# MLP Coursework 2: Exploring Convolutional Networks

Leon Overweel

## Abstract

In this coursework, we implement the convolutional and max pooling layers of a convolutional neural network. We also explore different techniques for broadening the context of a unit in such a network, through the character classification task on the EMNIST dataset. Specifically, we experiment with changing the architecture of the network in terms of kernels per layer (constant per layer, increasing with network depth, and decreasing with network depth) across context broadening techniques including max pooling, average pooling, strided convolutions, and dilated convolutions with three different dilations-per-layer settings. We discover that an equal number of kernels per layer yields the best accuracy all other settings equal, but that models that increase their number of kernels per layer perform better at small scales and small training budgets. We also discover that, at lower dilation settings, all kernels-per-layer architectures perform better than their default-dilation equivalents.

## 1. Introduction

In recent years, artificial intelligence systems have become an increasingly important part of software and workflows across many industries. Such systems work by analyzing many example data points to attempt to "learn" fundamental properties of the data, which they can later apply to make predictions about previously unseen data (s1837379, 2018).

A common approach for leaning such properties for image data is to use convolutional neural networks (LeCun et al., 1998). Rather than fully connecting each input to each output in a layer as in standard deep neural networks, convolutional nets employ kernels that are passed over the previous layer, enabling them to recognize features independent of their location. The size and complexity of the features that a kernel can recognize depend on the amount of context it can process. This in turn depends on the size of the kernel and the depth at which the kernel is used. In this paper, we explore additional techniques to increase context: pooling layers, striding and dilation. Specifically, we investigate how changing the number of kernels or feature maps per layer affects network performance in terms of accuracy and training time across different context broadening techniques and scales.

For our investigation, we use the handwritten character classification dataset EMNIST, which is "a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format" (Cohen et al., 2017). With 26 uppercase and lowercase letters and 10 digits, EMNIST has 62 classes. We use a version of the dataset that merges letters whose upper- and lowercase versions are hard to distinguish at a normalized scale (such as $C$, $I$ and $M$), leaving 47 classes. EMNIST has a total of 131,600 examples, which we split into 100,000 data points (76%) for training, 15,800 (12%) for validation, and 15,800 (12%) for testing (s1837379, 2018).

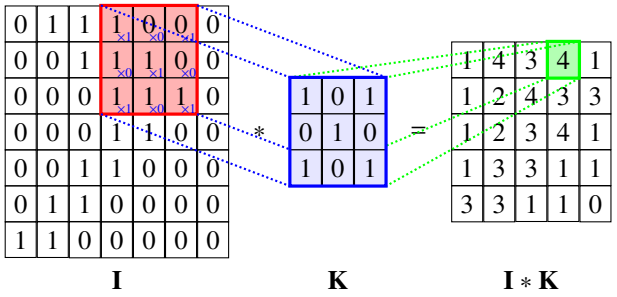


*Figure 1.* A convolution of a $3 \times 3$ kernel $K$ over image $I$, resulting in feature map $I * K$. Illustration by (Velickovic, 2016).

The rest of this paper is structured as follows. In Section 2, we introduce convolutional layers (Section 2.1) and max pooling layers (Section 2.2) and describe a few aspects of our implementation of these layers in the MLP framework; in Section 3, we introduce the concept of context for convolutional neural networks and use the literature to describe the theoretical interaction of pooling layers, striding, and dilation; we also introduce our research questions (Section 3.4). In Section 4, we introduce our experimental setup. In Section 5, we discuss our experimental results. Finally, in Section 6, we conclude.

## 2. Implementing convolutional networks

Fully-connected neural networks trained with back-propagation are a popular model for machine learning because they offer a great deal of flexibility in what kind of different functions they can learn that map input data to output data. However, they do not work particularly well for images, the type of input data that we are interested in, because they have no concept of locality: a feature in an image (such as the end of a line) may appear in many locations but a fully-connected neural network is not equipped to model these occurrences in a single, abstracted way, which leads to modeling and computational inefficiencies (LeCun et al., 1998).

### 2.1. Convolutional Layers

Convolutional layers in neural networks solve this locality problem by using feature detectors ("kernels") that are passed over the image to produce feature maps with hidden units that have local receptive fields (LeCun et al., 1998), as illustrated in Figure 1. As an added benefit, since the kernel's weights are shared between each input location, the number of parameters to train

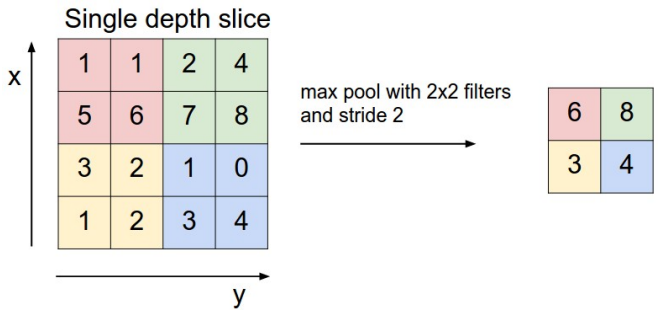


*Figure 2.* A max pool operation with filter size $2 \times 2$, which looks at a group of four neighboring activations and propagates only the highest-value one. In this example, a $2 \times 2$ kernel passed over the right layer has a receptive field of $4 \times 4$ on the left layer. Illustration by (Karpathy, 2015b).

is vastly smaller than it would be for a fully-connected network.

Typically, a number of different kernels will be passed over the same input image to produce different feature maps: for the first layer of a handwritten character recognition network, these may be detectors for different orientations of edges and line endings (LeCun et al., 1998). To keep the image size the same from layer to layer, we may employ padding, which involves adding a border of $\frac{k-1}{2}$ pixels around the original image, where $k$ is the kernel size.

#### 2.1.1. Implementation

We implement convolutional layers in the MLP framework as follows. We choose to use the approach using SciPy's `convolve2d` method over the `im2col` serialization method. The latter is faster because it uses fewer, larger matrix multiplications, but since we are building a naive CPU-based implementation, neither method will be anywhere near as fast as optimized frameworks anyway. We therefore prefer the more intuitive `convolve2d` method. To handle bad inputs, we also add assertions to check that...

- all channel counts and layer and kernel sizes are integers, and
- the kernel is not larger than the input.

We do not implement caching on the convolutional layer.

### 2.2. Max Pooling Layer

As we progress beyond the first convolutional layer, we want to start to detect larger features in our image: instead of looking for just line parts and endings, for example, we're interested in how those lines fit together. To do this, we must extend the size of the piece of the image that the kernel in a hidden layer reasons about, called the "receptive field".

Instead of the obvious solutions of increasing the size of the kernel itself or adding more layers to the network, each of which would increase the receptive field but could also could adversely affect the location invariance and/or generalizability of the network (Luo et al., 2016), we can employ a technique called "pooling", as illustrated in Figure 2. The figure illustrates max pooling, where the kernel only passes on the maximum value of the values it sees. Other functions, such as averaging, are also possible. According to the literature, max pooling is

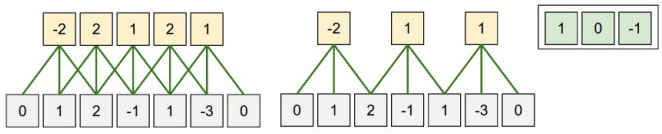


*Figure 3.* Stride demonstration with a kernel of size 3 (top right, green) being passed over a one-dimensional image of size 7 (gray). At stride 1 (left), the kernel is passed over every pixel (except the two edges) and produces an output of size 5; at stride 2 (right), the kernel is passed over every other pixel (except the two edges) and produces an output of size 3. Illustration by (Karpathy, 2015c).

especially well-suited to the separation of sparse binary features, and the size of the pooling kernel should increase as the number of feature maps per layer increases (Boureau et al., 2010).

Pooling also helps decrease computational complexity, by reducing the size of the next convolutional layer quadratically by pooling filter size. Although pooling has been successfully employed to increase the accuracy of convolutional neural networks ((LeCun et al., 1998)), more recent literature suggests that pooling will be employed less in the future, in favor of other techniques to increase a model's context (Karpathy, 2015a); see Section 3 for examples of such techniques.

#### 2.2.1. Implementation

We implement max pooling layers in the MLP framework as follows. As before, we use `conv2d`. We implement caching by saving the index of each maximum on the forward pass. Then, on the backward pass, we can simply create a zero matrix and iterate over the cache to fill in the gradients at the maximum indices. Besides being more computationally efficient, it also makes the implementation of back propagation quite simple. To handle bad inputs, we also add assertions to check that...

- the input size, kernel size, and stride are integers,
- the size and stride are not larger than the input, and
- the kernel will cover the entire image: `input_size−size` mod `stride` = 0.

## 3. Context in convolutional networks

In the rest of this report, we explore different modifications to convolutional neural network architectures that can be used to improve the network's concept of context. This is also known as the *receptive field* or *field of view* that a unit on a particular layer of the network has of the original image (Luo et al., 2016). A network can reason about context in several ways, including pooling layers (see Section 2.2), striding, and dilation[1]. We describe the latter two in more detail here.

### 3.1. Stride

The stride affects the way in which we pass the kernel over the image. The larger the stride, the more pixels the center of the kernel skips between positions, and the larger the receptive field or context of the output.

Intuitively, we can see this in Figure 3. Imagine a secondary kernel of size 2 being passed over the output of the convolution

[1]Network depth and kernel size also contribute to a unit's field of view (Luo et al., 2016), but we do not focus on these in this work.

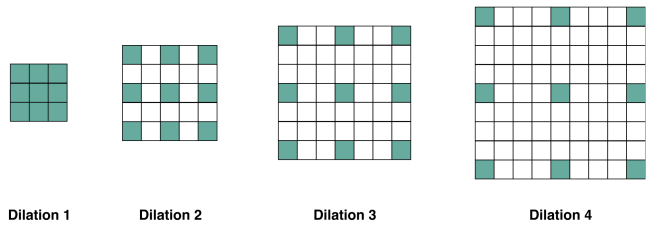


*Figure 4.* Kernel shapes at different dilation settings. Green pixels are captured by the kernel, while white pixels are not considered. At dilation 1, the kernel is the same as a non-dilated kernel. At higher dilations, the receptive field of the kernel increases (from $3 \times 3$ at dilation 1 to $9 \times 9$ at dilation 4) while the number of parameters stays constant (at 9). Illustration by (Antoniou et al., 2018).

in Figure 3 (in yellow), starting at the first two pixels on the left. For the left output, which was convolved with a stride of 2, its first two pixels were constructed from pixels 1 through 3 (with values $0, 1, 2$) and 2 through 4 $(1, 2, -1)$ of the original image (in gray); this makes the receptive field on the original image of secondary kernel 4: it can detect features of at most size 4. For the right output, its first two pixels were constructed from pixels 1 through 3 $(0, 1, 2)$ and 3 through 5 $(2, -1, 1)$; this makes its receptive field 5: it can detect larger features than the previous configuration.

On top of increasing the receptive field of the next layer, a higher stride will also decrease the size of that layer and the reduce computational load of processing that layer. It is common practice to only use a stride greater than 1 on pooling layers and keep the stride at 1 for convolutional layers (Karpathy, 2015a).

### 3.2. Dilation

Another approach to increasing the context of a convolutional layer is to use a dilated kernel, as illustrated in Figure 4. A dilated convolution is similar to the *algorithme à trous* algorithm, except that it is constructed by adjusting the convolution operator instead of the the naive (and inefficient) approach of simply creating a larger filter padded with zeros where the white pixels are in Figure 4 to create the effect of a dilation (Yu & Koltun, 2015).

The benefit of using dilation as expansion technique is that (with appropriate padding around the source layer), we can increase our receptive field without decreasing resolution and therefore plug dilated convolutions into existing architectures easily. This allows us to learn higher-order features without decreasing dimensionality (Antoniou et al., 2018). However, not reducing resolution also means that we do not get the computational benefit of decreasing the size of our layers as we progress to deeper layers.

### 3.3. Combining Pooling, Stride, and Dilation

Although all these dimensionality reduction techniques affect the receptive field, they are not mutually exclusive. For example, we can use different strides for convolutional layers, pooling layers, and dilated convolution layers; for each of these, we are still passing a kernel over an image or a hidden layer, and the amount by which we move the kernel on each step is a variable to experiment with.

Of course, we must adjust these hyperparameters with care in respect to each other. For example, if we apply a dilated convolution with a stride equal to the dilation (and greater than 1), we are essentially subsampling the image: for dilation 2 and stride 2 on a $3 \times 3$ kernel, for example, the kernel would only be looking at pixels that are on both odd rows and odd columns, and completely disregarding all pixels whose row or column is even!

### 3.4. Research Question

Many seminal convolutional neural network architectures that employ forms of dimensionality reduction also vary the number of kernels (or feature maps) that each layer has: LeNet-5 has two convolutional layers with 6 and 16 kernels per layer (LeCun et al., 1998); and AlexNet has five convolutional layers with 48, 128, 128, 192, and 192 kernels per layer (Krizhevsky et al., 2012). Therefore we ask the following question:

***Q1:*** *For different dimensionality reduction types, how does changing the network architecture in terms of number of kernels / feature maps per layer affect the network's performance in terms of accuracy and speed?* Is it better to increase the number of feature maps as we progress to further layers, decrease it, or keep it the same? We report our experiments to answer this question in Section 4.1. Based on the results of these experiments, we investigate this question in more depth for dilated convolutions with the following question:

***Q2:*** *How do different architectures for dilations per layer and kernels per layer affect the network's performance in terms of speed and accuracy?* Since our images are small, would lower dilations work better than the default $2, 3, 4, 5$ dilations per layer?

## 4. Experiments

Both research questions **Q1** and **Q2** deal with varying the number of kernels / feature maps per layer. We identify three ways to vary this architecture:

- **Group A:** Use a **constant amount** of kernels on each layer. Since this is the default in the MLP framework, this will serve as our baseline.
- **Group B:** Use an **increasing amount** of filters per layer as we progress to deeper layers. Intuitively, this approach makes sense to use with expanding contexts: as we get deeper in the network and our receptive fields are larger, we will want to identify more complex features, so having more kernels tuned to pick up different complex features on deeper layers makes sense.
- **Group C:** Use a **decreasing amount** of filters per layer as we progress to deeper layers. In computer science (and in life in general), testing the for the opposite of our intuition is always a good idea.

For each of these three settings, we must also consider scale: a group may, for example, perform best in terms of accuracy at a large scale (interesting if we have a large training budget) while another might work best in terms of time on a small scale (interesting if we have a small training budget). We experiment on the following kernels-per-layer architectures:

- For Group A, we try five constant kernel counts of 4, 8,

| | Max Pooling | | | Average Pooling | | | Strided Conv. | | | Dilated Conv. | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Kernels/Layer** | **ID** | **Acc** | **s/Ep** | **ID** | **Acc** | **s/Ep** | **ID** | **Acc** | **s/Ep** | **ID** | **Acc** | **s/Ep** |
| 4, 4, 4, 4 | *1* | 0.746 | 5.1 | *2* | 0.781 | 5.0 | *3* | 0.815 | 7.3 | *4* | 0.813 | 11.1 |
| 8, 8, 8, 8 | *5* | 0.850 | 5.4 | *6* | 0.851 | 5.4 | *7* | 0.869 | 7.7 | *8* | 0.866 | 13.6 |
| 16, 16, 16, 16 | *9* | 0.875 | 6.3 | *10* | 0.882 | 6.4 | *11* | 0.878 | 8.5 | *12* | 0.883 | 18.2 |
| 32, 32, 32, 32 | *13* | **0.886** | 8.9 | *14* | 0.888 | 9.1 | *15* | 0.882 | 11.2 | *16* | 0.887 | 25.7 |
| 64, 64, 64, 64 | *17* | 0.884 | 17.1 | *18* | **0.889** | 17.6 | *19* | **0.887** | 20.5 | *20* | **0.891** | 61.7 |
| 4, 8, 16, 32 | *21* | 0.876 | 5.4 | *22* | 0.878 | 5.4 | *23* | 0.870 | 7.7 | *24* | 0.884 | 15.0 |
| 8, 16, 32, 64 | *25* | 0.883 | 6.8 | *26* | 0.884 | 6.8 | *27* | 0.879 | 9.3 | *28* | 0.885 | 22.8 |
| 16, 32, 64, 128 | *29* | 0.885 | 10.4 | *30* | 0.886 | 10.4 | *31* | 0.881 | 15.0 | *32* | 0.890 | 70.1 |
| 32, 16, 8, 4 | *33* | 0.835 | 7.1 | *34* | 0.839 | 7.3 | *35* | 0.860 | 9.4 | *36* | 0.867 | 18.0 |
| 64, 32, 16, 8 | *37* | 0.873 | 11.0 | *38* | 0.881 | 11.4 | *39* | 0.875 | 14.8 | *40* | 0.881 | 34.6 |
| 128, 64, 32, 16 | *41* | 0.881 | 25.2 | *42* | 0.888 | 25.7 | *43* | 0.880 | 31.1 | *44* | 0.890 | 85.5 |

*Table 1.* Highest validation set accuracy (**Acc**) achieved and the average training time per epoch in seconds (**s/Ep**) for each combination of dimensionality reduction type and number of feature maps per convolutional layer, where models are uniquely identified by their **ID**s. Best accuracies are highlighted for each dimensionality reduction type. Horizontal lines separate groups A, B and C.

16, 32, and 64 per layer[2].

- For Group B, we increase kernel counts on three different scales: 4, 8, 16, 32; 8, 16, 32, 64; and 16, 32, 64, 128.
- For Group C, we decrease kernel counts on three different scales: 128, 64, 32, 16; 64, 32, 16, 8; and 32, 16, 8, 4.

We use these same groups and scales of kernels per layer for all experiments across **Q1** and **Q2** experiments. Unless otherwise specified, we use a seed of 424242 to train our models, batch size 100, and the PyTorch default of uniform random weights/biases initialization between $\pm\frac{1}{\sqrt{1ik}}$, where $k$ are kernel sizes.

### 4.1. Q1: Kernels Per Layer

For each of architectures, we investigate how it performs in terms of validation set accuracy and average epoch training time across different context techniques / dimensionality reduction types. For each of these types, unless otherwise specified, we use the default MLP framework settings of kernel size 3, padding 1, stride 1, dilation 1. The types we test for are Max Pooling, Average Pooling, Strided Convolution (with stride 2), and Dilated Convolutions (dilation $i + 2$, where $i$ is the zero-indexed index of the layer).

In total, this yields 11 kernels-per-layer architectures times 4 dimensionality reduction types equals 44 different models to train for our first set of experiments. We label them as models *1 - 44*. The results of running these experiments are summarized in Table 1 and Figure 5; we discuss these results in Section 5.1.

### 4.2. Q2: Dilations

In our experiments for **Q1**, all kernels per layer architectures performed best on Dilated convolution (see Table 1 and Section 5.1), so we decide to investigate dilations further. Again, we use MLP framework default settings of stride 1, padding 1 and kernel size 3. We try the following settings for dilations, with the first number corresponding to the dilation for the first layer, etc.:

- 2, 3, 4, 5: This is the default; so we already ran these settings in models *4, 8, 12, ..., 44*.
- 1, 2, 3, 1: This setting is inspired by (Yu & Koltun, 2015), which first increases and then decreases dilation per layer.
- 1, 2, 3, 4: This setting is inspired by the default, but scaled down because, after all, we are dealing with relatively small $28 \times 28$ pixel images.

In total, this yields 11 kernels-per-layer architectures times 2 additional dilations settings equals 22 models to train for our second set of experiments. We label these as models *45 - 66*. The results of these experiments are summarized in Table 2 and Figure 5; we discuss these results in Section 5.2.

### 4.3. Final Comparison

For our final model comparison, we calculate and report test scores and error bars for the few best and most interesting models. This includes models *13*, *18*, *19* and *20*, which had the best accuracies in our first set of experiments; models *54* and *59*, which had the best accuracies for the second set; and model *55*, which scored surprisingly high for its kernel counts and training time.

We run each model three times, with seeds 424242, 87654 and 34958. We calculate error bars on accuracy as the mean of these runs ± the standard deviation. The results of these runs are in Table 3; we discuss the results in Section 5.3.

## 5. Discussion

### 5.1. Q1: Kernels Per Layer

Table 1 and the first four columns of Figure 5 provide two views on the 44 experiments we ran to answer **Q1**. The table allows us to compare models in terms of their highest validation set score vs. their training time, while the graph shows us where different models underfit or start to underfit. We present our conclusions from each in turn.

First, we can see from the highlighted accuracy best scores in Table 1 that Group A (the baseline with equal kernel counts per layer) slightly outperforms Groups B and C in terms of highest overall validation set accuracy. For average pooling, strided convolutions, and dilated convolutions, the models with 64 kernels per layer perform best (models *18*, *19*, and *20*, respectively); for max pooling, the model with 32 kernels per layer

[2]Initial experiments showed that training four full layers of 128 kernels would take too long.

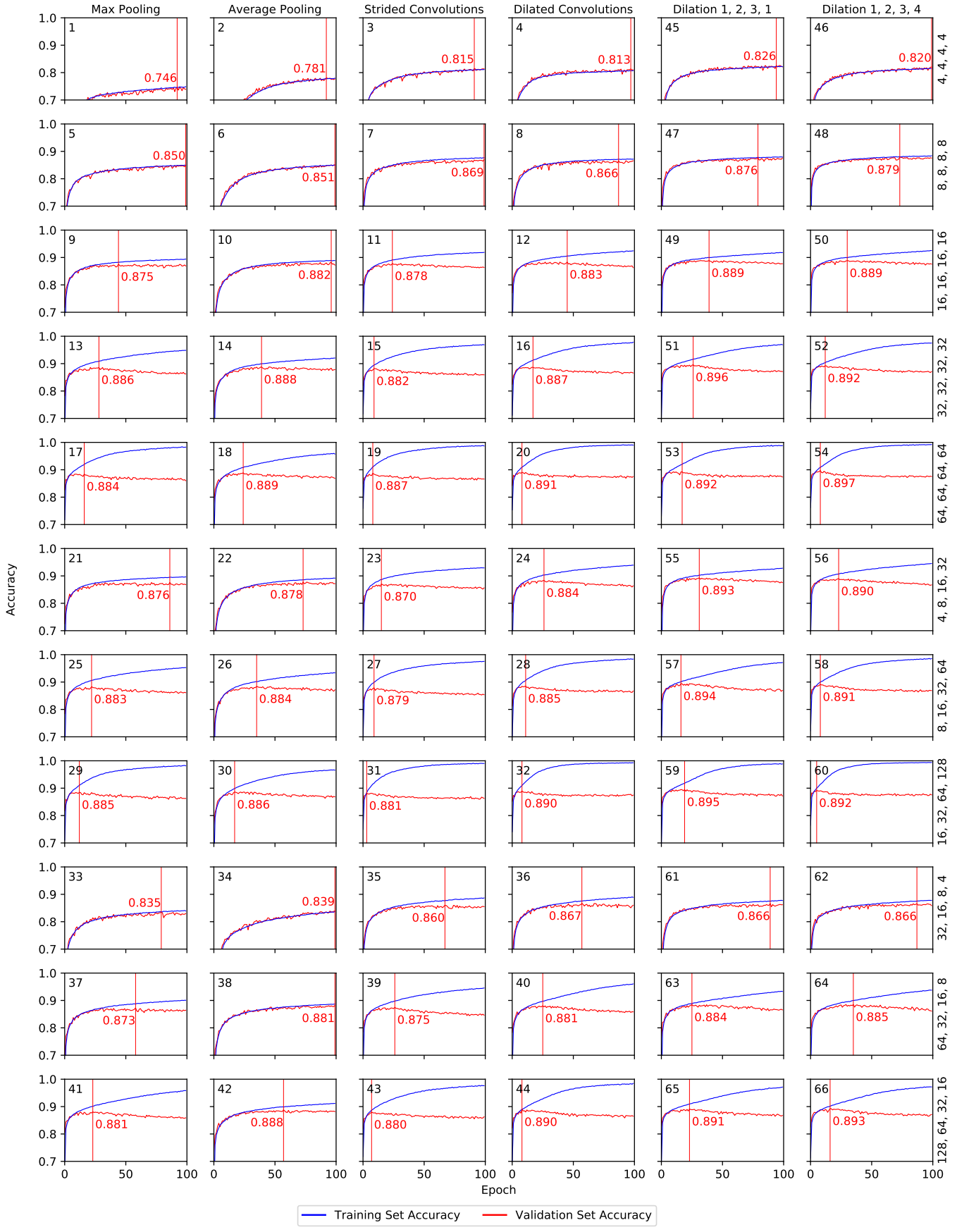


*Figure 5.* Training (blue) and validation (red) curves for models 1 - 66, resulting from the experiments described in Sections 4.1 (first four columns) and 4.2 (last two columns). The black number at the top left of each graph indicates the model ID. The red line indicates the epoch that reached the highest validation score; the red number indicates the highest validation score. Labels at the top indicate dimensionality reduction type; labels on the right indicate the architecture in terms of kernels per layer.

| | Dilation 1, 2, 3, 1 | | | Dilation 1, 2, 3, 4 | | |
|---|---|---|---|---|---|---|
| **Kernels/Layer** | **ID** | **Acc** | **s/Ep** | **ID** | **Acc** | **s/Ep** |
| 4, 4, 4, 4 | *45* | 0.826 | 15.0 | *46* | 0.820 | 14.4 |
| 8, 8, 8, 8 | *47* | 0.876 | 18.2 | *48* | 0.879 | 17.6 |
| 16, 16, 16, 16 | *49* | 0.889 | 23.9 | *50* | 0.889 | 23.1 |
| 32, 32, 32, 32 | *51* | 0.896 | 33.3 | *52* | 0.892 | 31.8 |
| 64, 64, 64, 64 | *53* | 0.892 | 76.9 | *54* | **0.897** | 73.6 |
| 4, 8, 16, 32 | *55* | 0.893 | 22.7 | *56* | 0.890 | 21.1 |
| 8, 16, 32, 64 | *57* | 0.894 | 35.5 | *58* | 0.891 | 32.3 |
| 16, 32, 64, 128 | *59* | **0.895** | 77.0 | *60* | 0.892 | 67.9 |
| 32, 16, 8, 4 | *61* | 0.866 | 21.9 | *62* | 0.866 | 21.1 |
| 64, 32, 16, 8 | *63* | 0.884 | 33.7 | *64* | 0.885 | 32.7 |
| 128, 64, 32, 16 | *65* | 0.891 | 86.3 | *66* | 0.893 | 27.9 |

*Table 2.* Highest validation set accuracy (**Acc**) achieved and the average training time per epoch in seconds (**s/Ep**) for each combination of dilation setting per layer and number of feature maps per convolutional layer, where models are uniquely identified by their **ID**s. Best accuracies are highlighted for each dimensionality reduction type. Horizontal lines separate groups A, B and C.

| **ID** | **Accuracy** |
|---|---|
| *13* | 0.875 ± 0.0016 |
| *18* | 0.881 ± 0.0010 |
| *19* | 0.875 ± 0.0011 |
| *20* | 0.882 ± 0.0016 |
| *54* | 0.884 ± 0.0010 |
| *55* | 0.882 ± 0.0027 |
| *59* | 0.889 ± 0.0020 |

*Table 3.* Test set accuracies of select models with error bars. Error bars are computed as the mean of running with three different ± the standard deviation of these three runs.

performs best (model *13*). So, in terms of accuracy, changing the number of kernel maps to either be increasing or decreasing from layer to layer (at least in the configurations we tried) does not improve network performance.

More interesting results emerge if we consider accuracy in relation to training time. We can compare the models that take up to 5.5 seconds per epoch to train, for example. Here, Group A models *1*, *2*, *5* and *6* all have much worse accuracies than Group B models *21* and *22*. We can see similar patterns in Table 1 for the relatively small models trained using other dimensionality reduction types as well. So, on constrained training budgets, an architecture of increasing the number of kernels per layer improves training accuracy. Another timing-related observation is that dilated convolutions are relatively slow to train compared to other dimensionality reduction types, and even more so for Groups B and C than for Group A.

Finally, Table 1 also confirms our intuition that increasing kernels per layer performs better than decreasing kernels per layer: across the board Group C models train more slowly when compared to equivalently-size Group B models or Group A models with similar training times; they also generally result in relatively worse accuracies.

We can also gain some insight from the graphs in Figure 5. First, the common rule of thumb that models with more parameters overfit more on the training set holds. The opposite holds as well: the smaller models (*1 - 8*, *20 - 24*, and *33 - 37*) appear to underfit and would probably get better accuracies if they were trained longer (whenever the red line in Figure 5 hugs the right axis of a graph, this indicates that the model underfitted). Another observation from this figure is that, irrespective of kernel count per layer, both strided and dilated convolutions overfit faster than max and average pooling.

### 5.2. Q2: Kernels and Dilations Per Layer

Table 2 and the last two columns of Figure 5 show the results of the experiments we ran to answer **Q2**.

First, we can see a pronounced improvement: for both dilations $1, 2, 3, 1$ and $1, 2, 3, 4$, every single kernels-per-layer architecture gets to a better best validation set accuracy score than the equivalent models trained with max pooling, average pooling, strided convolutions, or the default $2, 3, 4, 5$ dilated convolutions. The best model, *55*, has the highest validation set accuracy of 0.897. They also, however, all take much longer to train.

If we examine the relatively small models again, we can see the improvement is especially large: model *55*, with $4, 8, 16, 32$ kernels per layer trained with dilations $1, 2, 3, 1$, for example, outperforms the best model (*20*) from the first set of experiments in terms of accuracy (0.893 vs 0.891) and training time (22.7 vs 61.7 s/epoch).

### 5.3. Final Comparison

For our final model comparison, as described in Section 4.3, we compute error bars on the test set accuracy of a few interesting models. Across the board, the test set accuracies are about 1% worse than the validation set accuracies, which could indicate that we slightly overfitted our architecture and hyperparameters on the validation set.

However, the general trend we discovered in our **Q2** experiments holds: the models with dilations $1, 2, 3, 1$ and $1, 2, 3, 4$ outperform the model with the default dilations $2, 3, 4, 5$. Also, model *55* once again performed similarly to bigger-scale models, showing that a small model with increasing kernels per layer and dilations $1, 2, 3, 1$ indeed outperforms the baselines.

## 6. Conclusions

In this work, we experimented with changing the number of kernels per layer in four-layer convolutional neural networks across different context broadening techniques, including max pooling, average pooling, strided convolutions, and dilated convolutions with three different dilations-per-layer settings. From research question **Q1**, we discovered that, although the baseline of having an equal number of kernels per layer yields the highest overall accuracy, models that increase their number of kernels per layer with layer depth perform better at small scales: with relatively short training time and a relatively low number of parameters to train, these models score higher in accuracy than models of similar scale. From research question **Q2**, we discovered that smaller dilations perform better than the default dilations in the MLP network.

## References


Antoniou, Antreas, Słowik, Agnieszka, Crowley, Elliot J, and Storkey, Amos. Dilated densenets for relational reasoning. *arXiv preprint arXiv:1811.00410*, 2018.

Boureau, Y-Lan, Ponce, Jean, and LeCun, Yann. A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 111–118, 2010.

Cohen, Gregory, Afshar, Saeed, Tapson, Jonathan, and van Schaik, André. Emnist: an extension of mnist to handwritten letters. *arXiv preprint arXiv:1702.05373*, 2017.

Karpathy, Andrej. Convolutional Neural Networks, 2015a. URL http://cs231n.github.io/convolutional-networks/. [Online; accessed November 21, 2018].

Karpathy, Andrej. Max Pooling (illustration), 2015b. URL http://cs231n.github.io/convolutional-networks/. [Online; accessed November 19, 2018].

Karpathy, Andrej. Stride (illustration), 2015c. URL http://cs231n.github.io/convolutional-networks/. [Online; accessed November 21, 2018].

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Luo, Wenjie, Li, Yujia, Urtasun, Raquel, and Zemel, Richard. Understanding the effective receptive field in deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 4898–4906, 2016.

s1837379. MLP Coursework 1: Learning Algorithms and Regularization, 2018. [Accessed November 22, 2018].

Velickovic, Petar. 2D Convolution (illustration), 2016. URL https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution. [Online; accessed November 18, 2018].

Yu, Fisher and Koltun, Vladlen. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.