Capstone 2, Milestone Report 2

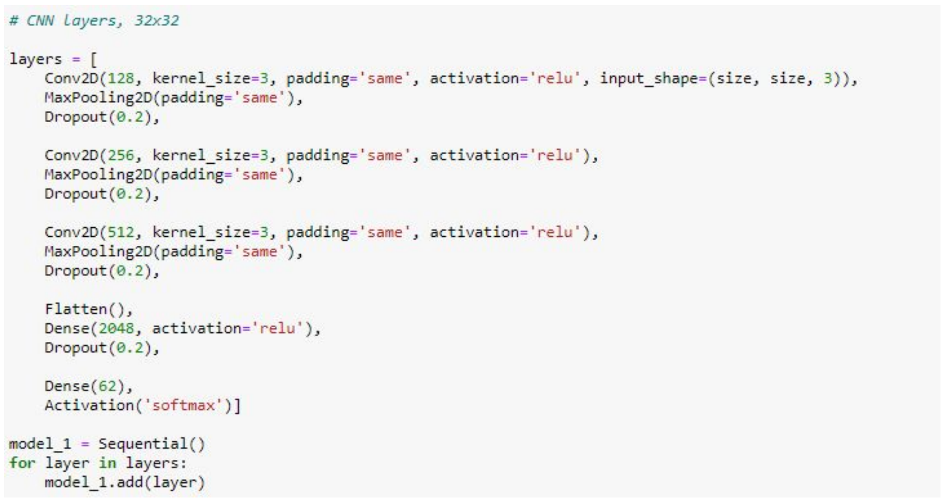
Overview

A technique central to my strategy in this project is progressive resizing, which is an application of transfer learning. Transfer learning involves using the layers and weights from a pre-trained model as the basis for a new model. When applied to progressive resizing, this means that each iteration or generation of the model will use layers from the previous generation, plus any layers needed to take the resized images as input.

Using the dimensions I selected during exploration (32x32, 64x64, and 128x128), I created a three-generation model where each successive generation takes the next largest input size . Each generation will use a training set from different random selections of the full dataset.

Building and Training the Model

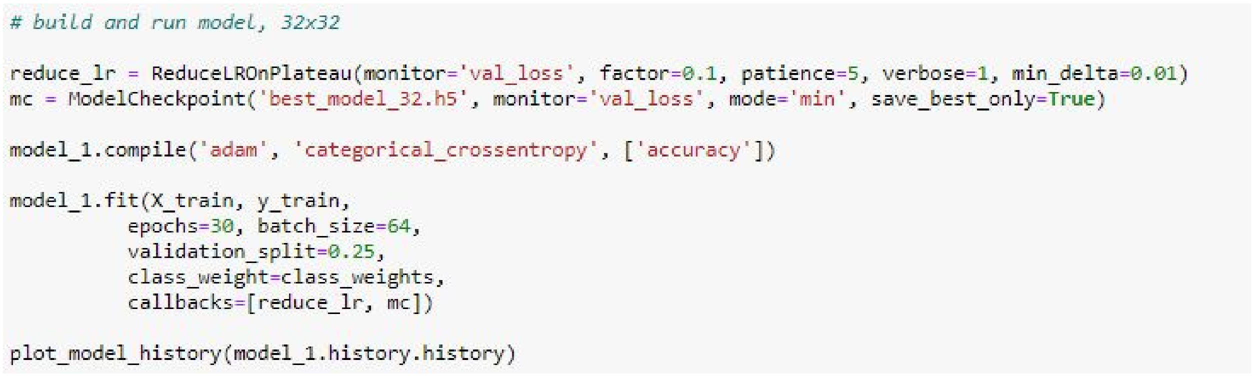
Setting on an architecture for the layers of the convolutional neural network took time and experimentation. I tested several variations in the kernel size, initial number of filters, dropout percentage, frequency of dropout layers, and the number of convolutional layers to use between each max pooling operation. The architecture I settled on for the initial model was ultimately the one printed below.



```
# CNN layers, 32x32

layers = [
    Conv2D(128, kernel_size=3, padding='same', activation='relu', input_shape=(size, size, 3)),
    MaxPooling2D(padding='same'),
    Dropout(0.2),

    Conv2D(256, kernel_size=3, padding='same', activation='relu'),
    MaxPooling2D(padding='same'),
    Dropout(0.2),

    Conv2D(512, kernel_size=3, padding='same', activation='relu'),
    MaxPooling2D(padding='same'),
    Dropout(0.2),

    Flatten(),
    Dense(2048, activation='relu'),
    Dropout(0.2),

    Dense(62),
    Activation('softmax')]

model_1 = Sequential()
for layer in layers:
    model_1.add(layer)
```

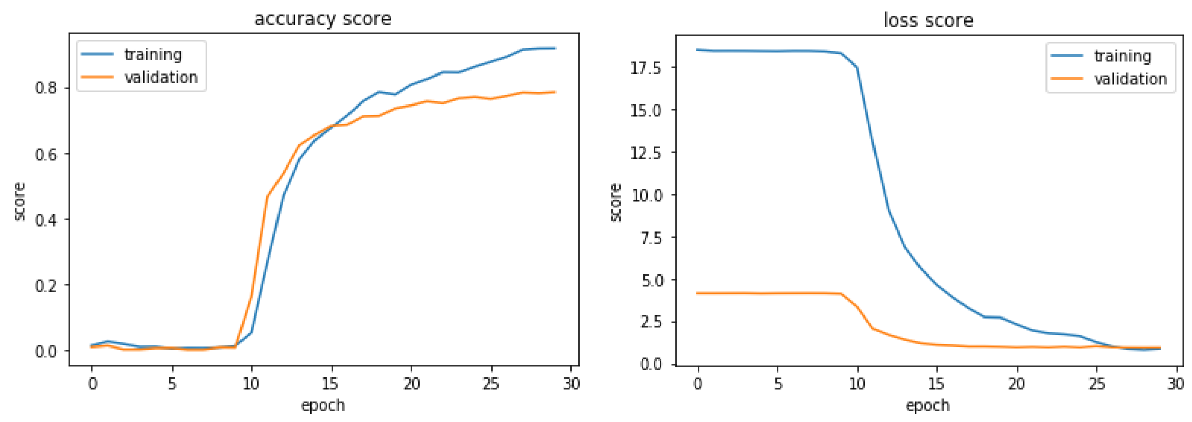
During the convolution phase, the layers are divided into a repeating pattern of three layers: one convolutional layer with the ReLU activation function, one max pooling layer, and one dropout later. After the last repetition, the input is flattened into a shape that can be fed into a dense layer. One more instance of dropout occurs before the final dense layer, which instead uses the softmax activation function to output the per class probabilities for each input.

Each generation of the model uses two callbacks: one to exponentially reduce the learning rate if no significant improvement in the validation loss occurs after a number of epochs, and a second to save the best version of the model to use as the next generation's prior. The model is compiled using the Adam optimizer, categorical crossentropy as the loss function, and accuracy as an additional metric to evaluate during training and validation. Validation is performed using 25% of the test set.



```
# build and run model, 32x32

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, verbose=1, min_delta=0.01)
mc = ModelCheckpoint('best_model_32.h5', monitor='val_loss', mode='min', save_best_only=True)

model_1.compile('adam', 'categorical_crossentropy', ['accuracy'])

model_1.fit(X_train, y_train,
            epochs=30, batch_size=64,
            validation_split=0.25,
            class_weight=class_weights,
            callbacks=[reduce_lr, mc])

plot_model_history(model_1.history.history)
```

After fitting the model, I plot the accuracy and validation loss scores for both the training and validation sets.

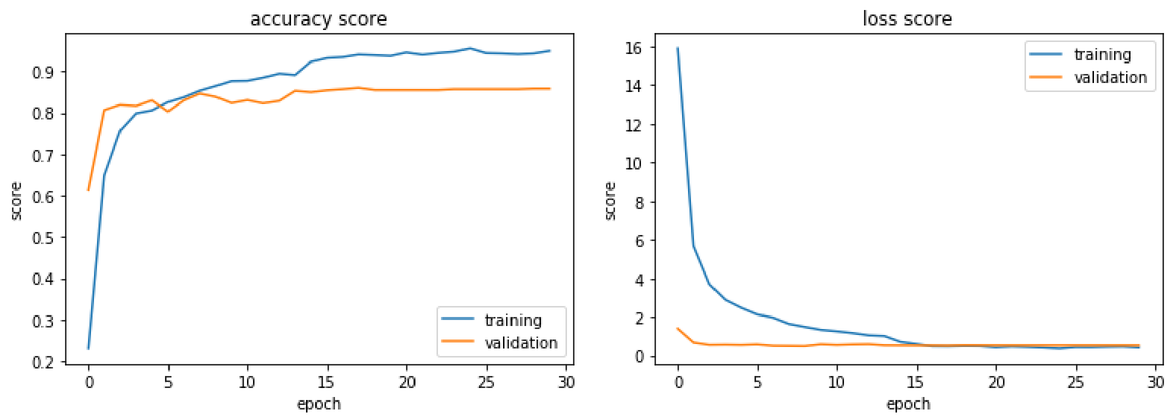


The training loss is initially much higher than the validation loss, which I thought was unusual. As best as I can tell, this is probably a result of the training model also being subject to regularization loss via the dropout layers. The gap between the two loss scores shrinks to nothing after the model begins its climb down the gradient descent. The 29th epoch displays the best validation loss, and the validation accuracy at that stage is about 78.05%. This is the version of the model that gets saved for future use.

The second generation of the model begins with a new set of input layers to accommodate data that has now doubled in size before the layers from the previous model, along with their corresponding weights, are added.

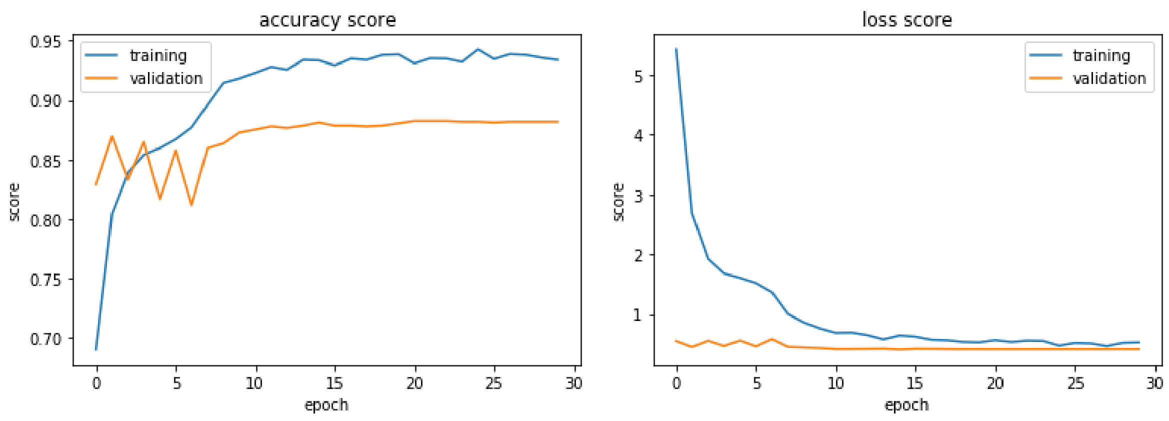
```
# CNN layers, 64x64

model_2 = Sequential()

model_2.add(Conv2D(64, kernel_size=3, padding='same', activation='relu', input_shape=(size, size, 3)))
model_2.add(MaxPooling2D(padding='same'))
model_2.add(Dropout(0.2))

model_2.add(Conv2D(128, kernel_size=3, padding='same', activation='relu'))

prior = load_model('best_model_32.h5')
for layer in prior.layers[1:]:
    model_2.add(layer)
```

The same callbacks and compiler parameters are used for the second generation as the first, and the loss and accuracy scores are plotted in the same manner as before.



The new model shows some improvement on the prior model, with the saved version scoring 0.4954 on the validation loss and 83.83% on the validation accuracy. Perhaps as a result of inheriting weights from the prior model, it takes far fewer epochs for this generation to begin to converge on a maximum accuracy and minimum loss.

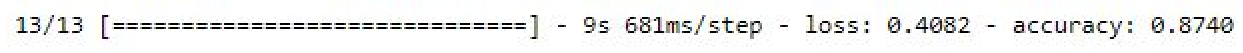
Next, I train, validate, and plot metrics for the third generation.



Of all three generations, this latest one appears to benefit the most from learning rate reductions, seeing how the metrics were slowly trending in the wrong directions for the first few epochs. By the time this model finishes training, it achieves the best metric scores so far, with a validation loss of 0.4096 and a validation accuracy of 88.12%. The final saved version of the model is the one that will be used to perform evaluation in the following section.

Evaluation

Upon evaluating the savel model on the test set, I obtain the following results:

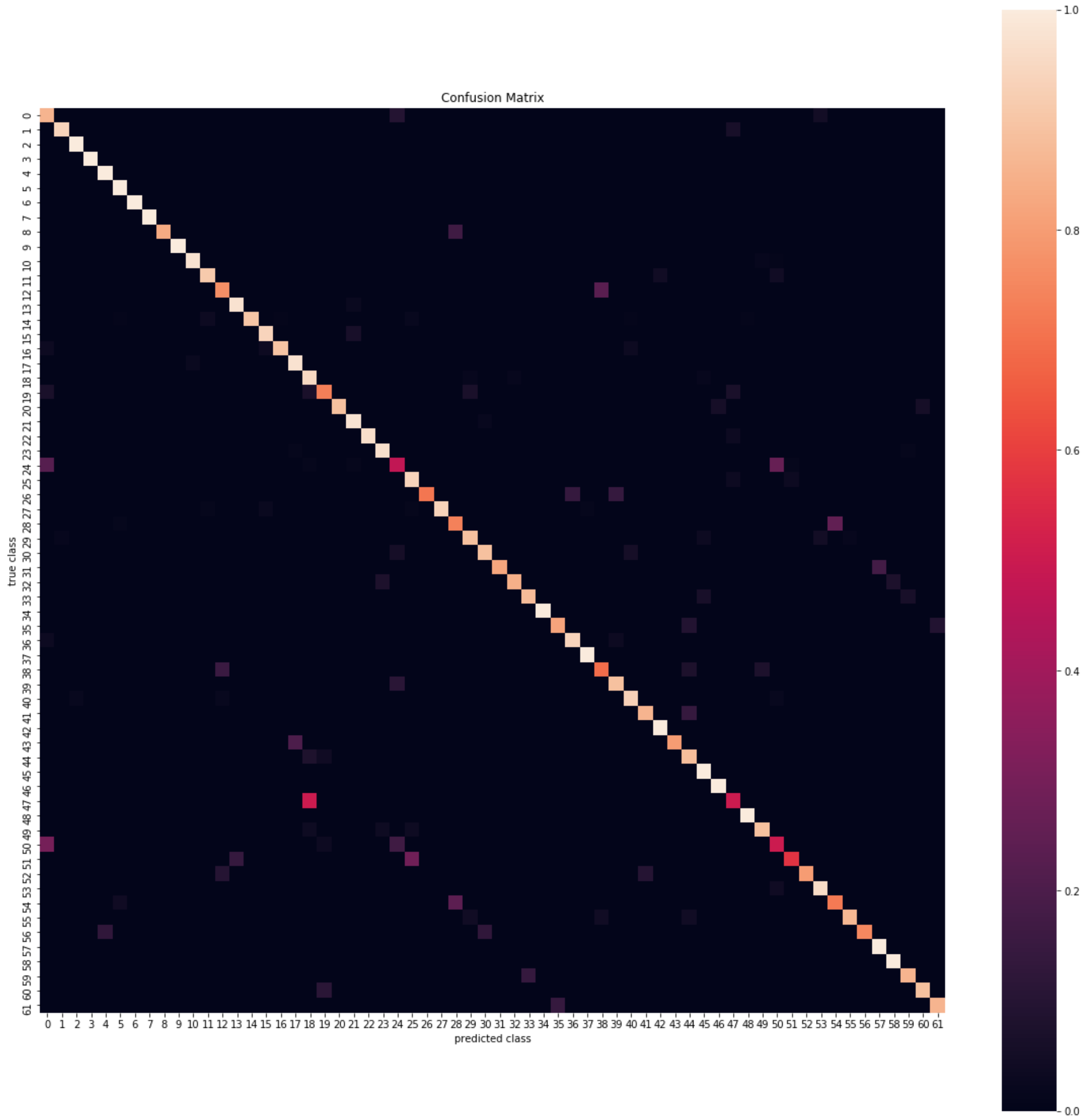


```
13/13 [==============================] - 9s 681ms/step - loss: 0.4082 - accuracy: 0.8740
```

The loss score of 0.4082 and the accuracy score of 87.40% are within a reasonable margin of error of the validation scores for the test set, and based on other work that has been done on this dataset, I believe these final results indicate strong model performance.

Although most of the work I found on this dataset uses accuracy as the primary evaluation metric, I believe it is also valuable that we examine the precision, recall, and f_1 scores of the model. To do this, I generate a confusion matrix of the data. The confusion matrix shown in this paper is a simple heatmap that effectively conveys general impressions of each class's performance; the sheer number of classes unfortunately makes it difficult to display a confusion matrix with explicit numeric data in this format.

A larger, more detailed confusion matrix can be found in the Jupyter Notebook that accompanies this report.



The most noticeable trend I see in this matrix, besides the prominent main diagonal, is the presence of two other diagonals that each represent a set of misclassified samples. Inspecting these diagonals reveals that both of them are the result of confusing uppercase and lowercase letters for each other. Given the number of letters that maintain the same shape in each of these forms, I find the model's confusion in this area to be understandable.

Using the data from this confusion matrix, I compute a precision score of 87.06%, a recall score of 87.96%, and an f_1 score of 87.51%; on average, it seems the model performs well enough on individual classes in spite of the unbalanced nature of the data. However, the performance across classes is clearly not uniform, so it would be valuable to examine and diagnose the lowest-performing classes, which may offer insight into ways to further improve the model.