Assignment 4

December 8, 2017

Carolyne Pelletier 101054962
Akhil Dalal 100855466

1 Modifying CNN for CIFAR-10 Dataset

1.1 Dataset

The CIFAR-10 dataset contians 60000 color images of dimensions 32x32x3 [1]. It is split into 50000 training images and 10000 test images.

The images are split into 10 categories (making this a 10 class problem).

We use the built-in convinience function in Tensorflow (v1.4): tf.keras.datasets.cifar10.load_data() to make it easier to load the training and testing sets.

We also normalize our data (divide it by 255.0 to get everything between 0 and 1).

1.2 Hyperparameters

1.2.1 Kernel sizes for convolution layers

Most commonly used kernel sizes are 3x3 and 5x5. A smaller kernel tends to pick up more details than a bigger one. During our initial tests to see which size worked best for the cifar-10 dataset, we observed that a 3x3 kernel works best here.

1.2.2 Kernel sizes for max pooling

Most commonly used kernel sizes are 2x2 and 3x3. Pooling is used to reduce the dimension of the data as we study it further in the network. This also helps in controlling overfitting.

Reducing dimensions also implies losing data. So the size of the kernel matters. Depending on the the size, we call the resulting layer overlapping or non-overlapping pooling.

Overlapping pooling has been shown to do better in practice. Overlapping regions allow for less loss of surrounding spatial information while reducing similar (if not same) amount of parameters. However, larger pooling sizes can generally be more destructive.

Here we use both 2x2 and 3x3 kernels.

1.2.3 Strides

This determines how the kernel will convolve over the input. Larger strides result in smaller output volumes spatially. We use the most common settings. A stride of 1 for convolutional kernels, and a stride of 2 for pooling kernels.

1.2.4 Weight initialization

Since we use ReLU as our activation function, we can use the Xavier Glorot initialization for weights. This is a way to calculate the standard deviation for a normal distribution from which we choose our weights uniformly at random.

The formula is the following:

$$\sqrt{2}\sqrt{\frac{2}{n_{inputs}+n_{outputs}}}$$

1.2.5 Number of feature/activation maps

The number of activation maps correspond to the number of kernels we use. The idea is that each kernel/filter learns something about the input.

Here it is common to use multiples of 32. We use 32 for the first pooling, 64 for the second, and so on.

1.2.6 Size of fully connected layer(s)

The size of the fully connected layers affects the output of the network. If it is too small, it will be difficult to learn/capture information coming in from the network. If it is too big, it might take longer to learn something.

Here we use two fully connected layers on all our networks. We use 512 neurons in each fully connected layer.

1.2.7 Number of convolution layers

A convolution layer learns features from the input given to it.

We experiment with different designs:

C - Conv, P - Max pool, I - input, O - output, FC - fully connected

#	Design
1	I -> C -> P -> FC -> FC -> O
2	I -> C -> P -> C -> P -> FC -> FC -> O
3	I -> C -> P -> C -> P -> C -> FC -> FC -> O
4	I -> C -> C -> P -> C -> C -> P -> C -> C -> P -> FC -> FC -> O

Notice that the fourth design has 2 convolutions before pooling. This allows the convolution layer to extract more complex features before going for the destructive pooling operation. It might be a better choice to do this for deep networks.

1.3 Performance

Each network was trained with 50000 training images. The networks were updated stochastically with a batch size of 128.

We test the accuracy using 1000 randomly chosen images.

1.3.1 Network 1

Input -> Conv1 -> Pool1 -> FC -> FC2 -> Output

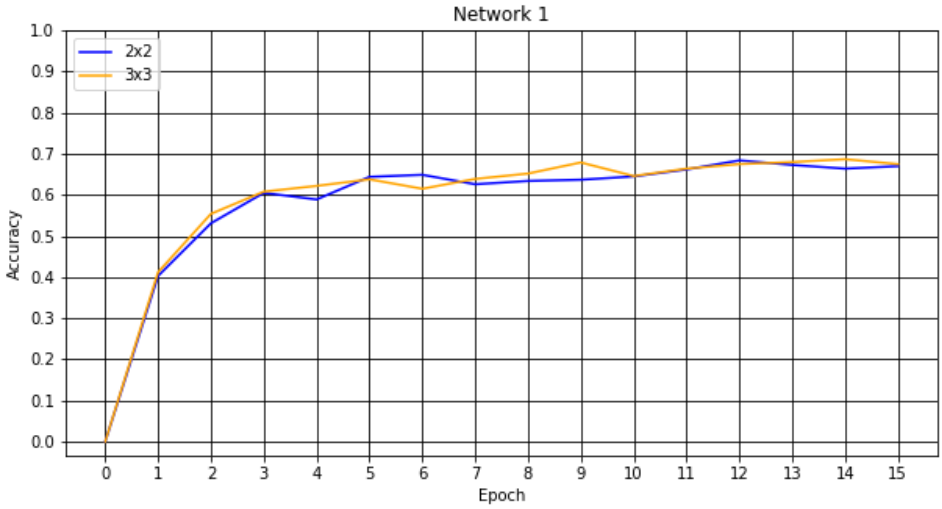
Conv1: Used 32 3x3 filters with stride 1.
Pool1: Used 2x2/3x3 filter with stride 2.
FC: fully connected layer with 512 outputs.
FC2: fully connected layer with 512 outputs.
Output: Output layer with 10 outputs.

Epoch	Accuracy (2x2)	Accuracy (3x3)
1	0.403	0.412
2	0.531	0.553
3	0.605	0.608
4	0.589	0.622
5	0.644	0.638
6	0.649	0.615
7	0.626	0.639
8	0.634	0.652
9	0.637	0.679
10	0.645	0.646
11	0.662	0.664
12	0.684	0.675
13	0.673	0.680
14	0.664	0.687
15	0.670	0.675

Plotted below are the accuracies:

```
In [34]: from IPython.display import Image
         Image(filename="img/cnn-1_acc_graph.png")
```

Out[34]:



1.3.2 Network 2

Input -> Conv1 -> Pool1 -> Conv2 -> Pool2 -> FC -> FC2 -> Output

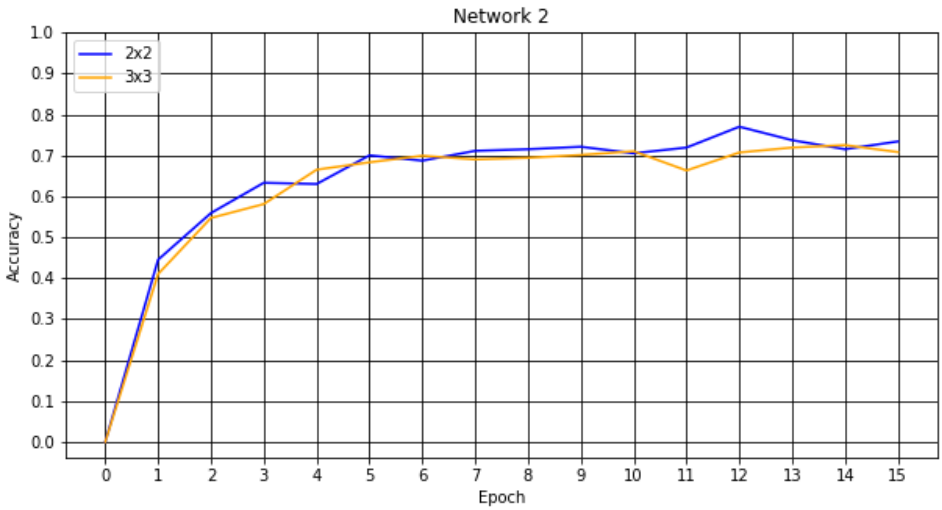
Conv1: Used 32 3x3 filters with stride 1.
Pool1: Used 2x2/3x3 filter with stride 2.
Conv2: Used 64 3x3 filters with stride 1.
Pool2: Used 2x2/3x3 filter with stride 2.
FC: fully connected layer with 512 outputs.
FC2: fully connected layer with 512 outputs.
Output: Output layer with 10 outputs.

Epoch	Accuracy (2x2)	Accuracy (3x3)
1	0.445	0.436
2	0.559	0.511
3	0.633	0.574
4	0.630	0.576
5	0.700	0.555
6	0.687	0.574
7	0.711	0.589
8	0.715	0.567
9	0.721	0.568
10	0.705	0.587
11	0.719	0.614
12	0.770	0.571
13	0.737	0.545
14	0.715	0.572
15	0.734	0.629

Plotted below are the accuracies:

```
In [35]: from IPython.display import Image
         Image(filename="img/cnn-2_acc_graph.png")
```

Out[35]:



1.3.3 Network 3

Input -> Conv1 -> Pool1 -> Conv2 -> Pool2 -> Conv3 -> FC -> FC2 -> Output

Conv1: Used 32 3x3 filters with stride 1.
Pool1: Used 2x2/3x3 filter with stride 2.
Conv2: Used 64 3x3 filters with stride 1.
Pool2: Used 2x2/3x3 filter with stride 2.
Conv3: Used 128 3x3 filters with stride 1.
FC: fully connected layer with 512 outputs.
FC2: fully connected layer with 512 outputs.
Output: Output layer with 10 outputs.

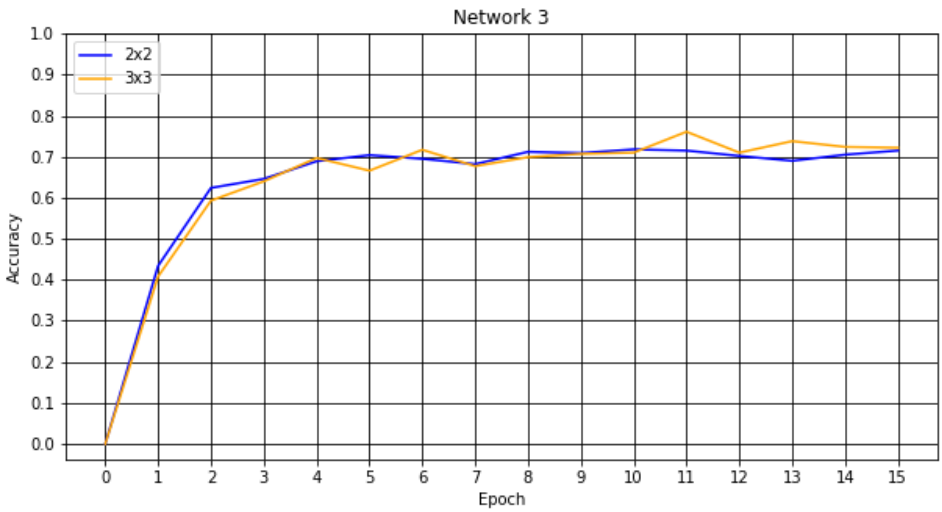
Epoch	Accuracy (2x2)	Accuracy (3x3)
1	0.434	0.408
2	0.624	0.593
3	0.646	0.640
4	0.689	0.697
5	0.704	0.666
6	0.695	0.717
7	0.682	0.677
8	0.712	0.699

Epoch	Accuracy (2x2)	Accuracy (3x3)
9	0.709	0.707
10	0.761	0.718
11	0.715	0.761
12	0.702	0.710
13	0.690	0.738
14	0.70	0.724
15	0.715	0.722

Plotted below are the accuracies:

```
In [36]: from IPython.display import Image
         Image(filename="img/cnn-3_acc_graph.png")
```

Out[36]:



1.3.4 Network 4

Input -> [Conv1 -> Conv2 -> Pool1]*3 -> FC -> FC2 -> Output
Conv1: Used 32 3x3 filters with stride 1.
Conv2: Used 32 3x3 filters with stride 1.
Pool1: Used 2x2/3x3 filter with stride 2.
Conv3: Used 64 3x3 filters with stride 1.
Conv4: Used 64 3x3 filters with stride 1.
Pool2: Used 2x2/3x3 filter with stride 2.
Conv5: Used 128 3x3 filters with stride 1.
Conv6: Used 128 3x3 filters with stride 1.
Pool3: Used 2x2/3x3 filter with stride 2.

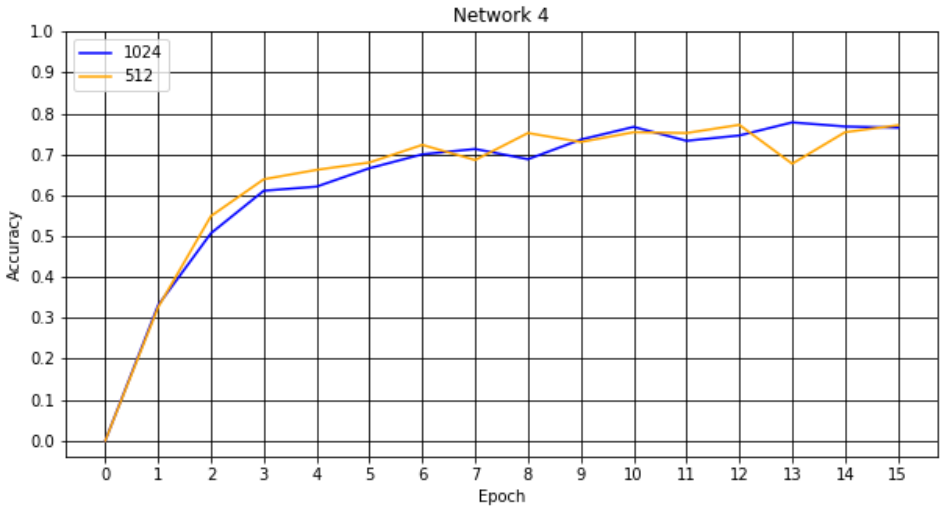
FC: fully connected layer with 512 outputs.
FC2: fully connected layer with 512 outputs.
Output: Output layer with 10 outputs.

Epoch	Accuracy (2x2)	Accuracy (3x3)
1	0.331	0.326
2	0.507	0.549
3	0.611	0.639
4	0.621	0.662
5	0.666	0.680
6	0.700	0.723
7	0.713	0.686
8	0.688	0.752
9	0.736	0.730
10	0.767	0.754
11	0.733	0.752
12	0.746	0.772
13	0.778	0.677
14	0.768	0.754
15	0.765	0.771

Plotted below are the accuracies:

```
In [37]: from IPython.display import Image
         Image(filename="img/cnn-4_acc_graph.png")
```

Out[37]:

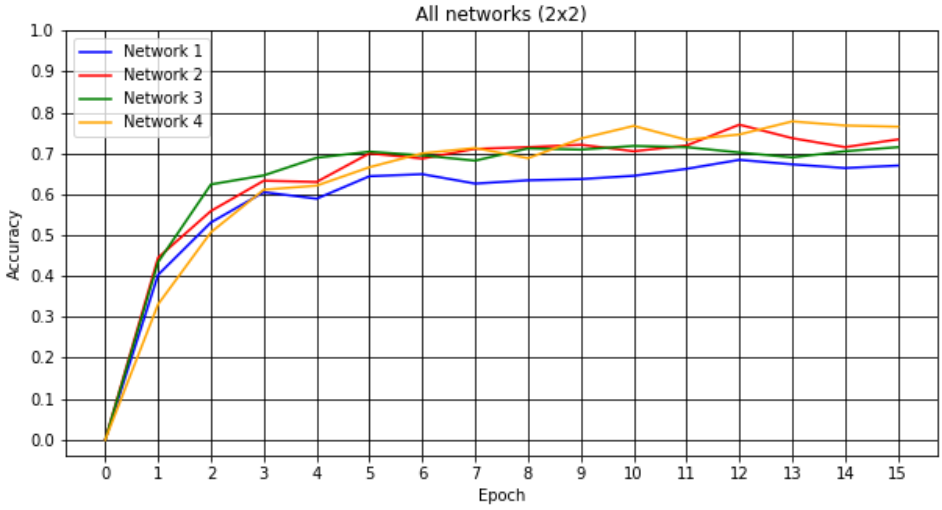


1.3.5 All networks

Here are two graphs, one for 2x2 filters and one for 3x3 filters, of all 4 networks. Note how deeper networks tend to learn more/better. Which makes sense since there are more convolution layers to learn a variety of features.

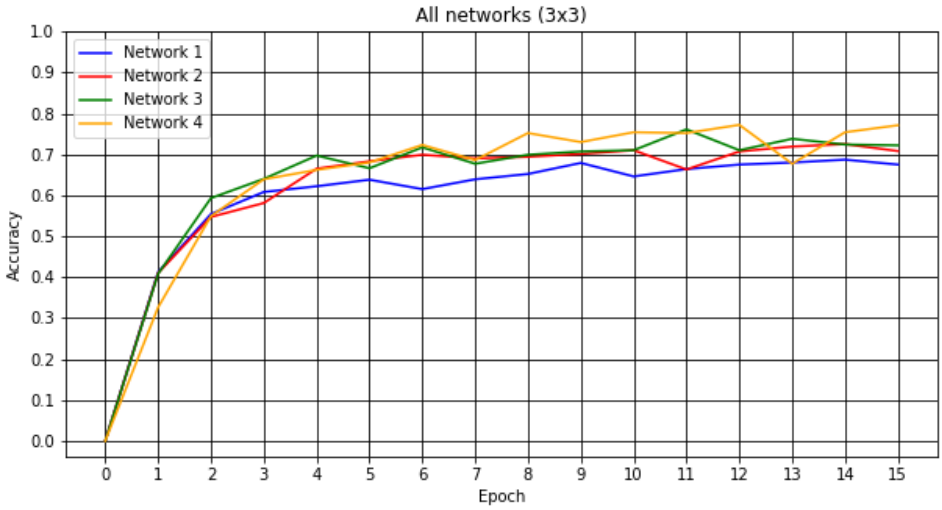
In [38]:
```
from IPython.display import Image
Image(filename="img/cnn-all2x2_acc_graph.png")
```

Out[38]:



In [39]:
```
from IPython.display import Image
Image(filename="img/cnn-all3x3_acc_graph.png")
```

Out[39]:



1.4 References

[1] Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.