4-2020

Optimization Study of an Image Classification Deep Neural Network

Rose Ault
Grand Valley State University

Advisor:
Greg Wolffe

Rose Ault
HNR 499
Final Report

Optimization Study of an Image Classification Deep Neural Network

Abstract

Machine Learning is an important and growing field within Artificial Intelligence. It is particularly useful in situations where developing an algorithm to perform the task in a conventional way would be extremely difficult. Instead of being programmed specifically to complete a task, a program embodies a trained model that can recognize patterns present in given example data, and is able use that model to make predictions on future data. Neural networks are a prominent example of machine learning models used for this purpose. Neural networks are models that are based on how brains work, with massive numbers of connected processing elements called nodes. Training the model is a process of iteratively assigning different weights to node connections; in effect, the network "learns" to recognize patterns.

This process of training a neural network to perform a task involves repeatedly testing it with example data and using the results to modify the parameters or weights within the model in such a way as to minimize error. The component that makes these changes is called the optimizer. As the field of machine learning has grown and matured, many different types of optimizers have been proposed and developed, each with their own set of advantages and disadvantages. The goal of this research project is to conduct a performance evaluation of different optimizers, quantifying the speed and accuracy with which they perform an image classification task.

Introduction

Image classification is an easy task for humans. Most humans can tell at a glance if a photo contains a dog or a cat, although it may be hard for them to precisely articulate how they made that determination. For these and other reasons, a simple recognition task is a more complicated problem for a computer program. Neural networks provide a means to solve this problem, and allowing autonomous systems to classify large amounts of images with greater speed than a human could ever hope to produce.

Neural networks contain many nodes that are connected together in ways that allow the computer to make a choice about what information it has been given, whether a simple binary decision, e.g. Is this a cat or a dog?, or a more complex decision, e.g. Where are the obstacles in this video? Each connection between nodes is activated or not based on the input data and the weights associated with each node.

Convolutional Neural Networks (CNN) are very popular in image analysis because of their ability to extract features by convolution. The use of filters to convolve the input into features reduces the number of parameters needed for the network to function, allowing the network to be lighter and faster. Between these convolution layers are pooling layers. These reduce the size of the data being fed through the neural network. Ultimately the convolved information is fed through a fully connected activation layer to classify the input, and the output of the network is obtained. [1]

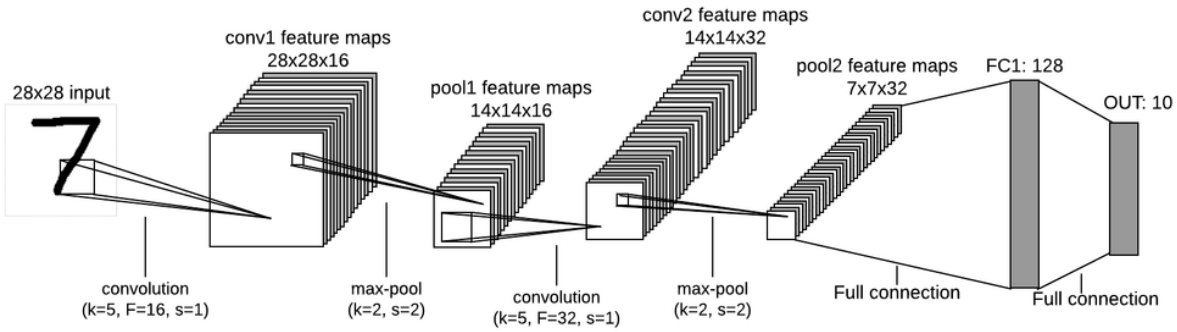


Figure 1: CNN for identifying handwritten digits

Figure 1 diagrams two different convolution layers that extract features and two pooling layers that reduce the size of the processed data. What starts out as a 28x28 pixel input goes through several feature maps, then is reduced to 14x14 datasets, goes through more feature maps and size reduction, and feeds forward through several fully connected layers that produce the final output (the classification).

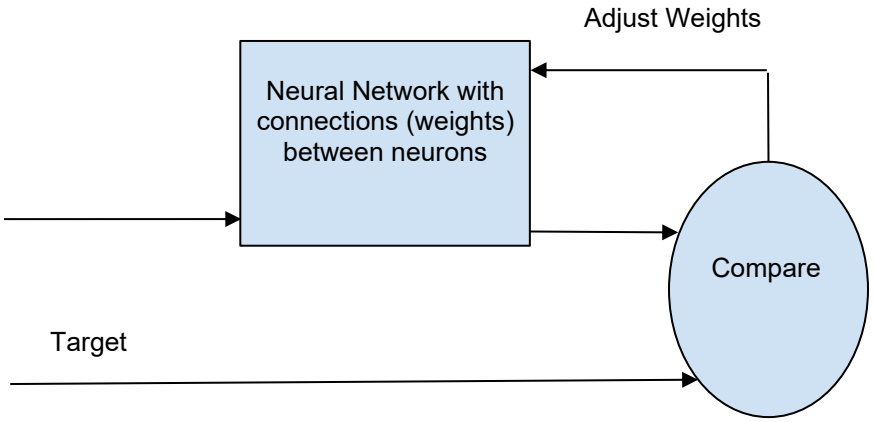


Figure 2: Neural network training process

Figure 2 shows the basic process involved in training a neural network. The input to the neural network consists of images from a labeled dataset, which are fed forward through the neural network. This generates a classification, or a guess of what category the image belongs in. This classification is the output of the neural network, which is then compared to the target, or actual classification of the image. The loss function is what performs this

comparison, and returns the success (or error) of the classification. The output of the loss function directs the adjustment of the weights within the neural network; this is where the optimizer function is applied. Optimizers update the model in response to the output of the loss function. The amount that the optimizer changes each weight in the model is mediated by the learning rate. The learning rate combined with the output from the loss function controls how much the optimizer changes each weight. There are many ways the optimizer can calculate this change, which has led to continual development in the available types of optimizers.

Much research has been conducted on creating detailed algorithms for specific cases of image classification, but this project involves comparing the performance of different kinds of optimizers on existing neural networks.

Method

The primary goal of this project was to train a neural network model using different optimizers, and compare their respective accuracy and training time. As the goal was to rapidly get a functioning neural network up and running, a Python framework called Keras was used to handle the build and training of the neural networks used in this project.

Neural networks have many applications outside of image classification, but that use remains one of the most successful and widely deployed. For purposes of this project, a simple dataset was used, as the goal was to compare the performance of different models, changing the optimizer used in each one. Initially the Stanford Dogs Dataset[2] was employed, but fully implementing models based on this dataset proved computationally expensive. For this reason, the final dataset used was the Kaggle Dogs vs. Cats dataset[3].

In addition to identifying and evaluating appropriate datasets, a large part of the semester was spent researching the methods involved in building and training a neural network. Building a model from the ground up is unnecessarily intensive for comparing the performance of the optimizer, and potentially might not be successful in classifying data in a way that would be helpful for this project. For this reason, transfer learning was used to take advantage of pretrained models. Pretrained models are models that were built and trained on large benchmark datasets. These can be used as feature extractors for problems that have different but similar goals[4]. The Keras framework includes several different models, such as VGG16, Inception, and ResNet[5]. The approach used in this project was to use the pretrained model as a convolutional base feature extractor, and train a new classifier with the results of the pretrained model as input.

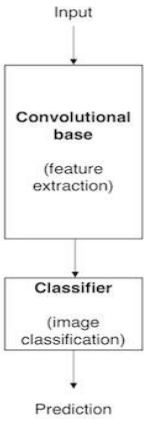


Figure 3: Basic outline of pretrained model

The VGG16 network was the pretrained model selected for this project because it is a simpler network without the auxiliary classifiers and skip connections of later models, while still being a deep CNN.

There are several different optimizers available within the Keras framework, four of which were selected to be compared in this project.

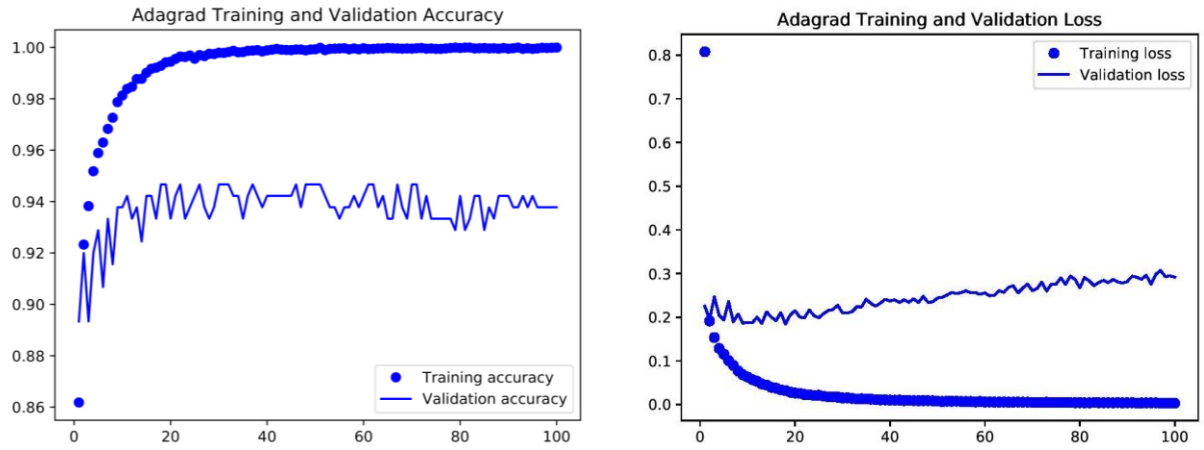
- **Adagrad** stands for Adaptive Gradient Algorithm. It is adaptive because each weight has a different learning rate, which is updated separately from the others. The rate of the updates are based on how often that parameter is used. This causes uncommon parameters to have a greater update rate than common parameters, which is beneficial when the dataset is sparse. However, because the calculation of learning rates is based on the historical squared gradients sum, the rates are continually decaying. This causes the algorithm to stop updating weights by any meaningful amount as training progresses.
- **RMSprop** was developed to address Adagrad's weaknesses. It attempts to fix the continual decay problem by calculating weight updates based on the running average of the squared gradients, not the sum of all squared gradients. This keeps the weight update responsive to the more recent past of the model.
- **Adadelta** is another optimizer that is very similar to Adagrad. Its goal is also to address the problem of continual decay. Adadelta only considers gradients that happened within some fixed window size. This allows it to keep learning even when many updates have been performed on the weights within the neural network. [6]
- **Adam**, the most recently developed of the optimizers considered, stands for Adaptive Moment Estimation. Adam, like Adagrad, creates an individual learning rate for each parameter. It uses both the decaying average of past gradients and the decaying average of past squared gradients to update parameters. [7, 8]

Results

To determine the performance of each optimizer, the model was iterated through 100 training epochs for each optimizer. The total time to train and the final classification accuracy of the model on the test data were primary evaluation metrics, along with average training accuracy, and validation accuracy. Also considered were the training and validation loss.

Adagrad

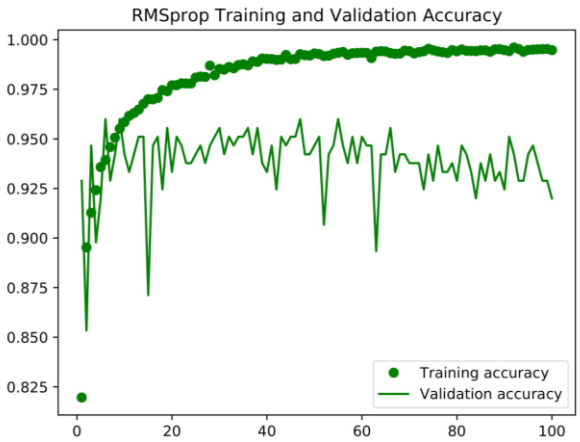
The model using the Adagrad optimizer took 306.93 seconds to train. The final classification accuracy of the model on the test set was 0.9222. This was similar to the average validation accuracy of 0.9375. The average validation loss was 0.245. The graphs below showcase the loss and accuracy for both training and validation data over 100 epochs.

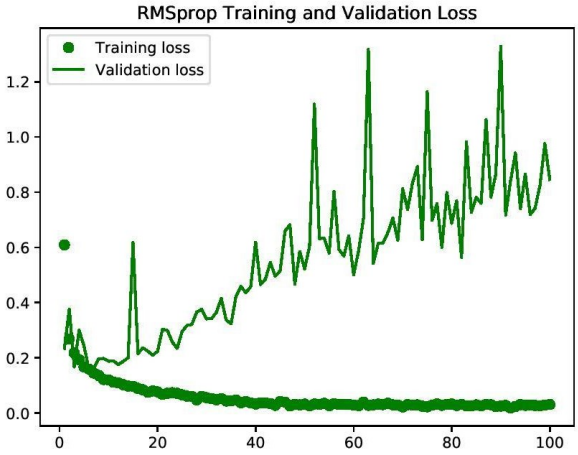


Figures 4 and 5 : Adagrad optimizer training and validation accuracy, and loss

RMSprop

The model using the RMSprop optimizer took 321.84 seconds to train. The RMSprop model had a final test accuracy of 0.9111. Its average validation accuracy was 0.9388. RMSprop had an average validation loss of .5591.

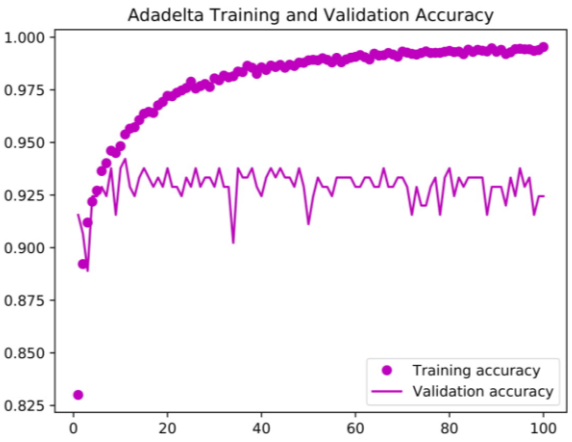


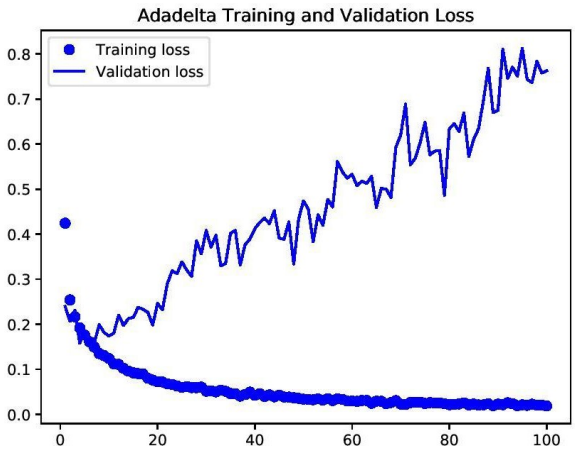


Figures 6 and 7 : RMSprop optimizer training and validation accuracy, and loss

Adadelta

The model using the Adadelta optimizer took 348.085 seconds to train. Its final test accuracy was at 0.9244. The average validation accuracy was very similar, at 0.9289. The average validation loss was .4547.

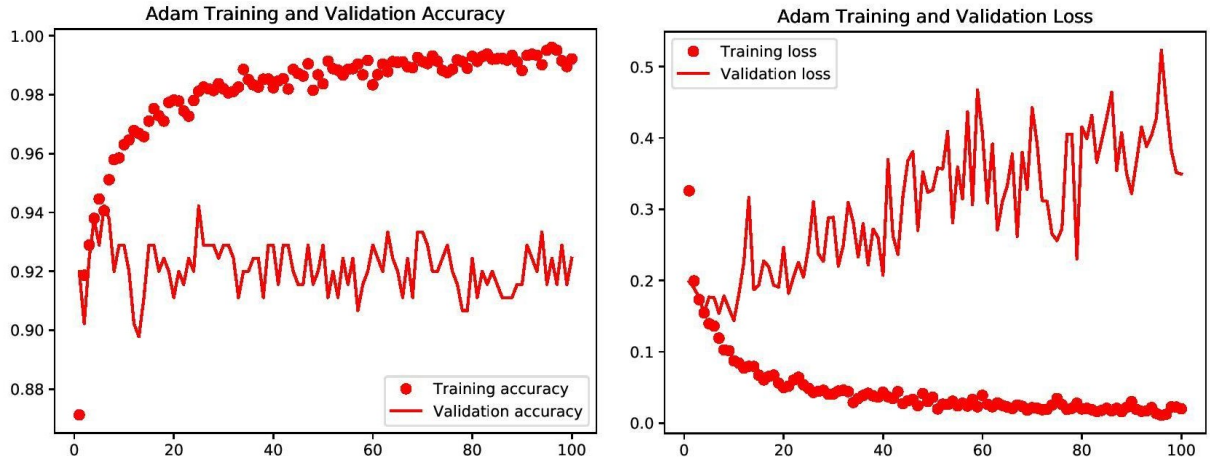




Figures 8 and 9 : Adadelta optimizer training and validation accuracy, and loss

Adam

The model with the Adam optimizer took 347.814 seconds to complete 100 epochs. For this model, the final test accuracy was 0.9288. This is similar to the average validation accuracy, which was 0.9212. These were both lower than the average training accuracy, 0.9811. The Adam optimizer produced an average training loss of 0.3038.



Figures 10 and 11 : Adam optimizer training and validation accuracy, and loss

Time to train

Each of the models were timed during training. Adagrad was the fastest to finish 100 epochs with 306.93 seconds, followed by RMSprop, then Adadelta. Adam had a very similar training time to Adadelta, both finishing in around 348 seconds.

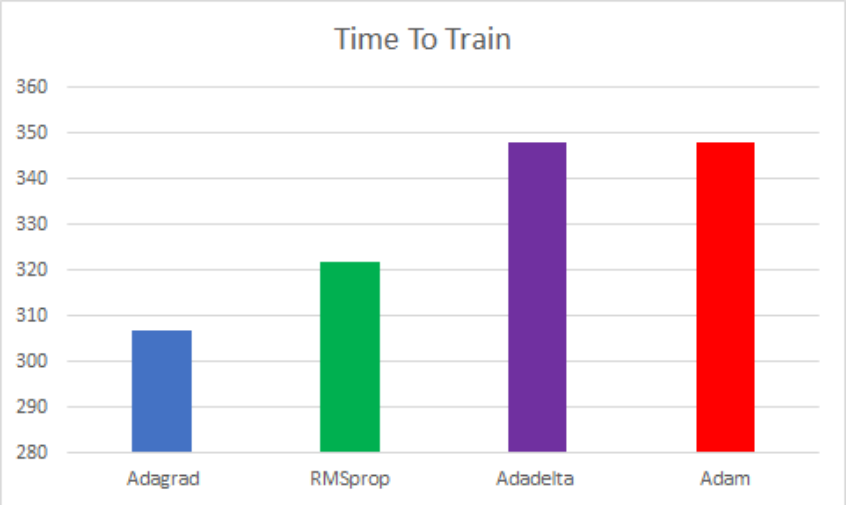


Figure 12: Training time for 100 epochs

Conclusion

The goal of this project was to investigate and quantify the effect of using different optimizers during the training process, as related to the outcome and speed of a deep neural network used in an image classification task. The optimizers tested provided the models with different strengths and advantages. In terms of accuracy, the four different models had very similar test accuracy. Because they performed so similarly in accuracy, it is useful to consider loss when comparing the models. Loss is the calculation of the error produced by the model during each epoch. The RMSprop and Adadelta models had the greatest average validation loss. This is shown in Figures 7 and 9, where the final validation loss was around the range of 0.8. When the model exceeded 40 epochs, there was no major gain in accuracy, but the loss continued to grow. The RMSprop model even experienced a slight decrease in accuracy. Large amounts of loss during the validation step can be indicative of a model that has been overfit (it has trained too specifically on the presented training data). The models using the Adagrad and Adam optimizers did not have these issues with loss.

The Adagrad model had the lowest training time, while the Adadelta and Adam models essentially tied for highest training time among the four. This was only by 25-40 seconds, so in light of the overfitting issues seen in the RMSprop and Adadelta models, the Adagrad and Adam models showed better overall performance. The Adagrad model still experiences continual decay, potentially causing problems on larger datasets where the learning rate will continue to decrease and make further training of the model impossible.

Therefore, our conclusion is that of the four models compared, the Adam optimizer demonstrated the best overall performance. Currently, Adam is one of the more popular optimizers for use in neural networks, and these results showcase that its popularity is well earned. As is typically the case, Adam benefitted greatly from improvements made to existing optimizers, thereby making it well suited for use in neural nets designed for image classification.

Bibliography

[1] Saha, S. (2018, December 17). A Comprehensive Guide to Convolutional Neural Networks-the ELI5 way. Retrieved from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[2] http://vision.stanford.edu/aditya86/ImageNetDogs/

[3] https://www.kaggle.com/c/dogs-vs-cats/data

[4] Marcelino, P. (2018, October 23). Transfer learning from pre-trained models. Retrieved from https://towardsdatascience.com/transfer-learning-from- pre-trained-models-f2393f124751

[5] Usage of Optimizers. (n.d.). Retrieved from https://keras.io/optimizers/

[6] Ruder, S. (2020, March 20). An overview of gradient descent optimization algorithms. Retrieved from https://ruder.io/optimizing-gradient-descent/index.html#fn14

[7] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[8] Smolyakov, V. (2018, January 10). Neural Network Optimization Algorithms. Retrieved from https://towardsdatascience.com/neural-network-optimization-algorithms-1a44c282f61d

Figure 1
https://www.easy-tensorflow.com/tf-tutorials/convolutional-neural-nets-cnns

Figure 3
https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751