know the source of the 9% error increase.

In order to tease out these two effects, it will be useful to define a new piece of data, called **training-dev set**, which is a new subset of data that we carve out from the training set and should have the same distribution as training set, but not used for training. So now we split data into train/train-dev/dev/test sets, where we still train the neural network on the training set. Let's say in the example, the training error is 1%, the error on the train-dev set is 9%, and dev error is 10%. When we go from training data to train-dev data, the error went up by a lot and the only difference between training data and train-dev data is that our neural network was not trained explicitly on train-dev data. This tells us that we have a variance problem because train-dev data comes from the same distribution as the training data.

Suppose, instead, that the training error is 1%, the error on the train-dev set is 1.5%, and dev error is 10%. Now, we have a pretty low variance problem. The error really jumps when we go to dev set error, so this is a **data mismatch problem**. In this case, the algorithm has learned to do well on a different data distribution than the one we care about.

Suppose now that the training error is 10%, the error on the train-dev set is 11%, and dev error is 12%. Remember the human-level proxy for Bayes optimal error is 0%. In this case, we really have an avoidable bias problem.

Suppose now that the training error is 10%, the error on the train-dev set is 11%, and dev error is 20%. It looks like we have two issues – avoidable bias problem as well as data mismatch problem.

Sometimes if the dev/test set distribution is much easier for the application we work on, then dev error and test error could be smaller than train and train-dev error.

### 8.5.3 Addressing Data Mismatch

Unfortunately there are not super systematic ways to address this problem, but there are a few things we can try that might help.

- Carry out manual error analysis to try to understand difference between training and dev/test sets. For example, if we are building speech activated rearview mirror, we might find a lot of dev set examples are noisy, so it differs from the training set. Or it's often misrecognizing street numbers because there are a lot more navigational queries which will have street addresses.

- After gaining insights, we can make training data more similar; or collect more data similar to dev/test sets. For example, we can simulate noisy in-car data to get better on noise problem. Or try to get more data of people speaking out numbers to help street number problem.

If our goal is to make the training data more similar to the dev set, we can do **artificial data synthesis**. Let's present the idea in the context of addressing car noise problem. For a speech recognition application, maybe we do not have a lot of audio that was actually recorded inside the car with background noise. Suppose we have recorded a large amount of audio without car background noise. One popular

example that's often tested in AI is "The quick brown fox jumps over the lazy dog." – it contains every alphabet from A to Z. We can also get a car noise recording. We can synthesize these two audios by adding them together.

Suppose we have 10,000 hours of data that was recorded against a quiet background and only 1 hour of car noise. We can repeat the car noise audio 10,000 times in order to add to the audios with quiet background. However, there is a risk that our learning algorithm will overfit to the 1 hour car noise. It is possible that using 10,000 hours of unique car noise rather than just 1 hour could result in a better performance to the learning algorithm. The challenge is that to human ears, all 10,000 hours all sound the same as 1 hour.

There is another example of artificial data synthesis. Suppose we are building a self-driving car where we want to detect vehicles. One idea is that one should use computer graphics to simulate tons of images of cars. Unfortunately, we might be synthesizing, or simulating, just a tiny subset of the space of all possible cars without realizing it – as it is hard for human to tell.

## 8.6 Learning from Multiple Tasks

### 8.6.1 Transfer Learning

One of the most powerful ideas is that sometimes we can take knowledge the neural network has learned from one task and apply the knowledge to a separate task. For example, we might have a neural network learned to recognize objects like cats and then we can use (a part of) that knowledge to help us do a better job reading X-ray scans. This is called **transfer learning**.

Suppose we have a neural network for image recognition task. If we want to adapt, or transfer, the neural network to radiology diagnosis, for example, then we can delete the last output layer and all the weights fed into that layer. If we have a small dataset, then we can create a set of randomly initialized weights just for the last layer and have that output radiology diagnosis. If we have enough data, we can also retrain all the other layers of the neural network. If we retrain all the parameters in the neural network, then the initial phase of training on image recognition is called **pre-training**. Then the training on radiology data is called **fine-tuning**. This can be helpful because learning a lot of low level features, such as detecting edges, curves, and parts of an object, from a very large image recognition dataset might help our learning algorithm do better on radiology task.

Suppose we have trained a speech recognition system where inputs are audio snippets and outputs are some transcripts. Now if we want to build a wake/trigger word detection system. In order to do this, we might take out the output layer. We can also create several new layers to the neural network to create output label of the wake word detection system.

Transfer learning (try to learn from Task A and transfer the knowledge to Task B) makes sense when

- Task A and Task B have the same input $X$.
- We have a lot more data for Tasks A than Task B.
- Low level features from A could be helpful for learning B.

### 8.6.2 Multi-task Learning

For multi-task learning, we start simultaneously, trying to have one neural network do several things at the same time and each of these task helps, hopefully, all other tasks.

Suppose we are building an autonomous vehicle. Then our self-driving car needs to detect several things – pedestrians, other cars, stop sign, traffic lights, and other things, so we would have multiple output labels. Suppose we have four possible labels. Then, the cost function could be defined as follows:

$$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)}),$$

where $\mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)}) = -y_j^{(i)}\log(\hat{y}_j^{(i)} - (1 - y_j^{(i)})\log(1 - \hat{y}_j^{(i)})$.

Unlike softmax regression, now we can have multiple labels for each image. If we train a neural network to minimize the cost function above, then we are carrying out multi-task learning. One other thing we could've done is to train four neural networks, instead of training one neural network to do four things, but if some of the earlier features in the neural network can be shared among these different tasks, then we find that training one neural network on multiple tasks would result in a better performance. It turns out that multi-task learning also works even if some of the images have missing labels of some objects. With data like this, we can still train the neural network to do four tasks at the same time. For the cost function in this case, we would only sum over $j$ where object $j$ has 0/1 label.

Multi-task learning makes sense when

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually: Amount of data you have for each task is quite similar. If we have 100 tasks, each having 1,000 training examples, then training together would augment the training data from 1,000 for individual tasks to 100,000 examples.
- Can train a big enough neural network to do well on all the tasks. If we can train a big enough neural network, then multi-task learning rarely hurt performance.

In practice, though, multi-task learning is used much less often than transfer learning. There are more applications of multi-task learning in computer vision.

## 8.7 End-to-end Deep Learning

There have been some data processing systems, or learning systems, that require multiple stages of processing. End-to-end Deep Learning can take all those multiple stages and replace them with just a single neural network.

Take speech recognition as an example, where our goal is to take an input $x$ such as an audio clip and map it to the output $y$ such as the transcript of the audio clip. Traditionally, speech recognition required many stages of processing. Firstly, we need to extract some hand-designed features of the audio by MFCC. Then, we

might apply a machine learning algorithm to find phonemes in the audio clip, where phonemes are the basic units of sounds. We string phonemes together to form individual words and finally form the transcript of the audio clip. In contrast to this pipeline with multiple stages, we can train a huge neural network to just input the audio clip and have it directly output the transcript. This really obsoleted many years of research in some of the intermediate stages of processing.

One of the challenges of end Deep Learning is that we might need a lot of data before it works well. If we train on 3,000 hours of data to build a speech recognition system, then the traditional pipeline works really well. Only if we have 10,000 hours of 100,000 hours of data will the end approach start to suddenly work really well. For the intermediate size of data, maybe we can bypass certain stages, which is a step towards end-to-end learning, but not all the way there.

There is a face recognition system built by a researcher Yuanqing Lin at Baidu, where a camera looks at a person approaching the gate and if it recognizes the person, then the turnstile automatically lets the person through, without needing the person to carry an RFID badge. One thing we could do is to look at the image that the camera is capturing and try to learn a function mapping from the image to the person's identity. It turns out this is not the best approach. The person could approach in all different directions. Instead, the best approach is a multi-step approach. First, we run one piece of software to detect the person's face. Then, we zoom in to that part of the image. The image is then fed into the neural network to estimate a person's identity. Research has found that breaking the problem into two simple steps allows the learning algorithms two solve two much simpler tasks and result in a better overall performance. The two-step approach works better because

1. each of the two problems we are solving is actually much simpler.

2. we have a lot of data for each of the two sub-tasks.

In contrast if we were to learn everything at the same time, there is much less data of the form $(x, y)$, where $x$ is the image taken from the turnstile camera and $y$ is the person's identity. If we have enough data for the end-to-end approach, then end-to-end approach would work better.

Another example is machine translation. Traditionally, machine translation systems also has a long complicated pipeline, where we first take, say, English text and then do text analysis with multiple steps, and finally output a translation of the input text. For machine translation, we do have a lot of pairs of translation, end-to-end deep learning works pretty well for this task.

Suppose we want to look at an X-ray image of a hand of a child and estimate child's age. A non-end-to-end approach would be looking at an image, segmenting out bones, going to a lookup table showing average bone lengths to estimate age. In contrast, if we were to go straight from image to age, then we would need a lot of data. This approach does not work well today just because there isn't enough data to train this task in an end-to-end fashion.

### 8.7.1 Whether to use Eng-to-end Deep Learning

In this section, we present the pros and cons of the end-to-end deep learning approach, so we can come away with some guidelines of whether to use an end-to-end approach for our application.

Pros:

- Let the data speak. If we have enough $(x, y)$ data, then if we train a big enough neural network, hopefully the neural network will figure it out.
- Less hand-designing components needed.

Cons:

- May need a large amount of data.
- Excludes potentially useful hand-designed components. If we do not have a large amount of data, then the learning algorithm does not have much insight that can be gained from the data. Therefore, hand-designed features could inject manual knowledge into the algorithm.

Key question: Do we have sufficient data to learn a function of the complexity needed to map $x$ to $y$?

- We use Deep Learning to learn individual components.
- Carefully choose $(x, y)$ mapping that we want to learn, depending on the data we can get for.

# Part IV

# Convolutional Neural Networks

# Chapter 9

# Foundations of ConvNets

The obejectives of this chaper are

- Explain the convolution operation
- Apply two different types of pooling operations
- Identify the components used in a convolutional neural network (padding, stride, filter, ...) and their purpose
- Build a convolutional neural network
- Implement convolutional and pooling layers in numpy, including forward propagation
- Implement helper functions to use when implementing a TensorFlow model
- Create a mood classifer using the TF Keras Sequential API
- Build a ConvNet to identify sign language digits using the TF Keras Functional API
- Build and train a ConvNet in TensorFlow for a binary classification problem
- Build and train a ConvNet in TensorFlow for a multiclass classification problem
- Explain different use cases for the Sequential and Functional APIs

## 9.1 Introduction to ConvNets

### 9.1.1 Computer Vision

There are two reasons to be excited about deep learning for computer vision are 1) rapid advances in computer vision are enabling brand new applications to be built, which were impossible just a few years ago. With the tools, we might be able to invent some of the new products and applications and 2) as the computer vision community has been creative and inventive in coming up with new neural network architectures and algorithms, it creates a lot cross-fertilization into other areas.

There are several examples of computer vision problems.

- Image classification/recognition where we take as input a 64 by 64 image and try to figure out if it is a cat.

- Object detection where, self-driving cars for example, we don't just need to figure out if there are other cars in the image, but we figure out the positions of the other cars in the picture.

- Neural Style Transfer. Given a content image and a style image, we want the content image to be repainted into the style of style image.

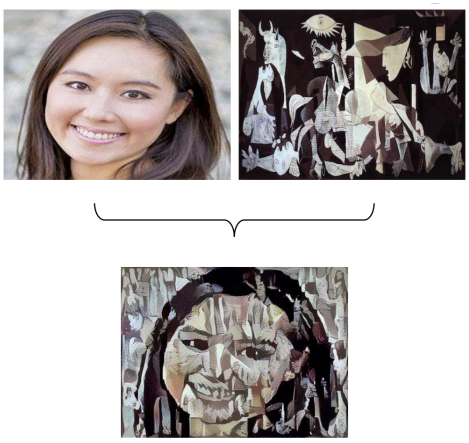


Figure 9.1: Neural Style Transfer Example

One of the challenges of computer vision problems is that the inputs can get really big. Previously, we have worked with 64 by 64 by 3 images, which is 12288 dimensional. However, that is a very small image. If we work with larger images like 1000 by 1000, which is just one megapixel, the dimension of the input features will be 1000 by 1000 by 3, which is 3 million dimensional. This means we would have 3 million $x_i$'s. If we have just 1000 hidden units in the first layer, then the total number of weights in $w^{[1]}$, if we use standard fully connected network, will be a 1000 by 3M dimensional matrix, which has 3 billion parameters, which is insanely large. With such many parameters, it is difficult to get enough data to prevent the neural network from overfitting. Also, the computational requirements and memory requirements to train the neural network is a bit infeasible.

In computer vision applications, we don't want to just deal with small images. We want to deal with large images too. To do this, we need to better implement the convolution operation, which is one of the fundamental building blocks of convolutional neural networks.

### 9.1.2 Edge Detection Example

We've showed how early layers of neural network might detect edges and then some later layers might detect parts of an object, and then parts of complete objects.

For a computer to figure out what are the objects in the picture, the first thing we might do is to detect vertical edges in the image. We might also want to detect horizontal edges. How can we do that? Here is a 6 by 6 grayscale image

| | | | | | |
|---|---|---|---|---|---|
| 3 | 0 | 1 | 2 | 7 | 4 |
| 1 | 5 | 8 | 9 | 3 | 1 |
| 2 | 7 | 2 | 5 | 1 | 3 |
| 0 | 1 | 3 | 1 | 7 | 8 |
| 4 | 2 | 1 | 6 | 2 | 8 |
| 2 | 4 | 5 | 2 | 3 | 9 |

To detect vertical edges, we can construct a 3 by 3 matrix, which is called a **filter**, or **kernel** in many research papers, as follows:

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

We are going to **convolve** the 6 by 6 image, denoted by the the asterisk (*) mark, with the 3 by 3 filter. The output of this convolution operator will be a 4 by 4 matrix. To compute the upper-left element of the 4 by 4 matrix, we take the 3 by 3 filter and paste it on top of the 3 by 3 region of our original input image. Then, we take element-wise product. In this case, we would have the following transformation:

$$\begin{bmatrix} 3 & 0 & 1 \\ 1 & 5 & 8 \\ 2 & 7 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 3\times 1 & 0\times 0 & 1\times(-1) \\ 1\times 1 & 5\times 0 & 8\times(-1) \\ 2\times 1 & 7\times 0 & 2\times(-1) \end{bmatrix} = \begin{bmatrix} 3 & 0 & -1 \\ 1 & 0 & -8 \\ 2 & 7 & -2 \end{bmatrix}.$$

We then add up all the entries, which gives us -5. So the upper-left entry of the output matrix is -5. For the second element in the first row of output matrix, we do the similar computation but on the 3 by 3 matrix obtained by shifting to one column to the right:

$$\begin{bmatrix} 0 & 1 & 2 \\ 5 & 8 & 9 \\ 7 & 2 & 5 \end{bmatrix}$$

The computation gives us -4. We do computation on other such matrices similarly, and obtain

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}.$$

If we implement this in a programming language, most of the languages will have some different functions rather than an asterisk to denote convolution. For example, Python has $conv\_forward$. In TensorFlow, there is $tf.nn.conv2d$. In keras, we have $Conv2D$.

Here is another example. Suppose we have the following grayscale image:

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}.$$

This detects the vertical edge between dark and lighter region down the middle. The convolution operation gives us a convenient way to specify how to find these vertical edges in the image.

### 9.1.3 More Edge Detection

In this section, we show the difference between positive and negative edges – the difference between light to dark versus dark to light edge transitions. Also, we present other types of edge detectors, as well as how to have an algorithm learn, rather than having us hard code an edge detector.

For the previous example, if we have dark side on the left and brighter side on the right, then the convolution will give us the following:

$$\begin{bmatrix} 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \end{bmatrix}.$$

-30 shows this is dark to light transition, as opposed to light to dark transition. If we do not care about distinguishing these two cases, then we can take the absolute values of the output matrix.

It might not be a surprise that horizontal edge detector, where pixels are relatively bright on the top and relatively dark in the bottom row, is given as follows:

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| -1 | -1 | -1 |

We can give an example as follows:

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 30 & 10 & -10 & -30 \\ 30 & 10 & -10 & -30 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

We could have other 3 by 3 filters. **Sobel filter** is given as follows:

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

which puts a little bit more weight to the central row and makes the filter more robust.

Computer vision researchers use other sets of numbers too, like **Scharr filter** as follows:

| | | |
|---|---|---|
| 3 | 0 | -3 |
| 10 | 0 | -10 |
| 3 | 0 | -3 |

We can also have nine parameters like

| | | |
|---|---|---|
| $w_1$ | $w_2$ | $w_3$ |
| $w_4$ | $w_5$ | $w_6$ |
| $w_7$ | $w_8$ | $w_9$ |

and learn these parameters using back propagation. The goal is that when input image is convolved with the parameter matrix, it gives us a good edge detector.

### 9.1.4 Padding

In order to build deep neural networks, we need to use padding to modify the basic convolutional operation.

Suppose we have an n by n input image and a f by f filter. Then the output matrix would be $n-f+1$ by $n-f+1$. There are two downsides of it: 1) our image shrinks every time we apply a convolutional operator. If we have a hundred layer deep net, then if the image thrinks a bit on every layer, then we end up with very small image after a hundred layers. 2) the edge pixels are only used in one of the outputs, whereas the pixels in the center are used much more. This means we are throwing away a lot of information from the edges of the image.

To address these two problems, we can pad the image, say one additional boarder, before apply the convolutional operation. By convention, we pad with 0's. Suppose $p$ is used to denote the padding amount, where $p = 1$ if we pad one additional layer. Then the output matrix would have dimension $n + 2p - f + 1$ by $n + 2p - f + 1$. Therefore, edge pixels are counted more.

There are two valid choices of padding: **valid and same convolutions**. Valid convolution means no padding, where $n \times n$ image, combined with $f \times f$ filter, gives $n - f + 1 \times n - f + 1$ dimensional output. Same convolution means we pad so that output size is the same as the input size, where output has dimensions $n + 2p - f + 1 \times n + 2p - f + 1$. Therefore,

$$n + 2p - f + 1 = n \Rightarrow p = \frac{f-1}{2}.$$

By convention in computer vision, $f$ is almost always odd and we rarely see even number filters in computer vision. The reason is that if $f$ is even, we need some asymmetric padding. Also, when we have an odd dimensional filter, then the filter has a central pixel.

### 9.1.5 Strided Convolutions

Stride convolution is another piece of basic building block of convolutions as used in convolutional neural networks.

We can have the following example with $stride = 2$, where we take two steps at

a time to apply the kernel:

$$\begin{bmatrix} 2 & 3 & 7 & 4 & 6 & 2 & 9 \\ 6 & 6 & 9 & 8 & 7 & 4 & 3 \\ 3 & 4 & 8 & 3 & 8 & 9 & 7 \\ 7 & 8 & 3 & 6 & 6 & 3 & 4 \\ 4 & 2 & 1 & 8 & 3 & 4 & 6 \\ 3 & 2 & 4 & 1 & 9 & 8 & 3 \\ 0 & 1 & 3 & 9 & 2 & 1 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 91 & 100 & 83 \\ 69 & 91 & 127 \\ 44 & 72 & 74 \end{bmatrix}$$

If we have $n \times n$ input image and $f \times f$ filter, with padding $p$ and stride $s$, then the output dimensions are

$$\lfloor \frac{n+2p-f}{s} + 1 \rfloor \times \lfloor \frac{n+2p-f}{s} + 1 \rfloor.$$

In case the fraction is not an integer, we round the number down.

So far, what we are doing is actually cross-correlation, instead of convolution. If we were to do convolution, as presented in some math and signal processing textbook, we would need to flip the rows and columns of the filter as follows:

$$\begin{bmatrix} 3 & 4 & 5 \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 7 & 9 & -1 \\ 2 & 0 & 1 \\ 5 & 4 & 3 \end{bmatrix}$$

However, in deep learning literature, we just call cross-correlation as convolution. By convention in deep learning, we do not bother with the flipping operation.

Convolution enables us to enjoy associativity, as $(A * B) * C = A * (B * C)$, which is nice for some signal processing applications. However, for deep neural networks, this does not really matter.

### 9.1.6 Convolutions Over Volumns

In this section, we show how to implement convolutions over three dimensional volumns.

Suppose we want to detect features not just in a grayscale image, but in a RGB image, where it could be a stack of three $6 \times 6$, which has dimensions $6 \times 6 \times 3$ images. We can convolve this with a 3D filter of dimensions $3 \times 3 \times 3$. In this case, the output would be a $4 \times 4$ matrix. To apply convolution, we will place the $3 \times 3 \times 3$ filter to the top-left corner of the 3D input. Then, we multiply each of 27 numbers with its corresponding numbers in the filter. By adding them up, we get the first element in the output matrix. In this case, if we want to detect edges in the red channel of the image, then we can apply the following filter:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

We can also convolve the input image on two different filters and stack two filter outputs together as follows:

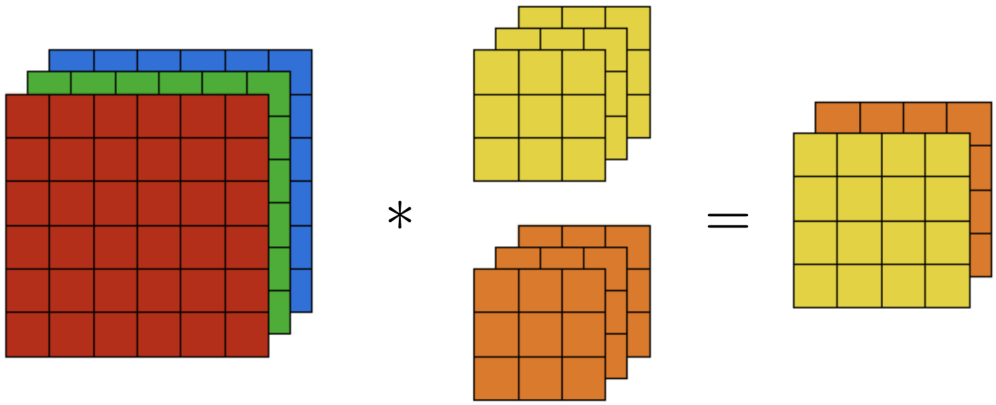


Figure 9.2: Convolution with Two Filters

More generally, if we have $n \times n \times n_c$ dimensional image, where $n_c$ is the number of channels, or depth of the 3D volumn, and convolve it with $f \times f \times n_c$ filter, then the output has dimensions $n - f + 1 \times n - f + 1 \times n'_c$, where $n'_c$ is the number of filters we use, assuming we use stride of 1 and no padding.

### 9.1.7 One Layer of a Convolutional Network

After applying convolution with multiple filters, each filter will give us a output matrix. To turn this into a convolutional neural network layer, we add a bias term and then apply a non-linearity, which gives us the output of the neural network layer. We can visualize it as follows:

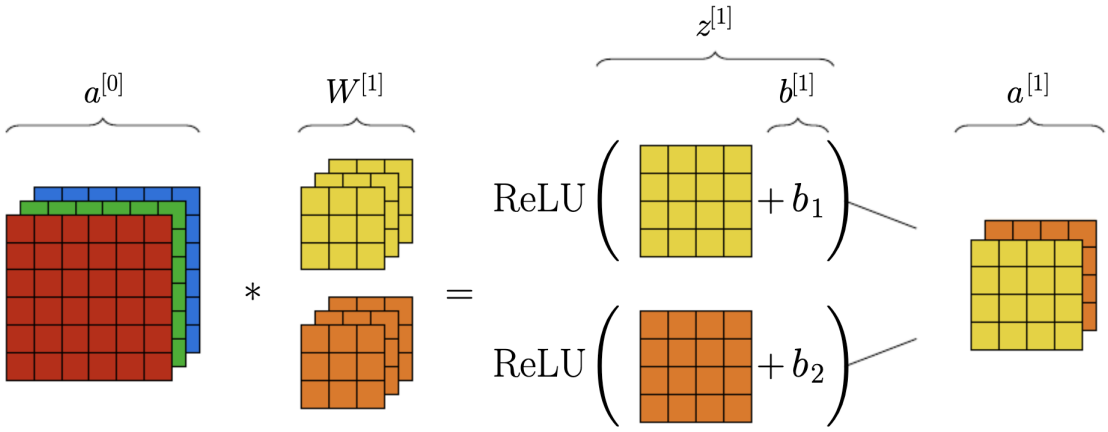



Figure 9.3: One Layer of Convolutional Network

Suppose we have 10 filters that are $3 \times 3 \times 3$in one layer of a neural network, how many parameters does that layer have? Each filter has 27 parameters and one bias, which results in a total of 28 parameters for each filter. Then, for 10 filters, there are $28 \times 10 = 280$ parameters.

We now summarize the notations. If layer $l$ is a convolutional layer, with input of size $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$, which means the output size is $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$, where

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

then

- $f^{[l]}$ = filter size
- $p^{[l]}$ = amount of padding
- $s^{[l]}$ = stride
- $n_c^{[l]}$ = number of filters
- Each filter has size $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
- Activation $a^{[l]}$ has dimensions $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$. If we have $m$ examples, then $A^{[l]}$ has dimensions $m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- Weights $W^{[l]}$ would have size $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
- Size of bias is given by $n_c^{[l]}$. In code, it is more convenient to represent it as $(1, 1, 1, n_c^{[l]})$ dimensional tensor.

There are three types of layers in a convolutional network:

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

Although it is possible to design a pretty good neural network using just convolutional layers, most neural network architectures will also have a few pooling layers and a few fully connected layers.

### 9.1.8 Pooling Layers

ConvNets also use pooling layers to reduce the size of the representation, to speed up the computation, as well as make some of the features that detects a bit more robust.

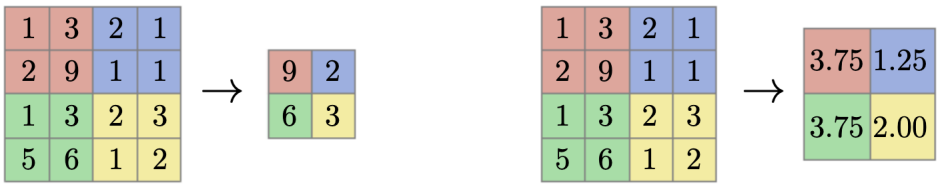
Let's go though an example of pooling. Suppose we have a 4 by 4 input as follows:

$$\begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix}$$

We want to apply a type of pooling called **max pooling**. Suppose filter size is $f = 2$ and stride is $s = 2$. The output would be a 2 by 2 matrix, where we segment matrix into four 2 by 2 matrices and then each element in the output matrix would be the max of the corresponding region. The max operation preserves the features detected in the quadrants. The main reason why people use max pooling is because it's been found in a lot of experiments to work well. Once we fix $f, s$, there is nothing for gradient descent to learn.

There is another type of pooling that is not used very often – **average pooling**, where we average within each filter. Sometimes, we might use average pooling very

deep in a neural network to collapse our representation from, say, $7 \times 7 \times 1000$ to $1 \times 1 \times 1000$.




**(a)** Max pooling **(b)** Average pooling

Figure 9.4: Pooling Layers with $f = 2, s = 2$

In either case, the hyperparameters are

- $f$ : filter size
- $s$ : stride
- $p$ : padding (rarely used when we do max pooling, which does not use padding)

The common choices are $f = 2$ and $s = 2$. This has the effect of roughly shrinking the height and width by a factor of about 2. $f = 3, s = 2$ is also used sometimes.

Max pooling is used most commonly because it generally produces better results. Some people hypothesize that max pooling performs better because it captures in each filter if a certain aspect of the image is present, while average pooling can be saturated with a lot of data.

### 9.1.9 CNN Example

Suppose we are inputting a $32 \times 32 \times 3$ image, which contains digit 7. We want to build a neural network to do digit recognition. The following is inspired by LeNet-5 by Yann LeCun.

Suppose we have $f = 5, s = 1$, then our apply 6 filters, add a bias, and apply ReLU non-linearity. Our CONV1 would be of dimensions $28 \times 28 \times 6$. Then, we apply a pooling layer with max pooling, with $f = 2, s = 2$. The output POOL1 has dimensions $14 \times 14 \times 6$. When people report the number of layers in a neural network, usually people just report the number layers that have weights. As pooling layers have no weights, we put CONV1 and POOL1 together as Layer 1. If we apply non-linearity with $f = 5, s = 1$ and 16 filters, we obtain CONV2 with dimensions $10 \times 10 \times 16$. Then, we apply max pooling with $f = 2, s = 2$, the output POOL2 would be of dimensions $5 \times 5 \times 16$. Similarly, CONV2 and POOL2 make up of Layer 2. We flatten the output POOL2 into a $400 \times 1$ dimensional vector, as $5 \times 5 \times 16 = 400$. Then, we take this 400 units and build the next layer with, say, 120 fully connected layer FC3, where the weight matrix $w^{[3]}$ would have dimensions $120 \times 400$ and bias $b^{[3]}$ would have dimensions $(120, 1)$. Lastly, we take 120 units and have another smaller fully connect layer FC4 – say with 84 units. We then have 84 real numbers that we can feed into a softmax unit. If we want to classify digits $1, 2, \ldots, 9$, then softmax would have 10 outputs. The whole neural network is summarized as follows:

| | Activation shape | Activation Size | # parameters |
|---|---|---|---|
| Input: | (32,32,3) | 3,072 | 0 |
| CONV1 (f=5,s=1) | (28,28,6) | 4,704 | 456 |
| POOL1 | (14,14,6) | 1,176 | 0 |
| CONV2 (f=5,s=1) | (10,10,16) | 1,600 | 2,416 |
| POOL1 | (5,5,6) | 400 | 0 |
| FC3 | (120,1) | 120 | 48,120 |
| FC4 | (84,1) | 84 | 10,164 |
| Softmax | (10,1) | 10 | 850 |

One common guideline for setting hyperparameters is to look at the literature to see what hyperparameters worked for others. There is a chance that the same values of hyperparameters would work well for our application too.

As we go deeper into the network, $n_H$ and $n_W$ usually decrease. $n_c$ will increase. Another common neural network pattern we see is that we have one CONV layer, followed by a POOL layer, so on and so forth. At the end, we have some fully connected FC layers, followed by a softmax.

### 9.1.10 Why Convolution?

There are two main advantages of convolutional layers over just using fully connected layers: 1) parameter sharing and 2) sparsity of connections.

Let say we have a $32 \times 32 \times 3$ dimensional image, which has size of 3,072. We use $f = 5$ with 6 filters, which gives us a $28 \times 28 \times 6$ dimensional output, which has size of 4,704. If we were to connect every one of these neurons, then the number of parameters in the weight matrix would be $3,072 \times 4,704 \approx 14M.$, which is a lot of parameters to train. Today we can train neural networks with even more parameters, but the input image is really small in this case. If the input has dimensions $1000 \times 1000 \times 3$, then the number of features would be infeasibly large. There are $(5 \times 5 \times 3 + 1)6 = 456$ parameters for the first CONV layer, which remains quite small. The reason that a ConvNet has relatively small number of parameters is that

- **Parameter sharing**, which is motivated by the fact that a feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image. This is true for low-level features, as well as high-level features like detecting the eye that indicates a face or a cat.

- **Sparsity of connections**, where in each layer, each output value depends only on a small number of inputs.

Through these two mechanisms, a neural network has a lot fewer parameters, which allows it to be trained with smaller training set and is less prone to be overfitted. Sometimes we hear about ConvNets being very good at capturing translation invariance. This is the observation that if we shift a picture of cat by a couple of pixels to the right, it is still pretty clearly a cat. The convolutional structure helps the neural network encode the fact that an image shifted by a few pixels should result in pretty similar features and should probably be assigned the same output label.

How can we train one of these networks? Suppose we have an image as input $x$ and a label for many classes as output $y$. If we choose ConvNet structure, then we would define cost function similar to before, as we have parameters $w, b$:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

We run gradient descent, or another optimization algorithm, to optimize parameters to reduce the cost $J$. We can build a very effective cat detector or some other detectors out of this.

# Chapter 10

# Deep Convolutional Models

The objectives of this chaper are

- Implement the basic building blocks of ResNets in a deep neural network using Keras
- Train a state-of-the-art neural network for image classification
- Implement a skip connection in your network
- Create a dataset from a directory
- Preprocess and augment data using the Keras Sequential API
- Adapt a pretrained model to new data and train a classifier using the Functional API and MobileNet
- Fine-tine a classifier's final layers to improve accuracy

## 10.1 Classic Networks

In this section, we present some of the classic neural network architectures like LeNet-5, AlexNet, and VGGNet.

### 10.1.1 LeNet-5

LeNet-5 was named after the author Yann LeCun. The goal of LeNet-5 was to recognize handwritten digits, where input images are grayscale with dimensions $32 \times 32 \times 1$. In the first step, we use a set of 6 $5 \times 5$ filters with stride $s = 1$ and no padding. The output dimensions are $28 \times 28 \times 6$. Then, the neural network applies average pooling (when paper was published this was used much more), with $f = 2, s = 2$. This reduces the height and width by a factor of 2 and the output dimensions are $14 \times 14 \times 6$. Then, we have another convolutional layer, with 16 $5 \times 5$ filters, $s = 2$, and no padding. The output would have dimensions $10 \times 10 \times 16$. Then, we apply another average pooling layer with $f = 2, s = 2$, which gives us $5 \times 5 \times 16$ dimensional output. The next layer is a fully connected layer that fully connects each of the 400 nodes with every one of 120 neurons. This is followed by another fully connected layer with 84 units. The next step is to use these 84

features to give one final output $\hat{y}$, where $\hat{y}$ takes 10 possible values corresponding to recognizing each of the digits from 0 to 9. The modern neural network would use a softmax layer to produce 10 dimensional output, though back then LeNet-5 used a different classifier – one that's useless today. The network is summarized below:

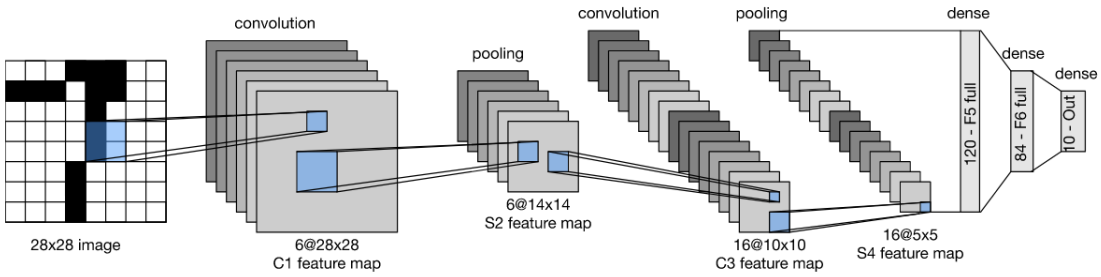



Figure 10.1: LeNet-5 Architecture

One pattern we see in this neural network is that $n_H, n_W$ go down while $n_C$ goes up. Another pattern is that we have CONV layers followed by one or more POOL layer. Back then in the paper, people used sigmoid and tanh non-linear functions, instead of ReLU. To save some computation time and some parameters, the original LeNet-5 had complicated ways with different filters for different channels of the input block, but modern implementation would not have that complexity. Also, the original LeNet-5 applied sigmoid non-linearity after pooling layers. If we were to read the paper, it is recommended to focus on sections II and III.

This neural network is small by modern standard, which has 60K parameters. Nowadays we would often see neural networks with 10M to 100M parameters.

### 10.1.2 AlexNet

The second example of neural network is AlexNet, named after the first author Alex Krizhevsky.

AlexNet's inputs start with $227 \times 227 \times 3$ images. Note that the paper refers to $224 \times 224 \times 3$, but the numbers only make sense with $227 \times 227 \times 3$ inputs. The first layer applies a set of 96 $11 \times 11$ filters with a stride of $s = 4$. As it uses a large stride of 4, the dimensions shrink to $55 \times 55 \times 96$. Then, we apply a max pooling layer with a $3 \times 3$ filter and stride of $s = 2$. This reduces the volumn to $27 \times 27 \times 96$. Then, it performs a $5 \times 5$ SAME convolution layer, with 256 filters and padding $p = 2$. This gives us a $27 \times 27 \times 256$ dimensional output. We then apply a max pooling layer with a $3 \times 3$ filter and stride $s = 2$, which gives us the output of dimensions $13 \times 13 \times 256$. Then, we apply the SAME convolution layer **twice**, with 384 $3 \times 3$ filters and padding $p = 1$, which results in the output of volumn $13 \times 13 \times 384$. After that, we apply another SAME convolution layer, with 256 $3 \times 3$ filters and padding $p = 1$, which results in the output of volumn $13 \times 13 \times 256$. After that, we apply a max pooling layer with a $3 \times 3$ filter and stride $s = 2$, which brings the dimensions down to $6 \times 6 \times 256$. If we multiply out these numbers, we have $9,216$. Therefore, we unroll these into $9,216$ nodes. Then, we have two fully connected layers with $4,096$ units each. Finally, we uses a softmax layer to output a 1000 dimensional vector, representing each of 1000 possible classes.

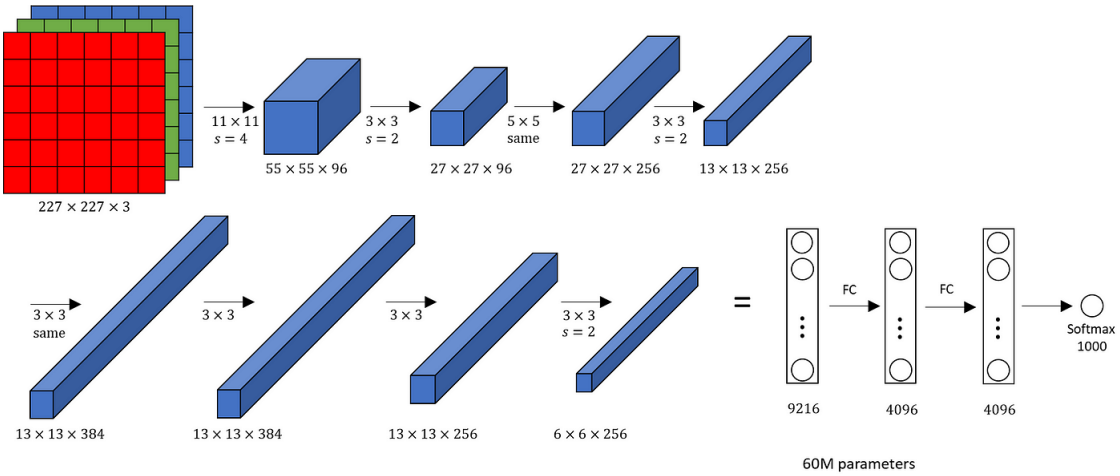



Figure 10.2: AlexNet Architecture

AlexNet has some similarities with LeNet-5, but is much bigger. Whereas LeNet5 had 60K parameters, AlexNet has 60M parameters. They follow a similar building block, but AlexNet has a lot more hidden units and trains on a lot more data. Also, AlexNet uses ReLU as the activation function, which also contributed to why it performed remarkably well. Back in the days GPUs were still a little slow, so AlexNet paper showed a complicated way of training on two GPUs, where a lot of layers are split across two different GPUs and there was a thoughtful way for when the two GPUs would communicate with each other. The original AlexNet also had another set of layers called **Local Response Normalization (LRN)**, which is not used much. Suppose we have a volumn of dimensions $13 \times 13 \times 256$. Then LRN will look at one position of $13 \times 13$ positions, look down across all the channels, and normalize them. The motivation is that we do not want too many neurons with a very high activation. However, subsequent researchers have found that this does not help too much. For completeness, one example is shown below:

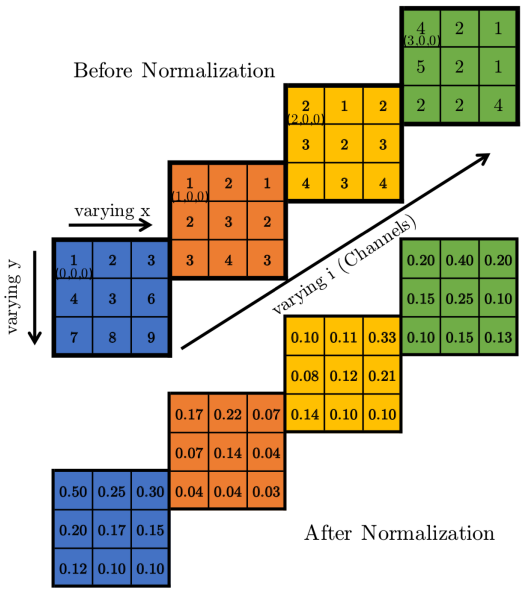



Figure 10.3: Local Response Normalization (LRN) Example

### 10.1.3 VGGNet

It was the AlexNet paper that convinced the computer vision community to take a serious look at deep learning. However, whereas AlexNet had a relatively complicated architecture, there are a lot of hyperparameters. There is a third network called VGG, or VGG-16, network. A remarkable thing about this network is that instead of having so many hyperparameters, we use a much simpler network, where we focus on having SAME CONV layers with filters of $3 \times 3$ and stride $s = 1$ and MAX-POOL layers with a $2 \times 2$ filter and stride $s = 2$. This simplifies the neural network architecture presented in AlexNet. VGG-16 refers to the fact that this neural network has 16 layers that have weights. This is a pretty large network even by modern standard – it has a total of about 138M parameters, but the simplicity makes it quite appealing.

More specifically, with input dimensions $224 \times 224 \times 3$, we first apply the SAME CONV layer with 64 $3 \times 3$ filters and stride $s = 1$ **twice**, both giving us an output of dimensions $224 \times 224 \times 64$. Next, we use a MAX-POOL layer, which gives us output dimensions of $112 \times 112 \times 64$. Then, we apply the SAME CONV layer with 128 $3 \times 3$ filters and stride $s = 1$ **twice**, both giving us an output of dimensions $112 \times 112 \times 128$. This is followed by a default MAX-POOL layer, which gives us output dimensions $56 \times 56 \times 128$. Next, we apply the **three** SAME CONV layer with 256 $3 \times 3$ filters and stride $s = 1$, each giving us an output of dimensions $56 \times 56 \times 256$. Next, we have a default MAX-POOL layer, which gives us output dimensions $28 \times 28 \times 256$, followed by **three** SAME CONV layer with 512 $3 \times 3$ filters and stride $s = 1$, each giving us an output of dimensions $28 \times 28 \times 512$. Another MAX-POOL layer is followed to give us $14 \times 14 \times 512$ output volumn. We repeat to apply **three** 512 filter SAME CONV layers, each giving us an output of dimensions $14 \times 14 \times 512$, and a MAX-POOL layer, which gives us an output of dimensions $7 \times 7 \times 512$. We then feed the output into **two** consecutive fully connected layer with 4096 units each. Finally, we apply a softmax that outputs one of 1000 classes.

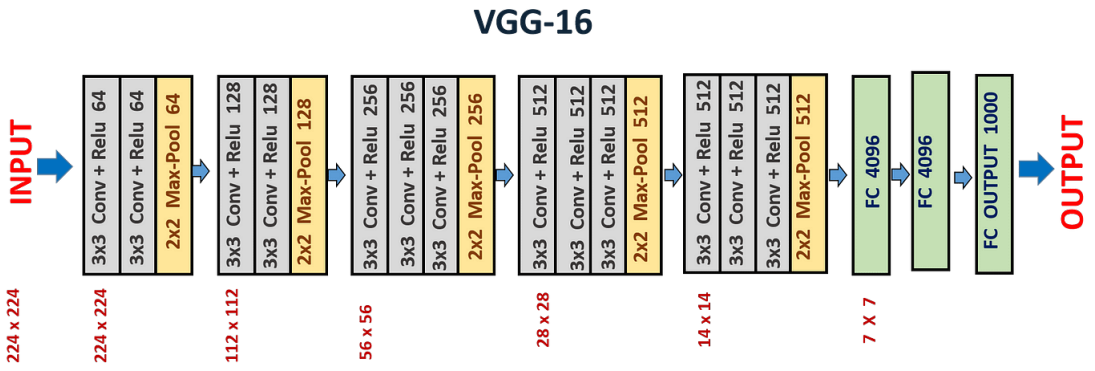



Figure 10.4: VGG-16 Architecture

The main downside of VGG-16 network is that it is a pretty large network in terms of the number of parameters we have to train. If we see literature, people sometimes talk about VGG-19, which is a bigger version of this network. Because VGG-16 does almost as well as VGG-19, a lot of people just use VGG-16.

## 10.2 Residual Networks (ResNets)

Very deep neural networks are difficult to train, because of vanishing and exploding gradients. In this section, we present skip of connections that allows us to take the activation from one layer and suddenly feed it to another layer that is much deeper in the neural network. This allows us to train very deep neural networks, sometimes with even over 100 layers.

ResNets are built out of **residual blocks**. Originally, we follow a "main path" to obtain $a^{[l+1]}$ from $a^{[l]}$ as follows:

$$a^{[l]} \xrightarrow{W^{[l+1]}a^{[l]}+b^{[l+1]}} z^{[l+1]} \xrightarrow{g(z^{[l+1]})} a^{[l+1]} \xrightarrow{W^{[l+2]}a^{[l+1]}+b^{[l+2]}} z^{[l+2]} \xrightarrow{g(z^{[l+2]})} a^{[l+2]}.$$

Rather than following the main path, we can fast forward $a^{[l]}$ as follows:

$$a^{[l]} \xrightarrow{g(z^{[l+2]}+a^{[l]})} a^{[l+2]},$$

which is referred to as **"short cut", or skip connection**. In this case, $a^{[l]}$ skips over a layer and pass information deeper into the neural network. The inventors of residual network, Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, found that using residual blocks allows us to train much deeper neural networks. The way we build a ResNet is by taking many of these residual blocks and stacking them together to form a deep network. The ResNet structure could be visualized as follows:

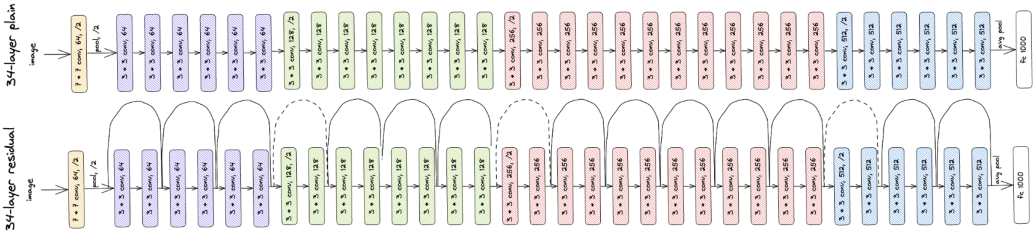



Figure 10.5: ResNet Architecture

Suppose we fast forward $a^{[l]}$ prior to applying activation for $a^{[l+2]}$. Then we have

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

For the sake of argument let's say $W^{[l+2]} = 0$ and $b^{[l+2]} = 0$ and we are using ReLU activation function for all layers. Then, $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$. This shows identity function is easy for residual block to learn because of the skip connection. Adding additional two layers in the neural network does not hurt our ability to do as well as the neural network without extra layers. Sometimes we get lucky and the additional layers even helps performance. When we train on plain neural networks, it is very difficult to choose parameters that learn even the identity function when we go deep into the network, which is why a lot of layers end up making our result worse.

One thing to note is that we assume $z^{[l+2]}$ and $a^{[l]}$ have the same dimensions, so we see ResNet uses a lot of SAME CONV layers, so the dimensions would be the

same and we can do the short-circuit connection. In case these two does not have the same dimensions, we add an extra matrix $W_s$ that maps $a^{[l]}$ to the dimension of $z^{[l+2]}$. $W_s$ could be a matrix of parameters to be learned, or a fixed matrix that applies zero padding. Either of these could work.

## 10.3 $1 \times 1$ Convolutions

In terms of designing architectures, one of the ideas that really help is using a $1 \times 1$ convolution.

While a $1 \times 1$ convolution would simply scale up each value in a 2D input, with 3D inputs, we can do much more. It's like a fully connected neural network layer, which takes in $n_W \times n_H$ inputs and gives a $\#\ filters$ dimensional output. This can carry out a pretty non-trivial computation on the input volumn. This is also sometimes called **network in network**.

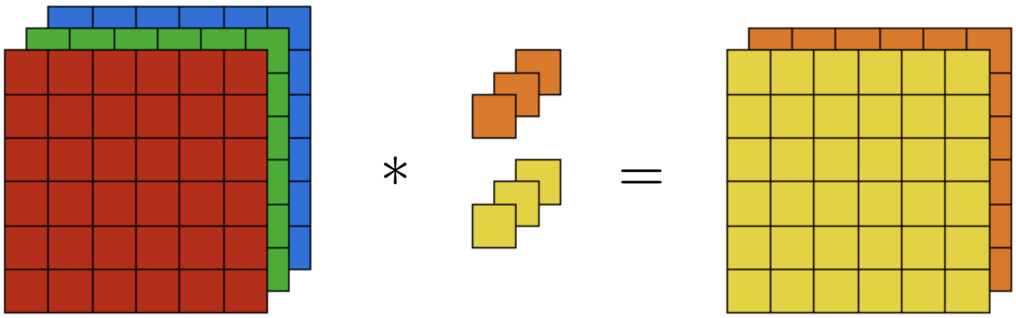


Figure 10.6: $1 \times 1$ Convolution

There are two things we can do with $1 \times 1$ convolution: 1) shrink the number of channels in the input and save computations, and 2) if the output dimensions are the same, then it adds non-linearity to learn more complex function.

## 10.4 Inception Network

When designing a layer for a ConvNet, we might have to pick the dimensions of the filter. Inception Network enables us to do them all. This makes the neural network architecture more complicated, but it also works remarkably well.

Suppose we have a $28 \times 28 \times 192$ input. For one layer, we can apply 64 $1 \times 1$ filters and gives a part of output as $28 \times 28 \times 64$. Then, we can apply one SAME CONV layer with 128 $3 \times 3$ filters, which gives an output of dimension $28 \times 28 \times 128$. We can then add another SAME CONV layer with 32 $5 \times 5$ filters, which gives output of dimensions $28 \times 28 \times 32$. Then, we can apply a MAX-POOL layer and have output $28 \times 28 \times 32$. To match the dimensions, we would need to use stride $s = 1$ and padding, which is unusual. By concatenating all these outputs together, we get a single $28 \times 28 \times 256$ dimensional output.

There is a problem with the inception layer, which is computational cost. Let's look at the SAME CONV layer with 32 $5 \times 5$ filters. In this case, we need to compute $28 \times 28 \times 32$ numbers, each having $5 \times 5 \times 192$ computations. If we multiply out

all these numbers, it gives us 120M. While we can do 120M multiplications on a modern computer, it is still a pretty expensive operation. By using $1 \times 1$ CONV layers, we can reduce the computational cost by a factor of about 10.

Prior to applying the $5 \times 5$ CONV layer, we apply a $1 \times 1$ CONV layer with 16 filters. Then, we have a smaller volumn of dimensions $28 \times 28 \times 16$. Then, we apply the $5 \times 5$ CONV layer to the smaller volumn to give us the final output of the dimensions $28 \times 28 \times 32$. Sometimes the result of $1 \times 1$ filter is called the bottleneck layer. The cost of computing the intermediate $28 \times 28 \times 16$ dimensional output is $28 \times 28 \times 16 \times 192 = 2.4M$, as we need to do 192 multiplications for each of them. For each output in the second output, we do $5 \times 5 \times 16$ computations, thus resulting in a total of $28 \times 28 \times 32 \times 5 \times 5 \times 16 = 10.0M$. Therefore, the total number of computation would be 12.4M, which reduced by a factor of about 10. Shrinking down the representation size so dramatically would not hurt the performance of our neural network so long as we apply the bottleneck layer. For MAX-POOL layer, we apply 32 $1 \times 1$ filters to shrink the output of dimensions $28 \times 28 \times 192$ down to $28 \times 28 \times 32$, so we don't end up having MAX-POOL layer taking up most of the channels in the final output.

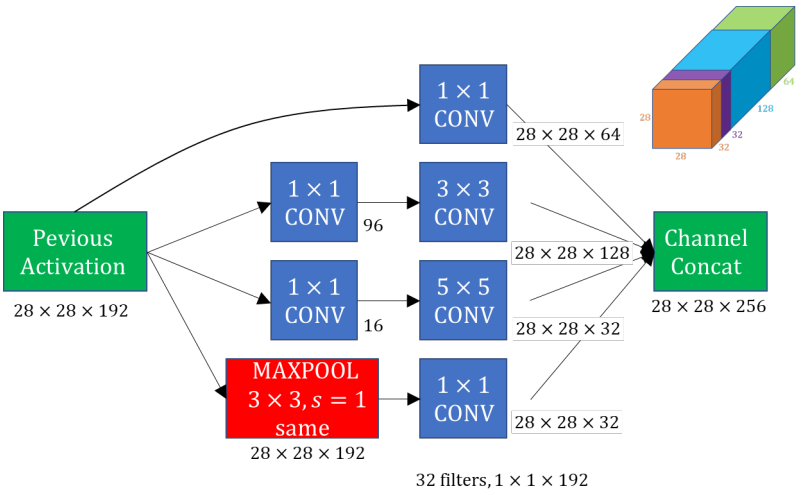



Figure 10.7: Inception Module Structure

Inception Networks more or less puts a lot of these inception modules together, shown as follows:

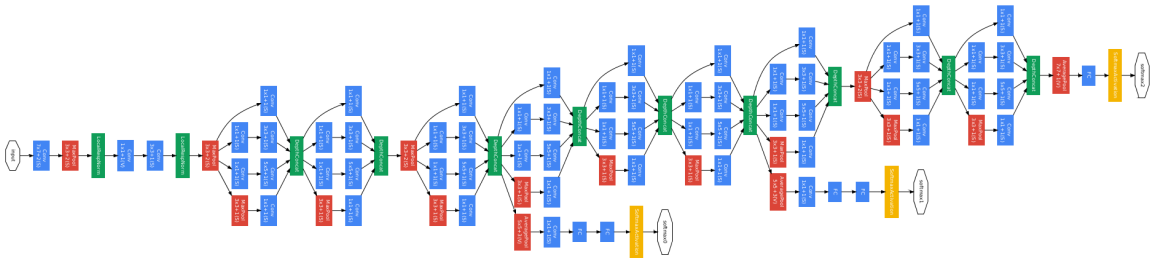


Figure 10.8: Inception Network Architecture

This is the inception network developed by people at google and it's called **googLeNet**. We notice that there are additional side branches. The last few layers of the network are fully connected layers, followed by a softmax layer to make a prediction. The side branches take some hidden layers, pass through a few fully connected layers and have a softmax to make a prediction. This helps ensure that features computed, even in the intermediate layers, are not too bad for predicting the output class of the image. This appears to have a regularizing effect on the inception network and helps prevent this network from overfitting.

Finally, the name *inception* is motivated to build deeper neural networks, and the inception network authors cite the meme


Figure 10.9: Meme cited by Inception Network Authors

## 10.5 MobileNet

MobileNet is another foundational neural network architecture used for computer vision. Using a MobileNet allows us to build and deploy new networks that work even in low compute environments, such as mobile phone.

All the neural network architectures we have presented are quite computationally expensive. If we want the neural network to run on a device with less powerful CPU or a GPU at deployment, MobileNet performs much better. The key idea is **depthwise-separable convolutions**.

In a normal convolution, we have $n \times n \times n_C$ input and convolves it with $n'_C$ $f \times f \times n_C$ filters, which gives a smaller $n_{out} \times n_{out} \times n'_C$ output. The computational cost is given by

$$\#\text{ filter parameters} \times \#\text{ filter positions} \times \#\text{ filters}$$

Suppose we apply 5 $3 \times 3 \times 3$ filters to the input of size $6 \times 6 \times 3$. We would have output of dimensions $4 \times 4 \times 5$. The computational cost is given by

$$(3 \times 3 \times 3) \times (4 \times 4) \times 5 = 2160.$$

The depthwise-separable convolution has two steps. We first use a **depthwise convolution**, followed by a **pointwise convolution**. First, for depthwise convolution, we have input dimensions $n \times n \times n_C$ and apply $n_C$ $f \times f$ filters to it. We apply one of each filters to one of the corresponding input channels. Collectively, they give

a $n_{out} \times n_{out} \times n_C$ dimensional output. For the example we have, the computational cost is given by

$$(3 \times 3) \times (4 \times 4) \times 3 = 432.$$

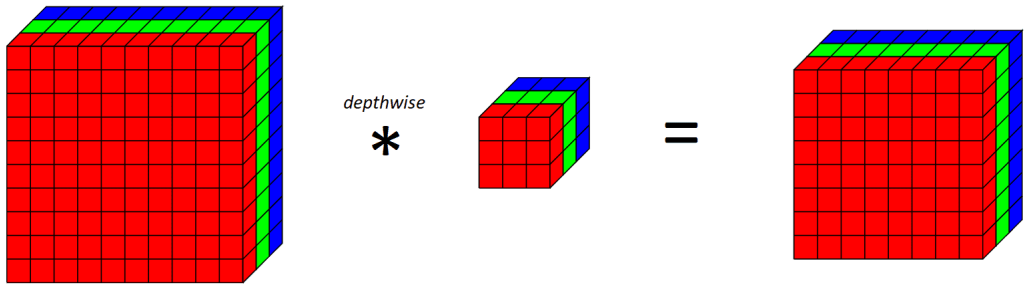



Figure 10.10: Depthwise Convolution

For the pointwise convolution, we take a input of dimensions $n_{out} \times n_{out} \times n_C$ and convolve it with $n'_C$ filters of dimensions $1 \times 1 \times n_C$, which gives us the output of dimensions $n_{out} \times n_{out} \times n'_C$. The computational cost, for our example, is given by

$$(1 \times 1 \times 3) \times (4 \times 4) \times 5 = 240.$$

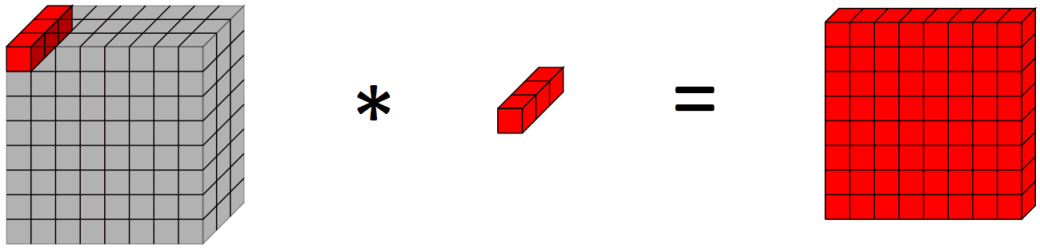



Figure 10.11: Pointwise Convolution

In total, we have a total cost of computation as

$$432 + 240 = 672.$$

This turns out to be $672/2160 = 0.31$ as computationally expensive as cost of normal convolution. In general, the ratio of the cost is given by

$$\frac{1}{n'_C} + \frac{1}{f^2}.$$

In a normal network, $n'_C$ would be fairly large, say 512 and $f$ would be 3. Therefore, roughly the ratio in practice would be about 1/10.

By taking the above building block, we can build a MobileNet. The idea is that we use much less expensive depthwise-separable convolution operation, comprising the depthwise convolution operation and poinwise convolution operation. The **MobileNet v1** paper used the block of depthwise-separable convolution 13 times. After these 13 layers, the last layers are the usual POOL layer followed by a FC layer, and then followed by a softmax. This turns out to be performing well while being much less computationally expensive than earlier architectures.

The **MobileNet v2** by Sandler et al in 2019 has an improvement with two main changes 1) addition of residual connection, or skip connection, and 2) adds an **expansion layer** before the depthwise convolution. MobileNet v2 happens to repeat such a block 17 times, followed by the usual POOL, FC, and softmax layers. The block is also called the **bottleneck block**.

MobileNet v2 has an input of size, say, $n \times n \times 3$. Then, we apply an expansion operation with a large number of $1 \times 1 \times 3$ filters, say 18 of them, so we end up with a $n \times n \times 18$ dimensional output. Expansion factor of 6 is quite typical in MobileNet v2. Then, we apply a depthwise separable convolution. With some padding, we can obtain an output with the same dimensions. Finally, we apply a pointwise convolution with 3 $1 \times 1 \times 18$ dimensional filters. We end up with the output of dimensions $n \times n \times 3$. Since the dimensions wend down, we call the pointwise convolution step in the bottleneck block as **projection step**. It turns out the bottlenet block accomplishes two things:

1. By using the expansion operation to increase the size of the representation within the bottleneck block, neural network can learn a richer function.

2. When deploying on a mobile device, we will often be heavily memory constrainted. Therefore, the pointwise convolution projects the volumn down to a smaller set of values, so when we pass to the next block, the memory needed to store the values is reduced back down.

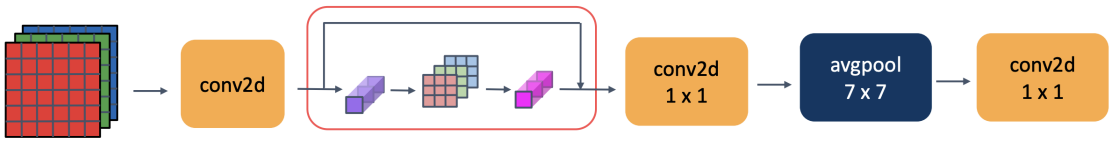



Figure 10.12: MobileNet v2 Architecture

## 10.6 EfficientNet

EfficientNet (Tan and Le, 2019) gives us a way to scale up or down neural networks for a particular device.

There are three things we could do to scale things up or down.

1. Use a higher resolution image.

2. Make the network much deeper.

3. Make the layers wider.

Depending on the computational resources we have, we can also use **compound scaling**, where we simultaneously scale up or scale down the resolution of images and the depth/width of the neural network. However, at what rate do we scale up to get the best possible performance within our computational budget? An open-source implementation would help us choose the trade-off among these three numbers.

## 10.7 Practical Advice for Using ConvNets

### 10.7.1 Using Open-Source Implementation

It turns out that a lot of the neural networks are difficult, or finicky, to replicate because of a lot of details about tuning of the hyperparameters 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. If we can get the author's implementation, we can get going much faster than trying to reimplementing from scratch.

### 10.7.2 Transfer Learning

If we are building a computer vision application, rather than training the weights from scratch and random initialization, we often make much faster progress if we download weights that someone else had already trained on the network architecture and use that as pre-training and transfer that to a new task that we are interested in. Sometimes the training takes several weeks and might take many GPUs. The fact that someone else has done this and gone through the painful hyperparameter searching process means that we can often download open-souce weights that take someone else many weeks or months to figure out. We can use that as a very good initialization for our own neural network.

Suppose we are building a cat detector to recognize our own cats. Let's say our cats are Tigger and Misty, which are common cat names according to the Internet. So we would have a classification task with three classes – Tigger, Misty, and Neither. We ignore the case where both cats appear in the picture.

We probably do not have a lot of pictures of Tigger or Misty, so our training set will be small. We can download some open-source implementation of a neural network – not just the code but also the weights. There are lots of networks that have been trained on, for example, ImageNet dataset, which has 1000 different classes. We can get rid of the softmax layer and create our own softmax unit that outputs three classes. We can think of all intermediate layers as frozen and we only train parameters associated with the softmax layer. By using someone else's pre-trained weights, we might be able to get pretty good performance on the task even with small training dataset. A lot of deep learning frameworks support this mode of operation. There is one other neat trick that may help for some implementations. Because all of the early layers are frozen, there is some fixed function that doesn't change, which takes an input image $x$ and maps it to some set of activations in a later layer. The trick to speed up training is that we can pre-compute the activationss of that layer and just save them to disk.

When we have a larger training set, we can then freeze fewer layers and train on later layers. We can take the weights of last few layers and run gradient descent to train the weights. Or, we can blow away the last few layers and just use our own hidden layers. In extreme cases, if we have huge amount of training data, then we can take the open-source neural network weights as initialization and train the whole network, with a replaced softmax layer.

In computer vision, transfer learning is something that we should almost always do, unless we have exceptionally large dataset to train everything else from scratch by ourselves.

### 10.7.3 Data Augmentation

Data augmentation is one of the techniques that is often used to improve the performance of our computer vision systems. Computer vision is a pretty complicated tasks where we need to learn descently complicated functions to do the task. In practice, for almost all computer vision tasks, having more data will help.

There are several common data augmentation methods.

- The simplest method is **mirroring** on the vertical axis. If we mirror an image of cat, we still get a cat image. If the mirroring preserves the objects we try to recognize, then mirroring would be a good data augmentation technique to use.

- Another commonly used technique is **random cropping**, where we crop on different parts of an image. This is not a perfect data augmentation technique, as there is a possibility that we miss the main object that we try to detect, but in practice it works well, so long as we randomly crops reasonably large subsets of the actual image.

- In theory, we can also do rotation, shearing, local warping, and so on, though in practice they seem to be used a bit less.

- We can also use **color shifting**. We can add to the RGB channels different distortions. For example, we can add 20 to red and blue channels and subtract 20 from the green channel, which would make the image more purply. In practice, we draw RGB from some distribution. Even though we have color distortion, the main identity stays the same. There are many ways to sample RGB values. One of the ways to implement color distortion uses **Principle Component Analysis (PCA)** algorithm. The detail is given in the AlexNet paper. This is sometimes called **PCA color augmentation**. If the image mainly has red and blue tints but very little green, then PCA color augmentation will add and subtract a lot to red and blue, but relatively little to green channel, so the overall color of the tint the same.

Similar to other parts of training a deep neural network, the data augmentation process also has some hyperparameters, such as how much color shifting do we implement and parameters we use for random cropping. Therefore, a good starting point would be using someone else's open-source implementation.

Here are some implementation details about distortion during training. We might have data stored in the hard disk. If we have a small training set, then we can do almost anything. However, if we have a large training set, then we have a CPU thread that is constantly loading images from the hard disk. We use CPU thread to implement distortions needed to form a batch, or many batches (mini-batches), of data. This data is constantly passed to some other thread for training. This could be done on CPU or GPU. Often, the image distortion and training can run in parallel.

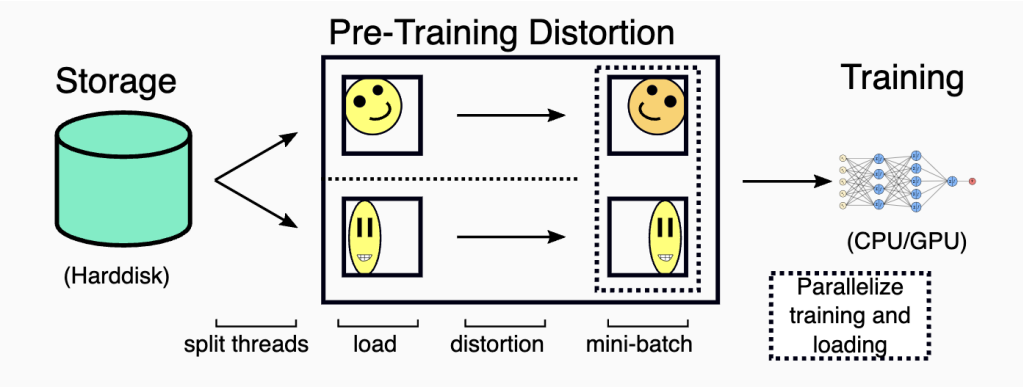



Figure 10.13: Implementing Distortions during Training

### 10.7.4 State of Computer Vision

Deep learning has been successfully applied to computer vision, natural language processing, speech recognition, online advertising, logistics, many, many, many problems. There are a few things that are unique about the application of deep learning to computer vision, about the status of computer vision.

We can think of most machine learning problems falling in the spectrum between having little data and having lots of data. Today, we have a descent amount of data for speech recognition. Even though there are reasonably large datasets today for image recognition, or image classification, this is still a complicated problem. It feels over millions of images is still not enough. So it is more towards little data end relative to speech recognition. For object detection, we have even less data because of the cost of getting the bounding boxes. It is more expensive to label the objects and bounding boxes, so we tend to have less data for object detection than for image recognition.

On average, when we have a lot of data, people tend to get away with using simpler algorithms and less hand-engineering features. In contrast, if we have little data, then people engage in more hand-engineering features and more “hacks." There are two sources of knowledge for learning algorithms: 1) labeled data $(x, y)$ and 2) hand engineered features/network architecture/other components. Since computer vision is trying to learn a really complex function, it often feels like we do not have enough data for computer vision. Even though datasets are getting bigger and bigger, we often do not have as much data as we need. This is why the state of computer vision has replied more on hand engineering and rather complex architectures. When we do not have enough data, hand engineering is a very difficult, skillful task that requires a lot of insights. Fortunately, transfer learning helps a lot when we have little data.

From computer vision literature and the ideas out there, we find people are really enthusiastic. They are really into doing well on standardized benchmarks dataset and winning competitions. For computer vision, if we do well on thebenchmarks, it is easier to get papers published. This helps the whole community figure out what are the most effective algorithms. We also see in the papers that people do things that allow us to do well on a benchmark, but not in a production or a system.

Here are some tips on doing well on benchmarks and winning competitions:

- **Ensembling**. After we figure out what neural network we want, train several

neural networks independently and average their output $\hat{y}$'s, not the weights. This might allow us to do 1% or 2% better on some benchmark. 3-15 networks is quite typical. However, this is almost never used in production to serve actual customers, unless we have a huge computational budget. We would need to keep all these 3-15 networks around. This takes up a lot more computer memory.

- **Multi-crop** at test time. This is a form of applying data augmentation to our test image as well. We run classifier on multiple versions of test images and average results. One example would be 10-crop, where we crop central, top-left, top-right, bottom-left, bottom-right parts of both the original image and the mirrored image. This, compared to ensembling, does not suck up as much memory as we only keep one neural network, but it still slows down our runtime.

Because of a lot computer vision problems are in the small data regime, others have done a lot of hand engineering of network architectures. A neural network that works well on one vision problem often surprisingly work well on other vision problems. Therefore, we often do well starting off with someone else's neural network architecture. We can use open source code as follows:

- Use architectures of networks published in the literature.
- Use open source implementations if possible.
- Use pretrained models and fine-tune on our dataset.

# Chapter 11

# Object Detection

The objectives of this chapter are

- Identify the components used for object detection (landmark, anchor, bounding box, grid, ...) and their purpose
- Implement object detection
- Implement non-max suppression to increase accuracy
- Implement intersection over union
- Handle bounding boxes, a type of image annotation popular in deep learning
- Apply sparse categorical crossentropy for pixelwise prediction
- Implement semantic image segmentation on the CARLA self-driving car dataset
- Explain the difference between a regular CNN and a U-net
- Build a U-Net

## 11.1 Detection Algorithms

### 11.1.1 Object Localization

For **classification with localization**, the algorithm not only detects an object but also is responsible for putting a bounding box, or a red rectangle, around the position of the object in the image, where the term *localization* refers to figuring out where in the picture is the object we have detected. For both image classification and classification with localization, we usually have 1 object in the middle of the image that we try to recognize. In detection problem, however, there could be multiple objects, potentially of different categories, within a single image.

In addition to the neural network structure we use for image classification, we can have a few more output units that output, say $b_x, b_y, b_h, b_w$, that parametrize the bounding box of the detected object, where the top-left corner of the image has coordinates $(0, 0)$ and lower-right has $(1, 1)$ and the mid-point coordinates of the bounding box is denoted by $(b_x, b_y)$.

Now, we can formally define the target label $y$. Suppose we have four classes: pedestrian, car, motorcycle, and background and the image has only one of these object. We would need to output $b_x, b_y, b_h, b_w$ and class label (1-4). The target label $y$ is a vector. The first element $p_c$ denotes if there is an object. If the object is pedestrian, car, motorcycle, then $p_c = 1$. If the classification is background, then $p_c = 0$. We can think of $p_c$ as the probability that one of the classes we try to detect is there. Next, if there is an object, we want to output $b_x, b_y, b_h, b_w$. Finally, if $p_c = 1$, we also want to output $c_1, c_2, c_3 \in \{0, 1\}$, denoting the prediction of one of the classes.

We can define the loss function to be the squared error function as follows:

$$\begin{cases} \sum_{i=1}^{8} (\hat{y}_i - y_i)^2 & \text{if } y_1 = p_c = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = p_c = 0 \end{cases}$$

where 8 comes from the dimension of our target label $y$.

### 11.1.2 Landmark Detection

In more general cases, we can have a neural network just output $x$ and $y$ coordinates of important points in the image, sometimes called **landmarks**.

Suppose we are building a face recognition application and we want the algorithm to tell where is the corner of someone's right eye. In this case, we can have the neural network output two more layers $l_x, l_y$ to tell the coordinates. If we want four corners of both eyes, then we can modify the neural network to output $l_{1x}, l_{1y}, \ldots, l_{4x}, l_{4y}$. Say we want more points – 64 points, or landmarks, for example. By selecting a number of landmarks and generating a labeled training set that contain all these landmarks, we can have a neural network to tell us where are the key positions on a face. In case of 64 landmarks, our output would contain $p_c, l_{1x}, l_{1y}, \ldots, l_{64x}, l_{64y}$, which is $128 + 1 = 129$ dimensional. This is the basic building block of recognizing emotions from faces, special effects for AR augmented reality filters on entertainment apps like Snapchat, and computer graphics effects that warp the face or draw various special effects like putting a crown or a hat on the person. To train a network like this, we would need a labeled training set, so someone has to go through and laboriously annotate all these landmarks.

### 11.1.3 Object Detection

In this section, we show how to use a ConvNet to perform object detection using **sliding windows detection algorithm**.

Suppose we want to build a car detection algorithm. We can create a labeled training set, with closely cropped examples of cars, where input $x$'s are all just cars centered in the entire image. Given the training set, we can train on a ConvNet that inputs an image and outputs $y \in \{0, 1\}$. Then we can use the ConvNet in sliding windows detection. We start by picking a specific square window size and input a small rectangular region of the windows size from the original image into the ConvNet and let ConvNet to make prediction. We repeat the process until we have slid the window across every position, with some stides, in the image. After running it once, we repeat the whole process with larger window sizes using some stride. The hope is that so long as there is a car somewhere in the image, there will

be a window where ConvNet outputs 1 for the car region. However, there is a huge disadvantage of sliding windows detection, which is the computational cost, As we are cropping out so many different square regions in the image and running each of them independently through a ConvNet. If we use a very coarse stride, then the number of windows we need to pass through the ConvNet will reduce, but that may hurt performance.

Before the rise of neural networks, people used to use much simpler classifiers like a simple linear classifier over hand-engineered features in order to perform object detection. Because each classifier was relatively cheap to compute, sliding windows detection ran okay. However, running a single classification with ConvNet is much more expensive and thus sliding windows detection is infeasibly slow.

### 11.1.4 Convolutional Implementation of Sliding Windows

We can turn fully connected layers in the neural network into convolutional layers. Suppose we have the following neural network where we have $14 \times 14 \times 3$ input images:

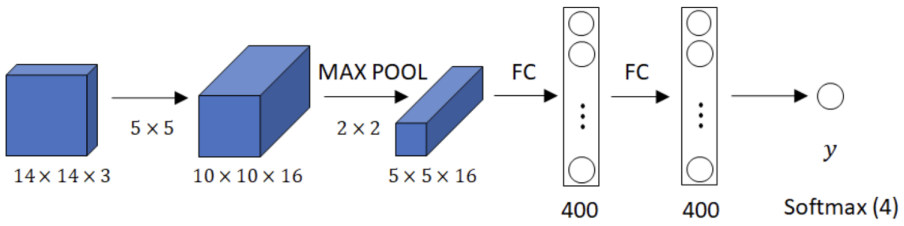



Figure 11.1: ConvNet Example

In order to predict the label of four classes, we can change the softmax layer to output 4 numbers corresponding to the probabilities for four class. Another way to implement the first fully connected layer is to apply 400 $5 \times 5$ filters. This gives us a $1 \times 1 \times 1 \times 400$ dimensional output. Instead of viewing 400 as a set of nodes, we now view it as a $1 \times 1 \times 400$ volumn. Mathematically, it is the same as a fully connected layer. Each of the 400 values is some arbitrary linear function of the $5 \times 5 \times 16$ activations in the previous layer. Next, we implement a $1 \times 1$ convolution with 400 filters. This gives us another fully connected layer with output dimensions $1 \times 1 \times 400$. Finally, we apply another $1 \times 1$ filter followed by a softmax activation, which gives us a $1 \times 1 \times 4$ volumn.

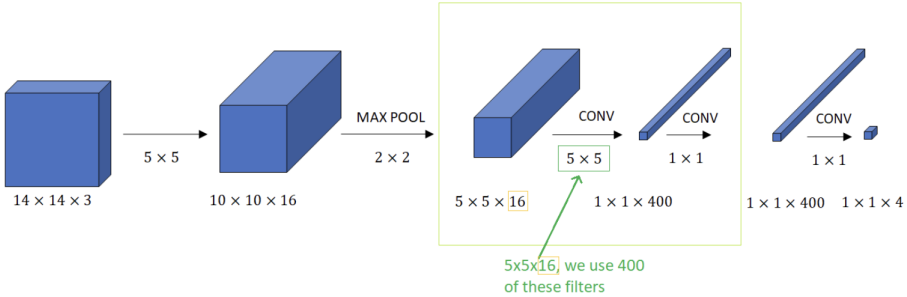



Figure 11.2: Turn FC Layers into CONV Layers

With this conversion, we can implement a convolutional implementation of sliding windows object detection. Previously, we implement the sliding windows detection in ConvNet as follows:

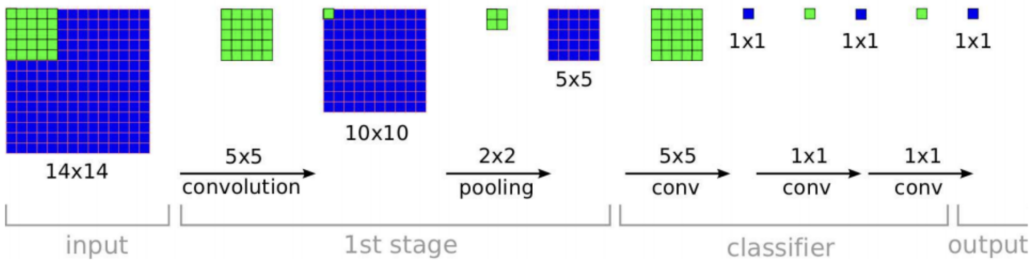



Figure 11.3: Normal Sliding Windows Detection

We can add two additional pixels for row and column and use a stride of 2 to slide over as follows:

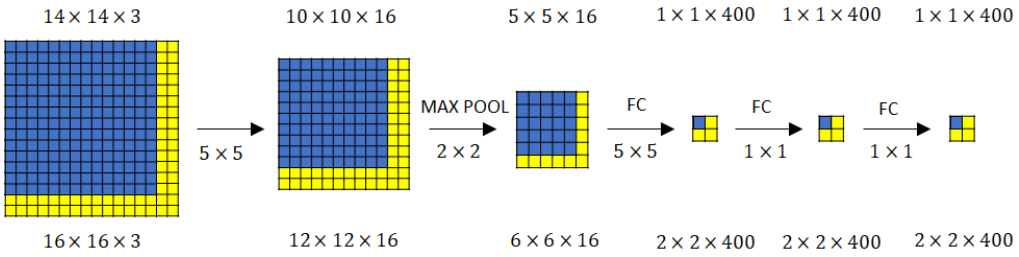



Figure 11.4: Sliding Windows Detection with Stride of 2

The blue $1 \times 1 \times 4$ subset gives us the result of upper-left hand corner $14 \times 14$ image. The upper-right one gives us the result of running ConvNet for the shifted window region. Other volumns give us similar results for different sliding windows. The max pooling of $2 \times 2$ corresponds to running our neural network with stride of 2 for the original image. With the new convolutional implementation, instead of applying ConvNet sequentially, we can implement the entire image and convolutionally make all the predictions at the same time by one forward pass the the big ConvNet.

### 11.1.5 Bounding Box Predictions

Even though the computational implementation of sliding windows is more computationally efficient, it still has a problem of not quite outputting the most accurate bounding boxes. A good way to get the more accurate bounding boxes is with the **YOLO (You Only Look Once) algorithm** by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi.

Suppose we have a $100 \times 100$ image. We put a grid on the image – it could be $3 \times 3$ or a finer on $19 \times 19$ in practice. We apply the image classification and localization algorithm to each of the nine grids in the $3 \times 3$ case. For each grid cell, we would have a 8 dimensional output $y$. Once an object is detected, YOLO algorithm takes the midpoint of the object and assign it to the grid cell containing the midpoint. The advantage of this algorithm is that the neural network outputs precise bounding boxes. For each of the output for individual grid, we can read off

$p_c$ as 1 or 0 to see if there is an object detected. If there is an object, we can know the label and box positions precisely in that grid cell. As long as we do not have more than one object in each grid cell, the algorithm should work okay. For each object, even if the object spans multiple grid cells, the object is only assigned to one of the cells. One nice thing about the YOLO algorithm is that it is a convolutional implementation, which runs pretty fast. Therefore, this works even for real time object detection.

Now, we show more about the details of how we encode $b_x, b_y, b_h, b_w$. $b_x, b_y$ are straightforward as we can find the coordinates of midpoint relative to $(0, 0)$ and $(1, 1)$ positions as defined previously. The height of the bounding box is specified as the fraction of the overall height of the grid cell. Similarly, the width of the bounding box is specified as the fraction of the overall width of the grid cell. Therefore, by definition, $0 \leq b_x, b_y \leq 1$ and it is possible for $b_h, b_w > 1$. If we want to read the YOLO paper, it was one of the harder papers to read.

## 11.2 Intersection Over Union

How do we tell if our object detection algorithm is working well? We can use a function called **Intersection Over Union (IoU)** to evaluate our object detection algorithm and add another component to our algorithm to make it work even better.

IoU computes the union and intersection of the predicted and actual bounding boxes. We then divide the size of intersection by the size of union. By convention, we would say the answer is correct if $IoU \geq 0.5$. If the predicted and the ground-truth bounding boxes overlapped perfectly, the IoU would be 1. The higher the IoU, the more accurate the bounding box. More generally, IoU is a measure of the overlap between two bounding boxes.

## 11.3 Non-max Suppression

One problem of the object detection so far is that our algorithm may find multiple detections of the same objects. **Non-max Suppression** helps us make sure that our algorithm detects each object only once and is a tool we can use to make the outputs of YOLO algorithm work even better.

If we use $19 \times 19$ grid cells, then many of the cells could have $p_c$ labeled as 1 when we have a few objects. Therefore, we may end up having multiple detections for each object. Non-max suppression cleans up the detections, so we end up just having one detection for each object. It looks at the probabilities of each detection and takes the largest one. Then, we look at all the remaining rectangles and suppress the ones with the highest IoU's with the rectangle that has the highest probability. Next, we go through the remaining rectangles and find one with the highest probability. Then, we get rid of the other rectangles with high IoU's with the rectangle we just found. We repeat the process until all rectangles have either been found and highlighted or darkened due to high IoU value.

The exact details of non-max suppression is as follows. For each 8 dimensional output prediction, we would discard all boxes with $p_c \leq 0.6$. While there are any remaining boxes, we pick the box with the largest $p_c$. Output that as a prediction.

Then, we discard any remaining box with $IoU \geq 0.5$ with the box output in the previous step.

## 11.4 Anchor Boxes

What if our grid cells want to detect multiple objects? We can use the idea of **anchor boxes**.

Suppose we have midpoints of two objects in the same grid cell. We can predefine two different shapes of anchor boxes and associate two predictions with different anchor boxes. In this case, we define the label $y$ with two sets of $p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3$ values, which gives us a 16 dimensional vector. With two anchor cells, each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

In practice, two objects rarely happen to have their midpoints in the same grid cell, especially when we use $19 \times 19$ grids. A better motivation for anchor boxes is that it allows our learning algorithm to specialize better. To choose anchor boxes, people do this by hand and choose 5-10 anchor box shapes that span a variety of shapes that seems to cover the types of objects we detect. One better way to do this is to use a K-means algorithm to group together two types of object shapes we tend to get.

## 11.5 Region Proposals

If we look at the object detection literature, there is a set of ideas called **region proposals** that's been very influential in computer vision.

Using the standard sliding windows detection is pretty computationally expensive. We can run the algorithm convolutionally, but it classfies a lot of regions where there is clearly no object. **R-CNN (Regions with CNNs** tries to pick just a few regions that make sense to run the ConvNet classifiers on. The way to perform region proposals is to run a **segmentation algorithm** to figure out what could be objects. We find about 2,000 blobs and place bounding boxes around about these 2,000 blobs and run classifier on just these blobs. This could be a much smaller number of positions where we run the ConvNet classifier.

It turns out R-CNN is still quite slow. There is a line of work to explore how to speed up this algorithm. These are summarized below:

- Basic R-CNN: Propose regions. Classify proposed regions one at a time. Output label + bounding box. However, this is quite slow.

- Fast R-CNN: Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions. The clustering step to propose regions is still quite slow.

- Faster R-CNN: Use convolutional network to propose regions.

## 11.6 Semantic Segmentation with U-Net

The goal of **semantic segmentation** is to draw a careful outline around the object that is detected so we know exactly which pixels belong to the object and which do not. This is useful for many commercial applications as well today.

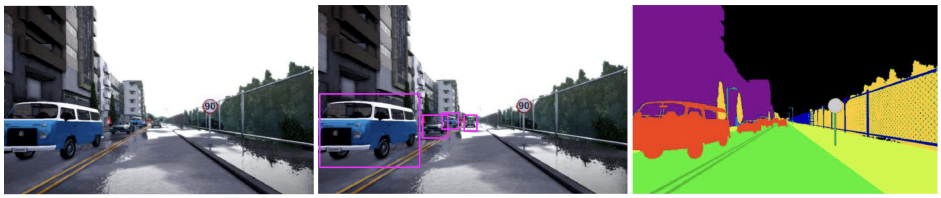



Figure 11.5: Object Detection vs. Semantic Segmentation

Semantic segmentation attempts to label every pixel to some class. One of the uses is that this algorithm is used by many self-driving car teams to figure out exactly which pixels are safe to drive over.

The use of semantic segmentation in medical imaging makes it easier to spot irregularities and diagnose serious diseases and also help surgeons with planning out surgeries. The algorithm to generate the result is called the **U-Net**.

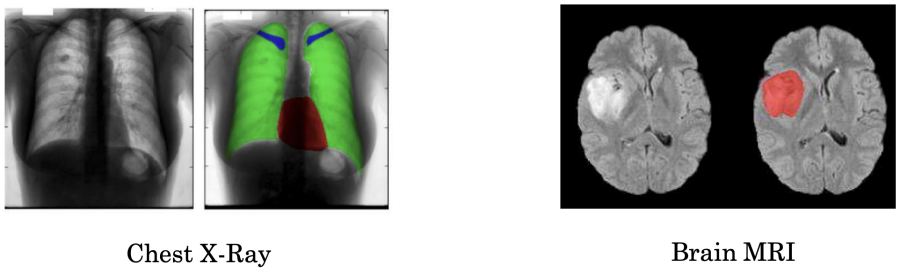



Figure 11.6: Semantic Segmentation in Medical Imaging

Suppose we want to segment out a car in the image. In this case, we have two class labels – 1 for car and 0 for not car. The job of segmentation algorithm is to output 1 or 0 for each pixel of the image. Alternatively, if we want to segment the image more finely, we can have three classes – 1 for car, 2 for building, and 3 for ground or road.

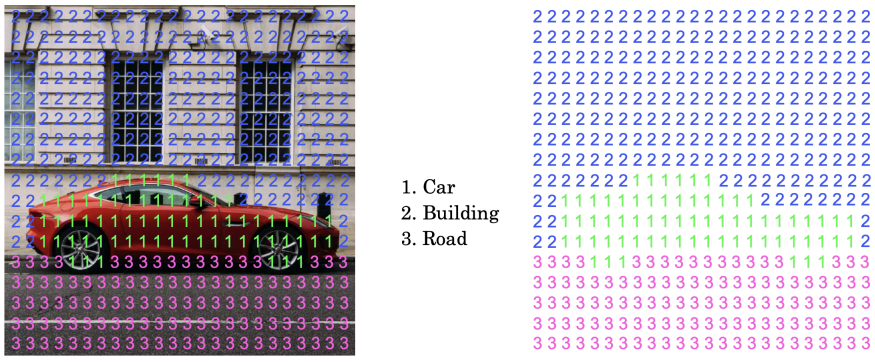



Figure 11.7: U-Net Segmentation Map

Now, instead of generating a single class label or a class label and coordinates needed to specify bounding boxes, the neural network, U-Net, needs to generate a whole matrix of labels. One key step of semantic segmentation is that whereas the dimensions of the images are generally getting smaller as we go deeper into the network, the dimensions need to get bigger so we can gradually blow it back up to a full-size image that we want to output.

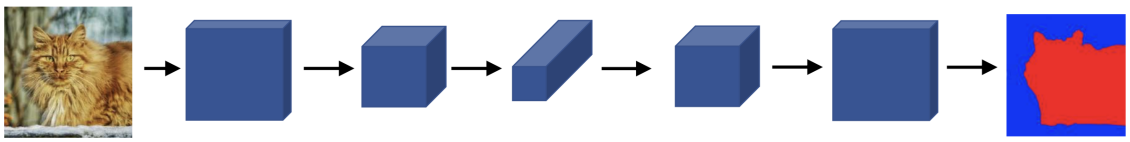


Figure 11.8: Brief UNet Outline

## 11.7 Transpose Convolutions

Transpose convolution is a key part of the U-Net architecture and it blows up the output dimensions.

Suppose we take a $2 \times 2$ input $\begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}$. We now convolve it with a, say, $3 \times 3$ filter $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix}$ and end up with a $4 \times 4$ output. Suppose further that we have padding $p = 1$ on the output and stride $s = 2$. In the transpose convolution, we place the filter on the output, instead of the input. In this case, we first take the top-left element 2 of the input matrix and have the filter multiplied with it. Then, we paste the result in the top-left $3 \times 3$ region of output. Then, we ignore the padding region of the output and throw in remaining four values to the output. This gives us

$$\begin{bmatrix} 0 & 2 & * & * \\ 4 & 2 & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}.$$

Then, we have the filter multiplied by the top-right element: 1. With the stride of 2, there are some places where we have an overlap between the region we have pasted and the region we want to paste. In this case, we add overlap values together, which results in the following output:

$$\begin{bmatrix} 0 & 2+2 & 0 & 1 \\ 4 & 2+0 & 2 & 1 \\ * & * & * & * \\ * & * & * & * \end{bmatrix} = \begin{bmatrix} 0 & 4 & 0 & 1 \\ 4 & 2 & 2 & 1 \\ * & * & * & * \\ * & * & * & * \end{bmatrix}.$$

By repeating the process, we get the following $4 \times 4$ output:

$$\begin{bmatrix} 0 & 4 & 0 & 1 \\ 10 & 7 & 6 & 3 \\ 0 & 7 & 0 & 2 \\ 6 & 3 & 4 & 2 \end{bmatrix}.$$

Here is another example of transpose convolution from the book *Dive into Deep Learning*:

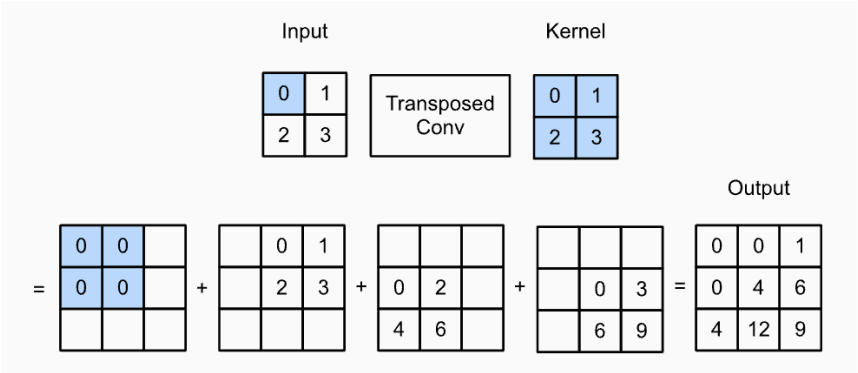



Figure 11.9: Transpose Convolution Example

## 11.8 U-Net Architecture

In U-Net architecture, we use skip connection from the earlier layers to deeper layers. Therefore, the earlier blocks of activations is copied directly to the later block. Two types of information are useful the deep, or close to output, layers. Lower resolution, high level contextual information is obtained from previous layers of the deep layer, but a very detailed, fine grained spatial information is missing, as the set of activations have lower resolution. Therefore, early layers provide high-resolution, low level feature information where it could capture, for every pixel position, how much fairy stuff is there for each pixel.

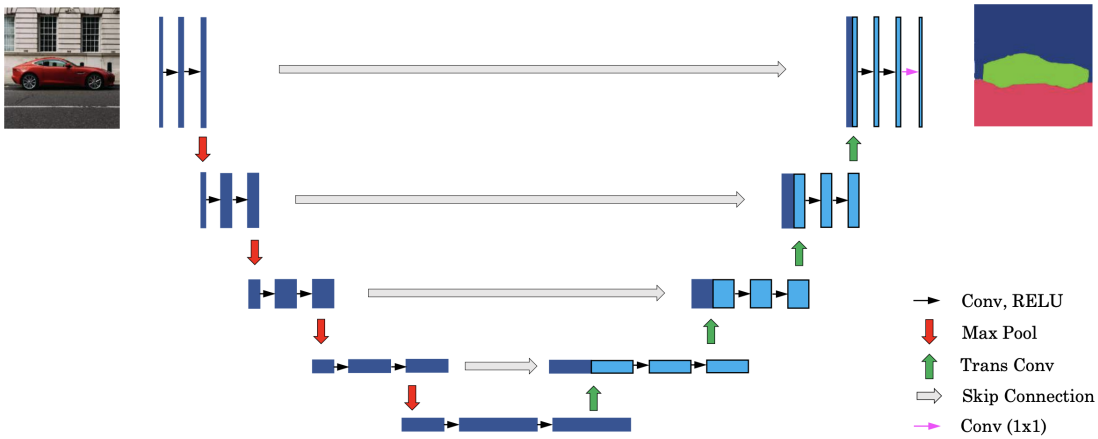



Figure 11.10: U-Net Architecture

The input of U-Net is a $h \times w \times 3$ RGB image. The first part uses a normal feedforward CONV layers where we increase the number of channels by a little bit. Then, we use MAX-POOL layer to reduce the height and width. Then, we have two more normal feedforward CONV layers, followed by a MAX-POOL layer. We repeat the process again and obtain the result for the first half.

Now, the height and width are very small, so we start by applying transpose convolution layers where we do not increase the height and width but reduce the

number of channels and add in skip connection by copying the set of activations (blue part in the figure). We apply a couple more regular CONV layers with ReLU activation function, followed by a transpose convolution wherewe increase the height and width of the image. Again, we apply skip connection by copying over directly. Then, we repeat the process twice more, followed by two normal feedforward CONV layers. Finally, we use a $1 \times 1$ CONV layer to map the output to the segmentation map, which has the dimensions $h \times w \times n_{classes}$.

# Chapter 12

# Special Applications

The objectives of this chapter are

- Differentiate between face recognition and face verification
- Implement one-shot learning to solve a face recognition problem
- Apply the triplet loss function to learn a network's parameters in the context of face recognition
- Explain how to pose face recognition as a binary classification problem
- Map face images into 128-dimensional encodings using a pretrained model
- Perform face verification and face recognition with these encodings
- Implement the Neural Style Transfer algorithm
- Generate novel artistic images using Neural Style Transfer
- Define the style cost function for Neural Style Transfer
- Define the content cost function for Neural Style Transfer

## 12.1 Face Recognition

In the face recognition literature, people often talk about face verification and face recognition. For face verification, we input an image, as well as name/ID of a person, the job of the system is to verify if the input image is that of the claimed person. This is sometimes called **one-to-one problem**. Face recognition problem is much harder than the verification problem. In this case, given an input image and a database of K persons, we output ID if the image is any of the K persons (or "not" recognize). Suppose we have a verification system that is 99% accurate. To implement a face recognition system, we would need a verification task that has probably 99.9% accurate for the system to work well.

### 12.1.1 One Shot Learning

For most face recognition applications, we need to be able to recognize a person given just one single image, or given just one example of that person's face. Historically, deep learning algorithms don't work well if we have only one training example.

Suppose we have four faces to recognize. One solution to this is to train a ConvNet using a softmax activation to give 5 outputs. However, if we have an additional face to recognize, which means the softmax needs to give 6 outputs, we need to retrain the ConvNet. This does not seem to be a good approach. Instead, we learn a "similarity" function, where

$$d(img1, img2) = \text{degree of difference between images}$$

If two images are of the same person, we want this to output a small number. If two images come from two different people, we want this to output a large number. More specifically, if $d(img1, img2) \leq \tau$, we predict that two images are of the same person. Otherwise, if $d(img1, img2) > \tau$, we predict that two images are from different people. This is how we address the face verification problem. To use this for face recognition tasks, given a new picture, we use function $d$ to compute the difference between the image and all other images in the database. If there is no match, $d$, hopefully, would output large numbers for all pairwise comparisons and we say this person in the additional image is not in our database. This helps with one shot learning problem.

### 12.1.2 Siamese Network

The job of the function $d$ is to input two faces and tell how different they are. A good way to do this is to use a Siamese Network.

For an input image $x^{(1)}$, suppose we have a 128 dimensional hidden layer in the ConvNet prior to softmax activation. We denote this as $f(x^{(1)})$, which is seen as an encoding of $x^{(1)}$. For some image $x^{(2)}$, we would get another 128 numbers represented as $f(x^{(1)})$, which encodes the second picture. If we believe the encoding is a good representation of the images, we can define

$$d(x^{(1)}, x^{(2)}) = \|f(x^{(1)}) - f(x^{(2)})\|_2^2.$$

More formally, the ConvNet gives an encoding $f(x^{(i)})$ for each input image $x^{(1)}$. We learn parameters so that

- If $x^{(i)}, x^{(j)}$ are the same person, $\|f(x^{(i)}) - f(x^{(j)})\|_2^2$ is small.
- If $x^{(i)}, x^{(j)}$ are different people, $\|f(x^{(i)}) - f(x^{(j)})\|_2^2$ is large.

### 12.1.3 Triplet Loss

One way to learn the parameters of the neural network, so that it gives us a good encoding for our picture of faces, is to define and apply gradient descent on the triplet loss function.

If two images are of the same person, we want the encoding to be similar. Otherwise, we want very different encodings of the two images. In the triplet loss, when we look at one **anchor image**, we want the distance between anchor image and

**positive image** to be similar, whereas we want anchor image and **negative image** to be much further apart. The name triplet comes because we always look at three images at a time – Anchor (A), Positive (P), and Negative (N) images. More formally, we want

$$d(A, P) = \|f(A) - f(P)\|_2^2 \leq \|f(A) - f(N)\|_2^2 = d(A, N).$$

By moving the terms, we obtain

$$\|f(A) - f(N)\|_2^2 - \|f(A) - f(P)\|_2^2 \leq 0$$

To make sure neural network does not just output zeros for all encodings, which would trivially satisfy the inequality, we modify the objective by adding an margin $\alpha$, which is another hyperparameter, as follows:

$$\begin{aligned} &\|f(A) - f(P)\|_2^2 + \alpha \leq \|f(A) - f(N)\|_2^2 \\ \Rightarrow &\|f(A) - f(N)\|_2^2 - \|f(A) - f(P)\|_2^2 + \alpha \leq 0 \end{aligned}$$

Now, we formally define the triplet loss function. Given 3 images A,P,N, we define the loss as follows:

$$\mathcal{L}(A, P, N) = \max\{\|f(A) - f(N)\|_2^2 - \|f(A) - f(P)\|_2^2 + \alpha, 0\}.$$

The effect of max is that so long as the whole term is less than 0, the loss we have would be 0. The neural network does not care how negative it is. The overall cost function for m training examples is given by

$$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)}).$$

If we have 10K pictures of 1K people in the training set, we can take 10K pictures to select triplets. Then, we use gradient descent on the cost function defined above for training. For the purpose of training our system, we do need a dataset where we have multiple pictures of the same person. One problem is that during training, if A,P,N are chosen randomly, $d(A, P) + \alpha \leq d(A, N)$ is easily satisfied. Therefore, the neural network will not learn much from the training process. To construct the training set, we want to choose triplets that are hard to train on. Specifically, we want to choose triplets such that

$$d(A, P) \approx d(A, N).$$

Choosing triplets this way, the neural network will push $d(A, P)$ down and $d(A, N)$ up, so there is at least a margin $\alpha$ between two sides. The effect of choosing triplets like this is that it increases the computational efficiency of the learning algorithm. If we choose triplets randomly, many training examples would be quite easy and the neural network ends up learning not much.

### 12.1.4 Face Verification and Binary Classification

Besides the using triplet loss to learn the parameters for face recognition task, face recognition can also be seen as a straight binary classification problem.

Another way to train the neural network is to take a pair of Siamese networks and have them both compute the 128 dimensional, or high dimensional, embeddings. Then, we input the embeddings to a logistic regression unit to then make a prediction, where the target output would be 1 if both images are the same person and 0 if two images are two different people. This is the alternative to the triplet loss. In this case, we can represent $\hat{y}$ as follows:

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b\right),$$

where the subscript $k$ means we select out the kth component of the embedding vector.

There are a few other variations of the way we compute $\hat{y}$. Another formula could be

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k} + b\right),$$

which is sometimes called the $\chi^2$ **similarity**.

Since the parameters of the Siamese Network are the same, one computational trick to help deployment is that we can precompute the encoding, so we do not need to store all the input images and compute the encoding every single time.

## 12.2 Nerual Style Transfer

For neural style transfer, we have a content image (C), a style image (S), and generate a image of the content image with the style of the style image.

Figure 12.1: Hoover Tower with Nerual Style Transfer

### 12.2.1 What are Deep ConvNets Learning?

Suppose we have trained a ConvNet and we want to visualize what the hidden units in different layers are computing. Here is what we can do.

Pick a unit in layer 1. Find the nine image patches that maximize the unit's activation. We can repeat for other units.

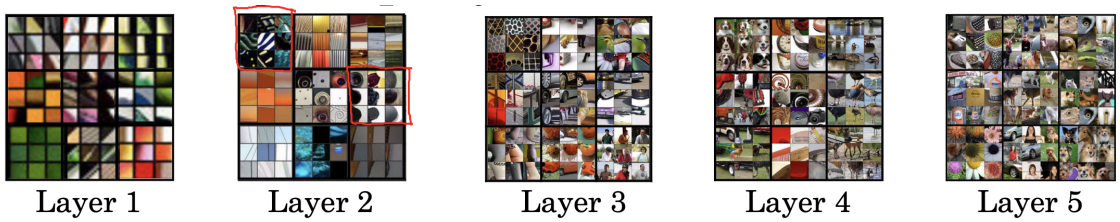



Figure 12.2: Visualizing Deep Layers

### 12.2.2 Cost Function

To build a Neural Style Transfer system, we would need to define a cost function for the generated image. By minimizing this cost function, we can generate the image that we want.

Suppose we use C to denote Content image, S to denote Style image, and G to denote Generated image. Then, we define two parts to the cost functions:

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G).$$

We would initialize $G$ randomly where $G$ as $100 \times 100 \times 3$ dimensional image. Then, we use gradient descent to minimize $J(G)$ as

$$G := G - \frac{\partial}{\partial G} J(G),$$

which we end up updating pixel values of the initialized image.

### 12.2.3 Content Cost Function

Say we use hidden layer $l$ to compute content cost. If $l$ is very small, where hidden layer is in the early part of the network, then it will force the generated image pixel values to be similar to the content image. If we use deep layer, the generated image becomes more flexible. In practice, we choose layer $l$ somewhere in between of shallow and deep layers. We can use pre-trained ConvNet (e.g. VGG network). Let $a^{[l](c)}$ and $a^{[l](G)}$ be the activation of layer $l$ on the images. If $a^{[l](c)}$ and $a^{[l](G)}$ are similar, both images have similar content. We will define the cost function as follows:

$$J_{content}(C, G) = \frac{1}{2} \| a^{[l](c)} - a^{[l](G)} \|_2^2,$$

where $\frac{1}{2}$ is the normalization constant. This could be adjusted by the hyperparameter $\alpha$ as well. This incentivizes the algorithm to find an image $G$ so that these hidden layer activations are similar.

### 12.2.4 Style Cost Function

Say we use layer $l$'s activation to measure style. We can define style as correlation between activations across channels and ask how correlated are the activations across different channels. More formally, given an input image, we can compute a **style matrix**, where $a^{[l]}_{i,j,k}$ = activation at $(i, j, k)$. $G^{[l]}$ is $n^{[l]}_c \times n^{[l]}_c$ and $n_c$ is the number of channels. In particular, $G^{[l]}_{kk'}$ measures how correlated are the activations in channel $k$ compared to $k'$, where $k = 1, \ldots, n^{[l]}_c$. $G^{[l]}_{kk'}(S)$ is for style image, whereas $G^{[l](G)}_{kk'}$ is for the generated image.

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n^{[l]}_H} \sum_{j=1}^{n^{[l]}_W} a^{[l](S)}_{i,j,k} a^{[l](S)}_{i,j,k'}.$$

Notice that if both of these activations tend to be correlated, then $G_{kk'}$ will be large. Otherwise, $G_{kk'}$ might be small. We would do the same for the generated image as follows:

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n^{[l]}_H} \sum_{j=1}^{n^{[l]}_W} a^{[l](G)}_{i,j,k} a^{[l](G)}_{i,j,k'}.$$

In linear algebra, these are called the **gram matrix**.

Finally, the style cost function on layer $l$ between S and G could be given as follows:

$$\begin{aligned} J^{[l]}_{style}(S, G) &= \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \|G^{[l](S)} - G^{[l](G)}\|^2_F \\ &= \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \sum_k \sum_{k'} \left(G^{[l](S)} - G^{[l](G)}\right)^2 \end{aligned}$$

It turns out we get more visually pleasing result if we use a style cost function from muultiple different layers as follows:

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S, G),$$

where $\lambda^{[l]}$ is a style of weighted hyperparameters.

### 12.2.5 1D and 3D Generalization

Many ideas we have shown not just apply to 2D images but also apply to 1D data, as well as 3D data.

For example, EKG signal, or electrocardiogram, is a 1D data, where we place a electrode over our chest to measure the voltages that vary across the chest as our heart beats. This is a time-series data showing the voltage at each instant in time. We might have 13 dimensional input. In this case, we can convolve it with a 5 dimensional filter. Similar to 2D data, we apply the 5 dimensional filter to a lot of different positions throughout the 1D signal. This gives us a 10 dimensional output. If we have 16 such filters, then our output would be $10 \times 16$ dimensional. Then, if we have another set of 32 $5 \times 16$ filters, this would give us a $6 \times 32$ dimensional output.

Suppose we have a 3D input volumn of numbers. An example is CT scan, which takes different slices through our body. We can look at different slices of the human torso to see how they look. Now, when we have a 3D input volumn, say $14 \times 14 \times 14 \times 1$, where 1 is the number of channels, we can apply 3D filters. If we concolve it with 16 $5 \times 5 \times 5 \times 1$ filters, then this would give us a $10 \times 10 \times 10 \times 16$ volumn. If we then convolve the output with 32 $5 \times 5 \times 5 \times 16$ filters, then the output would be $6 \times 6 \times 6 \times 32$ dimensional output, where we have $6 \times 6 \times 6$ volumn across 32 channels.

Another example of data that we can treat as a 3D volumn is movie data, where different slices could be different slices in time through a movie. We can use this to detect motion or people taking actions in movies.

# Part V

# Sequence Models

# Chapter 13

# Recurrent Neural Networks

- Define notation for building sequence models
- Describe the architecture of a basic RNN
- Identify the main components of an LSTM
- Implement backpropagation through time for a basic RNN and an LSTM
- Give examples of several types of RNN
- Build a character-level text generation model using an RNN
- Store text data for processing using an RNN
- Sample novel sequences in an RNN
- Explain the vanishing/exploding gradient problem in RNNs
- Apply gradient clipping as a solution for exploding gradients
- Describe the architecture of a GRU
- Use a bidirectional RNN to take information from two points of a sequence
- Stack multiple RNNs on top of each other to create a deep RNN
- Use the flexible Functional API to create complex models
- Generate your own jazz music with deep learning
- Apply an LSTM to a music generation task

Models like recurrent neural networks (RNN) have transformed speech recognition, natural language processing, and other areas. There are a few examples where sequence model can be useful.

- In speech recognition, we are given an audio clip and map it to text transcript, both of which are sequence data – audio clip plays out over time and text transcript is a sequence of words.

- Music generation is another example, but only the output is a sequence. The input can be an empty set or a single integer representing the genre of the music.

- In sentiment classification, the input is a sequence of phrase. Then we predict how many starts the review would be.

- Sequence model is also useful for DNA sequence analysis, where the input is a DNA sequence and we output a labeled subsequence that corresponds to a protein.

- In machine translation, we are given a input sentence and output the translation into a different language.

- In video activity recognition, we are given a sequence of video frames and asked to recognize the activity.

- In the name entity recognition, we are given a sentence and asked to identify the people in that sentence.

All these examples can be addressed as supervised learning with labeled data $(x, y)$ as the training set, but there are many types of sequence models.

## 13.1 Notation

As a motivating example, suppose we want to build a sequence model to input a sentence like

"Harry Potter and Hermione Granger invented a new spell."

We want a sequence model to tell us where are people's names in the sentence. This is called **name entity recognition** and this is used by search engines, which can be used to find people's names, companies' names, times, locations, countries' names, currency names, and so on. We can have a model that outputs $y$ that has one 0/1 output for each input word, which tells if each input word is a part of a person's name. We can have a more complicated representation that includes the start and end of each person's name. In our example, there are nine sets of features to represent the nine words $x^{<1>}, x^{<2>}, \ldots, x^{<t>}, \ldots, x^{<9>}$. Similarly, we can denote the output as $y^{<1>}, \ldots, y^{<9>}$. Additionally, we use $T_x$ to denote the length of the input sequence and $T_y$ to denote the length of the output sequence. In the example, $T_x = T_y = 9$. But $T_x$ and $T_y$ can be different.

To denote the $t$th element in the sequence of training example $i$, we use $x^{(i)<t>}$. $T_x^{(i)}$ is used to denote the input sequence length for training example $i$. Similarly, $y^{(i)<t>}$ means the $t$th element in the output sequence of the $i$th training example and $T_y^{(i)}$ would be the length of the output sequence in the $i$th training example.

Next, we show the way to represent individual words in the sentence. First, we

need to come up with a dictionary, a list of words that we will use.

$$Vocabulary: \begin{bmatrix} \text{a}(1) \\ \text{aaron}(2) \\ \vdots \\ \text{and}(367) \\ \vdots \\ \text{harry}(4,075) \\ \vdots \\ \text{potter}(6,830) \\ \vdots \\ \text{zulu}(10,000) \end{bmatrix}.$$

Now we have a dictionary of size 10K. This is quite small by modern NLP applications. For commercial applications, dictionary sizes of 30K to 50K are more common and 100K is not uncommon. Some of the large Internet companies will use dictionary sizes that are maybe a million words or even bigger. One way to build the dictionary would be looping through the training set and find the top 10K occurring words. We can also look through some of the online dictionaries that tell us the most common 10K words in the English Language. We can use one-hot representation to represent each of these words. For example, $x^{<1>}$ would be a vector of all zeros except for the 1 in position 4075 – the position of "harry" in the dictionary. We can represent other words similarly.

If we encounter a word that is not in our vocabulary, then we create a new token, or a new fake word, called **unknown word** to represent words not in the vocabulary.

## 13.2 Recurrent Neural Network Model

We can use the standard neural network to do the task in previous section, where we have 9 inputs and have 9 output values. This turns out not to work well because

- Inputs and outputs can be different lengths in different examples. If each sentence has a max length, maybe we can pad zeros, but this still does not seem like a good representation.
- Does not share featrues learned across different positions of text.

Recurrent neural network resolves both of the disadvantages. We have the first word $x^{<1>}$ and feed it into a neural network layer and have the neural network predict the output $\hat{y}^{<1>}$. Then, the neural network read the second word. The hidden layer not only gets $x^{<2>}$ but also the activation value $a^{<1>}$ from step 1. We repeat until we reach the end of the sentence. For the first hidden layer, we would have some made-up activation $a^{<0>}$, usually a vector of zeros. The parameters governing the connection from $x^{<1>}$ to the hidden layer would be some set of parameters $W_{ax}$, where we use the same parameters $W_{ax}$ for every time step. The activation connections. would be governed by some set of parameters $W_{aa}$, which is used for every time step. Similarly, we use $W_{ya}$ to govern the output predictions.

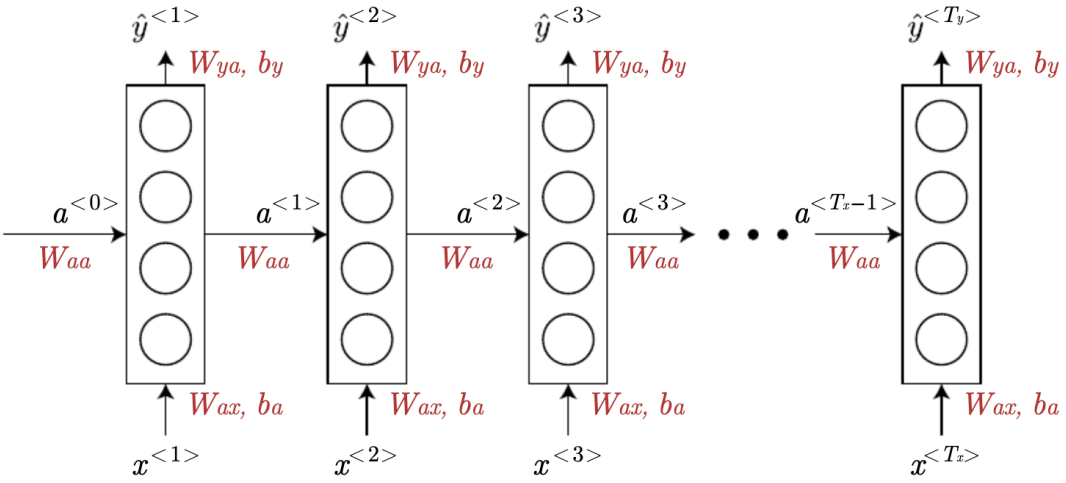



Figure 13.1: Recurrent Neural Network Architecture from Matt Deitke

The weakness. of this RNN is that it only uses the information that is earlier in the sequence to make a prediction. When making prediction for $\hat{y}^{<3>}$, we do not use information $x^{<4>}, x^{<5>}, \ldots$. This is a problem. If we are given a sentence

He said, "Teddy Roosevelt was a great president."

In order to decide whether or not the word *Teddy* is a part of a person's name, it would be useful to know not just the information from the first two words but to know information from later words. The sentence could have been

He said, "Teddy bears are on sale!"

Given only the first two words, it is not possible to know for sure whether the word *Teddy* is a part of a person's name. This will be addressed when we show **Bidirectional Recurrent Neural Network (BRNN)** later.

The forward propagation could be represented, where $a^{<0>} = \vec{0}$, as follows:

$$a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$$
$$\hat{y}^{<1>} = g(W_{ya}a^{<1>} + b_y)$$

where the activation function is often tanh and sometimes ReLU for $a^{<1>}$ and sigmoid is often used for $\hat{y}^{<1>}$ is we have a binary classification task or a softmax for multi-class classification.

More generally, at time step $t$,

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$
$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

We can simplify the notation as

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$
$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

where $W_a = \begin{bmatrix} W_{aa} & W_{ax} \end{bmatrix}$, $[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$

If $a^{<t-1>}$ is 100 dimensional and $x^{<t>}$ is 10,000 dimensional, then $W_{aa}$ has shape $(100, 100)$ and $W_{ax}$ has shape $(100, 10000)$. Therefore, $W_a$ is $(100, 10100)$ a dimensional matrix and $[a^{<t-1>}, x^{<t>}]$ is a 10100 dimensional vector.

## 13.3 Backpropagation Through Time

As usual, when we implement backprop in one of the programming frameworks, the programming framework will automatically take care of backpropagation. However, it is still useful to have a rough sense of how backprop works in RNN.

We can define an element-wise loss (cross entropy loss) at one time step as follows:

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log(\hat{y}^{<t>}) - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>}).$$

Therefore, the loss for the entire sequence is given by

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$

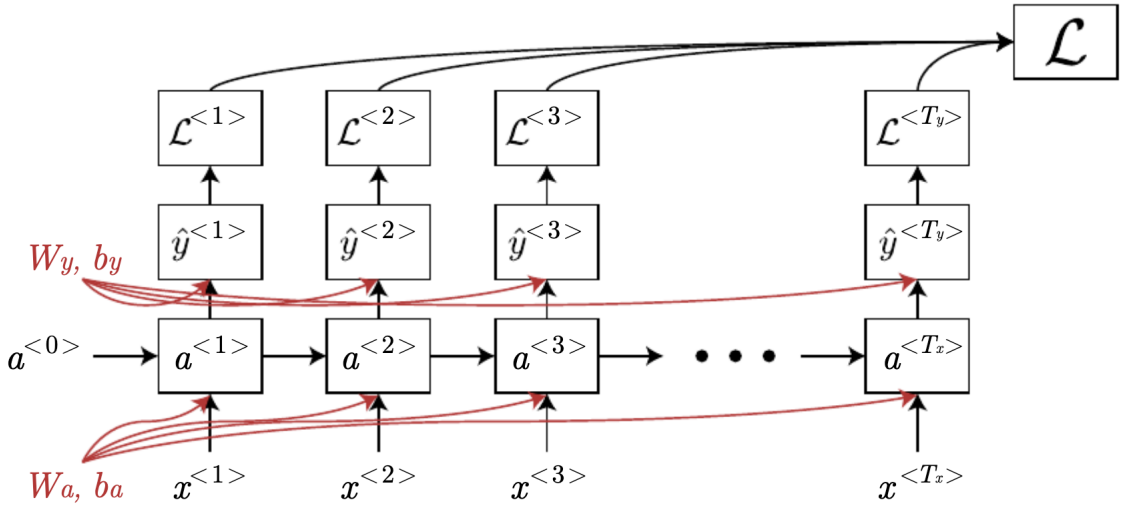



Figure 13.2: Recurrent Neural Network Computation Graph

Backpropagation is doing computations, or passing messages, in the opposite direction. This allows us to compute all the quantities and update parameters using gradient descent.

## 13.4 Different Types of RNNs

In the previous example, we've shown the **many-to-many** RNN architecture for problems where the input length and output length are the same. However, there are also some problems where the input length and output length are different. It turns out we can modify the RNN architecture to address the problem.

Suppose we want to address sentiment classification. Here, input might be a piece of text sequence and output might be 0/1 representing positive/negative review or a number from 1 to 5. Rather than having outputs at each time step, we can have the RNN read in the entire sentence and output one single $y$ at the end. This is **many-to-one** RNN architecture. There is also **one-to-one** RNN architecture. This is less interesting because it is more like the standard neural network. We can also have **one-to-many** RNN architecture used for tasks like music generation.

We usually take the synthesized output and feed it to the next layer, along with the activation. For tasks like machine translation, which also uses **many-to-many** RNN architecture, the input length and output length are different. In this case, we first read in the entire input, which constitute the encoder part. Then, we have the neural network output translation. This is the decoder part of the entire neural network.

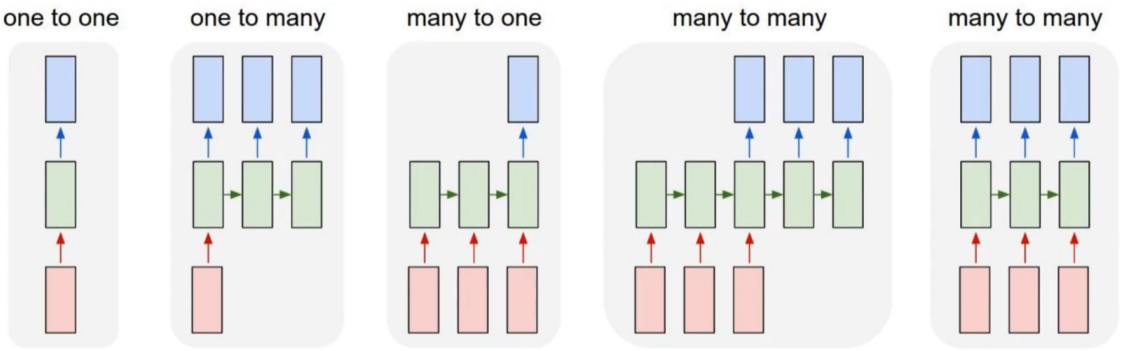



Figure 13.3: Different Types of RNN Architectures

## 13.5 Language Model and Sequence Generation

Language modeling is one of the most basic and important task in natural language processing. It is also one tasks that RNNs do very well.

Suppose we are building a speech recognition system and we hear the sentence

The apply and *pair* salad or The apple and *pear* salad

These two sentences sound exactly the same, but the second is more likely. In this case,

$$P(\text{The apply and pair salad}) = 3.2 \times 10^{-13}$$
$$P(\text{The apply and pear salad}) = 5.7 \times 10^{-10}$$

Since the second sentence is more likely, the speech recognition system would pick the second choice. The job of a language model is that given a sentence, it tells the probability of that particular sentence $P(y^{<1>}, y^{<2>}, \ldots, ^{<T_y>})$.

To build a language model, we first need a training set comprising of large corpus (body/set in NLP term) of text. Given a sentence, we would tokenize the sentence by mapping words using vocabulary. We also add an extra token called <**EOS**> to help us figure out the end of the text. If a word is not in our vocabulary, then we use a unique token <**UNK**> to represent the word.

At time 0, we compute some activation $a^{<1>}$ as some function of the input $x^{<1>}$. By convention, both $a^{<0>}$ and $x^{<1>}$ are set to $\vec{0}$. $a^{<1>}$ will make a softmax prediction to try to figure out the probability of first word as $\hat{y}^{<1>}$. Then, the RNN steps forward and have some activation $a^{<2>}$, which tries to figure out the probability of second word, given the correct first word. This implies that $x^{<2>} = y^{<1>}$. We go on to the next step of RNN, we are given the first two words. Therefore, $x^{<3>} = y^{<2>}$. We repeat until the end where we compute the chance of <EOS> given the whole

sentence. Hopefully the RNN would predict there is a high chance of <EOS> at the end.

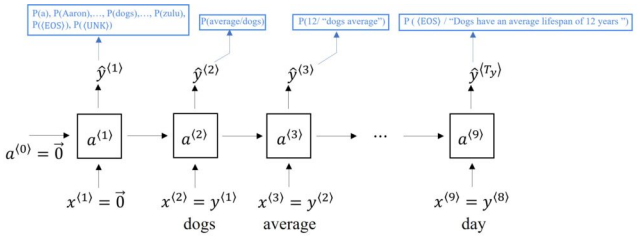



Figure 13.4: RNN Model for a Language Model

To train the network, we define the loss function as follows:

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log(\hat{y}_i^{<t>}),$$

where the cost function is given by $\mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$.

## 13.6 Sampling Novel Sequences

After we train a sequence model, one of the ways we can informally get a sense of what it has learned is to have a sample novel sequences.

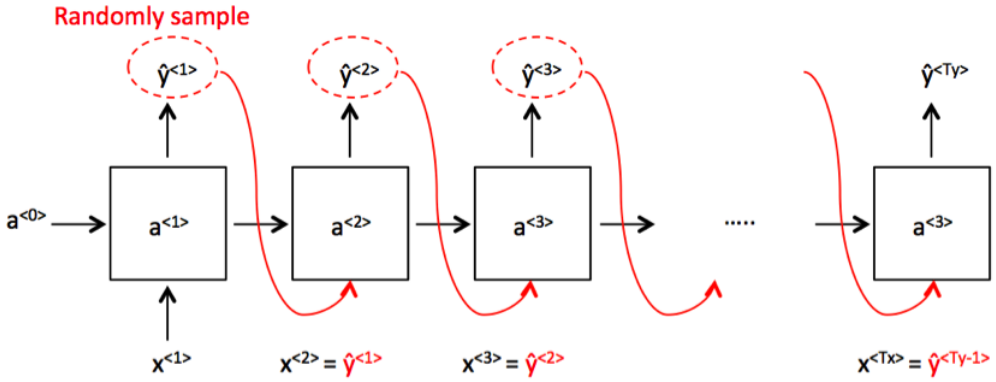



Figure 13.5: Sampling Novel Sequence

By feeding in $a^{<0>}, x^{<1>}$ at first time step, we would have some softmax probability over possible outputs. Then, we randomly sample according to the softmax probability distribution using *np.random.choice*. Then, we take the $\hat{y}^{<1>}$ we just sampled and pass it as the input of the second time step and generate another softmax probability distribution $\hat{y}^{<2>}$, which corresponds to the probability of possible second word given the first word we have sampled. We keep going until we reach the last time step. If the <EOS> token is a part of our vocabulary, we can keep sampling until we generate the <EOS> token. Alternatively, if <EOS> is not in the vocabulary, we can decide to sample some specific number of words, say 20 or 100. This particular procedure will sometimes generate an unknown word token. If

we want to avoid this, we can reject any sample that comes out as the unknown word.

So far, we are building a word-level RNN. Depending on the application, we can also build a character-level language model. In this case, the vocabulary will just be lower-case alphabets, upper-case alphabets, space, punctuation, and numbers. In this case, our sequence $y^{<1>}, y^{<2>}, \ldots$ would be the individual characters in the training data.

By using the character-level RNN, we do not have to worry about unknown word tokens. This is able to assign a sequence like *Mau* a non-zero probability, whereas the word-level RNN may assign *Mau* to the unknown word token. However, the main disadvantage is that we end up with much longer sequences. This is not as good at capturing long range dependencies between how the early parts of a sentence affect the later parts of a sentence. Also, it is more computationally expensive to train. For most part, word-level language model are still used, but as computers get faster, there are more and more applications where people start to look at character-level models.

## 13.7 Vanishing Gradients with RNNs

One of the problems of the basic RNN model is that it runs into vanishing gradient problems.

Suppose we have the following two sentences

The **cat**, which already ate ..., **was** full.

The **cats**, which already ate ..., **were** full.

This is one example when language can have very long-term dependencies where one word much earlier can affect what needs to come much later in a sentence. However, the basic RNN we have shown is not very good cat capturing long-term dependencies. Before, when we talk about deep neural networks, we said that gradient from the output $\hat{y}$ would have a very hard time propagate back to affect the weights of earlier layers. In the context of RNN, we have the same vanishing gradient problem for the output $\hat{y}^{<T_y>}$ at later time step to affect the computations that are earlier. In English, the words in the middle could be arbitrarily long, which means we need to memorize singular/plural for a long time.

We also talked about exploding gradient in deep neural networks. It turns out the vanishing gradient tends to be the biggest problem with training RNNs. Exploding gradient is easier to spot because the parameters have blown up, which we might see results of a numerical overflow like *NaN*. One solution to exploding gradient is to apply **gradient clipping** – look at the gradient vectors and if some vectors are greater than some threshold, we rescale these gradient vectors.

## 13.8 Gated Reccurent Unit (GRU)

Gated Recurrent Unit (GRU) is a modified RNN hidden layer that makes it much better at capturing long-range connections and helps a lot with the vanishing gradient problem.

First, we see a simplified GRU. As we read a sentence from left to right, a GRU unit has a new variable called $c$, which stands for **memory cell**. It provide a bit of memory to remember, for example, whether *cat* is singular or plural, where $c^{<t>} = a^{<t>}$. At every time step, we consider overwriting the memory cell with $\tilde{c}^{<t>}$ where

$$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c).$$

The **update gate** of GRU $\Gamma_u \in \{0, 1\}$ is given by

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u).$$

For most of the possible ranges of the input, the sigmoid function is either very close to 0 or very close to 1, so we can think of $\Gamma_u$ as either 0 or 1.

We were thinking of updating $c$ with $\tilde{c}$. And then the gate will decide whether or not to update. The key equation of GRU is given as follows:

$$c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>}.$$

For most of the word values in the middle of two connected words, $\Gamma_u = 0$.

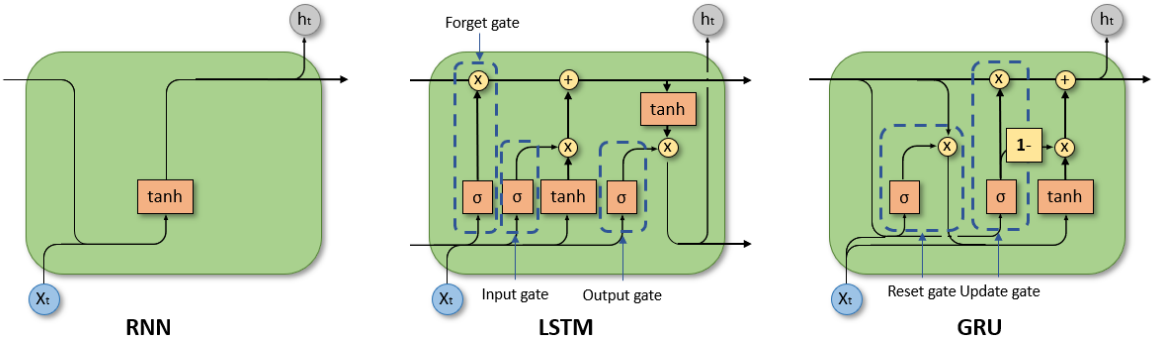



Figure 13.6: RNN, LSTM, GRN Models

Because $\Gamma_u$ can be as small as 0.0000001 or even smaller, the value of $c^{<t>}$ is maintained even across many time steps. Therefore, we suffer less from the vanishing gradient problem.

The full GRU unit is given by

$$\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[\Gamma_r \times c^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_u &= \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \\
\Gamma_r &= \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \\
c^{<t>} &= \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>}
\end{aligned}$$

where $\Gamma_r$ tells the relevance between $c^{<t-1>}$ and $c^{<t>}$.

Over the years, researchers have experimented with many different possible versions of such units to capture long-term memory and address vanishing gradient issue. GRU is one of the most commonly used versions that's robust and useful for many different problems.

## 13.9 Long Short Term Memory (LSTM)

Another type of unit that allows us to learn very long range connections in a sequence is **LSTM**, or Long Short Term Memory units. This is even more powerful than GRU.

The LSTM is a slightly more powerful and more general version of the GRU. The equations are given as follows:

$$\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_u &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \\
\Gamma_f &= \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \\
\Gamma_o &= \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \\
c^{<t>} &= \Gamma_u \times \tilde{c}^{<t>} + \Gamma_f \times c^{<t-1>} \\
a^{<t>} &= \Gamma_o \times \tanh(c^{<t>})
\end{aligned}$$

where we no longer have the case $a^{<t>} = c^{<t>}$. $\Gamma_u$ stands for update gate, $\Gamma_f$ stands for forget gate, and $\Gamma_o$ stands for output gate,

One good property about LSTM is that instead of having one update gate, we have two separate terms to control the update.

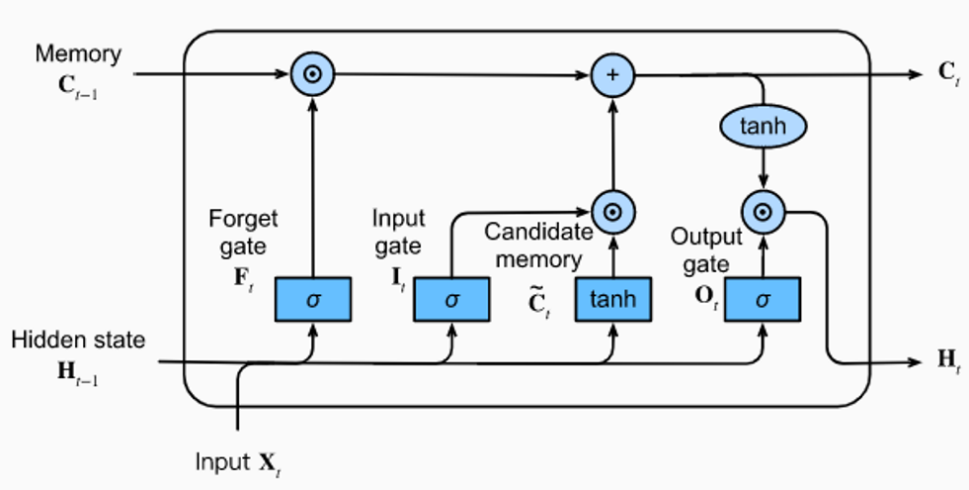



Figure 13.7: LSTM Model

There are a few variations of LSTM. The most common one computes gate values with $a^{<t-1>}$, $x^{<t>}$ and $c^{<t-1>}$, **peephole connection**. There is no widespread consensus on when to use LSTM and when to use GRU. On different problems, different algorithm wins out. Therefore, there is no superior one, but LSTM is more powerful and flexible as it does have three gates instead of two.

## 13.10 Bidirectional RNN

Bidirectional RNN lets us, at the point in time, take information from both earlier and later points in the sequence.

The network's hidden layer will have a forward recurring components and backward connections. For input $x^{<1>}$, it is connected to both $\vec{a}^{<1>}$ and $\overleftarrow{a}^{<1>}$. Also,

both $\vec{a}^{<1>}$ and $\overleftarrow{a}^{<1>}$ are connected to the output $\hat{y}^{<1>}$. The network defines an acyclic graph.

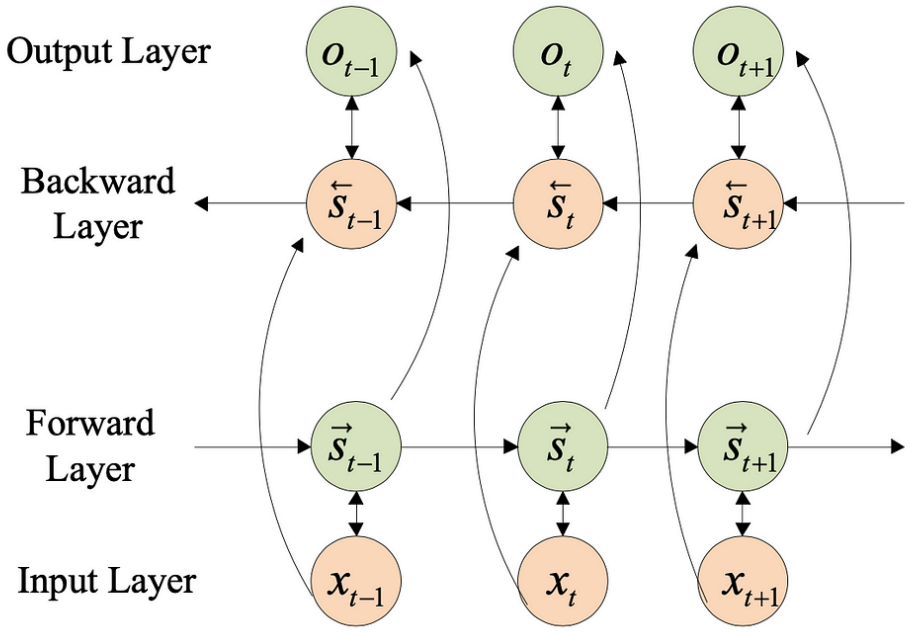



Figure 13.8: Bidirectional RNN Model

The blocks could be standard RNN blocks, GRU blocks, or LSTM blocks. In natural language processing problems, a bidirectional RNN with a LSTM appears to be commonly used. The disadvantage of the bidirectional RNN is that we do need the entire sequence of data before we can make predictions anywhere. To build a speech recognition system, if we use the straightforward implementation, we would need to wait for the person to stop talking to get the entire utterance before we can process it and make a speech recognition prediction.

## 13.11 Deep RNNs

The different versions of RNNs already work quite well by themselves. However, for learning very complex functions, sometimes it is useful to stack multiple layers of RNNs together to build even deeper versions of these models.

We use $a^{[l]<t>}$ to denote layer $l$ associated with time step $t$. To see how each activation value is computed, we can take an example of $a^{[2]<3>}$. $a^{[2]<3>}$ has two inputs – $a^{[1]<3>}$ and $a^{[2]<2>}$. It can be computed as follows:

$$a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]}).$$

For RNNs, having three layers is already quite a lot. Because of the temporal dimension, these networks can already get quite big even if we have just a small handful of layers. We do see sometimes that after recurrent layers, we might take the output and have a bunch of deep networks, which do not have horizontal connections, used to predict $\hat{y}^{<t>}$.

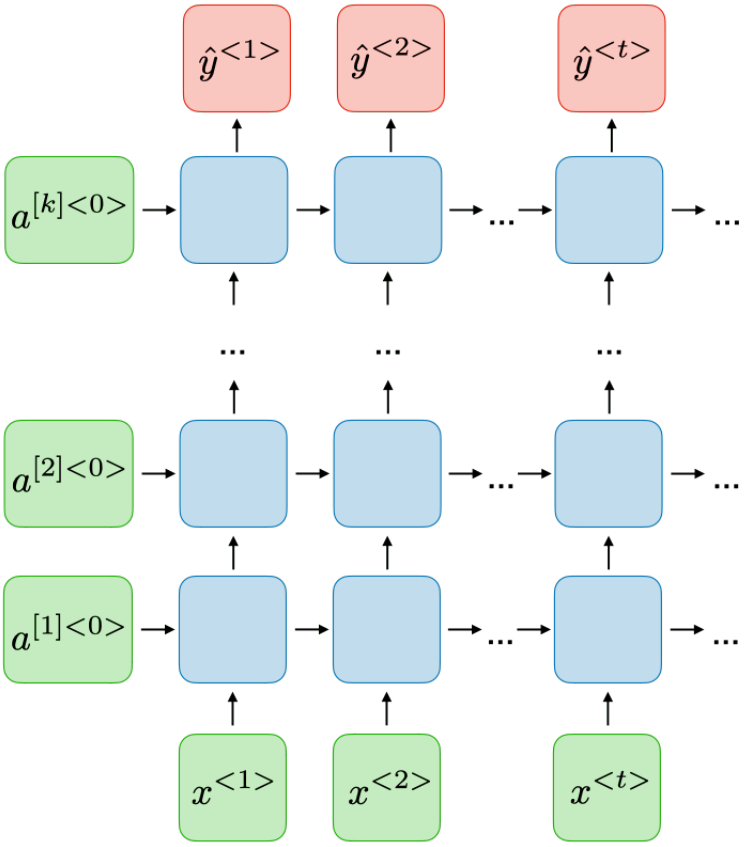



Figure 13.9: Deep RNN Model