Robert Kaszubski

CSC578 – Section 710

Homework 7 – Convolution Neural Networks

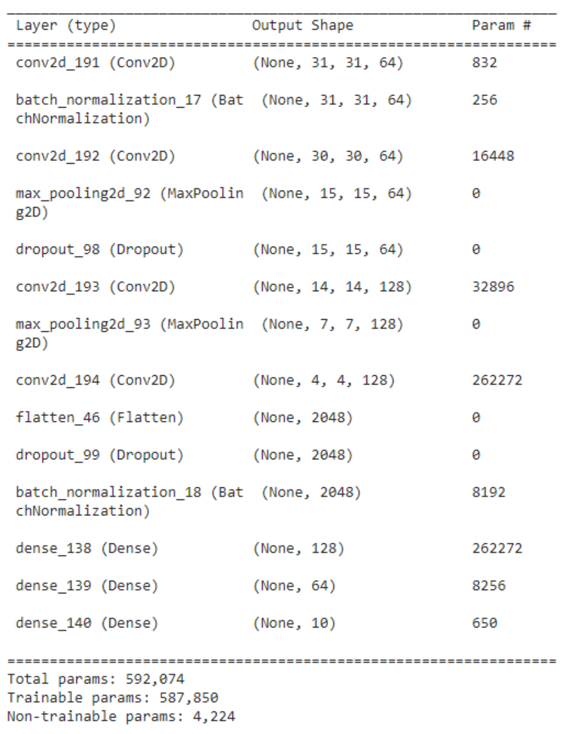
Kaggle name: **Robert**

Rank on Public Leaderboard as of November 11: **5th / .58783**

Section A – Best Model

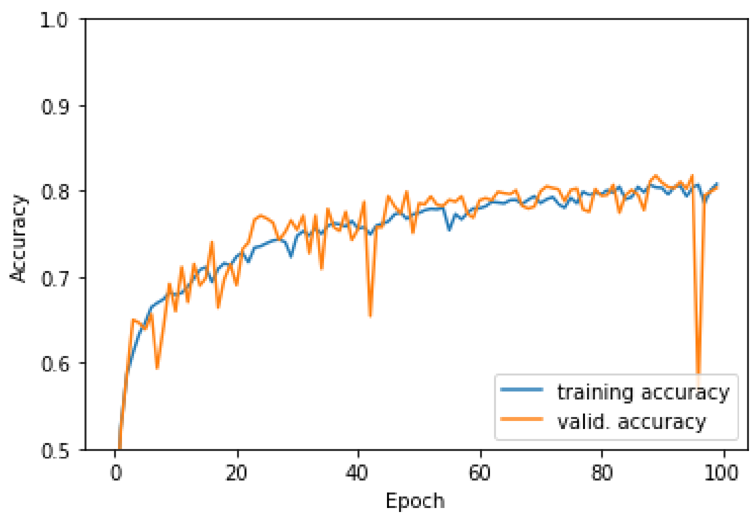
My best model achieved a score of **.58783** on Kaggle.

The model begins with a convolution layer of 64 filters of size 2x2. The output of the first convolution filter is fed into a batch normalization layer. This is then passed into an identical convolution layer then into a max pooling layer (size 2x2). The first dropout layer is applied, which drops 50% of the input units by setting them to 0. The data is then passed into another convolution layer, this time with 128 filters of size 2x2. A max pooling layer follows before another convolution layer with 128 filters of size 4x4. The data is then flattened to be passed into the fully connected layers. Before, it reaches the fully connected layers, another dropout layer is applied removing 50%, and then another batch normalization layer. This is done to prevent overfitting. The data first enters a fully connecter layer of 128 nodes, matching the 128 filters of the last convolution layer, then passes into a fully connected layer of 64 nodes, before finally entering the output layer with 10 nodes, each corresponding to one of the class labels.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_191 (Conv2D)         (None, 31, 31, 64)        832

 batch_normalization_17 (Bat  (None, 31, 31, 64)       256
 chNormalization)

 conv2d_192 (Conv2D)         (None, 30, 30, 64)        16448

 max_pooling2d_92 (MaxPoolin  (None, 15, 15, 64)       0
 g2D)

 dropout_98 (Dropout)        (None, 15, 15, 64)        0

 conv2d_193 (Conv2D)         (None, 14, 14, 128)       32896

 max_pooling2d_93 (MaxPoolin  (None, 7, 7, 128)        0
 g2D)

 conv2d_194 (Conv2D)         (None, 4, 4, 128)         262272

 flatten_46 (Flatten)        (None, 2048)              0

 dropout_99 (Dropout)        (None, 2048)              0

 batch_normalization_18 (Bat  (None, 2048)             8192
 chNormalization)

 dense_138 (Dense)           (None, 128)               262272

 dense_139 (Dense)           (None, 64)                8256

 dense_140 (Dense)           (None, 10)                650

=================================================================
Total params: 592,074
Trainable params: 587,850
Non-trainable params: 4,224
```

To get the optimal model, I used a callback with early stopping. This tracked the training process and noted the best weights from the epoch with the lowest validation loss. The model was originally trained over 10 epochs however better results were achieved training the model for 100 epochs and using callback. The lowest loss was at epoch 96 with .5349 and an accuracy of .8176.



Section B – Hyperparameter Tuning

The original code given to use netted:

Loss .9588, Accuracy .6839

Adjusting number of filters/nodes:

Filters	Loss	Accuracy	Runtime
16/32/64 | 64	.9757	.6712	12s
32/64/128 | 64	.9821	.6905	11s
35/64/128 | 128	1.1602	.6917	12s
50/100/120 | 64	.9367	.6963	16s
64/128/128 | 64	.9245	.7130	17s
64/128/128 | 32	.9769	.7061	16s
64/64/128 | 128	1.0849	.6948	14s
64/64/128 | 64	.9312	.6913	13s
64/128/256 | 64	.9998	.7004	17s

I was originally only changing the number of filters but also experimented with changing the number of nodes in the fully connected layer. It seems that for the most part, the more filters used the better the accuracy. I got the highest accuracy using 64 filters for the first convolution layer, and 128 for the 2nd and 3rd layer. I think adding more convolution layers will improve this further, as well as perhaps using different filter sizes.
I'm going to be keeping this configuration going forward in terms of the number of filters.

Changing filter sizes:

Filter Sizes	Loss	Accuracy	Runtime
3x3,3x3,5x5	.9846	.6939	17s

3x3,5x5,3x3	.9443	.6969	18s
5x5,3x3,3x3	1.0351	.6926	17s
5x5,5x5,5x5	1.2027	.6725	15s
7x7, 7x7, 5x5	1.2187	.6054	13s
2x2, 2x2, 3x3	.8577	.7190	16s
3x3, 2x2, 2x2	.9284	.7211	11s
2x2,3x3,2x2	.9009	.7046	17s
2x2,2x2,2x2	.9901	.7102	12s
2x2,2x2,4x4	.8764	.7314	12s
2x2,3x3,4x4	1.0340	.6648	18s
2x2, 2x2, 2x4	.8544	.7222	11s
2x2, 2x2, 4x2	.8829	.7208	11s
2x1, 2x2, 4x4	.9629	.6776	14s
1x1, 2x2, 4x4	1.0616	.6446	15s

Using larger filters tended to increase the runtime while decreasing the accuracy. This makes sense as its essentially condensing more of the image each time data is passed through a convolution layer. I ran into an interesting issue when trying to use all 7x7 filters, where the data was reduced too far by the time it reached the third convolution layer. I found the best results using a 2x2 filter for the first two convolution layers, then a 4x4 filter for the final convolution layer. I also wanted to experiment with filters that aren't squares. This reduced the accuracy slightly but didn't change the overall results too much.

Adding more convolution layers:

I first added an identical convolution layer right after the first original layer and before the first pooling layer. This netted a loss of .8282, and an accuracy of .7432 and a run time of about 17 seconds per epoch. I'm also starting to suspect some overfitting at this point as the training loss is very low at .2703 and accuracy getting high at around .88. I tried adding a third convolution layer in the same spot, this wasn't as effective with a loss of 1.0500 and accuracy of .7239.

I then tried adding a second convolution layer after the first pooling layer. This netted a loss of .9534 and accuracy of .7326. I tried again this time removing the extra layer added right after the first original convolution layer as described in the previous paragraph. This brought the loss up to 1.0210 and the accuracy to .7074.

I removed the added convolution layer after the first pooling layer, and instead added it after the second pooling layer. This returned a loss of .9652, and accuracy of .7369. Not quite as good as before. The runtime was also a little longer at 19 seconds per epoch.

Adding Dropout layers:

I first added a single dropout layer right after the first max pooling layer and before the third convolution layer. The dropout was set to .2 and netted a loss of .7650 and accuracy of .7480 with a run time of about 19 seconds.

I tried several different dropout values. Dropout is random so retraining the model does get different results:

Dropout	Loss	Accuracy	Runtime
.2	.7650	.7480	19s
.1	.8046	.7549	17s
.25	.7905	.7409	18s
.15	.7733	.7578	18s
.5	.7829	.7338	19s

It seems like .15 is the best value for dropout percentage. Although .2 and .1 both netted similar results.

I then added more dropout layers in other places. Adding a dropout layer between the first two convolution layers decreased accuracy to about .7519. Adding a dropout layer before the final convolution layer decreased accuracy to about .7413. This is much lower than keeping only the single dropout layer. I also tried adding dropout prior to the fully connected layer. This greatly improved results. Loss became .7048 and accuracy went up to .7656.

Adding fully connected layers:

I kept these two dropout layers (one after the first pooling layer, one before the fully connected layer) with a .15 dropout percentage. I then tried adding more fully connected layers. Adding another fully connected layer with the same 64 nodes decreased the accuracy down to .7556. I changed this layer to only 32 nodes to observe any changes. The accuracy then decreased to .7538. I went back and modified the number of nodes of both fully connected layers to 128 and 64 respectively as I expected that having more nodes would improve performance, this netted a loss of .6879 and accuracy of .7678, a new best. I also attempted to add another fully connected layer with 64 nodes for a total of 3 (excluding the output), this resulted in worse results. I kept the two fully connected layers, the runtime remained at around 19 seconds per epoch during training.

Adding Batch Normalization (non-best but interesting model):

I wanted to try batch normalization. Batch normalization is meant to standardize the inputs so at the very least should make the network a little faster. I attempted to add a batch normalization layer in a few different places with all default parameters. Adding it before the fully connected layers improved the accuracy slightly to .7686 but also increased the loss to .6924. I tried adding a few more layers before the convolution layers except for the first two for a total of 3 Batch Normalization layers, this resulted in the accuracy rising to .7817 and the loss at .6885. The loss is like before, but the accuracy is almost a full 2 points higher, however when submitting this on Kaggle it performed slightly worse than the previous model. What I noticed with these models is that they are a bit more random than before in the sense that the loss can have huge shifts between epochs in either direction. I decided to save this model (included at the end of my code file) as it was interesting but took out batch normalization for the time being.

Trying Data Augmentation:

At this point I wanted to try out data augmentation. I used keras preprocessing layers to add random image flipping and rotation. This however proved unsuccessful. Perhaps I made a mistake in my implementation but the best accuracy I was able to achieve was around .65. I did have to upgrade to Colab Pro at this point as I had used up all my GPU runtime, so going forward the runtimes aren't comparable to my previous results. However, with the added processing power I wanted to try training my models longer.

Training over more Epochs:

I tried training my current best model (the one with .15 dropout that netted an accuracy of .7678 and loss of .6879) for 50 epochs. I noticed quite quickly that soon after 10 epochs, the loss started growing rapidly. The training accuracy was approaching .95 yet the validation accuracy was hardly improving. It was apparent that there was still overfitting. I increased dropout to .5 and trained the model again to find much better results. The new accuracy was .78 and the loss down to .6433. There doesn't seem to be much overfitting either. I tried to alter the model further by bringing back batch normalization and training over 100 epochs. This netted an accuracy of .81 and loss of only .6135.

Final Touches and Callback:

I saw that during training, it had better result during certain epochs. I did some research and implemented call back early stoppage. I set this to keep track of the validation loss and to save the weights of the epoch with the best loss. This achieved great success. When training the model, especially with batch normalization, the loss oscillated quite a bit, sometimes dramatically rising 10 or more percent. So, if you used the wrong number of epochs for training, then you may get an unreasonably high loss. This became my best and final model with an accuracy .82 and loss of around .5349 at epoch 96.

Section C – Conclusions

My final model successful classifies with an accuracy of around 82% and loss of .5349. According to Kaggle, the loss was .58783. It successfully eliminates overfitting as seen by the similar accuracy between the training and testing. The model is generalized and works well with brand new data which was my goal. The model uses batch normalization and dropout to achieve this. The original iterations had quite a bit of overfitting after the first few epochs of training that led to the loss increases. The model's convolution layers initially use 64 filters and later layers use 128 filters to account for the more complex features being captured and the fact that the data's height and width gets condensed with each convolution and pooling layer.

A lot of experimentation went into finding this model. Each parameter was initially adjusted individually to find the best performance starting with the number of filters as mentioned above. Next, the filter size was adjusted. I found that the smaller the filter the more information was preserved and even ran into an interesting problem when using larger filters where the data was dimensionally too small for the filter. The only time a larger filter was used was for the final convolution layer which finds the highest-level features.

It was interesting to see batch normalization initially being detrimental to the model. My initial testing was training with only 10 epochs. It wasn’t until I started attempting to train with 50 or 100 epochs that the benefits of batch normalization and a high dropout started taking effect. This was essential to lowering the loss and raising the accuracy for this final model.

I believe this model is successful and would work well classifying any data because of how generalized it is.

Section D – Reflections

This was my favorite assignment so far. I loved the competitive aspect of it with the Kaggle competition. I had used Kaggle before to find datasets but never participated in a competition on there. It was just fun to compete, and I think I might enter some more competitions after the class ends. I tried my best, and I think I scored reasonably well. I look forward to seeing what the top scoring students did. I spend hours testing and experimenting and I still fell way short from the top students.

I didn’t really find this assignment all that difficult, just a little time consuming due to how long the training took for each iteration of the model. There was only a little bit of coding involved which was adding the export to csv which wasn’t too complicated. I did find adding data augmentation to be difficult, I tried to follow some guides, but I was getting errors and when I did get it to run, it ran poorly.

I was quite happy with my final best model, which at the moment of writing this is ranked number five. That’ll probably go down by the time the competition closes but I’m still proud of this result. My initial goal was to get my score to around .7 and ended up getting lower than .6.