Deep Learning Introduction And Applications

Mohamad Fakhouri
Supervised by Gökhan Yıldırım

August 25, 2020

Abstract

In this report, we study the mathematical foundations of neural networks. In particular, we focus on convolutional neural networks and consider some applications. We also study a method that is used to learn from data that is available in the form of a graph.

1 Introduction

Deep learning is a branch of machine learning that is inspired by the biological brain. It models the brain by formalizing layers and neurons as we will see later on in this report. To motivate neural networks, we begin by describing an example.

Assume that an employee at a bank is asked to design an algorithm, or a tool, that predicts whether a customer's loan application will bring some profit to the bank. Each customer has some features, such as age, customer-age, current funds, average income, credit card score, and so on. The bank provides the employee with historical loan data that describes each application and whether that application made profit in the future or not. The employee then aims to design an algorithm, in place of the black box in Figure 1.

In this example, the input can be modeled by a vector $x \in \mathbb{R}^6$ and the associated output is a value $y \in \{0, 1\}$. To design this black box, the employee needs to set up a model and then train it. Neural Networks are one of the many possible models that the employee can choose from. Using the previous loan data (a set of vectors of inputs and the real outputs), the employee can train this network to learn the patterns, and the rules, that the bank uses to decide whether to grant the loan or not without actually having to write the rules out themselves.

A lot of data that is available nowadays is in a picture format. Recently, Convolutional Neural Networks have proven themselves to be an important tool in solving image-related data. However, many data is also available in a non-organized fashion as graphs. Such graphs can represent social networks, connections, and more. We explore a technique that allows us to learn from graph data in the later sections.

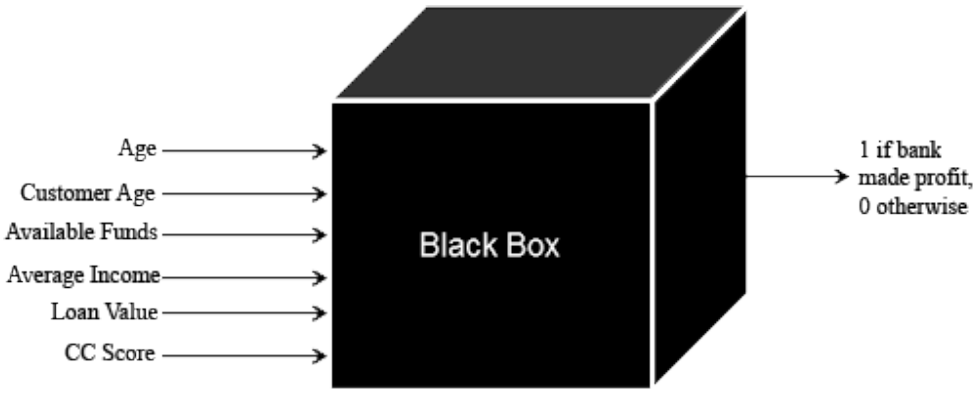

Figure 1: Black box Model

2 Neural Networks

Earlier, we mentioned how Neural Networks can be used to address the bank-loan example. Now, we describe what a Neural Network is in detail. A Neural Network (NN) is computational system, inspired by the biological brain, that understands and translates an input data to an output of (usually) a different form [1]. NNs have three fundamental components: neurons, activation functions, and layers, which we will see in the subsequent sections. In this section, we follow the presentation of [2].

2.1 Neurons

Let $n \in \mathbb{N}$, $b \in \mathbb{R}$, $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, $w = (w_1, \ldots, w_n) \in \mathbb{R}^n$. b is called the **bias**, x is called the **input** vector and w is called the **weight** vector. The neurons apply the NN's **activation function** $\sigma \colon \mathbb{R} \mapsto \mathbb{R}$ and compute the output, y, as follows:

$$y = \sigma(x \cdot w + b) = \sigma(\sum_{i=1}^{n} x_i w_i + b) \tag{1}$$

2.2 Activation Function

The activation function is a nonlinear function that the network uses to transform inputs. The function has to be nonlinear; otherwise, the entire network can be represented by a linear (simple) function which is not desired. To allow the network to be able to learn complex patterns, activation functions must be nonlinear. We will mainly focus on one example of an activation function: the sigmoid function. Let $t \in R$, then the sigmoid function σ is a differentiable function defined by

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{2}$$

The sigmoid function's derivative can be expressed by itself as follows:

$$\sigma'(t) = \sigma(t)(1 - \sigma(t)) \tag{3}$$

We will extend this definition to vectors and matrices by applying the sigmoid function element wise. Let $n, m \in \mathbb{N}$. We will use σ for all of these functions - which of these functions can be inferred from the context. Let $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Then we define $\sigma \colon \mathbb{R}^n \mapsto \mathbb{R}^n$ by

$$(\sigma(x))_i = \sigma(x_i) \tag{4}$$

Similarly, we extend the definition to real matrices. Let $X \in \mathbb{R}^{m \times n}$. We define $\sigma \colon \mathbb{R}^{m \times n} \mapsto \mathbb{R}^{m \times n}$ by

$$(\sigma(X))_{ij} = \sigma(X_{ij}) \tag{5}$$

where A_{ij} is the element of the matrix A at the i^{th} row and j^{th} column. This notation will be useful as we discuss layers later on.

Another activation function that is commonly used is the ReLU (Rectified Linear Unit) function. It is given by $\sigma(x) = \max(0, x)$. In our report, we will focus on the sigmoid function unless specified.

2.3 Layers

We can now start by building what a layer is. Assume that we have m neurons in a layer and that the layer takes $x \in \mathbb{R}^n$ as an input. Then the layer produces the output $y = (y_1, \ldots, y_m) \in \mathbb{R}^m$ by making each neuron apply the activation function with its respective weights and biases - meaning that y_i is the output of the i^{th} neuron when given the input x. A network simply consists of $L \in \mathbb{N}$ layers, where the 1^{st} layer is called the input layer, the L^{th} layer is the output layer, and the layers $2, \ldots, L-1$ are called the hidden layers.

We now describe how we keep track of the weights and biases using matrices. Let $l \in \mathbb{N}$ such that $2 \leq l \leq L$. Suppose that the l^{th} layer contains n_l neurons. Then n_1 is the dimension of the input, and n_l is the number of neurons (and number of outputs) of the l^{th} layer. Hence, the network maps $x \in \mathbb{R}^{n_1}$ to $y \in \mathbb{R}^{n_L}$.

Each neuron contains one weight for each output of the previous layer. For layer l, we denote the weights by $W^{[l]} \in \mathbb{R}^{n_l, n_{l-1}}$. Hence, $w_{ij}^{[l]}$ is the weight that the i^{th} neuron in layer l applies to the j^{th} neuron in layer $l-1$. Each neuron in the l^{th} layer also has one bias. We represent the biases of the l^{th} layer by $b^{[l]} \in \mathbb{R}^{n_l}$.

Assume that the network takes in $x \in \mathbb{R}^{n_1}$ as input. We introduce $a^{[l]} \in \mathbb{R}^{n_l}$ to represent the output of the l^{th} layer. Then, we can summarize what the network is doing by the following recurrence relation:

$$a^{[1]} = x \in \mathbb{R}^{n_1} \tag{6}$$

$$a^{[l]} = \sigma(W^{[l]} a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l} \quad \text{for } 1 < l \leq L \tag{7}$$

2.4 Learning from Data

Assume we are working with a data set that has $N \in \mathbb{N}$ samples given by $\{(x^i, y^i)\}_{i=1}^N$ where $x^i \in \mathbb{R}^{n_1}$ and $y^i \in \mathbb{R}^{n_L}$. We now define a function that measures the prediction error that the network makes with the current weights and biases. We will call this function the **cost function**. We will focus on the following cost function:

$$\text{Cost(weights, biases)} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}||y^i - a^{[L]}(x^i)||_2^2 \tag{8}$$

where $a^{[L]}(x^i)$ is the output of the network when x^i is passed as the input. We write Cost(weights, biases) informally; however, we mean that Cost is a function of $w_{ij}^{[l]}$ for all valid i, j, l and $b_j^{[l]}$ for all valid j, l. We aim to *train* the network to on this data set. Training the network means finding the weights and biases which minimize the cost function. Hence, the problem can be written as

$$\text{(weights, biases)}^* = \underset{\text{(weights, biases)}}{\arg\min}\ \text{Cost(weights, biases)} \tag{9}$$

We can employ Gradient Descent or Stochastic Gradient Descent, which are discussed in Section 2.5, to solve this problem. However, to do that, we would need to calculate the partial derivatives of the cost function with respect to $w_{ij}^{[l]}$ and $b_j^{[l]}$. We also focus on the cost function with respect to one sample only since we can write

$$\text{Cost(weights, biases)} = \frac{1}{N}\sum_{i=1}^{N} C_i \tag{10}$$

where $C_i = \frac{1}{2}||y^i - a^{[L]}(x^i)||_2^2$. For now, we drop the index on C and just focus on

$$C = \frac{1}{2}||y - a^{[L]}||_2^2 \tag{11}$$

Hence, we need to compute $\frac{\partial C}{\partial w_{ij}^{[l]}}$ and $\frac{\partial C}{\partial b_j^{[l]}}$ for all i, j, l. We already have a variable to denote the output of the l^{th} layer ($a^{[l]}$). Now, we introduce a new notation for the *weighted input* that will simplify the calculations. Let

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l} \quad \text{for } 1 < l \leq L \tag{12}$$

Then the network can be described as

$$a^{[1]} = x \tag{13}$$

$$a^{[l]} = \sigma(z^{[l]}) \quad \text{for } 1 < l \leq L \tag{14}$$

Now, we also introduce a new variable to describe how much the cost function is affected by little changes in the input - or the sensitivity of the cost function to the input at that specific neuron. Let $\delta^{[l]} \in \mathbb{R}^{n_l}$ be given by

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} \quad \text{for } 1 < l \leq L \text{ and } 1 \leq j \leq n_l \tag{15}$$

Theorem 2.1 (Backpropagation)**.** *Let $\circ$ denote the Hadamard (pairwise) product. Then, in the setting defined above, we have the following:*

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^{[L]} - y) \tag{16}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \qquad \text{for } 1 < l \leq L-1 \tag{17}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for } 1 < l \leq L \tag{18}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for } 1 < l \leq L \tag{19}$$

Before we begin the proof, we need the following lemma.

Lemma 2.2. *We can write $\frac{\partial C}{\partial z_i^{[l]}}$ as follows:*

$$\frac{\partial C}{\partial z_i^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_i^{[l]}}$$

Proof. Consider $z_j^{[l+1]}$. This is the j^{th} input at the $l+1^{th}$ layer and, using 12, it can be written as

$$z_j^{[l+1]} = \sum_{s=1}^{n_l} w_{js}^{[l+1]} \sigma(z_s^{[l]}) + b_j^{[l+1]} \tag{20}$$

Hence, each $z_j^{[l+1]}$ depends on every input of the previous layer. Hence, we can write

$$\frac{\partial C}{\partial z_i^{[l]}} = \frac{\partial C}{\partial z_1^{[l+1]}} \frac{\partial z_1^{[l+1]}}{\partial (z_i^{[l]}} + \cdots + \frac{\partial C}{\partial z_{n_{l+1}}^{[l+1]}} \frac{\partial z_{n_{l+1}}^{[l+1]}}{\partial z_i^{[l]}} \tag{21}$$

$$= \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_i^{[l]}} \tag{22}$$

□

Equipped with Lemma 2.2, we can now prove Theorem 2.1 where we follow the proof in [2].

Proof. The proof heavily depends on the Chain Rule. We begin by proving 16. By 14 and 4, we have $a_j^{[l]} = \sigma(z_j^{[l]})$. Hence, we have $\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \sigma'(z_j^{[l]}) \frac{\partial z_j^{[l]}}{\partial z_j^{[l]}} = \sigma'(z_j^{[l]})$. Now, by 11, we can write $C = \frac{1}{2} \sum_{i=1}^{n_l} (y_k - a_k^{[l]})^2$. Hence,

$$\begin{aligned}
\frac{\partial C}{\partial a_j^{[l]}} &= \frac{1}{2} \frac{\partial}{\partial a_j^{[l]}} \sum_{i=1}^{n_l} (y_k - a_k^{[l]})^2 \\
&= \frac{1}{2} (0 + \cdots + 2(y_j - a_j^{[l]}) \frac{\partial}{\partial a_j^{[l]}} (-a_j^{[l]}) + \cdots + 0) \\
&= -(y_j - a_j^{[l]}) = a_j^{[l]} - y_j
\end{aligned}$$

Then, by 15, we get

$$\begin{aligned}\delta_j^{[l]} &= \frac{\partial C}{\partial z_j^{[l]}} = \frac{\partial C}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \\ &= (a_j^{[l]} - y_j)\sigma'(z_j^{[l]}) = \sigma'(z_j^{[l]})(a_j^{[l]} - y_j)\end{aligned}$$

Hence, we have 16.

To show 17, we use Lemma 2.2 and focus on $\frac{\partial C}{\partial z_k^{[l+1]}}$ and $\frac{\partial z_k^{[l+1]}}{\partial z_i^{[l]}}$. Note that by 15, we have $\delta_k^{[l+1]} = \frac{\partial C}{\partial z_k^{[l+1]}}$. For the second term, we apply the Chain Rule by using the definition of $z_k^{[l+1]}$ from 12. We get:

$$\begin{aligned}\frac{\partial z_k^{[l+1]}}{\partial z_i^{[l]}} &= \frac{\partial}{\partial z_i^{[l]}}(W^{[l+1]}a^{[l]} + b^{[l+1]})_k \\ &= \frac{\partial}{\partial z_i^{[l]}}\left(\sum_{s=1}^{n_l} w_{ks}^{[l+1]} a_s^{[l]} + b_k^{[l+1]}\right) \\ &= \frac{\partial}{\partial z_i^{[l]}}\left(\sum_{s=1}^{n_l} w_{ks}^{[l+1]} \sigma(z_s^{[l]}) + b_k^{[l+1]}\right) \\ &= (0 + \cdots + w_{ki}^{[l+1]}\sigma'(z_i^l) + \cdots + 0) + 0 \\ &= w_{ki}^{[l+1]}\sigma'(z_i^l)\end{aligned}$$

Hence, by Lemma 2.2, we have

$$\begin{aligned}\delta_i^{[l]} = \frac{\partial C}{\partial z_i^{[l]}} &= \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_i^{[l]}} \\ &= \sum_{k=1}^{n_{l+1}} \left(\delta_k^{[l+1]}\right)\left(w_{ki}^{[l+1]}\sigma'(z_i^l)\right) \\ &= \sigma'(z_i^l)\sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{ki}^{[l+1]} \\ &= \sigma'(z_i^l)\left((W^{[l+1]})^T \delta^{[l+1]}\right)_i\end{aligned}$$

where the last line defines the elements of the vector in equation 17 as needed.

To show 18, we begin by noting that $\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$ which we get by differentiating 12 with respect to $b_j^{[l]}$. Hence, we get

$$\begin{aligned}\frac{\partial C}{\partial b_j^{[l]}} &= \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} \\ &= \delta_j^{[l]}\end{aligned}$$

which implies 18.

To show 19, note that $z_j^{[l]}$ and $w_{jk}^{[l]}$ are related by

$$z_j^{[j]} = \sum_{s=1}^{n_{l-1}} w_{js}^{[l]} a_s^{[l-1]} + b_k^{[l]}$$

Hence, we have

$$\begin{aligned}
\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} &= \frac{\partial}{\partial w_{jk}^{[l]}} \left(\sum_{s=1}^{n_{l-1}} w_{js}^{[l]} a_s^{[l-1]} + b_k^{[l]} \right) \\
&= (0 + \cdots + \frac{\partial}{\partial w_{jk}^{[l]}} (w_{jk}^{[l]} a_k^{[l-1]}) + \cdots + 0) \\
&= a_k^{[l-1]}
\end{aligned}$$

Hence,

$$\begin{aligned}
\frac{\partial C}{\partial w_{jk}^{[l]}} &= \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} \\
&= \delta_j^{[l]} a_k^{[l-1]}
\end{aligned}$$

which shows 19. □

We now can compute the partial derivative, and we are ready to formulate the learning algorithm. Essentially, we need to do two passes: one *forward pass* to calculate the δ's (which uses the inputs to and pushes them through the network), and one *backward pass* to update the weights and biases. For an algorithm pseudocode, please see [2].

2.5 Training of A Neural Network

For simplicity, we will assume that we are dealing with a convex optimization problem where we need to find w^* such that

$$w^* = \arg\min_w f(w)$$

Assume that we denote the weight vector in the i^{th} iteration by $w^{(i)}$. Let $\mu > 0$ be the step size, and let $\epsilon > 0$ be our difference that we will use as a stopping criteria. The Gradient Descent algorithm is shown in Algorithm 1.

Algorithm 1 Gradient Descent

```
initialize w^(1) randomly
for k = 1, 2, ... do
  w^(k+1) = w^(k) − μ∇f|_{w^(k)}
  if ||w^(k+1) − w^(k)|| < ε then
    stop
  end if
end for
```

The choice of μ is important; if μ is chosen to be too large, the algorithm diverges. If μ is too small, then the algorithm takes a large number of iterations. If f is a convex function, then we can guarantee that we can find a good μ such that the algorithm converges to a global minimum [3]. If f is a nonconvex function that is bounded from below, we can guarantee that the algorithm converges to a saddle point or a local minimum (almost surely) [4]

We will consider an example of applying GD on the square loss function f. Assume that we know the real output values y, and we have their corresponding input values in a matrix X. Further assume that $f(w) = ||y - Xw||_2^2$. Then, given a weight vector w, the function f returns the error of the predicted value Xw. f is a convex function by Theorem A.7.

Theorem 2.3. *In the setting above, there exists a μ such that the sequence $f(w^{(1)}), f(w^{(2)}), \ldots$ converges to a global minimum.*

In the proof of the following theorem, we follow the video lectures of Professor Rebecca Willet [5].

Proof. We know that f is bounded below. Hence, all we need to do is find μ such that for every k, $f(w^{(k+1)}) < f(w^{(k)})$. Assume that the largest singular value of X is σ_1. Let $\mu < \frac{1}{\sigma_1^2}$. Remember that $f(w) = ||y - Xw||_2^2 = (y - Xw)^T(y - Xw) = y^Ty - 2w^TX^Ty + w^TX^TXw$, hence $\nabla f = -X^Ty + 2X^TXw$. The update rule then becomes $w^{(k+1)} = w^{(k)} - 2\mu X^T(Xw^{(k)} - y)$. We want

$$\begin{aligned}
f(w^{(k+1)}) &= \|y - Xw^{(k+1)}\|_2^2 \\
&= \|y - Xw^{(k)} + 2\mu XX^T(Xw^{(k)} - y)\|_2^2 \\
&= \|y - Xw^{(k)}\|_2^2 + 4\mu^2\|XX^T(Xw^{(k)} - y\|_2^2 + 4\mu(y - Xw^{(k)})^TXX^T(Xw^{(k)} - y) \\
&= \|y - Xw^{(k)}\|_2^2 + 4\mu^2\|XX^T(Xw^{(k)} - y\|_2^2 - 4\mu(X^T(Xw^{(k)} - y))^TX^T(Xw^{(k)} - y) \\
&= f(w^{(k)}) + 4\mu^2\|XX^T(Xw^{(k)} - y\|_2^2 - 4\mu\|X^T(Xw^{(k)} - y)\|_2^2
\end{aligned}$$

By Theorem A.3, we have

$$\|XX^T(Xw^{(k)} - y\|_2^2 \leq \sigma_1^2\|X^T(Xw^{(k)} - y\|_2$$

Hence, we get

$$\begin{aligned}
f(w^{(k+1)}) &= f(w^{(k)}) + 4\mu^2\|XX^T(Xw^{(k)} - y\|_2^2 - 4\mu\|X^T(Xw^{(k)} - y)\|_2^2 \\
&\leq f(w^{(k)}) + 4\mu^2\sigma_1^2\|X^T(Xw^{(k)} - y\|_2 - 4\mu\|X^T(Xw^{(k)} - y)\|_2^2 \\
&= f(w^{(k)}) + 4\mu(\mu\sigma_1^2 - 1)\|X^T(Xw^{(k)} - y)\|_2^2 \\
&< f(w^{(k)})
\end{aligned}$$

where the inequality comes from the fact that $0 < \mu < \frac{1}{\sigma_1^2}$ which implies that

$$\mu\sigma_1^2 - 1 = \frac{1}{\sigma_1^2}\sigma_1^2 - 1 < 1 - 1 = 0$$

and that $\|X^T(Xw^{(k)} - y)\|_2^2 \geq 0$ Hence, we know that the algorithm converges to a minimum, say, w^*. Since f is a convex function, by Theorem A.8, w^* is also a global minimum. □

The main issue with using Gradient Descent is that in every iteration, reading all the sample data is required to calculate ∇f, which is an $O(n)$ operation where n is the number of samples. For this reason, Stochastic Gradient Descent is used. Moreoevr, Stochastic Gradient Descent is known to escape saddle points [6]. The difference is that instead of using all the samples to calculate the ∇f, we approximate its value using only one sample. Assume that we have n many samples and that we can write

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$$

Each f_i represents the cost of the i^{th} sample. We outline the algorithm in Algorithm 2.

Algorithm 2 Stochastic Gradient Descent

initialize $w^{(1)}$ randomly
for $k = 1, 2, \ldots$ **do**
  select $t_k \in \{1, 2, \ldots, n\}$
  $w^{(k+1)} = w^{(k)} - \mu \nabla f_{t_k}|_{w^{(k)}}$
  if $||w^{(k+1)} - w^{(k)}|| < \epsilon$ **then**
    stop
  end if
end for

The versions of the Stochastic Gradient Descent algorithm differ by how they select t_k. Sampling $t_k \sim Unif(n)$ is appealing because then we would have

$$\begin{aligned}
\mathbb{E}[\nabla f_{t_k}(w)] &= \sum_{i=1}^{n} \frac{1}{n} \nabla f_i(w) \\
&= \nabla \sum_{i=1}^{n} \frac{1}{n} f_i(w) \\
&= \nabla f(w)
\end{aligned}$$

However, other choices for the choice of the selection methods exist. For discussion on these methods, and for convergence analysis, please see [7].

The activation function used here was assumed to be differentiable. Hence, it is worth it to consider the case where ReLU is used as an activation function (since it is not differentiable everywhere). In the case of non-differentiable functions, the GD and SGD algorithms can also be used by replacing the gradient with a *subgradient*. For more information, please see [8]

3 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a NN that uses convolution instead of full matrix multiplication in at least one layer. Full matrix multiplication means that every neuron

uses every output of the previous layer; CNNs do not necessarily do that thanks to the convolution operation. We will explore the convolution operation first and then see how it is applied in CNNs. Here, we mainly follow The Deep Learning book's introduction and examples [9]. The classical convolution operation is defined as an integral. For two functions $x, w\colon \mathbb{R} \mapsto \mathbb{R}$, a new function $y\colon \mathbb{R} \mapsto \mathbb{R}$ is defined as follows:

$$y(t) = \int_{-\infty}^{\infty} x(\tau)w(t-\tau)d\tau \tag{23}$$

In our context, x is called the *input*, w is called the *kernel*, and the output y is called a *feature map*.

We can also define a discrete convolution. Let $x, w\colon \mathbb{Z} \mapsto \mathbb{R}$, then we can define a new function $y\colon \mathbb{Z} \mapsto \mathbb{R}$ as follows:

$$y(t) = \sum_{i=-\infty}^{\infty} x(i)w(t-i) \tag{24}$$

If x, w's support is a finite subset of $\mathbb{Z}$, then y is well defined.

In machine learning, data is typically a multidimensional array of data called **tensor**

Definition 3.1. Let $m \in \mathbb{N}$. Let $N\colon \underbrace{\mathbb{N} \times \cdots \times \mathbb{N}}_{m \text{ times}} \mapsto \mathbb{R}$ such that $\text{supp}(N)$ is finite. Then N is an m-dimensional tensor.

These tensors can be to represent many types of input data. For example, a $2D$ tensor can be used to represent a black and white image, where each $N(i, j)$ represents the luminosity at the $(i, j)^{th}$ pixel. For a simple example of a 5×6 BW (black and white) image, please see Figure 2

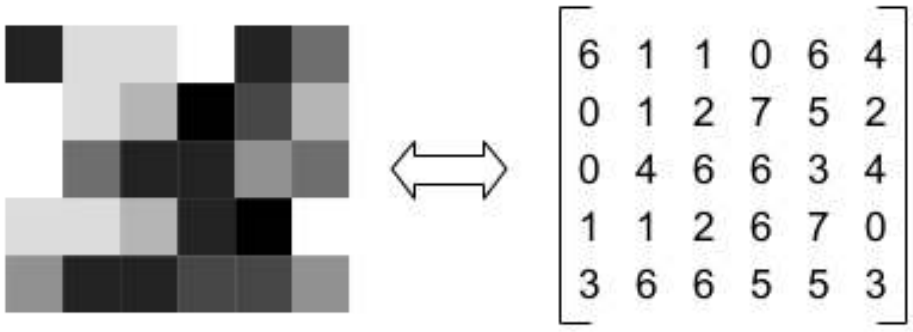

Figure 2: BW Image with its matrix representation

An $m \times n$ colored image can be represented by a $m \times n \times 3$ tensor. The last dimension represents the red, green and blue components of the image respectively. For an example, please see Figure 3.

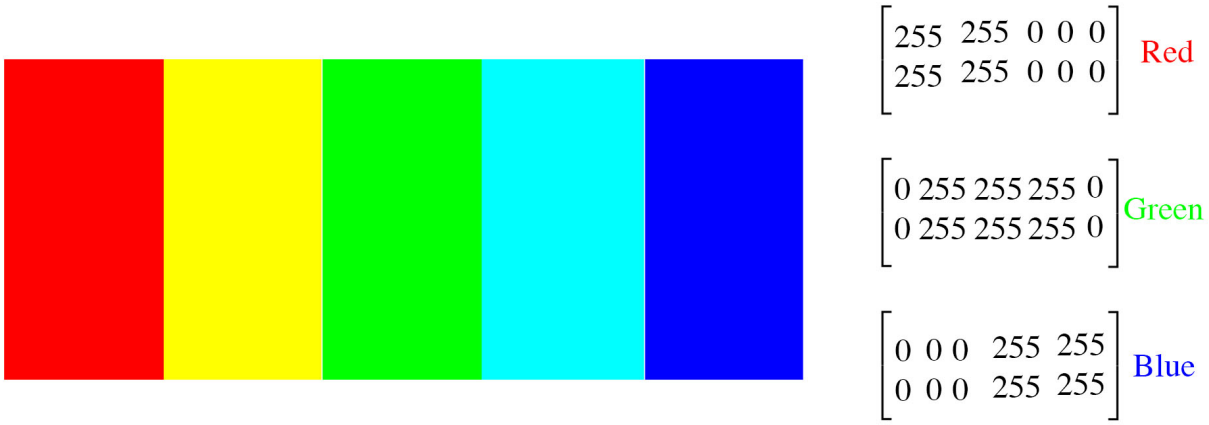

Figure 3: A 2×5 Colored Image Representation

When dealing with CNNs, the meaning of convolution is slightly different. Let X and K be an m-dimensional tensors where X is the input and K is the kernel. Then, we define the *convolution* (denoted by $\star$) of X by K by:

$$(X \star K)(i_1, \ldots, i_m) = \sum_{k_1} \cdots \sum_{k_m} X(i_1 + k_1, \ldots, i_m + k_m) K(k_1, \ldots, k_m) \tag{25}$$

Which corresponds to "sliding" the Kernel K across X, multiplying pairwise elements and then summing them up. An example is shown in Figure 4

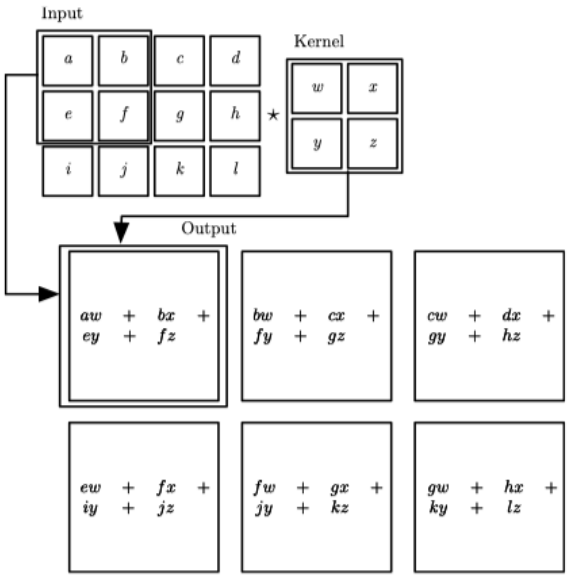

Figure 4: Convolution Example on $2D$ Tensors

We also define *flipped convolution* (denoted by $\circledast$) of X with K by

$$(X \circledast K)(i_1, \ldots, i_m) = \sum_{k_1} \cdots \sum_{k_m} X(i_1 - k_1, \ldots, i_m - k_m) K(k_1, \ldots, k_m) \tag{26}$$

Note that the flipped convolution is commutative, whereas the convolution that we defined is not. In practice, flipping the kernel does not affect the output of the network as the network will learn the best kernel; if the best one is the flipped kernel, it will flip it.

We can also represent the convolution operation using matrix multiplication. The operation shown in Figure 4 can be represented by

$$\begin{bmatrix} w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{bmatrix}$$

For more information on how to represent convolution using matrix multiplication, please see [10]. Hence, we can easily add a convolutional layer to a NN. These (small) kernels can be used to extract *feature maps* from the inputs. Please see Figures 5, 6 and 7

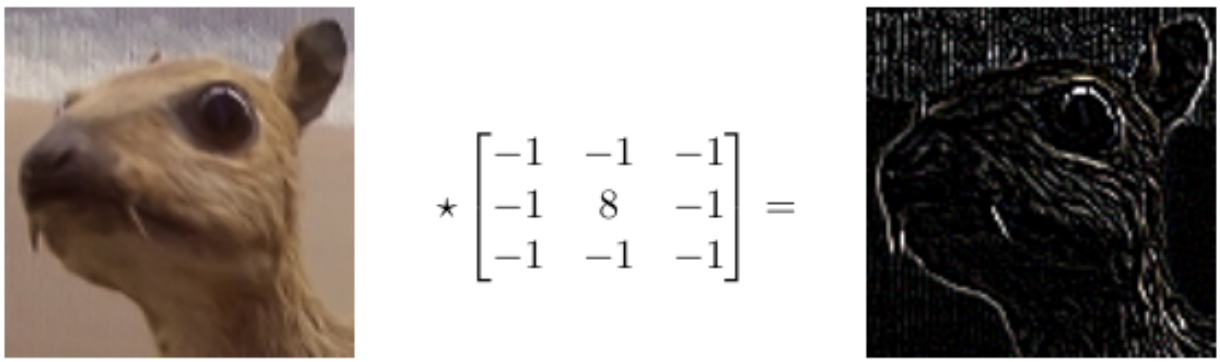

Figure 5: Edge Kernel [11]

$$\star \begin{bmatrix} 0 & -1 & 5 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$

Figure 6: Sharpening Kernel [11]

$$\star \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} =$$

Figure 7: Blurring Kernel [11]

CNNs have two main features that allow them to be extremely useful:

1. Sparse Connectivity
2. Parameter Sharing

Sparse Connectivity means that each neuron is connected to a few neurons from the previous layers. Of course, this depends on the size of the kernel. These creates a sense of locality at the neurons in the layers. Moreover, neurons in deeper layers will be deeply connected to many other neurons in the previous layers.

Parameter sharing is extremely important for optimization. Basically, the CNN only needs to learn the parameters of the kernels that they use. These parameters are shared across all neurons of that layer; hence, the algorithm needs to find a relatively small number of parameters.

Example 3.1. Consider a 180×320 input image with a kernel that subtracts the left value from the right value ($K = \begin{bmatrix} -1 & 1 \end{bmatrix}$). Then the convolutional layer has only 2 weights -

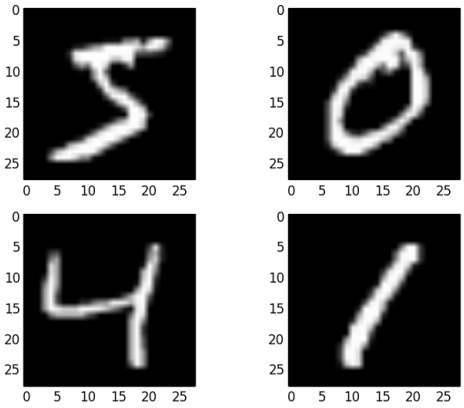

Figure 8: Example Digit Images

specifically, the -1 and 1 entries of the kernel.
Now consider describing the same transformation with a fully connected layer. then, we need $\underbrace{180 \times 320}_{\#input} \times \underbrace{180 \times 319}_{\#output}$ weights
If we represent convolution by matrix multiplication, then we need $2 \times 180 \times 319$ weights, which is $28,800\times$ less weights.

This decreases the number of parameters of the cost function, which makes training much faster!

After the convolutional layer, we can use *pooling* as a dimensionality reduction technique. Some examples of pooling include:

1. Max pooling
2. Mean pooling

and they mean to represent the data and its nearby statistics. We can also increase the stride (how much a kernel slides) to reduce the dimension as well.

We will consider the problem of handwritten digit detection. Assume that we are dealing with the MNIST dataset [12] which contains a total of $70,000$ images. Each image (please see Figure 8) contains a single handwritten digit and its label (the label tells what digit is written inside it). We can build a CNN that can learn from the dataset to predict the digit written inside an image. The input size is a 28×28 BW image, and the output is $x \in \mathbb{R}^{10}$ that represents the probability that the image contains the corresponding image (for example, x_0 tells the probability of the image containing 0). One way to build the model is to have a convolutional layer that uses 32 Kernels of size 3×3, and then follow it by a maxpooling and a dense (fully connected) layer. After being trained, this model can predict the digits with more than 97% accuracy. For the complete model with the code, please see [13]. The architecture of this network is shown in Figure 9. This network has around 2 million parameters that need to be optimized.

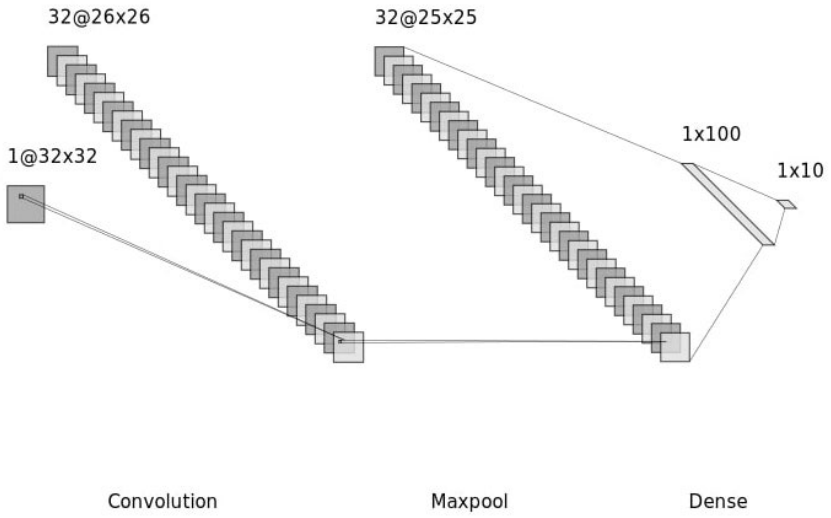

Figure 9: CNN Network Architecture

4 Graph Neural Networks

In this section, we aim to study a method that allows us to generalize some of the previous concepts to graphs. Graphs can be used to represent many real life situations such as social networks, citation networks and more. In graphs, the situation is different as the shape of the input data is not fixed unlike the tensor type inputs defined previously. Here, we follow the introduction given in [14].

Assume that we have an undirected graph $\mathbb{G} = (V, E)$, where V is a set of vertices (their indices), and E is a set of edges. For example, if $V = 1, 2, \ldots, 10$, then we have 10 vertices. If $(1, 4) \in E$, then the 1^{st} vertex is connected to 4^{th} vertex. We wish to find a function that takes

1. $X \in \mathbb{R}^{N \times D}$ where N is the number of vertices and D is the number of features per vertex

2. $A \in \mathbb{R}^{N \times N}$, the adjacency matrix of the corresponding graph

and produces an output $Z \in \mathbb{R}^{N \times F}$, where F is the number of output features per vertex. In order to get graph-level outputs, we can use pooling techniques.

Every layer rule can be written as

$$H^{[l+1]} = g(H^{[l]}, A) \tag{27}$$

where g is the update function that is used in every layer, $H^{[i]}$ represents the vertex features at the i^{th} layer, A is the corresponding matrix and σ is the function defined in 5. Note that A does not change unless we use some pooling techniques that reduce the number

of vertices of the graph. Suppose that the weights of the features of the vertices in layer l are $W^{[l]}$. We will consider

$$g(H^{[l]}, A) = \sigma(AH^{[l]}W^{[l]})$$

Then, $\text{row}_i(A)\text{col}_j \cdot (H^{[l]})$ means that we add up the values of the j^{th} feature of vertices connected to the i^{th} vertices. If we want to account for the value existing at the i^{th} vertex, we would add the identity matrix to A. Multiplying by $W^{[l]}$ transforms the inputs to the outputs ready to be passed to the next layer.

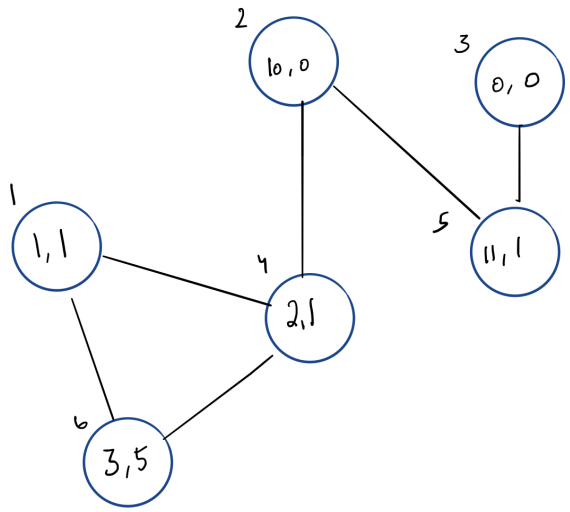

Figure 10: Input Graph

Example 4.1. Consider the graph shown in Figure 10. Then, we have

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \text{ and } X = \begin{bmatrix} 1 & 1 \\ 10 & 0 \\ 0 & 0 \\ 2 & 1 \\ 11 & 1 \\ 3 & 5 \end{bmatrix}$$

Hence, we have $AX = \begin{bmatrix} 5 & 6 \\ 13 & 2 \\ 11 & 1 \\ 14 & 6 \\ 10 & 0 \\ 3 & 2 \end{bmatrix}$ which shows added features of the neighbors of the vertices.

Assume that we have $W = \begin{bmatrix} 1 & 0 & -1 & 5 \\ 0 & 1 & 10 & -1 \end{bmatrix} \in \mathbb{R}^{2\times 4}$ which means that in the next layer, the

vertices will have 4 features instead of 2. Also, we have

$$AXW = \begin{bmatrix} 5 & 6 & 55 & 19 \\ 13 & 2 & 7 & 63 \\ 11 & 1 & -1 & 54 \\ 14 & 6 & 46 & 64 \\ 10 & 0 & -10 & 50 \\ 3 & 2 & 17 & 13 \end{bmatrix} \text{ and } \sigma(AXW) = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 0.9 & 1.0 & 1.0 \\ 1.0 & 0.7 & 0.3 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 0.5 & 0.0 & 1.0 \\ 1.0 & 0.9 & 1.0 & 1.0 \end{bmatrix}$$

Notice that the values grow out of proportion, which is an issue for feature scaling. This simple approach, while powerful, has two main issues.

The first issue is that the value at the center vertex is not taken into account. A solution is to create a new adjacency matrix $\tilde{A} = A + \mathbb{I}_n$ and then use it instead of A in 27. The second issue is that the values of the features grow outside of the feature scale. This is an issue because if some features are out of scale, it can slow the rate of which GD or SGD converge. This does not have a straight forward solution but we will explore two of them. The straight forward solution is to simply average out the values while adding up the values of the connected vertices. This can be done by replacing A by $D^{-1}A$ where D is the degree matrix.

Example 4.2. We continue Example 4.1. Here, we have $D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$

and we have

$$D^{-1}AXW = \begin{bmatrix} 2.5 & 3.0 & 27.5 & 9.5 \\ 6.5 & 1.0 & 3.5 & 31.5 \\ 11.0 & 1.0 & -1.0 & 54.0 \\ 4.7 & 2.0 & 15.3 & 21.3 \\ 5.0 & 0.0 & -5.0 & 25.0 \\ 1.5 & 1.0 & 8.5 & 6.5 \end{bmatrix} \text{ and } \sigma(D^{-1}AXW) = \begin{bmatrix} 0.9 & 1.0 & 1.0 & 1.0 \\ 1.0 & 0.7 & 1.0 & 1.0 \\ 1.0 & 0.7 & 0.3 & 1.0 \\ 1.0 & 0.9 & 1.0 & 1.0 \\ 1.0 & 0.5 & 0.0 & 1.0 \\ 0.8 & 0.7 & 1.0 & 1.0 \end{bmatrix}$$

Notice how the output of $\sigma(\cdot)$ here has more variance than in Example 4.1.

The drawback of this method is that only the connectivity of the neighbors is considered. If the neighbor is connected only to one vertex, then this connection is stronger than a neighbor that is connected to every vertex. To counter this, Kipf et al. [14] recommended a method similar to the laplacian's symmetric normalization. We normalize A by $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.

Example 4.3. We use the same setting in Example 4.1. Here, we have

$$D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW = \begin{bmatrix} 2.3 & 2.9 & 26.8 & 8.7 \\ 6.3 & 0.9 & 2.8 & 30.7 \\ 7.8 & 0.7 & -0.7 & 38.2 \\ 5.7 & 2.4 & 18.8 & 26.1 \\ 5.0 & 0.0 & -5.0 & 25.0 \\ 1.3 & 0.9 & 7.8 & 5.7 \end{bmatrix} \text{ and } \sigma(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW) = \begin{bmatrix} 0.9 & 0.9 & 1.0 & 1.0 \\ 1.0 & 0.7 & 0.9 & 1.0 \\ 1.0 & 0.7 & 0.3 & 1.0 \\ 1.0 & 0.9 & 1.0 & 1.0 \\ 1.0 & 0.5 & 0.0 & 1.0 \\ 0.8 & 0.7 & 1.0 & 1.0 \end{bmatrix}$$

For more indepth discussion of the differences and other methods, please see [15]. In practice, each normalization technique has its weaknesses and strength; hence, it is recommended to try different methods to see how they affect the results. Using symmetric normalization and self-loops, the update rule then becomes

$$g(H^{[l]}, \tilde{A}) = \sigma(D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}H^{[l]}W^{[l]}) \tag{28}$$

where $\tilde{A} = A + \mathbb{I}_n$ and D is the diagonal matrix of $\tilde{A}$. For more in-depth discussion of GNNs and other methods, please see [16].

We will consider the problem of classifying articles from the Cora dataset [17]. Cora is a dataset that contains 2708 vertices and 5429. Each vertex represents a publication, and edges represent a citation between two publication. Moreover, each vertex has a set of words that are mentioned in it (described by a vector). Each vertex is classified into one of 7 classes ("Genetic_Algorithms", "Case_Based" and such). The goal is to create a GNN model that is able to learn on and classify the articles correctly. We can use the method that was discussed earlier for that purpose. By building a GNN network of 3 layers that uses 28, Kipf et al. were able to build a model that predicts the labels of the publications with 81.5% accuracy [14] [16].

5 Conclusion

In this report, we gave a mathematical foundation for the theory of Neural Networks. We focused on Convolutional Neural Networks, gave examples of the Kernels, and discussed an application of them. We also discussed a method that allows us to generalize some of the methods used in CNNs to data that is available as a graph, and discussed one application that uses this method.

A Mathematical Background

In this section, we will go over the some mathematical concepts that are needed in our previous discussions. We assume the reader knows basic Linear Algebra (even though we will revise the Singular Value Decomposition), multivariable calculus, and basic probability.

A.1 Singular Value Decomposition

We know that if we are given a matrix $B \in \mathbb{R}^{n\times n}$, we can decompose it using its eigenvectors. More specifically, we can write $B = QDQ^{-1}$, where Q is a square $n \times n$ matrix whose ith column is the ith eigenvector of B, and D is a diagonal $n \times n$ matrix whose diagonal elements are the corresponding eigenvalues, ie, $D_{ii} = \lambda_i$. This decomposition has two main drawbacks:

- The eigenvectors of B are not necessarily orthogonal
- The matrix B has to be square.

We wish to find another decomposition that does not have these drawbacks. Assume that we are given a matrix $A \in \mathbb{R}^{m \times n}$ of rank r. We wish to decompose A as such $A = U\Sigma V^T$ where U is a $m \times m$ orthonormal matrix, V is a $n \times n$ orthonormal matrix, and Σ is a $m \times n$ rectangular diagonal matrix with non-negative reals on the main diagonal. Let the ith column of U be denoted by u_i, the ith column of V by v_i, and the ith diagonal entry of Σ be denoted by σ_i. The σ's are called the singular values of A and are normally ordered in decreasing order, so $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_r \geq 0$. This decomposition gives basis for the four fundamental spaces as follows:

- $u_1, \ldots, u_r$ is an orthonormal basis for the column space
- $u_{r+1}, \ldots, u_m$ is an orthonormal basis for the nullspace of A^T
- $v_1, \ldots, v_r$ is an orthonormal basis for the row space
- $v_{r+1}, \ldots, v_n$ is an orthonormal basis for the nullspace of A

and then the vectors v_i and u_i diagonalize A as

$$Av_i = \sigma_i u_i \text{ for } i = 1, \ldots r$$

This formulation also allows us to describe A as a sum of r-many rank 1 matrices, as follows:

$$A = \sum_{i=1}^{r} u_i \sigma_i v_i^T$$

We now will show that for any matrix A, we can find u_i's, σ_i's and v_i's that give us the Singular Value Decomposition (SVD). We will base the derivation on the eigendecomposition.

Assume that A is a $m \times n$ real matrix. Then $A^T A$ is a symmetric and positive semi-definite matrix $n \times n$ matrix. Hence, $A^T A$ has an eigenvalue decomposition $A^T A = QDQ^{-1}$ where Q is a $n \times n$ matrix eigenvectors as its columns and D is a diagonal $n \times n$ matrix with the eigenvalues as its diagonal entries. Denote the eigenvalues by $\lambda_1, \ldots, \lambda_n$ and the corresponding eigenvectors by $q_1, \ldots, q_n$. Since $A^T A$ is symmetric, q_i's are orthogonal. Hence $A^T A = QDQ^{-1} = QDQ^T$. Since $A^T A$ is positive semi-definite, then we have $\lambda_i \geq 0$ for $i = 1, \ldots, n$. We want to find U, V, and Σ such that $A = U\Sigma V^T$. We calculate $A^T A$:

$$A^T A = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma^T U^T U \Sigma V^T = V(\Sigma^T \Sigma)V^T$$

Since U is orthogonal and Σ is diagonal. Since we know the eigendecomposition of $A^T A$, then we have $V(\Sigma^T \Sigma)V^T = QDQ^T$, so this means that $V = Q$ and $\sigma_i^2 = \lambda_i$ for $i = 1, \ldots, n$. Now we need to find the u_i's. Remember that the SVD diagonalizes A by $Av_i = \sigma_i u_i$, then $u_i = \frac{Av_i}{\sigma_i}$. What remains is that we need to show that these u_i's are orthogonal.

Lemma A.1. *The vectors u_i (as constructed above) are pair-wise orthogonal*

Proof. Assume that $i \neq j$. Then we show that their dot product is zero:

$$\begin{aligned} u_j^T u_i &= \left(\frac{Av_j}{\sigma_j}\right)^T \left(\frac{Av_i}{\sigma_i}\right) = \frac{1}{\sigma_j \sigma_i} v_j^T A^T A v_i \\ &= \frac{1}{\sigma_j \sigma_i} v_j^T (\sigma_i^2 v_i) && \text{(since } v_i \text{ is an eigenvector of } A^T A\text{)} \\ &= \frac{\sigma_i}{\sigma_j} v_j^T v_i \\ &= 0 && \text{(since eigenvectors of } A^T A \text{ are orthogonal)} \end{aligned}$$

□

We now show prove a lemma that we will use in the proof of the next theorem.

Lemma A.2. *Let Q be an orthonormal matrix $n \times n$ and $x \in \mathbb{R}^n$. Then $||Qx|| = ||x||$*

Proof. We focus on the square norm. We have:

$$||Qx||^2 = (Qx)^T (Qx) = x^T (Q^T Q) x = x^T x = ||x||^2$$

since Q is an orthonormal matrix. Hence, $||Qx|| = ||x||$. With a similar argument, we can show that $||Q^T x|| = ||x||$ as well. □

Now, we show that we can bound $||Ax||$ by the use of its singular values in the following theorem.

Theorem A.3. *Assume that A is a $m \times n$ real matrix and that σ_1 is its largest singular value. Then, for any $x \in \mathbb{R}^n$, we have $||Ax|| \leq \sigma_1 ||x||$*

Proof. Assume that A has the SVD $A = U\Sigma V^T$. Then

$$\begin{aligned} ||Ax|| = ||U\Sigma V^T x|| &= ||\Sigma V^T x|| && \text{(since } U \text{ is an orthonormal matrix)} \\ &\leq ||\sigma_1 I_n V^T x|| && \text{(since } \Sigma \text{ is a diagonal matrix)} \\ &= \sigma_1 ||V^T x|| && \text{(since } \sigma_1 \geq 0 \text{ and } I_n \text{ is identity)} \\ &= \sigma_1 ||x|| && \text{(since } V \text{ is an orthonormal matrix)} \end{aligned}$$

and we are done. □

A.2 Convexity and Optimization

We begin by defining what a convex set is. Intuitively, a set is convex if the line segment that connects any two points from the set is in the set. Formally, we define it as follows:

Definition A.1. Let S be a vector space over $\mathbb{R}$. A subset $C \subset S$ is called *convex* if

$$\forall x, y \in C \ \forall t \in [0,1] \ \ (1-t)x + ty \in C$$

Examples of the convex subsets of $\mathbb{R}$ are intervals or points. A disk (not a circle) is a convex subset of $\mathbb{R}^2$.

We now shift our focus to functions.

Definition A.2. Let X be a convex set. Suppose that $f\colon X \mapsto \mathbb{R}$. Then f is called a *convex function* if

$$\forall x, y \in X \ \forall t \in [0,1] \quad f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$$

For a picture of this definition, please see Figure 11. Here, in the language of the definition, we fix $x = 2, y = 5$ and the purple line connecting them. The green line is the line segment representing $\forall t \in [0,1]\ tf(x) + (1-t)f(y)$ and the blue segment is the segment representing $\forall t \in [0,1]\ f(tx + (1-t)y)$. The blue line is always under (less than or equal to) the green line. For an example of a function that is not convex, please see Figure 12. The segment definitions are the same as in the previous paragraph. However, notice that the blue segment is not always under the green line. Hence, $f(x) = \sin(2x) + 2$ is not a convex function.

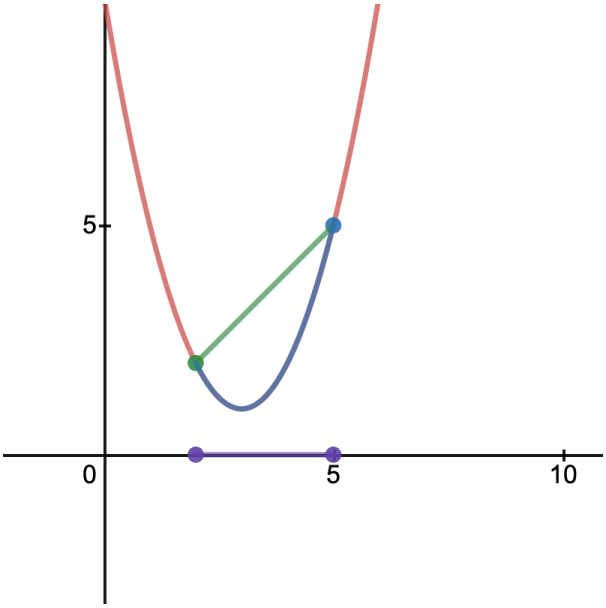

Figure 11: $f(x) = (x - 3)^2 + 1$ is a convex function

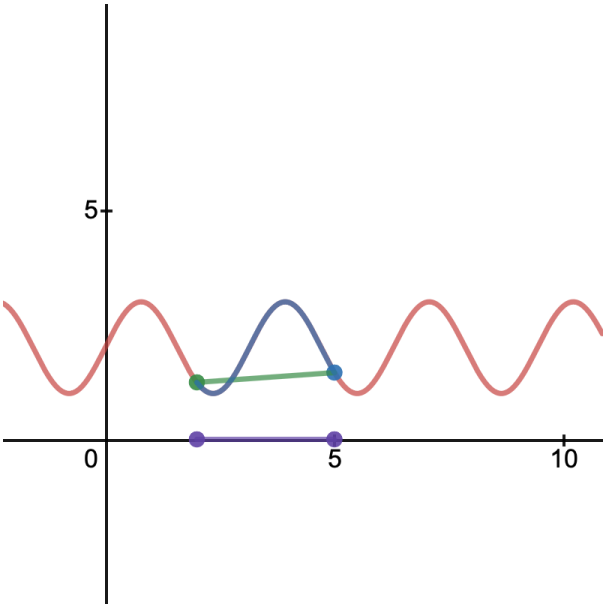

Figure 12: $f(x) = \sin(2x) + 2$ is not a convex function

If f is differentiable, then we can also define its convexity using the gradient.

Definition A.3. Let X be a convex set. Suppose that $f\colon X \mapsto \mathbb{R}$ is differentiable. Then f is called a *convex function* if

$$\forall x, y \in X \ f(x) \geq f(y) + \nabla f(y)^T (x - y)$$

Essentially, this definition is saying that the function is above all of its tangents. For a pictorial example (again, on $\mathbb{R}$), please see Figure 13. In this figure, we let x vary and fixed two examples for y, specifically $y = 2$ and $y = 5$. Notice that the tangent lines at these values are always under (or on) f.

For functions defined on $\mathbb{R}$ that are twice differentiable, we have the following equivalent definition:

Definition A.4. Let $f\colon \mathbb{R} \mapsto \mathbb{R}$ be a twice differentiable function on A. If $f''(x) \geq 0$, then f is convex.

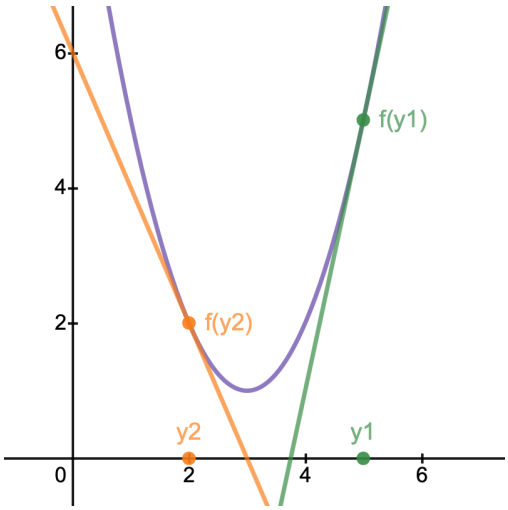

Figure 13: $f(x) = (x - 3)^2 + 1$ is a convex function (using the second definition)

Why are convex functions so important? Turns out, many optimization problems can be described with convex functions. We will explore some examples now. For the setting, assume that $y \in \mathbb{R}^n$, $X \in \mathbb{R}^{n \times p}$ and $w \in \mathbb{R}^p$ where $n, p \in \mathbb{N}$. In the machine learning context, n is the number of samples, p is the number of features, y are the observed outputs, the ith row, $\text{row}_i(X)$, contains the observed features for the i^{th} sample, w contains the weights for the features, and Xw is the current prediction. Essentially, we wish to find the best weights that minimizes the function. Assume that $f\colon \mathbb{R}^n \mapsto \mathbb{R}$ (called the cost function). Then, we wish to find w^* that minimizes f, or

$$w^* = \arg\min_w f(w)$$

The function f can be many different things. We will give two examples here before proceeding on.

Example A.1 (Least Squares)**.** With the above formulation, consider $f(w) = \|y - Xw\|_2^2$. This equation is used as an objective function to find a linear equation that best fits a data set.

Example A.2 (Ridge Regression)**.** With the above formulation, consider $f(w) = \|y - Xw\|_2^2 + \lambda\|w\|_2^2$ where $\lambda \geq 0$.. This function is used as an objective function, similar to least squares, to find a linear equation that fits the data. The use of the parameter λ is helpful to not overfit the data.

Both of these examples are examples of convex functions. We will prove that they are convex functions in the following series of lemmas.

Lemma A.4. *Let $x \in \mathbb{R}$. Then $f(x) = x^2$ is a convex function.*

Proof. We can show this using the third definition of convexity for single variable functions. However, we will show that the first definition holds for completion. Assume that $t \in [0, 1]$.

Let $x, y \in \mathbb{R}$. Then, we wish to show that$(tx + (1-t)y)^2 \leq t(x)^2 + (1-t)(y)^2$ is true. Expanding, we get:

$$\begin{aligned} t^2x^2 + 2t(1-t)xy + (1-t)^2y^2 &\leq tx^2 + (1-t)y^2 \\ t^2x^2 + 2t(1-t)xy + y^2 - 2ty^2 + t^2y^2 &\leq tx^2 + y^2 - ty^2 \\ (t^2-t)x^2 + 2(t-t^2)xy + (t^2-t)y^2 &\leq 0 \\ (t^2-t)x^2 - 2(t^2-t)xy + (t^2-t)y^2 &\leq 0 \\ (t^2-t)(x^2 - 2xy + y^2) &\leq 0 \\ (t^2-t)(x-y)^2 &\leq 0 \end{aligned}$$

If $x = y$, then the last inequality is true since $x - y = 0$. If $x \neq y$, then, since $t \in [0,1]$, then $t^2 - t = t(t-1) \leq 0$ and $(x-y)^2 \geq 0$, the last inequality is true. Hence f is a convex function □

Lemma A.5. *Let $f, g \colon \mathbb{R}^n \mapsto \mathbb{R}$ be convex functions. Then $f + g$ is a convex function.*

Proof. Assume that f and g are convex functions as in the first definition. Let $t \in [0,1]$ and $x, y \in \mathbb{R}^n$. Then, we have

$$\begin{aligned} f(tx + (1-t)y) + g(tx + (1-t)y) & \\ &\leq tf(x) + (1-t)f(y) + tg(x) + (1-t)g(y) \\ &\leq t(f(x) + g(x)) + (1-t)(f(x) + g(x)) \end{aligned}$$

Hence, $f + g$ is also a convex function. □

Using induction, it is easy to prove that a finite sum of convex function is also a convex function. We omit the proof.

Lemma A.6. *If $f \colon \mathbb{R}^n \mapsto \mathbb{R}$ be a convex function, then for any $A \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^n$, the function*

$$h(x) := f(Ax + b)$$

is convex.

Proof. Assume the hypothesis, $t \in [0,1]$ and $x, y \in \mathbb{R}^m$. Then, we have

$$\begin{aligned} h(tx + (1-t)y) &= f(A(tx + (1-t)y) + b) \\ &= f(Atx + (1-t)Ay + tb + (1-t)b) \\ &= f(t(Ax + b) + (1-t)(Ay + b)) \\ &\leq tf(Ax + b) + (1-t)f(Ay + b) \qquad \text{(since } f \text{ is a convex function)} \\ &= th(x) + (1-t)h(y) \end{aligned}$$

Hence h is also a convex function. □

Now, we are ready to prove that the square loss objective function is a convex function.

Theorem A.7. *Assume $y \in \mathbb{R}^n$, $X \in \mathbb{R}^{n \times p}$ and $w \in \mathbb{R}^p$ where $n, p \in \mathbb{N}$. Let f be the square loss objective, ie, $f(w) = \|y - Xw\|_2^2$. Then f is a convex function.*

Proof. Let X_{i*} be the ith row of X, and y_i be the ith component of y. Then, we can write f as

$$f(w) = \|y - Xw\|_2^2 = \sum_{i=1}^{n} (y_i - X_{i*}w)^2$$

We know that $(\cdot)^2$ is a convex function by Lemma A.4. Then, since $(y_i - X_{i*}w)^2 = (X_{i*}w - y_i)^2$, by Lemma A.6, $(y_i - X_{i*}w)^2$ is also convex. Then, f is a finite sum of convex functions, hence by Lemma A.5, f is a convex function. □

The appeal to convex functions is that they have a *single* minima.

Theorem A.8. *Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a convex function with local minimum at $x \in \mathbb{R}^n$. Then x is also a global minimizer.*

Proof. We prove the theorem by contradiction. Assume that f is a convex function. Assume that x_l is a local minimizer and x_g is a global minimizer such that $f(x_g) < f(x_l)$ and $x_l \neq x_g$. Since x_l is a local minimizer, we can find δ such that

$$\forall x \in \mathbb{R}^n \ \|x - x_l\|_2 \leq \delta \implies f(x_l) \leq f(x)$$

Now, we form a line segment between x_l and x_g and choose $t \in [0, 1]$ to be small enough such that $x_t := tx_l + (1 - t)x_g$ satisfies $\|x_t - x_l\|_2 \leq \delta$. Then, $f(x_l) \leq f(x_t)$. Then, we have

$$\begin{aligned}
f(x_l) &\leq f(x_t) \\
&= f(tx_l + (1-t)x_g) && \text{(definition of } x_t\text{)} \\
&\leq tf(x_l) + (1-t)f(x_g) && \text{(since } f \text{ is a convex function)} \\
&= tf(x_l) + f(x_g) - tf(x_l) \\
&= f(x_g) \\
&< f(x_l) && \text{(since } f(x_g) < f(x_l)\text{)}
\end{aligned}$$

which is a contradiction. Hence, $x_l = x_g$, and every local minimizer is a global minimizer □

Hence, once a local minimum is found, we are done (since this theorem guarantees that it is a global minimum).

References

[1] DeepAI, "Neural network," May 2019.

[2] C. F. Higham and D. J. Higham, "Deep learning: An introduction for applied mathematicians," 2018.

[3] R. Tibshirani, "10-725: Optimization lecture notes," 2013.

[4] J. D. Lee, M. Simchowitz, M. I. Jordan, and B. Recht, "Gradient descent converges to minimizers," 2016.

[5] R. Willet, "Mathematical foundations of machine learning," 2019.

[6] R. Ge, F. Huang, C. Jin, and Y. Yuan, "Escaping from saddle points — online stochastic gradient for tensor decomposition," 2015.

[7] D. P. Bertsekas, "Incremental gradient, subgradient, and proximal methods for convex optimization: A survey," 2015.

[8] N. Z. Shor, K. C. Kiwiel, and A. Ruszcayundefinedski, *Minimization Methods for Non-Differentiable Functions*. Berlin, Heidelberg: Springer-Verlag, 1985.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[10] M. Dukhan, "The indirect convolution algorithm," 2019.

[11] Wikipedia contributors, "Kernel (image processing) — Wikipedia, the free encyclopedia," 2020. [Online; accessed 23-August-2020].

[12] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[13] J. Brownlee, "How to develop a cnn for mnist handwritten digit classification," Jan 2020.

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016.

[15] Y. Dubois, "Interpretation of symmetric normalised graph adjacency matrix?." Mathematics Stack Exchange.

[16] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, p. 18–42, Jul 2017.

[17] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," *Inf. Retr.*, vol. 3, p. 127–163, July 2000.