# Final Project - Practicing CNN with CIFAR-10

**Isidora Conic**

**CS6140 - Machine Learning**
**Fall 2021, Professor Craig Martell**

## Assignment:

The aim of this final project was to practice using, creating, and modelling CNNs to then perform image detection (predict the class of an image) with the CIFAR-10 dataset. The CIFAR-10 dataset has 60,000 32x32x3 images, which are already split into a 50,000 image training set and a 10,000 testing set. There are 10 categories of labels (classifications of the images), which include airplane, automobile, bird, cat, deepr, dog, frog, horse, ship, and truck. For this assignment, we were allowed to use any architecture, with or without regularization.

## Library Used:

For the assignment, I used Keras and TensorFlow to build up CNNs from scratch. I also used keras_applications to import pre-made pre-trained models for transfer learning. For the pre-trained weights, I used the ImageNet dataset.

## Different CNN Architectures Attempted:

Before even coding up any CNNs, I first split the whole dataset into a test and train split (10,000 and 50,000 images, respectively, as mentioned above). I then took the training test, and further split that into a 30/70 test/train split, for the initial modelling and testing. I would then use the initial test dataset, with 10,000 images, to test the accuracy of my final model.

### *(1) LeNet-5*

I started with a simple model, called the LeNet-5 architecture. I built this model up from scratch. This is a very basic and classic architecture model which was originally designed in 1990 to classify handwritten digits. It has two convolutional layers + average pooling layers on top of each other. After this, the inputs are fed into a flattening convolutional layer, two fully-connected layers, and then to perform the final classification, a softmax classifier layer.[1]

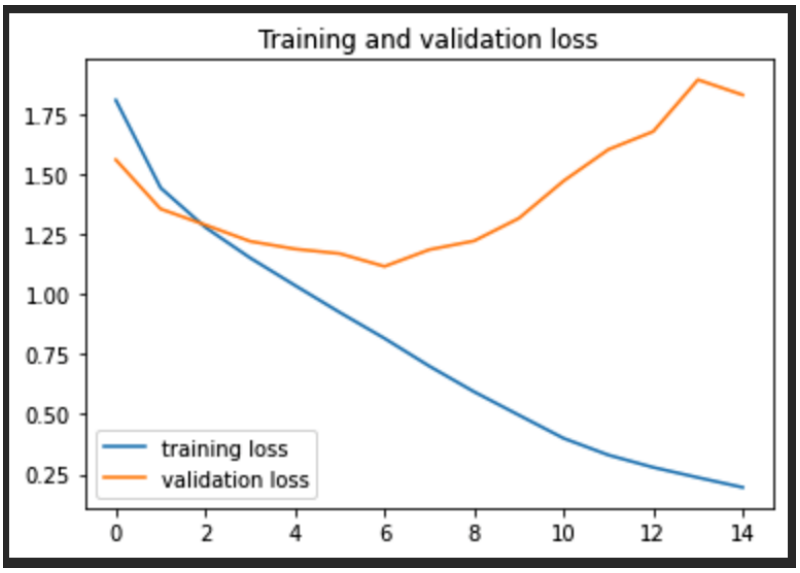
I used a relu activation function, and varying numbers of filters in the different layers, ranging from 64 to 256 (all powers of 2) with a kernel size of (3,3). I generally stuck with these numbers throughout the assignment, varying different combinations in the interest of improving the accuracy of the model.

The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier. I coded up these layers first with no regularization, and the model had a performance of around 60%.

[1] https://www.datasciencecentral.com/profiles/blogs/lenet-5-a-classic-cnn-architecture

For reference in terms of training and validation loss curves:

- Training loss = how well the model is fitting the training data
- Validation loss = how well the model fits new data



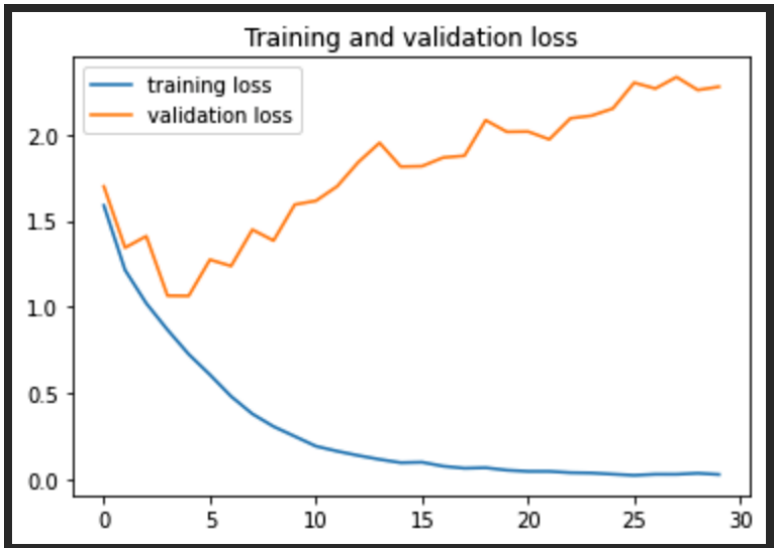

Above, we can see that initially the model is heavily overfitting after about 2 epochs. This isn't much of a surprise since we have not included any regularization.

While this is an okay starting point, there was a lot of performance improvement that could be added with regularization. I performed the following regularization with the following results:

1. **Batch Normalization**: I first added batch normalization layers to the original LeNet-5 model. This served to standardize the inputs into a layer in order to reduce instability in the distributions of layer activation and the internal covariate shifts in a CNN.[2]
    a. The result was an improvement in accuracy of classification of about 5%, to bump the CNN up to an accuracy of 65%.
    b. Below, in the training and validation loss curve, we can see that now the overfitting only begins after about 5 epochs, which is better, but still seems like the model requires some additional regularization.

---

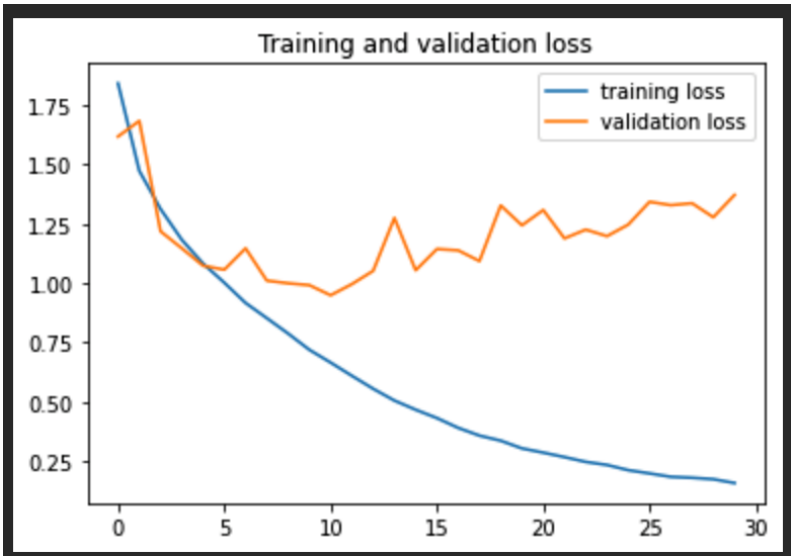
2 https://analyticsindiamag.com/hands-on-guide-to-implement-batch-normalization-in-deep-learning-models/




*Training and Validation Loss for LeNet-5 with Batch Normalization*

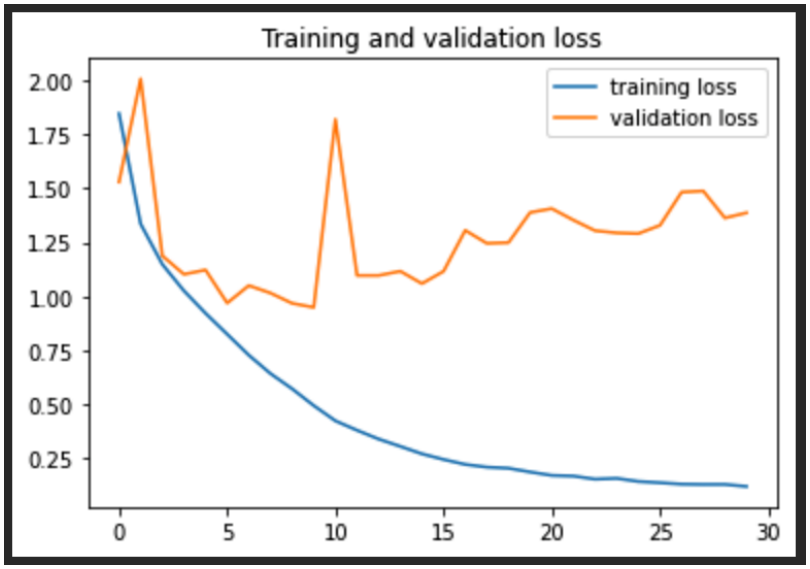
2. **Dropout Layer**s: After the batch normalization layers, I added dropout layers. Dropout layers serve to reduce overfitting of the model, by making the training process noisy. They allow having a number of layer outputs to be randomly ignored or dropped, which simulates a layer as having a different number of nodes and connectivity to the previous layer.[3]
    a. I added two dropout layers, the first one with 25% dropout (0.25), and the second one with 50% dropout (0.50).
    b. This served to improve the accuracy of the model by another 5%, to bump the CNN up to an accuracy of approximately 70%.
    c. Further, we can see below in the training and validation loss curve, the validation loss is less than that of the model with only batch normalization. With only batch normalization, it was reaching approximately 2.2 after 30 epochs, but with the addition of the dropout layers, the validation loss is only about 1.25. This is an improvement.

[3] https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/



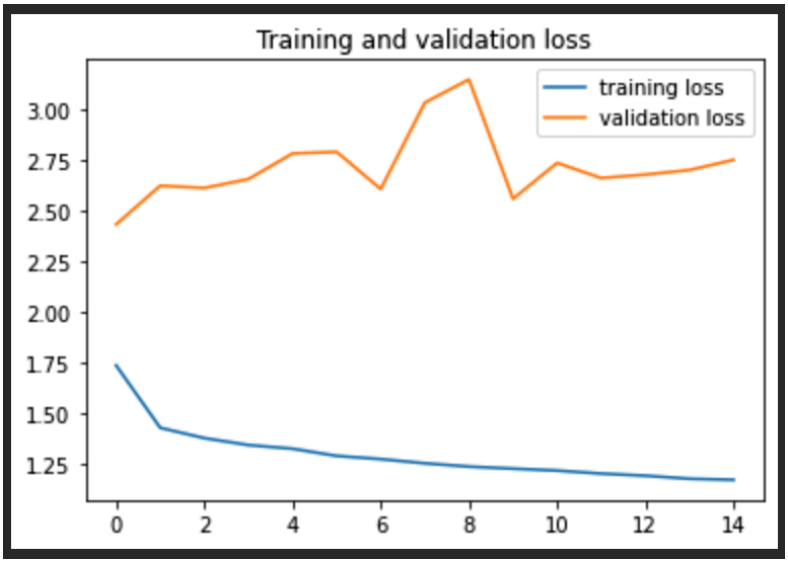

*Training and Validation Loss for LeNet-5 with Dropout Layers and Batch Normalization*

3. **Adam Optimizer:** While this isn't exactly a regularization technique, I also tried to use a different optimizer after adding the batch normalization and dropout layers. In this case, I used the Adam optimizer with a learning rate of 0.001.
    a. This didn't have much of an effect on the accuracy of the model, since it stayed around 70%.
    b. It seems like the Adam optimizer didn't change the model's overfitting problem too much from before (when looking at the training and validation loss curves). The values are about the same after 30 epochs. Further, this could suggest that the learning rate is too fast (although I picked a fairly standard value).




*Training and Validation Loss for LeNet-5 with Dropout Layers, Batch Normalization with Adam optimizer*

4. **Data Augmentation:** Finally, I also performed data augmentation. This serves to create a more robust and generate an artificially larger dataset, by taking the images we are already working with, and transforming them (flips, vertical and horizontal shifts, brightening, zooming, pixel shifts, etc.). This was performed by splitting the data I was working with already, applying these transformations using the ImageDataGenerator, and then fitting the model on them.
    a. This actually made the accuracy of the model much worse, around 10%. I suspect that this is because the transformed images aren't actually representative of the dataset. In other words, if we modify an image, it starts to not even look like the original object, and then might serve to actually "confuse" the model instead of augmenting it.
    b. Based on the training and validation loss curves below, we can confirm that point (a) is correct. It seems that the model was unable to learn the training set well, and this would make sense, since I suspect that I added images that make no sense when I performed this data augmentation with so many transformations.




*Training and Validation Loss for LeNet-5 with Dropout Layers, Batch Normalization, and Data Augmentation with Adam optimizer*

Final Comments: In order to improve this model, there are a couple things that could be done. First, try the model with an even smaller learning curve, since the overfitting and divergence of the training loss and validation loss curves that we are seeing could be caused by this. Secondly, when performing data augmentation, we could augment that data with less transformations, in order to have the resulting images be more "normal" and representative of the rest of the given training images. Since this CNN is very simple, I will implement these changes in the more complex model below.

***(2) Yann LeCun Architecture***

The next model architecture I tried was that of Yann LeCun (who was also responsible for the LeNet-5 architecture). This architecture is somewhat similar to the CNN above, but I included a third convolutional layer, as well as pooling layers. Firstly, the addition of an additional

convolutional layer was in the hopes of extracting more features. Then, the pooling layers were added to down sample the resulting feature maps (to summarize the presence of features in patches of the feature map).[4]

The model summary is:

```
Model: "sequential_34"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_66 (Conv2D)          (None, 30, 30, 32)        896

 max_pooling2d_18 (MaxPoolin  (None, 15, 15, 32)       0
 g2D)

 batch_normalization_98 (Bat  (None, 15, 15, 32)       128
 chNormalization)

 conv2d_67 (Conv2D)          (None, 13, 13, 64)        18496

 max_pooling2d_19 (MaxPoolin  (None, 6, 6, 64)         0
 g2D)

 dropout_57 (Dropout)        (None, 6, 6, 64)          0

 batch_normalization_99 (Bat  (None, 6, 6, 64)         256
 chNormalization)

 conv2d_68 (Conv2D)          (None, 4, 4, 128)         73856

 max_pooling2d_20 (MaxPoolin  (None, 2, 2, 128)        0
 g2D)

 dropout_58 (Dropout)        (None, 2, 2, 128)         0

 batch_normalization_100 (Ba  (None, 2, 2, 128)        512
 tchNormalization)

 flatten_32 (Flatten)        (None, 512)               0

 dense_99 (Dense)            (None, 256)               131328

 dense_100 (Dense)           (None, 120)               30840

 dense_101 (Dense)           (None, 10)                1210
```

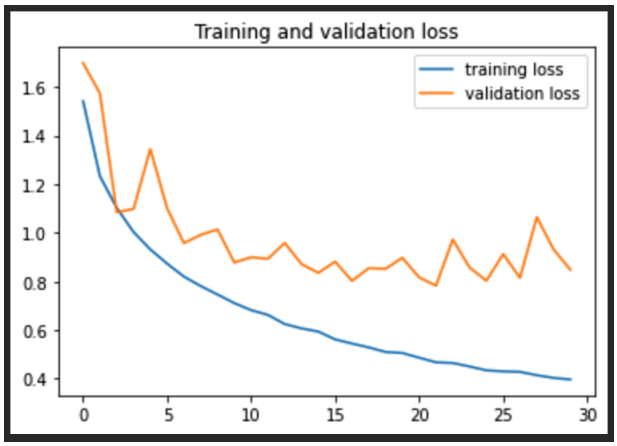
[4] https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

```
=================================================================
Total params: 257,522
Trainable params: 257,074
Non-trainable params: 448
_________________________________________________________________
```

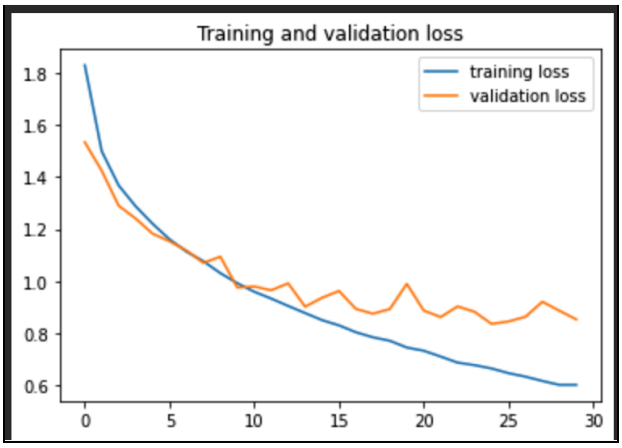
Next, to account for some of the insights from the LeNet-5 model, I added in batch normalization, dropout layers, and used the Adam optimizer. With this configuration, I tried two different learning rates to account for the issues in the previous CNN model. The results were:

- Learning rate = 0.001 → Accuracy = approx. 74%
- Learning rate = 0.0001 → Accuracy = approx. 71%

The training and validation loss curves were:




*Training and Validation Loss for CNN #2 (LeCun) with Learning Rate of **0.001***




*Training and Validation Loss for CNN #2 (LeCun) with Learning Rate of **0.0001***

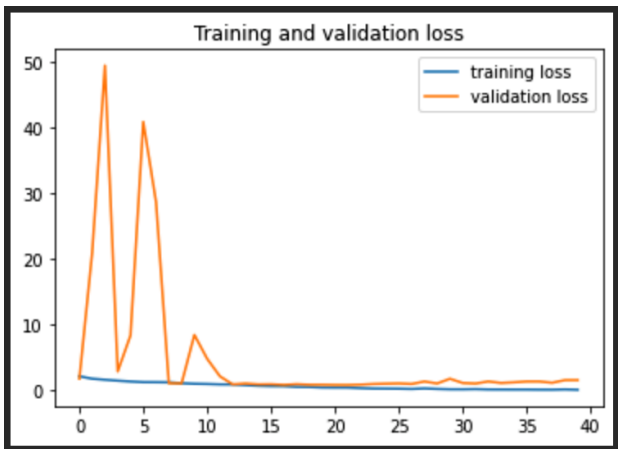
While it seems from the training and validation loss curves that the lower learning rate performed better, the higher learning rate had a better accuracy when predicting.

***(3) Transfer Learning with ResNet50***

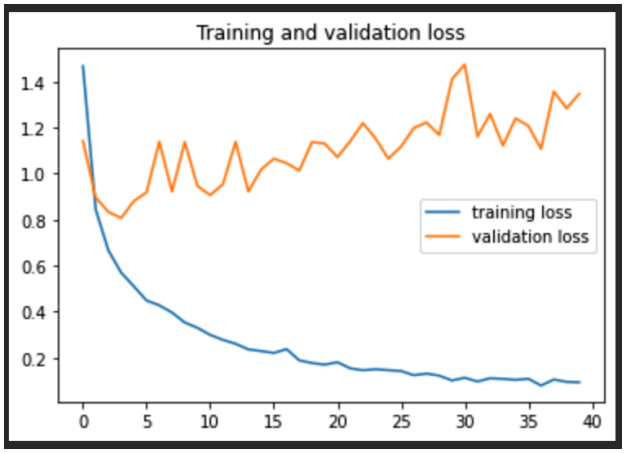
In the last part of my assignment, I tried to implement transfer learning with ResNet50. ResNet50 is CNN with 50 layers; 48 convolutional layers, 1 max pool layer, and 1 average pooling layer. It is a very popular deep learning model, and performs very well with image classification. Further, I wanted to attempt transfer learning, since I could then use pre-trained weights for a ResNet50 model trained on the ImageNet dataset, then freeze the top layers, and then add layers for both feature extraction and fine tuning of my own dataset (CIFAR-10). The hypothesis here was that the model would already be good at image classification based on the training of most of the layers from the ImageNet dataset, but then it would be improved for the CIFAR-10 dataset through the feature extraction and fine tuning done on the CIFAR-10 dataset.

The steps to do this were as follows:

1. Create base model with ResNet50 pre-trained on ImageNet
    a. Set include_top=False in order to add layers trained on CIFAR-10 after.
2. Freeze all layers of the ResNet50 base for **feature extraction**.
3. Add a data preprocessing layer and try out data augmentation layer.
4. Add dense layers with batch normalization and dropout layers.
5. Add classification layer (softmax).
6. Train the CNN.
7. Unfreeze ResNet50 base layers for **fine tuning**.
8. Train the CNN.
9. Predict.

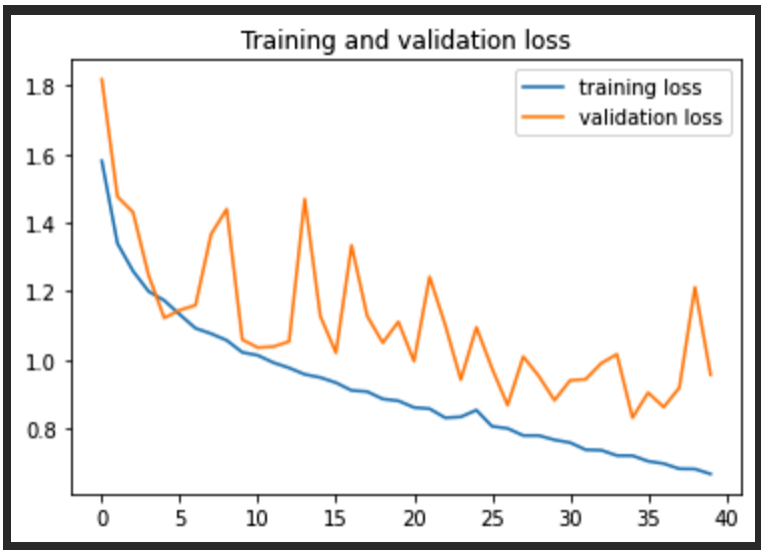



*Training and Validation Loss for ResNet50 Transfer Learning CNN with Batch Size of 32*




*Training and Validation Loss for ResNet50 Transfer Learning CNN with Batch Size of 128*

Then, I also added data augmentation as the first additional layer to feed into the model, to see how it would affect it (I was also using a batch size of 128 for this).

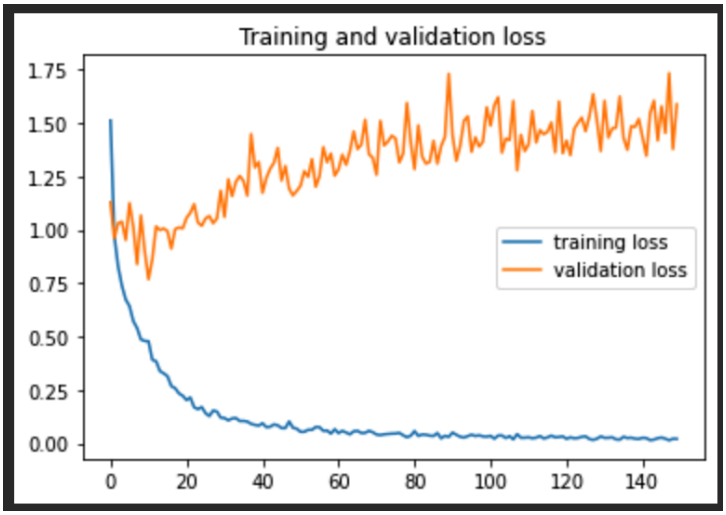



*Training and Validation Loss for ResNet50 Transfer Learning CNN with Batch Size of 128 with Data Augmentation*

Note that in the interest of time, the CNNs above were run (fit) with 40 epochs, but for my final results I did it with 150 epochs (and a batch size of 128).

## Final Results:

For my final submission, I decided to use my third model; the ResNet50 transfer learning model. In this case, I trained/fit the second part of the model (after unfreezing layers, during fine-tuning) with 150 epochs. I used this model to evaluate the classification on the test dataset that I had set aside at the very beginning of the code (the 10,000 images), and it performed with an accuracy of about 0.7785.




*Training and Validation Loss for ResNet50 Transfer Learning CNN with Batch Size of 64 with 150 Epochs*

However, it looks like the model is overfitting quite a bit. This is something that should be tuned in the future.

Some other final thoughts for future optimizations of all/any of the CNNs I used would be to further explore different learning rates (I even saw some instances online where people varied their learning rates throughout different layers of the CNN). Also, I would explore different batch sizes and how that would affect my models. Finally, I would also look into better data augmentation; it seems as though every time I tried it in my assignment, it didn't help any of my CNNs, when in theory, it should have.

## Other notes:

I also played around with a VGG16 CNN architecture and pre-trained model. I was having trouble implementing the transfer learning so was hoping that trying another model might work. Further, I was constantly trying to increase the accuracy of whichever CNN I was working with and wanted to try a handful of different types. I have included those files in my submission as well, but please note that they are not cleaned up and commented properly, and code was taken from various sources and I do not claim it as my own. Finally, the transfer learning VGG16 file doesn't even run in the end.