Austin Peay State University

Senior Seminar

CSCI-4800

Exploring Deep Learning

Author:

Michael Timbes

Professor:

Dr. Jiang Li

Contents

Abstract

Deep learning is an important sub-field of Artificial Intelligence. This paper will explore the goal of deep learning, which is to take a set of data ranging from single data points to multidimensional data sets such as images and have a general model for classification or to help intelligent agents in AI make educated decisions based on complex data. This paper will cover details on the learning aspect which is done through mathematical manipulation of coefficients of a logistically based model; as well as insight into the stages of mathematical manipulation abstracted in what are referred to as*layers* that are then connected to form networks. There will also be discussion on the pre-processing strategies often used to avoid common issues such as over-fitting and numerical issues such as under/overflow. Finally, an example application will be discussed to illustrate the practical uses of deep learning.

Introduction

In the broad field of AI, machine learning is an important part of some AI system architectures. From search engines to face recognition applications, AI and machine learning are used in critical systems of everyday life. As the application of AI grows, so too does the importance of understating what is going on behind the scenes. The focus in this paper will be a small subsection of machine learning, deep learning, which has proven to be incredibly powerful and is normally the standard learning method in many large and complex use cases such as self-driving vehicles.

The history of deep learning can be traced back to Allen Turning when he proposed a hypothetical test that determined if a machine can think on its own known as the *Turing Test*[9]. This test consists of two agents (Agent A and B) and a human participant. The goal of the test is to determine if the participant can point out which agent is a human and which is a machine by asking each agent complex and self-reflective questions. There are some philosophical issues with this test, for example, some people don't believe that the test goes far enough; and this argument is supported by example in modern day AI-powered agents which can (to a level) pass this test. However, the proposition that machines could potentially *think* on their own sparked a movement to investigate the question.

Soon after Allen Turing, computer scientists began writing programs that could seemingly *learn* from their mistakes. Additionally, it seemed that computer programs could simulate implicit (as opposed to explicit if/then clauses) logic based on environment variables. Algorithms such as the *K-Nearest Neighbor* and Numerical Logistic/Linear Regression Analysis paved the way for computers to learn and make predictions based on data. However, developments in learning algorithms didn't really pick up until the 1990's with IBM's *Deep Blue* project which beat a world champion at chess.

Soon after, companies realized that with the large amount of data they had and the growing popularity of the internet, machine learning could really give them an advantage. Companies like Google, IBM, Facebook and others began to research how to further improve machine learning models/methods and how to incorporate that into their AI-powered applications.

What is Deep Learning

Deep learning refers to a set of learning methods that use interconnected *layers* of probability weights whose output is a set of probabilities. Essentially, it is modeled on how the brain learns. For example, when a person learns a new word, that person's brain performs a set of actions to ensure the information can be used in the future for reference and to make inferences.

The way the model stores the information is not in explicit definitions but rather in a weight matrix. The matrix represents a set of weights that are applied to the input data and then sent through a function called a *sigmoid function* or depending on the application, a *logit function*; both functions will produce a number or set of numbers that represent where the input data lies bounded by the vertical axis by $[0, 1]$ in the case of the sigmoid or from $[-1, 1]$ in the case of the logit. The outputs of the functions serve as Boolean true/false values that help to classify the input.

The training process takes the output of the functions and compares it to what the output should be, say 0 or 1 if using the sigmoid function. Just as if you were to balance a scale to find the weight of something, the comparison function calculates the amount the weight matrix coefficients need to change in order to be closer to the actual output and updates the matrix. The next time the same or *similar* input is given the sigmoid or logit functions should produce a value that is exactly or very close to what the actual value should be. This is the simple explanation of what is going on when an algorithm *learns* something.

Deep learning is a type of neural network architecture that takes this learning approach and scales it up to multiple output values and then uses those output values as inputs to another layer (or layers). The process is then repeated through many more layers until a final output layer returns a set of values that describe the original input data. The next few sections of the paper provides more detail on the Logistic functions (sigmoid/logit Functions), the training process, and the role that data structure and reprocessing plays.

Logistic Model

Logistic regression is used for classification based on learned statistical weights and bound by the sigmoid function which acts as more of a logical function (Boolean type logic).These weights in conjunction with the function produce what is considered a decision boundary which serves to separate the data into different classifications. It was first used for binary classification of different features to help classify an event based on that values of the features[3].

To understand logistic regression, consider the issue of attempting to classify an item into one of two categories (essentially binary classification problem). There should be a function that produce a well defined boundary that is either category-A or B. A logistic function does

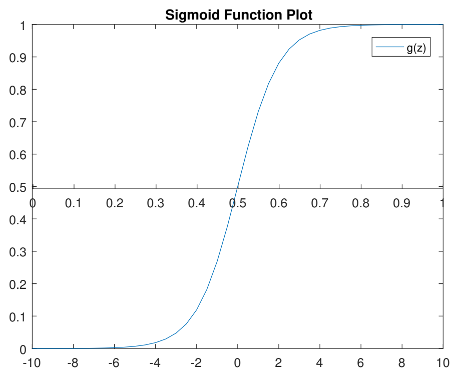

Figure 1: sigmoid function Plot

just that, it has a well defined line that passes right through a specific point which can serve as the defining point at which data point goes from one category to another. The basis for using this function is that the limit of an exponential function at infinity approaches some constant and there is always a single continuous curve that has an upper and lower bound.

The weight matrix (illustrated in the following equation as θ_n values) times the input data is the input of the Sigmoid. Matrix multiplication rules apply so for N data points there must be a corresponding N number of weights coefficients.

$$h_\theta(x) = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \begin{pmatrix} \theta_0 & \theta_1 & \dots & \theta_n \end{pmatrix}$$

The $h_\theta(x)$ value is considered the hypothesis function. It is used to represent the output values that are passed to the sigmoid function. The hypothesis can be thought of as a current *best guess* based on the coefficient values of the weight matrix.

In order to fine tune the values of the weight matrix, there is a step in the training process that compares the best guess provided passing the hypothesis function into the sigmoid function and calculating the *cost* or how far off the hypothesis is from the actual value. This calculation is in the form of a cost function.

What is a Cost Function

When determining how far off a hypothesis is from the actual value expected, a cost function is used. What this function does is calculates the numerical difference between an output number and the value expected. In the context of a deep neural network, the output layer produces a value and that value is checked against a training example. The *check* is how far off the output is from the training value. This is calculated using a simple formula similar to how one may calculate the distance between two points using a square difference as shown below.

$$Cost = \sum_{i=0}^{n} (hypothesis(x_i) - actual(y_i))^2 \tag{1}$$

Where *hypothesis* is the output value from the model and *actual* is the expected known value of the training example. This cost is a sum because this is repeated for all the training input. By calculating how far off the output is, the next step is to then figure out how to adjust the model to have more accurate results. The next step handles the job of finding the best value to add to the weights to produce more accurate results.

In the context of logistic regression, the cost function looks more complex but achieves the same. In this case, the goal is to adjust the cost based on a binary decision line, namely when $y = 0$ or $y = 1$. This means that the function must reflective a high cost if the hypothesis is zero when it should be one and vice versa. This is accomplished with two $\log(x)$ functions, each reflects both cases.

$$J(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x)) \tag{2}$$

Cost Optimization

In the previous section explains how to find how far off a model's output is from the true value which is relatively straight forward with a sum of squared differences. The step after that finds how to adjust the model to be more accurate is more complicated.

There is a class of optimization problems (convex optimization problem) that aim to find the smallest number possible of a function in a given interval, also known as the local minimum of a function. This is useful in the cost reduction step to find the best offset so that the cost function from earlier approaches zero. One algorithm that is popular in machine learning is *Gradient Descent*.

Algorithm 1 Gradient Descent Implementation

```
1: procedure GradientDescent(cost_value)
2:     $step\_size = 0.1$
3:     $prev, current =$ cost_value
4:     $temp \leftarrow 0$
5:     while prev > some small number
6:       $temp \leftarrow current$
7:       $current \leftarrow$ current + $step\_size *$ Cost_Function($temp$)
8:       $prev \leftarrow |current - temp|$
```

What makes this optimization problem tricky is when there is more than one variable, one has to account for each variable independently. Let's first consider the easier case of the convex optimization problem, when there is only a single variable which is essentially a line; gradient descent *steps* through the range of the variable, starting from some point and ending at a very small number. Once a step starts to produce a non-decreasing number, it is determined that the previous step is the local minimum.

In the case of a multivariate optimization problem, each variable is treated as part of a larger system. If we were to take each variable and treat them as individual lines, we could apply the same approach mentioned before with just one variable and expand it to all the variables. Once the local minimum is found for every variable, each is updated with their own respective offset to reduce the cost with respect to that variable. The effect is by adding the cost reductions, the overall cost of the model is also minimized.

Another lurking problem with minimizing the cost and even in calculating cost is numerical over-flow and under-flow. Since the cost function and gradient descent are numerical routines that deal with floating-point math, one has to be aware of the issues that arise with data points that are so large that they cause over-flow or they could be incredibly small

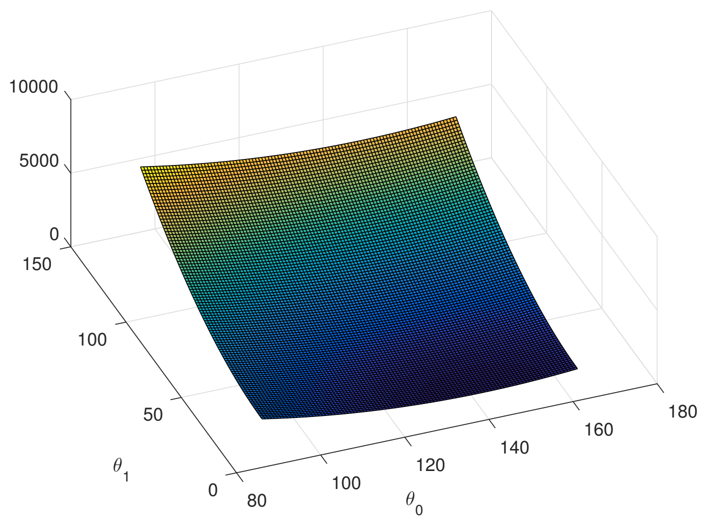

Figure 2: Example Gradient Descent Converging to Zero with 2 Weights

so that they cause under-flow. An additional problem arises when *fitting* data to a model called *over-fitting* which causes the model to only produce accurate results when given the training data and fails when new data is introduced. The methodology behind preventing these issues is normalizing training data.

Data Normalization is Important

Depending on the type of data, there are varying methods to normalizing data. The idea behind normalization stems from normalizing vectors in the context of Linear Algebra. A vector is considered normalized if the length of the vector is equal to one. The reason it is helpful is that a normalized data-set gathers the points around a common center and they aren't as spread out. Normalizing reduces the possibility of over-fitting by effectively breaking the structure of the data and standardizing the data-set which is ideal since the goal of training a model is to predict outcomes from data other than the training set. It also reduces the possibility of numerical over/under-flow by reducing the deviation of the values to where they are closer together.

There are different ways to normalize data and the methods really depend on the type of data. In the case of data points, a statistical approach can be used by adjusting the standard-deviation so that the mean is zero and the standard deviation is one. This is also known as the *Z-Score* normalization and is used to create a unit-less data-set helpful to avoid numerical over-flow.

Another example is building a regularization factor into the model. This is an additional piece to the hypothesis function that will scale the effect of the weight matrix. If one doesn't build the regularization into the hypothesis, then they would have to manipulate the matrices that the input features are multiplied by to be able to have accurate output values when testing new data.

$$J(h_\theta(x^i), y^i) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Cost Function with Regularization Term (from Andrew Ng's Course on ML)

A final example, is in the case of images. The first question to consider is does the image depend on a number of channels (i.e. RGB, LAB, etc.) one of the most obvious ways to reduce the number of input is reducing the channels. Other than reducing the channels, normalization is also the same by normalizing the pixel values by subtracting the mean and dividing by the standard deviation but normally this isn't done, instead the picture is saved as a black and white or gray scale image.

Putting it Together- Shallow Network Example

To illustrate the concepts above, there is enough information to build a simple *single-layer network* also known as a shallow network. Let's build the network around the requirements for a binary classifier that attempts to determine if a person is at risk for a heart attack based on a person's age. The classification labels are true (1) or false(0) which mean, at risk or not at risk respectively.

Form the Hypothesis

The first step is to define the structure of the overall model and gain a better understating of what the hypothesis function contains. From the information above we know that this problem is dependent on a single feature- the age. This means the input (or input layer) X is only going to have a length of one, again because we are assuming that there is only one data point that determines the risk. The weight vector W will have two weights to account for the two possibilities (two weights are use for illustration but only needs one); also, there must be a vector to hold what is known as the bias B which helps give some flexibility to the model. Finally, the output vector Y will have two elements- one to hold the probability that the data point is true and another for false; this vector is a result of applying the sigmoid function to vector Z.

$$z_1 z_2 = x\theta_1\theta_2 + (b_1 b_2)$$

$$\downarrow$$

$$Z = X \cdot \Theta + B$$

The next step is to apply the sigmoid function. This forms the hypothesis function $h(z)$. Then, of course, is choosing the appropriate cost function which is defined in the cost function section of this paper.

$$h(\hat{Z}) = Y \rightarrow Sigmoid(y_1\ y_2)$$

Training

The next crucial step before training is to normalize the data-set. This is done by subtracting the mean from each data point and then dividing by the standard deviation. It is important to note that when testing, the test input should also be normalized.

After feeding the data in, the training portion is next. With each training input value (and their respective labels) the cost function is evaluated and goes through the gradient

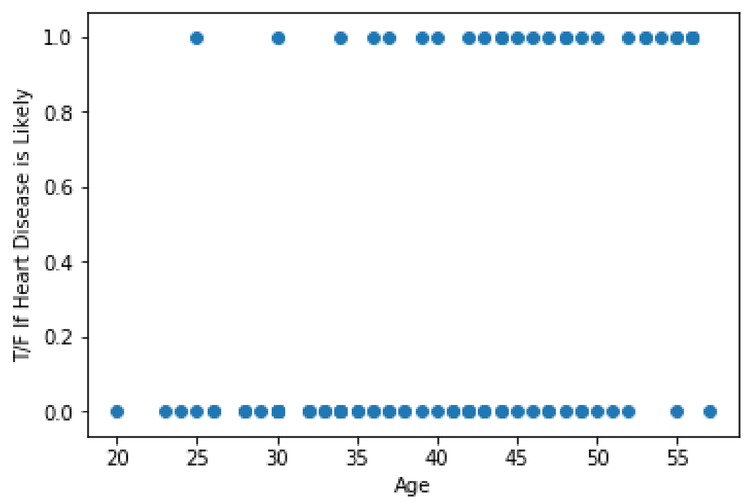

Figure 4: Input Data

descent optimization step where the gradient step alpha is set to 0.1, there is no set rule for what the alpha should be and usually trial and error (comparing results with different alphas) will dictate the best step value. The optimization step sends the correction values to the weights matrix that minimizes the cost. This is repeatedly on the same data points for however long is needed until the accuracy is improved (in this case, repeating 20 times seemed reasonable).

Testing and Results

After the training step, the testing process gives insight to how accurate the model is. Below are the results. Included in the results are the values for the weights and constants.
Accuracy: 73%

$$W = \begin{pmatrix} 10.136 \\ -10.136 \end{pmatrix} \quad B = \begin{pmatrix} -250.95 \\ 250.95 \end{pmatrix}$$

The accuracy is not realistically acceptable and basing if someone is at risk of heart disease on a single feature would be irresponsible but this is a good illustration of the process to train a logistic classifier. The training data was based on the first 80 points of the input data and the test is based on the remaining 20 points. The training data is randomized

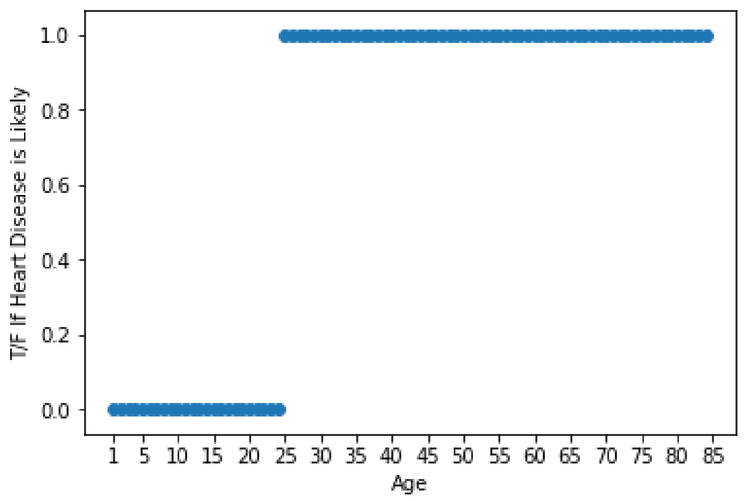

Figure 5: Plotting Results- Looks Similar to Input

in the training process to avoid over-fitting and the test data is completely unique to the training data (the model has never seen the test values before) so for the model to make an educated guess with only 80 data points and get most correct shows just how powerful logistic regression is in classifying.

Layered Systems

It is important to go through the details of logistic regression since it is the foundation of neural networks. In fact, neural networks are a system of logistic regression subsystems stacked on top of one another. Each layer that is not an input or output layer is considered a hidden layer, these hidden layers is where the cost is calculated, minimization occurs, and updates to the weights are made. The hidden layers are connected by their input vectors and depending on the type of network, they share cost minimization information which can be propagated forward and backward. There are three common types of neural networks: feed forward network multi-layer perceptron (MLP) with back propagation, convolutional neural networks (CNN), and recurrent neural networks (RNN).

Multi-layer Perceptron- Feed Forward and Back Propagation

A shallow network is considered to have an input layer, a single hidden layer, and output layer. An MLP can have at a minimum of three layers and of the three, one must be a hidden layer. So, a shallow network is considered a type of MLP but it does not perform well for applications that demand more than one classification. Instead, the layers add for more flexibility in the model to evolve the hypothesis to accommodate multiple classifications.

The flow from input layer to output layer starts with the first application of the sigmoid function which takes the input and multiplies it by the first vector column weights of the weight matrix; the outputs from this operation are considered the *activations* of a hidden layer the sum of these activations is considered the value of the activation node at that layer. Activations could be though of as the interpretation of the input from the point of view of a hidden layer and the weights associated with that layer. A similar process follows with the second layer with the same column vector of the weight matrix which produces its own set of activation values which are summed to be the activation node value of that layer. This process is called feed forward or forward propagation and is standard practice in the training step of MLP networks but is only part of the overall training process.

Just like before with a shallow network, the cost is calculated and then compared to the expected value. Using the activation node values, these outputs are used to calculate the cost of the overall model. Back propagation starts from the layer before the output layer; it calculates the adjustment needed to make that weight value more accurate by using the adjustment values from layer in front of it as a basis, the weight is updated, and the process is repeated from the output layer to the first hidden layer.

Convolutional Neural Networks

A convolutional neural network is a little different from an MLP before although it shares similar roots to the logistic regression learning model explained earlier. In image processing, a filter is a set of values in a square matrix that is convoluted through an image. Convoluted

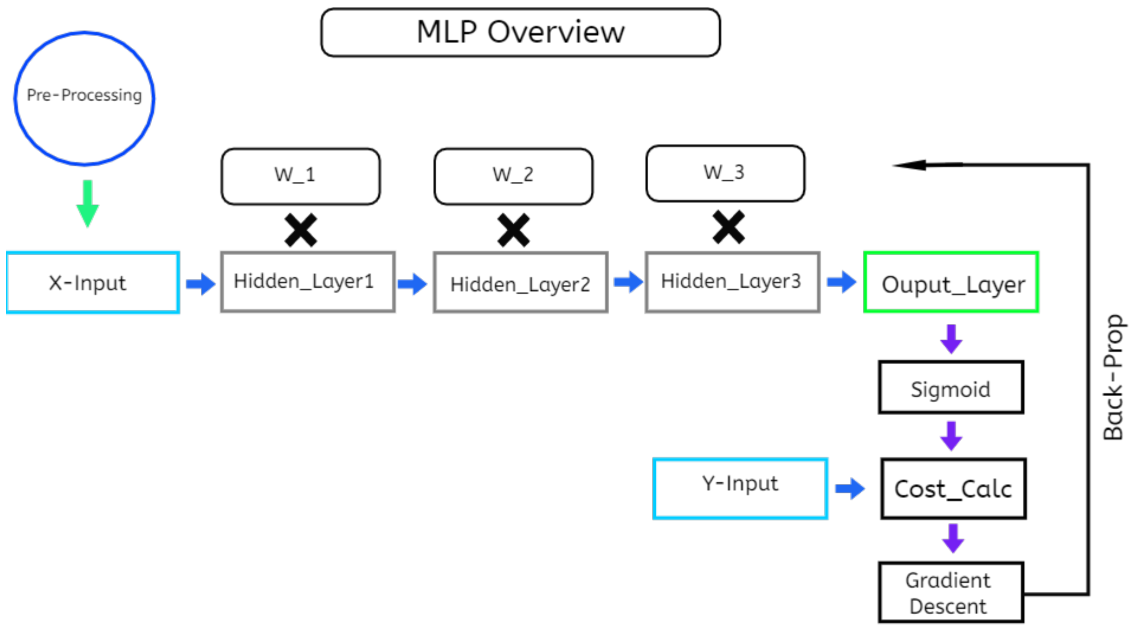

Figure 6: Example MLP Architecture

meaning inner products of the image and the transpose of the filter; this is done in sections equal to the size of the filter from left to right so that the result is the same image but with new values that are updated by the same filter. Convolution is used in filtering noise and amplifying edges in an image. The training data is normally encoded as a binary vector

Convolutional neural networks take this process and apply it to a logistic learning model. The reason for using convolution rather than flattening an image and passing it through an MLP is that scaling the model would mean that for a 32x32 pixel image in RGB color space, there would be 3,072 inputs. This architecture is unsustainable for high definition images and would mean that a lot of information would be lost due to down-scaling.

Another key difference in a CNN is that the main function used to produce a hypothesis is not the Sigmoid but the Relu. This is to help numerical stability by essentially narrowing the outputs to zero or some positive real number. In addition to stability, using the Relu is a way to increase the speed of the training process by helping convergence in the gradient descent algorithm.

Although a CNN is different, the flow of operations is still similar. First, a weight matrix is still used but instead of a single matrix, there is a weight matrix for each color channel of the image. This means that the weight matrix is three dimensional- Columns x Rows x Color Channels. The weight matrices are seen as filters for each channel and are convoluted over a 2-D slice of an image at some specified stride which can be thought of as a step size (i.e. a stride of 1 means that the filter moves in 1 pixel increments). Sometimes, to make dimensions work between layers a zero padding is given which essentially makes the filter size fit the input volume without adding anything to the input values. The convolution produces a 3-D set of values that are used as inputs to the Relu function known as an output volume.

A type of layer that is exactly similar to an MLP layer is the fully connected layer. It is the length of an input volume that is flattened and is multiplied by a column vector of weights (just like an MLP). This is used when an object is known to normally be in one localized area of an image. There is another type of layer called a locally connected layer which is essentially a fully connected layer in the form of a convolutional layer and is used for objects that could be located anywhere within the image. Depending on the design and goals of the implementation, either one of these layers could be used as an output or final layer.

The Relu function then produces the activation values similar to the MLP network, the difference is that in a multi-layer CNN the activations are used as inputs to the next layer. The layer that receives the activation values from the previous layer has a set of weight filters that are convoluted through the activation values and values are produced for that layer. Passing through each layer affects the number of output values depending on the layer's hyper parameters and the input volume dimensions. For example, an image that is 20 x 20 x 3 (total of 1,200 values) and a convolutional layer that is of size 4 x 4 x 8 with a stride of 1 and no padding, will produce an output of 8 x 8 x 8 (total of 512 values). Where as a convolutional layer that is 6 x 6 x 32 with stride 1 and zero padding of 1, will produce an output of 8 x 8 x 32 (2,048 values).

Input Hyper Parameters:

W_1 = Input Volume Width

H_1 = Input Volume Height

D_1 = Number of Channels

K = Number of Filters

F = Filter Size

S = Stride

P = Zero-Padding

Output Hyper Parameters:

W_2 = Output Volume Width

H_2 = Output Volume Height

D_2 = Number of Output Channels

$$H_2 = W_2 = \frac{W_1 - F + 2P}{S + 1}$$

$$D_2 = K$$

Hyper Parameters for a Convolutional Layer. Equation to find the output volume from one layer to another[1]

Pooling and decreasing volume after about two layers of convolution is recommended but there is no strict rule with when a pooling layer is implemented. The pooling method, *max pooling*, takes an N x N area of a matrix and keeps the maximum of the subset discarding the other values and *average pooling* which is similar but calculates and uses an average of the subset. Some still use pooling layers as a means to decrease overall volume (number of activation values) but it's been shown that intermittently between layers, using a larger stride achieves the same effect without dropping as much data that could be valuable.

Another alternative to max pooling is *dropout* which is a method to reducing volume by randomly dropping some of the activation values at a layer. This is a very powerful method by still potentially causes important values to be lost. In many architectures, a combination of methods to reduce volume are used or a dynamic programming algorithm may be used to figure out what activations are actually important to the accuracy.

The final layer of a CNN is a dense layer (either fully connected or locally connected). It is a flattened vector of results from the activation values of the convolutional layers before. This is different from the MLP in that the values that were passed along were the sum of all the activation values that made up the activation node values at each layer. In this case the activation values from the layers before are passed as inputs to the next layer and so there is no need to sum the activations.

Usually to speed up training, the network is trained by what is called batch training which is taking a number of images and labels and feeding them all at the same time (they are still trained in a queue). At the dense layer output, the cost is calculated and the gradient descent algorithm is used to find the offset to reduce cost. Back propagation is applied similar to an MLP but the values produced by gradient descent are applied as an transposed convoluted filter through each layer.

A CNN is a powerful network design for multi-dimensional input with multiple classification possibilities. It can be used for classification of images, audio, and other data structures that are expressed in multiple dimensions. A prime example is the bounded box object detection problem where a CNN detects and finds the region where a known object is within an image.

Recurrent Neural Networks

Recurrent neural networks is the last network to be reviewed, much detail will be omitted since this architecture is more advanced but it is important to understand the basics of the inner workings of this type of network. RNNs are incredibly powerful and are often used in natural language processing applications such as translation and automatic word prediction. There are a few similarities between the previously described networks and an RNN; for example, they all use probability weights when calculating the hypothesis values, they use the same method to calculate costs, and also use gradient descent to calculate the corrective offsets for the weights. However, this network is different in many aspects compared to the logistic based models of MLP and CNNs.

One such difference is that there is an internal state that is taken into consideration when creating a new hypothesis. There is still an input vector (or possibly a set of input vectors) that is used to create a hypothesis but it is saved for the next future hypothesis. Another input vector passes through the hypothesis function, the previous hypothesis is included, and this is repeated. Another is in how the output vector is calculated, rather than use a sigmoid or Relu function, an RNN generally uses a hyperbolic tangent function for a non-linear output and to address the numerical instability of gradient descent on a second order derivative (since we have to account for the previous state).

Depending on the flavor of an RNN, there can be a single output vector at the last layer with multiple inputs or there can be many outputs with a single input vector. For networks with multiple input vector steps (i.e. a series of letters/words) and a single output, they are considered a *many to one* RNN. For the latter, where there are multiple outputs given one input vector step, this is considered a *one to many* network which could be used to extract multiple probabilities similar to how a CNN creates activation values for a volume.

In both cases, the previous hypothesis value is used as part of the new output vector. As before, the costs are calculated when training and back propagation is used to update the weight vector which is shared among the various layers.

One example of using an RNN for word generation given a sequence of characters. First the network is trained with a vocabulary of words that are made of characters. Then from the training set, the network can produce somewhat sensible sentences. Future work with this network is to pair it with AI to produce *original* material that makes more sense to humans and that might even use literary devices such as metaphors to form an illusion that the system can produce content without telling it what to do explicitly.

Original Example of a Convolutional Network

To gain a better understanding of how a CNN works, let's suppose we want to build a network that will recognize one of four objects: a laptop, cup, chair, or car. Services, like Google's vision API is trained to detect and recognize millions of objects and usually consists of layers upon layers of convolutional filters. This application will attempt to only recognize four types of objects with a small number of examples.

The first step before implementing the code needed is to consider the data pre-processing step and the model design. Since images are being used, the data normalization process is different compared to just data points. After, there are a few challenges and hardware limitations that need to be discussed. Finally, I'll go over the results and discussion on how to improve the network.

Image Data Processing

The images used are from the Caltech101 image collection. They are not pre-processed so that people can use them in their machine learning applications. Each network starts with the input layer which is a fixed size in a CNN so this means that the images must be a fixed size as well. In addition to the height and width, we must make sure that the images are in the same color space (RGB for this application).

The size of the image must not be too large because the convolutions will produce results that could be larger than the image size and might be too much for the hardware to handle. For this example, it seemed that dimensions of 64x64x3 would be a good choice for the images. Any larger and risk running out of memory, any smaller and the images would be a poor quality and useless in terms of teaching the network the structure of the items we want it to recognize. Each image label will be serialized in four separate categories. This is so we can have an output that corresponds to a specific category.

Sometimes smoothing filters used to help make the images easier to process and cut down on noise. Depending on the application, there are cases where the RGB color space is not needed and one can cut the images down to a gray or black and white. There isn't a strict method and usually it is to the discretion of the people designing the network to decide what kind of data is best.

Convolutional Network Design Overview

To avoid distraction and difficult to read code screen shots I will explain the abstract components of the CNN (if the reader is interested in the code, I've included a link to my GitHub at the end of the paper). Below is a map of the CNN design, it's meant to help illustrate the structure of the network and be a reference throughout the rest of this section. The 'L' number is the index of the type of layer, not in reference to the actual layer number. The abbreviations for the layers are: CONV_L (convolutional layer), MAX_PL (max pooling layer), and FC_L (fully-connected layer).

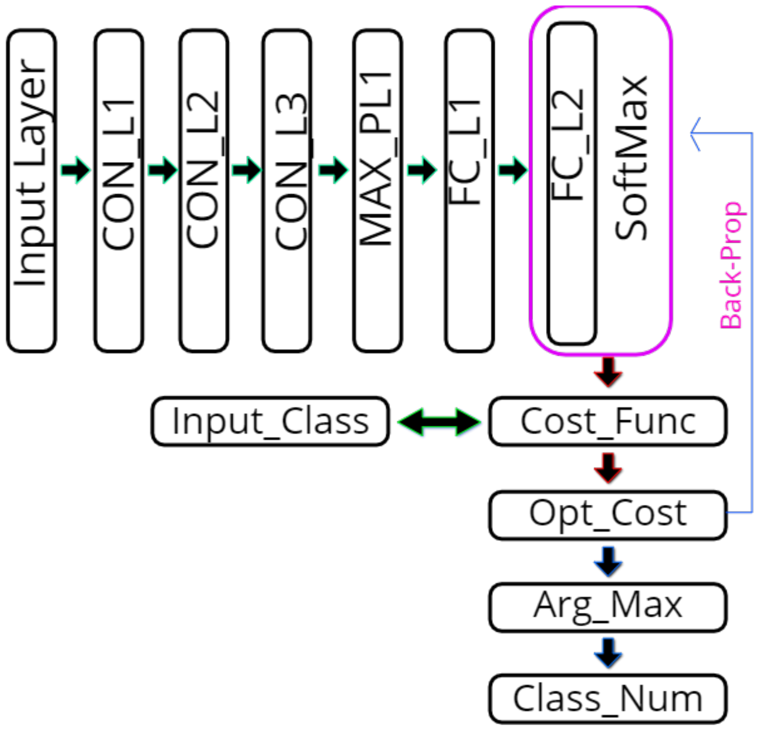

Figure 7: Convolutional Network Design Overview

Layer by Layer Walk-through

The first layer is the input layer, this is a matrix that is equal to the dimensions of the input image, it hands off the input image to the first convolutional layer. In every layer with weights, the weights are initialized to a random small number to ease the gradient descent algorithm's initial calculation and each one uses the Relu function to calculate the activations.

CONV_L1 has dimensions 6x6x64 which is convoluted through the input image pixel values. The output matrices (64 in total) then go to CONV_L2 which applies 3x3x32 filters to the input with zero padding to ensure the dimensions between the filter and the input agree. The final convolutional layer applies 2x2x22 filters and at this point, there are a total of 64x64x22 input features (90,112 inputs) that the filters are applied to.

After the third convolutional layer, the output matrix for the max pooling layer which has a kernel size of 12x12 and stride of 2 is 27x27x22 which is 16,038 outputs. The max pooling layer helps to reduce the number of weights which speeds up computations for the two fully connected layers. The first fully connected layer is structured like an MLP layer except uses the Relu function as opposed to the sigmoid. The final fully connected layer actually doesn't use the Relu function because the outputs will go through another function called *Soft-Max* which is a way to normalize the output and spread the results across the different classifications.

Training and Testing

The training loop consists of an inner and outer loop. The outer loop randomizes the training data to help avoid over fitting. The inner loop controls the batch training process. During training, the cost function is active and compares the results from the final fully-connected layer to the expected classification. The cost is calculated and sent to the optimization step (Opt_Cost) which then drives the back propagation to correct the filters in each layer.

If the model is being tested, the last layer goes from the Soft-Max results to the maximum arguments function (Arg_Max) which simply returns the index of the largest value. Depending on if the person implementing the wants to serialize the data (using one hot encoding), the way to extract the class number may vary but in this case, the returned index represents the class number.

Results of the Convolutional Network and Conclusion

I used Python (version 3.6) and TensorFlow (version 1.1- GPU bound implementation) a machine learning package by Google. Both the language and the package are used by major companies to power their A.I. applications and services. There were 227 training images and 96 testing. All images were size 64x64x3 in RGB color space and were one hot encoded.

After 90 training iterations and the small number of images used to train the model, the accuracy was 57.26% which is acceptable for a small example, but there are certainly ways to improve it. The first obvious way is to add more training data. Usually there are millions of training images used in large scale projects and normally and are usually larger than 64x64 pixels.

One of the other things that can be done to the model is adding more layers. The more layers, the better the model can learn more complex features. Another adjustment could be to add a dropout layer to loosen the model and allow gradient descent to be more accurate. In fact, after adding a dropout layer to the network, the accuracy did increase to 64.52%. At times, a part of designing a network may be adjusting and testing parameters or design to further improve the accuracy and how well the model can generalize (apply to unknown data).

Deep learning is a powerful tool from categorizing data to generating a coherent paragraph from little to no information. The topic has come a long way in just the past ten years and continues to grow. Only a small fraction of the many methods and possibilities are included in this paper but it isn't too difficult to realize why industries across the world are rushing to take advantage of the tools and capabilities that deep learning provides.

References

[1] Karpathy A. Cs231n convolutional neural networks. Lecture notes on Convolutional Neural Networks.

[2] Ng A. Cs229 lecture notes. Lecture notes on Machine Learning/Logistic Regression.

[3] Cox. The regression analysis of binary sequences. *JSTOR*, 20(2983890):215–242, 1958.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Google(ABC). TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Abdi H. Normalizing data. In Neil Salkind, editor, *Encyclopedia of Research Design*. SAGE, 2010.

[7] R. Fergus L. Fei-Fei and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. CalTech was the source of the training/testing images.

[8] S. Yeung L. Fei-Fei, J. Johnson. Cs231n lecture 10: Recurrent neural networks. Lecture notes on Convolutional Neural Networks.

[9] Graham Oppy and David Dowe. The turing test. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2016 edition, 2016.

Github Links:

Logistic Regression Example:

https://github.com/MichaelTimbes/MLTF/tree/master/LOGREG

CNN Example:

https://github.com/MichaelTimbes/MLTF/tree/master/CNN