intel Nervana™ AI Academy

INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS

LAST TIME

Managed to get seemingly good results with basic network

98% Test Accuracy on MNIST:

- ReLU
- 3 hidden layers of depth 1200
- 15 epochs

98% for a minimal amount of training time seems pretty good!

What are we missing?

CONSIDERATIONS

MNIST has relatively clean images

Numbers are:

- Centered
- Approximately same size

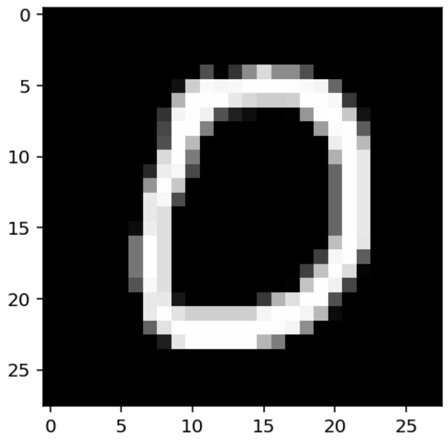
Image only has number in it – background is black

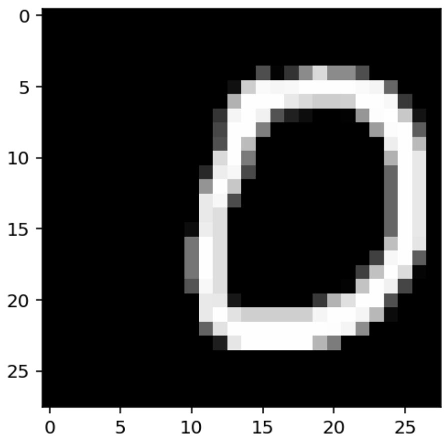
PROBLEM 1: TRANSLATION INVARIANCE

Each pixel is independent input

If we translate the input, the model breaks down

- We need to train (and test) models on translated data for more realistic scenario





PROBLEM 2: HUGE NUMBER OF PARAMETERS

1200x1200 matrix of weights = 1.4 *million* weights

- More weights → need more data
- More weights → hard to scale on hardware
  - Memory constraints!

What can we do?

KERNELS

WHAT ARE KERNELS?

Square grid of weights overlaid on image, centered on one pixel, and moved around the image

Each weight multiplied with pixel underneath it

Output for the centered pixel is $\sum_{p=1}^{P} W_p \cdot pixel_p$

Used for traditional image processing techniques:

- Blur
- Sharpen
- Edge detection
- Emboss

EXAMPLE: 3X3

Input

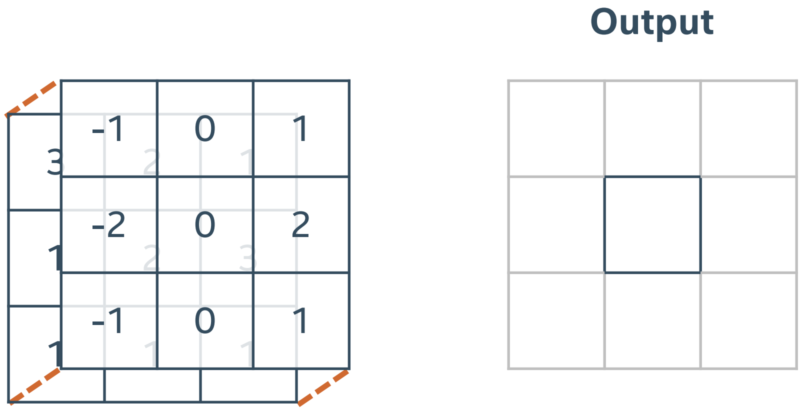
3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

IMAGINE KERNEL IS STACKED ON TOP OF INPUT



EXAMPLE: 3X3

Input

3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

$$= (3 \cdot -1)$$

EXAMPLE: 3X3

Input

3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

$$= (3 \cdot -1) + (2 \cdot 0)$$

EXAMPLE: 3X3

Input

3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

$$= (3 \cdot -1) + (2 \cdot 0) + (1 \cdot 1)$$

EXAMPLE: 3X3

Input

3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

$$= (3 \cdot -1) + (2 \cdot 0) + (1 \cdot 1) + (1 \cdot -2)$$

EXAMPLE: 3X3

Input		
3	2	1
1	2	3
1	1	1

Kernel		
-1	0	1
-2	0	2
-1	0	1

Output		
	2	

$$= (3 \cdot -1) + (2 \cdot 0) + (1 \cdot 1) + (1 \cdot -2) + (2 \cdot 0) + (3 \cdot 2) + (1 \cdot -1) + (1 \cdot 0) + (1 \cdot 1)$$

$$= -3 + 1 - 2 + 6 - 1 + 1$$
$$= 2$$

HERE'S WHAT THE PROCESS LOOKS LIKE OVER A LARGER INPUT

1	2	0	3	1
1	0	0	2	2
2	1	2	1	1
0	0	1	0	0
1	2	1	1	1

Input

-1	1	2
1	1	0
-1	-2	0

Kernel

-2		

Output

INTERACTIVE KERNEL DEMONSTRATION

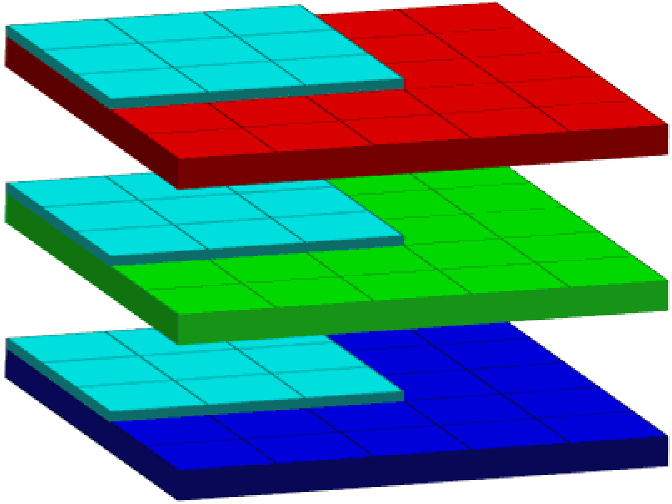
http://setosa.io/ev/image-kernels/

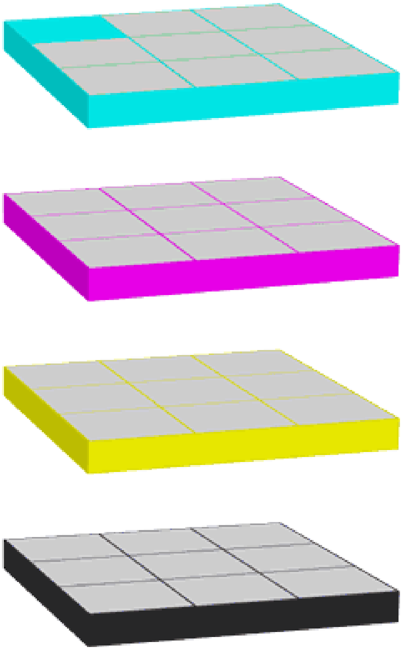
CONVOLUTIONAL NEURAL NETWORKS

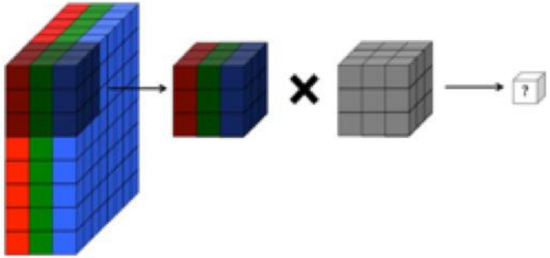
CONVOLUTIONAL NEURAL NETWORKS

Idea: let neural network learn suitable kernels for task

CONVOLUTION OPERATION







CONVOLUTION SETTINGS

HEIGHT AND WIDTH

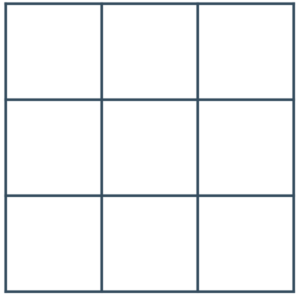
Number of pixels the kernel operates on

Both dimensions must be odd

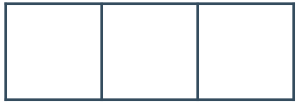
- B/c we need a reasonable center pixel

Kernel doesn't have to be square

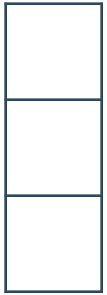
Height: 3, Width: 3



Height: 1, Width: 3



Height: 3, Width: 1



STRIDE

***Stride* is the step size from center to center**

Also has height/width component

- Generally height/width are the same

If greater than 1, will scale down the output dimensions

STRIDE 2 CONVOLUTION

1	2	0	3	1
1	0	0	2	2
2	1	2	1	1
0	0	1	0	0
1	2	1	1	1

Input

-1	1	2
1	1	0
-1	-2	0

Kernel

-2	

Output

PADDING

Notice: the standard convolution down samples input

1	2	0	3	1
1	0	0	2	2
2	1	2	1	1
0	0	1	0	0
1	2	1	1	1

Input
[5x5]

-1	1	2
1	1	0
-1	-2	0

Kernel

-2		

Output
[3x3]

PADDING

Padding adds pseudo-pixels off-the-edge of the input

- Padding is all zero values

One unit of padding means one ring of zero pixels around the input

Amount of padding is usually either:

- No padding
  - TensorFlow calls this 'VALID' (i.e., use only *valid* input size)
- Enough to offset the kernel size and output the same dimensions
  - TensorFlow calls this 'SAME' (i.e., same input/output size)

3x3 kernel → padding 1
5x5 kernel → padding 2
7x7 kernel → padding 3

PADDING: 1 (‘SAME’)

0	0	0	0	0	0	0
0	1	2	0	3	1	0
0	1	0	0	2	2	0
0	2	1	2	1	1	0
0	0	0	1	0	0	0
0	1	2	1	1	1	0
0	0	0	0	0	0	0

Input

-1	1	2
1	1	0
-1	-2	0

Kernel

-1				

Output

DEPTH—NUMBER OF OUTPUT CHANNELS

Channels: multiple numbers (colors) associated with same pixel

- 3-color RGB → 3 channels
- 4-color CMYK → 4 channels

Number of separate kernels needed in a layer

OUTPUT CHANNELS: 2

1	2	0	3	1
1	0	0	2	2
2	1	2	1	1
0	0	1	0	0
1	2	1	1	1

Input

-1	1	2
1	1	0
-1	-2	0

kernel 1

0	1	-1
0	1	1
1	0	-2

kernel 2

-2		

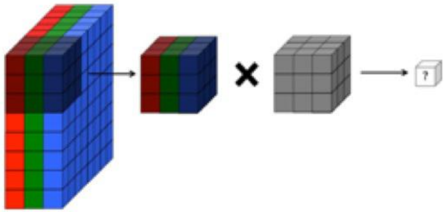
output (layer1)

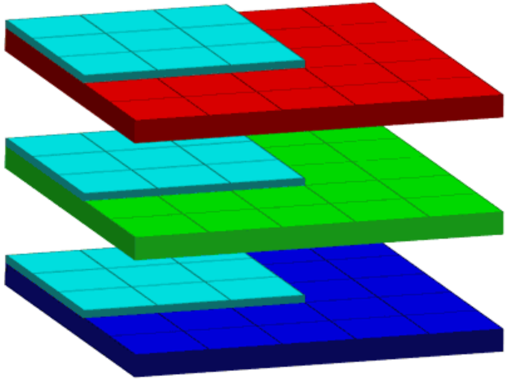
output (layer2)

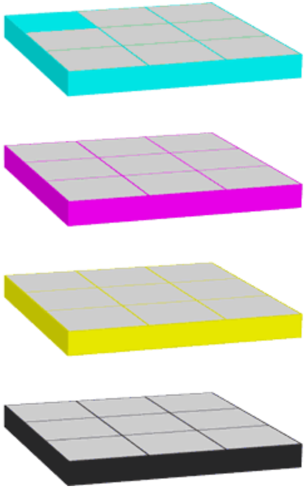
INPUT DEPTH

Each kernel has the same depth as the number of input channels

Each input on each channel has a single weight associated with it







CONVOLUTION IN TENSORFLOW

`tf.nn.conv2d(input, filter, strides, padding)`

`input`: 4d tensor [batch_size, height, width, channels]

`filter`: 4d: [height, width, channels_in, channels_out]

- Generally a Variable

`strides`: 4d: [1, vert_stride, horiz_strid, 1]

- First and last dimensions must be 1 (helps with under-the-hood math)

`padding`: string: ‘SAME’ or ‘VALID’

POOLING

POOLING

Idea: reduce neighboring pixels

Reduce dimensions of inputs (height and width)

No parameters!

MAX POOLING

2	1	0	-1
-3	8	2	5
1	-1	3	4
0	1	1	-2

maxpool

8	5
1	4

AVERAGE POOLING

2	1	0	-1
-3	8	2	5
1	-1	3	4
0	1	1	-2

avgpool

2	1.5
.25	1.5

GLOBAL POOLING

2	1	0	-1
-3	8	2	5
1	-1	3	4
0	1	1	-2

(Average pool over whole layer)

global pool

1.3125

ADDITIONAL CONVOLUTION OPERATION RESOURCE

Andrej Karpathy's convolutional network website

Created for Stanford's CS231n course

http://cs231n.github.io/convolutional-networks/

XAVIER (AND HE) INITIALIZATION

XAVIER INITIALIZATION

Want to initialize our weights such that the variance of the output of our *activation* is 1

Xavier Glorot and Bengio derived the following initialization scheme for activations with mean zero inputs:

$$W = TruncNormal(0.0, \sqrt{\frac{2}{n_{in} + n_{out}}})$$

RECOMMENDATION FOR RELUS

He et al. derived an initialization scheme specifically for ReLUs (which don't have a zero mean)

$$W = TruncNorm(0.0, \sqrt{\frac{2}{n_{in}}})$$

SIMPLIFIES THE TRAINING PROCEDURE.

Allows us to train “end-to-end”, without pre-training

Less time spent dealing with exploding gradients

No longer have to hand-tweak everything

Nice explanation:

Initializing neural networks

intel® Nervana™ AI Academy

Nervana AI Academy

ConvNets and Transfer Learning

Review

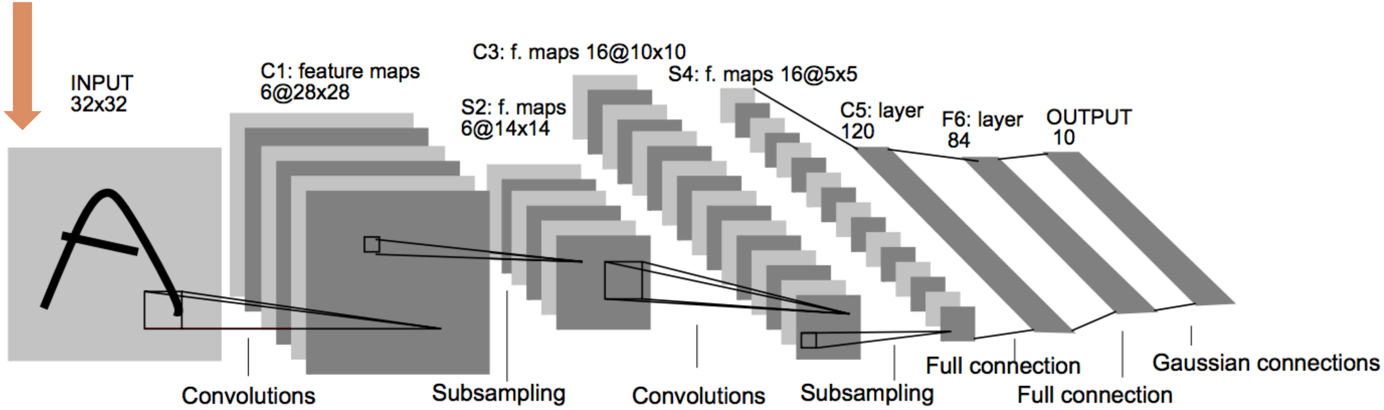
- Do some review of concepts from the last lecture
- We will revisit kernel, stride, and pooling in the context of the Le-Net 5 model

LeNet-5

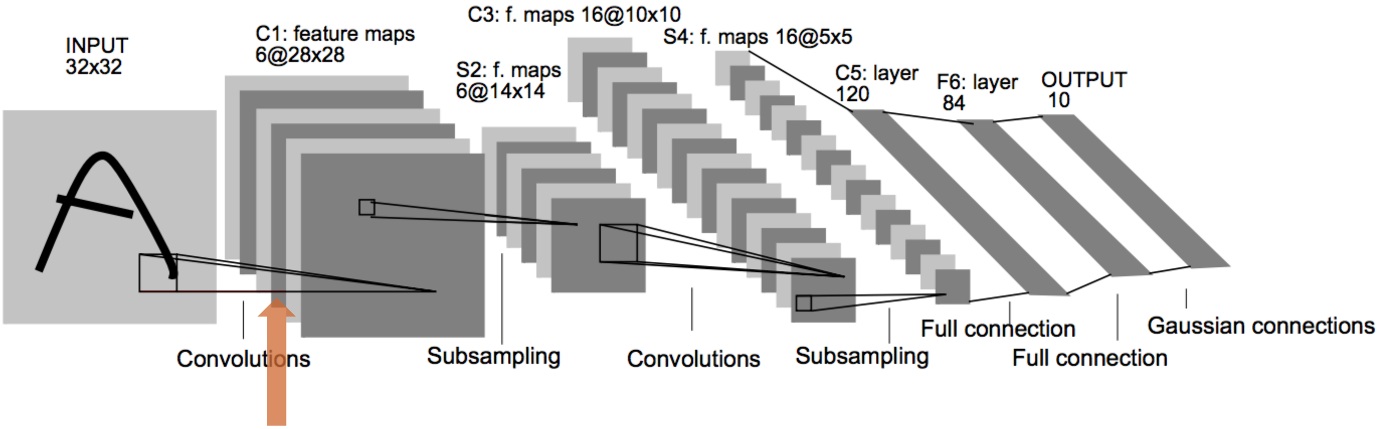
- Created by Yann LeCun in the 1990s
- Used on the MNIST data set
- Novel Idea: Use convolutions to efficiently learn features on data set

LeNet—Structure Diagram

Input: A 32 x 32 grayscale image (28 x 28) with 2 pixels of padding all around.

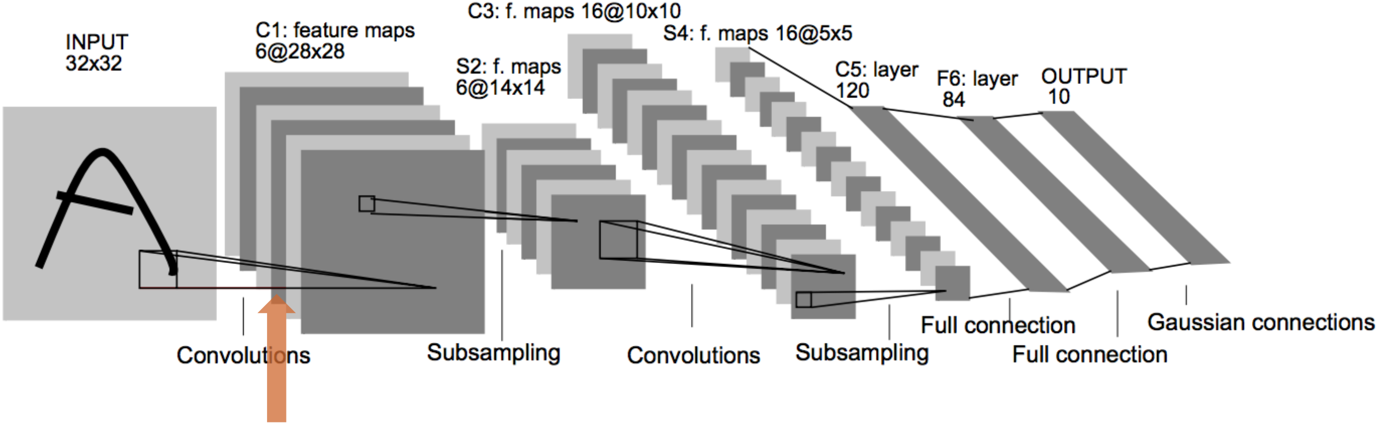


LeNet—Structure Diagram



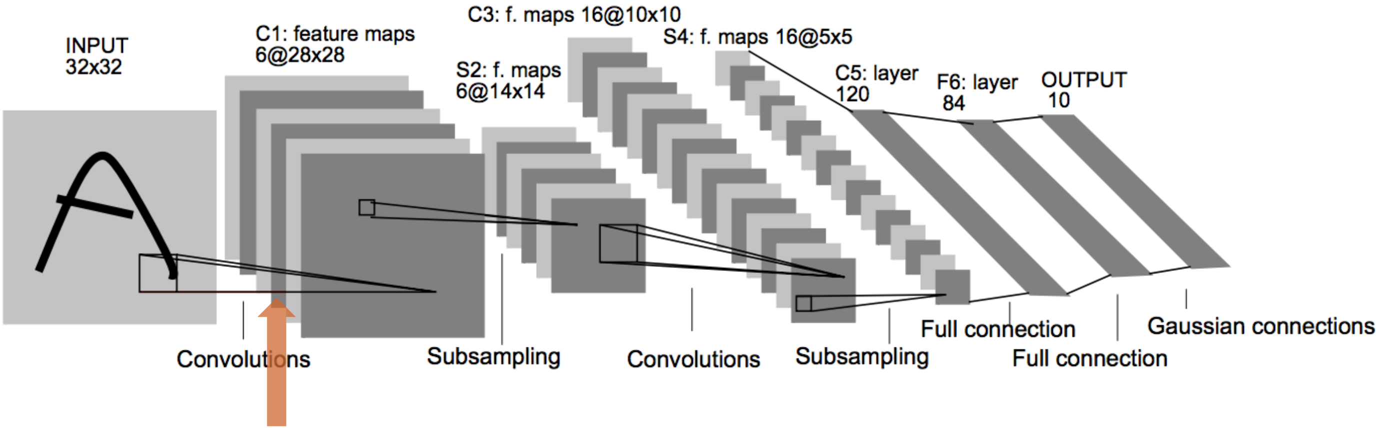
Next, we have a convolutional layer.

LeNet—Structure Diagram



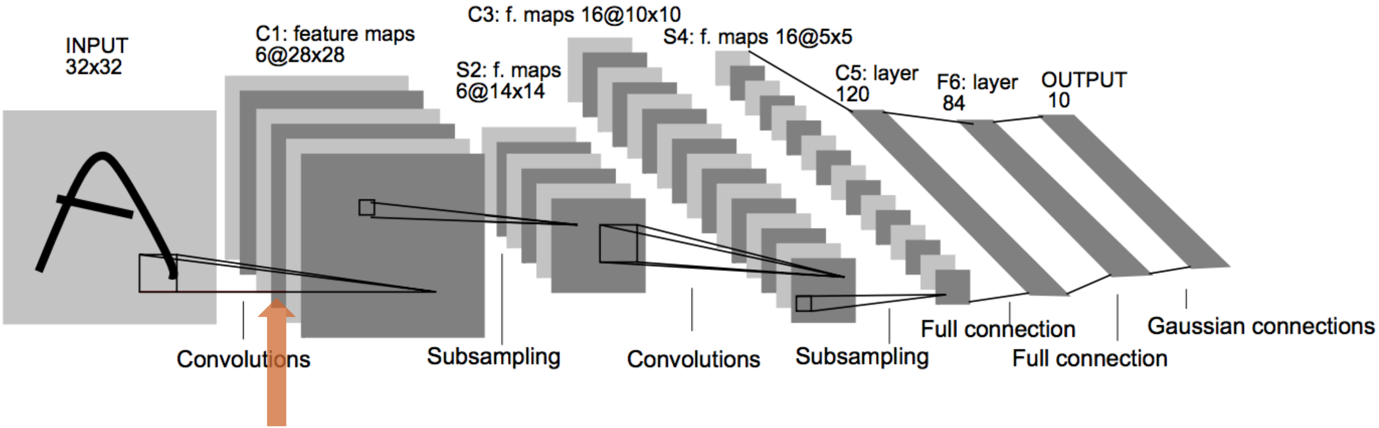
This is a 5x5 convolutional layer with stride 1.

LeNet—Structure Diagram



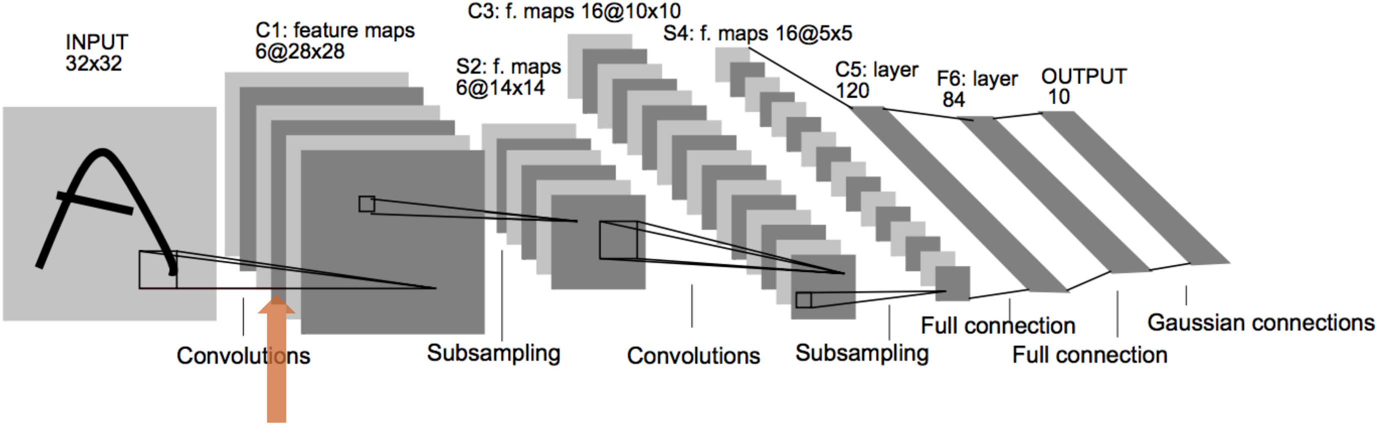
This means the resulting "filter" has dimension 28x28. (Why?)

LeNet—Structure Diagram



They use a depth of 6. This means there are 6 different kernels that are learned.

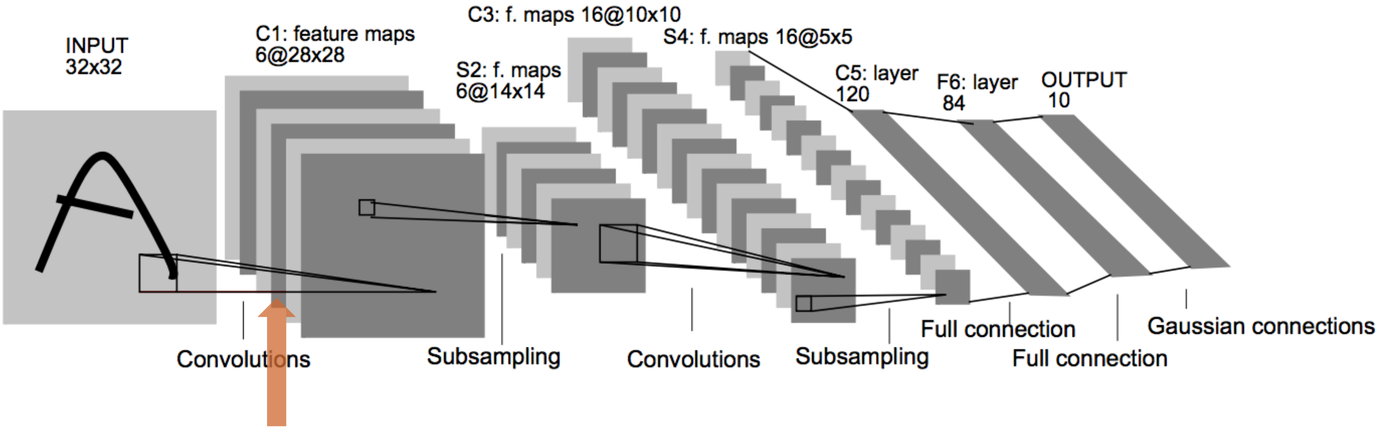
LeNet—Structure Diagram



They use a depth of 6. This means there are 6 different kernels that are learned.

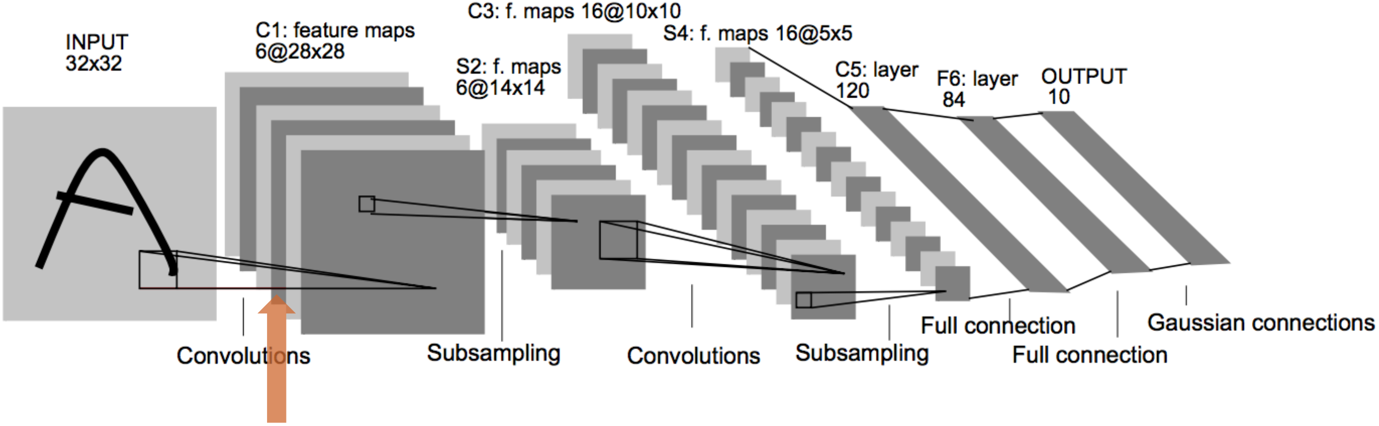
So the output of this layer is 6x28x28.

LeNet—Structure Diagram



What is the total number of weights in this layer?

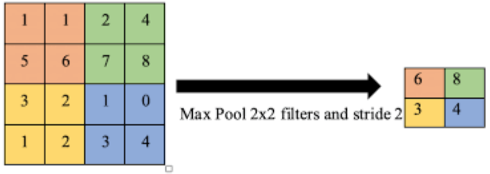
LeNet—Structure Diagram



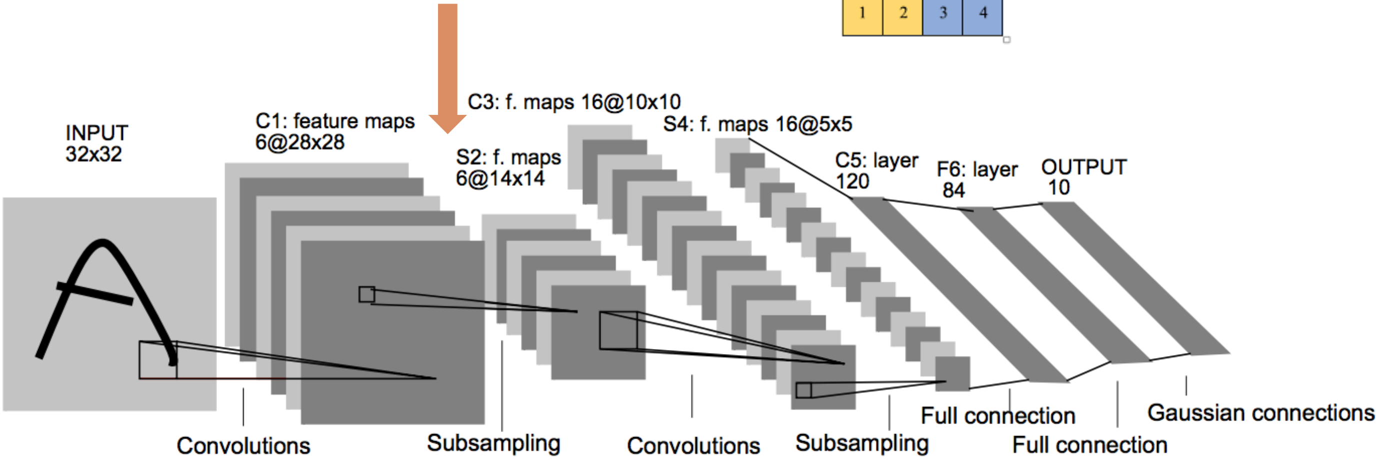
What is the total number of weights in this layer?

Answer: Each kernel has 5x5=25 weights (plus a bias term, so actually 26 weights). So total weights = 6x26 = 156.

LeNet—Structure Diagram

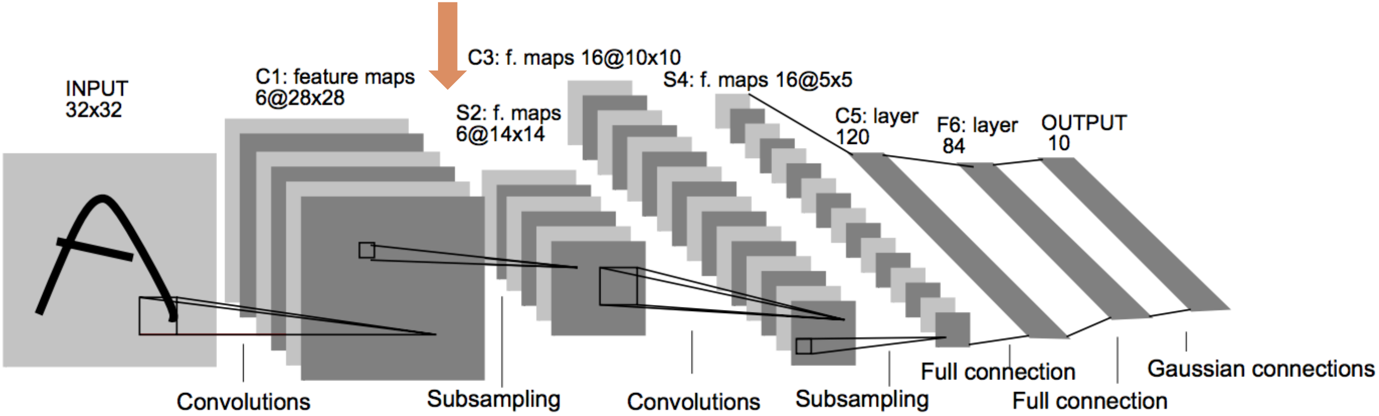


Next is a 2x2 pooling layer. (with stride 2)



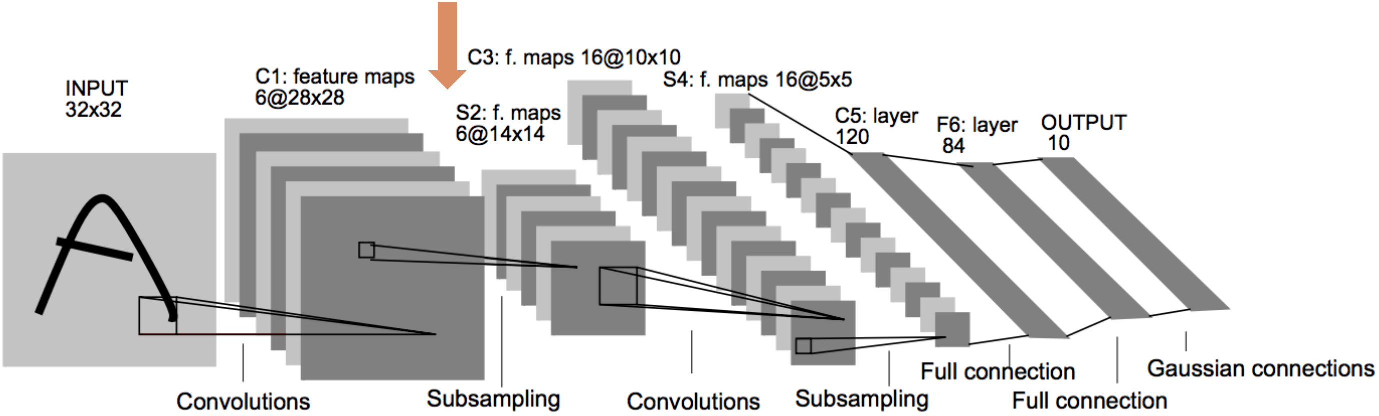
LeNet—Structure Diagram

So output size is 6x14x14.
(we downsample by a factor of 2)



LeNet—Structure Diagram

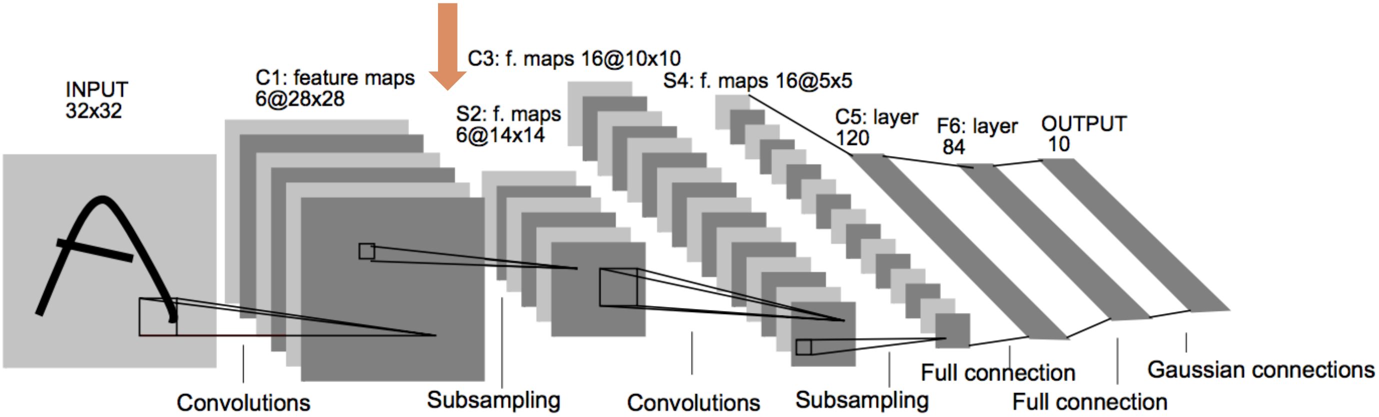
So output size is 6x14x14.
(we downsample by a factor of 2)



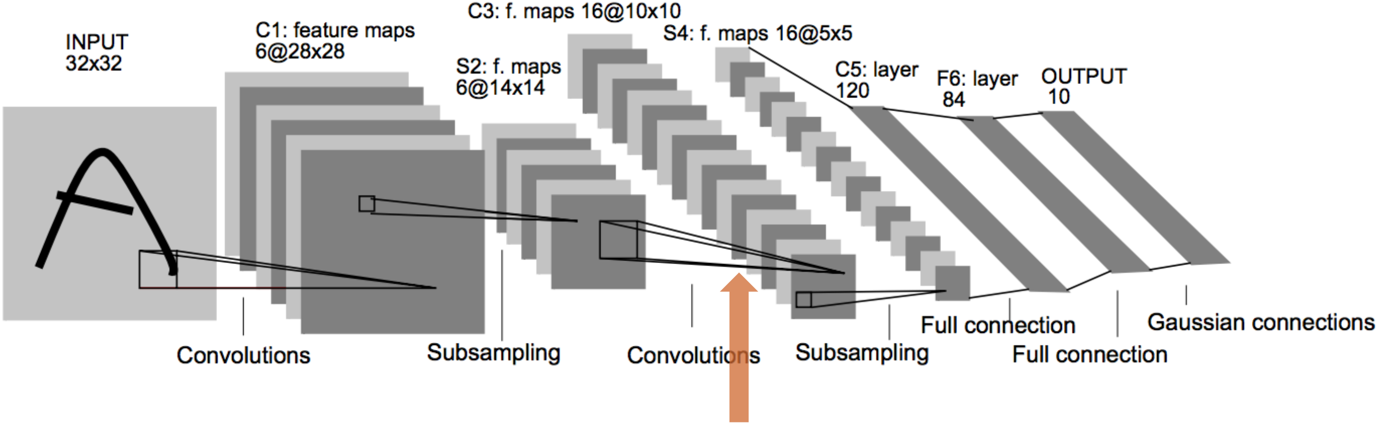
Note: The original paper actually does a more complicated pooling then max or avg. pooling, but this is considered obsolete now.

LeNet—Structure Diagram

No weights! (pooling layers have no weights to be learned – it is a fixed operation.)

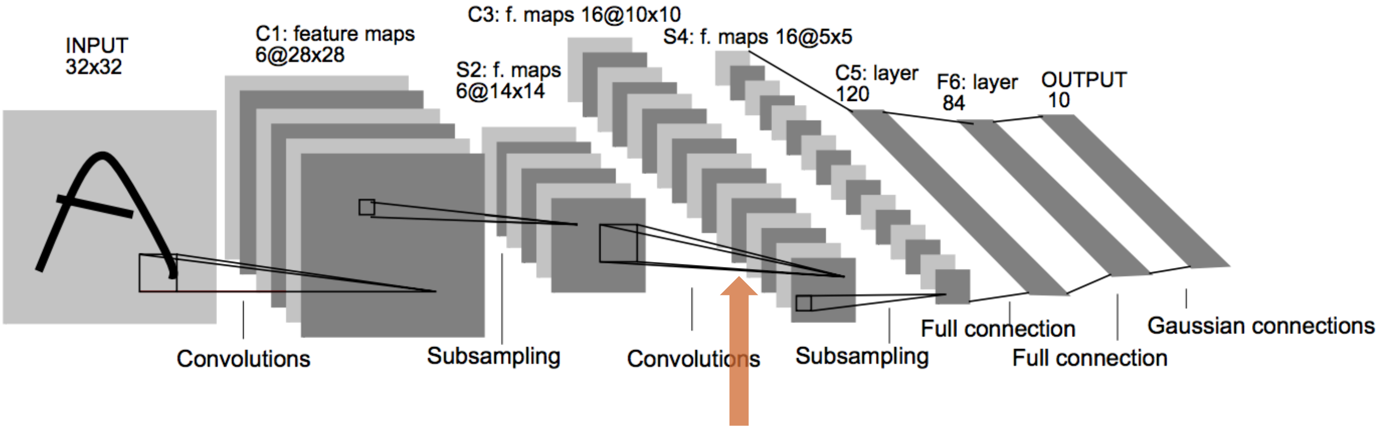


LeNet—Structure Diagram



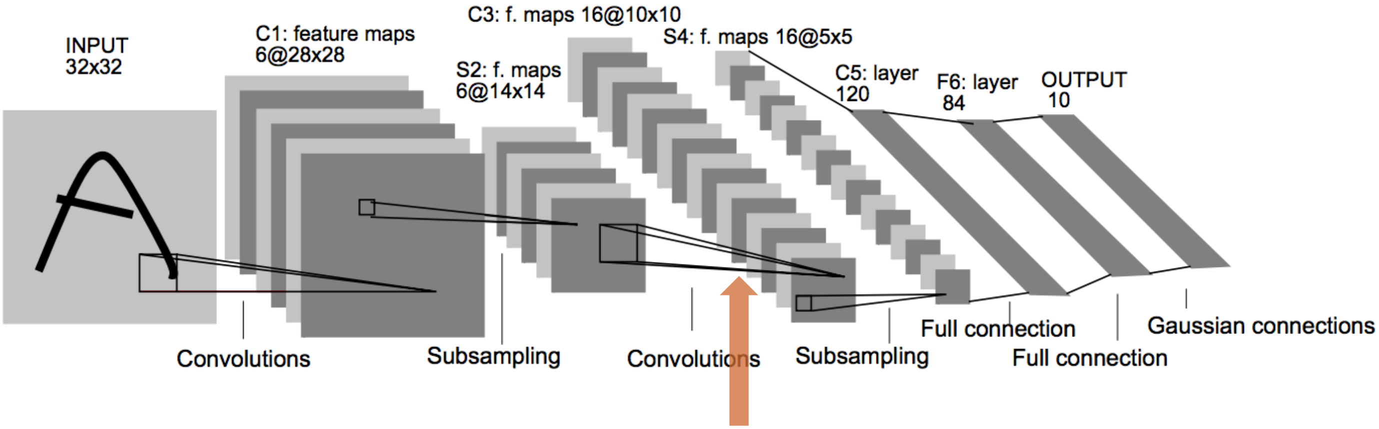
Another 5x5 convolutional layer with stride 2. This time the depth is 16.

LeNet—Structure Diagram



Output size: 16 x 10 x 10 How many weights? (tricky!)

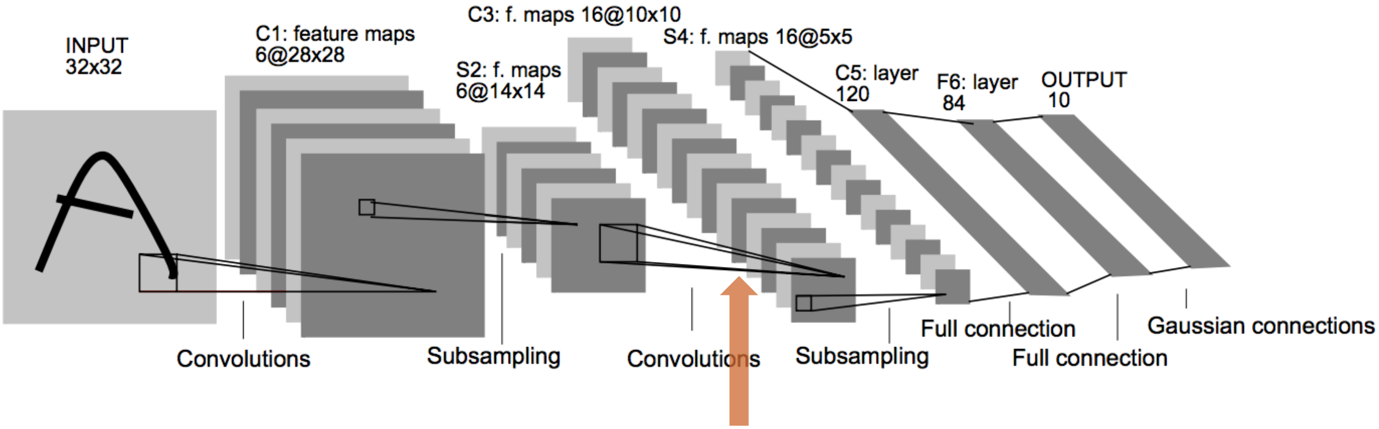
LeNet—Structure Diagram



The kernels “take in” the full depth of the previous layer. So each 5x5 kernel now “looks at” 6x5x5 pixels.

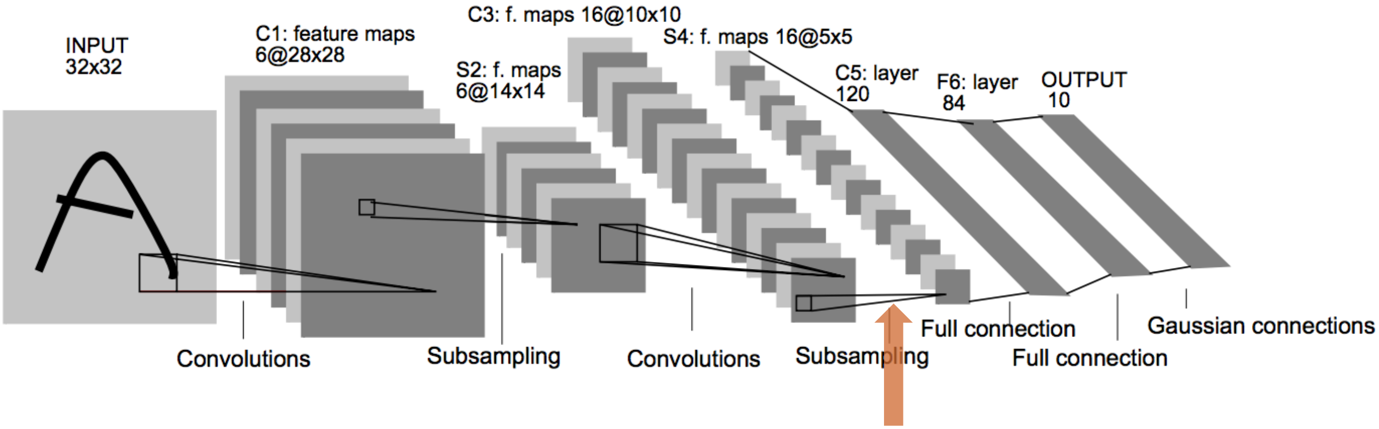
Each kernel has 6x5x5 = 150 weights + bias term = 151.

LeNet—Structure Diagram



So, total weights for this layer = 16*151 = 2416.

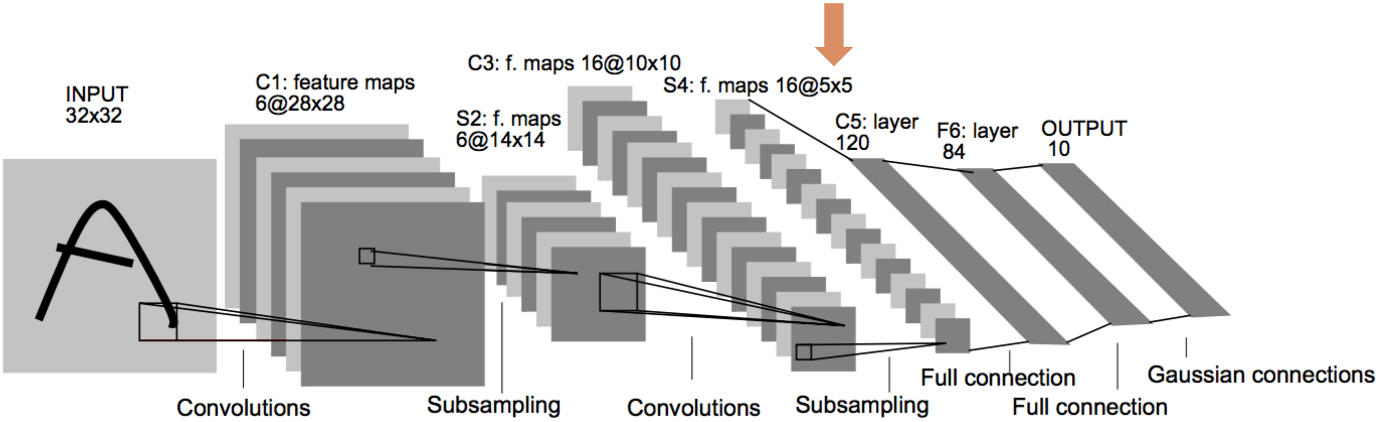
LeNet—Structure Diagram



Another 2x2 pooling layer.
Output is 16 x 5 x 5.

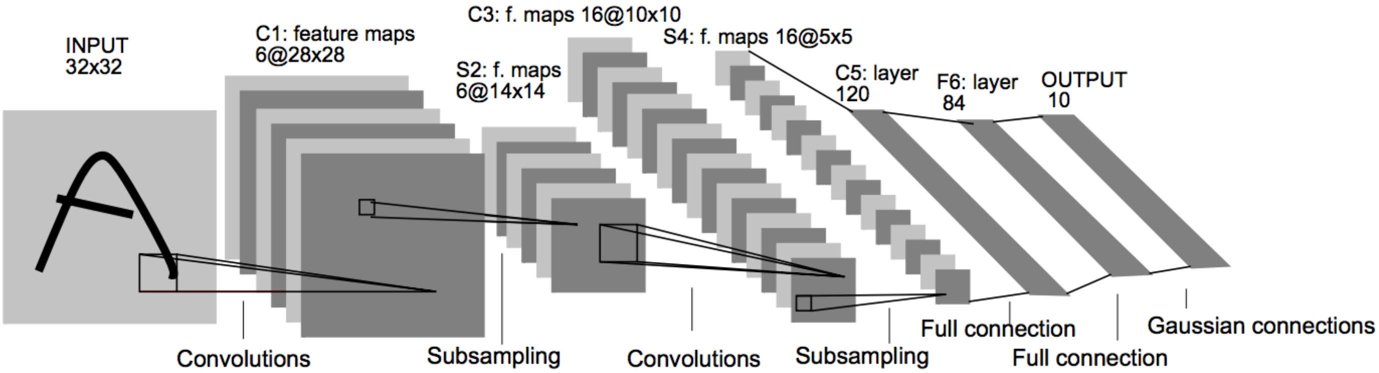
LeNet—Structure Diagram

We "flatten" this to a length 400 vector. (not shown)



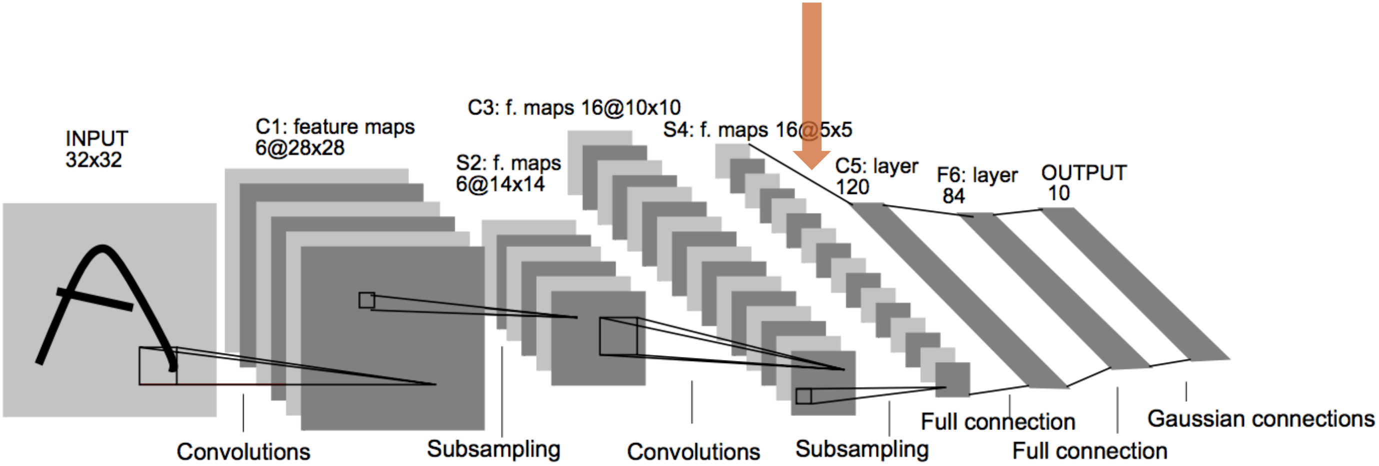
LeNet—Structure Diagram

The following layers are just fully connected layers!



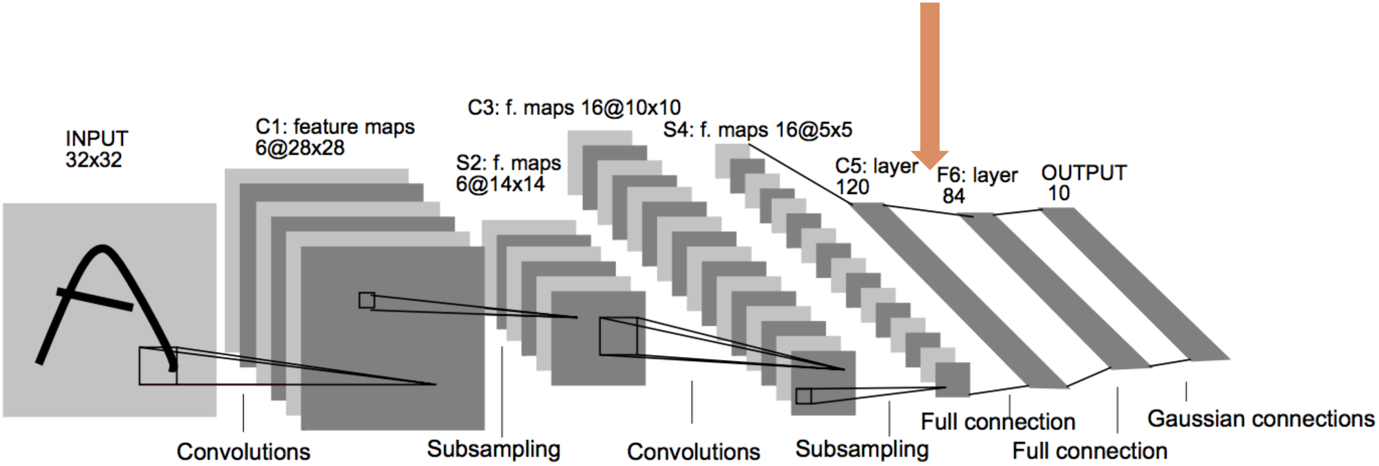
LeNet—Structure Diagram

From 400 to 120.



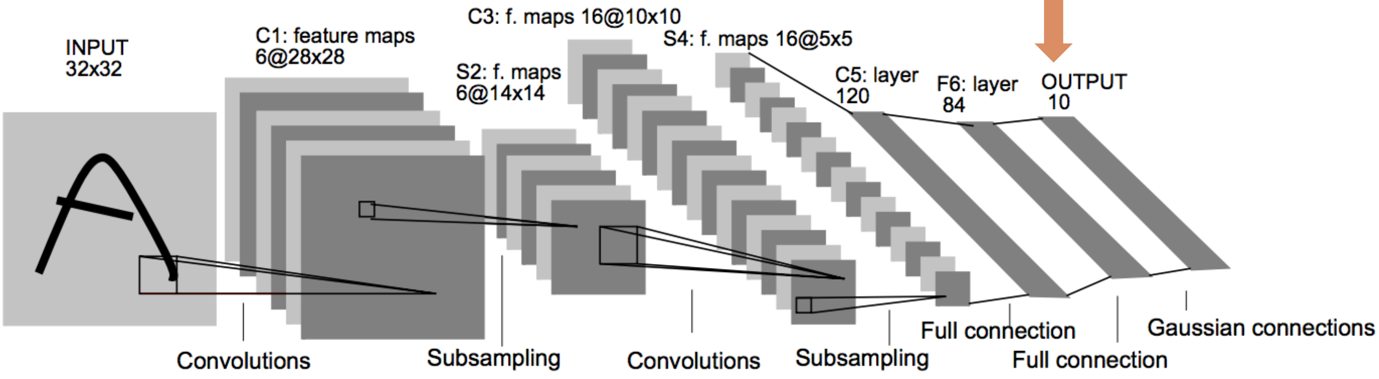
LeNet—Structure Diagram

Then from 120 to 84.



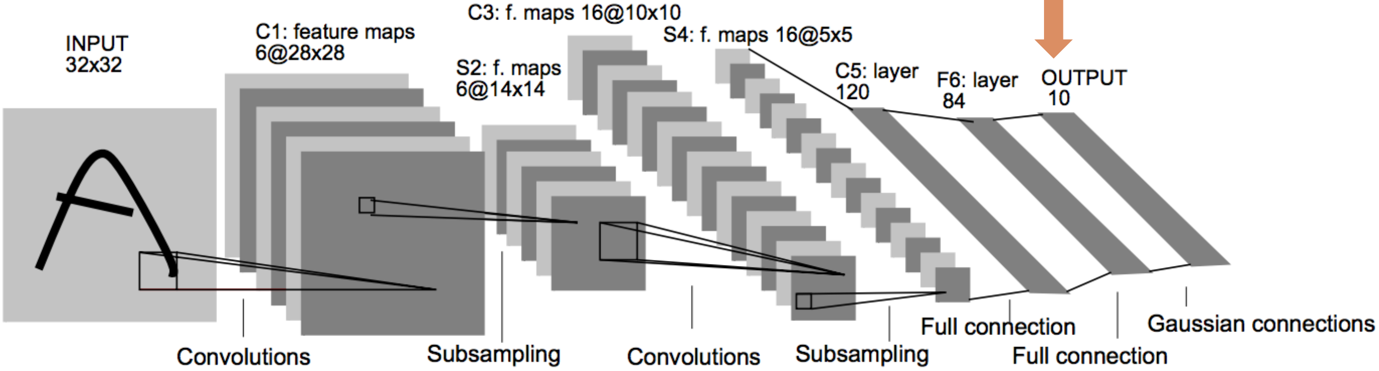
LeNet—Structure Diagram

Then from 84 to 10.



LeNet—Structure Diagram

And a softmax output of size 10 for the 10 digits.



LeNet-5

How many total weights in the network?

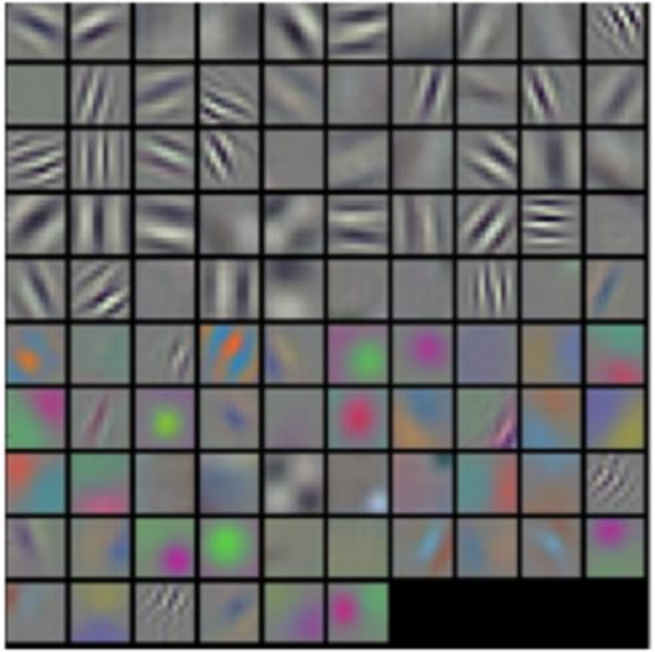
Conv1: 1*6*5*5 + 6	=	156
Conv3: 6*16*5*5 + 16	=	2416
FC1: 400*120 + 120	=	48120
FC2: 120*84 + 84	=	10164
FC3: 84*10 + 10	=	850
Total:	**=**	**61706**

Less than a single FC layer with [1200x1200] weights!

Note that Convolutional Layers have relatively few weights.

Motivation

- Early layers in a Neural Network are the hardest (i.e. slowest) to train
- Due to vanishing gradient property
- But these ”primitive” features should be general across many image classification tasks



Motivation

- Later layers in the network are capturing features that are more particular to the specific image classification problem
- Later layers are easier (quicker) to train since adjusting their weights has a more immediate impact on the final result

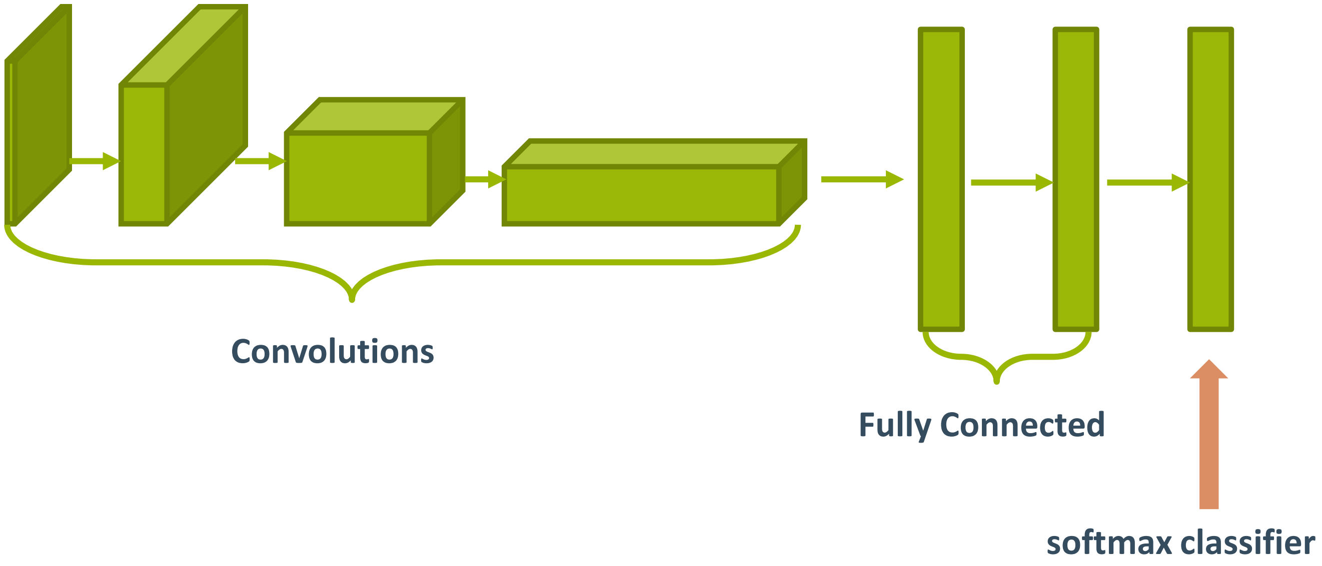
Motivation

- Famous, competition-winning models are difficult to train from scratch
  - Huge datasets (like ImageNet)
  - Long number of training iterations
  - Very heavy computing machinery
  - Time experimenting to get hyper-parameters just right

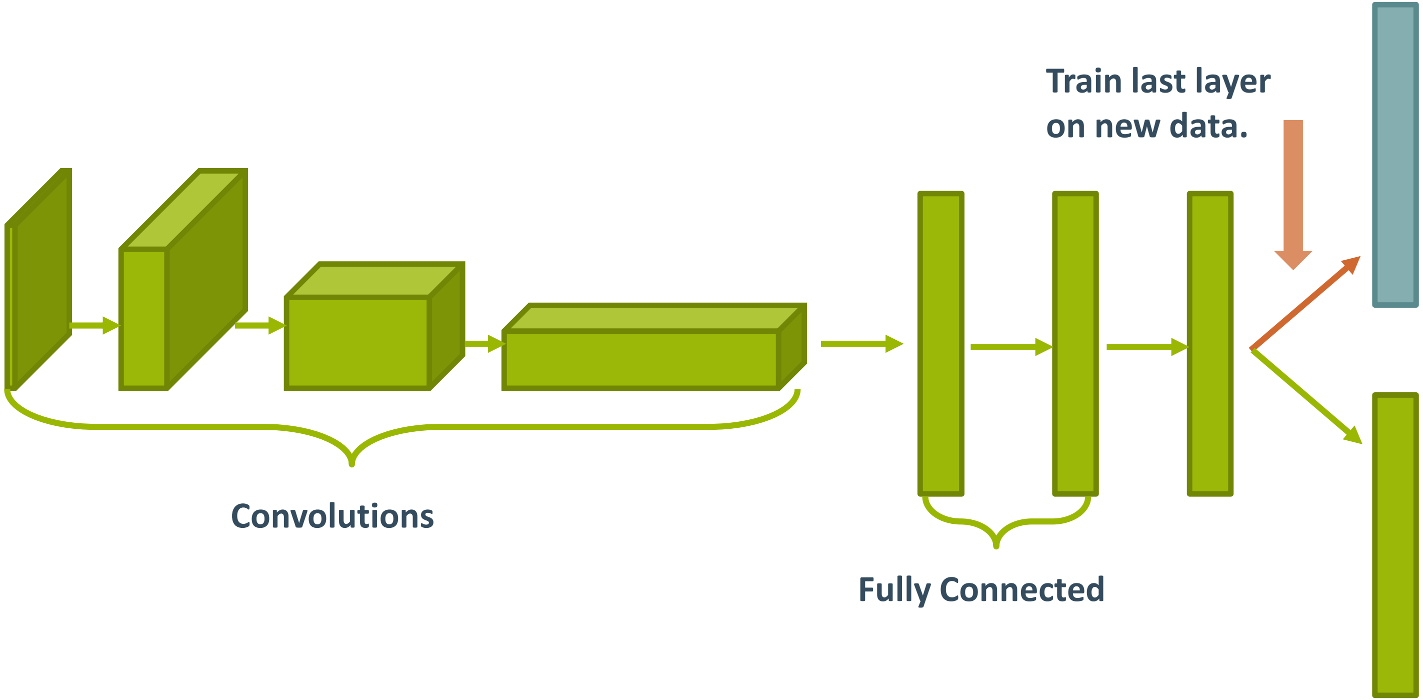
Transfer Learning

- However, the basic features (edges, shapes) learned in the early layers of the network *should* generalize
- Results of the training are just weights (numbers) that are easy to store
- Idea: keep the early layers of a pre-trained network, and re-train the later layers for a specific application
- This is called ***Transfer Learning***

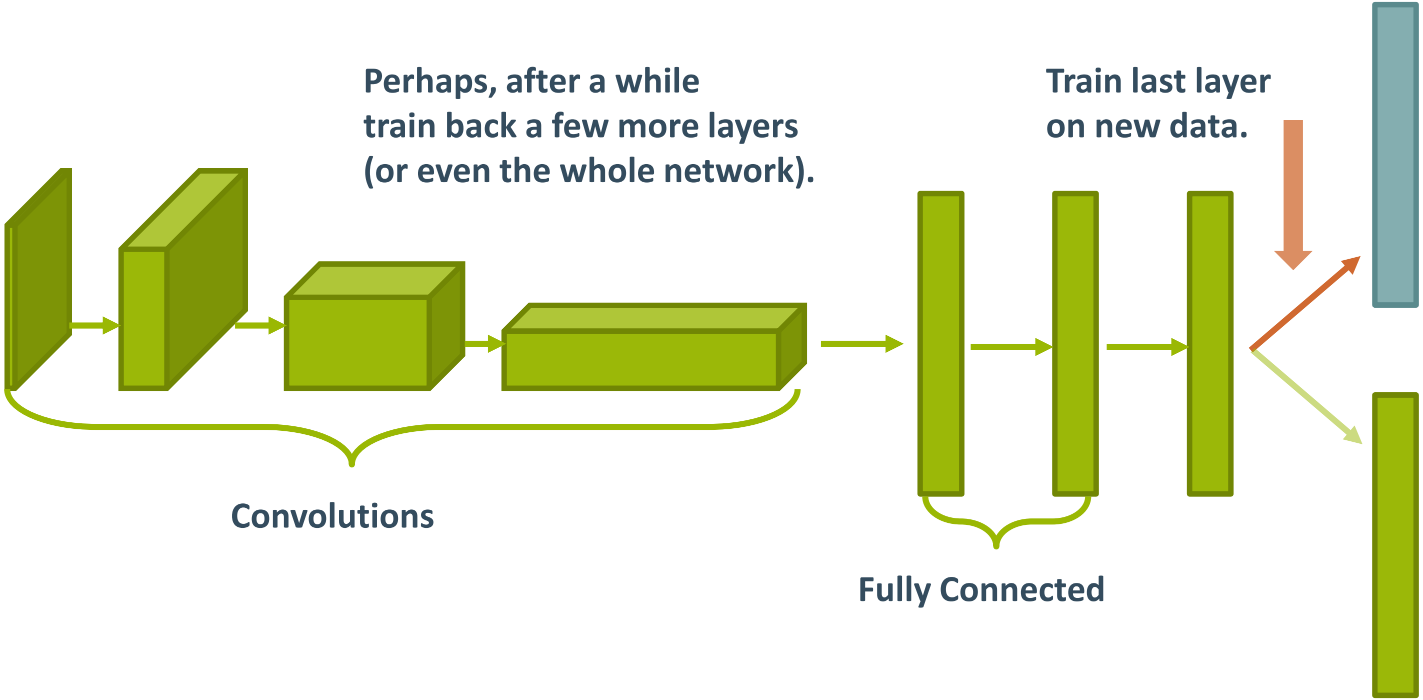
Transfer Learning



Transfer Learning



Transfer Learning



Transfer Learning Options

- The additional training of a pre-trained network on a specific new dataset is referred to as "Fine-Tuning"
- There are different options on "how much" and "how far back" to fine-tune
  - Should I train just the very last layer?
  - Go back a few layers?
  - Re-train the entire network (from the starting point of the existing network)?

Guiding Principles for Fine-Tuning

While there are no "hard and fast" rules, there are some guiding principles to keep in mind.

1) The more similar your data and problem are to the source data of the pre-trained network, the less fine-tuning is necessary

E.g. *Using a network trained on ImageNet to distinguish "dogs" from "cats" should need relatively little fine-tuning. It already distinguished different breeds of dogs and cats, so likely has all the features you will need.*

Guiding Principles for Fine-Tuning

2) The more data you have about your specific problem, the more the network will benefit from longer and deeper fine-tuning

E.g. *If you have only 100 dogs and 100 cats in your training data, you probably want to do very little fine-tuning. If you have 10,000 dogs and 10,000 cats you may get more value from longer and deeper fine-tuning.*

Guiding Principles for Fine-Tuning

3) If your data is substantially different in nature than the data the source model was trained on, Transfer Learning may be of little value

E.g. *A network that was trained on recognizing typed Latin alphabet characters would not be useful in distinguishing cats from dogs. But it likely would be useful as a starting point for recognizing Cyrillic Alphabet characters.*

intel® Nervana™ AI Academy

ADVANCED TECHNIQUES FOR CNNS AND KERAS

DATA AUGMENTATION

- One practical obstacle to building image classifiers is obtaining labeled training data.
- Finding images is difficult.
- Labeling images is time consuming and costly.
- How can me make the most out of the labelled data we have?

DATA AUGMENTATION

If this is a chair:

DATA AUGMENTATION

If this is a chair…

Then so is this!

DATA AUGMENTATION

If this is a chair...

Also this:

DATA AUGMENTATION

If this is a chair…

Also this:

DATA AUGMENTATION

- By slightly altering images, we can increase our effective data size.
- Also allows the neural network to learn invariance to certain transformations.
- But we need to be careful—this can have unintended consequences.

DATA AUGMENTATION

Would not want a self-driving car to think these mean the same thing!

DATA FLOWS IN KERAS

- Keras has a convenient mechanism for Data Augmentation.
- Requires use of “Data Generators”
- To date, we have used the standard `model.fit` mechanism
- Holds entire dataset in memory
- Reads the batches one by one out of memory

DATA FLOWS IN KERAS

- Alternative mechanism is to use a "data generator"
- Idea: define a generator object which "serves" the batches of data.
- Then use **`model.fit_generator`** instead of **`model.fit`**
- Generators can be used to serve images from disk to conserve working memory

IMAGEDATAGENERATOR

- Keras has an `ImageDataGenerator` class which permits “real-time” data-augmentation.
- When a batch of images is served, you can specify random changes to be made to the image.
- These include shifting, rotating, flipping, and various normalizations of the pixel values.

IMAGEDATAGENERATOR

```
keras.preprocessing.image.ImageDataGenerator(
featurewise_center=False, samplewise_center=False,
featurewise_std_normalization=False, samplewise_std_normalization=False,
zca_whitening=False,
rotation_range=0.,
width_shift_range=0.,
height_shift_range=0.,
shear_range=0., zoom_range=0., channel_shift_range=0., fill_mode='nearest',
cval=0.,
horizontal_flip=False, vertical_flip=False,
 rescale=None, preprocessing_function=None,
data_format=K.image_data_format())
```

Lots of options! We'll discuss a few.

SHIFTING IMAGES

```
keras.preprocessing.image.ImageDataGenerator(
width_shift_range=0.,
height_shift_range=0.,
...)
```

- These determine the range of possible horizontal or vertical shifts to make to the image.
- Measured as a percentage of the image size.
- So if an image is 200 x 200, and **`width_shift_range=0.1`**, then it will shift up to 20 pixels to the left or right.

SHIFTING IMAGES (HOW TO FILL IN)

```
keras.preprocessing.image.ImageDataGenerator(
...,
fill_mode='nearest', cval=0.,
...)
```

- When shifting, we don't wish to change the proportions of the image.
- We need to "fill in" the pixels on the other side.
- Options are "**`constant`**", "**`nearest`**", "**`reflect`**", "**`wrap`**"
- The cval is the value when "constant" is specified.

ROTATING IMAGES

```
keras.preprocessing.image.ImageDataGenerator(
...,
rotation_range=0.,
...)
```

- This allows us to specify a range of possible rotations
- Measured in degrees
- So rotation_range=30 means up to a 30 degree rotation (in either direction)

FLIPPING IMAGES

```
keras.preprocessing.image.ImageDataGenerator(
...,
horizontal_flip=False, vertical_flip=False,
...)
```

Whether or not to randomly flip in a horizontal or vertical direction.

Intel Nervana™ AI Academy