

Introduction to Neural Network, Convolution Neural Networks, and Deep Learning

Paul Rodriguez, PhD
(SDSC)

August 2025

Agenda

- 9:30 – 10:45 am Intro to NNs and CNNs (including a PyTorch intro)
- 10:45 - 11:00 am Break
- 11:00 - 12:00 am Practical Guidelines for training on HPC (w/ multinode execution)
- 12:00 - 1:30 pm Lunch
- 1:30 - 2:15pm: DL Layers and Architectures
- 2:15 - 3:15pm: DL Transfer Learning
- 3:15 - 3:30pm: Break
- 3:30 - 4:30pm: DL Special Connections and Transformers
- 4:30 - 4:45 pm: Q&A

From Machine Learning, Data Science, to AI

AI/Deep Learning – neural networks that can learn from large datasets of unstructured data (e.g. text and/or images)

Here, we focus on AI b/c of current trends on our systemss

Nature Survey, 2023

25%
Proportion
of articles

20

15

10

5

0

1983 1993 2003 2013 2023
©nature

AI and science: what 1,600 researchers think

A Nature survey finds that scientists are concerned, as well as excited, by the increasing use of artificial-intelligence tools in research.

By Richard Van Noorden & Jeffrey M. Perkel

— Computer science
— Physical sciences
— Life sciences
— Social sciences
— Health and medicine
— Total

Outline

- **Part I**

Overview of Neural Networks (aka Multilayer Perceptron)

Convolution Neural Networks and Scaling

Exercise, MNIST classification

- **Part II**

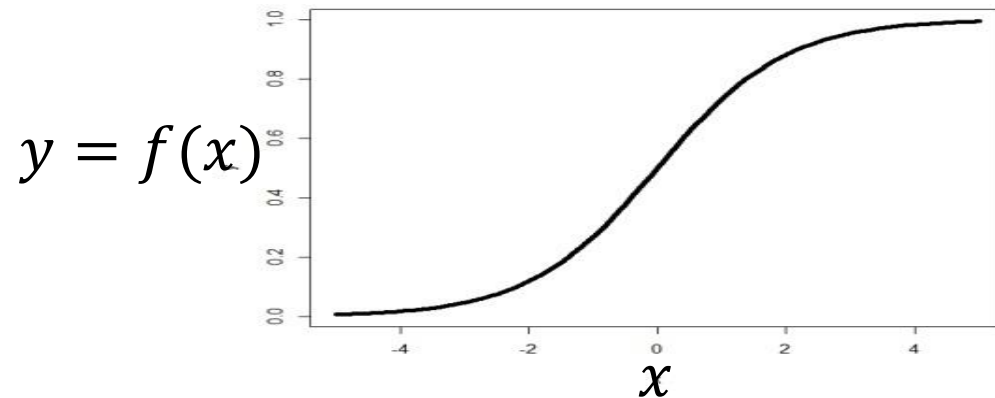
**Practical Guidelines: Hyperparameters, Workflows,
Batchjobs, GPUs**

Exercise, Multinode MNIST

Logistic Regression to Neural Network

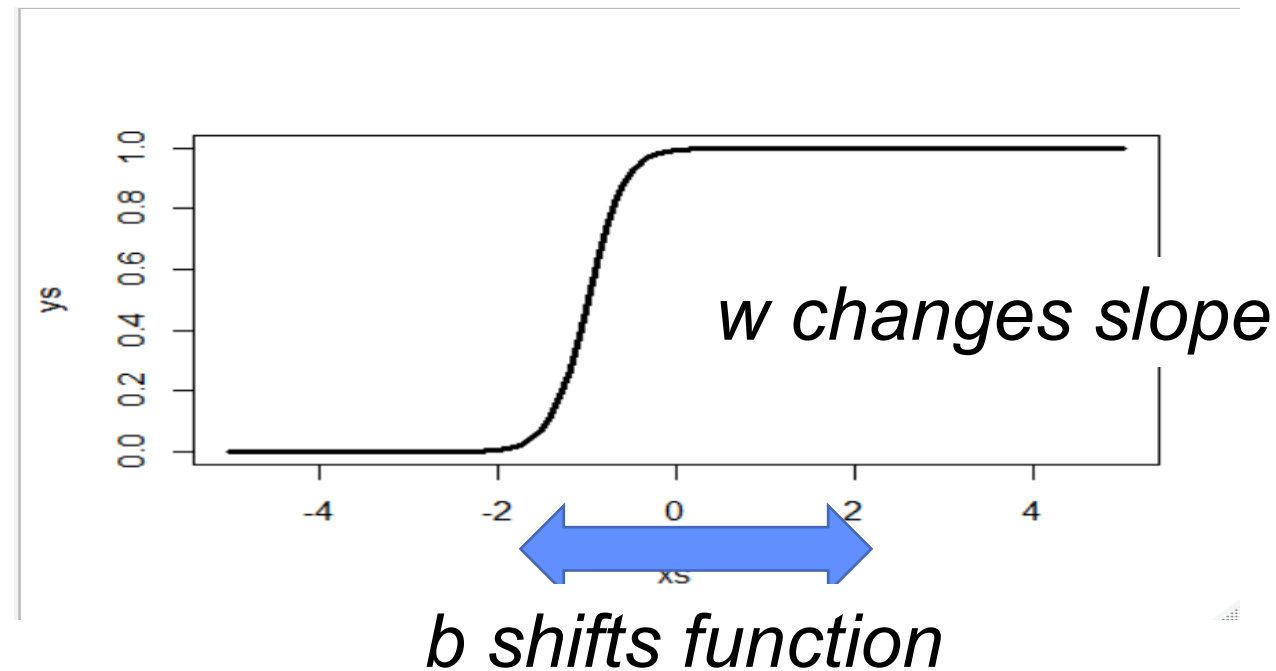
$$f(x, b, w) = \frac{\exp^{(b+wx)}}{1 + \exp^{(b+wx)}} = \frac{1}{1 + \exp^{-(b+wx)}}$$

for parameters: $b = 0$, $w_1 = 1$

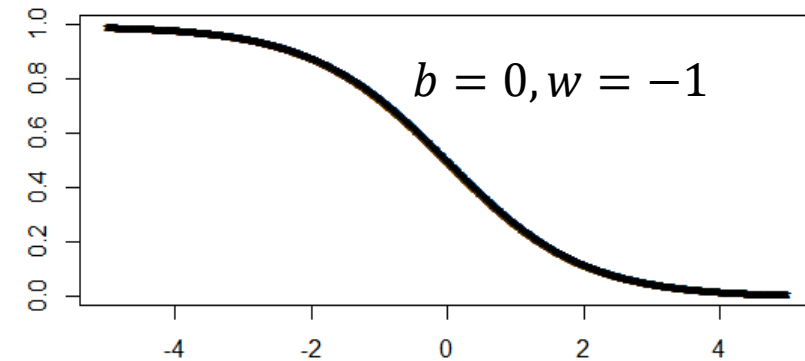
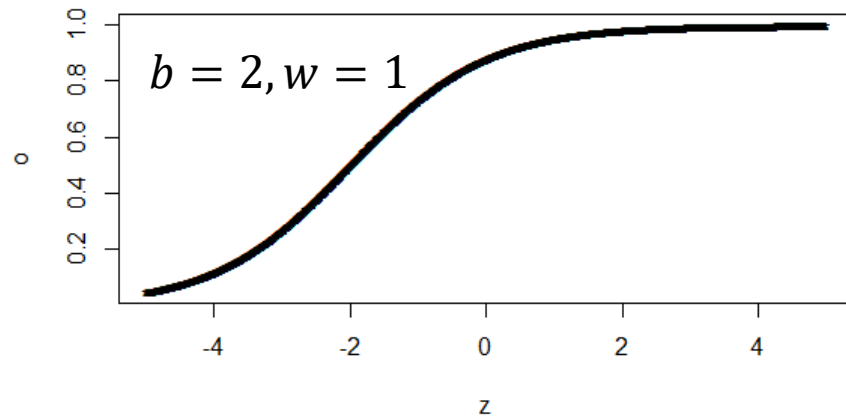
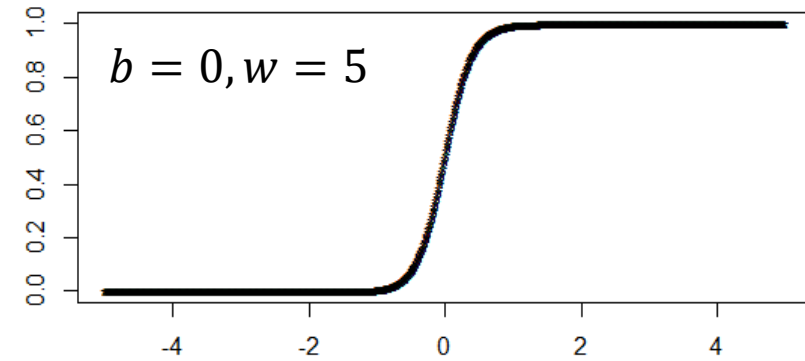
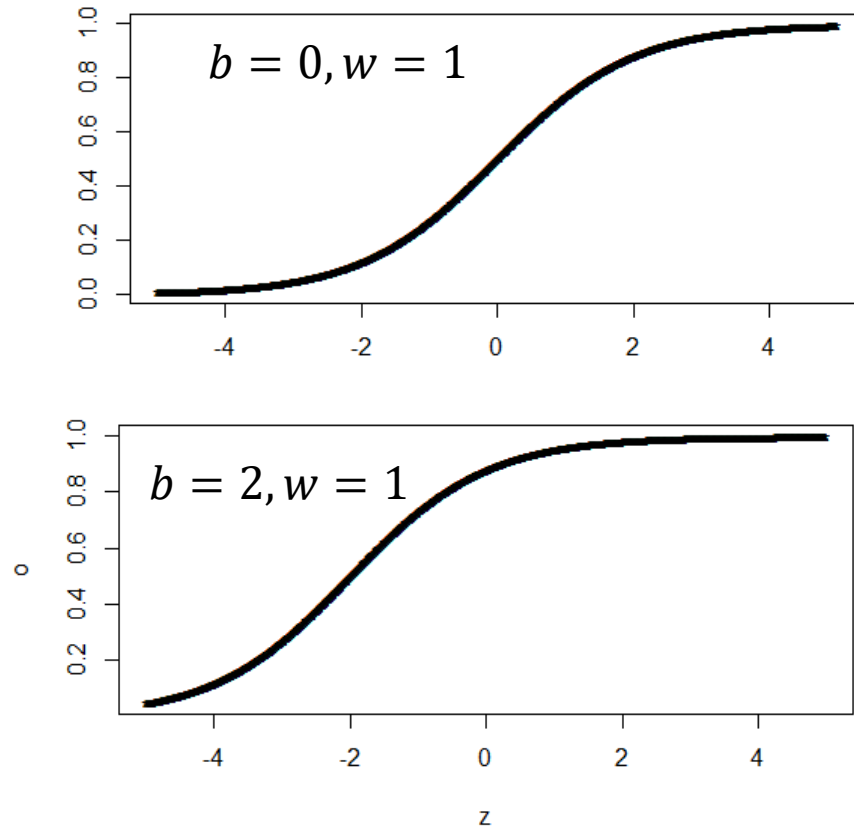


Logistic Regression to Neural Network

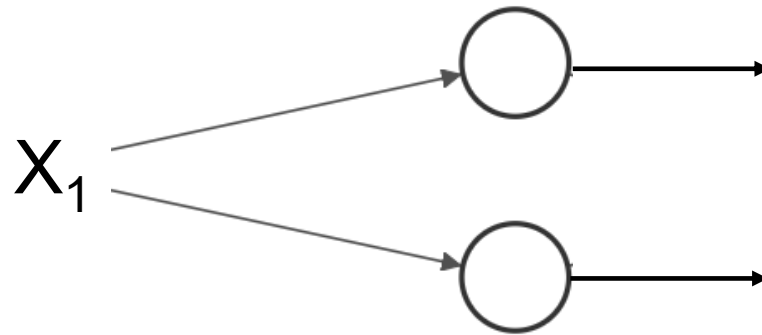
$$f(x, b, w) = \frac{\exp(b+wx)}{1 + \exp(b+wx)}$$



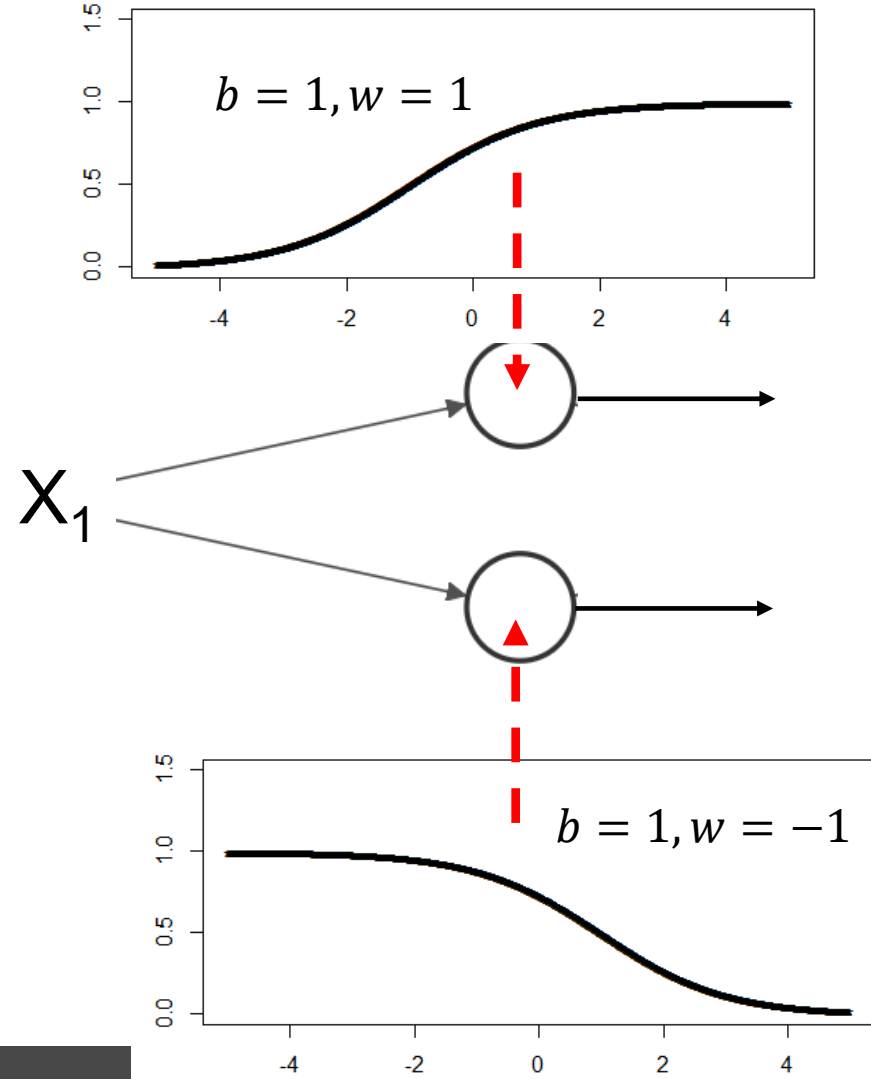
Logistic function w/various weights



Example: 1 input into 2 logistic units

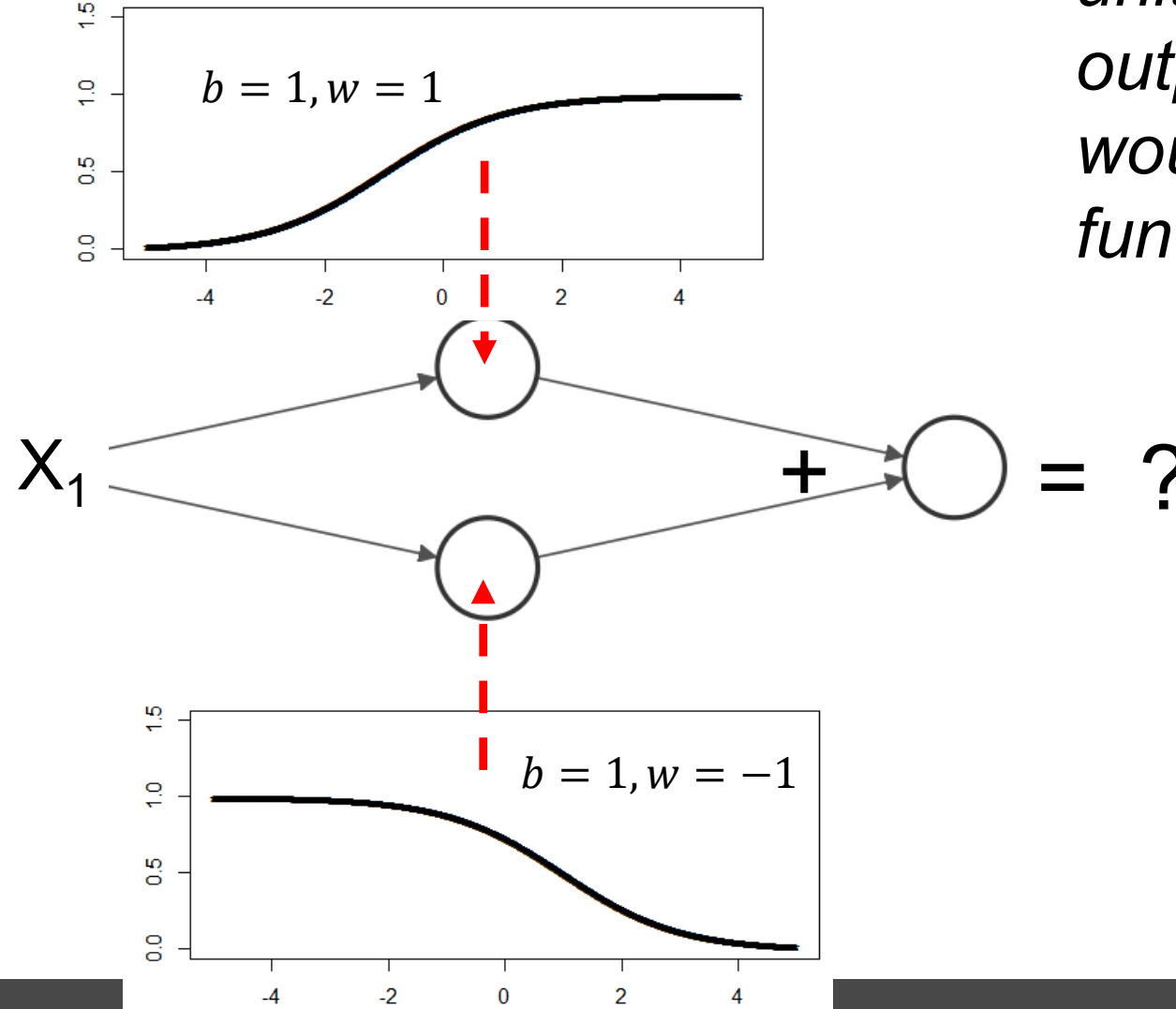


Example: 1 input into 2 logistic units with these activations



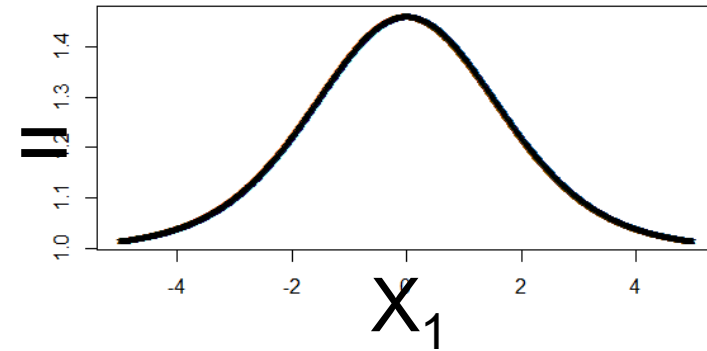
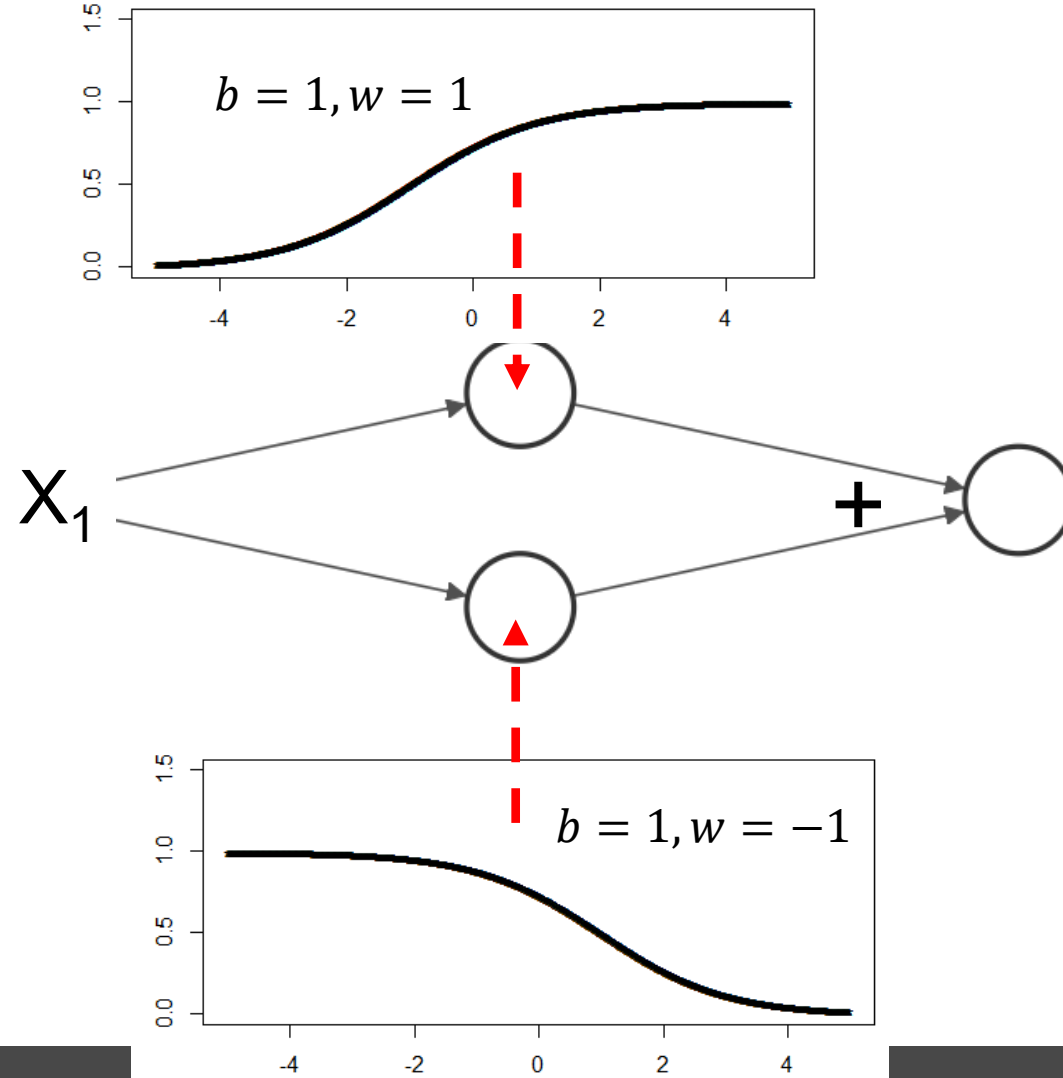
Example: 1 input into 2 logistic units with these activations

If you add these 2 units into a final output unit what would the output function look like?



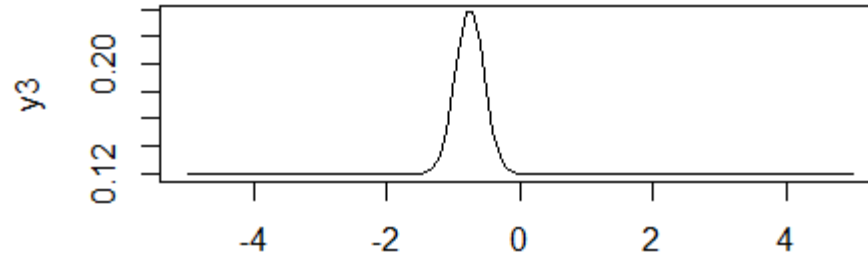
Example: 1 input into 2 logistic units with these activations

If you add these 2 units into a final output unit what would the output function look like?

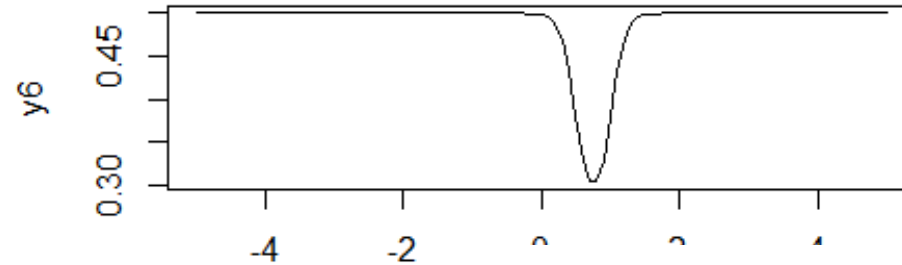


Higher level function combinations

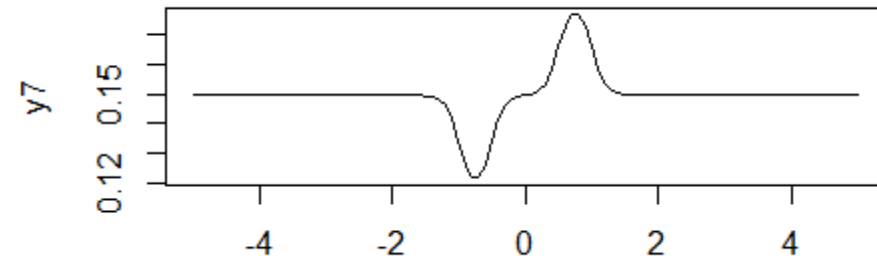
```
x=seq(-5,5,.1)
y1=1/(1+exp(10+ 10*x))
y2=1/(1+exp(-5+(-10)*x))
y3=1/(1+exp(1+1*y1+1*y2))
plot(x,y3,type="l")
```



```
y4=1/(1+exp(10+ (-10)*x))
y5=1/(1+exp(-5+(10)*x))
y6=1/(1+exp(1-1*y4-1*y5))
plot(x,y6,type="l")
```

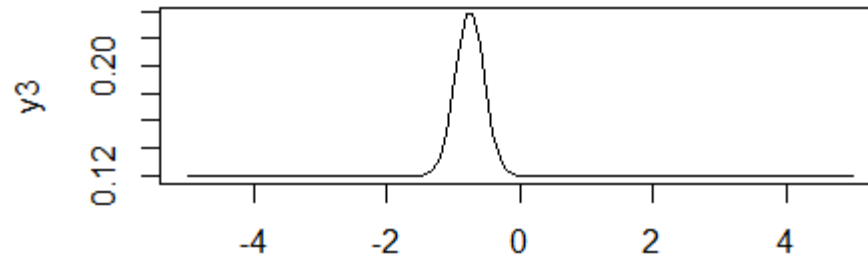


```
y7=1/(1+exp(1+2*y3+1*y6))
plot(x,y7,type="l")
```



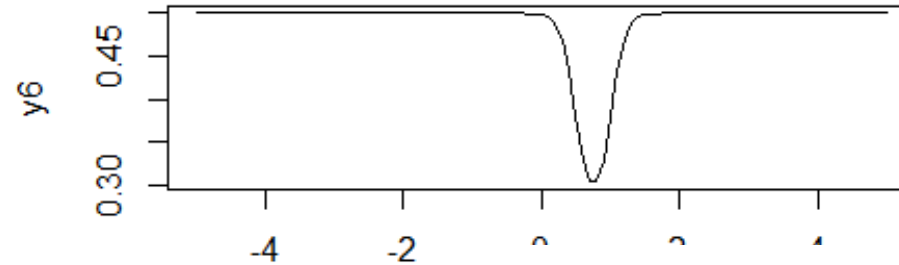
Higher level function combinations

```
x=seq(-5,5,.1)
y1=1/(1+exp(10+ 10*x))
y2=1/(1+exp(-5+(-10)*x))
y3=1/(1+exp(1+1*y1+1*y2))
plot(x,y3,type="l")
```

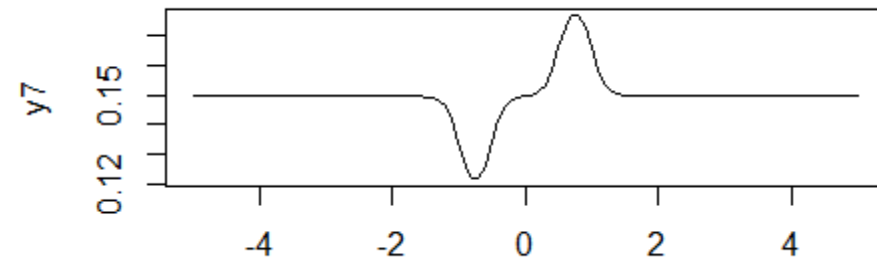


Multiple layer networks can represent any logical or real-valued functions (unbiased, but potential to overfit)

```
y4=1/(1+exp(10+ (-10)*x))
y5=1/(1+exp(-5+(10)*x))
y6=1/(1+exp(1-1*y4-1*y5))
plot(x,y6,type="l")
```



```
y7=1/(1+exp(1+2*y3+1*y6))
plot(x,y7,type="l")
```




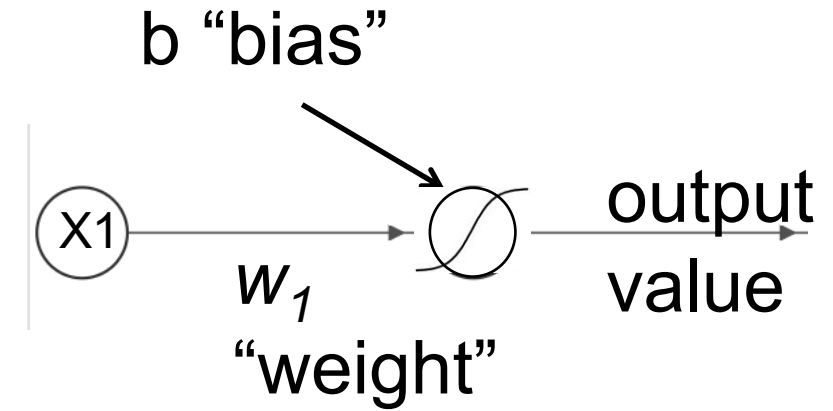
Logistic to Neural Network model

$$f(x, b, w) = \frac{\exp^{(b+wx)}}{1 + \exp^{(b+wx)}}$$

Draw out function as a little graph, 1 input


Logistic to Neural Network model

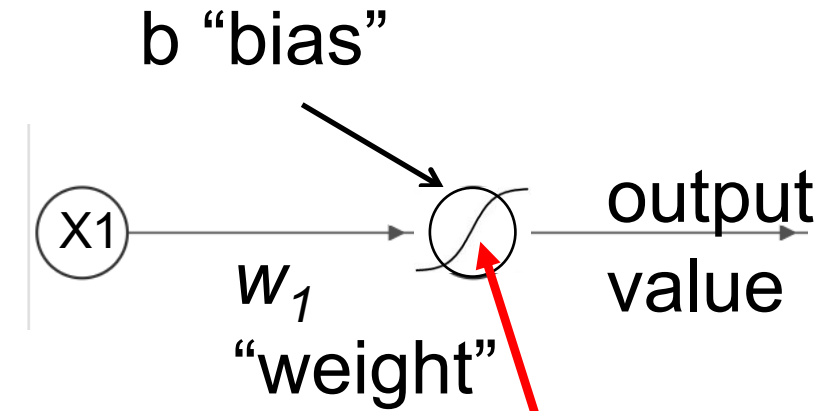
$$f(x, b, w) = \frac{\exp^{(b+w*x)}}{1 + \exp^{(b+w*x)}}$$




Draw out function as a little graph, 1 input

Logistic to Neural Network model

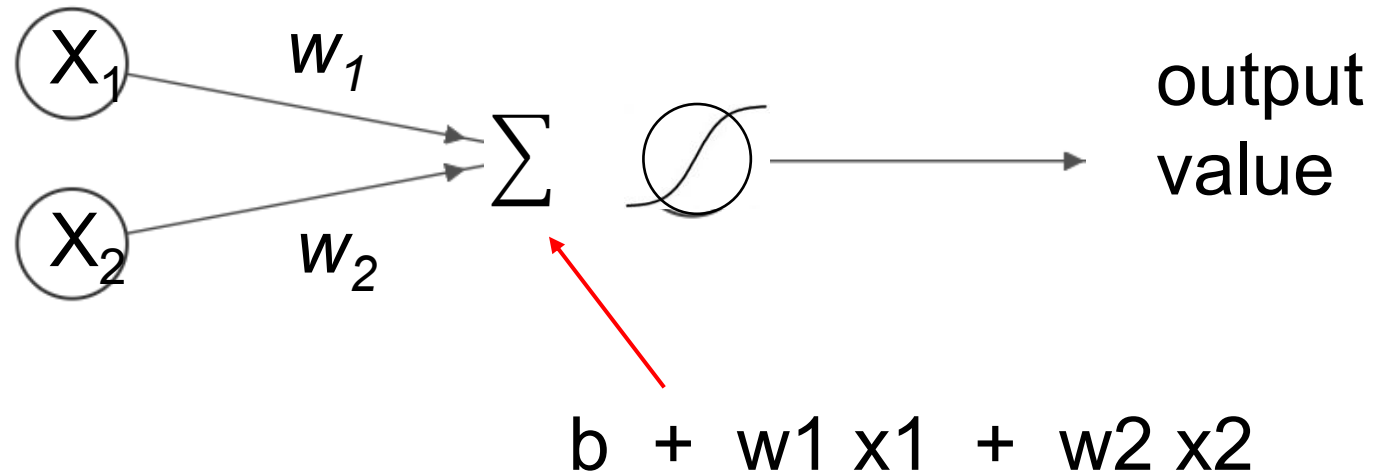
$$f(x, b, w) = \frac{\exp^{(b+w*x)}}{1 + \exp^{(b+wx)}}$$




Draw out function as a little graph, 1 input

logistic function will transform input to output – call it the ‘activation’ function

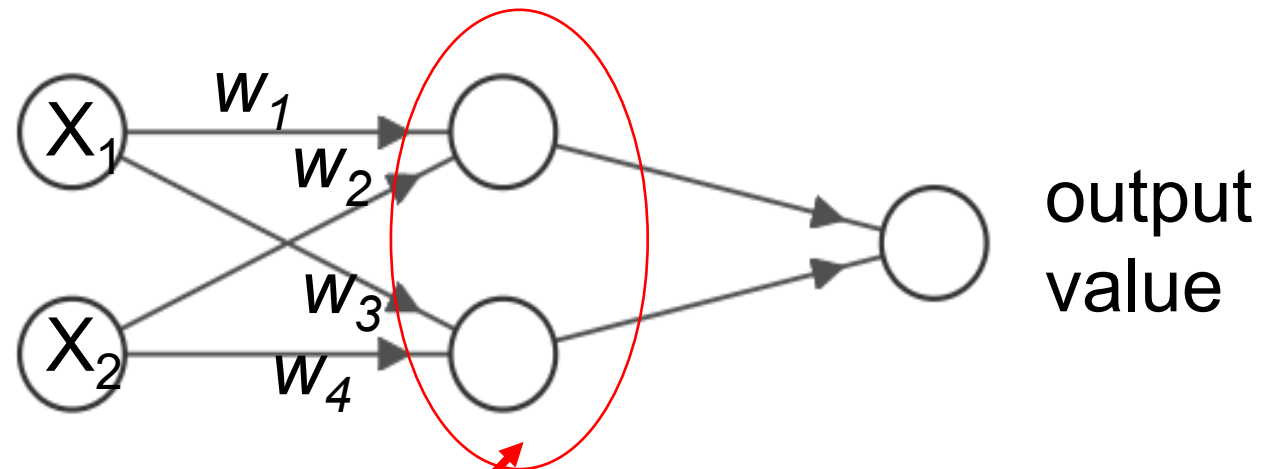
Using 2 input units, the graph model would be:



We usually don't draw the bias.

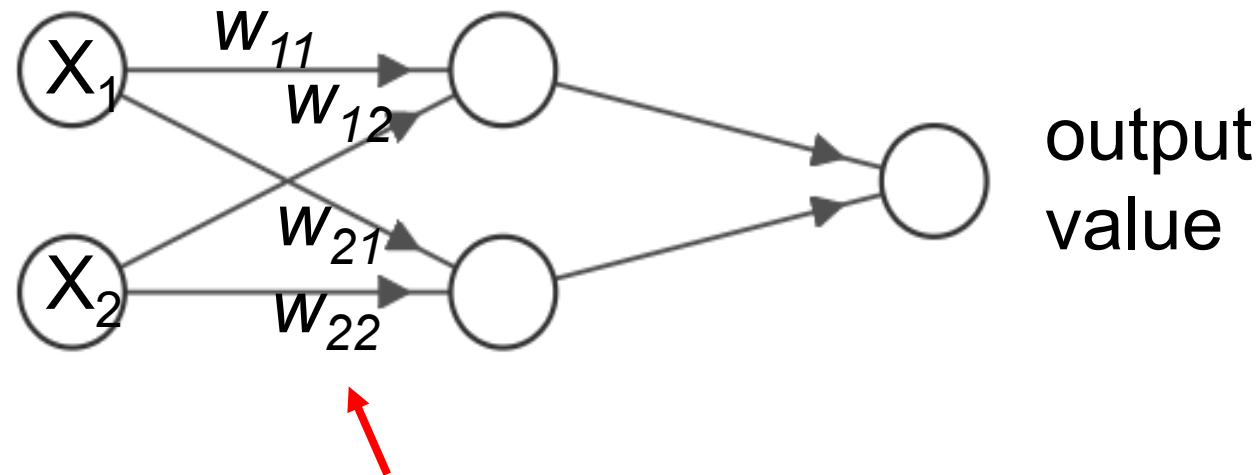
We assume inputs*weights are summed (a dot product)

Using 2 input units, 2 intermediate units, and 1 output:



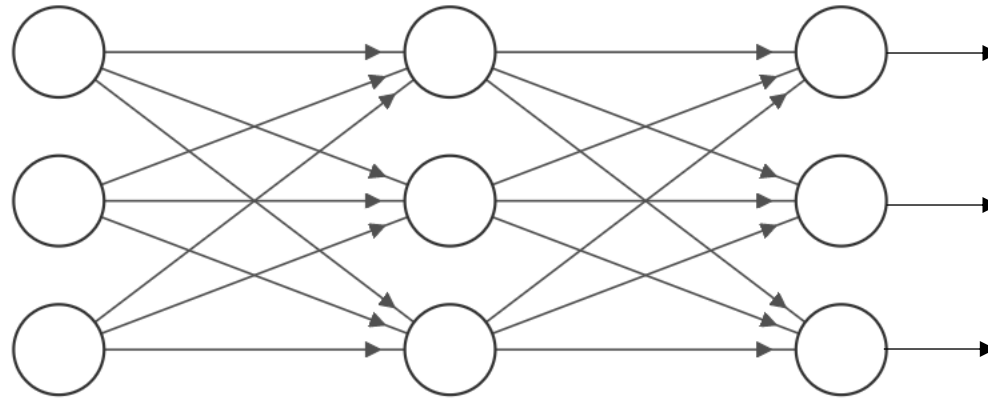
Call these "hidden units"

Using 2 input units, 2 intermediate units, and 1 output:

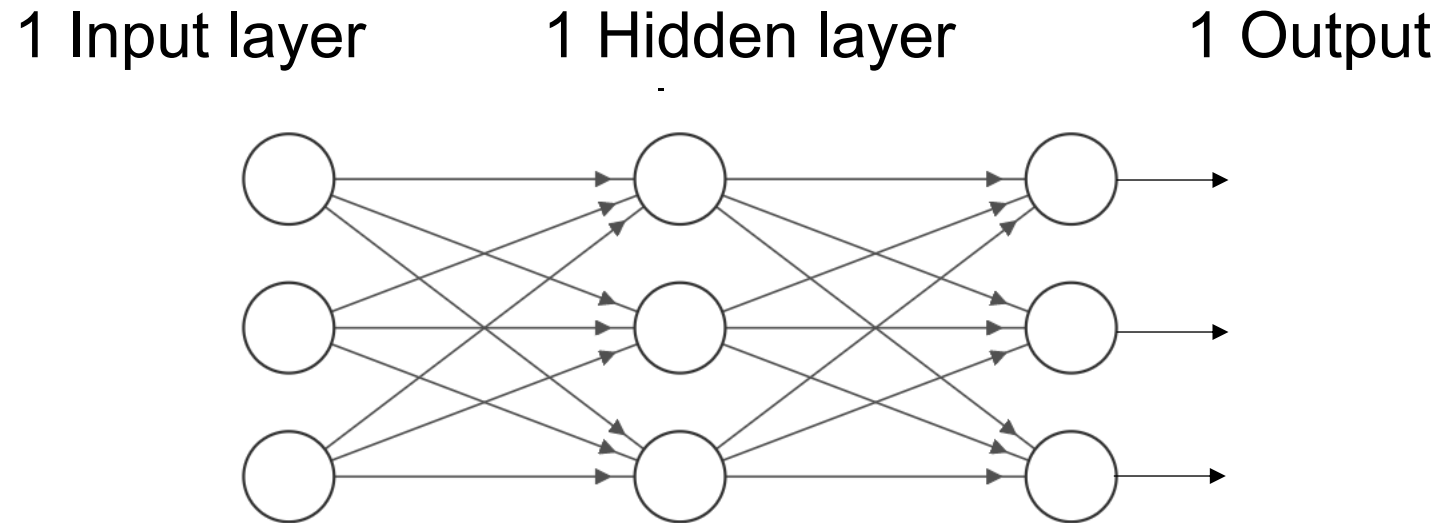


For X a $P \times 1$ vector, we set up a weight matrix W so that:
 $W * X = \text{incoming activations}$

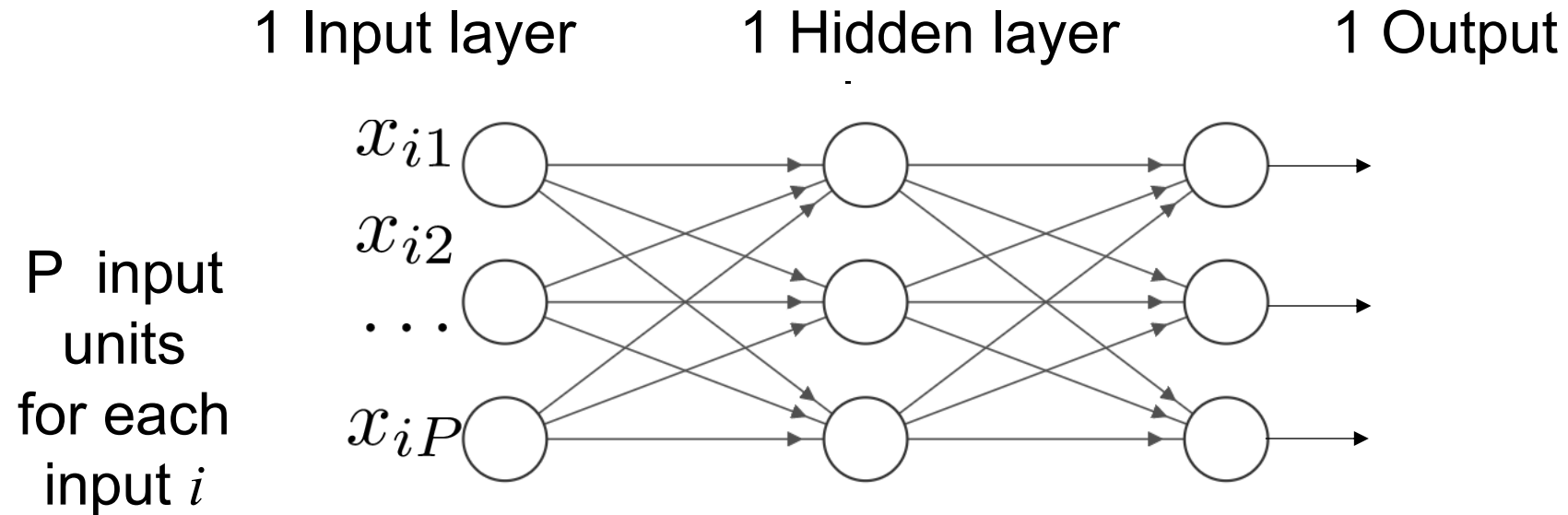
More generally, we can add a hidden layer,
and have many inputs and outputs



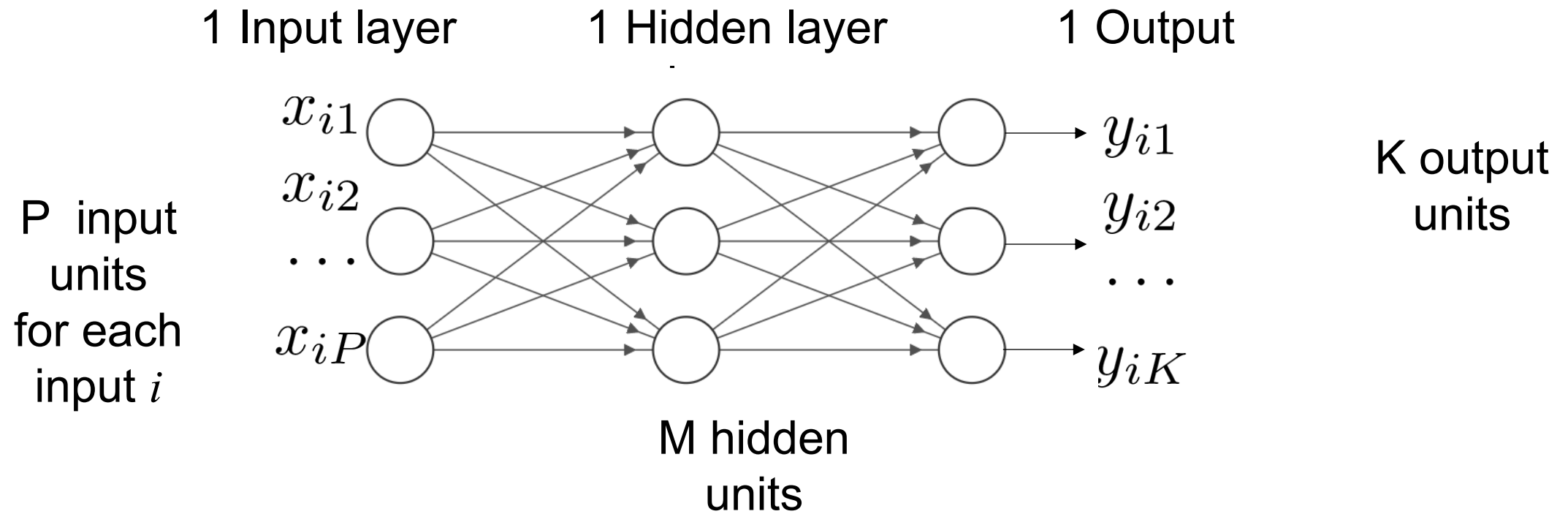
A “Multilayer Perceptron”



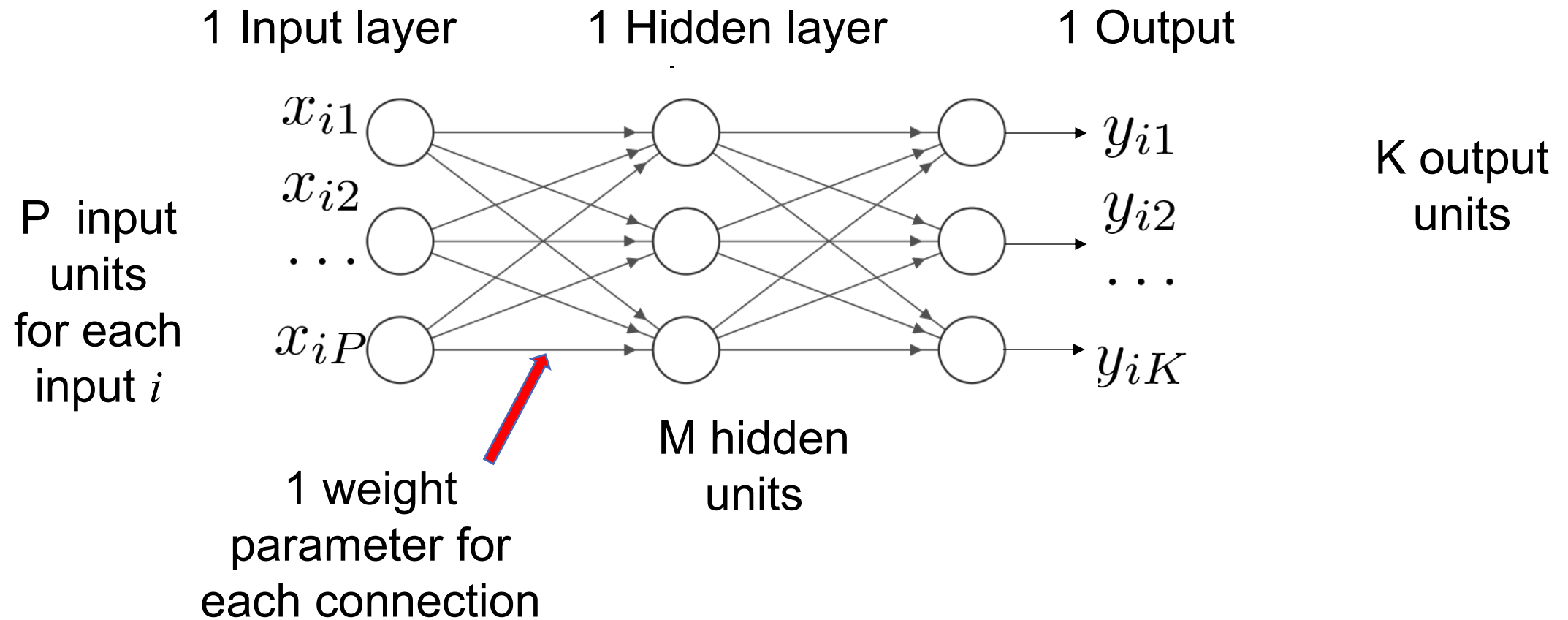
A “Multilayer Perceptron”



A “Multilayer Perceptron”

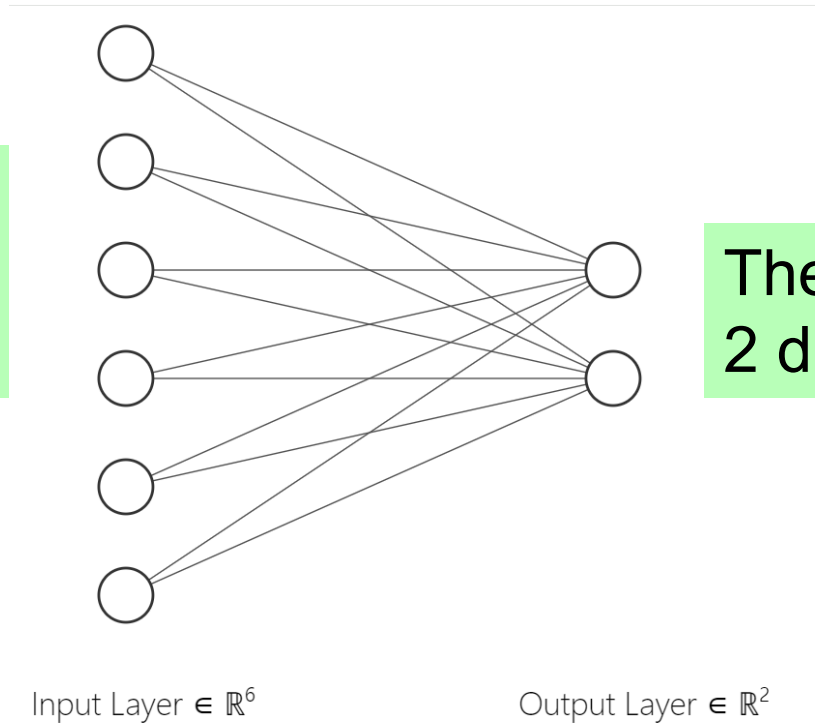


A “Multilayer Perceptron”



Quick side note: fewer units at each layer creates an '**embedding**' of the X input into a lower dimension output

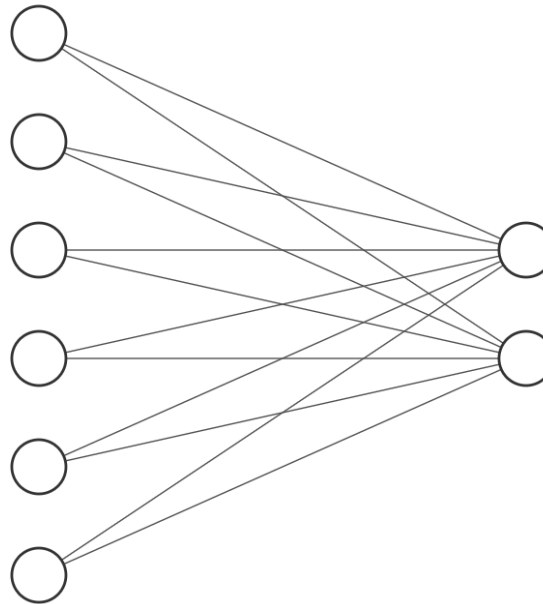
Here, the input vector has 6 values, so it's 6 dimensions



The output vector has 2 dimensions

Quick side note: fewer units at each layer creates an ***‘embedding’*** of the X input into a lower dimension output

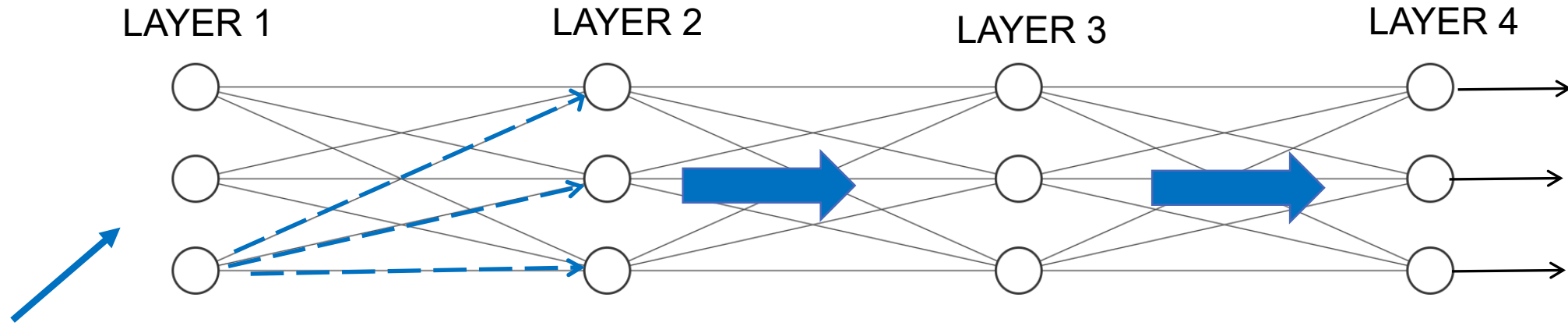
Here, the input vector has 6 values, so it's 6 dimensions



The output vector has 2 dimensions.

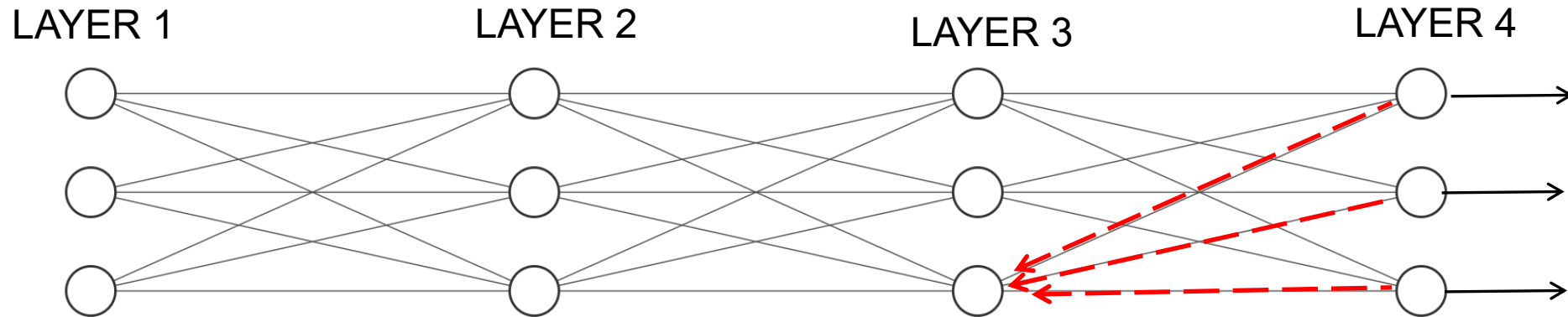
Learning the most relevant information usually means learning good embeddings!

Algorithm steps



1. FORWARD PROPAGATE
AN ENTIRE BATCH OF
INPUTS

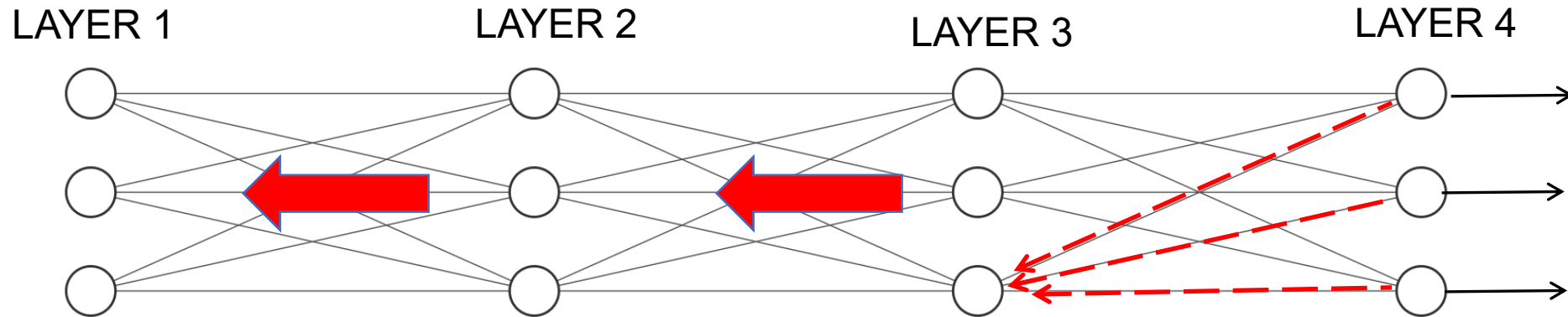
Algorithm steps



2. BACKWARD PROPAGATE ERROR FOR WHOLE BATCH USING DERIVATIVE CHAIN RULE:

$$\frac{dE}{dw_{mp}} = \sum_k^K \frac{dE_k}{d\hat{y}_k} \frac{d\hat{y}_k}{da_k} \frac{da_k}{dh_m} \frac{dh_m}{da_m} \frac{da_m}{dw_{mp}}$$

Algorithm steps and Vanishing Gradients

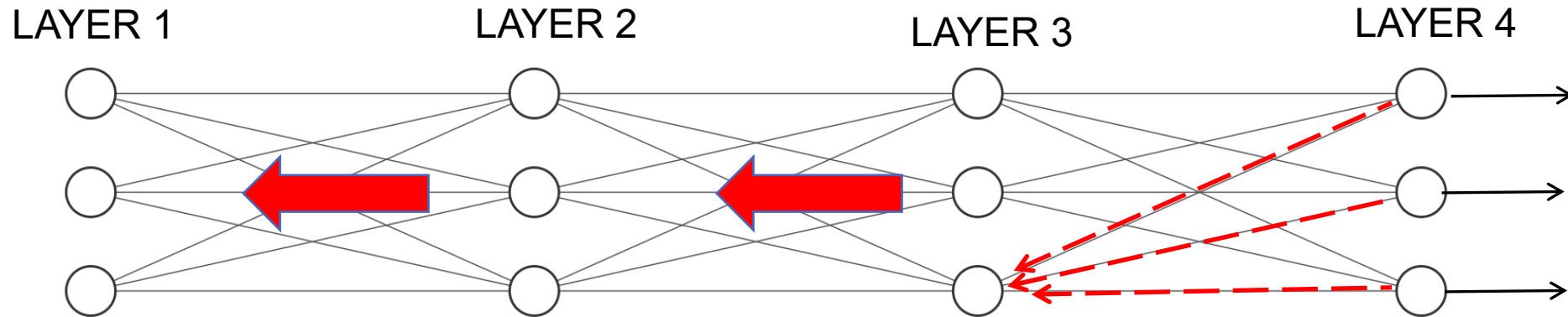


2. BACKWARD PROPAGATE ERROR FOR WHOLE BATCH USING DERIVATIVE CHAIN RULE:

Note: As you go farther back, the error information gets diluted and the error gradient starts 'vanishing'

$$\frac{dE}{dw_{mp}} = \sum_k^K \frac{dE_k}{d\hat{y}_k} \frac{d\hat{y}_k}{da_k} \frac{da_k}{dh_m} \frac{dh_m}{da_m} \frac{da_m}{dw_{mp}}$$

Algorithm steps and Vanishing Gradients



2. BACKWARD PROPAGATE ERROR FOR WHOLE BATCH USING DERIVATIVE CHAIN RULE:

Note: As you go farther back, the error information gets diluted and the error gradient starts 'vanishing'

A different activation function helps ...

$$\frac{dE}{dw_{mp}} = \sum_k^K \frac{dE_k}{d\hat{y}_k} \frac{d\hat{y}_k}{da_k} \frac{da_k}{dh_m} \frac{dh_m}{da_m} \frac{da_m}{dw_{mp}}$$

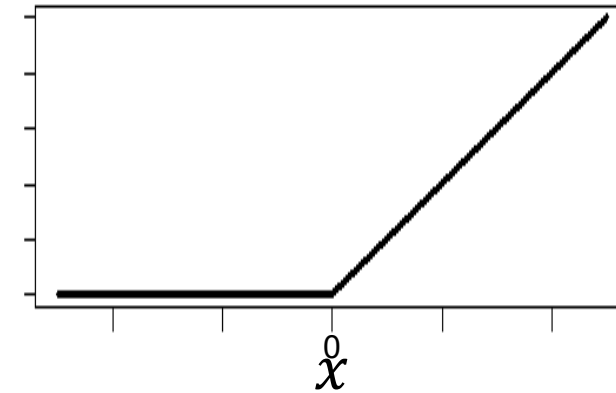
The rectified linear unit (RELU)

RELU (rectified linear unit)

RELU activation function

It is unscaled (bad!)

But df/da is constant (good!)



$$f(a) = \begin{cases} a & a > 0 \\ 0 & a \leq 0 \end{cases}$$

where $a = XW$

Overall, RELU mitigates vanishing gradients

The Neural Network Algorithm

INITIALIZE weights to small value (for example: ± 0.3)

LOOP until stopping criterion:

The Neural Network Algorithm

INITIALIZE weights to small value (for example: +/- <0.3)

LOOP until stopping criterion:

FORWARD PROPAGATION: calculate all node activations

BACKWARD PROPAGATION: calculate all error derivatives to *minimize Loss*

The Neural Network Algorithm

INITIALIZE weights to small value (for example: +/- <0.3)

LOOP until stopping criterion:

FORWARD PROPAGATION: calculate all node activations

BACKWARD PROPAGATION: calculate all error derivatives to *minimize Loss*

UPDATE WEIGHTS: $w \leftarrow w - \text{learning_rate} * \frac{dL}{dw}$

The Neural Network Algorithm

INITIALIZE weights to small value (for example: +/- <0.3)

LOOP until stopping criterion:

FORWARD PROPAGATION: calculate all node activations

BACKWARD PROPAGATION: calculate all error derivatives to *minimize Loss*

UPDATE WEIGHTS: $w \leftarrow w - \text{learning_rate} * \frac{dL}{dw}$

STOP: when validation error reaches minimum or after a max number of epochs

The Neural Network Algorithm [and heuristics]

INITIALIZE weights to small value (for example: +/- <0.3)

LOOP until stopping criterion:

[work in batches of input]

FORWARD PROPAGATION: calculate all node activations

BACKWARD PROPAGATION: calculate all error derivatives to *minimize Loss*

UPDATE WEIGHTS: $w \leftarrow w - \text{learning_rate} * \frac{dL}{dw}$

[adapt learning rate,
use momentum]

STOP: when validation error reaches minimum or after a max number of epochs

[several metrics of loss are possible]

Neural Network main options to choose:

- **1 Architecture: number of hidden units & layers**
- **2 Optimizer and learning rate**
- **3 Loss function depends on task**
- **Note: more hidden layers, more hidden units => more potential for overfitting**

terminology and cheat sheet on output activations (for reference):

Type of Problem	Y outputs	Output Activation Function (this gives a SCORE \hat{Y} :)	Output PREDICTION (what you decide to predict)	Output Loss Function	Evaluative Measure
Regression: map into to K real valued predictions	if $Y \in (-\infty, +\infty)^K$	$\hat{Y} = XW$	\hat{Y} :	Mean Squared Error (MSE)	Mean Squared Error (MSE)
Multivariate output of 0's and 1's	if $Y \in [0, 1]^K$	$\hat{Y} = \frac{1}{1 + \exp^{-(XW)}}$	1 or 0	MSE	MSE
Binary Classification	if $Y \in \{0, 1\}$	$\hat{Y} = \frac{1}{1 + \exp^{-(XW)}}$	A probability given by \hat{Y} : $P(y = 1 x)$	Cross Entropy $L = -y \log(\hat{y}) - (1 - y)(\log(\hat{y}))$	Accuracy, ROC
Multiclassification	if $Y \in \{0, 1\}^K$	$\hat{Y}_k = \frac{\exp^{-(XW_k)}}{\sum_k \exp^{-(XW_k)}}$	Max class	Cross Entropy $L = - \sum_k y_k \log(\hat{y}_k)$	Accuracy

Summary:

Pro:

Multilayer Perceptron, and Neural Networks in general, are flexible powerful learners

Hidden layers learn a nonlinear transformation of input

Summary:

Pro:

Multilayer Perceptron, and Neural Networks in general, are flexible powerful learners

Hidden layers learn a nonlinear transformation of input

Con:

Lots of parameters

Hard to interpret

Needs more data

**A neural network can discover visual features using
'convolutions'**

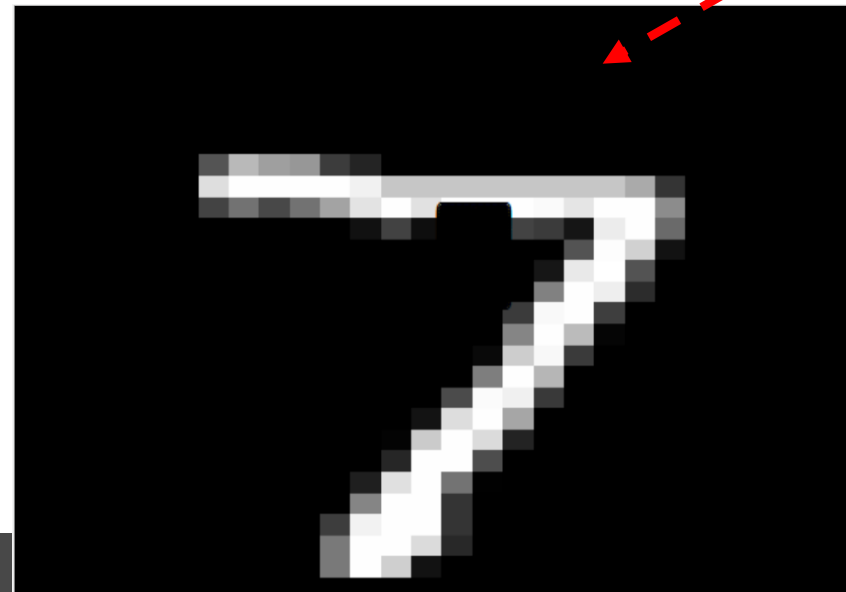
Next: Image classification of digits

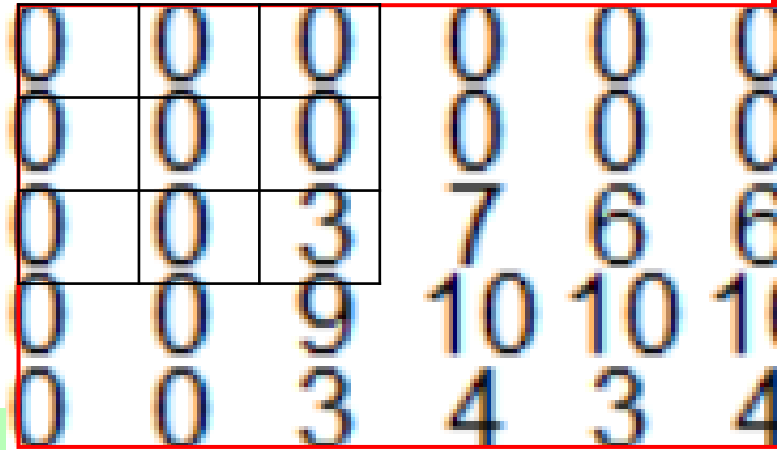
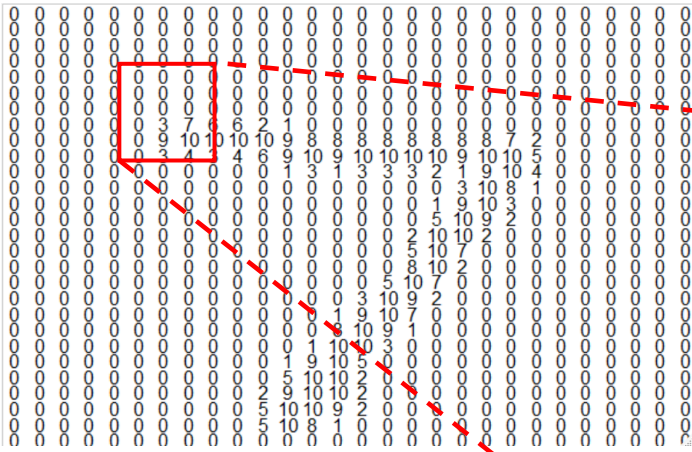
Image features

- **MNIST - A database of handwritten printed digits**
(National Inst. of Standards and Technology)



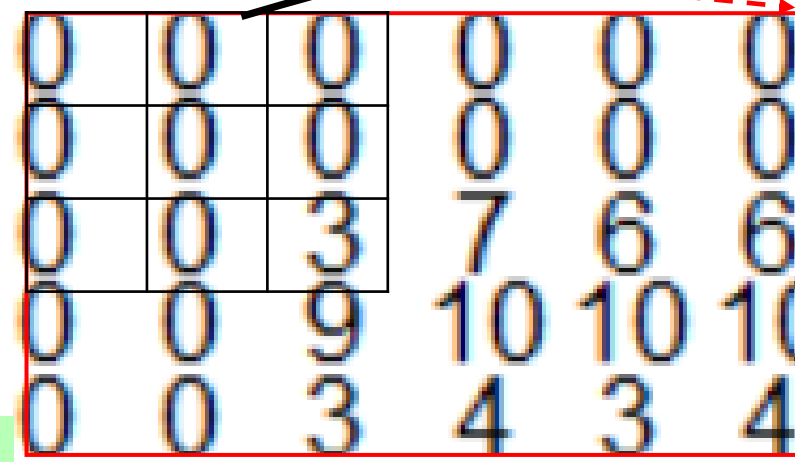
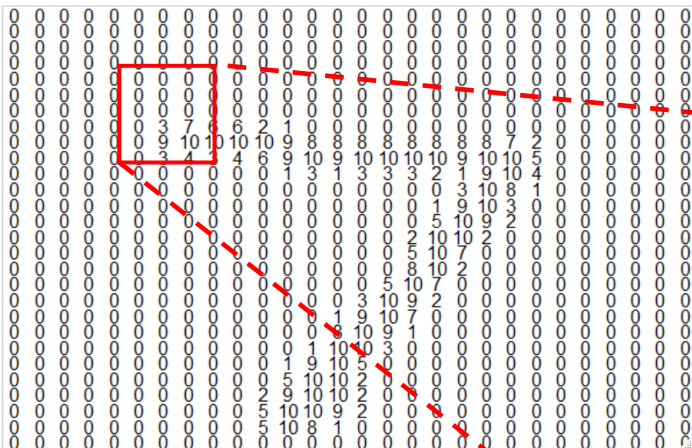
How to classify digits?





Let's zoom into 5x6 window of pixels near the tip of '7'

Take a 3x3 patch of pixels and apply a 'filter' template – designed to find an edge



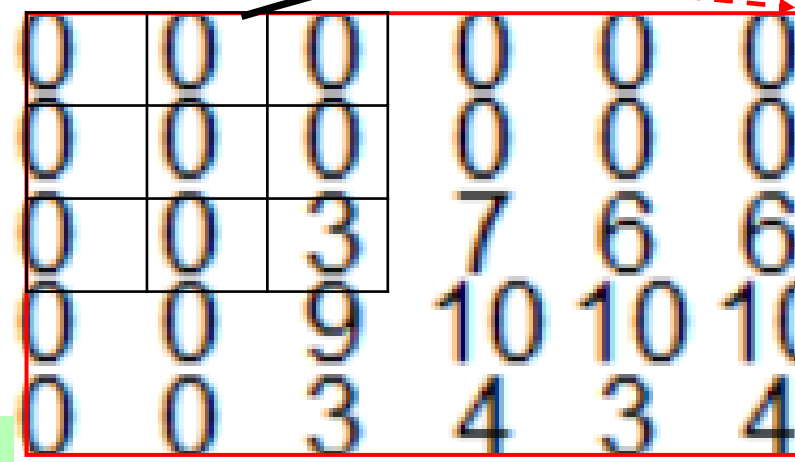
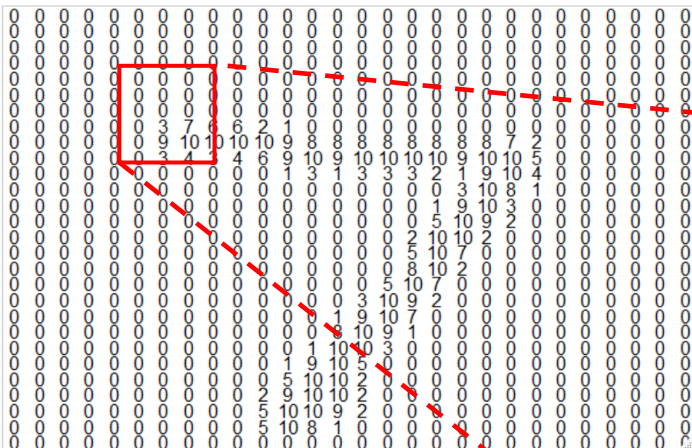
X

-1	0	+1
-1	0	+1
-1	0	+1

1. Multiply 3x3 patch of pixels with 3x3 filter

Let's zoom into 5x6 window of pixels near the tip of '7'

Take a 3x3 patch of pixels and apply a 'filter' template – designed to find an edge



(our weight parameters)

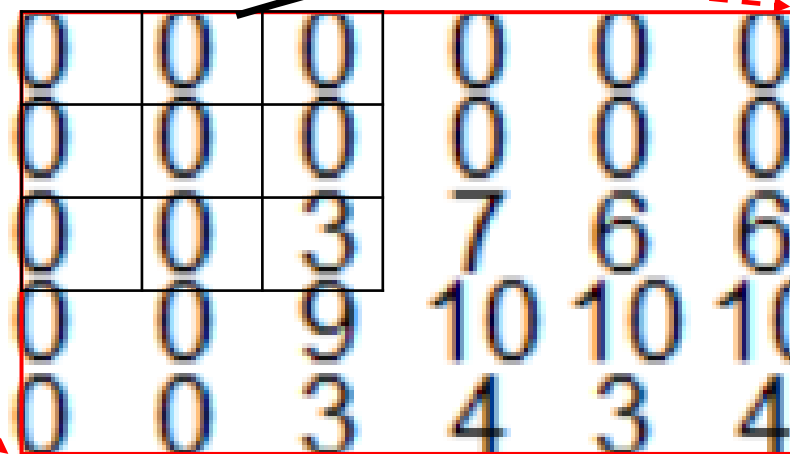
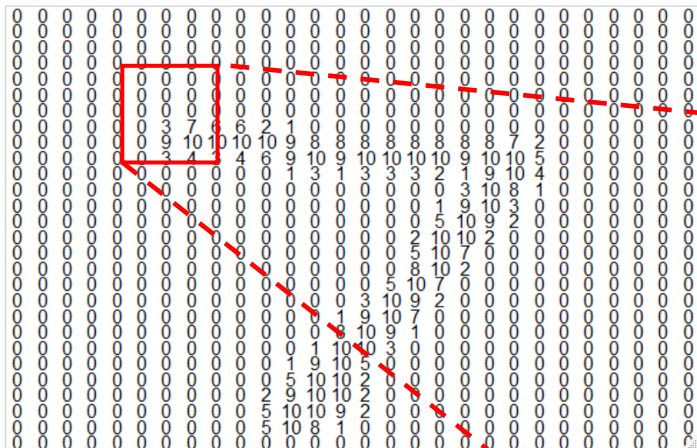
-1	0	+1
-1	0	+1
-1	0	+1

X

1. Multiply 3x3 patch of pixels with 3x3 filter “W”

Let's zoom into 5x6 window of pixels near the tip of '7'

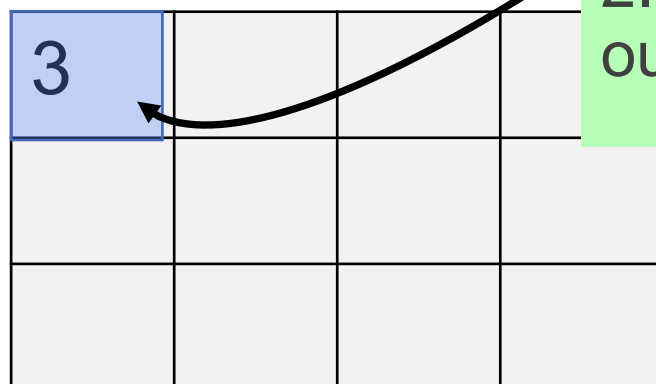
Take a 3x3 patch of pixels and apply a 'filter' template – designed to find an edge



X

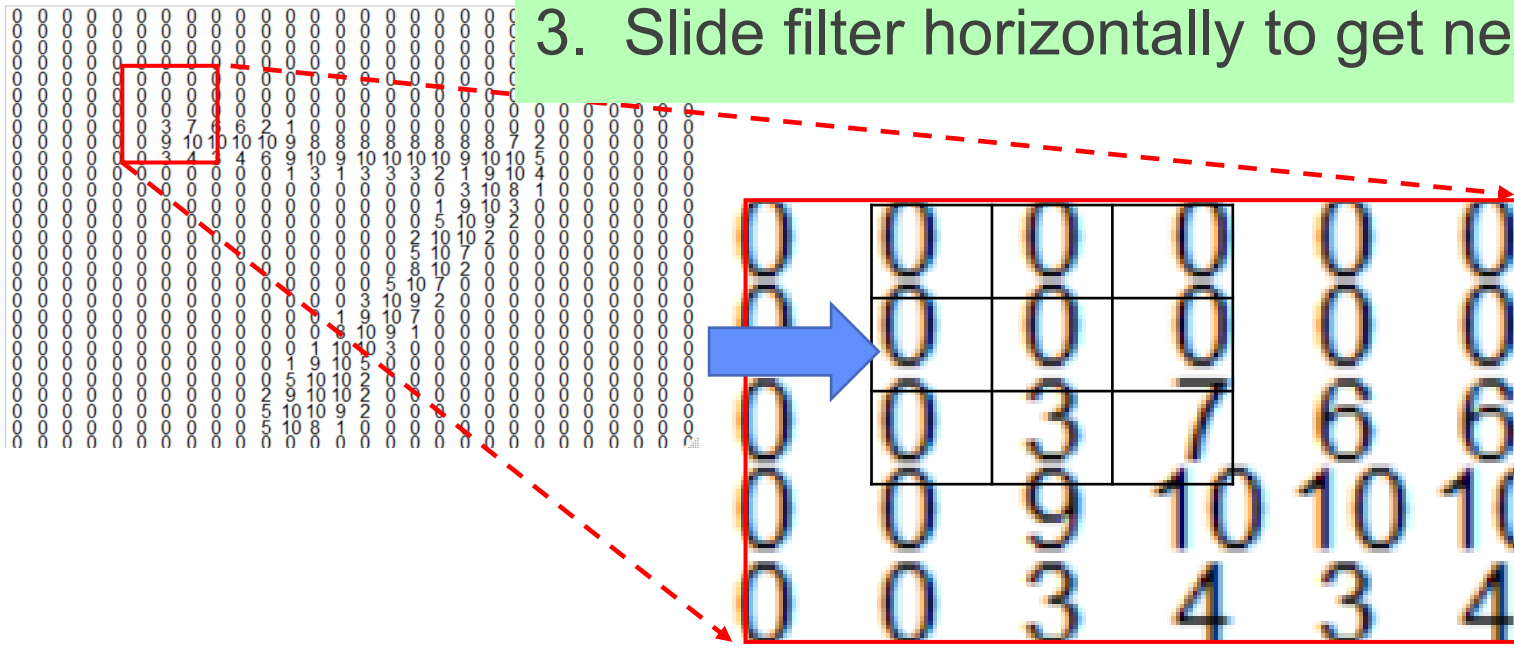
-1	0	+1
-1	0	+1
-1	0	+1

1. Multiply 3x3 patch of pixels with 3x3 filter "W"

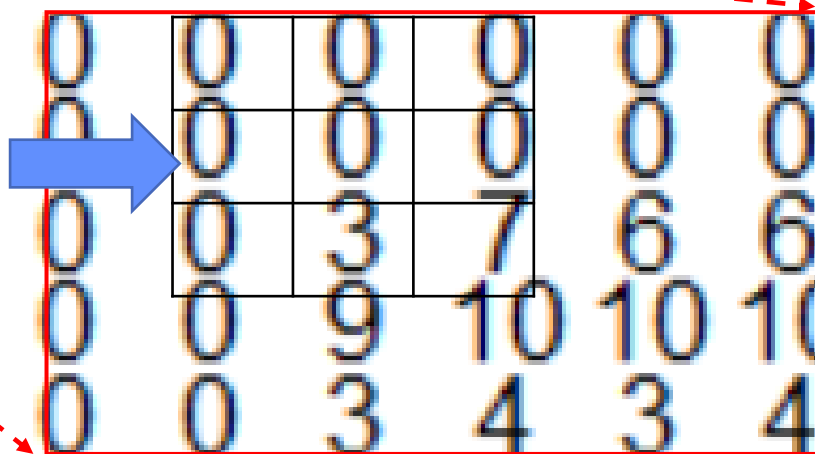
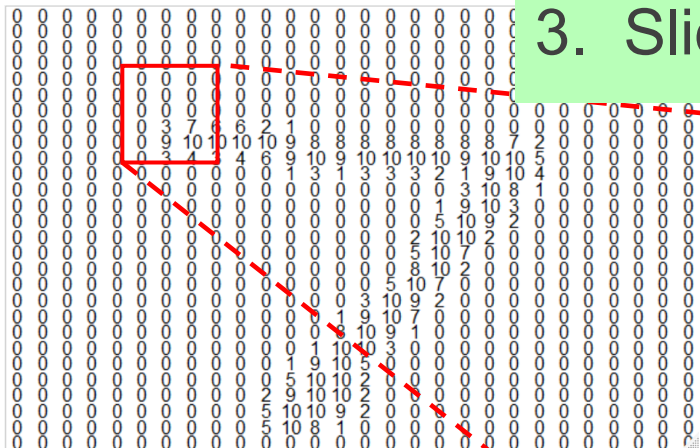


2. Put sum in new cell of output map

3. Slide filter horizontally to get next output value



3. Slide filter horizontally to get next output value



-1	0	+1
-1	0	+1
-1	0	+1

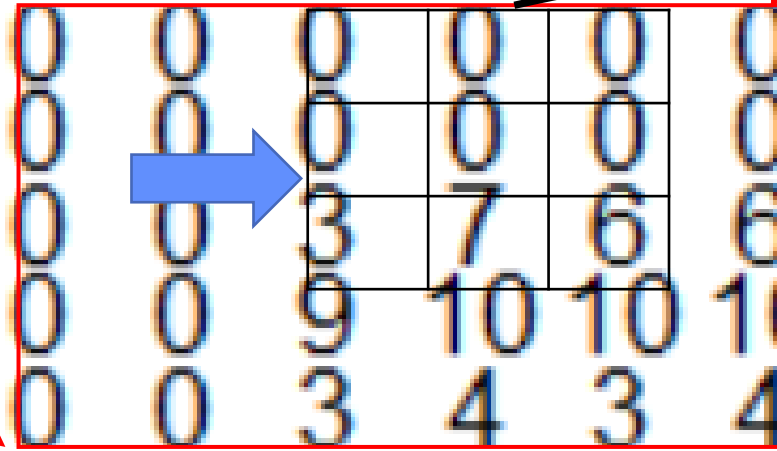
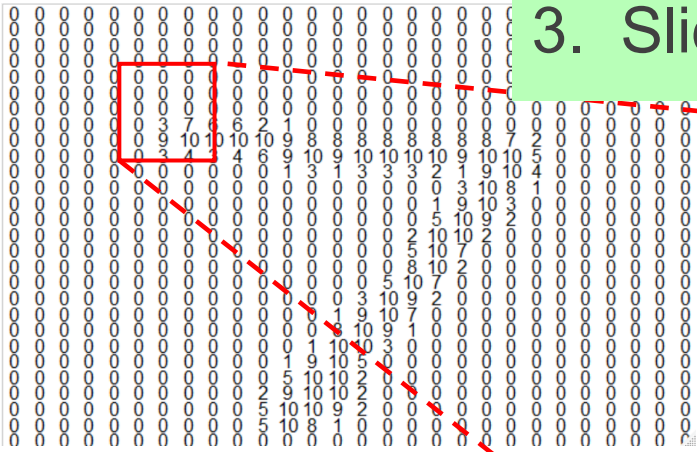
X

1. Multiply 3x3 patch of pixels with 3x3 filter "W"

2. Put sum in new cell of output map

3	7		

3. Slide filter horizontally to get next output value



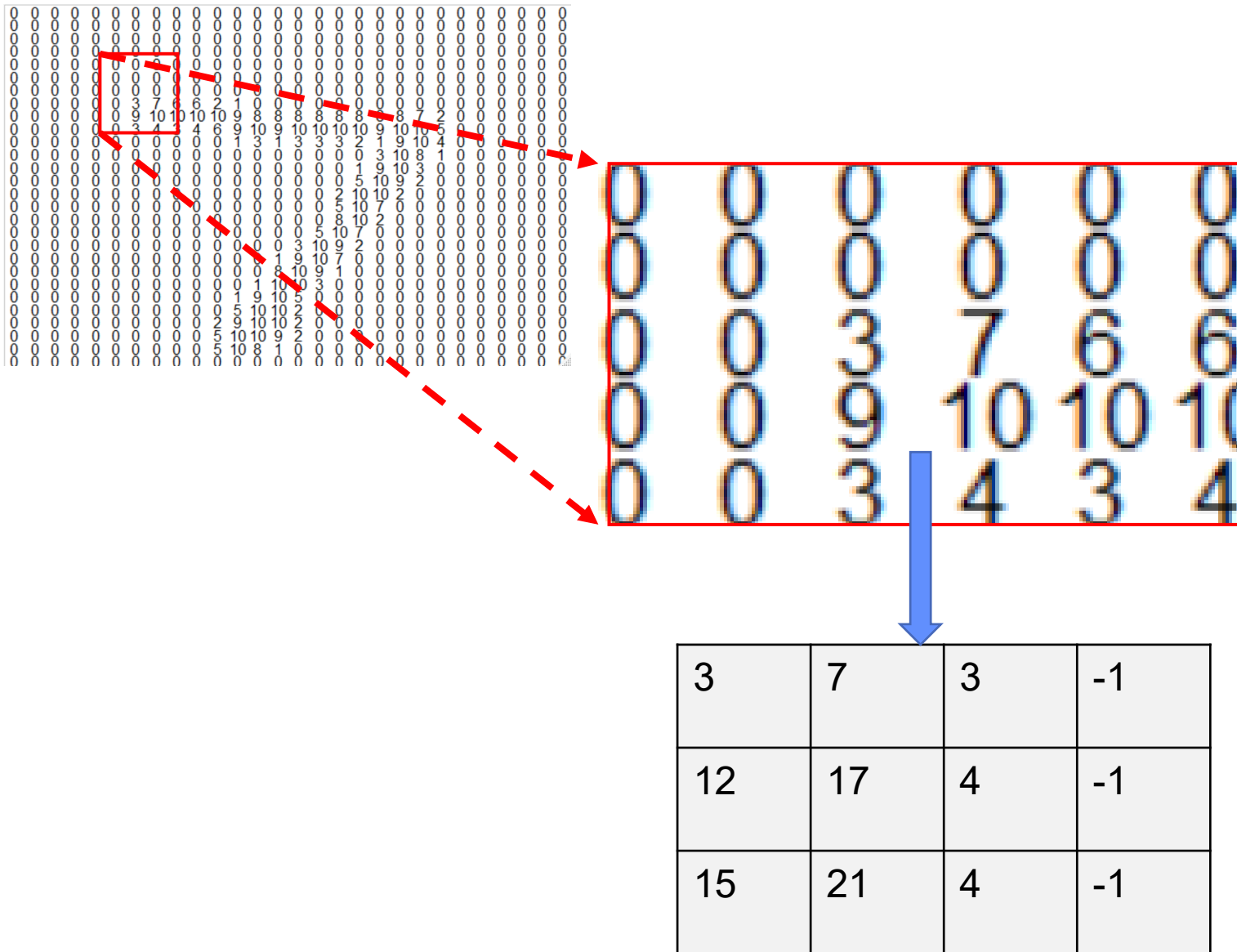
-1	0	+1
-1	0	+1
-1	0	+1

1. Multiply 3x3 patch of pixels with 3x3 filter "W"

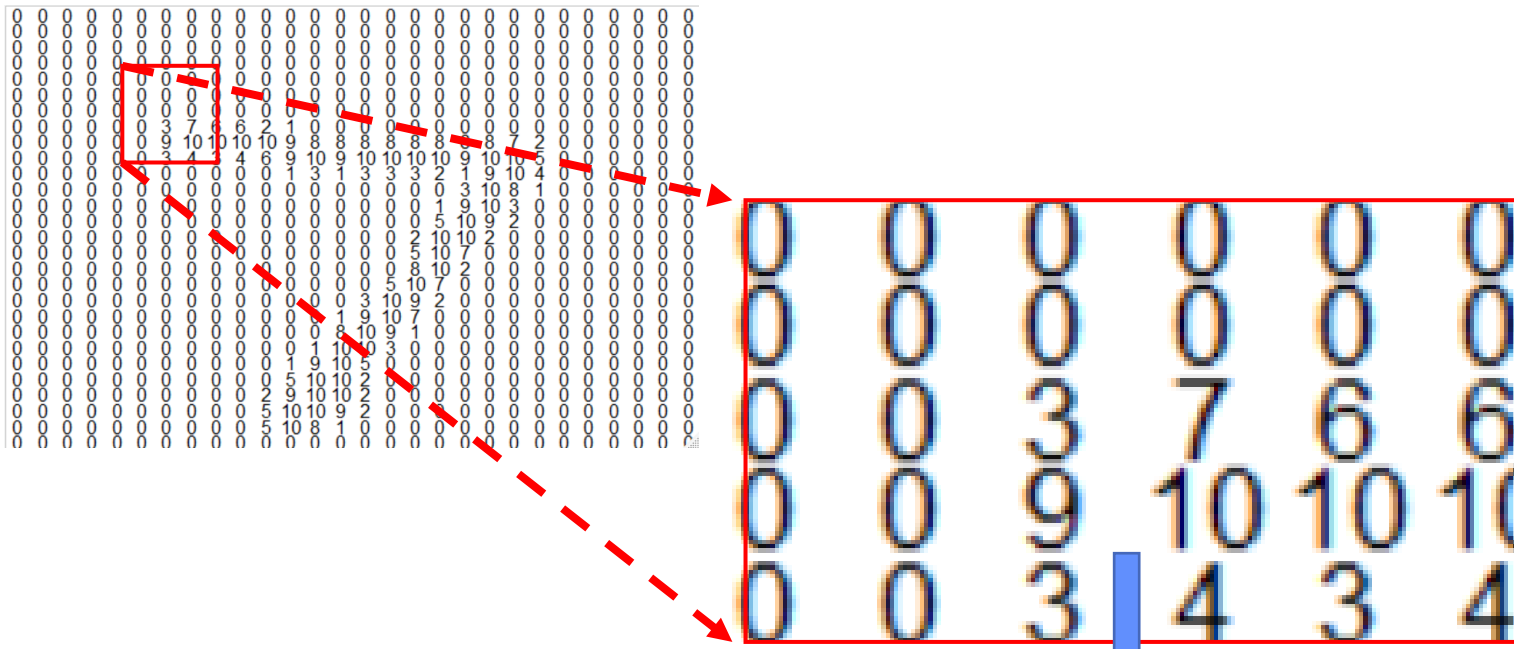
2. Put sum in new cell of output map

NOTE: sliding a filter is known as a "convolution" operation

3	7	3	



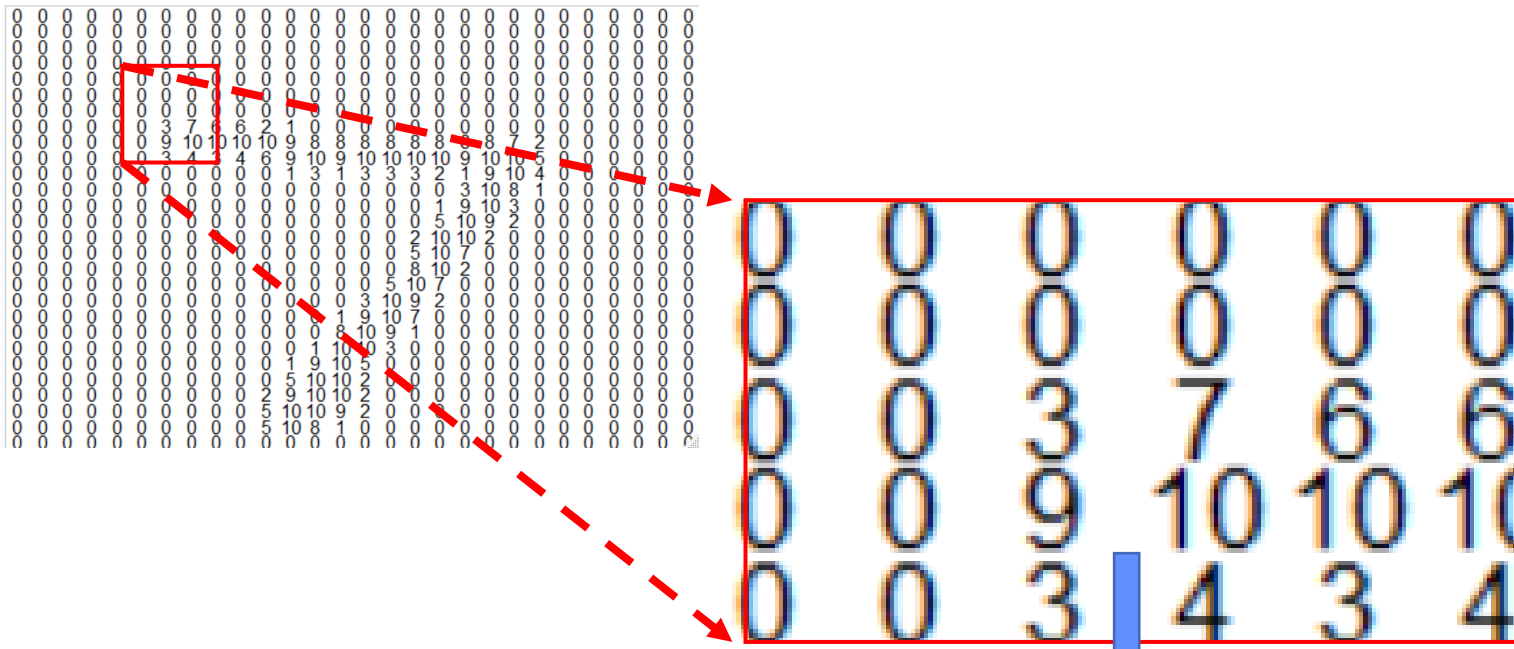
After vertical and horizontal sliding the 5x6 patch is now a 3x5 feature map.



3	7	3	-1
12	17	4	-1
15	21	4	-1

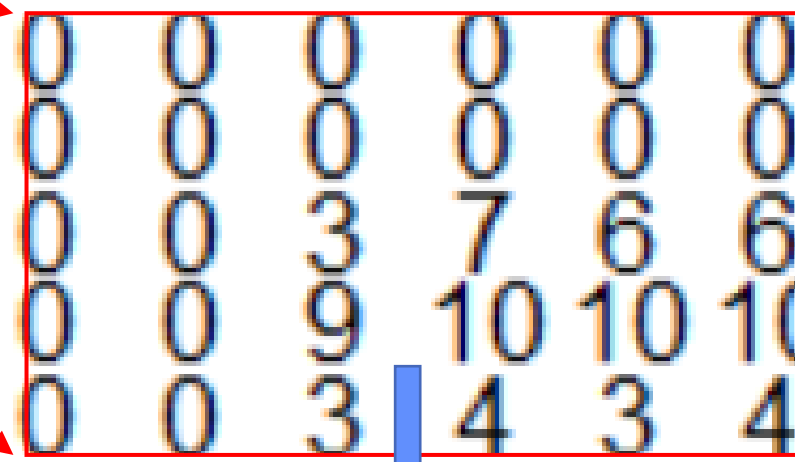
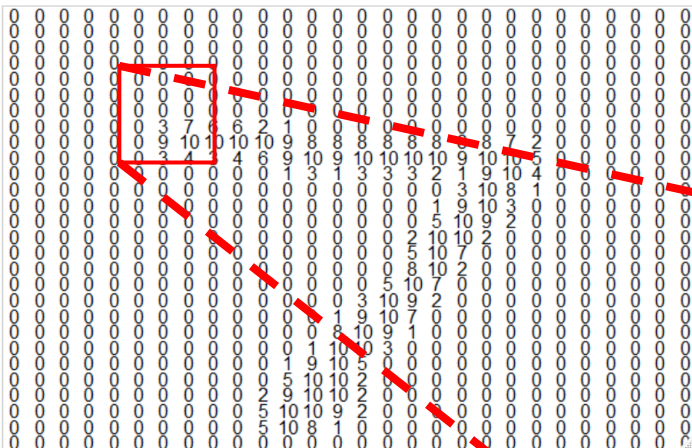
After vertical and horizontal sliding the 5x6 patch is now a 3x5 feature map.

What do the highest values in the feature map represent?



3	7	3	-1
12	17	4	-1
15	21	4	-1

Optional next step:
Use another filter, and take maximum over elements -
“max pooling”

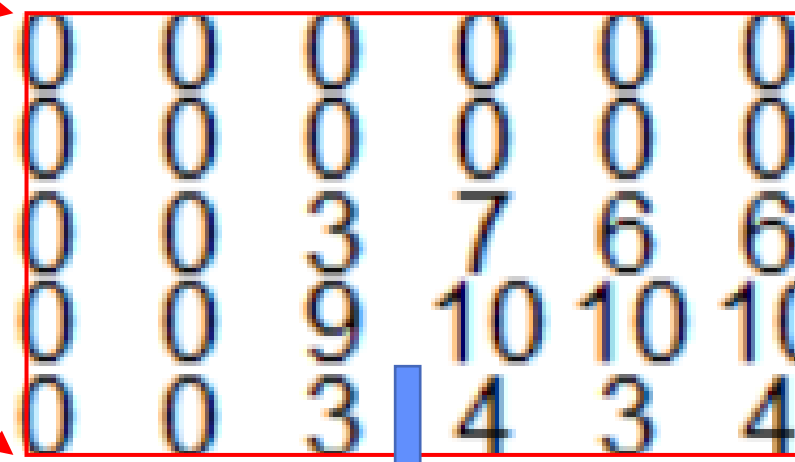
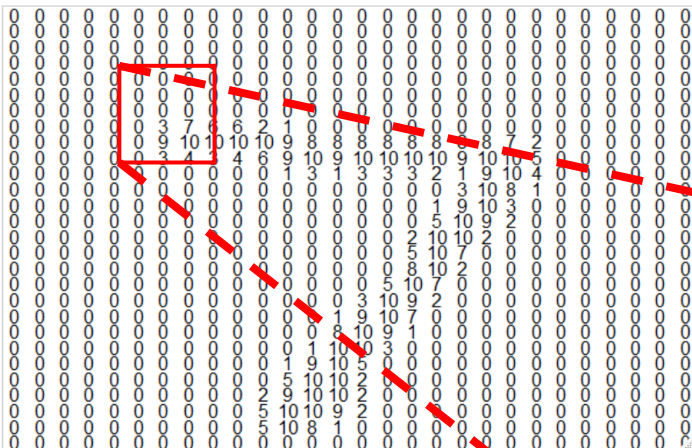


3	7	3	-1
12	17	4	1
15	21	4	-1

Optional next step:
Use another filter, and take maximum over elements -
“max pooling”

2x2 filter has max=17

17		

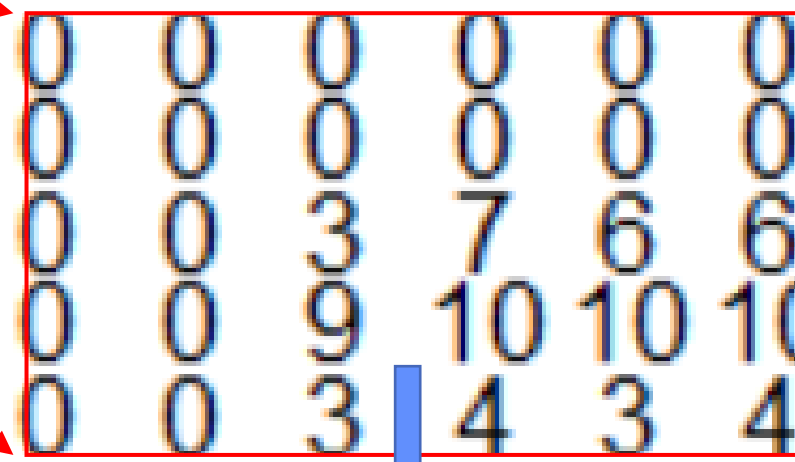
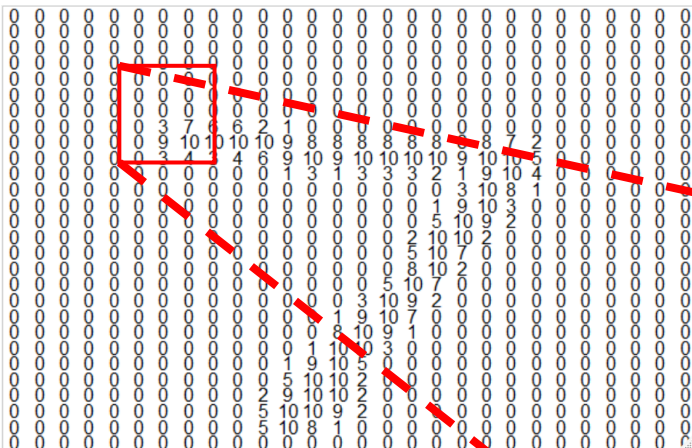


3	7	3	-1
12	17	4	-1
15	21	4	-1

Optional next step:
Use another filter, and take maximum over elements -
“max pooling”

Slide filter ...

17	17	4
21	21	4

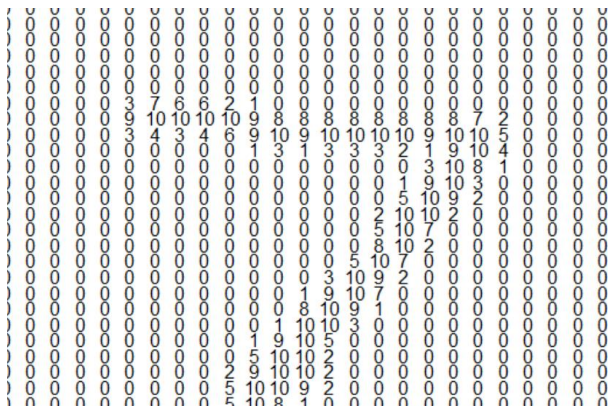
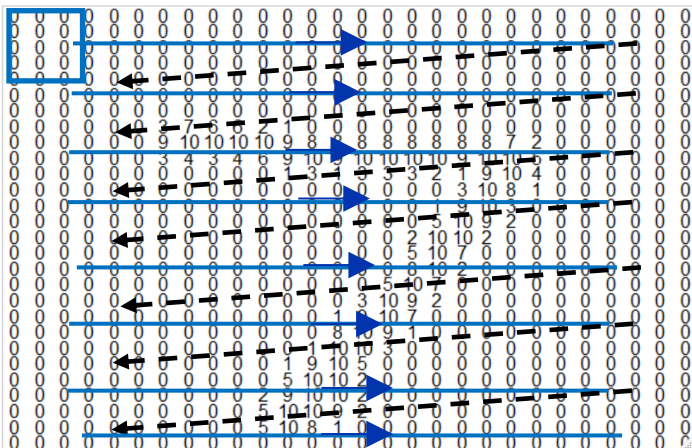


After convolution and pooling, the 5x6 patch is **transformed** into a 2x3 feature map of 'edge gradients'

3	7	3	-1
12	17	4	-1
15	21	4	-1

Slide filter ...

17	17	4
21	21	4



A convolution of one filter is applied to the entire image across and down.

*The entire 28x28 input is **transformed** into a smaller feature map of 'edge gradients'*

Pooling is optionally applied

Convolution Neural Network (CNN)

In CNNs the filter values are weight parameters that are learned (**feature discovery**)

W_{11}	W_{12}	W_{13}
W_{21}	W_{22}	W_{23}
W_{31}	W_{32}	W_{33}

Convolution Neural Network (CNN)

In CNNs the filter values are weight parameters that are learned (**feature discovery**)

W_{11}	W_{12}	W_{13}
W_{21}	W_{22}	W_{23}
W_{31}	W_{32}	W_{33}

A convolution layer is a set of feature maps, where each map is derived from convolution of 1 filter with input

Convolution Neural Network (CNN)

More hyperparameters:

Size of filter (smaller is more general)

Convolution Neural Network (CNN)

More hyperparameters:

- Size of filter (smaller, like 3x3, is more general)

- Number of pixels to slide over (1 or 2 is usually fine)

Convolution Neural Network (CNN)

More hyperparameters:

- Size of filter (smaller, like 3x3, is more general)

- Number of pixels to slide over (1 or 2 is usually fine)

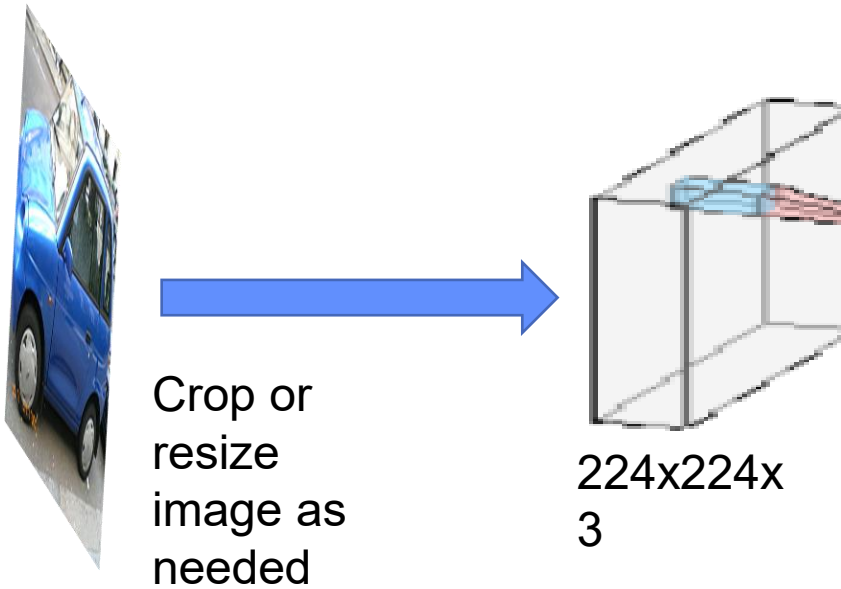
- Max pooling or not (usually some pooling layers)

- Number of filters (depends on the problem!)

- **A large CNN example**

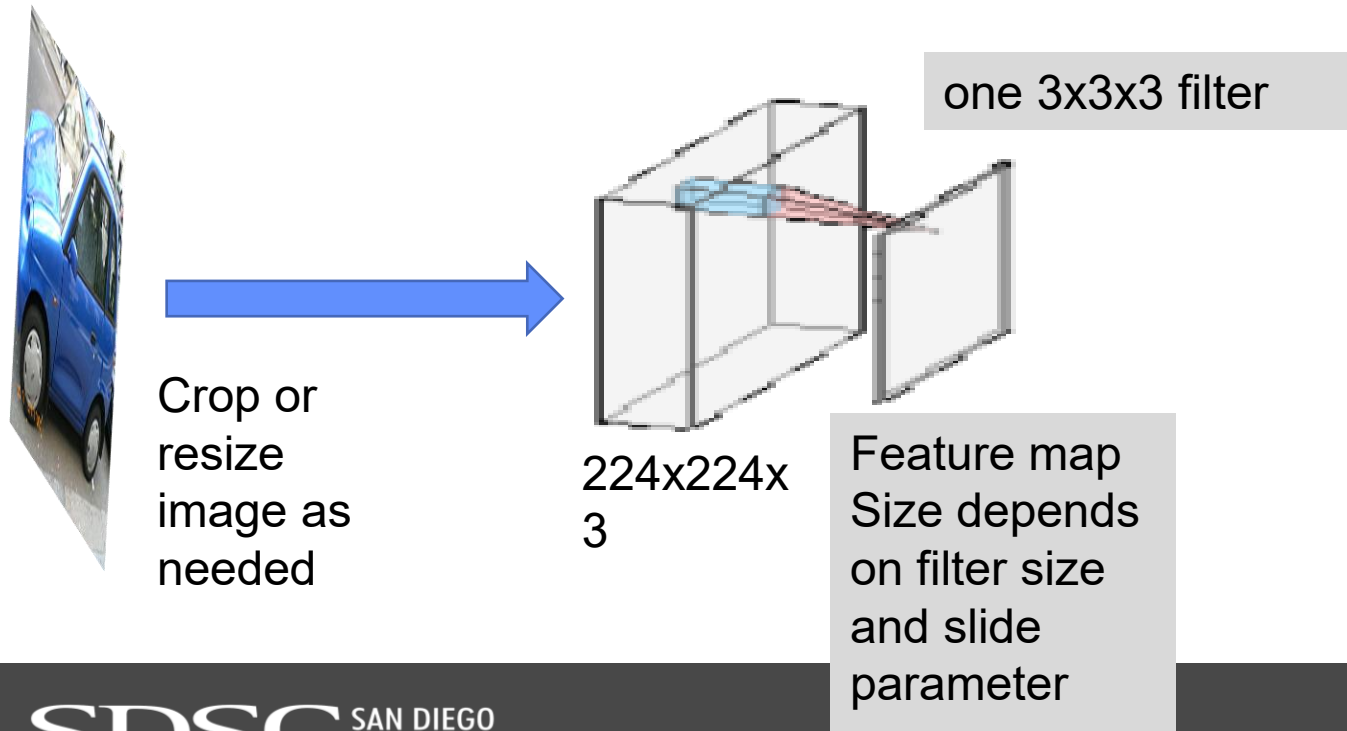
Convolution with image

- Make 1 layer, using HxWx3 image (3 for Red,Green,Blue channels)



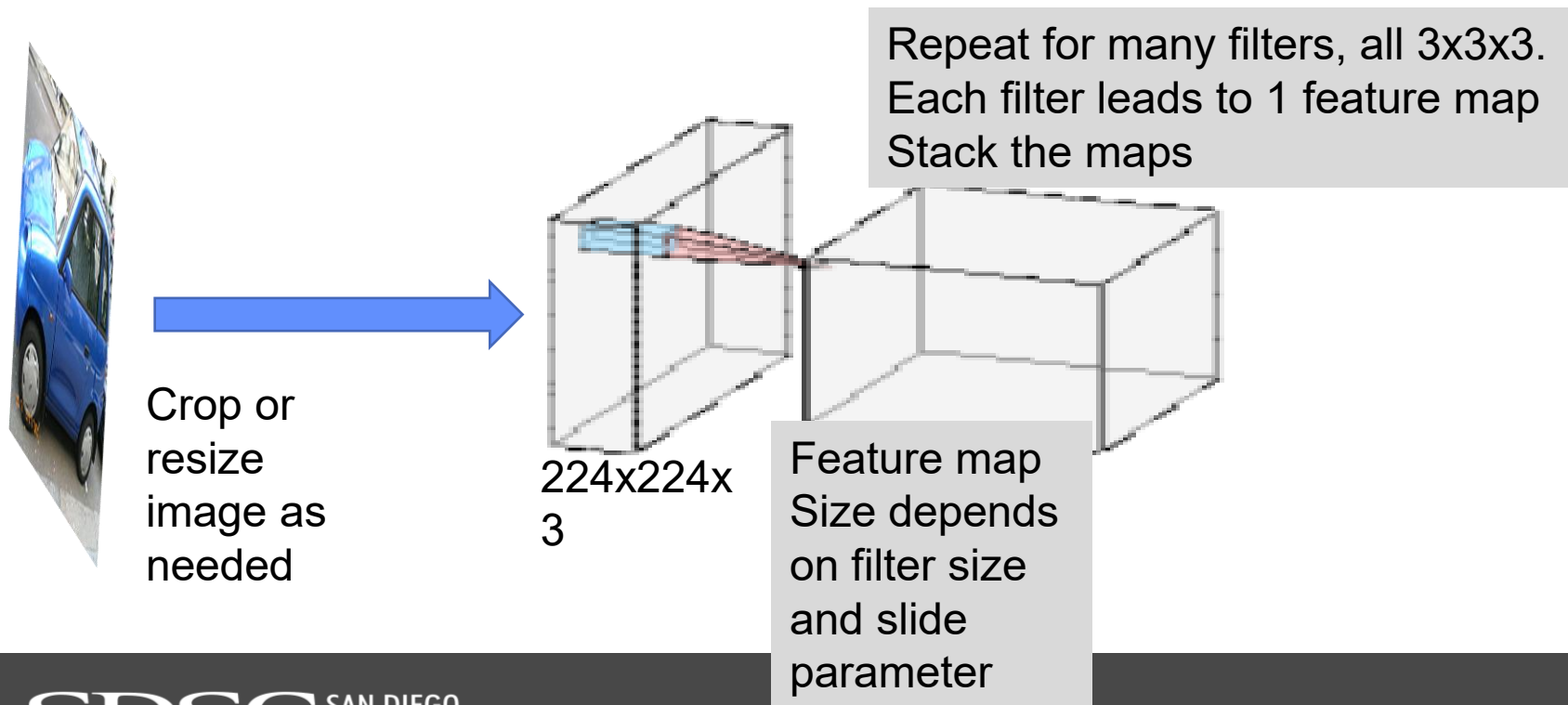
Convolution with image

- Make 1 layer, using HxWx3 image (3 for RGB channels)



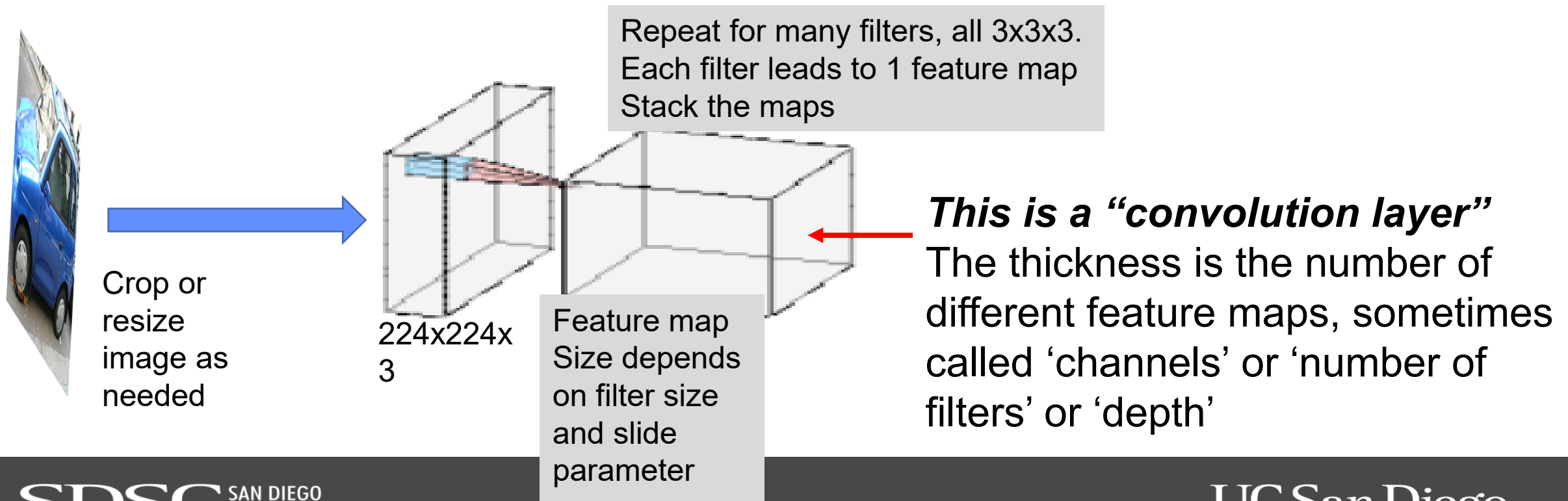
Convolution with image

- Make 1 layer, using $H \times W \times 3$ image (3 for RGB channels)



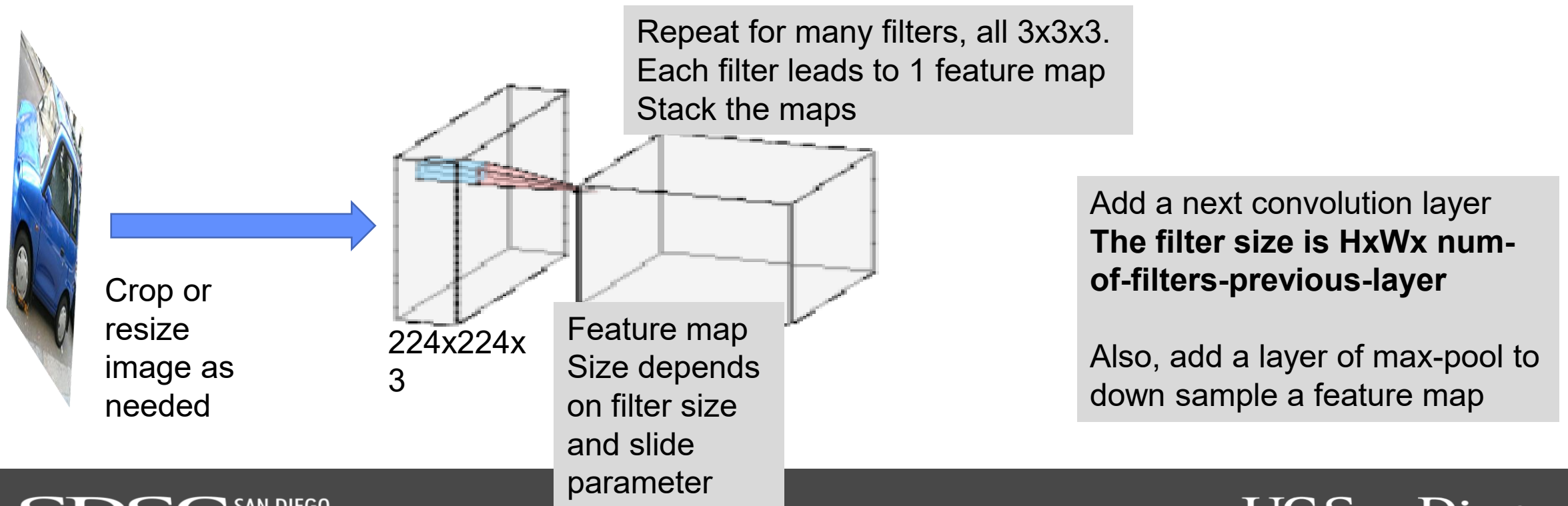
Convolution with image

- Make 1 layer, using HxWx3 image (3 for RGB channels)



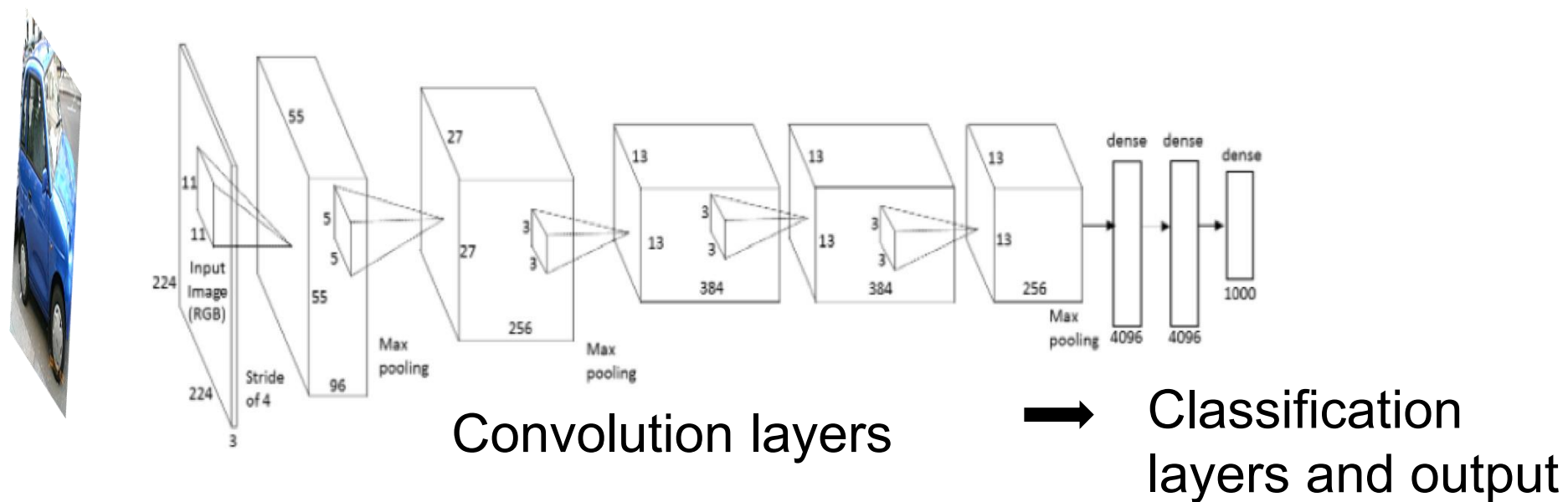
Convolution with image

- Make 1 layer, using $H \times W \times 3$ image (3 for RGB channels)

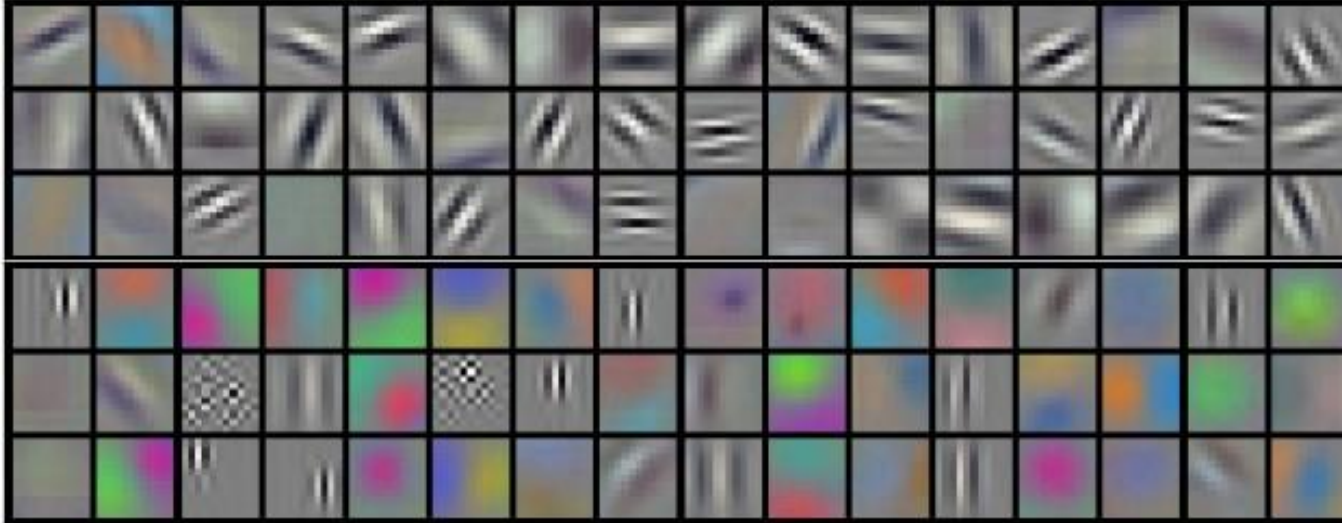


Large Scale Versions

- Large Convolution Networks – Alexnet, VGG19, ResNet, GoogLeNet, ...



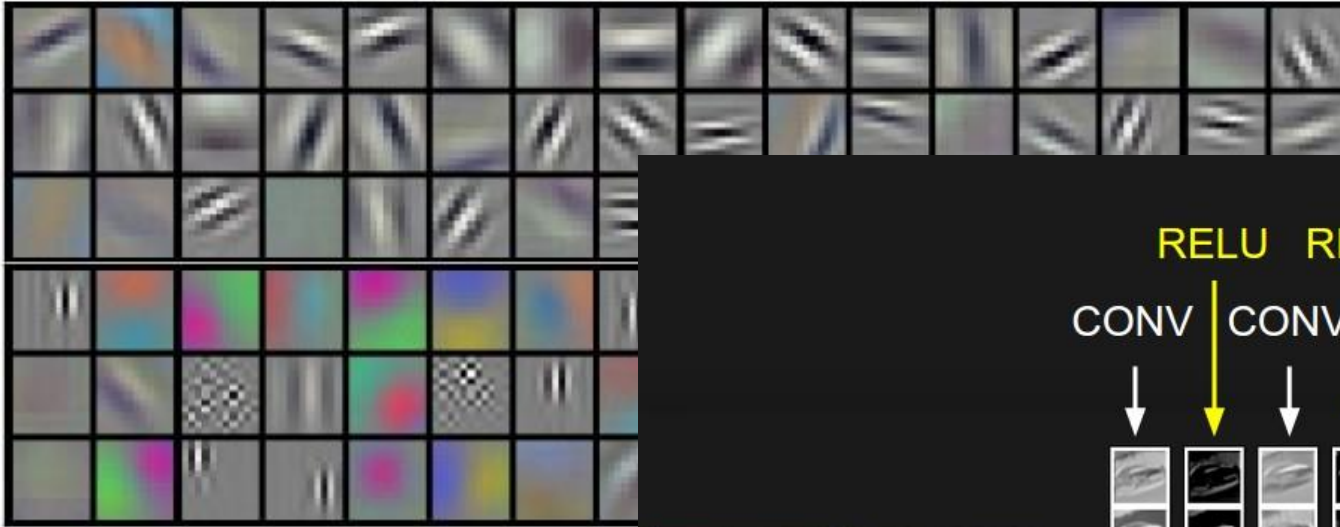
First convolution layer filters are simple features



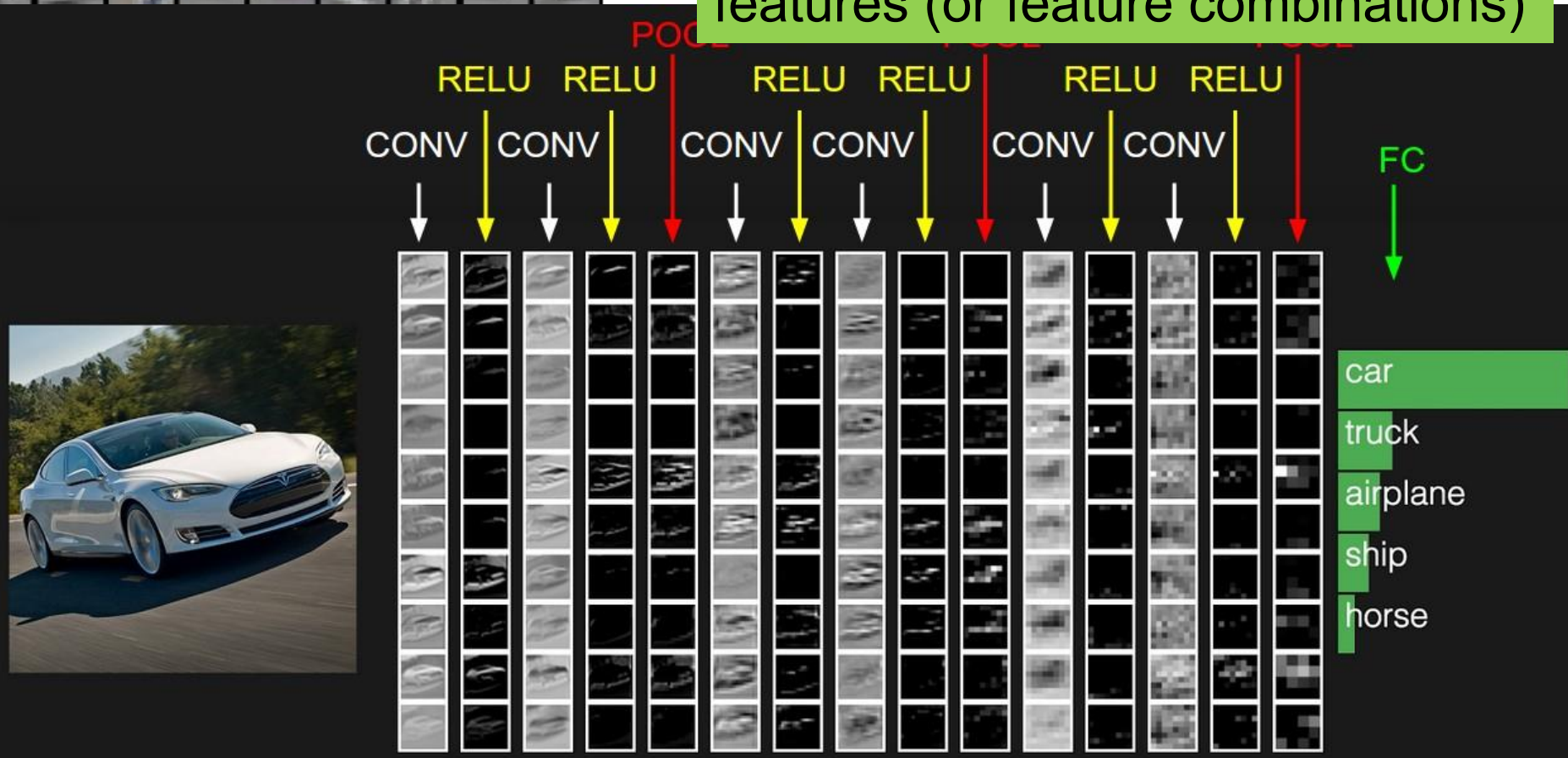
What Learned
Convolutions
Look Like

What Learned Convolutions

First convolution layer filters are simple features



Higher layers are more abstract features (or feature combinations)



Convolution Neural Network Summary

CNNs work because convolution layers have a special architecture and function – it is biased to do certain kind of transformations

Low layers have less filters that represent simple local features for all classes

Higher layers have more filters that cover large regions that represent object class features

What is deep learning?

Deep learning refers to learning complex and varied transformations of the input

Deep learning refers to **discovering** useful features of the input

Deep learning is a neural network with many layers

- **Next, notebook demo**

Exercise CNN for Digit Classification

- The 'hello world' of CNNs
- It uses MNIST dataset and Pytorch coding
- First: Quick intro to Pytorch and CNN layers coding
- Second: We will login and start a notebook



Python code for a 'Sequential' neural network

```
def MyNet(numfilt):  
    model=torch.nn.Sequential(  
        #Conv:  input size 1 channel, output is number of filters, the  
        # actual batch of input is implicit  
        # see:  https://docs.pytorch.org/docs/stable/generated/torch.n  
        torch.nn.Conv2d(in_channels=1,out_channels=numfilt,  
                        kernel_size=kernel_size2use,stride=1),  
        torch.nn.ReLU(),  
        ...  
    )
```

*Add convolution layer,
relu activation,
Max pooling layer*

Python code for a 'Sequential' neural network

```
def MyNet(numfilt):  
    model=torch.nn.Sequential(  
        #Conv:  input size 1 channel, output is number of filters, the  
        # actual batch of input is implicit  
        # see:  https://docs.pytorch.org/docs/stable/generated/torch.n  
        torch.nn.Conv2d(in_channels=1,out_channels=numfilt,  
                        kernel_size=kernel_size2use,stride=1),  
        torch.nn.ReLU(),  
        ...  
    )
```

*Add convolution layer,
relu activation,
Max pooling layer*

*Images are 1x28x28,
so input is 1 channel of a 2D image*

Output will be given by 'num filters' argument

*Output matrix shape will depend on filter size
(kernel size) and stride*

Python code for a 'Sequential' neural network

```
def MyNet(numfilt):  
    model=torch.nn.Sequential(  
        #Conv:  input size 1 channel, output is number of filters, the  
        # actual batch of input is implicit  
        # see:  https://docs.pytorch.org/docs/stable/generated/torch.n  
        torch.nn.Conv2d(in_channels=1,out_channels=numfilt,  
                        kernel_size=kernel_size2use, stride=1),  
        torch.nn.ReLU(),  
        torch.nn.MaxPool2d(3,2),  
        torch.nn.Flatten(),  
        torch.nn.Linear(numfilt*reduced_size*reduced_size,16), #after m  
        torch.nn.ReLU(),  
        torch.nn.Linear(16, 10),  
        torch.nn.LogSoftmax(dim=1))  
    return model
```

*Add convolution layer,
relu activation,
Max pooling layer*

*Then flatten into a vector
for classification layers*

Python code for a 'Sequential' neural network

```
def MyNet(numfilt):  
    model=torch.nn.Sequential(  
        #Conv:  input size 1 channel, output is number of filters, the  
        # actual batch of input is implicit  
        # see:  https://docs.pytorch.org/docs/stable/generated/torch.n  
        torch.nn.Conv2d(in_channels=1,out_channels=numfilt,  
                        kernel_size=kernel_size2use,stride=1),  
        torch.nn.ReLU(),  
        torch.nn.MaxPool2d(3,2),  
        torch.nn.Flatten(),  
        torch.nn.Linear(numfilt*reduced_size*reduced_size,16), #after m  
        torch.nn.ReLU(),  
        torch.nn.Linear(16, 10),  
        torch.nn.LogSoftmax(dim=1))  
    return model
```

*Add convolution layer,
relu activation,
Max pooling layer*

*Then flatten into a vector
for classification layers*

- Every layer has some input, output
- Network layers need input and output size (e.g. channels or hidden units)
- Not every layer/function has trainable parameters – like which one of these?

Python code for a functional (non-sequential) neural network

```
# -----  
class MyNet(torch.nn.Module):  
    def __init__(self):  
        super(MyNet, self).__init__()
```

*Pytorch 'nn' class objects
are very flexible*

'MyNet' inherits functionality from 'torch.nn' package

Python code for a functional (non-sequential) neural network

```
# -----  
class MyNet(torch.nn.Module):  
    def __init__(self):  
        super(MyNet, self).__init__()  
        #Conv:  input size 1 channel, output is number of filters, the  
        # actual batch of input is implicit  
        # see:  https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html  
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=numfilt, kernel_size=  
        self.linear1 = torch.nn.Linear(numfilt*reduced_size*reduced_size, 16) #after m  
        self.linear2 = torch.nn.Linear(16, 10)
```

*Pytorch 'nn' class objects
are very flexible*

*In the initialization you
define layer functions*

'MyNet' inherits functionality from 'torch.nn' package

Python code for a functional (non-sequential) neural network

```
# -----  
class MyNet(torch.nn.Module):  
    def __init__(self):  
        super(MyNet, self).__init__()  
        #Conv: input size 1 channel, output is number of filters, the  
        # actual batch of input is implicit  
        # see: https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html  
        self.conv1 = torch.nn.Conv2d(in_channels=1,out_channels=numfilt,kernel_size=  
        self.linear1 = torch.nn.Linear(numfilt*reduced_size*reduced_size,16) #after max pool  
        self.linear2 = torch.nn.Linear(16, 10)  
  
    def forward(self, x):  
        x = self.conv1(x)  
        x = F.relu(x)  
        x = F.max_pool2d(x, 3, 2)  
  
        # <<<<<<<<<<<<-----  
        #Uncomment this to see what the size actually is after max pooling  
        #print('MYINFO fwd, after max, x shape:',x.shape)  
  
        x = torch.flatten(x, 1)  
        x = self.linear1(x)  
        x = F.relu(x)  
        x = self.linear2(x)  
        #not sure i need this x = F.relu(x)  
        output = F.log_softmax(x, dim=1) #log softmax for classfct or binary?  
        return output
```

Pytorch 'nn' class objects are very flexible

In the initialization you define layer functions

In the forward function you indicate the inputs and lay out the model

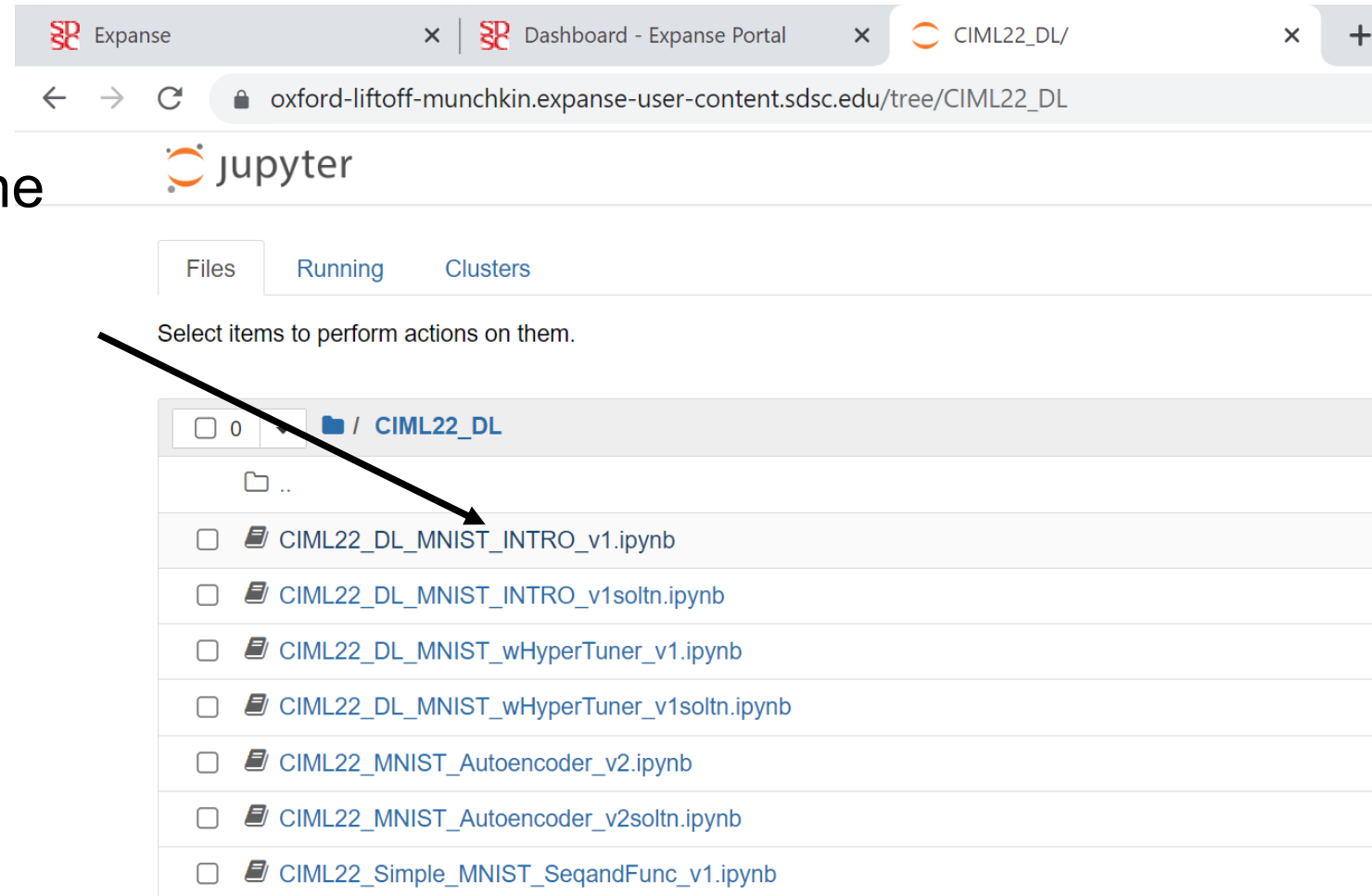
You can mix layers and torch functions in flexible ways

Pytorch coding notes:

\$ jupyter-gpu-shared-pylight

In jupyter notebook session open the
CIML_MNIST_Intro notebook

Follow instructions in the notebook



Notebook code overview

1. Import statements
2. Define parameters (arguments)
3. Get data and set up Pytorch dataloader

Notebook code overview

1. Import statements
2. Define parameters (arguments)
3. Get data and set up Pytorch dataloader

the loader is a Python iterator to help retrieve a batch of data at a time

```
dataset1 = datasets.MNIST(data_path, train=True, download=True, transform=transform)
dataset2 = datasets.MNIST(data_path, train=False, download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(dataset1,
                                             batch_size=batch_size, sampler=None,
                                             num_workers=num_worker2use, pin_memory=True, drop_last=True)
test_loader = torch.utils.data.DataLoader(dataset2,
                                             batch_size=batch_size, sampler=None,
                                             num_workers=num_worker2use, pin_memory=True, drop_last=True)
```

Notebook code overview

1. Import statements
2. Define parameters (arguments)
3. Get data and set up Pytorch dataloader
4. Define functions to create network and do a forward pass
5. Initialize model, move to GPU device (if available)

Notebook code overview

1. Import statements
2. Define parameters (arguments)
3. Get data and set up Pytorch dataloader
4. Define functions to create network and do a forward pass
5. Initialize model, move to GPU device (if available)

A gpu node has 4 gpu devices (cards). For single device execution just set to 'current_device'.

```
use_cuda = torch.cuda.is_available()
if use_cuda:
    num_gpu = torch.cuda.device_count()
    print('INFO,  cuda, num gpu:', num_gpu)
    device = torch.cuda.current_device()
```

Notebook code overview

1. Import statements
2. Define parameters (arguments)
3. Get data and set up Pytorch dataloader
4. Define functions to create network and do a forward pass
5. Initialize model, move to GPU device (if available)

A gpu node has 4 gpu devices (cards). For single device execution just set to 'current_device'.

We also move model and data to device before training

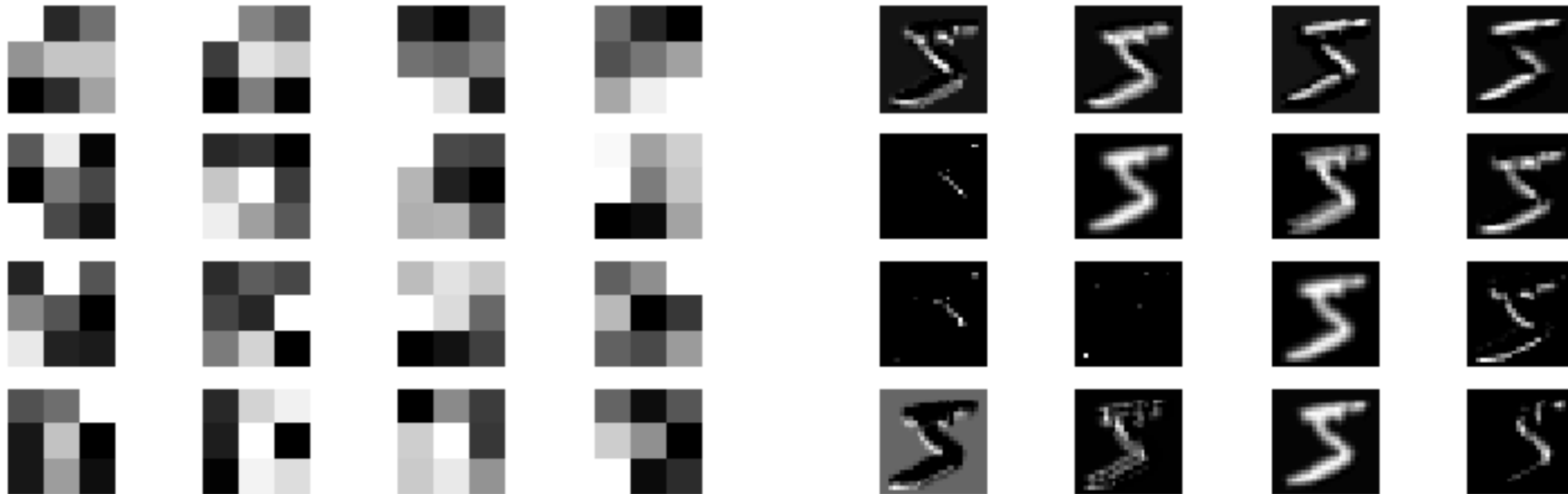
```
use_cuda = torch.cuda.is_available()
if use_cuda:
    num_gpu = torch.cuda.device_count()
    print('INFO,  cuda, num gpu:', num_gpu)
    device = torch.cuda.current_device()
```

```
my_model = MyNet().to(device)
```

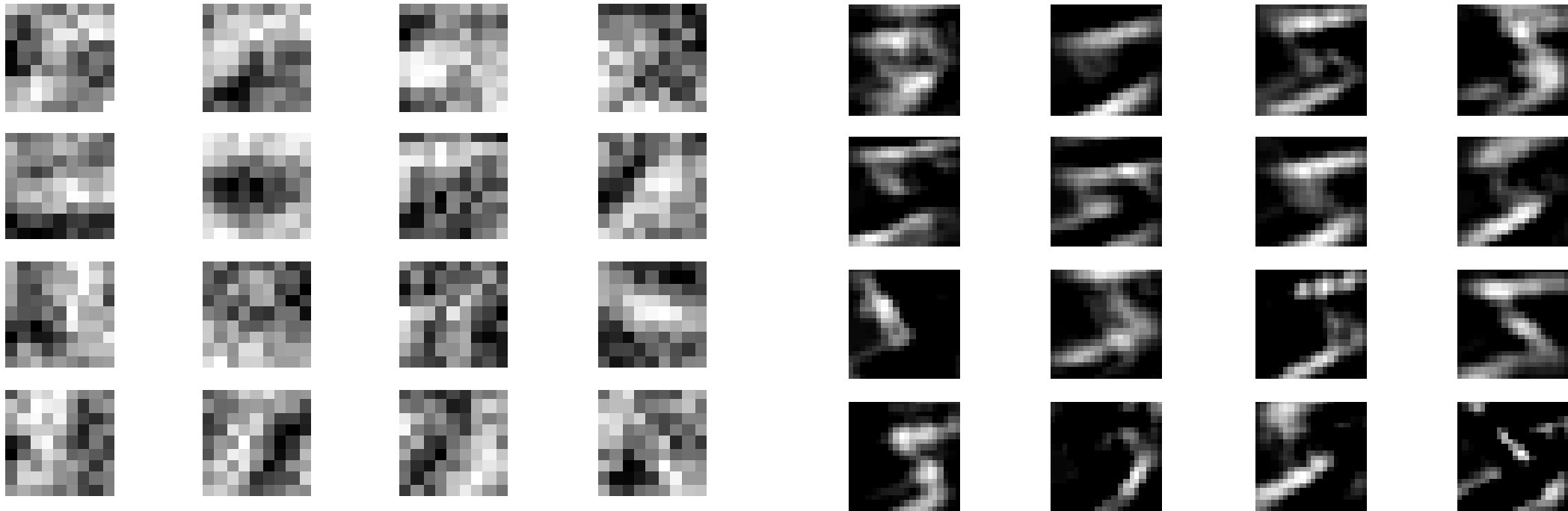
Notebook code overview

1. Import statements
2. Define parameters (arguments)
3. Get data and set up Pytorch dataloader
4. Define functions to create network and do a forward pass
5. Initialize model, move to GPU device (if available)
6. Run training loops of forward pass, get loss, run loss.backward (Pytorch keeps track of gradients)
7. Display results

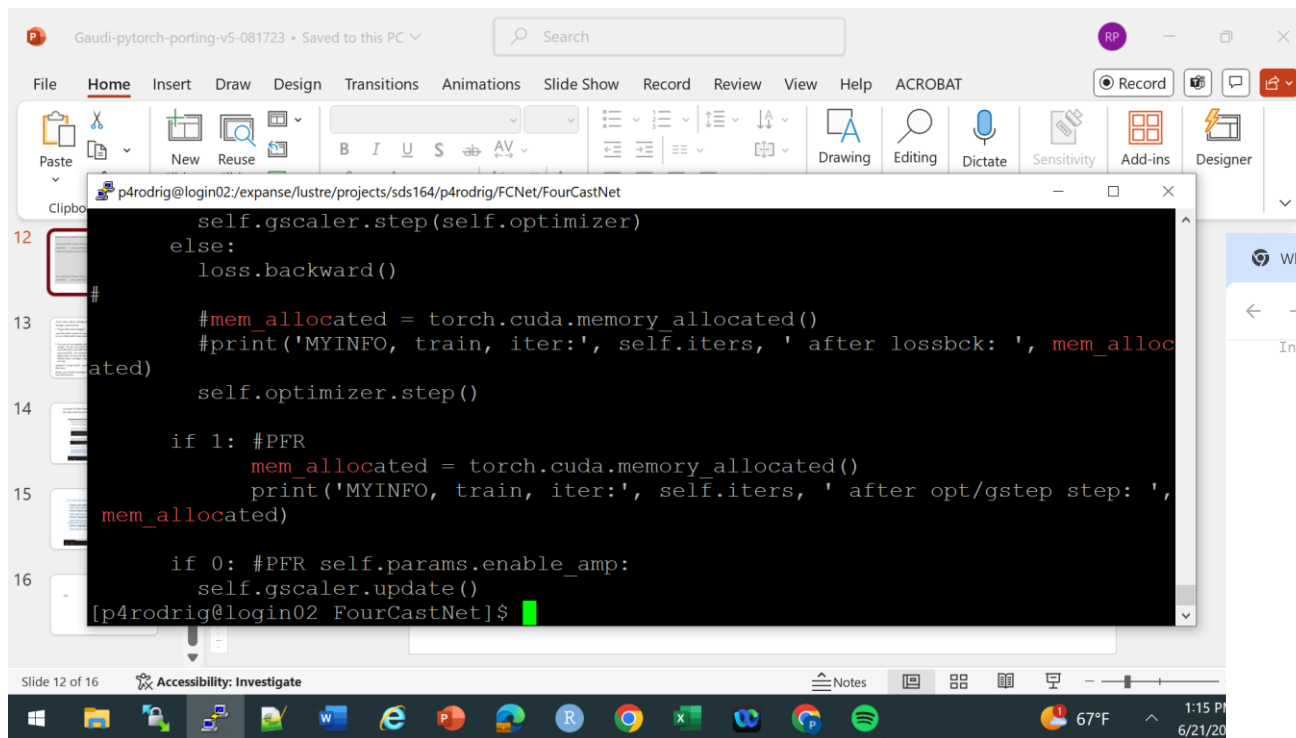
Exercise notes: 3x3 first convolution layer filter and activation



9x9 first convolution layer filter and activation



End



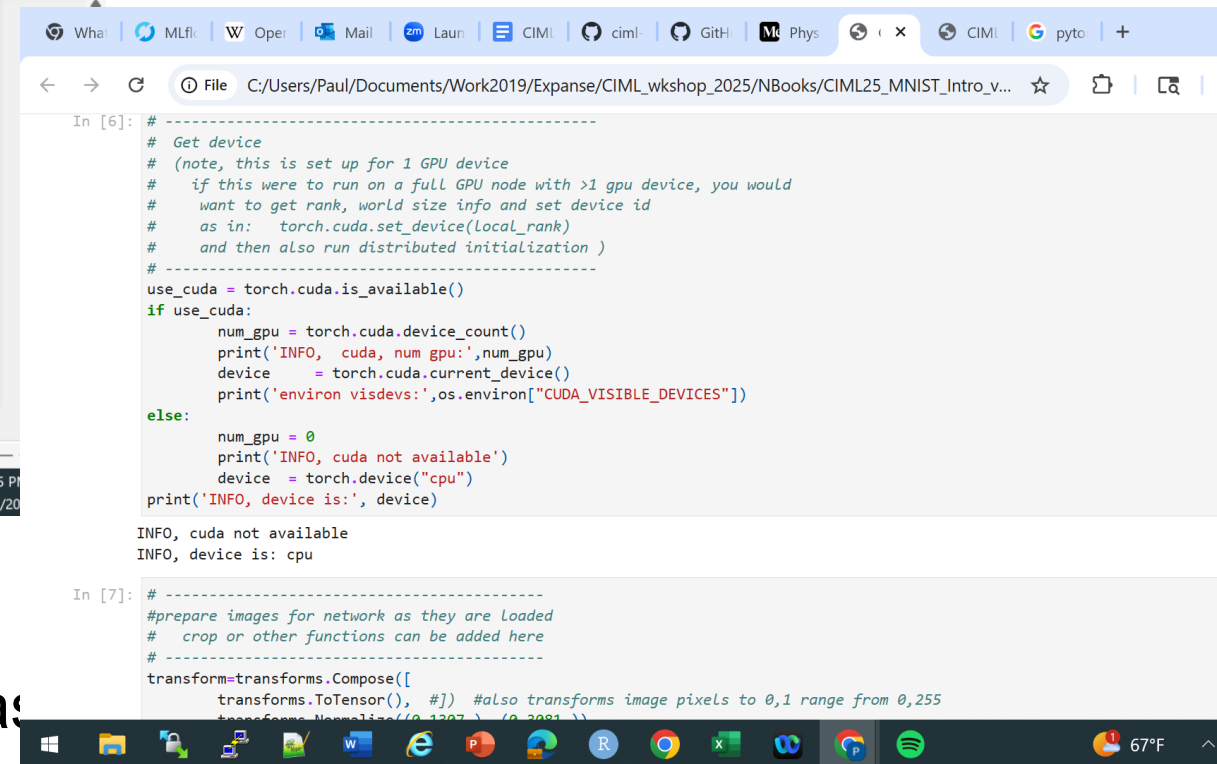
```
self.gscaler.step(self.optimizer)
else:
    loss.backward()

#
#mem_allocated = torch.cuda.memory_allocated()
#print('MYINFO, train, iter:', self.iters, ' after lossbck: ', mem_alloc
ated)

self.optimizer.step()

if 1: #PFR
    mem_allocated = torch.cuda.memory_allocated()
    print('MYINFO, train, iter:', self.iters, ' after opt/gstep step: ',
mem_allocated)

if 0: #PFR self.params.enable_amp:
    self.gscaler.update()
[p4rodrig@login02 FourCastNet]$
```



```
# Get device
# (note, this is set up for 1 GPU device
# if this were to run on a full GPU node with >1 gpu device, you would
# want to get rank, world size info and set device id
# as in: torch.cuda.set_device(local_rank)
# and then also run distributed initialization )
# -----
use_cuda = torch.cuda.is_available()
if use_cuda:
    num_gpu = torch.cuda.device_count()
    print('INFO, cuda, num gpu:', num_gpu)
    device = torch.cuda.current_device()
    print('environ visdevs:', os.environ["CUDA_VISIBLE_DEVICES"])
else:
    num_gpu = 0
    print('INFO, cuda not available')
    device = torch.device("cpu")
print('INFO, device is:', device)

INFO, cuda not available
INFO, device is: cpu

In [7]: # -----
#prepare images for network as they are loaded
# crop or other functions can be added here
# -----
transform=transforms.Compose([
    transforms.ToTensor(), #] #also transforms image pixels to 0,1 range from 0,255
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

On a Gaudi node, all devices are known as
Nvidia-smi