Addressing issues with training deep networks
Eddie Gil 922002267

Abstract: Recent work towards improving Convolutional Neural Networks (CNNs) has focused on adding network depth and complexity in nodal connectivity. Very deep networks however, still suffer from training difficulties due to problems with unstable gradients. I hypothesize that this largely stems from the way errors are back-propagated through the networks. In order to test whether or not adding simple connectivity pathways aid in trainability and overall performance, I trained neural networks of various depths and varying number of kernels on CIFAR-10 and compared their testing accuracy and learning curves. The results obtained in this work show that concatenating down sampled outputs from earlier layers in the network with the ones meant to be passed to fully connected layers does improve trainability and overall testing accuracy. Increased network depth can also be added using this techique, however the advantages gained in doing so were not worth the increase in depth. Thus, I conclude that a well designed network of a certain depth can offer more performance advantages over arbitrarily deep networks.

Introduction and Motivation: There has been a push for the use of deeper networks with more complicated architectures in the past few years. CNNs with impressive performance on the COCO, ILSVRC, and CIFAR-100[1,2,3] benchmarks have had network depths greater than 12 layers, and in some cases have high complexity in their nodal connectivity. Networks like VGG16, VGG19, and ResNet have been introduced boasting 16, 19, and 150 layers respectively and perform exceedingly well on those benchmarks[4,5]. Complexity in layer connectivity has also been explored by numerous groups developing networks like Dense Nets, networks with LSTM based gates, and Highway way nets[6,7,8]. However, there may be some redundancy in connectivity and feature map similarity, increasing the memory costs for these networks.

It is important to note that with added network depth the difficulty of training networks increases. This is thought to be due to exploding and vanishing gradients. While batch normalization may help to some extent since it keeps firining of each neuron in a confined range, the true issue lies in use of the chain rule in the back propogation algorithm. For simplification we can constrain our analysis to a neural network using a categorical cross entropy loss function ,

$$C(\vec{y}, \vec{y'}) = -\sum y_i \log(y'_i) \text{ Eq (1).}$$

where y is a one hot represented vector describing a sample's class label, and y' is a vector describing the predicted label. The output *y'* can be written as a function of the weights and biases *(w, b)* in the last layer (*L*) that is

$$y' = softmax(w z + b) \text{Eq (2).}$$

The activation of some layer z in this work is given by

$$z_L = ReLU(w_{L-1} z_{L-1} + b_{L-1}) \text{ Eq(3).}$$

In order to train the network the goal is to minimize the cost with respect to the training weights and biases in the network, that is

$$min_{w,b} C(\vec{y}, \vec{y'}(w, b)) \text{Eq (4).}$$

To accomplish this gradient descent is performed with backpropagation. For the final layer that means the partial derivative with respect to the weights of the final layer can be found using the chain rule,

$$\frac{dC}{dW_L} = \frac{dC}{dy'} \frac{dy'}{dW_L} \text{Eq(5).}$$

In the layers prior to the final layer, the chain rule is again applied depending on the connectivity , however, it boils down to something of the form

$$\frac{dC}{dW_{l-n}} = \frac{dC}{dy'}\frac{dy'}{dW_l}\frac{dW_l}{dW_{l-1}}\frac{dW_{l-1}}{dW_{l-2}}\ldots\frac{dW_{l-(n-1)}}{dW_{l-n}} \text{Eq(6).}$$

Where *l* is the *l*th layer and *n* is the number of layers before it. That being said the partial derivatives are actually taken more rigorously with respect to the activation as well, adding even more intermediate steps. However, it is clear that the use of the chain rule leads to cumulative multiplication in order to backpropogate errors in these networks. When implementing in Keras and Tensorflow, these partial derivatives are determined using Reverse-Mode Autodifferentiation[9], which first creates a graph based on the connections between each layer in a given network, then performs the chain rule to find the derivative of the cost with respect to the weights and biases in each layer. Taking these concepts into account allows us to consider that the reason that connectivity patterns in Dense Units and shortcut paths in Residual Units is that the chain rule for earlier layers is shortened making, the cumulative product that propagates the error back to them more stable. That being said, Residual units only consider skipping 1 to a few layers, and although every layer in a dense unit takes into account all previous layers, it is constrained to that particular Dense block. This lead me to wonder if more simple pathways could be used to achieve the same effect.

Methodology: Performance on CIFAR-10 was used to evaluate networks using various depths, numbers of kernels, and connectivity patterns. Images loaded from the CIFAR-10 data were reshaped so that they had the shape N images x 32 rows x 32 columns x 3 color channels, where N is the number of images in a batch. Doing so allowed the entire training and testing set to be loaded into the memory. The images were scaled to be between 0 and 1 by dividing by 255. All networks trained in this project were trained using Keras with TensorFlow in the backend. Categorical Cross Entropy was used as the loss function, Stochastic Gradient Descent with a learning rate of 0.01, and batch size of 500 was used to train each network for 200 epochs. Training was done using a NVIDIA GTX 1060 graphics card. For all networks, 3x3 convolutional kernels were used with a stride of 1, and zero padded so that they had the same input size as output. 3 maxpooling layers were used in each network using a 2x2 pools size so that the output of any max pooling step had half the number of rows and columns as its input. Batch Normalization was used after each of the 3 max pooling steps. For each network, after the final maxpooling step, the output was flattened and fed to a fully connected layer using 2048 units, then 1024 units, then 10 units. The 2048, and 1024 unit layers, used 50% dropout. All layers were activated using ReLUs except for the final 10 unit layer. For that, softmax activation was used. Below are more details about the various networks

Baseline Variant: This network was restricted to the basic feedforward architecture. Variants using 12, 69, and 99 layers were implemented. A max pooling step was placed every 4, 23, and 33 layers for the 12,69, and 99 layer networks respectively. I trained copies of the 12 layer network using 3 kernels per layer ,16 kernels per layer, and 64 kernels per layer, to use as a baseline of performance. Due to memory constraints, I was only able to train versions using 3 kernels in the 69, and 99 layer variants. The connectivity map for this network is shown in **Figure 1.**

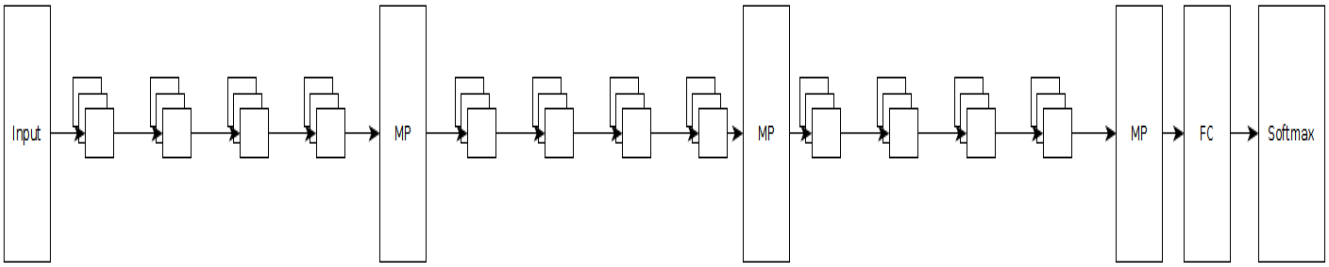

Figure 1. Basic Feed forward network. Each group of 3 squares represents a convolution, the large rectangles marked MP are max pooling layers. The rectangle marked FC represents the fully connected stages of the network and the Softmax rectangle is the softmax layer.

Layer Output Concatenation: This network contains a basic feed forward path. Additionally, the output from every kernel in each layer of the network is maxpooled so that it can be concatenated with the output from the third major maxpooling layer. Variants using, 12, 69, and 120 layers were trained. The same rules for kernels were followed for this variant as in the baseline. The same rules for maxpooling and for the number of kernels from before were followed here as well. A map for the 12 layer version can be seen in **Figure 2.**

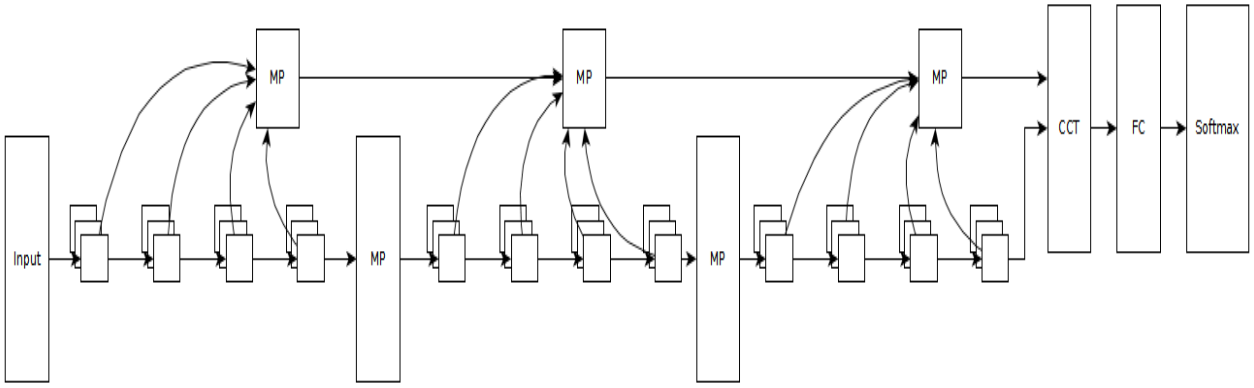

Figure 2. Concatenation variant. Each convolutional layer's output is max pooled and immediately concatenated with the channels for the output that gets fed into the fully connected stages. The abbreviation CCT stands for concatenation.

Layer Output Summation: This is the same as in the the concatenation variant however the layer outputs were summed instead of concatenated in the final step. While the concatenation pathway adds a path to the end of the network, it increases the number of outputs in the intermediate steps of the network. Summing instead of concatenation means only 1 additional map is fed to the fully connected stages. Variants using, 12, 69, and 120 layers were trained. The same rules for kernels were followed for this variant as in the baseline. The same rules for maxpooling and for the number of kernels from before were followed here as well. The network layout is shown in **Figure 3.**

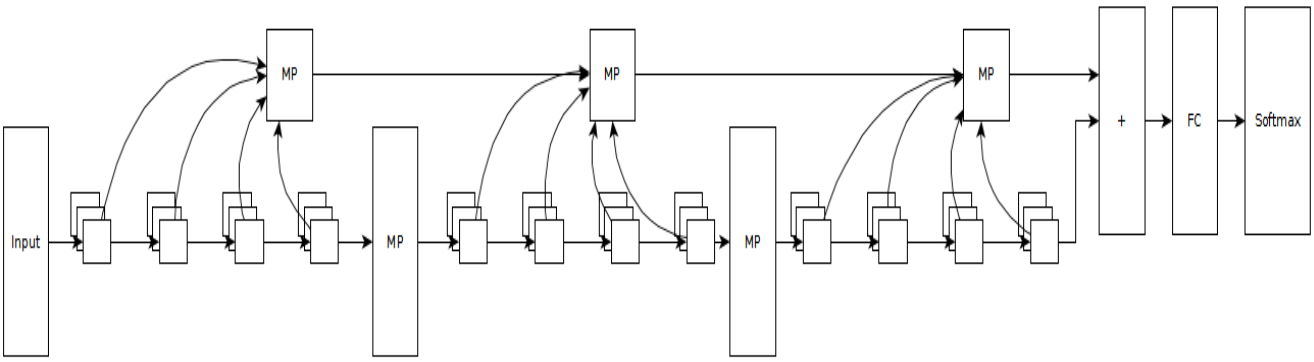

Figure 3: This is the same as the concatenation network, however the layer outputs are summed instead of being concatenated. While the concatenation network increases the final number of layers, this one reduces them while still adding a path to the end of the network.

Results: Table 1 shows the final training and testing accuracy for each model variant at different depths using a different numbers of kernels. From the Table, we can see that generally, the models using 16 kernels seem to offer the best performance, in that they have some overfitting, but better testing accuracy than networks using only 3 kernels. The best testing accuracy was obtained by the Concatenation variant using 16 kernels, and 12 layers. The 64 kernel variant on that group did have better testing accuracy but overfit significantly. One thing to point out is that if we compare across groups using the same number of kernels, the concatenation networks outperform the summing networks which outperform the baseline. However, when we compare the best network in each category there is not much difference between the techniques.

Architecture	Number of Layers	Number of Kernels	End Training Accuracy	End Testing Accuracy
Baseline	12	3	0.498	0.485
		16	**0.801**	**0.676**
		64	0.999	0.64
	69	3	0.101	0.099
Concatenation net	12	3	0.595	0.588
		16	**0.834**	**0.686**
		64	0.999	0.74
	69	**3**	**0.647**	**0.625**
	69	**16**	**0.932**	**0.74**
	120	**3**	**0.782**	**0.666**
Summing net	12	3	0.519	0.511
		16	**0.751**	**0.668**
		64	**1**	**0.714**
	69	3	0.499	0.499

Table 1: End Training and Testing Accuracy for network variants

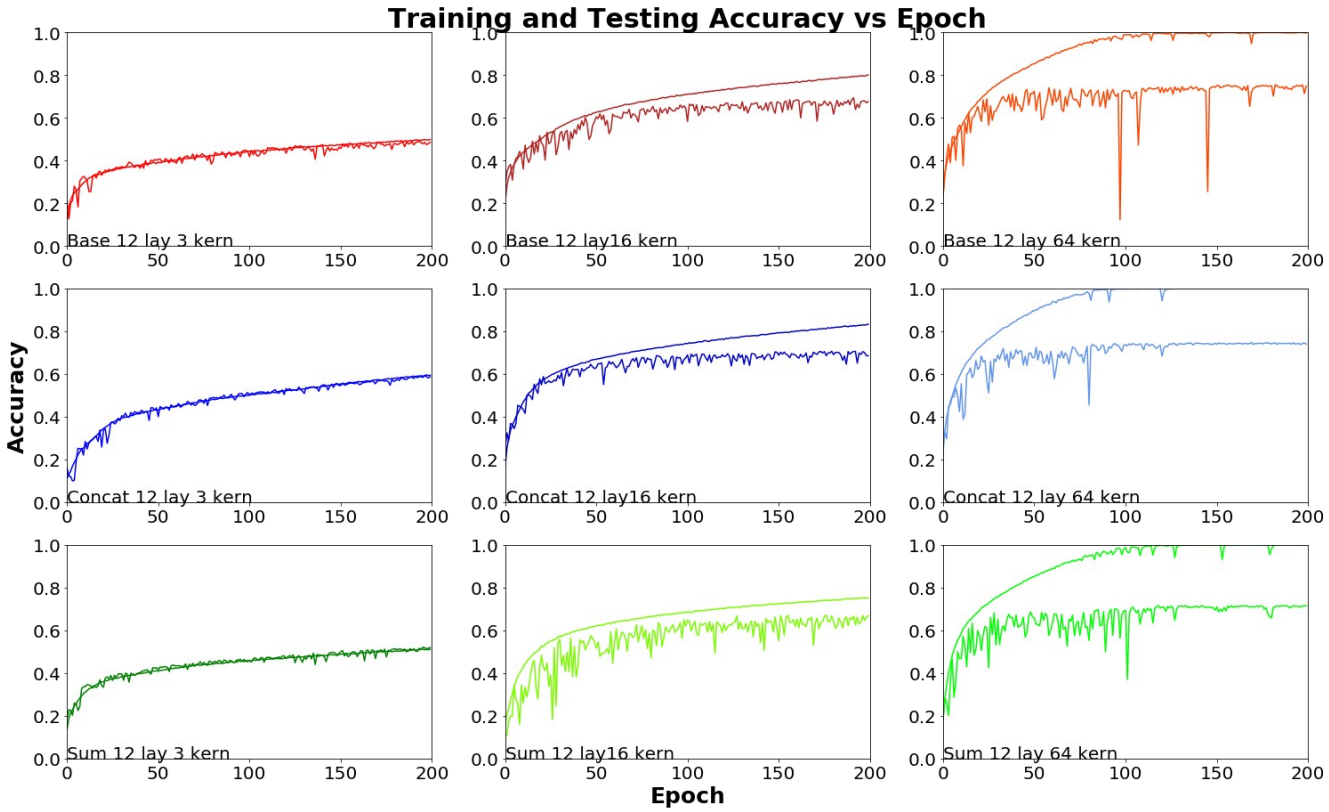

Figure 4. Training and Testing Accuracy for each of the model variants. From top to bottom each row is the Baseline model, followed by the Concatenation model, followed by the summing model. Each column from left to right is the 3 kernel variants, 16 kernels, then 64 kernel variants.

We can get a better feel for this concept from the Training and Testing accuracy plots shown in **Figure 4.** Notice that the testing accuracy when adding kernels to the networks doesn't improve all that much. Across each of the connectivity variants, the testing accuracy also seem to converge to the same point. This suggests that if I had trained multiple initializations for each architecture and gotten a mean and standard deviation of the best models from each group there may not have been significant differences in performance outcomes. One key observation is that as the number of training kernels increases, in every case overfitting increases. Additionally, by observing the points where training and testing loss diverge we can see that increases in the number of kernels shorten the number of epochs necessary to obtain a decent model. However, for instances beyond the scope of this project where we need to train for a long time, restricting the number of kernels to a small number may be more desirable.

Examination of the training losses in **Figure 5,** helps to confirm that the networks with more kernels did in fact minimize the training loss more quickly than others. There also doesn't seem to be much of a difference between the three techniques as the number of kernels increases, however, when the number of kernels picked is in a kind of middle ground (16 kernels in this case) the Concatenation technique works best to minimize the training loss without sending the losses to zero. Meaning that additional training time could yield additional improvement in that case.

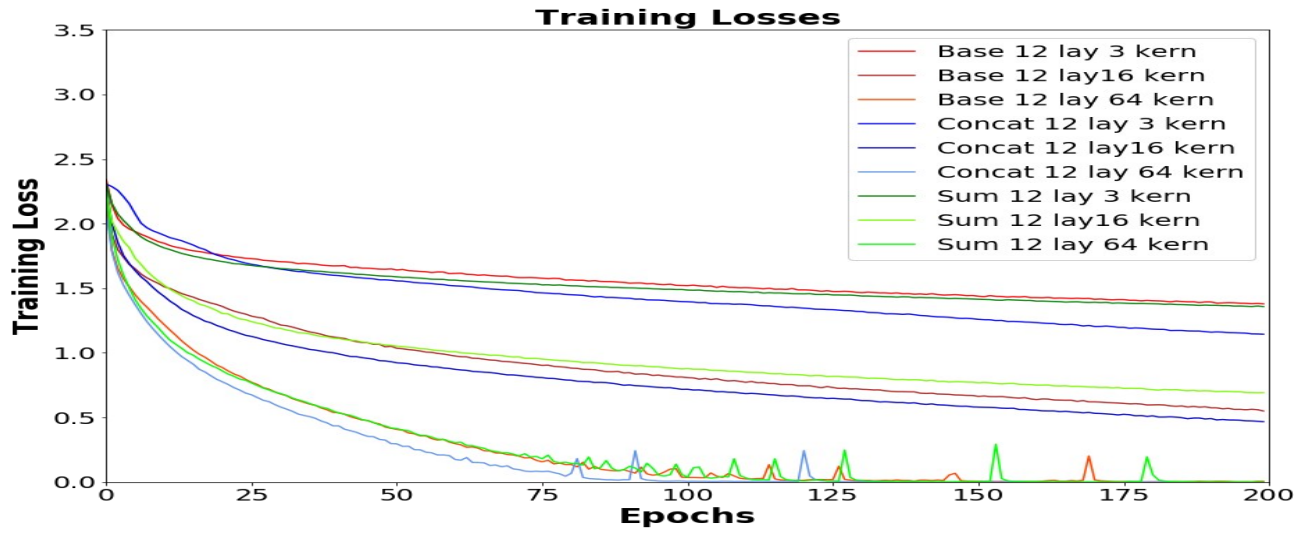

Figure 5.Training loss for the different network varaints.

In **Figure 6**, we can see the effect of increasing layer depth on the testing accuracy for the concatenation network. In **Table 1,** we saw that increasing the network depth in a basic feedforward network offers some improvements in final testing accuracy, however after some depth, the network becomes untrainable. For the Concatenation and summation variants we see that is not the case. However, **Figures 6 and 7** shows that while the deeper networks become trainable, the added depth does not linearly improve the testing accuracy of the networks.

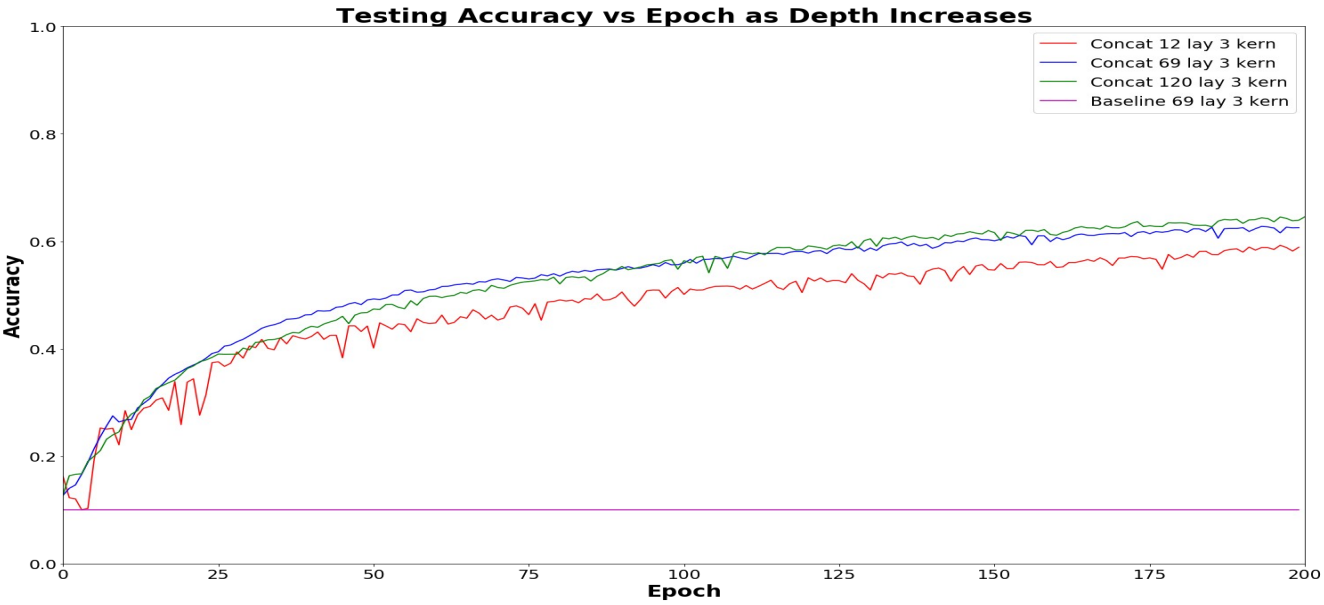

Figure 6. Testing accuracy as concatenation network depth increases for the Concatenation variant. Baseline with 69 layers is shown as well

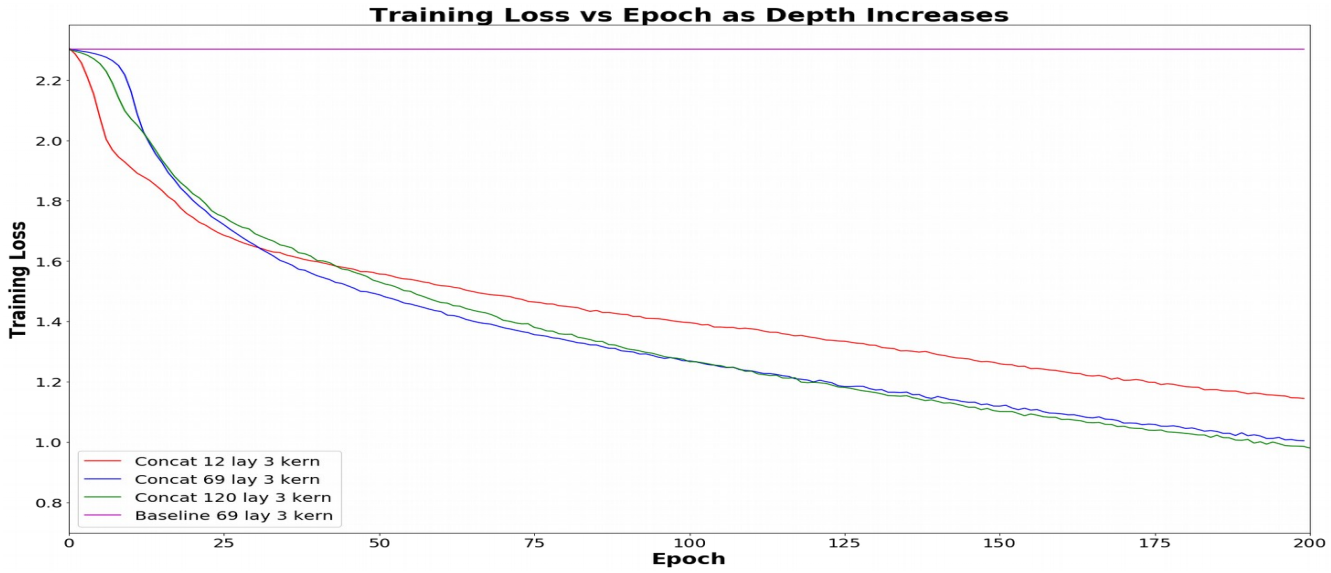

Figure 7: Training Loss versus epoch as network depth increases for 3 kernel per layer networks. Baseline with 69 layers is shown as well.

Conclusions: The biggest impact from variation in the models tested came from the number of kernels selected per layer. All of the results suggest that increasing the number of kernels increases the testing accuracy of the networks developed at the cost of increasing the amount of overfitting. This suggest that it may be possible to tune networks for their optimal number of kernels in an algorithmic way. This trend seems to hold true when the number of layers increases. That is, for the concatenation variants, as the number of layers increased from 12 to 69 to 120 layers, the split between training and test accuracy was minimal for the 3 kernel variants. When increasing from 3 to 6 kernels, the testing accuracy improved at the cost of overfitting. Additionally, in this highly constrained scenario we can see that added depth by itself, does not guarantee performance improvements over more shallow networks. Using an optimal number of kernels does allow for performance boosting, and in the event that more network depth is needed, simple fixes like adding a pathway between early layers and the later stages can allow end to end training.

To improve the quality of this work it could be useful to train multiple initializations of each network and get a mean and standard deviation for training and testing accuracy endpoints. Although training went on for long enough to show the networks converged to some end point it would allow us to verify these results with more certainty. It would also be good to test how the concatenation and summation networks compare against variants with the same number of layers but more max pooling stages. Assuming we had inputs with size of 128x128x3 would addition of 2 more max pooling stages help under the same constraints on number of layer and number of kernels.

More importantly, these results suggest that there may exist an algorithmic way for designing a network by tuning the number of kernels, network depth, and connectivity between nodes in the network. ResNets and Dense Nets started the discussion by allowing for the flow of information between blocks in the network, this work continues it by showing that the flow could take much larger steps. I think the end point is that there may be a more graphical way to explore the flow of information in networks. For future research it would be interesting to see how the use of generative approaches like Evolutionary Algorithms could tune the above parameters, while testing different connectivity mappings to design a much better network than what has been formulated here.

References:

[1] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollr, P., and Zitnick, C. L., “Microsoft coco: Common objects in context,” Computer Vision ECCV 2014 Lecture Notes in Computer Science, 740755 (2014).

[2] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L., “ImageNet Large Scale Visual Recognition Challenge,” International Journal of Computer Vision (IJCV) 115(3), 211–252 (2015).

[3] Krizhevsky, A., “Learning multiple layers of features from tiny images,” tech. Rep. (2009).

[4] Simonyan, K. and Zisserman, A., “Very deep convolutional networks for large-scale image recognition,” CoRR abs/1409.1556 (2014).

[5] He, K., Zhang, X., Ren, S., and Sun, J., “Deep residual learning for image recognition,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016).

[6] Huang, G., Liu, Z., Maaten, L. V. D., and Weinberger, K. Q., “Densely connected convolutional networks,” 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017).

[7] Srivastava, Rupesh Kumar, et al. *Training Very Deep Networks*. 2015. *EBSCOhost*, ezproxy.library.tamu.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edsarx&AN=edsarx.1507.06228&site=eds-live.

[8] Huang ,G., Chen, D., Li, T., Wu, F., van der Maaten, L., and Weinberger,. K. “Multi-scale Dense Networks for Resource Efficient Image Classification” 2018 ICLR conference paper (2018).

[9] Géron, Aurélien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow : Concepts, Tools, and Techniques to Build Intelligent Systems. First Edition. Aurélien Géron*. Sebastopol, CA : O’Reilly Media, 2017., 2017. *EBSCOhost*, ezproxy.library.tamu.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat03318a&AN=tamug.5743869&site=eds-live.