

# Deep Learning



## Deep Learning

Lecture: Convolutional Neural Networks  
(Part 2)

Ted Scully

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

-1	0	1
-1	0	1
-1	0	1

Filter (Kernel)

**Feature Map**  
(Result of Convolution)


- ▶ A convolution involves repeatedly applying a **filter** (kernel) to an original image.
- ▶ A filter allows us to extract specific features from the original image.

5 -1	1 0	3 1	5	7	9
3 -1	4 0	4 1	0	7	8
6 -1	1 0	8 1	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

-1	0	1
-1	0	1
-1	0	1

**Feature Map**  
(Result of Convolution)

1			

$$\begin{aligned}
 &(5 * -1) + (1 * 0) + (3 * 1) + \\
 &(3 * -1) + (4 * 0) + (4 * 1) + \\
 &(6 * -1) + (1 * 0) + (8 * 1)
 \end{aligned}$$

Notice that we move (slide) the filter over one pixel width to the right

5	1 -1	3 0	5 1	7	9
3	4 -1	4 0	0 1	7	8
6	1 -1	8 0	6 1	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

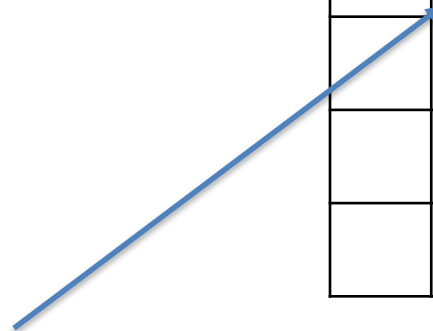
Original Image

-1	0	1
-1	0	1
-1	0	1

$$\begin{aligned}
 &(1*-1) + (3*0) + (5*1) + \\
 &(4*-1) + (4*0) + (0*1) + \\
 &(1*-1) + (8*0) + (6*1)
 \end{aligned}$$

Feature Map  
(Result of Convolution)

1	5		



5	1	3-1	5 0	7 1	9
3	4	4-1	0 0	7 1	8
6	1	8-1	6 0	3 1	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

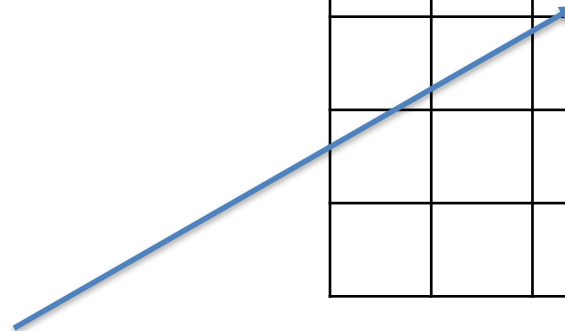
Original Image

-1	0	1
-1	0	1
-1	0	1

$$\begin{aligned}
 &(3*-1) + (5*0) + (7*1) + \\
 &(4*-1) + (0*0) + (7*1) + \\
 &(8*-1) + (6*0) + (3*1)
 \end{aligned}$$

**Feature Map**  
(Result of Convolution)

1	5	2	



5	1	3	5 -1	7 0	9 1
3	4	4	0 -1	7 0	8 1
6	1	8	6 -1	3 0	4 1
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

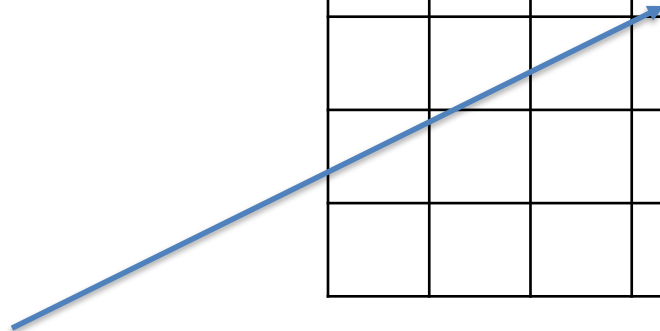
Original Image

-1	0	1
-1	0	1
-1	0	1

$$\begin{aligned}
 &(5*-1) + (7*0) + (9*1) + \\
 &(0*-1) + (7*0) + (8*1) + \\
 &(6*-1) + (3*0) + (4*1)
 \end{aligned}$$

**Feature Map**  
(Result of Convolution)

1	5	2	10



5	1	3	5	7	9
3 -1	4 0	4 1	0	7	8
6 -1	1 0	8 1	6	3	4
1 -1	3 0	5 1	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

-1	0	1
-1	0	1
-1	0	1

$$\begin{aligned}
 &(3*-1) + (4*0) + (4*1) + \\
 &(6*-1) + (1*0) + (8*1) + \\
 &(1*-1) + (3*0) + (5*1)
 \end{aligned}$$

Feature Map  
(Result of Convolution)

1	5	2	10
7			

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8 -1	4 0	6 1
1	7	5	3 -1	1 0	2 1
7	2	2	6 -1	4 0	6 1

Original Image

**Feature Map**  
(Result of Convolution)

1	5	2	10
7	.	.	.
.	.	.	.
.	.	.	.

Notice here we have a **6\*6** matrix initially. When we convolve it with a **3\*3** filter we get a **4\*4** matrix.

We can calculate the row dimension of the resulting matrix as follows:

**$n - k + 1$**  where:

- **n** is the number of *rows* in the original matrix
- **k** is the number of rows in the filter.

We can apply the same equation to calculate the number of columns (where n is original number of columns and k is number of filter columns).



5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

?	?	?
?	?	?
?	?	?

Filter (Kernel)

1	5	2	10
7	.	.	.
.	.	.	.
.	.	.	.

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

?	?	?
?	?	?
?	?	?

Filter (Kernel)

1	5	2	10
7	.	.	.
.	.	.	.
.	.	.	.

$+$  bias

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

?	?	?
?	?	?
?	?	?

Filter (Kernel)

$$\text{RELU}\left( \begin{array}{|c|c|c|c|} \hline 1 & 5 & 2 & 10 \\ \hline 7 & . & . & . \\ \hline . & . & . & . \\ \hline . & . & . & . \\ \hline \end{array} + \boxed{\text{bias}} \right)$$

- ▶ Two types of padding (**Valid** and **Same**)

0	0	0	0	0	0	0	0
0	5	1	3	5	7	9	0
0	3	4	4	0	7	8	0
0	6	1	8	6	3	4	0
0	1	3	5	8	4	6	0
0	1	7	5	3	1	2	0
0	7	2	2	6	4	6	0
0	0	0	0	0	0	0	0

## Original Image

?	?	?
?	?	?
?	?	?

## Filter (Kernel)

## Result of Convolution

[illegible]

# Stride

- By default when performing our convolution we slide the filter in steps of **one pixel at a time**.
- The stride is a integer parameter that is used to specify the **step size in pixels** when performing convolutions (in the previous example our stride  $s = 1$ ).
- Notice below we are moving the filter in two pixels steps ( $s = 2$ )

0 ?	0 ?	0 ?	0	0	0	0
0 ?	5 ?	1 ?	3	5	7	0
0 ?	3 ?	4 ?	4	0	7	0
0	6	1	8	6	3	0
0	1	3	5	8	4	0
0	1	7	5	3	1	0
0	0	0	0	0	0	0

# Stride

- By default when performing our convolution we slide the filter in steps of one pixel at a time.
- The stride is a integer parameter that is used to specify the **step size in pixels** when performing convolutions (in the previous example our stride  $s = 1$ ).
- Notice below we are moving the filter in two pixels steps ( $s = 2$ )

0	0	0 ?	0 ?	0 ?	0	0
0	5	1 ?	3 ?	5 ?	7	0
0	3	4 ?	4 ?	0 ?	7	0
0	6	1	8	6	3	0
0	1	3	5	8	4	0
0	1	7	5	3	1	0
0	0	0	0	0	0	0

# Stride

- By default when performing our convolution we slide the filter in steps of one pixel at a time.
- The stride is a integer parameter that is used to specify the **step size in pixels** when performing convolutions (in the previous example our stride  $s = 1$ ).
- Notice below we are moving the filter in two pixels steps ( $s = 2$ )

0	0	0	0	0 ?	0 ?	0 ?
0	5	1	3	5 ?	7 ?	0 ?
0	3	4	4	0 ?	7 ?	0 ?
0	6	1	8	6	3	0
0	1	3	5	8	4	0
0	1	7	5	3	1	0
0	0	0	0	0	0	0







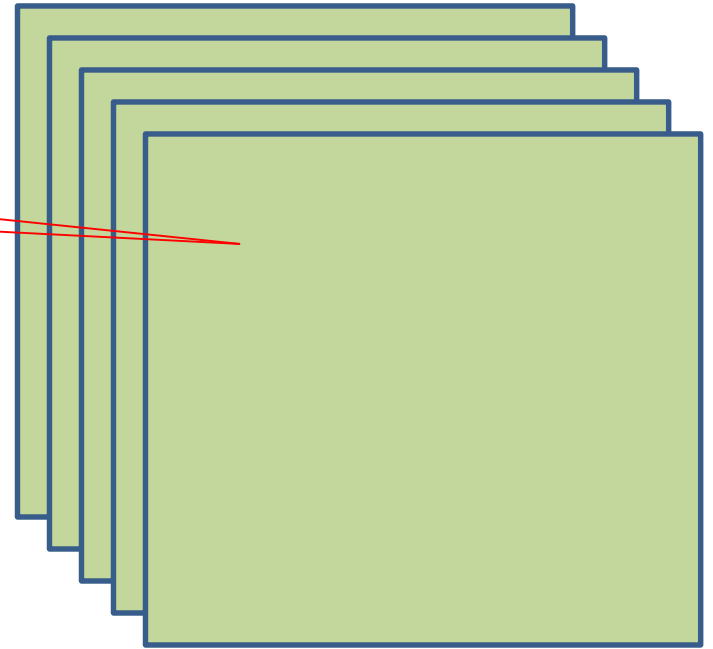
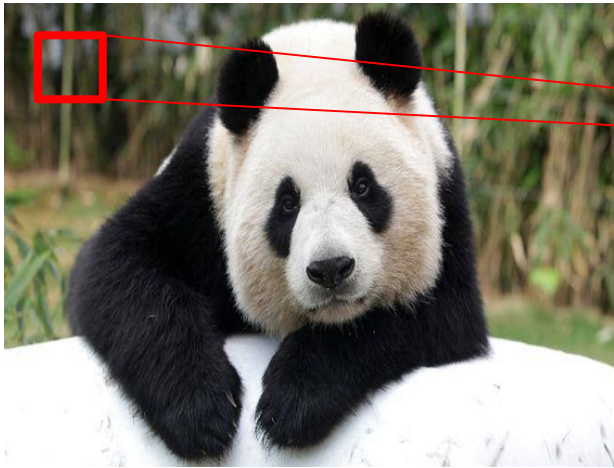






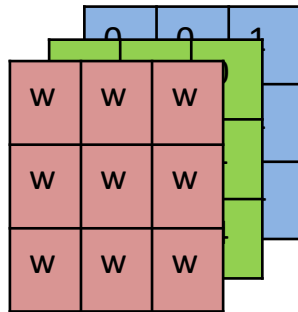


**Feature Map: A map of the filter over the input** (indicating the response of that filter pattern at different locations in the input. )



**Feature Map**

You will remember that previously we said that after the convolution process is performed, we take the resulting **feature map**, **add a bias** and pass the result through a **ReLU** activation function.

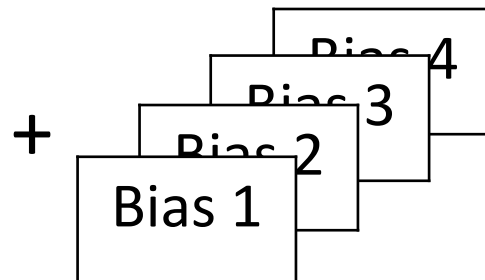
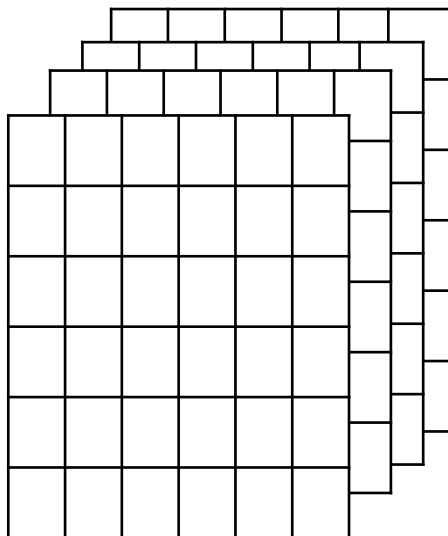
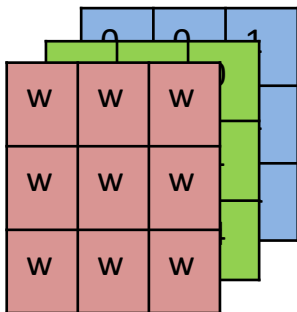
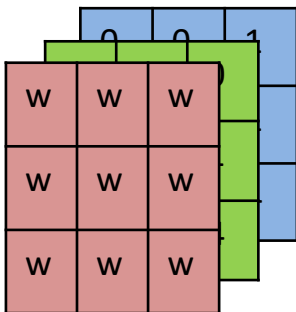
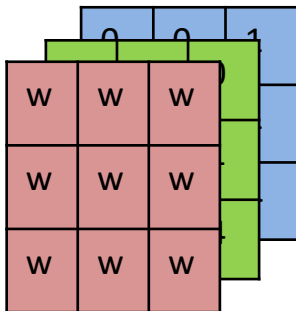
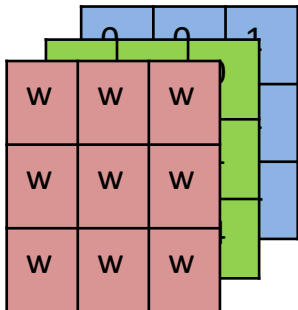


ReLU(

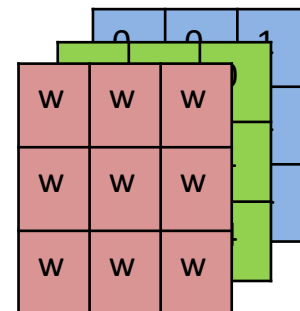
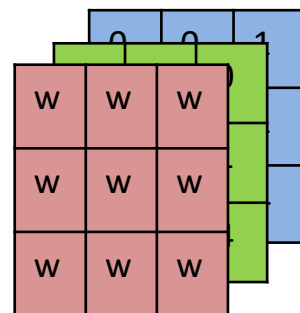
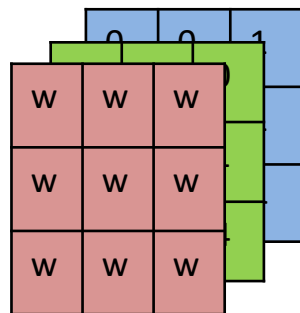
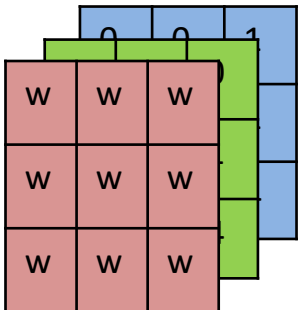

+

Bias 1

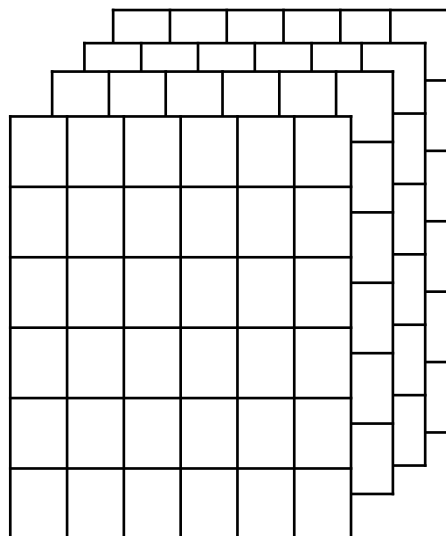
)



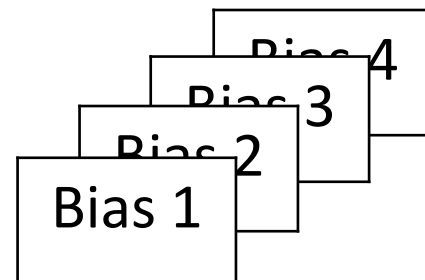




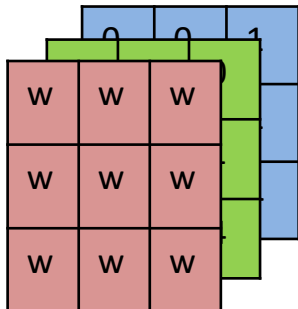
RELU(



+

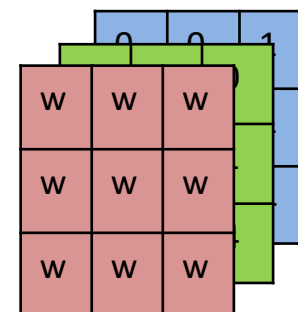
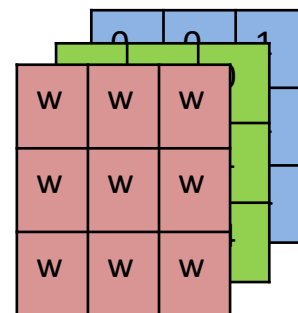
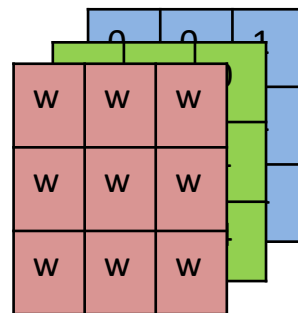


)

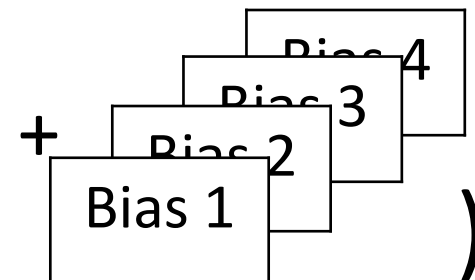
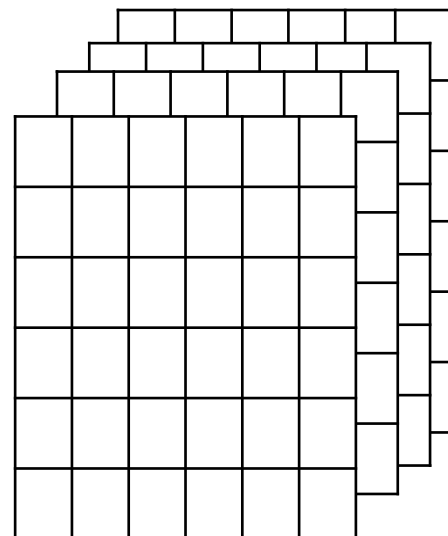


This is a typical first layer of a CNN. Notice how similar it is to what we have previously seen with neural networks. Each of our filters contain weights that our network must learn.

We perform our convolutions, which is similar to multiplying weights by input value. We add a bias to the result of this and pass it through a non-linearity such as ReLU. The result flows into the next layer of our neural network.



RELU(



)

In most architectural diagrams the bias addition and ReLU operation are **not depicted**.

# Question

In our Convolutional Layer we are going to apply 10 filters.

The shape of each filter is 3\*3\*?

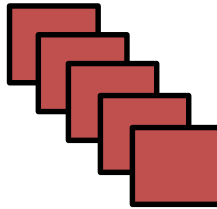
How many learnable parameters are there in this convolutional layer?



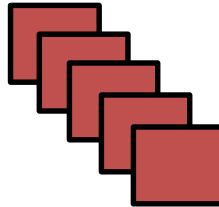
# Question

Each filter must have the same depth as the original feature map (therefore 5 in this case).

10 Filters in Total



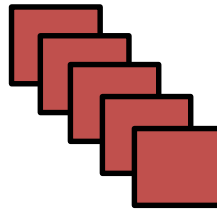
...



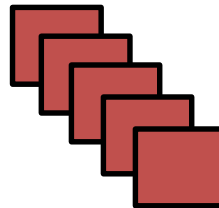
Configuration of  
each filter is  $3 \times 3 \times 5$

# Question

10 Filters in Total



⋮



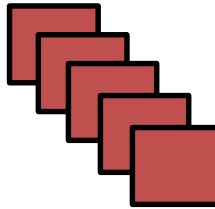
Configuration of  
each filter is  $3 \times 3 \times 5$

- Each filter is  $3 \times 3 \times 5$ .
- Therefore, each filter has 45 learnable parameters.
- Ten filters means we have 450 ( $10 \times 45$ ) learnable parameters in the filters.
- For each filter we must add a bias, which now means we have 460 ( $450 + 10$ ) learnable parameters.

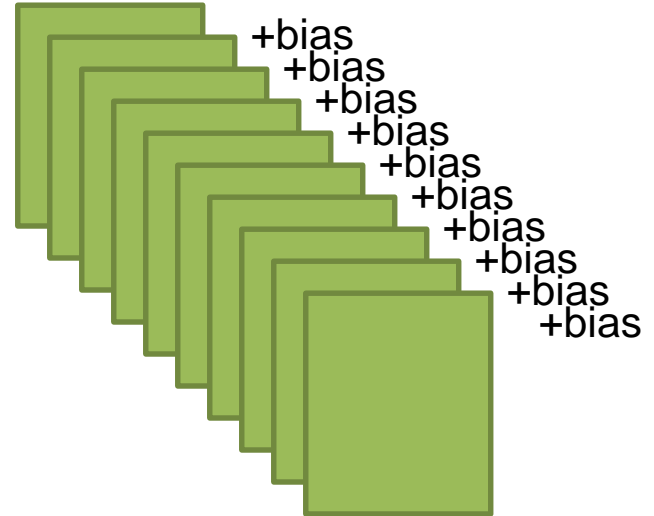
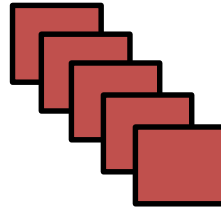
- Each filter is  $3 \times 3 \times 5$ .
- Therefore, each filter has 45 learnable parameters.
- Ten filters means we have 450 ( $10 \times 45$ ) learnable parameters in the filters.
- For each filter we must add a bias, which now means we have 460 ( $450 + 10$ ) learnable parameters.

# Question

10 Filters in Total



...



Configuration of  
each filter is  $3 \times 3 \times 5$

# Components of a CNN

- ▶ A convolution neural network can consist of some of the following components
  - ▶ Convolutional Layer
  - ▶ Padding
  - ▶ Stride
  - ▶ **Pooling Layer**
  - ▶ 1\*1 Convolutional Layer
  - ▶ Fully Connected Layers

# Pooling Layers (Downsampling)

- ▶ A layer commonly found in CNNs is called a Pooling layer and is typically positioned between convolutional layers.
- ▶ Its primary objective is to reduce the size of the representation, which will in turn speeds up computation.
- ▶ The pooling layer itself has no learnable parameters. However, it has two hyper-parameters:
  - ▶ The first is the **stride length** ( $s$ )
  - ▶ The second is the **pool size** ( $w$ )
- ▶ A common configuration is  $s=2$  and  $w=2$
- ▶ The most common type of pooling is called **max pooling**. If we perform max pooling with  $s=2$  and  $w=2$  we step through each exclusive  $2 \times 2$  grid and take the max number from this grid.
- ▶ Every max would in this case be taking a max over 4 numbers ( $2 \times 2$  region). Therefore, we would end up **discarding 75% of all values**.



Applying Pool Size (w=2, s=2)

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	1	5	3	1	2
1	2	2	6	4	6

5		

Result of Max Pooling Layer

Applying Pool Size (w=2, s=2)

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	1	5	3	1	2
1	2	2	6	4	6

5	5	

Result of Max Pooling Layer

Applying Pool Size (w=2, s=2)

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	1	5	3	1	2
1	2	2	6	4	6

5	5	9

Result of Convolution

Applying Pool Size (w=2, s=2)

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	1	5	3	1	2
1	2	2	6	4	6

5	5	9
6		

Result of Max Pooling Layer

Applying Pool Size (w=2, s=2)

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	1	5	3	1	2
1	2	2	6	4	6

5	5	9
6	8	6
2	6	6

Result of Max Pooling Layer

In this example we apply max pooling with a stride and pool size of 2 ( $s=2$ ,  $w=2$ ). Therefore, we step through our image in the usual way, except rather than performing a convolution we now take the max value from each grid.

The max value from each grid is highlighted in bold. Notice how we reduce our dimensionality from  $8*8*3$  to  $4*4*3$

In this example we apply max pooling with a stride and pool size of 2 ( $s=2$ ,  $w=2$ ). Therefore, we step through our image in the usual way, except rather than performing a convolution we now take the max value from each grid.

The max value from each grid is highlighted in bold. Notice how we reduce our dimensionality from  $8*8*3$  to  $4*4*3$

	2	8	8	3	4	1	0	9	
	1	6	7	3	2	1	3	2	3
1	2	3	0	6	4	8	5	1	1
3	<b>9</b>	1	<b>3</b>	5	<b>7</b>	<b>9</b>	7	9	8
4	3	4	4	0	<b>7</b>	<b>8</b>	1	9	5
3	<b>6</b>	1	<b>8</b>	6	3	4	7	4	7
0	1	3	5	<b>8</b>	4	<b>6</b>	3	5	7
1	<b>1</b>	<b>7</b>	5	3	1	2	4	3	1
5	<b>7</b>	2	<b>2</b>	<b>6</b>	4	<b>1</b>	1	7	
4	2	1	2	5	3	0	0		

A 4x4 grid of numbers is shown, with overlapping colored rectangles. The numbers in the grid are:

9	3	7	9
6	8	7	8
1	7	8	6
7	2	6	1

Overlapping rectangles are shown in blue, green, and red. The red rectangle covers the entire 4x4 grid. The green rectangle covers the first three rows and the first three columns. The blue rectangle covers the first two rows and the last two columns.

In this example we apply max pooling with a stride and pool size of 2 ( $s=2$ ,  $w=2$ ). Therefore, we step through our image in the usual way, except rather than performing a convolution we now take the max value from each grid.

The max value from each grid is highlighted in bold. Notice how we reduce our dimensionality from  $8*8*3$  to  $4*4*3$

	2	8	8	3	4	1	0	9	
1	6	7	3	2	1	3	2	3	
0	5	3	0	2	0	1	1	1	
1	3	9	4	0	7	8	9	8	
5	6	9	8	6	3	4	9	5	
7	1	4	5	8	4	9	4	7	
8	1	5	3	3	1	2	5	7	
9	1	1	1	1	0	6	3	1	
2	4	0	0	0	0	6	7		

6	7	2	3	
6	9	7	9	
8	5	8	9	
9	1	1	7	

# Pooling?

- ▶ The input to your pooling layer is a feature map (more specifically the activations of a feature map).
- ▶ Each layer in the input represents a specific feature. **A high value would indicate the presence of this feature while a low value would indicate the absence of the feature.**
- ▶ For example, in the blue quadrant we have a high value of 9 indicating that it has detected this feature. In the yellow quadrant a low value of 1 indicates the feature is not present in this quadrant.
- ▶ With max pooling once the feature is detected in a particular quadrant then it is retained and sent to the resulting matrix (but in essence we are reducing the resolution).

1	2	3	0	6	4	8	5
3	9	1	3	5	7	9	7
4	3	4	4	0	7	8	1
3	6	1	8	6	3	4	7
0	1	3	5	8	4	6	3
1	1	7	5	3	1	2	4
5	7	2	2	6	4	1	1
4	2	1	2	5	3	0	0

- ▶ It should also be noted that there is currently a debate about the merits of pooling. Some papers don't include pooling and instead opt for increasing the stride size within a convolutional layer.
- ▶ The following paper [Striving for Simplicity: The All Convolutional Net](#) argues that the max-pooling layer can be replaced by a convolutional layer with increased stride without loss in accuracy on several image recognition benchmarks.



# Components of a CNN

▶ The following are elements commonly found in convolutional neural networks.

▶ Convolutional Layer

▶ Padding

▶ Stride

▶ Pooling Layer

▶ **1\*1 Convolutional Layer**

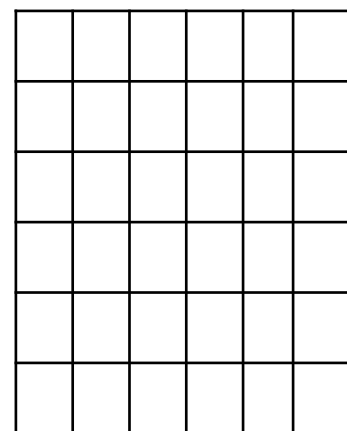
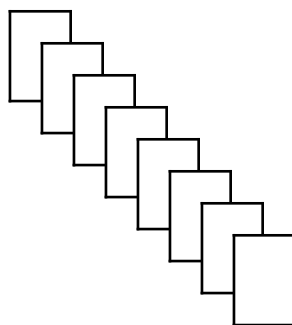
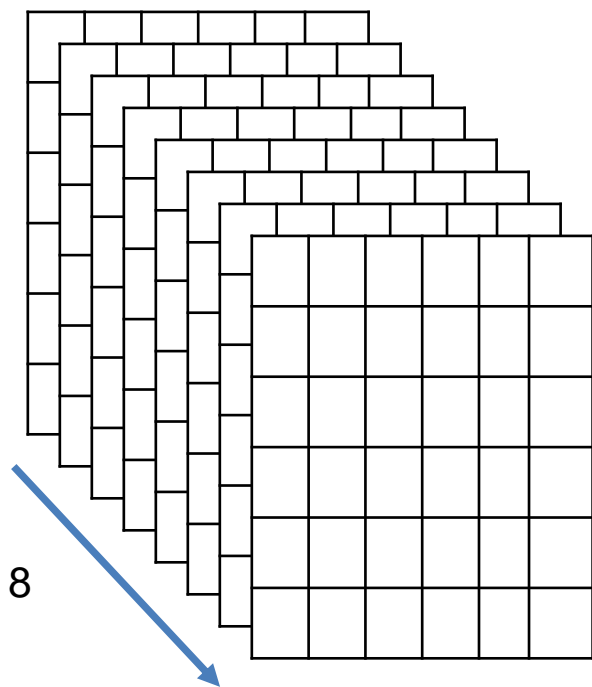
▶ Fully Connected Layers

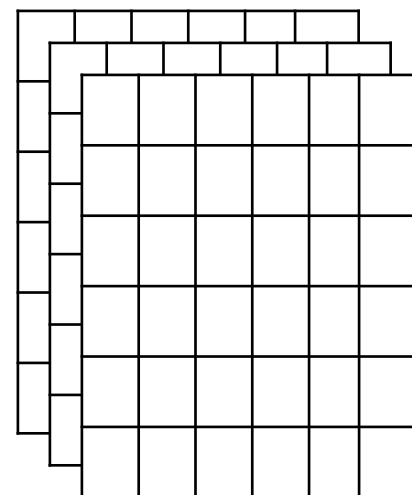
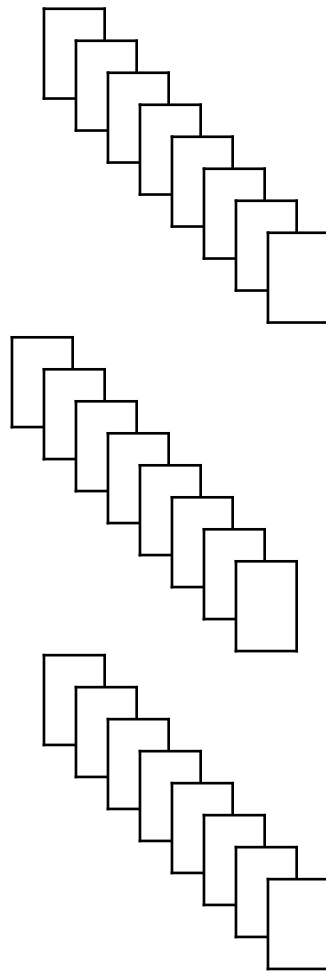
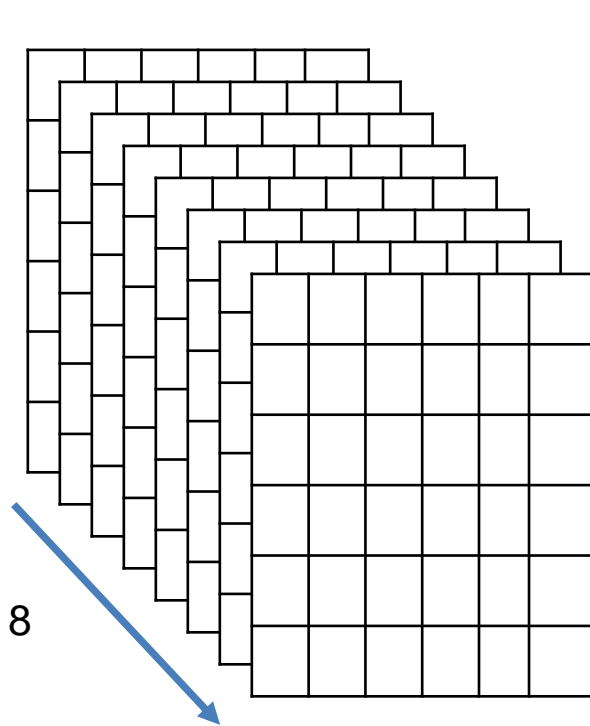
	1	3	7	5	1	2
	3	4	4	0	7	6
5	1	3	5	7	9	2
3	4	4	0	7	8	8
6	1	8	6	3	4	6
1	3	5	8	4	6	3
1	7	5	3	1	2	
7	2	2	6	4	6	

Original Image  
(Feature Map)

### Filter (Kernel)







# Components of a CNN

▶ The following are elements commonly found in convolutional neural networks.

▶ Convolutional Layer

▶ Padding

▶ Stride

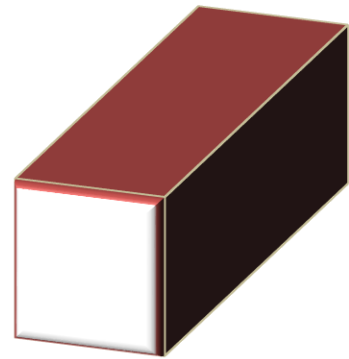
▶ Pooling Layer

▶ 1\*1 Convolutional Layer

▶ **Fully Connected Layers**

# Fully Connected Layer for CNNs

- ▶ It is very common to include one or more fully-connected layers as the **final layers of a CNN**.
- ▶ In the example on the next slide we have already performed a number of convolutional and pooling layers and have produced the following output (a matrix with dimensions  $10*10*50$ ).
- ▶ To include a fully connected layer we **flatten this array into a single linear array that contains 500 values** ( $10*10*5$ ), this would then be fed into a fully connected layer (which may in turn be connected to another fully connected layer).
- ▶ In classification problems the final layer of neurons would typically be fed directly to a Softmax layer.



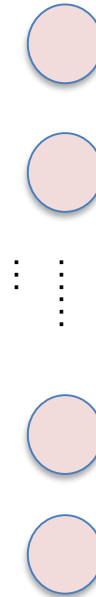
$10 * 10 * 50$

Flat array with 5000 values

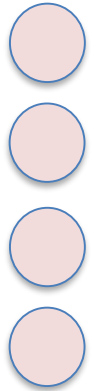
Fully  
Connected  
Layer with  
128 neurons



Fully  
Connected  
Layer with 64  
neurons



SoftMax  
Layer





# Components of a CNN

- ▶ Now that we have examined the composite building blocks of a CNN we will now look at implementing convolutional neural networks using Keras.
- ▶ In the first example, we will apply a simple CNN to the **CIFAR 10** dataset.
- ▶ We will also look at some of the standard architectures for convolutional neural networks.

# CNNs in Keras

The following are the primary components of a CNN in Keras:

1. **Conv2D**. The following are the main parameters that need to be provided:

- **filters (f)**: The number of filters in the convolution.
- **kernel\_size (k)**: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides (s)**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions.
- **padding (p = valid or p= same)**: one of "valid" or "same" (case-insensitive).
- **activation**: Activation function to use
- As with dense layers we can also specify how to **initialize** the kernels and the bias as well as apply **regularization** to both.

# CNNs in Keras

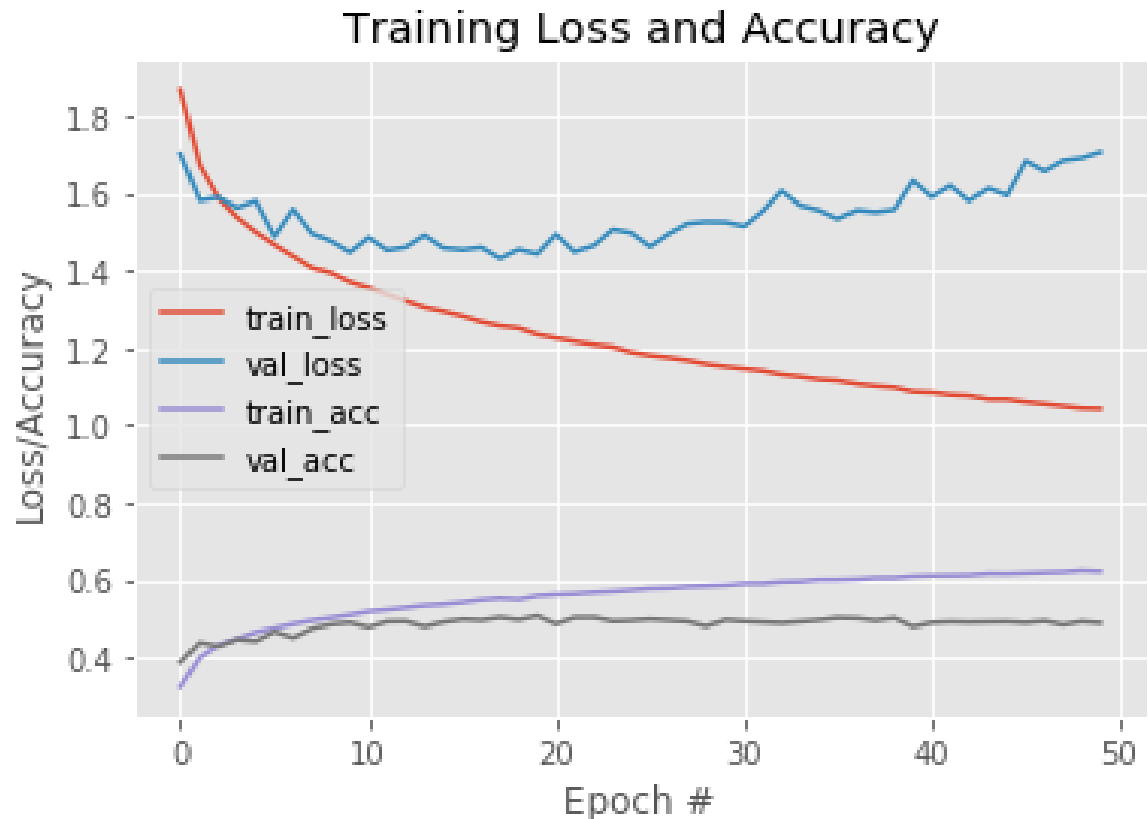
The following are the primary components of a CNN in Keras:

2. **MaxPooling2D**. The following are the main parameters that need to be provided:
  - **pool\_size (k)**: integer or tuple of 2 integers (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
  - **strides (s)**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to pool\_size.

# CIFAR 10 Dataset

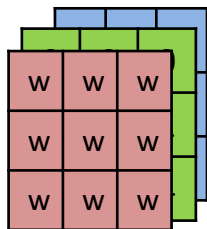
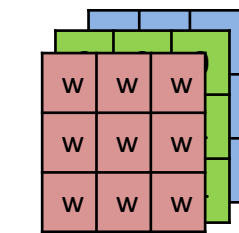
1. You will remember that CIFAR-10, a collection of 60,000,  $32 \times 32$  RGB images and is a 10 class classification problem.
2. We previously built a neural network for this problem that obtained an **accuracy of 0.48**.
3. Over the next few slides we will put together a simple convolutional network for this problem. It will consist of just a single convolutional layer connected to a softmax layer.

Please note you can find a copy of the code on the next few slides on Colab [here](#).

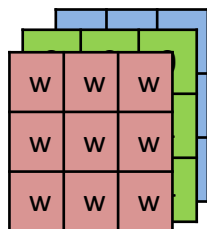




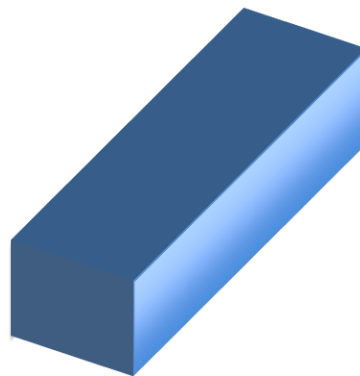
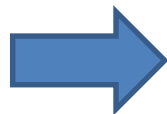
$32 \times 32 \times 3$



...



64 filters



$32 \times 32 \times 64$



Flat array with 65,536 values



Softmax layer  
with 10 neurons

1. Here we create a class called `ShallowNet` class that contains our architecture for our basic CNN.
2. Notice the input shape we have specified is 3D (as we have 3 layers in our image, one for each colour channel). TensorFlow uses what is called a channel last configuration with images.
3. In `ShallowNet` we use 64 filters (kernels), the size of each filter is  $3 \times 3$ . We haven't specified a stride so it defaults to 1. We use 'same' padding.
4. The 3D output from the Convolutional layer is then flattened and connected to a softmax layer.

```
import tensorflow as tf

class ShallowNet:

    @staticmethod
    def build(width, height, depth, classes):

        # initialize the model along with the input shape to be
        # "channels last"
        model = tf.keras.Sequential()
        inputShape = (height, width, depth)

        # define the first (and only) CONV => RELU layer
        model.add(tf.keras.layers.Conv2D(64, (3, 3), padding="same",
            input_shape=inputShape, activation='relu'))

        # softmax classifier
        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(classes, activation='softmax'))

        return model
```

1. Notice this code is similar to what we have seen before. In this case I create an instance of my shallowNet.
2. Next I call `model.summary()`, which prints out the overall structure of the network.

```
NUM_EPOCHS = 60

# load the training and testing data, then scale it into the
# range [0, 1]
((trainX, trainY), (testX, testY)) = tf.keras.datasets.cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0

# initialize the optimizer and model
print("Compiling model...")

opt = tf.keras.optimizers.SGD(lr=0.01)
model = ShallowNet.build(width=32, height=32, depth=3, classes=10)
print ( model.summary() )

model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
              metrics=["accuracy"])

# train the network
print("Training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
              batch_size=32, epochs=NUM_EPOCHS)
```

1. Notice this code is similar to what we have seen before. In this case I create an instance of my shallowNet.

```
NUM_EPOCHS = 60
```

```
# load the training and testing data, then scale it into the  
# range [0, 1]  
((trainX, trainY), (testX, testY)) = tf.keras.datasets.cifar10.load_data()  
trainX = trainX.astype("float") / 255.0
```

2. Next I call

```
m = Model: "sequential"
```

pr  
st

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 32, 32, 64)	1792
-----		
flatten (Flatten)	(None, 65536)	0
-----		
dense (Dense)	(None, 10)	655370
=====		
Total params: 657,162		
Trainable params: 657,162		
Non-trainable params: 0		

```
batch_size=32, epochs=NUM_EPOCHS)
```

pt,



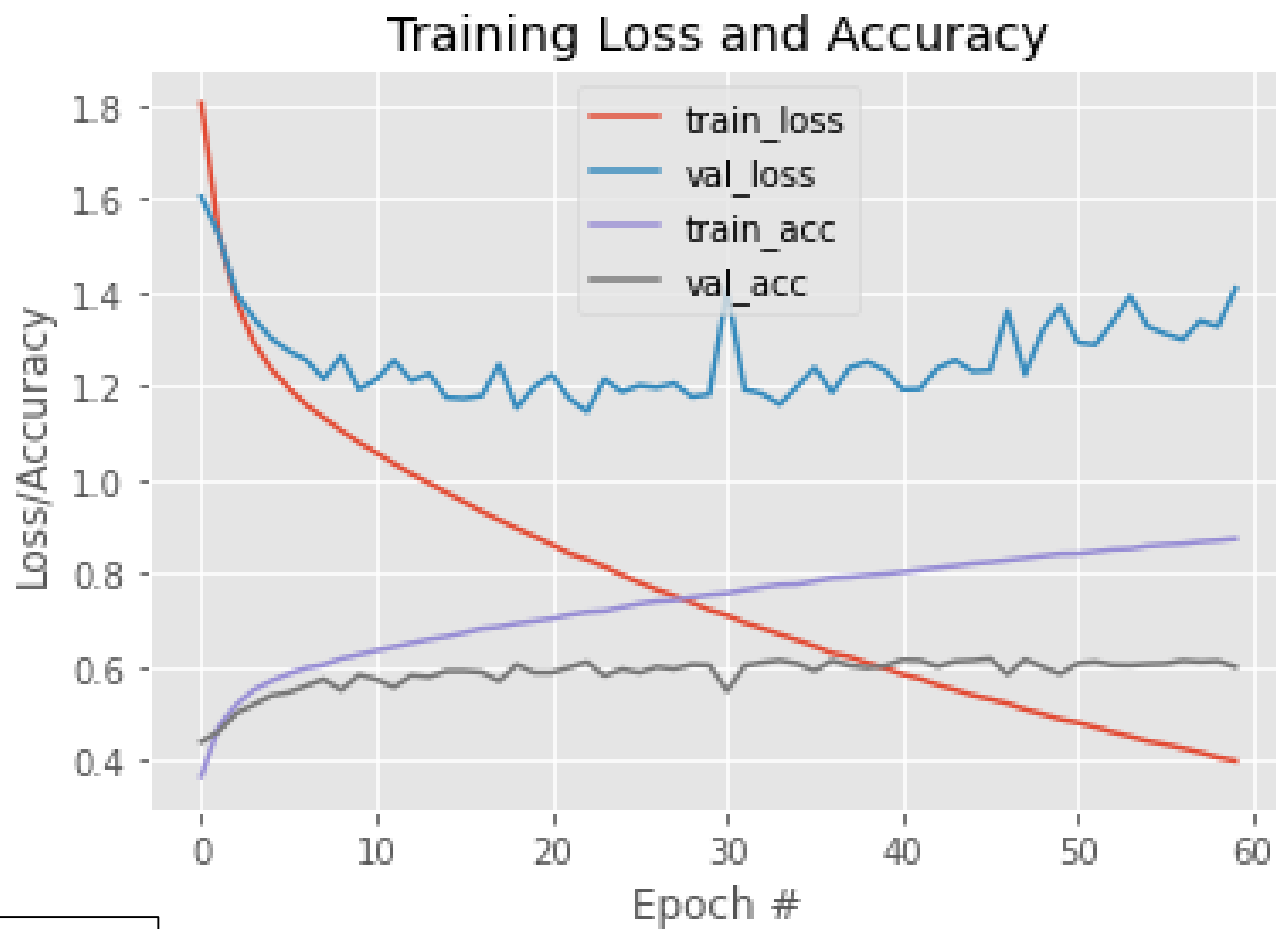
1. Notice this code is similar to what we have seen before. In this case I create an instance of my shallowNet.

2. Next I call

```
Model: "s
Layer (ty
=====
conv2d (C
flatten (
dense (De
=====
Total par
Trainable
Non-train
```

NUM\_EPOCHS = 60

# load the training and testing data, then scale it into the



Despite being a very simple CNN model it has still managed to push up our accuracy by over 10%.

# Existing CNN Architectures

▶ Over the next few slides we will look at common CNN architectures.

▶ LeNet 5

▶ AlexNet

▶ VGG16

▶ Inception

▶ In the next few slides we will adhere to the following notation:

▶  $s$  – stride length

▶  $k$  – filter size

▶  $p$  – padding (if padding is used then we include  $p$ )

▶  $n$  – size of original matrix (for simplicity we assume  $n * n$ )

▶  $f$  – number of filters

▶  $w$  – pool size

# LeNet 5 Architecture

- ▶ The LeNet architecture was proposed by Yann Le Cun et al. and is a **basic example** of a convolutional neural network.
- ▶ The associated paper was called [Gradient Based Learning Applied to Document Recognition](#)
- ▶ This is not a particularly deep CNN but is a good starting point for looking at a CNN architecture. The following table summarized the structure of LeNet.
- ▶ When this model was first produced **padding was not used** so you will notice that the size will notice below that the height and width continue to decrease.

Layer Type	Output Size	Configuration
Input	32*32*1	
Conv	28*28*6	f = 6, k =5, s = 1
Pooling	14*14*6	w =2 , s=2
Conv	10*10*16	f = 16, k =5, s = 1
Pooling	5*5*16	w =2 , s=2
Fully Connected	120*1	
Fully Connected	84*1	
Softmax		

s – stride length

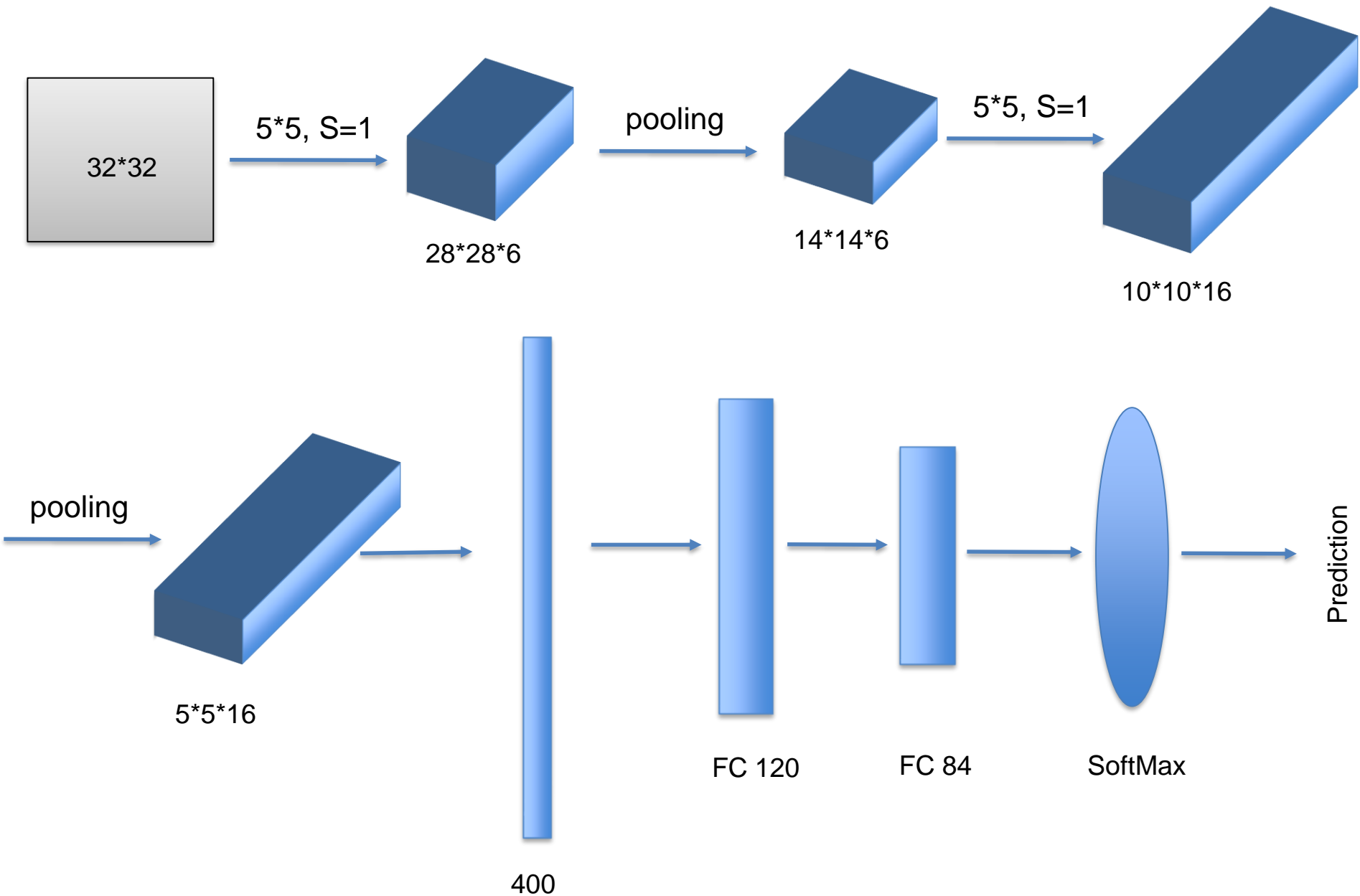
k – filter size

p – padding

f – number of filters

w – pool size

# LeNet 5 Architecture



# LeNet 5 Architecture

- One pattern that you will notice in the LeNet architecture is that we tend to have **convolutional layers** followed by **pooling layers**. This is a very common configuration in many CNNs.
- While the pooling performed in LeNet was **average pooling** (it was more widely used at that time) a modern variant would use **max pooling**.
- Likewise the original architecture had an output layer consisting of 10 neurons. However, modern variants just use a **Softmax** layer.
- Another issue is that the original architecture used **Sigmoid** and **Tanh** activation functions but modern variants use **ReLU**.
- Over the next few slides we will look at implementing a variant of **LeNET for the MNist** dataset. We were previously able to get 0.97 accuracy on the MNIST test set. Lets see if we can improve on that using a variant of the LeNET architecture.

This code is similar to what we have seen before for the ShallowNet example.

We start off with a convolutional layer (with 20 filters) followed by a pooling layer.

We then have another conv layer (containing 50 filters) followed by a pooling layer.

We then flatten the resulting volume and feed the result into a fully connected layers of 500 neurons.

Finally we feed the result of this into a softmax function.

```
import tensorflow as tf

class LeNet:

    @staticmethod
    def build(width, height, depth, classes):

        # initialize the model
        model = tf.keras.models.Sequential()
        inputShape = (height, width, depth)

        # first set of CONV => RELU => POOL layers
        model.add(tf.keras.layers.Conv2D(20, (5, 5), padding="same",
                                           input_shape=inputShape, activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        # second set of CONV => RELU => POOL layers
        model.add(tf.keras.layers.Conv2D(50, (5, 5), padding="same",
                                           activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        # first (and only) set of FC => RELU layers
        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(500, activation='relu'))

        # softmax classifier
        model.add(tf.keras.layers.Dense(classes, activation='softmax'))

        return model
```

This code is similar to what we have seen before for the ShallowN

We start convolut filters) fo layer.

We then layer (co followed

We then volume a into a ful 500 neurons.

Finally we feed the result of this into a softmax function.

```
import tensorflow as tf

class LeNet:
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 28, 28, 20)	520
max_pooling2d_2 (MaxPooling2	(None, 14, 14, 20)	0
conv2d_3 (Conv2D)	(None, 14, 14, 50)	25050
max_pooling2d_3 (MaxPooling2	(None, 7, 7, 50)	0
flatten_1 (Flatten)	(None, 2450)	0
dense_2 (Dense)	(None, 500)	1225500
dense_3 (Dense)	(None, 10)	5010
=====		
Total params: 1,256,080		
Trainable params: 1,256,080		
Non-trainable params: 0		


```
# softmax classifier
model.add(tf.keras.layers.Dense(classes, activation='softmax'))

return model
```

When we initially load the MNIST dataset its shape is (60000, 28, 28).

The **Conv2D class** expects the incoming data structure to have a **row \* column \* depth** structure.

Therefore, we must reshape the loaded dataset so that it now has **4 dimensions**.



We build our model in the usual way and then train for 100 epochs.

The resulting model obtains 99% on the validation dataset.

```
from conv.lenet import LeNet
from tensorflow.python.keras.optimizers import SGD
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

NUM_EPOCHS = 100

#load, and reshape data
mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1) # Returns np.array
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1) # Returns np.array

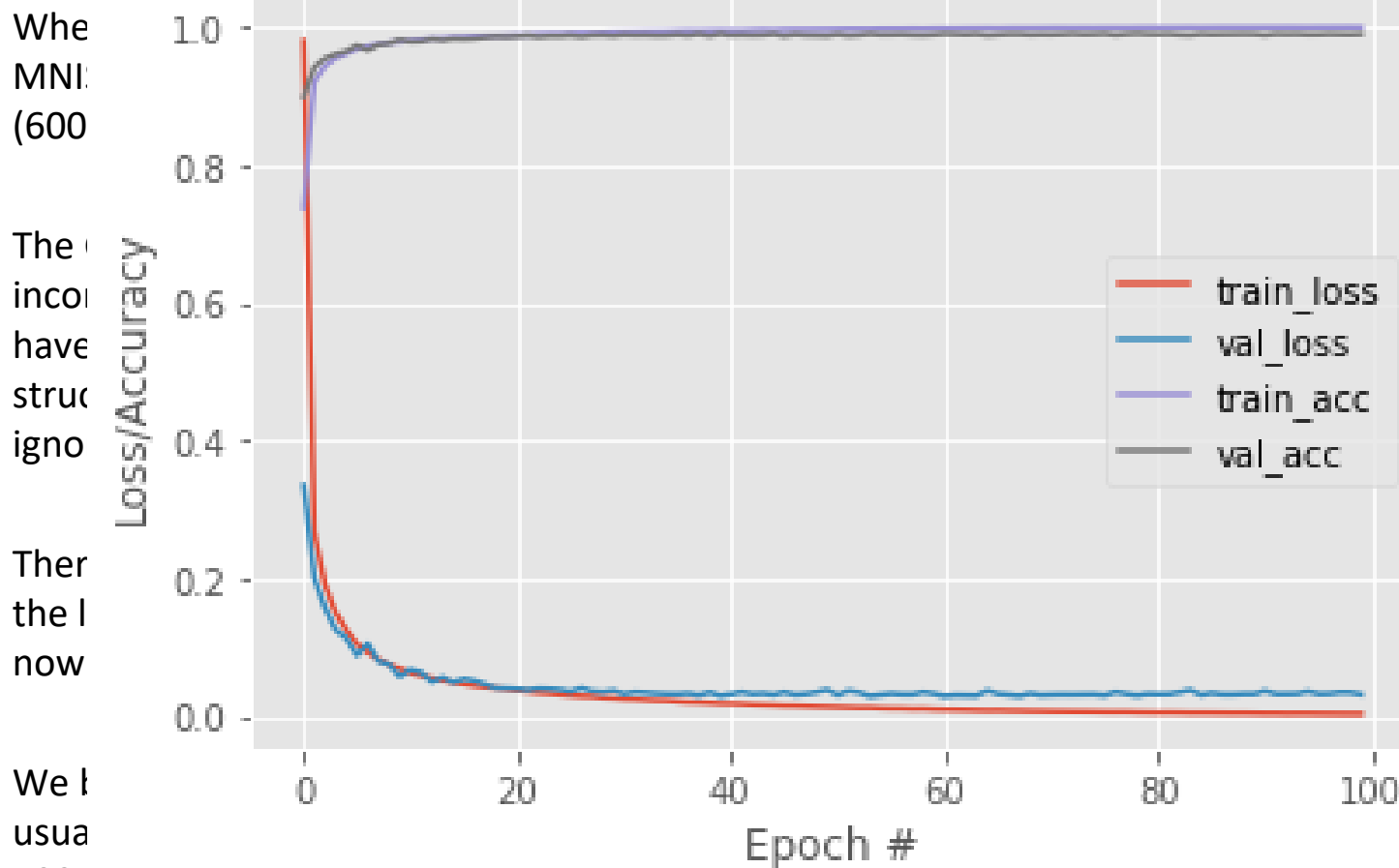
#normalize the data
x_train, x_test = x_train / 255.0, x_test / 255.0

# initialize the optimizer and model
print("Compiling model...")
opt = SGD(lr=0.01)
model = LeNet.build(width=28, height=28, depth=1, classes=10)
model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
               metrics=["accuracy"])

# train the network
print("Training network...")
H = model.fit(x_train, y_train, validation_data=(x_test, y_test),
              batch_size=128, epochs=NUM_EPOCHS, verbose=1)
```



# Training Loss and Accuracy



# Returns np.array  
Returns np.array

`model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,`

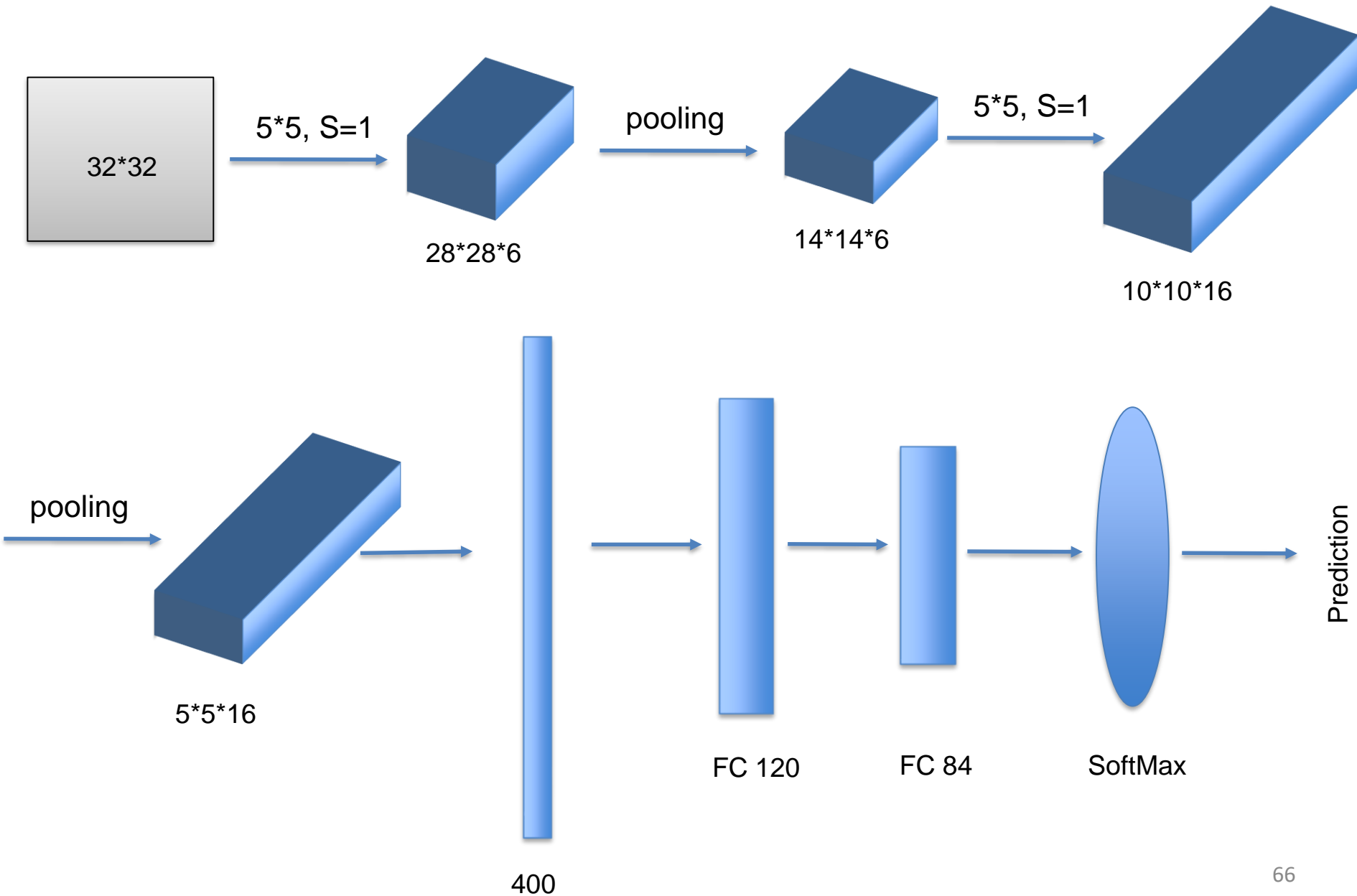
`classes=10)`

Obtain a test accuracy of accuracy of 0.988

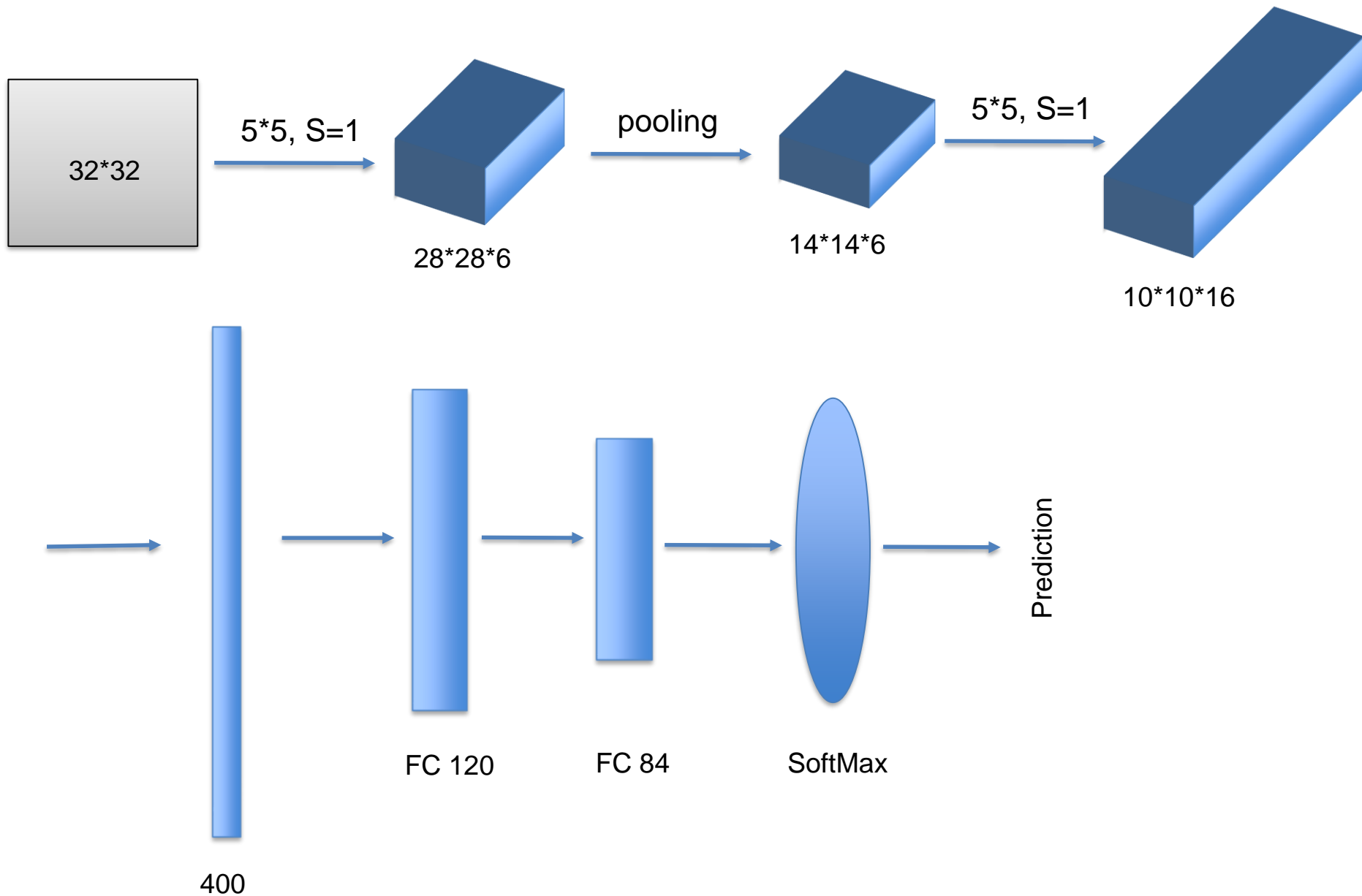
Click [here](#) for full code and results

`(x_test, y_test),  
batch_size=128, epochs=NUM_EPOCHS, verbose=1)`

# Impact of Pooling



# Impact of Pooling



# Learnable Parameters with Pooling

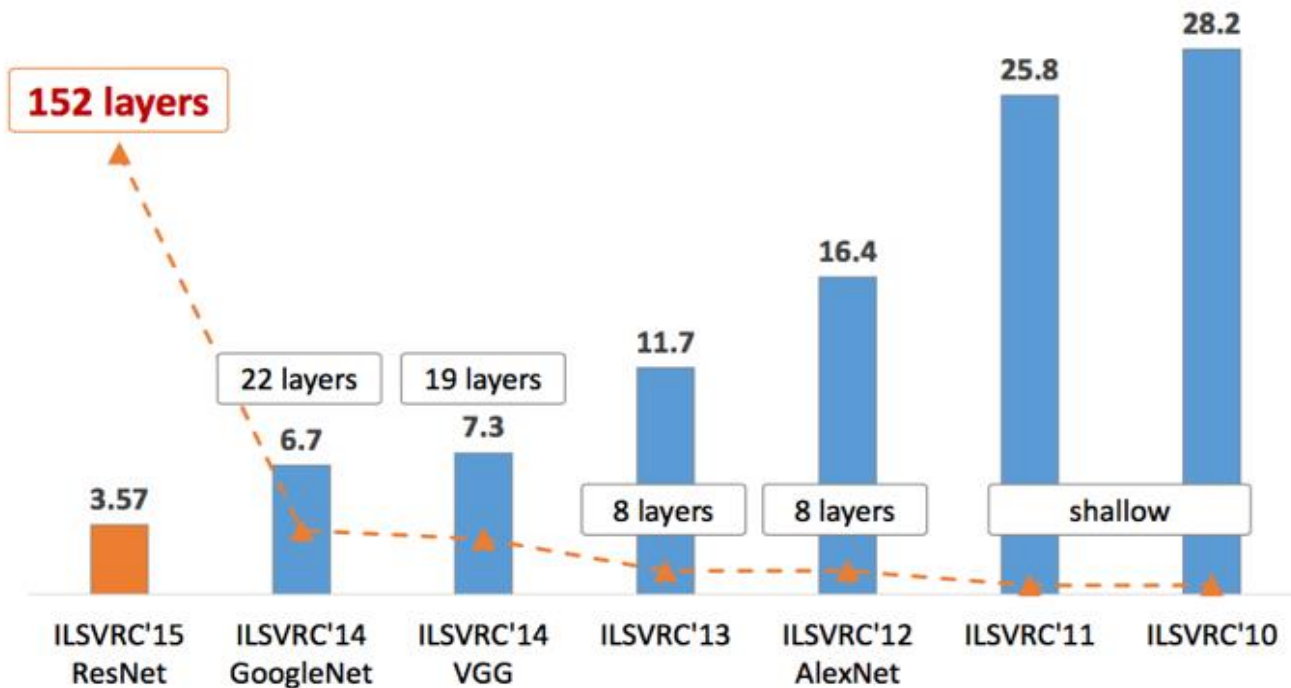
Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 28, 28, 20)	520
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 20)	0
conv2d_3 (Conv2D)	(None, 14, 14, 50)	25050
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 50)	0
flatten_1 (Flatten)	(None, 2450)	0
dense_2 (Dense)	(None, 500)	1225500
dense_3 (Dense)	(None, 10)	5010
Total params: 1,256,080		
Trainable params: 1,256,080		
Non-trainable params: 0		

# Learnable Parameters without Pooling

Layer (type)	Output Shape	Param #
=====		
conv2d_2 (Conv2D)	(None, 28, 28, 20)	520
<hr/>		
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 20)	0
<hr/>		
conv2d_3 (Conv2D)	(None, 14, 14, 50)	25050
<hr/>		
flatten_1 (Flatten)	(None, 9800)	0
<hr/>		
dense_2 (Dense)	(None, 500)	4900500
<hr/>		
dense_3 (Dense)	(None, 10)	5010
=====		
Total params: 4,931,080		
Trainable params: 4,931,080		
Non-trainable params: 0		
<hr/>		

# AlexNet Architecture

- ▶ Since 2010, the ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes (there are **1.2 million training images**, and **100,000 testing images, 1000 classes**).
- ▶ In **2012** a submission called AlexNet achieved a **rank-5 error of 16%**, more than 10.8 percentage points ahead of the runner up. You can find the full paper [here](#), which currently has over 59000 citations.

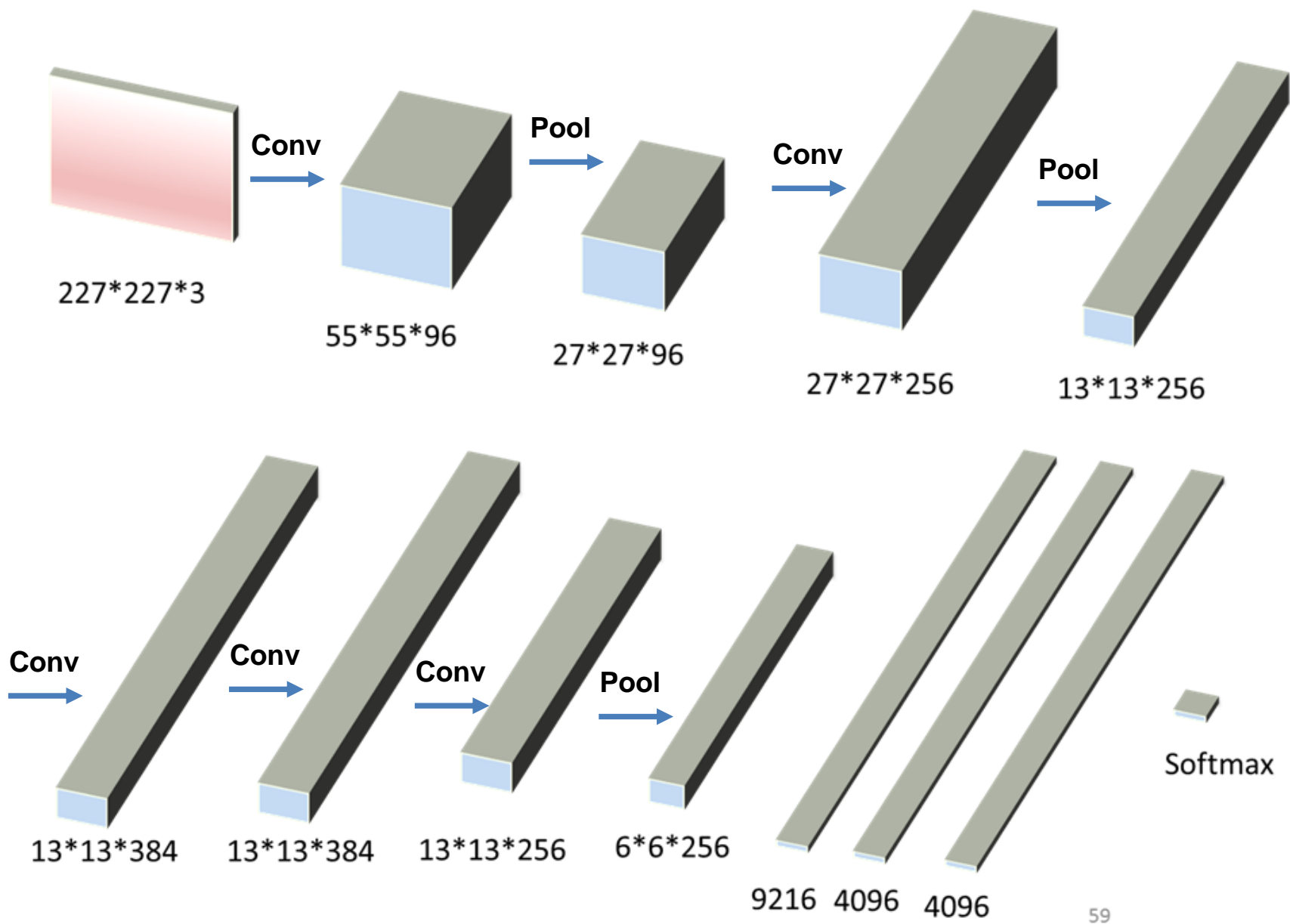


# AlexNet Architecture

▶ The following table summarizes the AlexNet Architecture:

Layer Type	Output Size	Configuration
Input	227*227*3	
Conv	55*55*96	f = 96, k =11, s = 4
Pooling	27*27*96	w =3 , s=2
Conv	27*27*256	f = 256, k =5, s = 1, p
Pooling	13*13*256	w =3 , s=2
Conv	13*13*384	f = 384, k =3, s = 1, p
Conv	13*13*384	f = 384, k =3, s = 1, p
Conv	13*13*256	f = 256, k =3, s = 1, p
Pooling	6*6*384	w =3 , s=2
Fully Connected	9216	
Fully Connected	4096	
Fully Connected	4096	
Softmax		

- s – stride length
- k – filter size
- p – padding
- f – number of filters
- w – pool size





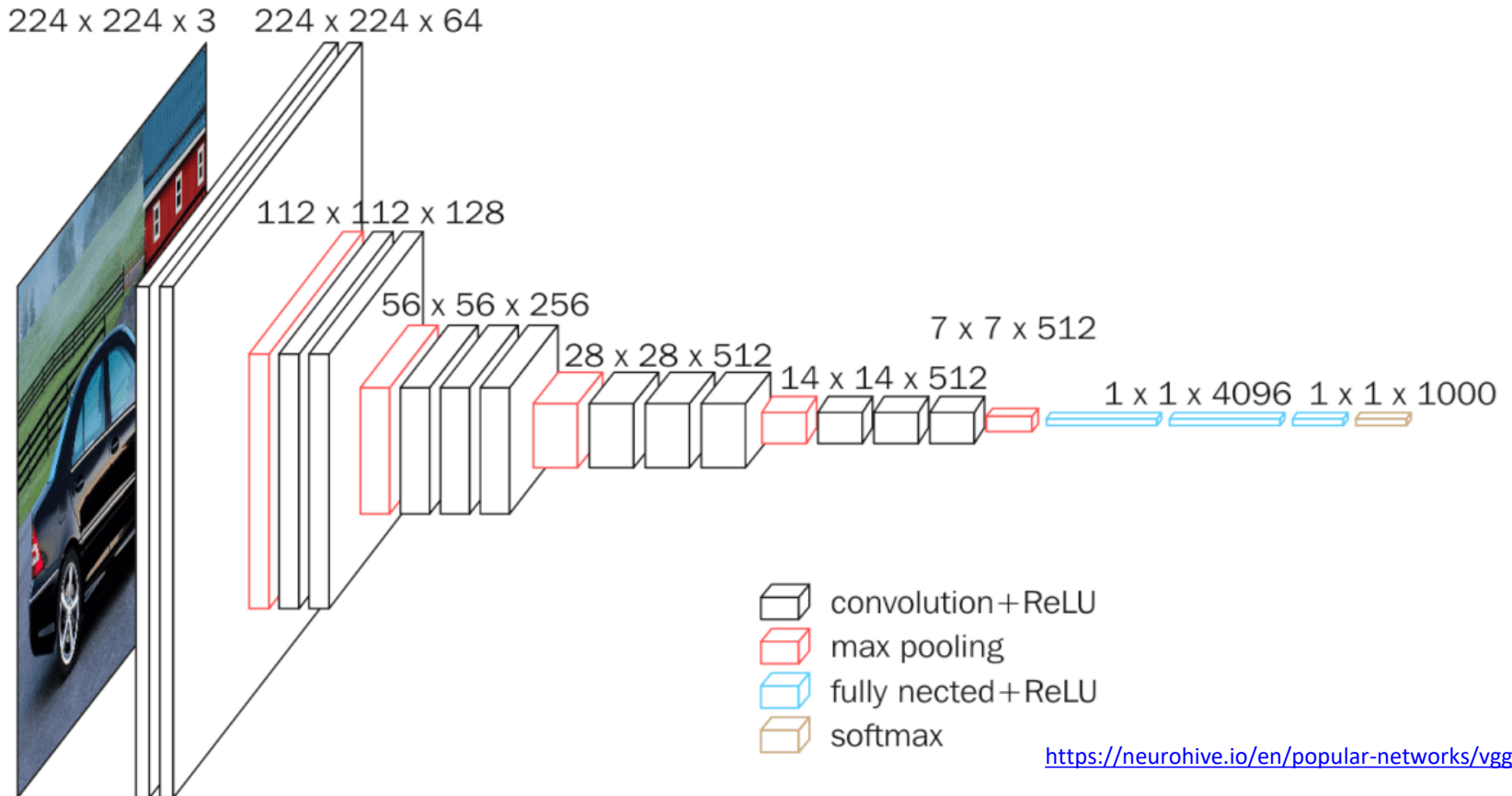
# VGG 16 Architecture

- ▶ In the AlexNet architecture you will have noticed that there is **significant variation in the configuration of each of the network layers**.
- ▶ We might be using:
  - ▶ A large **filter size** of 11 or a small filter size of 3.
  - ▶ Different **number of filters** at different layers (96, 256, 384)
  - ▶ A conv layer following directly by a pooling layer or a conv layer followed by another conv layer.
- ▶ The VGG network is a deep convolution network that is much **more regular** and applies the **same sequence iteratively** to build the network.
- ▶ This network is described in the paper “[Very Deep Convolutional Networks for Large Scale Image Recognition](#)”.

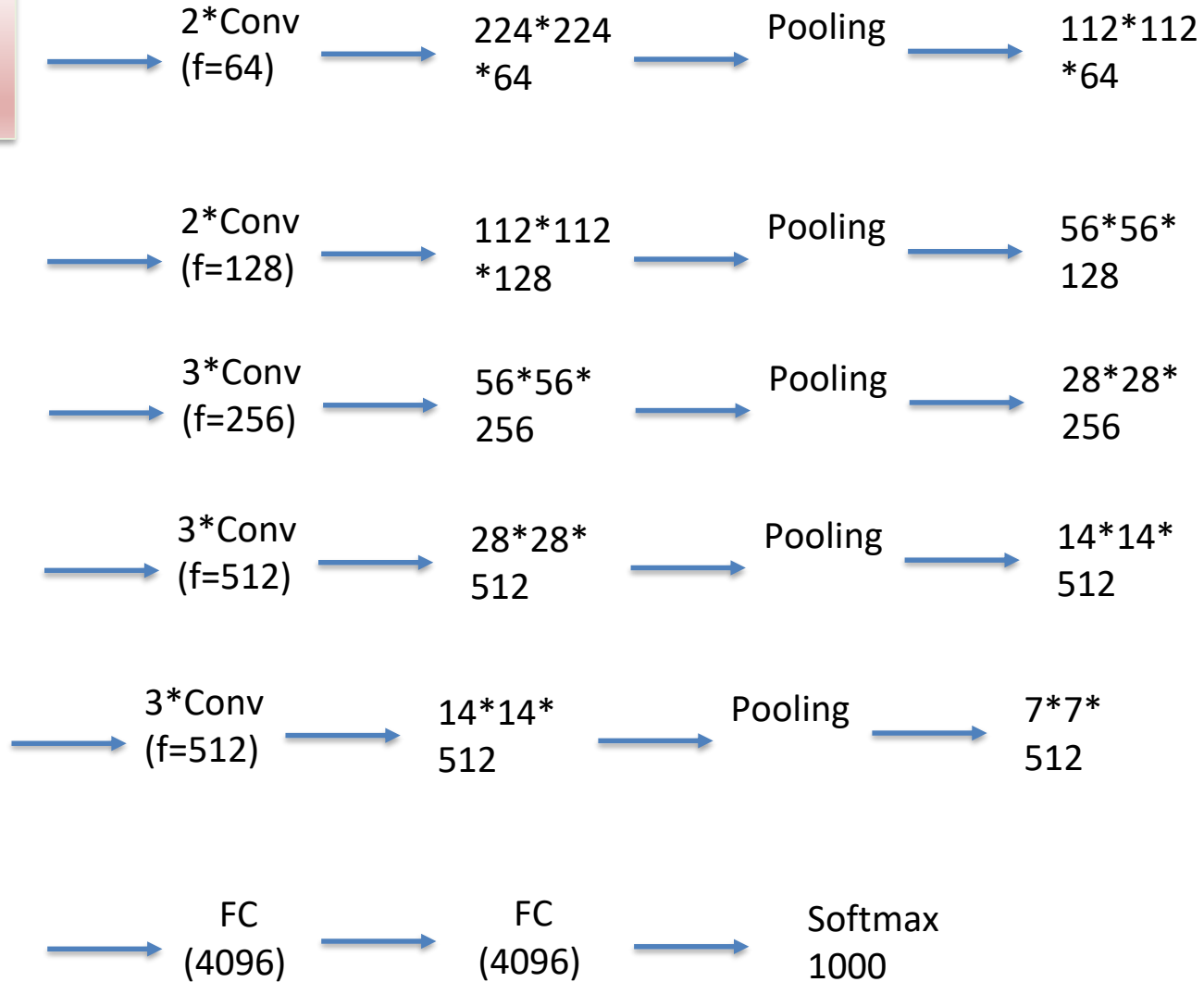
# VGG 16 Architecture

- ▶ For convolution layers VGG16 uses the following  $k=3, s=1, p$
- ▶ For pooling it uses the following  $w=3, s=2$
- ▶ Notice this network structure is quite regular (2\* (2 convolutions followed by a pooling layer) followed by 3\* (3 convolutions followed by a pooling layer)).
- ▶ Also notice the number of filters **doubles** after each “block”

- $s$  – stride length
- $k$  – filter size
- $p$  – padding
- $f$  – number of filters
- $w$  – pool size



224\*224\*3



# VGG 16 Architecture - ShallowVGGNet

1. Over the next few slides we will develop a shallow VGG net and apply to the CIFAR10 dataset.
2. We focus on a shallow version because the full architecture takes a very long time to train from scratch even on a GPU.
3. **ShallowVGGNet** consists of:
  - A) CONV
  - B) CONV
  - C) POOL
  - D) CONV
  - E) CONV
  - F) POOL
  - G) FC
  - H) FC

# Shallow VGG 16 Architecture

- s – stride length
- k – filter size
- p – padding
- f – number of filters
- w – pool size

Layer Type	Output Size	Configuration
Input	32*32*3	
Conv	32*32*32	f = 32, k =3, s = 1,p
Conv	32*32*32	f = 32, k =3, s = 1,p
Pooling	16*16*32	w =2 , s=2
Conv	16*16*64	f = 64, k =3, s = 1,p
Conv	16*16*64	f = 64, k =3, s = 1,p
Pooling	8*8*64	w =2 , s=2
Fully Connected	512	
Softmax	10	

```
import tensorflow as tf
from tensorflow import keras
```

```
class ShallowVGGNet:
```

```
    @staticmethod
```

```
    def build(width, height, depth, classes):
```

```
        model = keras.models.Sequential()
```

```
        inputShape = (height, width, depth)
```

```
        # first CONV => CONV => POOL layer set
```

```
        model.add(keras.layers.Conv2D(32, (3, 3), padding="same",
                                         input_shape=inputShape, activation='relu'))
```

```
        model.add(keras.layers.Conv2D(32, (3, 3), padding="same", activation='relu'))
```

```
        model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
```

```
        # second CONV => CONV => POOL layer set
```

```
        model.add(keras.layers.Conv2D(64, (3, 3), padding="same", activation='relu'))
```

```
        model.add(keras.layers.Conv2D(64, (3, 3), padding="same", activation='relu'))
```

```
        model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
```

```
        # first (and only) set of FC => RELU layers
```

```
        model.add(keras.layers.Flatten())
```

```
        model.add(keras.layers.Dense(512, activation='relu'))
```

```
        # softmax classifier
```

```
        model.add(keras.layers.Dense(classes, activation='softmax'))
```

```
        # return the constructed network architecture
```

```
        return model
```

Notice we have **two conv layers** followed by a **max pooling** layer

This is then followed by another sequence with 2 conv layers and a pooling layer.

We then have a dense fully connected layer that connects to a softmax layer.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
flatten (Flatten)	(None, 4096)	0
dense (Dense)	(None, 512)	2097664
dense_1 (Dense)	(None, 10)	5130

=====  
 Total params: 2,168,362  
 Trainable params: 2,168,362  
 Non-trainable params: 0

**Previously we obtained an accuracy of 61% on CIFAR 10** using a very shallow convolutional neural network.

**The validation accuracy we now obtain is 70%**

You can find the full code for this example [here](#).

Please note that even on a GPU this can take a long time to train.

```
NUM_EPOCHS = 25
((trainX, trainY), (testX, testY)) = keras.datasets.cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0

# initialize the optimizer and model
print("Compiling model...")
opt = keras.optimizers.SGD(lr=0.01)
model = ShallowVGGNet.build(width=32, height=32, depth=3, classes=10)
model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
              metrics=["accuracy"])

print (model.summary())

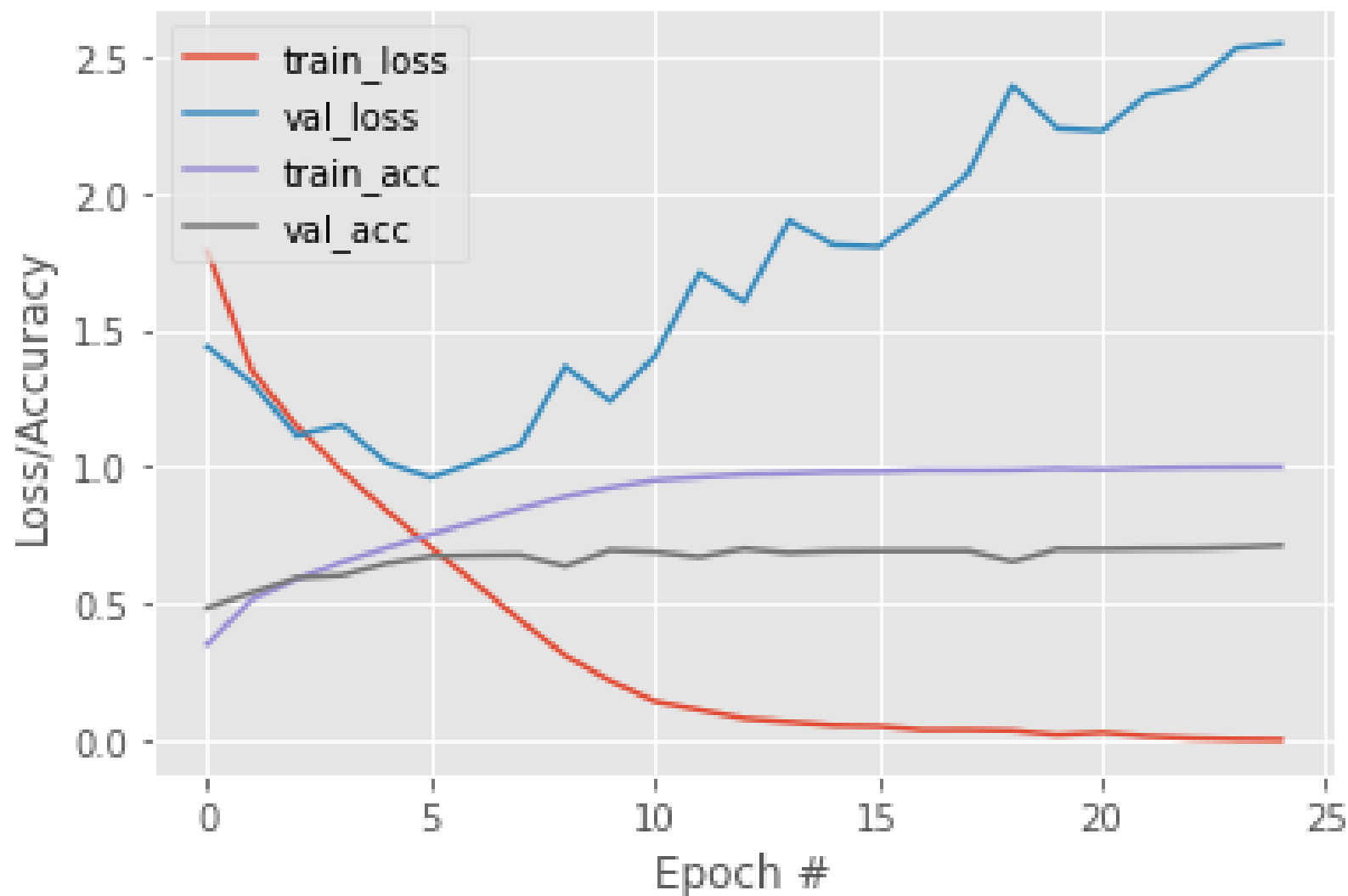
print("Training network...")
H = model.fit(trainX, trainY, batch_size=16,
             epochs=NUM_EPOCHS, validation_split=0.1)

print ("Test Data Loss and Accuracy: ", model.evaluate(testX, testY))

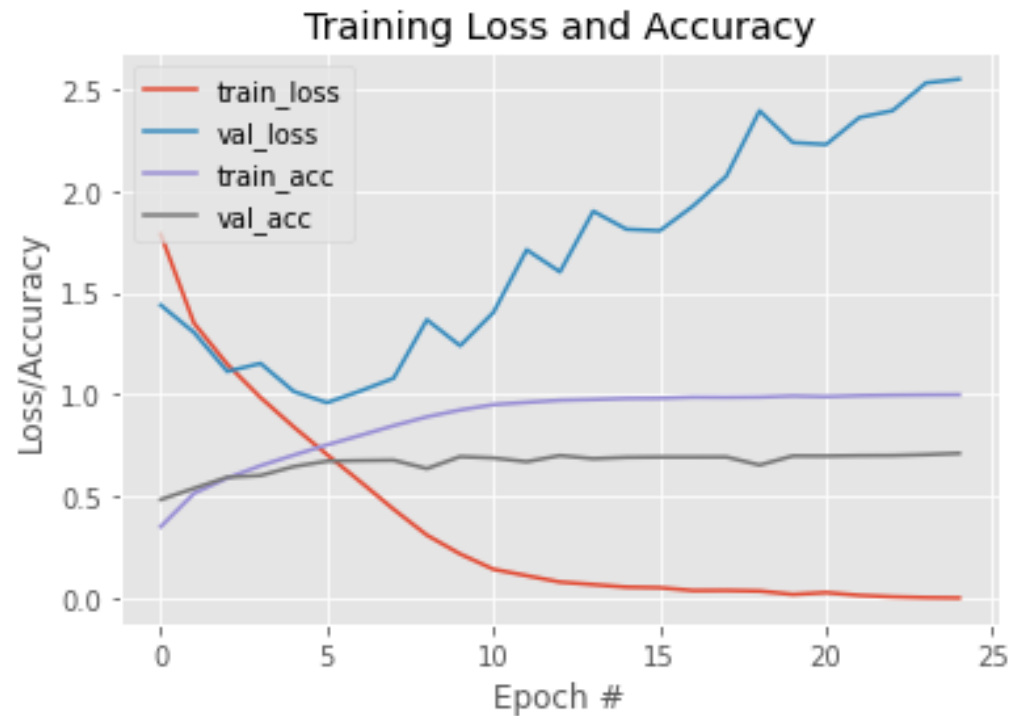
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, NUM_EPOCHS), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, NUM_EPOCHS), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, NUM_EPOCHS), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, NUM_EPOCHS), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```



# Training Loss and Accuracy



- This graph shows a model that is **significantly overfitting** on the training data.
- After 4 or 5 epochs the model begins to overfit. We can see the validation loss going back up and the training loss continue to fall.
- ShallowVGGNet is a deep network and is likely to overfit on the training data.
- Let's apply a **regularization (Dropout)** techniques to help address this issue.



```
import tensorflow as tf
```

```
class ShallowVGGNet:
```

```
    @staticmethod
```

```
    def build(width, height, depth, classes):
```

```
        model = keras.models.Sequential()
```

```
        inputShape = (height, width, depth)
```

```
        # first CONV => RELU => CONV => RELU => POOL => Dropout Layer
```

```
        model.add(keras.layers.Conv2D(32, (3, 3), padding="same",  
                                         input_shape=inputShape, activation='relu'))
```

```
        model.add(keras.layers.Conv2D(32, (3, 3), padding="same", activation='relu'))
```

```
        model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
```

```
        model.add(keras.layers.SpatialDropout2D(rate=0.25))
```

```
        # second CONV => RELU => CONV => RELU => POOL => Dropout Layer
```

```
        model.add(keras.layers.Conv2D(64, (3, 3), padding="same", activation='relu'))
```

```
        model.add(keras.layers.Conv2D(64, (3, 3), padding="same", activation='relu'))
```

```
        model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
```

```
        model.add(keras.layers.SpatialDropout2D(rate=0.25))
```

```
        # first (and only) set of FC => RELU layers
```

```
        model.add(keras.layers.Flatten())
```

```
        model.add(keras.layers.Dense(512, activation='relu'))
```

```
        model.add(keras.layers.Dropout(rate=0.5))
```

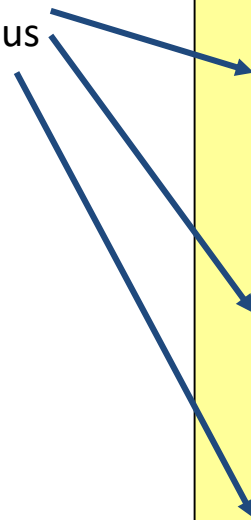
```
        # softmax classifier
```

```
        model.add(keras.layers.Dense(classes, activation='softmax'))
```

```
        # return the constructed network architecture
```

```
        return model
```

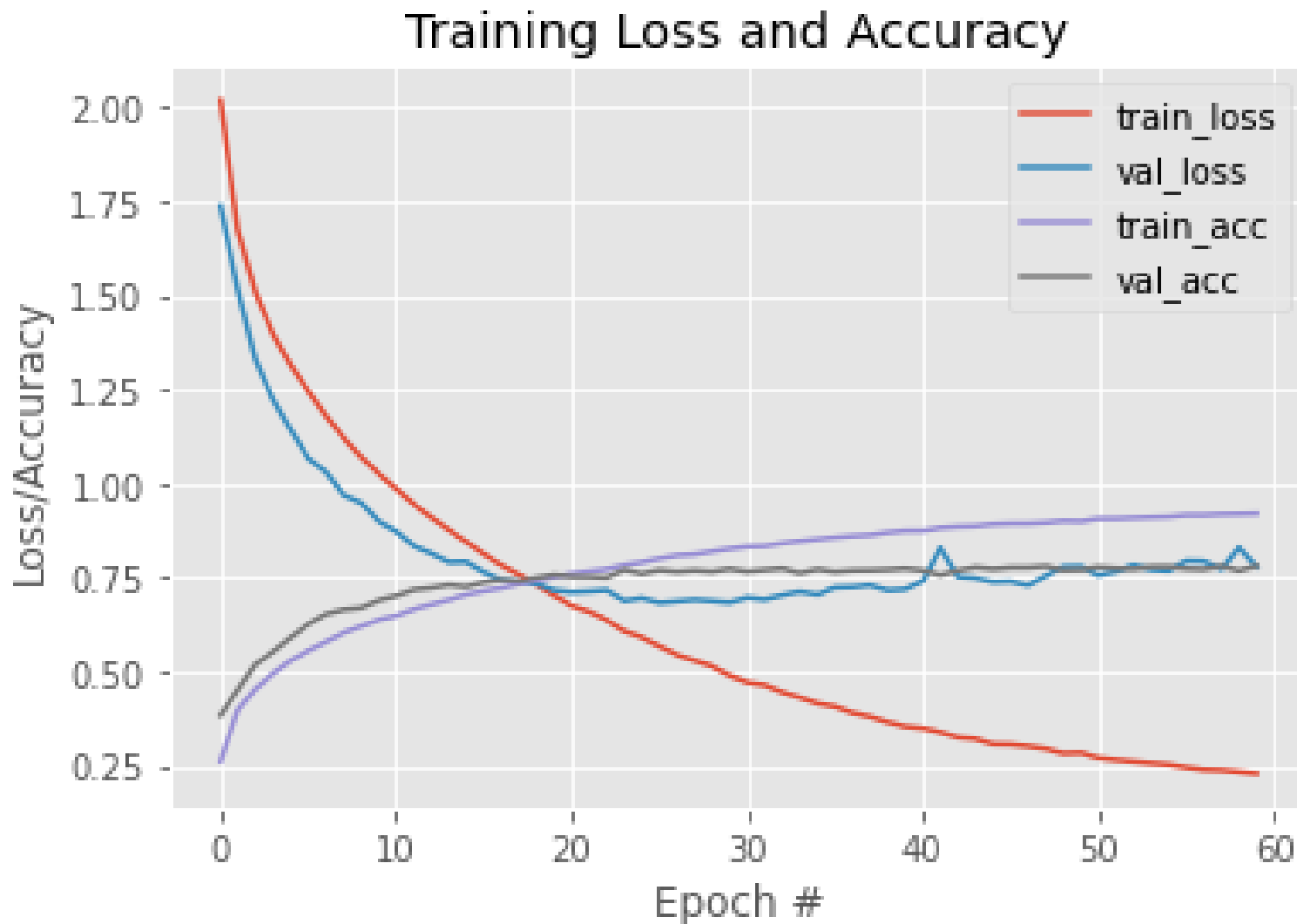
You will notice the difference between this implementation and the last is that we introduce Dropout as a means of controlling the overfitting issue which was very evident in the previous slide.



Notice the implementation of Dropout significantly reduces the impact of overfitting and allows us to increase our validation accuracy still further on this problem.

In fact we were able to obtain an accuracy of **0.76**.

The full code for this can be found [here](#).



```

model = keras.models.Sequential()
inputShape = (height, width, depth)

# first CONV => RELU => CONV => RELU => POOL => Dropout Layer
model.add(keras.layers.Conv2D(32, (3, 3), padding="same", input_shape=inputShape, activation='relu'))
model.add(keras.layers.Conv2D(32, (3, 3), padding="same", activation='relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.SpatialDropout2D(rate=0.25))

# second CONV => RELU => CONV => RELU => POOL => Dropout Layer
model.add(keras.layers.Conv2D(64, (3, 3), padding="same", activation='relu'))
model.add(keras.layers.Conv2D(64, (3, 3), padding="same", activation='relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.SpatialDropout2D(rate=0.25))

# third CONV => RELU => CONV => RELU => POOL => Dropout Layer
model.add(keras.layers.Conv2D(128, (3, 3), padding="same", activation='relu'))
model.add(keras.layers.Conv2D(128, (3, 3), padding="same", activation='relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.SpatialDropout2D(rate=0.25))

# first (and only) set of FC => RELU layers
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dropout(rate=0.5))

# softmax classifier
model.add(keras.layers.Dense(classes, activation='softmax'))

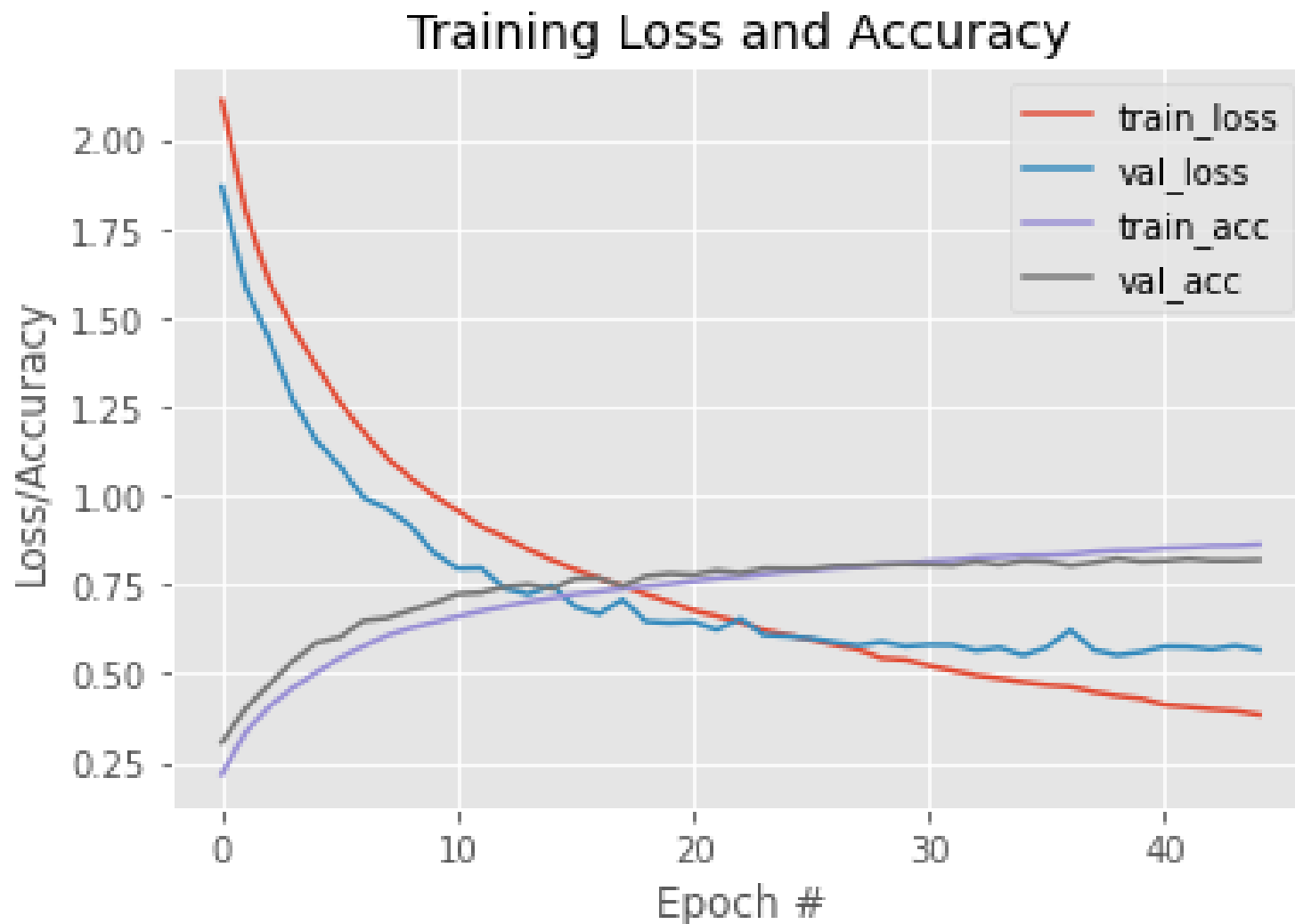
# return the constructed network architecture
return model

```

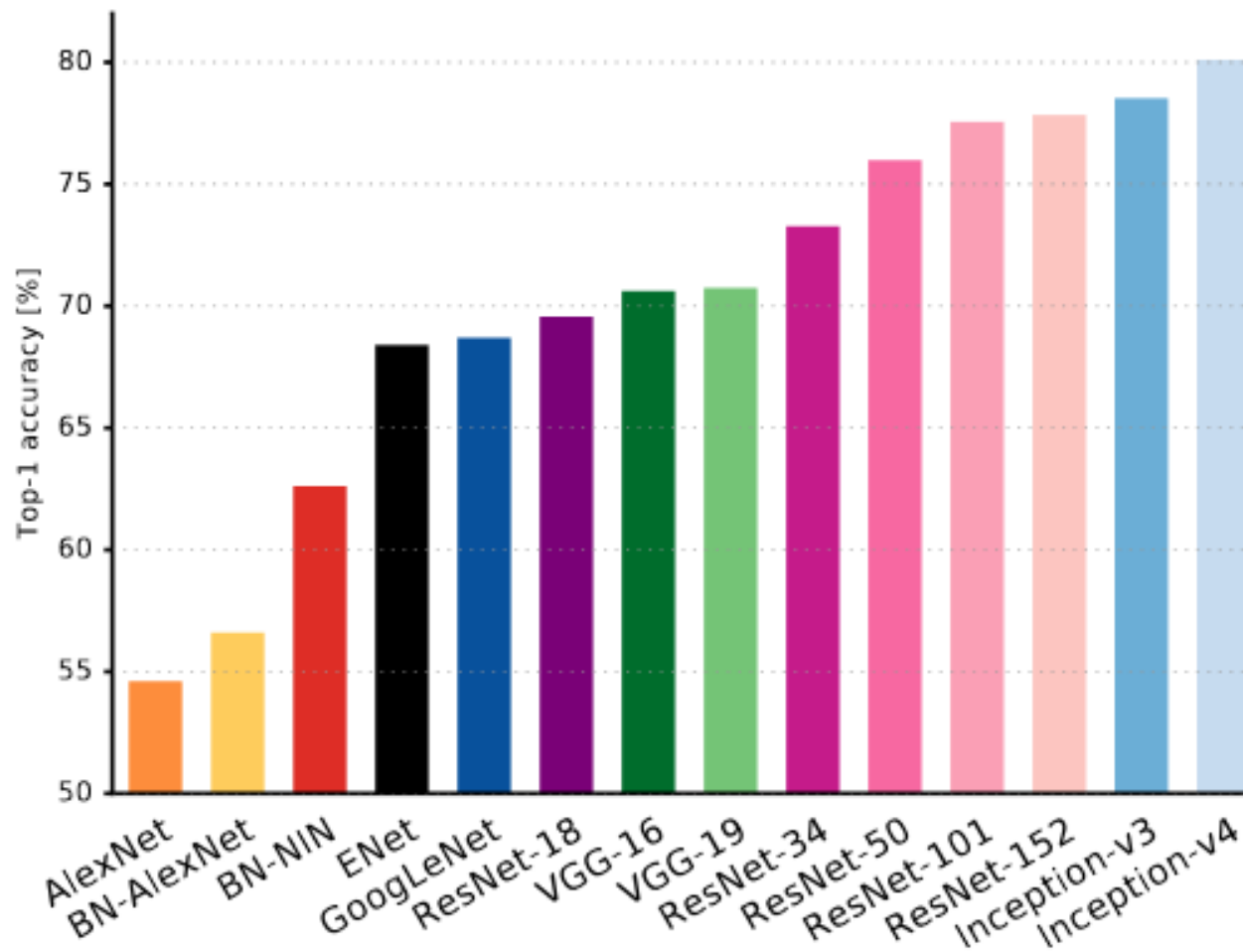
In this code we create a deep version of VGG again. Notice we have added another 2\*Conv2D + Pool + Dropout

In the graph below we now have the deeper VGG network, which provides another bump in accuracy up to **0.80**.

The full code for this can be found [here](#).



The graph below shows the rank 1 accuracy of all the major deep network architecture of ImageNet. This graph and the graphs on the following slides are from the following 2016 paper [An Analysis of Deep Neural Network Models for Practical Applications](#)



This graph represents the **rank 1 accuracy** versus the **number of operations** required for a single forward pass.

The size of the circles is proportional to the **number of network parameters**.

Interestingly we can see that VGG, even though it is widely used in many applications, is by far the most expensive architecture — both in terms of computational requirements and number of parameters (**138M**).

