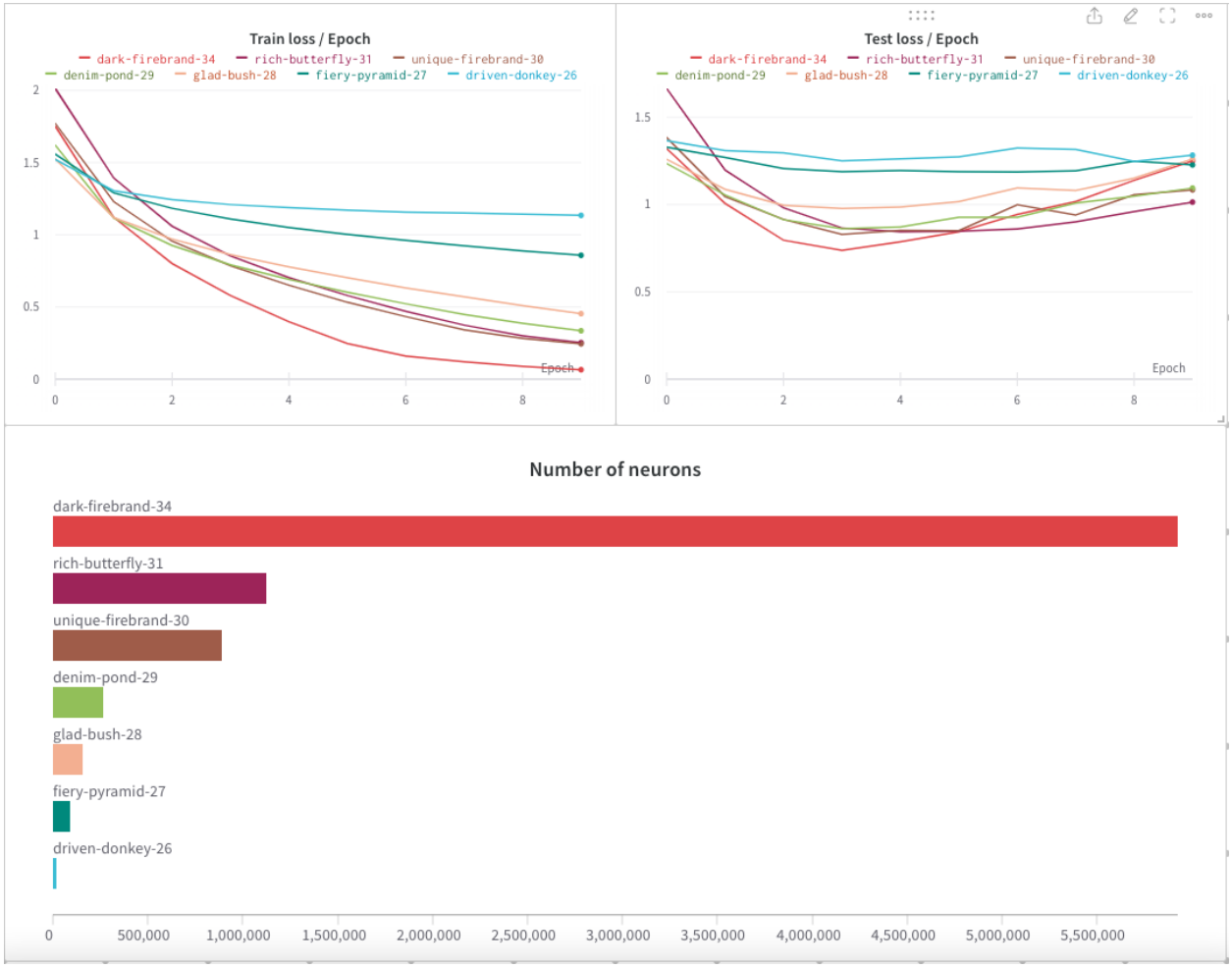
671033 - NN4I 2023
First Assignment

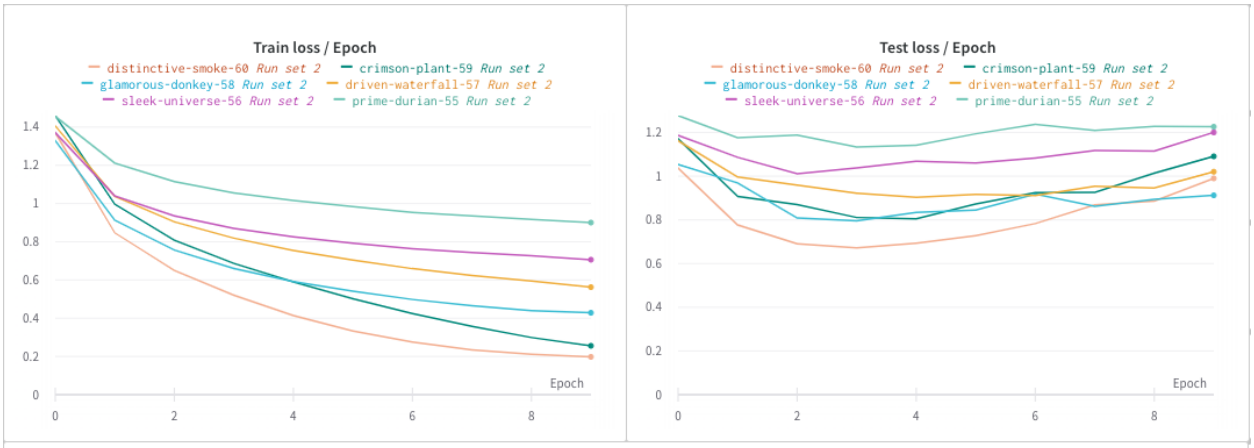
Architecture. Implement a “standard” CNN consisting of a few convolutional layers that reduce the spatial image dimensions (via pooling), while increasing the number of feature maps (number of channels). Complete this construction by reshaping the responses into a vector and applying an output FC layer that produces the output.

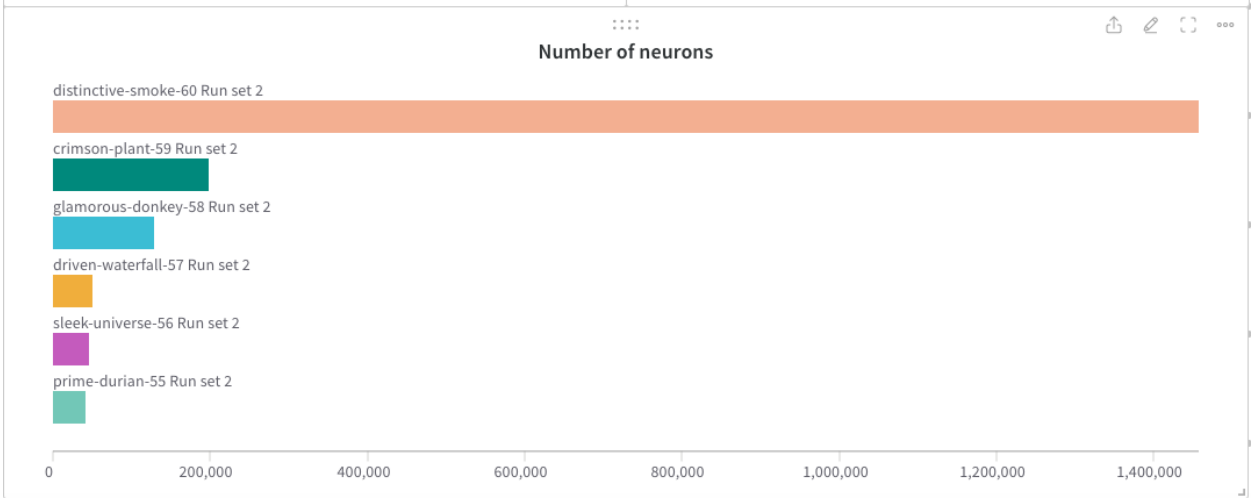
a. Plot the train and test losses and explore how they change as you change the total number of filters (neurons) in the network to the extent of overfitting and underfitting (also report the total number of learnable parameters).

Answer:
I made this report using W&B, in each run, I set the config parameters to sync the code on Google colab with the report.
I found it very convenient to use the W&B framework to track the different runs and experiments I performed.
Both the train and test loss normalized over the whole train/test dataset, I used 50,000 for the train and 10,000 for the test.



After class in week 3, I realized there is a way to make the network deeper if I use a pooling layer after more than one CL. This also makes the receptive field bigger and the learning function more expressive as we learned from alexNet and other networks.
So I did some changes to my code defining a Block as a unit that I can make deeper CL and perform the pooling in the end. I also added skip-connection if the block gets very deep to avoid gradient vanishing. Another thing was that I used in that session only one FC layer and in general, the number of learning parameters is much lower and the best network is much deeper (9 layers) So here is the second report:





Name (6 visualized)	#neurons	Notes	device	Test loss / E	Train loss /	Runtime
distinctive-smoke-60	1457066	Net depth = 3, Block depth = 3	cuda:0	0.99	0.1978	22m 43s
crimson-plant-59	198410	Net depth = 3, Block depth = 1	cuda:0	1.091	0.2556	10m 35s
glamorous-donkey-58	129194	Net depth = 2, Block depth = 3	cuda:0	0.9124	0.4283	13m 23s
driven-waterfall-57	50698	Net depth = 2, Block depth = 1	cuda:0	1.021	0.5625	9m 25s
sleek-universe-56	46058	Net depth = 1, Block depth = 3	cuda:0	1.201	0.7059	10m 32s
prime-durian-55	41418	Net depth = 1, Block depth = 1	cuda:0	1.226	0.9002	8m 13s

b. Find a configuration resulting in the lowest test error and use it for the following sections.

Answer:
The best configuration I found was the one with the most neurons in the second report as you can see in light brown color run number 60. The best test loss I got with this model was an average of 0.67 using cross-entropy loss and accuracy of 78% after 4-5 epochs, more epochs made the model overfit the training dataset and the numbers got worst.

c. Report these tests and their results in your report. The search space is huge and hence you can choose a strategy as to how you will increase and decrease the network's size.

Answer:
First report,
Throughout all the different configurations, my strategy was to use a different number of convolution layers in each configuration. I expected that the deeper my network will be the better, also because the input dimensions are 32x32 I couldn't make the network too deep, after each convolution layer there is a pooling layer that reduces the dimensions by a factor of two so the deepest network I achieve was with 4 convolution layers.
The network architecture was 1-4 CL with MaxPooling and Relu (I tried also average pooling but it was not performing as well as the max pooling layer), flattened the Tensor to a 1d vector, and then applied 2-3 FC layers which in the end output the classification vector with 10 dimensions for the 10 classes.
Most of the models (models that were expressive enough with a large number of neurons) after 5+- epochs started to overfit the training dataset. Although they overfitted I still got small improvements in accuracy and test loss with the more neurons models I trained.
Second report,
After realizing I need to use only one FC layer and because I was curious about how can I make the network deeper and better I changed all my network architecture and built a strategy for scaling my network fast, I made a Block containing few CL and Max Pool in the end. My network was built out of these blocks. I run 6 different runs to find the best model for all the combinations for Network depth [1, 2, 3] and for Block depth - [1, 3] as you can see in my code. I'm very happy with my changes and the final result. It was super fun to play and train a real neural network.

Importance of Non-Linearity. What happens to the performance of the network once you remove all the non-linear components it contains. List these components and explain the results obtained. Increase the network's size and see if the performance improves. Did this work? Explain.

Answer:
The nonlinear components in the CNN I created were the activation functions and Max pooling layers, as we saw in class multiple numbers of CL are equivalent to one linear operation because convolution is linear. So after removing all the non-linearities we got a simpler less expressive model. As we imagined the model performed not well without the nonlinearities, causing a 15% drop in accuracy. Even if we increase the network size we don't get the best results because the phenomena - multiple number of matrix multiplication is equal to one matrix. In the end, we want to learn a model that is nonLinear (because it's more expressive and better at classifying real-world data)

Cascaded Receptive Field. In order to assess the importance of using a multi-scale CNN to obtain a large receptive field, let us achieve this using a shallower network. Thus, rather than shrinking the image size using repeated convolutions interleaved with pooling operators, let us directly map the output of the first convolutional layer to the output layer. This can be implemented in two ways: (i) use an FC layer on the activations of the first conv layer, or (ii) use a "global average pooling" operator where you basically compute the average of the activations in each channel (i.e., the image x and y dimensions collapse) and then apply an FC layer. Consider both options, make a choice, and adapt the architecture to obtain a reasonably-sized model. Report your choices and their justification. Did this new network perform any better than the one in Part 1? Can you explain why?

Answer:
In the case of using only one CL that is connected to the FCL and outputs a classification without any pooling, we will get a large amount of learning parameters in general, our input images are relatively small so it's not so bad. But it performed very poorly in comparison to the model from the first question mostly because it had a low amount of neurons (60,000+-) and also a low receptive field.
In the second part using global average pooling we could average the whole channel at once which caused the learning parameters to drop quickly in our case it resulted in a very low amount of learning parameters (around 600) although the receptive field is slightly improved this network is very poorly expressive enough and so the performance. While using a deep CL with pooling layers we get a higher receptive field the deeper we get in the network which leads us to more complex feature matching and this is why it's performing much better.

Theoretical Questions:

1. Formally show that the condition L[x(i+k)](j) = L[x(i)](j+k) over a linear operator L receiving a 1D signal x(i) with offset k (the outer parentheses refer to the output signal), corresponds to a convolution. L operates over signals, i.e., it receives an input signal and outputs a signal, and the output signal index is denoted by j. Hint: decompose the signal x(i) into a weighted sum of translated delta functions, and then use the linearity of L. What input signal will output the underlining filter of the convolution?

$L[x(i)](j)$: operator L recived 1D signal $x(i)$

(j) refer to the output signal

We'll use the hint given to decompose the signal into a weighted sum of Kronecker delta functions

Let denote $\delta(i-j)$ as 1 if $i=j$, 0 otherwise

$$L[x(i)] = L\left[\sum_{k=0}^{\infty} x(k)\,\delta(i-k)\right]$$

$$= \sum_{k=0}^{\infty} L[x(k)\,\delta(i-k)] \qquad \text{Linearity of } L$$

$$= \sum_{k=0}^{\infty} x(k)\,L[\delta(i-k)] \qquad x(k) \text{ is constant in } i$$

$$\overset{*}{=} \sum_{k=0}^{\infty} X(k)\,h(i-k)$$

$(*)$ $L(\delta(i-k)) = h(i-k)$ by detenition which is the impulse response of the linear operator L

Now we can write the output of convolution of $x(i)$ with $h(i)$

$$x(i) * h(i) = \sum_{j=0}^{\infty} \sum_{k=0}^{\infty} x(k)\,h(i-k)[j-i]$$

$$L[x(i)] = x(i) * h(i) \quad \text{iff} \quad h(j-k)[j] = L(\delta(i-k))$$

Therefore the impulse response of L corresponds to the underlying filter of convultion

2. When reshaping a 2D activation map (resulting from a convolutional layer) and feeding it into an FC layer, how important is the order (is there a good ordering or a bad ordering)? Why?

In general, the 2D activation map is flattened into a 1D vector before being passed to the FC layer. The order of the flattened elements can have an impact on the learning dynamics of the network. For example, if adjacent elements in the 2D activation map are also adjacent in the flattened vector, then their weights will be updated together during backpropagation. This can be beneficial because it allows the network to learn spatially-coherent patterns in the input.

Another important consideration when flattening a 2D activation map and feeding it into a fully connected layer is the size of the flattened vector. If the activation map is large, flattening it into a very long vector can result in a very large number of parameters in the fully connected layer. This can lead to overfitting and slower training times, as well as increased memory usage during inference.

3. The convolution operator is LTI (linear translation invariant). What about the following operators:
    a. The ReLU activation function
    b. The strided pooling layer (i.e., the one that always picks the top-right pixel in each block).
    c. The addition of a bias
    d. Multiplication with a fully-connected matrix

   Explain your answers.

The ReLU activation function and the strided pooling layer are both nonlinear and translation invariant. The ReLU function applies the same nonlinear transformation to each element of the input, regardless of its location, and does not depend on the specific input values or location. Similarly, the strided pooling layer operates on non-overlapping windows of the input and picks a fixed pixel in each window, regardless of its location. Thus, the output of the ReLU and strided pooling layers can be shifted in the same way as the input without affecting their outputs.

The addition of a bias is linear but not translation invariant. The bias adds a fixed constant value to each element of the input, and therefore it does not depend on the location of the input. However, shifting the input will change the output of the bias layer because the same constant value will be added to different input values.

Multiplication with a fully-connected matrix is also linear but not translation invariant. The fully-connected layer applies a linear transformation to the input, but the weights of the matrix are learned parameters that are not necessarily invariant to shifts in the input. Therefore, shifting the input will result in a different output for the fully-connected layer.