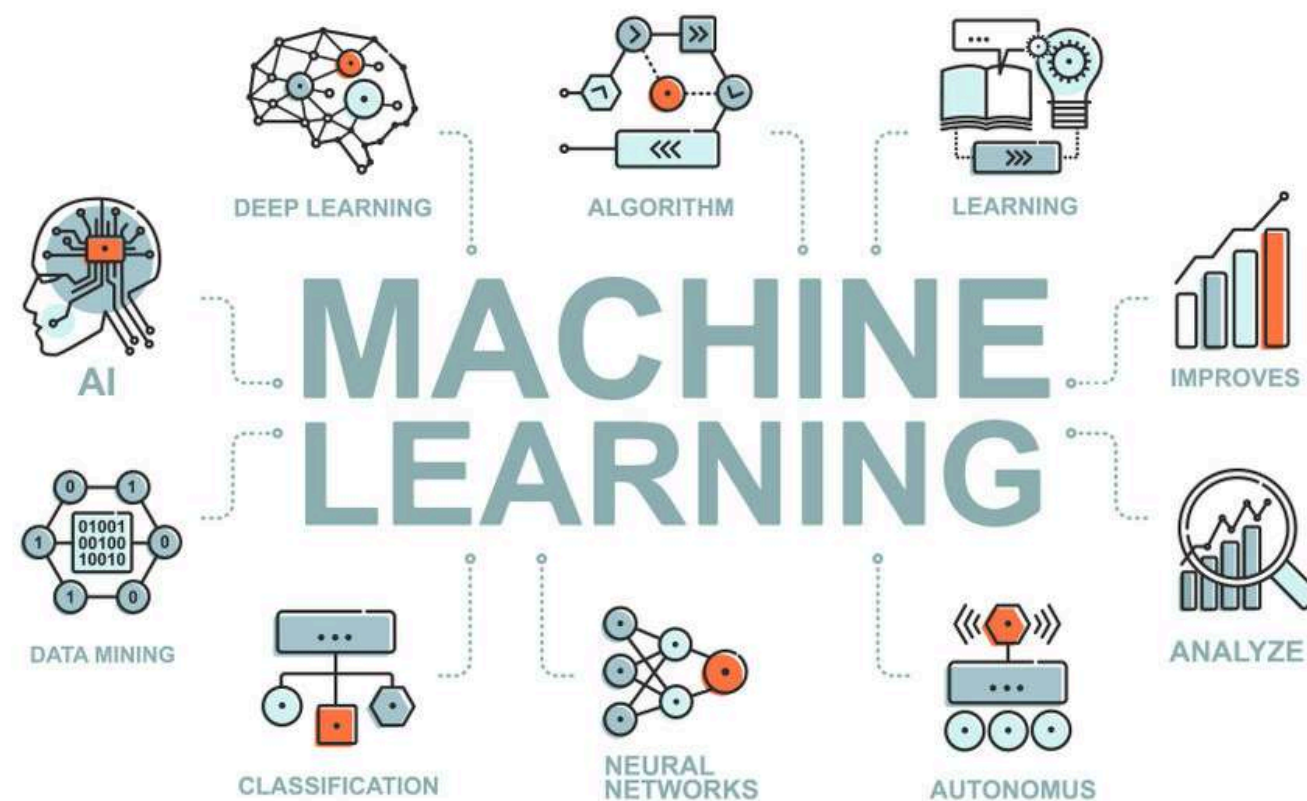
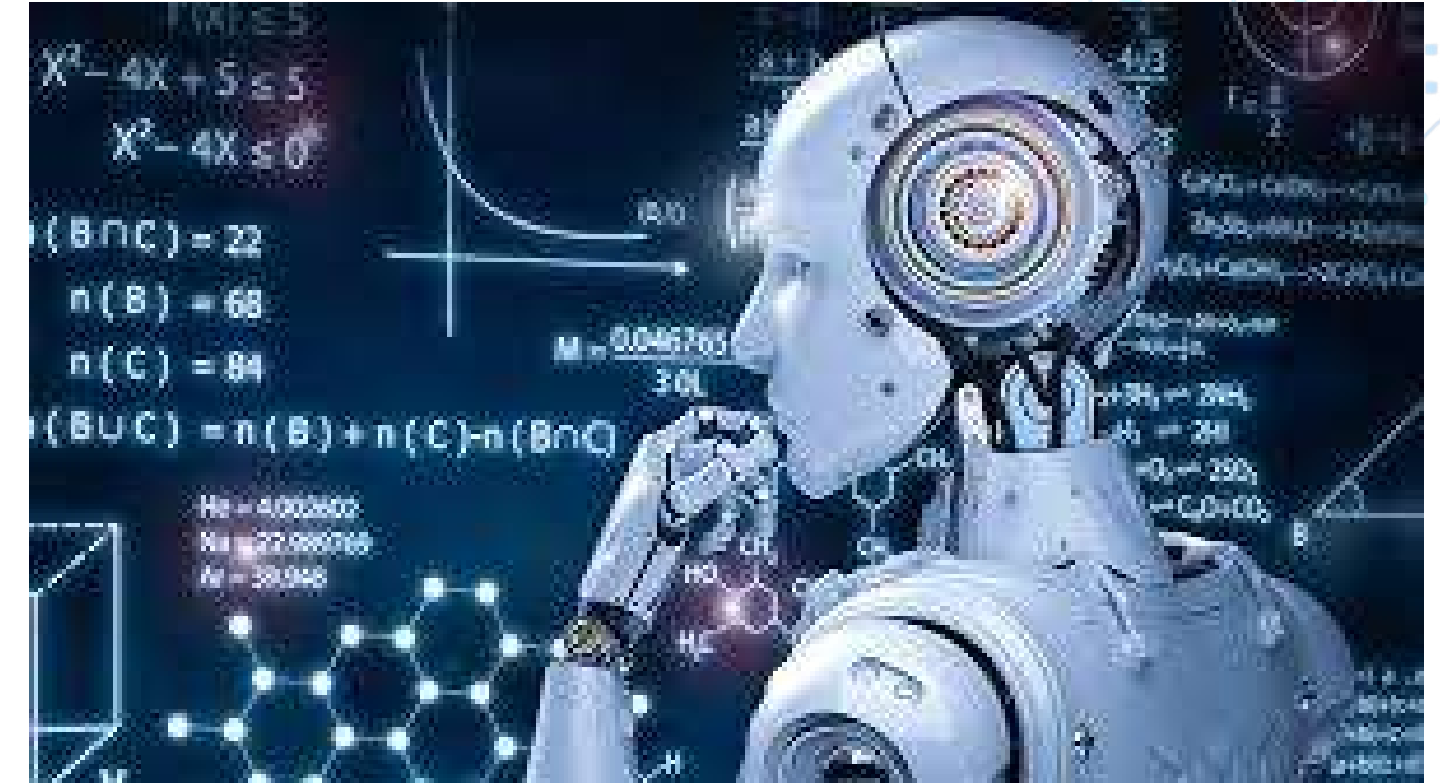


Introduction To Deep Learning

By Fikireab Mekuriaw

Introduction

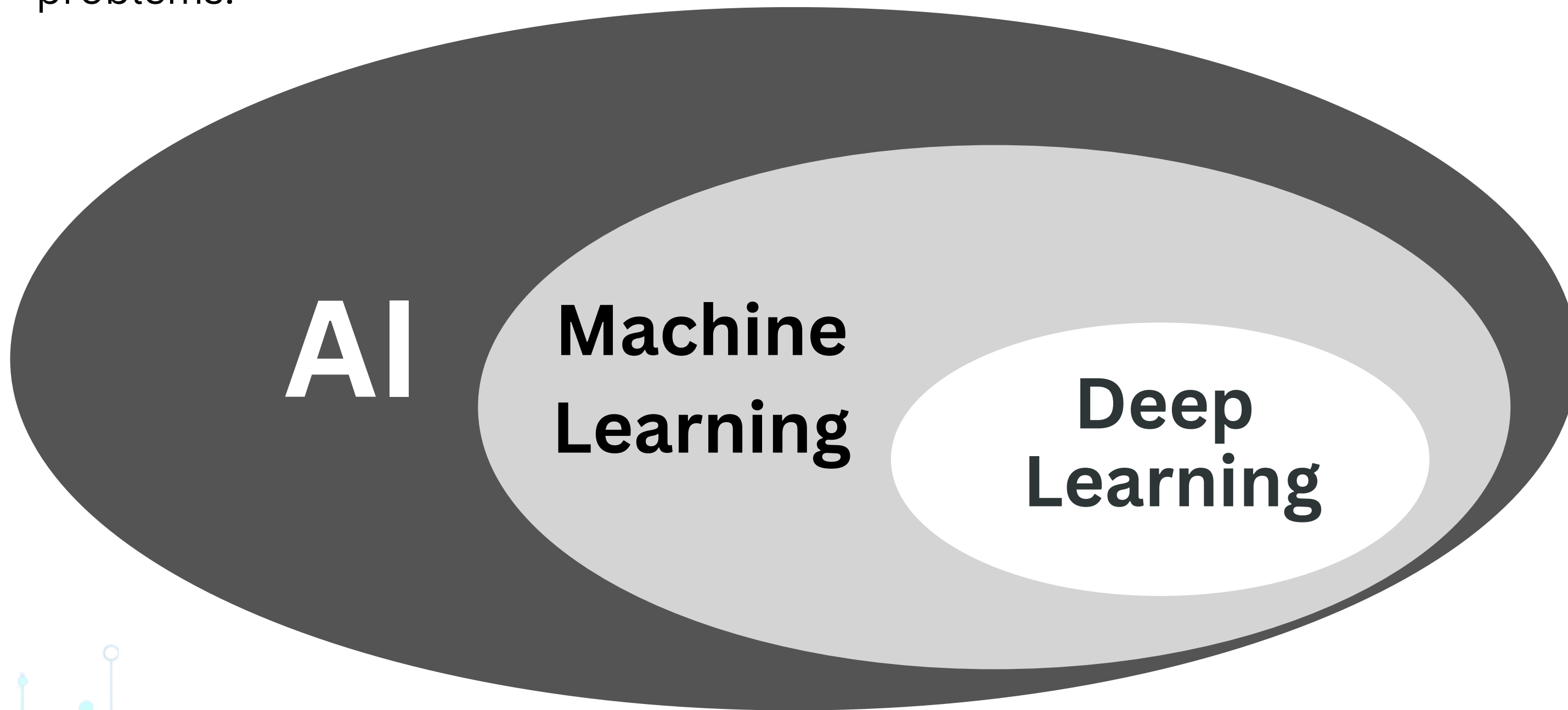
- AI refers to the simulation of human intelligence in machines.
- Machines programmed to think, learn, and make decisions in ways that mimic human abilities, like recognizing patterns, solving problems, or even understanding languages.



- Machine Learning (ML) focuses on teaching computers to learn from data and improve their performance without being explicitly programmed.

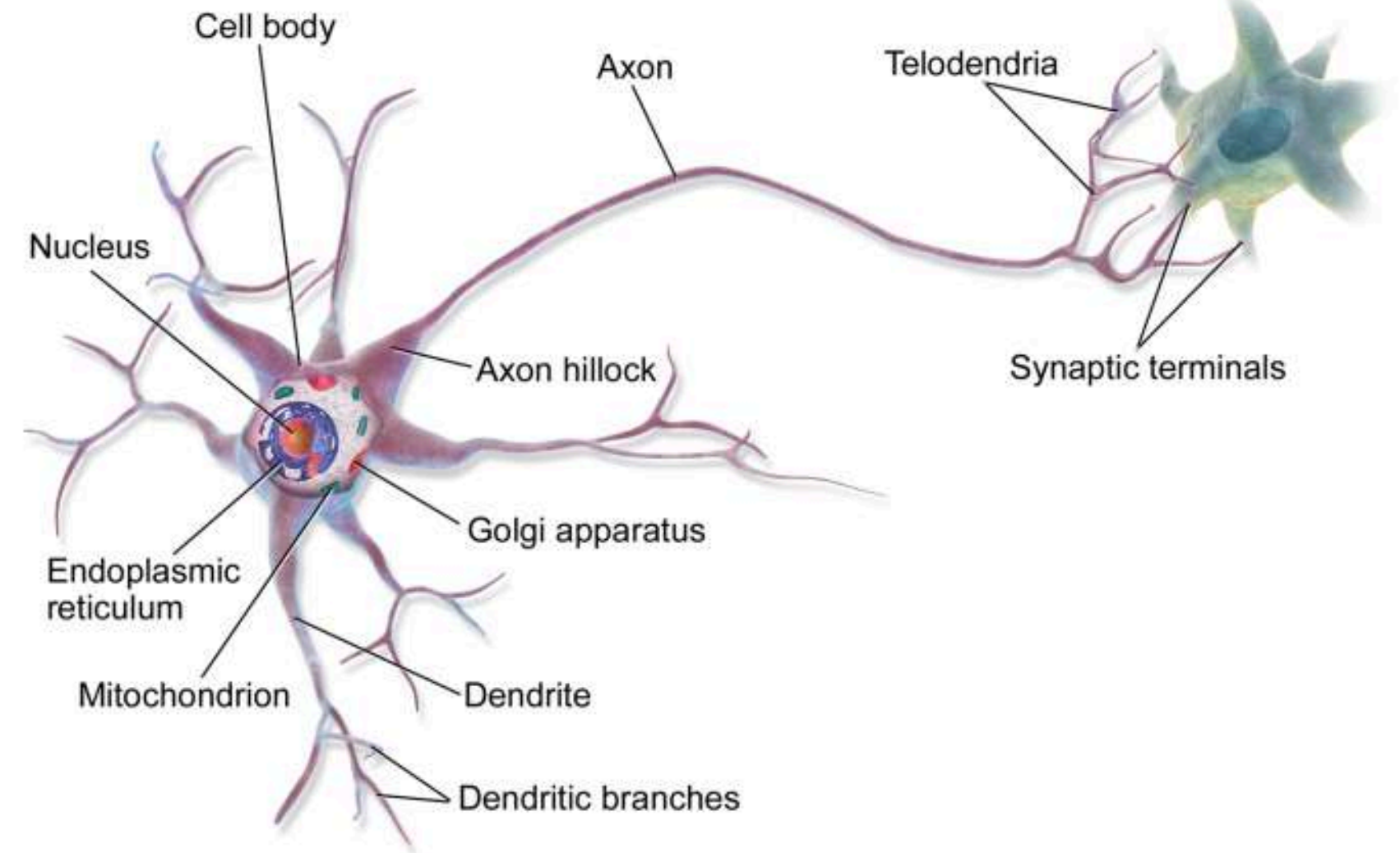
Deep Learning?

- Deep Learning (DL) is a specialized subset of machine learning that uses artificial neural networks (inspired by the human brain) to model and solve complex problems.



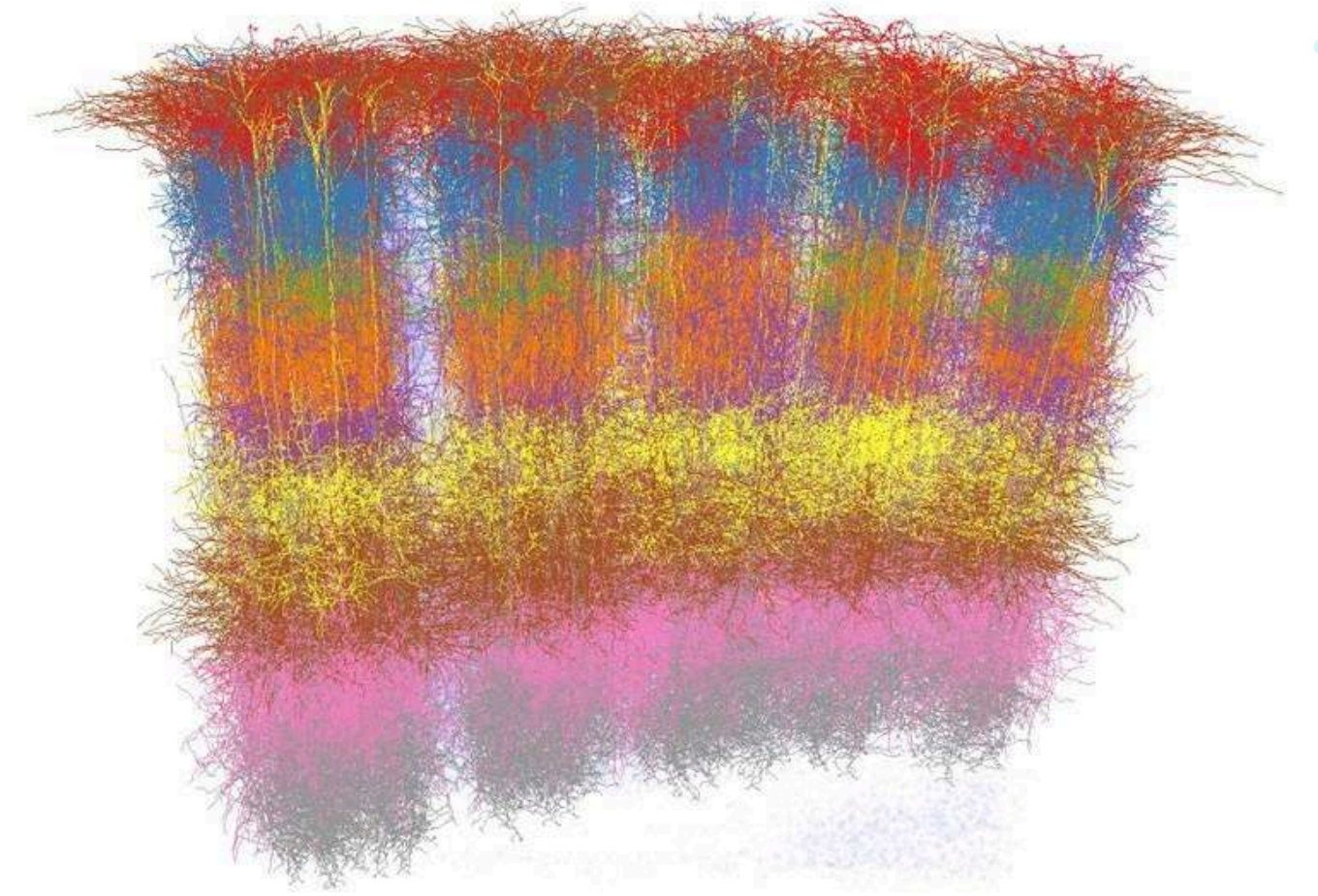
The Biological Inspiration

- Neurons in your cerebral cortex are connected via axons
- A neuron “fires” to the neurons it’s connected to, when enough of its input signals are activated
- Very simple at the individual neuron level – but layers of neurons connected in this way yields learning behavior.
- Billions of neurons, each with thousands of connections, yields a mind



The Cortical Columns

- Neurons in our cortex seem to be arranged into many stacks, or “columns” that process information in parallel
- “Mini-columns” of around 100 neurons are organized into larger “hyper-columns”. There are millions mini-columns in our cortex



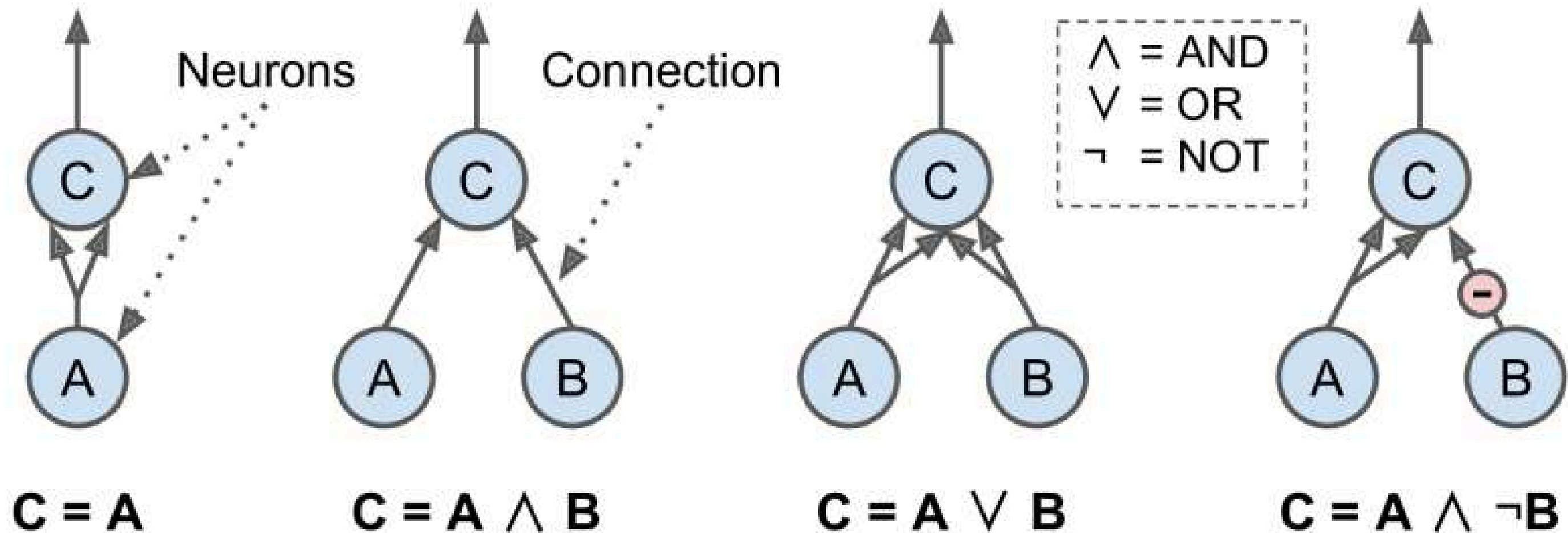
Some history

- Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts.
- In their landmark paper, McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic.
- **This was the first artificial neural network architecture.**

The first artificial neuron



- It has one or more binary (on/off) inputs and one binary output.
- By connecting these neurons together, we can construct logical operators.



The perceptron

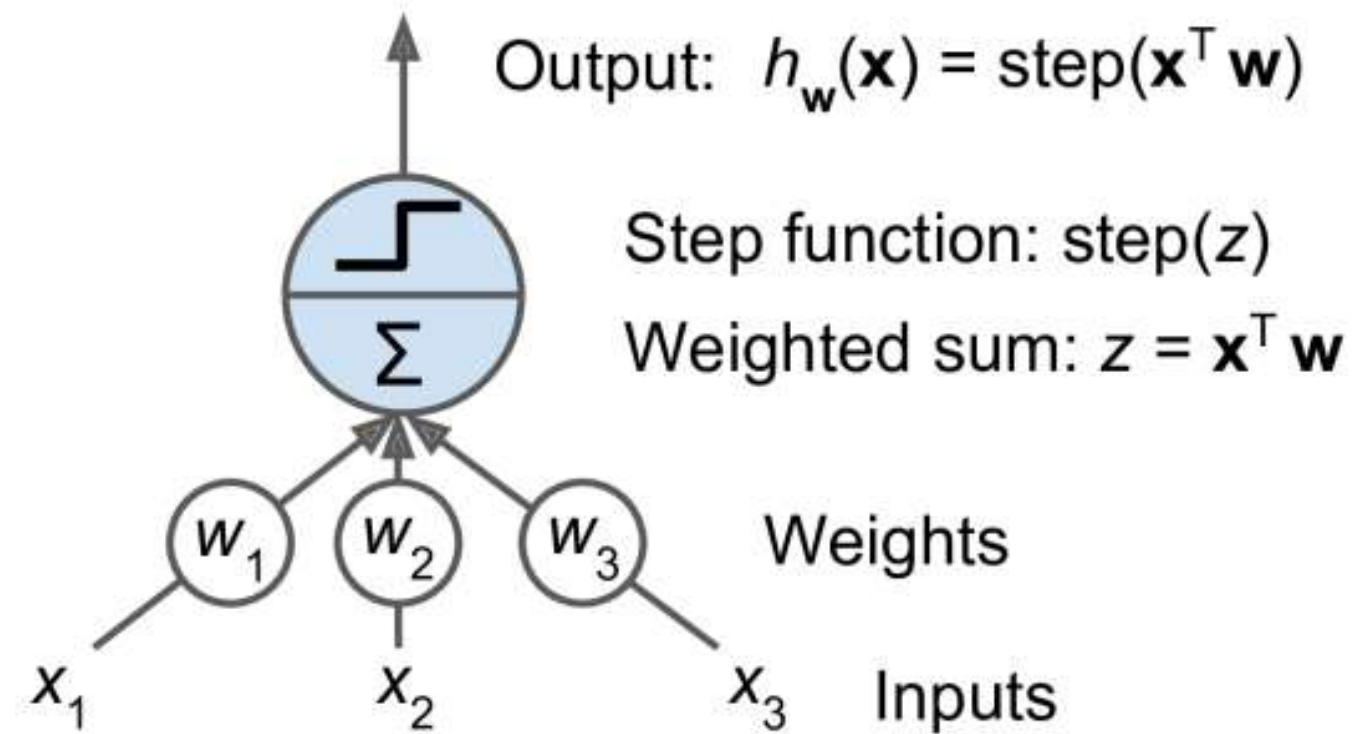


- The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a slightly different artificial neuron called threshold logic unit (TLU), or sometimes a linear threshold unit (LTU).
- The inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight.
- The TLU computes a weighted sum of its inputs then applies a step function to that sum and outputs the result.

A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs



The perceptron



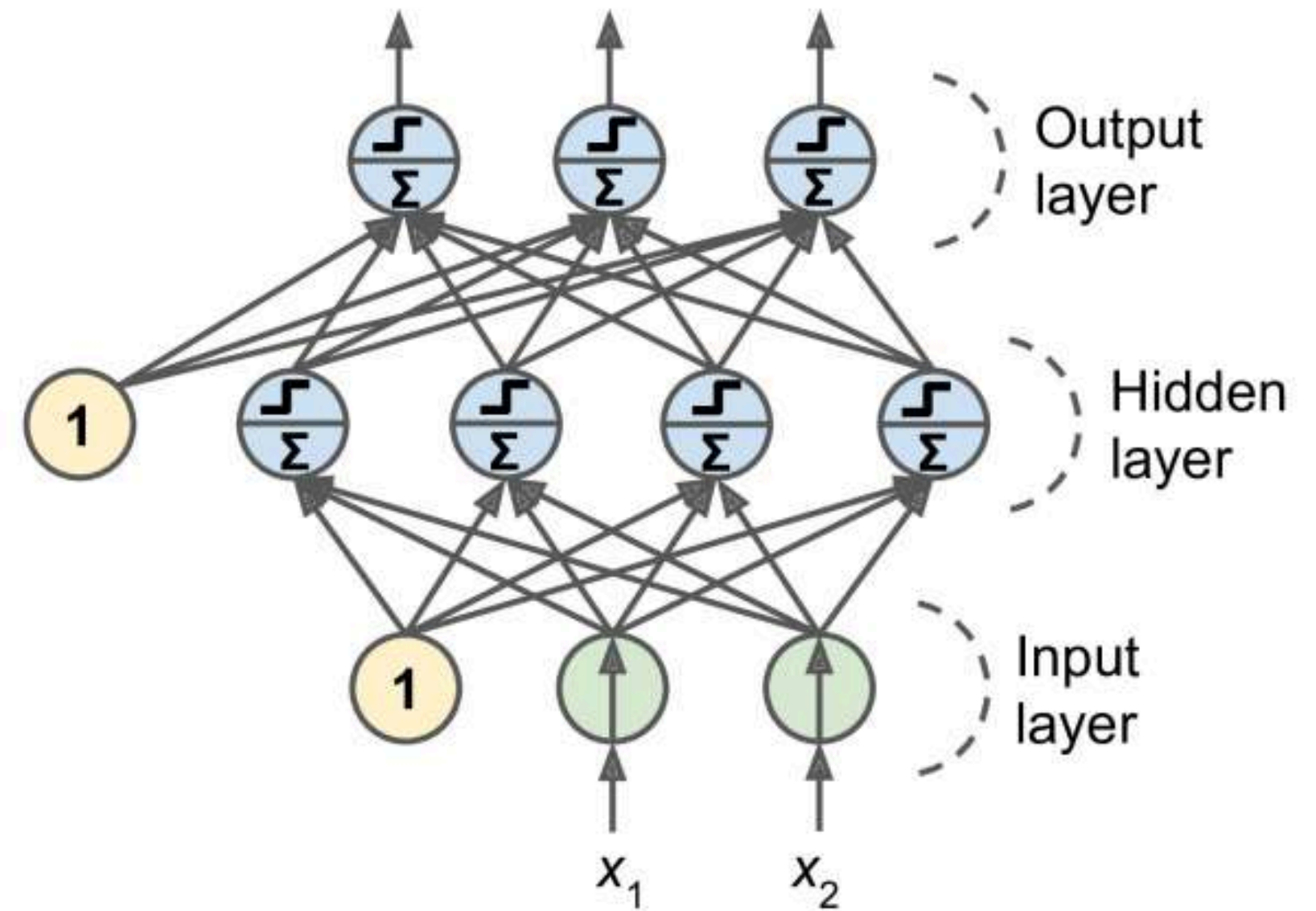
$$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{x}^T \mathbf{w}$$
$$h(\mathbf{x}) = \text{step}(z)$$

- The most common step function used in Perceptrons is the Heaviside step function.

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

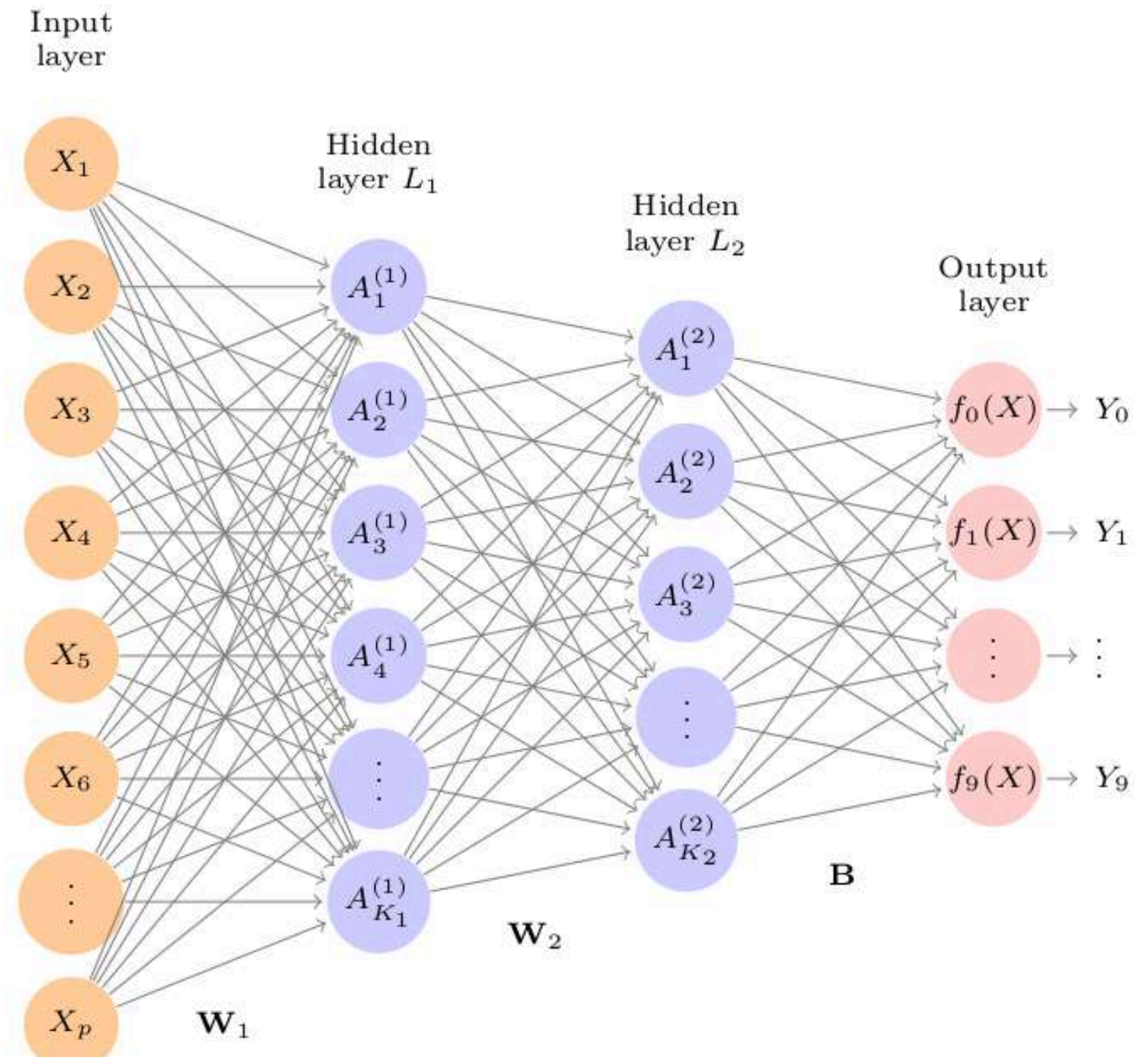
Multi-layer perceptrons

- An MLP is composed of one passthrough(input) layer, one or more layers of TLUs, called hidden layers, and one final layer of TLUs called the output layer.
- The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers.
- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN).
- The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations



Modern deep neural networks

- Complex neural network architecture with multiple hidden layers, often including advanced techniques like convolutional layers (for image data), recurrent layers (for sequential data), attention mechanisms.
- Replace step functions with something better
- Apply softmax to output



Backpropagation



- Backpropagation is how a neural network learns from its mistakes. It helps adjust the network's internal settings so that it makes better predictions over time.

How does it work?

- For each training instance:
 - Compute the output error
 - Compute how much each neuron in the previous hidden layer contributed
 - Back-propagate that error in a reverse pass
 - Tweak weights to reduce the error using gradient descent



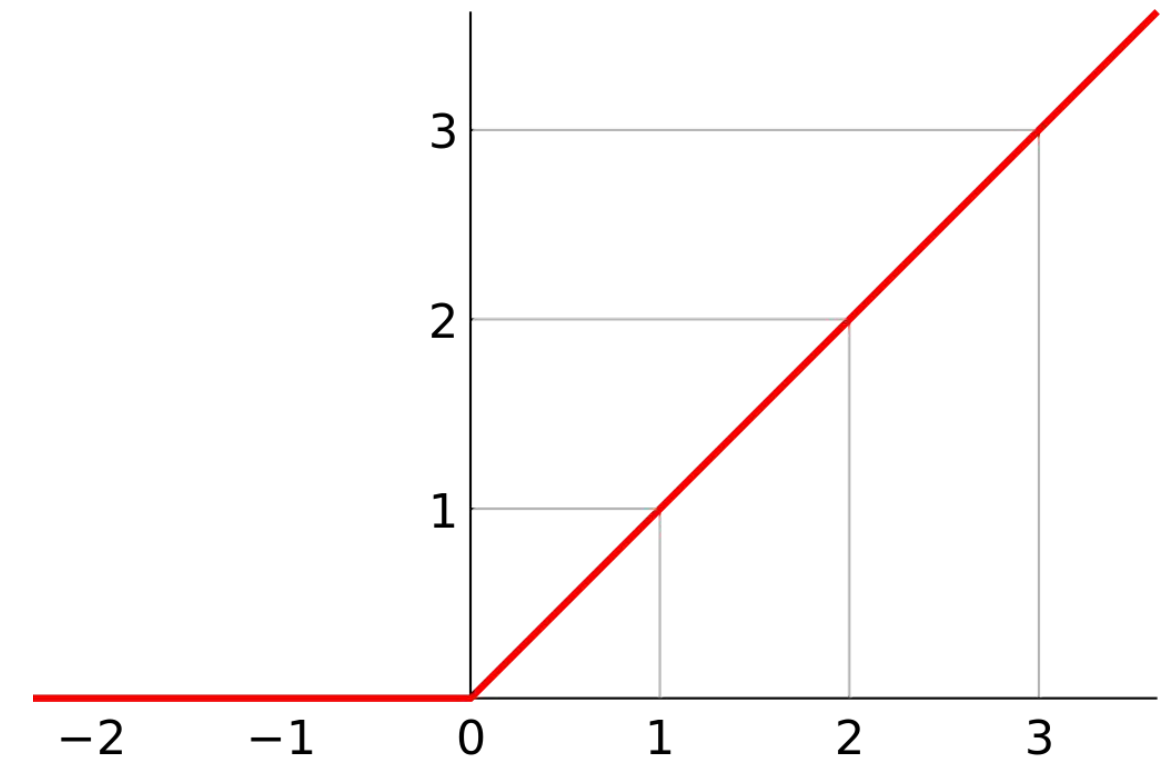
Activation functions (aka rectifiers)



- Step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface)

Alternatives:

- Logistic function
- Hyperbolic tangent function(tanh)
- Exponential linear unit(ELU)
- Rectified linear unit (ReLU)
- ReLU is common. Fast to compute and works well.
- ELU can sometimes lead to faster learning.(when resource is not our issue)



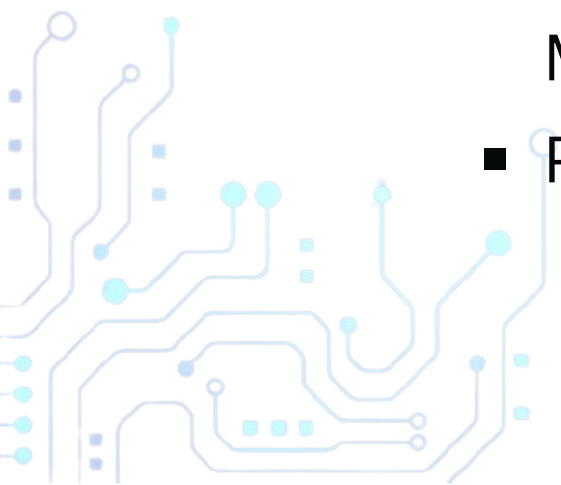
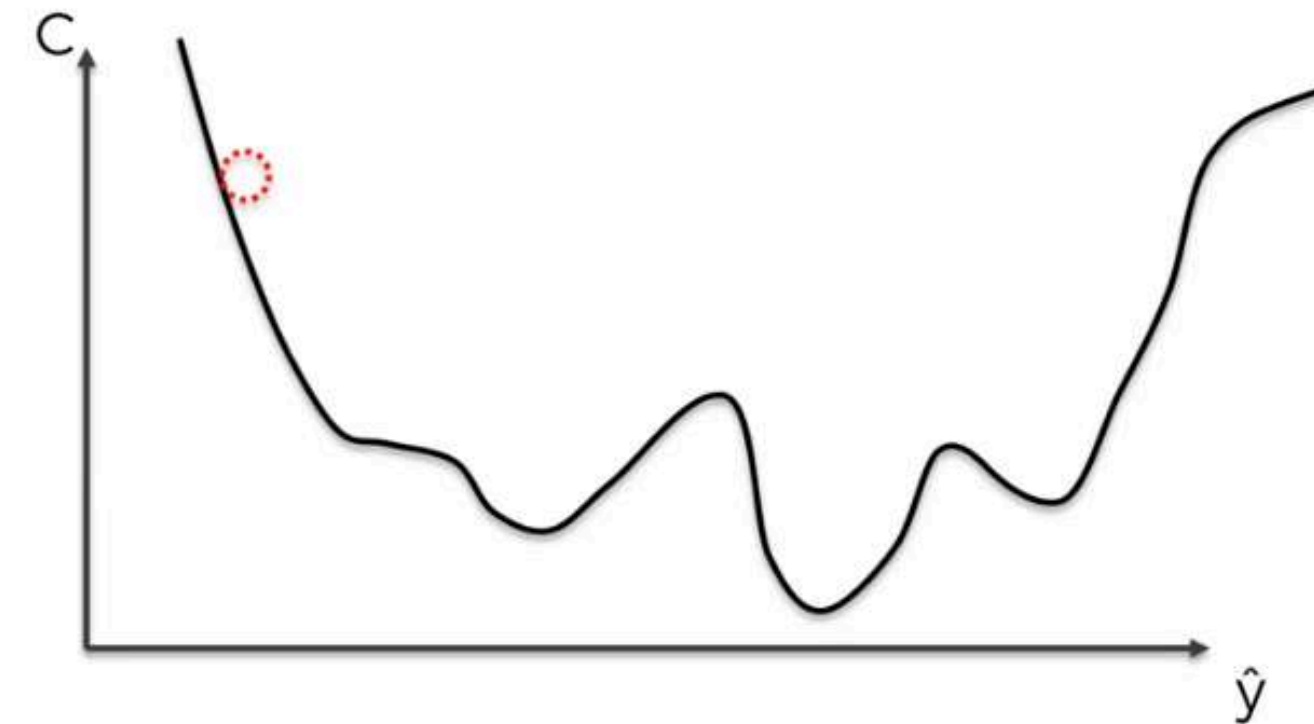
Graph of ReLU function



Optimization functions



- There are faster (as in faster learning) optimizers than gradient descent
 - **Momentum Optimization**
 - Introduces a momentum term to the descent, so it slows down as things start to flatten and speeds up as the slope is steep
 - **Nesterov Accelerated Gradient**
 - A small tweak on momentum optimization – computes momentum based on the gradient slightly ahead, not where it's now.
 - **RMSprop** (Root Mean Square Propagation)
 - Adaptive learning rate to help point toward the minimum
 - **Adam** (Adaptive moment estimation)
 - It combines the best features of two other optimizers: Momentum and RMSProp
 - Popular choice today, easy to use

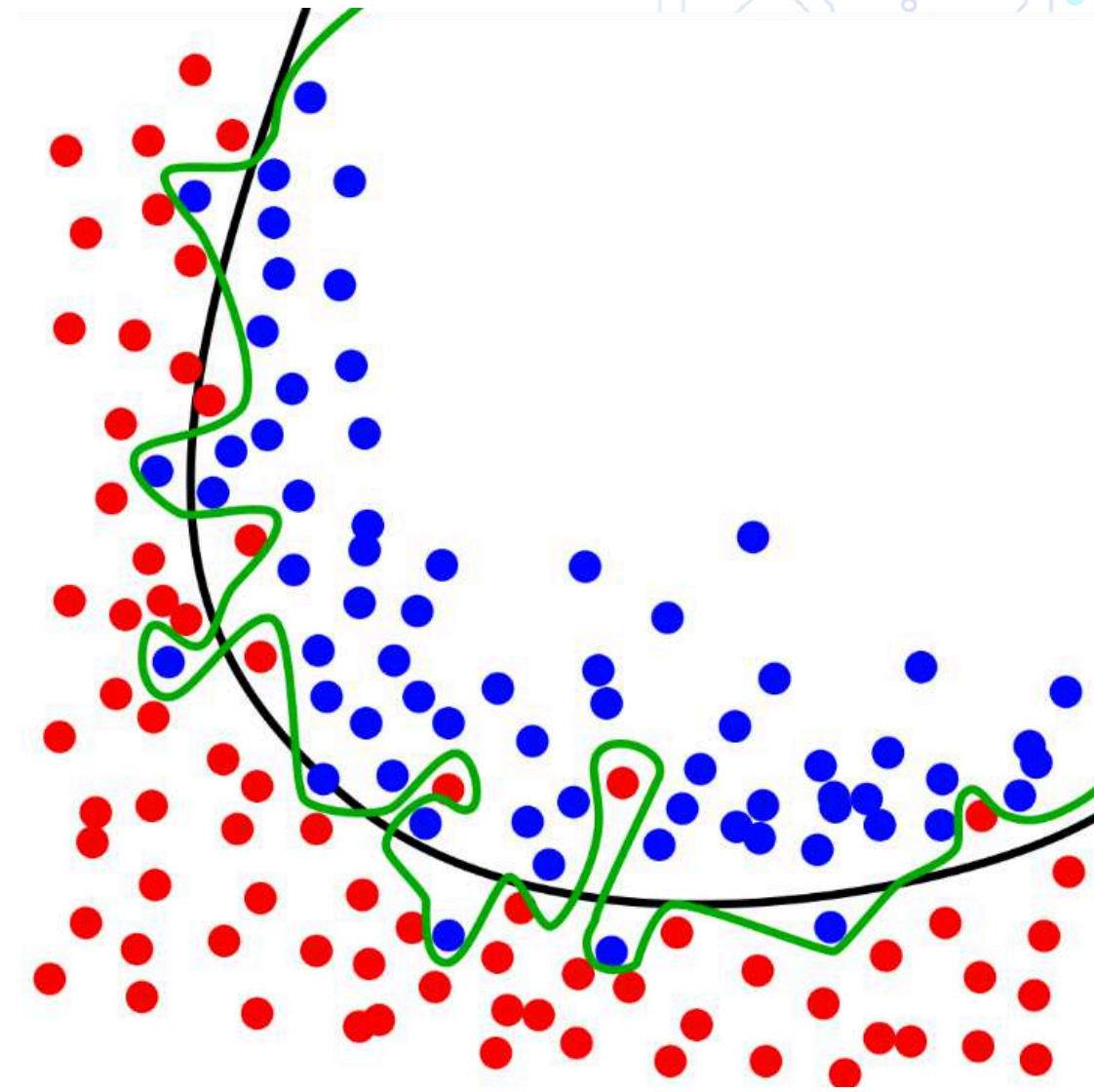


Avoiding Overfitting

- Overfitting happens when a machine learning model learns the training data too well, including noise and details that don't help in making good predictions on new data.
- It's like a student memorizing answers instead of understanding concepts.
- With thousands of weights to tune, overfitting is a problem

How to prevent it?

- Dropout (for Neural Networks) - randomly ignoring some neurons during training helps prevent the model from relying too much on specific details.
- Early stopping (when performance starts dropping)
- Regularization terms added to the cost function during training-force the model to focus on the important features instead of irrelevant detail.



Tuning your topology



- Trial & error is one way
 - Evaluate a smaller network with less neurons in the hidden layers
 - Evaluate a larger network with more layers - try reducing the size of each layer as you progress
- More layers can yield faster learning than more neurons in a single layer
- Or just use more layers and neurons than you need, and don't care because you can early stop.
- Use “model zoos”




Tensorflow



- TensorFlow is an open-source machine learning framework developed by Google.
- A tensor is just a fancy name for a multi-dimensional array (like a table of numbers).

why tensorflow?

- Easy to Use – Provides high-level APIs like **Keras**, which makes building models simple.
 - Optimized for Speed – Uses GPUs (Graphics Processing Units) to run models faster.
 - Scalable – Works on small devices (like phones) and large systems (like cloud servers).
 - Supports Many AI Applications – Used in self-driving cars, medical diagnosis, chatbots, and more.
- 

Using tensorflow

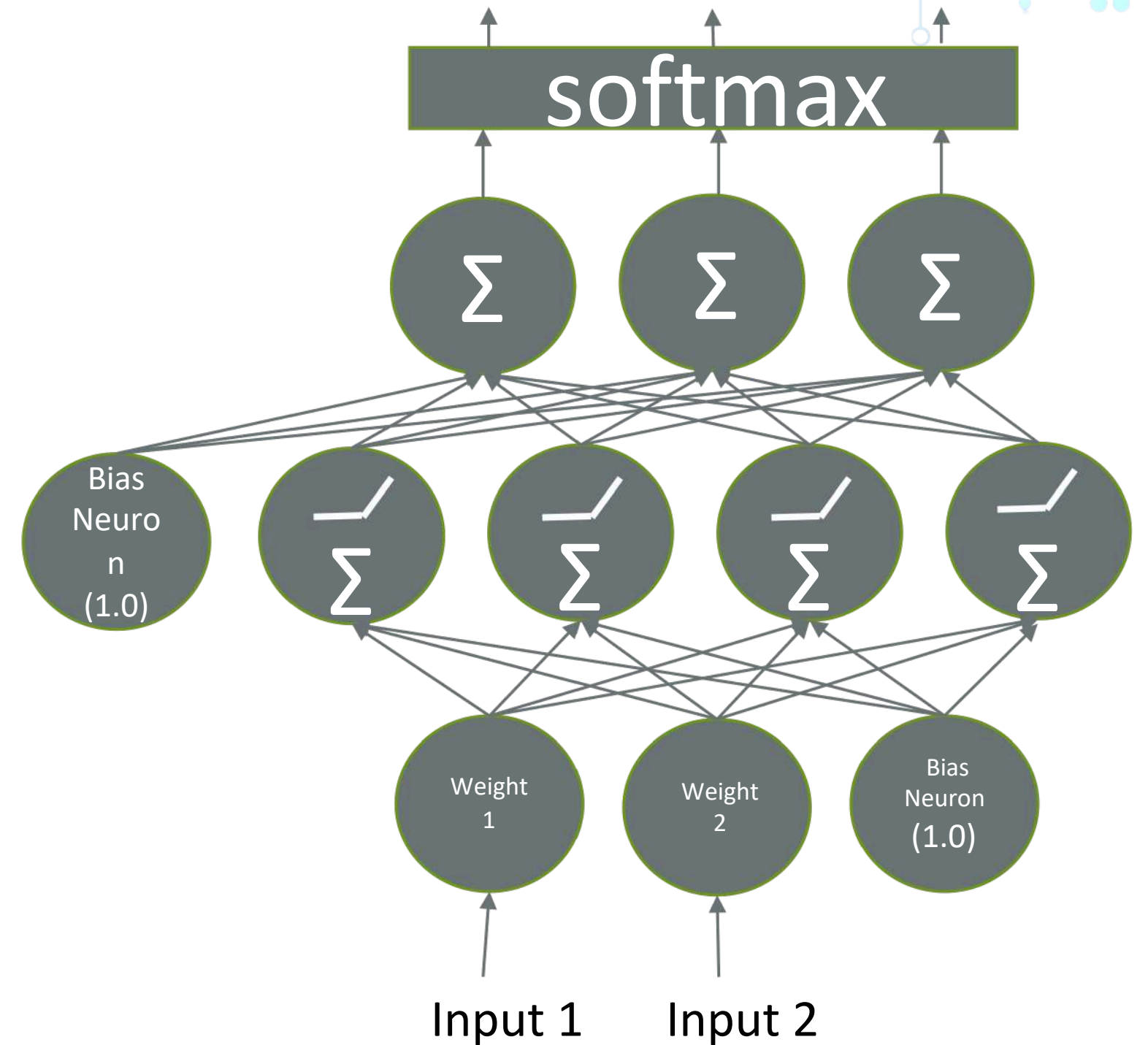
- Can install it with `conda install tensorflow` or `conda install tensorflow-gpu`
- Construct a graph to compute your tensors
- Initialize your variables
- Execute that graph

```
import tensorflow as tf
a = tf.Variable(1, name="a")
b = tf.Variable(1, name="b")
f = a + b

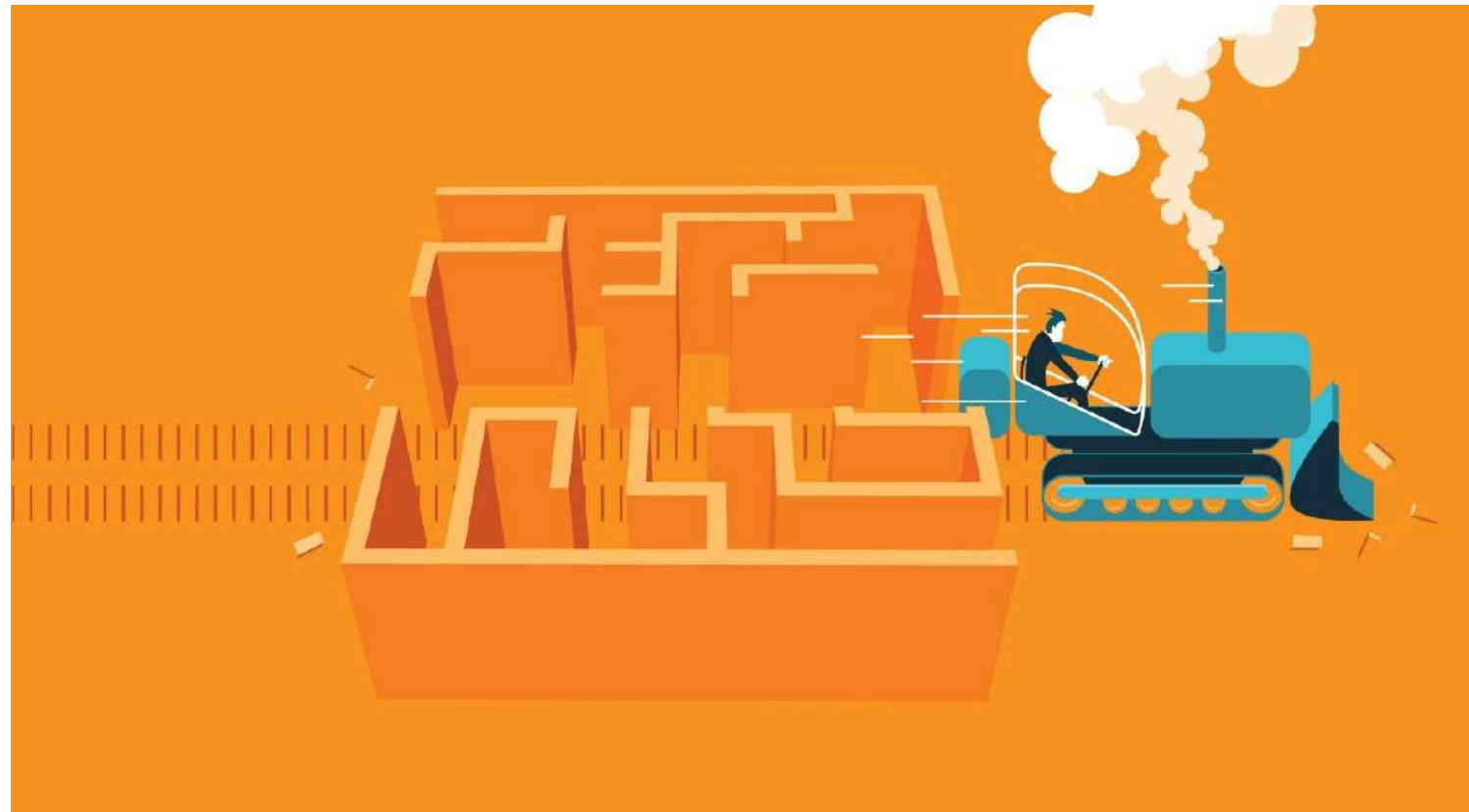
tf.print(f)
```

Creating a NN with tensorflow

- Load up our training and testing data
- Normalize your input data - the real goal is that every input feature is comparable in terms of magnitude.
- scikit_learn's **StandardScaler** can do this for you.
- Construct a graph describing our neural network
- Associate an optimizer (ie gradient descent) to the network
- Run the optimizer with your training data
- Evaluate your trained network with your testing data



Keras



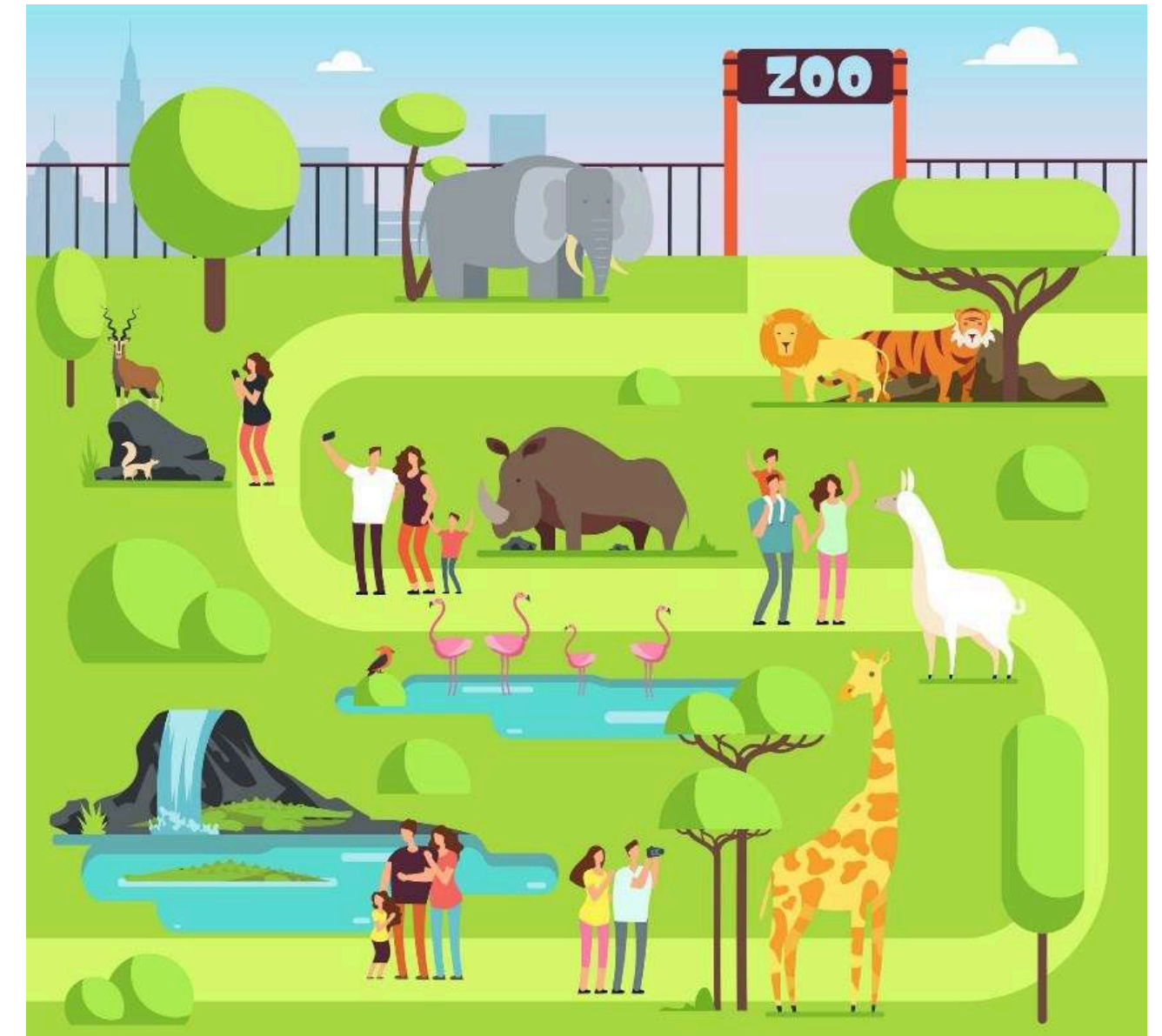
- Easy and fast prototyping
- Runs on top of TensorFlow
- scikit_learn integration
- Less to think about – which often yields better results without even trying
- This is really important! The faster you can experiment, the better your results.

Transfer Learning

- For many common problems, instead of training a model from scratch, we take an already-trained model and fine-tune it for our needs. This saves time, requires less data, and improves performance.
- Image classification (ResNet, Inception, MobileNet, Oxford VGG)
- NLP (word2vec, GloVe)
- Use them as-is, or tune them for your application

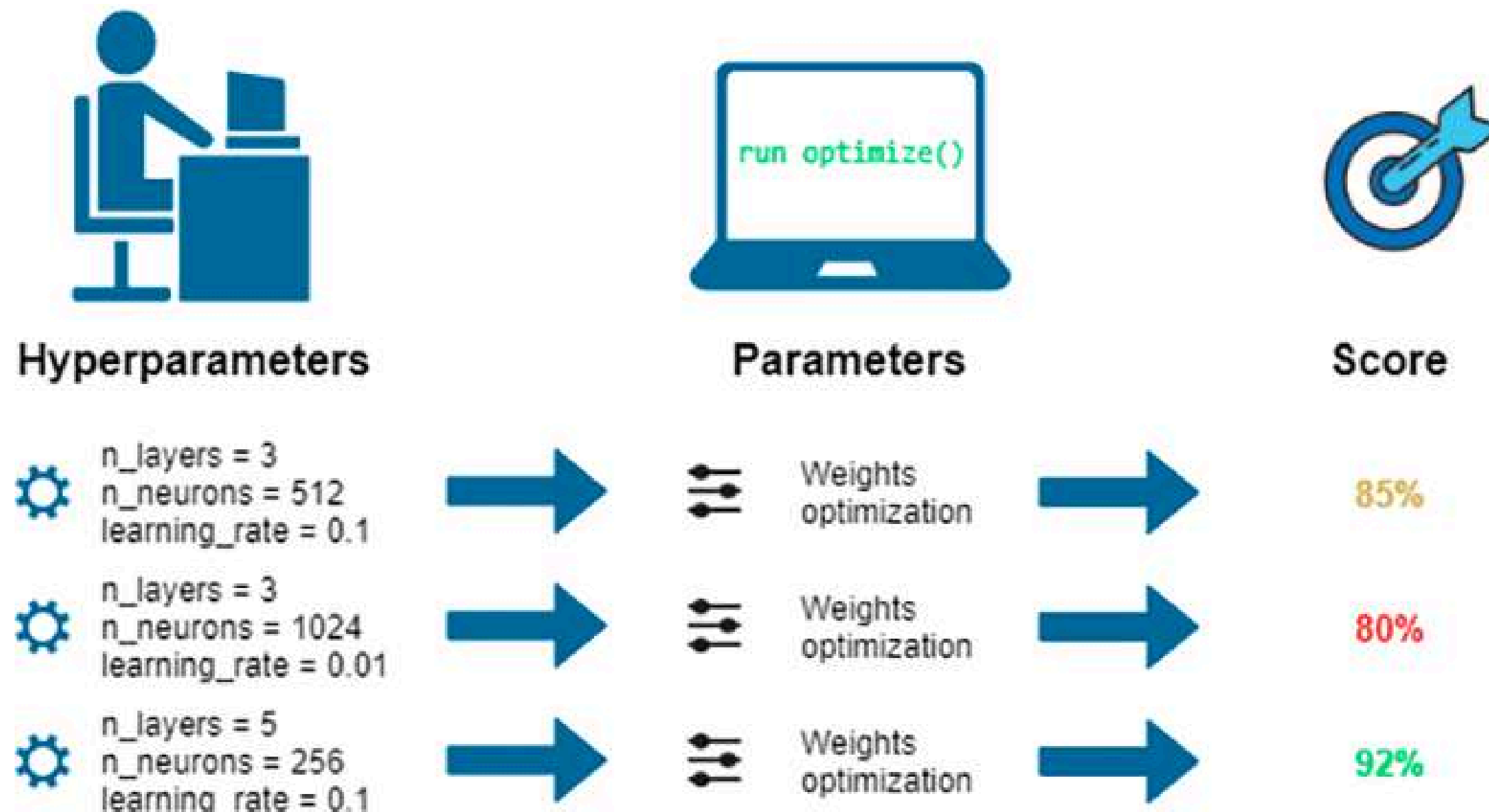
Where to find pre-trained models?

- `Model Zoos` - most popular place



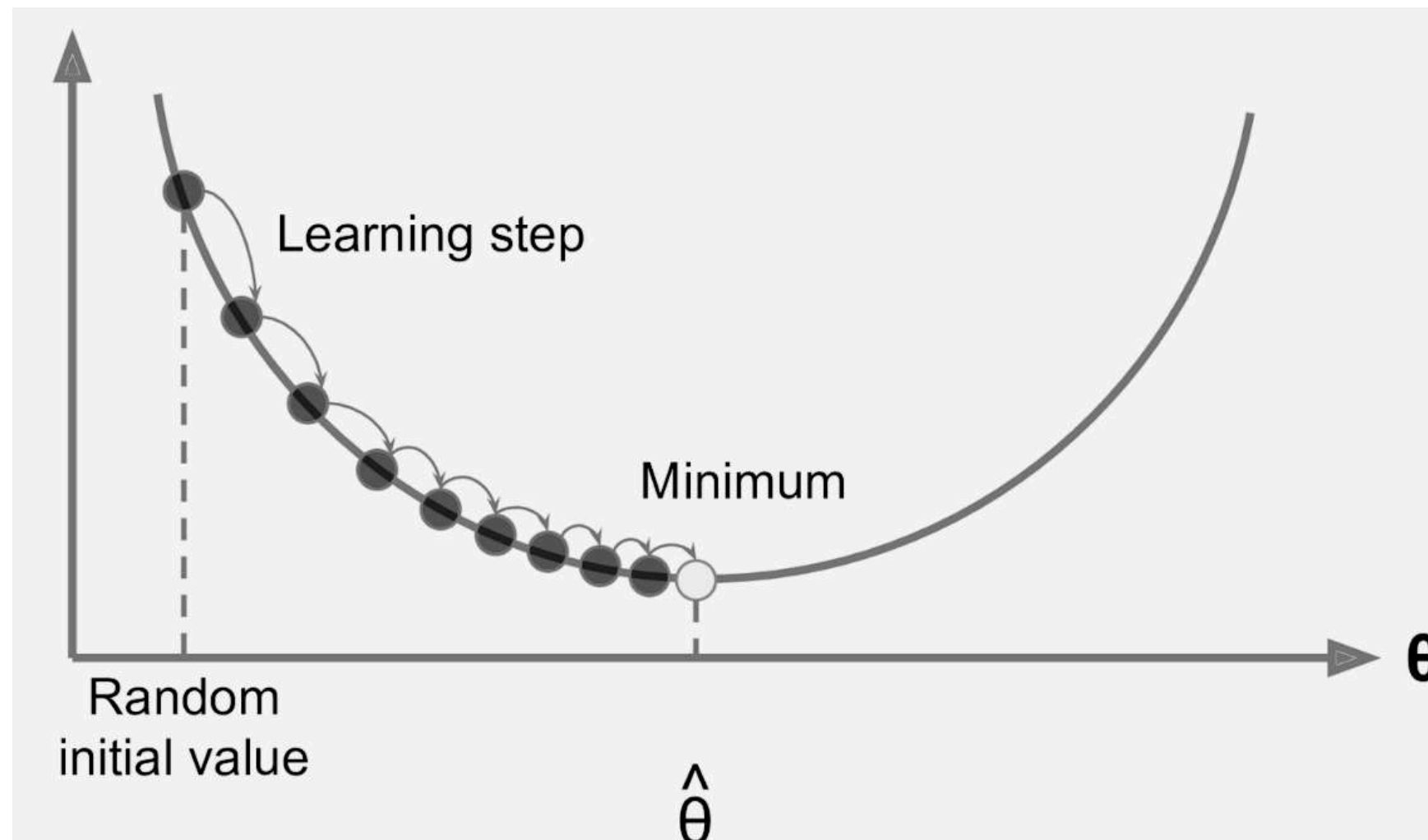
Tuning Neural Networks

- Adjusting it to improve its accuracy and performance.
- This is done by tweaking different settings (called **hyperparameters**) or modifying the model's layers during training.



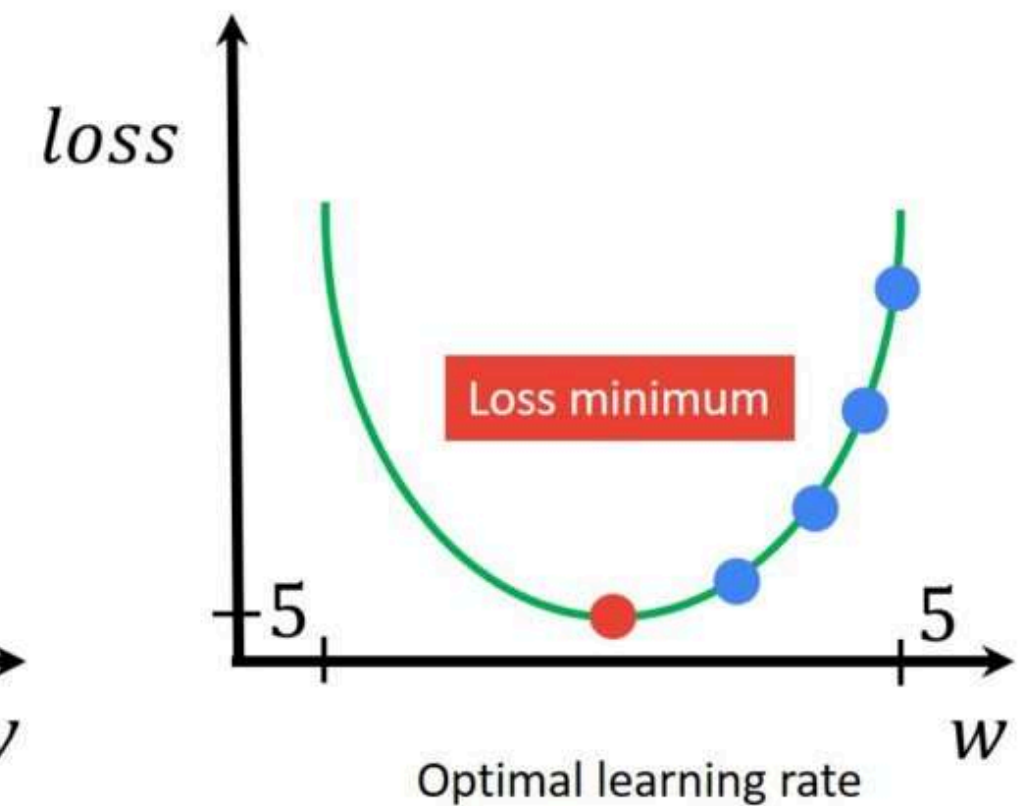
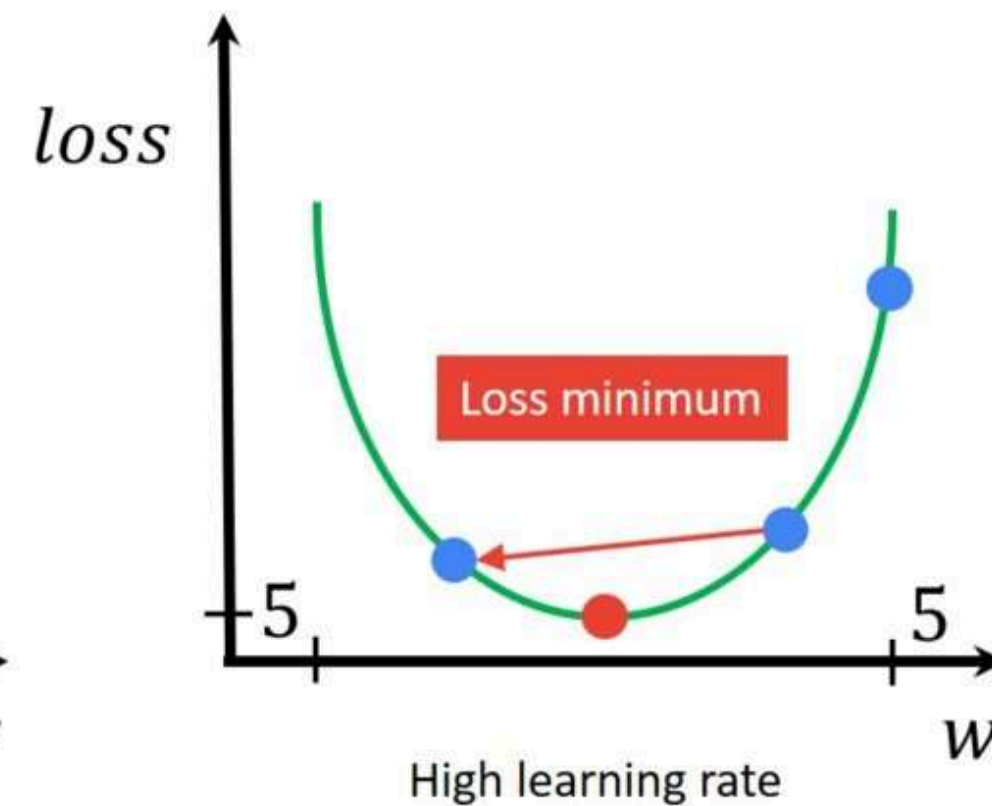
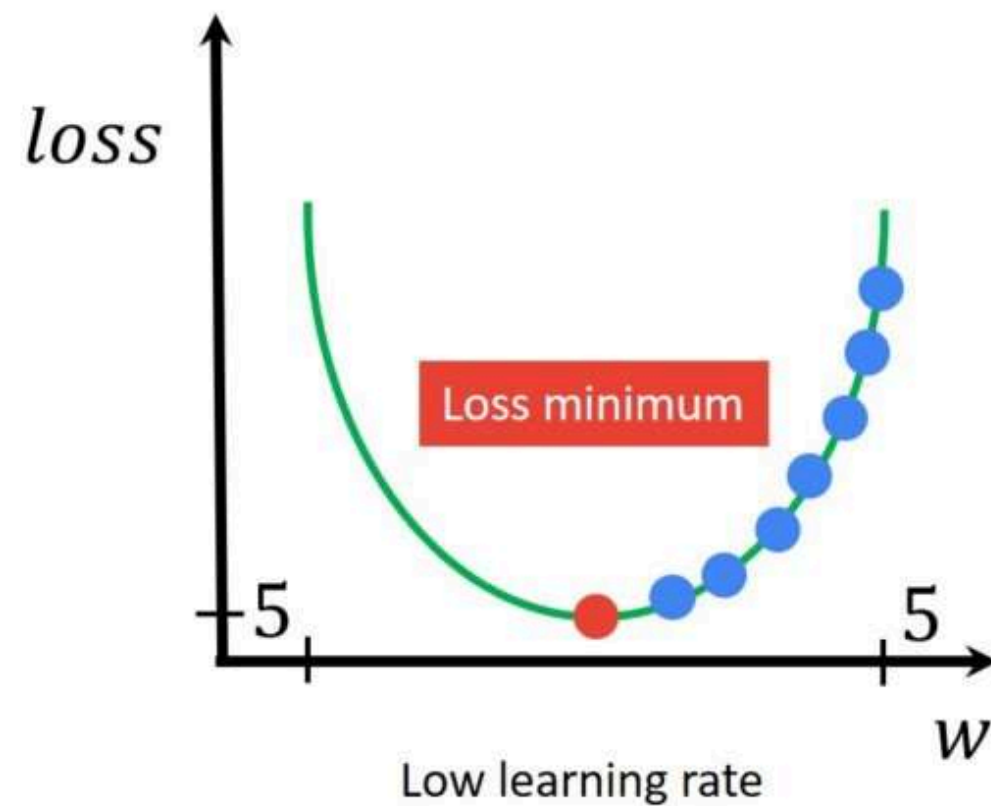
Learning Rate

- Neural networks are trained by gradient descent (or similar means)
- We start at some random point, and sample different solutions (weights) seeking to minimize some cost function, over many epochs
- How far apart these samples are is the learning rate



Effect of learning rate

- Too high a learning rate means you might overshoot the optimal solution!
- Too small a learning rate will take too long to find the optimal solution



Batch Size

- How many training samples are used within each epoch
- Smaller batch sizes can work their way out of “local minima” more easily
- Batch sizes that are too large can end up getting stuck in the wrong solution

