VGG-Based Convolutional Neural Network on Modified MNIST Dataset

Mini Project 3 - COMP 551: Applied Machine Learning

Alexander Harris: 260688155 Le Nhat Hung: 260793376 Abbas Yadollahi: 260680343

McGill University - March 18, 2019

Abstract

Classifying the MNIST database's handwritten numerical digits from 0-9 is a classic introductory problem in machine learning [15]. We tackle a modified version of the challenge: an input image containing multiple digits of random sizes and orientations on a noisy gray scale background, the goal being to identify the largest sized digit (covering the largest area, not numerical value). Here, we document our experimentation with many CNN variations, and present our best model: an ensemble model based on the VGG16 architecture, achieving an accuracy of 99.7% on the validation set and 97.5% on the Kaggle test set.

1 Introduction

Computer vision, image processing or image recognition techniques aim to extract and identify high level objects or concepts, and are applied for diverse tasks: object recognition, facial recognition, emotion classification, image generation, image caption generation, etc.

A textbook example problem is to classify handwritten numerical digits from 0-9 (e.g. see *Pattern Recognition and Machine Learning*, 2006, page 2; *The Elements of Statistical Learning*, 2008, page 4) [1] [4]. The input images for this problem are generally taken from the Modified National Institute of Standards and Technology (**MNIST**) database. The problem, commonly referred to as the MNIST challenge, is as follows: the inputs are images each with a digit between 0 and 9 in the center, whose size and orientation remains more or less consistent throughout the dataset. The goal is simply to identify the digit.

Our work will tackle a slightly more complex handwritten digit recognition challenge, which we will further describe in Section 3.1. This report will explore and compare the performances of multiple Convolutional Neural Network (CNN) variations based on the **VGG** neural network architecture [13], first as independent models and then as **ensemble models**. Our models' are based on VGG, but actually have fewer layers to accommodate for the fact that our input images only have 1 channel instead of 3, and are much smaller than the images used in the original paper. Our ensemble models all use the Bagging technique, which is suitable as the independent models all share the VGG architecture. We will train models with the Keras deep learning library [2], which provides a high-level neural networks API to a lower-level machine learning backend. For this task we will use the Tensorflow backend [10]. Through Keras, we will experiment with models varying in terms of: **architecture, number of epochs, batch size, optimizer, with or without batch normalization and dropout.** Our models are: VGG6 (6 layer VGG variant), VGG7, VGG9, VGG10, Ensemble3 (ensemble of 3 independent VGG9 models), Ensemble6 and Ensemble10.

1.1 Preliminaries

Convolutional Neural Networks (CNNs) are a class of deep neural networks most commonly applied in computer vision, i.e. image recognition [7]. It has 3 main types of layers [6]:

1) **Convolutional layers** leverage the idea of local connectivity. Each layer neuron connects only to a local region of the preceeding layer. Used primarily for feature extraction.

2) **Pooling/subsampling layers,** commonly inserted between successive convolutional layers, these progressively reduce the spatial size of the representation and thereby the number of parameters and computations in the network. They therefore control overfitting.

3) **Fully connected layers** have full connections to all activations in the previous layer, like in regular neural networks. Typically inserted once at the end of the CNN to output the prediction as a vector of probabilities assigned to all classes.

The **architecture** of a neural network refers to the configurations of its layers: how many, in which order and what types of layers, and for each layer: what are its dimensions, what is the activation function, etc. An **epoch** is when the entire training data is passed forward and backward through the neural network once. As the number of epochs increases, the higher the number of times the weight is changed in the network. The **batch size** is the number of examples in a batch, where a batch is a subset of the training data. Because fitting the whole training data into the neural network is computationally costly for a large dataset, the latter can be divided into batches, able to go through the network one by one.

VGG, or VGG16, is a 16-layer CNN architecture (pooling layers excluded) created by the Visual Geometry Group from the University of Oxford, which achieved a top-5 test accuracy of 92.7% on ImageNet - a dataset of over 14 million images belonging to 1000 classes [13].

An **ensemble** combines the predictions of individually trained classifiers to give its own prediction. Empirical studies have shown ensemble predictions almost always surpass those of individual classifiers [9]. In our case we decided to implement a fairly naive ensembling technique called bagging.

1.2 Important Findings

Regarding CNN configurations, adding batch normalization and dropout dramatically improved performance on both the validation and Kaggle test sets. Next, we found a large batch size does not directly correlate to a higher accuracy. Among the independent models, VGG9 is the best performer in both the validation and Kaggle test sets, whereas with ensemble models, Ensemble10 scored highest in the validation set, but Ensemble6 scored highest in the Kaggle test set.

2 Related Work

The current state of the art accuracy on the standard MNIST dataset is a 0.21% error rate [15], using a Convolutional Neural Network (CNN). This motivated us to build and test CNNs the modified MNIST dataset for this work.

[Karen Simonyan and Andrew Zisserman, 2014] ([13]) presented VGGNet: a CNN which now has the 5^{th} highest accuracy on ImageNet, despite its relatively simple and uniform architecture. This simplicity plays a big role in VGG's appeal, which is what prompted us to base our implementation on the VGG architecture.

[Richard Maclin and David W. Opitz, 2011] ([9]) presented an empirical study of Bagging and Boosting ensembling techniques, tested on 23 datasets, and with the independent models being Feed Forward Neural Networks (FFNN). The results demonstrate that "a Bagging ensemble nearly always outperforms a single classifier." Although this study involved datasets with continuous and/or discrete features, and FFNNs instead of CNNs, we arrived at the same conclusion when testing with CNNs: once our best ensemble model, Ensemble6, was built from our multiple instances of our best standalone model, VGG9, validation accuracy increased from 97.11% to 99.71%, and test accuracy from 97.07% to 97.53%.

[Pavlo M. Radiuk, 2017] ([8]) investigated the influence of batch size on CNN performance on the MNIST and CIFAR-10 datasets. The study concluded the batch size parameter "has a crucial effect on the accuracy of image recognition. The greater the parameter value, the higher the image recognition accuracy." However, that conclusion is challenged here, as we have found on our best model, higher batch sizes led to lower accuracy during validation.

3 Dataset and Setup

3.1 Dataset

We will be working with a modified MNIST dataset of 32,000 images. Each image contains more than one digit and the goal is to find which number occupies the most space in the image, i.e. the number with the largest square bounding box. Each image is represented as a 64×64 matrix of pixel intensity values, i.e. the images are grey-scale not color. See Figure 1 for example inputs.

Our task can be divided into 3 phases:

1) **Create training and validation sets:** done randomly for each model using Scikit-Learns's [11] `train_test_split` function, with an 80:20 train to validation ratio.

2) **Build and train models:** with varying architectures, hyperparameters and optimizers.

3) **Test and compare results of models:** each model is evaluated on a held out validation set as well as Kaggle's test set.

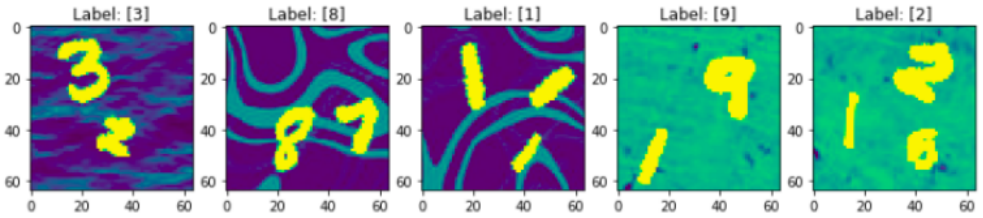


Figure 1: Example input images tackled by our work, and their associated labels. Note that the images are grayscale, but these images were generated using a colormap for improved readability.

3.2 Augmentation

Building a CNN requires a lot of data to be able to properly train a model, but having access to large amounts of data isn't always readily available. The modified MNIST dataset we are using is by no means small, although to help our model generalize better on different image sets, we implemented Keras' ImageDataGenerator which allows us to create new images by modifying our training data on the fly [2]. The idea behind the data generator is to prevent the model from overfitting on the training set, which in turn makes the model more robust and accurate. To do so, we slightly modify the dataset during each epoch by applying a set of transformations so that, to a certain extent, we train on a different set of images. Even though, the set of augmentations we can apply is very small due to digits being very sensitive to transformations. As a result, we only apply minimal changes to the rotation, width and height shift, shear and range of the image.

4 Proposed Approach

Before implementing our own CNN, we read a lot on the VGG16 network since it is known for being accurate on ImageNet and easy to fine tune [3] [13]. When we tried to train our VGG16 model, we realized it wouldn't be possible with our dataset since the network had too many pooling layers and, as a result, it reached a point where convolutions between features were of size 1×1. We decided to build our first network off that knowledge, which gave birth to our VGG6 architecture. The network is fairly simple, sporting 4 Conv layers, 2 max pooling layers and 2 fully connected layers (see Figure 4). Underwhelmed by its performance, we built our VGG7 network (see Figure 5 by adding an additional fully connected layer, 2 dropout layers and by using batch normalization after each set of Conv layers. After seeing an improvement in performance, we continued modifying the model by adding 3 Conv layers, except this time we had a layer of batch normalization after each Conv layer. As a result, we had a total of 7 batch normalization layers and an additional max pooling layer, which we called VGG7 (see Figure 7). We began seeing a small amount of overfitting on

the test set, so with that we decided to remove the last fully connected and dropout layers, which gave us our best independent model, also known as VGG9 (see Figure 6). After continuously modifying our model, our performance began to plateau. As a result, we tried using an ensemble of our best model, being VGG9. We built an ensemble using 3, 6 and 10 separately trained VGG9 models, which proved to improve our performance by a decent margin.

Through many iterations, we realized that applying a layer of batch normalization after each convolutional layer helped regularize our data, which in turn lead to improved results [5]. We also found that using Keras' implementation of dropout layer helped our model generalize better, as can be seen by the accuracy of VGG10 versus that of VGG9 in Table 2 [14].

Model	**Convolutional Layers**	**Max Pooling Layers**	**Batch Normalization Layers**	**Dropout Layers**	**Fully Connected Layers**	**Hidden Layers**
VGG6	4	2	0	0	2	1
VGG7	4	2	2	2	3	2
VGG10	7	3	7	2	3	2
VGG9	7	3	7	1	2	1

Table 1: Architectures of tested models

5 Results

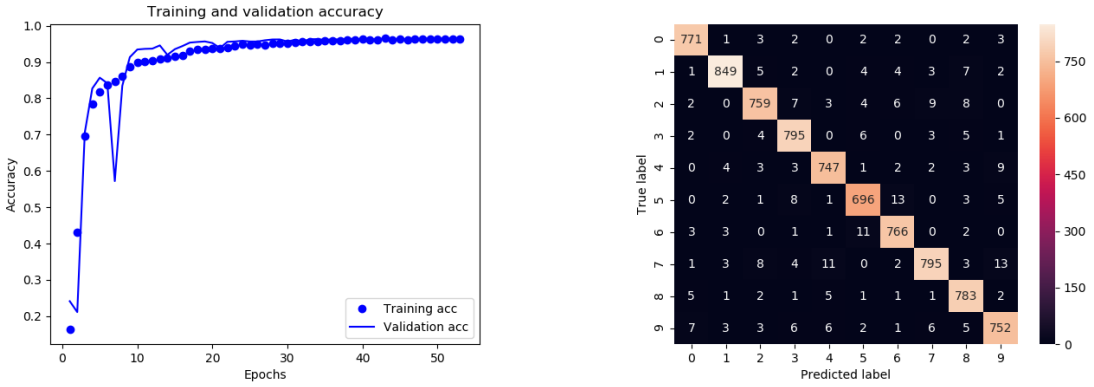
5.1 Model Performance

As can be seen from Table 2, our best performing single model was the VGG9 model with batch normalization and dropout after training for 53 epochs. It was also one of the fastest to train out of the ones we tested, only taking about 17 seconds per epoch on our hardware. The plot of the validation and training accuracy, as well as the confusion matrix for our best model can be found in Figures 2a and 2b respectively.

When we started encountering difficulties with improving this single model, we decided to attempt an ensemble of a few examples of this model in order to see if we could improve our accuracy. Initially, we trained 3 separate versions of our best model by not setting the random seed for Scikit-Learn's `train_test_split`. This ensured each model had a different training/validation split. When we saw that this gave us an increase of over 2% on the validation set and 0.5% on the test set, we tried with even more examples. We found that an ensemble of 6 models gave us our highest score on the test set at 97.53%, with a validation accuracy of 99.71%. Attempting the same with 10 models did increase the validation accuracy by a small amount, however we began to encounter overfitting as this decreased our accuracy on the test set. All models used for the project were trained on a local machine using an Nvidia GTX 1080.

Model	**Training Accuracy**	**Validation Accuracy**	**Test Accuracy**	**Epoch Runtime**	**Parameters**	**Epochs**	**Training Runtime**
VGG6	0.9815	0.9171	0.9077	75	Adadelta, batch_size=128	11	825
VGG7 (+ BatchNorm + Dropout)	0.9842	0.9406	0.9497	530	Adadelta, batch_size=512	10	5300
VGG10 (+ BatchNorm + Dropout)	0.9887	0.9701	0.9613	109	Nadam, batch_size=256	55	5995
VGG9 (+ BatchNorm + Dropout)	0.9630	0.9711	0.9707	17	Adam, batch_size=64	53	901
Ensemble3	N/A	0.9948	0.9750	N/A	N/A	N/A	N/A
Ensemble6	N/A	0.9971	**0.9753**	N/A	N/A	N/A	N/A
Ensemble10	N/A	**0.9973**	0.9747	N/A	N/A	N/A	N/A

Table 2: Results for tested models



(a) VGG9 training and validation accuracy

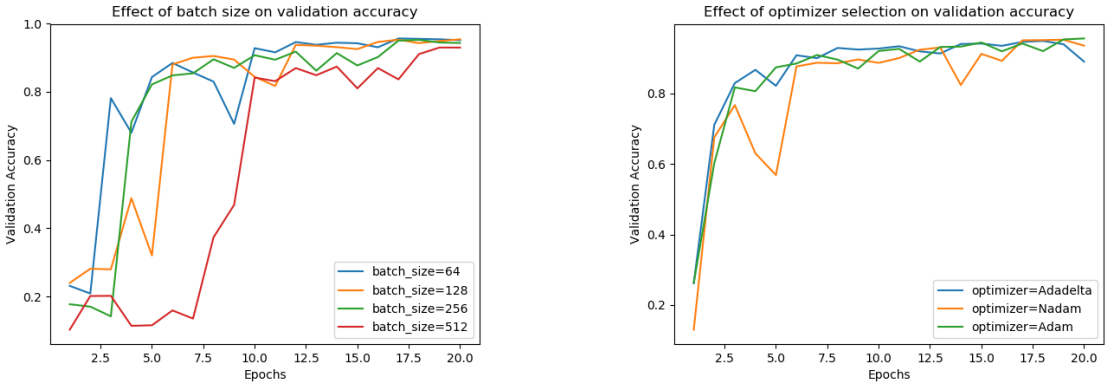
(b) VGG9 confusion matrix

Figure 2: VGG9 test results

5.2 Hyperparameter Effects

When it came to tune hyper-parameters such as batch size and number of epochs, we made a couple of anecdotal observations. Firstly, in our case, increasing batch size appeared to lead to lower scores on the test set, as can been seen from the results obtained from the VGG10 model, which used a batch size of 256. In comparison, our best model, which used a batch size of 64, achieved scores on the test set that were fairly close to its validation accuracy. However, in the former's case, we achieved scores on the test set that were near 1% lower than on the validation set. Additionally, our choice of optimizer seemed to affect not only the total time our model would take to train, but also the final accuracy of the model.

In order to better understand these parameters, we conducted a few experiments on our best-performing single model (VGG9) in order to quantify their effects. We first varied the batch size and trained over 20 epochs. As we can see from Figure 3a, a batch size of 64 appears to give the most consistent performance with our model, with higher batch sizes either taking more epochs or even being unable to reach the same level of accuracy on our validation set. As for different optimizers, we found the Adam optimizer gave us the best performance on our chosen model, as can be seen from Figure 3b. Other optimizers are able to eventually reach the same accuracy, however Nadam was quite unstable in early epochs.



(a) Batch size vs. validation accuracy on VGG9

(b) Optimizer vs. validation accuracy on VGG9

Figure 3: Effects of hyperparameter selection on VGG9

6 Discussion and Conclusion

In this project we explored applying convolutional neural networks towards a multi-class classification problem, in this case a modified version of the MNIST dataset. We researched previous state of the art models on various image datasets, and finally decided to implement a simplified version of VGG16 [13]. We found that it was best to allow our model's convolutional layers handle feature extraction, and decided to only normalize our training data before feeding in into the network. Keras' [2] data augmentation feature helped alot in increasing our model's accuracy given our relatively limited training data of only 40,000 images. We discovered that using techniques such as batch normalization and dropout could help improve our model's ability to generalize to other data as well as combat overfitting.

While designing our model, we found that increasing the dimensionality of the convolutional or hidden layers does not always lead to increased accuracy, and that there is a balance between how many parameters your model has and which hyperparameters you use to train for a particular dataset. Since our dataset contained images that were only 64×64, we had to cut down the original VGG16 model. We experimented with different numbers of convolutional and hidden layers, and found that 3 block of convolutions with max-pooling and batch normalization, followed by one hidden layer with dropout before our softmax layer (for a total of 9) gave us the best performance. We also observed that the choice of optimizer and batch size can effect not only speed of training but also the model's final accuracy.

When we began to encounter difficulties improving our model's accuracy beyond a certain point, we also attempted a simple ensembling technique called Bagging. In this case, we trained a number of examples of our VGG9 model, and then simply averaged the outputs of each to generate our final prediction. We found that this gave us a significant improvement of around 0.5% on the test set over our single model when ensembling 6. However, one characteristic we noticed with this technique was the much higher discrepancy between our validation accuracy and test accuracy. With our single model we achieved almost identical scores on the validation and test sets, however with our ensemble validation accuracy was over 2% higher. We believe this is due to the fact that since each of the models in our ensemble had a different train/validation partition, some models would have trained on examples from another validation set. So while this would have the effect of raising validation accuracy of our ensemble, it does not necessarily properly reflect our ensemble's ability to generalize to other data, as can be seen when comparing to our test set performance.

As for future investigation, one technique that we discussed implementing but did not have the time to do would be to use an object localization model such as Yolo [12] for this task. Our idea was to train Yolo on the regular MNIST dataset, possibly with some data augmentation as we used previously, then predict on the modified dataset to get bounding boxes. We would then find the largest bounding box detected in the image and output its predicted label as our prediction. This is a technique we would like to implement in the future if we revisit this dataset, as we would be interested to see if it would provide comparable performance to a more traditional model.

7 Statement of Contributions

All team members have made significant personal contributions towards this project. The amount of work for each is described as follows:

- Alexander Harris: Model design, training, validation and testing, write-up contribution.
- Abbas Yadollahi: Model design, write-up contribution
- Le Nhat Hung: Model design, write-up contribution.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Francois Chollet et al. *Keras*. https://keras.io. 2015.

[3] William L Hamilton. "Lecture 15 - Convolutional Neural Networks". In: *COMP 551* (Feb. 2019). URL: https://www.cs.mcgill.ca/~wlh/comp551/slides/15-conv_nets.pdf.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. Springer, 2008.

[5] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[6] Andrej Karpathy. "CS231n Convolutional Neural Networks for Visual Recognition". In: (2018). URL: http://cs231n.github.io/convolutional-networks/#convert.

[7] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[8] Pavlo M. Radiuk. "Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets". In: *Information Technology and Management Science* 20 (Dec. 2017). DOI: 10.1515/itms-2017-0003.

[9] Richard Maclin and David W. Opitz. "Popular Ensemble Methods: An Empirical Study". In: *CoRR* abs/1106.0257 (2011). arXiv: 1106.0257. URL: http://arxiv.org/abs/1106.0257.

[10] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python ". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[12] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018). arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767.

[13] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014). URL: http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

[15] Li Wan et al. "Regularization of Neural Networks using DropConnect". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1058–1066. URL: http://proceedings.mlr.press/v28/wan13.html.

8 Appendix

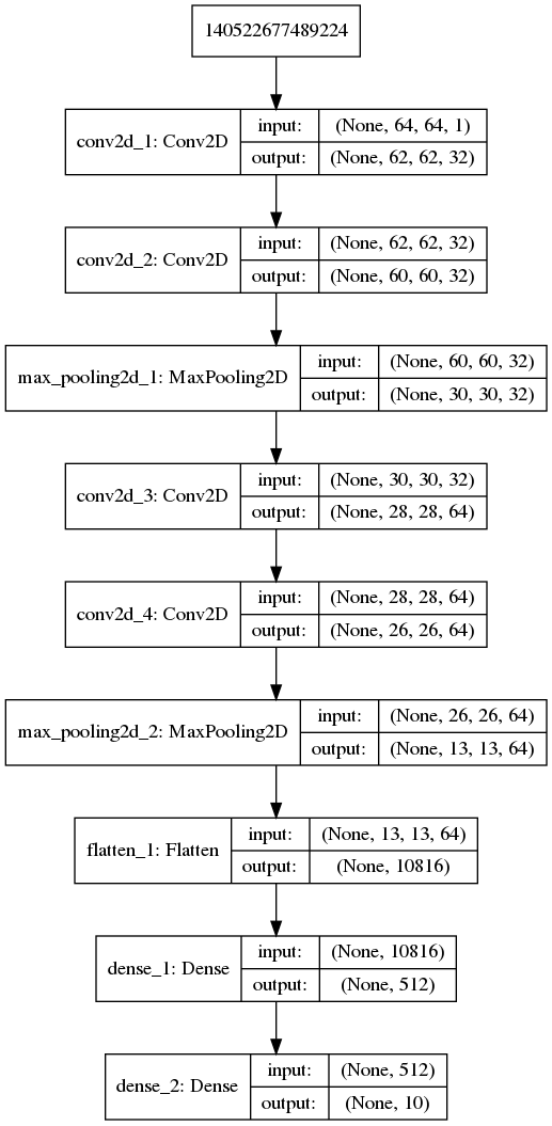


Figure 4: VGG6 Network Architecture

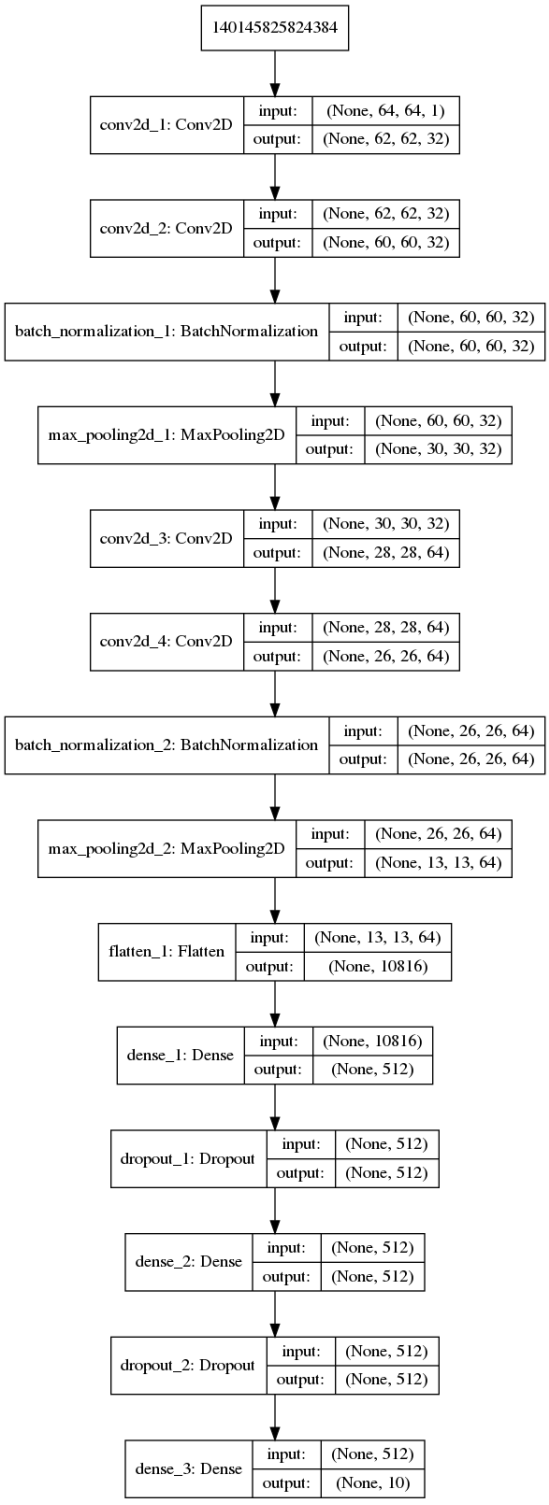


Figure 5: VGG7 Network Architecture

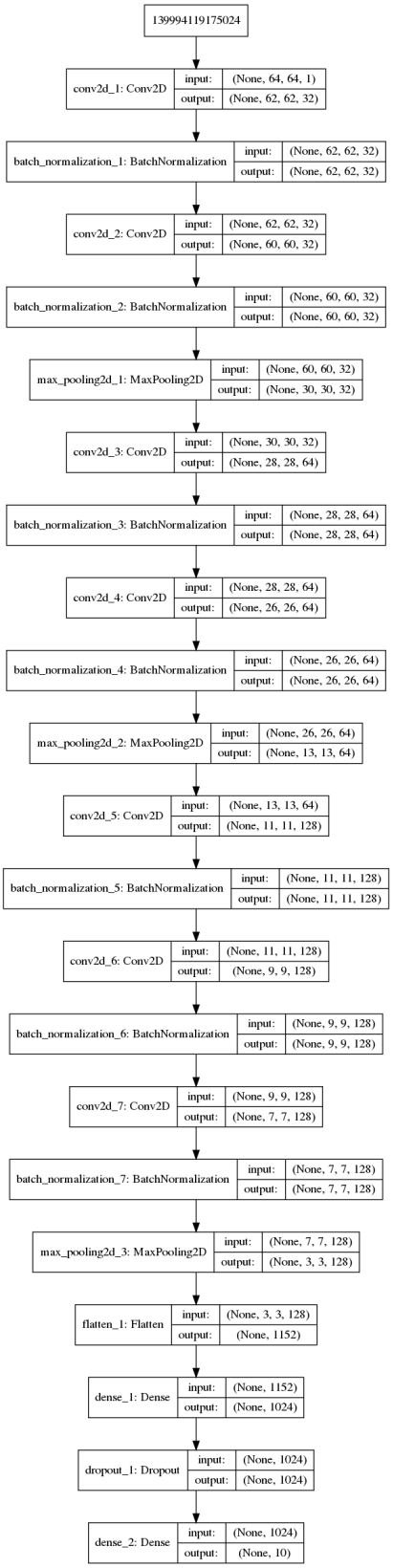


Figure 6: VGG9 Network Architecture

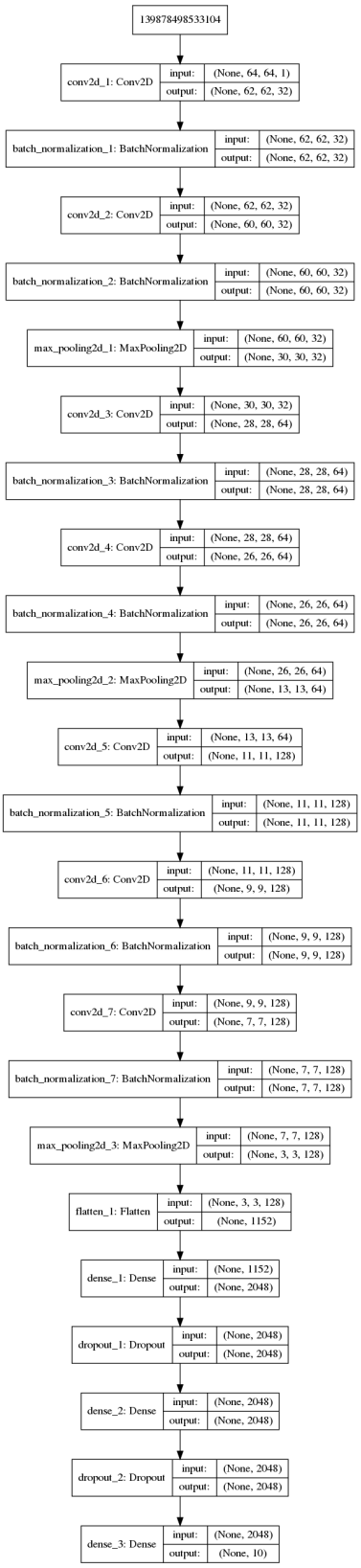


Figure 7: VGG10 Network Architecture