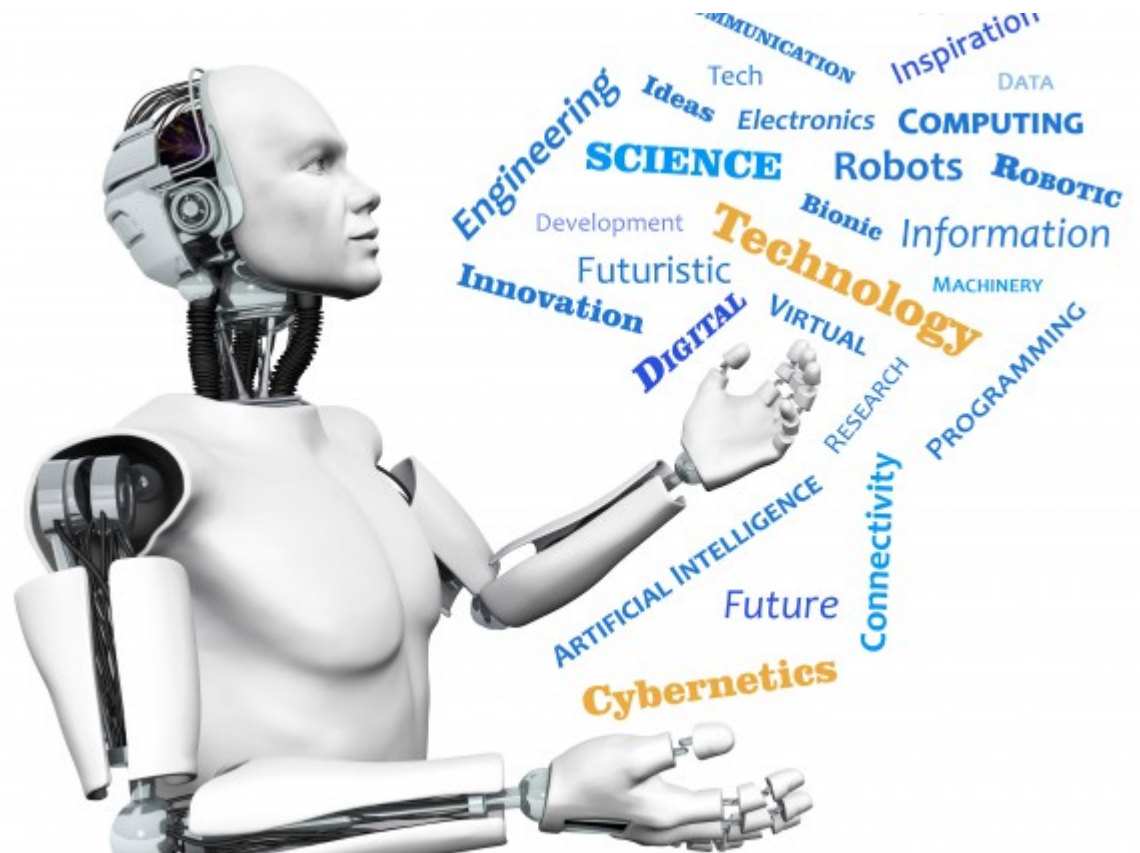


15-494/694: Cognitive Robotics

Dave Touretzky

Lecture 13:
Convolutional Neural Nets



TensorFlow Playground

Google's interactive backprop simulator.
<https://playground.tensorflow.org>

DATA

Which dataset do you want to use?



Ratio of training to test data: 50%



Noise: 0



Batch size: 10



REGENERATE

FEATURES

Which properties do you want to feed in?

- X_1
- X_2
- X_1^2
- X_2^2
- $X_1 X_2$
- $\sin(X_1)$
- $\sin(X_2)$

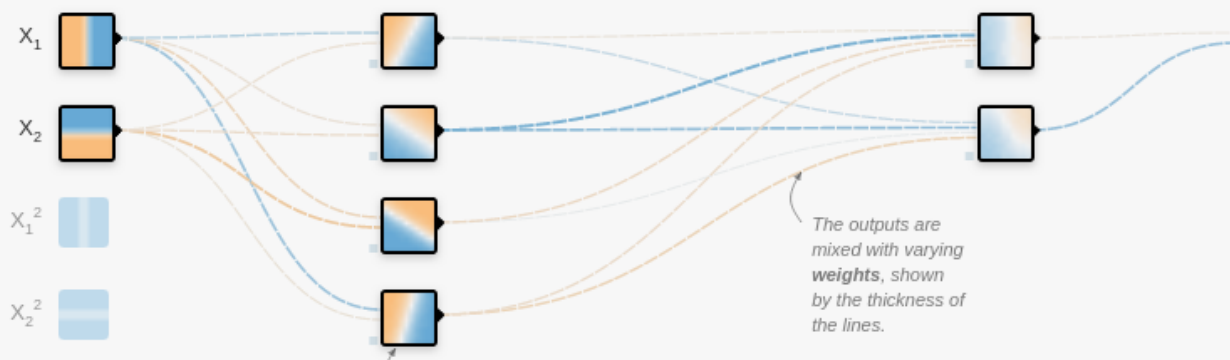
+ - 2 HIDDEN LAYERS

+ -

4 neurons

+ -

2 neurons



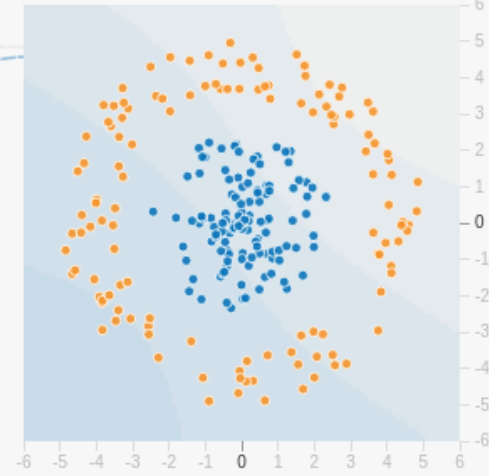
This is the output from one neuron. Hover to see it larger.

The outputs are mixed with varying weights, shown by the thickness of the lines.

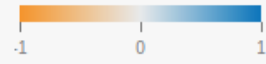
OUTPUT

Test loss 0.508

Training loss 0.508



Colors shows data, neuron and weight values.



Show test data

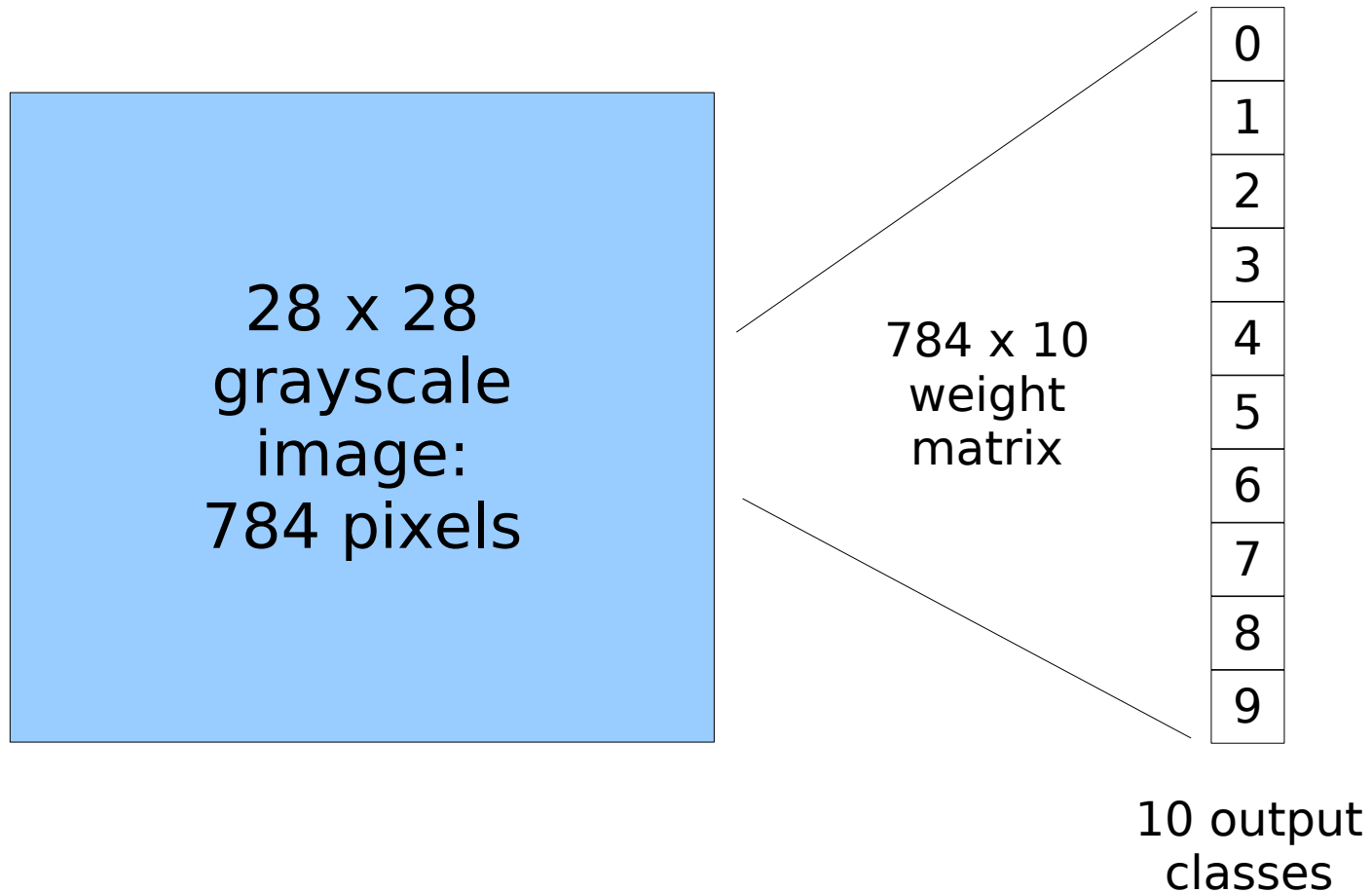
Discretize output

MNIST Dataset

- 60,000 labeled handwritten digits
- 28 x 28 pixel grayscale images



Recognition With a Linear Network



PyTorch

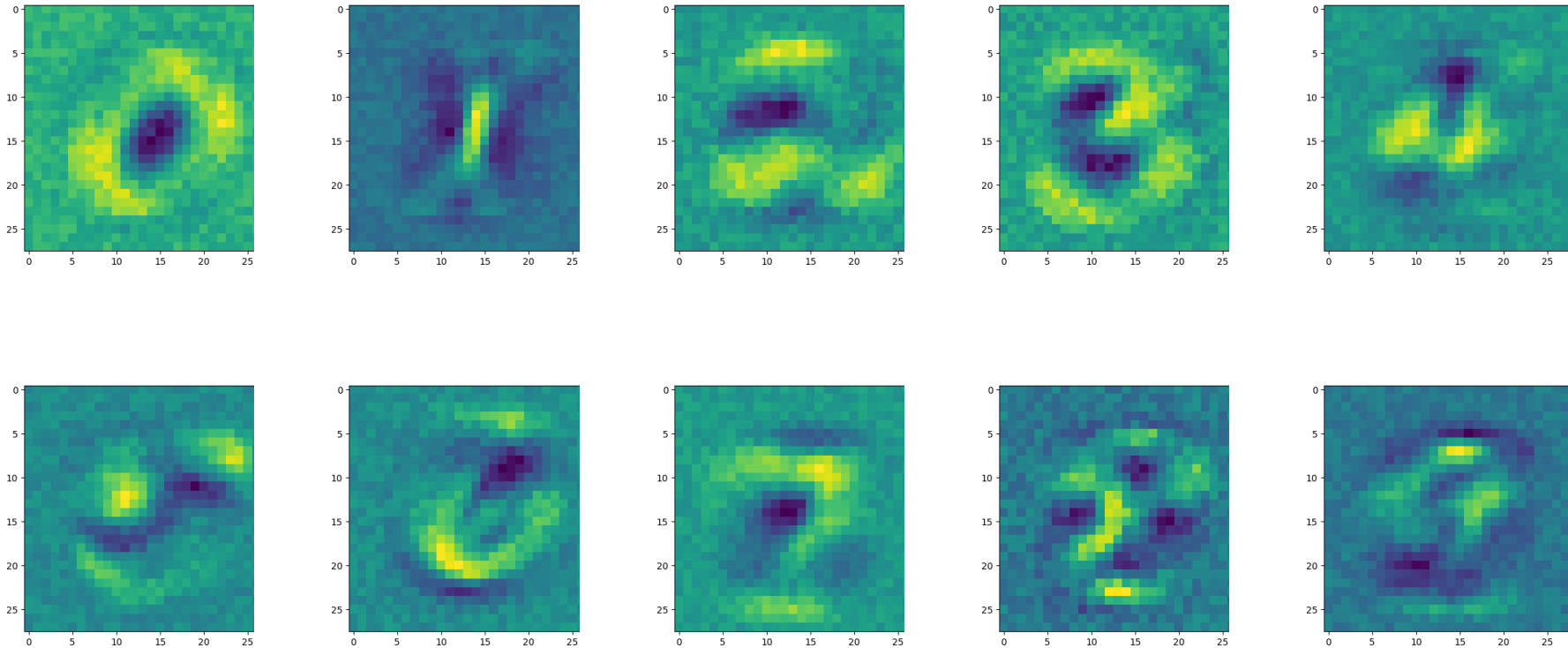
- Python package for tensor manipulation and vectorized computations, including neural net learning.
 - Replacement for numpy
 - Optimized for GPUs
- Tensors are multi-dimensional arrays, similar to numpy's ndarray structure.
- Code can run on either CPU or GPU.

Defining the Model mnist1

```
class MultiLogisticModel(nn.Module):  
  
    def __init__(self, in_dim, out_dim):  
        super(MultiLogisticModel, self).__init__()  
        self.linear = nn.Linear(in_dim, out_dim)  
  
    def forward(self, x):  
        out = self.linear(x)  
        return out
```

```
model = MultiLogisticModel(28*28, 10)
```

Learned Weights to Output Units

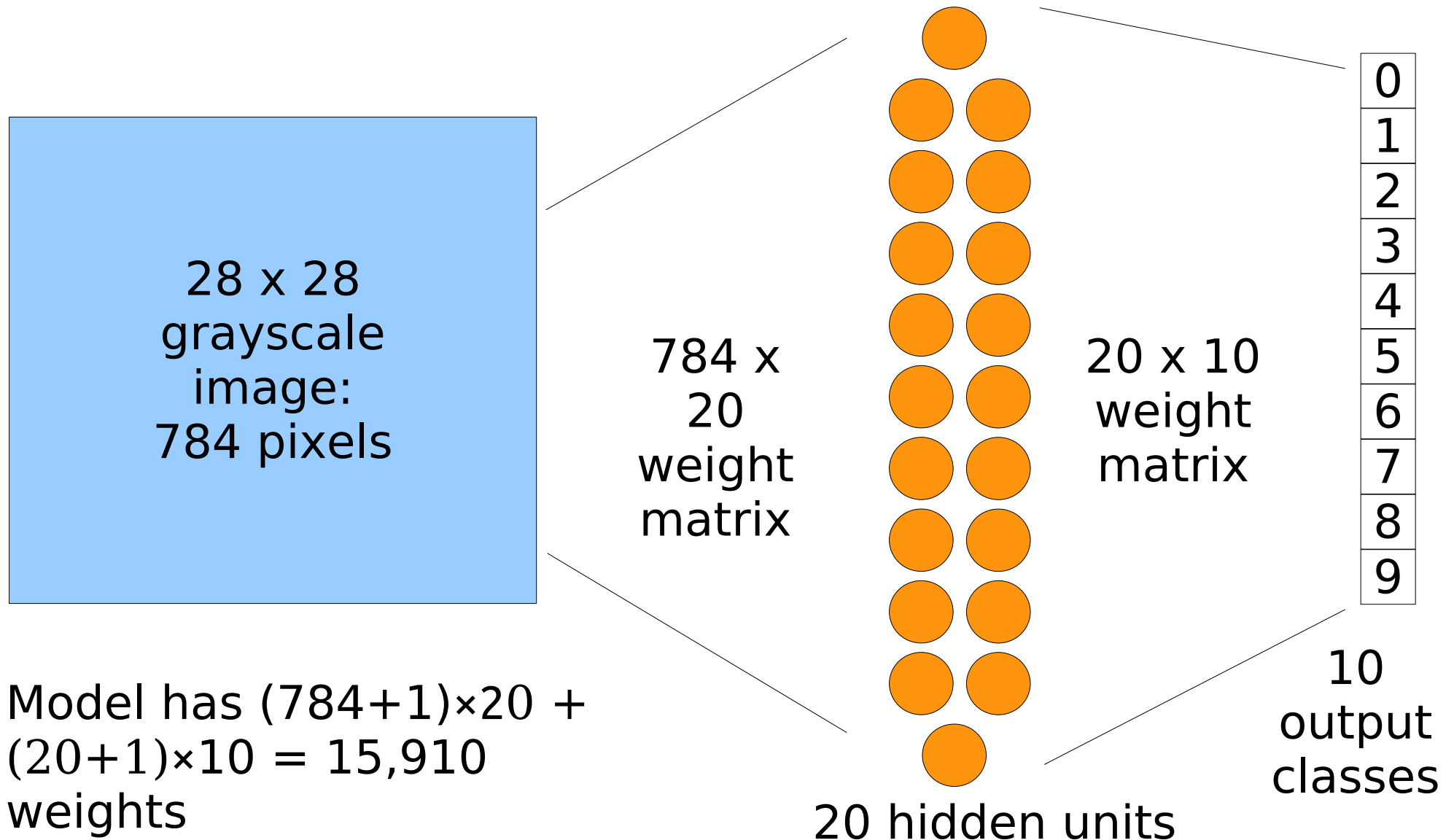


Training set performance: 89% correct.

Batch Size

- An *epoch* is one pass through all the training data.
- With a large training set (60,000 images), we don't need to see all the training examples in order to estimate the error gradient.
- We set a batch size of 100 to indicate we want to do a weight update after every 100 training examples.
 - The examples need to be mixed together.
 - What if we trained on all the 2's first?

Adding A Hidden Layer



Batch Normalization

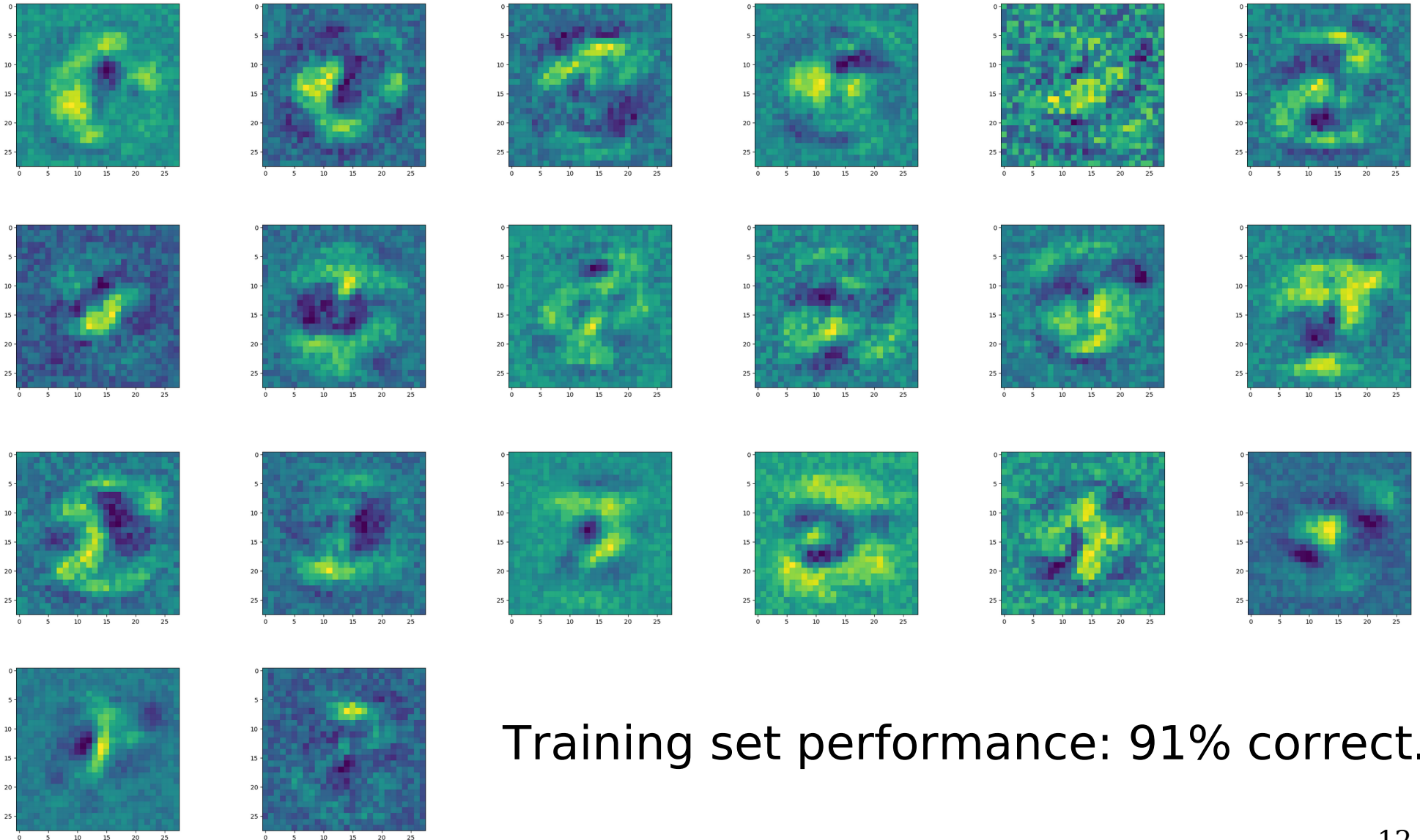
- We want the activity patterns in each layer to have nice statistical properties (mean and variance) because this helps speed up learning.
- But each weight update changes the statistical distribution.
- Solution: “batch normalization”, a trick for making the distributions more uniform.
- Built in to PyTorch.

Defining the Model mnist2

```
class OneHiddenLayer(nn.Module):  
  
    def __init__(self, in_dim, out_dim, nhiddens):  
        super(OneHiddenLayer, self).__init__()  
        self.network = nn.Sequential(  
            nn.Linear(in_dim, nhiddens),  
            nn.BatchNorm1d(nhiddens),  
            nn.ReLU(),  
            nn.Linear(nhiddens, out_dim)  
        )  
  
    def forward(self, x):  
        out = self.network(x)  
        return out
```

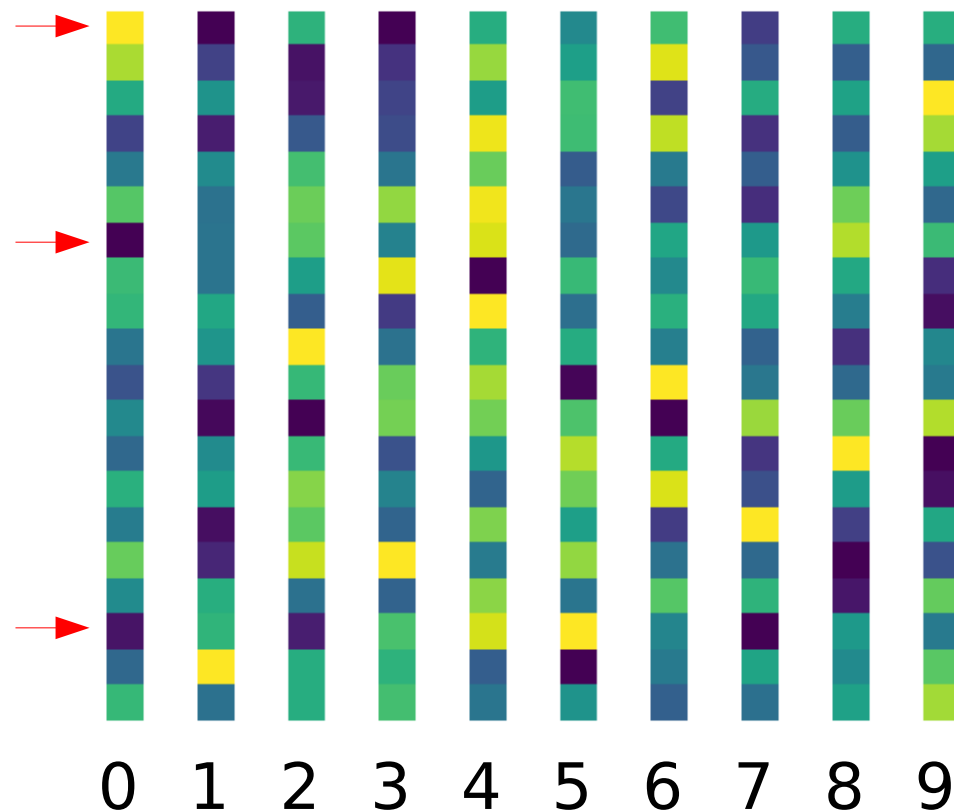
```
model = OneHiddenLayer(28*28, 10, 20)
```

Learned Weights to Hidden Units



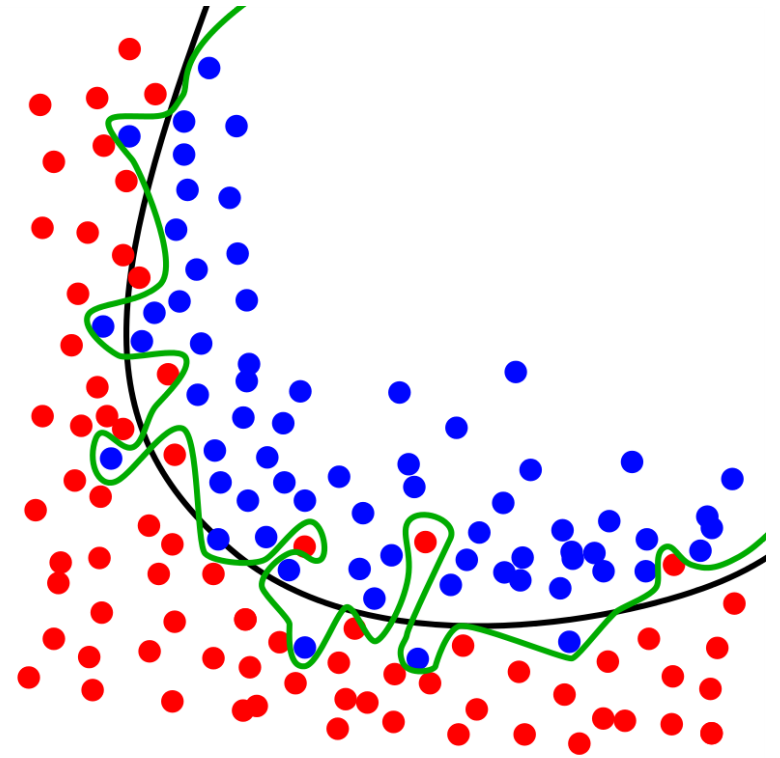
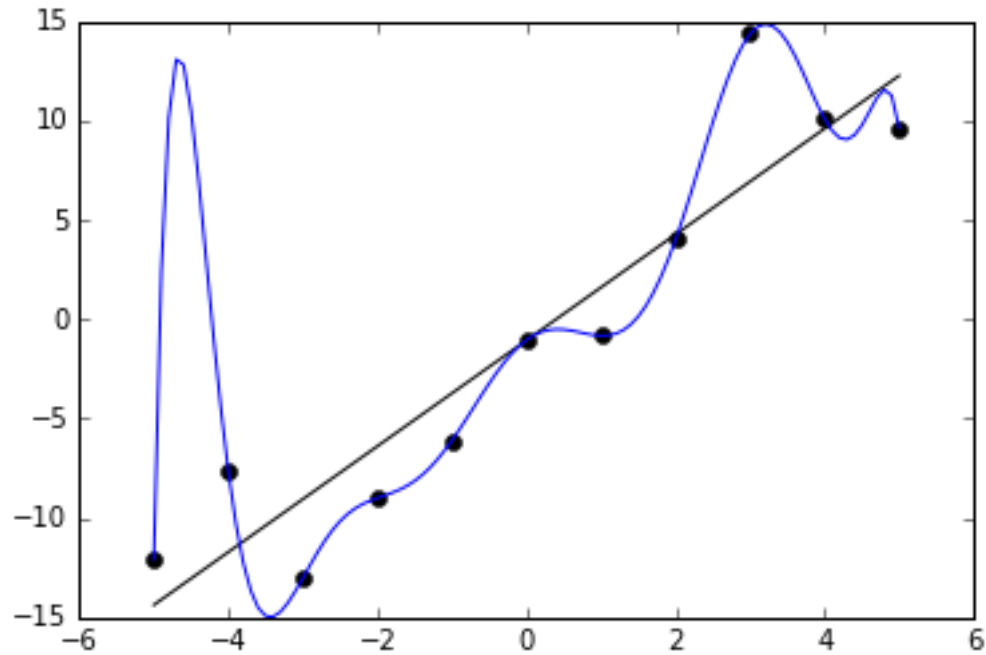
Training set performance: 91% correct.

Learned Weights to Output Units



Training set performance: 91% correct.

Overfitting

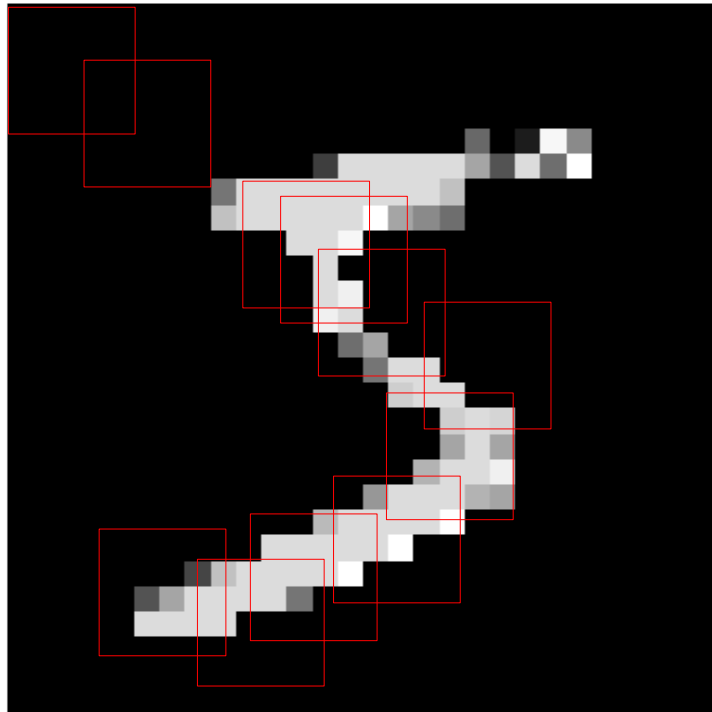


How to Avoid Overfitting

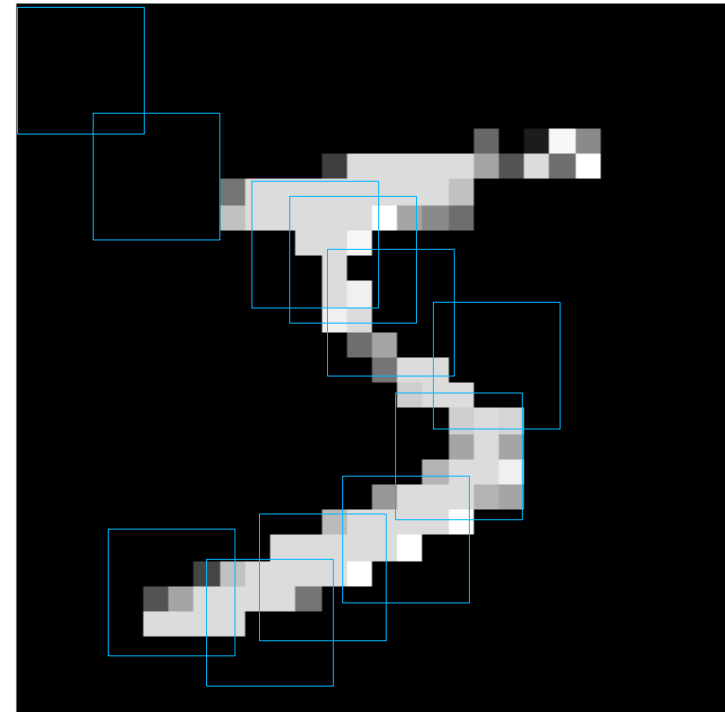
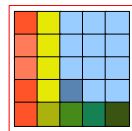
- Increase the size of the training set.
- Reduce the number of parameters:
 - Fewer hidden units
 - Shared weights (convolutional network)
- Regularization: penalize large weights to encourage making more weights be zero.
- Dropout: randomly disable some fraction of the connections on every iteration.
- Early stopping:
 - Maintain a separate cross-validation set
 - Stop training when the CV error rises

Convolutional Neural Networks

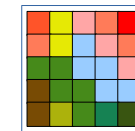
- Learn small (3x3 or 5x5) feature detectors or *kernels* that can be applied anywhere in the image.



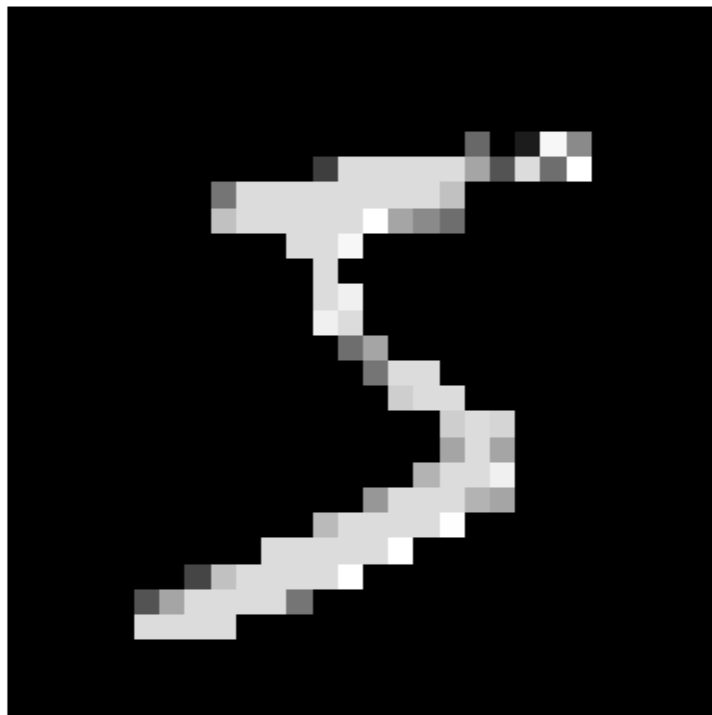
Feature 1:



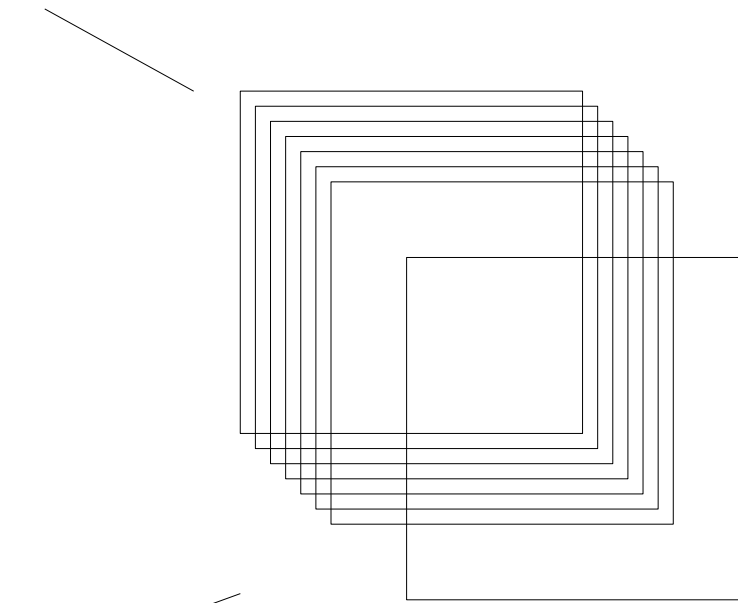
Feature 2:



Feature Maps



28 x 28 image



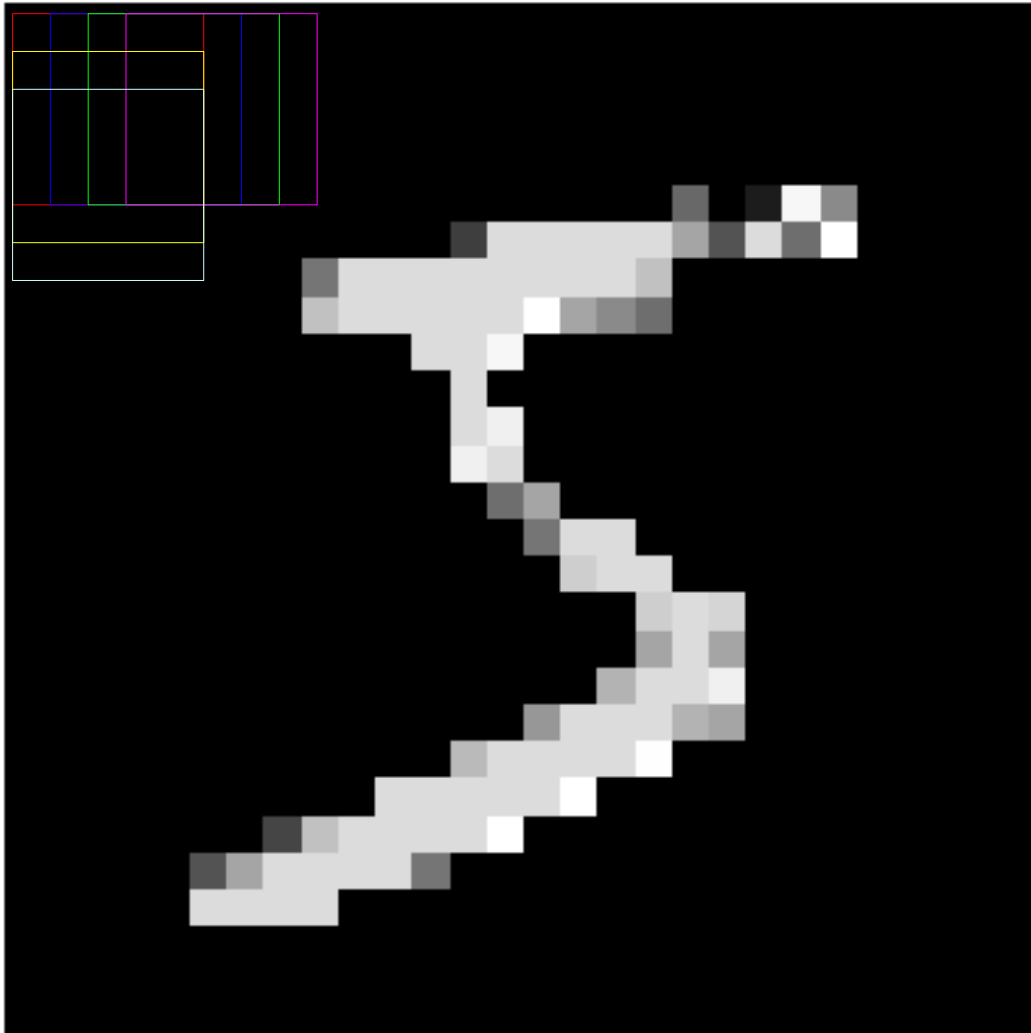
32 feature maps
26 x 26

32 kernels
5x5 pixels
stride 1
padding 1

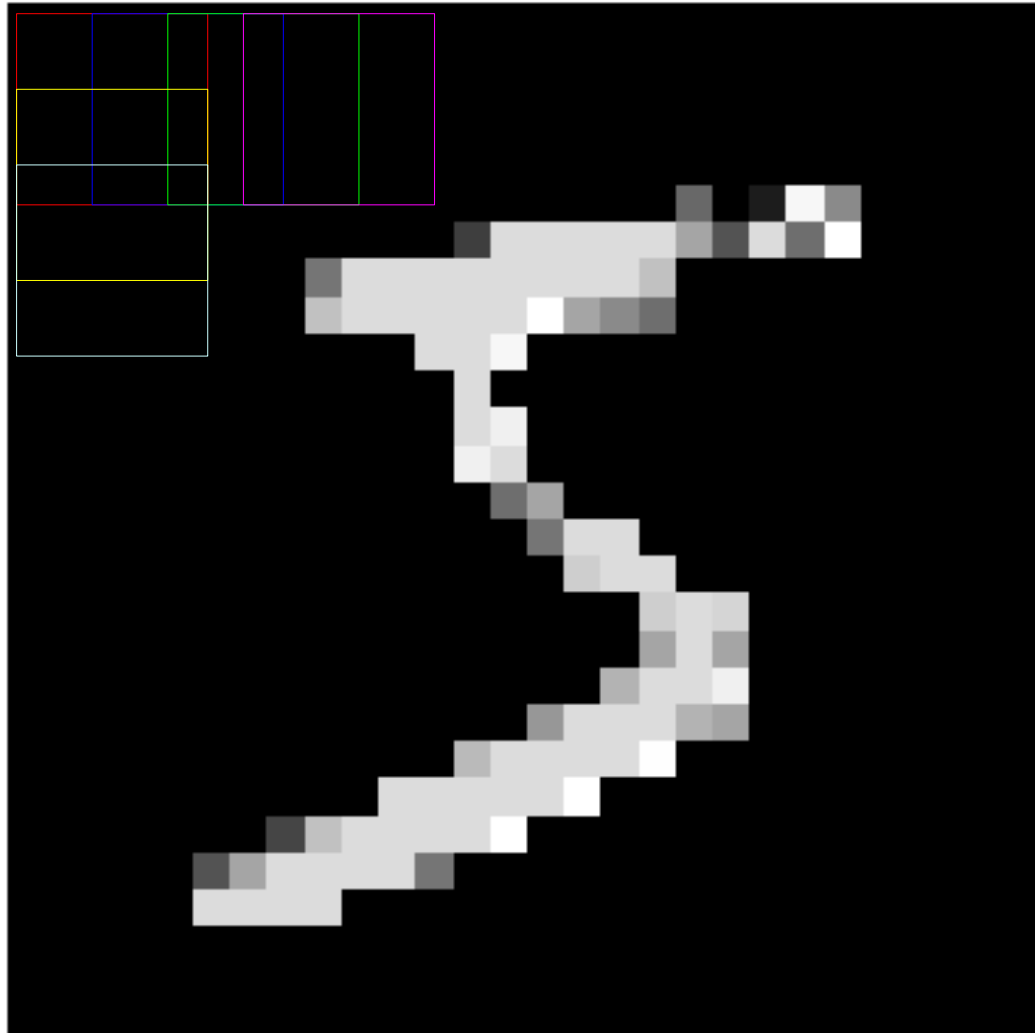
weights = $32 \times (5 \times 5 + 1) = 832$ (small!)

connections = $32 \times (26 \times 26) \times (5 \times 5 + 1) = 562,432$

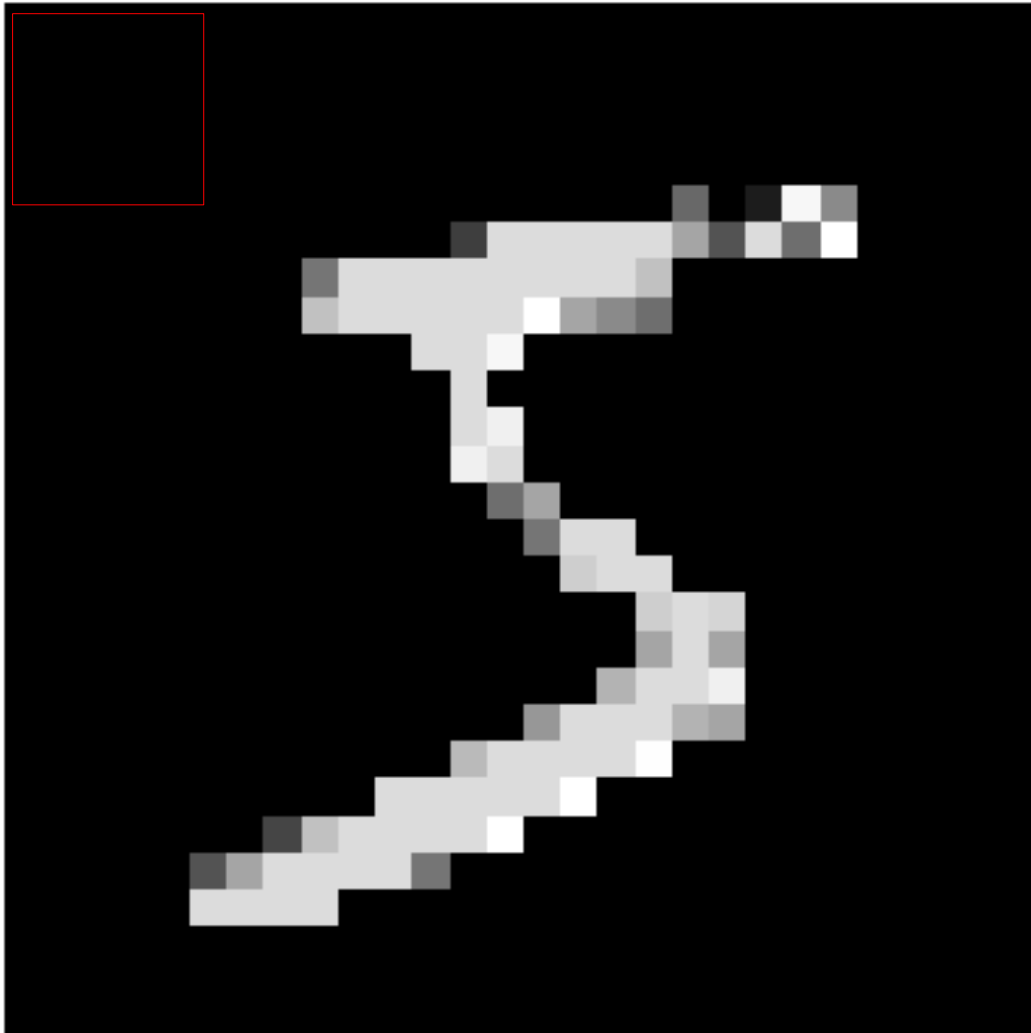
Stride 1



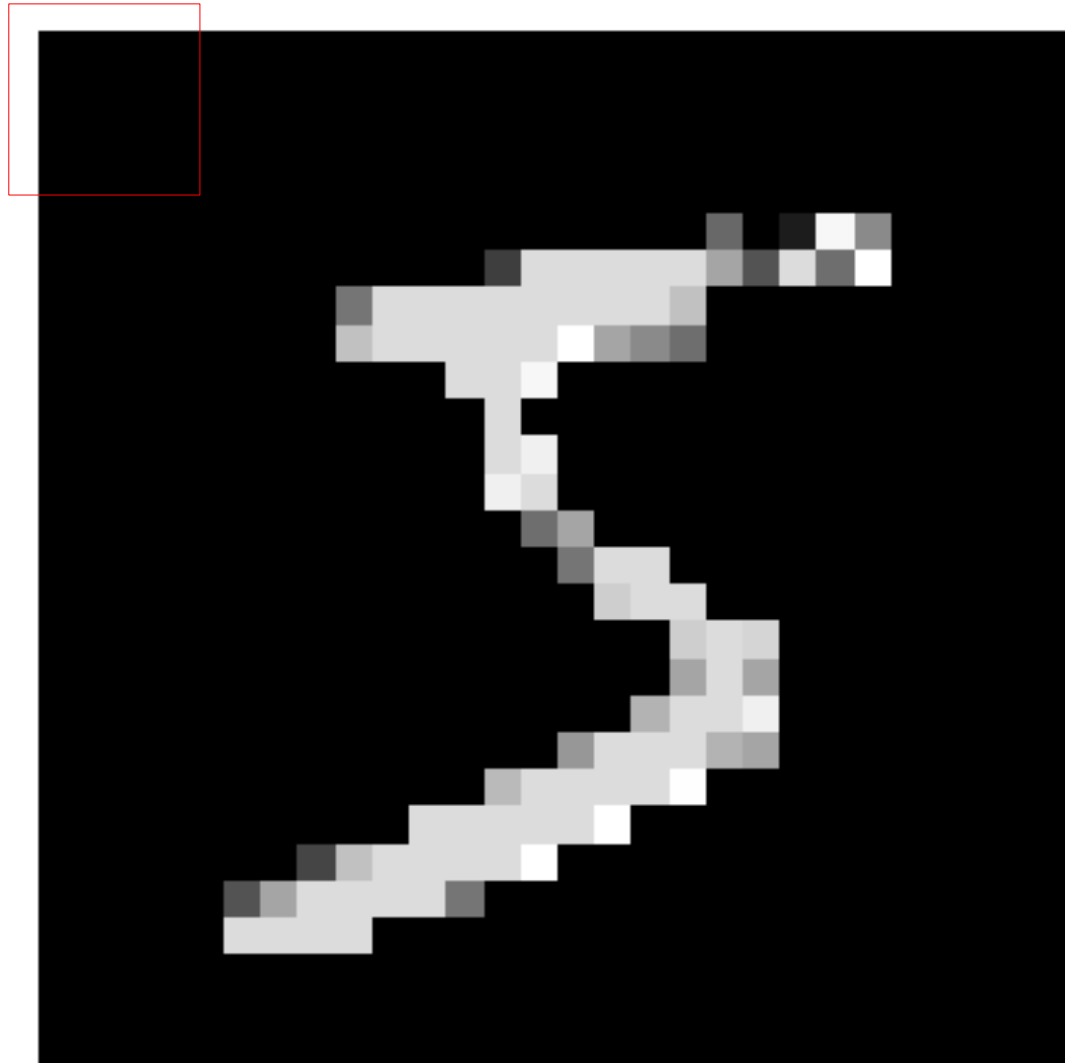
Stride 2



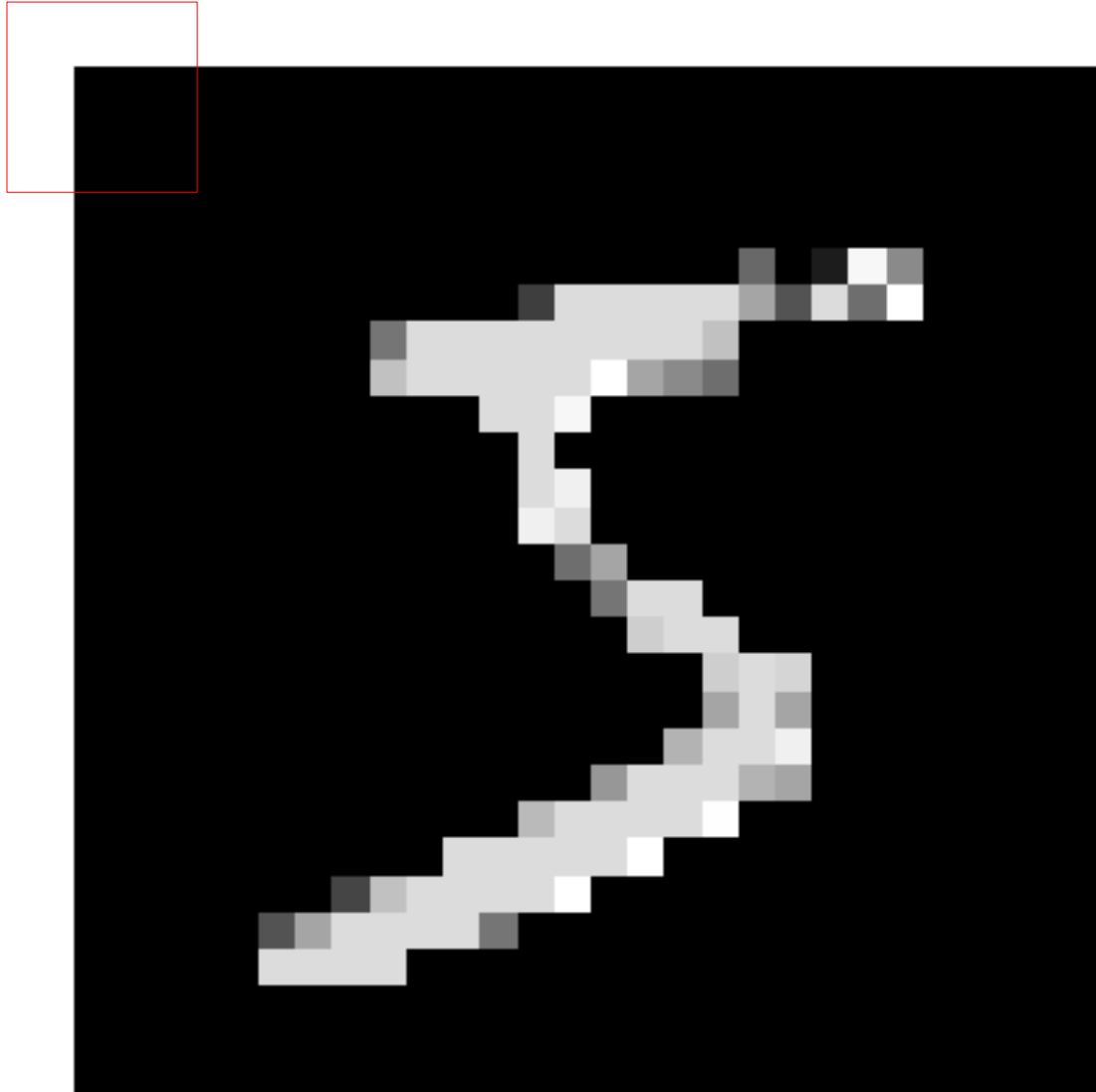
First Kernel: Padding 0



First Kernel: Padding 1

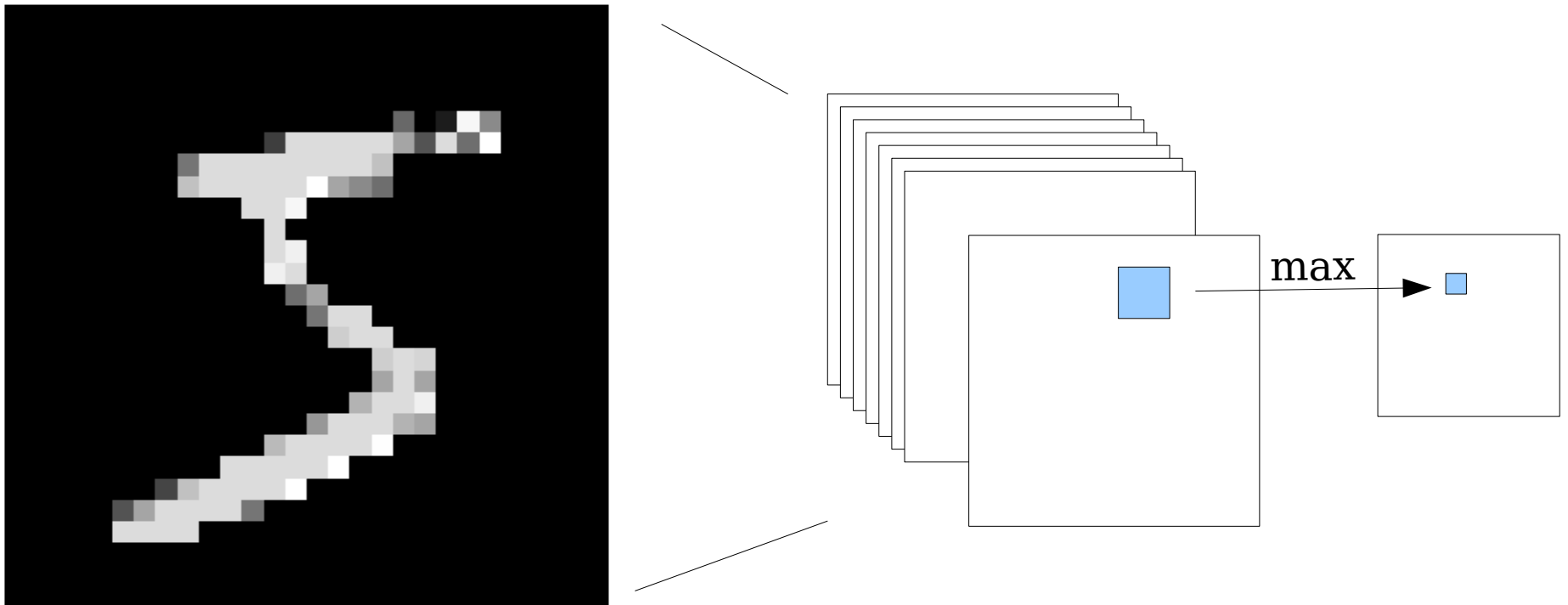


First Kernel: Padding 2



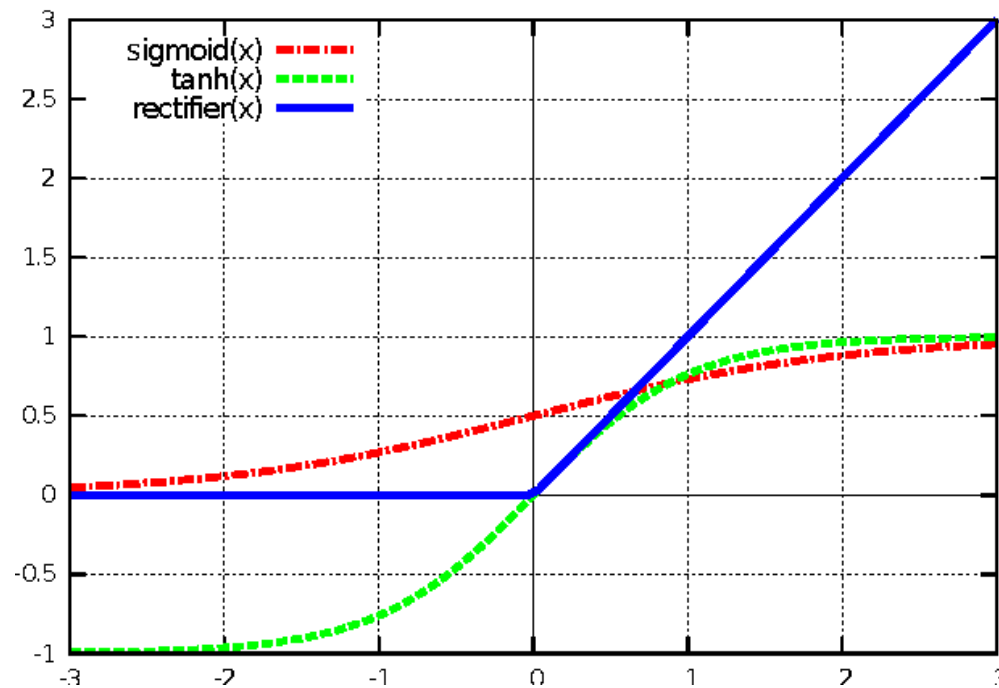
Max Pooling

- We might not care exactly where a feature appears in the image.
- Downsampling by max pooling reduces the number of units and connections.



Choice of Activation Function

- Sigmoid and tanh were popular early on:



- Now it's more common to use ReLU:
Rectified Linear Unit. $g(x) = \max(x, 0)$
 - Derivative doesn't go to zero for large x.

Choice of Loss Function

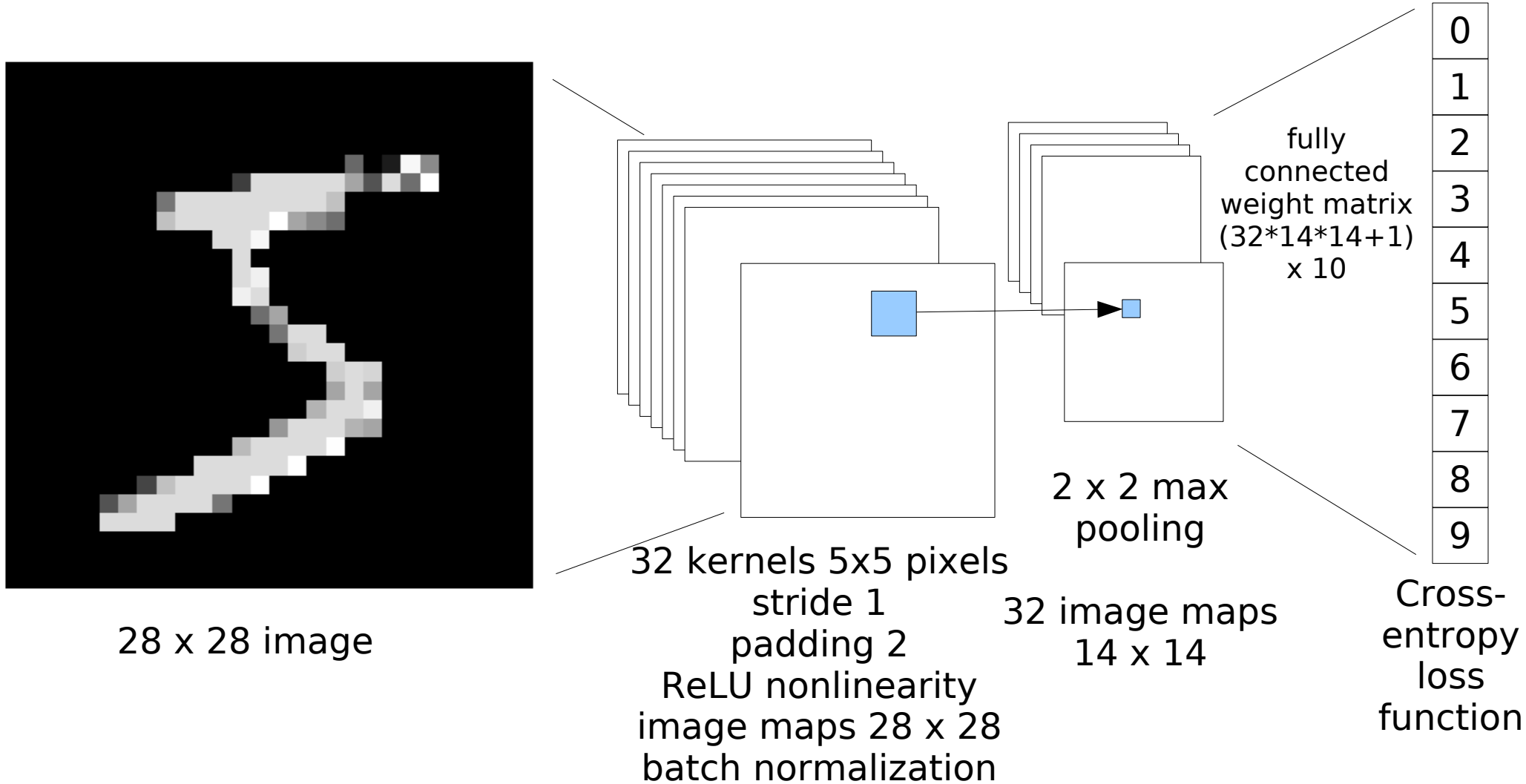
- Mean Squared Error is a general loss function but not always the best to use.

$$E = \frac{1}{2P} \sum_p (d^p - y^p)^2$$

- If desired outputs are probabilities (values between 0 and 1), use cross-entropy instead. Heavily penalizes really wrong outputs.

$$E = \sum_p -d^p \log(y^p) - (1 - d^p) \log(1 - y^p)$$

MNIST With A CNN



parameters = 63,626
How many connections?

Accuracy on training set: 98.7%

Defining the Model mnist3

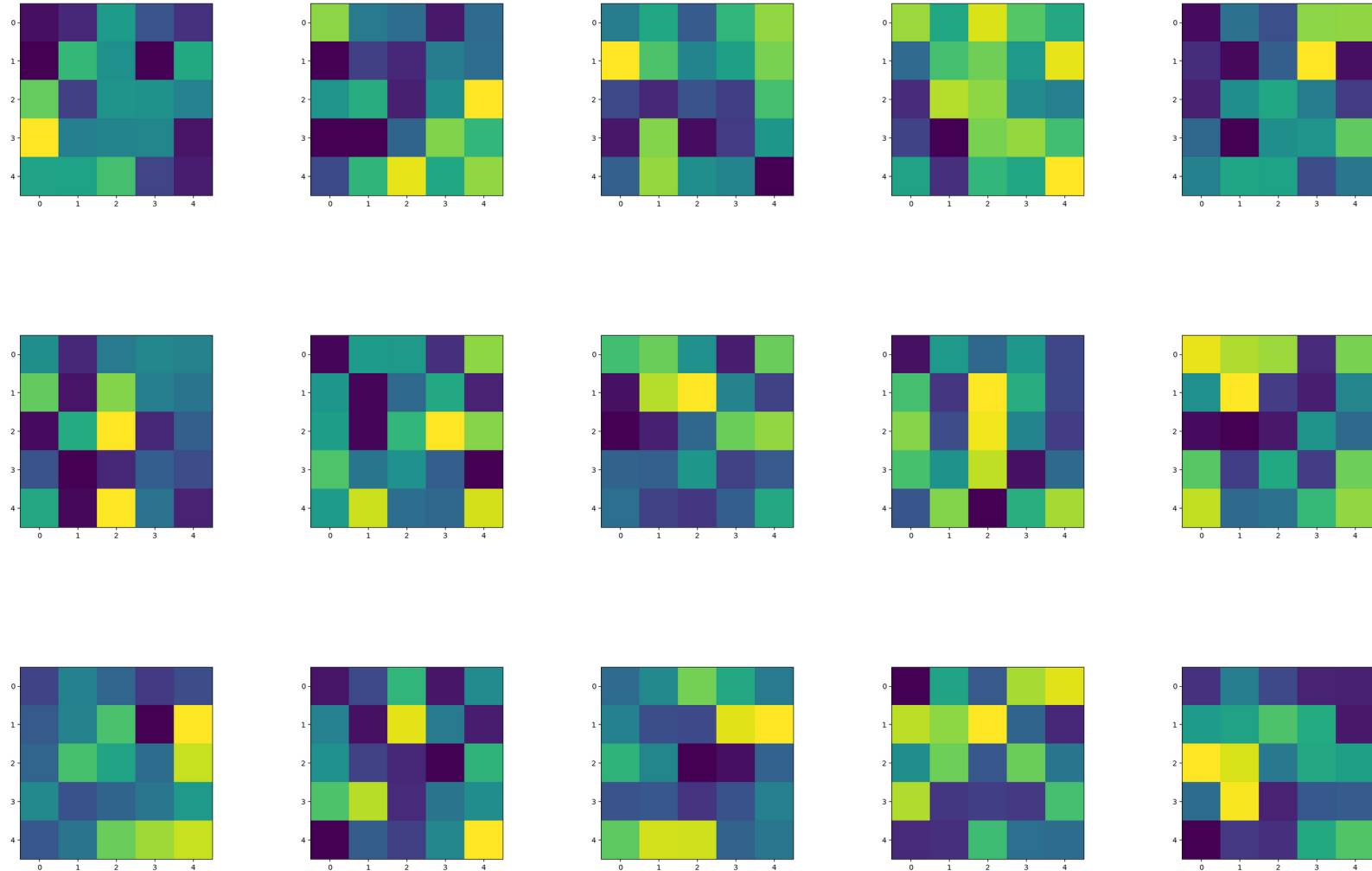
```
class OneConvLayer(nn.Module):  
  
    def __init__(self, in_dim, out_dim, nkernels):  
        super(OneConvLayer, self).__init__()  
        self.network1 = nn.Sequential(  
            nn.Conv2d(in_channels=1,  
                      out_channels=nkernels,  
                      kernel_size=5,  
                      stride=1,  
                      padding=2),  
            nn.BatchNorm2d(nkernels),  
            nn.ReLU(),  
            nn.MaxPool2d(kernel_size=2)  
        )  
        self.network2 = nn.Linear(nkernels*14*14,  
                                   out_dim)
```

Defining mnist3 (cont.)

```
def forward(self, x):  
    out = self.network1(x)  
    out = out.view(out.size(0), -1)  
    out = self.network2(out)  
    return out
```

```
model = OneConvLayer(28*28, 10, 32)
```

Sample Learned Kernels (32 Total)

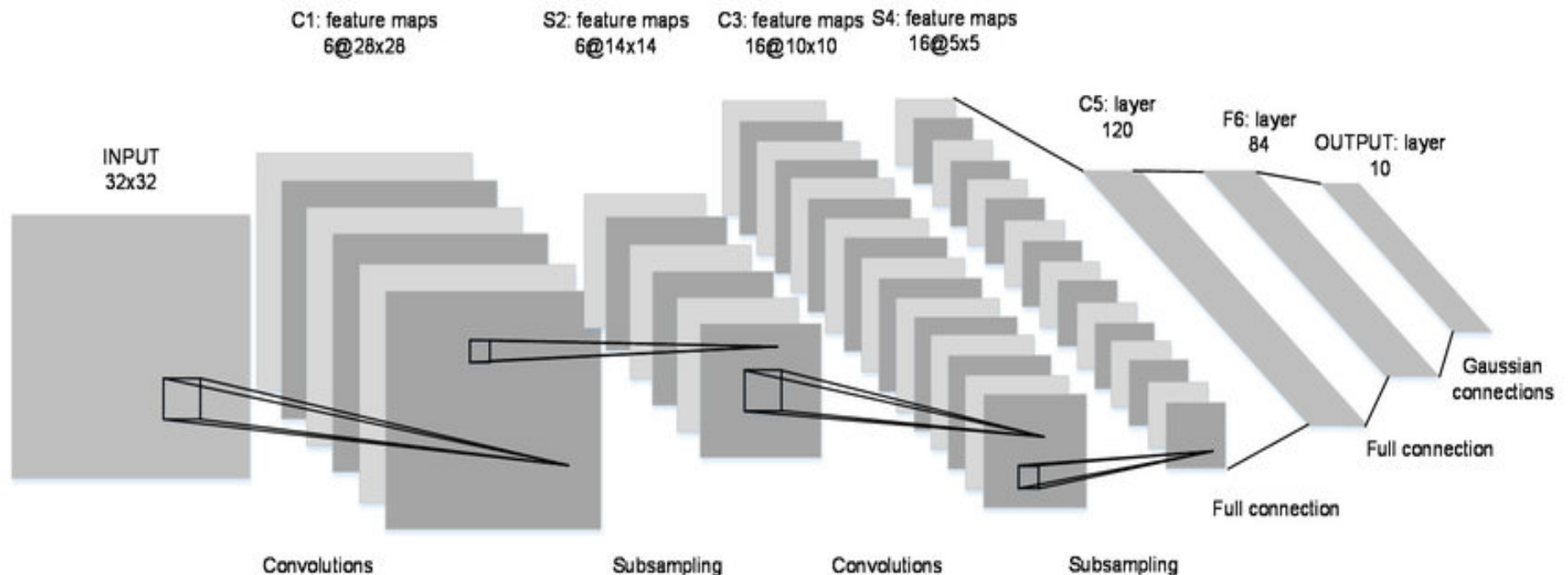


Deep Neural Networks

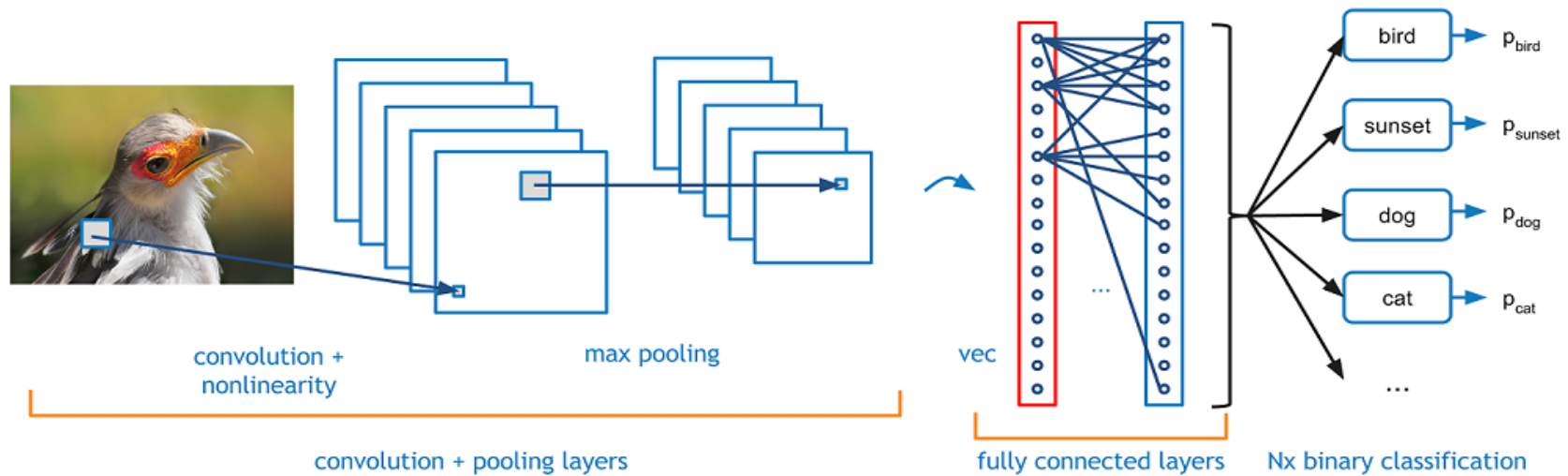
- For really hard problems (e.g., object recognition on color images) we may need many layers.
- Series of convolutional and max pooling layers, followed by some fully connected layers.
 - LeNet had 10 layers.
 - Inception V1 had 27 layers.
 - ResNet has 100 layers.
- GPUs required for training.

LeNet (Yann LuCun, 1990s)

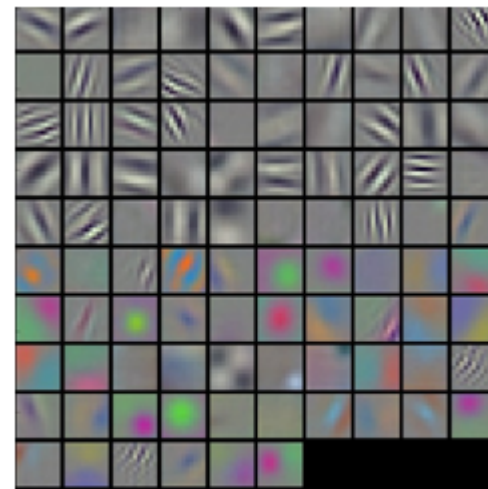
- Handwritten digit recognition



Object Recognition CNN

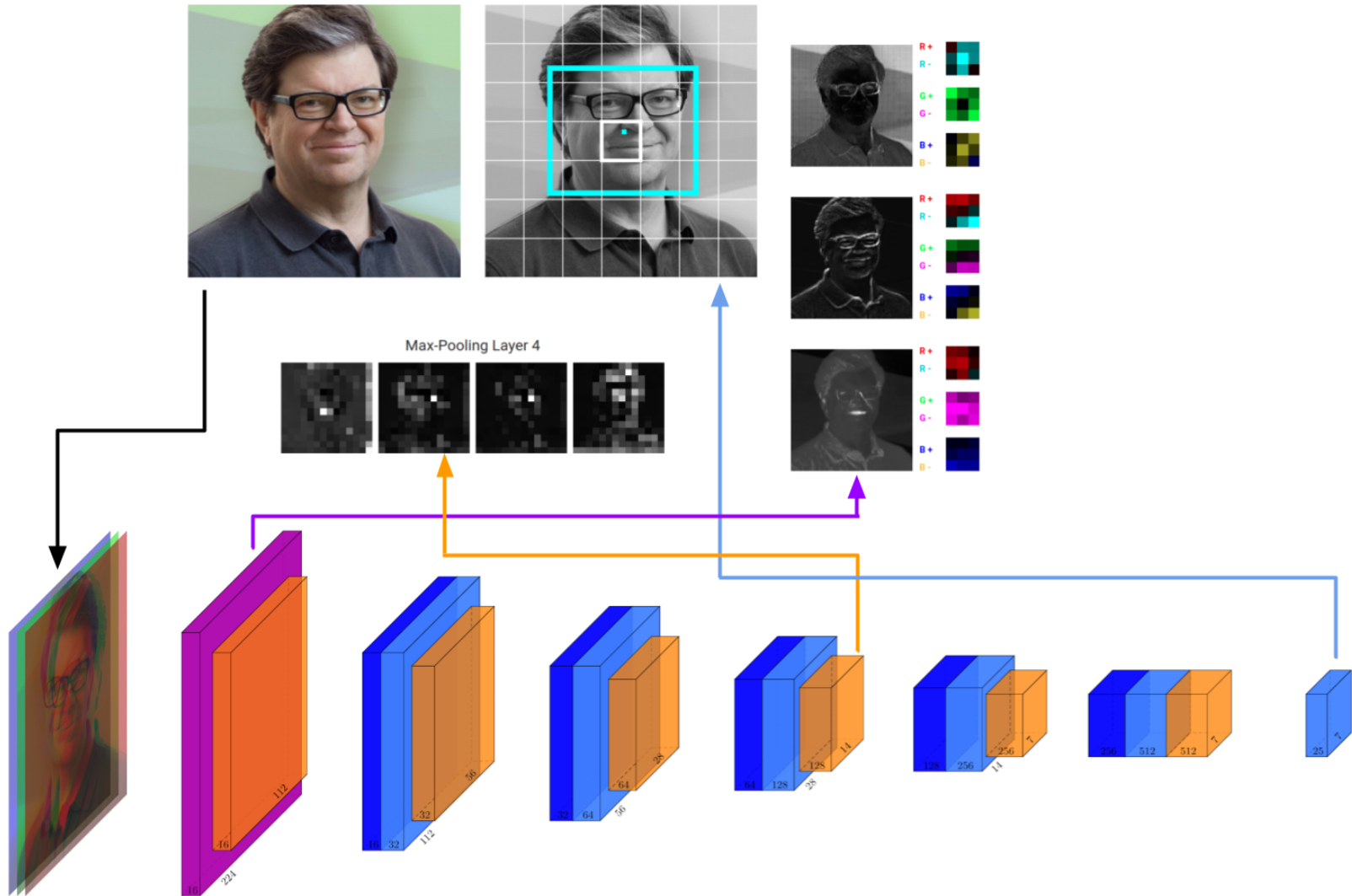


<https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>



Visualizations of filters

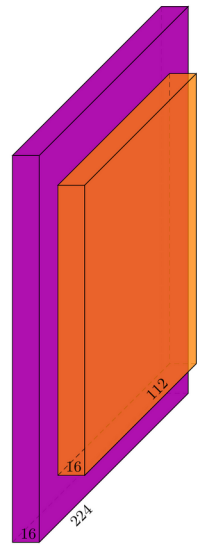
TinyYOLOV2 Face Recognition



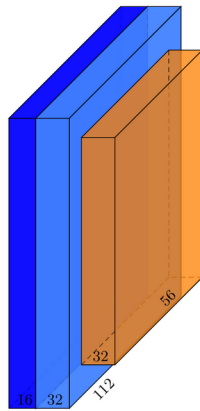
TinyYOLOV2 Architecture



Input Image
R, G, B Channels
224×224×3



conv1: [3,3,3]/1 × 16
224×224×16
max1: [2,2,1]/2 × 16
112×112×16

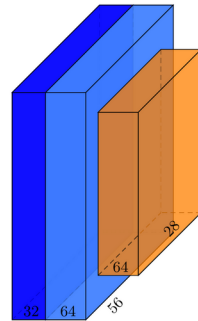


conv2: [3,3,1]/1 × 16
112×112×16

conv3: [1,1,16]/1 × 32
112×112×32

max2: [2,2,1]/2 × 32
56×56×32

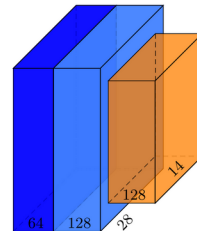
Conv & Pool Key: kernelType [kw,hw,inchan]/stride × outchan
Layer Dimension Key: width × height × channels



conv4: [3,3,1]/1 × 32
56×56×32

conv5: [1,1,32]/1 × 64
56×56×64

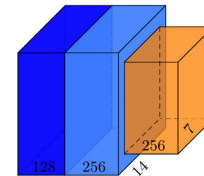
max3: [2,2,1]/2 × 64
28×28×64



conv6: [3,3,1]/1 × 64
28×28×64

conv7: [1,1,64]/1 × 128
28×28×128

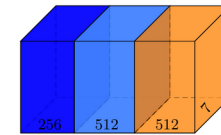
max4: [2,2,1]/2 × 128
14×14×128



conv8: [3,3,1]/1 × 128
14×14×128

conv9: [1,1,128]/1 × 256
14×14×256

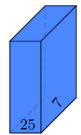
max5: [2,2,1]/2 × 256
7×7×256



conv10: [3,3,1]/1 × 256
7×7×256

conv11: [1,1,256]/1 × 512
7×7×512

max6: [2,2,1]/1 × 512
7×7×512



conv12: [1,1,512]/1 × 25
7×7×25

Purple: Full Convolution
Dark Blue: Depthwise Convolution
Light Blue: Pointwise Convolution
Orange: Max Pooling

Layer depths are drawn with logarithmic scaling. Max pooling layer widths and heights are shrunk by $\frac{1}{4}$ instead of $\frac{1}{2}$.