An Introduction to Neural Networks

Austin Barton[1]

[1]MATH 4210 - Mathematical Foundations of Data Science - Professor Short
[1]Georgia Institute of Technology

April 24, 2023

Contents

Introduction

Neural Networks are engineered systems used in machine learning that are inspired by the human brain. They are the heart of deep learning, a subset of machine learning.

This is a vague definition, but neural networks are a very broad field of learning models. They are comprised of node layers, containing an input layer, one or more hidden layers, and an output layer. The structure and properties of each of these change significantly between different models of neural networks.

Some applications of neural networks are:

- Image classification and recognition
- Computer vision
- Signature Verification
- Quality Analysis
- Sentiment Analysis
- Finance and trading (still just research heavy and not so common in actual firms yet, or so it seems)

Some well-known implementations are...

- ChatGPT
- GitHub Co-Pilot
- TAPAS (Google NLP Model)

To break down some of the underlying principles, in general, neural networks will take in p variables $X = (X_1, \ldots, X_p)$, build some non-linear function $f(X)$, and use that to predict the correct output variable Y.

The three models we will be studying and implementing are...

- Feedforward Neural Networks: These are the most fundamental model of neural networks. The goal is to approximate some true function f^*. It does this by constructing a directed acyclic graph comprised of layers of nodes feeding in one direction into the output, the sink of the network. The layers consists of an input layer, the hidden layers, and the output layer. Each i^{th} layer represents the mapping of some function $f^{(i)}$.

  A feedforward neural network consists of K hidden units, each of which consist of a function

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

  Each of these are *activation functions* and are typically non-linear.

1 Neural Networks

1.1 The Basic Idea/Paradigm Overview

1.2 The General Model for Neural Networks

What follows is a series of some basic groundwork necessary to begin to study neural networks. These ideas form distinct attributes that these methods have over other statistical learning methods and each attribute is common to other specific models and not necessarily (or usually) exclusive to one particular model of a neural network.

Definition 1.1. Input Layer

Definition 1.2. Hidden Layer

Definition 1.3. Output Layer

Definition 1.4. Node

Definition 1.5. Activation Function

Theorem 1.6.

Definition 1.7. A definition

2 Three Different Models

Here we introduce the overarching model of the FNN with a focus on the general reasoning behind this model of a neural network, the data it is designed best for, and the structure of the layers and neurons.

2.1 Feedforward Neural Network (FNN)

Overview

2.2 Convolutional Neural Network (CNN)

Overview

2.3 Recurrent Neural Network (RNN)

Overview

3 Feedforward Neural Networks

Here is a brief example of a basic feedforward nerual network learning the XOR function from "Deep Learning" by Goodfellow et. al. Their description of the example is slightly more thorough, however, we highlight some pieces of it for added clarity for students and readers of our level.

- The XOR function inputs two boolean variables typically encoded by 1 for true and 0 for false. Given an input vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ it outputs either 0 for false or 1 for true.
- The XOR function here is here target function $y = f^*(x)$ that we would like to approximate.
- Our model provides a function $y = f(x;\theta)$, and our learning algorithm will output θ to make f as similar to $f*$ as possible.
- We want our network to perform correctly on the set $\chi = \{\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}\}$
- We can treat this as a regression problem and use the MSE loss function $L(\theta) = (\theta - a)^2$ and our cost function will be,

$$J(\theta) = \frac{1}{4}\sum_{x\in\chi}(f^*(x) - f(x;\theta))^2$$

- Now, we must choose the form of our model, $f(x;\theta)$.
- Suppose we choose a linear model with θ consisting of w and b. That is,

$$f(x;w,b) = x^T w + b$$

- We can minimize $J(\theta)$ in closed form, obtaining,

$$w = (x^T x)^{-1} x^T y = 0 \; b = \frac{1}{2}$$

  (From the normal equations)
- This outputs 0.5 everywhere.
- One way to solve this is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- *FFN has entered the chatroom* Introduce a feedforward neural networks containing two hidden layers.
- This FFN has a vector of hidden units h that are computed by a function $f^{(1)}(x;W,c)$. Note that W is a matrix of weights. The values of these hidden units are then used as the input for the hidden layer/2nd layer.
- The 2nd layer is also the output layer on top of being a hidden layer. Because it is an output layer, this layer is a linear regression model, but not applied to h rather than to x.
- $h = f^{(1)}(x;W,c)$ and $y = f^{(2)}(h;w,b)$ with the complete model being $f(x;W,c,w,b) = f^{(2)}(f^{(1)}(x))$

- Most neural networks have an affine transformation controlled by learned parameters, followed by a fixed nonlinear function i.e. the activation function.
- Define $h = g(W^T x + c)$ where W provides the weights of a linear transformation and c is the bias of that linear transformation.
- We describe an affine transformation from a vector x to a vector h.
- Activation function g is typically chosen to be a function that is applied elementwise with $h_i = g(x^T W_{:,i} + c)$.
- THe default activation function is the rectified linear unit or ReLU defined as $g(z) = \max\{0, z\}$ for $z \in \mathbb{R}$.
- Our complete network is:

$$f(x; W, c, w, b) = w^T \max\{0, W^T x + c\} + b$$

4 Convolutional Neural Networks

4.1

4.2 The Convolution Layer

4.2.1 What is Convolution?

Definition 4.1. The Convolution Operator Consider two functions x and w where $x : T \to T'$ where $T, T' \subseteq \mathbb{R}$ and $w : S \to S'$ where $S, S' \subseteq \mathbb{R}$. Let $t \in T$ and $a \in S$. *Convolution* is then a linear operator denoted as $*$ that is defined as

$$x * w = s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

where $x(t)$ and $w(a)$ take on values of 0 for inputs outside of their domain.

It is commutative, thus,

$$x * w = w * x = \int x(t-a)w(a)da$$

The convolution operation $x * w$ essentially flips w around the "y" axis at point a and sums the products of x and w evaluated at each point from left to right. Of course, this is just a layman's terms explanation and shouldn't substitute a rigorous understanding of its actual definition.

We refer to the function x as the input and the function w as the kernel. x is the function that we are operating on using w as a sort of weighting function. The output is sometimes referred to as the feature map.

From L3Harris: "Convolution filters produce output images in which the brightness value at a given pixel is a function of some weighted average of the brightness of the surrounding pixels. Convolution of a user-selected kernel with the image array returns a new, spatially filtered image. You can select the kernel size and values, producing different types of filters."

An important note is that even though people often use kernel and filter interchangeably, the kernel is the actual mathematical object you could say and the filter is the actual implementation of a kernel typically in the form of a matrix of weights which slides/convolves over the pixels in an image.

4.3 Implementing Convolution

4.3.1 Sliding Method

This method involved creating a matrix of weights which act as linear transformations of values of our input function. The center square of this matrix/kernel "slides" over each pixel values, performs the transformation, and replaces the pixel at the center with that output. It must do this with every single pixel. Because pixels on the edge may not have the necessary adjacent data to do this transformation, they are either not included at all or some technique such as padding is applied.

4.3.2 FFT

The Fast Fourier Transform is an algorithm that determines the discrete Fourier transform of a sequence in a fast and efficient manner. What does this have to do with convolution? Well, without going too far outside the scope of this paper, Fourier Tranforms are transforms that convert functions into a form that describes the frequencies present in the original domain. And the convolution of two signals is equivalent to the multiplication of their Fourier transforms. Therefore, by transforming the input space into frequency space, a convolution becomes a single element-wise multiplication. What this means for us is that the input to a convolutional layer and kernel can be converted into frequencies using the Fourier Transform, multiply once, and then convert back using the inverse Fourier Transform. So, maybe it is possible that with a good implementations of this mapping and multiplication, we can make the computationally expensive convolution operation more efficient.

Good news, it is! In fact, it is MUCH more efficient.

4.3.3 Parameter Enumeration in CNNs

In order to calculate the number of parameters of a convolution layer we apply the equation

$$((h \times w \times p) + 1) \times d$$

where h - height of filters in current layer, w - width of filters in current layer, p - number of filters in the previous layer, d - number of filters in the current layer.

The intuition is for this that for each filter in a layer, we have $h \times w$ parameters to learn. But for each filter of shape (w,h) corresponds to a filter in the immediate previous layer. So we have $h \times w \times p$ parameters corresponding to each filter plus 1 for the bias. Since we have d filters, the total number of parameters in a layer is $((h \times w \times p) + 1) \times d$.

4.4 The Pooling Layer

4.4.1 Tensors

Internally, Keras represents the weights of a neural network with **tensors**. Tensors are basically TensorFlow's version of a Numpy array with a few differences that make them better suited to deep learning. One of the most important is that tensors are compatible with [GPU](https://www.kaggle.com/docs/efficie gpu-usage) and [TPU](https://www.kaggle.com/docs/tpu)) accelerators. TPUs, in fact, are designed specifically for tensor computations.

A model's weights are kept in its 'weights' attribute as a list of tensors.

4.5 The Fully Connected Layer

4.6 Some other layers and implementations

4.7 Parameter Sharing in Convolutional Neural Networks

5 Recurrent Neural Networks

6 Training and Optimization

Training and optimization are the bread and butter of statistical and deep learning. A lot of what makes up a good neural network model is our ability to train and optimize it. What follows are some of the theory behind training the models, algorithms in order to do so, and optimizations to them that may or may not depend on the dataset we are working on.

6.1 Loss Function

Definition 6.1. Loss Function. A *loss* function is a real valued function of two variables denoted as $L(\theta, a)$ where $\theta \in \Omega$ and $a \in \mathbb{R}$. It denotes a measure of "loss" in accuracy from estimating a parameter θ as a. If θ is exactly equal to a then $L(\theta, a) = 0$. The a is fixed in the loss, hence, when minimizing we are minimizing over all θ in our parameter space Θ.

This is essentially how our models will "know" how bad their results/predictions are and what direction they should go with their parameters in order to get better results i.e. minimize this loss function.

Note: We often use a *Cost Function* which acts as an aggregate measurement of loss in our batch of predictions. Typically the cost function is the sample mean of the loss function. That is,

$$J(\vec{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\theta_i, a_i)$$

Lots of times people use loss and cost interchangeably but here we distinguish the two since it is the most agreed upon notation. Loss is the measurement for ONE example and cost is the aggregate measurement of loss in an entire batch of observations. The cost function essentially summarizes and estimates the loss for each sample and its minimization corresponds to minimizing the loss, or so we hope. The behavior of the cost function is the important and multiplying it by a positive constant doesn't affect the problem of minimizing loss so much. For example, you may see people multiply $\frac{1}{2}$ by the cost function when mean squared error is used since it simplifies the derivative a bit.

In theory, the best cost function to minimize is the expected value of our loss function over the training set,

$$J(\theta) = \mathbb{E}_{(\theta,a)\in X}(L(\theta, a))$$

however, the sample mean is a good estimate when the probability distribution of our data generating distribution is uniform or normal. Otherwise, we may need to use a different estimator but the sample mean is typically good (by good, we mean that it is unbiased and consistent). Note in our equation above that the expected value is taken over all pairs (θ, a), where each a is some fixed real number or vector.

6.1.1 Loss for Classification versus Regression

It is important now to keep in mind how classification and regression often require much different loss functions. For a regression problem, it is very typical to use squared error as your loss function and mean squared error as your cost function. However, in a classification problem, the notion of distance as in regression is not so clear. And we typically utilize a probabilistic approach with a decision boundary in order to classify our problems. But how do we characterize a loss function that has desirable properties such as differentiability in order to find its minimum and thus, optimize our solution? That is what we will build in the next section.

6.2 Entropy

Definition 6.2. Entropy. If X is a discrete random variable supported on χ with probability mass function p, then the entropy of X is defined as

$$H(X) = -\sum_{x\in\chi} p(x)\log p(x)$$

Note that $H(X) = -\mathbb{E}[\log(p(X))]$

Lemma 6.3. $H(X) \geq 0$.

Proof. $p(x) \geq 0$ for all $x \in \chi$ and thus, $\log p(x) \leq 0$. Further, this implies that $p(x)\log p(x) \leq 0$ for all x. Therefore, $\sum_{x\in\chi} p(x)\log p(x) \leq 0$ which implies that $-\sum_{x\in\chi} p(x)\log p(x) \geq 0$. □

Remarks. Entropy is a measure of the "uncertainty" of a random variable. The term $\log p(x)$ is referred to as the "information content" and $p(x)$ is just the weighted probability of that event occurring.

Also note that log is typically base 2 but not necessarily. In the case that it is base 2, then we refer to the real number $H(X)$ as the expected number of "bits" required to encode X. It forms a lower bound on the number of binary questions needed in order to properly encode the random variable X.

Entropy is a measure of uncertainty of a random variable when we know the probability mass function of this random variable. But what if we don't know p? Well, it doesn't actually change the probability of the event x occurring, but it certainly does change the information content with each associated event observed or question asked. So, we might stipulate that there is some similar idea of entropy but that only measures the added relative uncertainty to a random variable with an assumption of probability mass function q that may or may not be correct.

Definition 6.4. Relative Entropy/Kullback-Leichler Divergence. Let X be a discrete random variable supported on χ. The *relative entropy* $D(p||q)$ of the probability mass function p with respect to q is defined by

$$D(p||q) = \sum_{x\in\chi} p(x)\log\frac{p(x)}{q(x)}$$

Note that it is equivalently defined as $D(p||q) = -\mathbb{E}[\log\frac{p(X)}{q(X)}]$.

Discussion. Relative entropy is a measure of the distance between two distributions. In statistics, it arises as an expected logarithm of the likelihood ratio. It is a measure of inefficiency of assuming that the distribution is q when in fact the true distribution is p (Cover and Thomas, 2006).

Theorem 6.5. Information Inequality. Let $p(x), q(x), x \in \chi$, be two probability mass functions. Then

$$D(p||q) \geq 0$$

with equality if and only if $p(x) = q(x)$ for all x.

Remark. The proof of this theorem essentially follows from Jensen's Inequality and the fact that $\log t$ is a strictly concave function of t. We omit the proof since it is not the focus of our paper and we are developing this section simply for a very basic understanding of cross-entropy. If you

want to see a proof then page 28 of "Elements of Information Theory" by Cover and Thomas has one.

Before we introduce something called cross entropy, let's discuss what we've seen so far in the context of loss and the goal of classification. Suppose that given an input of observations with some number of features, we would like to classify the input data with as high precision and accuracy as possible. The classes that the data correspond to may or may not be deterministic, such as data with noise. However, this doesn't change the fact that whether or not the input data has a deterministic correspondence to the classes, their is an underlying true probability distribution associated with the input data and each of the classes which we wish to find. Similar to logistic regression, we may classify according to the class with the highest probability of the data belonging to it given an input. This requires getting as close as possible to the true probability distribution of each of the classes. That is, in order to classify as best as possible, we seek to get as *close* as possible to this *true probability distribution*.

As discussed before, relative entropy measures a notion of "distance" (it is not literally distance by definition. Note that it is not symmetric) between a true underlying probability distribution p and an assumed probability distribution q. Consider,

$$\begin{aligned} D(p||q) = \sum_{x\in\chi} p(x)\log\frac{p(x)}{q(x)} &= \sum_{x\in\chi} p(x)\log p(x) - \sum_{x\in\chi} p(x)\log q(x) \\ &= -H(X) - \sum_{x\in\chi} p(x)\log q(x) \end{aligned}$$

First, note that by the information inequality, $D(p||q) \geq 0$ with equality if and only if $p(x) = q(x)$ for all $x \in \chi$. Additionally, $-H(X) \leq 0$. This alone is enough to show that the term $-\sum_{x\in\chi} p(x)\log q(x)$ must be not only non-negative but also at least as large as $H(X)$ in magnitude. Otherwise, the relative entropy would be negative.

q is our assumed probability mass function here. In our application of classification using neural networks, q is the probability distribution of our classifications given an input of features. We seek to get q as "close" as possible to p. We don't know what p is, but say that we assume that a class is assigned to an input deterministically. Then, that means,

$$-\sum_{x\in\chi} p(x)\log q(x) = -\sum_{x\in\chi} \log q(x)$$

Hence, that term becomes a negative log loss term. This also means that the term $-H(X)$ of relative entropy is now equal to 0 since $p(x) = 1$ for each $x \in \chi$, thus, $\log p(x) = 0$ for each x. This leaves us to minimize the non-zero term above. This makes calculations simpler, but the assumption may seem unjustified in some cases. Even if that is true, it should be noted that no matter what p is, $H(X)$ is a constant, and we are only trying to minimize $D(p||q)$ by getting q as similar to p as possible. Hence, we are trying to obtain the solution to

$$\text{minimize}[-\sum_{x\in\chi} p(x)\log q(x)]$$

Solving this equivalently minimizes the relative entropy as well. As stated above, in some cases it is reasonable to assume that a class is assigned to an input deterministically. Even more so, it is reasonable in most cases to assume that the true probability distribution for each of the classes will be close, equal to, 1 for one of the classes and close, or equal, to 0 for the rest of the classes.

Hence, for most cases, we assume a deterministic association of input to classes and attempt to obtain the solution to

$$\text{minimize}[-\sum_{x\in\chi}\log q(x)]$$

Now to formalize this unnamed term of relative entropy a bit.

Definition 6.6. Cross-Entropy. The cross-entropy of the probability mass function p with respect to the probability mass function q (or the cross-entropy of q from p) is defined as,

$$CE(p,q) = -\mathbb{E}_p[\log q(x)] = -\sum_{x\in\chi} p(x)\log q(x)$$

Remark. Note that cross-entropy is part of the relative entropy of p with respect to q. Specifically, $CE(p,q) = H(X) + D(p||q)$.

Theorem 6.7. The cross-entropy of p with respect to q satisfies the inequalities,

$$0 \leq H(X) \leq CE(p,q)$$
$$0 \leq D(p||q) \leq CE(p,q)$$

Proof. Consider,

$$\begin{aligned} D(p||q) &= \sum_{x\in\chi} p(x)\log p(x) - \sum_{x\in\chi} p(x)\log q(x) \\ &= -H(X) + CE(p,q) \end{aligned}$$

But $D(p||q) \geq 0$ so,

$$\begin{aligned} -H(X) + CE(p,q) &\geq 0 \\ \Rightarrow CE(p,q) &\geq H(X) \end{aligned}$$

Now, entropy is also non-negative and from the definition of cross-entropy we have, $CE(p,q) = H(X) + D(p||q)$. Therefore,

$$\begin{aligned} CE(p,q) &= D(p||q) + H(X) \\ \Rightarrow CE(p,q) &\geq D(p||q) \end{aligned}$$

Therefore,

$$0 \leq H(X) \leq CE(p,q)$$
$$0 \leq D(p||q) \leq CE(p,q)$$

□

Remark. The reason we opt for the notation CE to denote cross-entropy is for convenience and clarity. In some other texts, such as "Deep Learning" by Goodfellow et. al. they denote cross-entropy as $H(p,q)$, however, this is easily confused with the notation for joint entropy. The joint entropy between random variables X and Y is typically denoted as $H(X,Y)$. Using CE avoids this needless confusion.

Remark. It should be known that entropy is a strictly concave function. The proof behind this is introductory material, however, for the sake of brevity in this paper we omit it.

Remark. If X has the Bernoulli distribution with parameter p then the entropy of X is defined and denoted by,

$$H(X) = H(p) = -p\log p - (1-p)\log(1-p)$$

Now we introduce binary cross entropy. This is the cross entropy used when there are only two classifications of a dataset. After we develop our notion of binary cross entropy we will generalize it to categorical cross entropy.

Definition 6.8. Binary Cross-Entropy Let X be a random variable supported on $\chi = \{x_1, x_2\}$. Let p be the probability mass function of X and let q be a probability mass function. Then the binary cross-entropy of p with respect to q is,

$$CE(p,q) = -p(x_1)\log q(x_1) - p(x_2)\log q(x_2)$$

Note that $p(x_1)+p(x_2) = 1$ and $q(x_1)+q(x_2) = 1$. Thus, $1-p(x_1) = p(x_2)$ and $1-q(x_1) = q(x_2)$. Then this is,

$$-p(x_1)\log[q(x_1)] - (1-p(x_1))\log[1-q(x_1)]$$

We may shorten the notation of this definition if we let $p(x_1) = p$ and $q(x_1) = q$. Giving us,

$$-p\log q - (1-p)\log(1-q)$$

As we discussed earlier, the true probability mass function is typically assumed to be deterministic. In that case, let $y = I(X = x_1)$, where I is the indicator function. We may rewrite binary cross entropy using this as

$$-y\log q - (1-y)\log(1-q) = -y\log q(y) - (1-y)\log(1-q(y))$$

Where we used the fact that $q(x_1) = q(y)$. This is typically how you will see it written in articles and tutorials online. Remember, this is for one single sample. This equation is what will typically be used as our loss function.

Definition 6.9. The Binary Cross Entropy Cost Function Let $C = [C_1 \dots C_m]^T \in \mathbb{R}^{m\times 1}$ where each C_i is a random variable supported on $\{x_1, x_2\}$ given a parameter $d_i \in \mathbb{R}^{1\times n}$, where $n \in \mathbb{Z}^+$. The binary cross entropy cost of Y is,

$$-\frac{1}{m}\sum_{i=1}^{m}\Big[p(C_i = x_1|d_i)\log q(C_i = x_1|d_i) + p(C_i = x_2|d_i)\log q(C_i = x_2|d_i)\Big]$$

If the classification of each data point is deterministic, then this is

$$-\frac{1}{m}\sum_{i=1}^{m}\Big[y_i\log q(y_i) + (1-y_i)\log(1-q(y_i))\Big]$$

where $y_i = I(C_i = x_1|d_i)$.

The cost function can be made more general than this, but we are defining it this way to motivate it in this the context of classification problems.

Up until this point, if you have ever read about binary cross entropy anywhere else, this may seem very convoluted. However, in order to justify these definitions and ideas, one needs to keep in mind many of the subtle, yet simple, key notions of probability and information theory.

Definition 6.10. Categorical Cross Entropy In a classification setting, the classification of a data point is typically deterministic given an input.

Let $y_k = \mathbb{P}(Y = k|X) = I(Y = k|X)$ where I is the indicator function and Y is a deterministic random variable supported on the set of classes $C = \{1, \ldots, k\}$. Let $\hat{y}_k = \Pr(\hat{Y} = k|X)$ where $\hat{Y}$ is a random variable. The categorical cross entropy of y_k with respect to $\hat{y}_k$ is

$$-\sum_{k=1}^{K} y_k \log \hat{y}_k$$

Typically the softmax function is used for our class probability estimate.

Definition 6.11. Categorical Cross Entropy Cost Function Given the definition of categorical cross entropy as our loss function, the categorical cross entropy cost of a sample of size m is,

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

6.2.1 Categorical Cross Entropy vs. Sparse Categorical Cross Entropy

Definition 6.12. Label Encoding. Assign each category a distinct integer value.

Definition 6.13. One-hot Encoding. Consider a list of classes $\mathscr{C}$ in some sort of arbitrary order (typically, it's in alphabetical order). If there are C classes, then the one-hot encoded classification is a vector $y \in \mathbb{R}^C$ where y is normalized and consists of one non-zero value equal to 1 in the c^{th} row corresponding to the c^{th} class in $\mathscr{C}$ to classify the data as.

That is, in order to classify the data as the c^{th} class in $\mathscr{C}$, then we output the vector $y = e_c \in \mathbb{R}^C$ where e_c is the c^{th} standard basis vector in $\mathbb{R}^C$.

The difference between categorical cross entropy loss and sparse categorical cross entropy loss lies in how we encode the class to which our output and target variables belong.

6.3 Fitting Neural Networks

6.3.1 Gradient Descent

6.3.2 Convexity

6.3.3 Local and Global Minima/Maxima

6.3.4 The Gradient Descent Procedure

Here is some pseudocode for the general procedure of gradient descent for one sample of data.

Algorithm 1 Compute Gradient:

Input: $\mathbf{X}, \vec{y}, \vec{w}, b$

Output: $(\frac{\partial J(\vec{w},b)}{\partial \vec{w}}, \frac{\partial J(\vec{w},b)}{\partial b})$

$(m,n) \leftarrow \mathbf{X}$.shape

$\frac{\partial J(\vec{w},b)}{\partial \vec{w}} \leftarrow \vec{0}$ ▷ Note this is a vector in $\mathbb{R}^n$ where n is the number of features

$\frac{\partial J(\vec{w},b)}{\partial b} \leftarrow 0$ ▷ Note this is a scalar

for $i = 0$ to $m-1$ **do**

  $\varepsilon \leftarrow \mathbf{X}[i] \cdot \vec{w} + b - \vec{y}[i]$

  for $0 \leq j \leq n$ **do**

    $\frac{\partial J(\vec{w},b)}{}$

    3tial w$_j \leftarrow \frac{\partial J(\vec{w},b)}{}$

    3tial w$_j + \varepsilon * \mathbf{X}[i,j]$ ▷ ε is the error/residual

  end for

  $\frac{\partial J(\vec{w},b)}{\partial b} \leftarrow \frac{\partial J(\vec{w},b)}{\partial b} + \varepsilon$

end for

$\frac{\partial J(\vec{w},b)}{\partial \vec{w}} \leftarrow \frac{\partial J(\vec{w},b)}{\partial \vec{w}} / m$

$\frac{\partial J(\vec{w},b)}{\partial b} \leftarrow \frac{\partial J(\vec{w},b)}{\partial b} / m$

return $(\frac{\partial J(\vec{w},b)}{\partial \vec{w}}, \frac{\partial J(\vec{w},b)}{\partial b})$

Algorithm 2 Gradient Descent:

Input: $\mathbf{X}, \vec{y}, \vec{w}_{in}, b_{in}, \nabla J(\vec{w},b), \alpha, u$

. ▷ ∇J is a function (compute_gradient), u - number of iterations

Output: $(\vec{w}, b)$

$\vec{w} \leftarrow \vec{w}_{in}$ ▷ Assign to $\vec{w}$ a copy of $\vec{w}_{in}$ to avoid writing over global

$b = b_{in}$

for $i = 0$ to u **do**

  $(\frac{\partial J(\vec{w},b)}{\partial \vec{w}}, \frac{\partial J(\vec{w},b)}{\partial b}) = \nabla J(\vec{w},b)$

  $w \leftarrow w - \alpha * \frac{\partial J(\vec{w},b)}{\partial \vec{w}}$

  $b \leftarrow b - \alpha * \frac{\partial J(\vec{w},b)}{\partial b}$

end for

return $(\vec{w}, b)$

6.3.5 Common Problems and Known Solutions

6.4 Automatic Differentiation

6.4.1 Back Propogation

6.5 Regularization

Before we get into algorithms that are based off of gradient descent for finding the minimum cost function, we will discuss methods and techniques used in deep learning in order to increase test accuracy.

6.5.1 Parameter Norm Penalties

6.5.2 Dataset Augmentation

Since the amount of data we have is limited and in practice, especially for neural networks, it is usually best to have more data to train on, we can utilize what is called data augmentation to artificially create more training data.

Consider our image classification task. Our classifier takes in a highly complicated, high-dimensional input x, the image, and summarizes it with a single category identity y. In order to do this effectively, it is essentially required of the classifier to be invariant to a wide variety of transformations.

For example, in our Bird classification problem, if we receive multiple images that are all of a mockingbird, and each image is either slightly cropped, shifted, or something similar, then our classifier should still output a mockingbird class for each image.

We can generate new (x, y) pairs easily just by transforming the x inputs in our training set. (Goodfellow et al., 2017) This is more general to tasks and data sets outside of image classification, but utilizing basic translations on images has proved to be a very effective technique of data augmentation in image classification and object recognition.

6.5.3 Early Stopping

6.5.4 Parameter Sharing

6.5.5 Ensemble Methods

6.5.6 Dropout

6.5.7 Batch Normalization

6.6 Optimization Algorithms

6.6.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an extension of the gradient descent algorithm that was created in order to deal with the massively computationally expense nature of the gradient descent algorithm. The expense coming from the need to calculate every component of the gradient every time the gradient is computed. SGD essentially updates the gradient by only calculating one or a group of components of the gradient.

6.6.2 SGD with Nesterov Momentum

6.6.3 AdaGrad

6.6.4 RMSProp

6.6.5 Adam Optimizer

6.6.6 A Note on Simulated Annealing

Simulated annealing is an optimization technique that mimics the annealing phenomenon that occurs in nature. It does so by using probabilistic random restarts in the where we start our gradient descent that decay over time in a similar manner to cooling of hot objects.

6.7 Pre-processing and Analysis

6.7.1 What is already done for us?

6.7.2 A Qualitative Check/Overview

6.7.3 Principal Components Analysis

6.8 Post Analysis

6.8.1 The Learning Curve

6.8.2 ROC and AUC

7 Our Data

8 Code Overview

We used the TensorFlow machine learning platform for our project implementations. In addtion, we used to many other standard packages and libraries used in data science as shown below.

9 Results

9.1 Data 1

9.1.1 The primary model

Feedforward Neural Network.

9.1.2 Other models

- Multiple Logistic Regression
- Support Vector Machines
- Decision Trees

9.1.3 Test accuracy and error

- Feedforward Neural Network -
- Multiple Logistic Regression
- Support Vector Machines
- Decision Trees

9.1.4 Learning Curve

9.1.5 ROC

9.2 Data 2

9.2.1 The primary model

Convolutional Neural Network

9.2.2 Other models

- Multiple Logistic Regression
- Support Vector Machine with Radial Basis Function kernel
- Decision Trees

9.2.3 Test accuracy and error

- Convolutional Neural Network.
- Multiple Logistic Regression
- Support Vector Machines
- Decision Trees

9.2.4 Learning Curve

9.2.5 ROC

Definition 9.1. Precision

$$\frac{T_P}{T_P + F_P}$$

Definition 9.2. Recall

$$\frac{T_P}{T_P + F_N}$$

9.3 Data 3

9.3.1 The primary model

Recurrent Neural Network.

9.3.2 Other models

9.3.3 Test accuracy and error

- Recurrent Neural Network -
- Multiple Logistic Regression
- Support Vector Machines
- Decision Trees

9.3.4 Learning Curve

9.3.5 ROC

9.4 Benchmark Results and Comparisons

9.4.1 Data 1

9.4.2 Data 2

9.4.3 Data 3

10 Conclusions, Summary, Questions

How do our models compare to other modern approaches? How do they compare to models of the past? If there are any Kaggle implementations on the same dataset, what did they do differently?

References

Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory*. John Wiley & Sons, Inc.

Goodfellow, I., Bengio, Y., and Courville, A. (2017). *Deep Learning*. MIT Press.