An Introduction to Deep Learning

First Edition, 2021

Authors
Oh Tien Cheng, Wong Zhao Wu

October 21, 2021

Pre-Text

Welcome to SPAI's inaugural Deep Learning workshop. This book is meant as a companion piece to the workshop, with detailed explanations of concepts covered during the workshop itself.

The goal of the workshop is to provide an intuitive explanation in understanding some of the major building blocks of Deep Learning and Neural Networks. The workshop is suitable for individuals with a basic understanding of Machine Learning concepts. At the end of the workshop, Deep Learning will no longer be techno-wizardry, but something that you understand and even create!

We hope you have a smooth learning experience ahead!

Contents

Chapter 1

Day 1

1.1 The New Frontier

> "AI is the new electricity" - Andrew Ng

In recent decades, there has been an explosion in hype surrounding **artificial intelligence (AI)**. From the AI algorithms that recommend you new videos on YouTube and TikTok, to advances in technology like self-driving cars, AI has already revolutionized our world.

Artificial intelligence is defined as the science and engineering of making intelligent machines [1]. In short, the field of AI tries to teach machines (especially computer programs), to perform tasks that would normally require intelligence.

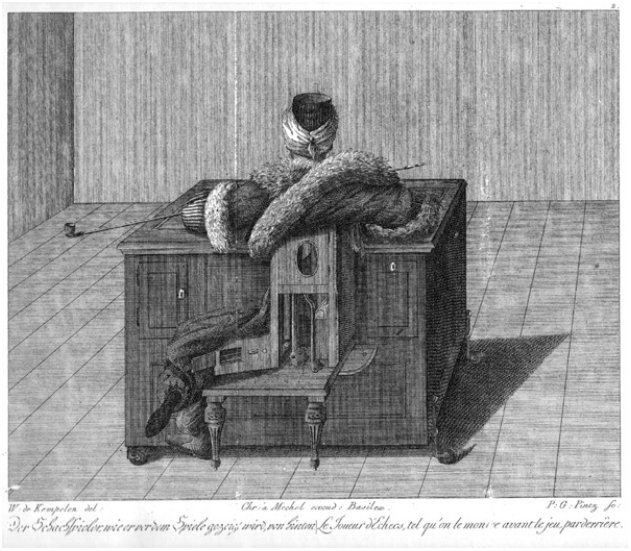


Figure 1.1: The Mechanical Turk

While the AI hype may seem relatively recent, humans have been dreaming of machines capable of thought for centuries. One such example was the Mechanical Turk [2] (see Figure 1.1). Built in the 1700s, it was purported to be a machine capable of playing chess by itself. It fascinated Kings and Emperors all over Europe, who were fascinated by the

idea of a machine being able to beat humans at their own game. However, it turned out to be a fraud, with a human operator secretly controlling the machine. It seemed that a machine capable of performing tasks requiring a human intellect would remain a fantasy. However, this would change with the birth of the field of AI in the 1950s by computer scientists hoping to make use of computing machines to make this dream of a thinking machine a reality.

1.1.1 Can Computers See Electric Sheep?

One of the earliest things humans were trying to accomplish in AI was to teach machines to see. This field is known as **Computer Vision** [3]. However, this turned out to be a difficult task.

A computer is fundamentally a calculating machine that can calculate really, really fast. While computers today are really powerful, even seemingly complicated tasks like displaying a video game is really just a lot of math behind the hood, with the computer constructing images from a constant stream of ones and zeros.

While the ability to do millions of calculations per second might sound impressive, computer scientists, in trying to create AI systems, discovered that computers struggled to do many other tasks that humans (and even small animals) can do easily. For example, what does the image shown in 1.2 show?

Figure 1.2: It's a Cat!

It's obviously a cat, but how do you teach a computer to know what a cat is? For scientists in the 1950s, trying to program a computer to recognize such images seemed to be an impossible task. Initially, they tried devising formal rules to code into an algorithm, but it turns out that devising rules to describe our extremely complex world is difficult. Then, a new paradigm of AI, called **machine learning**, came and tried to tackle this problem. It was based on a simple premise: instead of telling the computer what to do, why not let the computer figure itself out by extracting patterns from raw data? [4]

However, early machine learning algorithms were not up to the task of dealing with such a complex task. An early machine, called a Perceptron, tried to solve this by taking inspiration from neurons in the brain, but was found to be only capable of learning a linear decision boundary. As such, it was extremely limited in what tasks it could accomplish. [5]

However, researchers soon came to realise that although a single Perceptron was extremely limited in it's learning capacity, it was possible to chain Perceptrons together, forming a feedforward neural network (a **multi-layer perceptron**, or MLP for short). This led to the birth of the neural network, and thus, the field of deep learning.

1.2 What is Deep Learning?

> "Deep Learning is a subset of machine learning where artificial neural networks, algorithms based on the structure and functioning of the human brain, learn from large amounts of data to create patterns for decision-making. Neural networks with various (deep) layers enable learning through performing tasks repeatedly and tweaking them a little to improve the outcome." [6]

1.2.1 Representation Learning

When trying to determine how to teach a machine to understand a complicated concept, it's often useful to take a look at the ultimate proof by example: humanity

As humans, we typically learn complicated concepts by representing them as a composition of more basic concepts. For instance, to recognize an object in an image, we will first scan through the image and recognize simpler features like the lines and edges before combining them to form more complex features like the basic shape of the object. Finally, based on the shape and additional information like the texture and colour of the object, we can successfully identify the contents of the image.

In fact, when it comes to solving problems, we often find that how we represent these problems can make a world of difference. For instance, take the problem of adding two numbers together.

$$9 + 10 = ?$$

When we write the equation down like this, the answer is fairly obvious. But what would happen if we represented the numbers using the Roman numeral system?

$$IX + X = ?$$

Suddenly, the problem becomes a lot more daunting. We find the earlier expression easier to solve, since it is represented in a way that we are more familiar with.

Unfortunately, traditional machine learning algorithms are unable to form deep representations of problems. (Figure 1.3) As such, in the past, humans trying to get machines to perform complex tasks like image recognition have had to hand-engineer features based off what they felt would help the machine learning algorithm understand the problem better.

Deep Learning solves this problem by taking a whole new approach: instead of humans having to determine the best possible representation of the problem, *why not use a flexible machine learning algorithm which can figure out the best representation by itself?*

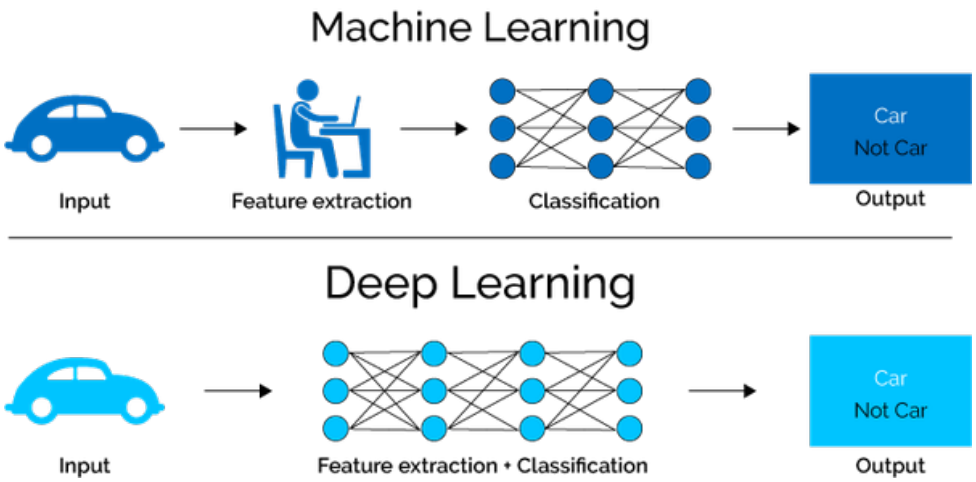


Figure 1.3: Traditional Machine Learning vs Deep Learning[7]

This approach has allowed for the success of neural networks in taking on challenging tasks that traditional machine learning algorithms simply fail at. As can be seen in Figure 1.4, a deep learning algorithm is able to form its own representation of the data. Generally, a neural network is comprised of many functions, called layers, which learn to transform the data into a different representation, and pass on that representation into another layer. This results in a **feature hierarchy**, where earlier layers are able to detect more low level features of the data, while later layers form a more complex representation of the data, which is then finally used for prediction. [8]

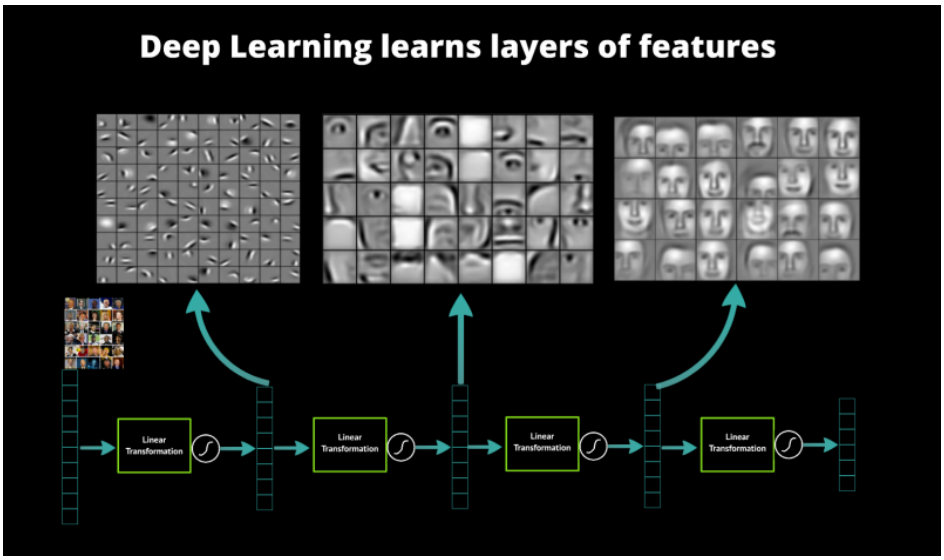


Figure 1.4: Representation Learning of Deep Learning

1.2.2 Applications of Deep Learning

As a result of the capabilities of deep learning, it has already seen wide application in a variety of fields, including:

- Computer Vision
  - Image Recognition

  - Object Detection
  - Semantic Segmentation
  - Facial Recognition
  - Optical Character Recognition
- Natural Language Processing
  - Speech Recognition
  - Speech Synthesis
  - Machine Translation
  - Question Answering
  - Conversational Chatbots
- Reinforcement Learning
- Generative Modelling
  - Deep Fakes
  - Style Transfer
- And More!

1.3 Anatomy of A Neural Network

1.3.1 The Neuron

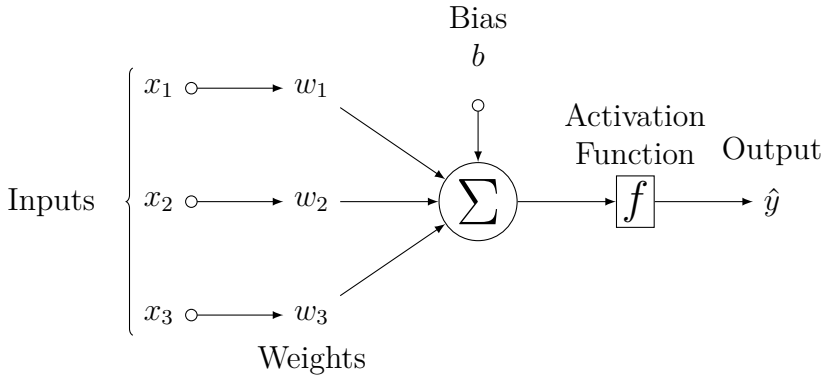


Figure 1.5: Neuron

The neuron is the most basic unit of the neural network. As such it is also referred to as a **Unit**. It is a mathematical abstraction of how a biological neuron looks like. Functionally, it is simply a function that takes in an input (often multiple inputs), and generates a single output.

A neuron computes its output in two stages: Firstly, we begin by feeding in our input features x into the unit. As each feature is passed into the unit, they are multiplied by a

set of weights w that correspond to each input, and added up together. A bias term b is then added to the final result. In Figure 1.5, the computation would look like this:

$$z = x_1 w_1 + x_2 w_2 + x_3 w_3 + b \tag{1.1}$$

Note 1

For those familiar with linear algebra, you may note that it is possible to represent this calculation as a dot product between an input vector and a weight vector, with a bias term added to the final result.

$$z = w^T x + b \tag{1.2}$$

In fact, such vectorized implementations are often used in deep learning, as they greatly speed up computation, but for the sake of accessibility, we will try to refrain from using linear algebra notation unless necessary.

After representing our inputs as a weighted sum z, we will then pass z into an **activation function** g.

$$a = g(z) \tag{1.3}$$

We will dive deeper into activation function later in Section 1.5, but for now, you could think of the first phase as mixing the ingredients needed to bake a cake, and the second phase (the activation function), as putting the mix into an oven and baking it. The final output is known as the **activation** of the unit.

Example 1

Suppose I am trying to predict whether or not a person will default on his loan. I have three features, x_1, x_2, x_3 representing a person's age, income, and bank account balance. It turns out that if I just use a single neuron to make a prediction as follows.

$$x = \begin{bmatrix} 29 \\ 4000 \\ 10000 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1.4}$$

$$w = \begin{bmatrix} 0.1 \\ 0.02 \\ 0.01 \end{bmatrix} \tag{1.5}$$

$$b = -180 \tag{1.6}$$

$$z = w^T x + b \tag{1.7}$$

$$= \begin{bmatrix} 0.1 & 0.02 & 0.01 \end{bmatrix} \begin{bmatrix} 29 \\ 4000 \\ 10000 \end{bmatrix} - 180 \tag{1.8}$$

$$= (0.1 * 29 + 0.02 * 4000 + 0.01 * 10000) - 180 \tag{1.9}$$

$$= 2.9 \tag{1.10}$$

I can then pass the sum into specific activation function called a *sigmoid function*, which will output a probability (between 0 and 1), indicating if the loan request should be accepted.

$$g(z) = \frac{1}{1 + \exp(-z)} \tag{1.11}$$

$$a = g(2.9) = 0.947 \tag{1.12}$$

Hence, a probability of 0.947 is the output of the neuron.

Note. We should note that when we only make use of a single neuron with this sigmoid activation, we are in fact just using a **logistic regression** model. Since a logistic regression model is in fact a linear model, a single neuron by itself is only capable of learning a linear decision boundary to decide if a loan should or should not be approved.

1.3.2 The Layer

In a neural network, a layer is formed by a set of units all acting in parallel (parallel, as each unit within the same layer receives the same set of inputs). The **width** of a layer is determined by the number of units/neurons inside the layer. The layer as a whole can be taught as a single function, that takes in a set of inputs (typically the output of the previous layer), and returns another set of outputs, which are the activations of the units within the layer. Figure 1.6 shows a visualisation of a 2 unit layer.

The computation for this simple layer would look something like this [9]:

$$z_1^{[l]} = w_1^{[l]T} x + b_1^{[l]}; \qquad a_1^{[l]} = g(z_1^{[l]}) \tag{1.13}$$

$$z_2^{[l]} = w_2^{[l]T} x + b_2^{[l]}; \qquad a_2^{[l]} = g(z_2^{[l]}) \tag{1.14}$$

Layer

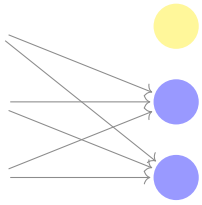


Figure 1.6: A 2 Unit Layer in a Network, with a Bias Term (in Yellow)

(where $w_i^{[l]}$ refers to the weights of the ith unit in the layer l)

Now, you can imagine that if I have a lot of units in a layer, it can get tedious if I have to have a separate calculation for each unit. Luckily, we can make use of matrix multiplication to to vectorize the equations to calculate the outputs of all the units in the layer at once.

$$Z^{[l]} = W^{[l]}x + b^{[l]} = \begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \end{bmatrix} \tag{1.15}$$

$$A^{[l]} = g(Z^{[l]}) = \begin{bmatrix} a_1^{[l]} \\ a_2^{[l]} \end{bmatrix} \tag{1.16}$$

In the vectorized equation, the parameters of each layer l are a weight matrix $W^{[l]}$, which consists of the weights of every unit within the layer, and a bias vector $b^{[l]}$. If you don't get the two equations above, don't worry too much about it, since your deep learning library will handle the calculations for you behind the scene.

Ultimately, this computation, known as **forward propagation**, is how our data, is passed from one layer to the next within a neural network.

The Input Layer

The first layer in a neural network is known as an input layer. Unlike the other layers, there is no actual computation that is done during this layer. It is merely where the input data are stored. As such, the number of units within this layer, n_x, is equal to the number of input features.

For instance, if I am creating a neural network to predict if a black and white picture is of a cat, each feature would be a pixel of the image. So for a 28 by 28 greyscale image, I would have $28 * 28 = 784$ input features.

On the other hand, if I wanted to use a neural network to make a prediction on a tabular data set (e.g. Housing Prices) with 7 features, I would have 7 input units in my input layer.

The Hidden Layer(s)

The hidden layers are the layers that come after the input layer, but before the final output layer. This is where the network will transform the data, forming a more complex representation of the data with each layer.

During computation, the data from the input layer will pass into the first hidden layer, which will then compute an intermediate result(i.e. activation of the first layer), which is passed on to subsequent layers.

In practice, we will use more than one hidden layer, as adding more layers allows for the network to learn better representations of our data. In fact, many deep learning practitioners consider any neural network with only one hidden layer to be a *shallow neural network*, while a network with many hidden layers is considered a *deep neural network*.

The number of units in the hidden layers, $n_h^{[l]}$ can be decided by you.

This concept of chaining hidden layers together to form better representations of the data was initially inspired from neuroscience (the study of the brain), but it's important to note that modern neural networks are not supposed to be a replica of the human brain, merely loosely inspired by it (hence modern neural networks have very little similarities to our actual brains). [4]

Question 1

Why do we call these layers "hidden"?

Answer 1.1

We call these layers "hidden" layers, since we don't actually know what we want the output of these intermediate layers to be [4]. (that is, I know that I want the result of the output layer to be as close to the real answer as possible, but I don't know what the proper output of the hidden layers should be)

The Output Layer

The output layer is the last layer in the network, and is supposed to output the value that we are trying to predict. You can think of this layer as a machine learning model that tries to make a prediction based on a set of features discovered by the hidden layers.

The number of units, n_y in the output layer depends on what we are trying to predict.

In a classification problem, the number of units is typically the number of classes. For example, if I were training a model to recognize handwritten digits, the number of units would be 10, with the output of each unit representing the probability that the image is of a particular digit.(we would then predict the digit with the highest probability) In the case of a binary classification problem (e.g. predicting if an image is of a cat or a dog), I could simply make use of a single unit, predicting the probability that the image is of a single class.

In a regression problem, the number of units would be equal to the number of features I am trying to predict. In most cases (e.g. predicting housing prices), we would only require one unit. In other problems however (e.g. object detection), I might need my network to predict more than one output (e.g. the coordinates of a bounding box).

1.4 The Neural Network As a Whole

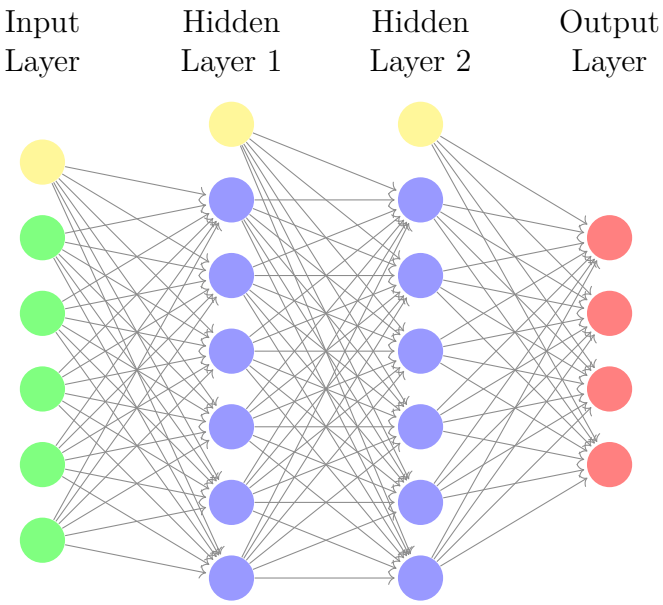


Figure 1.7: A 3 Layer Neural Network

When we look at the neural network as a whole, we can think of it as a composition of many functions (represented by each layer), with each layer acting as a function that takes in the input from the output of the previous layer (with the exception of the input layer). [4]

The hidden layers transform the data from the input layer into a more informative representation, which is then passed into the output layer to make a final prediction based on the new representation of the data.

Note 2

We should also note that the layers do not necessarily need to be connected in sequential order. In fact, certain architectures like Residual Networks (ResNets) have layers connected in sequence, but also include skip connections going from one layer to another layer deeper into the network [10]. (these skip connections allow for deeper neural networks to be trained without risk of over-fitting)

1.5 Activation Functions

When we first took a look at the neuron, we mentioned that the output of the neuron is the output of a linear combination of inputs, passed into an activation function g.

So, what is an activation function? An activation function is simply any non-linear function, that is used to introduce non-linearity to the network. It is this non-linearity that allows neural networks to learn complex patterns from our data. In fact, the **universal approximation theorem** tells us that a neural network with at least one hidden layer using a non-linear activation function (e.g. sigmoid), can approximate almost any function. (note that this means that it is possible for the network to represent almost any function, but does not guarantee that the training algorithm will be able to learn it) [4]

Why Do We Need Non-Linear Activation Functions?

To understand why the activation function is so important, let's take a look at what happens when we don't have an activation function, or to be more precise, when our activation function is only a linear function.

Figure 1.8: A Neural Network Without An Activation Function

We will let the activation function g be $g(z) = z$. This effectively means that we have no activation function. We will then make use of forward propagation to feed the two input features through the network.

$$W^{[3]}(W^{[2]}(W_1^{[1]}x_1 + W_2^{[1]}x_2)) = W^{[3]}W^{[2]}W_1^{[1]}x_1 + W^{[3]}W^{[2]}W_2^{[1]}x_2 \tag{1.17}$$

We can observe that without the nonlinear activation function, even a deep neural network is only capable of outputting a linear combination of the inputs, making it nothing more than a glorified linear regression model. [11]

In fact, we can even show that so long as my activation function is linear (e.g $g(z) = \frac{1}{2}z+5$), the output will still just be a linear combination of the inputs.

Note 3

That is not to say that a linear activation is useless. They can be useful at the output layer, if I am trying to solve a regression problem. It just means that we should not use a linear activation as part of our hidden layers.

Rectified Linear Unit (ReLU) Activation

The ReLU activation is by far one of the simplest, yet most effective activations out there. It is a piecewise function defined as follows.

$$ReLU(z) = \begin{cases} z, z \geq 0 \\ 0, z < 0 \end{cases} \tag{1.18}$$

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

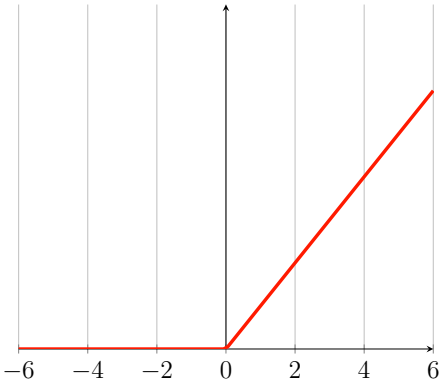


Figure 1.9: ReLU Activation

The ReLU activation, despite its simplicity, has gained rapid popularity as an activation function for units in the hidden layer, as it was found that it allowed for the training of deeper neural networks as compared to other activations (e.g. sigmoid, tanh). [12]

Question 2

Doesn't this just look like a linear function? I thought activation functions were supposed to be non-linear?

Answer 2.1

Not exactly, while it is very close to a linear function, the fact that all negative inputs become zero, makes it a non-linear function. In fact, it is this simple property that allow us to make use of many ReLUs together to approximate other functions.

For instance, let's try using a single layer of ReLUs to approximate the function $f(x) = x^2$.

We can let the approximation function be $a(x)$.

$$a(x) = ReLU(5x) + ReLU(-5x) + ReLU(7x - 24) + ReLU(-7x - 24) \tag{1.19}$$

When we plot out the approximation, we get the result in Figure 1.10

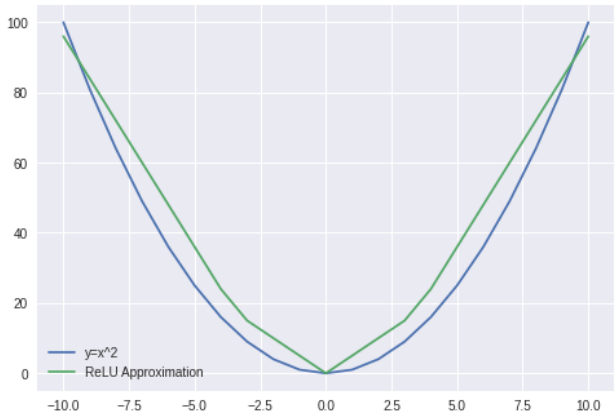


Figure 1.10: Approximating $f(x) = x^2$

Although the approximation is not perfect, it is easy to see that as I add more and more ReLUs, the approximation will only get better.

Sigmoid Activation

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

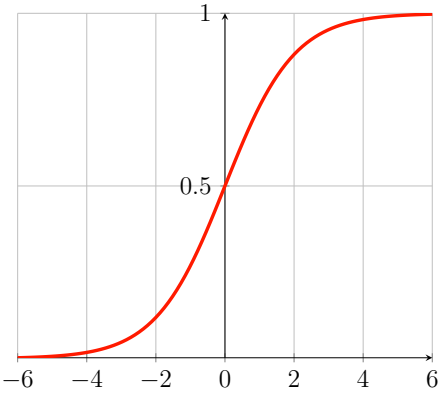


Figure 1.11: Sigmoid Activation

Before the popularity of ReLU, the sigmoid function used to be the most popular activation function for the hidden layers. However, the sigmoid activation still sees some use today in the output layer of a neural network. It is defined by the following equation.

$$g(z) = \frac{1}{1 + \exp(-z)} \tag{1.20}$$

The fact that the sigmoid function has a range of $[0, 1]$ allows us to train the neural network to output a probability between 0 (0%) to 1 (100%). This makes the sigmoid function useful for an output layer when trying to solve a binary classification problem (meaning, I am trying to predict a True (1) or False (0) outcome).

$$\begin{bmatrix} 3.82 \\ 5.35 \\ 1.44 \\ -1.26 \\ 2.71 \\ 1.98 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 0.16115195 \\ 0.74422819 \\ 0.01491471 \\ 0.00100235 \\ 0.05310907 \\ 0.02559374 \end{bmatrix}$$

$$softmax(z) = \frac{e^z}{\sum e^z}$$

Figure 1.12: Softmax Activation

Softmax Activation

While a sigmoid activation is useful for a binary classification problem, what happens if I wish to predict an outcome out of many different classes? (e.g. Predicting grades of a student)

This is where the softmax activation comes into play. It takes in an input, and outputs a normalised probability distribution vector $\hat{y}$ that sums up to 1. The j^{th} element of the probability vector $\hat{y}_j$ can be interpreted as the probability that the correct answer is class j. Such characteristics allow us to conveniently decide the predicted classes based on the neuron with highest probability value which makes softmax the best candidate in the context of multi-class classification problem.

1.6 How Does A Neural Network Learn?

In the previous sections, we saw how a multi-layer perceptron, takes an input x and produce an output $\hat{y}$, by passing the data through the layers in the network. This is called **forward propagation**. The output of forward propagation is primarily dependent on the weights W and biases b of the network.

But how does the network know what the weights and biases are? The answer is that the network must learn it on its own.

But how does a network learn? It turns out that we need to train (or fit) the network using an iterative method. To get an intuition as to how network is trained, let's take a look at how a simple neuron with a linear activation can be iteratively trained to convert a length from kilometers to miles.

Example 2

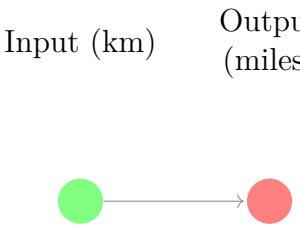


Figure 1.13: A Unit Converting Machine

Now, given this simple network (it's effectively just a linear regression model), we will try to train the model to learn the formula for converting between kilometers and miles. As we will find out, figuring out the correct weight is nothing more than a more complex version of **trial and error**.

Our model has only a single parameter to learn, w, and the final output is given by the formula: $\hat{y} = wx$

Our dataset is as shown in Table 1.1

Kilometres	Miles
0	0
100	62.137

Table 1.1: Data Set

To learn the correct value of w, we will start by choosing a **random** value. Let's try $w = 0.5$

When we try making a prediction, we get $\hat{y} = 100 * 0.5 = 50$

Based of the dataset, we can see that our **error**, the difference between the calculated answer and the actual answer, is $62.137 - 50 = 12.137$

Now that we know how wrong our guess is, we will use this error to help us guess a better number that will reduce our error.

We know that since our error is positive, we have undershot our target. Hence, we want to increase the value of w as it will increase the predicted value.

We will nudge the value of w by a tiny amount to 0.6 (if we increase w too much, we might end up overshooting our target).

$$w := w + 0.1$$

Our new prediction is now $\hat{y} = 100 * 0.6 = 60$, and our corresponding error is 2.137.

We can continue this process, until our error is very close to 0. Note that we didn't directly solve for the value of w, but instead adjusted an initial guess based on feedback from how the prediction using the parameters compare to the actual answer. (if

you've ever done trial and error back in Primary School math, this is basically a more advanced version of what you did back then)

Although the iterative method seems simple, the way that we train a full neural network is effectively just an extension of how we trained the converting machine.

The general training process shown as follows in Algorithm 1

Algorithm 1 Neural Network Training Procedure (Simplified)

$lr \leftarrow 0.01$ ▷ Learning Rate –> How big should each update be?
for each layer in network l **do**
  $W^{[l]} \leftarrow$ Randomly Initialize the Weights
  $b^{[l]} \leftarrow 0$
for each epoch e **do** ▷ Epoch is just a fancy word for iteration
  $\hat{Y} \leftarrow$ Forward Propagation to Generate Output
  $Loss \leftarrow J(Y, \hat{Y})$ ▷ Loss == Error
  for each layer in network l **do**
    Calculate update to parameters based on loss ▷ Parameters: Weights and Biases
    $W^{[l]} \leftarrow W^{[l]} -$ Weight Update $* lr$
    $b^{[l]} \leftarrow b^{[l]} -$ Bias Update $* lr$

In the subsequent sections, we will take a deeper look into how we tackle the task of training a neural network.

1.6.1 Loss Functions

> "If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle." - Sun Tzu, The Art of War [13]

When Sun Tzu wrote the quote above, he obviously wasn't thinking about deep learning, but it turns out that his words hold true in many contexts. What Sun Tzu was trying to say is that success is only possible if we not only know the problem we are trying to solve (our enemy), but also know ourselves: how close we are to solving the problem. Right now, we only have some vague notion that we want to make our model produce an accurate prediction. But this goal is vague: let's make our goal more concrete and measurable.

In the earlier example (converting kilometers to miles), we described calculating an error, a value denoting how far the model prediction is from the actual answer. In machine learning, we call the function that describes our error a **loss function**. This loss is calculated by comparing a set of predictions $\hat{y}$ with the actual target label y

For instance, a common loss function for regression problems is known as **mean square error**.

$$J(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

By having a measure of how badly the neural network is doing, we can recontextualize our problem from some vague notion of getting a good model, to a more quantifiable goal: **finding the parameters of the model that minimize the loss of the model**. That is, to find the following:

$$\arg\min_{W,b} J(y, \hat{y}) \tag{1.21}$$

We will not go into exact detail into the various loss functions until Day 2, but we will generally summarise the loss functions used for various problems.

- Mean Squared Loss: Regression
- Binary Cross Entropy: Binary Classification/ Multi-label Classification
- Categorical Cross Entropy: Multi-Class Classification
- Wasserstein Loss: Used in Generative Adversarial Networks

Question 3

What is the difference between a metric and a loss function?

Answer 3.1

We can define a loss and a metric as follows:

- Loss: A value, given by a loss function, that we use to optimize a model
- Metric: A value, that we use to judge the performance of a model

The primary difference between the two is their purpose. A metric is a value that we humans look at to see how good the model is at a task we want to do. Meanwhile, a loss is a value that the **optimization algorithm** uses to guide the training of the model. As such, a metric might be designed to be more interpretable for humans (e.g. accuracy), as compared to a loss (e.g. binary cross-entropy).

There are some cases where the metric we use and the loss function are the same (e.g. mean squared error), but not every metric is suitable to be a loss function. This is because the loss function needs to be **differentiable** as the optimization process uses the derivatives of the parameters with respect to the loss to derive the updates to the parameters. As such, certain metrics like accuracy, which are not differentiable, cannot be used as a loss function. Instead, we use a **proxy** to the accuracy metric, cross-entropy.

1.6.2 Gradient Descent

With our training objective clearly defined, let's discuss how a neural network is able to determine the optimal weights and biases that minimize our loss.

Now, in trying to find these parameters, we face a problem. There are so many parameters (a small neural network can easily have thousands of parameters to learn), that we cannot

solve for the parameters directly. Furthermore, there are so many combinations of weights that we cannot possibly try them out. So to solve this problem, we need to try and tackle the problem iteratively.

The specific technique we will is known as **gradient descent**.

Now imagine this, you are stuck somewhere in the wilderness. The landscape is very hilly, and you are out in the dead of night. You're stuck on some hill, and you need to get back to your camp at the bottom, but because it's night, you don't know where to go. Luckily, you have a flashlight, but it only lets you see so far.

In this situation, how do you get down? One strategy you might try is to use the torchlight to see which parts of the ground go downhill, and take small steps down in the direction where the slope of the ground is the steepest downwards. Ultimately, by following this strategy, we should find ourselves at the bottom of the hill, in spite of the fact that we lack a clear view of the landscape.

It turns out that gradient descent is very similar to this. But instead of finding our way down a hill, our optimization algorithm must find it's way down a mathematical function. And instead of using the slope of the ground to decide which way goes down, we use the slope, or **gradient** of the function (to be more specific, the partial derivatives of the loss function with respect to the weights and biases) to guide the way down.

To understand this, let's take a look at how we can use gradient descent to optimize a very simple function, to gain intuition into the technique.

Suppose I have a function $f(x)$ (see Figure 1.14), and I want to find the point $\hat{x}$, that gives me the minimum point of the function.

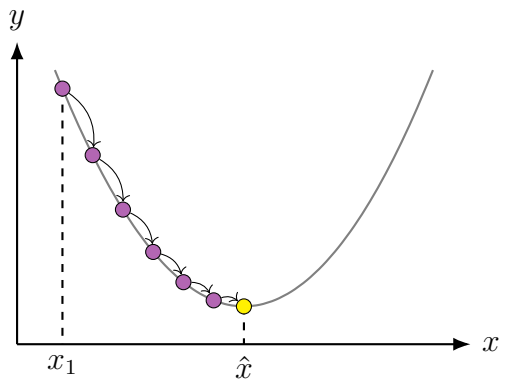


Figure 1.14: Graph of $f(x)$

Now, suppose that we start at random point on the graph x_1, so our initial guess for $\hat{x} = x_1$. At this point, we have a choice of moving in two directions: left or right

To decide which is the best direction to go to, we look at the slope, or gradient of the function. From the graph, we can tell that the slope at x_1 is negative. This means that when I increase $\hat{x}$, y will decrease. Mathematically, this means that the best choice is to move in the direction of the steepest negative gradient. We will thus do an update to $\hat{x}$ as follows:

$$\hat{x} := \hat{x} - \alpha \frac{dy}{dx} \tag{1.22}$$

In the above equation, we moderated the size of the update by using a learning rate α. We need this learning rate to slow down our movement as if we move too far to the right, we might just end up overshooting, and land in the opposite side of the graph. We will then continue this iterative process until we end up at the bottom of the graph.

Now that we've seen how gradient descent works on a small function, we simply need to use this process on the loss function of our neural network. The general idea is the same, but instead of a function with only a single variable x, we have a function with every weight and bias as a parameter. As such, we have to calculate the partial derivatives of the loss function with respect to each of the parameters. We form vectors of partial derivatives $\nabla_{W^{[l]}}$, $\nabla_{b^{[l]}}$, which are direction vectors, telling us the direction of steepest descent for the weights and biases in each layer.

Then, we can update the parameters of each layer l as follows:

$$W^{[l]} := W^{[l]} - \alpha \nabla_{W^{[l]}} \tag{1.23}$$

$$b^{[l]} := b^{[l]} - \alpha \nabla_{b^{[l]}} \tag{1.24}$$

This is known as the **update rule**.

Note 4

When it comes to optimizing a function, there are a few calculus terms that you will need to be aware of. [4]

- Derivative $\frac{dy}{dx}$: the slope, or gradient of a function at a certain point on the function. The derivative tells us how to make a small change in the input x, to obtain the corresponding change in the output y. For example, the derivative of $y = 2x$ is 2, which tells us, that if I increase x by 1, the output y will increase correspondingly by 2 times.
- Partial Derivative $\frac{\partial y}{\partial x_i}$: when we have a function with multiple inputs (e.g. x_1, x_2), we use partial derivatives. The partial derivative measures how the output of a function y changes as only the variable x_i increases at a certain point. If you know how to calculate a derivative, you already know how to calculate a partial derivative: just treat all other variables as constants.
- Gradient $\nabla_x f(x)$: the gradient of a function is the vector containing all of the partial derivatives. You can think of it as a vector which points in the direction of the slope.
- Local Minimum: a point where $f(x)$ is lower than all the neighbouring points (but not necessarily the lowest value in the whole function).

Since this is not a calculus course, we won't go much deeper into it. If you are confused by any of the calculus terms used, we suggest that you check out 3Blue1Brown's Essence of Calculus series for an introduction to the topic. [14]

Backpropagation

The algorithm by which the gradients are calculated is known as **backpropagation**. To summarize, backpropagation works by going through the layers in the network in reverse

order (output layer to input layer), making use of the *chain rule* from calculus. [15]

Note 5

The chain rule tells us how to differentiate a composite function. A composite function is a function, that takes in as input, the output of another function. For example if I had two functions $f(x) = \log(x)$, and $g(x) = x^2$, then a composite function would be $f(g(x)) = \log(x^2)$. To differentiate the function, we apply the chain rule, which tells us:

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x) \tag{1.25}$$

In practice, you do not need to implement, nor understand the process itself, as the computations are automatically done for you by any modern deep learning library.

Chapter 2

Day 2

In day 2, we will go through some practical advice for designing deep neural networks, as well as exploration of two classic neural network architectures: **Convolutional Neural Networks** and **Recurrent Neural Networks**.

2.1 Designing a Neural Network

2.1.1 Neural Network Architecture

When referring to the **architecture** of a neural network, the term refers to the overall structure of the network [4], of which there are two primary considerations we will discuss. Namely, number of layers in the network (the **depth**), and the number of units per hidden layer (the **width**).

Note 6

There is actually one other primary consideration, which is how the layers are connected to each other. We will ignore this for now, as we are currently dealing with only a single way of connection where each input unit is connected to every output unit of the previous layer. (what is known as a **Dense** or **Fully Connected (FC)** layer)

When we take a look at convolutional neural networks in a later section, we will see how they make use of specialized patterns of sparse connections [4] instead of the typical dense connections of a basic neural network.

2.1.2 Choosing the Number of Hidden Layers

We mentioned earlier back in Section 1.5 that a neural network with only a single hidden layer, could, in theory, approximate almost any function. It might then, seem tempting to build a neural network with only a single hidden layer. Despite that, the trend has been to build deeper and deeper networks. Certain networks, like ResNet, can go up to even 152 layers deep. The simple reason for this is that although a shallow network could perform as well as a deep network *in theory*, in practice we find that deep neural

networks typically perform better overall. The specific reasons for why this is the case is still unclear. [3]

However, there are a few leading hypotheses on the matter:

- Having many hidden layers supports the idea of representation learning, as each layer allows the model to discover more useful features, which are described by simpler features in previous layers. [4] (Figure 1.4)
- Deeper neural networks can represent the same functions in a more compact manner. That is, shallower neural networks require more neurons per layer to represent the same functions that a deep neural network can represent. This has been shown to be true in certain cases, such as in the case of the parity function. [16]

With that said, it turns out that there appears to be a limit to how deep a neural network can go (baring any changes to architecture). The authors of the ResNet paper [10] showed that (see Figure 2.1) very deep neural networks face a problem of degradation. This degradation is not due to overfitting (as the training error also gets worse), but because of difficulties in fitting a very deep network and problems like the **vanishing gradient problem**. [17]

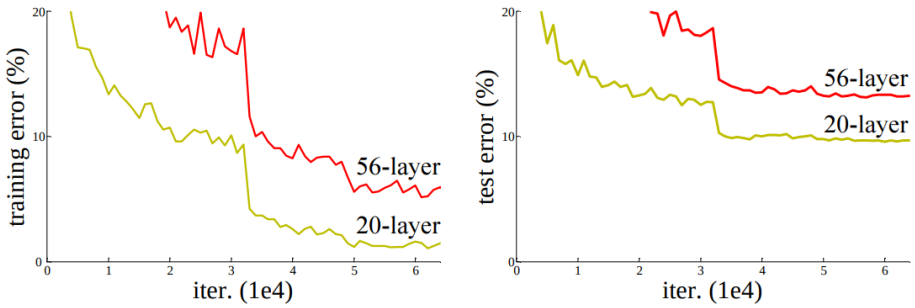


Figure 2.1: Training error (left) and test error (right) on the CIFAR-10 dataset with a 20-layer and 56-layer neural network. The deeper network has a higher training and test set error. [10]

Note 7

That is not to say that this problem cannot be solved. The ResNet paper showed that by making use of "skip connections", it was possible to train very deep neural networks without facing this problem. [10]

Vanishing Gradient Problem

The vanishing gradient problem is a problem that can occur with very deep neural networks and recurrent neural networks. During the optimization process, each parameter in the model is supposed to receive an update based on the partial derivative of the loss function with respect to the parameter. From the chain rule, we know that the gradients for the earlier layers are dependent on the gradients of the later layers. As such, if the gradients in the later layers are small (<1), the effect of having to multiply these small

gradients to obtain the gradients of the early layers means that the gradients of the early layers may become very small. As such, as we train the model, we may find that the gradient updates to early layers are too small to make any substantial changes to the parameters, slowing down, and potentially even stopping the training process entirely.

Example 3

Suppose I am training a very deep neural network, and I am currently performing backpropagation to calculate the gradients of each layer. Suppose I now wish to calculate $\frac{\partial J}{\partial W^{[1]}}$. By the chain rule,

$$\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial W^{[1]}} \tag{2.1}$$

Now, let's suppose that $\frac{\partial J}{\partial z^{[1]}} = 0.3$ and $\frac{\partial z^{[1]}}{\partial W^{[1]}} = 0.5$ (let's assume that each layer only has a single hidden unit to simplify the math). If we apply the chain rule, we will find that $\frac{\partial J}{\partial W^{[1]}} = 0.3 * 0.5 = 0.15$

Now, imagine if I had a learning rate α of 0.1. The final weight update would just be

$$W^{[l]} := W^{[l]} - 0.1 * 0.15 \tag{2.2}$$

The end result is that we have barely updated the weights in the first layer of the network. This shows how the vanishing gradients problem could hamper the training of a neural network.

2.1.3 Choosing the Number of Hidden Units

When selecting the number of hidden units per hidden layer, there are a few considerations we have to take.

We can think of adding more hidden units as like adding more features. As such, we generally expect that a wider network will have a lower bias, but a higher tendency to overfit. However, the optimal number of units per layer is largely dependent on your data, and so you should experiment with different widths of your network, or see the choices that others have made.

Generally, the trend in deep learning has been towards deeper networks, and networks have not really gotten much wider. This is mainly because going deeper instead of wider allows the network to generalize better.

2.1.4 Choice of Activation Functions

When creating a neural network, we have to decide on the activation functions for two different types of layers: the hidden layers, and the output layers.

Typically, **Rectified Linear Units (ReLU)** is used as the default activation function for the hidden layers in Dense layers. As for the selection of activation functions for the output layer, it really depends on the type of Machine Learning problem that you are trying to solve. Table 2.1 provides an overview of some common activation functions and their use cases.

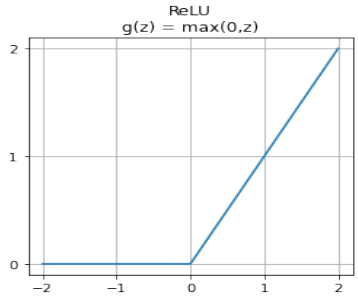
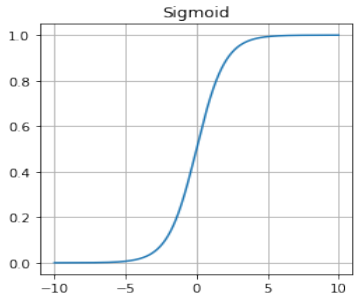
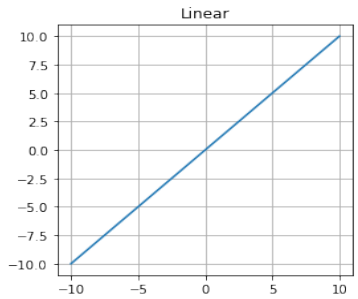
Activation Functions	Characteristics	Use Cases
ReLU Activation		Hidden Layers
Sigmoid Activation	Used to denote probability between 0 and 1. $g(z) = \frac{1}{1+e^{-z}} \in [0,1]$ 	Output Layer *(e.g. Binary Classification, Multi-label Classification)*
Linear Activation	Return the input value as activation output. $g(z) = z$ 	Output Layer *(e.g. Numerical Regression)*
Softmax Activation	Convert the output predictions into vector of probabilities that sum up to 1.	Output Layer *(e.g. Multi-class Classification)*

Table 2.1: Activation Functions and their Use Cases

Note 8

Multi-class classification refer to a single observation having **only a single label** from more than 2 classes, while **Multi-label classification** refer to a single observation having **multiple labels** from more than 2 classes. Refers to Figure 2.2 for a clearer explaination with example.

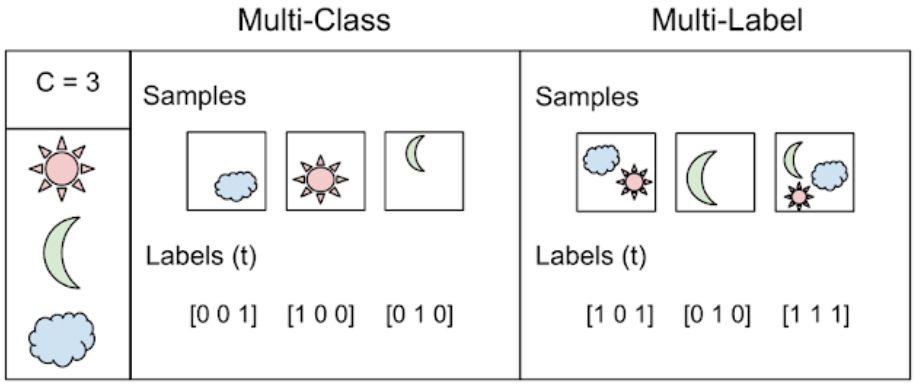


Figure 2.2: Multi-Class vs Multi-Label Classification[18]

Note 9

Unless the network architecture specifies otherwise, we typically stick with the same activation for all the hidden layers.

Why Do We Prefer ReLU?

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

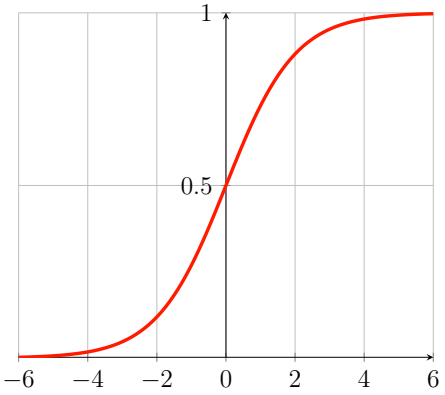


Figure 2.3: Sigmoid Activation

In the past, the *sigmoid activation* (or even the tanh activation) used to be the most popular pick for the hidden layers. However, it has since been subsumed by the simpler ReLU function. Why is this the case?

It turns out that the answer to this is the **vanishing gradients problem** (refer to Section 2.1.2).

When we look at the shape of the sigmoid activation, we observe that the slopes/gradients of the extreme ends of the function are very flat, meaning that their gradients are extremely small. As such, if the activation values are extremely large (i.e. approaching 1) or extremely small (i.e. approaching 0), the gradients calculated would be small, resulting in the vanishing gradients problem.

ReLU reduces the impact of the vanishing gradients problem by having a constant gradient of 1 for all positive inputs. As such, researchers found that simply replacing the sigmoid activation with ReLUs in their neural networks allowed them to train even deeper neural networks.

Note 10

You may note that the gradient of ReLU is 0 when an input is negative. This is in fact a flaw of the activation, as it can lead to a problem of **dead neurons**, where the weights of the layer cause some neurons to output a 0 value, but because of that, the weights associated with those neurons are unable to be updated. A modified version of ReLU, known as *Leaky ReLU* solves this problem. [19]

$$LeakyReLU(z) = \begin{cases} x & , x \geq 0 \\ 0.01 * x & , x < 0 \end{cases} \tag{2.3}$$

For your information, there also exist other variants of ReLU like *Parametric ReLU (PReLU)* and *Randomized ReLU (RReLU)*. You do not have to implement this yourself, as the activations are already included in most deep learning libraries.

2.1.5 Choice of Loss Functions

The choice of the loss function depends on the task that your model is trying to solve. Recall that a loss function is supposed to act as a proxy for the metric you care about, and so we must ensure that minimizing the loss function results in improving the metrics that matters in your context of problem.

Table 2.3 shows a summary of what loss functions to choose and the label format depending on the tasks.

Loss Functions	Use Cases	Label Format
Mean Squared Error	Numerical Regression *(e.g. Housing Price Prediction)*	[[1.3], [5.7], [6,6]]
Binary Cross-Entropy	Binary Classification *(e.g. Cat vs Dog)*	[[0], [1]]
	Multi-Label Classification *(e.g. Classify different items in the image)*	[[0, 1, 1], [1, 0, 1]]
Categorical Cross-Entropy	Multi-Class Classification *(e.g. Cat vs Dog vs Chicken)*	[[1, 0, 0] [0, 1, 0]]

Table 2.3: Summary of Loss Functions

Mean Squared Error

$$L(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 \tag{2.4}$$

The mean squared loss is generally used in regression problems. The loss is represented by the squared distance of the difference between the prediction and the actual value as shown in Formula 2.4. Other than MSE, Mean Absolute Error (MAE) is also another popular loss function for regression problem.

Binary Cross-Entropy

$$L(\hat{y}, y) = -\frac{1}{m}\sum_{i=1}^{m} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{2.5}$$

Binary Cross-Entropy (BCE) is commonly used when the actual label, y is either 0 or 1 and the model prediction, $\hat{y}$ is the predicted probability of an event or a particular class.

At a glance, the formula of Binary Cross-Entropy looks daunting. However, we can dissect formula 2.5 into two different cases: **Case when Actual Label,** $y = 1$ and **Case when Actual Label,** $y = 0$.

When **actual label,** $y = 1$, the red term of $(1 - y_i)\log(1 - \hat{y}_i)$ will disappear as $(1 - y_i = 1)\log(1 - \hat{y}_i) = 0$. And what is left is just the blue term of $-(y_i = 1)\log\hat{y}_i$ that will result in higher loss and thus penalize the model more when the output probability, $\hat{y}_i$ is closer to 0.

On the other hand, when **actual label,** $y = 0$, the blue term of $y_i \log(\hat{y}_i)$ will disappear as $y_i = 0 \log\hat{y}_i = 0$. And what is left is just the red term of $-(1 - y_i = 0)\log(1 - \hat{y}_i)$ that will result in higher loss and thus penalize the model more when the output probability, $\hat{y}_i$ is closer to 1.

Figure 2.4 shows a graphical summary of the explanations above.

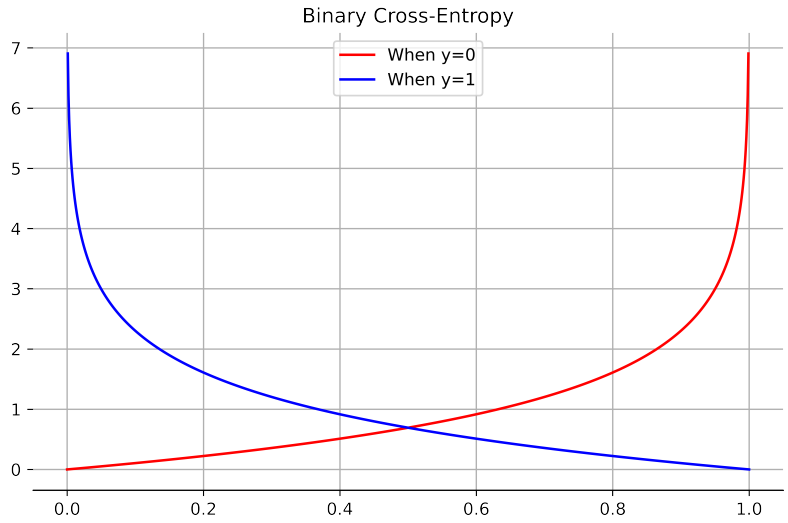


Figure 2.4: Binary Cross-Entropy Diagram

Categorical Cross-Entropy

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k}) \tag{2.6}$$

Categorical Cross-Entropy (CCE) is another variation of the entropy loss that is specialised in handling multi-class problem as shown in Figure 2.2.

In summary, CCE is computed by taking the averaged entropy loss by looping through each label classes for each observations as better illustrated in Algorithm 2.

Algorithm 2 Categorical Cross-Entropy Loss Function

Input: y, $\hat{y}$ which are matrices with dimensions of $(m \times k)$
Output: $J(y, \hat{y})$ ▷ CCE Loss function

```
function Loss(y,ŷ)
    SumLoss ← 0
    m ← length(y)          ▷ m ← Number of Observations
    K ← width(y)           ▷ K ← Number of Classes
    for i ← 1 to m do
        for k ← 1 to K do
            SumLoss ← SumLoss + (−y_{i,k} log(ŷ_{i,k}))
    return SumLoss/m       ▷ Return the averaged loss
```

2.2 Training a Neural Network

Once the architecture of the network and the loss function has been selected, there are a few more choices that we need to make during the training process.

2.2.1 Choice of Optimizer

One of the most important choices to make would be the optimization algorithm used during the training. While we explored **gradient descent** in Day 1, there are actually many variations and improvements on gradient descent that are utilized nowadays (although the general principle of gradient descent is still core to these learning algorithms).

As such, we will do an exploration of how a few of these algorithms work, before looking at which algorithm is generally recommended.

Stochastic Gradient Descent

When we perform gradient descent as described in Day 1, we were actually performing what is known as **batch/vanilla gradient descent**, where we update the parameters of the model after the model computes the gradients using the **entire training data**. As such, if we have a very large data set, we may find that the gradient descent process can be extremely slow. A common practice to solve this problem is to split the training data into **mini-batches**, and perform gradient descent based on each mini-batches. Stochastic Gradient Descent takes this principle to an extreme degree, by using only a single training example to calculate the gradients and update the parameters. This means that the algorithm will update the parameters for every training example.

Note 11

In most deep learning libraries, batch gradient descent, stochastic gradient descent and mini-batch gradient descent are merged, and simply referred to as stochastic gradient descent. [5]

Instead, the choice between the three methods comes down to specifiying a *batch size* hyperparameter in the optimizer.

- Batch Size $= 1$: Stochastic Gradient Descent
- $1 <$ Batch Size $< m$: Mini-Batch Gradient Descent
- Batch Size $= m$: Batch Gradient Descent

Note that m refers to the size of the training data set.

Typically, we will set the batch size to powers of two, such as 32, 64, or 128. (this is because the number of physical processors in the GPU or CPU is often a power of 2, and aligning the batch size results in better performance [20])

There are several benefits of using stochastic/mini-batch gradient descent, including,

- Faster convergence of the algorithm on large datasets,
- Lower memory usage (as compared to batch gradient descent), as we only need to load a single training example at a time

[21]

Stochastic Gradient Descent with Momentum

Momentum works just like normal SGD but with a slight tweak to the formula which helps accelerate gradients vectors based on the directions of past gradients, thus leading to faster converging.

Intuitively, Momentum adds inertia to gradient descent. Just like pushing a heavy ball down the hill, due to inertia, it is easier for the ball to move in the direction that is consistent with direction of previous steps. And on the same time, it is harder to just stop the ball immediately or completely move to a new direction that is not consistent to the previous steps.

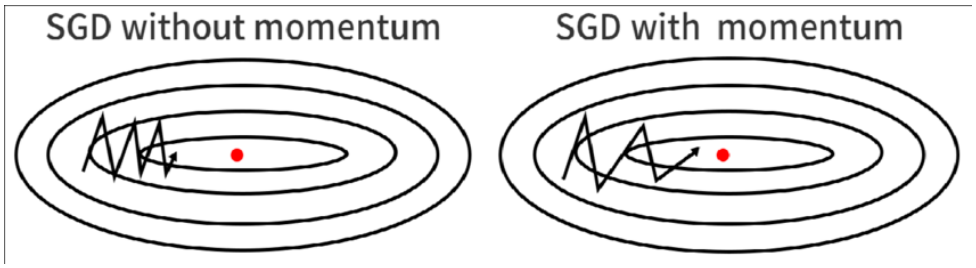


Figure 2.5: SGD with and without Momentum

In summary, the Momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence with reduced oscillation using Momentum. [22]

Adam

Adaptive Momentum Estimation (Adam), is another optimizer that is frequently recommended as the default optimizer to start with. [23]

Adam is an adaptive learning rate optimizer that is able to adjust individual learning rates for different parameter. Adam works by storing an exponentially decaying average of past squared gradients to inversely scale the learning rate like Adadelta and RMSprop, while keeping track of the exponentially decaying average of past gradients just like Momentum.

It is hard to explain Adam without establishing any mathematical formula, but intuitively, Adam works like pushing a heavy ball with rough surface and high friction. Other than having inertia which helps to move the ball in the direction based on the average direction of previous steps(i.e. Momentum), the new friction introduced will reduce the damping effect of the ball and make the ball moving more consistently in direction that leads to local minima. [20]

In summary, Adam is able to navigate through noisy loss terrain by reducing the damping effect and thus achieving faster convergence on most problems.

2.2.2 Choice of Learning Rate

Learning Rate, common denoted as α, is a configurable hyperparameter that represents the magnitude of changes towards the parameters of a model (i.e. Weights and Biases)

for each iteration of gradient descent. You can think of learning rate as how big should your footsteps be when you are descending down the loss valley.

When a higher learning rate is chosen, albeit the model is able to make larger steps in optimizing the parameters (i.e. train faster), sometimes the training steps might be too large and the loss function would ended up oscillating at a very high value. (Refer to Figure 2.6)

On the other hand, when a small learning rate is chosen, the model could guarantee of reaching local minima, but might suffers from longer training time or being stuck in a plateau for long period of time. (Refer to Note 12)

In summary, a good indication for an optimal learning rate is when the loss function is generally on a decreasing trend. Once you realised that your loss function has started to oscillate around some huge numbers, stop the training and switch to a smaller learning rate instead.

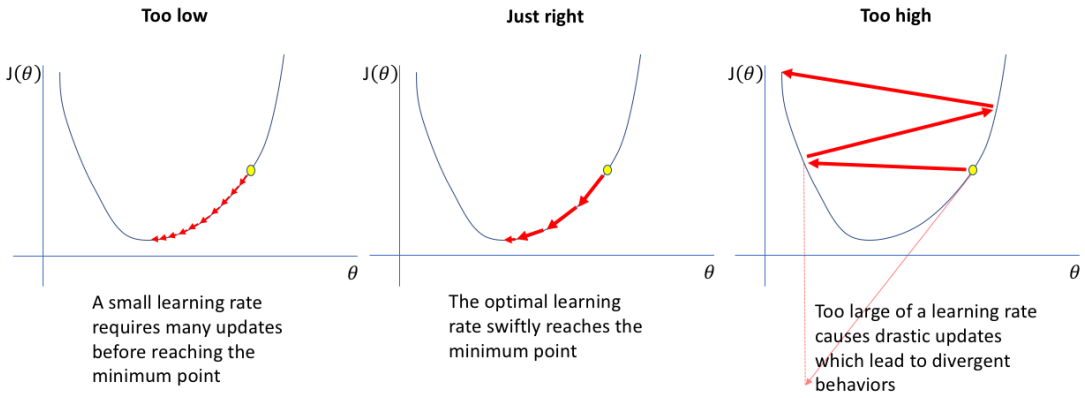


Figure 2.6: Choosing Learning Rate [24]

Note 12

Plateau is a region where the gradient of loss function w.r.t. its parameters is very close to zero for a long period of time.

Recall that in Formula 1.23, the gradient of loss w.r.t. to a parameter *(i.e.* $\nabla_{W^{[l]}}$*)* is one of the major component that defines the magnitude of step should the model take to update its parameters.

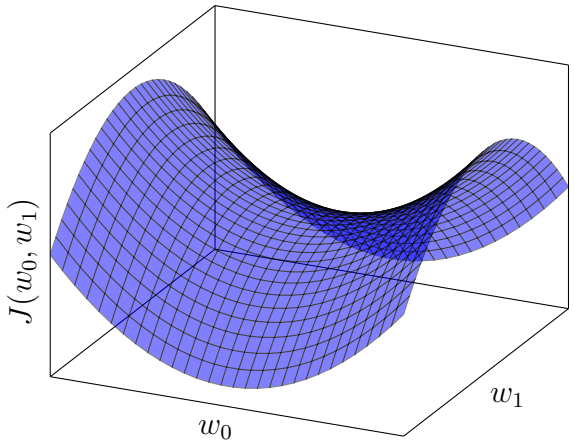


Figure 2.7: Loss Plateau

Hence, if the gradient is very close to zero, the amount of updates to the parameters of a model would be very small. As a result, the model would take a longer time moving down the plateau which in layman term, longer training time!

Note 13

Alternatively, there exist other methods to set your learning rate dynamically. Please refer to the following links for more detailed elaboration.

- Understand the Impact of Learning Rate on Neural Network Performance
- How to Configure the Learning Rate When Training Deep Learning Neural Networks

2.3 Bias and Variance in Deep Learning

In machine learning, one of the key considerations when creating a machine learning system is the **bias** and **variance** of the model.

Bias and variance measure two different sources of error in a model. [4]

Bias refers to errors made by the model due to erroneous assumptions made by the model. For example, a model that makes an assumption about the data, that does not turn out to be true, will have a high bias. In practice, neural networks tend to have a very low bias.

Variance refers to errors made by the model due to its sensitivity to variance in the training data set.

A model with high bias and low variance is said to have **underfit**. An underfit model is one that has a high error on the training data set.

A model with high variance and low bias is said to have **overfit**. An overfit model is one that has a low error on the training data set, but a high error on the validation data set. This means that the model has failed to generalize to new data.

Note 14

Note that it is possible for a model to have both high bias and variance in Deep Learning. This means that the model has high error on both the training and testing data set.

2.3.1 Dealing with High Bias

As mentioned before, high bias is not usually a problem that occurs with neural networks, but it can happen nonetheless.

It turns out that dealing with high bias is quite easy. We simply need to make the neural network more complex. This can be accomplished in a few ways:

- Increasing the number of units in each hidden layer
- Increasing the number of hidden layers
- Switching to a different type of neural network (e.g. a Convolutional Neural Network)
- Training the network for longer period of time

2.3.2 Dealing with High Variance

In general, there are a few ways to decrease the variance of a model:

- Decrease the complexity of the model (e.g. reduce the number of layers, implement techniques like regularization and dropout)
- Provide more data to the model (e.g. scraping more data online, data augmentation)

Dropout

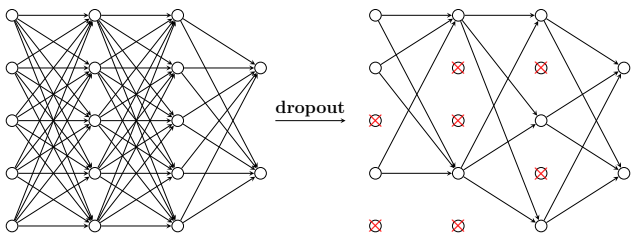


Figure 2.8: Dropout

One of the ways we can make our neural networks less prone to over fitting is to apply a technique known as **dropout** to the network. The idea of dropout is simple, during training, we will randomly remove some of the connections between the neurons. This prevents the network from using those specific connections during forward propagation. As a result, the network has to learn not to rely on any specific connections in the network during training.

Note 15

Note that during inference (i.e. making a prediction after the model is trained), dropout will be disabled, as we want to ensure all the connections of our network are used.

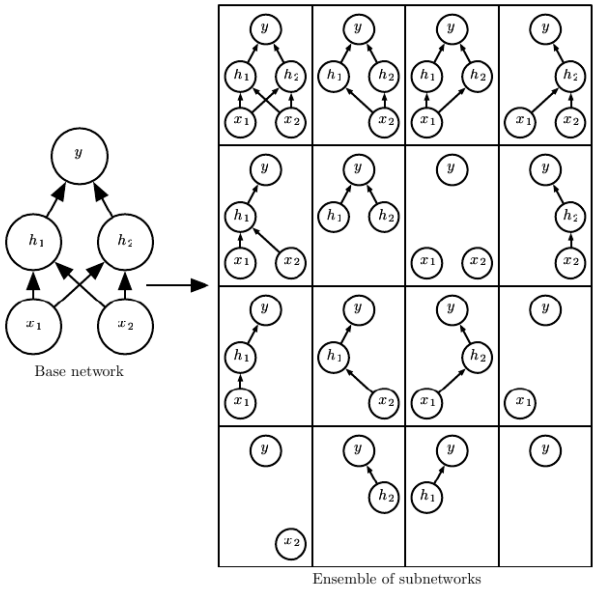


Figure 2.9: Dropout trains an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying base network.

The intuition behind why dropout helps to reduce the variance of our model is shown in Figure 2.9.

Data Augmentation

Data Augmentation is a way to easily increase the amount of training data we have, by creating fake data. The idea is to take existing data (e.g. an image), and create variations on that data by applying transformations to it.

For instance, given an image, we could flip the image, or perform a rotation on it, or applying different color filters, to make an new image as shown in Figure 2.10. By doing this, it ensures that the model would encounter a different augmented image during training (even though the "content" of the data would not change) and ultimately, the model would be more robust towards slight changes to the input data and becoming less overfitted.

Figure 2.10: Data Augmentation Techniques on Images

2.3.3 Batch Normalization

It turns out that as we are building a deeper and deeper network, it is harder for us to train it especially when the distribution of each layer's inputs keep changing during training, as the parameters of the previous layers are updated. The change in the distributions of internal nodes of a deep network, in the course of training, is also as Internal Covariate Shift. [25]

Up until this juncture, we are familarised with the notion of standardizing the input data such that it is easier for our model to learn and faster to train then. The idea of batch normalization is similar, to standardizes the inputs to a layer for each mini-batch in order to speed up the training process.

Batch normalization is able to reduce the amount of shifts of distribution for the inputs of hidden units and thus, limits the amount to which updating the parameters in earlier layers can affect distribution of values that the subsequent layer sees which helps speeding up the learning process of the whole network.

2.3.4 Transfer Learning

When training a neural network, we often require a large amount of data. For some tasks, finding enough data to train a model can be quite difficult. One way to solve it is by making use of an idea known as transfer learning.

It is an idea in deep learning that sometimes we can take knowledge a neural network has learned from one task and apply that knowledge to a similar task.

Example 4

For example, suppose I am training a neural network to detect different types of cats. Unfortunately, there just isn't enough data to train the model from scratch.

Instead, we can take a model that has already been trained on a task where there is a lot of data available. (e.g. a model for classifying dogs)

What we will do is to freeze the weights of the model, and replace the output layer. We will then train the output layer on our new task.

In fact, if I had more data, I could unfreeze the weights of the last few layers, and train them as well. This process is known as **fine tuning**.

And, if I had a lot of data, I could simply retrain the entire network, using the weights of the pre-trained model as a starting point. This is known as **pre-training**.

The intuition behind transfer learning is that the model has already learned to detect more general features from the other task (e.g how edges look like), which could still be transferable to the new task.

2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a specialized neural network designed to extract data that has a grid like structure (such as an image, which can be represented as a grid of pixels) [4]. CNNs have been employed successfully for computer vision tasks, due to their ability **to pick up patterns and features in images**.

The term "Convolutional" comes from the fact that certain layers in the network employ a mathematical operation known as **convolution**.

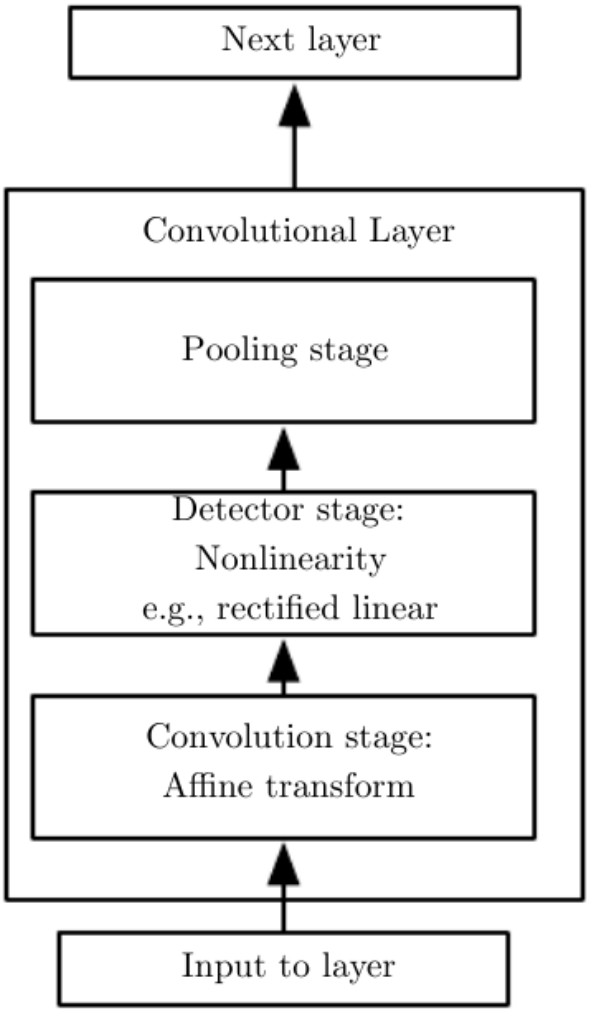


Figure 2.11: A convolutional layer is actually comprised of three stages: the convolution stage, the detector stage, and the pooling stage

2.4.1 Convolution Stage

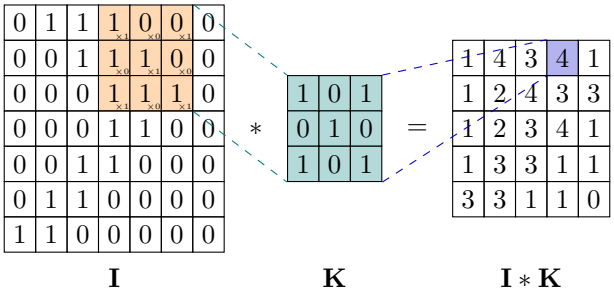


Figure 2.12: Convolution Operation

At the heart of the convolutional layer is the convolution operation. In this phase, we make use of what is known as a **kernel** or **filter**, which are convolved with the image. The kernel is basically a matrix of weights that is a trainable parameters in our model. For each image, the convolution operation works by extracting multiple patches of pixels

(e.g. 3x3 Grid) from the image to perform element-wise multiplication with the kernel. The output of the convolution process is called as the feature map.

The goal of the convolution stage is to extract useful features from the raw image or result of previous convolution stage, to form a rich **feature maps** that determines the intensity and location of a specific feature. (Refer to Visualisation from Figure 1.4 for better intuition)

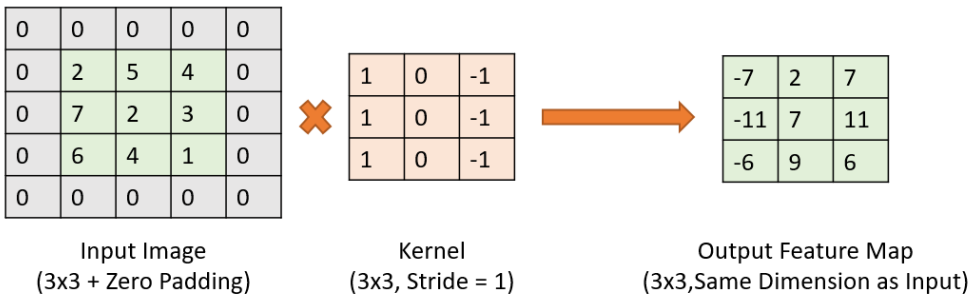
Example 5

The convolution stage works by mapping the kernels to the input images and calculate the sum of element-wise multiplication between the trainable-kernels to the input images.

For instance, Figure 2.12 shows that by mapping the green kernel, K to the input image, I, we managed to get a final output of a feature map.

The process of multiplication and summation is carried out for each steps across pixels and the number of steps for the kernel to slide through is known as **strides**.

Normally, the input images is **padded with zeros** around the image so as to maintain the dimensions of output feature maps. If the image is not padded, we will experience a very fast decrease in spatial size as we go to deeper layers and thus limiting the 'depth' of Convolution layers.



Note 16

Although we call this the convolution operation, in actuality, this operation is known as **cross-correlation**. In mathematics, cross-correlation and convolution are almost the same, but with one key difference. More precisely, a convolution in mathematics actually requires us to flip the kernel/filter relative to the input beforehand. Many machine learning libraries do not flip the kernel (since in the context of deep learning, this step is not important), and thus perform cross-correlation. However, the convention is to call both these operations convolution, and only to specify if the kernel is flipped when it actually matters. [4]

2.4.2 Non-linearity Stage

After building the feature maps (the outputs of the convolutional stage), we will pass the feature maps through a nonlinear activation function (e.g ReLU). The intuition behind the introduction of non-linearity towards our model is similar to Section 1.5, to allow our model to learn more complex representation of the image before parsing to the Dense layers for final decision making.

2.4.3 Pooling Stage

The last stage in a typical convolution operation is a pooling stage which aims to "summarize" the feature maps by performing mathematical operation like finding the maximum(i.e. max-pool) or average value(i.e. avg-pool) across multiple patches of pixels.

One of the desired characteristics of pooling is the ability to reduce the size of the representation without losing much information about the feature and its location as discovered in the convolution stage. (Section 2.4.1) This helps to speeds up computation and makes detected features more robust to small changes in the input image which is also known as translational invariance. (Refer to Note 17)

Note 17

Transitional Invariance refers to the phenomenon that our model is robust enough to ignore small shifts and translations of the object in the image. This makes sense intuitively as a picture of cat is still a cat no matters on which corner does the cat appears in the picture. One way to achieve this is through introducing the pooling stage.

The following example shows a CNN that is trained to identify the letter "L" in an image. We can notice that even if we have shifted down the letter "L" by one pixel in the second row of image, the final output of the feature maps from the max-pool layer are still the same due to the presence of max-pooling.

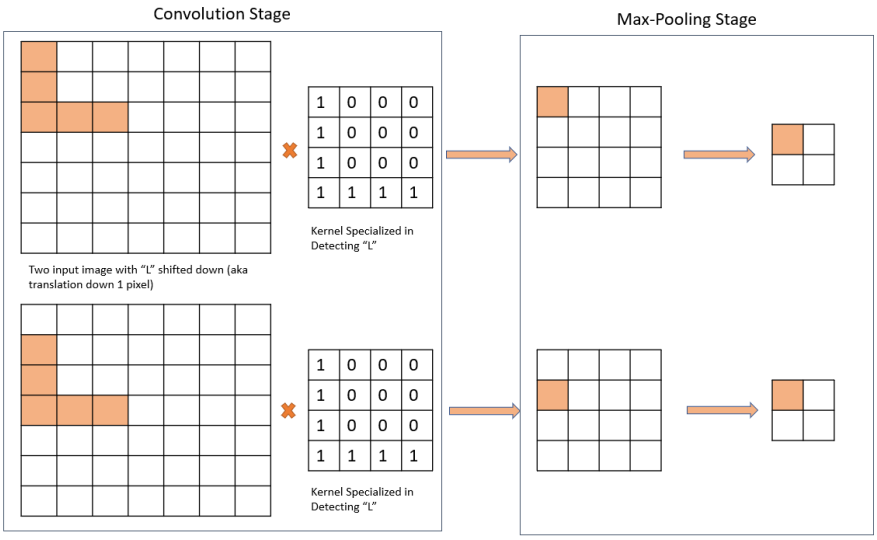


Figure 2.13: Transitional Invariance Example

2.5 Representing Text Data

A neural network cannot read raw text by itself. As such, it is necessary to encode text as a numerical matrix. There are two primary ways to encode text: one hot encoding, or word embeddings. So, let's explore how we might encode a single sentence.

2.5.1 Tokenization

Before anything, we first need to tokenize our sentence. To tokenize something means to split the sentence into units known as tokens. A token is typically a word, so tokenizing a sentence usually means that we will turn the sentence into a list of words. We will then map each token to a unique number, which is necessary for further encoding.

$$\text{Tokenizer("The movie was terrible! terrible!")} = [3,\ 1,\ 2,\ 4,\ 4] \tag{2.7}$$

From there, we have two choices on how to encode our data.

2.5.2 One Hot Encoding

The first, and easiest option is to one-hot encode our data. One-hot encoding means that each word can be represented by a vector of 0s and 1s, where each element represents the presence or absence of a word. For example, if I one hot encoded the sentence in the previous example, I would get:

$$y = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.8}$$

While this is an easy way to encode the data, we come into a problem if our vocabulary size is very big. For instance, suppose I had ten thousand unique words in my dataset, then each word would have to be represented by a 10000 dimensional vector (i.e. matrix with 10000 columns). In addition, using one hot encoding for each word, does not encode any meaning as to what the word actually means. As such, if the model were to see two words that were similar in meaning (e.g. large vs huge), then the model would treat these words as if they were completely unrelated.

2.5.3 Word Embeddings

To solve this, we can instead try to encode each word as a word embedding, a vector that encodes the meaning of the word in terms of numerical features. The idea then, is to train an embedding matrix, that can be used to transform a one-hot encoded sentence, to a matrix of word embeddings.

For instance, the previous sentence, if encoded using 3 feature word embeddings, could

be encoded like this.

$$y = \begin{bmatrix} 0.0 & 0.0 & 1.0 \\ 0.4 & 0.0 & 0.2 \\ 0.1 & 1.0 & 0.3 \\ 0.3 & 0.3 & 1.0 \\ 0.3 & 0.3 & 1.0 \end{bmatrix} \tag{2.9}$$

There are many algorithms for training word embeddings, but the most famous are the Word2Vec algorithm [26], and the GloVe algorithm [27].

Note that we can make use of the idea of transfer learning, by simply using the pre-trained we learned by these algorithms.

2.5.4 Padding Text

When dealing with sequence data, we often have to deal with inputs of variable length (i.e. each sentence have a different length or number of words). To ensure that all inputs are of the same length, we can **pad** the input sentence such that each sentence is of the same length.

To do this, we will add a special token, known as a padding token, to each sentence until it reaches a maximum length. In the case where the sentence is longer than the specified maximum length, we will simply truncate the input.

2.5.5 Special Tokens

Alongside the padding token, there are several other special tokens that we may use, namely an Unknown Word token (for any word that does not show up in the training vocabulary), and a start and end of sequence token (SOS, EOS), to signify the start and end of a sentence.

2.6 Recurrent Neural Networks

Recurrent neural networks (RNN) are specialized networks designed for sequential data. Sequential data refers to any data that is presented as a sequence (i.e. there is an ordering to the data). For instance, a sentence could be considered as a sequence of words $x^{<1>}, \ldots, x^{<T_x>}$. The main characteristic of an RNN, compared to the previous neural networks you've seen, is that they are able to remember information from other parts of the sequence.

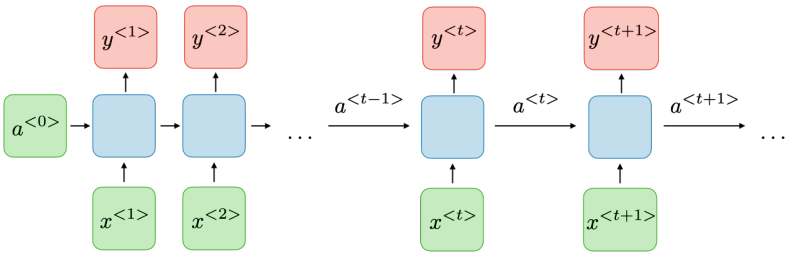


Figure 2.14: A Recurrent Neural Network [9]

2.6.1 The RNN Cell

An RNN is a network that is comprised of one or more RNN cells. A RNN cell works by processing an input sequence iteratively (e.g. I feed in each word in the sentence one at a time), and maintaining what is known as a *hidden state*, which contains information about the inputs. As each word is passed in to the network, it takes in information from the hidden state, which stores information on the previous inputs.

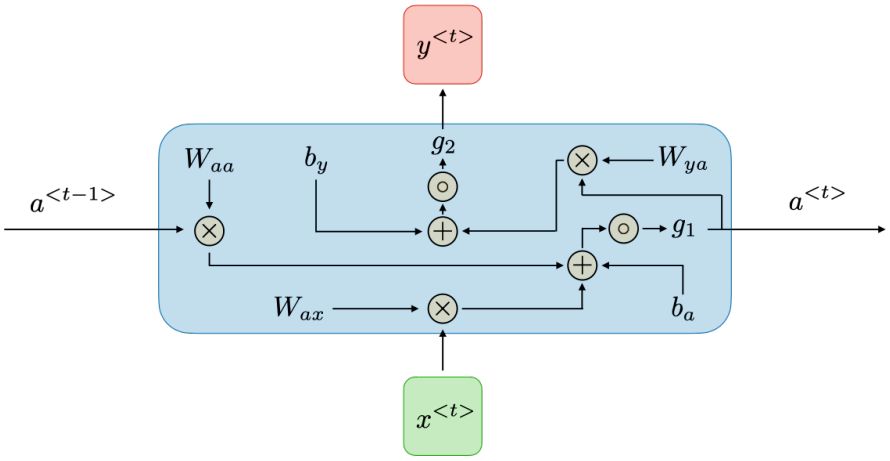


Figure 2.15: The RNN Cell in Detail [9]

To better understand how the RNN cell works, it is easier to look at the forward propagation equations for the RNN cell. At a certain time step t, the hidden state, and output of the RNN cell will be computed as follows.

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \tag{2.10}$$
$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y) \tag{2.11}$$

Basically, during the computation in the hidden layer, the model will take in the output of the hidden layer in the previous time step $t-1$ (known as the hidden state), alongside the current input. As such, as the RNN loops over each word in the sentence, it is able to retain information on the previous words it has seen in the sentence.

Since the RNN operates over a loop, it allows the RNN to process long sequences, of almost any length, without risk of the model growing too large in size. However, since we have to process each item in a sequence one by one, training and computing a prediction for an RNN is slower than other types of neural networks (e.g. CNNs, MLP).

2.6.2 Type of RNNs

It turns out that there are many types of RNNs, designed for different tasks, which we will go through in the next few subsections.

Many to One

A many to one architecture is used in tasks, where we wish to make a single prediction, given an input sequence. An example of one such task would be sentiment classification:

given a sentence (say, a review), predict if the sentiment of the sentence is positive or negative. In this case, we would build a network like that in Figure 2.16.

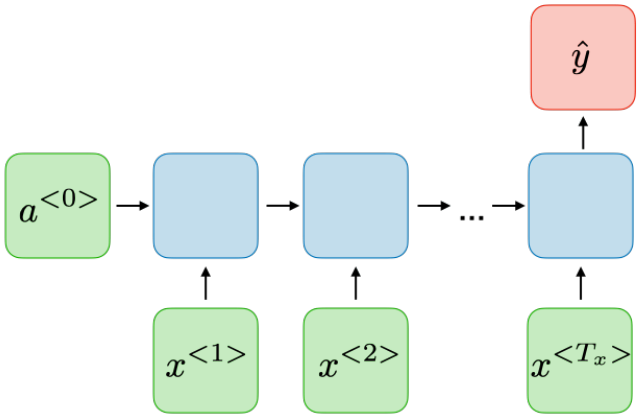


Figure 2.16: Many-to-One RNN [9]

Effectively, we feed in each word in the sentence, and make use of the output of the model at the final word.

One to Many

On the other hand, what if we don't have an input sequence at all, and simply want the network to generate a sequence for us? Say, suppose we wanted to create a model that could generate piano music by itself?

For this, we can make use of a one to many architecture, as shown in Figure 2.17. In this

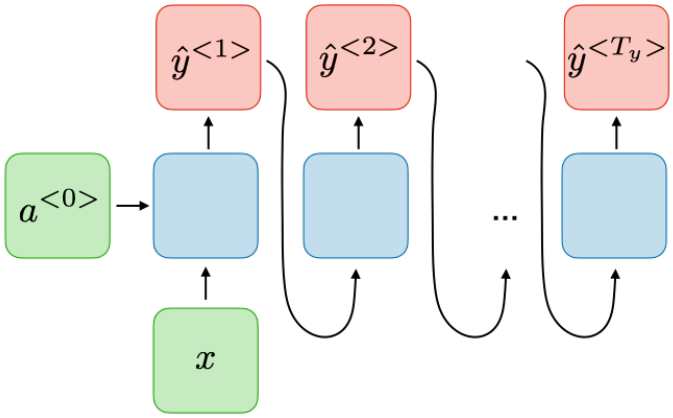


Figure 2.17: One-to-Many RNN [9]

design, we will typically feed in a random input into the RNN, which is then expected to classify the next item in the sequence (via a softmax classification head). We will then randomly sample from the output of the softmax activation, and select a predicted output. This predicted output is then used as the input to the RNN at the next time step. This process will then continue, until the network predicts the special end of sequence token (<EOS>).

Note 18

When trying to do sequence generation, we don't simply select the output of the softmax activation with the highest probability, since doing so would result in a very predictable output. As such, we choose to randomly sample from the probability distribution, such that while the most likely input has a high chance of being selected, it is also possible for other outputs to be predicted. This allows the network to be more creative in its generation.

Many to Many

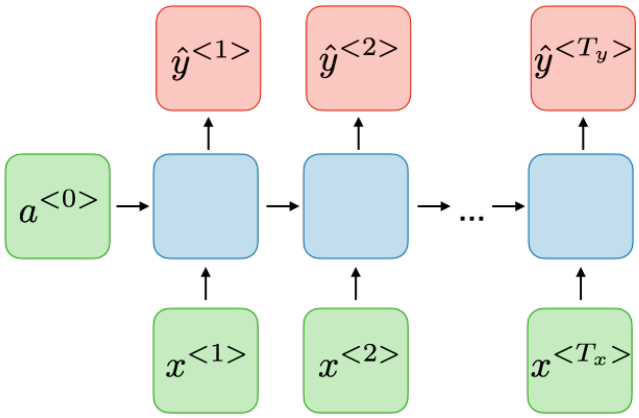


Figure 2.18: Many-to-Many RNN [9]

In a many to many RNN (see Figure 2.18), we make use of the prediction of the RNN for every input token. This means that given an input sequence of size T_x, the size of our prediction, T_y, will also be of size T_x. An example of an application that would require this architecture is a named entity recognition (NER) model. In the NER problem, we are trying to locate and classify named entities in a setence. This means, that given a sentence, we want to classify if a word in the sentence refers to a pre-defined category, such as the names of companies, locations, persons. As such, our prediction will be a token for each word in the sentence, with the token stating the category of the respective word it represents.

Encoder-Decoder

An encoder-decoder architecture (see Figure 2.19) is used for sequence-to-sequence tasks. That is, given an input sequence, attempt to predict an output sequence. Examples of such tasks include tasks like machine translation (e.g translating an English sentence into Chinese). For such tasks, it is possible that the output sequence length T_y, could be different from the input sequence length T_x.

To solve this problem, we make use of two networks. The first network, the Encoder, reads in the input sequence, and encodes it as a vector representation of that input sequence. The second network, the Decoder, reads that vector representation, and attempts to decode it, to form an output sequence (similar to a one-to-many RNN).

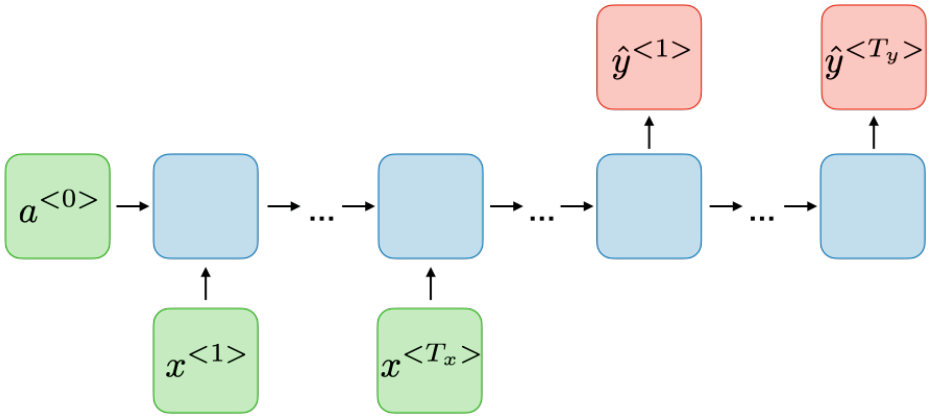


Figure 2.19: Encoder-Decoder [9]

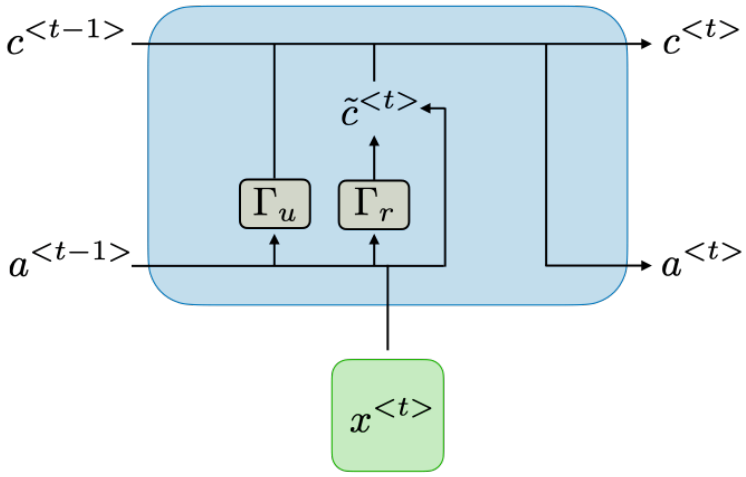


Figure 2.20: Gated Recurrent Unit [9]

2.6.3 Gated Recurrent Unit

It turns out that a basic RNN cell has trouble remembering information about past inputs that occurred very early on in the sequence. This is largely due to the vanishing gradients problem (where weights in the RNN stops being trained properly due to vanishing small gradient updates), which occurs in RNNs due to the fact that backpropagation has to occur over time, making this problem more likely when the model is trained on long inputs [28]. A Gated Recurrent Unit (see Figure 2.20) is a variant of an RNN cell tries to solve this problem.

It does this by replacing the hidden state with a gated hidden state. In a normal RNN cell, the hidden state always contains the activations of the RNN cell at the previous time step. In a GRU, a gate controls what information can be stored in the hidden state $c^{<t>}$. This effectively allows the GRU to decide when it wants to update its memory, allowing it to recall crucial information about earlier inputs.

To be more precise, let's look at the forward propagation equations. Firstly, the GRU cell will make a proposed hidden state (or candidate hidden state). In a normal RNN cell, we would pass in the hidden state of the cell and the current input. In a GRU, we will do the same thing, but multiply each value in the hidden state by a certain value (that is between 0 and 1). This has the result of blocking certain information in the hidden state

from being passed in. This is done by doing an element wise multiplication of the current hidden state matrix with a reset gate matrix Γ_r (pronounced "Gamma").

$$\widetilde{c}^{<t>} =_1 (W_{cc}\Gamma_r * c^{<t-1>} + W_{ax}x^{<t>} + b_c) \tag{2.12}$$

The reset gate itself is calculated using an equation, with the output decided by a set of weights which have to be trained.

$$\Gamma_r = \sigma(W_{ra}a^{<t-1>+W_{rx}x^{<t>}+b_r)} \tag{2.13}$$

To ensure the values in Γ_r are between 0 and 1, we make use of a sigmoid activation function. This also means that most of the values are very close to either 0 or 1, emulating the function of a gate as being either closed (0) or open(1).

The GRU will then have to decide whether or not to allow the current hidden state to be overridden by the candidate hidden state. This is done by using another gate, an update gate Γ_u, which is also calculated in the same fashion as the reset gate.

$$\Gamma_u = \sigma(W_{ua}a^{<t-1>} + W_{ux}x^{<t>} + b_u) \tag{2.14}$$

We will then use the following equation which decides if the hidden state should be updated.

$$c^{<t>} = \Gamma_u * \widetilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \tag{2.15}$$

Note that when Γ_u is high, $(1 - \Gamma_u)$ will be low. Since Γ_u is usually close to 1 or 0, the hidden state will either retain it's past values, or be overridden. Finally, we save the result as our hidden state, and use it to make our prediction (if any) at the current time step.

$$a^{<t>} = c^{<t>} \tag{2.16}$$

2.6.4 Long Short Term Memory Cell

A LSTM is a more powerful version of a GRU that is able to capture longer term dependencies. While the GRU only made use of two gates to control the flow of information through the network, an LSTM actually makes use of four gates: an update gate Γ_u, a forget gate Γ_f, a reset gate Γ_r and an output gate Γ_o.

The reset gate works exactly as it did in the GRU, but the update gate has been split into both an update gate and a forget gate. This means that instead of the update gate controlling both remembering and forgetting (Forgetting in GRU $= 1 - \Gamma_u$), we use a separate gate Γ_f. This gives the model the option of keeping the hidden state largely the same, and just updating it with new information, rather than overriding it.

$$\Gamma_f = \sigma(W_{fa}a^{<t-1>} + W_{fx}x^{<t>} + b_f) \tag{2.17}$$

$$c^{<t>} = \Gamma_u * \widetilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \tag{2.18}$$

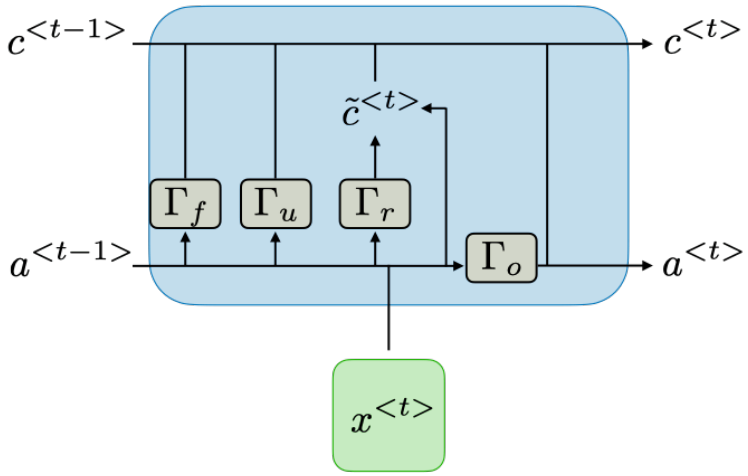


Figure 2.21: Long Short-Term Memory Cell [9]

Finally, it should be noted that the LSTM, unlike the GRU, will pass on two inputs to the next time step. This is done by applying an output gate Γ_o as follows.

$$\Gamma_o = \sigma(W_{oa}a^{<t-1>} + W_{ox}x^{<t>} + b_o) \tag{2.19}$$
$$a^{<t>} = \Gamma_o * g_1(c^{<t>}) \tag{2.20}$$

While the LSTM is more powerful than a GRU, it is also much more complex. As such it may be more difficult to train an LSTM network without overfitting if you only have a small dataset to work with.

Note 19

It should be noted that the LSTM actually came before the GRU, and that the GRU is basically a more simplified version of the LSTM.

2.7 Practical Tips for Deep Learning

This section aims to provide you some general tips and advises on things to take note of so that you could be more conscious and make more rationale and feasible decision on deciding which part invest more of your time to solving them.

2.7.1 Error Analysis

The performance of a deep learning model can be affected by more components as compared to classic ML models and oftentimes, a single value evaluation metrics like accuracy or mean squared error is not enough to properly diagnose the problems that a neural network is facing.

For that, we need to perform a thorough analysis towards the prediction of the model and identify potential issues that the model is facing. In the context of object detection (i.e. drawing a bounding box to the targeted object in an image), if we have a model

Image	Very large cat	Dog with cat-like ears	Blurry Image	Comments
1	0	1	0	Snapchat Filter
2	0	1	1	Rainy day
3	0	0	1	
...				
% Of Total	8%	43%	61%	

Example of Error Analysis

Figure 2.22: Error Analysis through Tabulation

that can identify cats and dogs in the picture, it is highly encouraged for us to visualise the prediction made for the test set to identify whether the model performs as good in identification for both animals or performs better to identify cats but worse at classifying dogs.

Figure 2.22 shows a systematic approach for us to record down the findings during error analysis of the wrongly classified examples where each column represents a possible problem with the image, and each row is a new image. Such tabulation of the errors allow us to better identify the elephant in the room in a systematic and scalable manner.

After performing error analysis, the project leader could better look into the issues and devise strategies to better improve the model's performance by, for instance, adding more pictures of dog with cat-like ears into the training data.

2.7.2 Taking a Data Centric Approach to AI

It is a common desire for us to improve the model's performance and getting better and better prediction from our model. Up until this juncture, what we have discussed thus far about choosing different network architechtures, different hyperparameters*(e.g. learning rate, optimizer)* can be classified as the **Model Centric Approach**(i.e. improve your model and traning process to get better result).

However, there exist another approach, that is more promising in most cases, to improve the model's performance, known as the **Data Centric Approach**. This consists of systematically changing/enhancing the datasets to improve the accuracy of an AI system which is usually overlooked as data collection is often treated as a one off task.[29]

The core essence of the Data Centric Approach to improving an AI model is a paradigm shift from appreciating **Big Data** (i.e. getting as much data as possible regardless of quality and consistency of the data) to **Good Data** (i.e. getting only quality data with high consistency across labels).

Up to this point, all of us have been caught up with the idea of Big Data and how getting more data is able to result in better performance of our model. While that statement is

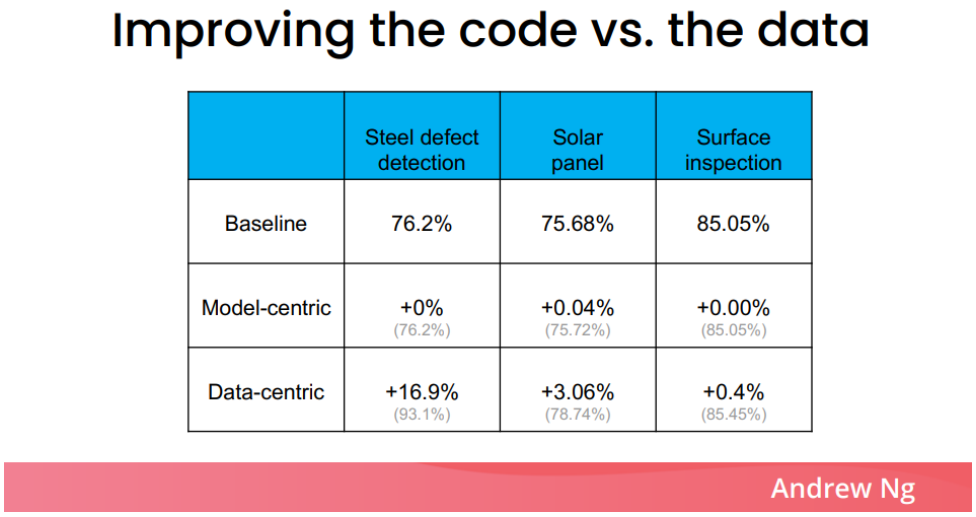


Figure 2.23: Model Centric vs Data Centric by Andrew Ng [30]

not completely wrong in the context of training a Deep Learning model, it turns out that getting high quality and clean data deserves a greater emphasise than just sheer volume of data due to the promising returns in model's performance. (Figure 2.23)

For instance, in one of the competition that we have participated, the task given is to identify the chicken in the picture. It turns out that these are the "chickens" that quietly sneak into my training data. (Figure 2.24)

It is indeed convenient for us to assume that the model will magically be able to infer that a roasted chicken is still a chicken but certainly such ambiguity will only confuse the model more. Hence, ensuring the cleanliness of the data and consistency of the labels could lead to better performance and shorter training time than the classic Model Centric Approach.

Note 20

The Model Centric Approach in AI is a new trend advocated by Andrew Ng to the AI Community. Do listen to his Presentation on YouTube[30] if you wish to know more about Data Centric AI.

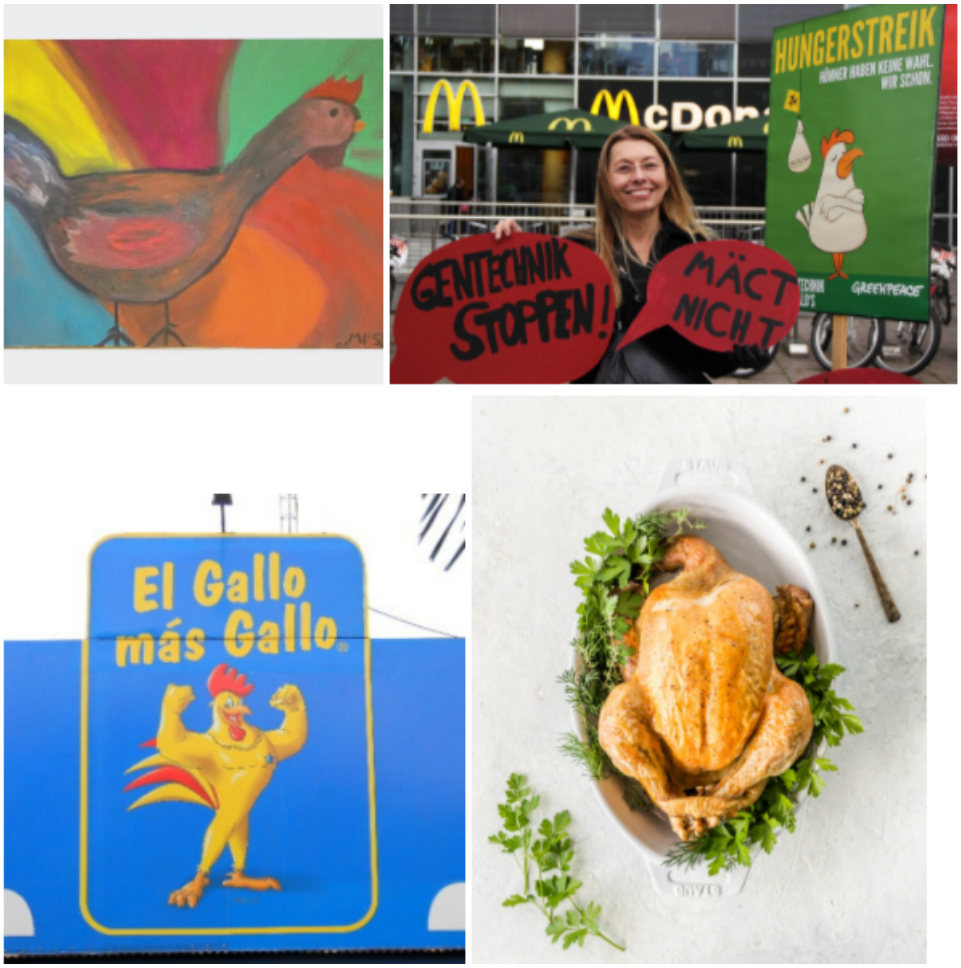


Figure 2.24: "Chicken" but not Chicken

Bibliography

[1] John McCarthy. "WHAT IS ARTIFICIAL INTELLIGENCE?" en. In: (), p. 14 (page 1).

[2] *Mechanical Turk*. en. Page Version ID: 1034693363. July 2021. URL: https://en.wikipedia.org/w/index.php?title=Mechanical_Turk&oldid=1034693363 (visited on 10/03/2021) (page 1).

[3] *What is the difference between a neural network and a deep neural network, and why do the deep ones work better?* URL: https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w (visited on 10/06/2021) (pages 2, 22).

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016 (pages 2, 9–11, 19, 21, 22, 32, 36, 38).

[5] *neural networks - Does Keras SGD optimizer implement batch, mini-batch, or stochastic gradient descent?* URL: https://stats.stackexchange.com/questions/406183/does-keras-sgd-optimizer-implement-batch-mini-batch-or-stochastic-gradient-des (visited on 10/08/2021) (pages 2, 29).

[6] *Deep Learning*. en. URL: https://www.coursera.org/specializations/deep-learning (visited on 10/03/2021) (page 3).

[7] *Artificial Intelligence vs. Machine Learning vs. Deep Learning | Towards Data Science*. URL: https://towardsdatascience.com/artificial-intelligence-vs-machine-learning-vs-deep-learning-2210ba8cc4ac (visited on 10/14/2021) (page 4).

[8] *A Primer on Deep Learning*. en-US. URL: https://www.datarobot.com/blog/a-primer-on-deep-learning/ (visited on 10/06/2021) (page 4).

[9] *CS 230 - Recurrent Neural Networks Cheatsheet*. URL: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks (visited on 10/14/2021) (pages 7, 41–45, 47).

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *arXiv:1512.03385 [cs]* (Dec. 2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (visited on 10/03/2021) (pages 10, 22).

[11] DeepLearningAI. *Why Non-linear Activation Functions (C1W3L07)*. Aug. 2017. URL: https://www.youtube.com/watch?v=NkOv_k7r6no (visited on 10/03/2021) (page 11).

[12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". en. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. ISSN: 1938-7228. JMLR Workshop and Conference Proceedings, June 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html (visited on 10/04/2021) (page 12).

[13] Sun Tzu. *The Art Of War*. English. First Thus edition. S.l: Filiquarian, Nov. 2007. ISBN: 978-1-59986-977-3 (page 16).
[14] *3Blue1Brown*. URL: https://www.3blue1brown.com/topics/3blue1brown.com (visited on 10/05/2021) (page 19).
[15] *4.7. Forward Propagation, Backward Propagation, and Computational Graphs — Dive into Deep Learning 0.17.0 documentation*. URL: https://d2l.ai/chapter_multilayer-perceptrons/backprop.html#backpropagation (visited on 10/05/2021) (page 20).
[16] Roger Grosse. "Lecture 5: Multilayer Perceptrons". en. In: (), p. 7 (page 22).
[17] Ok_Slice4231. *[D] Has the ResNet Hypothesis been debunked?* Reddit Post. Sept. 2021. URL: www.reddit.com/r/MachineLearning/comments/px3hzd/d_has_the_resnet_hypothesis_been_debunked/ (visited on 10/06/2021) (page 22).
[18] *TechViz - The Data Science Guy: Multi-Label classification with One-Vs-Rest strategy*. Feb. 2019. URL: https://prakhartechviz.blogspot.com/2019/02/multi-label-classification-python.html (visited on 10/11/2021) (page 25).
[19] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. 2013 (page 26).
[20] *machine learning - Is using batch size as 'powers of 2' faster on tensorflow?* URL: https://stackoverflow.com/questions/44483233/is-using-batch-size-as-powers-of-2-faster-on-tensorflow (visited on 10/08/2021) (pages 29, 30).
[21] Jingru Guo. *AI Notes: Parameter optimization in neural networks*. URL: https://www.deeplearning.ai/ai-notes/optimization/ (visited on 10/08/2021) (page 29).
[22] *An overview of gradient descent optimization algorithms*. URL: https://ruder.io/optimizing-gradient-descent/ (visited on 10/12/2021) (page 30).
[23] *CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/neural-networks-3/ (visited on 10/13/2021) (page 30).
[24] *Setting the learning rate of your neural network*. en. Mar. 2018. URL: https://www.jeremyjordan.me/nn-learning-rate/ (visited on 10/13/2021) (page 31).
[25] Jason Brownlee. *A Gentle Introduction to Batch Normalization for Deep Neural Networks*. en-US. Jan. 2019. URL: https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/ (visited on 10/14/2021) (page 35).
[26] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space". In: *arXiv:1301.3781 [cs]* (Sept. 2013). arXiv: 1301.3781 version: 3. URL: http://arxiv.org/abs/1301.3781 (visited on 10/14/2021) (page 41).
[27] Jeffrey Pennington, Richard Socher, and Christopher Manning. "Glove: Global Vectors for Word Representation". en. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: http://aclweb.org/anthology/D14-1162 (visited on 10/14/2021) (page 41).
[28] *The Vanishing Gradient Problem in Recurrent Neural Networks*. en. URL: https://nickmccullum.com/python-deep-learning/vanishing-gradient-problem/ (visited on 10/18/2021) (page 45).
[29] *From Model-centric to Data-centric Artificial Intelligence | by Urwa Muaz | Towards Data Science*. URL: https://towardsdatascience.com/from-model-

centric-to-data-centric-artificial-intelligence-77e423f3f593 (visited on 10/13/2021) (page 48).

[30] DeepLearningAI. *A Chat with Andrew on MLOps: From Model-centric to Data-centric AI*. Mar. 2021. URL: https://www.youtube.com/watch?v=06-AZXmwHjo (visited on 10/13/2021) (page 49).

Appendix A

Notation for Deep Learning

The notation we are using is based of the notation used in the CS230 Deep Learning module by Stanford University.

A.1 General

- superscript (i) denotes the i^{th} training example
- superscript [l] denotes the l^{th} layer

A.2 Sizes

- m : number of examples in the dataset
- n_x : input size
- n_y : output size
- $n_h^{[l]}$: number of hidden units of the l^{th} layer
- L : number of layers in the network

Appendix B

Further Reading

This document is only meant to provide the reader with a basic overview of deep learning. As such, if you wish to dive deeper into the topic, we recommend that you look out for the following books/courses.

- Deep Learning
- Dive into Deep Learning
- Deep Learning Specialization