Report

Implementation of Convolution Layer

Forward: Instead of running for loops across the samples, channels, height and weight, I used an Img2Col implementation to transform each image slice into a column forming a matrix. After transforming, a dot product with a reshaped weight matrix will give me a quick calculation of the output.

Backward: A naïve implementation of backward propagation for convolution using for loops across samples, channels, height and weight of the gradients. We have to update each weight individually in the loop.

Implementation of FC layer

Forward: A simple dot product of the inputs and weights plus bias.

Backward: Update gradient of bias and weights based on the formulas accordingly.

Implementation of ReLU layer

Forward: Outputs the maximum of either 0 or inputs.

Backward: We only pass back the gradient if the gradient > 0. If the gradient is < 0, we pass back 0 as well.

Implementation of Pooling layer

Forward: Instead of running for loops across the samples, channels, height and weight, I used an Img2Col implementation to transform each image slice into a column forming a matrix. If it is a max pooling, we shall pass on the maximum of the image slice (column). If it is average pool, we will pass on the average of the image slice (column).

Backward: A naïve implementation of backward propagation for pooling using for loops across samples, channels, height and weight of the gradients. For max pooling, the gradient from the next layer is passed back to only that neuron that has the max value. All other neurons get 0 gradient. For average pooling, the gradient is divided by # of neurons and distributed equally to each neuron.

Implementation of Dropout layer

Forward: Create a mask that tells you which neuron should be dropped out with probability p. Keep this ratio. Only inputs with mask (True) will be passed on to next layer. We also scale the outputs by 1/(1-p)

Backward: We only pass back the gradient to neurons that were allowed. This is determined by the same mask as forward.

MnistNet: Initial Designs

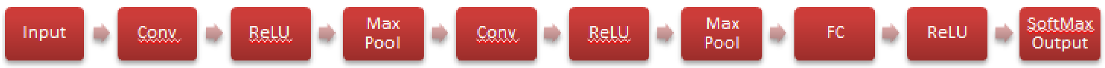
I wanted to design a similar model that could **perform better** than this model within **reasonable time (< 1 hour)** with the same resource **(my CPU, i7, 16 GB ram)**. It is the same context as those kernel-only competitions on Kaggle.

1. **Number of layers:** Since the original model was not over-fitting, I tried a similarly small but deeper CNN model that could perhaps capture more information from the training samples. By small, I mean that each convolution layer has only up to 32 filters. However I added more convolutions blocks to make it deeper (~8 convolutions blocks). Since we have a small image, I also added padding to both Conv and Maxpool layers. I was able to achieve ~97% testing accuracy within 2 epochs but it was taking > 1 hour.

2. **Dropout and # of epochs:** In my second try, I added Dropout layers to the original models after each Maxpool layers. I played around with the ratio from 15% to 50%. Since dropout has a regularising effect, I trained the model for 2 more epochs to train the model longer. Overall, it made my training progress slower and I still got 96% to 97% testing accuracy.

Final MNISTNet

Since the original architecture was already doing quite well with reasonable time. I decided to make some changes to the existing layers. The table below contains the summary of the difference:

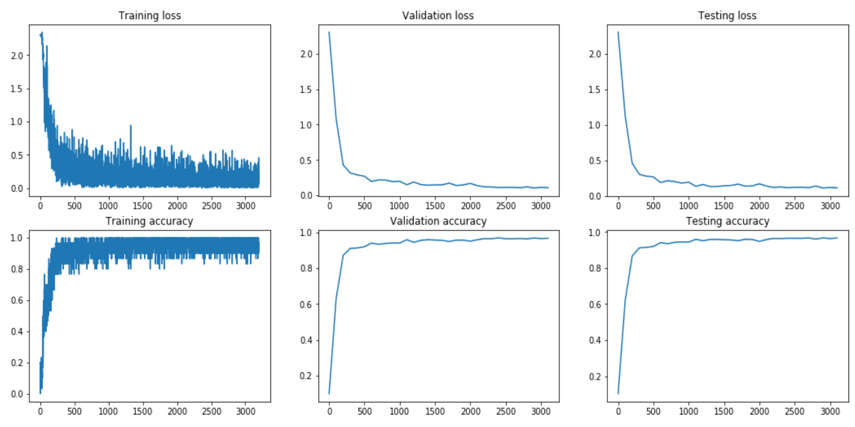


Original	Same	**6 filters**	Same	Same	**16 filters**	Same	Same	**400 inputs**	Same	Same
Final	Same	**16 filters**	Same	Same	**32 filters**	Same	Same	**800 inputs**	Same	Same

- In Conv 1, I switched from 6 filters to 16 filters.
- In Conv 2, I switched from 16 filters to 32 filters.
- In the FC layer, the number of inputs increased from 400 to 800 which contributed most to the computational intensity.

Initially I had wanted to increase the filter height and weight to 5x5. The idea was to increase the receptive field size and capture more pixels. However, I realised that the image was only 28x28 and a 5x5 filter will shrink the images by too much. So instead, since there are only 6 filters for first Conv layer, I decided to increase this number to 16. The rationale is the same. We can increase the complexity of the information captured by the first Conv layer and hopefully more information can be passed on to the later layers.

The later Conv layers usually have more filters to encode even richer representation of the data. To do that, I increased the second Conv layer from 16 to 32 filters. I stopped at 32 because the flatten layer will already create 800 inputs (400 more than original model). If I increase it to 64 or more, the model will definitely take longer than 1 hour to reach stability.



The model achieves > 90% validation and testing accuracy within 300 iterations. It can achieve the testing accuracy of ~97% with around 2000 iterations.

Figure 1: Results after 2nd epoch (New)

Next Steps

It seems promising to increase the number of filters for each layer. A combination of increasing filters and layers and adding dropout layers could allow us to train a much more accurate network without over-fitting.