



**Hyper-parameter Optimization of a
Convolutional Neural Network**

THESIS

Steven Chon, Captain, USAF
AFIT-ENS-MS-19-M-105

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENS-MS-19-M-105

HYPER-PARAMETER OPTIMIZATION OF A CONVOLUTIONAL NEURAL
NETWORK

THESIS

Presented to the Faculty
Department of Operations Research
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science Operations Research

Steven Chon, BS
Captain, USAF

March 2019

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENS-MS-19-M-105

HYPER-PARAMETER OPTIMIZATION OF A CONVOLUTIONAL NEURAL
NETWORK

THESIS

Steven Chon, BS
Captain, USAF

Committee Membership:

Dr. Daniel Steeneck
Chair

Dr. Bihl Trevor
Co-Chair

Lt. Col Jeremy Jordan, PhD
Member

Abstract

In the world of machine learning, neural networks have become a powerful pattern recognition technique that gives a user the ability to interpret high-dimensional data where as conventional methods, such as logistic regression, would fail. There exists many different types of neural networks, each containing its own set of hyper-parameters, that are dependent on the type of analysis required, but the focus of this paper will be on the hyper-parameters of convolutional neural networks. Convolutional neural networks are commonly used for classifications of visual imagery. For example, if you were to build a network for the purpose of predicting a specific animal, it would hopefully output, with high fidelity, the correct classification of a new animal introduced to the model. Traditionally, hyper-parameters were rarely optimized because it required a lot of computational power and time. If hyper-parameters were adjusted, analysts would manually change a few hyper-parameters, re-run the model, and hopefully get a better classification accuracy. However, because of the advancements in technology, hyper-parameter tuning can now be done through complex and powerful optimization algorithms to improve the model. This paper implements and compares three different optimization techniques: random search, Bayesian Optimization with Gaussian Process, and tree of parzen estimator approach. The best performing technique is then improved through the Kiefer-Wolfowitz approximation.

Acknowledgements

I would like to thank my advisor, Dr. Daniel Steeneck, for his support and guidance through my thesis process. He understood the difficulty I was having with my first topic dealing with a non-robust software program called UxAS, then recommended switching my topic to hyper-parameters.

Contents

	Page
Abstract	iv
Acknowledgements	v
List of Figures	ix
List of Tables	xi
I. Introduction	1
1.1 Hyper-parameters	12
1.1.1 Learning Rate	13
1.1.2 Number of Hidden Layers	14
1.1.3 Number of Dense Neurons	15
1.1.4 Batch Size	17
1.2 Convolutional Neural Network	17
1.3 Problem Statement	28
1.4 Research Scope	29
1.5 Research Objectives	30
1.6 Limitations	31
1.7 Summary	31
II. Literature Review	32
2.1 Overview	32
2.2 Random Search	33
2.3 Bayesian Optimization with Gaussian Process (BO-GP)	35
2.4 Tree of Parzen Estimator (TPE)	40
2.5 Kiefer-Wolfowitz	43
2.6 Summary	44
III. Research Methodology	45
3.1 Overview	45
3.2 TensorFlow	45
3.3 Keras	46
3.4 Data Set	50
3.5 Limitations	51
3.6 Applying Hyper-parameter Optimization Techniques	52
3.7 Metrics	52
3.8 Summary	53

	Page
IV. Results and Analysis	55
4.1 Introduction	55
4.2 Base CNN	55
4.3 Random Search	59
4.4 Bayesian Optimization with Gaussian Process (BO-GP)	61
4.5 Tree of Parzen Estimator (TPE)	65
4.6 Kiefer-Wolfowitz	68
4.7 Summary	70
V. Conclusions and Future Research	71
5.1 Overview	71
5.2 Implications	71
5.3 Future Considerations	72
5.4 Final Summary	73
Bibliography	75

Acronyms

Acronym	Definition
ANN	Artificial Network Network
CNN	Convolutional Neural Network
BO-GP	Bayesian Optimization with Gaussian Process
GP	Gaussian Process
TPE	Tree of Parzen Estimators

List of Figures

Figure		Page
1.	Simple Neural Network. From [1].....	3
2.	Sigmoid Function. From [1].....	5
3.	Single Neuron Model. From [2].....	6
4.	Large Neural Network. From [2].....	9
5.	A Large η may cause algorithm to overshoot global minimum. From [1].....	14
6.	Left: A standard network outwith drop-out. Right: A network with drop-out. From [3].	16
7.	Left: Artificial Neural Network. Right: Convolutional Neural Network. From [4].	18
8.	Top: ANN shows that each neuron in one layer is connected to every neuron in the next. Bottom: CNN shows that only a small region of neurons from the input later is connected to a neuron in the hidden layer. From [5].	18
9.	Typical CNN Architecture. From [6].....	19
10.	Left: Example of a input image. Right: Convolutional Filter/Kernel. From [6].	20
11.	Frobenius Product. From [6].....	20
12.	Next iteration of convolution. From [6].....	21
13.	3D Convolutions. From [6].	22
14.	Max Pooling. From [7].....	23
15.	After applying max pooling. From [6].....	24
16.	Flattening. From [8].	25
17.	Dense Layer to Softmax Layer. From [9].	26
18.	Adding more convolutional layers. From [6].....	30

Figure		Page
19.	Grid Search vs Random Search. From [10].	34
20.	Three iterations of Bayesian Optimization with Gaussian Process. From [11].	39
21.	Distribution of Good and Bad Groups. From [12].	42
22.	Architecture of Base CNN via Keras	47
23.	Architecture of Base CNN via TensorBoard	49
24.	Nine Examples from MNIST Database	50
25.	Base CNN Model Metrics	56
26.	Base CNN Confusion Matrix	57
27.	Random Search Confusion Matrix.	60
28.	Partial Dependence Plot	63
29.	BO-GP Confusion Matrix	64
30.	TPE Confusion Matrix	67
31.	TPE and TPE-Stochastic Approximation Comparison	68
32.	TPE-Stochastic Approximation Confusion Matrix	69

List of Tables

Table		Page
1.	Hyper-parameter Dimensions	29
2.	Hyper-parameter Dimensions	51
3.	Base CNN Model Metrics	56
4.	Base CNN Model Prediction on New Data Set	56
5.	Base CNN Confusion Matrix	58
6.	Base CNN PREC and SPEC	58
7.	Random Search Best Hyper-parameters	59
8.	Random Search Model Metrics	59
9.	Random Search Model Prediction on New Data Set	60
10.	Random Search Model Confusion Matrix	61
11.	Random Search Model PREC and SPEC	61
12.	BO-OP Best Hyper-parameters	62
13.	BO-OP Model Metrics	62
14.	Random Search Model Prediction on New Data Set	62
15.	BO-GP Model Confusion Matrix	64
16.	BO-GP PREC and SPEC	64
17.	TPE Best Hyper-parameters	65
18.	TPE Model Metrics	66
19.	TPE Model Prediction on New Data Set	66
20.	TPE Model Confusion Matrix	67
21.	TPE PREC and SPEC	67
22.	Kiefer-Wolfowitz Confusion Table	69

HYPER-PARAMETER OPTIMIZATION OF A CONVOLUTIONAL NEURAL NETWORK

I. Introduction

Advancements in computer technology has created a new field of study known as machine learning. A particular area of machine learning is artificial neural networks (ANN). Unlike human brains, machines cannot process and interpret complex real-world information to analyze data and make decisions, this gave rise to ANNs. ANNs, inspired by the human brain, which is connected by a numerous amount of neurons to process information, bring machines a step closer to handle old and new information in a similar manner. ANNs can be used for all types of data and process this data through different layers of mathematical formulations to be able to discern and produce meaningful results. Today, ANNs can be applied to almost anything, such as classifying information, predicting outcomes, and clustering data. Because of their ability to understand complex, dynamic, and non-linear data, ANNs provide a strong alternative to conventional methods that are restricted by assumptions of normality, independence, and linearity [13].

Neural networks process information differently than that of conventional algorithmic approach methods which has a inherent problem: if there are specific steps that are unknown, the computer cannot solve the problem [13]. Because of this inherent issue of conventional methods, researchers and analysts are automatically restricted to problems they know how to solve or data that can be axiomatic. Computers would be even more instrumental to problem solving if they were able to investigate data that cannot be analyzed with conventional approaches, hence neural

networks prove to be more effective in understanding complex high-dimensional data. The aim of this research is to determine if optimization techniques can be applied to neural networks to strengthen its use from conventional methods.

Neural networks were first developed in 1943 and were purely mathematically models. They could not be tested and validated due to primitive technology during this time period. Due to limited technology, neural networks were not used for researchers, but because of the computational power that exists in the market today, neural networks are rapidly being integrated with all aspects of industrial science. Then in 1958, a neural network called perceptron was developed which, theoretically, could train itself to learn any task with a finite set of parameters a user chooses. However, neural networks had a stagnant period due to limited computational power as well as a new problem researchers discovered called the XOR problem. The XOR problem was a classification issue in feed-forward networks, such as perceptrons, to predict an outcome with only two binary inputs. It was not until 1975, when back-propagation was invented, that fixed the XOR problem and also made neural networks more efficient. Finally, between 2009 and 2012, not only were neural networks further developed and tested extensively, but also technology had enough computational power to run large scaled networks efficiently [14].

A simple way of understanding a neural network is to think of it as a composite function: it takes some input and gives an output. There are two components that make up the architecture of a neural network. The first component is the neurons or nodes. A neuron receives data, performs a computation based on an activation function, and then output the restructured data to another neuron. The purpose of a neuron changes depending on its relation to the data and other other neurons. If a neuron receives the data it is in the input layer. If the neuron is in the hidden layer, it performs a computation. If it is in the output layer, it produces an output

for data prediction. The second component is the parameters of the network, also known as the weights, biases, or connection between each neuron. Throughout the training process, these parameters are altered incrementally so that the network is able to tailor itself to a given set of data. An example of a neural network is shown in Figure 1.

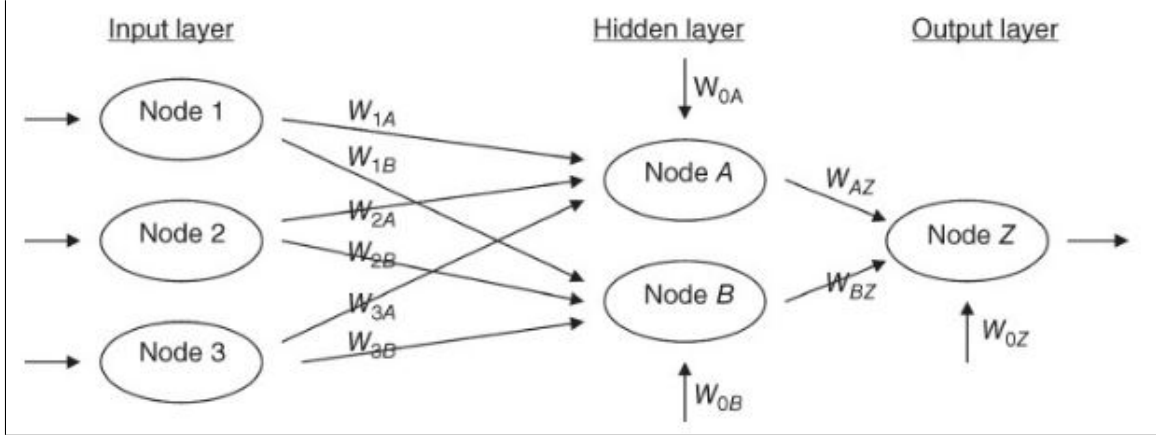


Figure 1. Simple Neural Network. From [1].

The neural network shown in Figure 1 portrays a fully connected feed-forward network. A feed-forward network means that when data flows through the network, it only flows in one direction with no cycling or looping [1]. A fully connected network means that every node in a layer is connected to every node in the sequential layer. However, nodes in the same layer are not connected to each other. The nodes between each layer are connected by weights, such as W_{2A} or W_{3B} . At the start of run for the network, weights are randomly assigned values between 0 and 1. Generally, the number of nodes in the input and output layer are determined by the attributes of the data set; however, the number of hidden layers and hidden nodes are configured by the user. Generally, if a user increases the amount of nodes in the hidden layer, it increases the power and flexibility of the network, but adding too many nodes leads to over-fitting the model resulting in model inaccuracy [1]. One important note is that

in order to use a neural network, all attribute values of the data must be standardized resulting in values between 0 and 1.

The flow of a ANN is that the input layer accepts attributes, x , from the data and sends them into the hidden layer without computational processing. Because of this, the input layer does share the same complexity as that of the hidden and output layers. The hidden layer receives the data from the input layer through a combination function, usually a linear combination of the weights and attributes is performed. Then some bias value, or a constant input, is added to the linear combination. For example, for Node A in the hidden layer of Figure 1:

$$Net_A = \sum_i W_{iA}x_{iA} = W_{0A} + W_{1A}x_{1A} + W_{2A}x_{2A} + W_{3A}x_{3A} \quad (1.1)$$

Equation 1.1 is a linear combination where W_{iA} is the weight connection from a neuron in the input layer to Node A in the hidden layer and x_{iA} is the data. After the network performs this linear combination, the value is transformed through an activation function. Activation functions are also known as mapping functions which take some input and output a value. The most common type of activation is the sigmoid function, other common activations that can be used are tanh, rectified linear unit (ReLU), and softmax functions. The sigmoid function is show in equation 1.2.

$$y = \frac{1}{1 + e^{-x}} \quad (1.2)$$

Figure 2 is a graph of the sigmoid function.

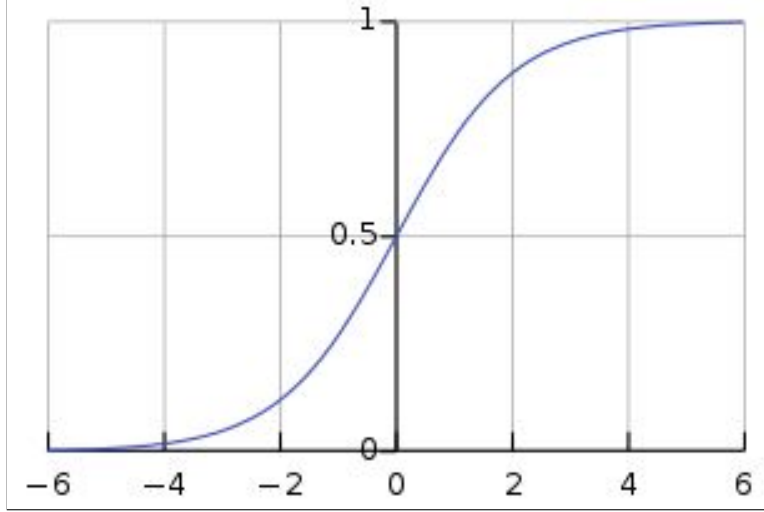


Figure 2. Sigmoid Function. From [1].

Based on an activation function a user sets, Node A from Figure 1, inputs the value it receives from equation 1.1 into x of equation 1.2 and outputs a value y . The output value, y , of Nodes A and B are then passed, via another linear combination, to Node Z in Figure 1. Lastly, another activation function is used on this weighted sum to produce an output from the neural network that represents the predicted value for a specific class of the target variables.

So once data is processed through the network, how does the network learn? Each observation in the data produces a predicted value that is compared to the actual value of a target variable. This comparison produces an error. This error is then sent backwards through the network and is used to update the weights and biases of the network [15]. This process is also known as back-propagation. Back-propagation repeatedly adjusts the weights of the connections between each node in the network in order to minimize the error [16]. However, there is a problem, due to the non-linear nature of the activation functions used in the network, there exists no solution to the set of weights that will minimize the error. Therefore, optimization methods, such as the gradient-descent method or the stochastic gradient descent (SGD) method must be implemented to approximate the set of weights that will minimize the error. The

combination of weights that back-propagation finds is considered to be the solution of the network, as in, it results in the highest prediction accuracy for the ANN [17]. To understand how SGD works for back-propagation, a single neuron model, Figure 3, is used to explain this process.

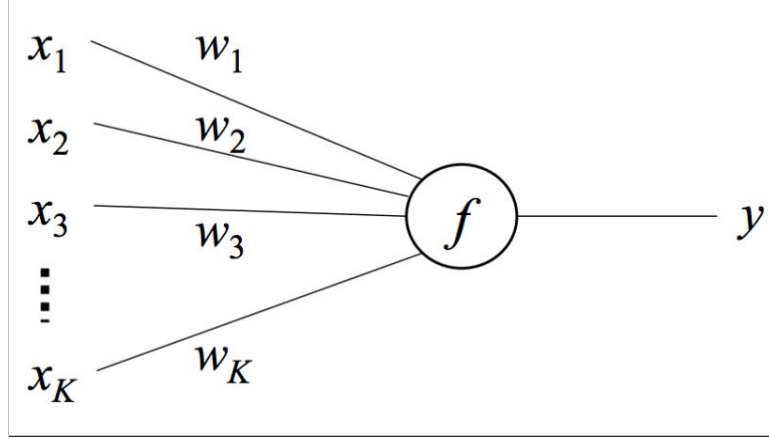


Figure 3. Single Neuron Model. From [2].

Figure 3 shows a single neuron network that combines an input vector, $\{x_1, \dots, x_k\}$, and a weight vector, $\{w_1, \dots, w_k\}$, into a nonlinear function, $\mathbf{f}$, such as the sigmoid function, to produce a scalar output y [?]. In a mathematical representation:

$$y = \mathbf{f} \left(\sum_{i=0}^K w_i x_i \right) = \mathbf{f}(w^T x) \quad (1.3)$$

Equation 1.3 shows the mathematical representation of the neural network taking a weighted sum of the inputs and weights into the sigmoid function, show in equation 1.2. In order to find the set of weights that will minimize the error in the network, this function must be differentiated. By using the chain rule, quotient rule, adding and subtracting techniques, the differentiation of the sigmoid function is shown below.

$$\frac{\partial}{\partial x} \mathbf{f}(x) = \frac{1}{1 + e^{-x}} \left(1 - \left(\frac{1}{1 + e^{-x}} \right) \right) \quad (1.4)$$

Interestingly enough, we can substitute Equation 1.2 back into Equation 1.4 and get the final differential equation for the sigmoid function.

$$\frac{\partial}{\partial x} \mathbf{f}(x) = \mathbf{f}(x)(1 - \mathbf{f}(x)) \quad (1.5)$$

Thus the derivative of the sigmoid function is the sigmoid function times one minus itself. The importance of this derivation of the sigmoid function is that it is extremely efficient in terms of computation speed on large networks composed of multiple layers and neurons. This derivation will be used to learn the weight vector w through SGD to minimize the error in a neural network. Since the objective is to minimize the error produced in the network, the error function typically used is the squared loss function. The squared loss function measures the difference between the actual output t with the predicted output y , or in this case, $\mathbf{f}(w^T x)$. Incorporating the square loss function for this single neuron network gives the following.

$$E = \frac{1}{2}(t - y)^2 = \frac{1}{2}(t - \mathbf{f}(w^T x))^2 \quad (1.6)$$

In Equation 1.6, weights must be found so that it minimizes the objective function error, E , and this is done through SGD. In SGD, the weights of the network are iteratively updated in the direction of the gradient of the loss function until a minimum is found. At each iteration, a single data point or a batch of data is randomly selected and the weights move in the direction of the randomly selected data point or batch. SGD has several advantages over traditional gradient descent. One advantage is that SGD is more computationally efficient since the traditional gradient descent requires loading of the entire data set before an update to the direction of the gradient is made. Therefore, in order to find the gradient, the objective function, E , is derived

with respect to the weight vector w . This gives the following equation.

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial y} \quad (1.7)$$

which equals,

$$= (y - t) \cdot y(1 - y) \cdot x_i \quad (1.8)$$

Now Equation 1.7 is used to update the old weights in the entire network of Figure 3 with a new weight vector. Also, a hyper-parameter known as the learning rate, η , is included, resulting in the following equation.

$$w^{new} = w^{old} - \eta \cdot (y - t) \cdot y(1 - y) \cdot x_i \quad (1.9)$$

This concludes how a single neuron network updates its weights by finding the gradient of the error objective function and substituting y with the values from Equation 1.5, t with the actual value, and the original input vector, x in Equation 1.9. But what about a large neural network with multiple neurons in the hidden and output layers of an ANN? Figure 4 shows how this network might be represented.

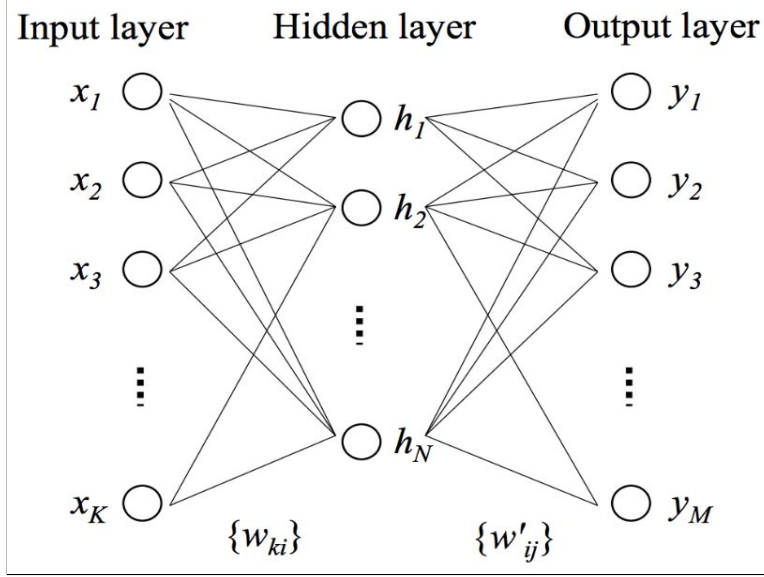


Figure 4. Large Neural Network. From [2].

Figure 4 consists of three layers: input, hidden, and output. The input layer consists of the input vector $\mathbf{x} = \{x_1, \dots, x_k\}$, the hidden layer consists a vector of N neurons $\mathbf{h} = \{h_1, \dots, h_N\}$, and the output layer consists of a vector $\mathbf{y} = \{y_1, \dots, y_M\}$ that represents each class the model needs to try and predict [2]. Each node in the input layer is connected to every neuron in the hidden layer by weight w_{ki} . This indicates which weight is associated with the k^{th} input node to the i^{th} hidden neuron. This indication structure of weights is the same for the hidden layer to the output layer designated by w'_{ij} . The output to each neuron in the hidden layer is shown below.

$$h_i = \mathbf{f}(u_i) = \mathbf{f}\left(\sum_{k=1}^K w_{ki}x_K\right) \quad (1.10)$$

And the output for each output neuron is,

$$y_i = \mathbf{f}(u'_i) = \mathbf{f}\left(\sum_{i=1}^N w'_{ij}h_i\right) \quad (1.11)$$

The error objective function now becomes,

$$E = \frac{1}{2} \sum_{j=1}^M (y_j - t_j)^2 \quad (1.12)$$

where y is the prediction value for a specific class and t is the actual value for the corresponding class. In order to find the gradient of the error, E , that will update the weights for both sets of weights, w_{ki} and w'_{ij} , the gradient of the error function with respect to w'_{ij} must be calculated first. Therefore, applying the chain rule to the error produces the following.

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial w'_{ij}} \quad (1.13)$$

Because differentiating the whole expression at once can be somewhat difficult, each term is isolated to make computing simpler. The first term, differentiating E with respect to w'_{ij} reduces down to $\frac{\partial E}{\partial y_j} = y_j - t_j$. Then, since the derivative of the function $\mathbf{f}$ in respect to its input is the same as Equation 1.5, the second term $\frac{\partial y_j}{\partial u'_j} = y_j(1 - y_j)$, where $y_j = \mathbf{f}(u'_j)$. And finally the last term, $\frac{\partial u'_j}{\partial w'_{ij}}$, where $u'_j = \sum w'_{ij} h_i$, is just h_i [?]. These isolated derivations can now be substituted back into Equation 1.13 to update the weights from the hidden layer to the output layer. To update the weights, Equation 1.13 is multiplied by the learning rate and subtracted from the old weights which is shown below in Equation 1.14.

$$w'_{ij}{}^{new} = w'_{ij}{}^{old} - \eta \cdot \frac{\partial E}{\partial w'_{ij}} \quad (1.14)$$

$$w'_{ij}{}^{new} = w'_{ij}{}^{old} - \eta \cdot (y_j - t_j) \cdot y_j(1 - y_j) \cdot h_i \quad (1.15)$$

Next, the gradient with respect to the weights, w_{kj} , from the input layer to the

hidden layer. Using the chain rule again, the gradient with respect to these weights is,

$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^M \left(\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial h_i} \right) \cdot \frac{\partial h_i}{\partial u_i} \cdot \frac{\partial u_i}{\partial w_{ki}} \quad (1.16)$$

The sum in Equation 1.16 is due to the fact that since the w_{kj} weights are connected to every hidden node, and each hidden node is connected to every node in the output layer, therefore, each of these gradients need to be included in the calculations [2]. The derivatives for $\frac{\partial E}{\partial y_j}$ and $\frac{\partial y_j}{\partial u'_j}$ have already been calculated, leaving $\frac{\partial u'_j}{\partial h_i}$, $\frac{\partial h_i}{\partial u_i}$, and $\frac{\partial u_i}{\partial w_{ki}}$ to be solved. These derivations are shown below.

$$\frac{\partial u'_j}{\partial h_i} = \frac{\partial \sum_{i=1}^N w'_{ij} h_i}{\partial h_i} = w'_{ij} \quad (1.17)$$

$$\frac{\partial h_i}{\partial u_i} = h_i(1 - h_i) \quad (1.18)$$

$$\frac{\partial u_i}{\partial w_{ki}} = \frac{\partial \sum_{k=1}^K w_{ki} h_i}{\partial x_k} = x_k \quad (1.19)$$

And through substitution, a new equation is formed to update the weights from the input layer to the hidden layer, as shown below.

$$w_{ki}^{new} = w_{ki}^{old} - \eta \cdot \frac{\partial E}{\partial w_{ki}} \quad (1.20)$$

$$w_{ki}^{new} = w_{ki}^{old} - \eta \cdot \sum_{j=1}^M [(y_j - t_j) \cdot y_j(1 - y_j) \cdot w'_{ij}] \cdot h_i(1 - h_i) \cdot x_k \quad (1.21)$$

Equation 1.21 shows the full equation to update the weights to the entire network of Figure 4. Originally, the steps to update the weights, w'_{ij} , from the hidden layer to the output layer, did not need to be shown. But by doing so, it streamlines

the process on how the gradient of the error objective function is found through this backward technique, which brings it back to how neural networks learn: a process called back-propagation [1].

This concludes the background portion of this research paper. First a brief history on how neural networks matured was discussed, then a simple example of the architecture of an ANN is explained and represented. Next, the computations that a ANN performs on the data is explained by taking the output of a linear combination into an activation function to produce a predicted output value of a data point. After this, an in-depth mathematically approach was used to discuss on how the network actually learns by tuning its weights from each node to another node through a process called back-propagation. Back-propagation updates the weights by finding the stochastic gradient descent of an error function. Next, hyper-parameters will be discussed that exists in the outside of the network, meaning the network does not update these hyper-parameters during the training process. However, hyper-parameters can drastically influence the rate at which the network learns as well as model performance.

1.1 Hyper-parameters

An artificial neural network consists of model parameters and hyper-parameters. Model parameters are attributes such the weights and biases that the model uses to tailor itself to fit the data. Hyper-parameters are attributes or properties that dictate the entire training process and need to be predefined [18]. Hyper-parameters must be predefined because they cannot be directly learned from the training process. Hyper-parameters define model complexity, its capacity to learn, and the rate of convergence for model parameters; thus finding the optimal value for hyper-parameters leads to better efficiency and results. The hyper-parameters a user can set include the learning

rate, number of hidden layers, number of hidden nodes, number of epochs, batch size, and the type of activation functions, among others. For the purpose of this research, the hyper-parameters considered are the learning rate, number of hidden layers, number of dense nodes, and the batch size.

1.1.1 Learning Rate.

The learning rate is one of many hyper-parameters that a user can tailor in a neural network to improve model accuracy. The learning rate is a constant between 0 and 1 that helps adjust the network weights towards a local or global minimum for an error objective function [1]. The learning rate can be thought of as the speed at which the network learns, meaning how fast the weights of the network converge [1]. When the learning rate is relatively small, the weight adjustments are small resulting in the network taking a long time to converge [1]. However, if the learning rate is increased too much, the learning becomes unstable and can overshoot the optimal solution. This results in the network oscillating back and forth across the error minimum [19]. To portray how this situation may occur, refer to Figure 5.

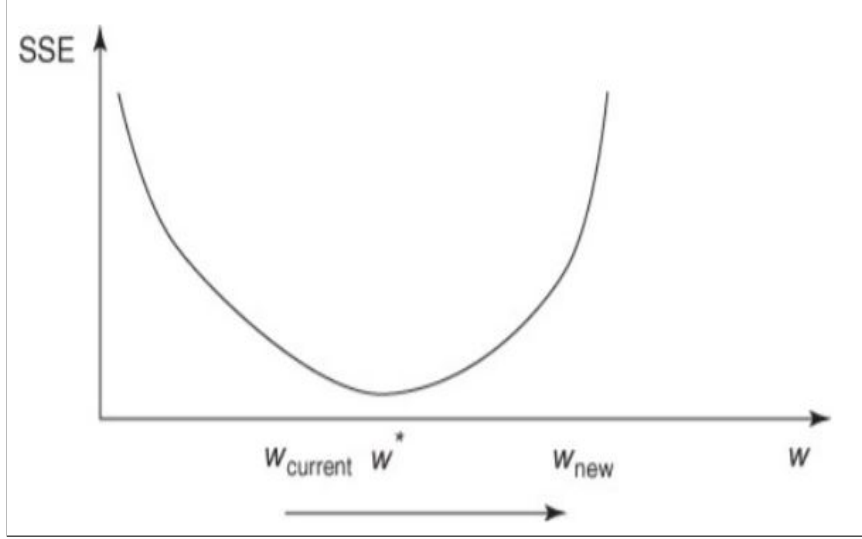


Figure 5. A Large η may cause algorithm to overshoot global minimum. From [1].

From Figure 5, W^* represents the optimal value for a set of weights in a neural network and $W_{current}$ is the current value of the weights. Applying the gradient descent method to this generalized problem means that $\Delta W_{current} = -\eta \cdot \left(\frac{\partial SSE}{\partial W_{current}} \right)$. This results in $W_{current}$ adjusting its weights in the direction of W^* . But if the learning rate is too large, which is a scalar multiple to the gradient, will cause $W_{current}$ to overshoot W^* to W_{new} . Then through the next iteration, W_{new} would be used to update the weights, but will overshoot W^* again because of the large learning rate and end back at the old $W_{current}$. This shows how oscillating between the two slopes of this “valley” could occur in the network and never find its way down to the bottom [?]. Therefore, choosing the appropriate value for the learning rate is extremely important, which is why it’s selected in this research.

1.1.2 Number of Hidden Layers.

The next hyper-parameters considered is the number of hidden layers. In convolutional neural networks, the number of hidden layers can be either referring to the number of convolutional layers or dense layers. In this research, the number of

hidden layers refers to the number of dense layers. The purpose of a hidden layer is straight forward: it transforms the data received by an activation function and sends this value to the output layer or another hidden layer. Currently, there is no in-depth study for a general guideline of how many convolutional layers one should use; however, problems that require more than two hidden layers are rare [4]. Although, this does not mean a user cannot add more. By adding more layers, it can increase the network's ability to learn more complex patterns within a data set by successively learning from each other. For example, the purpose of each convolutional layer may be different in a network used for image classification and the network automatically assigns its function. The first hidden layer might be used to detect "edges" of an image, the second layer is used to detect geometric shapes, and a third layer is used to detect facial features [4]. Although increasing the number of hidden layers may be beneficial, it will increase the computational cost of running the network resulting in more time required to train the network. As for the dense layers in a CNN, they act the same way as a standard ANN.

1.1.3 Number of Dense Neurons.

The third hyper-parameter chosen is the number of dense neurons in the each dense layer. Selection in the amount of hidden neurons is a major concern when using a neural network because it may cause over-fitting or under-fitting. If the user selects too many hidden neurons, over-fitting occurs. Over-fitting indicates that the network trains itself to the data so closely that it loses its generalization ability to predict on new data [4]. Under-fitting occurs when there are not enough hidden neurons in the model to detect the signal used to determine each target variable apart from others resulting in poor accuracy [20]. To combat this issue of over-fitting or under-fitting, a network can be built with a large number of hidden neurons, which

prevents under-fitting, and randomly selects neurons from the network to ignore to prevent over-fitting. This method that prevents the model from over-fitting is called the drop-out technique or rate. The general idea of the drop-out technique is to randomly drop neurons from the network during each iteration of gradient descent. This prevents in the network from adapting to the data excessively [3]. In logistic regression, regularization is used which reduces the likelihood of over-fitting by adding a penalty to the loss function. A neural network mimics regularization by the drop-out rate. Figure 6 represents the architecture of a neural network after the drop-out rate is applied.

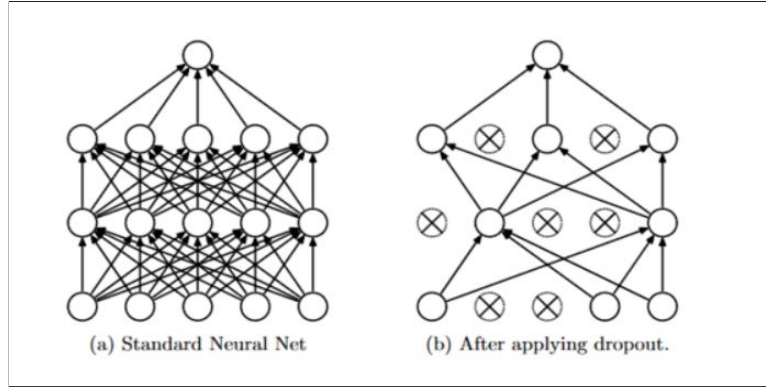


Figure 6. Left: A standard network outwith drop-out. Right: A network with drop-out. From [3].

In Figure 6, each neuron is dropped at some fixed probability $1 - p$ and kept with probability p [3]. The value for p can be different for each hidden layer. When the drop-out technique is added to a network, neurons cannot rely on any individual neurons output (since it may be dropped with some probability), thus forcing the neurons to reduce its dependency on others, and therefore prevents over-fitting [3]. Note that Figure 6 represents one iteration of the model training process. The next time a new batch of data goes through the model, new nodes will randomly be selected and dropped from the model. After the training process is completed, a

model average is calculated and used when the model is evaluated, thus including all nodes of the network. Also, during the evaluation phase, the outputs of all the neurons are multiplied by p so the input to the next layer has the same expected value [3].

1.1.4 Batch Size.

The last hyper-parameter chosen is batch size. The batch size is the number of training samples that will be used to make one update to the model parameters until all training samples have been fed through the network. Ideally, to train a network, each training sample would be used to update the model parameters; however, this is extremely inefficient and time exhausting. By using a batch of samples per iteration, the average of the gradients of each sample in the batch is calculated and then used to update the model parameters. Intuitively, batch size will affect the time it takes to train the network as well as the accuracy.

This concludes the hyper-parameters section and why each hyper-parameter chosen will be used to conduct hyper-parameter optimization. In summary, hyper-parameters are gears that tune the model parameters which leads to a more accurate learning model. Next section will discuss a particular ANN used for image classification which is known as a convolutional neural network.

1.2 Convolutional Neural Network

Convolutional neural networks (CNN) are similar to ANN where it contains weights and biases that the model learns during the training process as described in Section 1.1. However, the layers in CNN are structured in three dimensions: width, height, and depth. Depth for a CNN refers to the activation volume, not to how many nodes or layers there are in the overall network. Also, neurons in one layer do not connect

to all the neurons in another layer, only a region of it. Figures 7 and 8 provides a visualization of the difference between the two.

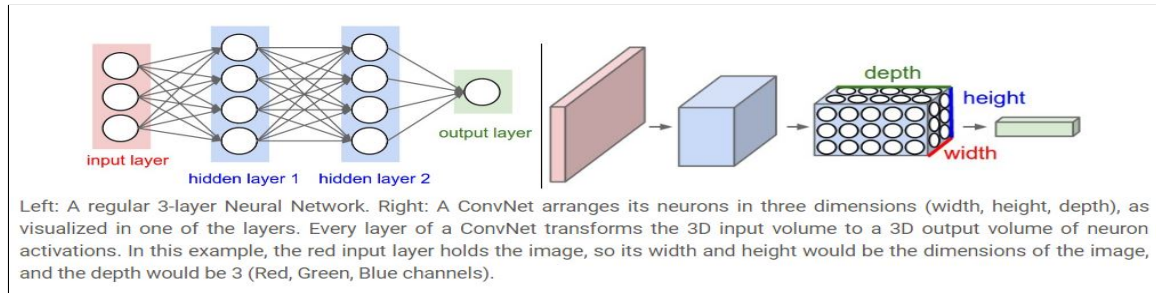


Figure 7. Left: Artificial Neural Network. Right: Convolutional Neural Network.
From [4].

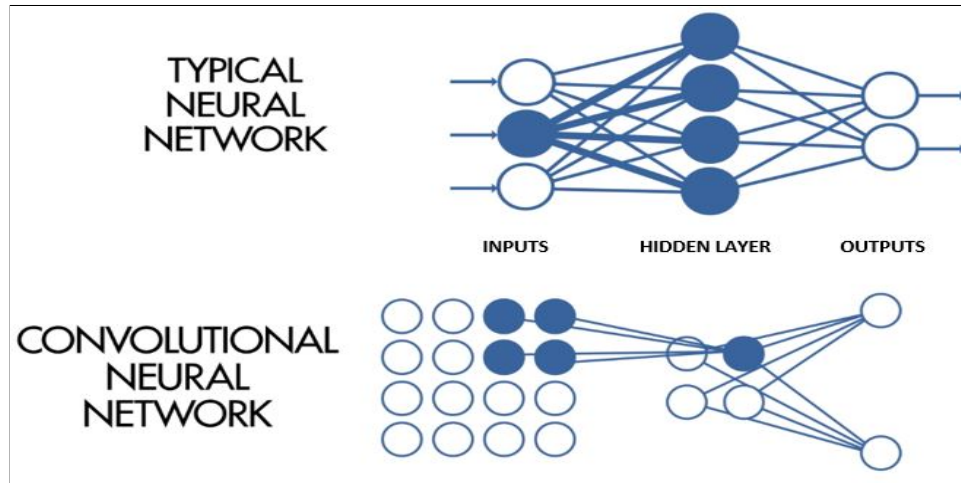


Figure 8. Top: ANN shows that each neuron in one layer is connected to every neuron in the next. Bottom: CNN shows that only a small region of neurons from the input later is connected to a neuron in the hidden layer. From [5].

Figure 8 shows that only a region of neurons in the input layer is connected to one node in the hidden layer. These regions of neurons in the input layer are known as local receptor fields and these fields are translated over to the hidden layer to create a featured map of values by a mathematical process called convolutions. All CNN models follow a similar architecture as shown in Figure 9, but a user can add as many layers to detect as many different features of the image he or she wants.

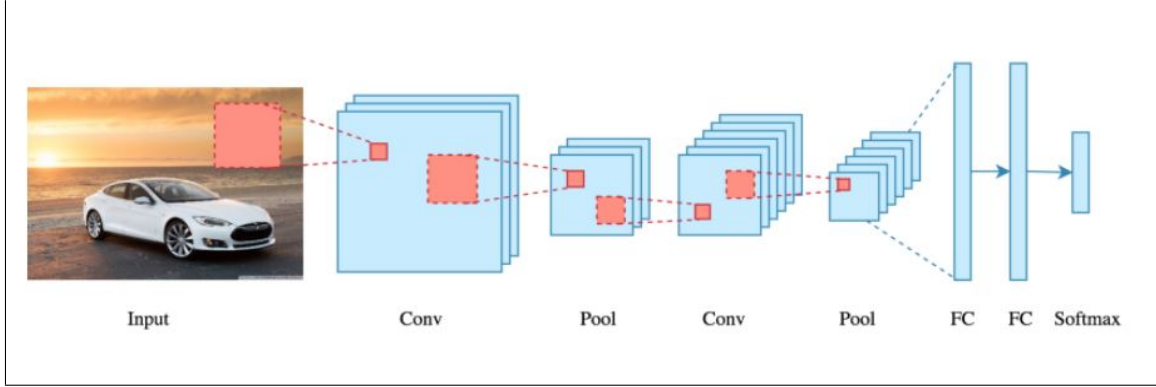


Figure 9. Typical CNN Architecture. From [6].

From Figure 9, the typical flow of a CNN is that it takes some input image that contains the raw pixel values for that image, for example a $[32 \times 32 \times 3]$ indicates that the image's width and height is 32 pixels with three color components: red, green, and blue. The convolution layer will create outputs that are fed through an activation function which creates a featured map. These featured maps are down sized through a method called max pooling. More convolutional layers are added if a user chooses and each will automatically detect meaningful features of the image. Once the image is fed through each convolutional layer, it is passed to a fully connected network that will be used to score each image. This section outlines each component of a CNN in detail.

In order to understand how CNN's transform the images it receives into data it can understand, convolutions first need to be discerned. Convolutions can be thought of as the main building block for the main layers of a CNN [6]. It is a mathematical process that merges two sets of information, or in a case for CNN, it combines input data with a convolutional filter or kernel to produce a featured map. To understand this process a little better, refer to Figure 10 which depicts an input image and a convolutional filter.

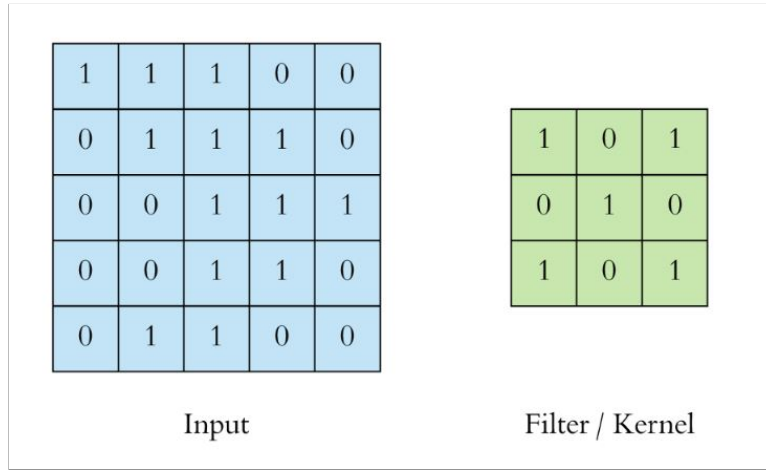


Figure 10. Left: Example of a input image. Right: Convolutional Filter/Kernel. From [6].

From Figure 10, the right $[3 \times 3]$ matrix is a example of a kernel and because of the size of the kernel, the network would perform a $[3 \times 3]$ convolution with the input matrix. The CNN will perform the convolution operation by sliding the kernel over the input matrix. At every location the kernel slides onto, an element-wise matrix multiplication is performed and the sum of each element produces an element of the featured map. This mathematical computation is known as Frobenius Product, which is show in Figure 11.

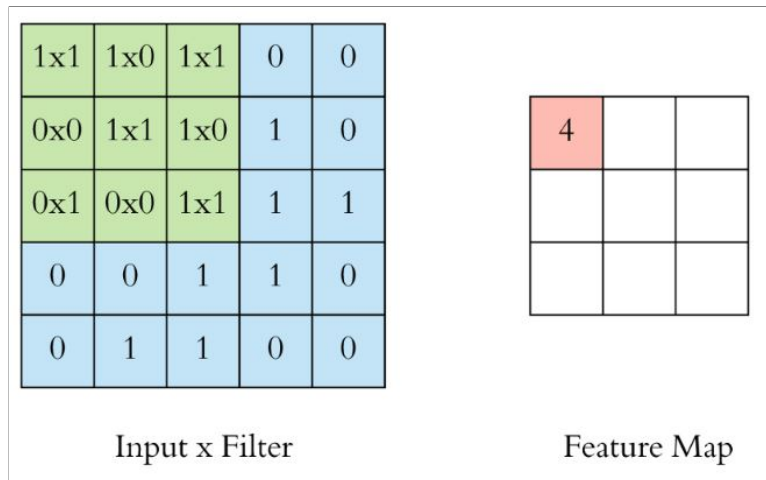


Figure 11. Frobenius Product. From [6].

In Figure 11, the kernel overlays itself onto the input matrix and performs the Frobenius Product to produce the first element in the featured map matrix, located to the right of the figure. The kernel then slides over one column and performs the convolution again, which is shown in Figure 12.

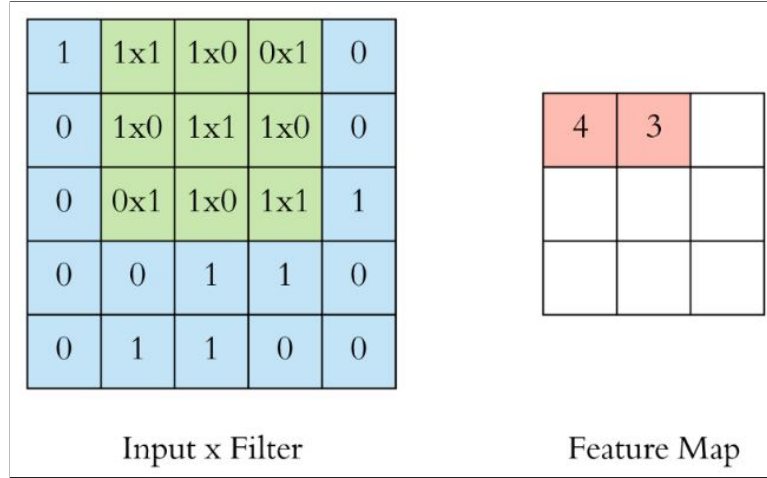


Figure 12. Next iteration of convolution. From [6].

The network continues to perform convolutions until the featured map is completed. The examples shown from Figures 10 to 12 represent convolutions that are two dimensional using a kernel size of three ($[3 \times 3]$); however, CNNs perform convolutions in three dimensions. In three dimensions, numerous convolutions are performed, each using a different filter (kernel) to produce a unique featured map [6]. Then, these unique featured maps are stacked together to create the final structure of the convolutional layer. Figure 13 shows convolutions being performed in three dimensions with a filter size of $[5 \times 5 \times 3]$ on an input image size of $[32 \times 32 \times 3]$ that results in one element $[1 \times 1 \times 1]$ of a distinct featured map until a $[32 \times 32 \times 1]$ map is created.

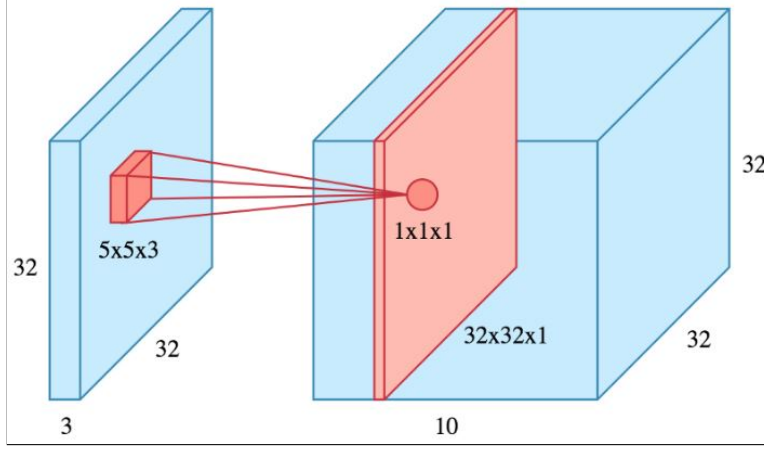


Figure 13. 3D Convolutions. From [6].

From Figure 13, the red slice that has the dimensions of $[32 \times 32 \times 1]$ is a completed distinct feature map. Figure 13 also shows that ten different filters or kernels are used to produce the convolutional layer which has a size of $[32 \times 32 \times 10]$. The number of filters that the network uses can be set by the user. Earlier, convolutions that were produced in two dimensions produced a smaller matrix for the featured map; however, in Figure 13, the size of the feature map is the same as the input width and height dimensions. This is known as padding. Padding is a technique used in CNN that surrounds the layer with zeros-valued pixels, thus preventing shrinking in the feature maps when multiple convolutional layers are used [6]. This will not only allow the spatial size of the feature map to remain the same as the input image, but also improves performance and makes sure the kernel and strides of the kernel will fit the input [7]. Strides refer to the step size the convolution filter moves each time. Strides are usually a size of one, meaning the filter slides one pixel at each iteration. If a user increases the strides, the kernel slides over the input with larger intervals resulting in less overlap between receptor fields and a smaller featured map [6].

Once each feature map is completed in the convolutional layer, the results of each map is passed through an activation function before being used as the final featured map of the layer. The most common activation function used in CNN is the ReLu

function. The ReLu function is a basic activation function which takes an input and returns either the same value, or zero if the input value is negative. The ReLu function is used because it was found to greatly accelerate the convergence of SGD compared to that of sigmoid or tanh functions [21]. After the convolutional layer is built, the max pooling technique is used to reduce dimensionality. Max pooling enables the network to reduce the number of parameters required to be trained, thus shorting the training time and combats over-fitting [7]. Max pooling takes the maximum value of a "window" that the user specifies. Figure 14 shows how max pooling works with a window size of $[2 \times 2]$ and a stride of two.

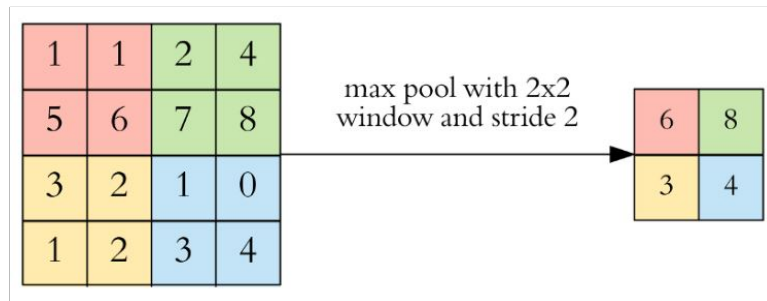


Figure 14. Max Pooling. From [7].

By using a max pooling with size $[2 \times 2]$ and a stride of two, this reduces the dimensionality of the feature maps shown from Figure 13 to $[16 \times 16 \times 10]$ as shown in Figure 15. In a sense, max pooling down samples the image while keeping the depth the same, this results in a lower resolution of an image.

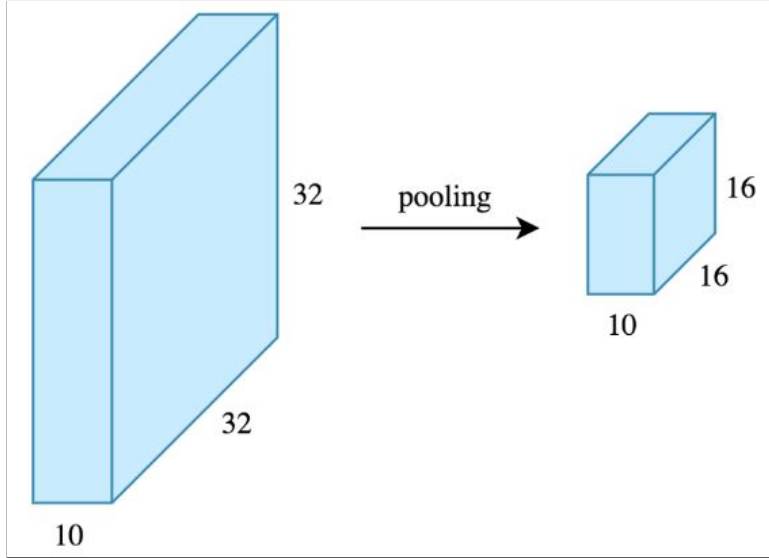


Figure 15. After applying max pooling. From [6].

By halving the height and width of the feature maps, the number of weights that need to be trained by is reduced 25% [6]. In CNN architectures, convolutions are usually done with a $[3 \times 3]$ filter, stride one, and padding, while max pooling is done with a $[2 \times 2]$ window, stride two, and no padding. To complete the network after convolutional layers and pooling, fully connected layers are added as shown in Figure 9. Since fully connected layers only accept one dimensional vector of numbers, the output of the final pooling layer is flattened. This means that the three dimensional data is rearranged into a significantly large one dimensional vector. An example of flattening for one pooled feature map is shown in Figure 16.

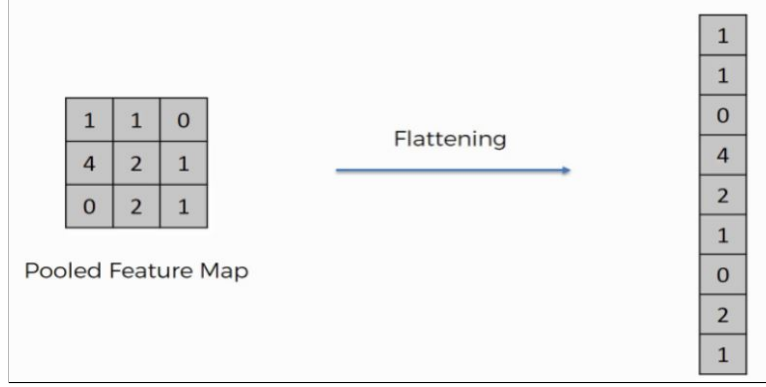


Figure 16. Flattening. From [8].

Once flattening occurs, the data is then fed into the fully connected layers, also known as dense layers, and perform in the same way as a standard ANN. The first fully connected layer (first dense layer) acts the same way as the input layer of an ANN. This layer takes the one dimensional data from the output of the last pooling layer, and the number of nodes in the dense layer is set by the user. The second fully connected layer (second dense layer), similar to that of a hidden layer, will perform computations and pass it onto the softmax layer (output layer). The output layer uses the softmax activation function to compute each classes score. The number of nodes in this layer will typically be the number of classes, C , the CNN is trying to predict. For example, if it was trying to classify and predict eight different kinds of animals, the output will be eight nodes each with a probability associated with each animal. The softmax activation function, f , is shown in Equation 1.22 that gives the probability of a unique class S_i .

$$f(x_i) = S_i = \frac{e^{(x_i)}}{\sum_{i=1}^C e^{(x_i)}}, \text{ for } i = 1, \dots, C \quad (1.22)$$

In Equation 1.22, x_i is the output from the last dense layer of the fully connected portion of the CNN. A diagram that shows the connection between the dense layer and the softmax layer is shown in Figure 17 below.

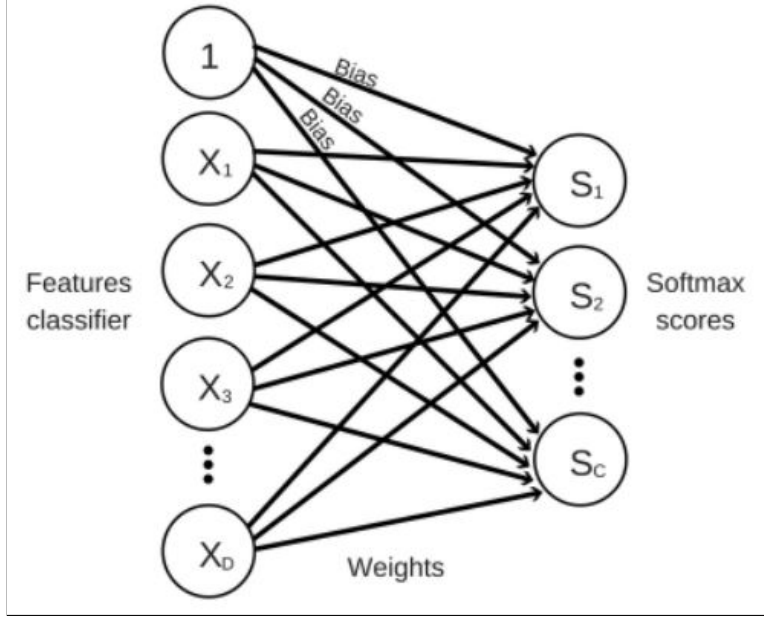


Figure 17. Dense Layer to Softmax Layer. From [9].

From Figure 17, the softmax classifier takes the input features, X_i , and multiplies them with a matrix of weights and adds a bias to it. This results to a score, x_i , for each i_{th} class that is used for the input of the softmax function. The softmax function calculates the probability distribution of a specific event over a number of different events, or in other words, the probability of each target class over all target classes [9]. The probabilities are calculated through an exponential function and take a value between 0 and 1 which sum up to 1. Because of the exponential property of the softmax function, higher values get emphasized more and lower values get suppressed more [9]. The softmax function is important for a CNN because without it, each class could end up having a high probability rendering the network useless for prediction [9].

As mentioned before, an ANN updates its parameters through back-propagation when the gradient descent of the error function is calculated with respect to the current weights. However, for a CNN the gradient descent is performed on a loss function, which is similar to an error function. The loss function for the softmax

function is called categorical cross-entropy which is shown in Equation 1.23 [22].

$$E = - \sum_i^{numClasses} t_i \log(S_i) \quad (1.23)$$

Where t follows a probability distribution that represents the correct classification output of the i_{th} target class. This parameter is typically a one-hot encoded vector and its size is based on the number of classes. One-hot encoded vector means that all the elements are zero except one, which corresponds to the correct classification. Now to determine the update for the weights in Figure 17, the gradient of the cross entropy function with respect to the weights for a i_{th} node in the softmax layer is shown in Equation 1.24 [22].

$$\frac{\partial E}{\partial s_j^1} = \sum_i^{numClasses} \frac{\partial E}{\partial x_i} \cdot \frac{\partial x_i}{\partial S_j} \cdot \frac{\partial x_i}{\partial s_j^1} \quad (1.24)$$

In Equation 1.24, s_j^1 refers to all the nodes in the hidden layer that are connected to one node in the softmax layer. Once again, each term is isolated to make differentiating simple. The term $\frac{\partial E}{\partial x_i}$ already incorporates the derivation of the activation function and reduces down to $(S_i - t_i)$. The second term, $\frac{\partial x_i}{\partial S_j}$ derives to just the weights w_{ij} . The third term, $\frac{\partial x_i}{\partial s_j^1}$, reduces down to $S_i(1 - S_j)$ [22]. Thus, substitution these derivations back into Equation 1.25 gives the following equation.

$$\frac{\partial E}{\partial s_j^1} = \sum_i^{numClasses} (S_i - t_i)(w_{ij})(S_i(1 - S_j)) \quad (1.25)$$

Equation 1.25 is used to updated the weights from all the neurons in the hidden layer (second dense layer) to a specific node j in the softmax layer.

This conclude this section of the architecture of a convolutional neural network. To summarize, a CNN will take a raw three dimensional as the input, then compute

the output of neurons that are connected to local regions of the input creating a featured map. This computations are done through convolutions and make up the convolutional layer. Then a ReLu activation is used on the featured maps to complete the output of the convolutional layer. The convolution layer automatically detects meaningful features of a given image and each layer builds on top of each other. Once this is done, max pooling is used to down size the image to a smaller spatial dimension but keep the depth the same. Finally, the output from the pooling layer is sent to the fully connected layers, also known as dense layers, which act the same was as a standard artificial network network. The last layer of the dense layers computes class scores through the softmax function. The architecture of CNN will be used as the base model and compared to other models that include hyper-parameter optimization. Next section discusses the motivation to why hyper-parameter optimization is important.

1.3 Problem Statement

As mentioned previously, hyper-parameters are important because they directly influence not only the rate at which a neural network learns, but also affects the performance. In an ideal world with an infinite amount of time and computational power, all possible values of each hyper-parameter should be tested and validated to determine the optimal set of values for a given network. The purpose and architecture of each neural network will always be different unless the data and architecture is the same, so finding the set of optimal hyper-parameters should always be considered. However, people do not live in an ideal world, people live in an imperfect world where technology is limited and time is of the essence. Therefore a compromise must be reached, which is why hyper-parameter optimization is important. Some networks could be built without hyper-parameter tuning and may end up with a

high prediction or classification accuracy. But even if a network produced a model with a high prediction accuracy of 95%, the 5% miss-classification rate would still be significant when a network has hundreds of thousands even millions of data points to sift through. There are many types of optimization algorithms that can be applied to a neural network, but since the architecture of a CNN is different, which optimization method is the best in terms of time and model performance? The optimization methods applied in this research are the random search, Bayesian Optimization with Gaussian Process, and tree of parzen estimator approaches. The reason why these methods were chosen is discussed in Chapter 2 of this paper.

1.4 Research Scope

The impact of hyper-parameter optimization allows the user to find an optimal set of values that will improve model performance. Hyper-parameter optimization techniques improves model performance by finding set of hyper-parameter, λ^* , that minimizes a generalization error objective function, E , for a given technique [23]. However, not all possible values can be explored, therefore, the spatial dimension of each hyper-parameter must be defined before implementing each optimization technique. Table 1 shows the dimensions for each hyper-parameter chosen.

Table 1. Hyper-parameter Dimensions

Hyper-parameter	Lower Bound	Upper Bound	Base
Learning Rate	0.0001	0.1	0.01
Number of Dense Layers	1	5	2
Number of Dense Nodes	5	512	128
Batch Size	16	512	200

From Table 1, the “base” column indicates the values of the hyper-parameters used in the base CNN. The base value for the learning rate was chosen due it being the default value for an application programming interface (API) called Keras. Keras

will be discussed later in the methodology chapter. The number of dense layers and dense nodes are arbitrarily picked because there exists no rule for the values of these hyper-parameters. The only information that is known about these hyper-parameters is that they do have the ability to affect how well the network interprets complex patterns. Last, the value for the base batch size was chosen based on Palvo Radius’s research who found that the optimal batch size values from 200 and greater depending on computational resources [24].

1.5 Research Objectives

The aim of this research is to determine if applying hyper-parameter optimization techniques to a simple convolutional neural network improves network performance. The word ”simple” implies that only two convolutional layers are used in the network. Adding more convolutional layers may benefit the overall model; however, will not only require more computational time, but also may actually have an inverse effect shown in Figure 18.

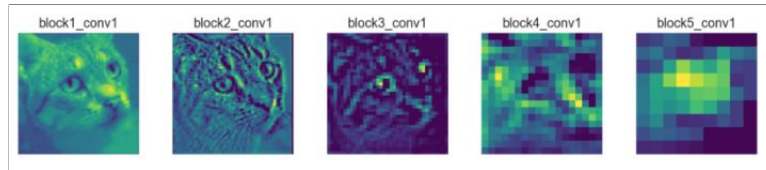


Figure 18. Adding more convolutional layers. From [6].

From Figure 18, adding more convolutional layers will not only extend the training process, but also produces less meaning feature maps that will not be beneficial to increasing model performance.

1.6 Limitations

The main limitation of this research is the computational resources which will affect how large the network is and the speed at which it trains. To combat this issue, the number and type of hyper-parameters were chosen, the dimensions of each hyper-parameter, and a limited network architecture. The type of hyper-parameters chosen exists in any type of artificial neural network. A convolutional neural network contains more hyper-parameters such as the number of convolutional layers to use, the size of the kernel, stride, and padding. However, these were not chosen because these hyper-parameters cannot be applied to other networks.

1.7 Summary

Hyper-parameter optimization should be included in the training process of every type of neural network. In this chapter, an in-depth explanation of how an artificial neural network learns by tuning its weights from each node to each layer is chronologically discussed. Then, not only the reasons behind which hyper-parameters were chosen, but also how they affect model performance was explained. Finally, the differences in the architecture between a standard neural network and convolutional neural networks was discussed as well as how each hyper-parameter chosen functions in each network. Chapter 2 discusses each optimization technique chosen.

II. Literature Review

2.1 Overview

Before hyper-parameter optimization techniques existed, the choice of hyper-parameters and their values were made by the user. However, not only does this require a significant amount of experience and time to conduct, but also results are typically not reproducible since every network architecture is different [23]. Studies have shown that not only more sophisticated and automated processes, such as random search, can find a better set of hyper-parameters to increase model performance, but also reduces the amount of time required to find them. Three optimization techniques chosen in this research are random search, Bayesian Optimization Gaussian Process (BO-GP), and tree of parzen estimators (TPE) approach.

Typically, the goal of any training optimization algorithm, $\mathcal{A}$, is to find a function, f , that will minimize some expected error or loss, $\mathcal{L}$, from independent samples, x , drawn from a some distribution G_x . The function is evaluated by the learning algorithm, $\mathcal{A}$, which is obtained after choosing a set of hyper-parameters, λ , and then maps a given data set, X^{train} , from G_x [10]. This can be denoted by by Equation 2.1

$$f = \mathcal{A}_\lambda(X^{train}) \quad (2.1)$$

What is really required by hyper-parameter optimization is to minimize a generalization error or loss function, $\mathbb{E}$, which is denoted by Equation 2.2.

$$\mathbb{E}_{x \sim G_x} = [\mathcal{L}(x; \mathcal{A}_\lambda(X^{train}))] \quad (2.2)$$

However, there is a problem with using Equation 2.2, the expected value cannot be evaluated over the unknown distribution G_x , which is the end goal to optimize.

Therefore, an approximation is used to guesstimate the results that will get close to the true optimum. This approximation is done by replacing the expected error with either an arguments of minimum for random search or arguments of maximum for BO-GP or TPE. These different techniques will later be discussed.

First, the random search approach is chosen because it is a widely used yet simple optimization method. This approach draws a random value from a pre-defined distribution as defined in Table 1 in Chapter 1 for each hyper-parameter chosen. Second, Bayesian Optimization is a type of black-box optimization where the user does not know anything about the objective function, but wants to find a minimization or maximization to the objective function. The objective function is also known as a surrogate model, which for this case, is the Gaussian rocess model. The Gaussian process then uses an acquisition function to determine which set of hyper-parameters to evaluate next. The last approach, TPE, collects new observations at each iteration and then decides which values of hyper-parameters to use next.

2.2 Random Search

The explanation of how random search is implemented is fairly short. Generally, two widely used optimization techniques are the grid search and random search. These two techniques are both simple to implement and powerful compared to that of manual based optimization. The grid search approach generates every possible combination of the hyper-parameter space and outputs the best resulting set. However, with higher dimensional hyper-parameter space, grid search will inevitably suffer from the phenomenon known as curse dimensionality [10]. When a unique hyper-parameter is added to the hyper-parameter space, it increases the amount of combinations required to be researched exponentially resulting in a slow convergence rate, this is curse dimensionality. Random search typically supersedes grid search because of its

ability to handle higher dimensions with a probability of a faster convergence rate, thus providing a relatively good solution quickly and easily [25]. Random search independently draws its samples from a uniform density space which has configuration space as that of the grid search. Figure 19 shows a comparison between grid search and random search.

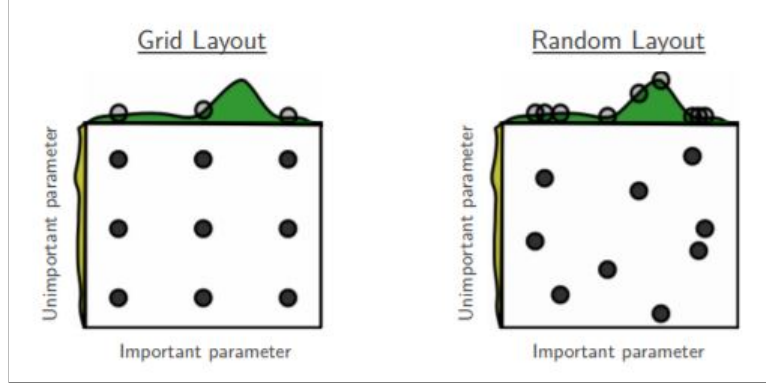


Figure 19. Grid Search vs Random Search. From [10].

From Figure 19, two functions are shown that could be used to estimate an objective function such as Equation 2.1. The distribution of these functions are shown in yellow and green, but are unknown to the user. Since the goal is to find a set of points that will maximize or minimize the objective function by combining the output of the two functions, the random search is advantageous. Grid search allocates a significant amount of time to explore dimensions that do not matter as well as poor coverage of the space [10].

In order for random search to evaluate each set of hyper-parameters that are chosen, it uses the K-fold cross-validation method. Cross-validation is a resampling procedure that splits the data set into K groups, which is set by the user. Once models are generated based on the K-fold cross-validation, the average error across all K trials is computed, thus returning the optimal values of the model. The basic strategy for random search is that for each hyper-parameter combination that is chosen, the K-fold

cross-validation score is evaluated and stored along with the hyper-parameters, and then outputs the set of values for the hyper-parameters that yielded the highest cross-validation score. However, there is no requirement for how the algorithm chooses the next set of hyper-parameters; they are just chosen randomly [10].

2.3 Bayesian Optimization with Gaussian Process (BO-GP)

Bayesian optimization (BO) is a formal process of sequential model-based optimization (SMBO) technique. SMBO algorithm use previous observations of a loss function to determine the next point to sample for this loss function. BO has two important components: prior function and acquisition function. The prior function is the probability measure on a black-box function, meaning the user has no idea what the objective function is. The prior function will be updated to a posterior function when new data is introduced to the prior. The acquisition function is the criteria used to decide where to evaluate the black-box function next in order to gain the maximum amount of information to where the true global optimum is for the black-box function. Because the black-box function may be computationally expensive to evaluate, a general model called the surrogate model is replaced with the black-box function which is computationally cheaper and used to solve the optimization problem. Gaussian processes (GP) are commonly used as the surrogate model in BO [26]. A Gaussian process is a generalization of the Gaussian probability distribution. Leaving mathematics aside, the idea behind GP is that if the properties of an unknown function is finite by a number of points, then the Gaussian process will produce the same answer as if it considered infinitely many other points [27]

The idea behind GP is that for every input x , there is an output y for $f(x)$, where f is a stochastic function. The inputs come from a Gaussian distribution with mean μ and standard deviation σ . The generalization of the Gaussian distribution comes

from the multivariate Gaussian distribution which is defined by a mean vector and a covariance matrix. Therefore, the generalization has a finite number of possible inputs. Again, the process for Bayesian Optimization with Gaussian Process (BO-GP) is that this technique incorporates a prior function about the objective function. Then updates the prior with samples in the space of the objective function to get a posterior function which is used to approximate the objective function [28]. The posterior function is a generalized Gaussian process model. The prior and posterior functions which are from a probability density function of the normal distribution, $\mathcal{N}$, are shown in Equation 2.3 and 2.4

$$p(f|X) = \mathcal{N}(f|\mu, K) \quad (2.3)$$

$$p(f_\star|X_\star, X, y) = \mathcal{N}(f_\star|\mu_\star, \Sigma_\star) \quad (2.4)$$

The Gaussian process defines any point, x , to be assigned a random variable $f(x)$ and the joint distribution of any random variables is Gaussian and defined by $p(f(x_1), \dots, f(x_n))$. In Equation 2.3, $f = (f(x_1), \dots, f(x_n))$, μ is the mean which is typically zero, and K is the positive definite kernel function of the covariance matrix of $(x_1), \dots, (x_n)$. In Equation 2.4, the posterior function is converted from Equation 2.3 after observing some data y . This function can now be used to make predictions $f_\star$ given a new input $X_\star$. The covariance matrix, $\Sigma_\star$, is generated after given the new set of inputs. By definition of the Gaussian process, the normal joint distribution of observed data y and predictions $f_\star$ is shown in Equation 2.5 [11].

$$\begin{pmatrix} y \\ f_\star \end{pmatrix} \sim \mathcal{N} \left(0, \begin{pmatrix} K_y & K_\star \\ K_\star^T & K_{\star\star} \end{pmatrix} \right) \quad (2.5)$$

In Equation 2.5, K_y equals $K + \sigma_y^2 I$, $K_\star$ equals $K(X, X_\star)$, and $K_{\star\star}$ equals $K(X_\star, X_\star)$. σ_y^2 is the noise term that is located in the diagonal of K_y . Here K is the matrix by applying the kernel function to the observed values of x , which results in the similarity between observations. The symbol $K_\star$ gives the similarity of the training values to the test values which are trying to be estimated. Lastly, $K_{\star\star}$ gives the similarity of the test values to each other. In order to obtain the values to use in Equation 2.4, the equations for $\mu_\star$ and $\Sigma_\star$ are shown below [11].

$$\mu_\star = K_\star^T K_\star^{-1} y \quad (2.6)$$

$$\Sigma_\star = K_{\star\star} - K_\star^T K_\star^{-1} K_\star \quad (2.7)$$

Equations 2.3 to 2.7 are the minimum required information to implement the Gaussian process. There are a number of kernel functions that can be used in the Gaussian process. The kernel function used to build the covariance matrix is the Matern kernel, which is not shown because it is outside the scope of this literature review. The Matern Kernel is a generalization of the squared exponential kernel.

Bayesian Optimization determines the next point to sample, $X_\star$, by using an acquisition function. The way acquisition functions work is that there is a trade-off between exploitation and exploration. Exploitation means that the acquisition function will look to sample from areas where the surrogate model (Gaussian model) predicts a high model value. Exploration means that the acquisition function will look to sample from areas where there is a high uncertainty of prediction. The goal here is to maximize the acquisition function to determine the next point to sample. Common acquisition functions implemented are maximum probability of improvement, upper

confidence bound, or expected improvement [28]. The acquisition function that will be used in this research will be expected improvement (EI) shown in Equation 2.8.

$$EI(x) = \mathbb{E} \max(f(x) - f(x^+), 0) \quad (2.8)$$

The next sample point, x_t , which is the argumentative maximum of the EI with respect to the current point and the previous point is found. The procedure for BO is as follows, for $t = 1, 2, \dots, n$:

1. Find the next sample point by optimizing EI from the Gaussian Model:

$$x_t = \operatorname{argmax}_x EI(x | D_{1:t-1}) \quad (2.9)$$

2. Obtain a sample, y_t , where $y_t = f(x_t) + \epsilon_t$
3. Add sample to previous samples, $D_{1:t}$, where $D_{1:t} = D_{1:t-1} \cup (x_t, y_t)$

From Equation 2.8, $f(x^+)$ is the value of the best sample, x^+ . The location of x^+ is from $\operatorname{argmax}_{x_i \in x_{1:t}} f(x_i)$. Since we are finding EI through the Gaussian Process, the analytical evaluation is shown in Equation 2.9.

$$EI(x) = \begin{cases} (\mu(x) - f(x^+) - \xi)\Phi(Z) - \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \quad (2.10)$$

where,

$$Z = \begin{cases} \frac{\mu(x) - f(x^+) - \xi}{\sigma(x)} & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \quad (2.11)$$

In Equations 2.9 and 2.10, $\mu(x)$ and $\sigma(x)$ are the mean and the standard deviation

of posterior function at x respectively. The cumulative density function of the standard normal distribution is represented by Φ . The probability density function of the standard normal distribution is represented by ϕ . Also, from the top entry in Equation 2.9, the first term is the exploitation and the second term is the exploration term. Parameter ξ determines the amount of exploration. Higher values of ξ leads to more exploration. This means that increasing ξ will lead to improvements decreasing relative to potential improvements from areas of high prediction uncertainty which are represented by large values of $\sigma(x)$ [11]. The default value for ξ used in this research is 0.01. Figure 20 shows how BO-GP performs with three iterations.

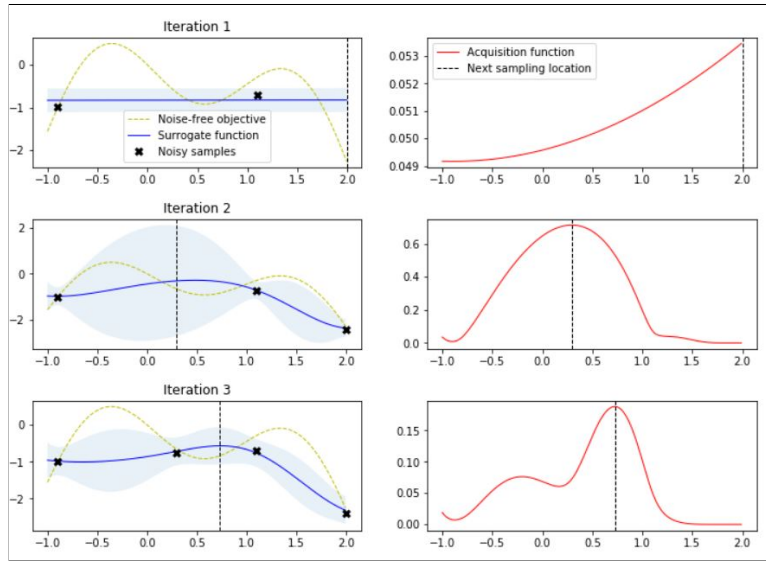


Figure 20. Three iterations of Bayesian Optimization with Gaussian Process. From [11].

From Figure 20, the left side of the graph shows the surrogate model (blue line) overlaying itself onto the actual function (yellow). The blue shaded regions represent the exploitation and exploration. This algorithm requires a starting point (black) which is used to produce the next point to sample for each iteration. How it determines the next point is by an acquisition function (red), which is shown on the right of the figure. The vertical line in all the sub-graphs represents the location of the next

sampling point.

Finally, how is the concept of BO-GP applied to hyper-parameters? In essence, the algorithm will find the next set of hyper-parameters, from an given initial starting point, that will maximize the acquisition function of the Gaussian process model, and in return, minimizes the validation of the error for the convolutional neural network after training and evaluation of the network based on the initial starting point.

2.4 Tree of Parzen Estimator (TPE)

As discussed in the previous section, SMBO is a formal process of Bayesian Optimization which refers to performing multiple trials sequentially to find an improved set of hyper-parameters by applying Bayesian reasoning and updating a probability model known as a surrogate. The main difference between each SMBO process is which criteria is used to evaluate the next set of hyper-parameters to use based on surrogate model. Although in the case of tree of parzen estimator (TPE) approach, this approach deviates from a standard SMBO because it does not define a predictive distribution over the black-box objective function. Instead, it creates two hierarchical processes, $\ell(x)$ and $g(x)$, that act as generative models for each hyper-parameter domain. Also, this method requires that the hyper-parameter space has a pre-defined distribution of how values are drawn [29]. A generative model is a statistical model of the joint probability distribution, thus these models represent the domains of the variables with a defined quantile, γ , to build an objective function shown in Equation 2.11.

$$p(x|y) = \begin{cases} \ell(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases} \quad (2.12)$$

From Equation 2.11, $p(x|y)$ is defined by two densities. $\ell(x)$ is the density formed by using the observations from a particular space, x^i , such that the corresponding loss

is less than y^* . y^* is the performance score that the algorithm generates on its own. $q(x)$ is the density formed by using the remaining observations. TPE chooses y^* based on some quantile (γ) of the observed values, y , meaning that the probability of $y < y^*$ equals γ [30]. The value used for γ is 25%. Both hierarchicals are formed by using the parzen-window density estimation, also known as the kernel density estimation.

However, before the TPE can generate these density estimates, it must first generate a small set of observations. Observations are the performance scores of a neural network. These observations are created by using the random search algorithm on pre-defined hyper-parameter search space. The performance score of the selected observations report back to the TPE algorithm, which then splits the observations into two groups (good or bad) based on the score.

The next part of the algorithm is to model likelihood probability for the posteriors for each of the groups. From the sampled hyper-parameters generated, a sample that is more likely to be in $\ell(x)$ and less likely to be in $q(x)$ is found by using Equation 2.12.

$$EI(X) = \frac{\ell(x)}{q(x)} \quad (2.13)$$

Equation 2.12 is the expected improvement for each sample made. To visualize the general idea of what TPE is doing, refer to Figure 21.

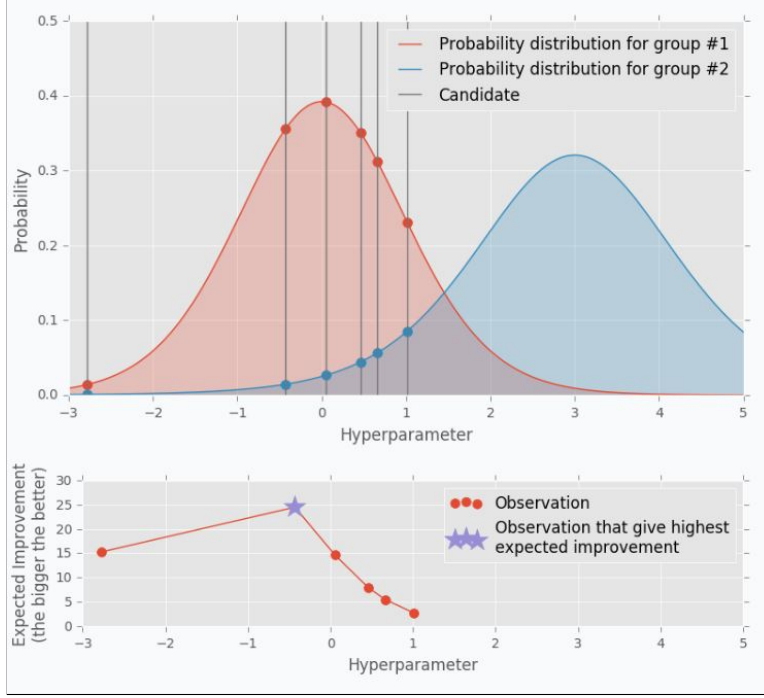


Figure 21. Distribution of Good and Bad Groups. From [12].

Figure 21 shows two pre-defined probability density functions of the t-distribution for $\ell(x)$ in red and $g(x)$ in blue. The bottom graph of the figure shows the expected improvement of six samples that were draw. In essence, given a set of hyperparameters, $\ell(x)$ is the probability that the set is in the good group and $g(x)$ is the probability in the bad group [12]. The ratio with the highest expected improvement is labeled with a purple star. This is the case where the hierarchical processes are known; however, that is not the case when it comes to hyper-parameter optimization since the distribution is not known [12]. Therefore, to determine which observations are good or bad, it uses the parzen-window density estimation. This density estimation is a way to build a probability density function from the observations that were generated [31] shown in Equation 2.12.

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right) \quad (2.14)$$

The main idea in Equation 2.12 is that in order to approximate an unknown function, f , a mixture of distributions, K , is used that are centered at x_i observations which are scaled by h . The constant h can be set by the user. Each sample that is generated defines a Gaussian distribution, K , with a unique mean (value of the hyper-parameter) and standard deviation. Then these distributions are stacked, in other words summed, together and normalized to get a probability density distribution [12]. The set of hyper-parameters with the highest EI is used for the next iteration. This concludes how the TPE algorithm is performed for hyper-parameter optimization.

2.5 Kiefer-Wolfowitz

The Kiefer-Wolfowitz is a stochastic approximation which, in essence, is a stepping algorithm that improves a function subjected to noise [32]. The purpose of this algorithm, in terms of this research, is to improve a continuous hyper-parameter to determine if it improves network performance. The optimization algorithms used in this research will always find a value that is close to the local minimum or maximum but never the global minimum or maximum. Therefore, Kiefer-Wolfowitz is used to try and get the current solution closer to the local minimum or maximum. Equation 2.15 shows the Kiefer-Wolfowitz approximation.

$$x_{n+1} = x_n + a_n \left(\frac{N(x_n + c_n) - N(x_n - c_n)}{2c_n} \right) \quad (2.15)$$

In Equation 2.15, x is the value of the learning rate at iteration n and $N(x)$ is the score of the convolutional neural network. The constants a_n and c_n are how the approximation steps closer to the local minimum or maximum where $a_n = \frac{1}{n}$ and

$$c_n = n^{-1/3}.$$

2.6 Summary

Hyper-parameter optimization is a crucial step when building a convolution neural network since they dictate the accuracy and performance of the network. Depending on how large the architecture of the network is as well as the spatial dimensions of the hyper-parameters, tuning hyper-parameters manually can be extremely inefficient and time constraining. To autonomously search the hyper-parameter space, three optimization techniques have been discussed: random search, BO-GP, and TPE. These three algorithms have been shown to find better results in other artificial neural networks, but how do they compare when applied to a convolutional neural network. This concludes Chapter 2, Chapter 3 will discuss on the methodology approach to implementing these three algorithms to a CNN.

III. Research Methodology

3.1 Overview

Chapter 3, Research Methodology, will discuss the development, software, and metrics that were implemented to conduct hyper-parameter optimization. The base software package that was used is package management environment called Anaconda. Through Anaconda, a convolutional neural network (CNN) was built through a software packaged called Spyder. Spyder is an open-source cross-platform integrated developmental environment (IDE) for scientific programming language called Python. Python is a high-level, general purpose computer language and its applications is almost endless. Just some of the domains that Python can be applied to are web and internet development, scientific computing, software, and graphical user interface development. To conduct this research, two application programming interfaces (API) were loaded into the Anaconda environment and ran through Spyder. These APIs are known as TensorFlow and Keras.

3.2 TensorFlow

TensorFlow, created and maintained by Google, is an open source software library for high performance numerical computation using data flow diagrams. Its flexible architecture allows easy deployment of computation across a variety of platforms and can be used on any type of operation system such as Linux and Windows [33]. TensorFlow allows users to create dataflow graphs that visualizes how data moves through a network or model through a series of processing nodes. These nodes in the graph represents a mathematical operation, and the connections between each node is a multidimensional array. These connections are called tensors. TensorFlow is a widely used API for machine learning problems and its ability to efficiently

utilize limited computation resources makes the application if it powerful. Because of TensorFlow's popularity, there are several unofficial extension packages (APIs) that increase its functionality. One such extension used in this research is called the Keras. Keras is used to build the architecture of the CNN and calls TensorFlow where data and analysis is stored. By doing so, this allow the user the access to the TensorBoard tool from TensorFlow. TensorBoard is a visualization tool to not only visualize scalars such as validation accuracy and loss, but also display a graph of the trained model [34].

3.3 Keras

Keras is a high-leveled neural networks API, written in python and ran in conjunction with TensorFlow. The goal of Keras is to allow a user to go from an idea to generating a result as fast as possible which is key to conduct an efficient and solid research [34]. There are two ways to write and save models into TensorFlow from Keras: sequential modeling or functional modeling. This research uses sequential modeling which means that the layers of the CNN are stacked sequentially in order of the way it was written. Keras is extremely easy to implement because each layer in the network can be represented with one line of code as opposed to writing a network block of code for a layer in a conventional way. The basic sequential model architecture in Keras has four components: model definition, compilation of model, model fitting, and performing predictions.

Before hyper-parameter optimization can be implemented, a base CNN was created and evaluated. As mention in Chapter 2, images are used as the input to the base CNN which has two convolution layers that are connected to a fully connected layer. The architecture of the network is shown in Figure 22.

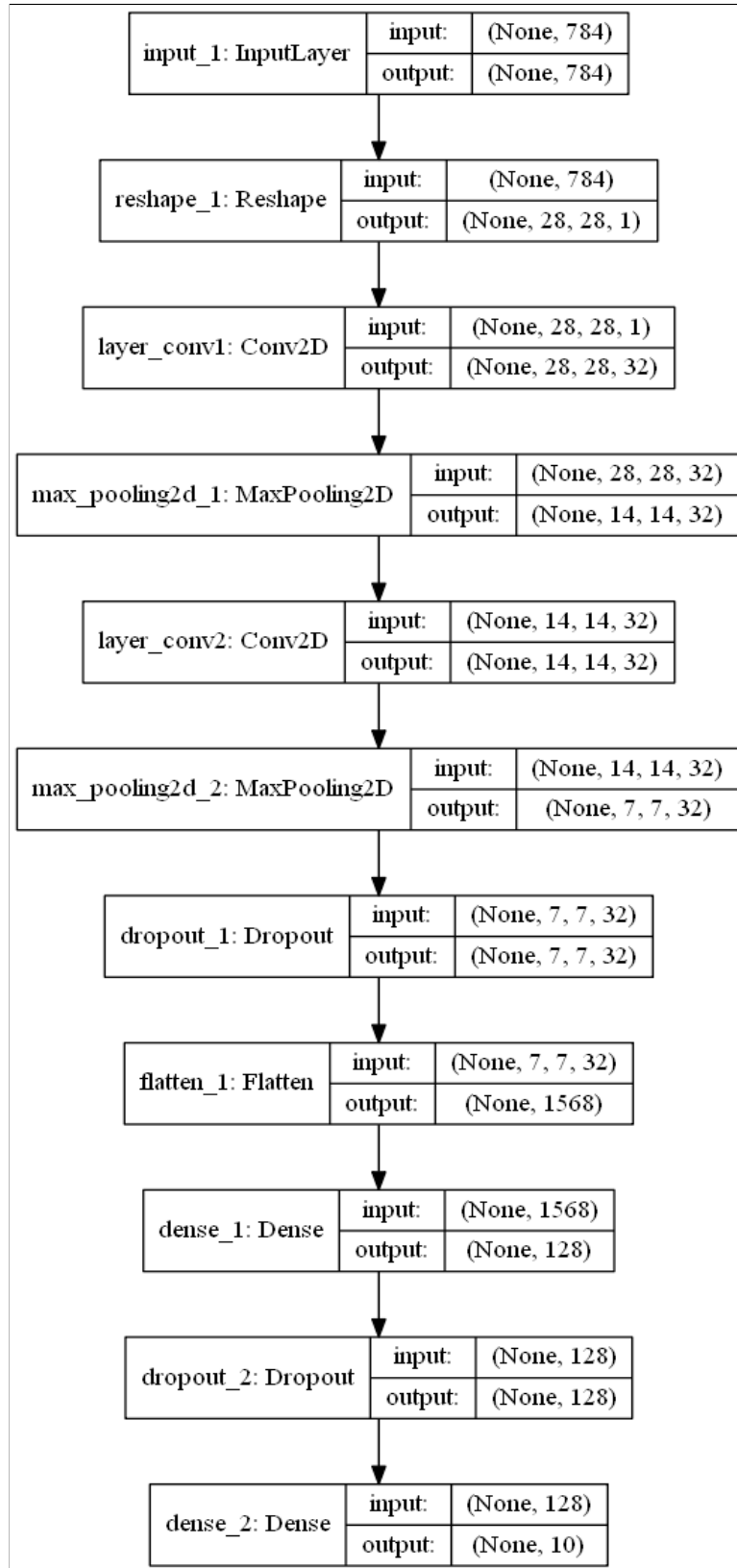


Figure 22. Architecture of Base CNN via Keras

Figure 22 shows the complete architecture from top to bottom. The input layer takes a raw image which contains 784 pixels (28x28) and reshapes the image into three dimensions that is required for a CNN. The reshaped image is fed into the first convolutional layer with a kernel size of 32. As expected, after max pooling, the width and height dimension is reduced, but the filter size remains the same. The image is then fed into the second convolutional layer and then pooled again. The output from the last max pooling layer is then flattened into one dimensional vector with size 1,568. Finally, the image is sent to the fully connected layer with the last layer classifying the image. A drop out rate is applied to both the convolutional network and the fully connected network. The "None" attribute in Figure 23 simply means that the layer can accept an input of any size. This network is ran with three epochs, which is a hyper-parameter, but is set at a fixed value. Epoch is the total number of times the network "sees" image and learn its patterns (tune weights) from it. Epoch directly affects model performance, but it significantly increases the computational time and resources required. Because of this, each model that is built with a different optimization approach is trained with three epochs. To view this CNN's architecture in TensorBoard, refer to Figure 23.

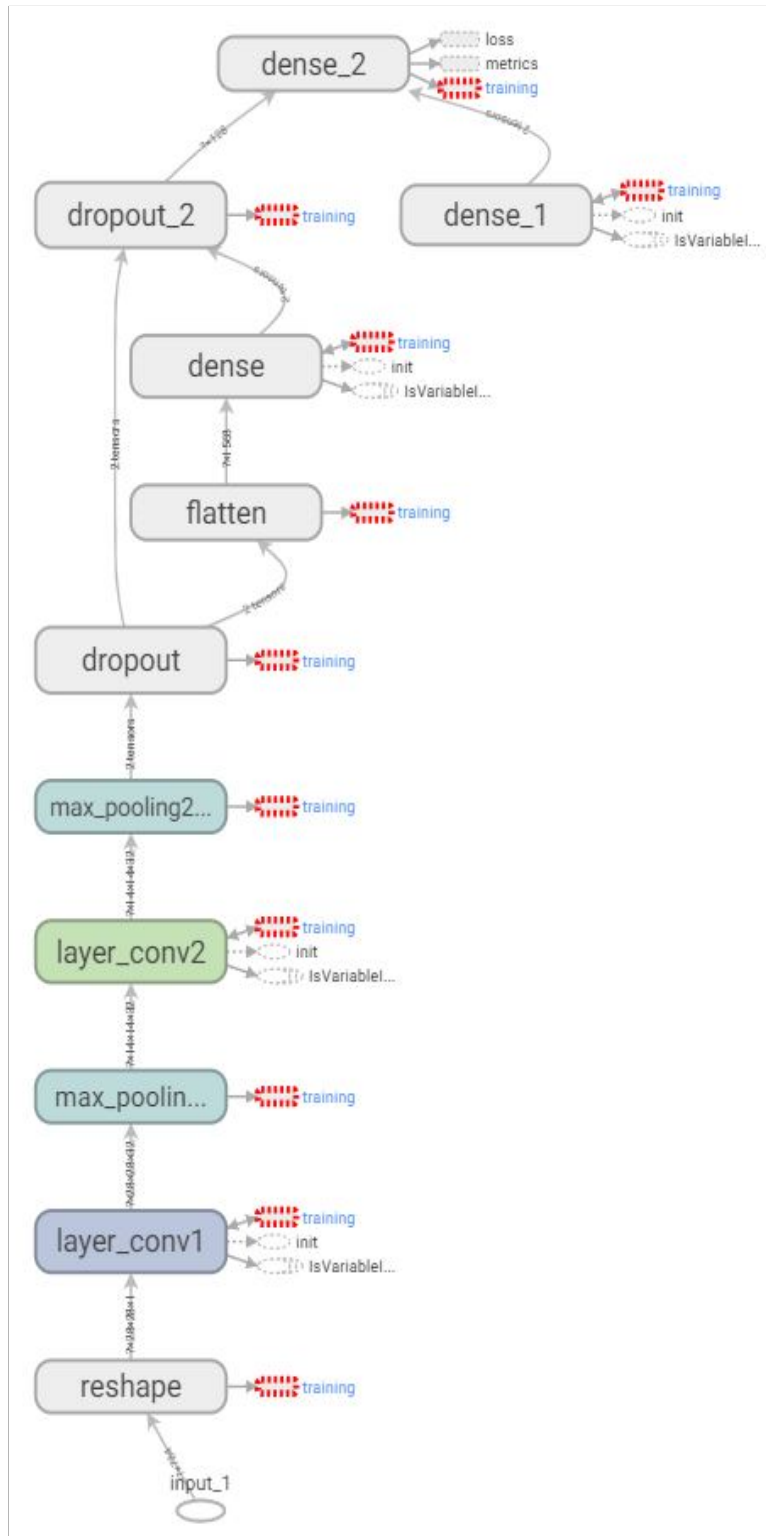


Figure 23. Architecture of Base CNN via TensorBoard

The base CNN will be unchanged throughout the process of applying different hyper-parameter optimization techniques. The data that was used for the network is explained in the next section.

3.4 Data Set

The data set that was used is from the Mixed National Institute of Standard and Technology (MNIST) [35]. The MNIST data set is composed of hand-written digital images. Each observation of the data set comes in pair: hand-written digital image and label. The digits range from 0 to 9 that are gray-scaled with size 28x28 pixels. The label is the actual number a digital image represents, also 0 to 9. Figure 24 shows what the nine samples that from the training set.

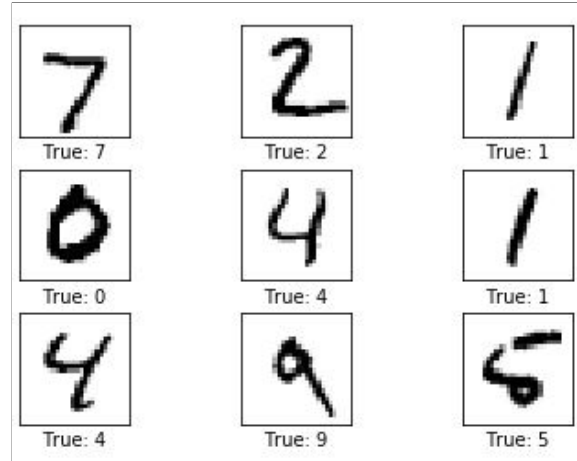


Figure 24. Nine Examples from MNIST Database

The data set is already partitioned out to a training set, test set, and validation set with a size of 55,000, 10,000, and 5,000, respectively. The model will be trained by the training set and tested on the test set to get an accuracy. Then the validation was used as an independent data set to determine model accuracy to new observations outside the training process. The model validation accuracy is used to determine

the next set of hyper-parameters either Gaussian process or tree of parzen estimator will use. Before each epoch, the training-set is shuffled again so the model does not become dependent on the order of the training set. The MNIST data set was chosen because it is a widely used image database used to train new CNN that are built by users all around the world.

3.5 Limitations

The limitations for this research mainly depends on computation resources and power. Due to this limitation, the hyper-parameter space is limited and only three epochs for each hyper-parameter set that is trained. Table 2 shows the hyper-parameter dimensions again.

Table 2. Hyper-parameter Dimensions

Hyper-parameter	Lower Bound	Upper Bound	Base
Learning Rate	0.0001	0.1	0.01
Number of Dense Layers	1	5	2
Number of Dense Nodes	5	512	128
Batch Size	16	512	200

The number of dense layers and the number of dense nodes are capped 5 and 512, respectively. These numbers are small relatively to other networks that can contain hundreds of layers and thousands of nodes. As mentioned earlier, the number of epochs used for the training process is fixed for each set of hyper-parameters chosen because it directly affects the time required to train each network. If the number was epochs was chosen as a hyper-parameter, it will indeed increase model performance. However, coupled with varying sizes of layers, nodes, and batch sizes, it could take hours for a single iteration to complete before an optimization algorithm chooses the next set of hyper-parameters to use.

3.6 Applying Hyper-parameter Optimization Techniques

The hyper-parameter optimization techniques discussed in Chapter 2 was coded through API packages in Python. The package `scikit-optimize`, `skopt`, and `hyperopt` were used to implement random search, Bayesian Optimization with Gaussian process, and tree of parzen estimator, respectively [36], the base CNN neural network that was coded is used as a function of each API. Once set up is complete and the algorithm is ran, the output of each API contains all the hyper-parameters that were search as well as model accuracy and validation accuracy.

3.7 Metrics

The metrics chosen to compare optimization algorithms is validation accuracy, validation loss, and total time. The Gaussian process, in simple terms, uses validation accuracy as the metric to determine the next set of hyper-parameters to use. The tree of parzen estimator also uses validation accuracy in terms of expected improvement to determine where to sample the next point. Validation accuracy is a value from 0 to 1, where the latter refers to the model classifying the model validation set with 100% accuracy. Validation loss is not a percentage, it is the summation of the errors that were made for each image. The validation loss value implies how well or poorly a certain model behaves after each iteration of optimization. Hence, one would expect validation loss to reduce after each epoch.

These metrics are important when it comes to comparing different optimization techniques; however, metrics not relating to optimization but model performance are also necessary to view model performance. The metrics that are important to compare to model performance are as follows [33]:

1. True Positive (TP) is an outcome where the model correctly predicts the true

- class of an image. Images is a 0, classifies it as a 0.
2. True Negative (TN) is an outcome where the model correctly predicts the wrong class of an image. Image is not a 0, classifies it as not a 0.
 3. False Positive (FP) is an outcome where the model incorrectly predicts the true class of an image. Image is a 0, classifies is not a 0, but correct label is an 0.
 4. False Negative (FN) is an outcome where the model incorrectly predicts the wrong class of an image. Image is not a 0, classifies it as a 0, but correct label is a 0.
 5. Precision (PREC) is the ratio of images that are correctly identified as positive amongst the returned images. PREC is how good the model is at identifying correct images. $PREC = TP / (TP + FN)$.
 6. Specificity (SPEC) is the ratio of images that are correctly identified as negative amongst the returned images. SPEC is how good the model is at avoiding false positives, in other words, correctly identifying wrong images. $SPEC = TN / (TN + FP)$

3.8 Summary

This chapter outline what software environments as well as APIs were used to conduct hyper-parameter optimization. Keras was the main API used that is integrated with TensorFlow to not only help visual the architecture of the CNN that was constructed, but also to build the neural efficiently with as few lines of code as possible. The data set used to conduct this research stems from the MNIST database. Because there are limit computation resources, only three epochs are use as well as

as limited hyper-parameter space. This concludes Chapter 3, Chapter 4 discusses the results and analysis of this research.

IV. Results and Analysis

4.1 Introduction

This chapter discusses the results after implementing the methodology in Chapter 3. The focus of this analysis is to see if hyper-parameter optimization not only improves the base convolutional neural network (CNN), but also which algorithm performs better. As mentioned before in Chapter 3, the metrics used to compare each model is the total time of each algorithm as model validation accuracy and loss. However, there are metrics, also mentioned in Chapter 3, that do not pertain to model optimization, but are still important in regards to how well the model performs to classify new data. It is expected the random search finds a set of hyper-parameters that increases the base CNN's performance. It is also expected that either Bayesian Optimization with Gaussian process or tree of parzen estimator performs better than random search.

4.2 Base CNN

A base CNN was first created and analyzed in order not only to examine which optimization method performs better, but also to analyze a model's ability to classify new images introduced to a model. As mentioned before, each model tunes its weights on three epochs. The model tunes its weights by the gradient of the categorical cross-entropy loss function. During the model training process, a batch of 200 images are fed into the network at a time to update the weights of the connections until a total of 55,000 images are processed by the network. This indicates the end of one epoch. At the end of each epoch, the current models loss and accuracy are evaluated on the test set. After each epoch is completed, the data is reshuffled and the model is updated again. Table 3 shows the metrics for each epoch run. Figure 25 is a graph showing

model performance over each epoch.

Table 3. Base CNN Model Metrics

Epoch	Run Time (sec)	Model Loss	Model Accuracy
1	42	1.6963	0.7309
2	47	0.5688	0.8539
3	48	0.3907	0.8980

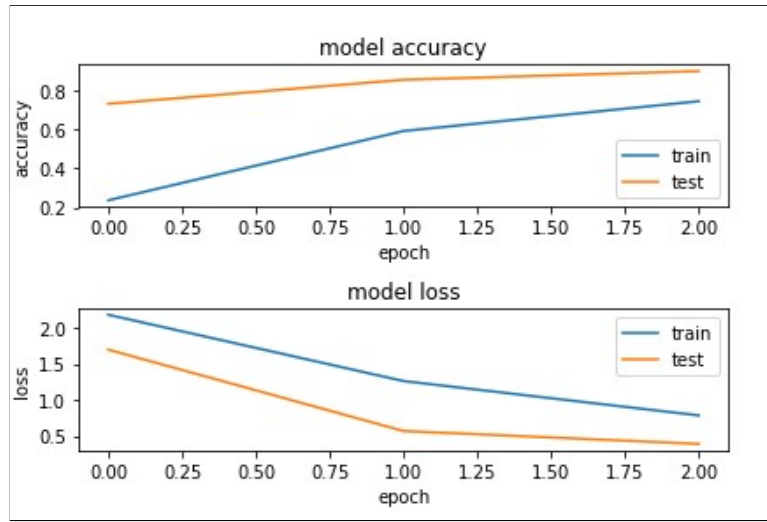


Figure 25. Base CNN Model Metrics

From Table 3 and Figure 25, model validation loss and accuracy improves after every epoch. The total run time for after three epochs was 137 seconds resulting in a model classification accuracy of 89.8%. The final model was then used to classify new digital images from the validation data set. The result is shown in Table 4.

Table 4. Base CNN Model Prediction on New Data Set

Validation Loss	0.4018
Validation Accuracy	0.8918

Overall, the model's ability to classify new images is about the same as its accuracy rate for the test set. Next the model's ability to classify all digit classes (0-9) is displayed through a confusion matrix in Figure 26. The confusion matrix is used to calculate model performance based on the metrics described in Chapter 3 which are the true positive (TP), true negative (TN), false positive (FP), false negative (FN), precision (PREC), and specificity (SPEC). PREC and SPEC are shown in Table 6.

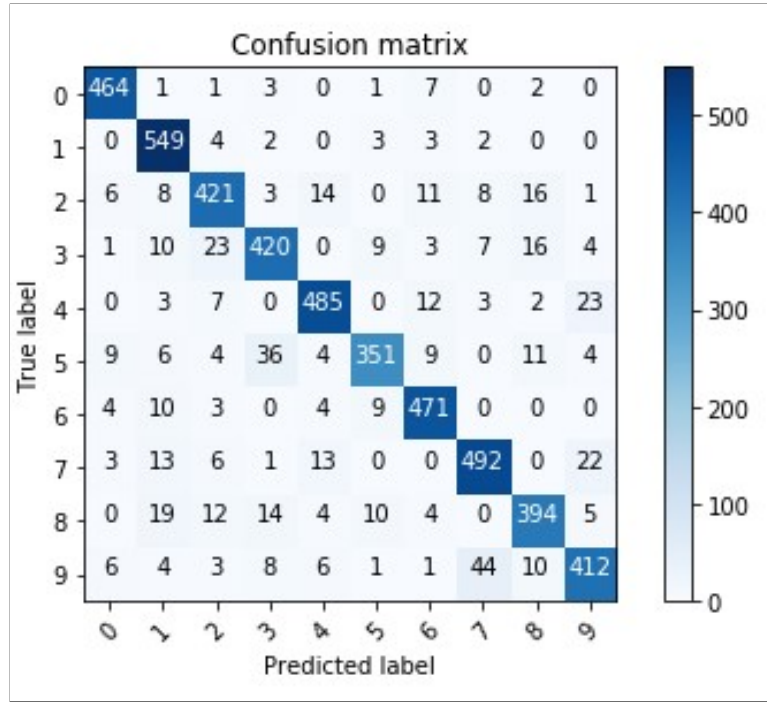


Figure 26. Base CNN Confusion Matrix

Table 5. Base CNN Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
TP	464	549	421	420	485	351	471	492	394	412
FP	29	74	63	67	45	33	50	64	57	59
FN	15	14	67	73	50	83	30	58	68	83
TN	4492	4363	4449	4440	4420	4533	4449	4386	4481	4446

Table 6. Base CNN PREC and SPEC

PREC	0.8918
SPEC	0.9890

From Figure 26, TP for each number is just the diagonal entry from the confusion matrix for the respective number. FP for each number is calculated by the sum of their respective column minus the diagonal entry for that number. FN for each number is calculated by the sum of their respective row minus the diagonal entry for that number. TN is calculated by removing the corresponding row and column for a number and then summing all the other rows and columns. The base CNN's ability to classify new digital images boils down to the values of PREC and SPEC. This means out of 5,000 images, the model's ability to correctly classify new images is 89.18%. PREC, though often times is the same value as accuracy, is calculated differently. PREC is essentially the probability the model correctly identifies any class, when in reality image represents the correct classified number. For example, the model has a probability of 89.18% to correctly identifying a 7 as a 7, and not another class. As for SPEC, the models' ability to avoid false positives is 98.9%. Avoiding false positive is an important performance metric because it rates how well the model identifies

a wrong image with respect to any other image class. Now that a base model has been built and results have been analyzed, next is to examine the first optimization algorithm: Random Search.

4.3 Random Search

Random search was ran five times, each given a chance to search the hyper-parameter space, shown in Table 2 of Section 3.5, eleven times. Each set of hyper-parameters that the search randomly selected was ran with three epochs. To maintain a clean and concise report, only the best found set of hyper-parameters is shown. Table 7 shows the best set of hyper-parameters. Table 8 shows the total run for the algorithm as well as the model loss and accuracy on the test set. Table 9 shows model loss and accuracy on the validation set.

Table 7. Random Search Best Hyper-parameters

Hyper-parameter	Value
Learning Rate	0.01104
Number of Dense Layers	2
Number of Dense Nodes	466
Batch Size	74

Table 8. Random Search Model Metrics

Model Loss	0.1531
Model Accuracy	0.9494
Total Run Time	76.3 min

Table 9. Random Search Model Prediction on New Data Set

Validation Loss	0.1557
Validation Accuracy	0.951

As expected, the random search algorithm was able to find a set of hyper-parameters to improve model performance. Next the confusion matrix as well as its ability to classify new images are shown in Figure 27 and Tables 10 and 11.

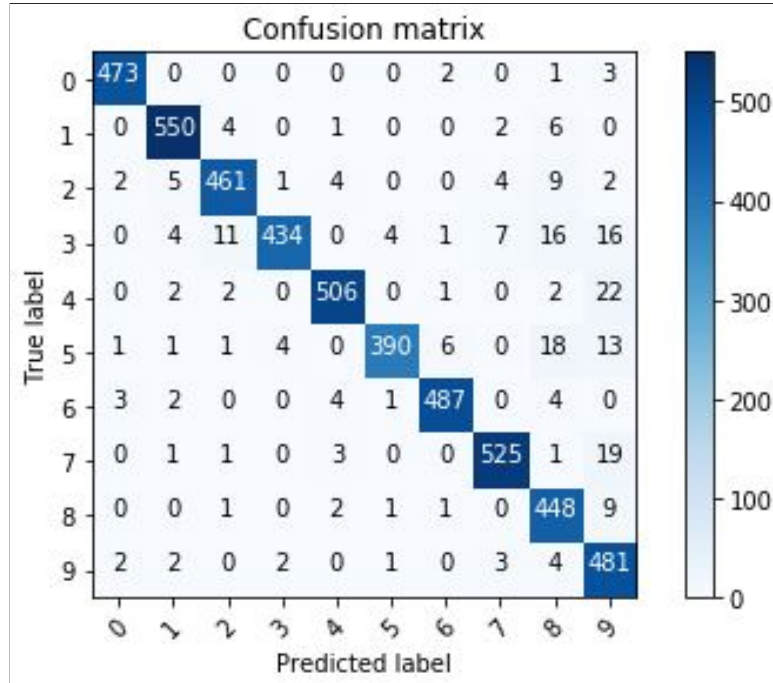


Figure 27. Random Search Confusion Matrix

Table 10. Random Search Model Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
TP	473	550	461	434	506	390	487	525	448	481
FP	8	17	20	7	14	7	11	16	61	84
FN	6	13	27	59	29	44	14	25	14	14
TN	4513	4420	4492	4500	4451	4559	4488	4434	4477	4421

Table 11. Random Search Model PREC and SPEC

PREC	0.9510
SPEC	0.9946

Overall, the random search algorithm was able to find a set of hyper-parameters that improves model accuracy and performance measures. Next the Bayesian Optimization with Gaussian process was conducted.

4.4 Bayesian Optimization with Gaussian Process (BO-GP)

BO-GP constructs a surrogate model (Gaussian process) for the search space of the hyper-parameter dimensions. This model gives the user an estimate of how the performance varies with changes to each hyper-parameter and will produce a new suggestion from a region of the search space that has not been explored yet. The new suggestion comes after the model accuracy is evaluated and reported back to the Gaussian process model, which is then determined on what BO-GP thinks will bring the most expected improvement. BO-GP was run five times, each given eleven chances to search the hyper-parameter space. The best hyper-parameter values that BO-GP

found is shown in Table 12. Table 13 shows the model metrics for this optimization approach. Table 14 shows model loss and accuracy on the validation set.

Table 12. BO-OP Best Hyper-parameters

Hyper-parameter	Value
Learning Rate	0.03691
Number of Dense Layers	2
Number of Dense Nodes	361
Batch Size	180

Table 13. BO-OP Model Metrics

Model Loss	0.0992
Model Accuracy	0.9696
Total Run Time	33.2 min

Table 14. Random Search Model Prediction on New Data Set

Validation Loss	0.1022
Validation Accuracy	0.9690

This approach was also given the chance to search the hyper-parameter space eleven times and each model was ran with three epochs. One key difference between the random search and BO-GP is the total run time. The BO-GP algorithm took over 50% less time to run than that of the random search and found a slightly better result. Therefore, if the amount of time available is limited, BO-GP will not only find

results that could potentially be better, but also will save a significant amount of time. However, these results are consistently not reproducible. When a new instance is ran, BO-OP could potentially not find a solution that will be better than the random search algorithm. The reason for this is every time the algorithm is performed, the surrogate model built for the search space may not be accurate. There are only eleven samples taken from the entire search space and the dimension "landscape" is not thoroughly examined. To visualize this, Figure 28 shows a partial dependence plot.

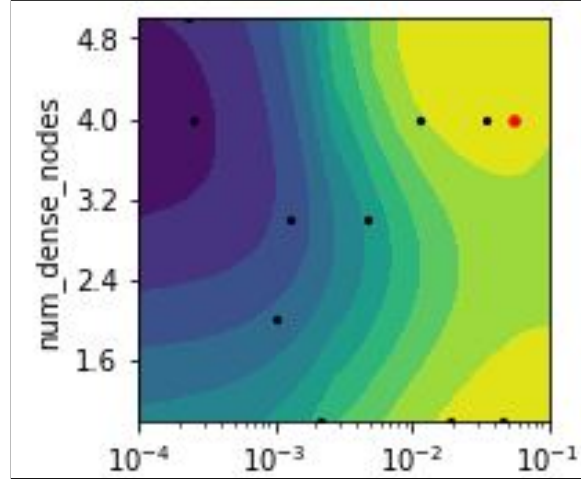


Figure 28. Partial Dependence Plot

Figure 29 shows a partial dependence plot between the number of dense nodes and the learning rate. A partial dependence plot which is a visualization technique used to view a high dimensional space in two dimensions. This plot shows the effect of changing these hyper-parameters when averaged over all the other hyper-parameters dimensional space. The plot is built from the last updated surrogate model of BO-GP where the areas shaded in yellow performed better than areas shaded in blue. Clearly, there are spaces of the landscape where the algorithm has not yet explored due to the fact it was only given a chance to search eleven different areas. Next, to see model performance for classifying a new data set, a confusion matrix is shown in Figure 29

as well as Table 15 and 16.

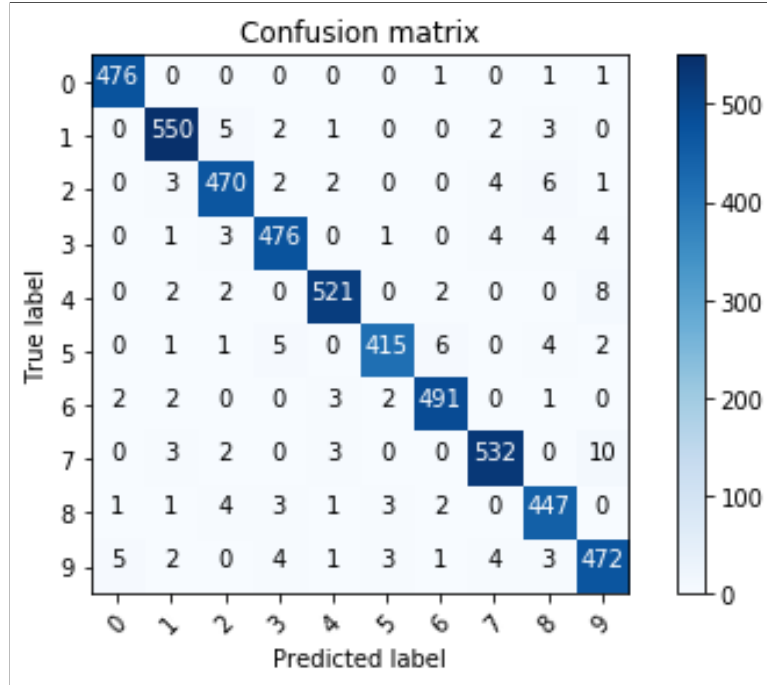


Figure 29. BO-GP Confusion Matrix

Table 15. BO-GP Model Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
TP	476	550	470	476	521	415	491	532	447	472
FP	8	15	17	16	11	9	12	14	22	26
FN	3	13	18	17	14	19	10	18	15	23
TN	4513	4422	4495	4491	4454	4557	4487	4436	4516	4479

Table 16. BO-GP PREC and SPEC

PREC	0.9690
SPEC	0.9967

Overall, even though this instance of the BO-GP run found a better hyper-parameter set than random search, the results are not entirely convincing. Not enough

samples are implemented to give the algorithm a chance to find adequately search the dimensional space of the hyper-parameters. Forty samples would probably not be enough, but increasing the number of samples will result in increasing the total time required to run the algorithm. Next the tree of parzen estimator was conducted.

4.5 Tree of Parzen Estimator (TPE)

The tree of parzen estimator (TPE), first performs a random search to build a distribution over the dimensional space of the hyper-parameters. TPE models the hyper-parameter space given the quality score produced by the random search and then by transforming the space into two different distributions. TPE was ran five times each given eleven times to select a set of hyper-parameters. The best hyper-parameter values that TPE found is shown in Table 17. Table 18 shows the model metrics for this optimization approach. Table 19 shows model loss and accuracy on the validation set.

Table 17. TPE Best Hyper-parameters

Hyper-parameter	Value
Learning Rate	0.02823
Number of Dense Layers	3
Number of Dense Nodes	461
Batch Size	17

Table 18. TPE Model Metrics

Model Loss	0.0392
Model Accuracy	0.9864
Total Run Time	31.8 min

Table 19. TPE Model Prediction on New Data Set

Validation Loss	0.0475
Validation Accuracy	0.9856

The TPE, just like the BO-OP algorithm, not only finds a better set of hyper-parameters, but also model metrics are slightly improved as well. Finally, model performance is shown with Tables 20 and 21 and visualized through a confusion matrix shown in Figure 30. Overall, these tables show that model performance was improved; however, this algorithm also had to be run multiple times in order to find results better than BO-OP. This result can be due to the random search portion before the algorithm is ran which creates a distribution that is not close to the true optimal hyper-parameter values.

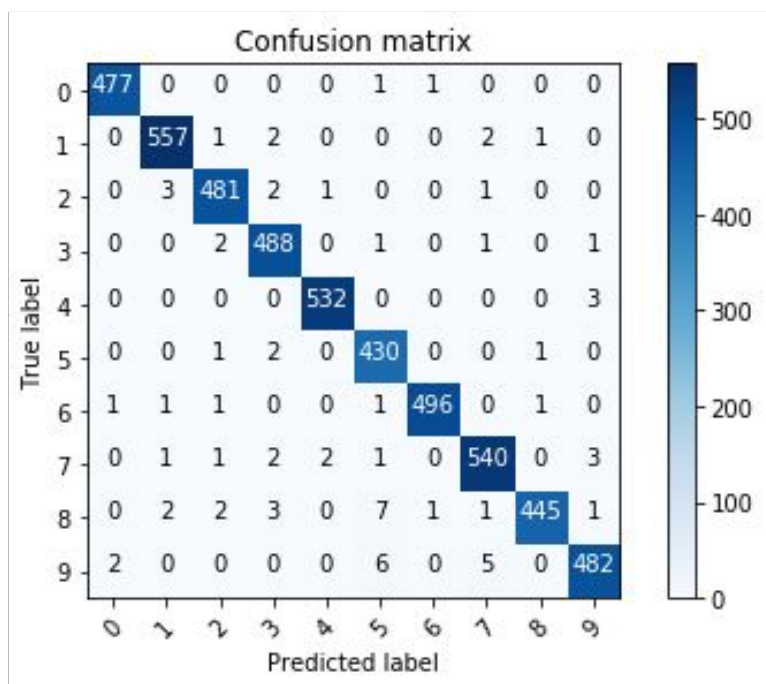


Figure 30. TPE Confusion Matrix

Table 20. TPE Model Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
TP	477	557	481	488	532	430	496	540	445	482
FP	3	7	8	11	3	17	2	10	3	8
FN	2	6	7	5	3	4	5	10	17	13
TN	4518	4430	4504	4496	4462	4549	4497	4440	4535	4497

Table 21. TPE PREC and SPEC

PREC	0.9856
SPEC	0.9984

4.6 Kiefer-Wolfowitz

Since the TPE approach found a set of hyper-parameters that resulted in the best model metrics and performance, the Kiefer-Wolfowitz approximation was applied to the learning rate of this set. The purpose of this approximation is to examine if it improves model metrics and performance after the implementation of the TPE approach. Figure 31 shows the different in learning rate as well as model metrics. Figure 32 shows the new confusion matrix after the stochastic approximation was applied. Table 22 is the confusion table of the confusion matrix.

	Hyper-parameter	
	Tree of Parzen Estimator	Tree of Parzen Estimator - Stochastic Approx
Learning Rate	0.02823	0.02864
Number of Dense Nodes	461	461
Number of Dense Layers	3	3
Batch Size	17	17

	Model Metrics and Performance	
	Tree of Parzen Estimator	Tree of Parzen Estimator - Stochastic Approx
Validation Loss	0.0475	0.0405
Validation Accuracy	0.9856	0.988
PREC	0.9856	0.988
SPEC	0.9954	0.997

Figure 31. TPE and TPE-Stochastic Approximation Comparison

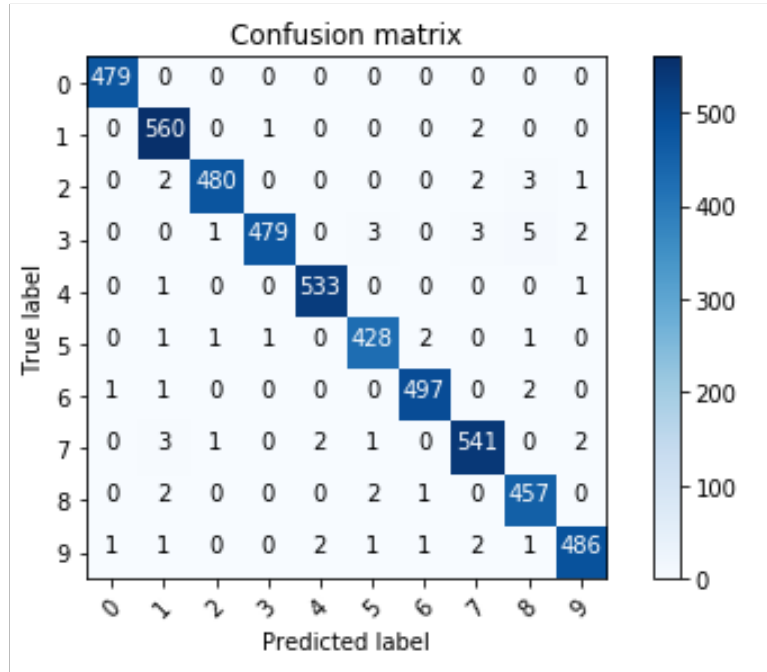


Figure 32. TPE-Stochastic Approximation Confusion Matrix

Table 22. Kiefer-Wolfowitz Confusion Table

Class	0	1	2	3	4	5	6	7	8	9
TP	479	560	480	479	533	428	497	541	457	486
FP	2	11	3	2	4	7	4	9	12	6
FN	0	3	8	14	2	6	4	9	5	9
TN	4519	4426	4509	4505	4461	4559	4495	4441	4526	4499

The results show that overall model metrics and performance was improved. Precision and specificity were increased by .24% and .16%, respectively. However, even though the changes in performance are small, the number of false positives and false negatives produced was decreased by 12%.

4.7 Summary

This concludes the result and analysis section. As mention before, random search was expected to find a set of hyper-parameters that will increase model metrics and performance. Both BO-GP and TPE were expected to find a better set of hyper-parameters to produce a CNN that will perform better than the CNN from random search. However, there were draw backs in using these methods. Both of these methods did not always find a better solution than that of the random search. Even though the total run time for both BO-GP and TPE was significantly less than the total run time of random search, random search produced consistent results.

The reason for inconsistent results could be due to the surrogate model drawn for BO-GP and the distribution built over the hyper-parameter space for TPE. Another reason for the inconsistent results is the number of samples BO-GP and TPE were allowed to draw from the search space. If a vast amount of computational resources were available, these algorithms could be run in parallel allowing significantly more samples to be drawn. Although, a user may run these algorithms once and find better results on the first try. In conclusion, BO-GP and TPE were both able to find hyper-parameter values that improved a simple CNN.

As for model performance, even though precision and specification were high, all the models had issues with false positives and false negatives relative to each class for each algorithm. However, after applying the Kiefer-Wolfowitz approximation after the TPE approach, not only improved the model metric, but also false positives and false negatives were reduced by 12%.

V. Conclusions and Future Research

5.1 Overview

This chapter provides further commentary on the result presented in this research. It also further justifies that hyper-parameter optimization should be a crucial step when building any kind of neural network. The base model, a convolutional neural network(CNN), was used to measure algorithm performance on the basis if it improved model metrics and performance. The hyper-parameters that were chosen were the learning rate, number of dense layers, number of dense nodes, and batch size. However, although obviously affecting model metrics and performance, maybe a different set or additional hyper-parameters should have been considered which is discussed in the Future Considerations section. The algorithms that were applied to search the hyper-space space and build a new model best found set were random search, Bayesian Optimization with Gaussian process (BO-GP), and tree of trazen estimator (TPE) approaches. This chapter talks about the implications of each algorithm, future research, and some final thoughts of conducting hyper-parameter optimization.

5.2 Implications

The analysis presented in Chapter 4 showed that each optimization technique chosen improved model metrics and performance. The best algorithm implemented was the TPE approach which showed that the validation accuracy increased from the base CNN to the TPE by 9.38%. The change in the validation accuracy is significant when even a one percent change can influence decisions from a data set containing hundreds of thousands of images.

Each optimization method chosen can be costly, as shown by random search. The algorithm took over an hour to complete and was the least accurate among the other

algorithms. However, although the slowest, is the most consistent when it came to finding a set of hyper-parameters that improved model performance. On the other hand, BO-GP and TPE did not always produce the expected results. Each algorithm was ran a considerable amount of time that could indicate random search may be better in terms of efficiency and performance. The reason for these inconsistent results is the algorithms themselves. BO-GP builds a surrogate model based on the search space and optimizes this model as opposed to the model involving the real search space. This may produce a model where the landscape of the real search space is not accurately represented resulting in a poor performance rate. Also, the number of samples BO-GP was allowed to search for may not be enough for the algorithm to grasp the landscape of the space in order to draw better results. As for TPE, his algorithm builds a distribution based on a limited set of randomly selected points from the search space before performing optimization hyper-parameters. This also will produce inconsistent results because again, the search space may not be accurately represented. Therefore, if there is a limited amount of time and computation resources, random search may be better to get sub-optimal hyper-parameters, but will suffer a loss in model performance to classify. If there is ample of time to run BO-GP or TPE for hours, then TPE would be the prefer method. The Kiefer-Wolfowitz approximation was implemented to the best algorithm that resulted in the best model metrics and performance. Kiefer-Wolfowitz did increase these network aspects and reduced the number of false positives and false negatives reported by a factor of 12%.

5.3 Future Considerations

During this investigation, the four hyper-parameters were considered clearly showed their ability to affect model metrics and performance. The most important hyper-parameter that significantly affected the CNN training process was the learning rate.

However, what about model architecture? A CNN architecture has two components: convolutional layers and fully connected network. The hyper-parameters chosen mainly affect the architecture of the fully connected layer and not the convolutional layers. Therefore, a future consideration would see how selecting the kernel size, number of convolutional layers, and different types of activation functions affects model metrics, performance, and time to train.

Another future consider is to use a entire new data series to validate the model created from the hyper-parameter optimization. For example, apply it to facial images, fictional characters, or maps to see if the hyper-parameter values hold true. The structure of the network would have to change slightly depending on the structure of the data itself. Also another consideration could be to add more hyper-parameters to the set that was tested in this research as well as increasing the hyper-parameter dimensions. Some hyper-parameters that could be considered are the number of convolution layers, size of the kernel or filters, number of kernels of filters, number of epochs, activation functions, or different optimizers other than the stochastic gradient descent.

5.4 Final Summary

This study on three different hyper-parameter optimization techniques on a simple two layer convolutional neural network proved that automated search procedures are valuable. Manually tuning hyper-parameters is not only inefficient, but also producing better results than an autonomous algorithm requires years of experience. The TPE algorithm shows to be the best algorithm amongst the other two. Although TPE inconsistencies may be based on limited resources, it would be beneficial to implement as a model training step. TPE will improve any neural network that is created, but must be given a significant time to perform. This study also shows that stochastic

approximation such as the Kiefer-Wolfowitz approximation can improvement model metrics and performance.

Bibliography

1. D. T. Larose and C. D. Larose, "Data Mining and Predictive Analytics," *Wiley Series on Methods and Applications in Data Mining*, vol. 25, no. 9, pp. 1682–1690, 2015.
2. A. Minnaar, "Deep Learning Basics: Neural Networks, Backpropagation and Stochastic Gradient Descent," 2015. Available: <http://alexminnaar.com/deep-learning-basics-neural-networks-backpropagation-and-stochastic-gradient-descent.html>, [Accessed: 01 - Jan - 2019].
3. S. Srivastava, Nitish; Geoffrey, Hinton; Alex, Krizhevsky; Ilya, Sutskever; Ruslan, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, 2014.
4. P. Kar, "CS231n Convolutional Neural Networks for Visual Recognition." Available: <http://cs231n.github.io/convolutional-networks/>, [Accessed: 03 - Jan - 2019].
5. P. Shyamal, Patel; Johanna, "Introduction to Deep Learning: What Are Convolutional Neural Networks?," 2017. Available: <https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html>, [Accessed: 03 - Jan - 2019].
6. A. Dertat, "Applied Deep Learning - Part 4: Convolutional Neural Networks," 2017. Available: <https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>, [Accessed: 01 - Jan - 2019].
7. D. Cornelisse, "An intuitive guide to Convolutional Neural Networks," 2018. Available: <https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050>, [Accessed: 04-Jan-2019].
8. SuperDataScienceTeam, "Convolutional Neural Networks (CNN): Step 3-Flattening," 2018. Available: <https://www.superdatascience.com/blogs/convolutional-neural-network-cnn-step-3-flattening>, [Accessed: 02-Jan-2019].
9. M. Buzau, "Gradient descent on a Softmax cross-entropy cost function," 2016. Available: <https://madalinabuzau.github.io/2016/11/29/gradient-descent-on-a-softmax-cross-entropy-cost-function.html>, [Accessed: 03-Jan-2019].
10. J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

11. M. Krasser, "Gaussian Process," 2018. Available: <http://krasserm.github.io/2018/03/19/gaussian-processes/>, [Accessed: 02-Jan-2019].
12. Y. Shevchuk, "Hyperparameter optimization for Neural Networks," 2018. Available: http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html, [Accessed: 01-Jan-2019].
13. M. S. B. Maind and M. P. Wankar, "Research Paper on Basic of Artificial Neural Network," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 2, no. 1, pp. 96–100, 2014.
14. A. Kurenkov, "A 'Brief' History of Neural Nets and Deep Learning," 2019. Available: <http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/>, [Accessed: [04-Jan-2019].
15. A. Hossain, M. Rahman, K. Prodhan, and F. Khan, "Implementation of back-propagation neural Network for isolated bangla speech recognition," *International Journal of Information Sciences and Techniques*, vol. 3, no. 4, pp. 1–9, 2013.
16. R. Rumelhart, David; Hinton, Geoffrey; Williams, "Learning representations by back-propagating errors," *Institute for Cognitive Science*, 1986.
17. F. Plume, M. Program, and A. Mapping, "The Backpropagation Algorithm," *Neural Networks-Springer-Verlag*, no. 1991, pp. 25–28, 1999.
18. Prabhu, "Understanding Hyperparameters and its Optimisation techniques," 2019. Available: <https://towardsdatascience.com/understanding-hyperparameters-and-its-optimisation-techniques-f0debba07568>, [Accessed: 01-Jan-2019].
19. K. Gurney, *An Introduction to Neural Networks*. CRC Press, 1st editio ed., 1997.
20. F. S. Panchal and M. Panchal, "International Journal of Computer Science and Mobile Computing Review on Methods of Selecting Number of Hidden Nodes in Artificial Neural Network," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 11, pp. 455–464, 2014.
21. A. Krizhevsky, I. Sutskever, and G. E. Hinton., "Imagenet classification with deep convolutional neural networks. In Advances in neural information," *Advances in neural information processing systems*, pp. 1097–1105, 2012.
22. P. Sadowski, "Notes on Backpropagation," *University of California Irvine*, no. 4, pp. 1–3.
23. T. Hinz, S. Magg, and S. Wermter, "Speeding up the Hyperparameter Optimization of Deep Convolutional Neural Networks," *International Journal of Computational Intelligence and Application*, no. 2, pp. 1–15.

24. P. M. Radiuk, "Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets," *Information Technology and Management Science*, no. December, pp. 20–24.
25. Z. B. Zabinsky, "Random Search Algorithms," *Wiley Encyclopedia of Operations Research and Management Science*, 2011.
26. R. B. Gramacy, "Bayesian Treed Gaussian Process Models," *University of California, Santa Cruz*, p. 150, 2005.
27. C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*, vol. 14. 2004.
28. P. I. Frazier, "A Tutorial on Bayesian Optimization," no. Section 5, pp. 1–22, 2018.
29. S. Dewancker, Ian;McCourt, Michael; Clark, "Bayesian Optimization Primer," *SIGOPT*, 2017.
30. J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, J. Shawe-taylor, R. S. Zemel, P. Bartlett, F. Pereira, K. Q. W. Annual, and J. Bergstra, "Algorithms for Hyper-Parameter Optimization," *Neural Information Processing Systems Foundation*, 2011.
31. O. S. Steinholtz, "A Comparative Study of Black-box Optimization Algorithms for Tuning of Hyper-parameters in Deep Neural Networks," *Lulea University of Technology, Department of Computer Science, Electrical and Space Engineering*, 2018.
32. S. Kullback and R. A. Leibler, "Stochastic Estimation of the Maximum of a Regression Function," *Ann. Math. Statist. Volume 22, Number 1 79-86.*, vol. 37, no. 3, pp. 688–697, 1951.
33. K. Pogorelov, K. R. Randel, T. D. Lange, D. Johansen, M. Lux, P. T. Schmidt, and M. Riegler, "Multi-Class Image Dataset for Computer Aided Gastrointestinal Kvasir," *KVASIR*, 2017.
34. R. J. Borgli, "Hyperparameter optimization using Bayesian optimization on transfer learning for medical image classification," *Faculty of Mathematics and Natural Sciences*, 2018.
35. C. Lecun, Yann; Cortes, Corinna; Burges, "THE MNIST DATABASE of handwritten digits," 1998. Available: <http://yann.lecun.com/exdb/mnist/>, [Accessed: 01-Jan-2019].
36. J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox, "Hyperopt: A Python library for model selection and hyperparameter optimization," *Computational Science and Discovery*, vol. 8, no. 1, 2015.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE (DD-MM-YYYY) 21-03-2019		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) Sept 2017 — Mar 2019	
4. TITLE AND SUBTITLE Hyper-parameter Optimization of a Convolutional Neural Network				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Steven H. Chon, Captain, USAF				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Operational Sciences (AFIT/ENS) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENS-MS-19-M-105	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Trevor Bihl, DAF, PhD Sensors Directorate Air Force Research Laboratory 2242 Avionics Circle Wright-Patterson AFB, OH 45431				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RYPAR	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT Distribution Statement A: Approved for Public Release; distribution unlimited.					
13. SUPPLEMENTARY NOTES This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.					
14. ABSTRACT In the world of machine learning, neural networks have become a powerful pattern recognition technique that gives a user the ability to interpret high-dimensional data where as conventional methods, such as logistic regression, would fail. There exists different types of neural networks, each containing its own set of hyper-parameters, that are dependent on the type of analysis required, but the focus of this paper will be on the hyper-parameters of convolutional neural networks. Convolutional neural networks are commonly used for classifications of visual imagery. For example, if you were to build a network for the purpose of predicting a specific animal, it would hopefully output, with high fidelity, the correct classification of a new animal introduced to the model. Traditionally, hyper-parameters were never optimized because it required a lot of computational power and time. If hyper-parameters were adjusted, analysts would manually change a few hyper-parameters, re-run the model, and hopefully get a better classification accuracy. But because of the advancements in technology, hyper-parameter tuning can now be done through complex and powerful optimization algorithms to improve the model. This paper implements and compares three different optimization techniques: random search, Bayesian Optimization with Gaussian Process, and tree of parzen estimator approach. The best performing technique is then improved through the Kiefer-Wolfowitz approximation.					
15. SUBJECT TERMS Term1; Term2; Term3					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 90	19a. NAME OF RESPONSIBLE PERSON Capt Steven H. Chon, AFIT/ENS
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (include area code) (937) 255-3636, x4624; steven.chon@afit.edu