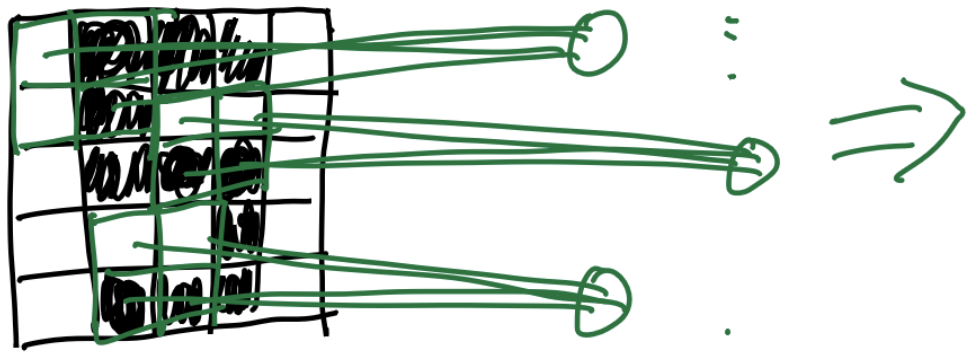
Convolutional Networks Cont.

Replicate the same neuron across the image!

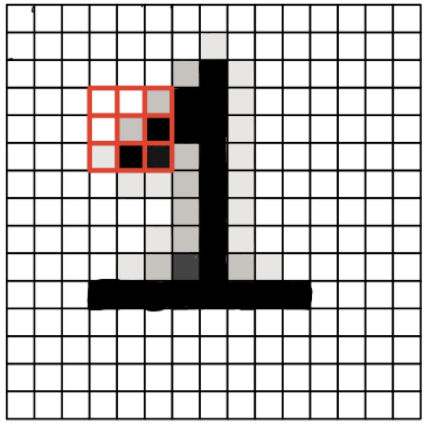
- That is, share the same weights for every neuron
- Activate on the same pattern anywhere



We call this replicated pattern of weights a kernel

w_1	w_2
w_3	w_4

Convolutions in 2-D



Align kernel in every 2-d location

$$\text{Conv2d}(x, k)_{ij} = \sum_{a=1}^{s} \sum_{b=1}^{s} x_{i+a, j+b} \cdot k_{ab}$$

[For padding = 'valid']

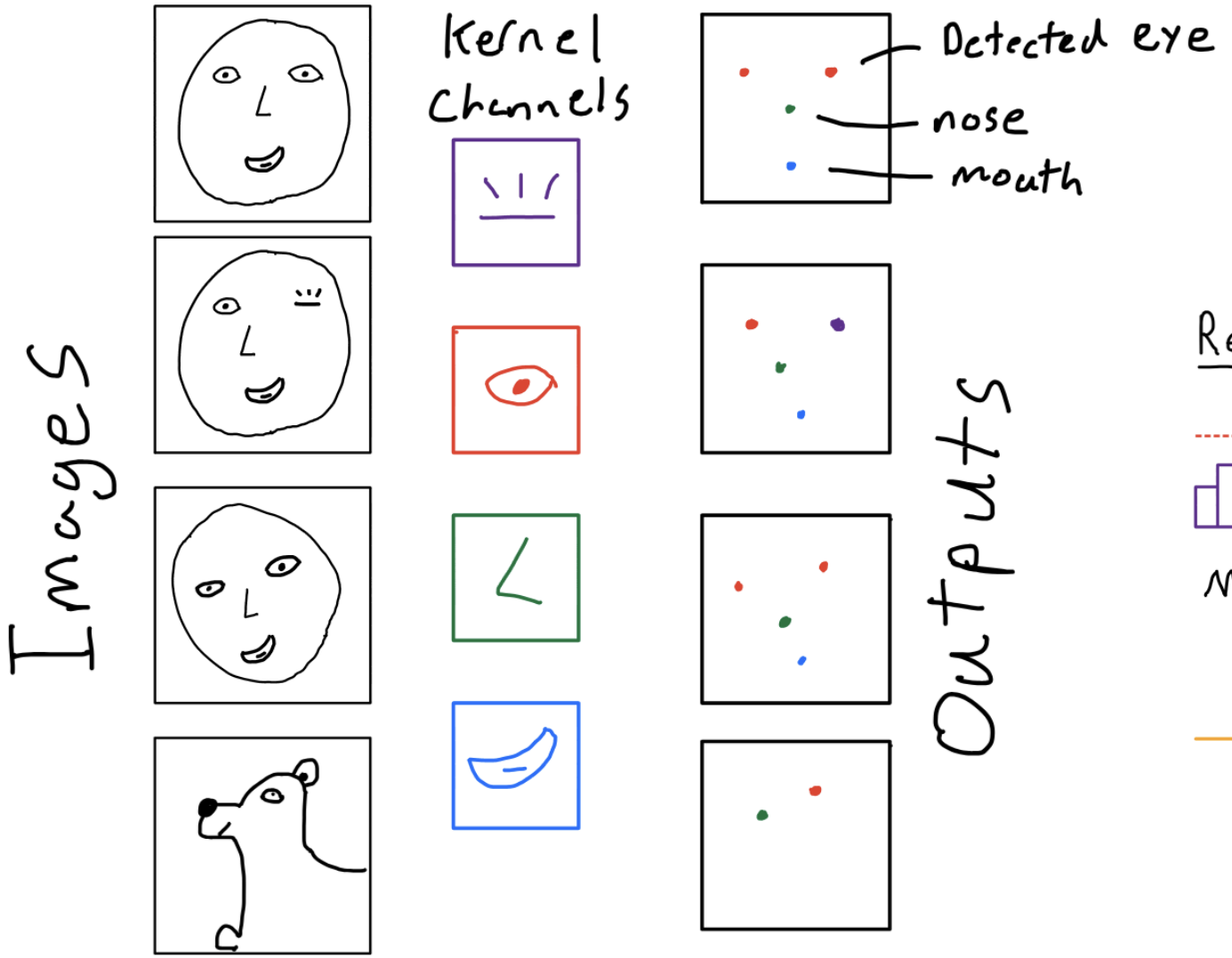
5	-1	3	8	4	6	-2
-3	4	9	1	1	7	-4
5	-6	3	2	1	-2	0
0	-8	5	5	4	-2	3
4	7	2	-7	3	1	4
4	-3	1	2	6	1	-1
0	0	3	-2	3	-1	0

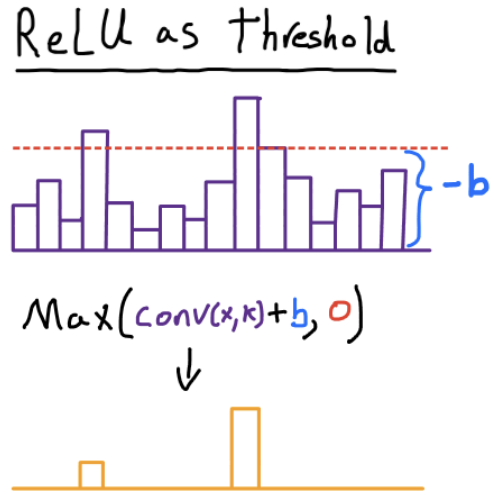
-1	-2	-1
0	0	0
1	2	1

			1	

$$= 1 \cdot (-1) + 1 \cdot (-2) + 7 \cdot (-1) + 2 \cdot 0 + 1 \cdot 0 +$$
$$(-2) \cdot 0 + 5 \cdot 1 + 4 \cdot 2 + (-2) \cdot 1$$

Convolutions as feature detectors





General 2-D Convolutional Layer

Input ($\underline{X}$): $N \times H \times W \times C$

- N: # Images (Batch)
- H: Height
- W: Width
- C: # Channels

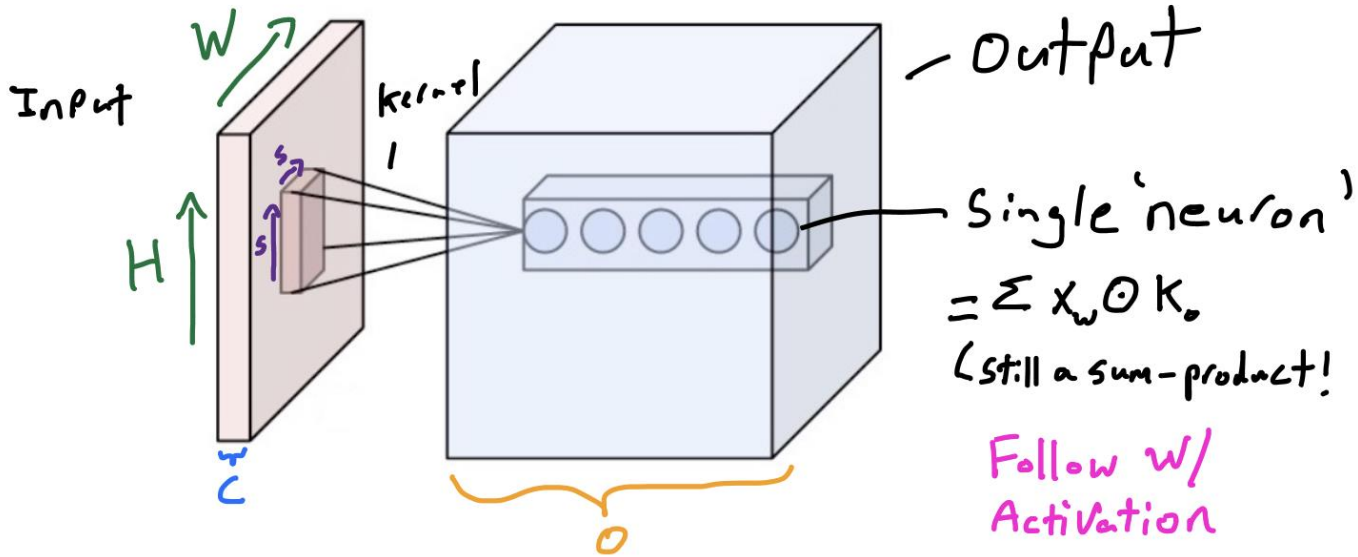
Kernel ($\underline{K}$): $S \times S \times C \times O$

- S: Kernel size
- C: # channels
- O: # output channels

Output (ϕ): $N \times H \times W \times O$

May change for padding $\neq$ 'same'

Diagram:



Single 'neuron'

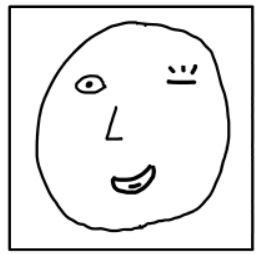
$= \Sigma X_w \odot K_w$

(still a sum-product!

Follow w/ Activation

Multi-Layer CNNs









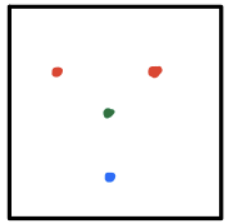
Detect low-level features





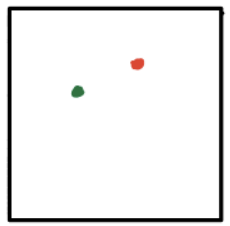












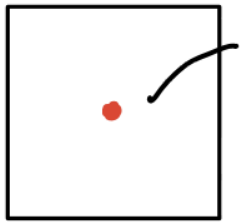
Detect high-level features



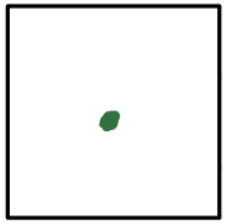


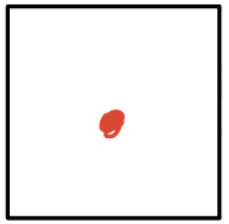


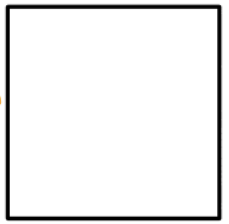
Exact location of features doesn't matter



Detected face

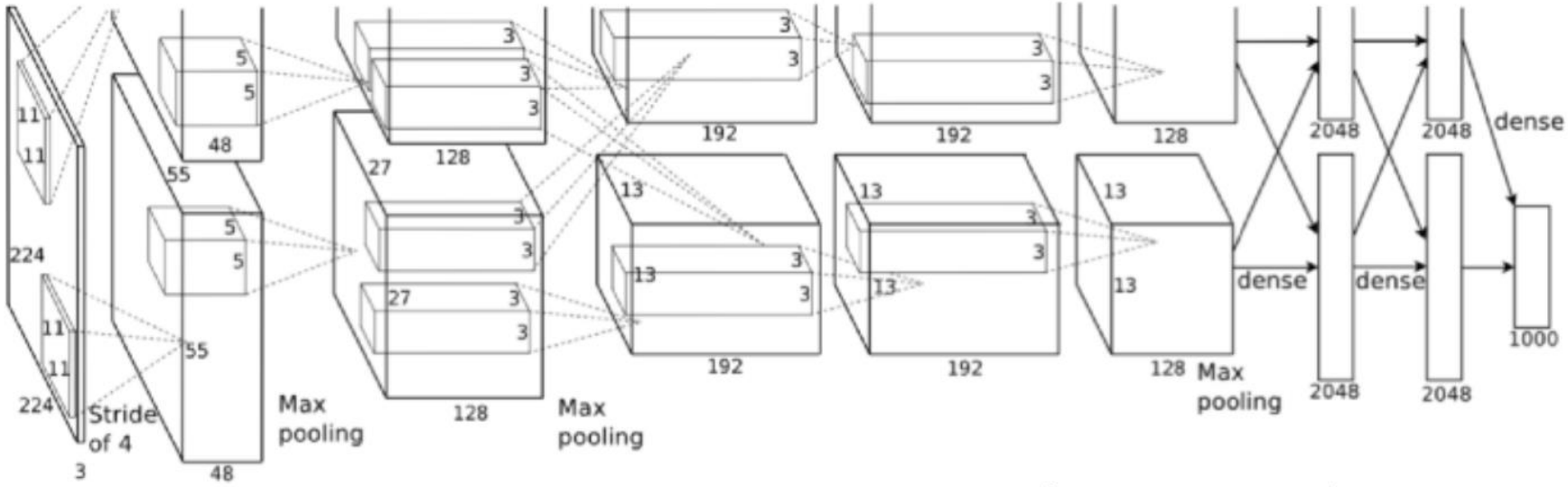




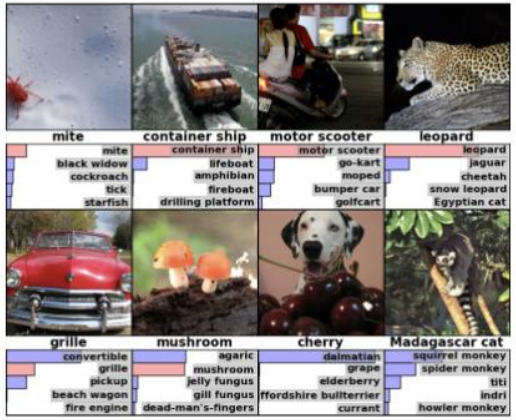


AlexNet (2012)

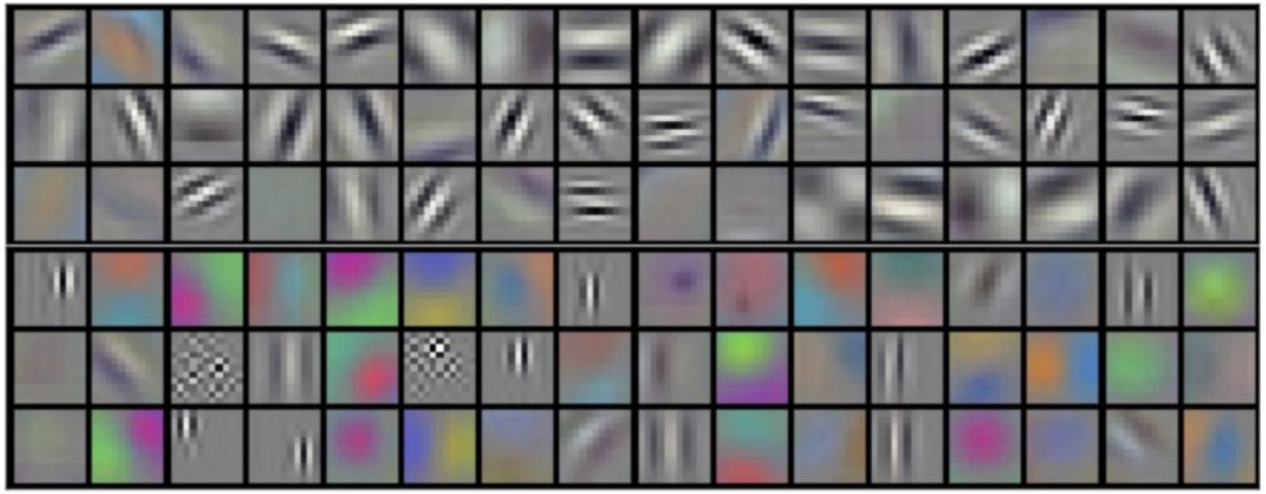
Architecture:



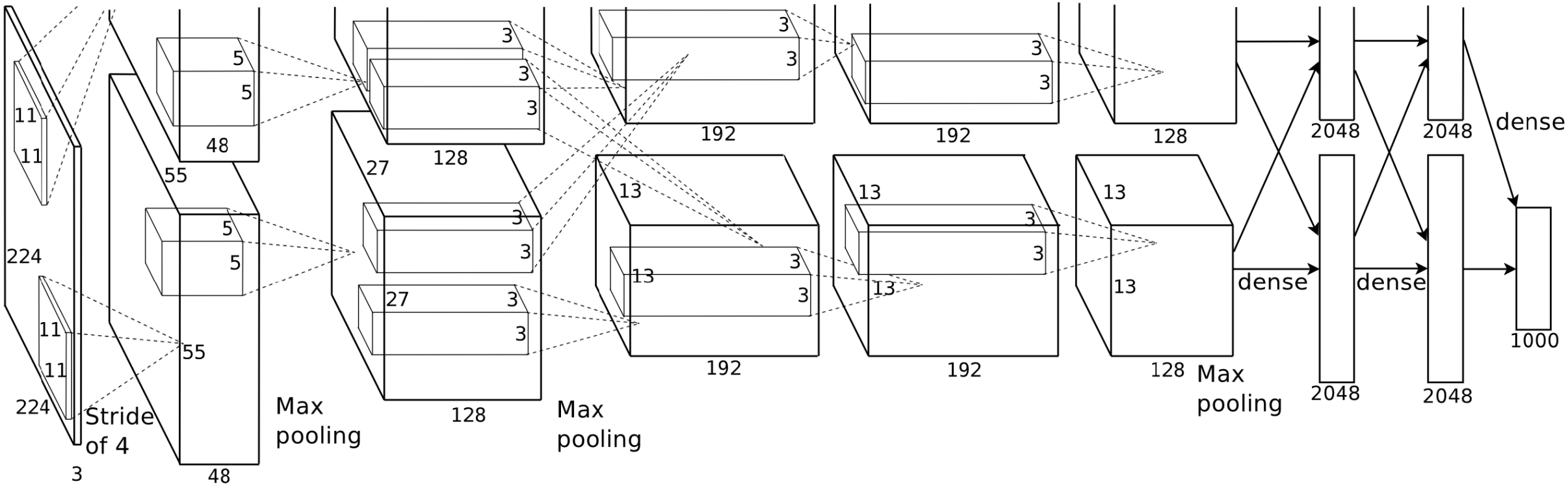
1000 Classes

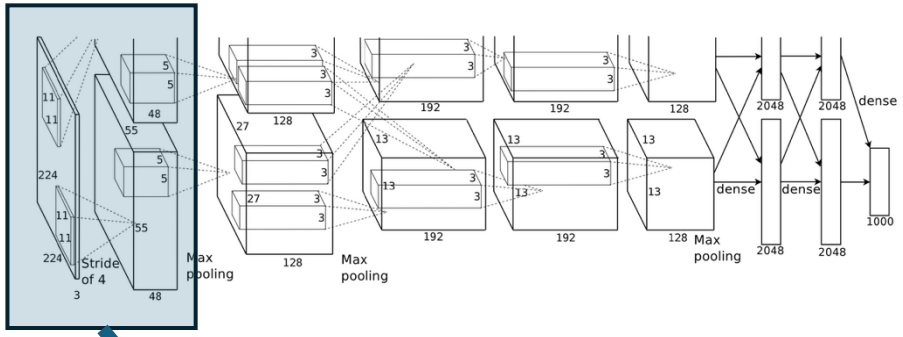


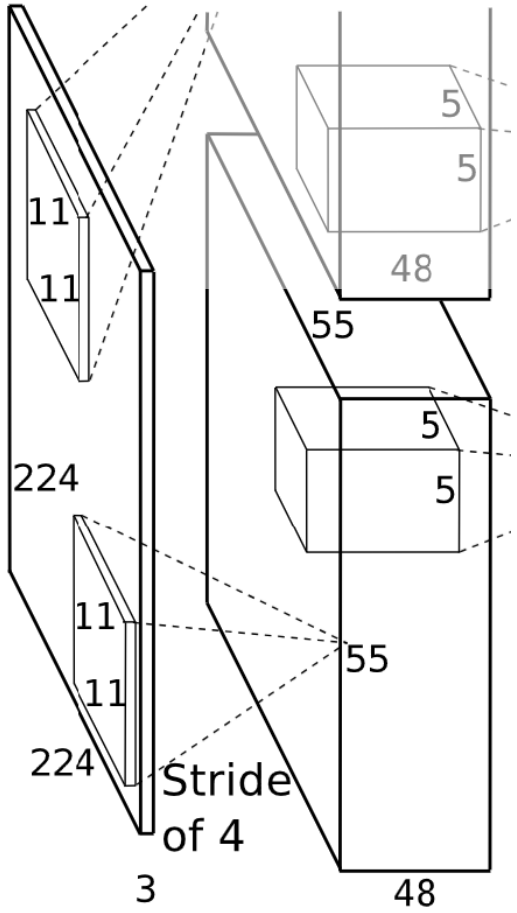
Learned kernels



Let’s break down AlexNet together!







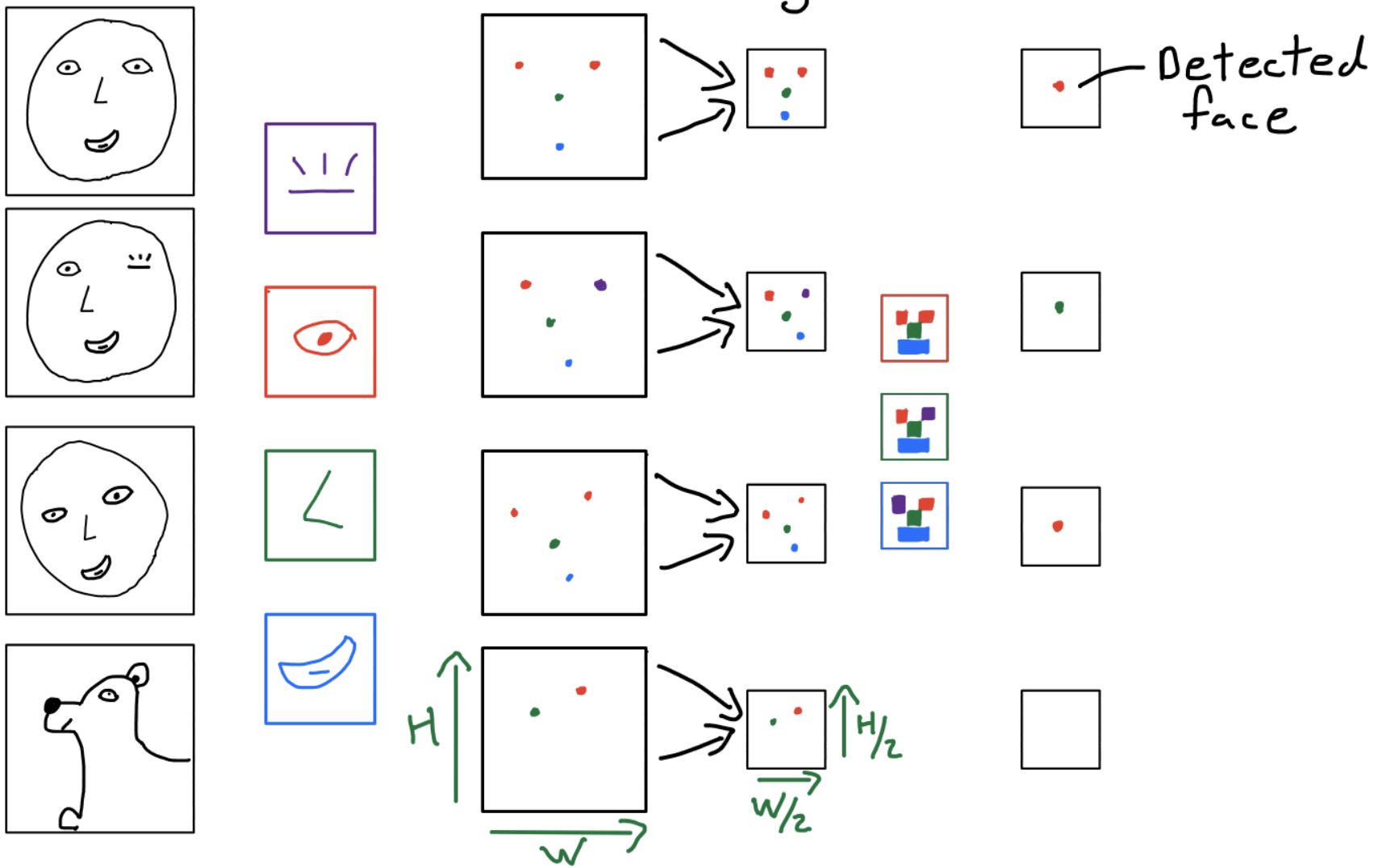
First layer

Questions:

- What is the resolution of the input image?
- Is it a color or grayscale image?
- What size is the *kernel*?
- How many neurons are in this layer?
- *What is the stride?*

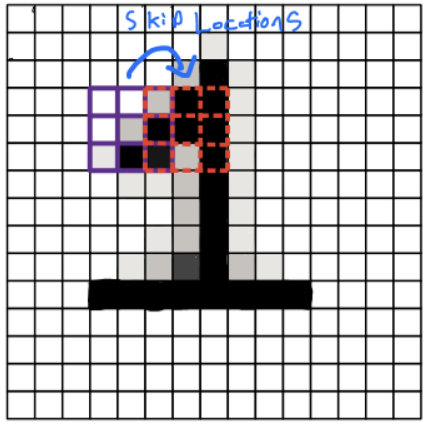
Down Sampling

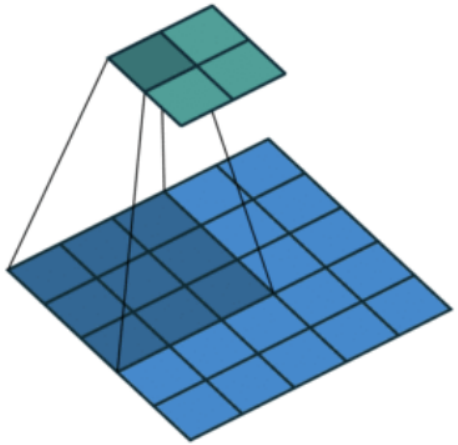
Reduce resolution → Easier and less expensive to find high-level features



Strided Convolutions (2-D)

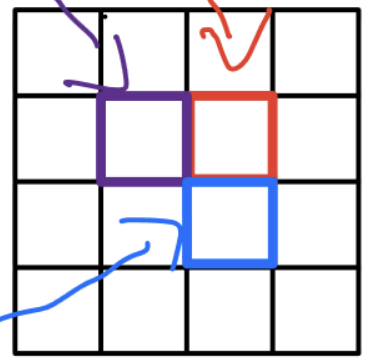
- Apply stride in each dimension





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

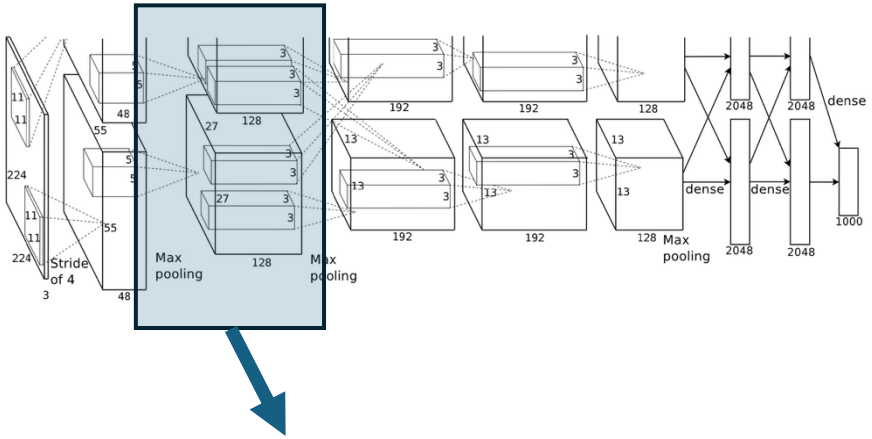
-1	-2	-1
0	0	0
1	2	1

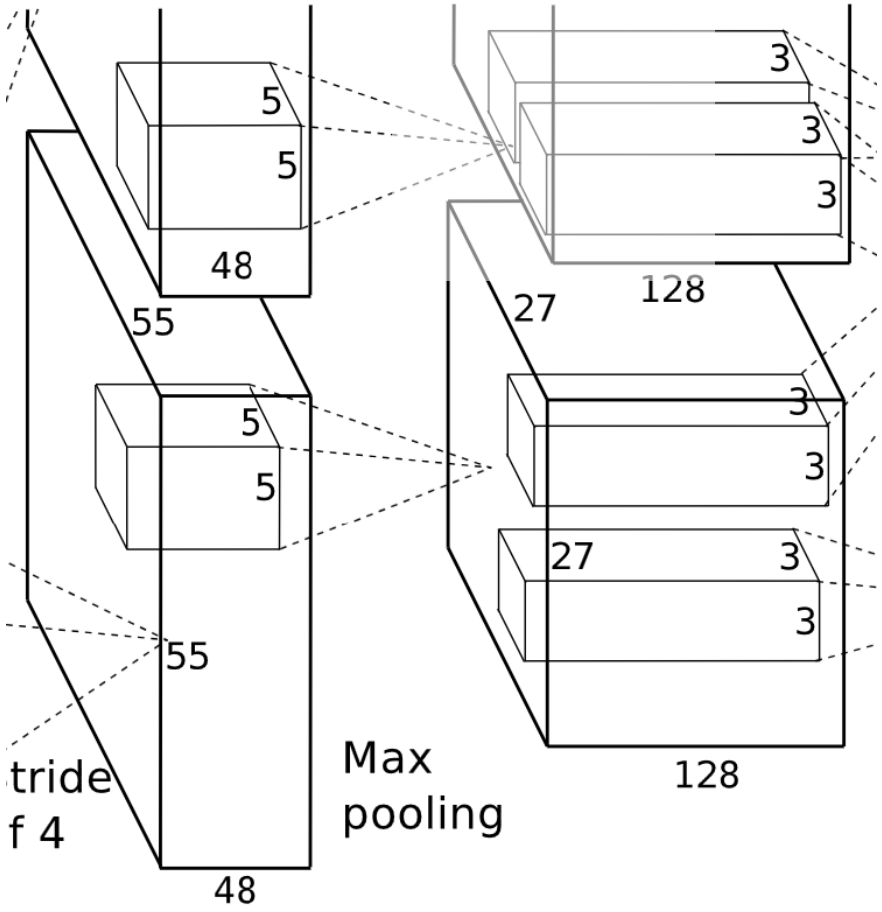


Stride = 2

What might be an issue with strided convolutions?

Foreshadowing...





Second layer

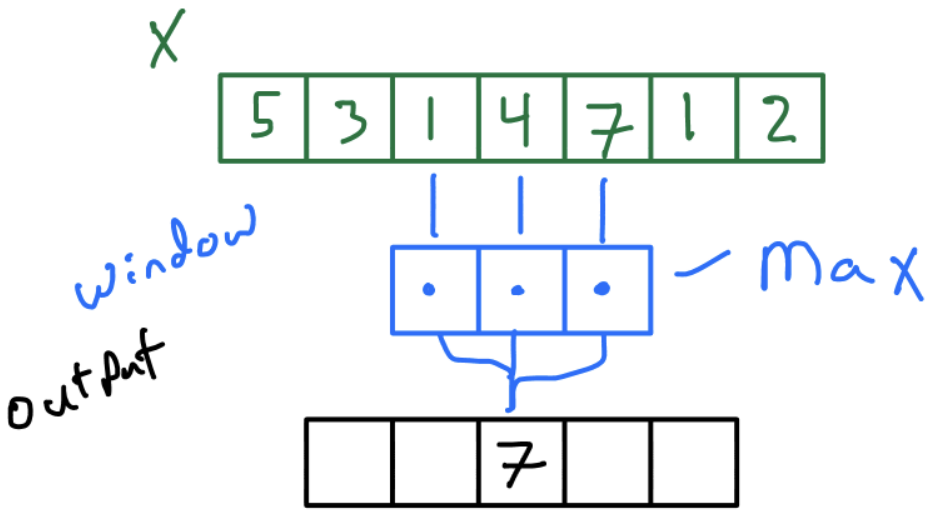
Questions:

- What size is the *kernel*?
- How many neurons are in this layer?
- *What is ”max pooling”?*
  - *How might we avoid the issues with strided convolutions we just identified?*

Pooling operator (1-D)

Inputs: x : Array of length d

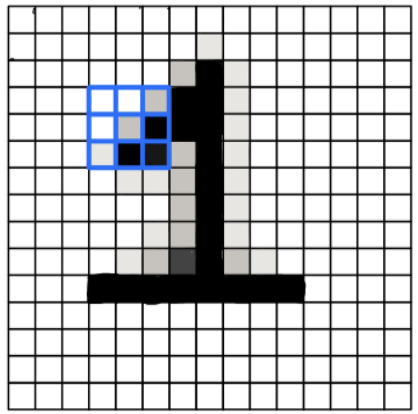
window size s



$7 = \max(1, 4, 7)$

- Take max output around each location before downsampling
- Make sure we don't miss any high activations
- Can also take Avg. (Average pooling)

Pooling in 2-D



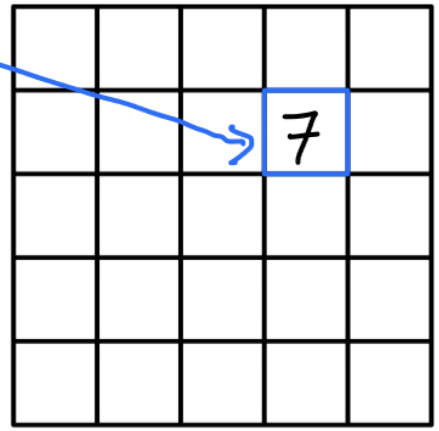
Align window in every 2-d location

$$\text{Max-Pool}(x)_{ij} = \max_{a=1}^{s}\left(\max_{b=1}^{s}\left(X_{i+a,j+b}\right)\right)$$

$$\text{Avg.-Pool}(x)_{ij} = \frac{1}{s^2}\sum_{a=1}^{s}\sum_{b=1}^{s} X_{i+a,j+b}$$

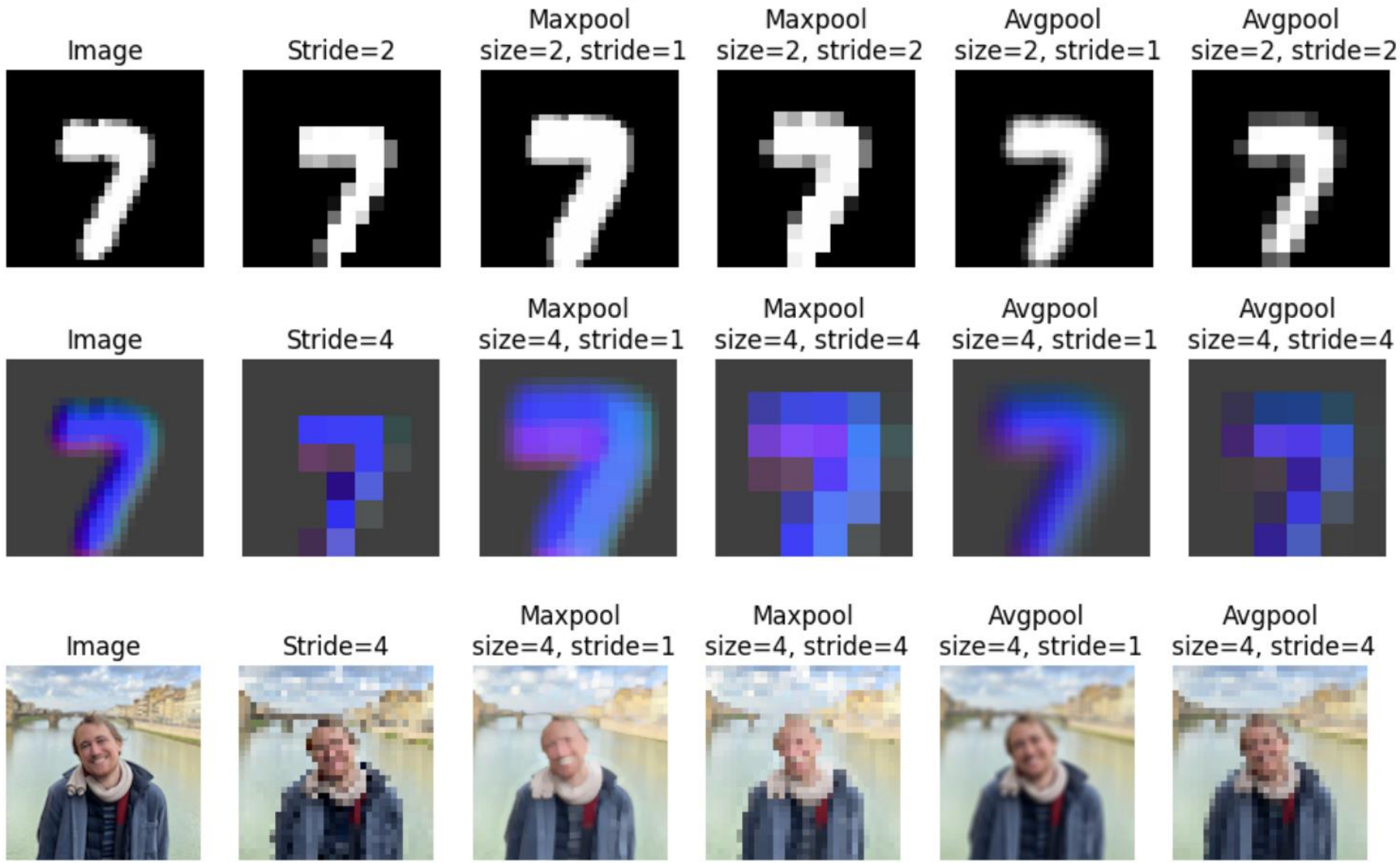
5	-1	3	8	4	6	-2
-3	4	9	1	1	7	-4
5	-6	3	2	1	-2	0
0	-8	5	5	4	-2	3
4	7	2	-7	3	1	4
4	-3	1	2	6	1	-1
0	0	3	-2	3	-1	0

max

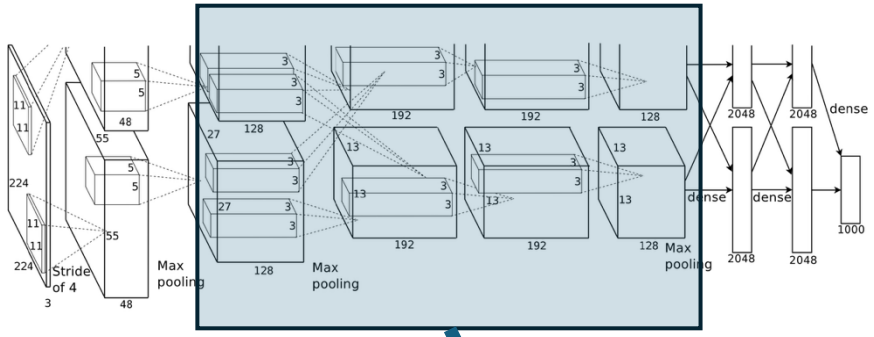


$$7 = \max(1, 1, 7, 2, 1, -2, 5, 4, -2)$$

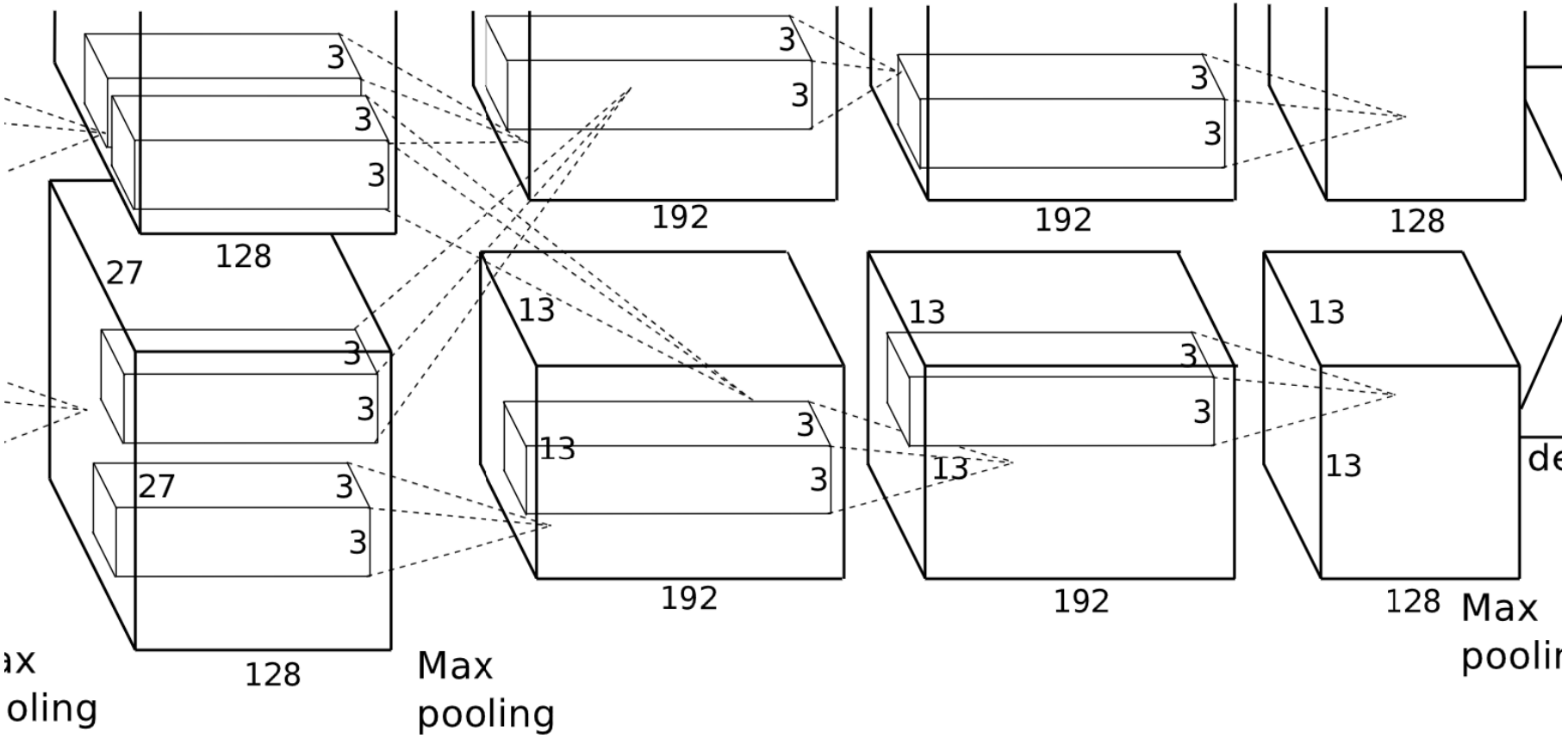
Down Sampling Comparison



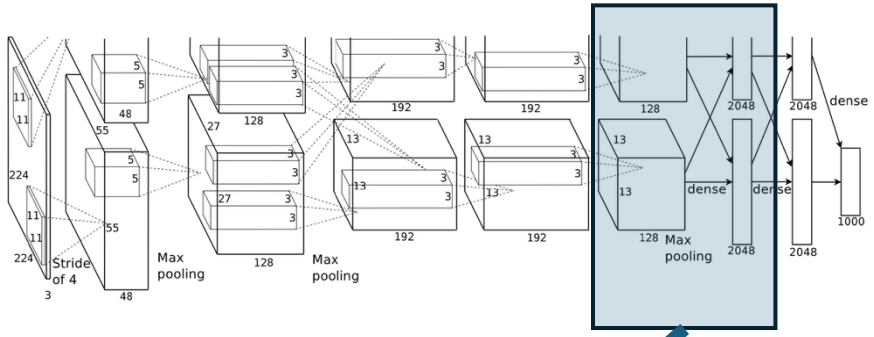
Layers 3-5

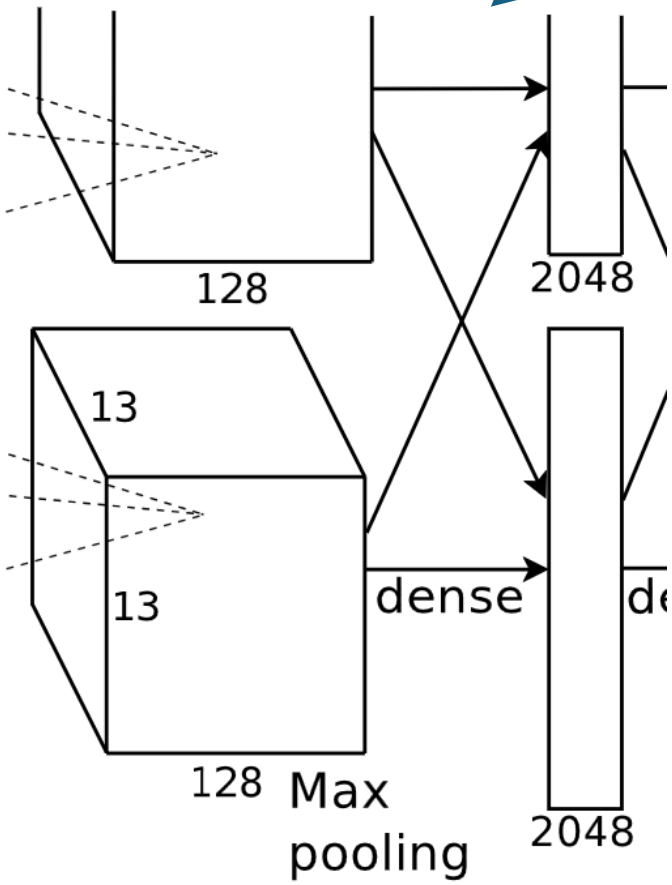


224
224
3
11
11
Stride of 4
55
55
48
48
5
5
Max pooling
27
27
128
128
3
3
Max pooling
13
13
192
192
192
192
128
128
Max pooling
dense
dense
dense
2048
2048
2048
2048
1000



3
3
3
128
27
3
3
27
3
3
128
Max pooling
192
192
128
13
13
192
13
13
192
13
13
128
Max pooling





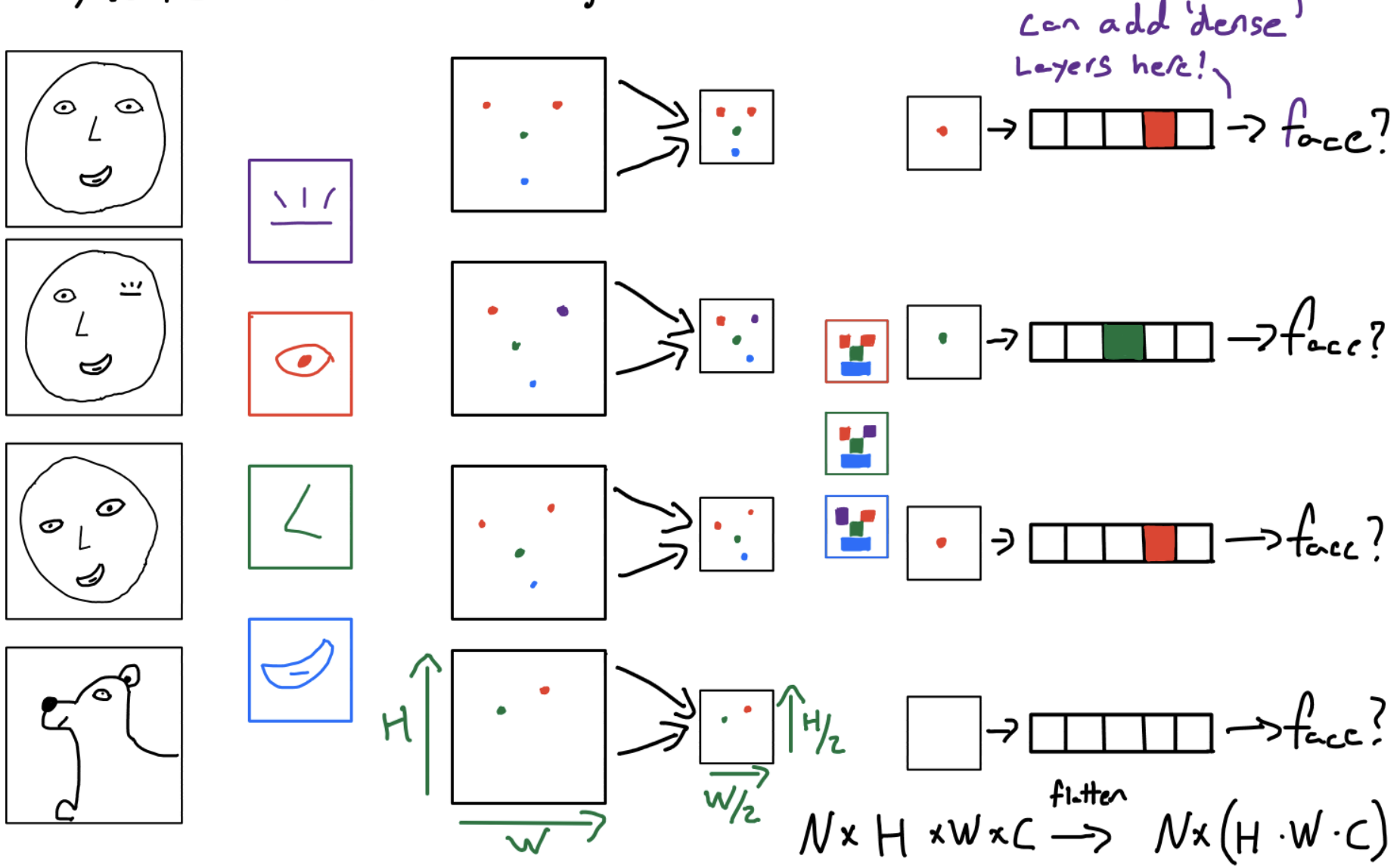
Layer 6

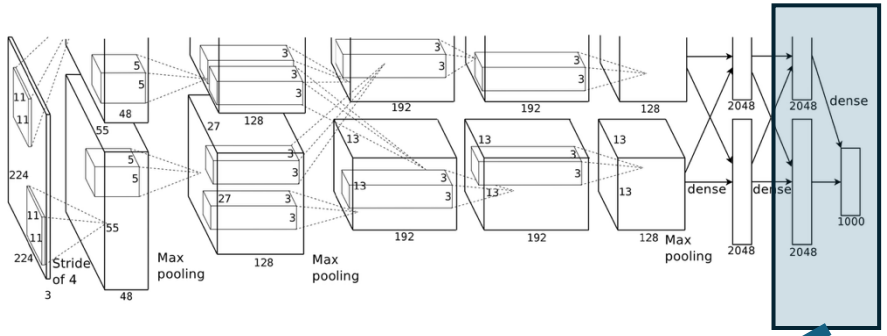
Questions:

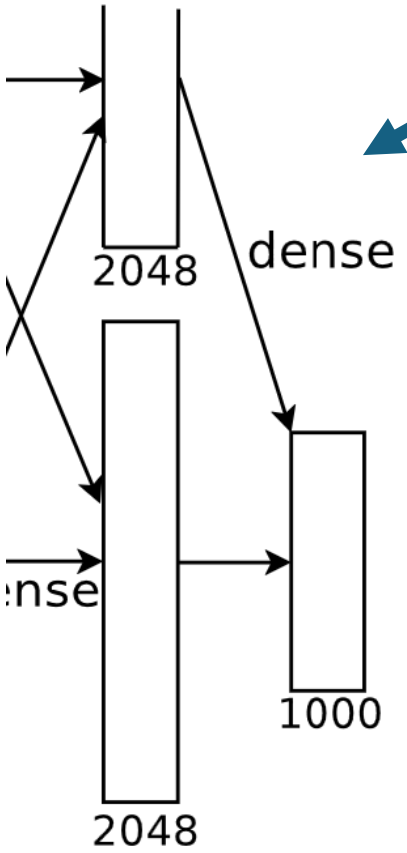
- What changed here? Why?
- What is meant by *dense*?
- How many neurons are in this layer?

Flattening

- After convolutions, flatten as before



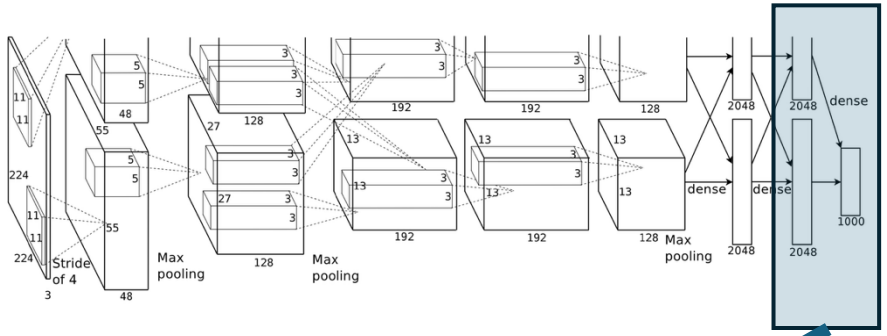


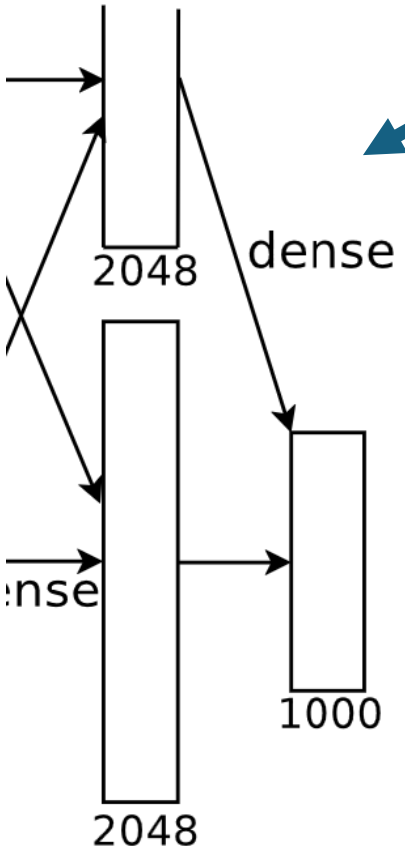


Layer 7-8

Questions:

- How many output neurons are there?
- Where does this number come from?

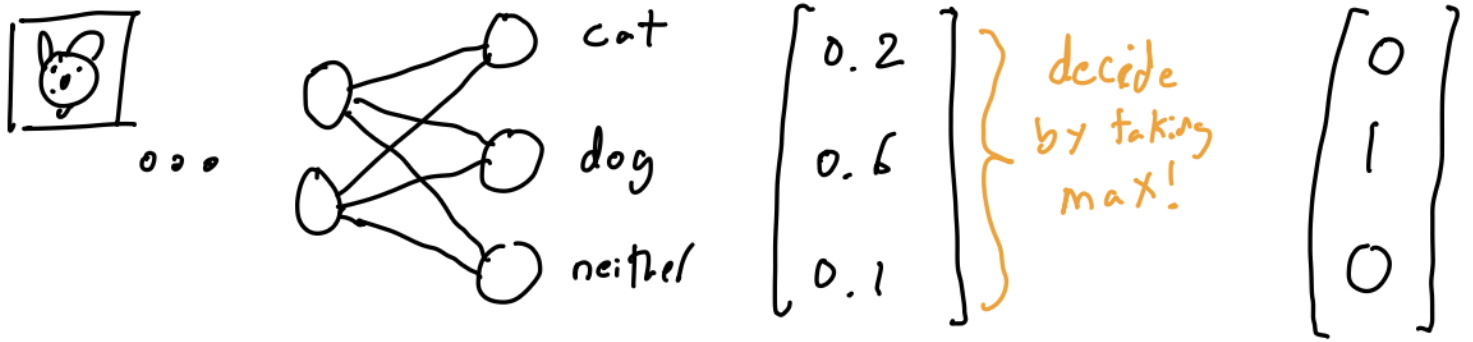




Layer 7-8

Recall (One-hot encoding):

Solution: Multiple outputs



Target (y) becomes a vector, 1 at index of correct response, 0 otherwise.

Applying neural networks in practice

Step 1: Identify our problem

What are our inputs? $[x_1, x_2 \ldots]$

- Should be a set of measurements about the state of the environment we want to react to.

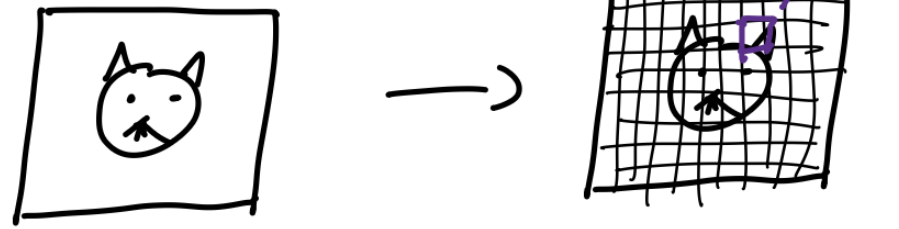
What is our Output? $[y]$

- Should be a decision we want to make based on our inputs

Example: Cat or dog?

Our agent should be able to decide if what its looking at is a cat or dog

Input: An Image



A set of pixel measurements, x_i is the brightness of pixel i

$$\begin{matrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{matrix}$$

[Note: could be a lot of inputs e.g. 256x256

Output: A Label

0 : Cat

1 : Dog

Step 2: Collect and process a Dataset

$$D: [(X_1, Y_1), (X_2, Y_2) \ldots (X_N, Y_N)]$$

Example: Find lots of images of cats and dogs

( , cat (0)), ( , dog (1)) ...

- Note: More data is generally better!

Tips for processing our collected data

1) The number of inputs for a network is fixed*, so examples need to be consistent

Example: Need to resize/crop images so they're all the same size
Don't mix color and greyscale images

2) We don't want too many inputs if they're not relevant

Example: Lowering image resolution overall can reduce irrelevant detail

3) Normalize inputs to a standard range (often [0,1] or [-1, 1]) $x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$

Example: Pixel values are often stored as 0-255

Convert to [0,1]: $x_i = \frac{x_i}{255}$

Step 3: Split the data into training and test sets

Dataset: $[(x_1, y_1), (x_2, y_2) \ldots (x_i, y_i)(x_{i+1}, y_{i+1}), \ldots (x_N, y_N)]$

Take 70-80% as "trainjng data"

Reserve 20-30% as "test data"

Goal: our test data will let us estimate how our model will perform in the wild.

— We won't use it to update the model so it will be "new"

Important: Always Shuffle your data before you split.

Example: If we collected our cats first, we might end up with a training set of all cats and a test set of all dogs :

Step 4: Train our network using back propagation

- We'll take multiple passes through our dataset ("Epochs"). We can stop early if we predict every action correctly

```
For epoch in NumEpochs:
    Ncorrect = 0 # Num. correct predictions this epoch
    For (x_i, y_i) in dataset: # Should also shuffle before new epoch
        ŷ = network.predict(x_i) # run forward pass
        network.backprop(y) # run backward pass
        if ŷ == y:
            Ncorrect += 1
    if Ncorrect == N: # Stop if we have 100% accuracy
        break
```

Lingo: Stochastic gradient descent

Step 6: Evaluate our network

Before we use our network, see how it does on our test data

```
Ncorrect = 0
for (xi, yi) in testDataset:
        ŷ = network.predict(xi)
        if ŷ == yi:
            Ncorrect += 1
accuracy = Ncorrect / N
```

What do we do if accuracy is not good?

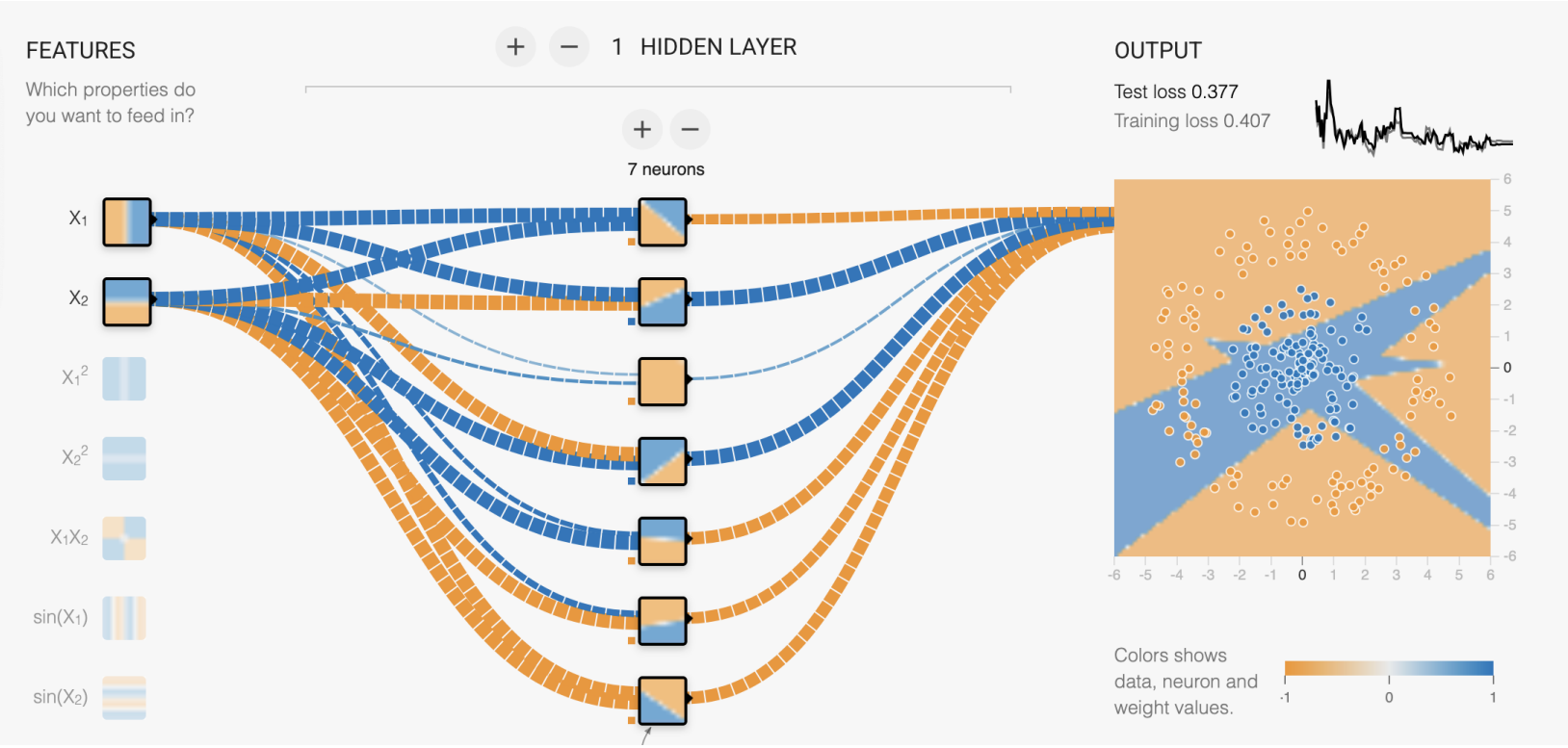
-> Change network structure

-> Change learning rate (γ)

-> Get more data

Then re-train and evaluate again

Diagnosing problems: Learning rate

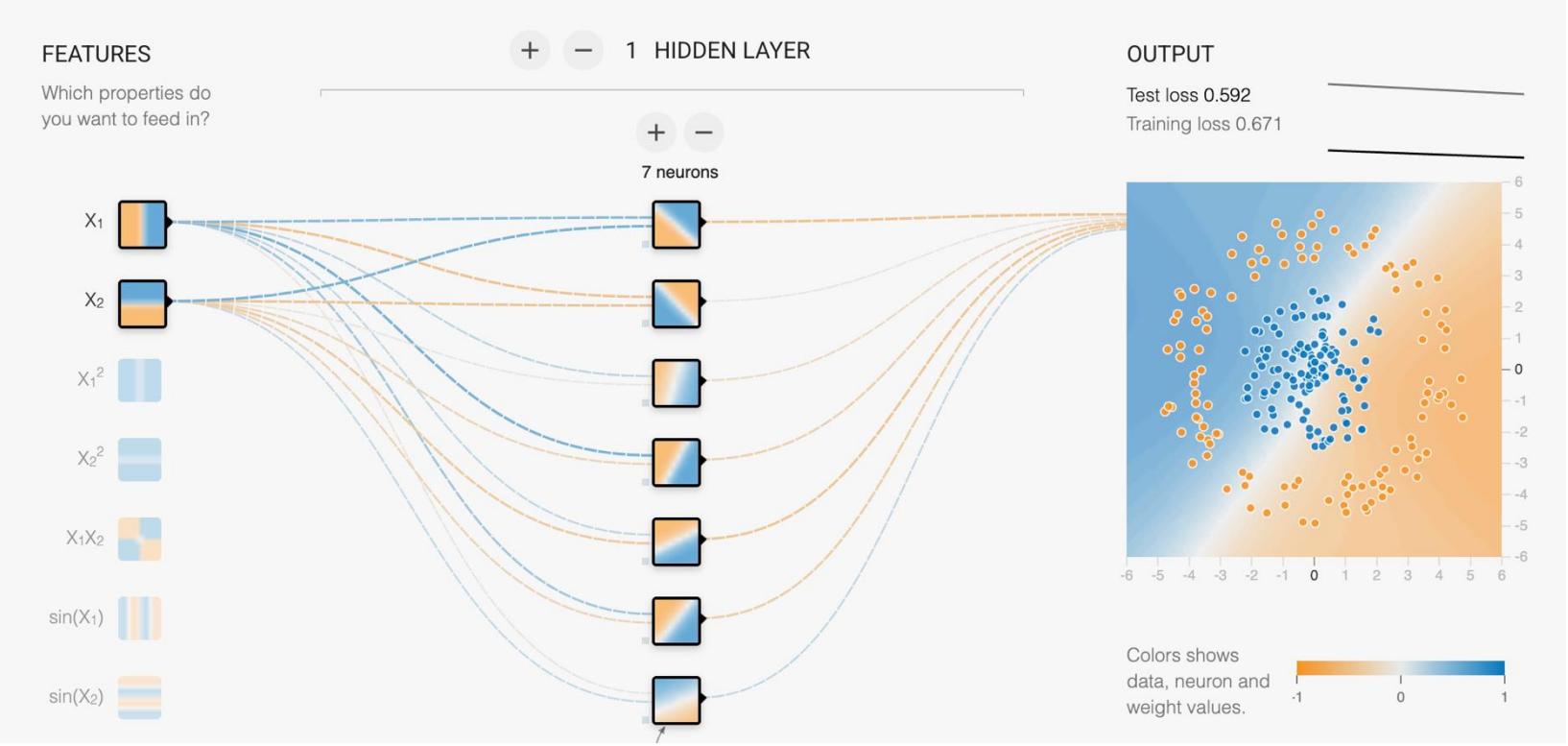


Watch loss as we train!

If it increases or oscillates:

- learning rate too high!

Diagnosing problems: Learning rate

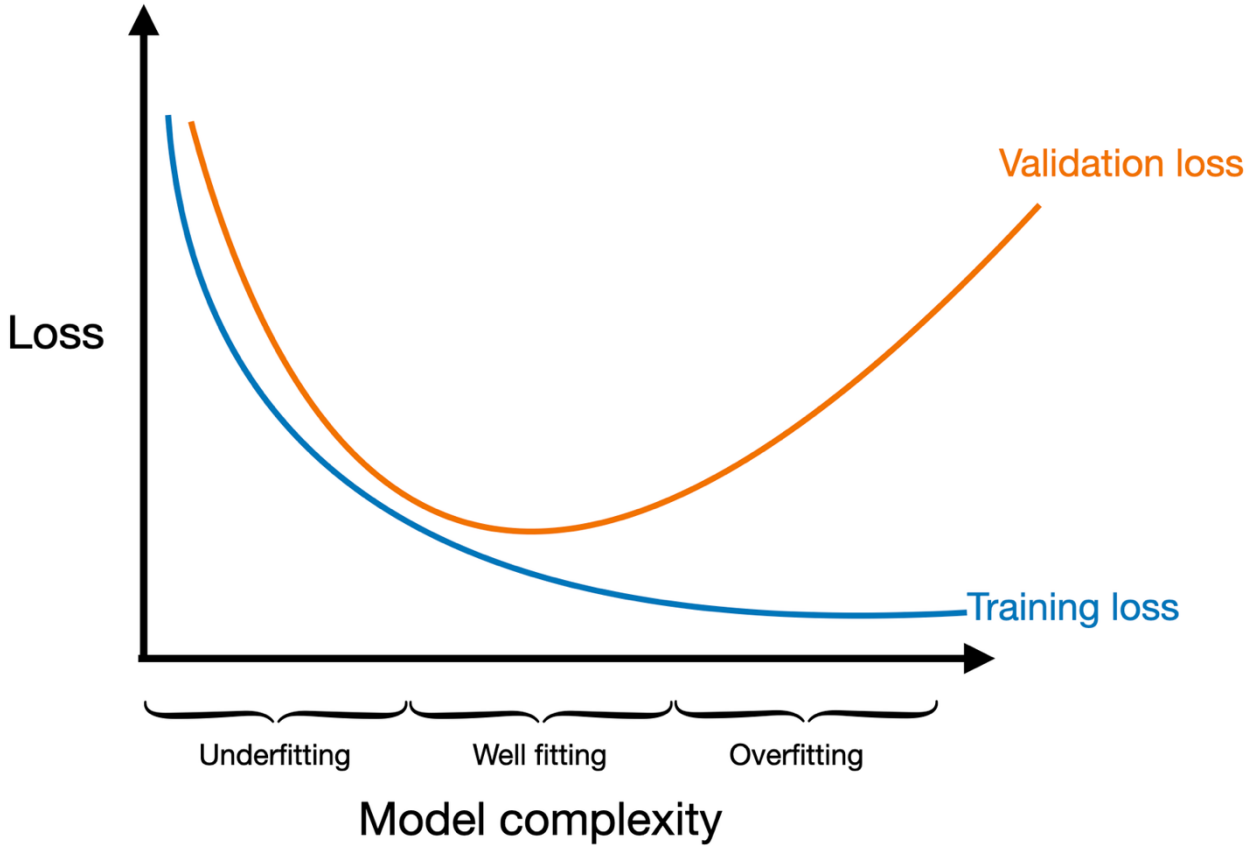


Watch loss as we train!

If it barely changes:

- learning rate too low!

Diagnosing problems: Overfitting and underfitting



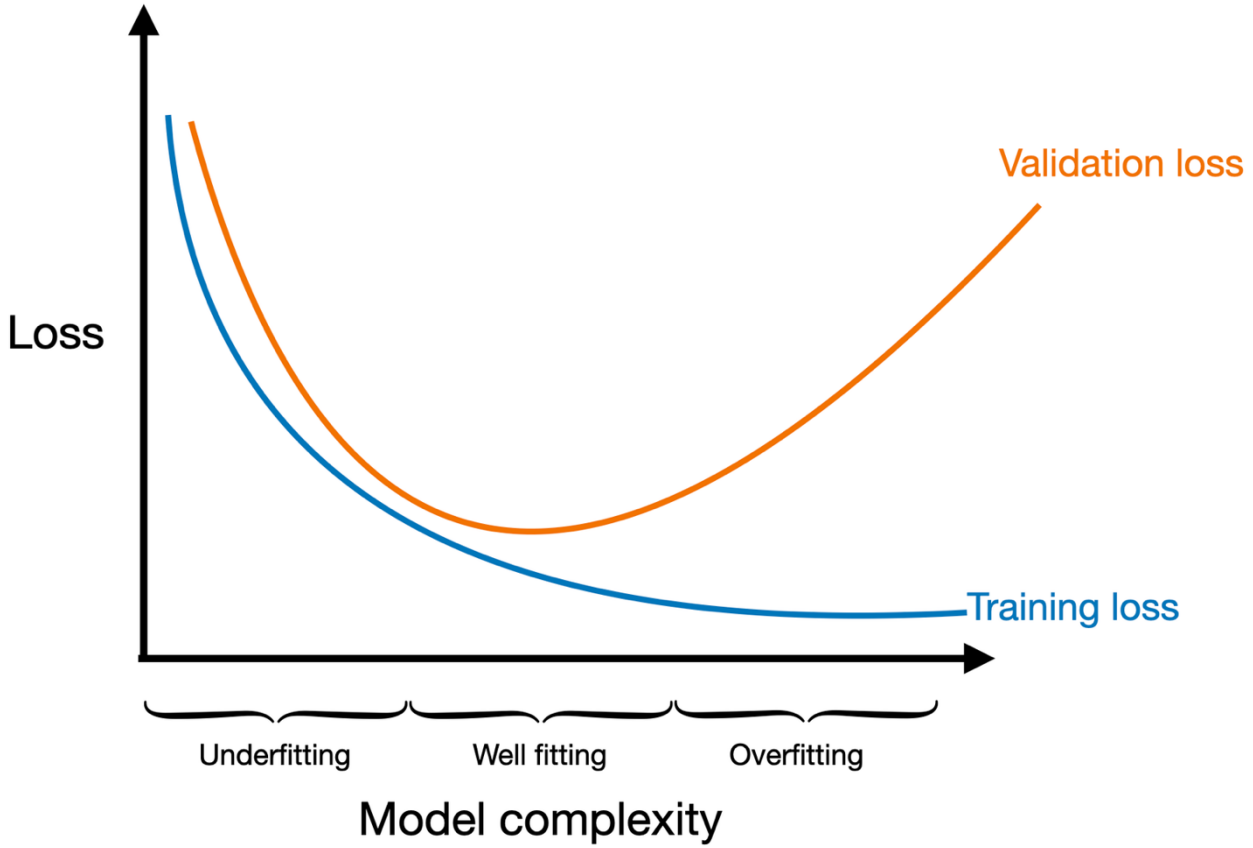
Compare loss/accuracy on training data to loss/accuracy on test data

(sometimes called "validation" data)

Both poor **(underfitting)**:

- Model not complex enough to capture data
- Add more neurons/layers!

Diagnosing problems: Overfitting and underfitting

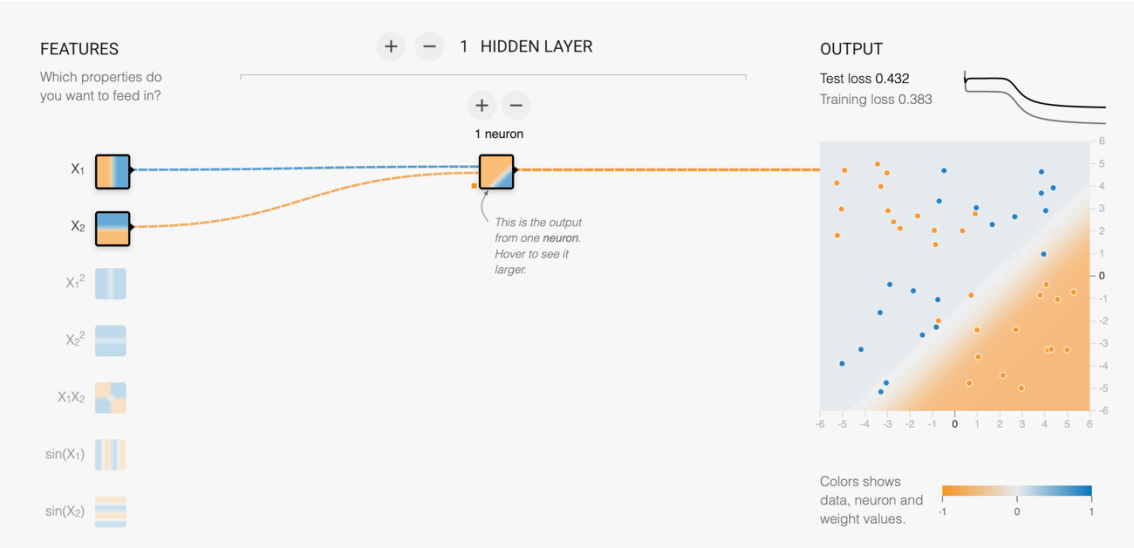


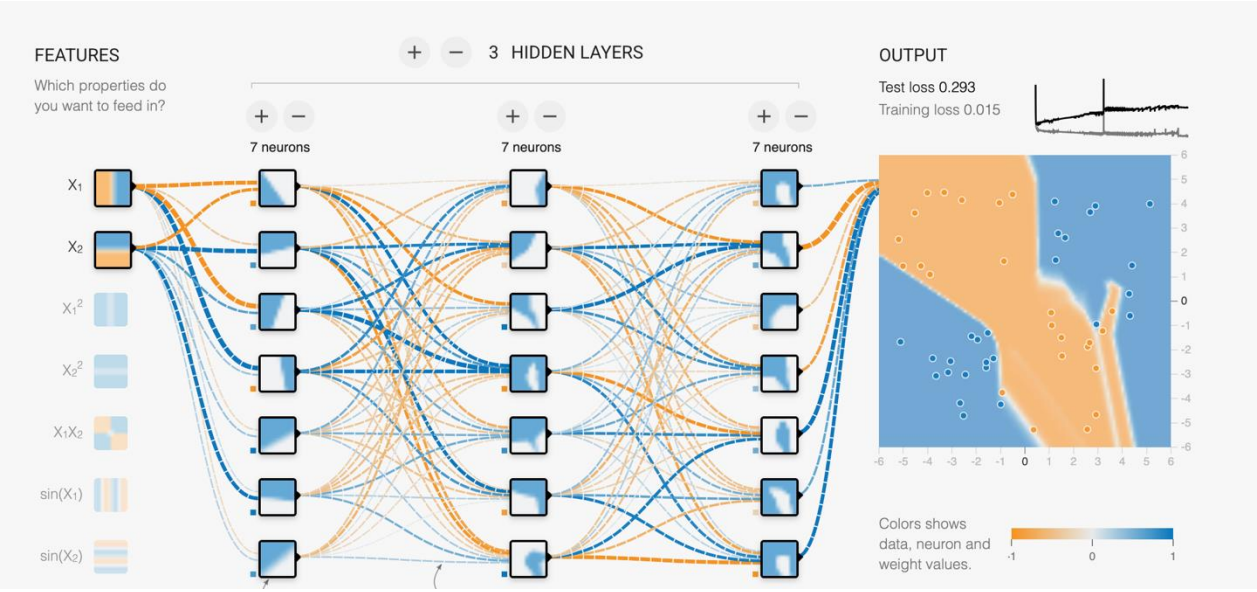
Compare loss/accuracy on training data to loss/accuracy on test data

(sometimes called "validation" data)

Only test poor **(overfitting)**:

- Model too complex, capturing not useful changes
- Reduce neurons/layers!





Underfitting:

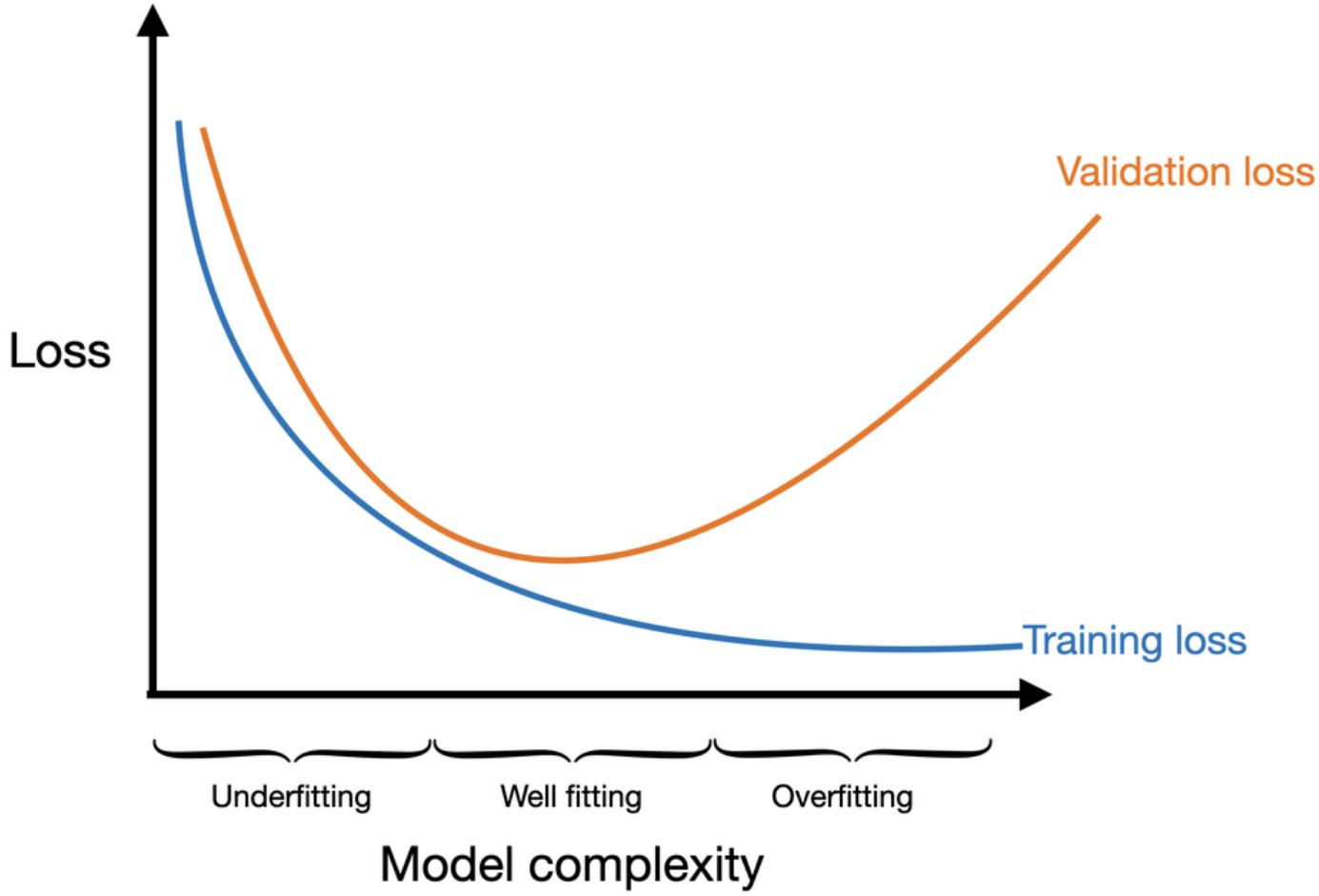
- Training and test loss/accuracy are poor

Overfitting:

- Training accuracy good, test is poor

Diagnosing problems: Overtraining





Early stopping

