

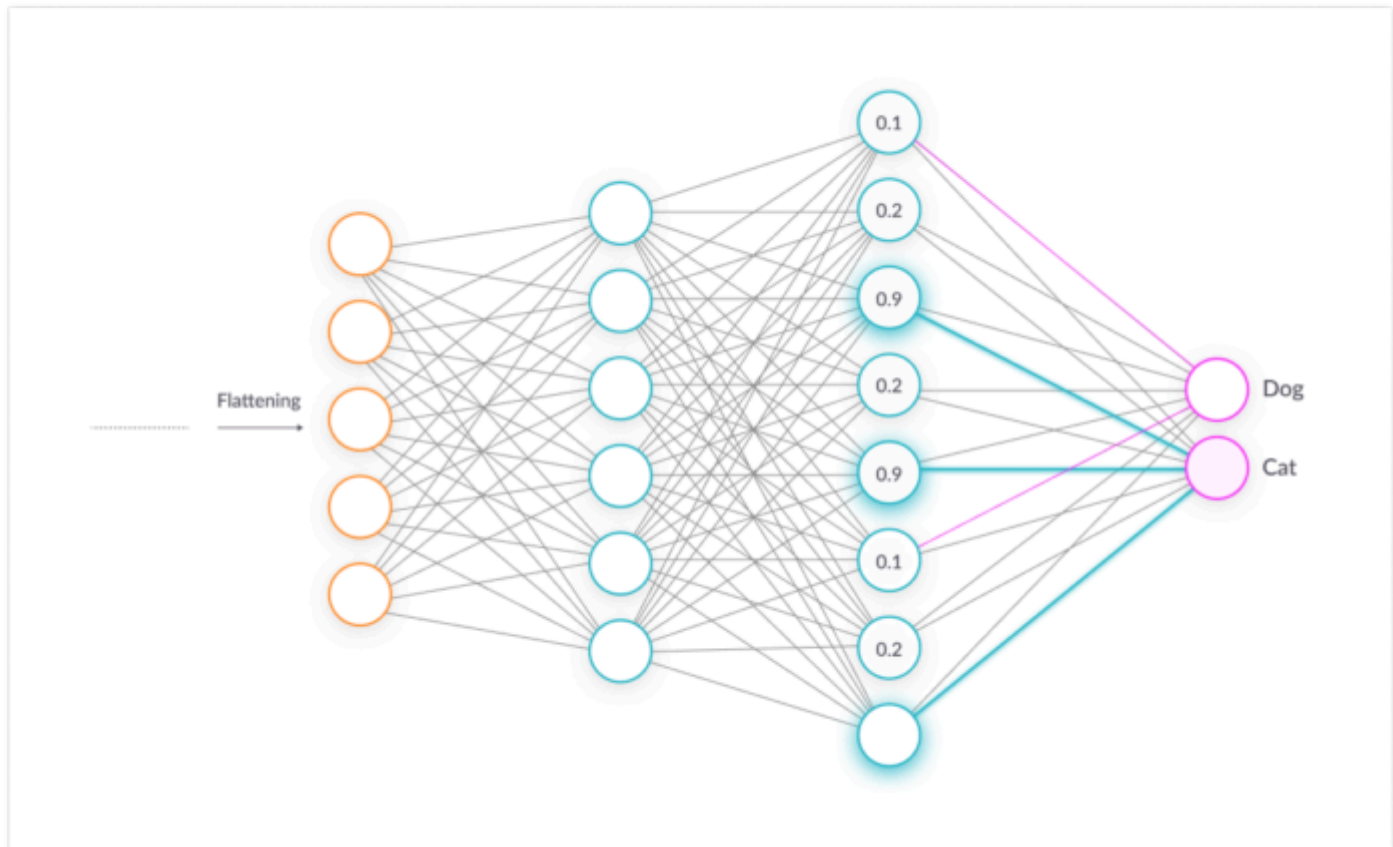
# S4

- Due No Due Date
- Points None
- Available after Feb 9, 2020 at 9:25am

## *SESSION 4 – ARCHITECTURAL BASICS*

### FULLY CONNECTED LAYERS





Each line you see is the **weight** we are training. Those circles are "temporary" values that will be stored. Once you train the model, *lines are what all matter!*

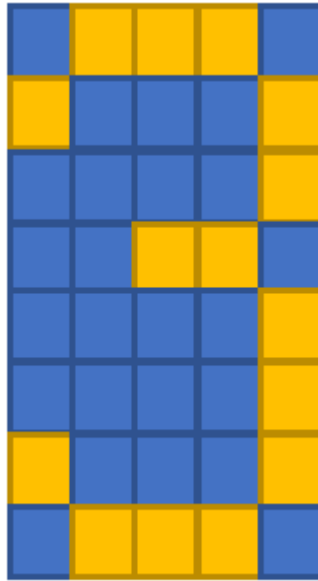


What digit do you think this pattern represents?

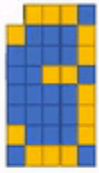


*Exactly, that's the point.*

This is what it actually represents.

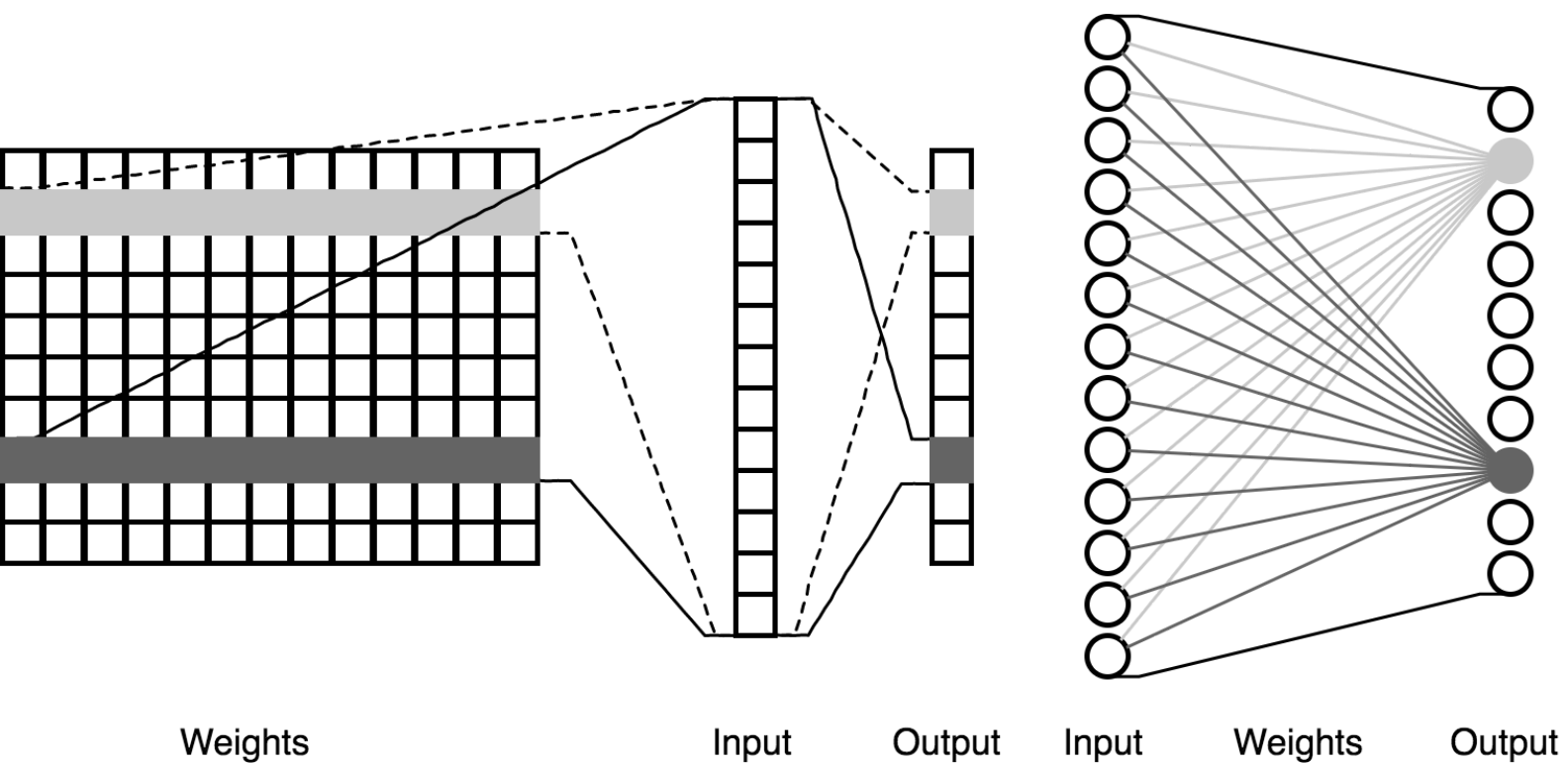


The process which converts our 2D data into 1D is shown below:

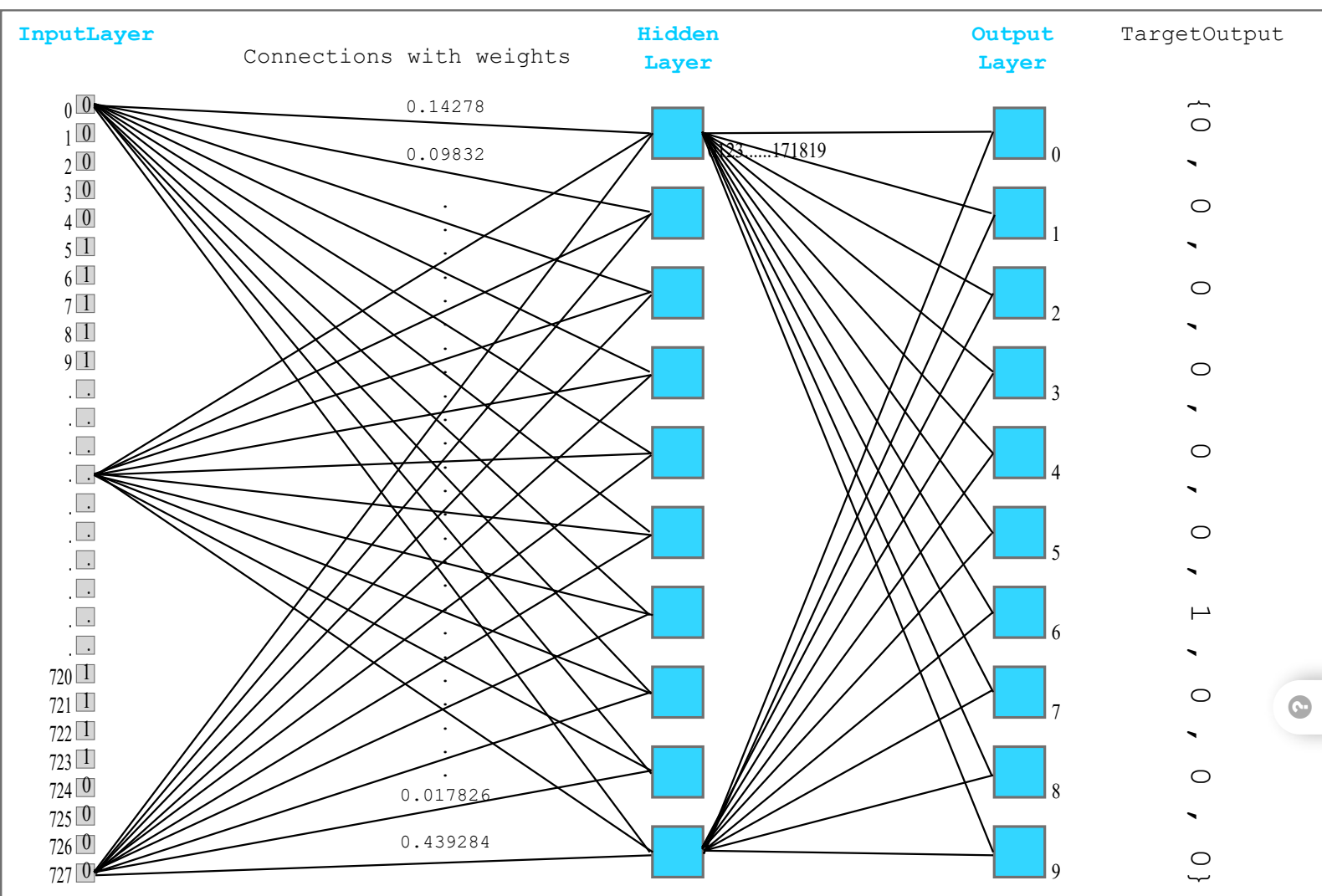


As you can see we lose the spatial information when we use fully connected layers.

*This is how weights and multiplications work in case of FC layers:*

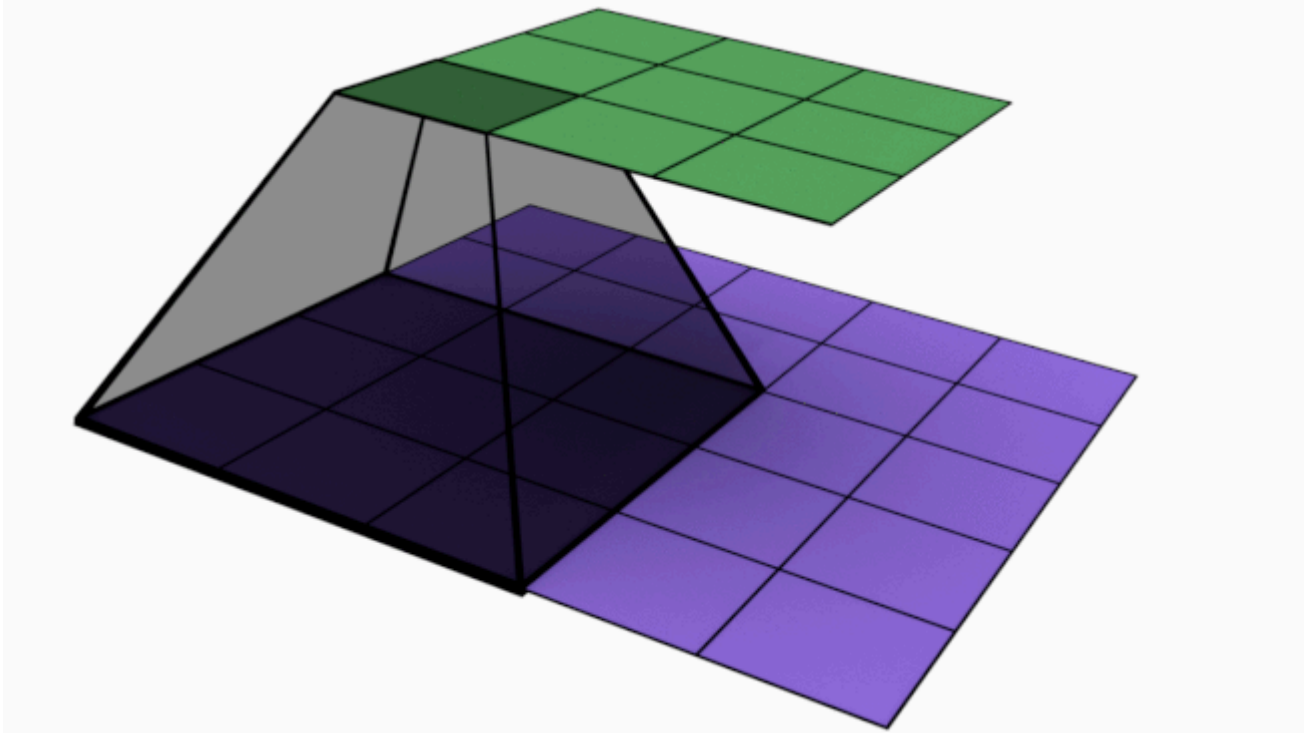


*Let's talk about this network below:*



This is a simple three-layer network, input neuron layer, hidden neuron layer, and output neuron layers.

*Getting old, and but still the new kid on the block*



## THE MODERN ARCHITECTURE



What we have covered (expand-squeeze channels) as architecture is for understanding only. This architecture is called Squeeze and Expand.

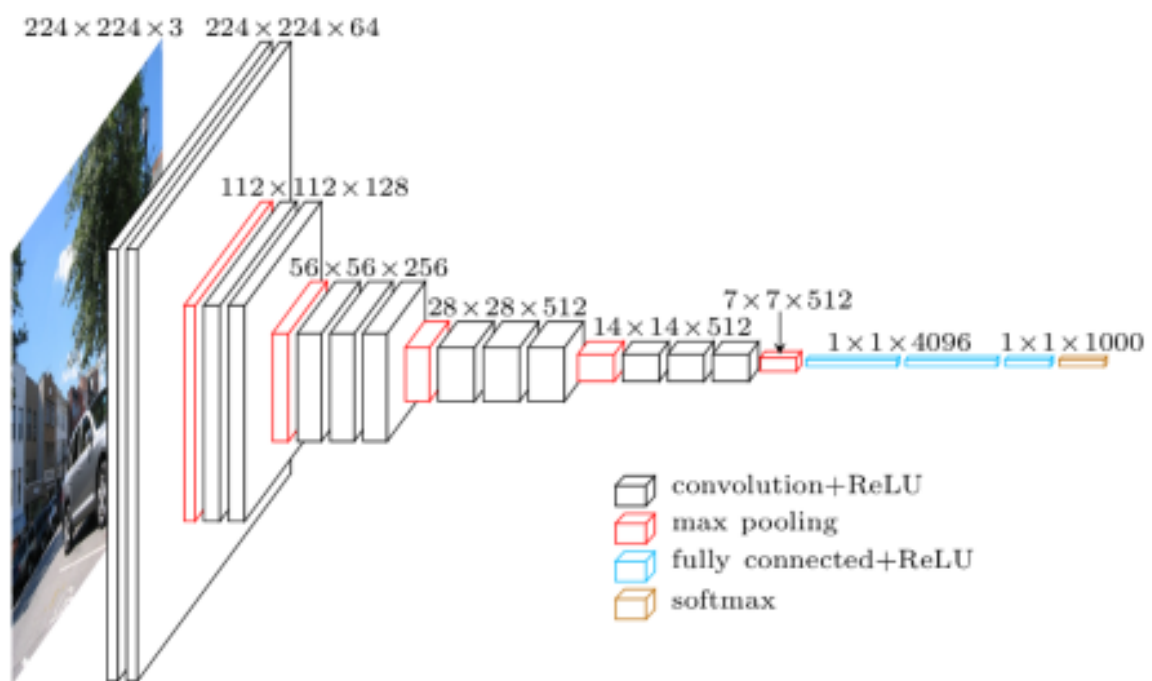
Most of the modern architecture follows this architecture:



#### Major Points:

1. ResNet is the latest among the above. You can clearly note 4 major blocks.
2. The total number of kernels increases from  $64 > 128 > 256 > 512$  as we proceed from the first block to the last (unlike what we discussed where at each block we expand to 512. Both architectures are correct, but  $64 \dots 512$  would lead in lesser computation and parameters.
3. Only the most advanced networks have ditched the FC layer for the GAP layer. In TSAI we will only use GAP Layers.
4. Nearly every network starts from 56x56 resolution!

*Before 2014*



Let's look at the parameters



Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 224, 224, 3)	0	
block1_conv1 (Convolution2D)	(None, 224, 224, 64)	1792	input_1[0][0]
block1_conv2 (Convolution2D)	(None, 224, 224, 64)	36928	block1_conv1[0][0]
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0	block1_conv2[0][0]
block2_conv1 (Convolution2D)	(None, 112, 112, 128)	73856	block1_pool[0][0]
block2_conv2 (Convolution2D)	(None, 112, 112, 128)	147584	block2_conv1[0][0]
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0	block2_conv2[0][0]
block3_conv1 (Convolution2D)	(None, 56, 56, 256)	295168	block2_pool[0][0]
block3_conv2 (Convolution2D)	(None, 56, 56, 256)	590080	block3_conv1[0][0]
block3_conv3 (Convolution2D)	(None, 56, 56, 256)	590080	block3_conv2[0][0]
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0	block3_conv3[0][0]
block4_conv1 (Convolution2D)	(None, 28, 28, 512)	1180160	block3_pool[0][0]
block4_conv2 (Convolution2D)	(None, 28, 28, 512)	2359808	block4_conv1[0][0]
block4_conv3 (Convolution2D)	(None, 28, 28, 512)	2359808	block4_conv2[0][0]
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0	block4_conv3[0][0]
block5_conv1 (Convolution2D)	(None, 14, 14, 512)	2359808	block4_pool[0][0]
block5_conv2 (Convolution2D)	(None, 14, 14, 512)	2359808	block5_conv1[0][0]
block5_conv3 (Convolution2D)	(None, 14, 14, 512)	2359808	block5_conv2[0][0]
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0	block5_conv3[0][0]
flatten (Flatten)	(None, 25088)	0	block5_pool[0][0]
fc1 (Dense)	(None, 4096)	102764544	flatten[0][0]
fc2 (Dense)	(None, 4096)	16781312	fc1[0][0]
predictions (Dense)	(None, 1000)	4097000	fc2[0][0]
Total params: 138357544			

## Softmax

This is how our code looked like after the second session:

```
x = F.relu(self.conv7(x))  
x = x.view(-1, 10)  
return F.log_softmax(x)
```

## First SoftMax

### What is softmax?

In mathematics, the softmax function, also known as softargmax[1] or normalized exponential function is a function that takes as input a vector of K real numbers and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers.

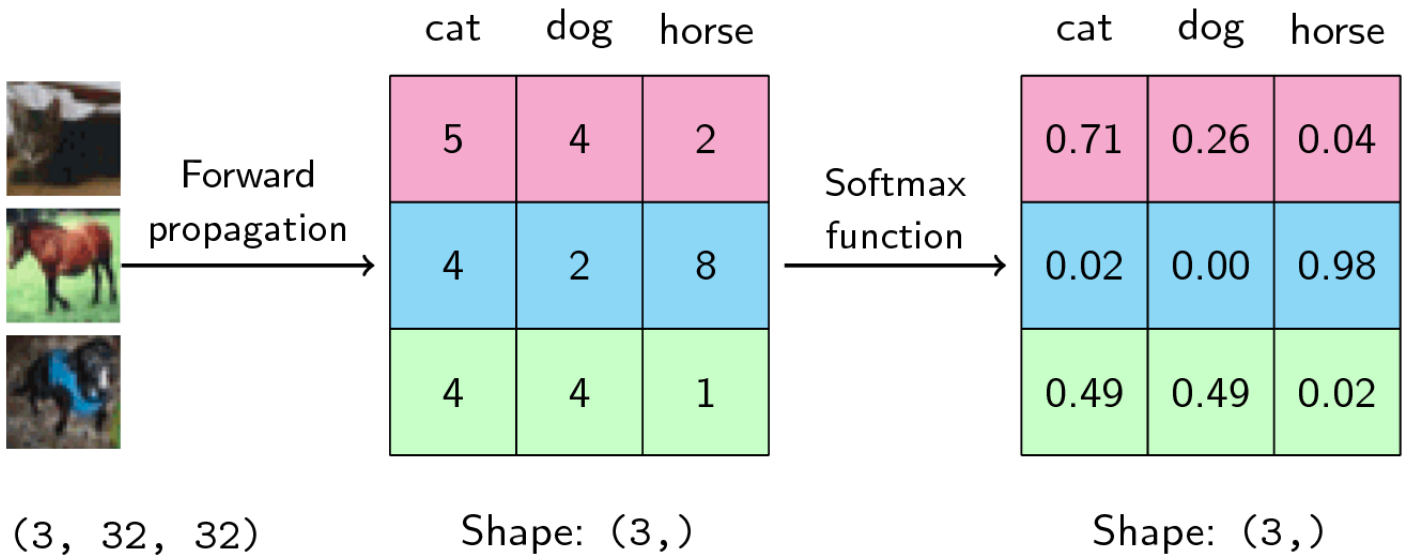
The **softmax** function is often used in the final layer of a neural network-based classifier.

Let's look at what it actually does:

Input pixels,  $x$

Feedforward output,  $y_i$

Softmax output,  $S(y_i)$



*Why Softmax is not probability, but likelihood!*

The output of the softmax describes the [likelihood](https://lvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood) (or if you may,

<https://lvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood>) (confidence) of the neural network that a particular sample belongs to a certain class. Thus, for the first example above, the neural network assigns confidence of 0.71 that it is a cat, 0.26 that it is a dog, and 0.04 that it is a horse. The same goes for each of the samples above.

We can then see that one advantage of using the softmax at the output layer is that it improves the interpretability of the neural network. By looking at the softmax output in terms of the network's confidence, we can then reason about the behavior of our model.

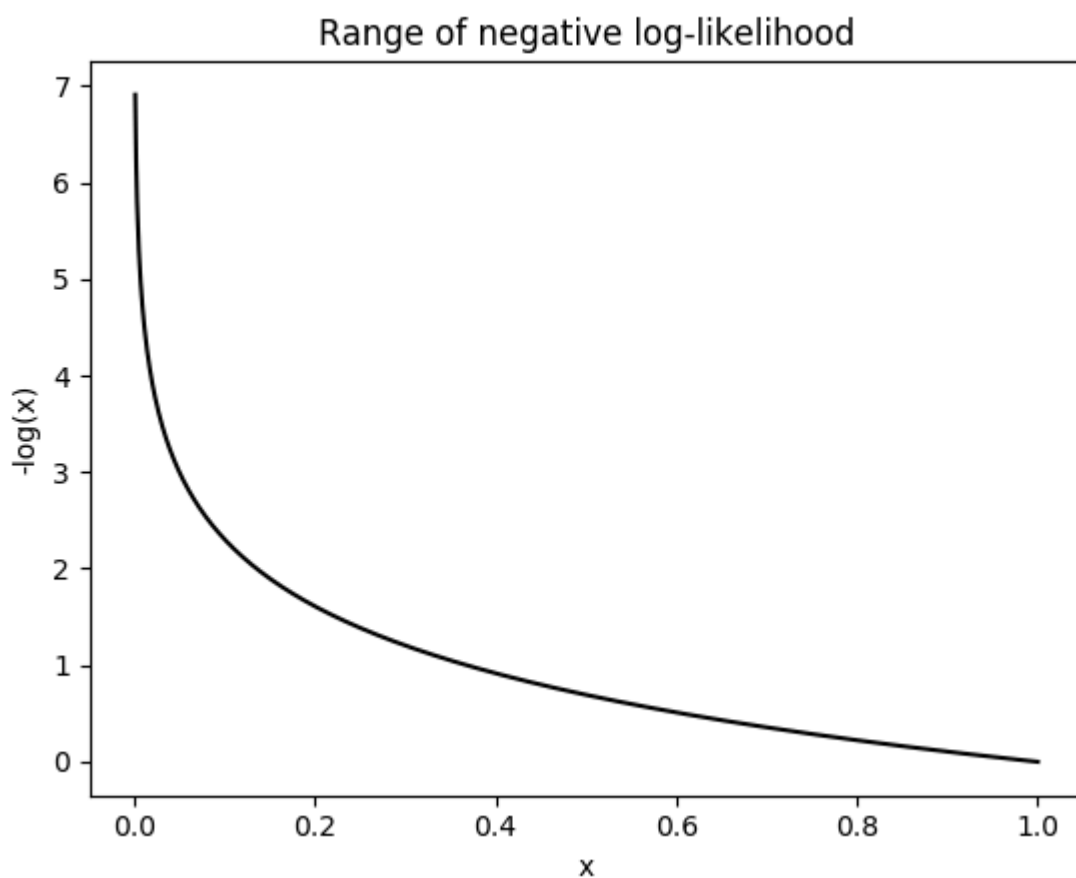
## Negative Log-Likelihood

In practice, the softmax function is used in tandem with the negative log-likelihood (NLL). This loss function is very interesting if we interpret it in relation to the behavior of softmax. First, let's write down our loss function:

$$L(y) = -\log(y)$$

When we train a model, we aspire to find the minima of a loss function given a set of parameters (weights). We can interpret the loss as the "unhappiness" of the neural network with respect to its parameters. The higher the loss, the higher the unhappiness, which we don't want. We want to make our network happy.

So if we are using the negative log-likelihood as our loss function, when does it become unhappy? And when does it become happy? Let's see a plot:



The negative log-likelihood becomes unhappy at smaller values, where it can reach infinite unhappiness (that's too sad), and becomes happy at larger values. *Because we are summing the loss function to all the correct*

classes, what's actually happening is that whenever the network assigns high confidence at the correct class, the unhappiness is low, and vice-versa.

Input pixels,  $x$



Softmax output,  $S(y_i)$

cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

The correct class is highlighted in red

Loss,  $L(a)$

NLL
0.34
0.02
0.71

Total: 1.07

$-\log(a)$  at the correct classes

Correct classes are known because we are training

Predictor confidence of **horse** is high. Lower unhappiness.

Predictor confidence of **dog** is low. Higher unhappiness.

*But why not  $\log(\text{SoftMax})$ ?*

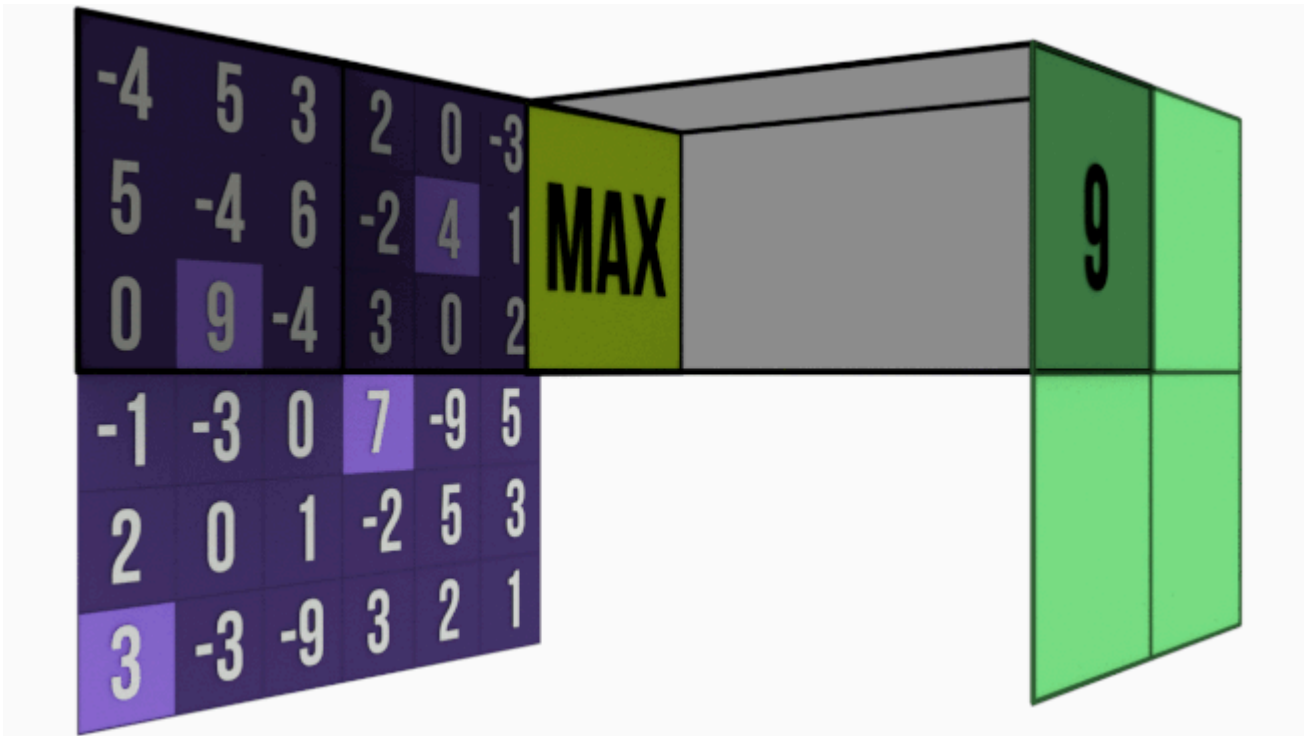
```
def log_softmax(input, dim=None, _stacklevel=3):
    r"""Applies a softmax followed by a logarithm.
    While mathematically equivalent to  $\log(\text{softmax}(x))$ , doing these two
    operations separately is slower, and numerically unstable. This function
    uses an alternative formulation to compute the output and gradient correctly.
    See :class:`~torch.nn.LogSoftmax` for more details.
    Arguments:
        input (Variable): input
        dim (int): A dimension along which log_softmax will be computed.
    """
```

[Ref <https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/>]

## Architectural Blocks

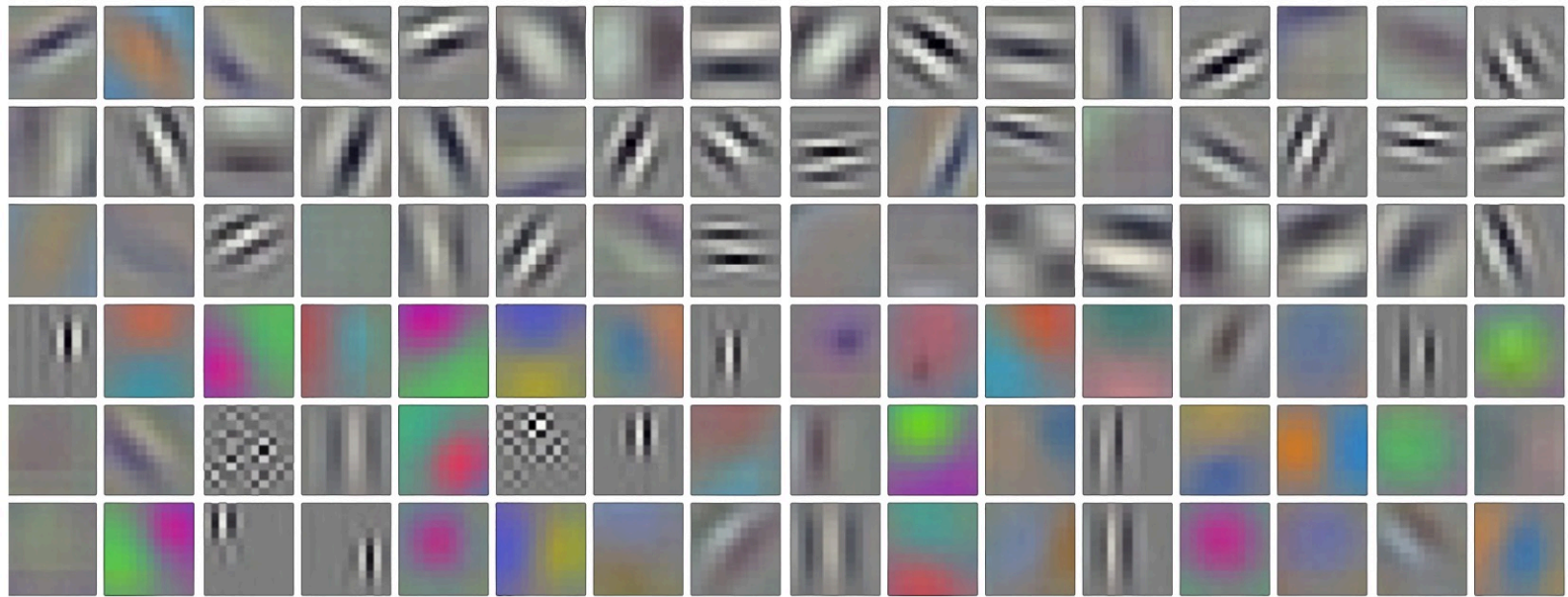
- *Fantastic Beasts and Where to Use them!*

## MAXPOOLING

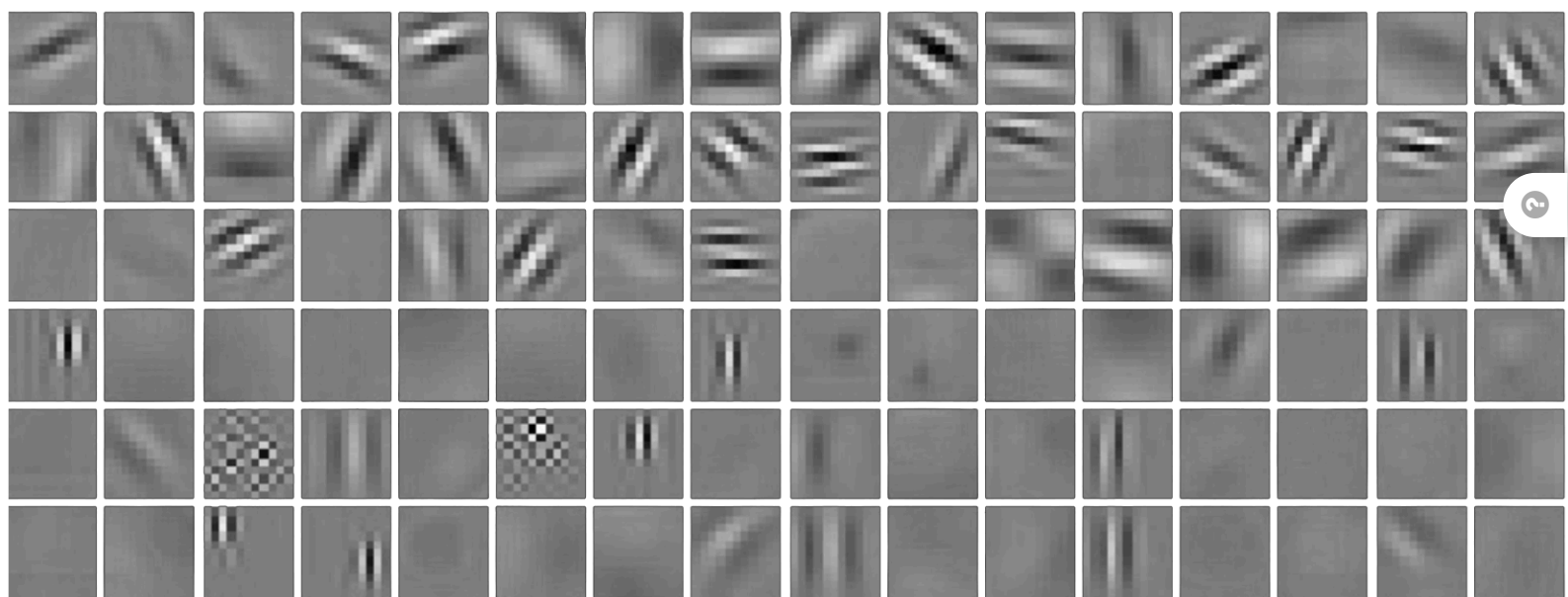


- used when we need to reduce channel dimensions
- not used close to each other
- used far off from the final layer
- used when a block has finished its work (edges-gradients/textures-patterns/parts-of-objects/objects)
- `nn.MaxPool2D()`

## BATCHNORMALIZATION

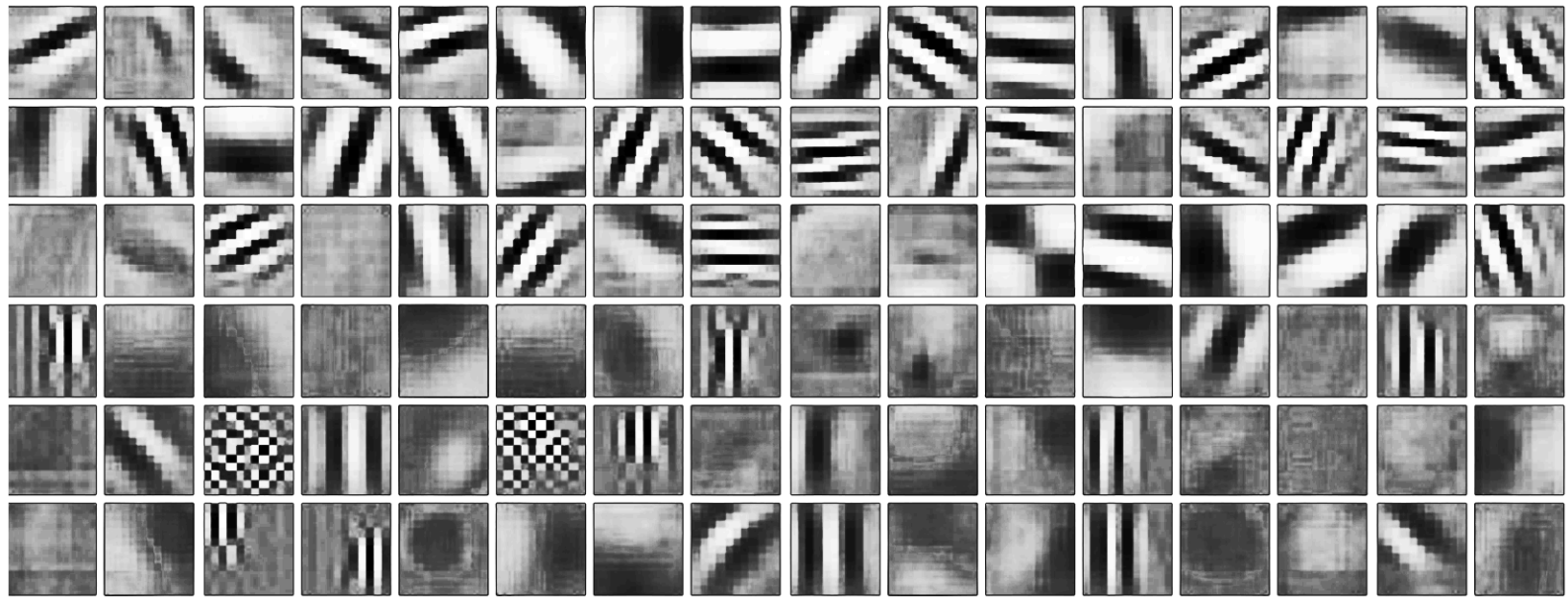


<https://canvas.instructure.com/courses/1587436/files/77236360/download?wrap=1>



<https://canvas.instructure.com/courses/1587436/files/77236380/download?wrap=1>

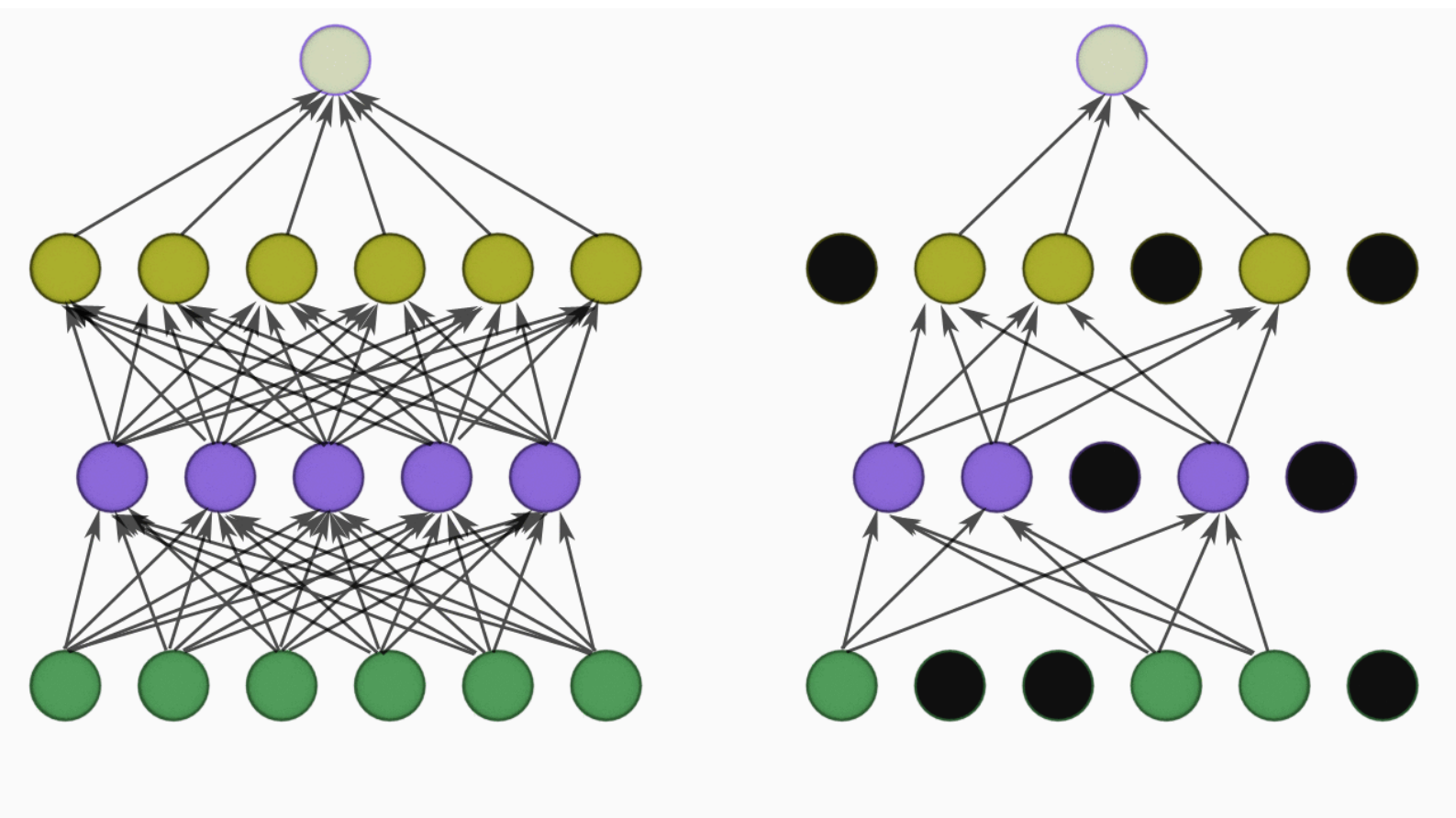




We need a whole dedicated session for BN (next session), so that's all for BN

- used after every layer
- never used before last layer
- *indirectly you have sort of already used it!*
- `nn.BatchNorm2d()`

## DROPOUT

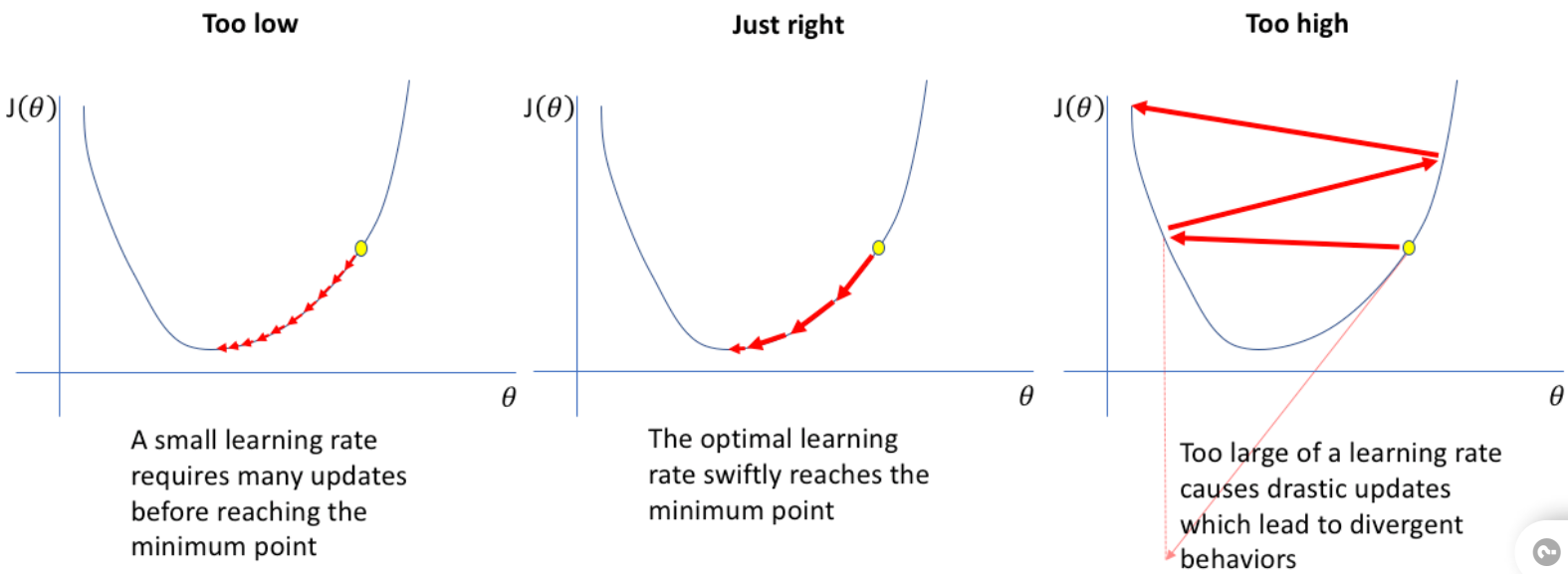


Needs further discussion that what we will have today (next session)



- used after every layer
- used with small values
- is a kind of regularization
- *not used at a specific location*
- `nn.Dropout2d()`

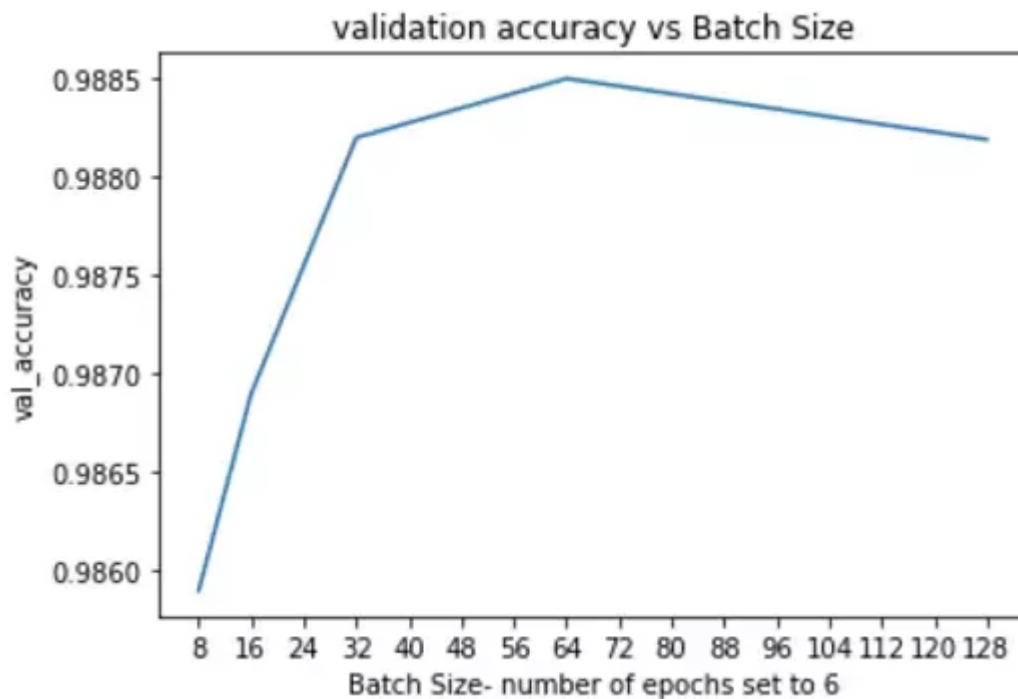
## LEARNING RATE



Needs a dedicated session (6-7)

- stick to suggested values
- the main tool to achieve moksh
- `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)`

## BATCH SIZE





Will be covered in further details in sessions to come

- don't break your head, look at this curve and see the diff in Val accuracies.
- used as a part of data\_loader
- `train_loader = torch.utils.data.DataLoader(`  
`datasets.MNIST('./data', train=True, download=True,`  
`transform=transforms.Compose([`  
`transforms.ToTensor(),`  
`transforms.Normalize((0.1307,), (0.3081,))`  
`]),`  
`batch_size=batch_size, shuffle=True)`

## Assignment:

1. We have considered many many points in our last 4 lectures. Some of these we have covered directly and some indirectly. They are:
  1. How many layers,
  2. MaxPooling,
  3. 1x1 Convolutions,
  4. 3x3 Convolutions,
  5. Receptive Field,
  6. SoftMax,
  7. Learning Rate,
  8. Kernels and how do we decide the number of kernels?
  9. Batch Normalization,
  10. Image Normalization,
  11. Position of MaxPooling,
  12. Concept of Transition Layers,

13. Position of Transition Layer,
  14. DropOut
  15. When do we introduce DropOut, or when do we know we have some overfitting
  16. The distance of MaxPooling from Prediction,
  17. The distance of Batch Normalization from Prediction,
  18. When do we stop convolutions and go ahead with a larger kernel or some other alternative (which we have not yet covered)
  19. How do we know our network is not going well, comparatively, very early
  20. Batch Size, and effects of batch size
  21. etc (you can add more if we missed it here)
2. Refer to this code: [COLABLINK](https://colab.research.google.com/drive/1uJZvJdi5VprOQHROtJIHy0mnY2afjNlx) 
- (<https://colab.research.google.com/drive/1uJZvJdi5VprOQHROtJIHy0mnY2afjNlx>)
1. **WRITE IT AGAIN SUCH THAT IT ACHIEVES**
    1. 99.4% validation accuracy
    2. Less than 20k Parameters
    3. You can use anything from above you want.
    4. Less than 20 Epochs
    5. No fully connected layer
    6. To learn how to add different things we covered in this session, you can refer to this code:  
<https://www.kaggle.com/enwei26/mnist-digits-pytorch-cnn-99>   
(<https://www.kaggle.com/enwei26/mnist-digits-pytorch-cnn-99>) DONT COPY ARCHITECTURE, JUST LEARN HOW TO INTEGRATE THINGS LIKE DROPOUT, BATCHNORM, ETC.
  3. This is a slightly time-consuming assignment, please make sure you start early. You are going to spend a lot of effort into running the programs multiple times
  4. Once you are done, submit your results in S4-Assignment-Solution
  5. **You must upload your assignment to a public GitHub Repository. Create a folder called S4 in it, and add your iPyNb code in it. THE LOGS MUST BE VISIBLE. Before adding the link to the submission make sure you have opened the file in an "incognito" window.**
  6. **If you misrepresent your answers, you will be awarded -100% of the score.**
  7. **If you submit Colab Link instead of notebook uploaded on GitHub, or redirect the GitHub page to colab, you will be awarded -50%**
  8. This assignment is worth 300pts

## Quiz:

1. Some of the lines are *italicized* here. Please make sure you understand them before you attempt the quiz.
2. Keep a calculator handy while taking the quiz
3. The Quiz is 45 minutes long
4. This quiz is worth 200 pts.

## Batch 2 Video:

EVA4 B2 P1S4



## Session Video - Wednesday

EVA 4 - Session 4 Wednesday



## Session Video - Sunday

EVA4 Session 4 - Sunday



