A Review of Deep Learning Architectures and Application of Transfer Learning for Image Classification

John D. Wise
jdwise@uab.edu

Baocheng Geng
bgeng@uab.edu

Abstract

Deep Learning is a vast field within computer science which has made tremendous progress in its application and abilities in the past decade. Some of the applications of deep learning include image recognition, language translation, neural audio effects, self-driving automobiles, and even cancer diagnosis. In this paper we will explore some of the foundational architectures of deep learning as well as their various applications. We will then go on to discuss our hands on experience with transfer learning using ResNet, an architecture for the task of image classification.

1 Introduction

Artificial Intelligence (AI) and Machine Learning (ML) have become modern buzzwords in the world of business and science, with ninety one percent of leading companies investing in AI on an on-going basis and one in twelve startups using some form of AI [1][2]. Some of the applications include online purchase recommendations, advertisements, fraud detection, autonomous driving cars, stream history-influenced video viewing recommendations, and orthopedic medicine [3]. With the vast array of applications of AI and ML, it's easy to understand why such a high percentage of companies are interested in the

Preprint. Under Review.

technology. In this paper we will explore some of the foundational architectures of deep learning as well as their various applications. We will then go on to discuss our hands on experience with transfer learning using ResNet architecture for the task of image classification.

Machine Learning is a subset of Artificial Intelligence which is focused in experiential learning on large datasets which ideally allow a model to produce a correct or reasonable output given an input [3]. In the traditional paradigm of software engineering, a programmer would write a program, which could be thought of as a series of procedures or rules, which would solve a real-world problem. The logic of the program would be explicitly written in the code. Machine Learning is a complete flip of this paradigm. In machine learning we train models to learn mappings from inputs to outputs or features to targets respectively. Using a loss function to know how bad its prediction was, the model adapts during training and gradually gets better at this mapping [4]. As stated by Chollet, "Learning, in the context of machine learning, describes an automatic search process for data transformations that produce useful representations of some data, guided by some feedback signal—representations that are amenable to simpler rules solving the task at hand."

Deep Learning is a subset of Machine Learning which uses *neural networks* to perform this mapping from inputs to outputs. The 'deep' in Deep Learning is a reference to the principal idea of using successive layers to create abstract representations from features of the input data. Depth, in this sense, refers to how many layers are contained within a model. To say one model has more depth than

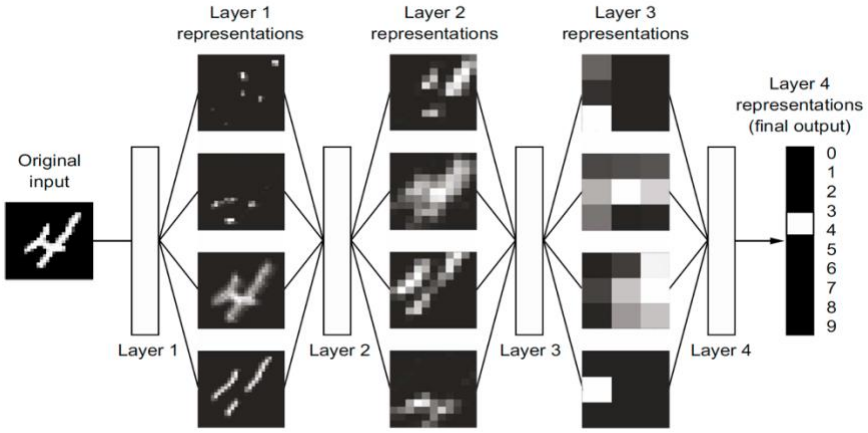


Figure 1: Data Representations learned by a digit-recognition model

another is not to say that it is more sophisticated, rather it just means that one model has more layers than another. In general, deep learning is a mathematical framework for learning and connecting representations of data [4].

1.1 A Brief History

The idea of Artificial Intelligence (AI) was a pioneered by Alan Turing in his 1950 paper, Computing Machinery and Intelligence [5]. Though AI was only presented as an idea in his paper, Turing laid some foundational thoughts that still impact how we think about AI, such as the Turing Test. AI transitioned from thought to reality in 1956 when John McCarthy organized a summer workshop called Dartmouth Summer Research Project on Artificial Intelligence (DSRPAI) where Allen Newell, Cliff Shaw, and Herbert Simon presented the Logic Theorist. The Logic Theorist was designed to mimic the problem-solving skills of a human, thus a realization of Turing's original work [4][5].

The early attempts to build AI systems involved programmers explicitly coding rules which were followed by a program. During this time most experts believed that human-level artificial intelligence could be achieved by creating a sufficiently large set of explicit rules for manipulating knowledge stored in explicit

databases. This approach is known as *symbolic AI* [4]. According to Chollet, symbolic AI "was the dominant paradigm in AI from the 1950s to the late 1980s, and it reached its peak popularity during the expert systems boom of the 1980s." [4].

In 2011, Dan Ciresan had the first practical success of modern deep learning when he began to win academic image-classification competitions with GPU-trained deep neural networks. But the tipping point came in 2012, with the entry of Hinton's group in the yearly large-scale image-classification challenge ImageNet. At the time, the top-five accuracy of the winning model, based on classical approaches to computer vision, was only 74.3%. Then, in 2012, a team led by Alex Krizhevsky was able to achieve a top-five accuracy of 83.6%. By 2015, the winner reached an accuracy of 96.4%, and the classification task on ImageNet was considered to be a completely solved problem. Since 2012, deep convolutional neural networks (convnets) have become the go-to algorithm for all computer vision tasks; more generally, they work on all perceptual tasks [4].

2 How Neural Networks Learn

As discussed earlier, a deep learning model "learns" abstract representations of input data in each successive layer of the network. In general, we can think of a machine learning model as a high dimensional function which maps inputs to outputs. This mapping is created by training the model on some pre-existing dataset. There are generally two paradigms when it comes to datasets and training neural networks, *supervised* and *unsupervised* learning. In supervised learning our data sets are already labeled. In unsupervised learning, ML models learn how to

analyze and cluster unlabeled data [7]. In this paper we will focus on supervised learning.

Let's discuss an overview of the learning process. At first, we separate our dataset into training and testing data. We present the network with training examples from the training data which consist of input data together with their desired outputs. We then quantify how closely the actual output of the network matches the desired output using a loss function. Next, we change the weight of each connection so that the network produces a better approximation of the desired output in a algorithm called backpropagation [6]. Once we have completed the training, we evaluate our model's performance on testing data. This allows us to get an understanding of how well our model performs on data it did not see during training, often referred to as generalization or how well the model generalizes.

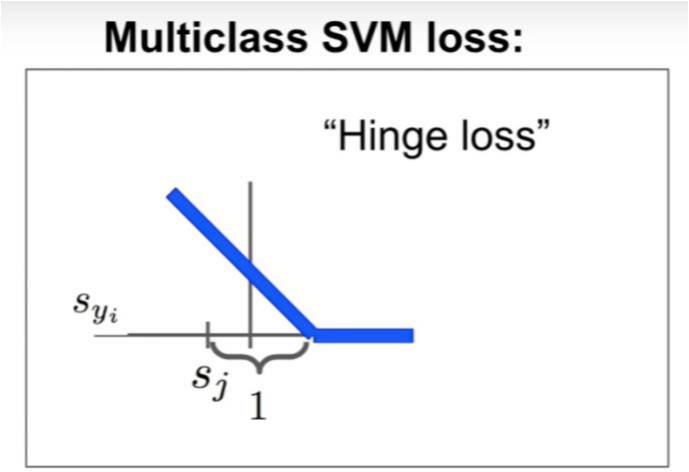
2.1 Loss Functions

Loss functions are mathematical methods we use to quantify how right or wrong the network is at mapping the input to its output. We use this information to steer the weights of the network such that we can reduce the loss over repeated training sessions [4]. We will use the multi-class SVM loss function as a way to introduce this concept.

Let's suppose we have example (x_i, y_i) where x_i is the input y_i is the label and we provide x_i to our model, our model will return s which is represented as a vector of scores shown in Layer 4 of Figure 1. We can thus define the SVM loss as follows:

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$
$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

where s_j is the score of the j-th category and s_{y_i} is the score of correct class in the i-th training set. Here we can see that the loss L_i is the sum of these differences between the correct category and all other categories. From this formula we can also see that negative losses result in a zero. This is called a *hinge loss* which is derived in the plot of the function as shown below .



The Multi-class SVM loss is one of many loss functions which can be used to quantify how right or wrong the network is at mapping the input to its output. Covering these loss functions and their domain application goes beyond the scope of this paper, however, now that we understand what the loss function is and how it can be used to objectively score a neural network model, we can move on to back propagation and gradient descent.

2.2 Optimizing the Network Using the Loss Function: Stochastic Gradient Descent and Backpropagation

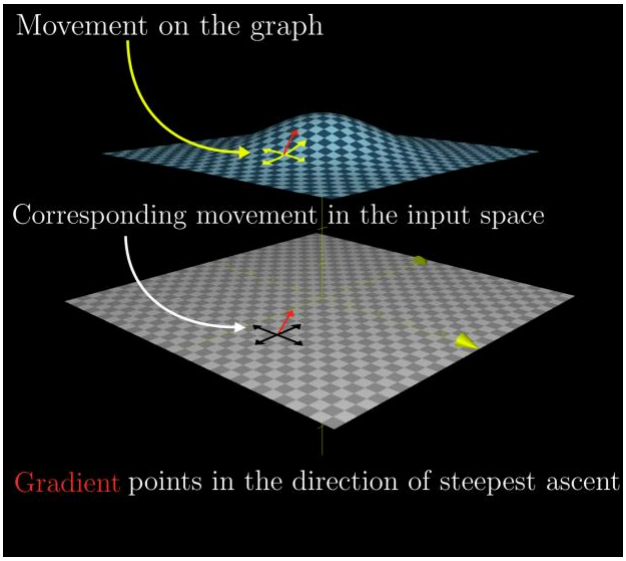
As discussed in the previous section, the loss function provides us a way to objectively quantify how bad a model is at mapping inputs to the right outputs. Therefore, we will seek a way to reduce the loss which is the goal of *optimization*. Let's discuss some of the methods we could use to optimize the loss function.

The most elementary method we could use is to randomly adjust the weights of the network and then evaluate whether the loss function has reduced. Though this may end up resulting in a reduce in the loss function, it is very inefficient. Every time the weights are updated, we have the cost of inference to see if the updates have resulted in a desired result. Maybe we can create a better way.

Let's explore the concept of directed adjusting of the weights. One method we can use for this is to perturb the value of a weight and evaluate if reducing or increasing its value results in a reduction of the loss function. Though this will allow us to improve the performance of the network more reliably than randomly adjusting the weights, this method is still inefficient. The reason is that we double how many times we must evaluate the performance of the network- once for a lower weight value and another for a greater weight value.

Though this past method was not the best approach, it does motivate a method that can be more efficient. When we perturb the value of the weight by a small value and evaluate the resultant value, it seems very similar to taking the derivative of the loss function at a certain point. Let's explore that idea in more detail.

Using the impetus of the derivative, we will evaluate a calculus-based concept called *gradient descent*. Given a point on the surface of a function, the gradient is a vector that represents which direction to move in order to increase the value of the function the most.



With gradient descent, we use the reciprocal of the gradient to find which direction to move in order to decrease the value of the function the quickest. This is exactly what we want for the loss function. We want to know how to adjust the weights of network such that the loss function reduces the fastest. Another huge benefit to this is that we don't need to evaluate the network. This is by far the most efficient method compared to the methods we have explored previously, but it still leaves many questions and challenges. How do we take these derivatives? Can we use gradient descent in every case?

One important thing to note is that we can only find this gradient if the function is *differentiable*. A function is differentiable if the derivative exists at every point in its domain. Consequently, the only way for the derivative to exist is if the function is *continuous* on its domain [8]. A function $f(x)$ is continuous at a point a,

if the function's value approaches $f(a)$ when x approaches a for all a in the domain of $f(x)$ [9].

Backpropagation is a way to use the derivatives of simple operations to easily compute the gradient of arbitrarily complex combinations of atomic operations [4]. In neural networks these atomic operations include operations such as addition, ReLU, or tensor product. A neural network consists of many of these operations chained together, each of which has a simple and known derivative. From calculus we know that we can compute the collective derivative of these chained operations using the *chain rule*. Applying the chain rule to compute the gradient values of a neural network gives rise to an algorithm called backpropagation [4].

There are many types of gradient descent algorithms. These include Stochastic Gradient Descent, Stochastic Gradient Descent with momentum, Mini-Batch Gradient Descent, Adam, and many more [10]. Which optimizer to use is largely dependent on your application [10]. We will discuss some of these in the next section.

2.3 Optimizers

Now that we have found a vector which defines what direction we should move in order to reduce the loss function, we have not discussed how far we should move in that direction, a hyperparameter called *step size*. During back propagation, we may find local minima which makes it seem like we have reduced the loss function as much as possible. However, if we were to increase the step size we may be able to move past that local minima such that we continue towards a location in

the weight space which is closer to the absolute minimum. This is what motivates us to use an *optimizer* to optimize the learning process.

An optimizer is a function or an algorithm that modifies the attributes of the neural network, such as weights, learning rate, and step size [10]. We briefly mentioned that momentum can be used as a factor when determining

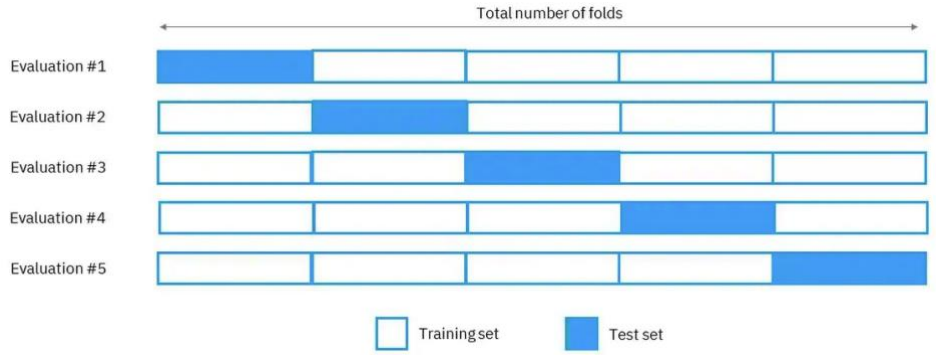
2.4 Over Fitting

As we've explored the concepts of the loss function and backpropagation, you may be thinking that our goal would be to reduce the loss function as much as possible such that our model has 100% accuracy on the data used during training. However, this is not ideal in the real-world and we will explain why in this section. Let's say that we train our model to have 100% accuracy on the data used during testing, but when we evaluate this model on data not seen during training the model achieves only 75% accuracy. This brings to light a problem faced in machine learning called *overfitting*.

Overfitting occurs when a statistical model fits exactly against its training data. When this happens, the algorithm cannot perform accurately against unseen data [9]. This defeats the goal that we have for our model, mainly that it performs well on data seen in real-world applications which may not have been seen during training. When the model memorizes and fits too closely to the training data, the model becomes "overfitted," and this is unable to generalize well to new data.

One of the most popular ways we can assess the accuracy of the model to data unseen during training is to use n-fold cross-validation. In n-folds cross-validation, data is split into n equally sized subsets, also called "folds." One of the n-folds will

act as the test set and the remaining folds will train the model. This process repeats until each of the folds has acted as a test set. After each evaluation, a score is retained and when all iterations have completed, the scores are averaged to assess the performance of the overall model [9]. This process can be shown graphically as shown below. Using the n-fold cross validation training technique, we can vary what data is seen during training and more accurately evaluate the performance of the network.



2.5 Activation Functions

Let's consider now linear layer in a neural network. Without an activation function, a dense layer would consist of only two linear operations – a dot product and an addition [4]. The output of this layer would be described by:

$$output = dot(input, W) + b$$

The problem is that the layer could only learn linear transformations of the input data, i.e. the hypothesis space of the layer would only be the set of all possible linear transformations of the input data. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations because a stack of linear layers would still implement a linear operation [4]. To access a richer hypothesis space that would benefit from deep representations, you need a non-

linearity – or activation function [4]. As you can imagine, there are many activation functions each with their own pros and cons and each performing better or worse than other options depending on the application.

There are many things to consider when choosing an activation function, but first, what makes a function a good candidate to be an activation function? According to Jain, we would need activation functions that have the following properties [11]:

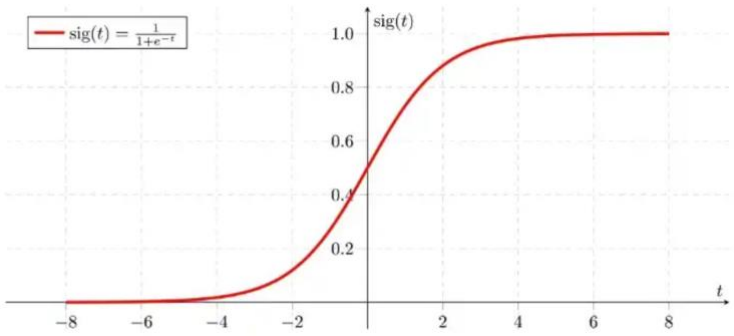
1) Zero-Centered
   Output of the activation function should be symmetrical at zero so that the gradients do not shift to a particular direction [11].

2) Computationally Inexpensive
   Activation functions are applied after every layer and need to be calculated millions of times in deep networks. Because of this, activation functions should be computationally inexpensive.

3) Differentiable
   As mentioned, neural networks are trained using the gradient descent process, hence the layers in the model need to differentiable or at least differentiable in parts. This is a necessary requirement for a function to work as activation function layer [11].

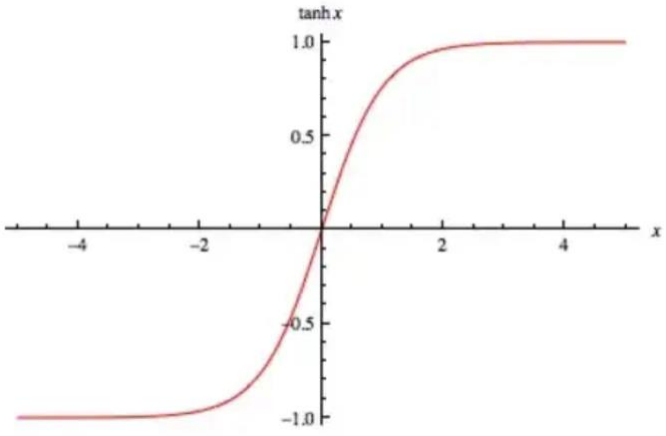
4) Avoids vanishing gradients
   When n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers [12]. This results in a network that learns too slowly or isn't able to learn at all.

With those four criteria, let's review some popular activation functions and discuss how well they match each criteria. The function that is used most often as an example when considering these functions is the Sigmoid function.



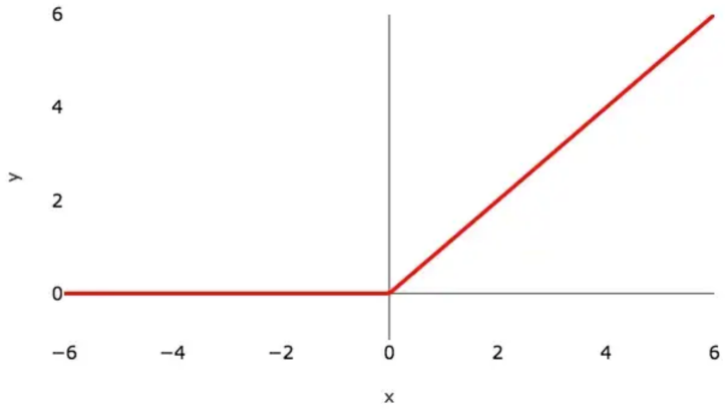
As you can see in the graph above, the sigmoid function is differentiable and squeezes any value of t to be between 0 and 1. However, it is not zero centered, it is computationally expensive due to the exponential in the denominator, and it does not avoid the vanishing gradient problem. The last point is not as straight forward, but let's consider a high value of t. At a high value in the positive or negative direction, the derivative gets closer and closer to 0. This creates the vanishing gradient problem during backpropagation. Because of these issues the sigmoid function is never recommended to use in an actual network and usually serves to educate rather than to be applied to a real-world network [11].

Next, we will discuss the hyperbolic tangent function, colloquially referred to as tanh.



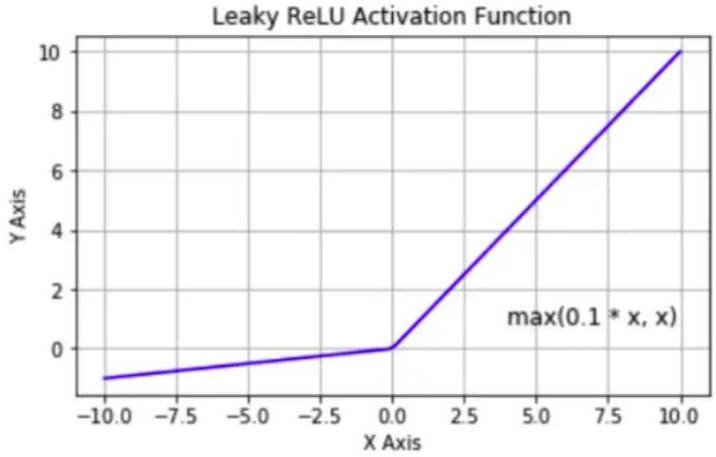
As we can see in the graph of tanh, we still have the same problems of vanishing gradient and computational expense that we faced with the sigmoid function. However, tanh is zero centered which is slightly better than the sigmoid function.

Next, we will discuss the Rectified Linear Unit, or ReLU, function.



This function is much different than the prior functions we have covered. In the ReLU function we see that it is in fact nonlinear, but there aren't many other pros to this function. We still face the problems of not being zero-centered, and the vanishing gradient problem for negative values. One of the biggest pros of this function though is that it is very computationally inexpensive because the positive regime is linear and the negative regime is constant. Because of this, this function is sometimes used in real-world models. However, the Leaky ReLU, a variant of the ReLU function, is more often used.

Let's discuss the pros and cons of the Leaky ReLU function, shown below.



In the Leaky ReLU we see that it is zero-centered, differentiable, computationally inexpensive, and avoids the vanishing gradient problem, unlike the standard ReLU function. Because of this, the Leaky ReLU is widely used in deep learning models.

2.6 Weight Initialization Schemes

Weight initialization is a critical component in deep learning, but it is often glossed over in texts. The reason may be that weight initialization is usually handled with whatever deep learning framework you use. However, it is important because having the right weight initialization will determine the behavior of the network and could determine if the network will converge at all [13]. Weight initialization is a procedure to set the weights of a neural network to small random values that define the starting point for the optimization of the neural network [14].

Usually, we will initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The scale of the initial distribution has a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize [13]. These random values are usually between -1 to 1, -0.3 to 0.3, or 0 to 1 [14]. Nevertheless, more modern approaches have been

developed that have become the defacto standard given they may result in a slightly more effective optimization (model training) process. These modern weight initialization techniques are divided based on the type of activation function used in the nodes that are being initialized, such as Sigmoid, Tanh, or ReLU [14].

The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or Tanh activation function is called Glorot or Xavier initialization after its inventor Xavier Glorot. The Xavier initialization method is calculated as a random number with a uniform probability distribution U between the range $-\frac{1}{\sqrt{n}}$ and $\frac{1}{\sqrt{n}}$, where n is the number of inputs to the node. The Xavier weight initialization was found to have problems when used to initialize networks that use the ReLU activation function [14]. The standard approach for weight initialization of neural networks that use the ReLU activation function is called "he" initialization after its creator Kaiming He. The He initialization method is calculated as a random number with a Gaussian probability distribution G with a mean of 0.0 and a standard deviation of $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the node [14].

2.7 Batch Normalization

As discussed in the prior sections, training a model with many layers can be difficult and initialization of the weights of a network can have a large effect on how well the network is able to perform. One possible reason for this difficulty is the distribution of the inputs to layers deep in the network will likely change after each mini batch when the weights are updated. This can cause the learning algorithm to

chase a moving target. This change in the distribution of inputs to layers in the network is referred to by the technical name *internal covariate shift*. *Batch normalization* is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini batch. *Batch normalization* has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks [15].

3 Architectures

Artificial neural network (ANN) is the underlying architecture behind deep learning. Based on ANN, several variations of the algorithms have been invented [16]. Now that we have discussed what neural networks are, how they learn, and the theory behind some of the foundational concepts, let's dive into the application of some common network architectures.

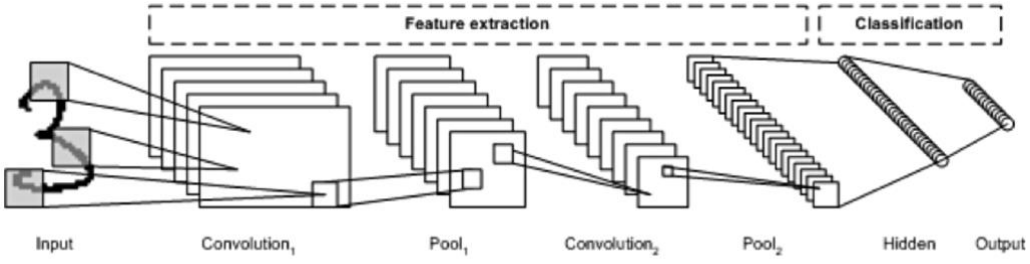
3.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) may be one of the most ubiquitous neural network architectures currently. A CNN is a multilayer neural network that is said to be inspired by the animal visual cortex. The first CNN was created by Yann LeCun in order to recognize handwritten characters, such as postal code interpretation. Early layers of the network recognize features and later layers recombine these features into higher-level abstractions of the input [16].

Networks used for image classification are usually two dimensional CNNs, and those used for audio effects and classification are usually one dimensional CNNs. The 2D CNNs are most popular and are referred to colloquially as CNNs. There are many different variations of convolutional neural networks. In fact, many of the

most famous architectures, such as ResNet, AlexNet, and LeNet, are CNNs. In this section we will just cover LeNet, however we will also explore ResNet in later sections.

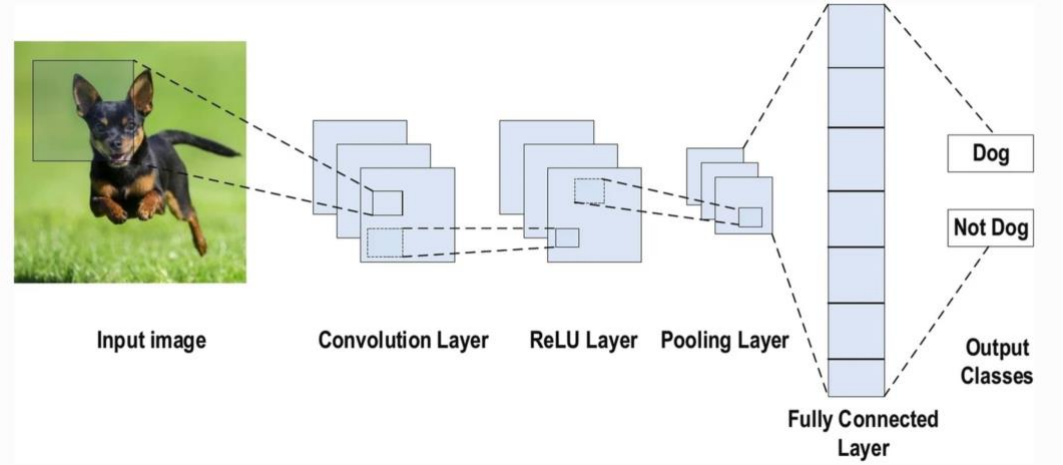
The LeNet CNN architecture is made up of multiple layers that implement feature extraction and then classification as shown below. The image is divided into receptive fields that feed into a convolutional layer by convolving filters called *kernels* over the image which serves to extract features from the input image. The next step is pooling, which is used to reduce the dimensionality of the extracted features through down-sampling while retaining the most important information. Another convolution and pooling step is performed that feeds into a fully connected multilayer perceptron. The final output layer of this network is a set of nodes that identify features of the image [16].



In addition to image processing, CNNs have been successfully applied to video analysis and various tasks within natural language processing [16]. Now that we have a general idea of what a convolutional neural network is, lets discuss the theory and function of these networks in more detail.

Unlike conventional fully connected (FC) networks, CNNs employ shared weights and local connections to make full use of 2D input-data structures like image signals. This operation utilizes an extremely small number of parameters,

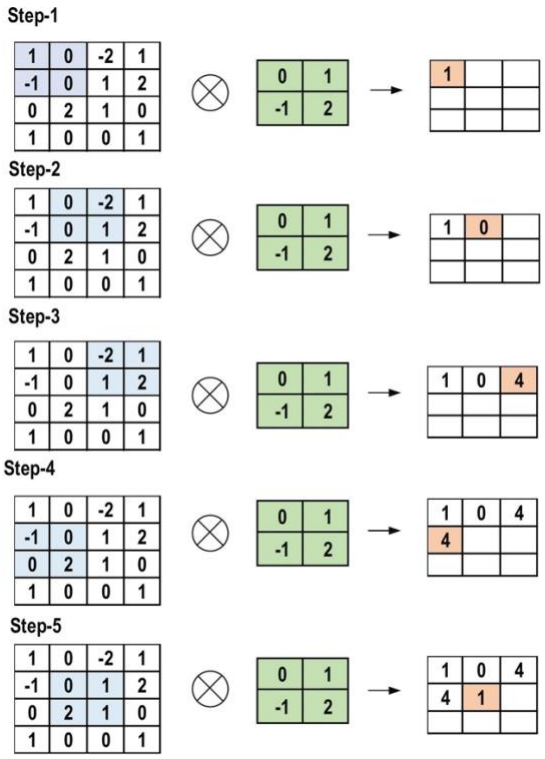
which serves to simplify the training process and speed up the network [17]. The most significant component of this architecture is the convolutional layer. It consists of a collection of convolutional filters (so-called kernels). The input image is convolved with these filters to generate the output feature map [17]. For the next few paragraphs lets move our attention from LeNet to a generic CNN as shown below.



A grid of discrete values, called the kernel weight, defines the kernel. As discussed in prior sections, random numbers are assigned to act as the weights of the kernel at the beginning of the training process. Next, these weights are adjusted at each training era; thus, teaching the kernel to extract features of interest.

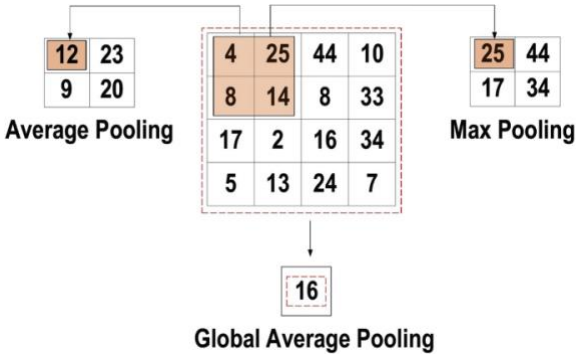
To understand the convolutional operation in more depth, let's discuss an example of a 4×4 gray-scale image with a 2×2 random weight-initialized kernel. First, the kernel slides, or convolves, over the whole image horizontally and vertically. During this process, the dot product between the input image and the kernel is determined, where their corresponding values are multiplied and then summed up to create a single scalar value. The calculated dot product values represent the feature map of the output. The figure below graphically illustrates

the primary calculations executed at each step. In this figure, the light green color represents the 2×2 kernel, while the light blue color represents the area of the input image. Both are multiplied; the end result after summing up the resulting product values – shown in a light orange color- represent an entry value to the output feature map [17].



Each convolutional layer is followed by a pooling layer. The main task of the pooling layer is the sub-sampling of the feature maps created in the convolution process. Similar to the convolutional operation, a kernel is used to convolve over the activation maps and both the stride and the kernel are initially size-assigned before the pooling operation is executed. Several types of pooling methods exists which include tree pooling, gated pooling, average pooling, min pooling, max pooling, global average pooling (GAP), and global max pooling. The most frequently utilized pooling methods are the max, min, and GAP pooling shown in the figure below. Sometimes the overall CNN performance is decreased as a result of the

pooling layer which is the main shortfall of the pooling layer. The reason for the decrease in performance is that the pooling layer helps the CNN to determine whether or not a certain feature is available in the particular input image, but focuses exclusively on ascertaining the correct location of that feature causing the CNN model misses the relevant information [17].

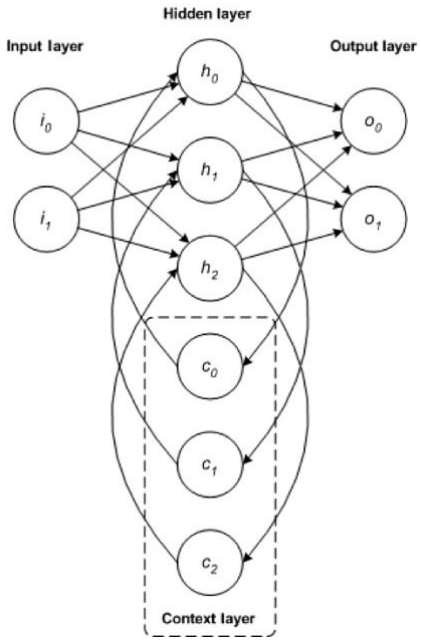


Usually a fully connected (FC) layer is located at the end of a CNN architecture. Inside a fully connected layer, each neuron is connected to all neurons of the previous layer. It follows the basic method of the conventional neural network. The input of the FC layer comes from the last pooling or convolutional layer. This input is in the form of a vector, which is created from the feature maps after flattening. The output of the FC layer represents the final CNN output and thus it is utilized as the CNN classifier [17].

3.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are another commonly employed architecture in deep learning and are mainly applied in the area of speech processing and NLP contexts where sequencing of information is important [17]. The primary difference between a typical multilayer network and a recurrent network is that rather than completely feed-forward connections, a recurrent network might have connections that feed back into prior layers. This feedback

allows RNNs to maintain memory of past inputs and model problems in time [16]. Since the embedded structure in the sequence of the data delivers valuable information, RNNs are fundamental to a range of different applications. For example, it is important to understand the context of the sentence in order to determine the meaning of a specific word in it. Thus, it is possible to consider the RNN as a unit of short-term memory, where x represents the input layer, y is the output layer, and s represents the state (hidden) layer [17]. A typical unfolded RNN diagram is illustrated below.



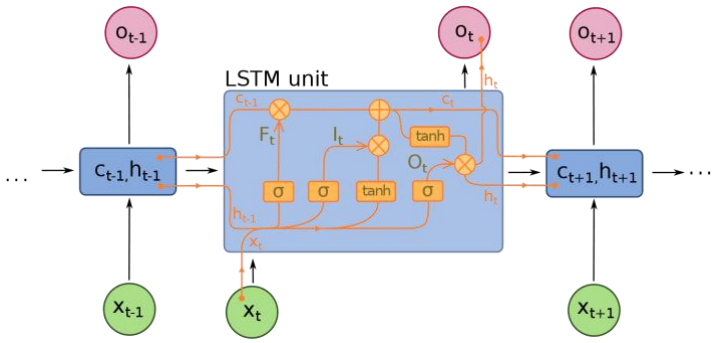
One of the main issues with RNNs are their sensitivity to the exploding gradient and vanishing problems. During the training process, the reduplications of several large or small derivatives may cause the gradients to exponentially explode or decay [17].

3.2.1 Long Short-Term Memory Unit

The Long Short-Term Memory Unit (LSTM) was created in 1997 by Hochreiter and Schimdhuber, and has grown in popularity in recent years as an RNN

architecture for various applications [16]. The LSTM introduced the concept of a *memory cell*. A memory cell can retain its value for a short or long time as a function of its inputs. This allows the cell to remember what's important and not just its last computed value [16].

The LSTM memory cell contains three gates that control when and how information flows into or out of the cell. The input gate controls when new information can flow into the cell. The forget gate controls when an existing piece of information is forgotten. This allows more recent data to be considered by the network over old data. Finally, the output gate controls when the information that is contained in the cell will be used in the output from the cell. The cell also contains weights which control each gate [16]. Shown below is a diagram of the memory cell shown in the greater context of the RNN network.

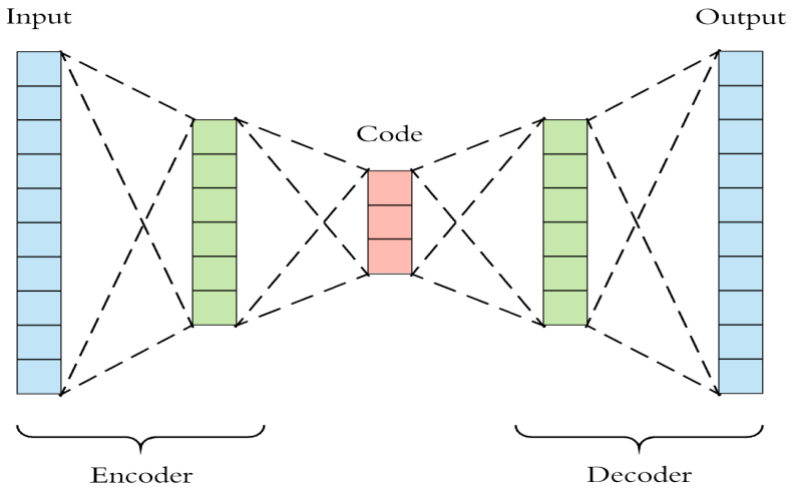


3.3 Auto Encoder/Decoder

The first known usage of Auto-Encoders (AEs) was found to be by LeCun in 1987 [16]. The central idea of AEs is to take some data of high dimension, represent this data in a lower dimensional latent representation of the input, which is the role of the encoder, and then up-sample the latent representation to try to most accurately reconstruct the input which is the role of the decoder. The term Auto Encoder is often used to describe an Auto Encoder/Decoder network; however, the

role of the encoder and decoder differ greatly. In this section we will further contribute to the overloading of the term and refer to an Auto Encoder/Decoder as an Auto Encoder or AE for short.

In general, this variant of an ANN is composed of input, hidden, and output layers. The input layer is encoded into the hidden layer using an appropriate encoding function. The number of nodes in the hidden layer is much less than the number of nodes in the input layer, creating a compressed or latent representation of the original input. Lastly, the output layer aims to reconstruct the input layer by using a decoder function.



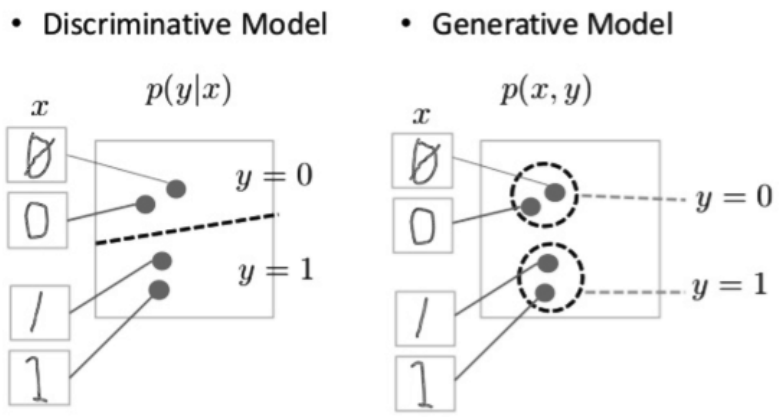
At first it may not make much sense as to why we would want to regenerate the input; however, there are a few applications where this makes sense. Let's say we have an image that is blurry, and we would like to up-sample it to reduce the noise. An Auto Encoder/Decoder can be used to recreate the image and the decoder could be trained to up-sample the image. We can also use the decoder part of the network in generative neural networks to create images from latent representations of a piece of text which is used to describe the image to be

generated by the network. It is this last application which motivates our next section on Generative Adversarial Networks.

3.4 Generative Neural Networks

The word "Generative" in Generative Neural Networks describes a class of statistical models that contrasts with the discriminative models which we have discussed in detail in sections prior. Informally, Generative models can generate new data instances. More formally, given a set of data instances X and a set of labels Y, Generative models capture the joint probability $p(X, Y)$, or just $p(X)$ if there are no labels [18].

A generative model for images might capture correlations like "things that look like cars are probably going to appear near things that look like roads" and "fingers are not likely to appear on feet." Both of these examples are very complicated distributions. In contrast, a discriminative model might learn the difference between "dog" or "not dog" by just learning from a few examples. Discriminative models attempt to find boundaries in the data space, while generative models try to model how data is placed throughout the space [18]. This is illustrated well in the figure below.



3.4.1 Generative Adversarial Networks

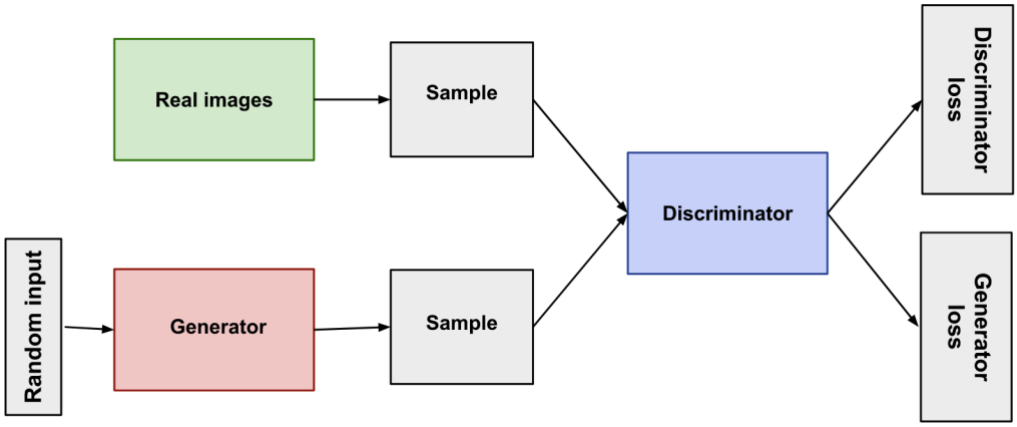
A generative adversarial network (GAN) has two parts: the generator and the discriminator. The *generator* learns to generate data which would be plausible in the data domain. The *discriminator* learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results which is where term adversarial comes from [19].

When training begins, the generator is not very good at producing convincing data so the discriminator can easily tell that it's fake [19].

As training continues, the generator becomes better at generating convincing data which is better able to fool the discriminator [19].

Finally, the generator becomes so good at generating convincing data that the discriminator can no longer tell the fake from the real data [19].

A diagram of the system is shown below.



4 A Deep Dive into Transfer Learning with ResNet

Now that we have strong foundational knowledge of deep learning, let's discuss the final project. In this section we utilize a method called transfer learning which allows us to transfer the knowledge from a pre-existing model to a new model in order to extend that models classification ability.

4.1 Objective

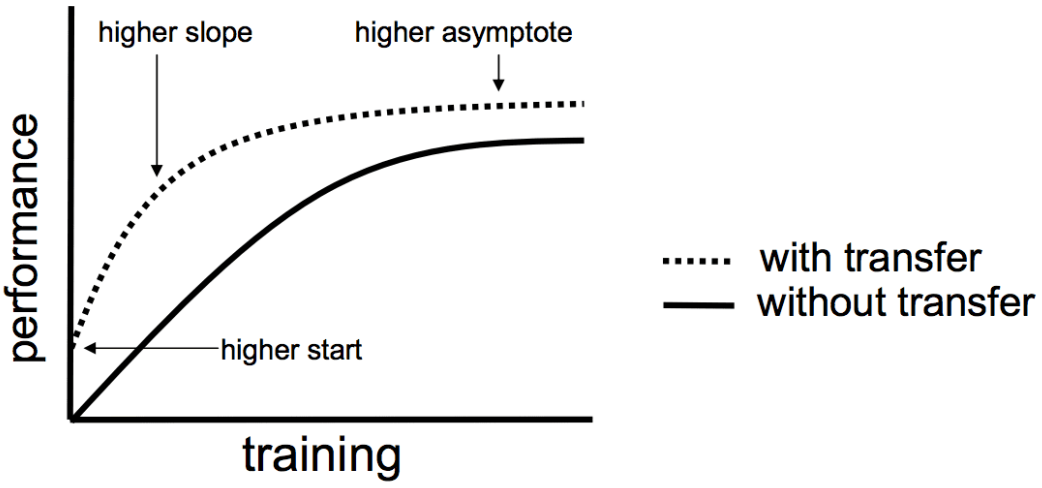
The objective for this assignment was to utilize transfer learning with a model called ResNet-50 and extend ResNet's functionality to classify dog breeds. We then tuned the network using various hyper parameter configurations. The data set for this assignment can be found here.

4.2 Transfer Learning

Transfer learning is a machine learning method where a model developed for a certain task is reused as the starting point for a model on a different task. It is a popular approach in deep learning where pre-trained models are used as the starting point given the vast compute and time resources required to develop

neural network models on certain problems and from the huge jumps in skill that they provide on related problems [20].

Lisa Torrey and Jude Shavlik describe three possible benefits to look for when using transfer learning. First, transfer learning enables a higher start meaning the initial skill (before refining the model) on the source model is higher than it otherwise would be. Second, transfer learning enables higher slope meaning the rate of improvement of skill during training of the source model is steeper than it otherwise would be. Lastly, transfer learning allows a higher asymptote meaning the converged skill of the trained model is better than it otherwise would be [20].



4.3 ResNet

The first ResNet architecture was the Resnet-34 which involved the insertion of shortcut connections in turning a plain network into its residual network counterpart. The plain network was inspired by VGG neural with the convolutional networks having 3×3 kernels. However, compared to VGGNets, ResNets have fewer filters and lower complexity. While the input and output dimensions were the same as VGG, the identity shortcuts were directly used [20].

The Resnet50 architecture is based on Resnet-34 with one major difference. In Resnet50 the building block was modified into a bottleneck design and each of the 2-layer blocks in Resnet34 was replaced with a 3-layer bottleneck block. This resulted in a much higher accuracy than the previous 34-layer ResNet model [20].

4.4 Method

In this project, we used PyTorch which is a python-based machine learning framework. Using PyTorch we are able to use a pre-trained ResNet50 model with the following line of code.

```
resnet = models.resnet50(weights='ResNet50_Weights.DEFAULT')
```

We kept the ResNet50 model the same by disabling gradients in the ResNet layers while we trained our network to learn classification of dog breeds. This can be seen in the following lines of code.

```
# freeze all model parameters
for param in resnet.parameters():
    param.requires_grad = False
```

Next, we were able to create new Fully Connected layers at the end of the model to perform the dog breed classification.

```
# new final layer with classes
num_ftrs = resnet.fc.in_features
resnet.fc = torch.nn.Linear(num_ftrs, NUM_BREEDS)
```

The training function, `train_model`, is the heart of the learning process for our model. For each epoch we first conduct a training phase which calculates the loss, performs back propagation, and finally updates the weights of the network using `optimizer.step()`. Next, we conduct a validation phase where we only

evaluate the loss and do not update the network. The full code can be found in the Appendix.

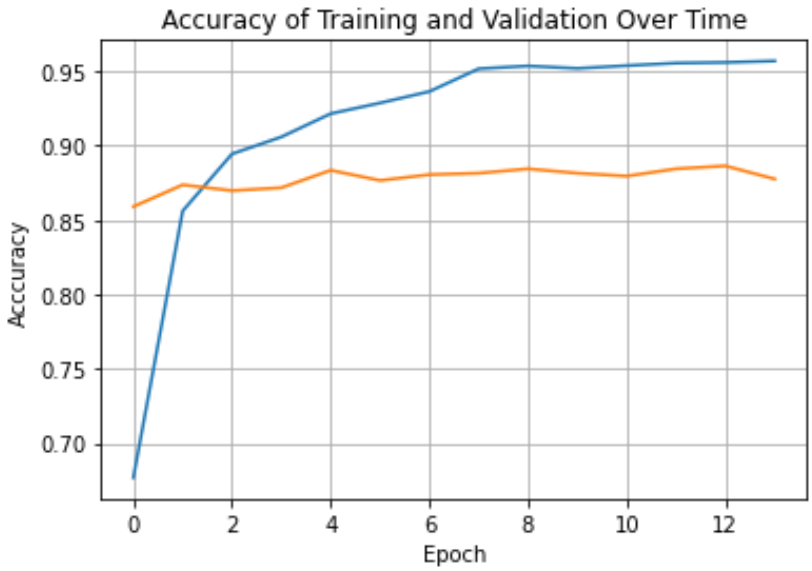
Once we had trained our first models, we began to vary the hyper parameters of the network and record how well the model performed. Finally, then graphed our model accuracy over multiple training epochs.

4.5 Findings

We found that using ResNet as our base model we were able to achieve a max accuracy of 88.6%. The following table shows the best accuracy for various configuration of hyperparameter values.

Batch Size	Workers	LR	Momentum	Num Epoch	Best Acc
16	2	0.01	0.9	14	0.886497
16	4	0.001	0.9	14	0.877202
16	4	0.1	0.8	14	0.852250
16	2	0.04	0.9	14	0.852250
16	2	0.05	0.9	14	0.878669
8	2	0.01	0.9	14	0.883562

From this table we can see that we achieved the best results when the hyperparameters of the network were equal to the first row. We tracked the accuracy of the model during training and validation phases which can be seen below.



From this chart we see that after two epochs the model starts to suffer from over fitting, where the accuracy of the model on the training dataset surpasses the accuracy of the model on the validation dataset. This is likely due to the large number of weights contained within the ResNet architecture. We saw similar results for the other configuration of hyper parameters shown in the table.

Though an accuracy of 88.7% could be considered an acceptable result, we believe more work is needed to improve the accuracy of the model overall. We believe that the ResNet model may have actually caused us to have relatively poor results due to the images in the dataset containing people and other objects which may have caused ResNet to classify that picture as a person rather than a dog. Because of this concern we also created a small convolutional network so we could compare the two.

4.6 Conclusion

Transfer Learning has become more popular in the field of Machine Learning as networks with high levels of accuracy have become more popular. Used as a tool, transfer learning allows you to build on top of a preexisting model, further specializing the model for the task you aim to solve. We will utilize this technique in our project to build a model which will successfully classify dog breeds. We utilize a model called ResNet in order to solve our classification task. We found that using ResNet as our base model we were able to achieve an accuracy of 88%. More work is needed to tune the hyperparameters of our network to see if a better accuracy can be realized. Also, more work is needed to compare this method to a more traditional CNN.

Acknowledgement

I would like to thank Dr. Geng for his time and for advising me on this journey to learn about deep learning. I'd also like to thank Dr. Johnstone for allowing me to take this course from which I have benefitted greatly. I'd like to thank the University of Alabama at Birmingham. I am eternally grateful for the people I have met, mentors I have found, and the friendships I have made during my time at UAB. The time I have spent in study with friends has lit ? ? passion within to pursue more, learn more, and strive to become better each day.

Lastly, I would like to thank my family, in particular my dad, Mike Wise. I would not be here without you. I owe all I have to the way you have loved and supported me.

References:

[1] A. Watters, “30+ artificial intelligence statistics; facts for 2022,” *COMPTIA*, 24-Feb-2022. [Online]. Available: https://connect.comptia.org/blog/artificial-intelligence-statistics-facts. [Accessed: 7-Nov-2022].

[2] P. Olson, “Nearly half of all 'ai startups' are cashing in on hype,” *Forbes*, 05-Mar-2019. [Online]. Available: https://www.forbes.com/sites/parmyolson/2019/03/04/nearly-half-of-all-ai-startups-are-cashing-in-on-hype/?sh=6ce3b7f4d022. [Accessed: 7-Nov-2022].

[3] J. M. Helm, A. M. Swiergosz, H. S. Haeberle, J. M. Karnuta, J. L. Schaffer, V. E. Krebs, A. I. Spitzer, and P. N. Ramkumar, “Machine Learning and Artificial Intelligence: Definitions, applications, and future directions,” *Current Reviews in Musculoskeletal Medicine*, vol. 13, no. 1, pp. 69–76, 2020.

[4] F. ois Chollet, *Deep Learning with Python, Second Edition*, 2nd ed. Greenwich, CT: Manning Publications, 2022.

[5] R. Anyoha, “The History of Artificial Intelligence,” *Science in the News*, 23-Apr-2020. [Online]. Available: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/. [Accessed: 10-Nov-2022].

[6] G. E. Hinton, “How neural networks learn from experience,” *Scientific American*, vol. 267, no. 3, pp. 144–151, 1992.

[7] J. DeLua, “Supervised vs. unsupervised learning: What's the difference?,” *IBM*. [Online]. Available: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning. [Accessed: 11-Nov-2022].

[8] Jenn, “Continuity and differentiability,” *Calcworkshop*, 22-Feb-2021. [Online]. Available: https://calcworkshop.com/derivatives/continuity-and-differentiability/. [Accessed: 23-Nov-2022].

[9] IBM Cloud Education, “What is overfitting?,” *IBM*, 21-Mar-2021. [Online]. Available: https://www.ibm.com/cloud/learn/overfitting. [Accessed: 23-Nov-2022].

[10] A. Gupta, “A comprehensive guide on Deep learning optimizers,” *Analytics Vidhya*, 24-May-2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/. [Accessed: 23-Nov-2022].

[11] V. Jain, “Everything you need to know about ‘activation functions’ in Deep learning models,” *Medium*, 30-Dec-2019. [Online]. Available: https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253. [Accessed: 23-Nov-2022].

[12] C.-F. Wang, “The vanishing gradient problem,” *Medium*, 08-Jan-2019. [Online]. Available: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484. [Accessed: 27-Nov-2022].

[13] I. Goodfellow, “Optimization for Training Deep Models,” in *Deep Learning*, Cambridge, MA: MIT Press Ltd, 2017, pp. 301–302.

[14] J. Brownlee, “Weight initialization for deep learning neural networks,” *MachineLearningMastery.com*, 07-Feb-2021. [Online]. Available: https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/. [Accessed: 28-Nov-2022].

[15] J. Brownlee, “A gentle introduction to batch normalization for Deep Neural Networks,” *MachineLearningMastery.com*, 03-Dec-2019. [Online]. Available: https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/. [Accessed: 28-Nov-2022].

[16] S. Madhavan and T. Jones, “Deep learning architectures,” *IBM developer*, 08-Sep-2017. [Online]. Available: https://developer.ibm.com/articles/cc-machine-learning-deep-learning-architectures/. [Accessed: 28-Nov-2022].

[17] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, “Review of Deep Learning: Concepts, CNN Architectures, challenges, applications, Future Directions,” *Journal of Big Data*, vol. 8, no. 1, 2021.

[18] G. Developers, “Background: What is a generative model? ,” *Google*. [Online]. Available: https://developers.google.com/machine-learning/gan/generative. [Accessed: 28-Nov-2022].

[19] G. Developers, “Overview of GAN Structure,” *Google*. [Online]. Available: https://developers.google.com/machine-learning/gan/gan_structure. [Accessed: 28-Nov-2022].

[20] J. Brownlee, “A gentle introduction to transfer learning for Deep learning,” *MachineLearningMastery.com*, 16-Sep-2019. [Online]. Available: https://machinelearningmastery.com/transfer-learning-for-deep-learning/. [Accessed: 28-Nov-2022].

[21] G. Boesch, “Deep residual networks (ResNet, RESNET50) - 2022 guide,” viso.ai, 22-Aug-2022. [Online]. Available: https://viso.ai/deep-learning/resnet-residual-neural-network/. [Accessed: 28-Nov-2022].

Appendix

Dog Breed Classification

A Study of Transfer Learning using ResNet

Author: John Wise
Semester: Fall 2022

Transfer Learning has become more popular in the field of Machine Learning as networks with high levels of accuracy have become more popular. Used as a tool, transfer learning allows you to build on top of a preexisting model, further specializing the model for the task you aim to solve. We will utilize this in our project to build a model which will successfully classify dog breeds. We utilize a model called ResNet, which was developed by Google, in order to solve our classification task.

In this project we will use a popular open source library called PyTorch to perform our machine learning tasks. We also use Numpy and Pandas in order to training and testing data.

References are given at the end.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Imports And Setup

Lets start by importing the libraries we will need in this project, as well as define some variables which will come in handy later in our code.

```
# Import PyTorch Dependencies
import torch
import torch.nn as nn
from torch.nn import functional as F
from torchvision import datasets, models, transforms
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable


# Import Pandas and Numpy Dependencies
import pandas as pd
```

```
# Import plotting Dependencies
from PIL import Image
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid


import os
import time


# Define some helpful constants
TRAIN_DIR = '/content/drive/MyDrive/TransferLearning/train'
TEST_DIR = '/content/drive/MyDrive/TransferLearning/test'
LABELS_PATH = '/content/drive/MyDrive/TransferLearning/labels.csv'
SAMPLE_PATH = '/content/drive/MyDrive/TransferLearning/sample_submission.csv'
INPUT_SIZE = 224

# Load Datasets
labels = pd.read_csv(LABELS_PATH)
breed_classes = pd.read_csv('/content/drive/MyDrive/TransferLearning/sample_submiss
NUM_BREEDS = len(breed_classes)
```

Prepare Data and Datasets

The code in this section was inspired by the *Use pretrained PyTorch models* tutorial. [2]

```
breed_list = breed_classes.tolist()
labels['target'] = 1
labels['rank'] = labels.groupby('breed').rank()['id']
labels_pivot = labels.pivot('id', 'breed', 'target').reset_index().fillna(0)

print(type(labels_pivot))
train = labels_pivot.sample(frac=0.9)
valid = labels_pivot[~labels_pivot['id'].isin(train['id'])]
print(train.shape, valid.shape)
```

```
<class 'pandas.core.frame.DataFrame'>
(9200, 121) (1022, 121)
```

Now we will want to create a class for our datasets which will allow us to perform useful operations on the datasets and the items in the datasets such as getting an item. This class will inherit from Dataset which is the heart of PyTorch data loading utility.

```
class DogDataset(Dataset):
```

```
    def __init__(self, labels, root_dir, subset=False, transform=None):
        self.labels = labels
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        img_name = '{}.jpg'.format(self.labels.iloc[idx, 0])
        fullname = os.path.join(self.root_dir, img_name)
        image = Image.open(fullname)
        labels = self.labels.iloc[idx, 1:].to_numpy().astype('float')
        labels = np.argmax(labels)
        if self.transform:
            image = self.transform(image)
        return [image, labels]
```

Now we will define a few transformations which will allow us to transform the data into the format needed for our model built on ResNet-50.

```
normalize = transforms.Normalize(
   mean=[0.485, 0.456, 0.406],
   std=[0.229, 0.224, 0.225]
)
ds_trans = transforms.Compose([transforms.Resize(224),
                               transforms.CenterCrop(224),
                               transforms.ToTensor(),
                               normalize])


train_ds = DogDataset(train, TRAIN_DIR, transform=ds_trans)

train_dl = DataLoader(train_ds, batch_size=16, shuffle=True, num_workers=2)

valid_ds = DogDataset(valid, TRAIN_DIR, transform=ds_trans)

valid_dl = DataLoader(valid_ds, batch_size=16, shuffle=True, num_workers=2)
```

Let's take a look at our data and see what we have!

```
def imshow(axis, inp):
    """Denormalize and show"""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
```

```
    axis.imshow((inp * 255).astype(np.uint8))

img, label = next(iter(train_dl))
print(img.size(), label.size())
fig = plt.figure(1, figsize=(16, 16))
grid = ImageGrid(fig, 111, nrows_ncols=(1, 16), axes_pad=0.05)
for i in range(img.size()[0]):
    ax = grid[i]
    imshow(ax, img[i])
```

```
torch.Size([16, 3, 224, 224]) torch.Size([16])
```

Create the Model

Here we will import the ResNet model.

```
use_gpu = torch.cuda.is_available()
resnet = models.resnet50(weights='ResNet50_Weights.DEFAULT')
if use_gpu:
    resnet = resnet.cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet50-11ad3fa6.pth" to /
100%  97.8M/97.8M [00:01<00:00, 66.2MB/s]
```

Train the Model

Lets define a function to perform our training. This function has been adapted from the *Use pretrained PyTorch models* article in [2].

```
train_results= []
validation_results= []
def train_model(dataloders, model, criterion, optimizer, num_epochs=25):
    scheduler = lr_scheduler.StepLR(optimizer, step_size=15, gamma=0.1)
    since = time.time()
    use_gpu = torch.cuda.is_available()
    best_model_wts = model.state_dict()
    best_acc = 0.0
    dataset_sizes = {'train': len(dataloders['train'].dataset),
                     'valid': len(dataloders['valid'].dataset)}
    loop = 0
    for epoch in range(num_epochs):
```

```
    print('Best val Acc: {:4f}'.format(best_acc))

    model.load_state_dict(best_model_wts)
    return model

print("Starting")
# freeze all model parameters
for param in resnet.parameters():
    param.requires_grad = False

# new final layer with classes
num_ftrs = resnet.fc.in_features
resnet.fc = torch.nn.Linear(num_ftrs, NUM_BREEDS)
if use_gpu:
    print("Using GPU")
    resnet = resnet.cuda()

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(resnet.fc.parameters(), lr=0.01, momentum=0.9)


dloaders = {'train':train_dl, 'valid':valid_dl}


start_time = time.time()
model = train_model(dloaders, resnet, criterion, optimizer, num_epochs=14)
print('Training time: {:10f} minutes'.format((time.time()-start_time)/60))
```

```
Starting
Using GPU
Phase:  train
Phase:  valid
Epoch [0/13] train loss: 0.1079 acc: 0.6758 valid loss: 0.0381 acc: 0.8620
Phase:  train
Phase:  valid
Epoch [1/13] train loss: 0.0368 acc: 0.8585 valid loss: 0.0295 acc: 0.8816
Phase:  train
Phase:  valid
Epoch [2/13] train loss: 0.0285 acc: 0.8834 valid loss: 0.0274 acc: 0.8650
Phase:  train
Phase:  valid
Epoch [3/13] train loss: 0.0231 acc: 0.9076 valid loss: 0.0263 acc: 0.8748
Phase:  train
Phase:  valid
Epoch [4/13] train loss: 0.0198 acc: 0.9225 valid loss: 0.0261 acc: 0.8650
Phase:  train
Phase:  valid
Epoch [5/13] train loss: 0.0177 acc: 0.9322 valid loss: 0.0265 acc: 0.8611
Phase:  train
Phase:  valid
Epoch [6/13] train loss: 0.0157 acc: 0.9386 valid loss: 0.0244 acc: 0.8738
Phase:  train
```

```
    print('Best val Acc: {:4f}'.format(best_acc))

    model.load_state_dict(best_model_wts)
    return model

print("Starting")
# freeze all model parameters
for param in resnet.parameters():
    param.requires_grad = False

# new final layer with classes
num_ftrs = resnet.fc.in_features
resnet.fc = torch.nn.Linear(num_ftrs, NUM_BREEDS)
if use_gpu:
    print("Using GPU")
    resnet = resnet.cuda()

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(resnet.fc.parameters(), lr=0.01, momentum=0.9)


dloaders = {'train':train_dl, 'valid':valid_dl}


start_time = time.time()
model = train_model(dloaders, resnet, criterion, optimizer, num_epochs=14)
print('Training time: {:10f} minutes'.format((time.time()-start_time)/60))
```

```
Starting
Using GPU
Phase:  train
Phase:  valid
Epoch [0/13] train loss: 0.1079 acc: 0.6758 valid loss: 0.0381 acc: 0.8620
Phase:  train
Phase:  valid
Epoch [1/13] train loss: 0.0368 acc: 0.8585 valid loss: 0.0295 acc: 0.8816
Phase:  train
Phase:  valid
Epoch [2/13] train loss: 0.0285 acc: 0.8834 valid loss: 0.0274 acc: 0.8650
Phase:  train
Phase:  valid
Epoch [3/13] train loss: 0.0231 acc: 0.9076 valid loss: 0.0263 acc: 0.8748
Phase:  train
Phase:  valid
Epoch [4/13] train loss: 0.0198 acc: 0.9225 valid loss: 0.0261 acc: 0.8650
Phase:  train
Phase:  valid
Epoch [5/13] train loss: 0.0177 acc: 0.9322 valid loss: 0.0265 acc: 0.8611
Phase:  train
Phase:  valid
Epoch [6/13] train loss: 0.0157 acc: 0.9386 valid loss: 0.0244 acc: 0.8738
Phase:  train
```

```
Phase:  valid
Epoch [7/13] train loss: 0.0144 acc: 0.9439 valid loss: 0.0253 acc: 0.8669
Phase:  train
Phase:  valid
Epoch [8/13] train loss: 0.0135 acc: 0.9514 valid loss: 0.0248 acc: 0.8699
Phase:  train
Phase:  valid
Epoch [9/13] train loss: 0.0125 acc: 0.9530 valid loss: 0.0252 acc: 0.8659
Phase:  train
Phase:  valid
Epoch [10/13] train loss: 0.0112 acc: 0.9620 valid loss: 0.0251 acc: 0.8738
Phase:  train
Phase:  valid
Epoch [11/13] train loss: 0.0107 acc: 0.9628 valid loss: 0.0256 acc: 0.8669
Phase:  train
Phase:  valid
Epoch [12/13] train loss: 0.0100 acc: 0.9666 valid loss: 0.0241 acc: 0.8767
Phase:  train
Phase:  valid
Epoch [13/13] train loss: 0.0099 acc: 0.9625 valid loss: 0.0266 acc: 0.8689
Best val Acc: 0.881605
Training time:  17.022427 minutes
```

Tuning Hyperparameters

Batch Size	Workers	LR	Momentum	Num Epoch	Best Acc
16	2	0.01	0.9	14	0.886497
16	4	0.001	0.9	14	0.877202
16	4	0.1	0.8	14	0.852250
16	2	0.04	0.9	14	0.852250
16	2	0.05	0.9	14	0.878669
8	2	0.01	0.9	14	0.883562

```
t_results, v_results = [], []

for result in train_results:
  t_results.append(result.item())

for result in validation_results:
  v_results.append(result.item())

plt.xlabel('Epoch')
plt.ylabel('Acccuracy')
plt.title('Accuracy of Training and Validation Over Time')
plt.plot(t_results)
plt.plot(v_results)
plt.grid(True)
plt.show
```

References

[1] J. Brownlee, "What is the difference between test and validation datasets?," Machine Learning Mastery, 14-Aug-2020. [Online]. Available: https://machinelearningmastery.com/difference-test-validation-datasets/. [Accessed: 08-Oct-2022].

[2] P. V. Lima, "Use pretrained pytorch models," Kaggle, 12-Oct-2017. [Online]. Available: https://www.kaggle.com/code/pvlima/use-pretrained-pytorch-models/notebook. [Accessed: 08-Oct-2022].

[3] B. Neo, "Building an image classification model with pytorch from scratch," Medium, 07-Feb-2022. [Online]. Available: https://medium.com/bitgrit-data-science-publication/building-an-image-classification-model-with-pytorch-from-scratch-f10452073212. [Accessed: 29-Nov-2022].