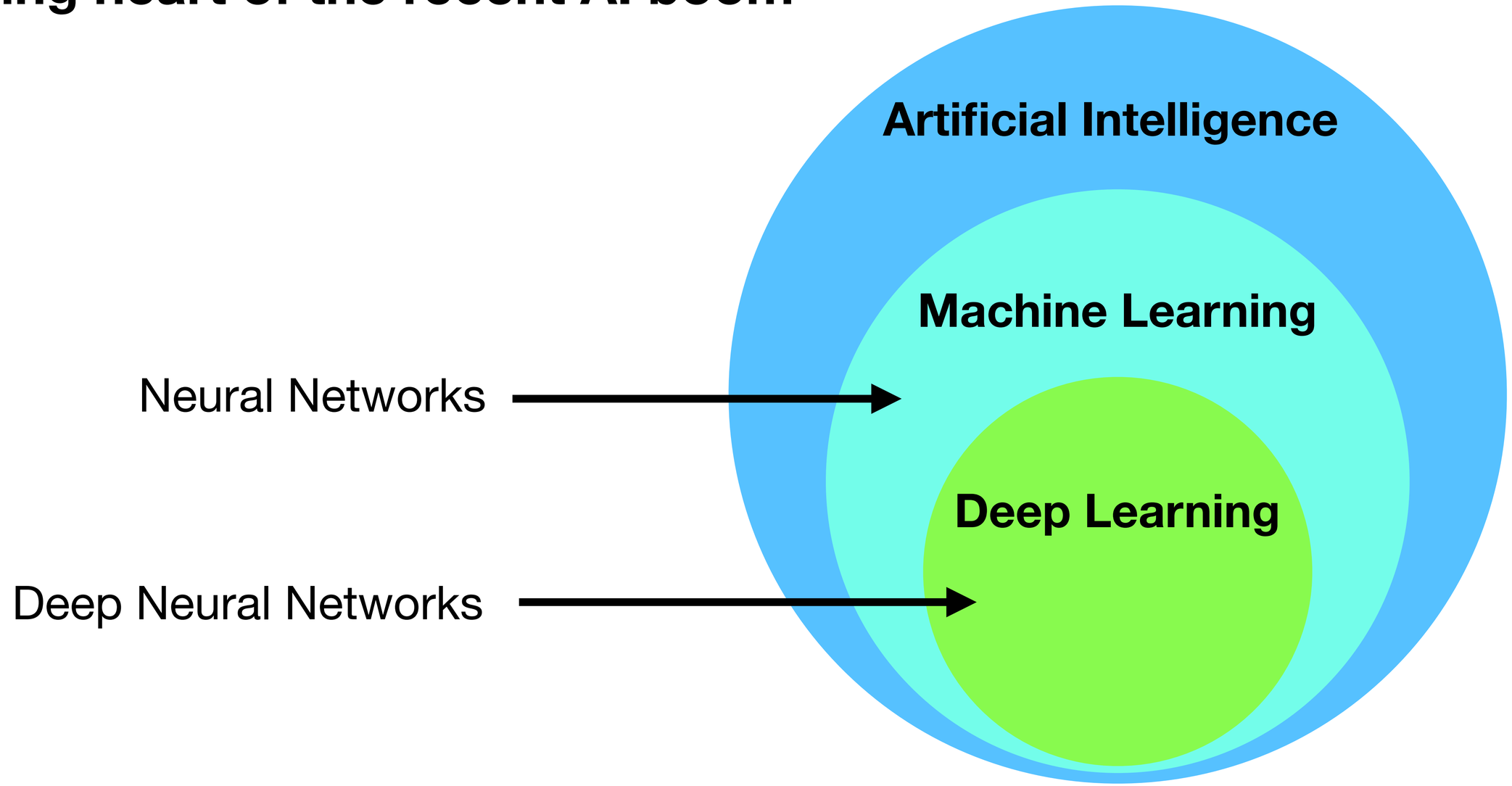
# Accelerating Neural Networks with CUDA

## Multi-Core Programming

Parsa Toopchinezhad - Dr. Jahangiry - Spring 2024

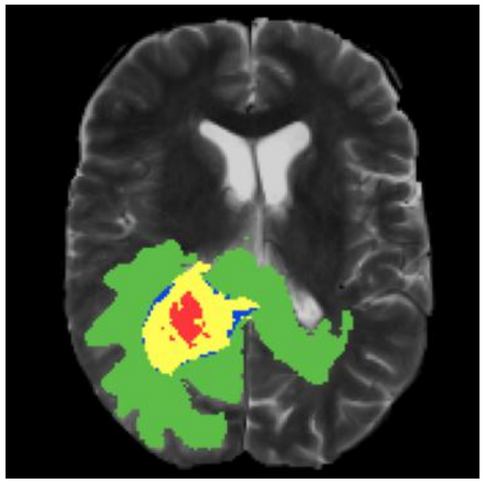
# What Are Neural Networks?

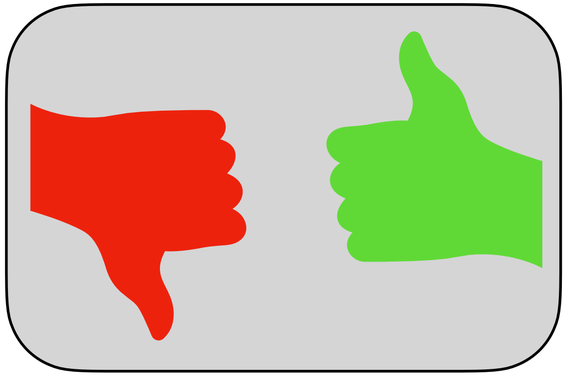
## The beating heart of the recent AI boom

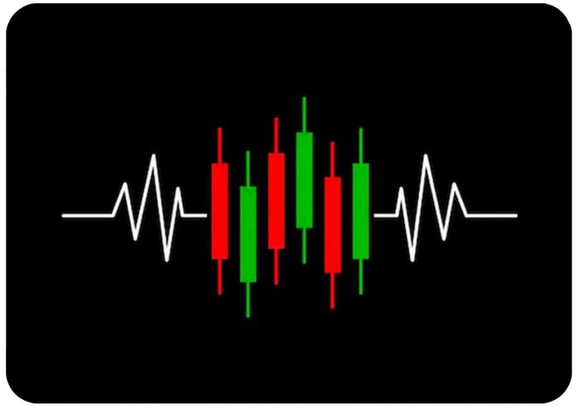


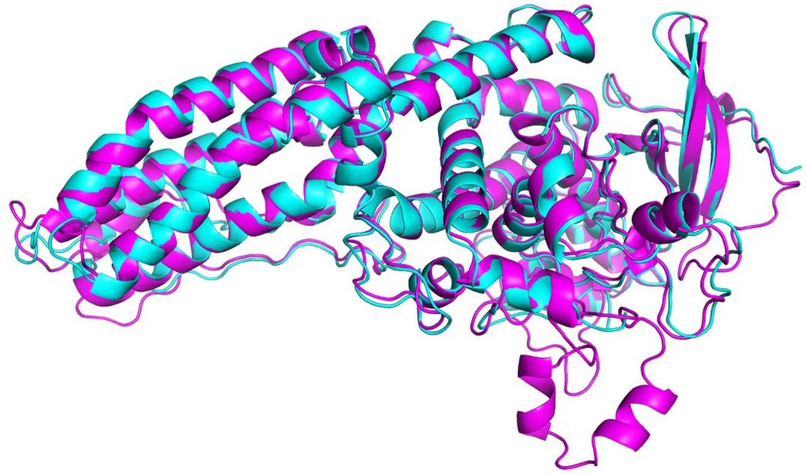
# Applications of Neural Networks

## Dawn of an AI powered world







# Novel Applications of NNs

## Discovering faster matrix multiplication (AlphaTensor)

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} \\ b_{5,1} & b_{5,2} & b_{5,3} & b_{5,4} & b_{5,5} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} & c_{1,5} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} & c_{2,5} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} & c_{3,5} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} & c_{4,5} \end{bmatrix}$$

- Traditional: 100
- Strassen: 80
- AlphaTensor: 76

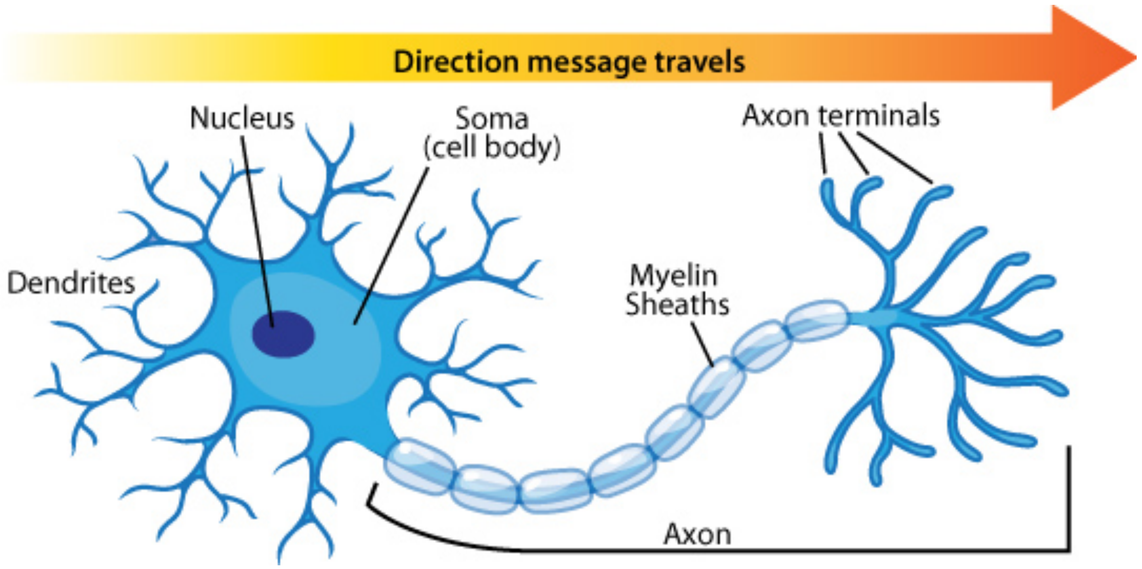
# Cool, but how do neural networks work?

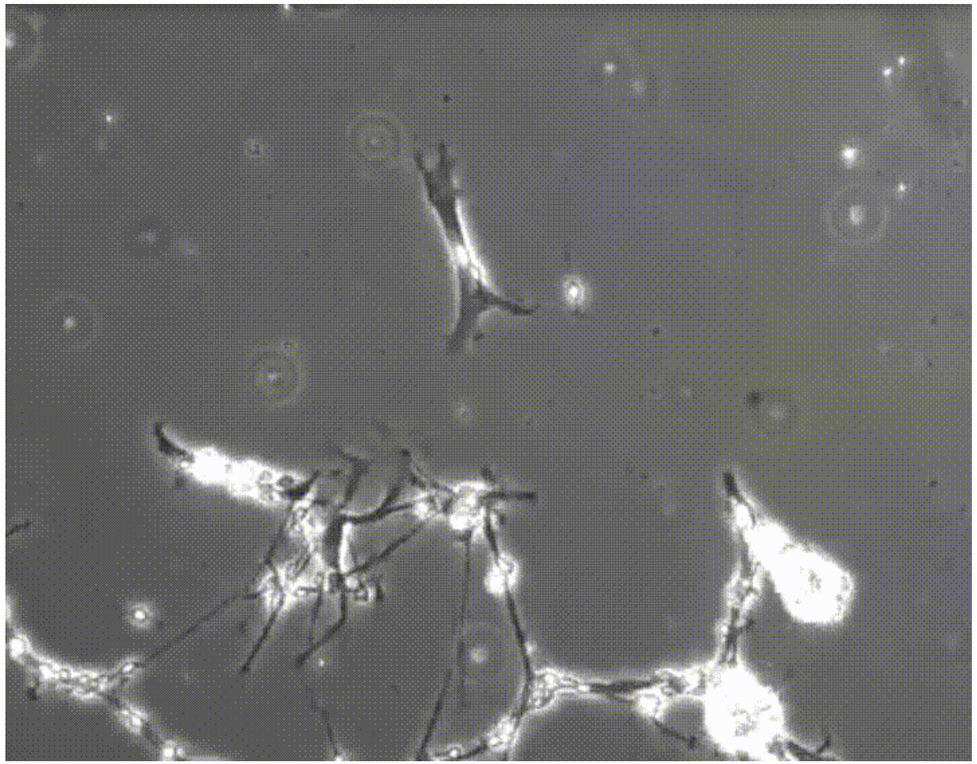
(First we'll look at biological neural networks)

# How Does The Brain Work?

## The least known part of our body

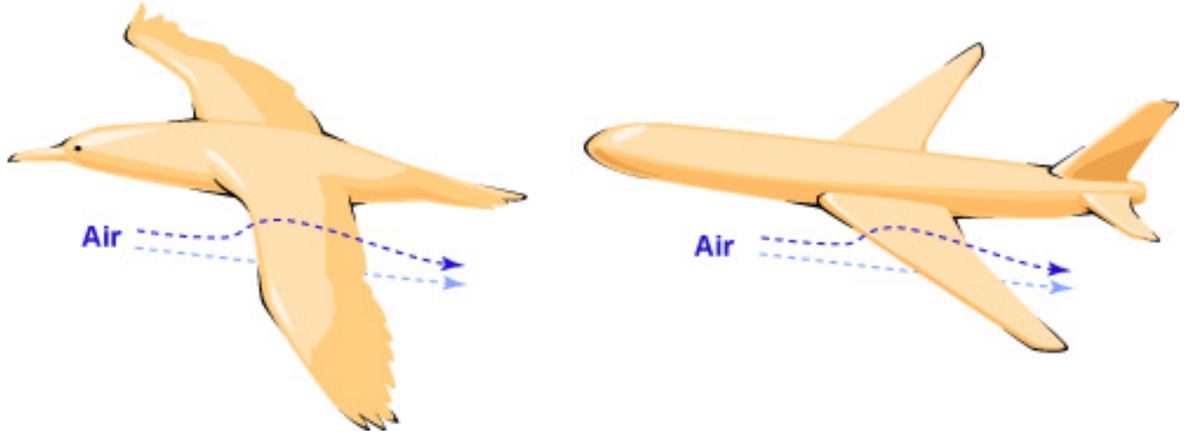
- “If the brain were so simple that we could understand it, we would be so simple that we couldn’t.” - Emerson M. Pugh
- Around 86 billion neurons

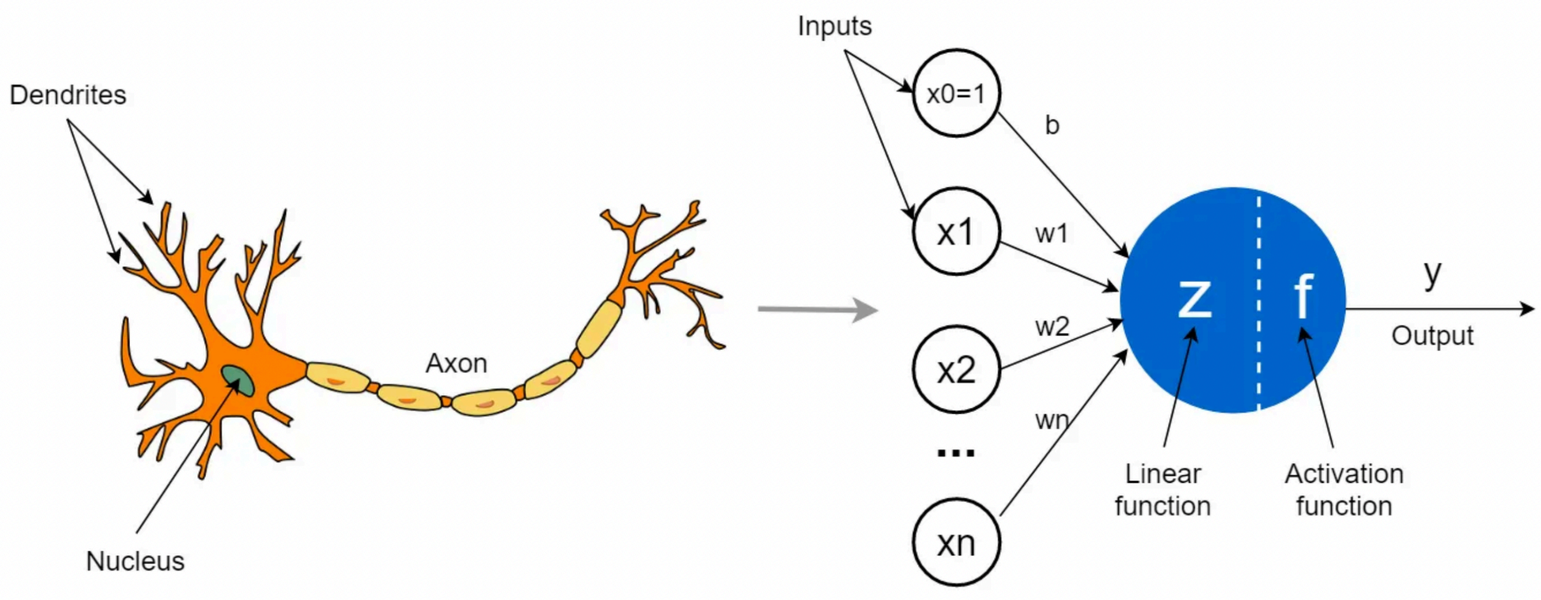


# Modelling Nature

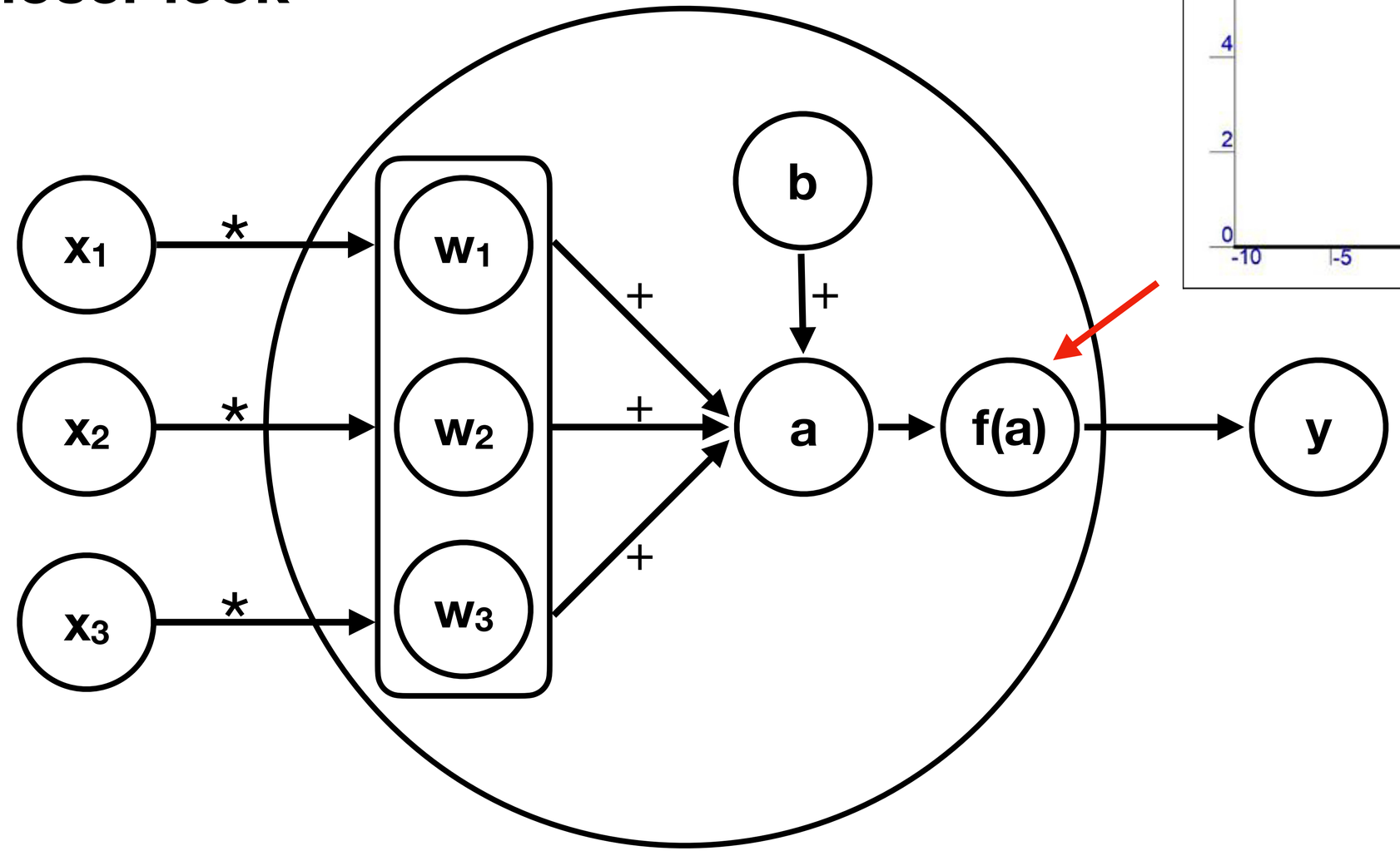
## Getting inspiration from the world around us

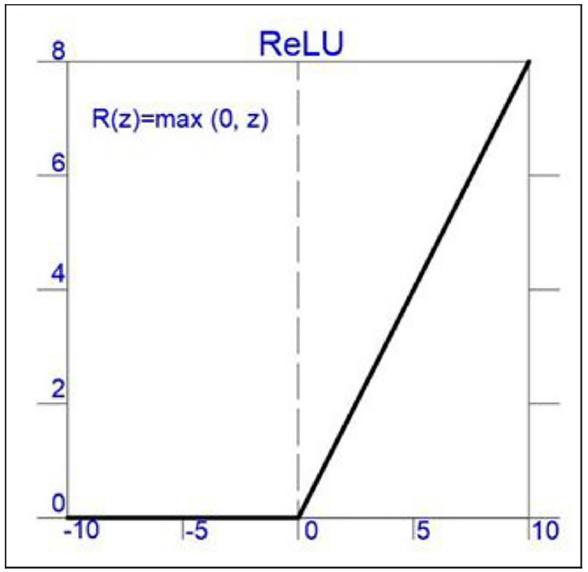


# Artificial Neuron

## A closer look

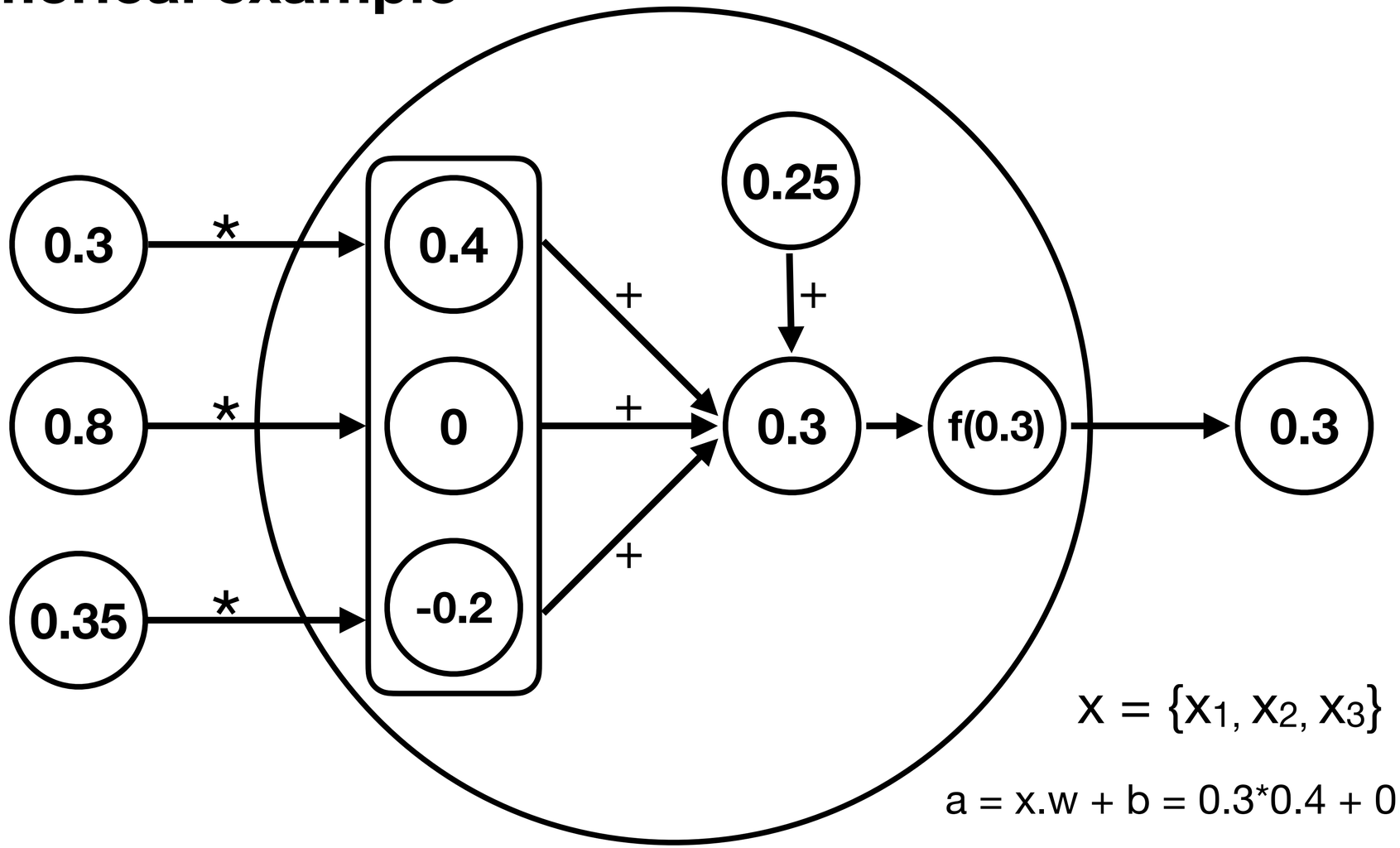



Activation Function

# Artificial Neuron

## Numerical example



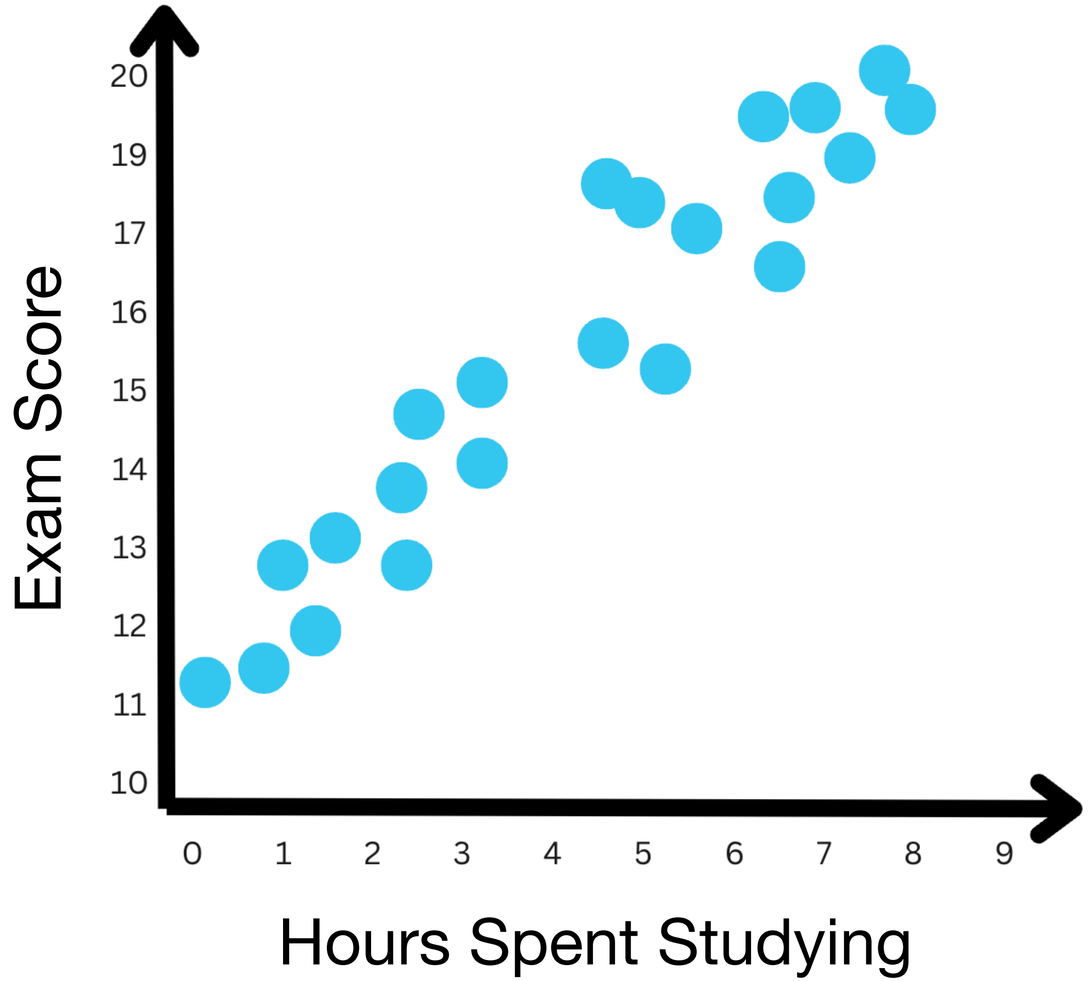

$x = \{x_1, x_2, x_3\}$ $w = \{w_1, w_2, w_3\}$

a = x.w + b = 0.3*0.4 + 0.8*0 + 0.35*-0.2 + 0.25 = 0.3

y = f(a) = ReLU(0.3) = 0.3

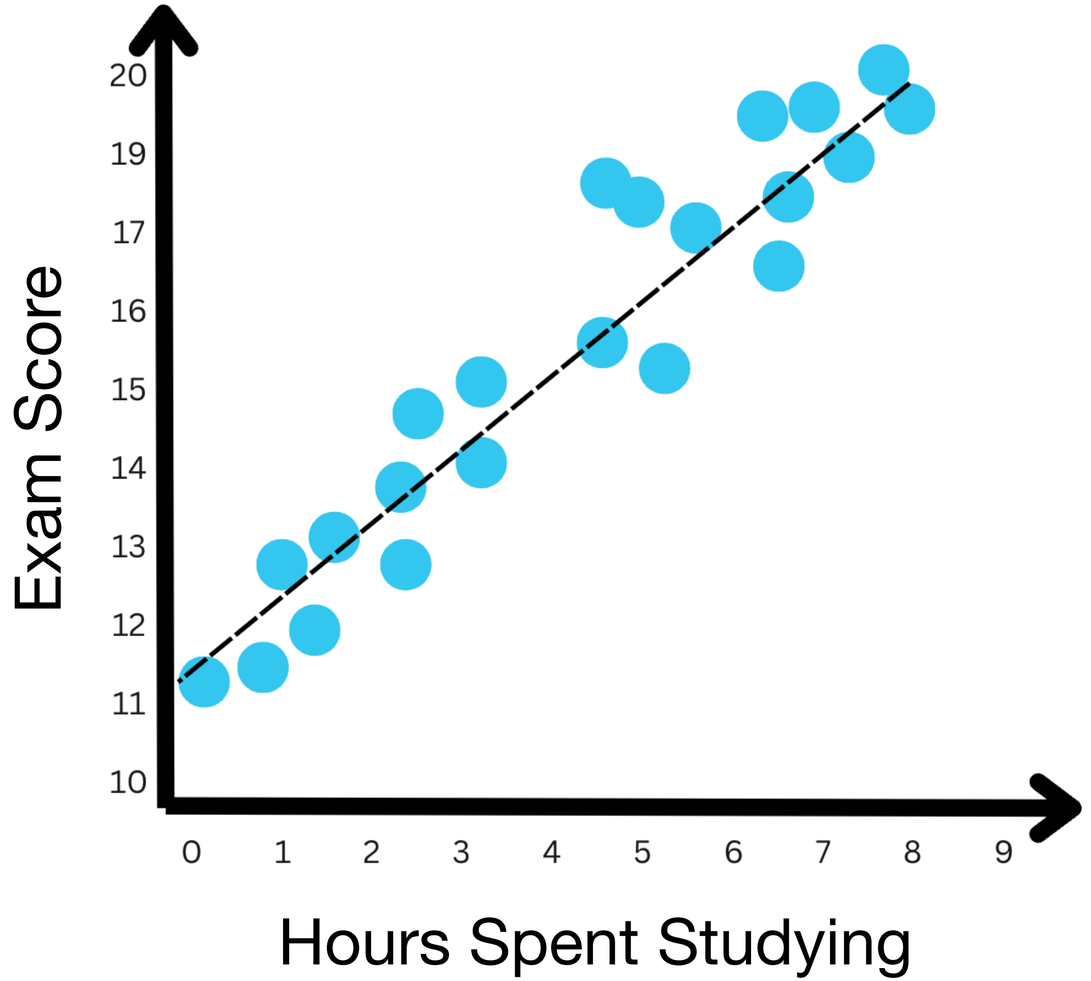
# What Can a Perceptron Do?

## Simple yet powerful



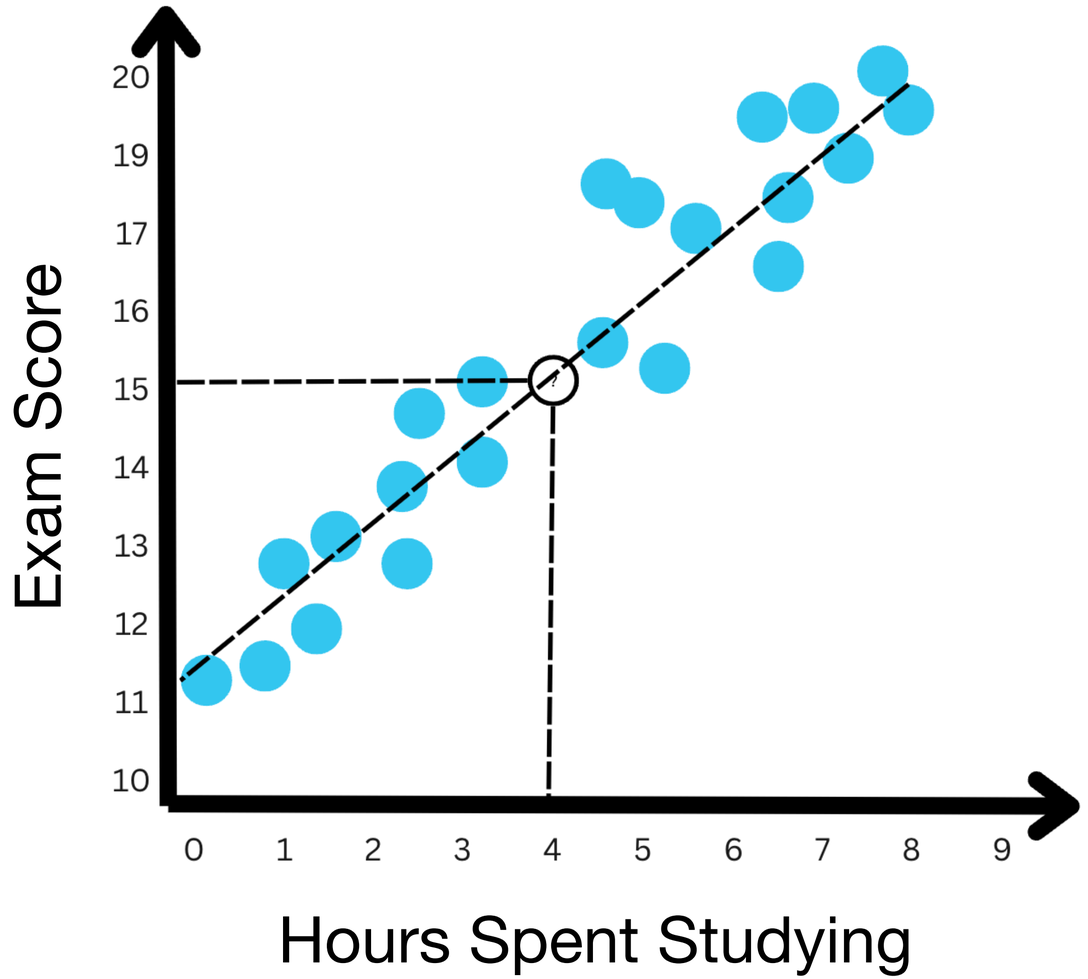
# What Can a Perceptron Do?

## Simple yet powerful



# What Can a Perceptron Do?

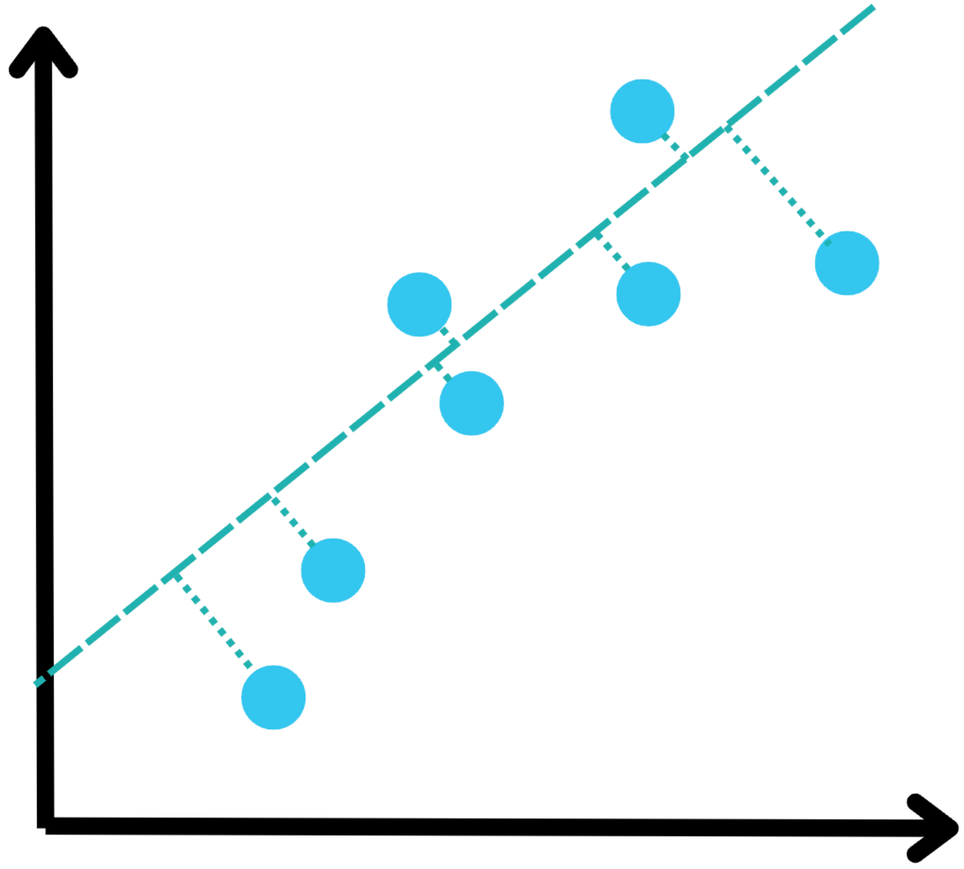
## Simple yet powerful




Goal: find a line that is close to all points

# Calculating the Error

## Formulating it to as an optimization problem



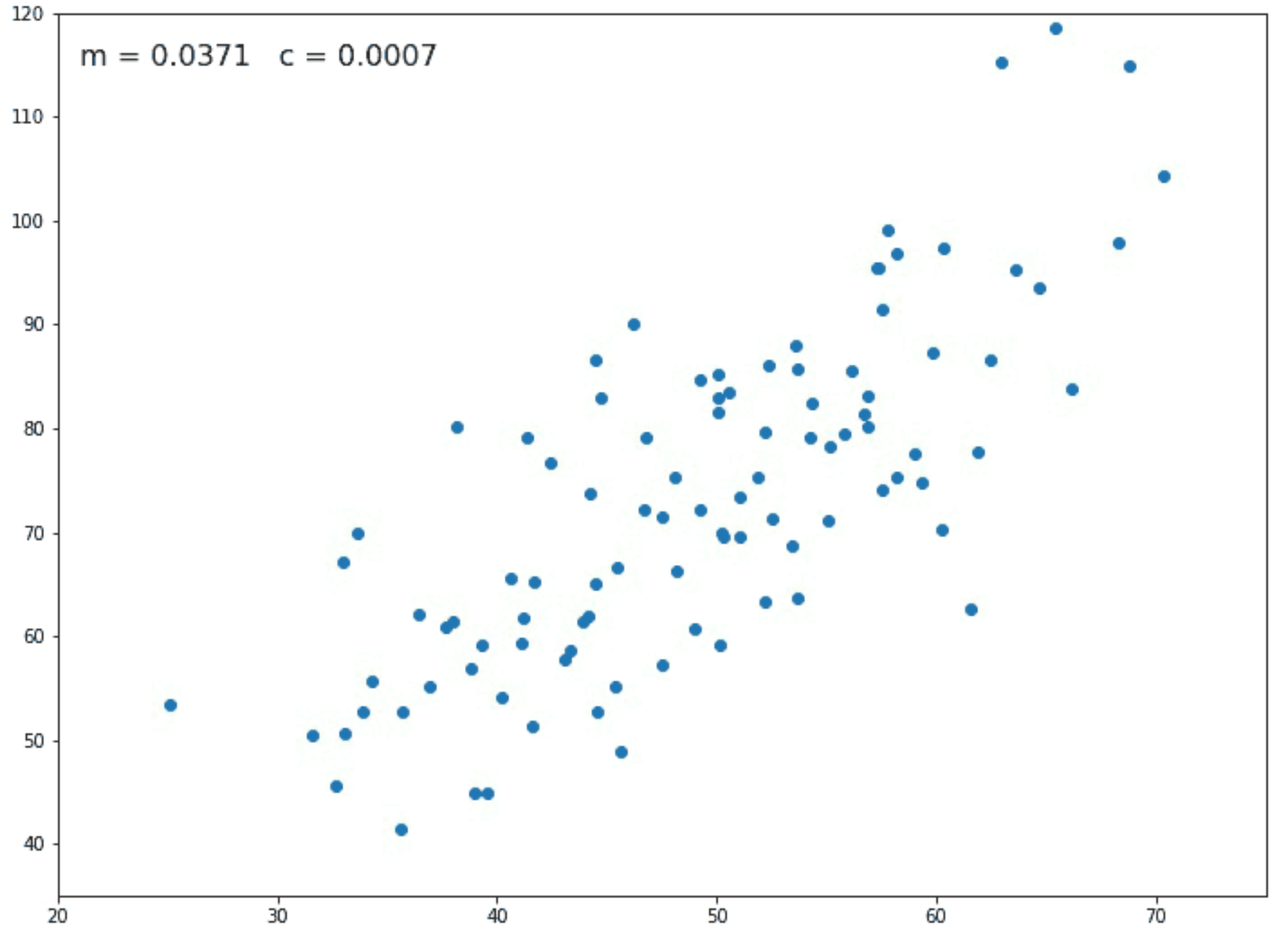
$$\text{MSE} = \underbrace{\frac{1}{n}}_{\text{میانگین}} \sum_{i=1}^{n} \underbrace{(Y_i - \hat{Y}_i)^2}_{\text{فاصله نقطه تا خط}}$$

Goal: find w and b such that they minimize the MSE

(These values are found via calculating and stepping in the opposite side of the derivatives, we wont focus on this today)

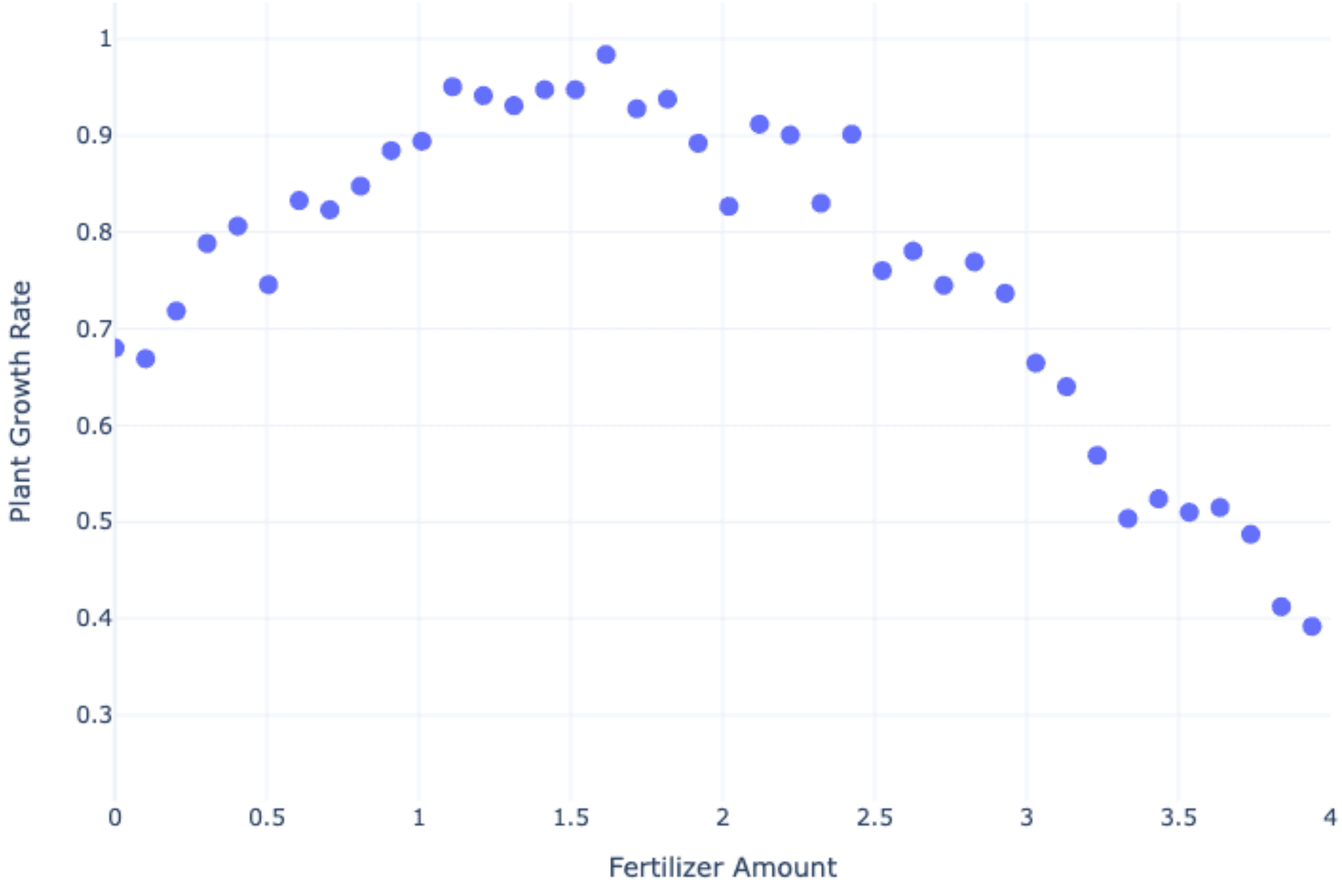
# Visualizing the Learning Process

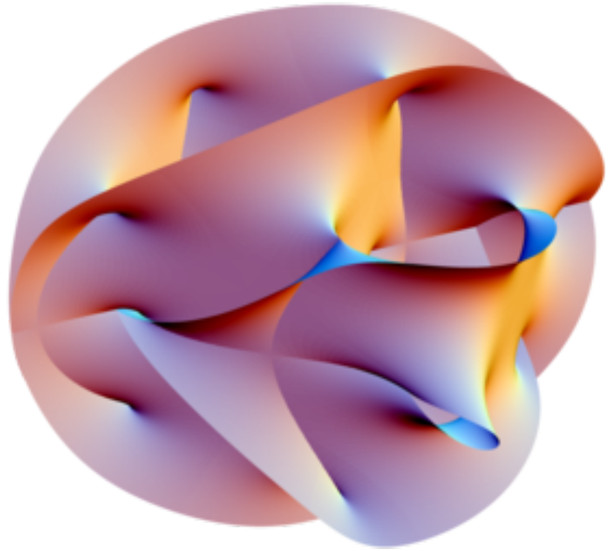
## A perceptron in action



# Limits of Single Perceptrons

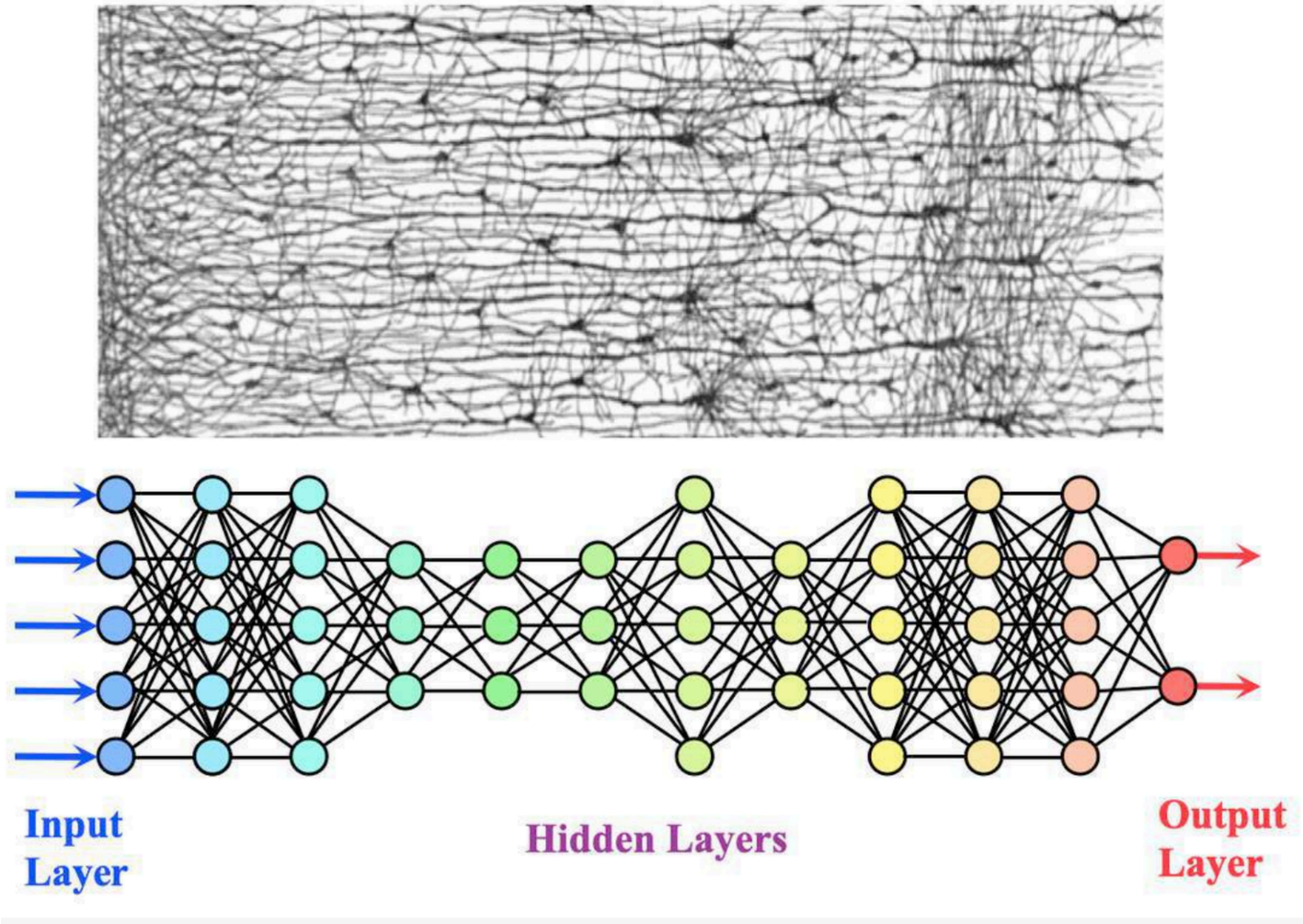
## We live in a high dimensional, non-linear world



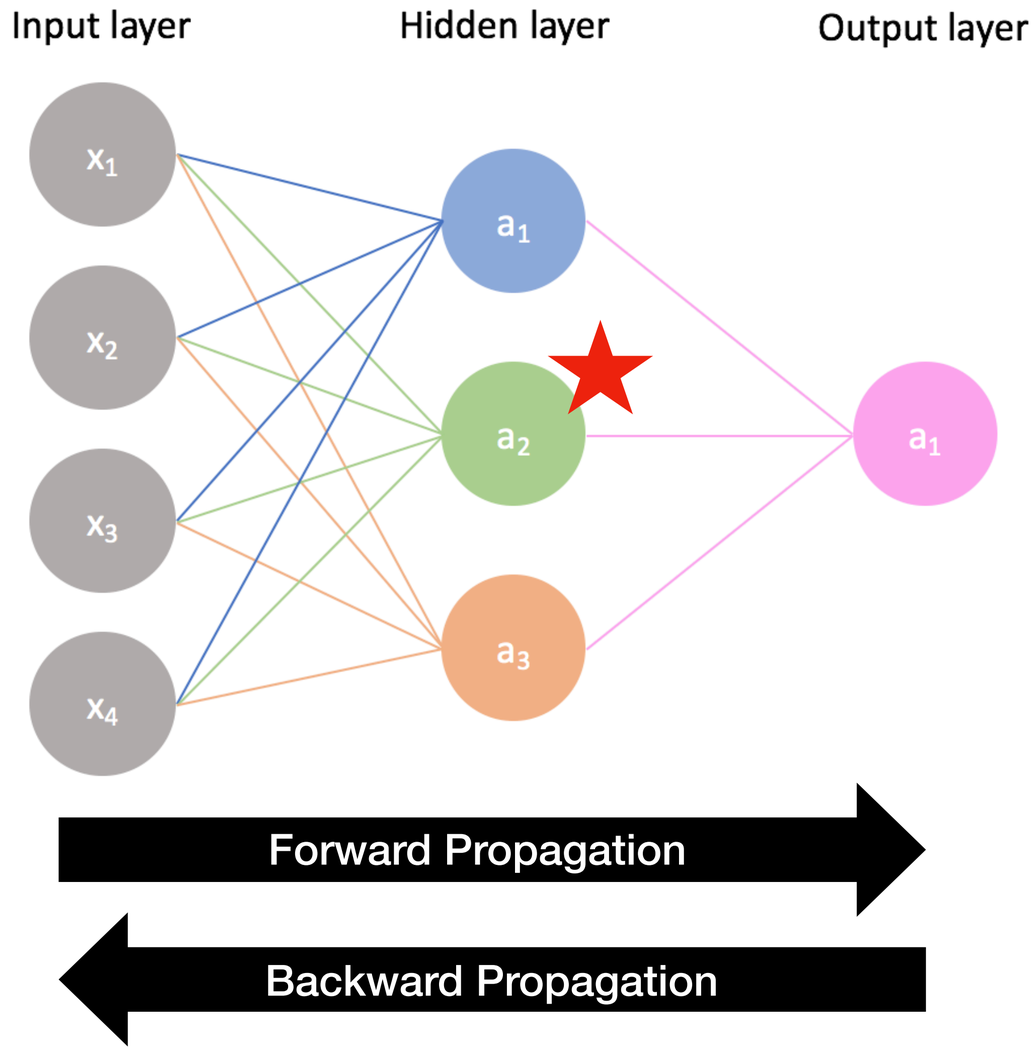
# Neural Networks

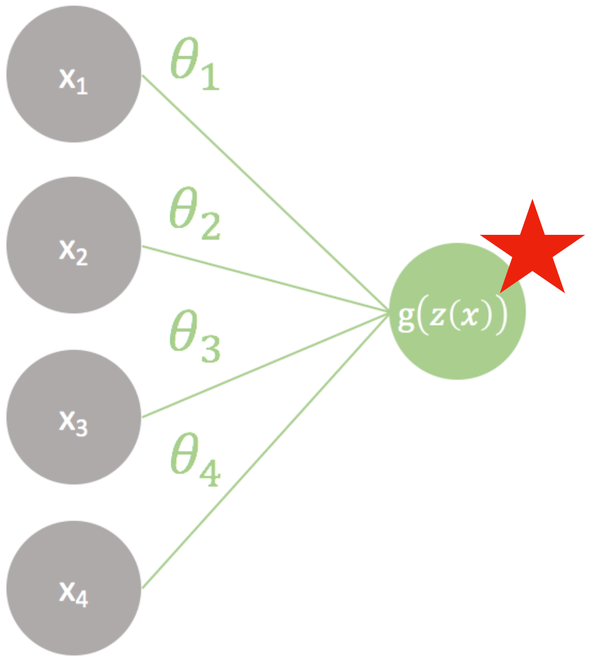
## From perceptrons to layers to networks



# Neural Networks

## A closer look

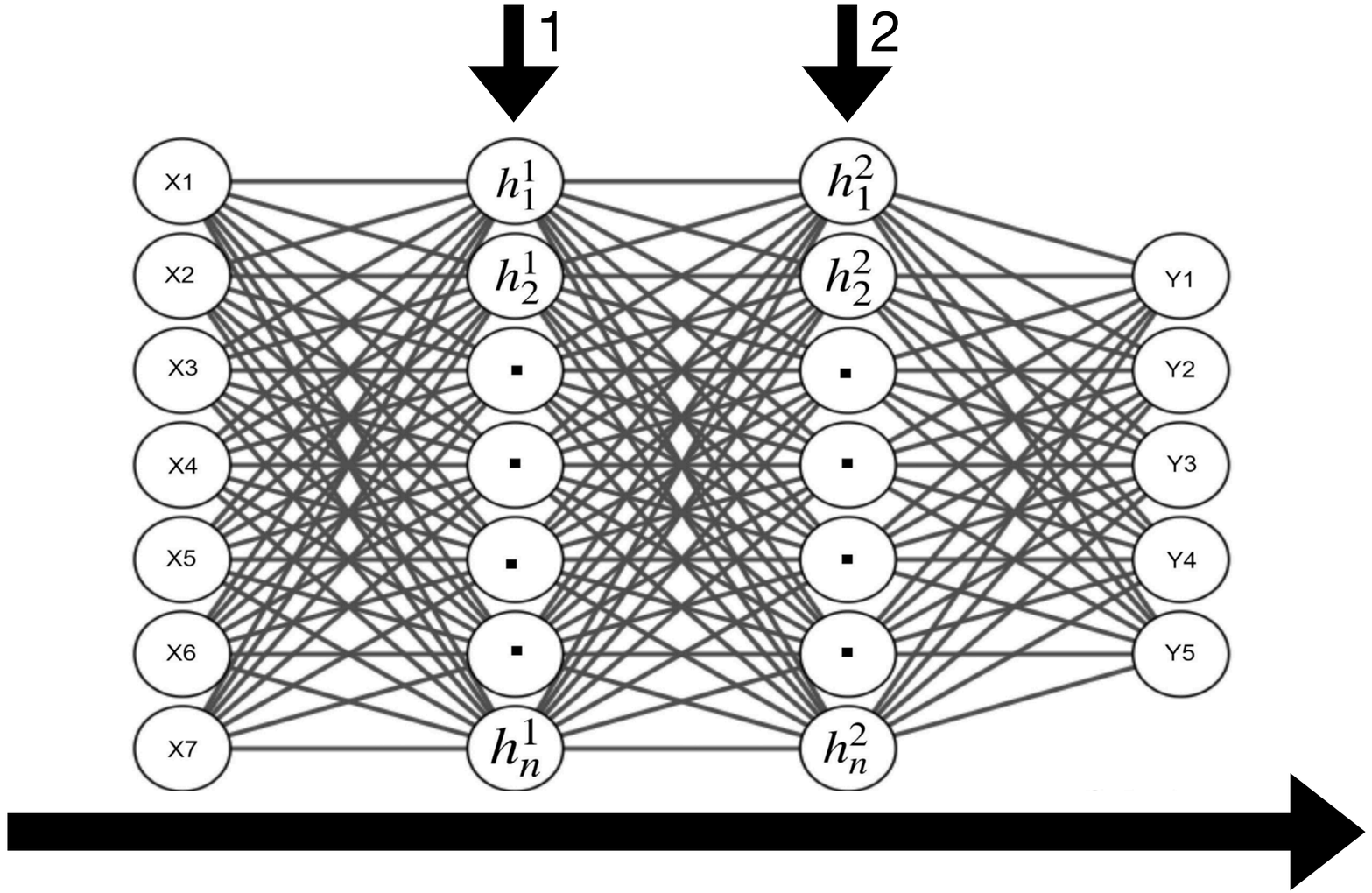


# How can we parallelize neural networks?

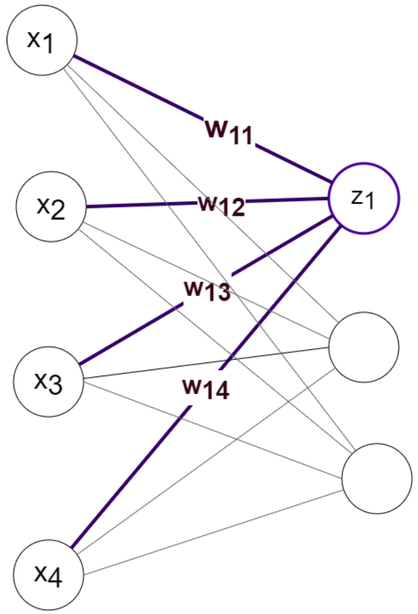
# What to Parallelize?

## A combination of sequential and parallel operations



# A Naive Parallel Implementation

## Thread 1



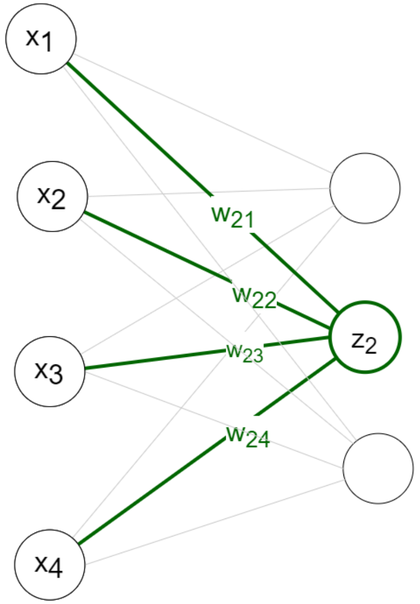

$$z_1 = w_{11} * x_1 + w_{12} * x_2 + w_{13} * x_3 + w_{14} * x_4 + b$$

Calculate Activation Function

z1 → sigmoid → a1

a1 = sigmoid(z1)

## Thread 2



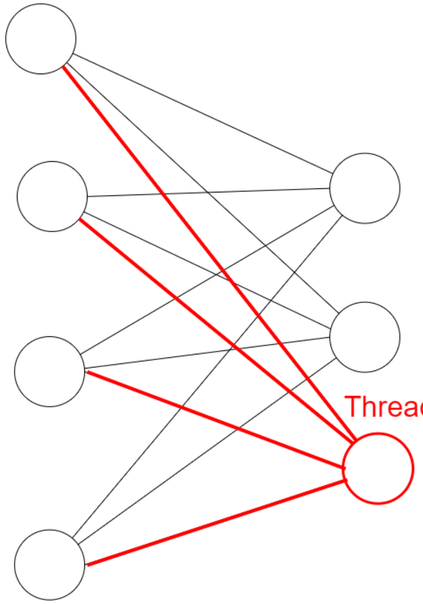

$$z_2 = w_{21} * x_1 + w_{22} * x_2 + w_{23} * x_3 + w_4 * x_4 + b$$

Calculate Activation Function

z2 → sigmoid → a2

a2 = sigmoid(z2)

## Thread 3




$$z_3 = w_{31} * x_1 + w_{32} * x_2 + w_{33} * x_3 + w_{34} * x_4 + b$$

Calculate Activation Function

z3 → sigmoid → a3

a3 = sigmoid(z3)

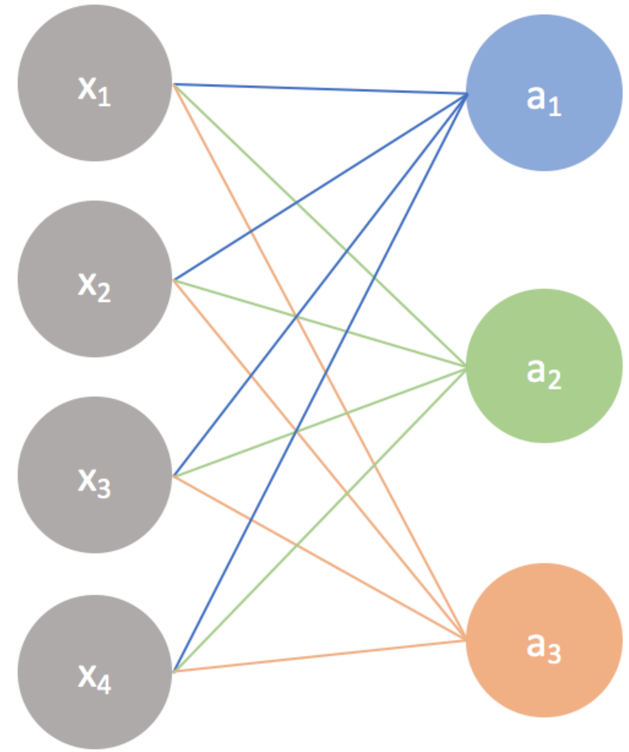
# A Naive Parallel Implementation

## Kernel code

```
__global__ void linear_layer_and_activation(float *weight_matrix, float *biases, float *x_inputs,
                                            float *z_values, float *activation_values,
                                            int nr_output_neurons, int nr_input_neurons)
{
        // We use the thread id so we can index different weights/neurons in each thread
        int id = threadIdx.x;

        // w*x
        // We loop over every incoming neuron and multiply it with it's weight and sum it to the current z_value
        for (int neuron_nr = 0; neuron_nr < nr_input_neurons; neuron_nr++)
        {
                z_values[id] += weight_matrix[(nr_input_neurons)* id + neuron_nr] * x_inputs[neuron_nr];
        }

        // w*x + b
        // Don't forget to add the bias
        z_values[id] += biases[id];

        // The we compute the activation value just like in part1 of the tutorial
        // sig(w*x + b)
        activation_values[id] = 1.0 / (1.0 + exp(-z_values[id]));

}
```

# Neural Networks

## Forward propagation as matrix multiplication



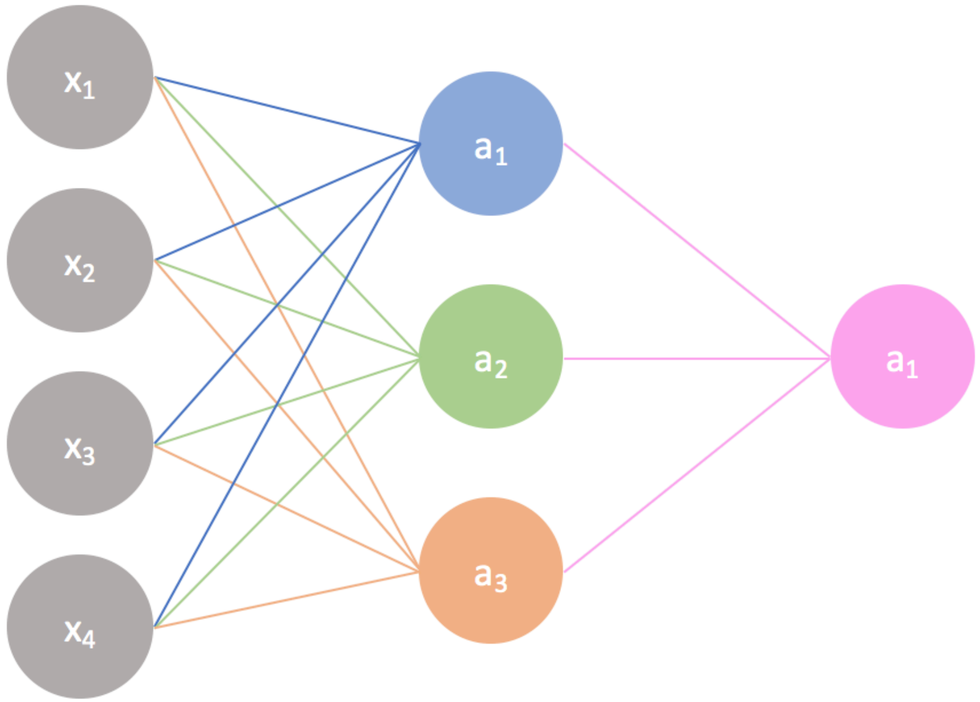

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow[activation]{} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Forward Propagation

The forward pass can be abstracted as a series of matrix multiplications, For which we have already studied efficient methods. We will come back to this.

# Creating NNs with PyTorch

## Simplicity of high-level DL frameworks




```
import torch
import torch.nn as nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(4, 3),
            nn.ReLU(),
            nn.Linear(3, 1)
        )

    def forward(self, x):
        return self.model(x)

# Create an instance of the NeuralNetwork class
model = NeuralNetwork()
print(model)
```

# Convolutional NNs:
# Neural Networks on Steroids

# Limits of Regular NNs

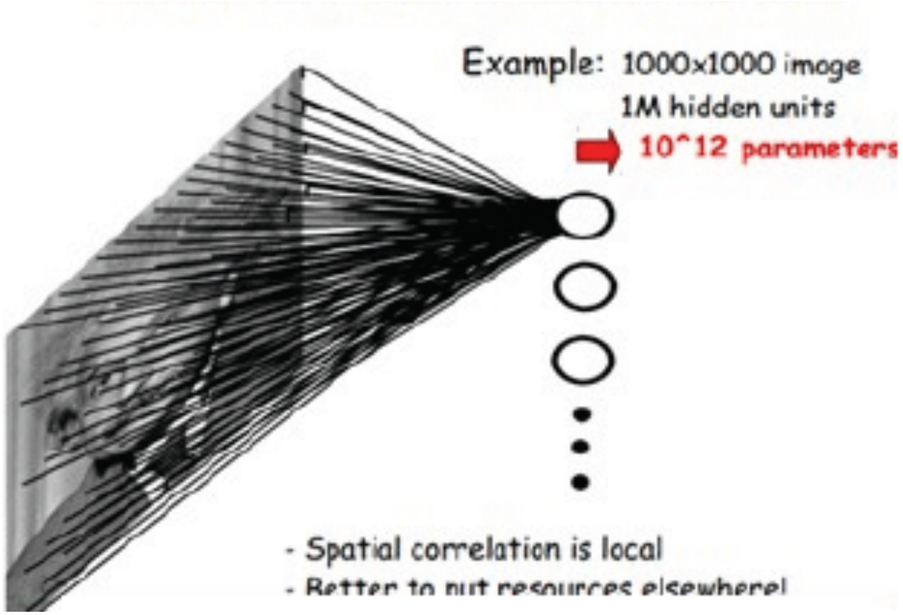
## Importance of feature extraction

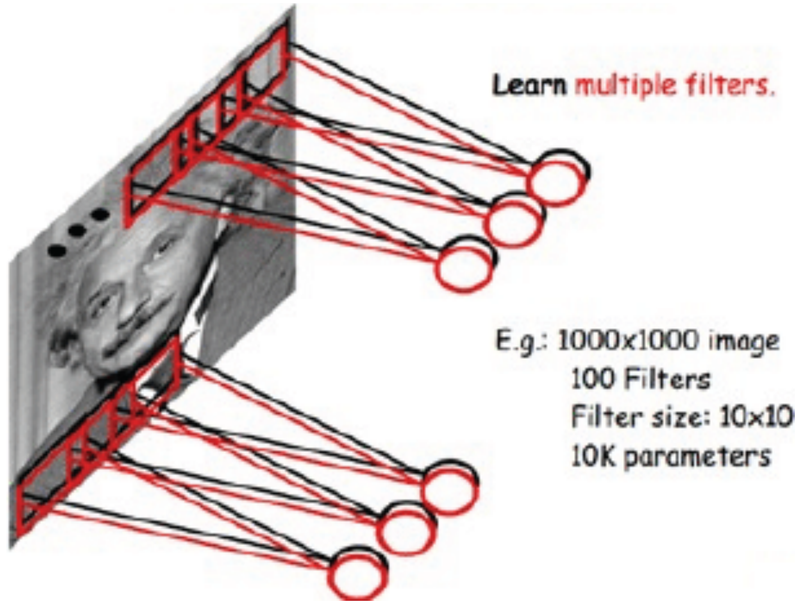

(a)


(b)

# Limits of Regular NNs

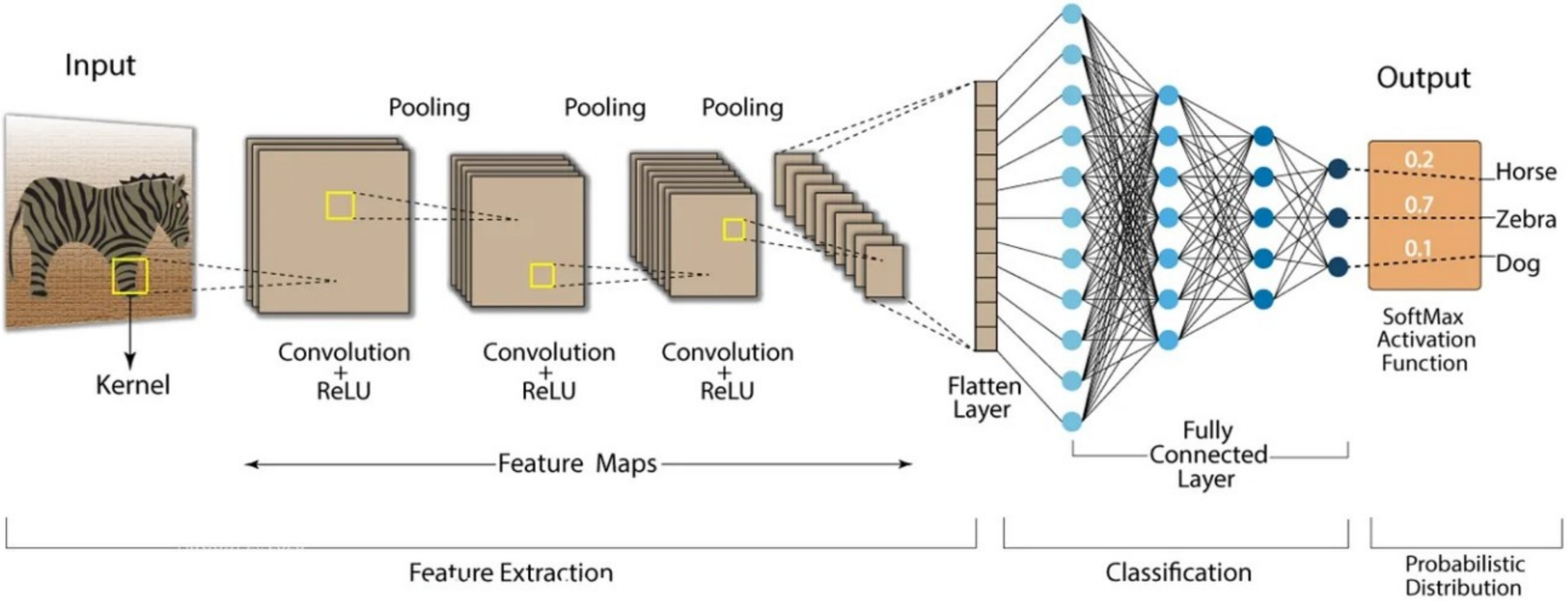
## Unfeasible for large inputs (e.g. images)



# Convolutional Neural Networks

## Giving computers the power understand images



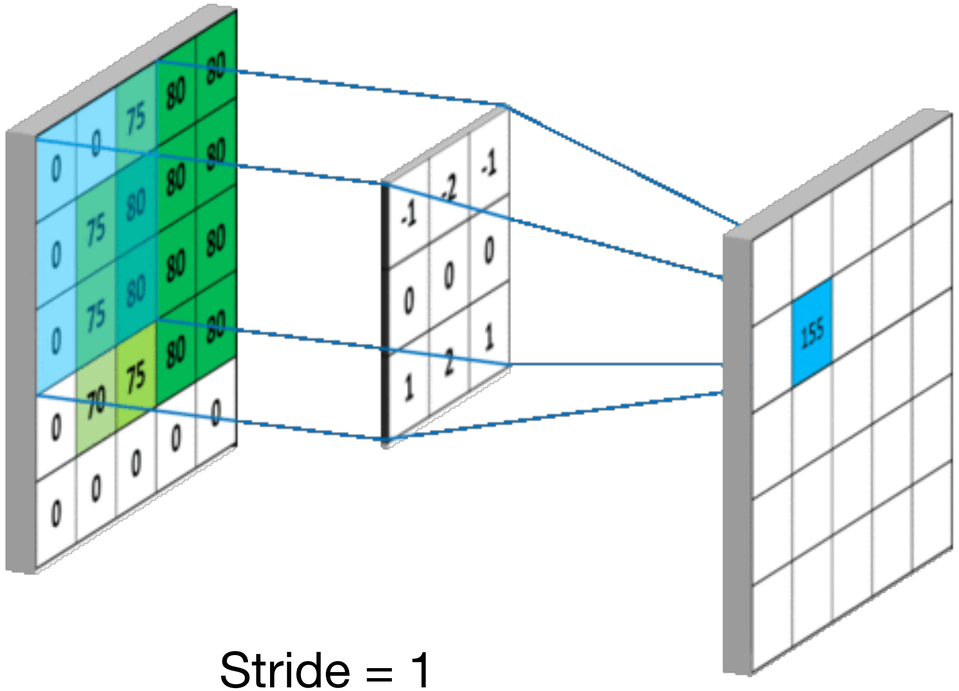
# The Convolution Operation

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

(a) Vertical edge detector

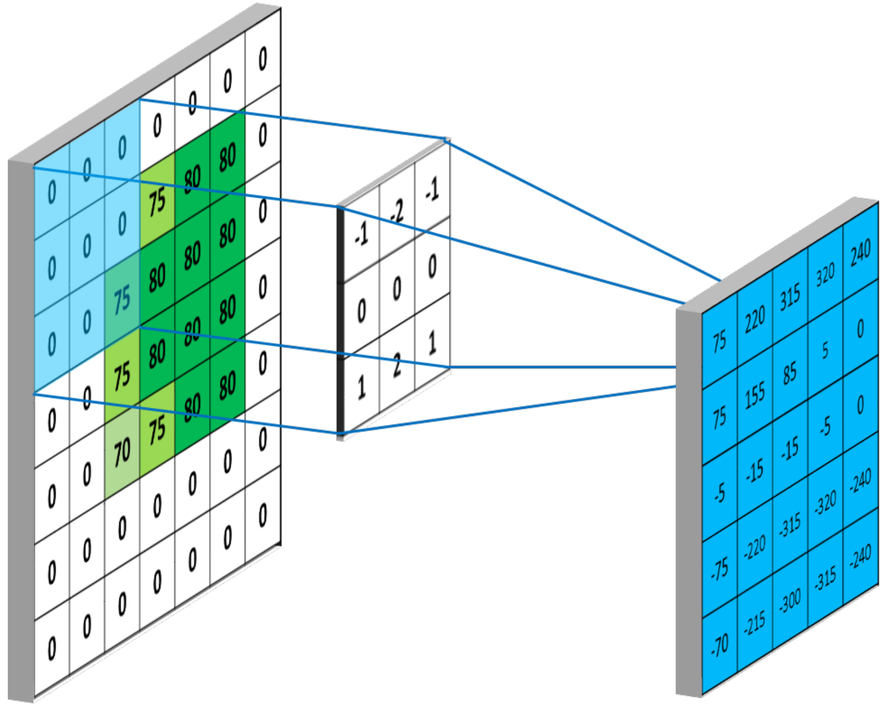
# Convolution Layers

## The first half of CNNs



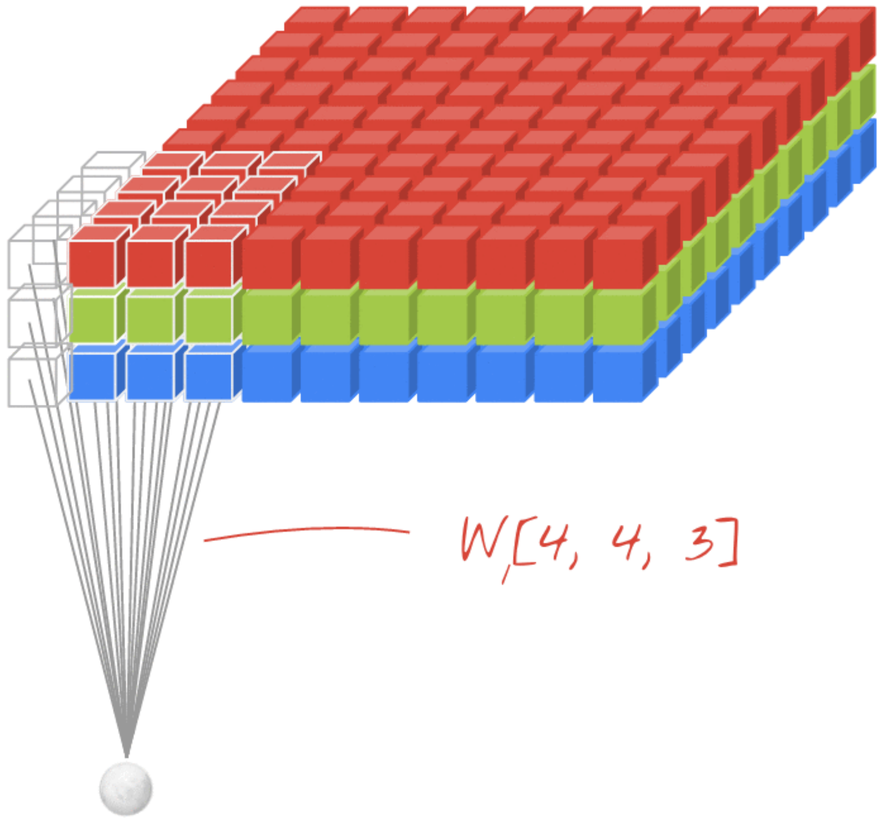

Stride = 1

Kernel Size = 3



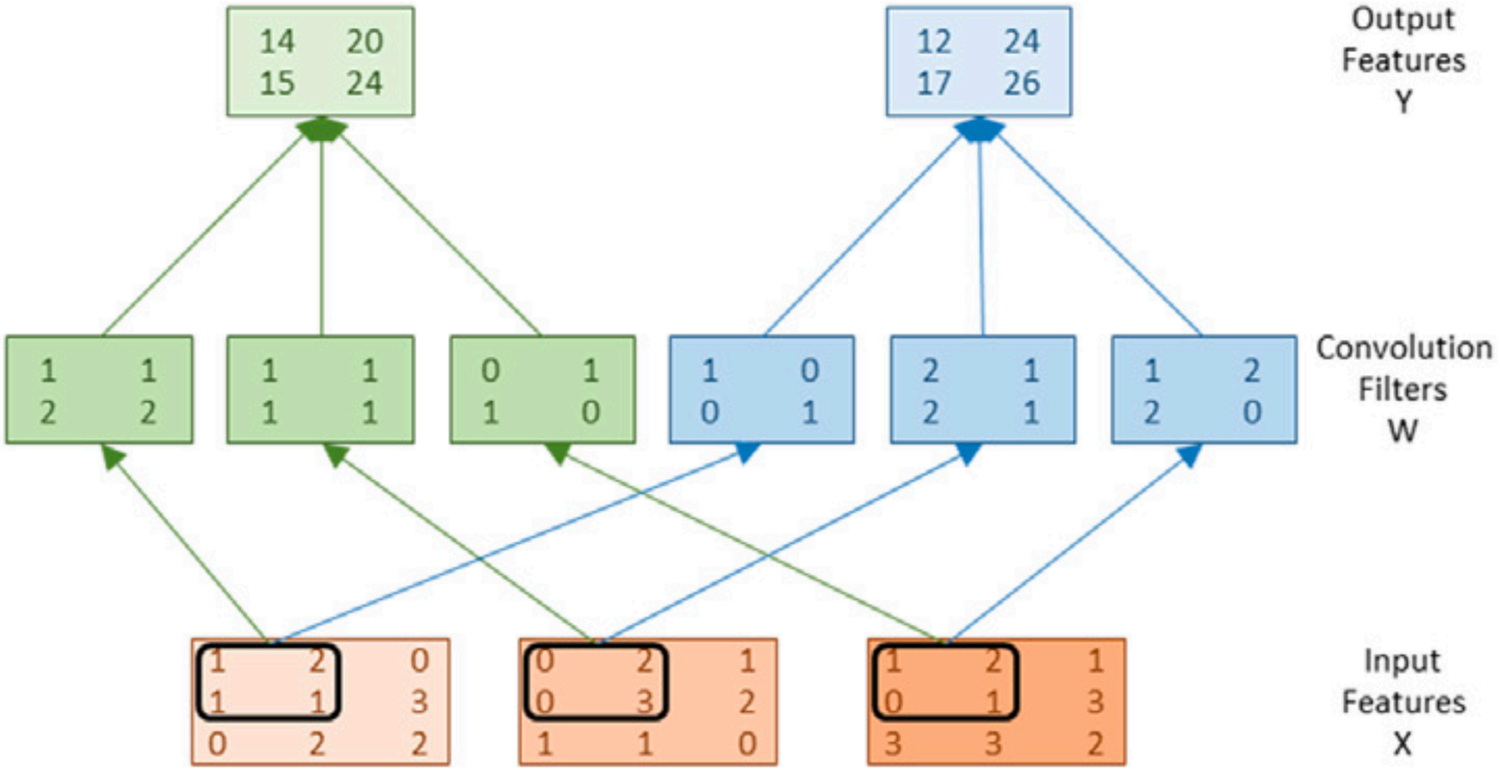
# Convolution Layers

## Multi-layer 3d layers



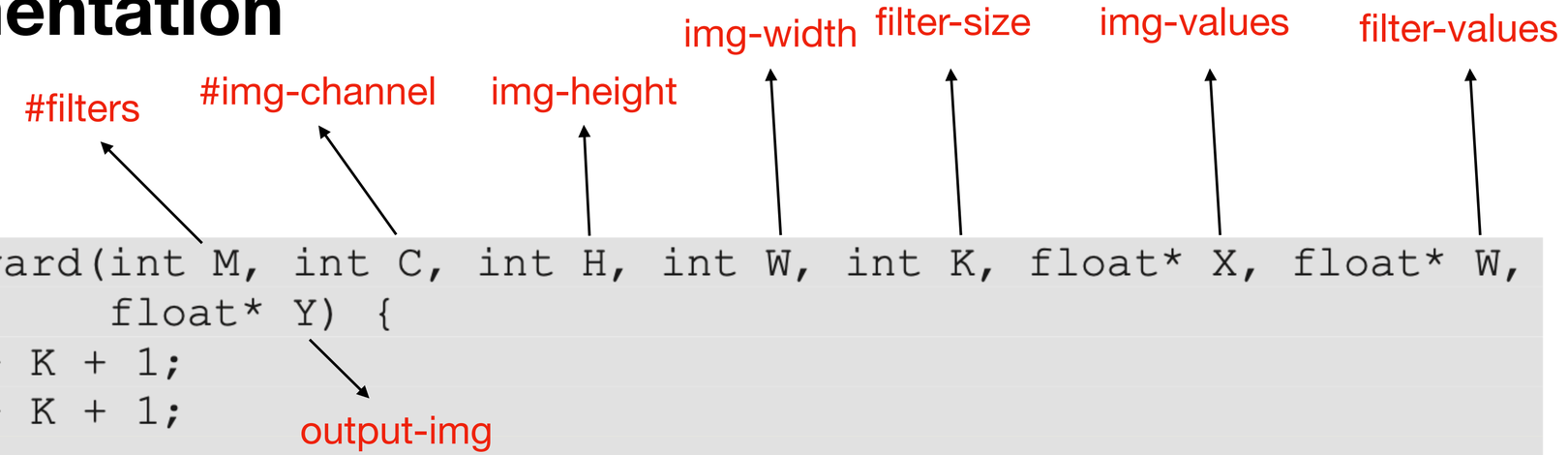
# Convolution Layers

## Yet another example



# Coding Convolutional Layers

## A serialized implementation




```
01  void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W,
                           float* Y) {
02      int H_out = H - K + 1;
03      int W_out = W - K + 1;

04      for(int m = 0;  m < M;  m++)              // for each output feature map
05         for(int h = 0; h < H_out; h++)         // for each output element
06            for(int w = 0; w < W_out; w++) {
07               Y[m, h, w] = 0;
08               for(int c = 0;  c < C; c++)     // sum over all input feature maps
09                  for(int p = 0; p < K; p++) // KxK  filter
10                     for(int q = 0; q < K; q++)
11                        Y[m, h, w] +=  X[c, h + p, w + q] * W[m, c, p, q];
12            }
13   }
```

# Parallel Implementation; Kernel Launch

- each thread computes one element of one output feature map
- each block is 2d and computes a tile of TILE_WIDTH x TILE_WIDTH pixels in one output feature map
- The grid is 2d, The first dimension (X) corresponds to the (M) output features maps covered by each block. The second dimension (Y) reflects the location of a block's output tile inside the output feature map.

```
01    # define TILE_WIDTH 16
02    W_grid = W_out/TILE_WIDTH;   // number of horizontal tiles per output map
03    H_grid = H_out/TILE_WIDTH;   // number of vertical tiles per output map
04    T = H_grid * W_grid;
05    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
06    dim3 gridDim(M, T, N);
07    ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

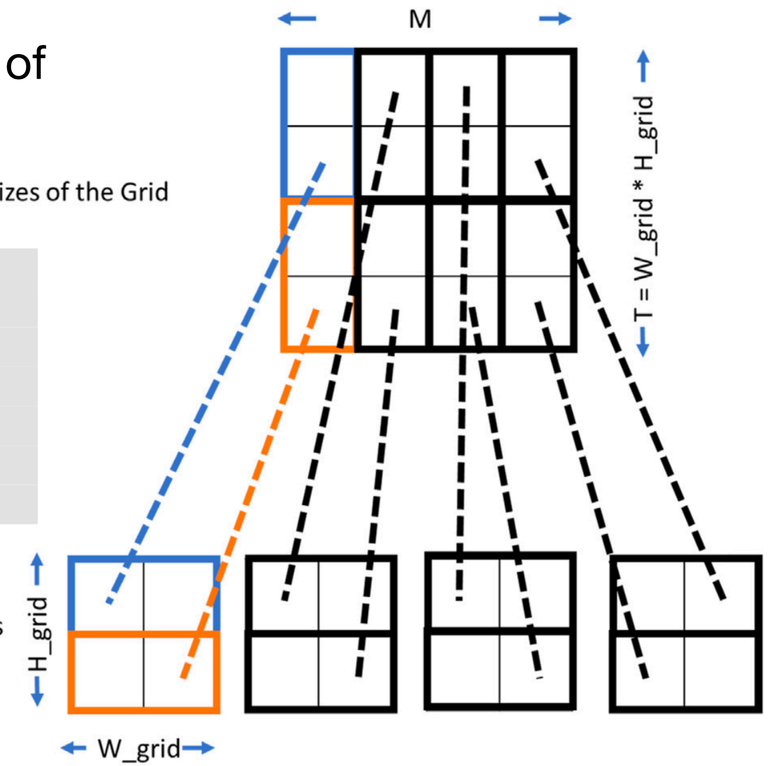
W_out = 128 —> W_grid = 128/16 = 8

H_out = 128 —> H_grid = 128/16 = 8

T = H_grid * W_grid = 8*8 = 64

| Grid | Block |
|---|---|
| 10x64 | 16x16 |

X-Y dimension sizes of the Grid



# Coding Convolutional Layers

## A parallel implementation

output-img

#img-channels

filter-size

img-values

filter-values

Filter number

Start pixel height

Start pixel width

Image batch #

```
01  __global__ void
02  ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W,
                            float* Y) {
03      int m = blockIdx.x;
04      int h = (blockIdx.y / W_grid)*TILE_WIDTH + threadIdx.y;
05      int w = (blockIdx.y % W_grid)*TILE_WIDTH + threadIdx.x;
06      int n = blockIdx.z;
07      float acc = 0.;
08      for (int c = 0;  c < C; c++) {          // sum over all input channels
09          for (int p = 0; p < K; p++)         // loop over KxK  filter
10              for (int q = 0; q < K; q++)
11                  acc += X[n, c, h + p, w + q] * W[m, c, p, q];
12      }
13      Y[n, m, h, w] = acc;
14  }
```

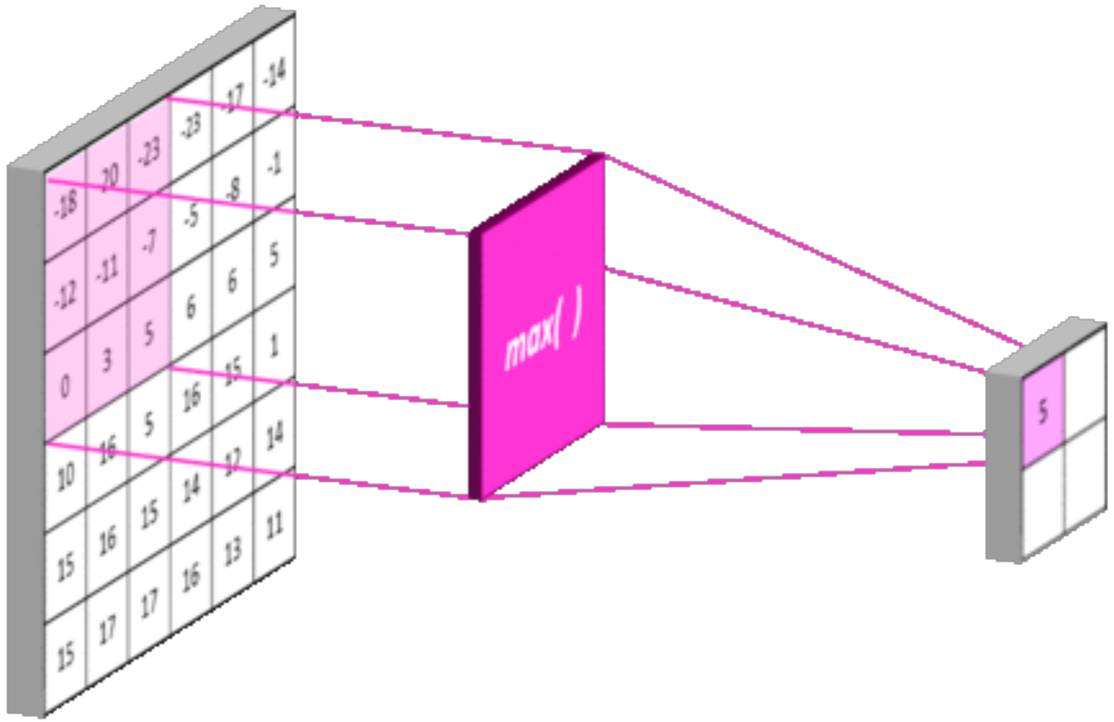
# Coding Convolution Layers

## Limitations and fixes

- High degree of parallelism
- Consumes too much global memory bandwidth
- Fix:
  - Using constant memory caching
  - Using shared memory tiling

# Pooling Layers

## The second half of CNNs




Stride = 3

Kernel Size = 3

# Coding Pooling Layers

## A serialized implementation


```
01   void subsamplingLayer_forward(int M, int H, int W, int K, float* Y, float*
S){
02      for(int m = 0;  m < M;  m++)                  // for each output feature map
03         for(int h = 0; h < H/K; h++)               // for each output element,
04            for(int w = 0; w < W/K; w++) {          // this code assumes that H and W
05               S[m, x, y] = 0.;                    // are multiples of K
06               for(int p = 0; p < K; p++) {        // loop over KxK input samples
07                  for(int q = 0; q < K; q++)
08                     S[m, h, w] += Y[m, K*h + p, K*w+ q] /(K*K);
09               }
                                    // add bias and apply non-linear activation
10               S[m, h, w] = sigmoid(S[m, h, w] + b[m]);
11            }
12   }
```

# Coding Pooling Layers

## A parallel implementation

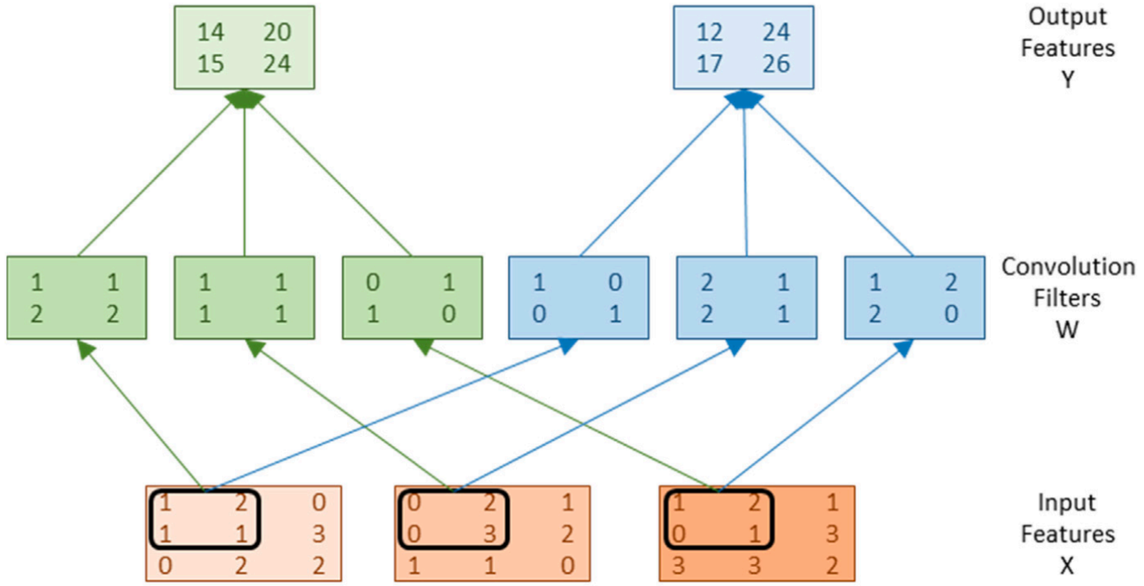
```
__global__ void averagePooling(float* input, float* output, int input_width, int input_height,
                               int output_width, int output_height, int pool_size) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < output_width && y < output_height) {
        int start_x = x * pool_size;
        int start_y = y * pool_size;
        int end_x = min(start_x + pool_size, input_width);
        int end_y = min(start_y + pool_size, input_height);

        float sum = 0.0f;
        int count = 0;

        for (int i = start_y; i < end_y; i++) {
            for (int j = start_x; j < end_x; j++) {
                sum += input[i * input_width + j];
                count++;
            }
        }

        output[y * output_width + x] = sum / count;
    }
}
```

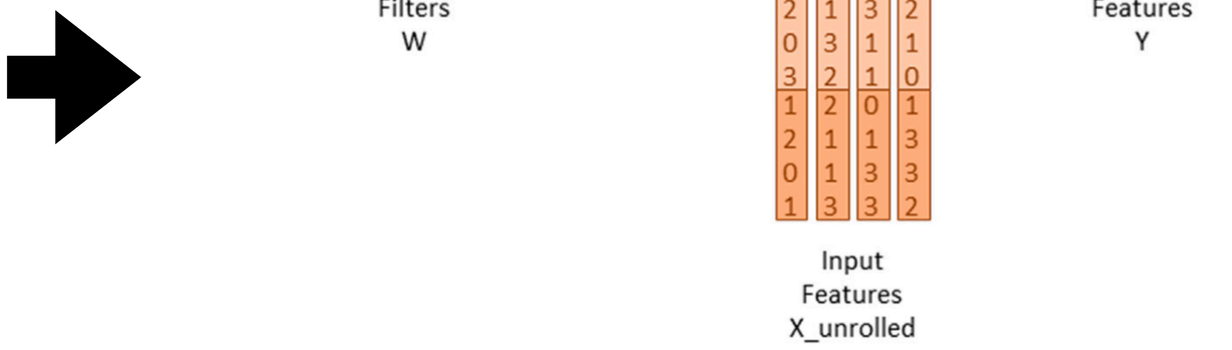
# This might look good, but we can do much better…!

Remember how we could represent a neural network as a series of matrix multiplications?

# Unrolling Convolution Layers

## Formulating the problem as matrix multiplication

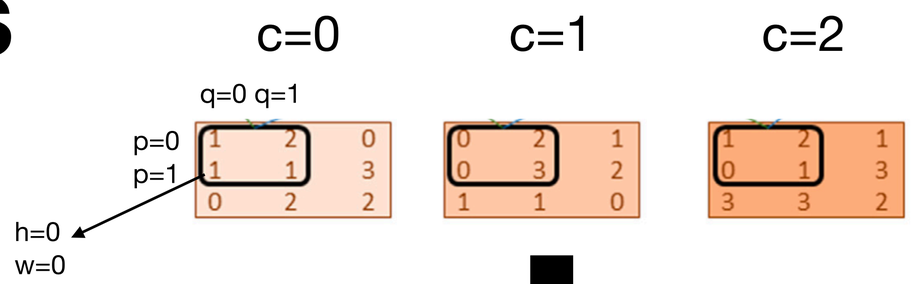



Once we unroll it, the resulting multiplication can be easily solved using warptiling or even cuBLAS

# Unrolling Convolution Layers

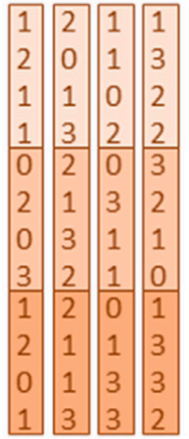
## Sequencial implementation




#img-channel

filter-size

```
01     void unroll(int C, int H, int W, int K, float* X, float* X_unroll) {
02         int H_out = H - K + 1;
03         int W_out = W - K + 1;
04 #channels for(int c = 0; c < C; c++) {
               // Beginning row index of the section for channel C input feature
               // map in the unrolled matrix
05     Start-index  w_base = c * (K*K);
06             for(int p = 0; p < K; p++) {
07                 for(int q = 0; q < K; q++) {
08                     for(int h = 0; h <  H_out; h++) {
09                         int h_unroll = w_base + p*K + q;
10                         for(int w = 0; w < W_out; w ++) {
11                             int w_unroll = h * W_out + w;
12                             X_unroll[h_unroll, w_unroll) = X(c, h + p, w + q);
13                         }
14                     }
15                 }
16             }
17         }
18     }
```



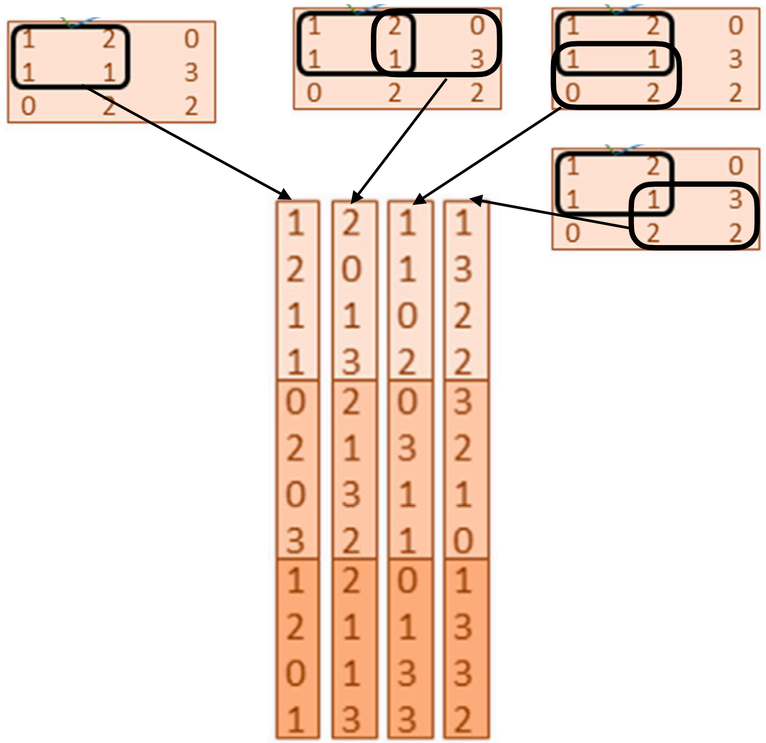
# Unrolling Convolution Layers

## Parallel implementation

#img-channel

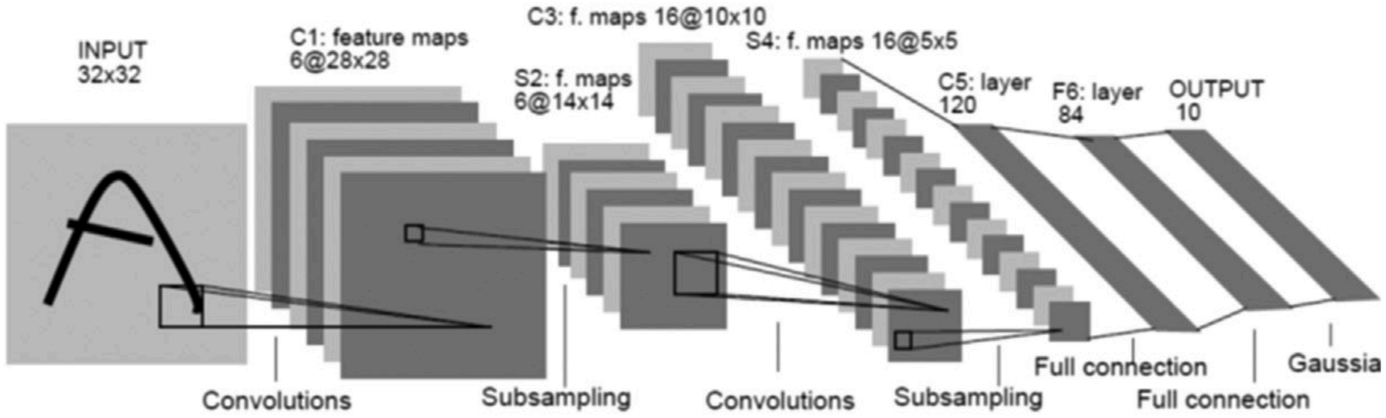
```
01    __global__ void
02    unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll) {
03        int t = blockIdx.x * blockDim.x + threadIdx.x;
04        int H_out = H - K + 1;
05        int W_out = W - K + 1;
          // Width of the unrolled input feature matrix
06        int W_unroll = H_out * W_out;
07        if (t < C * W_unroll) {
              // Channel of the input feature map being collected by the thread
08            int c = t / W_unroll;
              // Column index of the unrolled matrix to write a strip of
              // input elements into (also, the linearized index of the output
              // element for which the thread is collecting input elements)
09            int w_unroll = t % W_unroll;
              // Horizontal and vertical indices of the output element
10            int h_out = w_unroll / W_out;
11            int w_out = w_unroll % W_out;
              // Starting row index for the unrolled matrix section for channel c
12            int w_base = c * K * K;
13            for(int p = 0; p < K; p++)
14                for(int q = 0; q < K; q++) {
                      // Row index of the unrolled matrix for the thread to write
                      // the input element into for the current iteration
15                    int h_unroll = w_base + p*K + q;
16                    X_unroll[h_unroll, w_unroll] = X[c, h_out + p, w_out + q];
```

- Each CUDA thread will be responsible for gathering (K*K) input elements from one input feature map
- The total number of threads will be (C * H_out * W_out)



# LeNet-5

## A revolution in image processing




```python
class LeNet5(nn.Module):
    def __init__(self, num_classes):
        super(ConvNeuralNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(6),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.fc = nn.Linear(400, 120)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(120, 84)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(84, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out = self.relu(out)
        out = self.fc1(out)
        out = self.relu1(out)
        out = self.fc2(out)
        return out
```

Learn PyTorch in 1 hour:

PyTorch

**Everything you need to know to get started**

ARTIFICIAL INTELLIGENCE ASSOCIATION · RAZI UNIVERSITY ·

www.aparat.com/v/MOhuQ

# CuDNN: A Higher Level of Abstraction

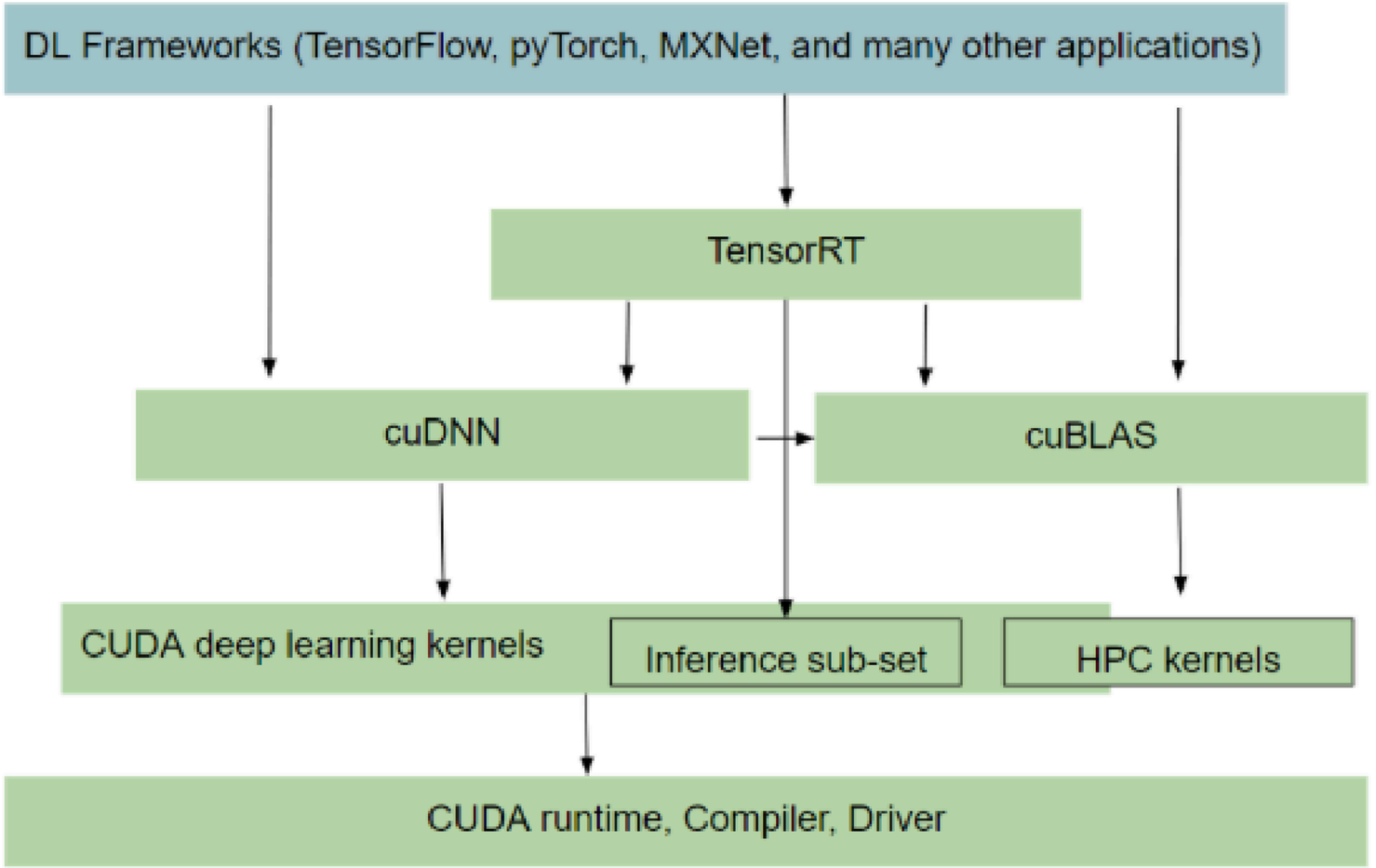
## NVIDIA's Deep Neural Network library

- A library of pre-written primitives CUDA kernels for deep neural networks
- Accelerated compute-bound operations like convolution and matmul
- Optimized memory-bound operations like pooling, softmax, normalization and activation
- Example APIs:

> cudnnConvolutionBackwardBias()
> cudnnConvolutionBackwardData()
> cudnnConvolutionBackwardFilter()
> cudnnConvolutionBiasActivationForward()
> cudnnConvolutionForward()
> cudnnCreateAttnDescriptor()
> cudnnCreateConvolutionDescriptor()
> cudnnPoolingBackward()
> cudnnPoolingForward()
> cudnnReduceTensor()
> cudnnReorderFilterAndBias()
> cudnnRNNGetClip_v8()
> cudnnRNNSetClip_v8()
> cudnnScaleTensor()
> cudnnSetAttnDescriptor()

# Where CuDNN Fits in the DL Ecosystem

## A backbone of modern DL Frameworks



# Summary and Conclusion

## Things you have (hopefully) learned:

- How neural networks are changing the world
- Brief explanation of how neural networks work under the hood
- CNNs, convolutions and pooling layers
- How to both naively and professionally parallelize neural networks
- How libraries like cuBLAS and cuDNN can simplify the process

# The End

## Thanks for your attention