Performance Enhancement of CKKS Encrypted Neural Networks During Training

by

Samuel Atkins

Supervisor: Prof. Glenn Gulak
April 2020

B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

Performance Enhancement of CKKS Encrypted Neural Networks During Training

by

Samuel Atkins

Supervisor: Prof. Glenn Gulak
April 2020

Abstract

Acknowledgements

Table of Contents

List of Figures

List of Tables

1. Introduction

In 2009, Gentry utilized lattice-based cryptography to construct the first fully homomorphic encryption (FHE) scheme [1]. Fully homomorphic encryption is named as such because it allows one to perform an arbitrary sequence of additions and multiplications on cipher texts. Given the security requirements of private data processing, FHE was a tremendous breakthrough for machine learning. A neural network modified to operate on homomorphically encrypted data is called a CryptoNet. CryptoNets are attractive because they allow a data owner to present their data, already encrypted, to a cloud service. This cloud service can then apply their CryptoNet to the encrypted data to provide encrypted inferences. These encrypted inferences can be sent back to the data owner to be decrypted. During this entire process, the cloud service remains unaware of the content of the data because the data owner maintains the private key.

In 2016, Microsoft Research developed a high-throughput CryptoNet capable of achieving 99% accuracy and 51,000 inferences per hour on a single PC [2]. This research trained their model using unencrypted data. Furthermore, while the inference throughput of the network is high, the network itself is not complex. More complex datasets require more complex networks. The affects of various structures, layer types, and parameter settings of an encrypted neural network will be investigated. Specifically, inference throughput and the time to train as a function of various model designs will be characterized. Moreover, the practically of CryptoNets will be investigated. This research is valuable because it increases our understanding of CryptoNets by providing insight into their practicality.

2. Literature Review

2.1 Encryption

2.1.1 Public-Key Cryptography

Public-key cryptography uses key-pairs to encrypt and decrypt data. Plaintext is unencrypted, readable text. Ciphertext, conversely, is encrypted text. To encrypt plaintext into ciphertext using public-key encryption, one first must generate a public key and a private key. The public key is used to encrypt plaintext into ciphertext, and the private key is used to decrypt the ciphertext back into plaintext. Public-key cryptography allows parties to communicate private information through public mediums. For a visualization of this process, see Appendix A.

2.1.2 Homomorphic Encryption

Homomorphic encryption is a form of encryption that supports arithmetic computation on ciphertexts. Using a homomorphic encryption scheme allows one to perform mathematical operations on ciphertexts. One can, using homomorphic encryption, create two ciphertexts, multiply these two ciphertexts together, and then decrypt the encrypted result to obtain the product of the two original plaintexts.

There are three different types of homomorphic encryption: partially homomorphic encryption (PHE), somewhat homomorphic encryption (SHE), and fully homomorphic encryption (FHE). PHE allows a smaller set of mathematical functions to be performed on ciphertexts an arbitrary number of times. SHE supports a limited number of operations that can only be performed a finite number of times. The most desirable of the three, FHE, allows one to perform both addition and multiplication on ciphertexts a semi-infinite number of times.

2.1.3 Fully Homomorphic Encryption

In 2009, Gentry introduced a limited SHE scheme [1]. The SHE scheme that he introduced was limited because after each multiplication operation between two ciphertexts, the noise budget corresponding to the result would grow. Ultimately, the noise budget would grow too large and render the decrypted result inaccurate. Gentry elucidates, however, that a modified

version of the proposed SHE scheme is *bootstrappable*. An encryption scheme is bootstrappable if the scheme can homomorphically evaluate its own decryption circuit plus one additional NAND gate. Gentry then illustrates that any SHE scheme that is bootstrappable can be recursively self-embedded to make it fully homomorphic. This discovery outlined the basis for the first FHE scheme.

The security of Gentry's FHE scheme was based on ideal lattices. The implementation of Gentry's lattice-based scheme showed that secure bootstrapping took up to 30 minutes per operation [3]. Following this implementation was an integer-based approach [4] that was simpler and more efficient than the previous lattice-based approach. Despite the efficiency improvement, the uses of homomorphic encryption were severely limited by computational efficiency bottlenecks due to the large number of mathematical operations involved. From 2011 to 2016, many variations of FHE schemes followed Gentry's breakthrough. These encryption schemes include the BGV [5], LTV [6], BFV [7] [8], BLLN [9], and CKKS [10] schemes.

The CKKS scheme is significantly faster than the other schemes because it utilizes a rescaling procedure. During rescaling, the ciphertext is truncated into a smaller modulus. As a result, the decrypted result is rounded and less accurate. The experimental models and parameter settings will be applied to the CKKS scheme.

2.2 Artificial Intelligence

2.2.1 Machine Learning Categories

There are two general categories of machine learning: supervised and unsupervised learning. Supervised learning requires labelled training data to improve the performance of a model through backpropagation. This research will focus on CryptoNets applied to supervised learning.

2.2.2 Neural Network Training and Forward Propagation

An artificial neural network, also known as a neural network, or multi-layer perceptron, is a circuit of neurons connected in a network. Neural networks are used to classify data and perform data driven tasks. Unlike traditional classification engines, neural networks learn from their input data to produce stronger results over time. The connections between the neurons in a neural network are known as the weights of the neural network. These weights are malleable and define the state of the network. As the neural network learns from its input data, the weights of the network change. The learning process in which the network learns from its input data is known as the training phase.

Prior to the training phase, the dataset must be split into three partitions: the training dataset, the validation dataset, and the testing, or holdout, dataset. Certain data splitting strategies lead to stronger generalization. The Kennard-Stone algorithm and SPXY systematic sampling method are two examples of algorithms that attempt to maximize the generalizability of the network through effective data splitting [11]. During training, we feed the data entries from the training dataset into the neural network. The neural network tries to infer the labels attached to the training data. It then uses the feedback from its correct and incorrect inferences to gradually update its inner parameters.

The structure of a neural network is one of the factors that dictates the post-training performance of a neural network. Figure 1 illustrates the structure of a simple neural network:

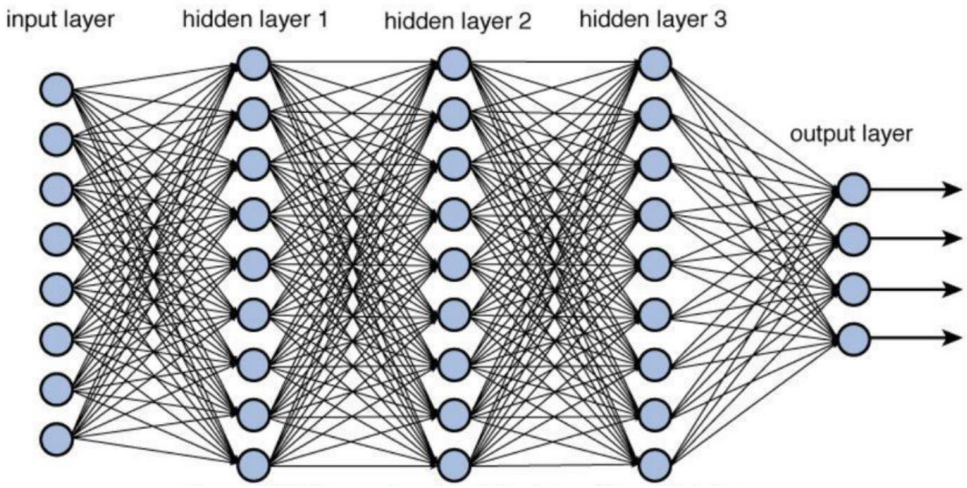

Figure 1: Structure of a multi-layer neural network [12]

The dimension of a layer in a neural network is defined by the number of nodes in that layer. The neural network in Figure 1 has an input layer of dimension 8, three hidden layers of dimension 9, and an output layer of dimension 4. The network above is called a *feed-forward, fully-connected* neural network. It is fully-connected because every node in each layer is connected to every node in the following layer. It is called a feed-forward network because during a process called *forward propagation*, the information propagates forward from the input layer to the next layer, and then to the subsequent layers that follow. An in-depth example that delineates the forward propagation process is detailed in Appendix B.

A crucial element that influences the predictive power of a neural network is the selected *activation function*. The activation function is applied to the output of every layer in a neural network. Activation functions introduce non-linearity into the weighted sums that define the connections between each layer. This non-linearity allows one to approximate "an arbitrary continuous function, defined on [0,1]" using "a multilayer feed-forward neural network with one hidden layer (that contains only finite number of neurons) using neurons with arbitrary activation functions in the hidden layer and a linear neuron in the output layer" [13] due to the Universal Approximation Theorem [13]. Examples of common activation functions are illustrated in Appendix C.

2.2.3 Backpropagation and Hyperparameter Tuning

Backpropagation is the process that complements forward propagation. During backpropagation, the neural network uses the results from each attempted inference to update its parameters. It uses a *loss function* to measure how well each prediction fares with respect to the ground truth label. The nature of the loss function is dictated by the task at hand. Each task requires a metric that successfully defines the quality of each prediction. The objective of the neural network is to minimize the defined loss function. Examples of typical binary classification and multi-class classification loss functions are detailed in Appendix D.

When using a gradient-descent-based backpropagation algorithm, the parameters are modified in a way that decreases the loss of the network by the greatest amount each iteration. The speed in which the parameters are updated is proportional to a pre-defined term called the *learning rate.* The learning rate greatly affects the nature of the training process. Small learning rates result in networks that train slowly, but carefully. If the learning rate is too small, however, the network may get stuck in a local minimum and stop learning. Conversely, if the learning rate is too large, the network may become unstable.

Another parameter that is significant during training is the *batch size*. The batch size, typically a power of 2, is the number of data entries that are used during each parameter update. A batch size of 128, for example, means that 128 data points will be used to calculate the gradient for the current update iteration. Large batch sizes result in controlled and accurate updates because the network uses more information to calculate each parameter update. The trade-off is that large batch sizes decrease the number of update iterations that the network experiences.

The batch size and the learning rate are two common examples of *hyperparameters*. Hyperparameters are parameters that are not adjusted automatically during training. These parameters are manually adjusted to increase performance during a process called hyperparameter tuning. Neural network training and hyperparameter tuning are executed intermittently to minimize the pre-defined loss metric on the validation dataset. This process is illustrated in Appendix E.

2.2.4 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of neural network that is applied to image classification problems. The following Figure illustrates the structure of a CNN:

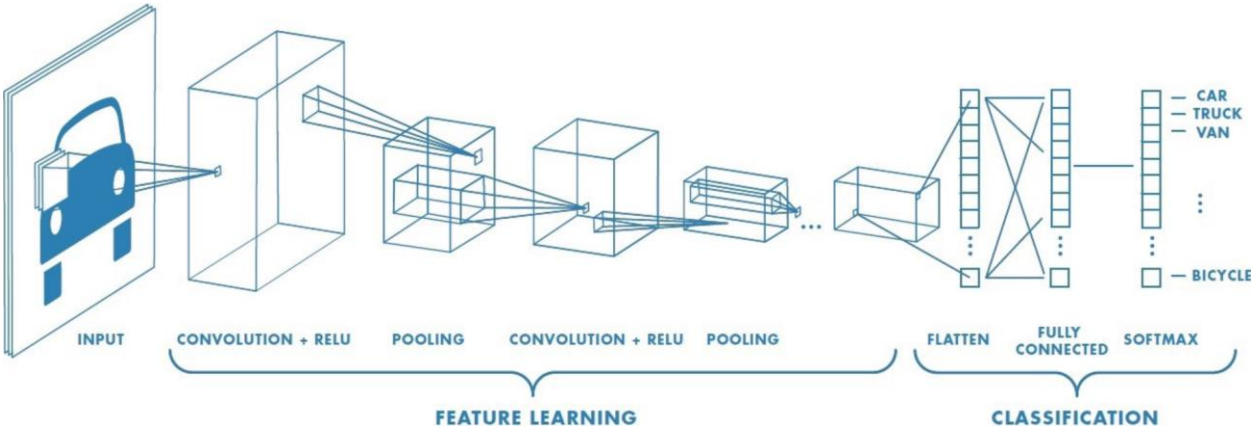

Figure 2: Illustration of the structure of a CNN [14]

The CNN in the above Figure is an example of a common convolutional structure. A CNN is comprised of a feature learning segment and a classification segment. During feature learning, learnable image filters called kernels are convolved over each convolutional layer. Using these learnable image filters, CNNs can discern which characteristics present in the input image data are relevant to the defined image classification problem. An explanation on learnable image filters and convolutions is included in Appendix F.

A pooling layer immediately follows each convolutional layer. Pooling layers reduce the size of the feature representations formed by the convolutional layers. These layers reduce the computational complexity of the network by extracting the most prominent features from the feature set generated by each convolutional layer. There are many ways in which a pooling layer can summarize the contribution from the previous layer. Various pooling layer types are discussed in Appendix G.

As evidenced by the common CNN configuration shown in Figure 2, the pattern of a convolutional layer followed by a pooling layer is repeated sequentially in the feature learning segment of a CNN. The learned features that are produced by the feature learning segment of the CNN are passed into the classification segment. This segment is a fully-connected neural network similar to the structure of the network defined in Figure 1. The number of convolutional

layers, pooling layers, fully-connected layers, and the ordering of these layers defines the architecture of the CNN. The architecture of the CNN, in conjunction with the other factors discussed, contribute to the accuracy of the network after training.

2.2.5 Modern CNN Architectures

In 1994, Yann LeCun's LeNet5 [15] was released. The network was designed for the generic handwritten digit recognition task. The LeNet5 architecture is illustrated by the following Figure:

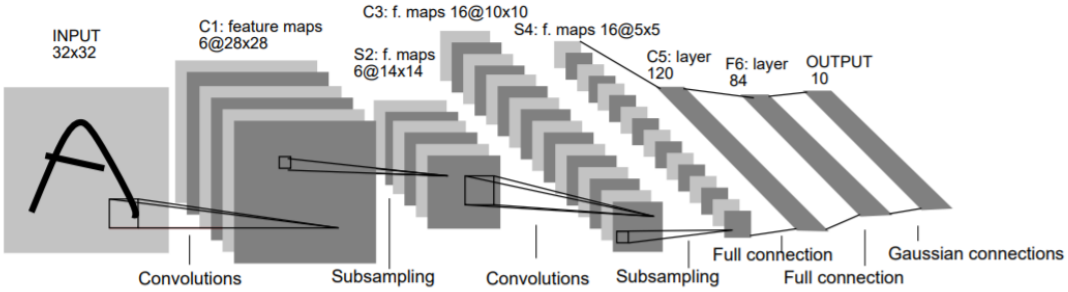

Figure 3: LeNet5 architecture [15]

The above architecture contains 60,000 parameters. This architecture rather simple in comparison to more modern architectures. This is partially due to the computational burden associated with training a network of this size as a result of the available computational power during the 1990s. In an age where manual heuristics were more common than learned approaches to image classification, LeCun argues that "better pattern recognition systems can be built by relying more on automatic learning and less on hand-designed heuristics" [15].

As GPUs became a general-purpose tool for computation, more complex architectures surfaced. In 2012, Alex Krizhevsky's AlexNet [16] won the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [17]. Figure 4 details the AlexNet architecture:

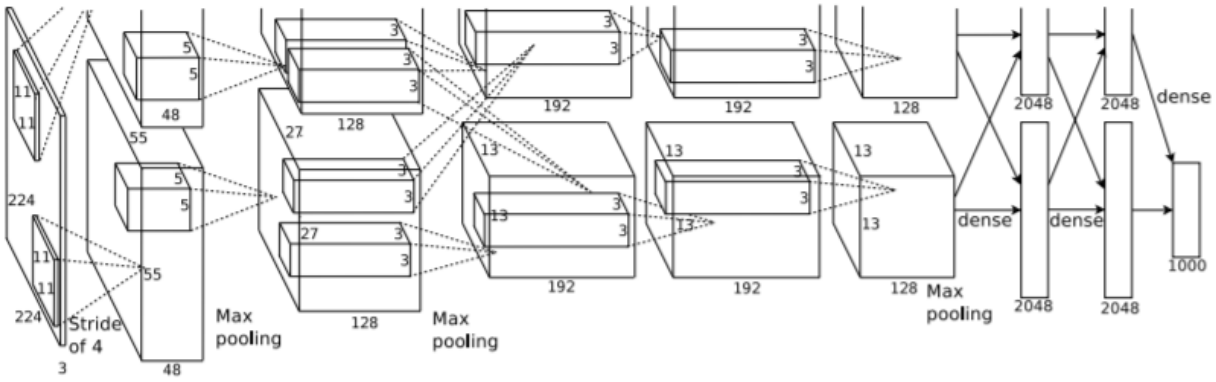

Figure 4: AlexNet architecture [16]

As illustrated in Figure 4, the architecture is much more complex. AlexNet has 60 million parameters and 650,000 neurons. Compared to the 60,000 parameters present in LeNet5, this is a tremendous increase in complexity. Consistent with the Figure above, the architecture consists of five convolutional layers and three fully-connected layers. Given the advances in GPU technology, the network was trained for a week on two NVIDIA GPUs.

In 2014, GoogLeNet [18], also known as Inception-v1, won the ILSVRC [19]. GoogLeNet boasted monumental accuracy improvements compared to the ILSVRC winners of 2012 and 2013. The fundamental factor responsible for the accuracy improvement yielded by GoogLeNet was the use of a novel neural network component called an inception module shown in Figure 5 below:

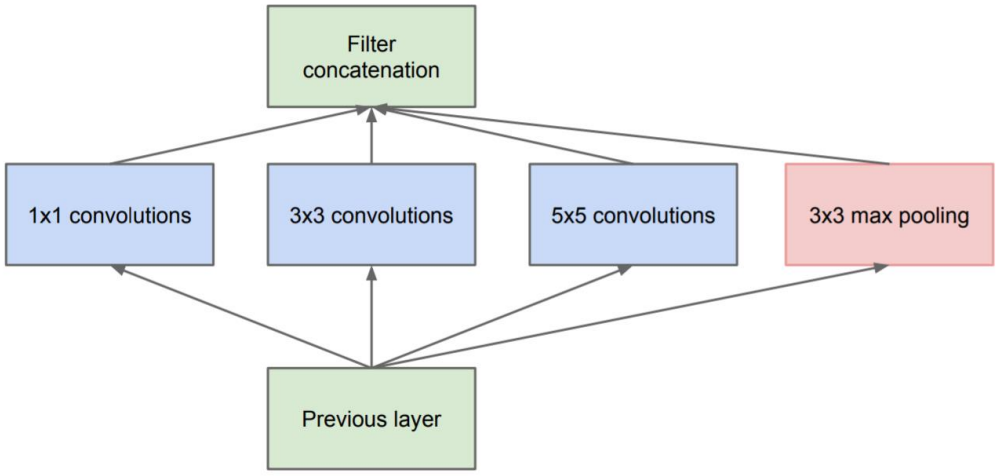

Figure 5: An illustration of the GoogLeNet inception module [18]

As delineated in Figure 5, the inception module synthesizes multiple filters of varying sizes stemming from the previous layer. The inception module “aligns with the intuition that visual information should be processed at various scales and then aggregated so that the next stage can abstract features from different scales simultaneously” [18]. Since Inception-v1, more modern architectures have presented minor accuracy improvements including Inception-v3 [20], ResNet-50 [21], Xception [22], Inception-v4 [23], Inception-ResNet-V2 [23], and ResNeXt-50 [24].

2.3 Homomorphic Permissibility

Fully-connected and convolutional layers rely primarily on two operations: addition and multiplication. Other neural network components, like activation functions, loss functions, and pooling layers, can situationally rely on division and comparison. When encrypting the inputs and parameters of a neural network, the ability to divide two encrypted numbers and compare two numbers is lost. This means that some neural network components that are essential to effective learning are lost. Network components that can be used in the encrypted domain shall be known as homomorphically permissible network components. Conversely, network components that cannot be utilized in the encrypted domain shall be known as homomorphically non-permissible network components. These components will be defined in detail in the following sections.

2.3.1 Layer Types

Both fully-connected layers and convolutional layers are homomorphically permissible. The operations required to propagate the inputs to each layer through to the outputs are addition and multiplication. Pooling layers, however, situationally require comparison. Average pooling layers take the average of a multiple pre-defined windows. This type of pooling is permissible because when averaging a set of numbers, the sum of the numbers is divided by a pre-defined unencrypted scalar. This scalar division is permissible. Max pooling, however, is not permitted. This layer type compares multiple sets of numbers and returns the largest in each set. It therefore relies on the comparison operator. Fortunately, this layer type is not fundamental to successful learning. With respect to layer types, CryptoNets are largely uninhibited.

2.3.2 Loss Functions

The primary regression loss function is the mean-squared error loss function. This function measures mean of the sum of the squared error between the targets and the predictions. This loss function is permissible because it, like the average pooling layer, requires addition, multiplication, and division by a scalar. Other regression loss functions, like the absolute loss function, only requires addition and is therefore also permissible.

With respect to classification, the most common loss function is the cross-entropy loss function. This loss function attempts to characterize the correctness of a multi-class prediction. Unfortunately, it is not permitted in the encrypted domain because it relies on a normalization factor that is dependent on the input data. For more details on the intricacies of this loss function, see Appendix D.

Fortunately, depending on our desired application, the permissibility of the loss function may not matter. If we wish to use a fully-encrypted model during training then the loss function must be permissible. If, however, we only need to use an encrypted model during inference, the loss function need not be permissible. This is true because the loss function is only used during training. During inference, we only need the parameters that propagate the inputs to the network to be permissible. Computing the loss while inferring using a pre-training model is unnecessary. Microsoft's CryptoNet research paper used a pre-train model and was therefore able to use a non-permissible loss function [2].

2.3.3 Activation Functions

The most significant performance bottleneck comes from the heavy restrictions imposed by the non-permissibility of the most common activation functions. For a fully-connected layer to be conducive to the learning capacity of a network, it must be followed by a non-linear activation function. The most common activation function used to succeed fully-connected layers is the sigmoid activation function. This activation function is non-permissible. Alternatives to the sigmoid activation function for the purpose of activating a fully-connected layer are the tanh, arctan, ReLU, and leaky ReLU functions, all of which are non-permissible. Similarly, convolutional layers require a non-linear activation function, most commonly the non-permissible ReLU function. For information on these non-permissible activation functions, see Appendix C. Given that these activation functions are non-permissible, sub-par approximations of these functions must be computed to successfully activate fully-connected layers in the encrypted domain. These approximations must be in the form of a polynomial. The linear, squared, and cubed activation functions will be delineated in the following section.

2.3.4 Homomorphically Permissible Activation Functions

Prior to discussing permissible activation functions, it is important to discuss the objectives of these activation functions. An activation function necessarily introduces non-linearity into a neural network. Without this non-linearity, the Universal Approximation Theorem no longer holds, and the network loses tremendous predictive power [13]. Furthermore, the performance of a network is largely dependent on which activation function was selected at each layer. The chosen activation function at each layer also affects the time that it takes to run the network. Given that the activation function impacts the empirical performance of the network as well as the time that it takes to run it, choosing a strong activation function is important.

The two homomorphically permissible activation functions that will be delineated are the linear activation function and the scaled squared activation function. These activation functions were selected because they have proven to show strong empirical results [2, 25]. Other homomorphically permissible activation function approximations, like polynomial approximations of the ReLU or sigmoid functions, exist. These functions, when implemented during the design process, did not provide noticeable advantages and were therefore not used.

2.3.4.1 Linear Activation Function

The linear activation function, $f(x) = x$, is equivalent to not applying an activation function. This function does not introduce non-linearity into the model and therefore, would significantly hinder the learning capacity of a single-layered model. In a multi-layered network utilizing a non-linear activation function, however, the linear activation function is conducive to learning and ahs the advantage of being the fastest possible activation function. Given the triviality of the function, this activation function will not be plotted.

2.3.4.2 Scaled Squared Activation Function

Microsoft's CryptoNet design used the squared activation function $f(x) = x^2$ to follow their first and second fully-connected layers in conjunction with the sigmoid activation function following their output layer [2]. This function introduces non-linearity into the network and unlocks tremendous learning capacity. Furthermore, given its simplicity, it is comparable in speed to the linear activation function. The problem with this function is that its derivative,

$f(x) = 2x$, is unbounded. The dire consequence of this is that the loss can situationally explode, especially when dealing with more complex models.

To rectify this issue, Microsoft attached a convolutional layer with no activation function after each squared activation function. Convolutional layers are used in image processing. When dealing with floating point features, convolutional layers are not practical. Therefore, in the context of floating point feature datasets, a modified version of the squared activation function must be implemented, the scaled squared activation function.

The scaled squared activation function is the squared activation function scaled by a complexity factor, c: $f(x) = c * x^2$. The complexity factor helps to constrain the explosion of the loss. Several complexity factor settings are plotted in Figure 6:

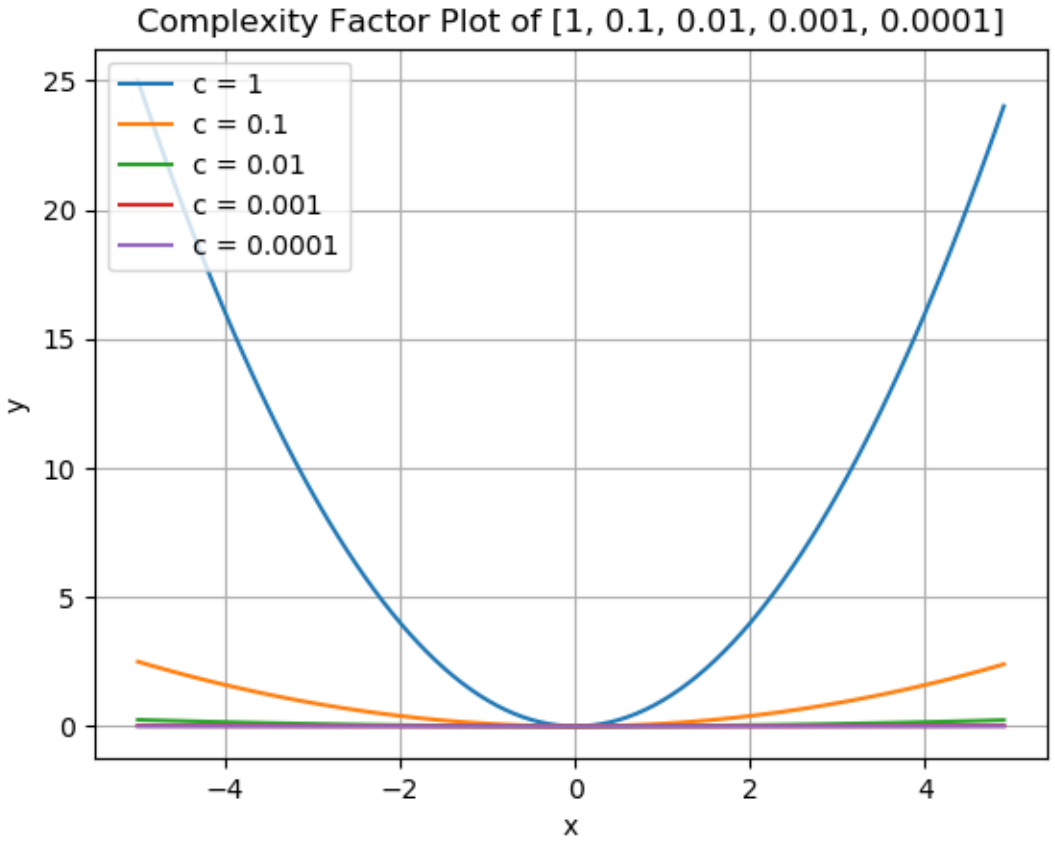

Figure 6: Complexity factory plots of f(x) = x^2

The complexity factor must be tweaked just as the learning rate and batch size. This factor will vary from network to network and can, in some cases, be incredibly small (< 0.0000000001).

3. Methods

3.1 Setup

3.1.1 Hardware

To perform the experimentation an Intel® Core™ i7-7700 CPU @3.60GHz and 16GB of RAM were used. Further, a Seagate BarraCude 2TB 7200RMP SATA 3.0 internal hard drive was used for storage. Given the complex computations executed during experimentation, these details are relevant. Alterations to the hardware could have a significant empirical impact when attempting to reproduce the subsequent results.

3.1.2 Environment

In 2018 Intel released HE Transformer for nGraph, an open-source CryptoNet research library for Python [26]. By importing the virtual environment defined by this library, one can encrypt any homomorphically permissible neural network. The meaning of a homomorphically permissible network will be explained in Section 3.3 CryptoNet Structures. To duplicate the virtual environment, a fresh Ubuntu 18.04 distribution was installed. Then, using IntelAI's publicly available GitHub repository containing build instructions [27], the virtual environment was installed. Since several steps relevant for the duplication of the environment were not included in the instructions, a more rigorous setup guide was defined in the GitHub research repository associated with this paper for posterity [28]. By following the instructions defined in "he-transformer-setup-instructions.sh" file in the base directory of the repository, one can expediently install IntelAI's HE Transformer for nGraph. IntelAI's HE Transformer for nGraph virtual environment was then defined as a PyCharm virtualenv environment to allow efficient TensorFlow model delineation and execution.

3.1.3 Parameter Configuration

After successfully establishing the TensorFlow environment, the encryption parameters had to be specified prior to execution. A coefficient modulus of [60, 40, 40, 60], a poly-modulus degree of 8192, and a security level of 128 were used. To be consistent with the parameters used by other prominent research groups, the aforementioned parameters were modelled after

Microsoft's CKKS encryption scheme example [29]. These parameters were held constant throughout experimentation to ensure fair model comparison.

3.2 Boston Housing Regression Dataset

When comparing models, different environments will have a significant impact on timing analysis. By measuring the slowdown when using an encrypted and an unencrypted model, the slowdown becomes relative and can be used as a baseline of sorts. To compute this relative slowdown, a simple and small Keras regression dataset called the Boston Housing regression dataset [30] was used.

The total length of the dataset is 506. It was partitioned into a training and a testing set. The training set had 404 elements while the testing set had 102. There are 13 input features for each training sample. These 13 features along with the target feature are detailed below:

Features:

- **CRIM**: per capita crime rate by town
- **ZN**: proportion of residential land zoned for lots over 25,000 sq.ft.
- **INDUS**: proportion of non-retail business acres per town
- **CHAS**: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- **NOX**: nitric oxides concentration (parts per 10 million)
- **RM**: average number of rooms per dwelling
- **AGE**: proportion of owner-occupied units built prior to 1940
- **DIS**: weighted distances to five Boston employment centres
- **RAD**: index of accessibility to radial highways
- **TAX**: full-value property-tax rate per $10,000
- **PTRATIO**: pupil-teacher ratio by town
- **B**: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- **LSTAT**: % lower status of the population

Target:

- **MEDV**: Median value of owner-occupied homes in $1000's

The targets range from 5 to 50 and have a mean of 22.53. The following Figure helps visualize the testing targets:

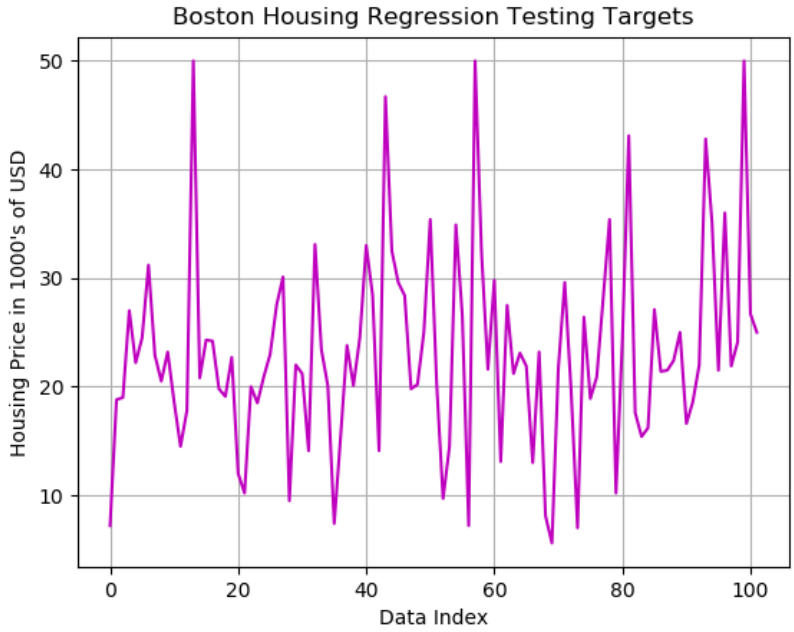

Figure 7: Boston Housing regression dataset testing targets

Since we are not concerned with the accuracy of the network, the dataset was not shuffled and there was no data parsing. The network simply learned and inferred using the unaltered training and testing data, respectively.

A simple network with one hidden layer was defined in both the encrypted and unencrypted domains. These networks had identical structures, except for the activation function, to characterize the timing loss when operating in the encrypted domain. The structure of this model is as follows:

3.2.1 Boston Housing Regression Model:

1. *Fully-Connected Layer*
    - Given that the dataset is a regression dataset with floating point features, we cannot take advantage of feature locality. When dealing with image inputs, for example, the locality of the image pixels contributes to the overall understanding of the image. In this case, interchanging the order of the columns has no effect on the target. Therefore, using a fully-connected layer is the most effective way to ensure effective learning.

2. *Activation Layer*
    - For the encrypted model the scaled squared activation function with a complexity factor of 1 was used. The unencrypted model utilized the sigmoid activation function.
3. *Fully-Connected Output Layer*
    - Given that the outputs are floating point targets ranging from 5 to 50, no activation function was used after the output layer. If, for example, a sigmoid activation function was added, the output would be bounded between 0 and 1 and the accuracy of the model would inevitably be 0.

Using the structure defined above, several trials using both the encrypted and unencrypted models were executed. The time to train and the time to infer were recorded.

3.3 Year Prediction Regression Dataset

The Boston Housing dataset, while efficient for small-scale timing comparison experiments, is not suitable for large-scale model comparison because it is too simple and does not have enough data entries. To compare models, a larger dataset is required because model comparison can only be unbiased if a validation set exists. The Boston Housing dataset has only 506 entries and is incapable of supporting a training set, validation set, and a testing set because of its size. Also, the Boston Housing dataset has only 13 features. This dataset is therefore relatively simple and more complex models may not be required.

The Year Prediction dataset is larger and much more complex. Each of its 515345 represents a song. It has 90 features. Each feature describes an element of the timbre of the song it pertains to. The target is an integer that delineates the year that the song was released. The following Table shows the first 7 columns of the first 5 data entries of this dataset:

Table 1: First 5 entries of the Year Prediction dataset

Index:	C1:	C2:	C3:	C4:	C5:	C6:	C7:	C8-C90:	Target:
1	39.13	-23.02	-36.21	1.68	-4.27	13.01	8.06		2008
2	37.66	-34.06	-17.36	-26.78	-39.95	-20.75	-0.1		2002
3	26.52	-148.16	-13.3	-7.26	17.22	-21.99	5.52	…	2004
4	37.68	-26.84	-27.11	-14.96	-5.87	-21.69	4.87		2003

5	39.12	-8.3	-51.38	-4.43	-30.07	-11.96	-0.85		1999

A plot of a subset of the testing data targets is illustrated in Figure 7:

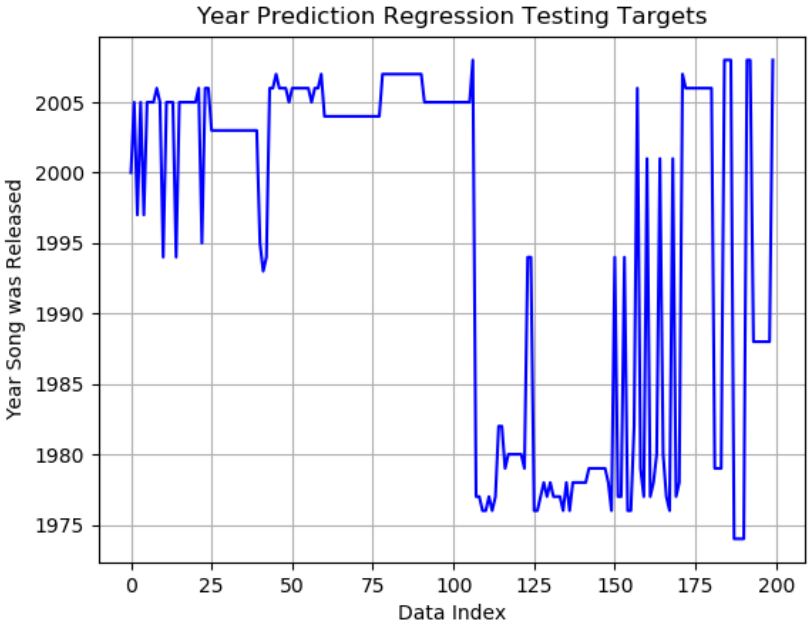

Figure 8: Year Prediction regression dataset testing targets

The targets range from 1922 to 2011, have a mean of 1998, and a median of 2002. Given the increased complexity of this dataset, more complex models are expected to perform better because of their increased learning capacity. This is important because if the model had a simple correlation between its inputs and outputs then comparing models would be trivial. All learned models would achieve similar performance.

When dealing with a song dataset, it is important to take steps to avoid the producer effect. The producer effect occurs when songs from the same artist appear in both the training and testing dataset. The model memorizes the patterns of the artist and fails to generalize to new artists. Fortunately, the dataset was preprocessed by the authors of the dataset to avoid the producer effect. The data folder was downloaded, split into training, validation, and test sets, and then shuffled. No empty values or undefined entries exist.

Three models of varying complexity were implemented in both the encrypted and unencrypted domain to characterize the trade-offs between performance and time. The reason that these models were also implemented in plaintext is to provide a frame of reference. This

frame of reference will provide insight into the performance advantages in the encrypted realm relative to plaintext performance advantages. The most success was found using the scaled squared activation functions in the activation layers for the encrypted model. Further, great success was found when scaling the number of nodes in each layer linearly from the input dimension through to the number of nodes in the final layer.

The first and most simple model has only 3 layers. The complexity factor of the activation layer may seem unreasonably small; however, it is necessary to effectively bound the output. This model is delineated below:

3.3.1 Year Prediction Dataset Model 1

1. *Fully-Connected Layer*
    - Input Dimension: 90
    - Number of Nodes: 180
2. *Activation Layer*
    - Encrypted Domain:
        - Function: Scaled Squared
        - Complexity Factor: 0.00000001
    - Unencrypted Domain:
        - Function: Sigmoid
3. *Fully-Connected Output Layer*
    - Input Dimension: 180
    - Output Dimension: 1

The second model is a little more complex. This model implements two extra layers. To deal with the problem of the unbounded gradient, both activation layers were scaled down aggressively:

3.3.2 Year Prediction Dataset Model 2

1. *Fully-Connected Layer*
    - Input Dimension: 90
    - Output Dimension: 180

2. *Activation Layer*
    - Encrypted Domain:
        - Function: Scaled Squared
        - Complexity Factor: 0.00000001
    - Unencrypted Domain:
        - Function: Sigmoid
3. *Fully-Connected Layer*
    - Input Dimension: 180
    - Output Dimension: 270
4. *Activation Layer*
    - Encrypted Domain:
        - Function: Scaled Squared
        - Complexity Factor: 0.00000001
    - Unencrypted Domain:
        - Function: Sigmoid
5. *Fully-Connected Output Layer*
    - Input Dimension: 562
    - Output Dimension: 1

The final model was the most complex of the three. In order to keep the gradient bounded, less nodes per layer were used. The linear scaling of the nodes per layer, however, remained. This model illustrated below:

3.3.3 Year Prediction Dataset Model 3

1. *Fully-Connected Layer*
    - Input Dimension: 90
    - Output Dimension: 180
2. *Activation Layer*
    - Encrypted Domain:
        - Function: Scaled Squared
        - Complexity Factor: 0.00000001
    - Unencrypted Domain:

        - Function: Sigmoid

3. *Fully-Connected Layer*
    - Input Dimension: 180
    - Output Dimension: 270
4. *Activation Layer*
    - Encrypted Domain:
        - Function: Scaled Squared
        - Complexity Factor: 0.00000001
    - Unencrypted Domain:
        - Function: Sigmoid
5. *Fully-Connected Layer*
    - Input Dimension: 270
    - Output Dimension: 360
6. *Activation Layer*
    - Encrypted Domain:
        - Function: Scaled Squared
        - Complexity Factor: 0.00000001
    - Unencrypted Domain:
        - Function: Sigmoid
7. *Fully-Connected Output Layer*
    - Input Dimension: 456
    - Output Dimension: 1

Using the encrypted versions of these three models, the time to train one batch of data was recorded through 5 trials. Furthermore, the number of epochs that it took to achieve a working model, the testing loss, the testing accuracy, and the time to test were also recorded. Lastly, a training plot that includes the training and validation curves created while training each model along with and a plot that illustrates a snippet of the testing outputs were generated.

3.4 MNIST Image Classification Dataset

To ensure that the analysis was not one-dimensional, a similar experimental procedure was executed on Yann Lecun's MNIST image classification dataset [31]. The inputs of this

dataset are 28x28 image pixel matrices that represent handwritten digits. The outputs of the dataset are the labels of the image entries, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. One such input is plotted in Figure 9:

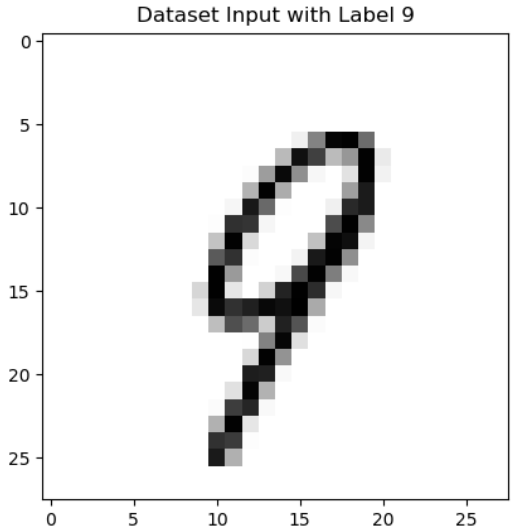

Figure 9: MNIST image input classified as a 9

The 9 shown in Figure 9 is on an angle and is not as easy to interpret as one would imagine. The MNIST dataset has a total of 31000 elements. 21700 of these elements were dedicated to the training set. The remainder of the elements were split evenly between the validation and testing set.

After extracting the dataset, two models of significantly different complexity were defined. Given that this dataset is an image classification dataset, convolutional layers are the most effective layers to extract meaning from the input data. This is because the spatial locality of the pixels in the image matter. If two, or more, pixel values are interchanged, the meaning of the image changes drastically. Convolutional layers, as explained before, take advantage of weight sharing and learnable image kernels. The value of these convolutional layers was investigated through the analysis of two encrypted models. Unencrypted models were not defined.

The first model utilizes one convolutional layer followed by the squared activation function and a pooling layer. Then, the convolutional output is flattened and passed into a fully-

connected neural network. The most success was found with mean pooling and aggressively small scaling factors. The first model has the following structure:

3.4.1 MNIST Classification Dataset Model 1

1. *Convolutional Layer*
    a. Input Depth: 1
    b. Kernel Width: 5
    c. Stride: 1
    d. Feature Maps: 10
2. *Activation Layer*
    a. Function: Scaled Squared
    b. Complexity Factor: 0.00000001
3. *Pooling Layer*
    a. Pooling Type: Mean
    b. Stride: 2
4. *Fully-Connected Layer*
    a. Input Dimension: 14 x 14 x 6
    b. Output Dimension: 2352
5. *Activation Layer*
    a. Function: Scaled Squared
    b. Complexity Factor: 0.00000001
6. *Fully-Connected Output Layer*
    a. Input Dimension: 2352
    b. Output Dimension: 10

The second model is larger. The convolutional segment of the network is three layers larger. An extra convolutional layer, activation layer, and pooling layer were utilized. This model is defined below:

3.4.2 MNIST Classification Dataset Model 2

1. *Convolutional Layer*

    a. Input Depth: 1
    b. Kernel Width: 5
    c. Stride: 1
    d. Feature Maps: 10
2. *Activation Layer*
    a. Function: Scaled Squared
    b. Complexity Factor: 0.00000001
3. *Pooling Layer*
    a. Pooling Type: Mean
    b. Stride: 2
4. *Convolutional Layer*
    a. Input Depth: 10
    b. Kernel Width: 7
    c. Stride: 1
    d. Feature Maps: 18
5. *Activation Layer*
    a. Function: Scaled Squared
    b. Complexity Factor: 0.00000001
6. *Pooling Layer*
    a. Pooling Type: Mean
    b. Stride: 2
7. *Fully-Connected Layer*
    a. Input Dimension: 7 x 7 x 18
    b. Output Dimension: 1764
8. *Activation Layer*
    a. Function: Scaled Squared
    b. Complexity Factor: 0.00000001
9. *Fully-Connected Layer*
    a. Input Dimension: 1764
    b. Output Dimension: 10

Using these two models the time to train one batch of data was recorded through 5 trials. Also, the testing loss, the testing accuracy, and the time to test were recorded.

4. Results

4.1 Boston Housing Regression Dataset Timing Analysis

Table 2: Time to train, number of epochs, and time to infer for encrypted Boston Housing regression model

Trial Number:	Time to Train (s):	Number of Epochs:	Time to Infer:
1	26.947	100	0.186
2	26.392	100	0.212
3	27.284	100	0.215
4	29.201	100	0.183
5	28.243	100	0.185

Table 3: Mean time to train and mean time to infer for encrypted Boston Housing regression model

Mean Time to Train:	Mean Time to Infer:
27.613 ± 1.114	0.1962 ± 0.0159

Table 4: Time to train, number of epochs, and time to infer for unencrypted Boston Housing regression model

Trial Number:	Time to Train (s):	Number of Epochs:	Time to Infer:
1	0.59	100	0.055
2	0.585	100	0.066
3	0.544	100	0.097
4	0.556	100	0.052
5	0.546	100	0.052

Table 5: Mean time to train and mean time to infer for unencrypted Boston Housing regression model

Mean Time to Train:	Mean Time to Infer:
0.5642 ± 0.0218	0.0644 ± 0.0191

4.2 Year Prediction Regression Dataset Model Comparison

4.2.1.1 Year Prediction Dataset Model 1 Encrypted Results

Table 6: Time to train 250 batches for the Year Prediction dataset encrypted model 1

One Layer Time Trials:	Time to Train 250 Batches (s):	Batch Size:
1	68.891	64
2	74.461	64
3	72.277	64
4	69.691	64
5	77.388	64

Table 7: Testing loss and testing accuracy for the Year Prediction dataset encrypted model 1

Testing Loss:	Testing Accuracy:
6599647	81.00%

Table 8: Time to test for the Year Prediction dataset encrypted model 1

Testing Trials:	Time to Test (s):
1	74.538
2	80.355
3	82.192
4	79.49
5	75.227

Table 9: Mean time to train and mean time to infer for Year Prediction dataset encrypted model 1

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Loss:
72.542 ± 3.489	78.360 ± 3.330	6599646.5

4.2.1.2 Year Prediction Dataset Model 1 Unencrypted Results

Table 10: Time to train 250 batches for the Year Prediction dataset unencrypted model 1

One Layer Time Trials:	Time to Train 250 Batches (s):	Batch Size:
1	0.203	64
2	0.188	64
3	0.184	64
4	0.17	64
5	0.192	64

Table 11: Testing loss and testing accuracy for the Year Prediction dataset unencrypted model 1

Testing Loss:	Testing Accuracy:
7841881	80.43%

Table 12: Time to test for the Year Prediction dataset encrypted model 1

Testing Trials:	Time to Test (s):
1	0.121
2	0.123
3	0.129
4	0.128
5	0.132

Table 13: Mean time to train and mean time to infer for Year Prediction dataset unencrypted model 1

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Loss:
0.187 ± 0.0120	0.127 ± 0.00451	7841881

4.2.2.1 Year Prediction Dataset Model 2 Encrypted Results

Table 14: Time to train 250 batches for the Year Prediction dataset encrypted model 2

One Layer Time Trials:	Time to Train 250 Batches (s):	Batch Size:
1	393.2	64

2	386.1	64
3	392.5	64
4	394.3	64
5	387.2	64

Table 15: Testing loss and testing accuracy for the Year Prediction dataset encrypted model 2

Testing Loss:	Testing Accuracy:
6577615	81.60%

Table 16: Time to test for the Year Prediction dataset encrypted model 2

Testing Trials:	Time to Test (s):
1	547.073
2	514.5
3	591.2
4	544.2
5	579.4

Table 17: Mean time to train and mean time to infer for Year Prediction dataset encrypted model 2

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Loss:
390.660 ± 3.737	555.275 ± 30.517	6577615

4.2.2.2 Year Prediction Dataset Model 2 Unencrypted Results

Table 18: Time to train 250 batches for the Year Prediction dataset unencrypted model 2

One Layer Time Trials:	Time to Train 250 Batches (s):	Batch Size:
1	0.233	64
2	0.219	64
3	0.239	64
4	0.24	64
5	0.232	64

Table 19: Testing loss and testing accuracy for the Year Prediction dataset unencrypted model 2

Testing Loss:	Testing Accuracy:
7124668	80.42%

Table 20: Time to test for the Year Prediction dataset encrypted model 2

Testing Trials:	Time to Test (s):
1	0.123
2	0.162
3	0.133
4	0.148
5	0.133

Table 21: Mean time to train and mean time to infer for Year Prediction dataset unencrypted model 2

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Loss:
0.233 ± 0.00838	0.140 ± 0.0153	7124668

4.2.3.1 Year Prediction Dataset Model 3 Encrypted Results

Table 22: Time to train 250 batches for the Year Prediction dataset encrypted model 3

One Layer Time Trials:	Time to Train 250 Batches (s):	Batch Size:
1	513.4	64
2	526.9	64
3	520	64
4	515.7	64
5	567.1	64

Table 23: Testing loss and testing accuracy for the Year Prediction dataset encrypted model 3

Testing Loss:	Testing Accuracy:
6419452.7	81.70%

Table 24: Time to test for the Year Prediction dataset encrypted model 3

Testing Trials:	Time to Test (s):
1	821.6
2	762.3
3	774.3
4	792.6
5	834.3

Table 25: Mean time to train and mean time to infer for Year Prediction dataset encrypted model 3

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Loss:
528.620 ± 22.116	797.020 ± 30.558	6419452.7

4.2.3.2 Year Prediction Dataset Model 3 Unencrypted Results

Table 26: Time to train 250 batches for the Year Prediction dataset unencrypted model 3

One Layer Time Trials:	Time to Train 250 Batches (s):	Batch Size:
1	0.304	64
2	0.315	64
3	0.31	64
4	0.309	64
5	0.307	64

Table 27: Testing loss and testing accuracy for the Year Prediction dataset unencrypted model 3

Testing Loss:	Testing Accuracy:
7034911	80.46%

Table 28: Time to test for the Year Prediction dataset encrypted model 3

Testing Trials:	Time to Test (s):
1	0.126
2	0.136
3	0.14

4	0.151
5	0.147

Table 29: Mean time to train and mean time to infer for Year Prediction dataset unencrypted model 3

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Loss:
0.309 ± 0.00406	0.140 ± 0.00977	7034911

4.3 MNIST Classification Dataset Model Comparison

4.3.1 MNIST Classification Dataset Model 1 Results

Table 30: Time to train 250 batches for the MNIST Classification dataset model 1

Model 1 Time Trials:	Time to Train 250 Batches(s):	Batch Size:
1	86.1	64
2	79.866	64
3	84.546	64
4	83.452	64
5	81.211	64

Table 31: Time to test for the MNIST Classification dataset model 1

Testing Trials:	Time to Test (s):
1	115.611
2	110.784
3	112.132
4	116.642
5	109.563

Table 32: Mean time to train and mean time to infer for MNIST Classification dataset model 1

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Accuracy:
83.036 ± 2.511	112.946 ± 3.0637	92.30%

4.3.2 MNIST Classification Dataset Model 2 Results

Table 33: Time to train 250 batches for the MNIST Classification dataset model 2

Model 2 Time Trials:	Time to Train 250 Batches(s):	Batch Size:
1	132.44	64
2	127.559	64
3	134.545	64
4	131.113	64
5	129.326	64

Table 34: Time to test for the MNIST Classification dataset model 2

Testing Trials:	Time to Test (s):
1	321.615
2	314.121
3	311.156
4	319.8463
5	325.124

Table 35: Mean time to train and mean time to infer for MNIST Classification dataset model 1

Mean Time to Train 250 Batches (s):	Mean Time to Test (s):	Optimal Testing Accuracy:
131.001 ± 2.706	318.372 ± 5.666	99.12%

5. Discussion

5.1 Boston Housing Regression Dataset Baseline Timing Analysis

As previously mentioned, the purpose of the timing analysis is to establish a baseline for future work. Tables 2 and 3, respectively, include the time that it took to train and test using both an encrypted and unencrypted regression model.

5.2 Year Prediction Regression Dataset Model Comparison

5.3 MNIST Image Classification Dataset Model Comparison

6. Conclusion

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," *In Proc. STOC,* pp. 169-178, 2009.

[2] N. Dowli, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," Microsoft Research, 2016.

[3] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *EUROCRYPT'11: Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology 2011*, p. 129–148, 2011.

[4] M. van Dijk, C. Gentry, S. Halevi and V. Vaikuntanathan, "Fully Homomorphic Encryption over the Integers," in *Advances in Cryptology -- EUROCRYPT 2010*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2010, pp. 24-43.

[5] Z. Brakerski, C. Gentry and V. Vaikuntanathan, "(Leveled) Fully homomorphic encryption without bootstrapping," in *ITCS '12: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 2012.

[6] A. Lopez-Alt, E. Tromer and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," *STOC '12: Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, p. 1219–1234, 2012.

[7] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP," *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology --- CRYPTO 2012*, vol. 7417, p. 868–886, 2012.

[8] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *IACR Cryptology ePrint Archive 2012/144*, 2012.

[9] J. W. Bos, K. Lauter, J. Loftus and M. Naehrig, "Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme," in *Cryptography and Coding*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2013, pp. 45-64.

[10] J. H. Cheon, A. Kim, M. Kim and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," *Cryptology ePrint Archive, Report 2016/421,* 2016.

[11] Y. Xu and R. Goodacre, "On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning," *Journal of Analysis and Testing,* vol. 2, 2018.

[12] R. Parmar, "Deep network architecture with multiple layers.," Towards Data Science, 2018.

[13] B. C. Csáji, "Approximation with Artificial Neural Networks," *MSc Thesis,* p. 22, 2001.

[14] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way," Towards Data Science, 2018.

[15] Y. Lecun, L. Bottou, Y. Benigo and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[16] A. Krizhevsky, G. Hinton and I. Sutskever, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25*, pp. 1097-1105, 2012.

[17] "ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)," [Online]. Available: http://image-net.org/challenges/LSVRC/2012/results.html. [Accessed 26 January 2020].

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1-9, 2015.

[19] "ImageNet Large Scale Visual Recognition Challenge 2014 (ILSVRC2014)," [Online]. Available: http://image-net.org/challenges/LSVRC/2014/results. [Accessed 26 January 2020].

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818-2826, 2016.

[21] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, 2016.

[22] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1800-1807, 2017.

[23] C. Szegedy, S. Ioffe, V. Vanhoucke and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, p. 4278–4284, 2017.

[24] S. Xie, R. Girshick, P. Dollár, Z. Tu and K. He, "Aggregated Residual Transformations for Deep Neural Networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5987-5995, 2017.

[25] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque and L. Fei-Fei, "Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference," *CoRR*, vol. abs/1811.09953, 2018.

[26] F. Boemer, Y. Lao, R. Cammarota and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," *CoRR*, vol. abs/1810.10121, pp. 1-13, 2018.

[27] F. Boemer, Y. Lao, R. Cammarota and C. Wierzynski, "nGraph-HE: Deep learning with Homomorphic Encryption (HE) through Intel nGraph," GitHub, [Online]. Available: https://github.com/IntelAI/he-transformer. [Accessed 17 4 2020].

[28] S. Atkins, "GitHub: Investigating CryptoNet structures for practical performance characterization," GitHub, [Online]. Available: https://github.com/atkinssamuel/UndergraduateCryptoNetThesisResearch. [Accessed 17 4 2020].

[29] "Microsoft SEAL (release 3.2)," Microsoft Research, Redmond, WA., February 2019. [Online]. Available: https://github.com/Microsoft/SEAL.

[30] F. C. JJ Allaire, "Keras Datasets," Keras, [Online]. Available: https://keras.io/datasets/. [Accessed 13 4 2020].

[31] Y. LeCun, C. Cortes and C. Burges, "THE MNIST DATABASE of handwritten digits," 1999. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed 27 January 2020].

[32] "Desmos Graphing Calculator," Desmos, 2020. [Online]. Available: https://www.desmos.com/calculator. [Accessed 27 01 2020].

[33] I. Goodfellow, Y. Bengio and A. Courville, in *Deep Learning (Adaptive Computation and Machine Learning series)*, MIT Press, 2016, p. 226.

[34] A. Brutzkus, O. Elisha and R. Gilad-Bachrach, "Low Latency Privacy Preserving Inference," *CoRR*, vol. abs/1812.10659, 2018 .

Appendix A: Public-Key Encryption

Suppose Bob wishes to send an encrypted to message to his friend, Alice. To do so, Alice would create two keys: a private key and a public key. The public key can encrypt messages into ciphertexts, and the private key can decrypt those messages back into plaintext. After creating this pair of keys, Alice would send the public key to Bob as in Figure 9, below:

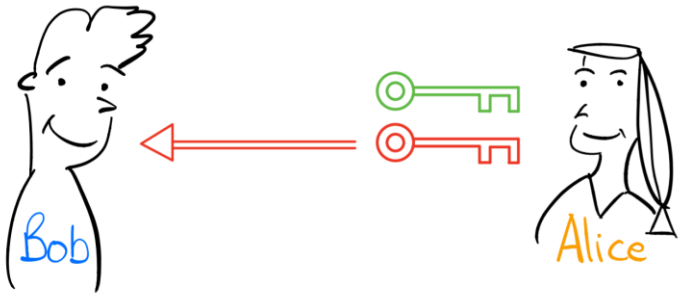

Figure 10: Alice sends Bob a public key (red), keeping her private key (green) to herself

Upon receiving the public key from Alice, Bob can use the public key to encrypt a message:

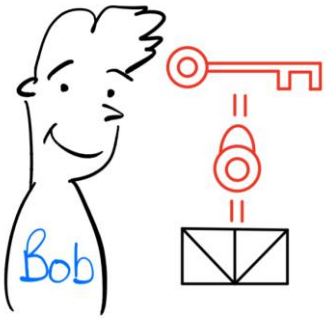

Figure 11: Bob uses Alice's public key to encrypt a message

Then, Bob can send his encrypted message to Alice:

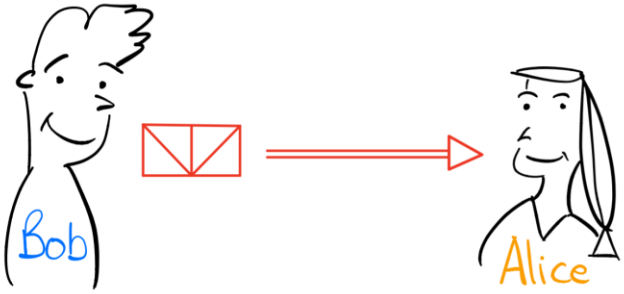

Figure 12: Bob sends his encrypted message to Alice

Using her private key, Alice can then decrypt Bob's encrypted message back into readable plaintext, as in Figures 12 and 13:

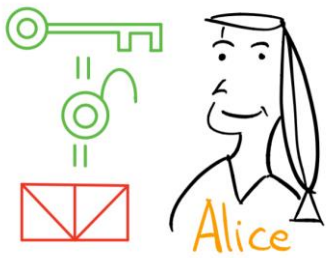

Figure 13: Alice uses her private key to decrypt Bob's encrypted message

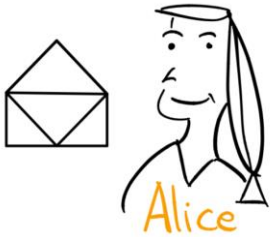

Figure 14: Alice can now view Bob's message

During this entire process, Alice has kept her private key secure. A third party listening in on Alice and Bob's conversation could have captured the contents of the public key, but would not be privy to Alice's private key, the essential element to decrypting Bob's messages.

Appendix B: Forward Propagation

The input to each neural network is vector of numbers of a constant dimension. During forward propagation, each node is assigned a calculated value based on the parameters of the network and the values of the nodes in the previous layer. Consider the following illustration of a simple neural network:

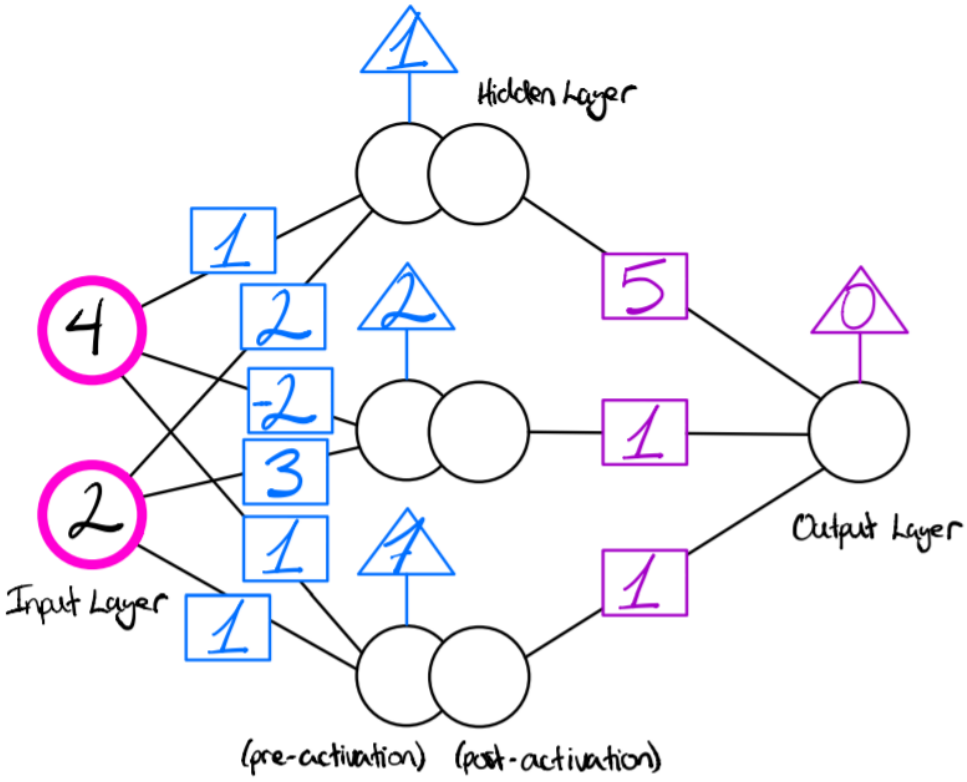

Figure 15: An example of a fully-connected neural network prior to forward propagation

The neural network detailed by Figure 14 has an input layer of dimension 2, one hidden layer of dimension 3, and an output layer of dimension 1. The weights connecting the input layer to the hidden layer are the numbers in the blue boxes. The biases associated with the hidden layer are the numbers in the blue triangles. Similarly, the weights connecting the hidden layer to the output layer and the bias term associated with the output layer are the numbers in the purple boxes and purple triangles, respectively. To illustrate the effects of the activation function, the hidden layer has been split into a pre-activation phase and a post activation phase.

During forward propagation, the input vector is propagated forward to the hidden layer. This propagation is the result of a multi-step process. First, the weighted sum of the input vector

and the parameters connecting the input layer and hidden layer is calculated for each node. These calculations are illustrated below:

$$4 * 1 + 2 * 2 = 8$$

$$4 * (-2) + 2 * (3) = -2$$

$$4 * 1 + 2 * 1 = 6$$

After calculating the weighted sum for each node, the bias terms associated with each node are added to their respective nodes. Figure 12 illustrates the result of these calculations:

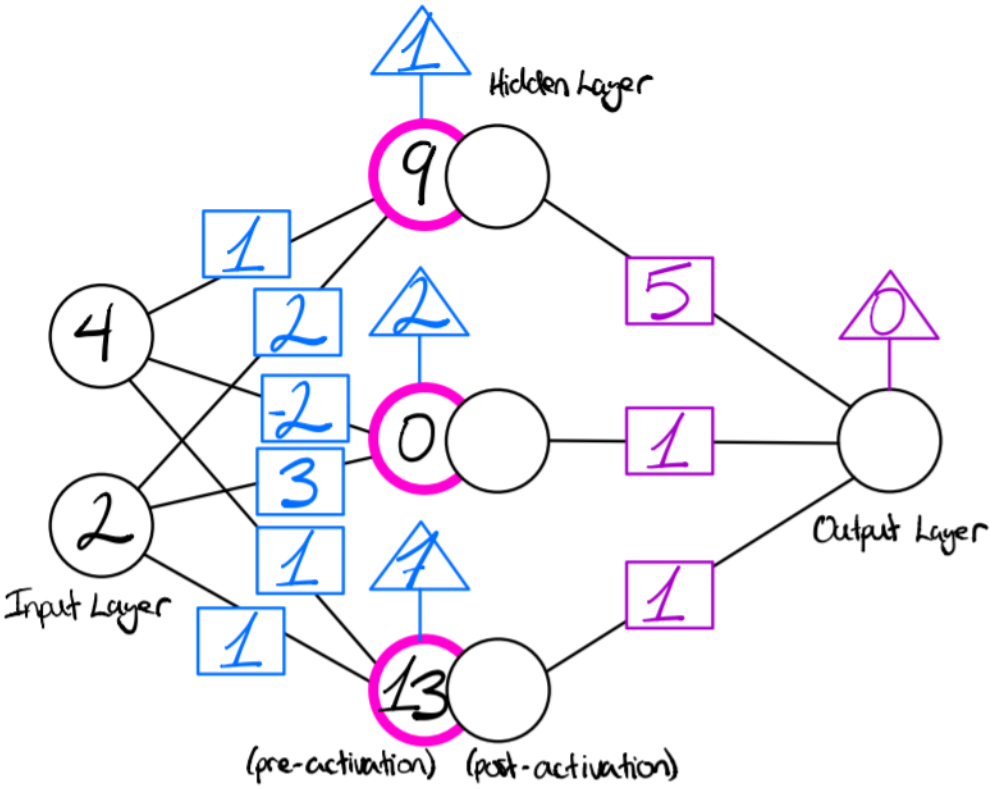

Figure 16: Illustration of the forward propagation process prior to the application of the activation function

Now, the values corresponding to each node are passed through an *activation function.* For a detailed explanation on activation functions, see Appendix D. The activation function used in this example is the Sigmoid activation function.

The values of the nodes in the hidden layer prior to being passed through the Sigmoid activation function are 9, 0, and 13, respectively. As such, we expect the Sigmoid activation function to activate the nodes with values of 9 and 13 and to mediate the contribution from the

node with a value of 0. Figure 16, below, illustrates the result of applying the Sigmoid activation function to the hidden layer:

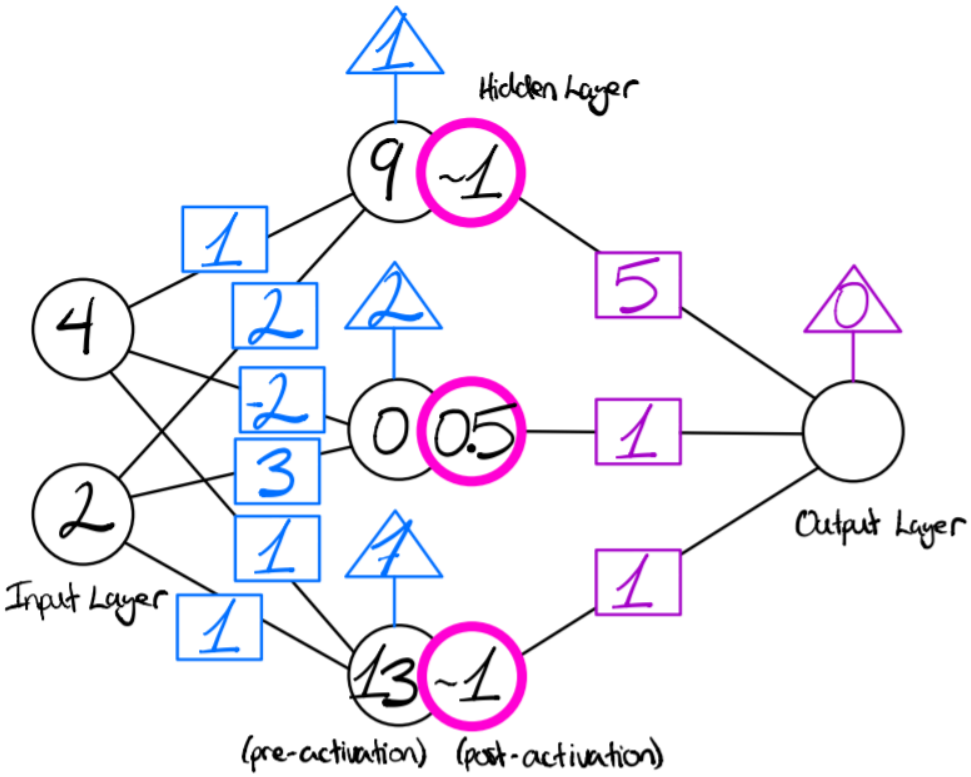

Figure 17: The forward propagation process after applying the activation function to the nodes in the hidden layer

As illustrated by Figure 16, the nodes in the hidden layer are transformed from values of 9, 0, and 13, to values of approximately 1, 0.5, and 1.

The values from the input layer have now propagated to the hidden layer. These values will continue to propagate in a similar fashion to the output layer. The output layer, in this case, uses a linear activation function. This is because there is a single output and the problem that is implicitly being solved by this simplified example is a regression problem. In a multi-class classification problem, however, one might use a Softmax activation function. For more information on common activation functions, see Appendix C. The final output of the neural network that was previously Figure 14 is delineated by Figure 17, below:

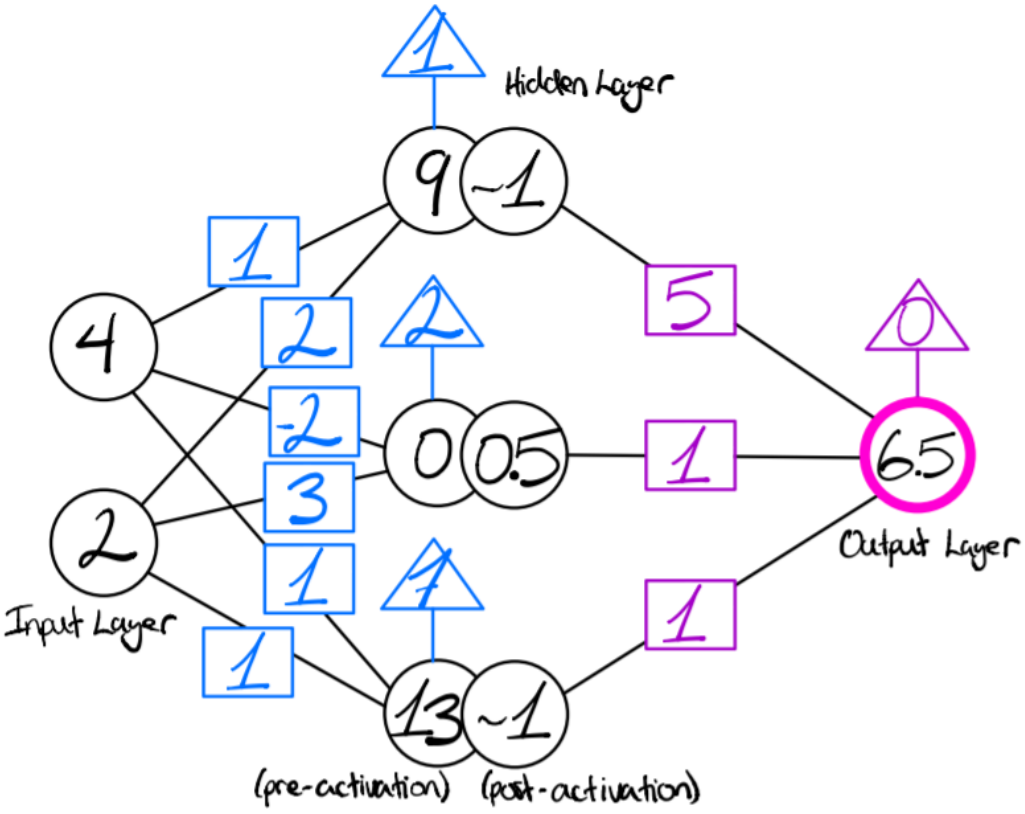

Figure 18: An illustration of the output of the neural network after forward propagation has occurred

The network has fully propagated the values from the input vector through to the output layer. The network has made an inference on the input vector and has completed the forward propagation process.

Appendix C: Activation Functions

A wide variety of activation functions are commonly applied to the output of a fully-connected neural network. Traditional activation functions include the Sigmoid function, the tanh function, and the arctan function. The equations and graphs corresponding to the functions listed above are included below:

C.1 Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$

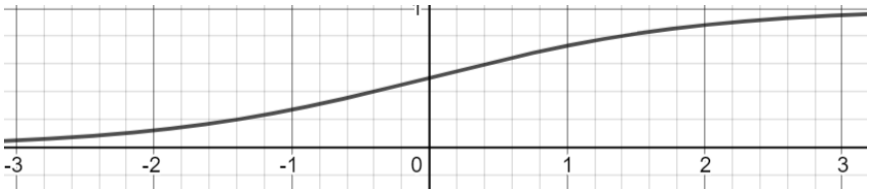

Figure 19: A graph [32] of the Sigmoid activation function

The Sigmoid activation function truncates large positive values to an output value close to 1 and truncates large negative values to an output value close to 0. The Sigmoid activation function activates nodes that yield a large positive value and suppresses nodes that yield a large negative value.

C.2 Tanh

$$f(x) = \tanh(x) = \frac{e^{x} + e^{-x}}{e^{x} + e^{-x}}$$

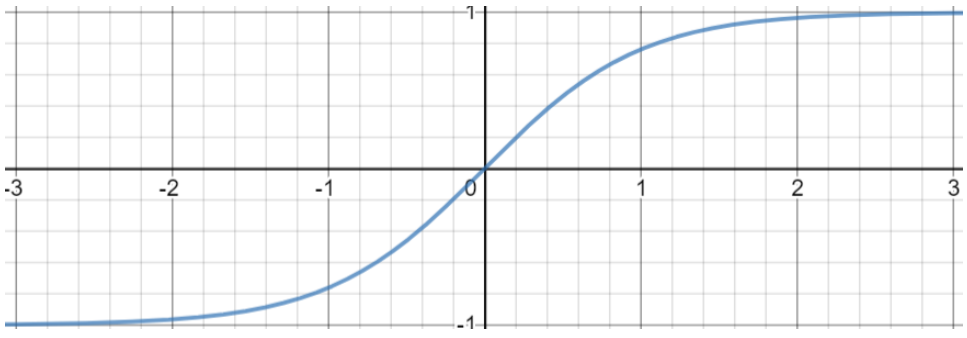

Figure 20: A graph [32] of the tanh activation function

Similar to the Sigmoid function, the tanh activation function truncates large positive values to an output value close to 1. Large negative values, however, are truncated to a value of -1 as opposed to 0.

C.3 Arctan

$$f(x) = \arctan(x)$$

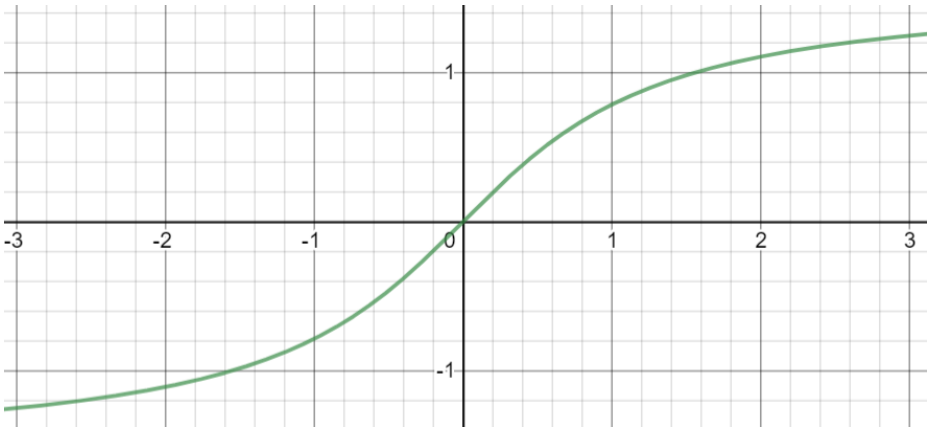

Figure 21: A graph [32] of the arctan activation function

The arctan function, as illustrated above, decays large positive and negative values as they grow. This ensures that the gradient of the activation function does not explode during backpropagation.

C.4 ReLU

In recent years, the rectified linear unit (ReLU) activation function has increased in popularity due to the observed performance increase: "[A] major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piecewise linear hidden units, such as rectified linear units" [33]. The following Equation and Figure illustrate the nature of the ReLU activation function:

$$f(x) = \max(0, x) = \{x \geq 0 : x, x < 0 : 0\}$$

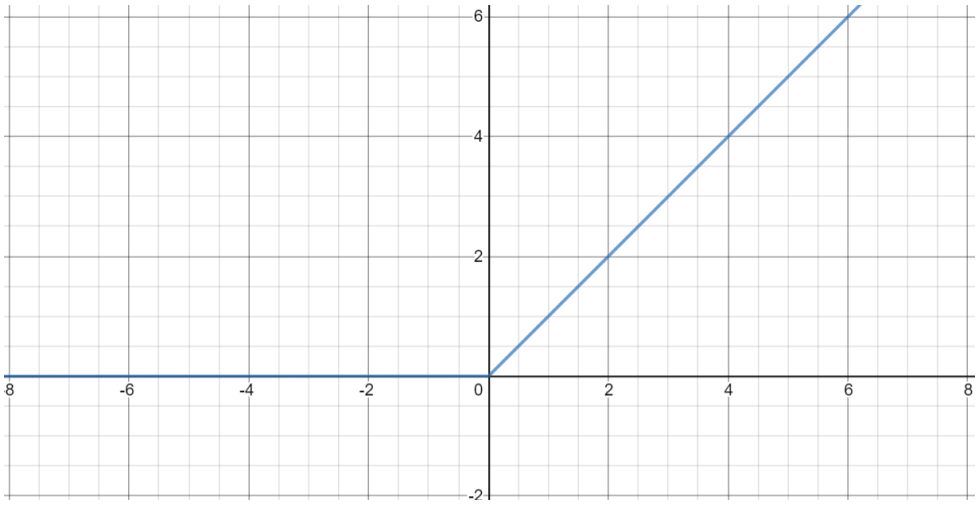

Figure 22: A graph [32] of the ReLU activation function

Unlike the above activation functions, the ReLU function truncates negative numbers and simply applies a linear function to the positive numbers. CNNs commonly use the ReLU activation function due to the aforementioned performance advantages.

C.5 Leaky ReLU

One problem that arises with the ReLU activation function is that due to the truncation of negative neurons, certain neurons may never contribute to the functionality of the network because the gradients of those neurons will always be zero. These neurons are called *dead neurons*. To address this issue, the Leaky ReLU function was conceived:

$$f(x) = \{x \geq 0 : x, x < 0 : 0.1x\}$$

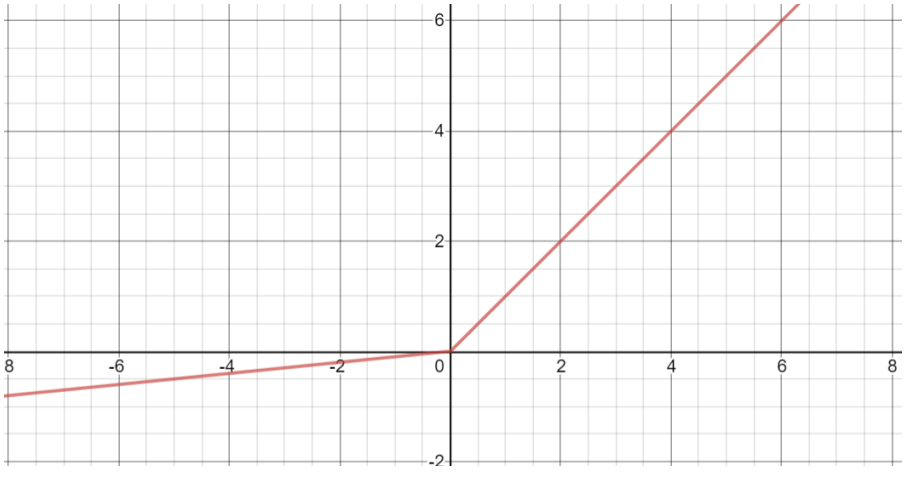

Figure 23: A graph [32] of the Leaky ReLU function

The Leaky ReLU function facilitates a small positive linear function for numbers less than 0. This effectively revives the dead neurons by allowing negative neurons to contribute to the predictive power of the network.

Appendix D: Loss Functions for Binary and Multi-Class Classification

For binary classification problems, a common loss function is the binary cross-entropy (BCE) loss function. This loss function is defined below:

$$Loss = -\sum_{c=1}^{2} y(c) * \log p(c)$$

In the above equation, the function *y(c)* is either 0 or 1, depending on which class the current data entry belongs to. The function *p(c)* is a discrete function containing the predicted probabilities of the two classes.

For multi-class classification problems, the BCE loss function can be extended to include multiple classes. This loss function is called the cross-entropy loss function and it is also defined below:

$$Loss = -\sum_{c=1}^{M} y(c) * \log p(c)$$

The classification problem pertaining to the defined loss function above has M classes. The function *y(c)* is 1 if the current data entry has class *c*. The discrete function *p(c)* contains the predicted probabilities of each class.

The binary cross-entropy and cross-entropy loss functions attempt to accurately define a loss metric that delineates how accurate each prediction with respect to the ground truth label. Figure 23 is a graph of the negative natural logarithm:

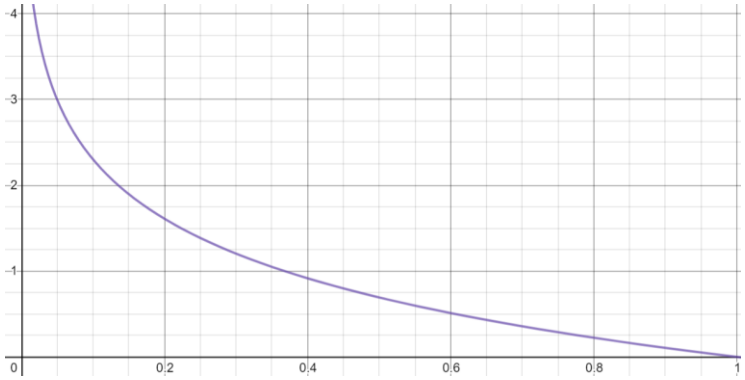

Figure 24: A graph [32] of the negative natural logarithm, y = -ln(x)

Recall that maximizing the cross-entropy loss is the same thing as minimizing the logarithmic loss. Thus, Figure 23 illustrates the impact of a prediction on a given ground truth label. If the network predicts the ground truth label with a low confidence, then the negative natural logarithm of Figure 23 will approach a very large number. If the network predicts the correct ground truth label with high confidence, however, the negative natural logarithm will be very low, as desired. This is the intuition behind using the negative natural logarithm as a loss function.

Appendix E: Loss Curves and Neural Network Training

The performance of a neural network during training is measured using a *loss curve*. The loss curve measures the pre-defined loss of a neural network as a function of the number of update iterations the network has experienced. The following is an example of a neural network validation loss curve on top of a training loss curve. In this example, the loss function being measured is the negative natural logarithm loss function.

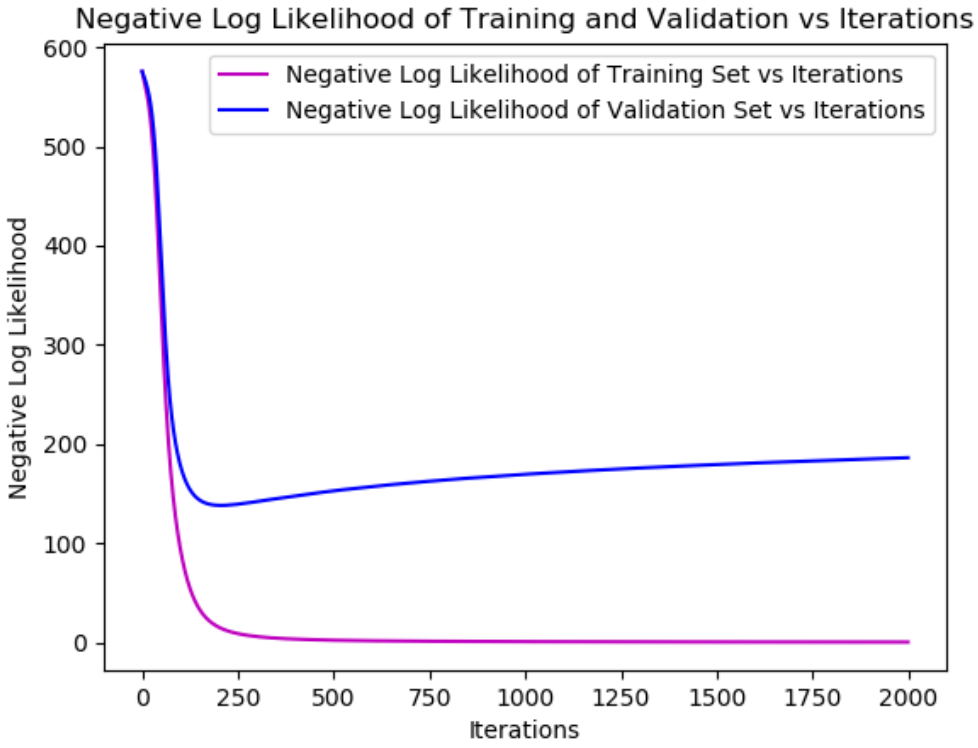

Figure 25: A typical graph of the validation and training loss of a neural network as a function of the update iterations

The Figure above is a strong representation of what typical validation and training loss curves look like. Initially, the validation loss decreases rapidly. Then, as the neural network continues to train on the training dataset, the validation loss increases as the training loss continues to decrease. This occurs because the neural network is starting to memorize the training data. Models that are able to make accurate predictions on unseen data are valuable. Models that can learn a certain set of data and only predict on seen data are completely useless. As such, validation and training loss curves are used to determine which version of a model should be selected. In this example, the version of the model at iteration 200 would be selected because it minimizes the validation loss.

Appendix F: Linear Image Filtering and Convolutions

CNNs use simple linear image filters to extract information from the input image. The following Figure is a grayscale image of Jenga blocks:

Figure 26: Grayscale image of Jenga blocks

The above image can be represented by a matrix of pixel values corresponding to the pixel intensities at each location in the image. Linear image filters can be convolved over this image to produce new versions of the above image that are more meaningful.

One such linear filter is the Sobel filter. The Sobel filter calculates the gradient of an input image in a desired direction. The 3x3 right Sobel, illustrated by Figure 26 below, computes the horizontal derivative of each pixel with respect to the adjacent pixels.

-1	0	1
-2	0	2
-1	0	1

Figure 27: Illustration of the 3x3 right Sobel filter

By applying the right Sobel filter to the grayscale image of Jenga blocks in Figure 25, the following image is obtained:

Figure 28: Result of the application of the right Sobel filter

The image above is smooth in the areas that do not have well-defined horizontal edges. The right Sobel filter accentuates the locations in the image that have significant horizontal derivates. In other words, the right Sobel filter captures the locations and magnitudes of the horizontal edges present in the original image. These edges may be useful in the context of image classification because the locations and magnitude of the edges could help indicate which class that image belongs to.

CNNs are powerful because they learn which image features matter with respect to the classification task at hand. Each convolutional layer uses a series of kernels of a pre-defined dimension. The dimension of a kernel is defined by the height and width of that particular kernel. The Sobel filter in Figure 26 has dimension 3, for example. The adjustable weights of a convolutional layer are the matrix entries of the filters associated with that layer. During backpropagation, the weights of each convolutional filter in a convolutional layer are updated to minimize the pre-defined loss function. In this way, each convolutional layer learns which input image characteristics are relevant to the meaning of the image as a whole.

Appendix G: Types of Pooling Layers

There are two categories of pooling layers: local pooling layers and global pooling layers. Local pooling layers divide the input matrix into windowed chunks. Then, the contribution from each windowed chunk is extracted. The two most common local pooling types are average pooling and max pooling. Global pooling layers compute a feature of a significantly smaller dimension using a global mathematical operation that captures the meaning of each feature map. Global average pooling is an example of a type of a global pooling layer.

Consider the following matrix of integers:

5	4	2	5
0	4	2	9
2	1	1	7
2	1	4	3

Figure 29: 4x4 matrix of integers representing the input to a pooling layer

Let the above matrix serve as an input to a pooling layer. The colors in the above Figure define the windowed chunks that the input matrix has been divided into. Average-pooling uses the average of each windowed chunk to define a smaller matrix that attempts to capture the contribution from each individual window. The result of applying average-pooling to the above Figure is a 2x2 output matrix shown below:

3.25	4.5
1.5	3.75

Figure 30: Result of applying average pooling to the input matrix in Figure 25

This matrix is able to retain some of the information from the input matrix. Furthermore, this matrix is half the size of the input matrix.

Max-pooling uses the maximum value from each windowed chunk to perform the same dimensionality reduction. The result of applying max pooling to the input matrix of Figure 28 is illustrated below:

5	9
2	7

Figure 31: Result of applying max-pooling to the input matrix of Figure 25

Max-pooling and average pooling each reduce the dimensionality of the input while attempting to retain the information conveyed by that input.

Average global pooling is a completely different approach. Instead of splitting the input matrix into windowed chunks, the algorithm takes the average of the entire input matrix:

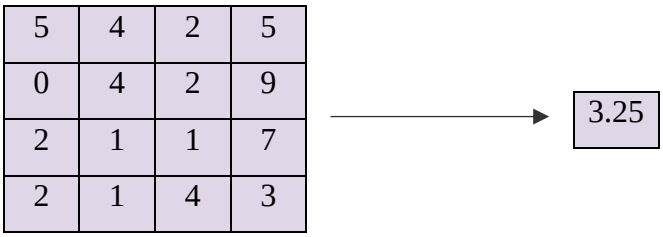

Figure 32: Result of applying global average pooling to the input matrix

The original 4x4 input matrix is reduced to a 1x1 matrix. Surprisingly, this pooling method can be very effective and is used as a fundamental component of the ResNet50 architecture [21].