Using Convolutional Neural Networks to Perform Classification on State Farm Insurance Driver Images

Diveesh Singh
Stanford University
650 Serra Mall, Stanford, CA
diveesh@stanford.edu

Abstract

For the State Farm Photo Classification Kaggle Challenge, we use two different Convolutional Neural Network models to classify pictures of drivers in their cars. The first is a model trained from scratch on the provided dataset, and the other is a model that was first pretrained on the ImageNet dataset and then underwent transfer learning on the provided StateFarm dataset. With the first approach, we achieved a validation accuracy of 10.9%, which is not much better than random. However, with the second approach, we achieved an accuracy of 21.1%. Finally, we explore ways to make these models better based on past models and training techniques.

1. Introduction

This paper will discuss using Convolutional Neural Networks (CNNs) to perform deep learning on an image dataset provided by State Farm Insurance. State Farm has provided a dataset that contains several pictures of drivers in their cars performing specific actions, detailed later in the paper. Our task is to be able to classify these images based on the action that the driver is performing. Many of these images show drivers performing actions that are unsafe while driving a car. The motivation behind this problem stems from State Farm's need to understand what kinds of behaviors drivers are engaging in so that they can set their insurance policies accordingly. For example, if texting and driving is a common cause for accidents, which it often is, insurance companies can increase their premium for drivers that are more likely to be texting and driving. By analyzing this data, we will not only learn what trends drivers tend to exhibit, but also be able to automate the process of pinpointing the actions that lead to accidents.

2. Previous Work and Background

Convolutional neural networks is where machine learning meets computer vision to tackle the problem of object detection. Every month, the state-of-the-art in convolutional neural gets pushed further and further. Currently, the leading model is the Inception-v4 model, that achieved a 3.08% top error rate on the ImageNet dataset; the Inception-v4 model has 75 trainable layers. These kinds of models take about 2 weeks to train, along with a substantial amount of computing power.
Another top framework in this field was proposed by Girshick *et al.*; this framework divides classification into two steps, where the first step performs object detection and the second step performs CNN recognition. Along with the two methods listed above, there are several other ways to modify and leverage CNNs in order to perform object recognition tasks [1]. Because of their previous success, we decided to explore their performance in our own visual detection task.

3. Techincal Approach

3.1. Dataset

The dataset contains about 4 GB of photos of drivers in their cars, which translates to approximately 25,000 pictures. Each driver has several images associated with them, and the images themselves are divided into 10 classes, which leads to about 2,500 images per class. The classes are as follows:

1. c0: Safe Driving
2. c1: Texting - right
3. c2: Talking on the phone - right
4. c3: Texting - left
5. c4: Talking on the phone - left
6. c5: Operating the radio

7. c6: Drinking
8. c7: Reaching behind
9. c8: Hair and Makeup
10. c9: Talking to passenger

Some examples from the dataset can be seen below

For the purposes of this paper, we divide up our dataset into a training set and a validation set. 80% of our dataset will make up our training set, and 20% will makeup the validation set.

3.2. Preprocessing

To ensure that the data is primed for usage in a convolutional neural network, we shrink each image to 224x224 pixels. This size was selected based on the default values that are used in a popular Convolutional Neural Network model known as VGGNet-19. Resizing of the images was done by a simple Python script.

3.3. Classification

For performing actual classification, there are various types of architectures and training techniques we can use. We will evaluate each different architecture and compare the results to see which architecture performs best. Also, within each architecture/technique, there are various different hyperparameters we can adjust in order to fine tune the model.

3.3.1 Approach 1: Small Network

The first architecture involves using a small, 10-layer CNN that contains 8 convolutional layers and 2 fully-connected layers at the end. The output of the final layer is an individual probability vector for each image, where each entry i in the vector indicates the predicted probability of that image being of category i

$$[0.1, 0.1, 0.7, 0, 0, 0, 0, 0.05, 0.05, 0]$$

To generate this probability vector, we will be applying a softmax activation to the final layer of the network. The probabilistic interpretation of the softmax classifier is as follows

$$P(y_i|x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$$

With convolutional neural networks, there are several different update rules one can use to update each of the weight matrices, bias vectors and convolutional filters. The update rule that has been known to work best in practice is known as the Adam update rule, which consists of the following:

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x -= learning_rate * m / (np.sqrt(v) + eps)
```

The hyperparameters that we set are the following

1. Learning rate = 1×10^{-7}
2. Beta1 decay = 0.9
3. Beta2 decay = 0.99
4. Weight initializer = Gaussian

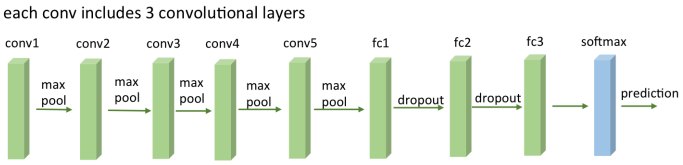
The purpose of using this architecture was to provide a baseline for how well a convolutional neural network can perform on this type of dataset.

3.3.2 Approach 2: Transfer Learning on VGG Network

This next architecture leverages an existing architecture that has been proven successful in the past, known as VGGNet-19. VGGNet-19 has been trained on the ImageNet dataset, and therefore is good at recognizing images in that dataset. However, we want our model to be able to distinguish the images in our dataset; to do this, we use a process called transfer learning. Transfer learning is the process of training only some of the layers in a CNN on a new dataset, while leaving the majority of the layers untouched. In our case, we can preload the pretrained weights of the VGGNet-19 model (found at https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3). Then, we can train the top layers on our State Farm dataset, without affecting the rest of the layers. By doing this, we get the advantage of using a pretrained model that can perform basic feature extraction (i.e edge detection, SIFT descriptors etc.) in the convolutional layers of the network, and then the final, fully-connected layers can use these features to make classifications specific to our problem. The hyperparameters that we set are the following

1. Learning rate = 1×10^{-9}
2. Beta1 decay = 0.9
3. Beta2 decay = 0.99
4. Weight initializer = VGGNet-19 weights

A visual of the model is provided below [3]



3.4. Technical Resources, Limitations and Solutions

Due to the nature of Convolutional Neural Networks, we need a significant amount of computing power and time in order to successfully train a model to the point where it can be used in practice. For example, training the Inception-v4 model took 2 weeks to train on 8 NVIDIA Tesla K40s, which are some of the best GPUs currently in the market. For the purposes of this paper, we created a GPU instance on Amazon AWS which had a reasonable amount of processing power and memory to perform the robust calculations required to train a convolutional neural network. However, it was not robust enough to handle a large training set, which is why we had to limit the number of training examples to 10,000. To get around this, we trained each model with the first 10,000 pictures for 25 epochs and noted down the results. Then we took that model, and trained it on a completely new set of 10,000 pictures for 25 epochs. Then finally, we took the remaining 5,000 images and trained the model for 25 epochs. While this was not ideally how the model would have been trained, given the computing resources we had, it provided us with a way to expose the model to all of the images in order to make it more generalized.

4. Results

4.1. Evaluation Metric

We measure the quality of our approach based on a loss function (the higher the loss, the lower the accuracy). The loss function used for accuracy is as follows

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij})$$

where N is the number of images in the test set, M is the number of image class labels, and y_{ij} is the true value of example i for attribute j, as posted on the Kaggle website.

The loss function above is essentially identical to the categorical cross-entropy loss function, so it was a straightforward choice to minimize that loss function during gradient descent.
However, intuitively, this loss function does not provide an intuitive evaluation of the model. Instead, we establish our own criterion for evaluating the model. As mentioned earlier, every training example outputs a vector of the following form

$$[0.1, 0.1, 0.7, 0, 0, 0, 0, 0.05, 0.05, 0]$$

Because each training example belongs to one class, we assign the training example to the class whose probability is the highest and generate a vector of binary values. Then, we compare the class assigned by the classifier and its true class to evaluate the accuracy. The official accuracy metric used

$$acc = \frac{numCorrectPredictions}{numExamples}$$

4.2. Loss and Accuracy Results

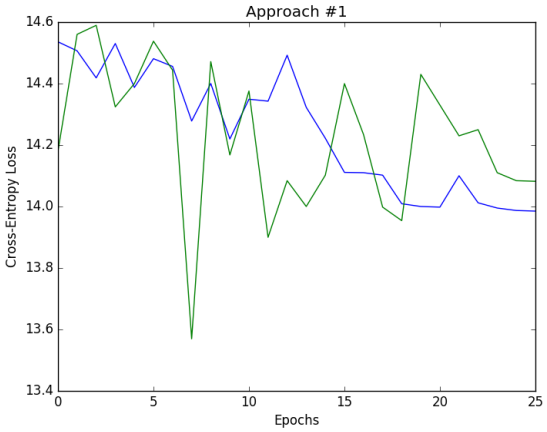
We trained both of the networks described earlier (a network trained from scratch, and a pretrained network with transfer learning performed on it). As detailed earlier, we had to train our model 3 spurts, twice with approximately 10,000 images and the final time with approximately 5,000 images due to technical limitations. Once we trained the whole model on these images, we evaluated our model by taking the 100 images that were not used in the training set and computing the categorical cross-entropy across all these examples. The training loss after 25 epochs of training on the first 10,000 examples was

$$loss_{train} = 13.985$$

and the validation loss was

$$loss_{val} = 14.082$$

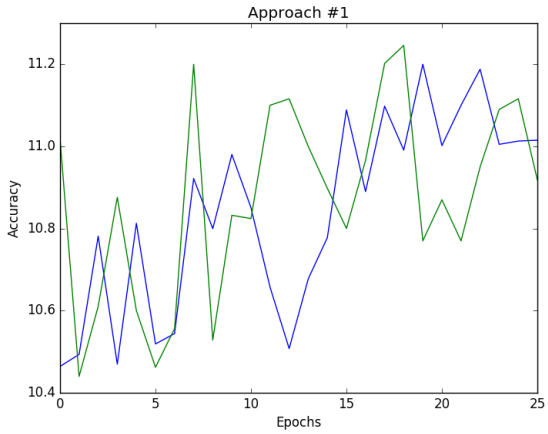
The graph is as follows



While we only trained for 25 epochs, it seemed that the loss still had not converged and required several more epochs for it to fully converge. This makes sense because we are training this model from scratch, and so it requires several epochs to actually understand the trends in the data. After exposing the model to the rest of the training set in chunks, we achieved losses of

$$loss_{train} = 11.235$$
$$loss_{val} = 12.111$$

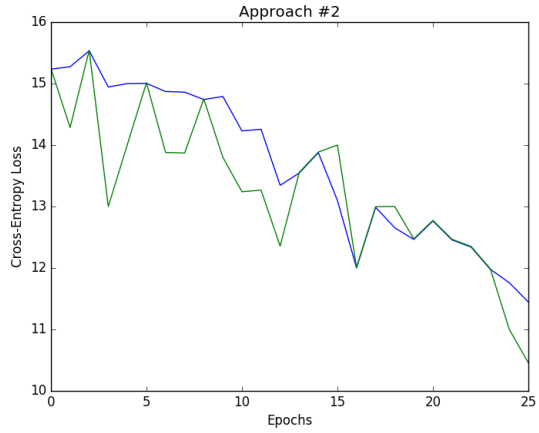
Now, with a trained model, we can evaluate the accuracy on a dataset of 100 images. The accuracy was 10.9%. Given that there are 10 possible classes for a training example, random guessing expects to achieve 10% accuracy. We explore reasons as to why this was the case in **Section 5**. We can also observe how the both the training accuracy and validation accuracy change as we train the model.



For the second architecture (VGGNet-19 with Transfer learning), the losses after 25 epochs on 10,000 examples was

$$loss_{train} = 11.445$$
$$loss_{val} = 11.923$$

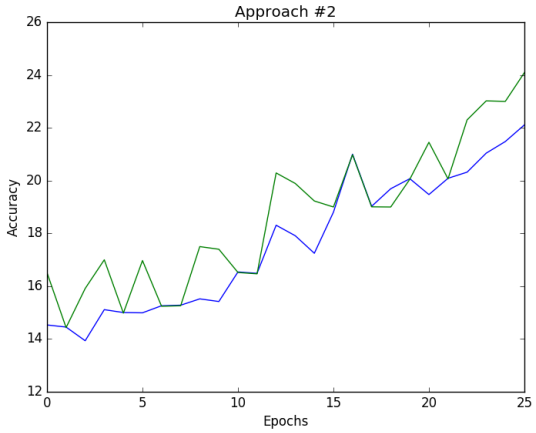
The graph of the loss is as follows



After exposing the model to all 25,000 images, we achieved losses of

$$loss_{train} = 10.905$$
$$loss_{val} = 11.421$$

After performing transfer learning on this model, we achieved a training accuracy of 21.1% and a validation accuracy of 24%. Clearly, this is better than random guessing, so our model was able to understand the trends in our data.



We analyze the reason for this in **Section 5**

5. Analysis

5.1. Dataset

A significant challenge that comes with trying to design an accurate model is overcoming issues with the dataset. While there wasn't much noise in the dataset (i.e images that were not of a driver), a lot of the images were very similar because they were all of a driver sitting in a car, so it was difficult for a model to learn appropriate distinctions. Specifically, the top layers of the network are responsible for learning what features contribute to an image's class. Some of the distinctions that were especially difficult for the model to learn were c8 (Hair and Makeup), c2 and c4 (Talking on the phone - left, right), due to how similar they were. While CNNs are generally able to implicitly learn subtle features like hand placement, facial recognition, and light-dark patches, these 3 categories were very similar in the sense that all of them involved having the driver's hand near their face. A potential solution could be to add more fully connected layers, as adding more parameters gives the model the ability to make more subtle distinctions. Perhaps allowing for more overfitting in the fully connected layers of the network would help.

Also, there is a general issue that comes up while doing transfer learning on a pretrained model when using your own dataset. Our dataset was exclusively pictures of people driving, whereas the pretrained model understood features

of all 1000 classes of the ImageNet dataset. Intuitively, it does not make sense that our model need to understand aspects of all 1000 classes, but should devote more of its attention to making the fine-grain distinctions between drivers.

5.2. Results

With the first approach, the results were unfortunately very close to that of random guessing. This can likely be attributed to the fact that the neural network did not have enough time to train on the existing dataset. As mentioned earlier, most successfuly neural networks that are trained from scratch require on the order of 2 weeks to train fully on very powerful GPUs. Regardless, it provided a guideline for how to design the second approach, as the main disadvantage with this first approach was the amount of computing resources needed to train this model. Given the availability of more resources, however, it would be worthwhile to try training for much longer, and decaying the learning rate over time. This is because when training begins, the learning rate needs to be large so that the updates on the weight matrix can happen on a larger scale; however, as the model gets closer and closer to convergence, a smaller learning rate is required as that will allow the model to fine-tune itself.

With the second approach, we were able to achieve an accuracy that was substantially better than random guessing, which meant that our model was able to learn some trends in our data. This is likely due to the fact that our pretrained model could already perform basic object recognition, so the entire training time could be devoted to learning the features of our dataset and adjusting the weight matrices accordingly. Something interesting to note, however, is that in most cases, the validation accuracy tends to be lower than the training accuracy (as the model has seen the examples that are used to calculate the training accuracy). Generally, when the validation accuracy is significantly lower than the training accuracy, it is a result of overfitting. However, in our case, the validation accuracy was higher than the training loss; the higher the validation accuracy, the more generalized the model is. The fact that the training accuracy was lower than the validation accuracy indicates that the model may have high variance, as its ability to recall images that had been seen before was weak. To fix this, it would be worthwhile experimenting with different learning rates and training the model for a much longer time. One of the causes of overfitting is training the model for too long; however, in this case, it was probably necessary to train the model for longer if the validation accuracy was higher than the training accuracy.

6. Further Improvements

There are several improvements that can be made to the above approaches. It would be interesting to explore the effects of different optimizing techniques (i.e. adaGrad, adaDelta, vanilla SGD etc.) to see how the speed of convergence changes and how it affects the accuracy. Also, we can try different nonlinearities after each layer of the network (leaky ReLU as opposed to regular ReLU) and see how that affects the above metrics.

One of the biggest drawbacks to using CNNs is the amount of time it takes to train one to the point where its usable; therefore computational efficiency is of utmost importance. Specifically with the first approach, training that network till convergence would have taken much more time and computing resources. Experimenting with fewer layers in this model is a potential way to reduce the amount of time it would take and also reduce the potential for overfitting. However, since the model was nowhere close to convergence, it is difficult to evaluate whether the model would have high bias or high variance given more time to train. Specifically for transfer learning on the second approach, it would make sense to try training more or fewer layers and see how that affects the rate of convergence and overall accuracy. Most of the time in transfer learning, we only train the fully connected layers becaue the convolutional layers act as feature extractors, but there have been advantages in also training the final convolutional layer, so this would be an avenue worth exploring.

As mentioned earlier, the dataset was not only a set of individual pictures, but could actually be grouped into sequences of frames from a video. While Convolutional Neural Networks do not normally work on videos by default, there have been some techniques researched by researchers here at Stanford that make classification of videos possible. Some of the techniques detailed in the paper are Early Fusion, Late Fusion, and Slow Fusion[2]. While these techniques are distinct, they all share the same general approach: while looking at a frame, it takes into account the previous frames and the following frames to incorporate the movement going on in the video in the process of classification.

In future work, we hope to evaluate other types of models, including ones that do not involve deep learning and compare the results; taking the predictions of several models and averaging them to come up with a more robust prediction would also be a productive avenue. In addition, we hope to design models that can take the temporal dimension into account with the techniques mentioned above to extract more information out of the dataset.

References

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Regionbased convolutional networks for accurate object detection and segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 2015.

[2] A. Karpathy et al. Large Scale Video Classification with Convolutional Neural Networks. CVPR, 2014.

[3] Z. Yan, H. Zhang, R. Piramuthu, V. Jagadeesh, D. DeCoste, W. Di, Y. Yu. HD-CNN: Hierarchical Deep Convolutional Neural Network for Large Scale Visual Recognition. ICCV, 2015.