

**LECTURER: Nghia Duong-Trung**

# **DEEP LEARNING**

---

## Introduction to Neural Networks and Deep Learning

---

1

---

## Network Architectures

---

2

---

## Neural Network Training

---

3

---

## Alternative Training Methods

---

4

---

## Further Network Architectures

---

5

**UNIT 2**

# **Network Architectures**



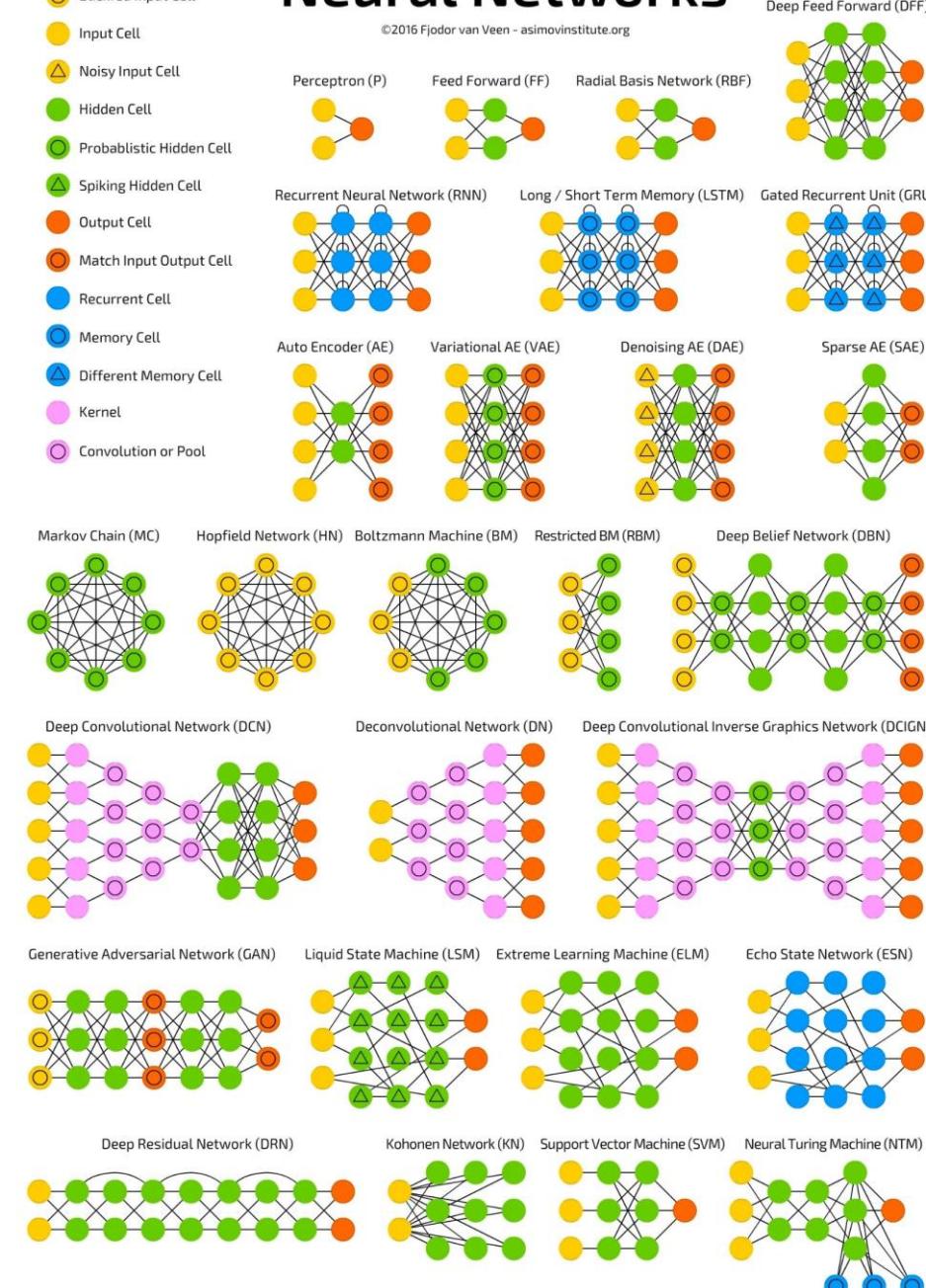
- After completing this unit you will be able to ...
  - ... identify the most important neural network architectures.
  - ... explain what feed-forward, convolutional, and recurrent networks are.
  - ... describe why convolutional neural networks are ideal for image analysis.
  - ... use recurrent neural networks to encode long sequences of events

## CHART OF NEURAL NETWORKS

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

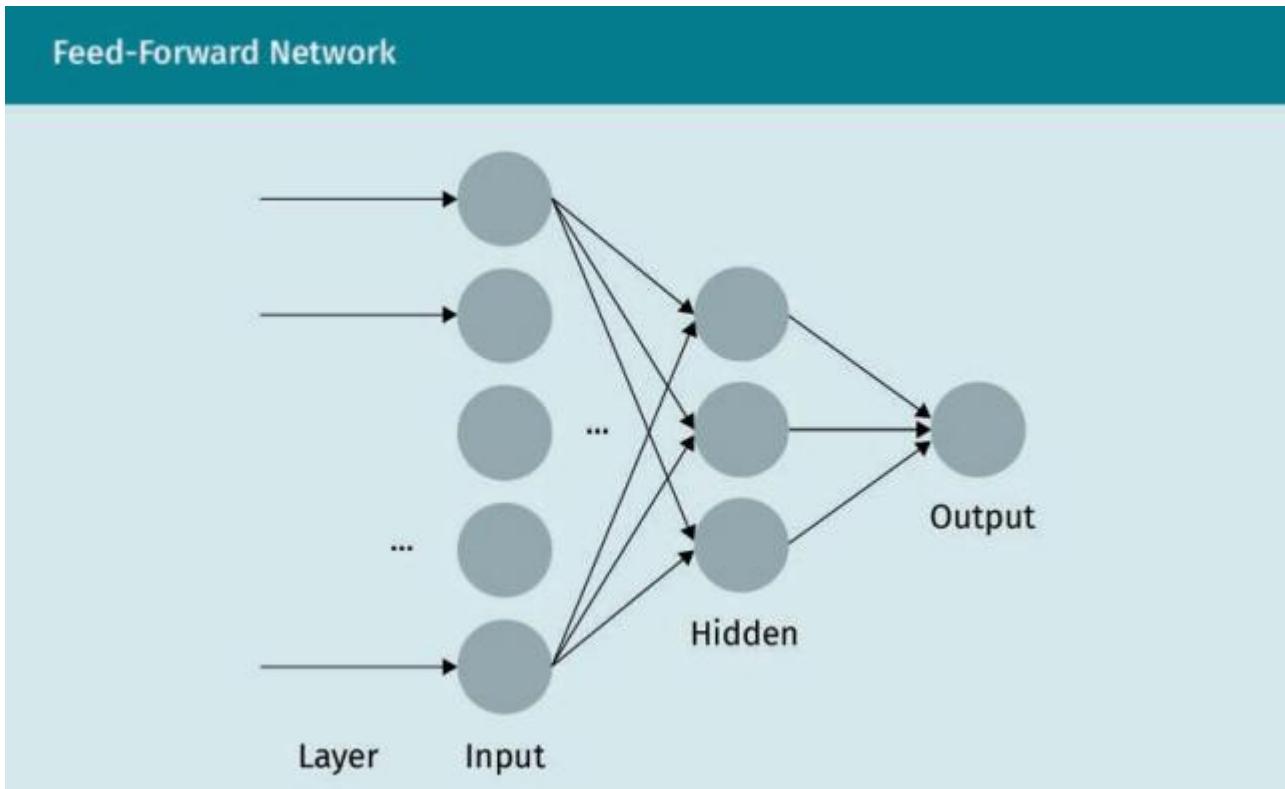
A mostly complete chart of  
**Neural Networks**

©2016 Fjodor van Veen - asimovinstitute.org



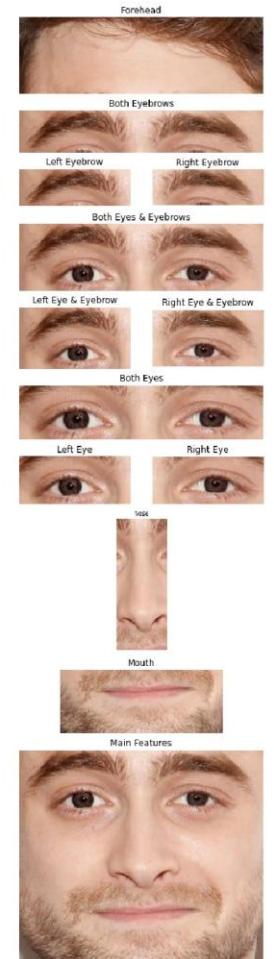
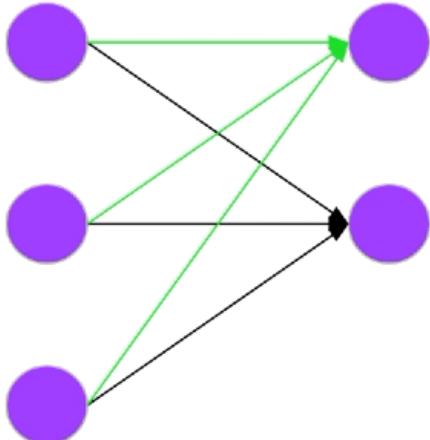
## FORWARD PROPAGATION

- A model is for making predictions.
- How do we make predictions with a neural network?
- We refer to this as “going in the forward direction”



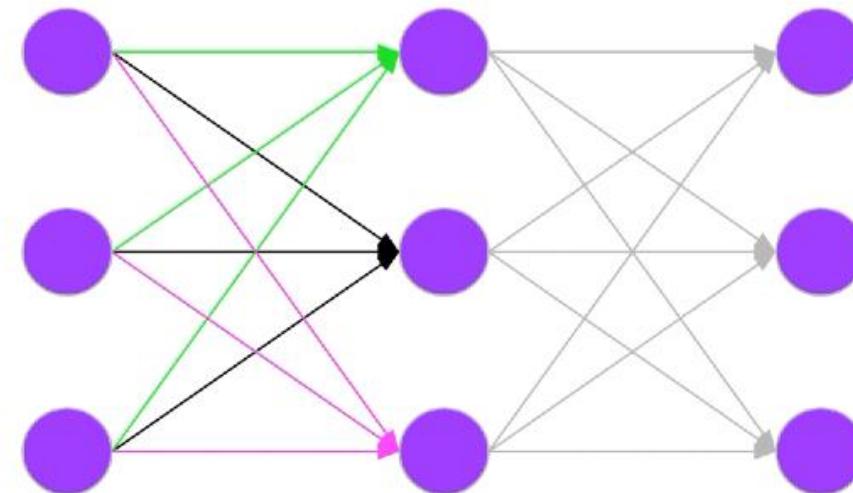
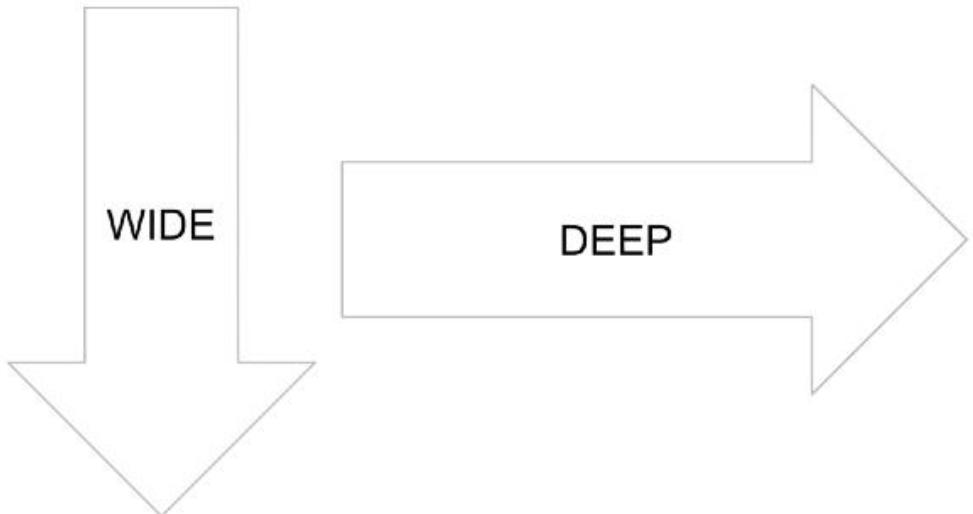
## REPEATING THE SINGLE NEURON

- Each of these neurons may calculate something different
  - Via different weights
- E.g. input is a face
  - One neuron looks for the presence of an eye
  - One neuron looks for the presence of a nose
- They are looking for different features



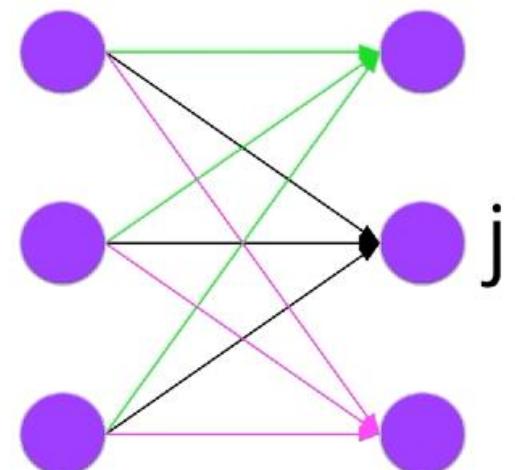
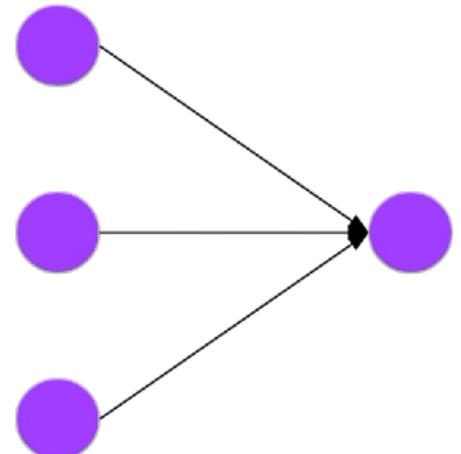
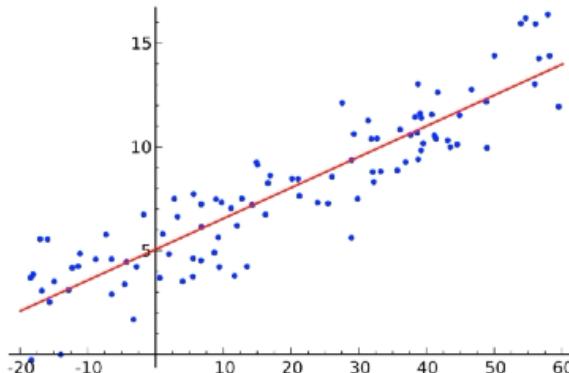
## REPEATING THE SINGLE NEURON

- Key insight: we can stack more layers of neurons: a chain of neurons
- We can model the brain as uniform structure
- 2 important ways to extend the single neuron
  - The same inputs can be fed to multiple different neurons, each calculating something different (more neurons per layer)
  - Neurons in one layer can act as inputs to another layer

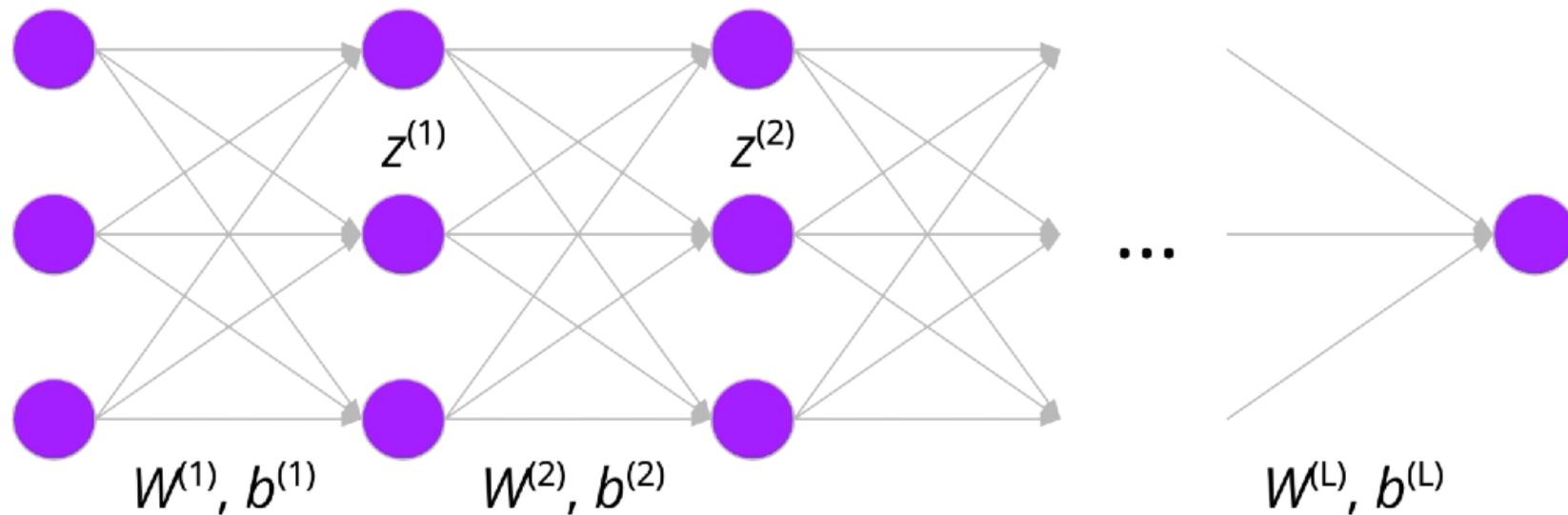


## LINES TO NEURONS

- A line:  $ax + b$
- A neuron:  $\sigma(w^T x + b)$
- Multiple neurons per layer
  - Call the output of the jth neuron  $z_j$ :  $z_j = \sigma(w^T x + b)$ , for  $j = 1 \dots M$
  - Vectorize the neuron:  $z = \sigma(W^T x + b)$
  - Shapes:
    - $z$  is a vector of size  $M$
    - $x$  is a vector of size  $D$
    - $W$  is a matrix of size  $D \times M$
    - $b$  is a vector of size  $M$
    - $\sigma()$  is an element-wise operation



## INPUT TO OUTPUT FOR AN L-LAYER NEURAL NETWORK



$$z^{(1)} = \sigma(W^{(1)T}x + b^{(1)})$$

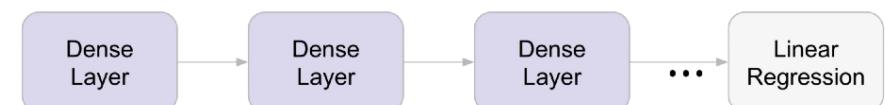
$$z^{(2)} = \sigma(W^{(2)T}z^{(1)} + b^{(2)})$$

$$z^{(3)} = \sigma(W^{(3)T}z^{(2)} + b^{(3)})$$

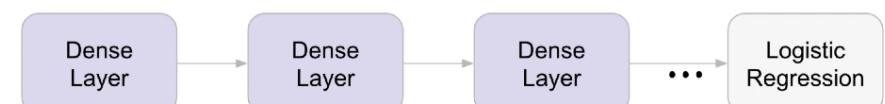
...

$$p(y = 1 | x) = \sigma(W^{(L)T}z^{(L-1)} + b^{(L)})$$

Regression:



Classification:

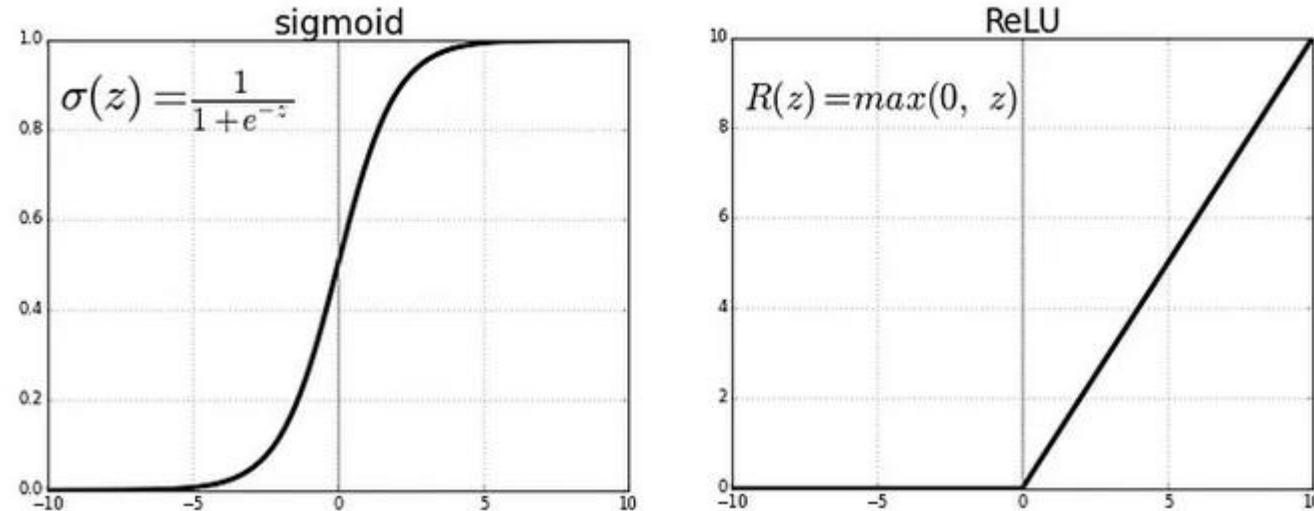


## ACTIVATION FUNCTIONS

- It's just a thing function that you use to get the output of node. It is also known as Transfer Function.
- It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).
- The Activation Functions can be basically divided into 2 types:
  - Linear activation function
  - Non-linear activation fuction
    - The Nonlinear Activation Functions are mainly divided on the basis of their range or curves
    - Sigmoid or Logistic Activation Function
    - Tanh or hyperbolic tangent Activation Function
    - ReLU (Rectified Linear Unit) Activation Function
    - Leaky ReLU

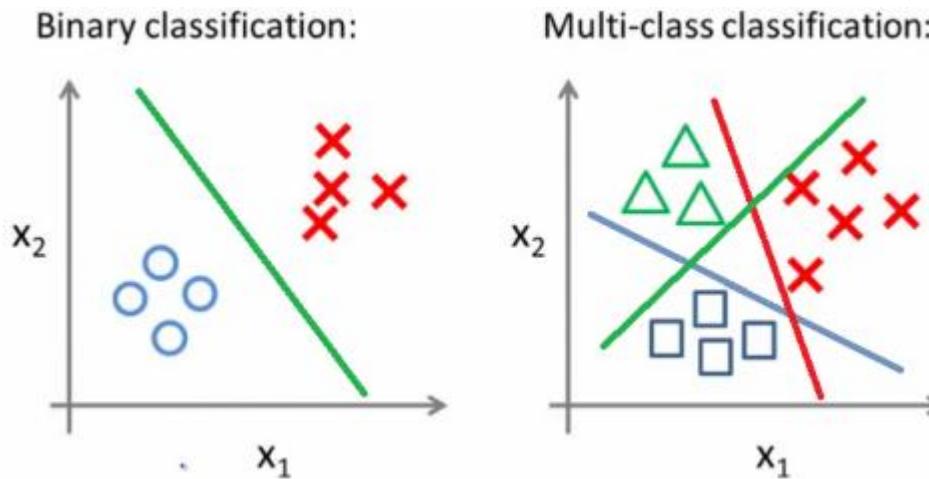
## ACTIVATION FUNCTIONS

- Most people use ReLU as a reasonable default.
- Sometimes, you'll find LReLU and ELU benefit.
- ML is experimentation, not philosophy.
- Never use your mind to guess the output of a computer program.
- 



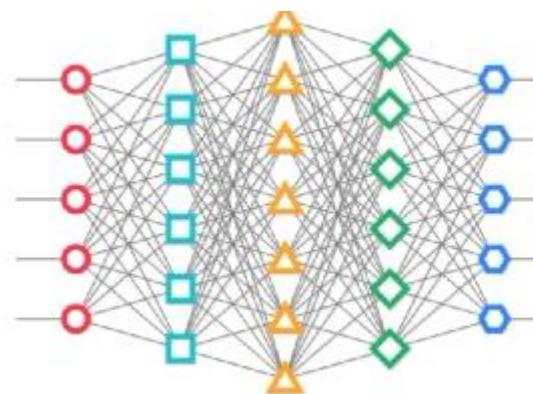
## MULTICLASS CLASSIFICATION

- For binary classification, we use a sigmoid at the output.
- We replaced sigmoids with ReLUs in the hidden layers.
- For output, sigmoid is still the right choice for binary classification.



## MULTICLASS CLASSIFICATION

- Suppose we calculate the value right before applying the final activation function.
- $a^{(L)}$  is a vector of size K (for a K-class classifier)
- How do we turn this vector into a set of probabilities for each of the K classes?
  - $a^{(L)} = W^{(L)T} Z^{(L-1)} + b^{(L)}$
- We need a probability distribution over K distinct values
  - Requirement 1:  $p(y = k | x) \geq 0$
  - Requirement 2:  $\sum_{k=1}^K p(y = k | x) = 1$



In this image, K=5

## THE SOFTMAX FUNCTION

### What softmax does

Given scores (logits)  $\mathbf{z} = (z_1, \dots, z_K)$ , softmax turns them into a probability distribution:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}, i = 1, \dots, K.$$

All outputs are in  $(0, 1)$  and sum to 1.

Higher scores get exponentially more probability.

| Task                      | Activation Function |
|---------------------------|---------------------|
| Regression                | None / Identity     |
| Binary classification     | Sigmoid             |
| Multiclass classification | Softmax             |

## THE SOFTMAX FUNCTION

scores  $\mathbf{z} = (2, 1, 0)$ .

$$e^2 = 7.389, e^1 = 2.718, e^0 = 1, \text{sum} = 11.107$$

$$p \approx \left( \frac{7.389}{11.107}, \frac{2.718}{11.107}, \frac{1}{11.107} \right) \approx (0.665, 0.245, 0.090).$$

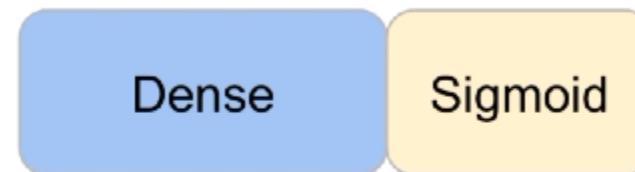
## NETWORK DESIGN PATTERNS

- Same pattern applies to CNNs, RNNs – the type of task corresponds only to the final activation function

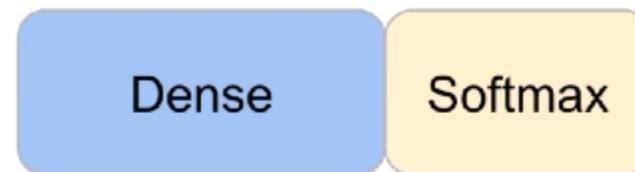
Linear Regression



Binary Logistic Regression



Multiclass Logistic Regression



ANN Regression



ANN Binary Classification

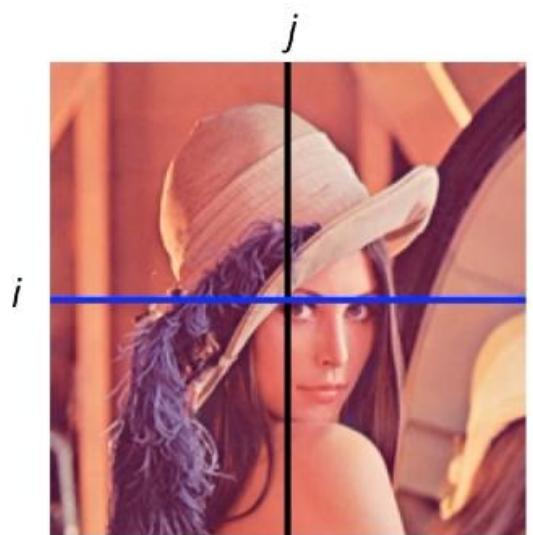


ANN Multiclass Classification



## HOW TO REPRESENT IMAGES

- We need 3 dimensions: height, width, color
- Color dimension always has size 3, corresponding to the RGB channels
- $A(i,j,k)$  stores the value of the  $i$ 'th,  $j$ 'th column,  $k$ 'th color
  - $k = \text{red, green, blue}$
- Physically, color is light, measured by light intensity
- It's a continuous value
- More precision -> Image takes up more space
- We've figured out that 8 bits (1 byte) is good enough
- $2^8 = 256$  possible values (0,1,2,...255)
  - This gives us  $2^8 2^8 2^8 = 16.8$  million possible colors
- How much space does a 500x500 image take up?
  - $500 \times 500 \times 3 \times 8 = 6$  million bits -> 750,000 bytes -> 732KB JPEG



## GRAYSCALE IMAGES

- Images that do not have color can be simplified.
- We call them grayscale because each pixel value can only be black, white, or some shade of gray
- Black = 0, White = 255
- Only requires a 2-D array (height, width)

## ANN CODE PREPARATION

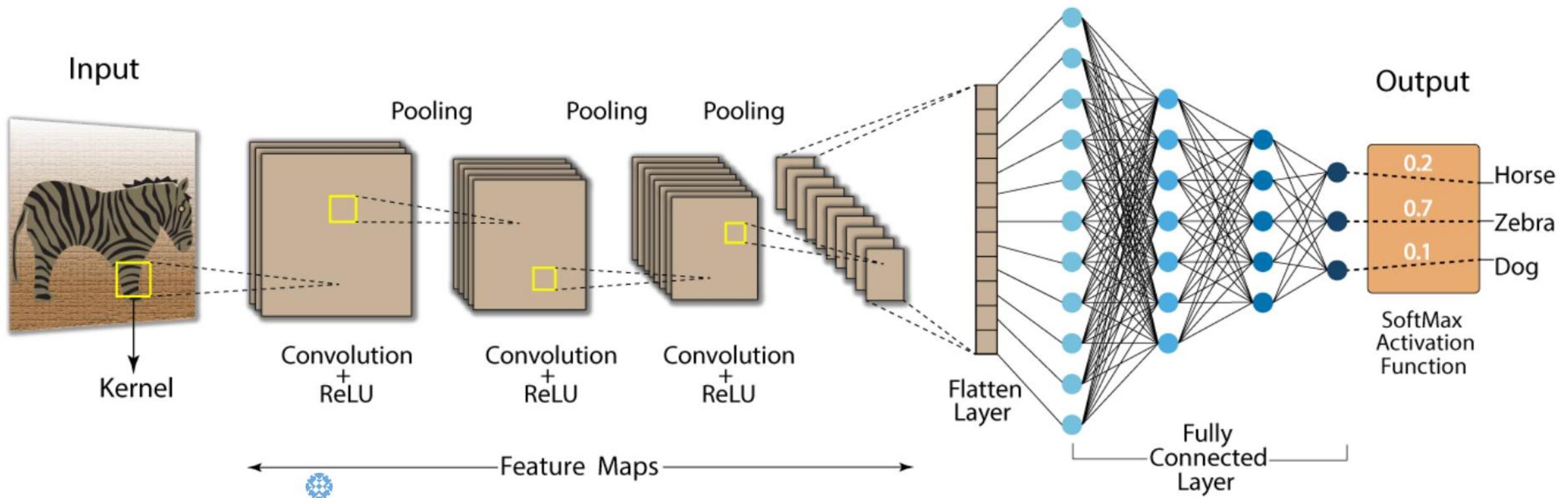
- Recap of the steps
  - #1 Load in the data
  - #2 Build the model
  - #3 Train the model
  - #4 Evaluate the model
  - #5 Make predictions

## **TRANSFER TASK**

- Try the notebook 02. PyTorch\_ANN\_MNIST.ipynb

## CONVOLUTIONAL NEURAL NETWORKS

- A CNN is a “neural network with convolution”
  - There are only 2 requirements: add and multiply
- Convolution = image modifier = pattern finding
  - The filter



## THE MECHANICS OF CONVOLUTION

- Given: input\_image, kernel
- Output\_height =  
 $\text{input\_height} - \text{kernel\_height} + 1$
- Output\_width =  
 $\text{input\_width} - \text{kernel\_width} + 1$
- Images themselves are not square – that's just the nature of cameras and screens.
- Some neural nets use square images for convenience
- Kernels/filters are almost always square
- Translational invariance
  - We want the same filter to look at all locations in the image

| Image   | $*$ | Filter | =  | Output |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
|---|-----|--------|----|--------|----|----|----|----|----|----|----|----|---|---|---|---|--|---|---|---|---|---|--|---|----|--|--|--|--|--|--|--|--|--|--|--|
| <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">10</td><td style="padding: 2px;">10</td><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">20</td><td style="padding: 2px;">30</td><td style="padding: 2px;">30</td><td style="padding: 2px;">20</td></tr> <tr><td style="padding: 2px;">10</td><td style="padding: 2px;">20</td><td style="padding: 2px;">20</td><td style="padding: 2px;">10</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">5</td><td style="padding: 2px;">5</td><td style="padding: 2px;">0</td></tr> </table> | 0   | 10     | 10 | 0      | 20 | 30 | 30 | 20 | 10 | 20 | 20 | 10 | 0 | 5 | 5 | 0 |  | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">2</td></tr> </table> | 1 | 0 | 0 | 2 |  | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">60</td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> </table> | 60 |  |  |  |  |  |  |  |  |  |  |  |
| 0   | 10  | 10     | 0  |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
| 20  | 30  | 30     | 20 |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
| 10  | 20  | 20     | 10 |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
| 0   | 5   | 5      | 0  |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
| 1   | 0   |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
| 0   | 2   |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
| 60  |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
|   |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
|   |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |
|   |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |  |  |  |  |  |  |  |  |  |  |  |

$$1*0 + 0*10 + 0*20 + 30*2 = 60$$

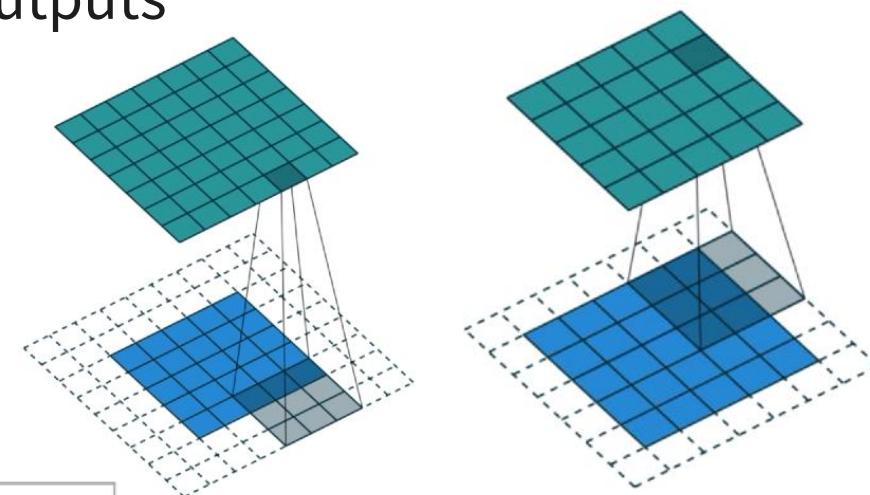
| Image   | $*$ | Filter | =  | Output |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
|---|-----|--------|----|--------|----|----|----|----|----|----|----|----|---|---|---|---|--|---|---|---|---|---|--|---|----|----|--|--|--|--|--|--|--|--|--|--|
| <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">10</td><td style="padding: 2px;">10</td><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">20</td><td style="padding: 2px;">30</td><td style="padding: 2px;">30</td><td style="padding: 2px;">20</td></tr> <tr><td style="padding: 2px;">10</td><td style="padding: 2px;">20</td><td style="padding: 2px;">20</td><td style="padding: 2px;">10</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">5</td><td style="padding: 2px;">5</td><td style="padding: 2px;">0</td></tr> </table> | 0   | 10     | 10 | 0      | 20 | 30 | 30 | 20 | 10 | 20 | 20 | 10 | 0 | 5 | 5 | 0 |  | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">2</td></tr> </table> | 1 | 0 | 0 | 2 |  | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">60</td><td style="padding: 2px;">70</td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> </table> | 60 | 70 |  |  |  |  |  |  |  |  |  |  |
| 0   | 10  | 10     | 0  |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
| 20  | 30  | 30     | 20 |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
| 10  | 20  | 20     | 10 |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
| 0   | 5   | 5      | 0  |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
| 1   | 0   |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
| 0   | 2   |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
| 60  | 70  |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
|   |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
|   |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |
|   |     |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |   |    |    |  |  |  |  |  |  |  |  |  |  |

$$1*10 + 0*10 + 0*30 + 2*30 = 70$$

| Image   | $*$ | Filter | =  | Output |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
|---|-----|--------|----|--------|----|----|----|----|----|----|----|----|---|---|---|---|--|---|---|---|---|---|--|--|----|----|----|----|----|----|----|----|----|
| <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">10</td><td style="padding: 2px;">10</td><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">20</td><td style="padding: 2px;">30</td><td style="padding: 2px;">30</td><td style="padding: 2px;">20</td></tr> <tr><td style="padding: 2px;">10</td><td style="padding: 2px;">20</td><td style="padding: 2px;">20</td><td style="padding: 2px;">10</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">5</td><td style="padding: 2px;">5</td><td style="padding: 2px;">0</td></tr> </table> | 0   | 10     | 10 | 0      | 20 | 30 | 30 | 20 | 10 | 20 | 20 | 10 | 0 | 5 | 5 | 0 |  | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">2</td></tr> </table> | 1 | 0 | 0 | 2 |  | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">60</td><td style="padding: 2px;">70</td><td style="padding: 2px;">50</td></tr> <tr><td style="padding: 2px;">60</td><td style="padding: 2px;">70</td><td style="padding: 2px;">50</td></tr> <tr><td style="padding: 2px;">20</td><td style="padding: 2px;">30</td><td style="padding: 2px;">20</td></tr> </table> | 60 | 70 | 50 | 60 | 70 | 50 | 20 | 30 | 20 |
| 0   | 10  | 10     | 0  |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 20  | 30  | 30     | 20 |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 10  | 20  | 20     | 10 |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 0   | 5   | 5      | 0  |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 1   | 0   |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 0   | 2   |        |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 60  | 70  | 50     |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 60  | 70  | 50     |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |
| 20  | 30  | 20     |    |        |    |    |    |    |    |    |    |    |   |   |   |   |  |   |   |   |   |   |  |  |    |    |    |    |    |    |    |    |    |

## PADDING

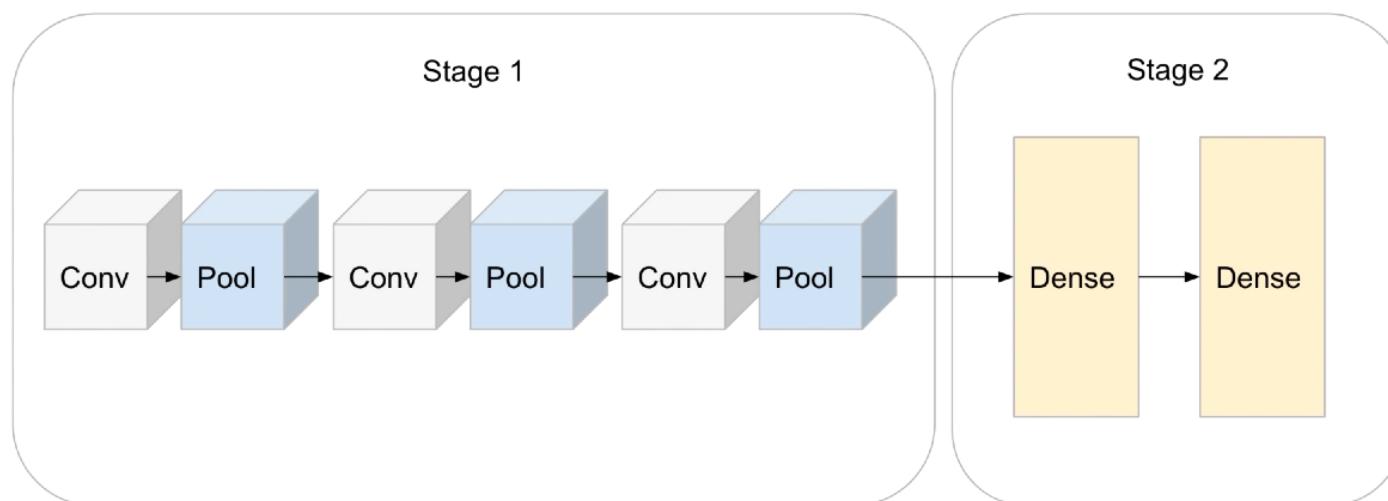
- What if we want the output to be the same size as the input?
- Then we can use padding (add imaginary zeros around the input)
- We could extend the filter further and still get non-zero outputs
- Full padding:
  - Input length =  $N$
  - Kernel length =  $K$
  - Output length =  $N + - 1$



| Mode  | Output Size | Usage    |
|-------|-------------|----------|
| Valid | $N - K + 1$ | Typical  |
| Same  | $N$         | Typical  |
| Full  | $N + K - 1$ | Atypical |

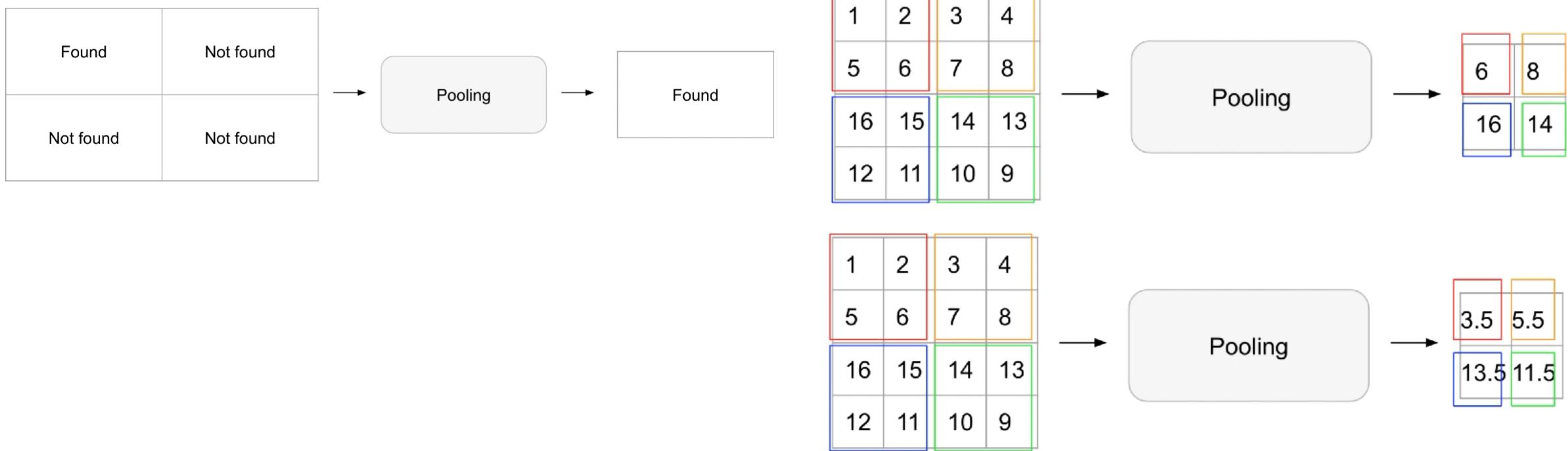
## TYPICAL CNN

- Hyperparameters
  - Learning rate, # hidden layers, # hidden units per layer
  - With CNNs, the conventions are pretty standard
    - Small filters relative to image, e.g., 3x3, 5x5, 7x7
    - Repeat: convolution -> polling -> ...
    - Increase #feature maps: e.g., 32 -> 64 -> 128



## POOLING

- There are 2 kinds of pooling: max and average
- Practical: if we shrink the image, we have less data to process
- Translational invariance: I don't care where in the image the feature occurred, I just care that it did

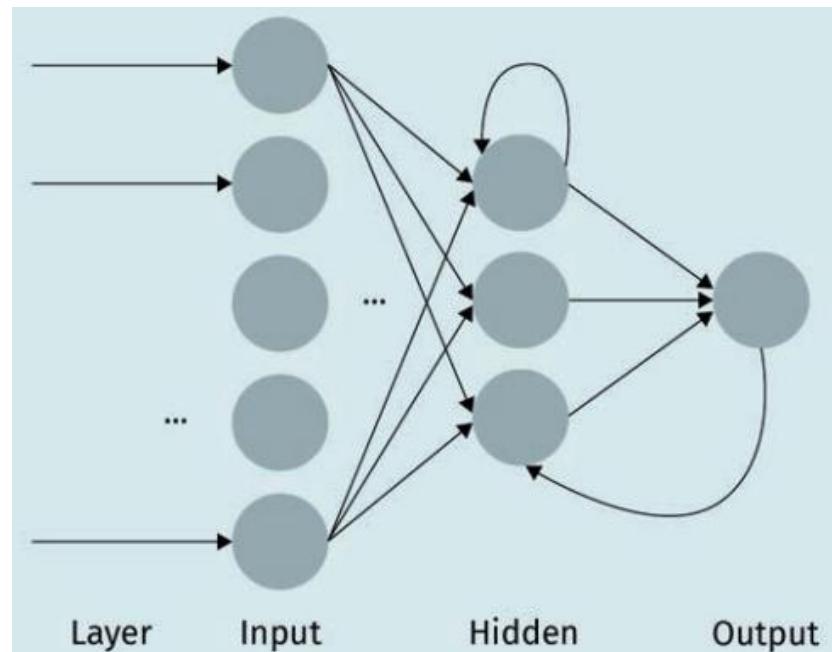


## **TRANSFER TASK**

- Run the notebook CNN\01. PyTorch\_CNN\_Fashion\_MNIST.ipynb

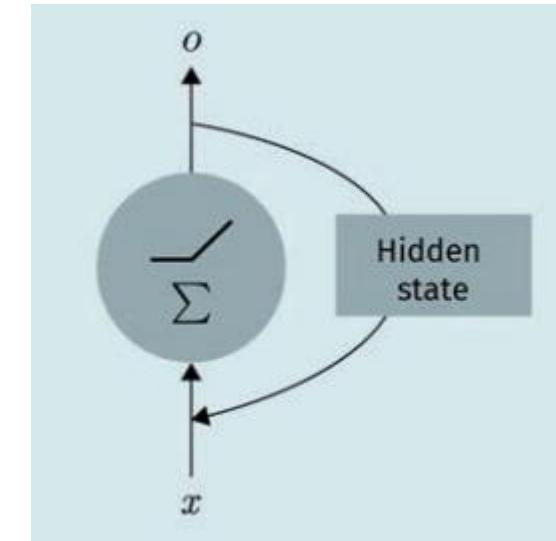
## RECURRENT NEURAL NETWORKS (RNN)

- RNN allows connections not only to the next layer but also to previous layers, the current layer, or even onto the neurons themselves
- This allows RNNs to develop a memory that can, for example, be used for language generation
- This means that the output of the neuron now depends on both the previous inputs to the neuron and the output it has calculated



## MEMORY CELLS

- In the simple recurrent neuron or layer of recurrent neurons, we sent the output computed by the neuron(s) back as additional input.
- However, we can make the recurrent neuron more powerful by extending this approach and including a hidden state into the loop, sending the output back and creating a memory cell.
- This hidden state is characterized by some function  $h(t)$ , which depends on the previous inputs, earlier outputs of the neuron, and the hidden state at earlier steps

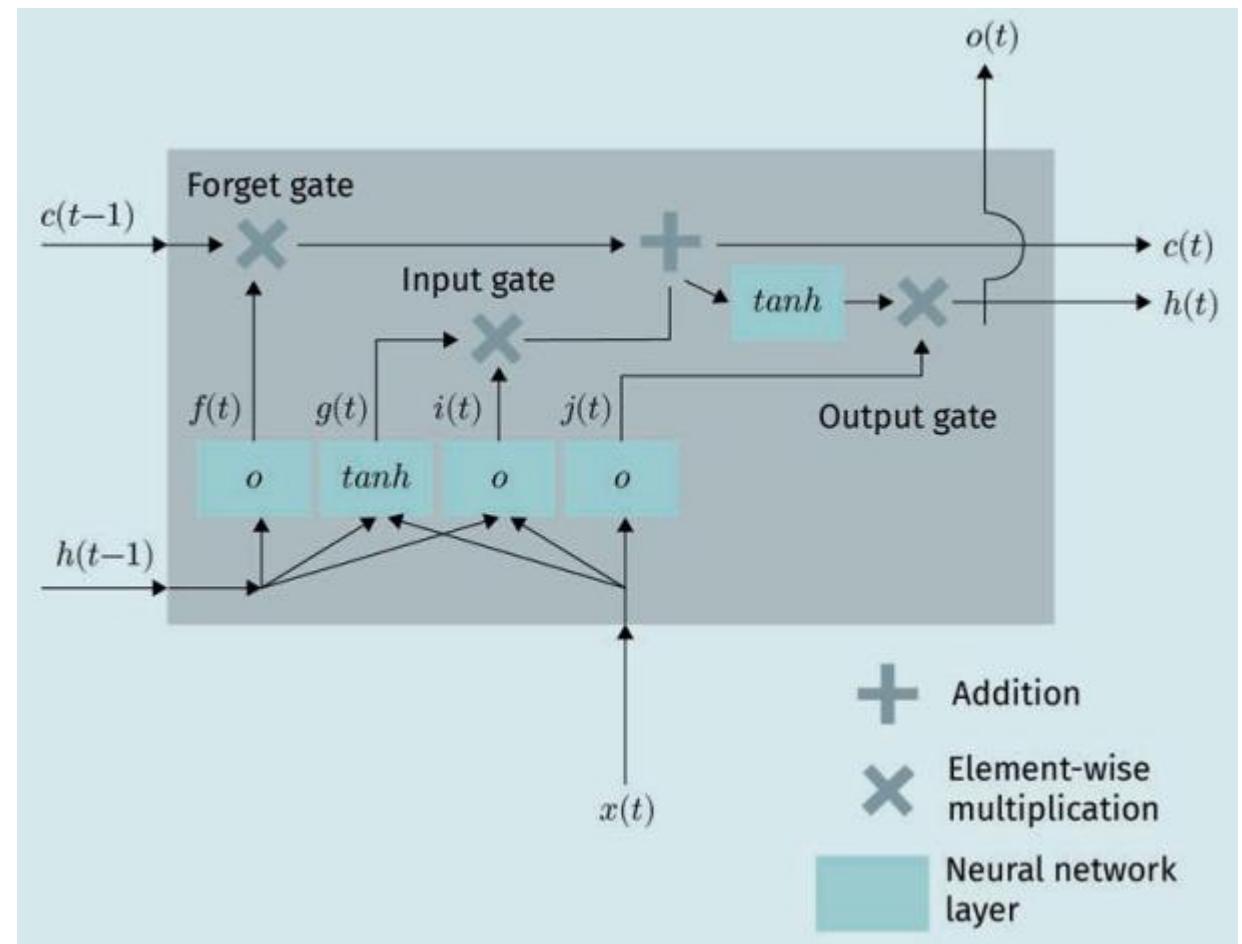


## LSTM

- One of the main challenges of using recurrent neural networks is learning long sequences, such as a very long sentence or a poem.
- Learning such long sequences requires that we take more than one previous output into account, i.e., the output does not only depend on time steps  $t$  and  $t - 1$ , but also steps  $t - 2, t - 3, \dots, t - n$ . This makes the training of RNNs very difficult and unstable.
- By creating a sophisticated hidden state, the RNN can perform much better.
- The general idea is to create a memory structure ( $M$ ) and decide during the training process which parts of the previously-seen input data should be remembered and which parts of the memory should be forgotten, as they are no longer relevant.
  - This then forms a candidate for the new memory ( $M'$ )
  - We can express this as  $M(t) = \text{remember}(t) * M(t - 1) + \text{add}(t) * M'(t)$ .

## LSTM CELL

- Compared to the simple recurrent cell, the hidden state is split into two functions:  $c(t)$  and  $h(t)$ .
  - $c(t)$ : the long-term hidden state
  - $h(t)$ : the short-term hidden state
- This is done via a series of *gates*



## REVIEW STUDY GOALS



- After completing this unit you will be able to ...
  - ... identify the most important neural network architectures.
  - ... explain what feed-forward, convolutional, and recurrent networks are.
  - ... describe why convolutional neural networks are ideal for image analysis.
  - ... use recurrent neural networks to encode long sequences of events

**SESSION 1**

# **Network Architectures**

© 2022 IU Internationale Hochschule GmbH

This content is protected by copyright. All rights reserved.

This content may not be reproduced and/or electronically edited, duplicated, or distributed in any kind of form without written permission by the IU Internationale Hochschule GmbH.