INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

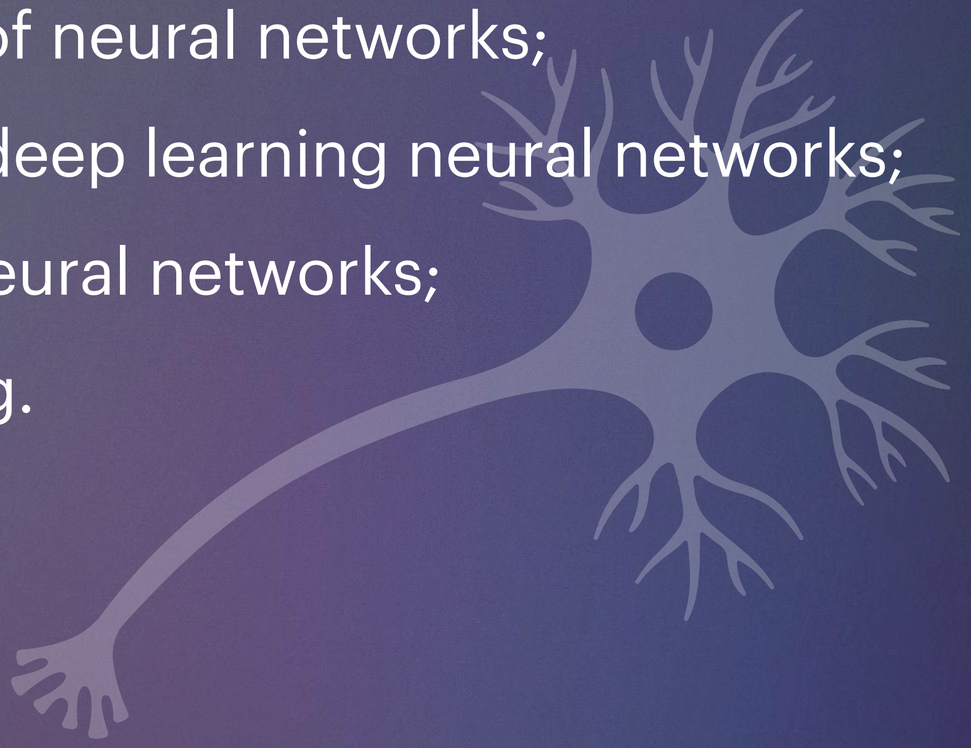
Deep Learning Overview

Prof. Me. Saulo A. F. Oliveira
saulo.oliveira@ifce.edu.br

Objectives

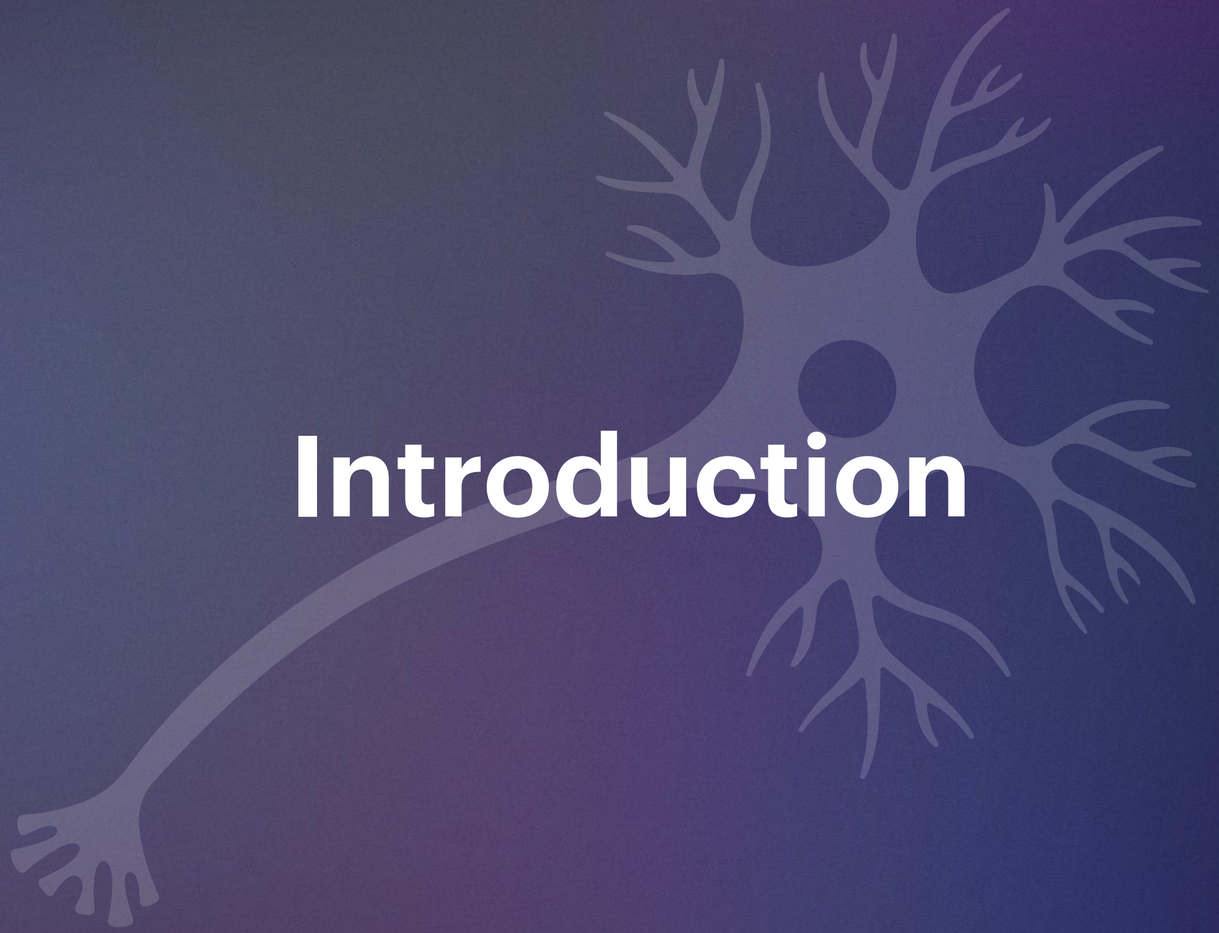
On completion of this course, you will be able to:

- Describe the definition and development of neural networks;
- Learn about the essential components of deep learning neural networks;
- Understand training and optimization of neural networks;
- Describe typical problems in deep learning.



01

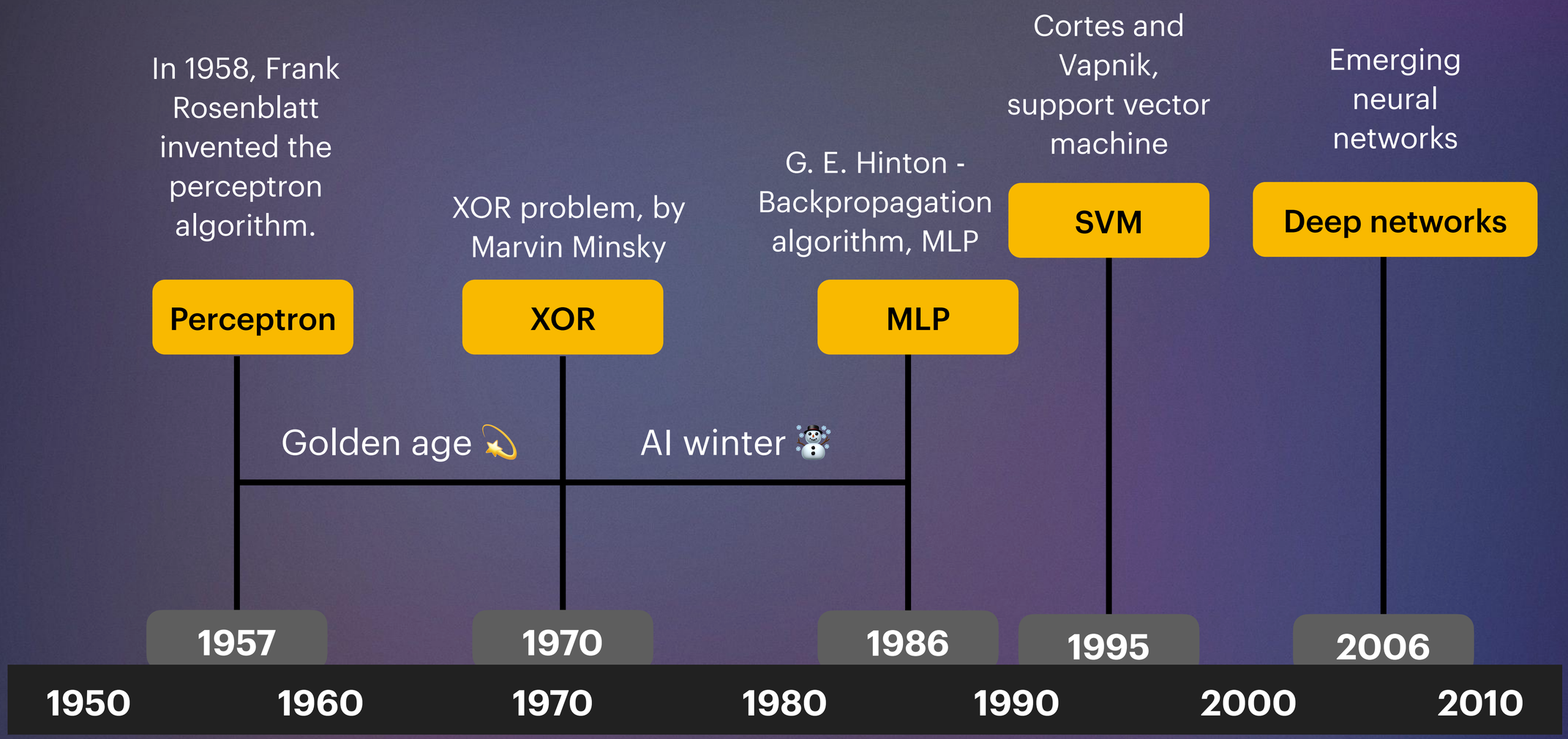
Introduction



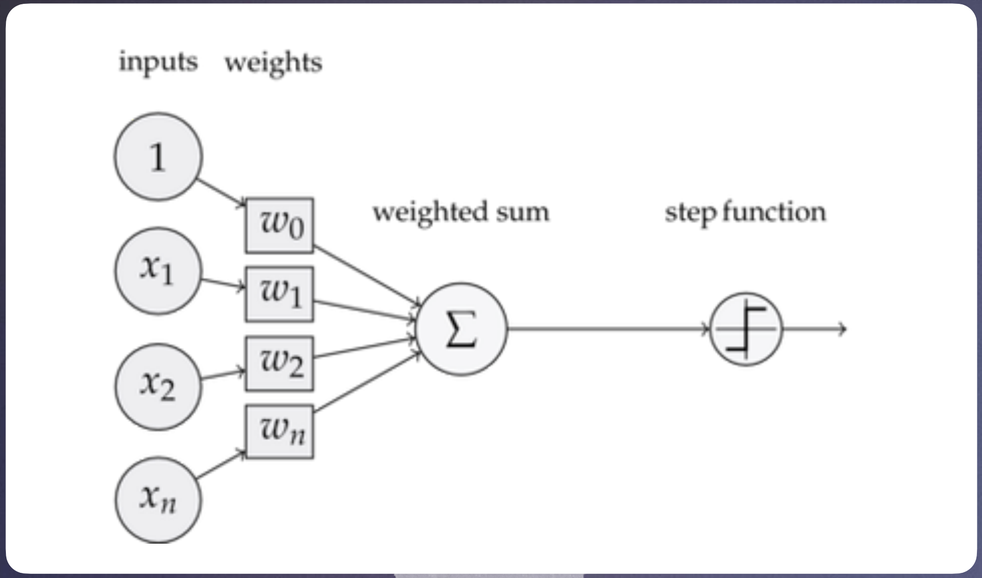
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

Development History of Neural Networks



Single Layer Perceptron



Graphical representation of a Single-Layer Perceptron

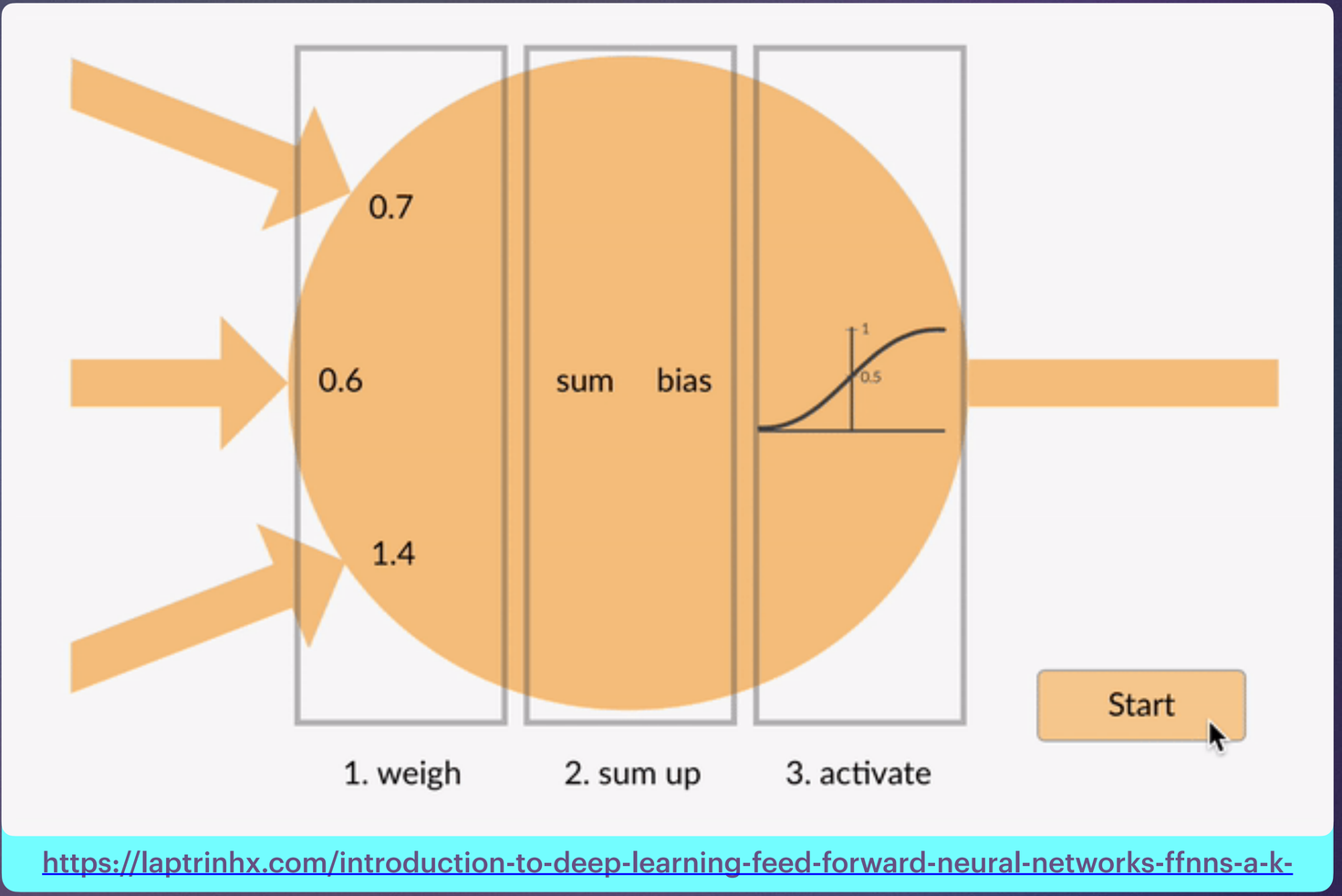
- Input vector:
  $\mathbf{x} = [x_1, x_2, \ldots, n_n]^\intercal$;
- Weights:
  $\mathbf{w} = [w_0, w_2, \ldots, w_n]^\intercal$, in which w_0 is the offset (bias);
- Activation function:

$$\hat{y} = \text{sign}(\mathbf{w}^\intercal \mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}^\intercal \mathbf{x} > 0, \\ -1 & \text{otherwise.} \end{cases}$$

The preceding perceptron is equivalent to a classifier. It uses the high-dimensional $\mathbf{x}$ vector as the input and performs binary classification on input samples in the high-dimensional space. When $\mathbf{w}^\intercal \mathbf{x} > 0$, $\hat{y} = +1$. In this case, the samples are classified into a type. Otherwise, $\hat{y} = -1$. In this case, the samples are classified into the other type. The boundary of these two types is $\mathbf{w}^\intercal \mathbf{x} = 0$, which is a high-dimensional hyperplane.

Single Layer Perceptron

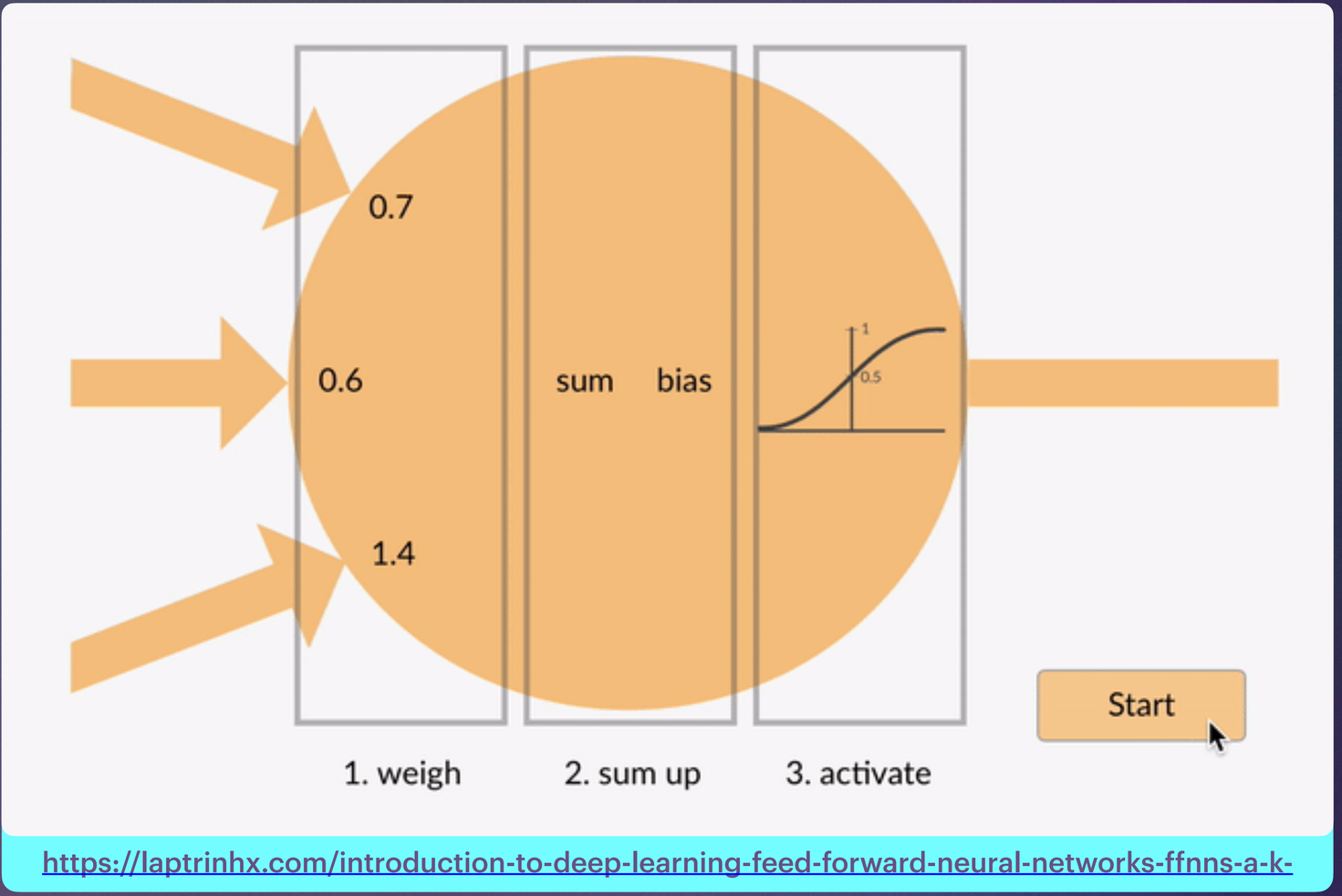
- Each neuron takes in some floating point numbers (e.g. 1.0, 0.5, -1.0) and multiplies them by some other floating point numbers (e.g. 0.7, 0.6, 1.4) known as weights (1.0 * 0.7 = 0.7, 0.5 * 0.6 = 0.3, -1.0 * 1.4 = -1.4). The weights act as a mechanism to focus on, or ignore, certain inputs. The weighted inputs then get summed together (e.g. 0.7 + 0.3 + -1.4 = -0.4) along with a bias value (e.g. -0.4 + -0.1 = -0.5).
- The summed value (x) is now transformed into an output value (y) according to the neuron's activation function (y = f(x)).



https://laptrinhx.com/introduction-to-deep-learning-feed-forward-neural-networks-ffnns-a-k-

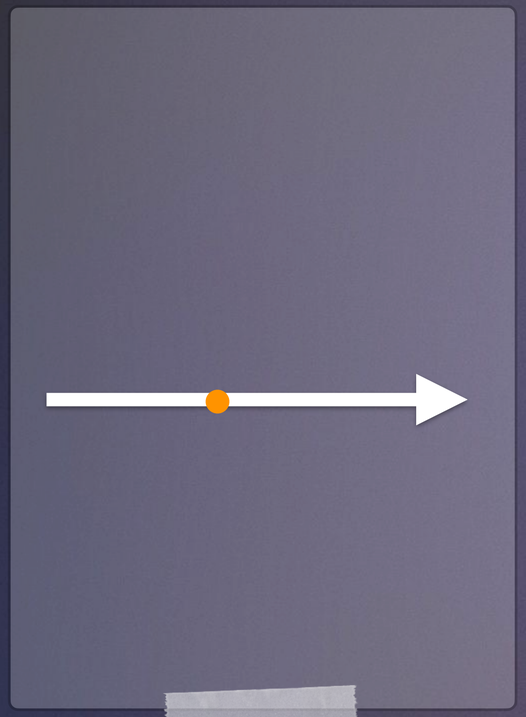
Single Layer Perceptron

- Each neuron takes in some floating point numbers (e.g. 1.0, 0.5, -1.0) and multiplies them by some other floating point numbers (e.g. 0.7, 0.6, 1.4) known as weights (1.0 * 0.7 = 0.7, 0.5 * 0.6 = 0.3, -1.0 * 1.4 = -1.4). The weights act as a mechanism to focus on, or ignore, certain inputs. The weighted inputs then get summed together (e.g. 0.7 + 0.3 + -1.4 = -0.4) along with a bias value (e.g. -0.4 + -0.1 = -0.5).
- The summed value (x) is now transformed into an output value (y) according to the neuron's activation function (y = f(x)).



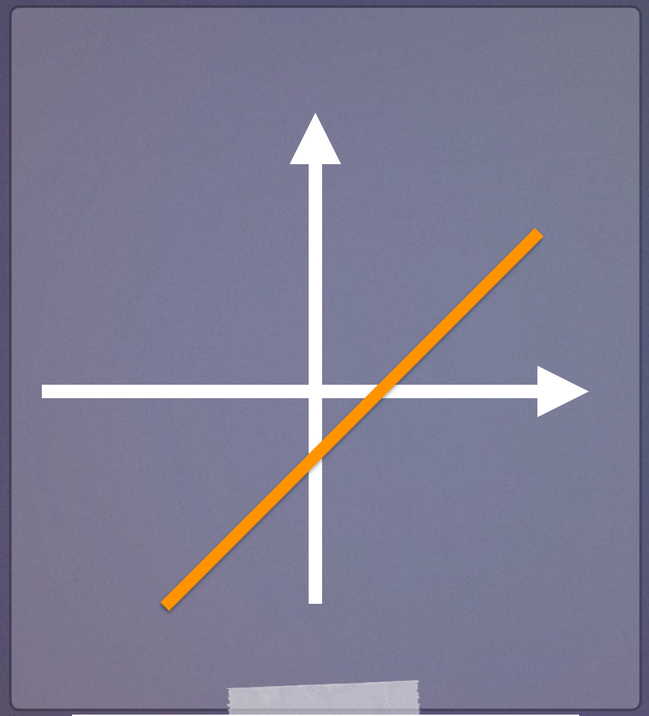
https://laptrinhx.com/introduction-to-deep-learning-feed-forward-neural-networks-ffnns-a-k-

Classification (Hyper)planes



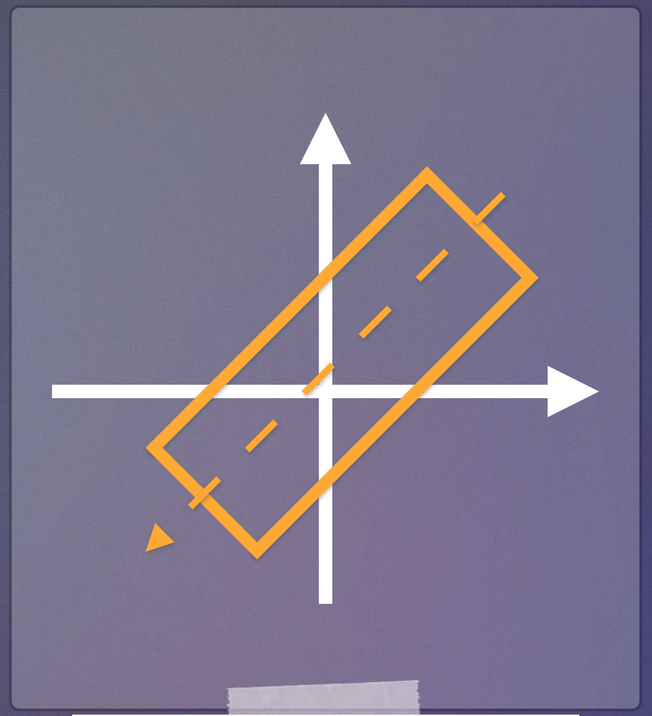
Classification point

$Ax + B = 0$



Classification line

$Ax + By + C = 0$



Classification plane

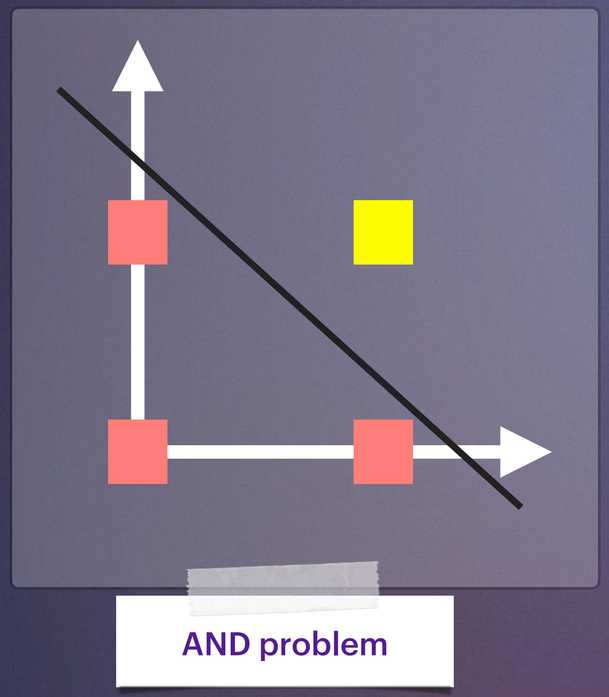
$Ax + By + Cz + D = 0$

Classification hyperplane

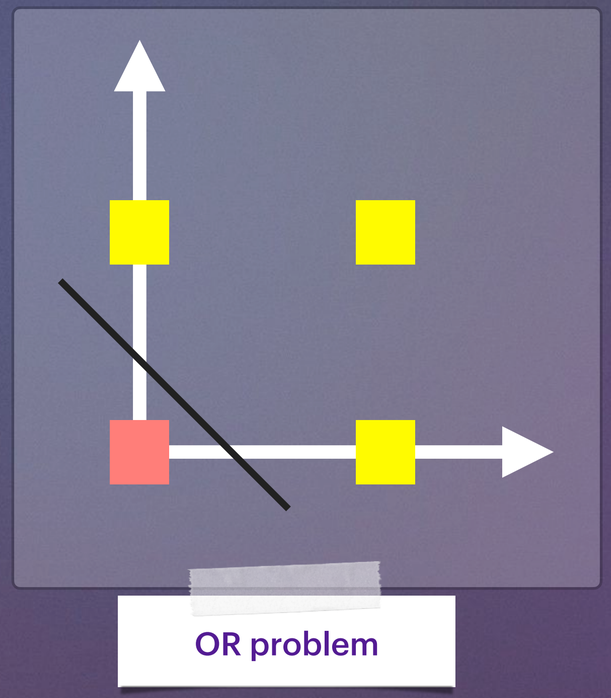
$\mathbf{W}^{\mathrm{T}}\mathbf{X} + b = 0$

XOR Problem

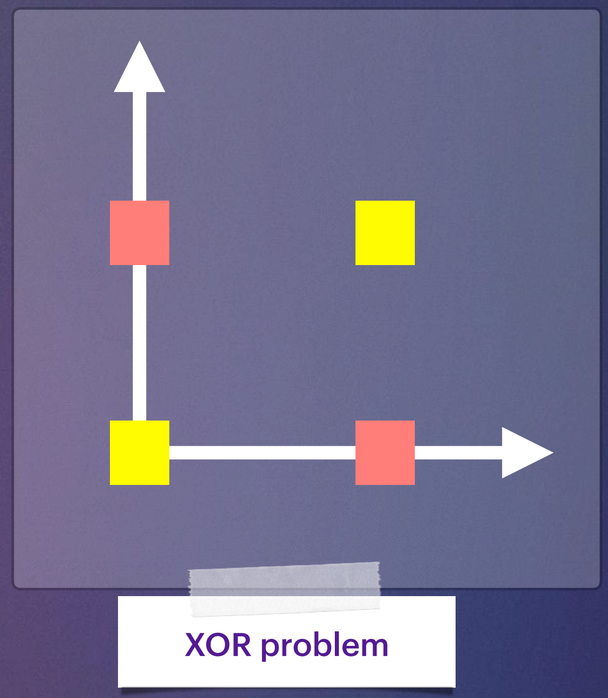
In 1969, Minsky, an American mathematician and AI pioneer, proved that a perceptron is essentially a linear model that can only deal with linear classification problems, but cannot process non-linear data.



AND problem

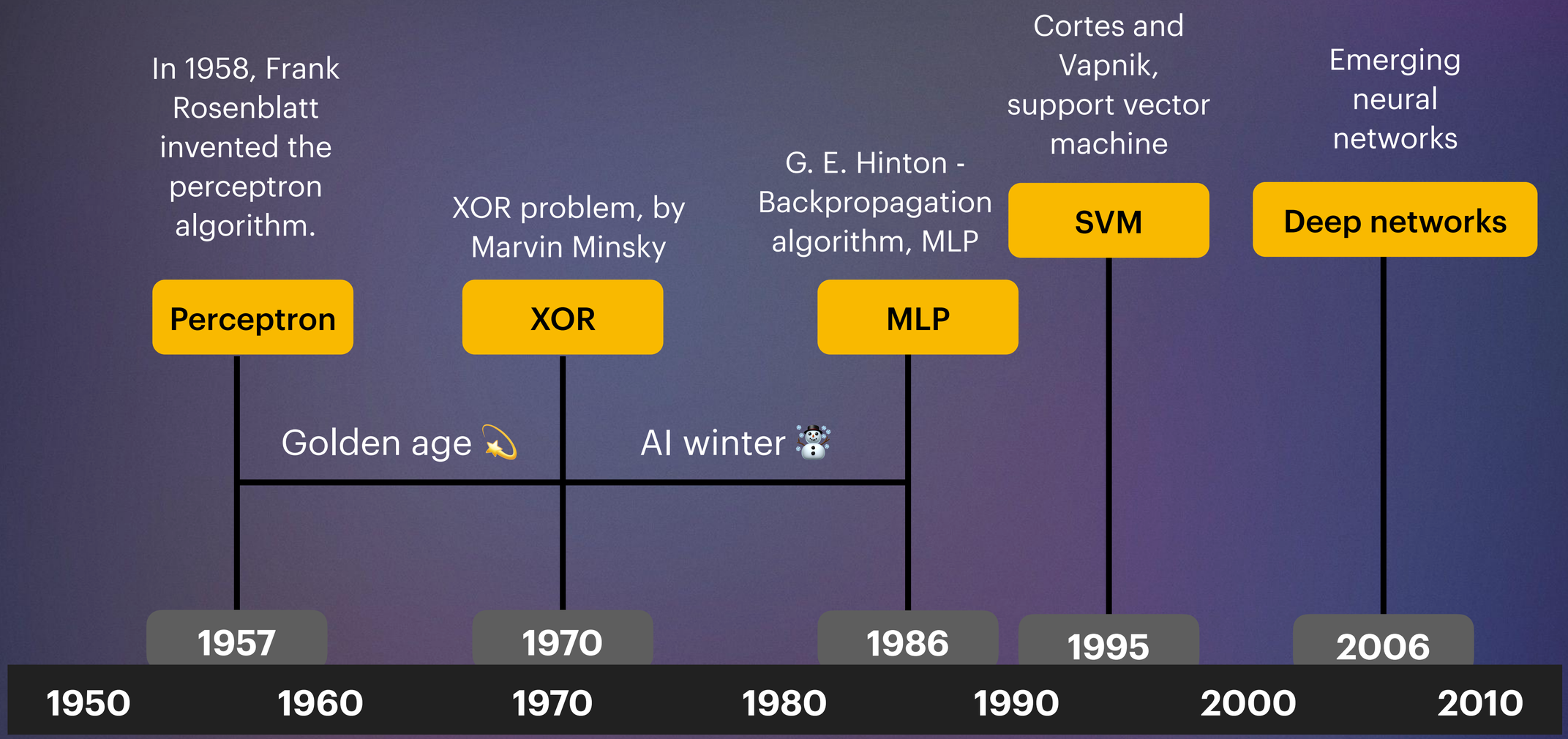


OR problem

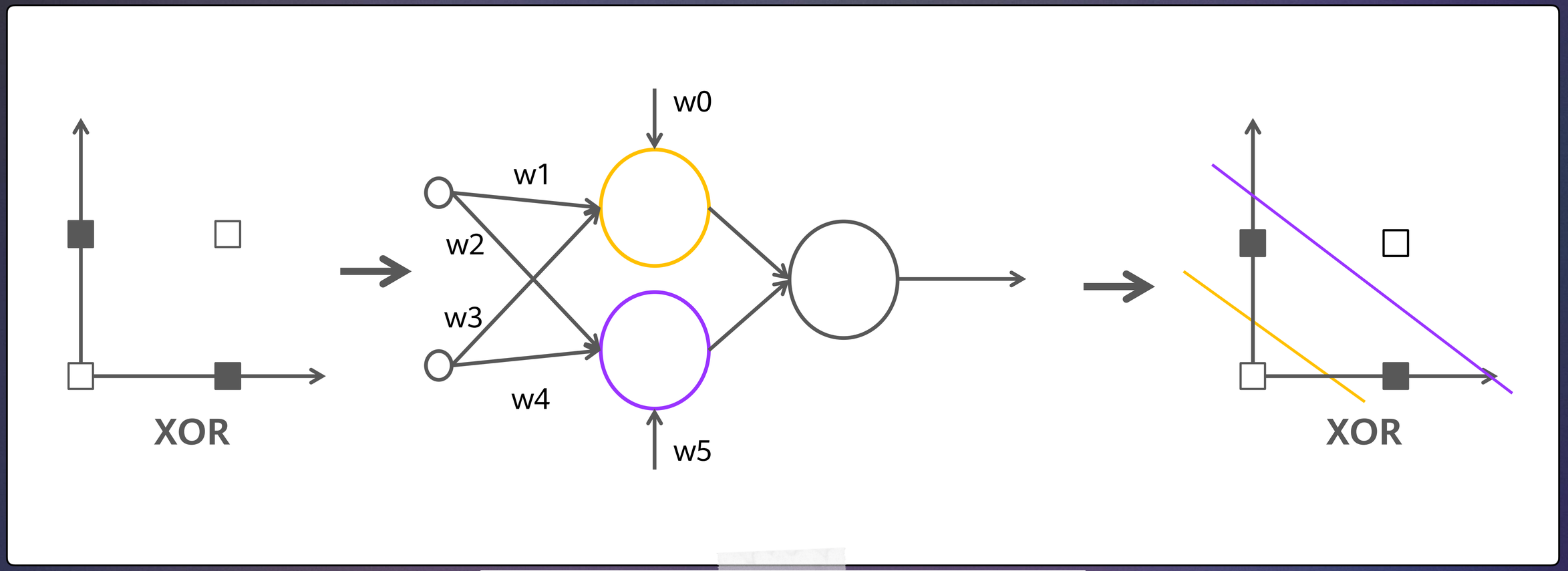


XOR problem

Development History of Neural Networks

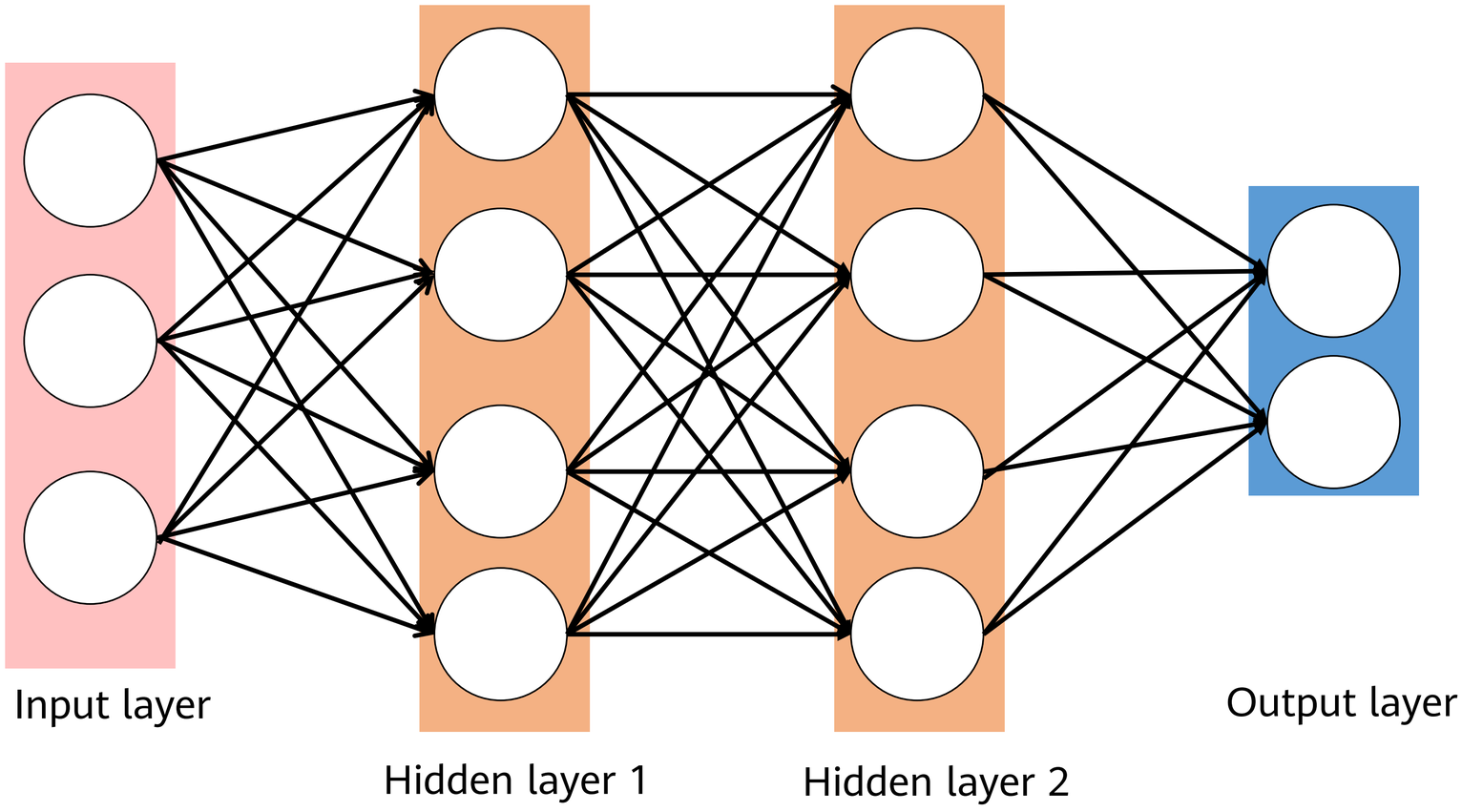


The magical hidden layer!



Now, we have peace!

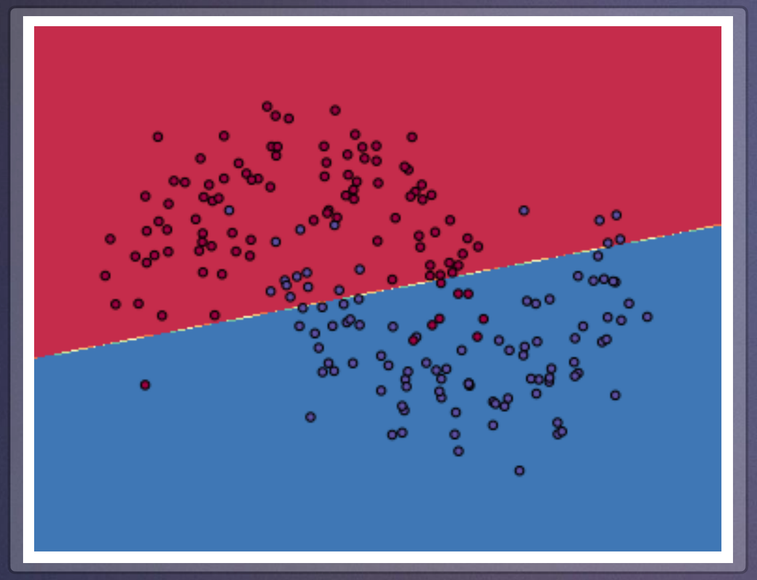
Hidden layers!



Typical feedforward architecture

Hidden layers: The more, the merrier?

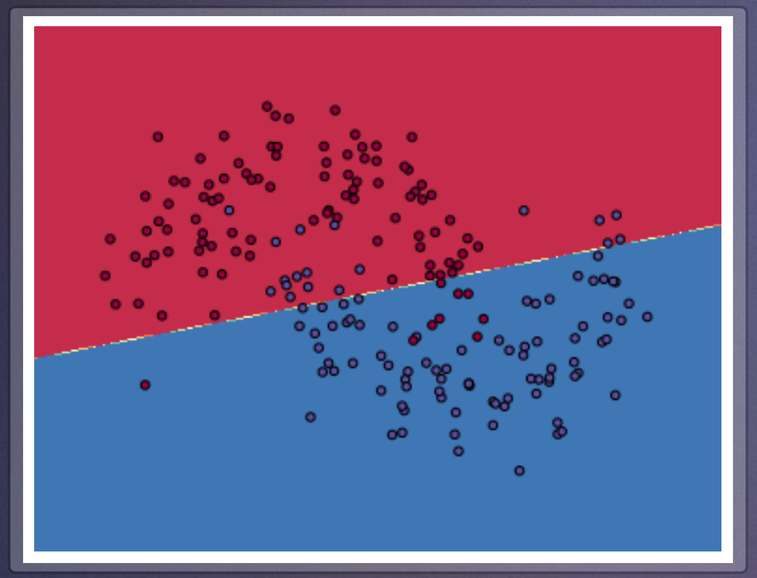
One (1) layer



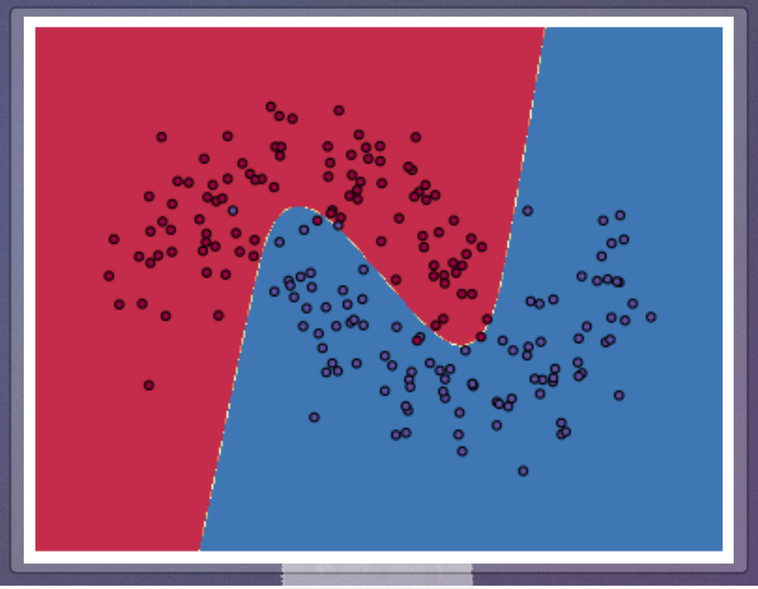
Boundary decisions according to the number of hidden layers

Hidden layers: The more, the merrier?

One (1) layer



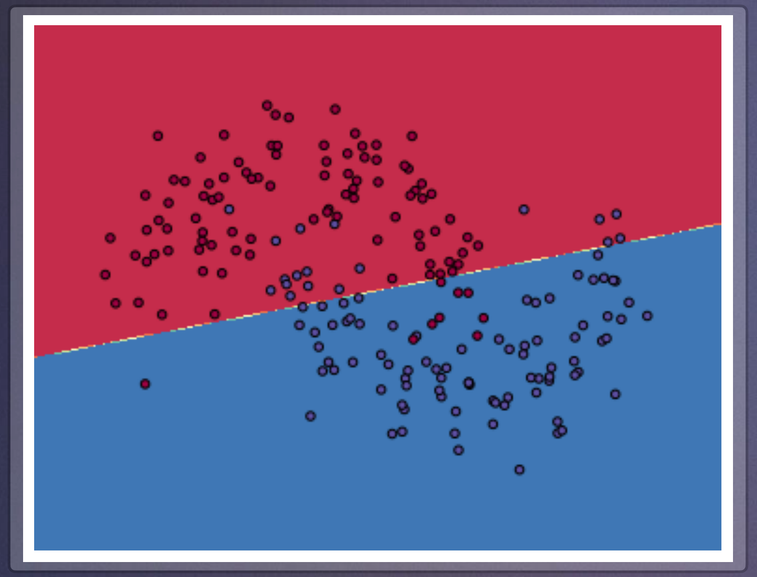
Three (3) layers



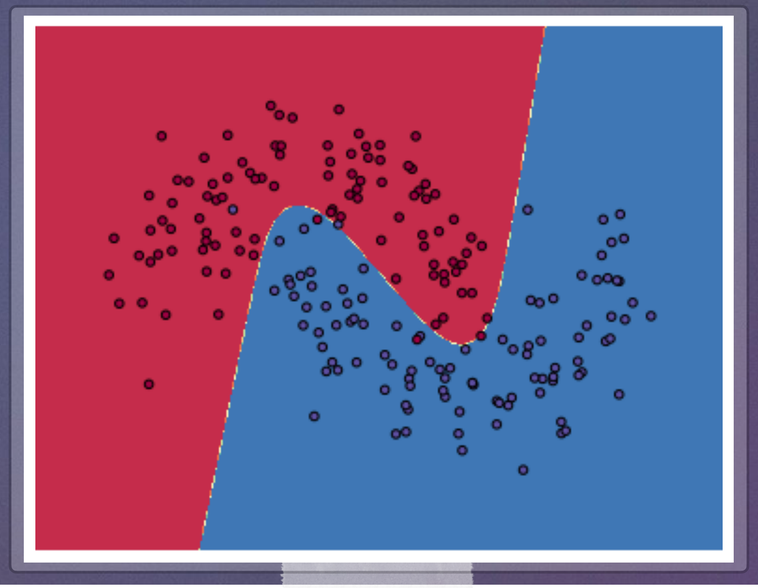
Boundary decisions according to the number of hidden layers

Hidden layers: The more, the merrier?

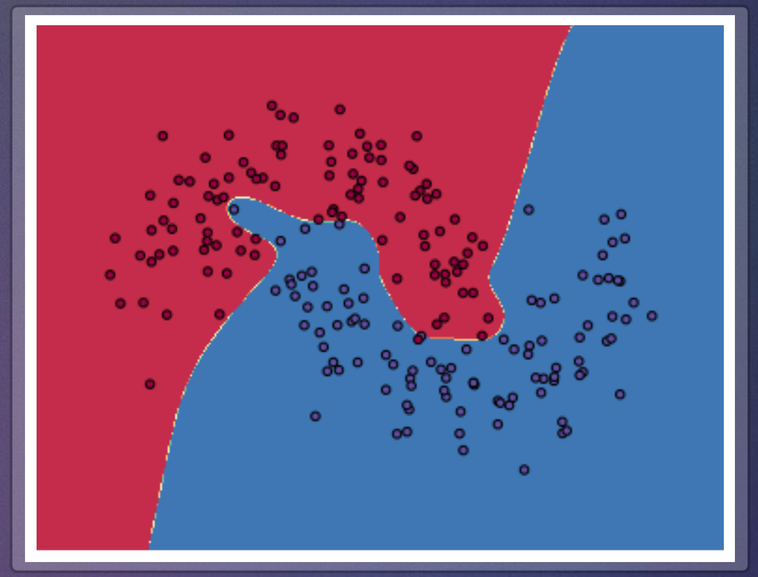
One (1) layer



Three (3) layers

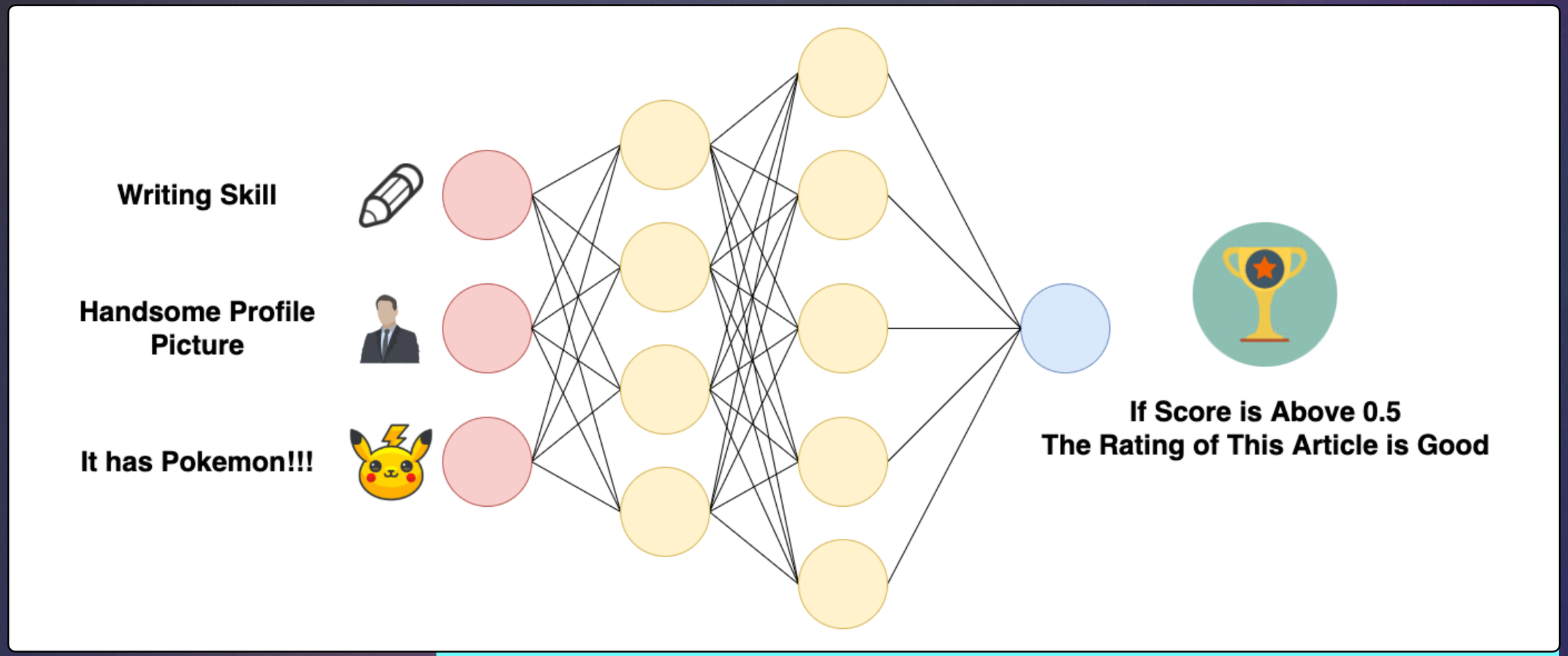


Fifty (50) layers



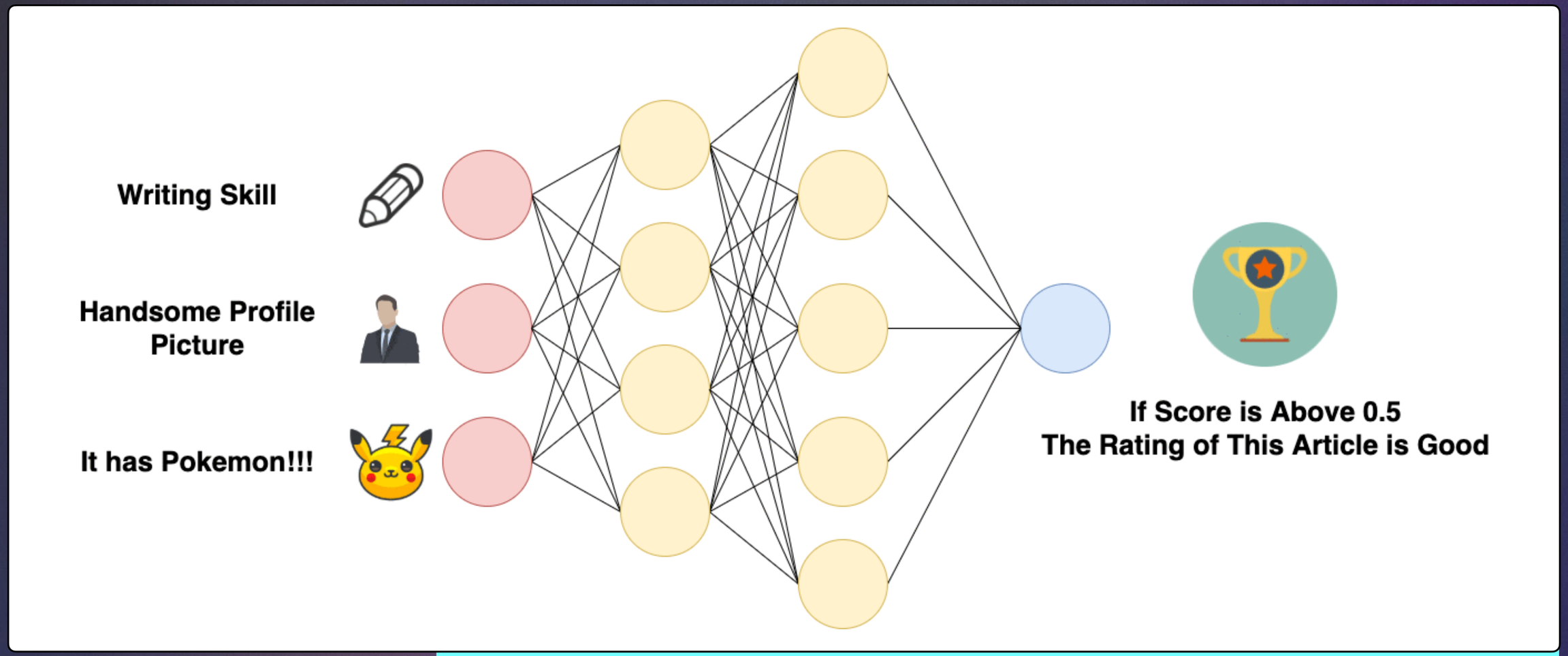
Boundary decisions according to the number of hidden layers

Forward propagation



https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae

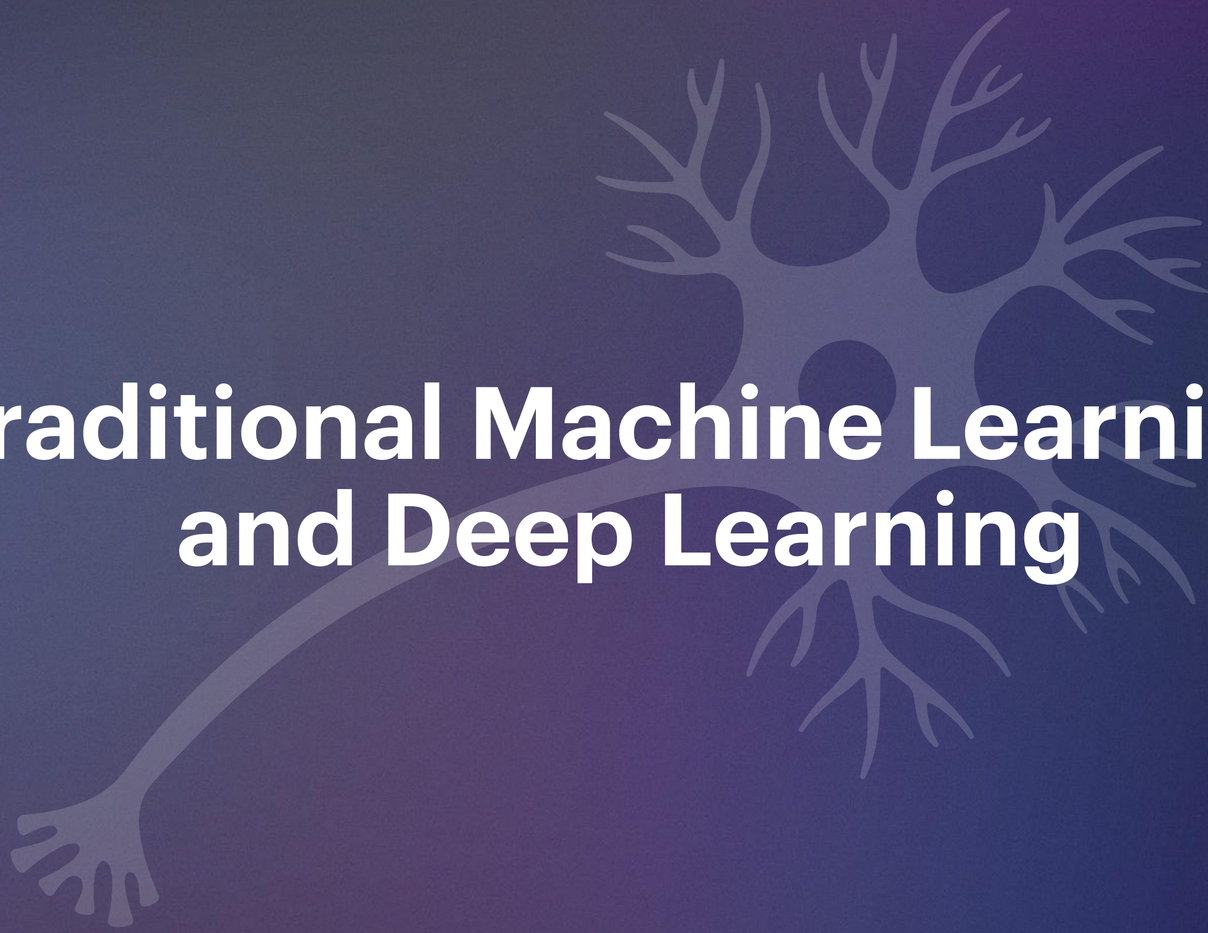
Forward propagation



https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae

02

Traditional Machine Learning and Deep Learning



INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

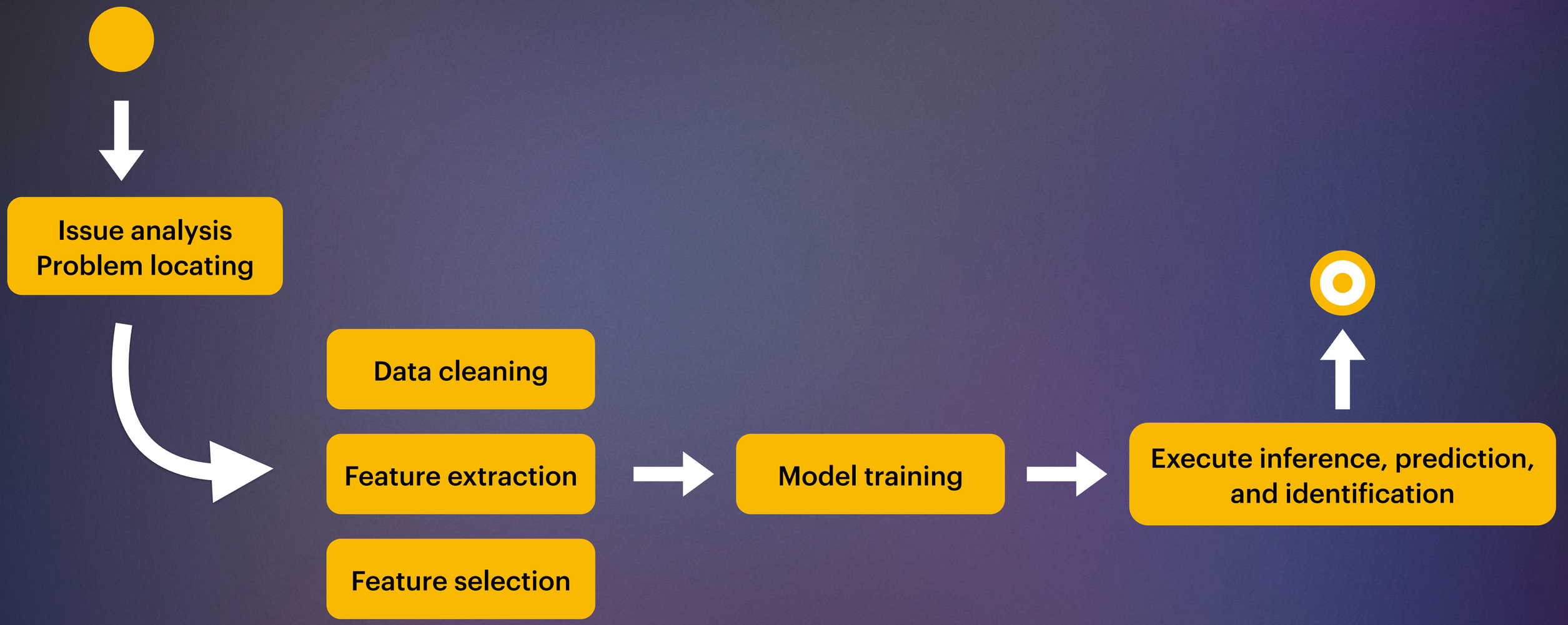
HUAWEI

Traditional Machine Learning and Deep Learning

As a model based on unsupervised feature learning and feature hierarchy learning, deep learning has great advantages in fields such as computer vision, speech recognition, and natural language processing.

Traditional Machine Learning	Deep Learning
Low hardware requirements on the computer: Given the limited computing amount, the computer **does not need a GPU** for parallel computing generally.	**Higher hardware requirements** on the computer: To execute matrix operations on massive data, the computer **needs a GPU** to perform parallel computing.
Applicable to training under a **small data amount** and whose **performance cannot be improved continuously as the data amount increases**.	Whose **performance can be high when high-dimensional weight parameters and massive training data** are provided.
Level-by-level problem breakdown	E2E learning
Manual feature selection	Algorithm-based automatic feature extraction
Easy-to-explain features	Hard-to-explain features

Traditional Machine Learning

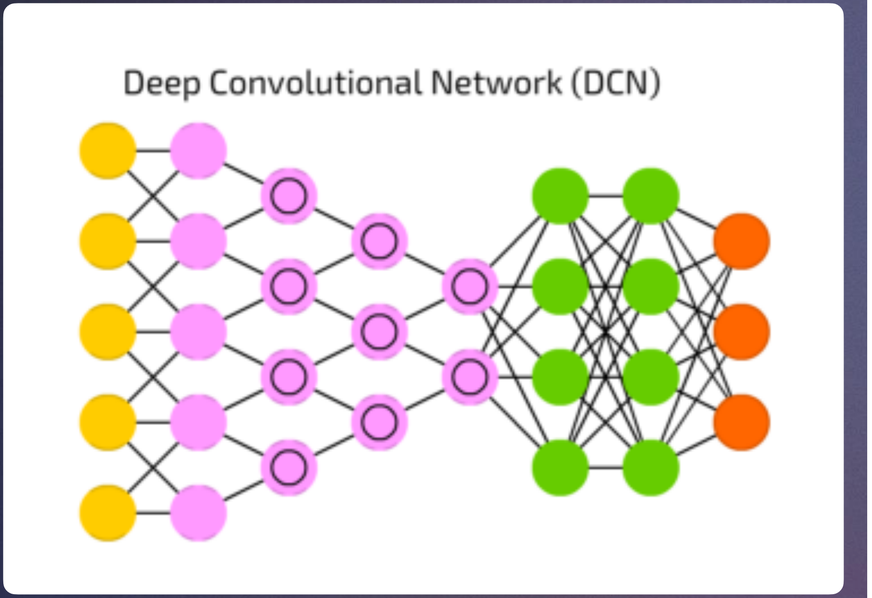


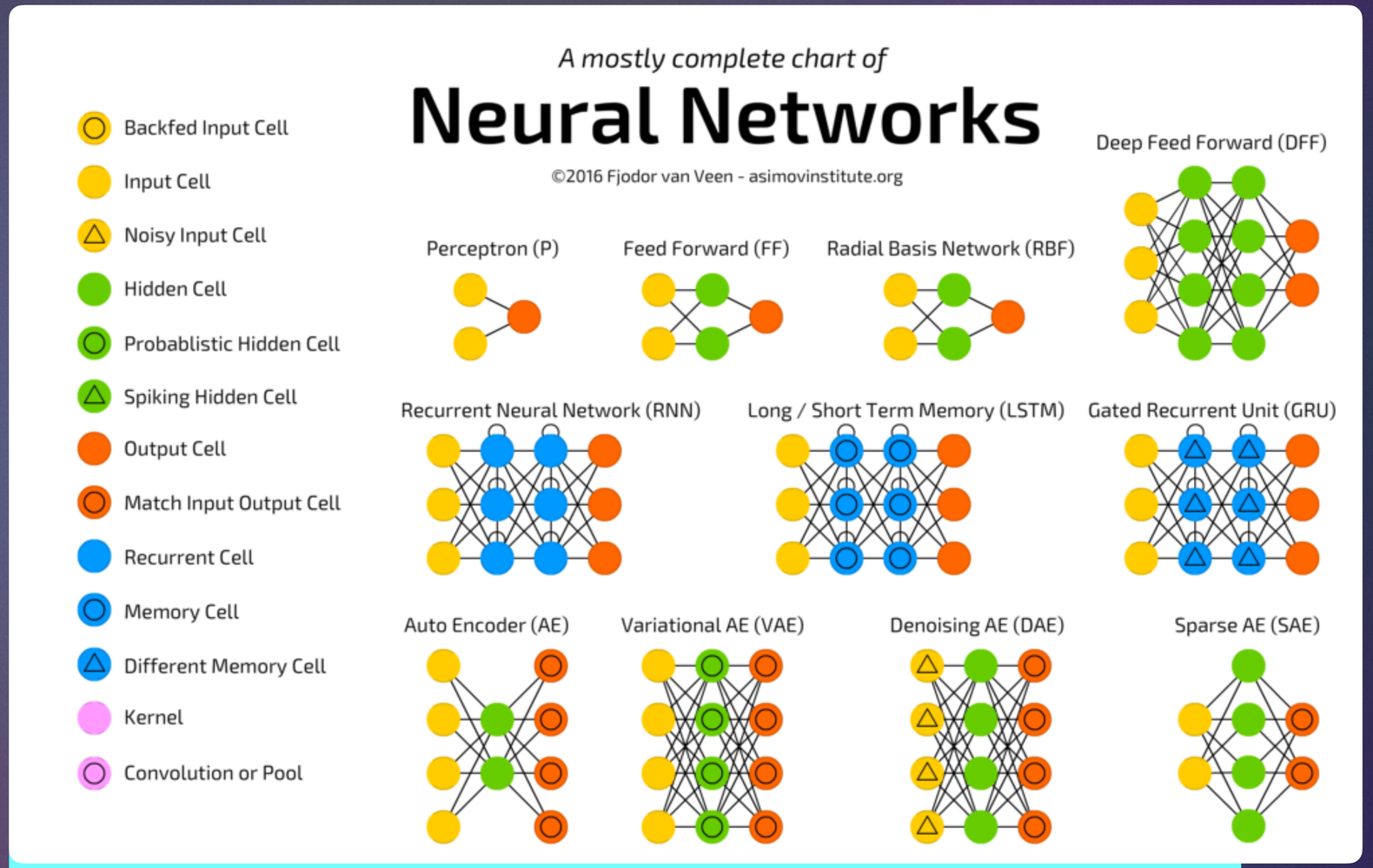
Neural Network

- Currently, the definition of the neural network has not been determined yet. Hecht Nielsen, a neural network researcher in the U.S., defines a neural network as **a computer system composed of simple and highly interconnected processing elements**, which process information by dynamic response to external inputs.
- A neural network can be simply expressed as an **information processing system designed to imitate the human brain structure and functions** based on its source, features, and explanations.
- Artificial neural network (neural network): **Formed by artificial neurons connected to each other, the neural network extracts and simplifies the human brain's microstructure and functions**. It is an important approach to simulate human intelligence and reflect several basic features of human brain functions, such as concurrent information processing, learning, association, model classification, and memory.

(Some) Neural Networks

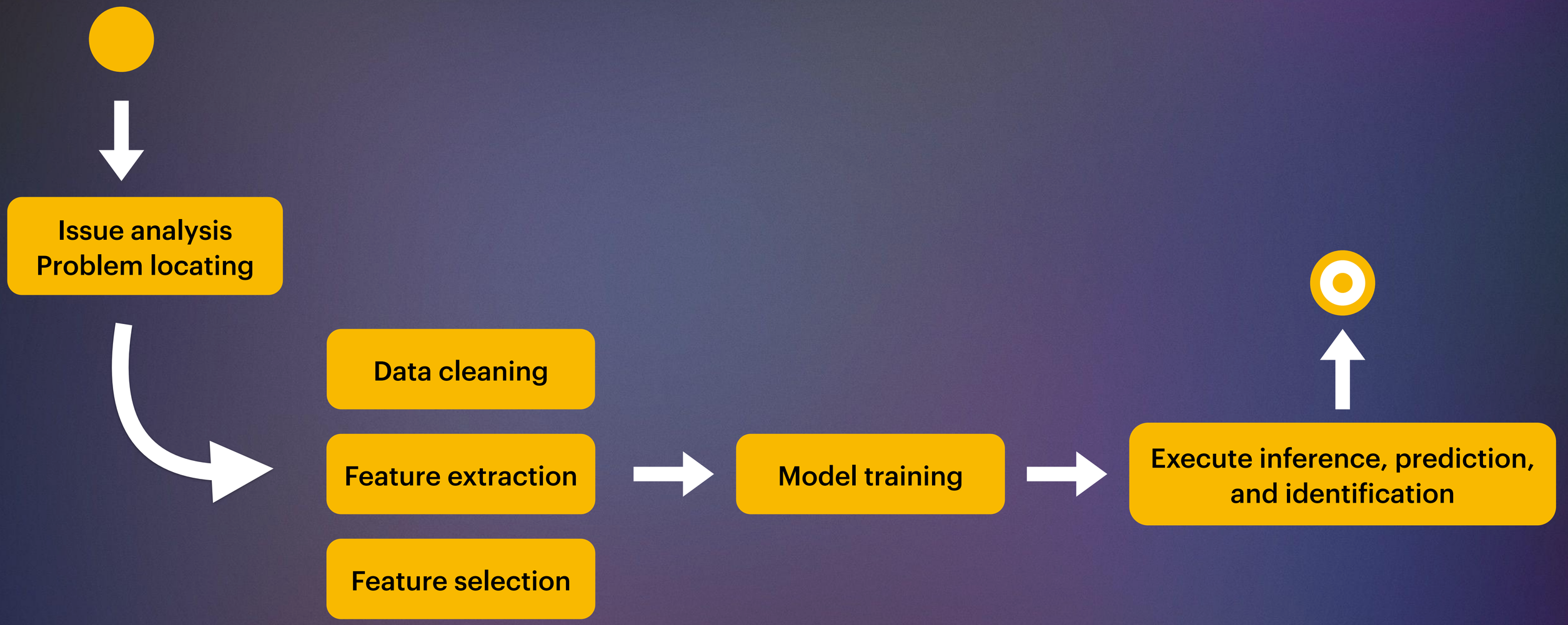
Usually, the deep learning architecture is a deep neural network. "**Deep**" in "**deep learning**" refers to the number of layers of the neural network.



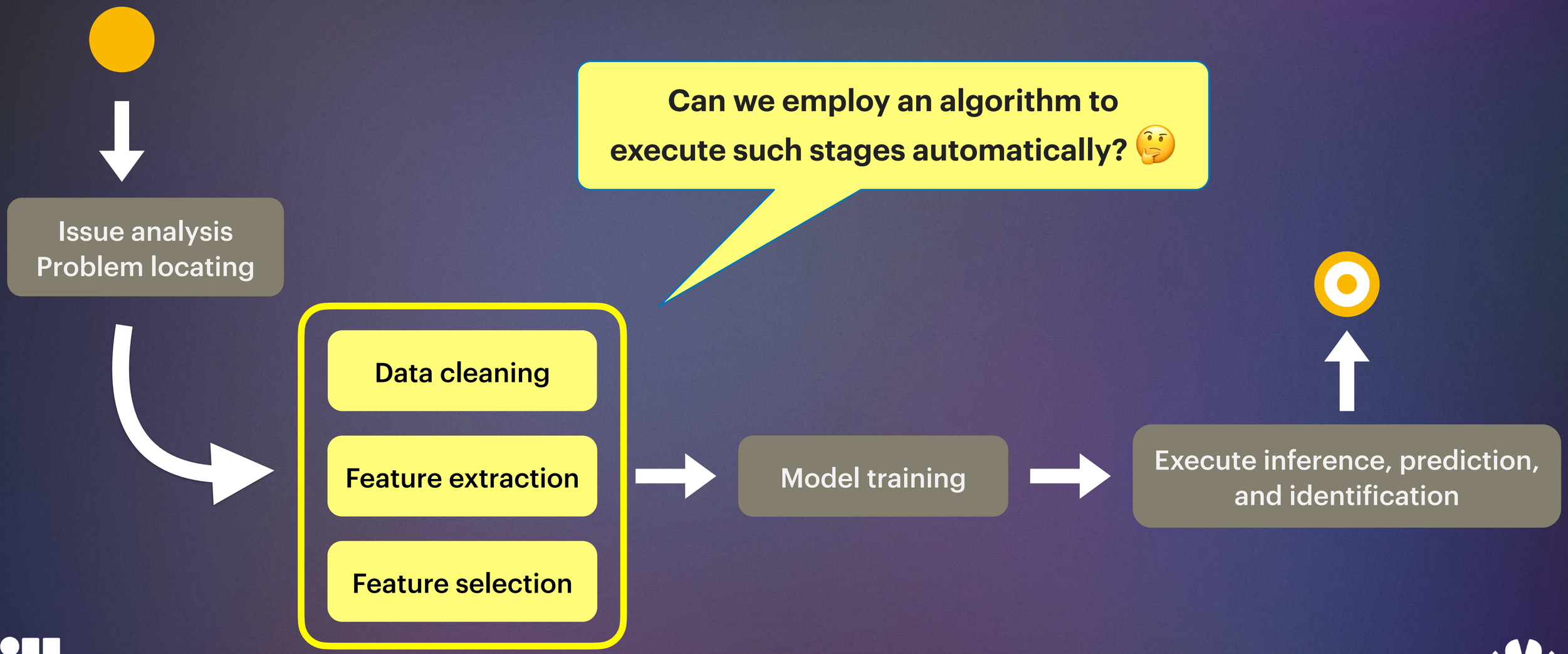


https://medium.com/predict/the-complete-list-to-make-you-an-ai-pro-be83448720b8

Traditional Machine Learning vs Deep Learning



Traditional Machine Learning vs Deep Learning



03

Training Rules

Training weights: a metaphor

Imagine you are a mountain climber on top of a mountain, and night has fallen. You need to get to your base camp at the bottom of the mountain, but in the darkness with only your dinky flashlight, you can't see more than a few feet of the ground in front of you. So how do you get down?

One strategy is to look in every direction to see which way the ground steeps downward the most, and then step forward in that direction. Repeat this process many times, and you will gradually go farther and farther downhill. You may sometimes get stuck in a small trough or valley, in which case you can follow your momentum for a bit longer to get out of it.

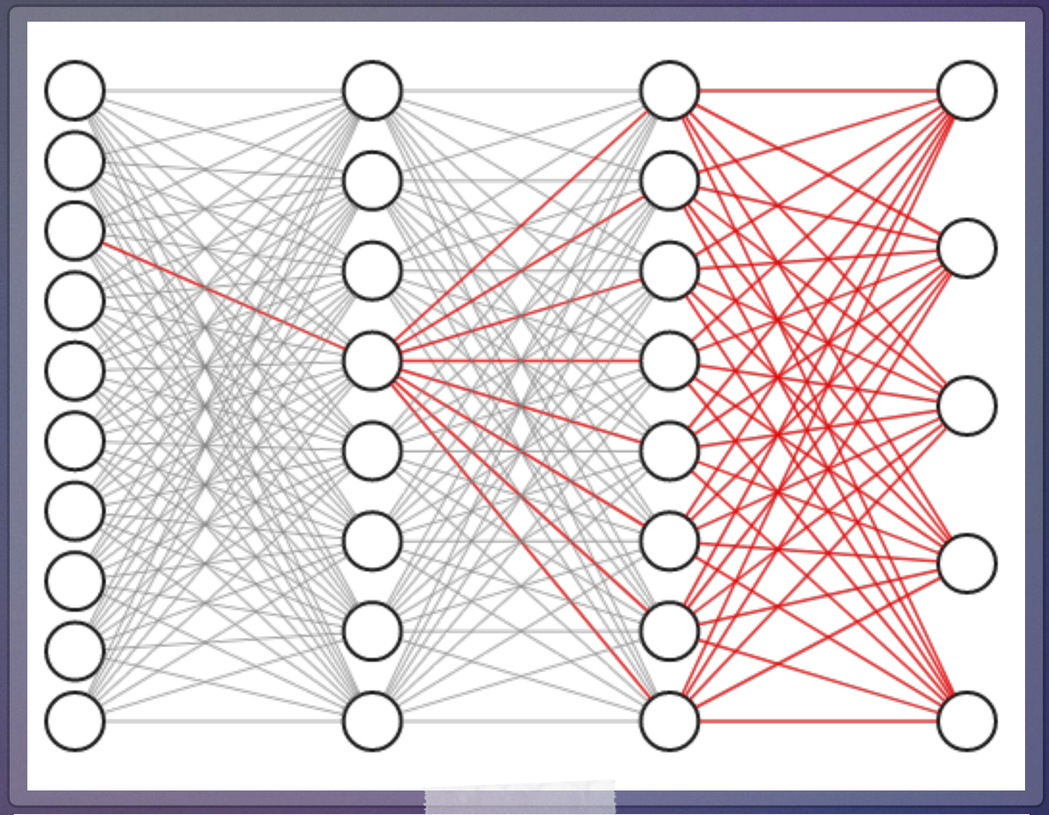
Caveats aside, this strategy will eventually get you to the bottom of the mountain.

Why training is hard?

The weights of a neural network with hidden layers are **highly interdependent**. To see why, consider the highlighted connection in the first layer of the three layer network after.

If we tweak the weight on that connection slightly, it will impact not only the neuron it propagates to directly, but also all of the neurons in the next two layers as well, and **thus affect all the outputs**.

https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

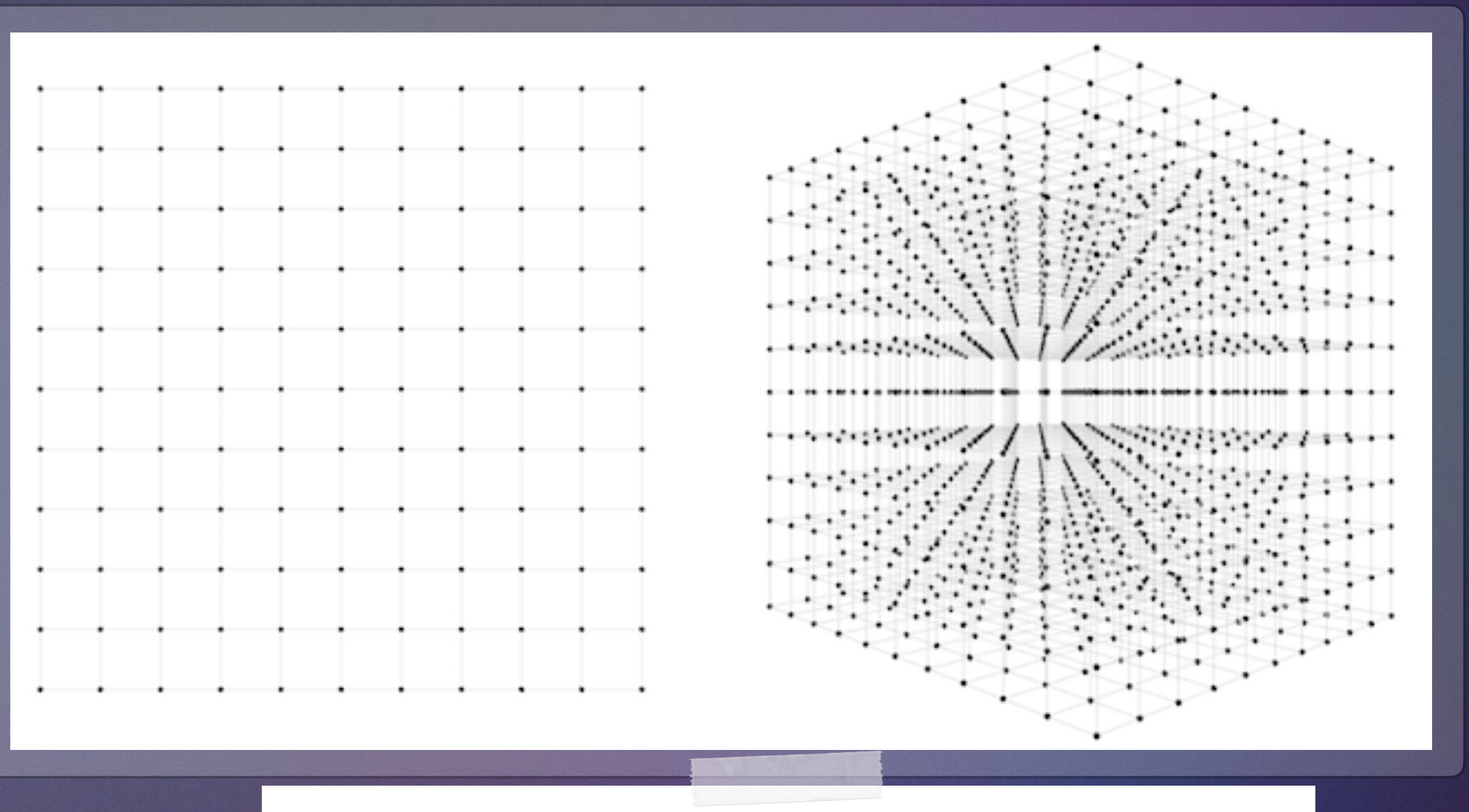


Tweaking the weight of one connection in the first layer will affect just one neuron in the next layer, but because of fully-connectedness, all neurons in subsequent layers will be changed.

“And now Joseph?”

So what happens when we try to use this approach to train our network for classifying MNIST digits? Recall that network has 784 input neurons, 15 neurons in 1 hidden layer, and 10 neurons in the output layer.

There are $178 \times 15 + 15 \times 10 = 11{,}935$ weights. Add 25 biases to the mix, and we have to simultaneously guess through 11,935 dimensions of parameters.

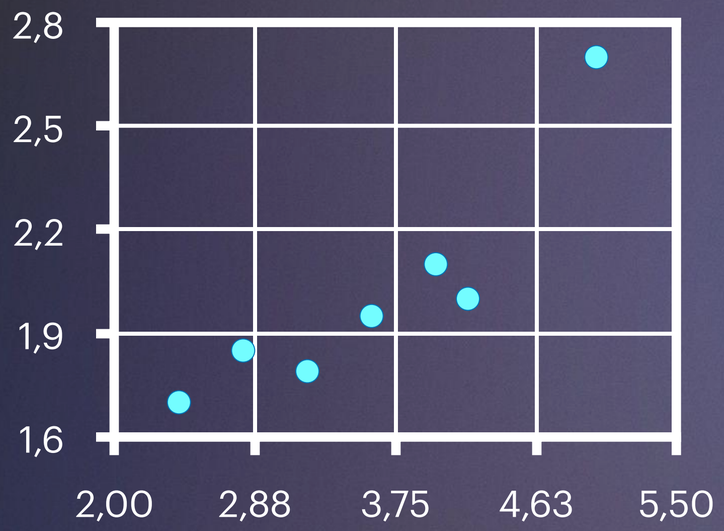


Left: a 2d square sampled to 10% density requires $10^2 = 100$ points.
Right: a 3d cube sampled to 10% density requires $10^3 = 1000$ points.

https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

Loss Function and Gradient Descent

Linear regression



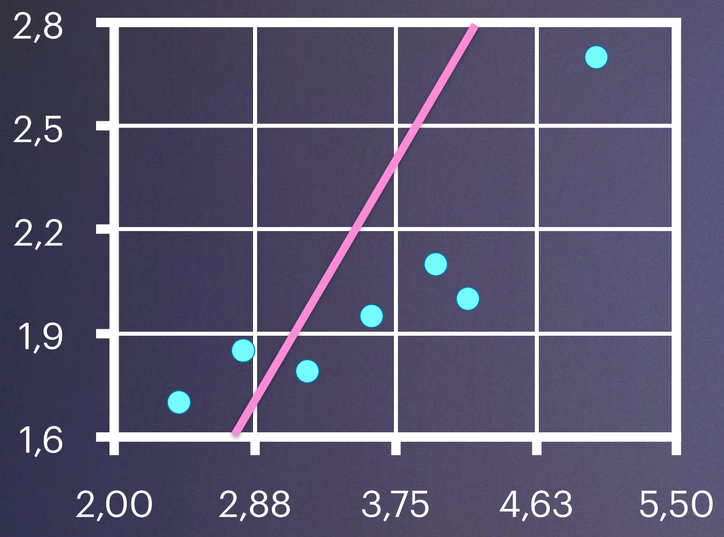
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

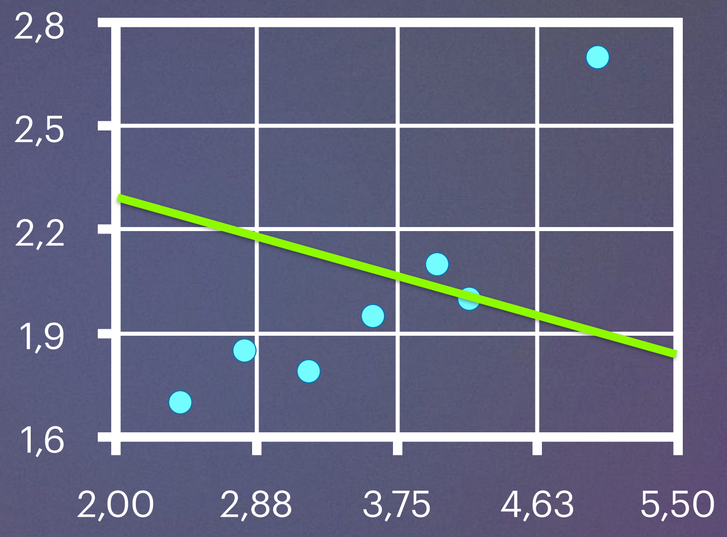
HUAWEI

Loss Function and Gradient Descent

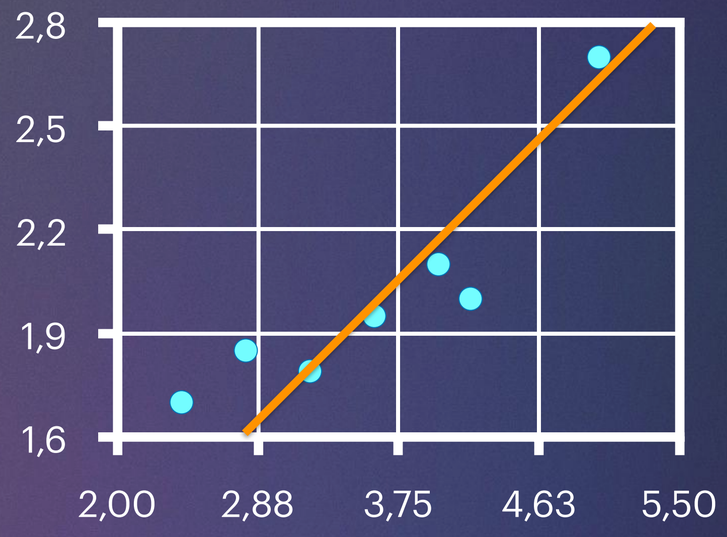
Linear regression



$f(x) = 0{,}92x - 1$

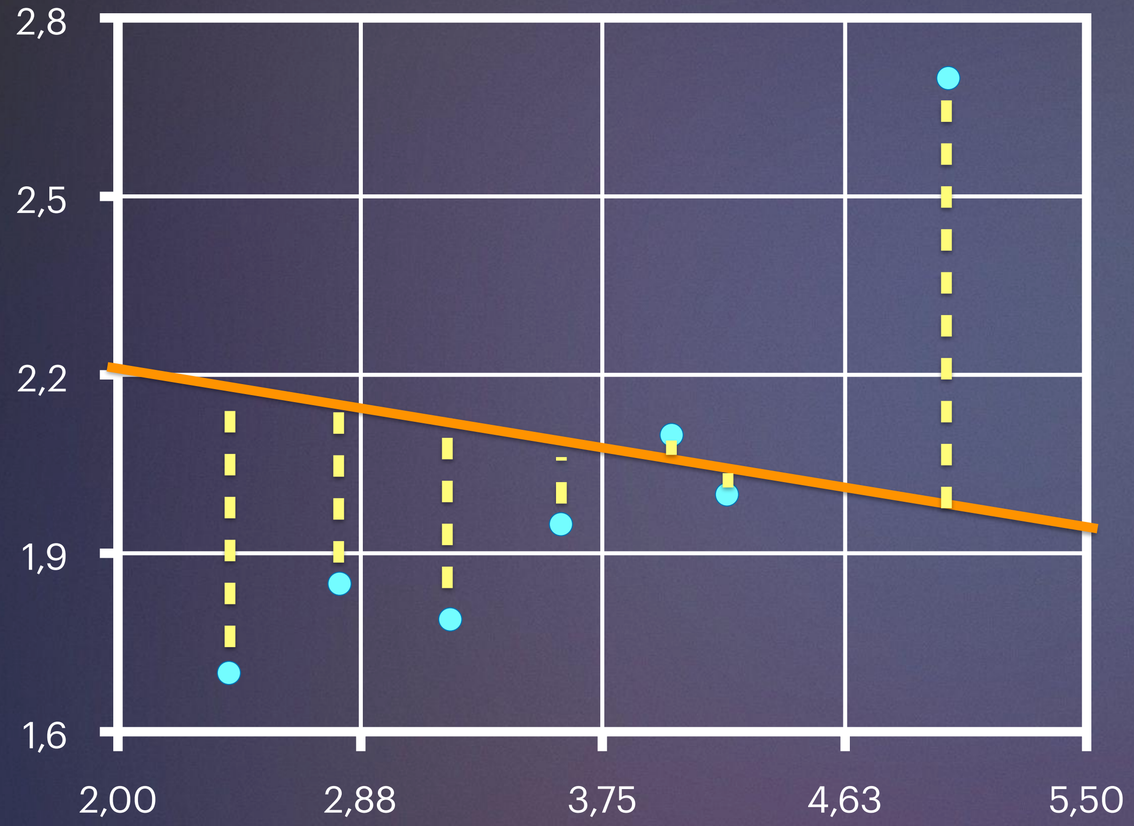


$f(x) = 0{,}11x + 2{,}5$



$f(x) = 0{,}52x - 0{,}1$

Loss Function (cost function)



The loss function – sometimes called a cost function – is a measure of the amount of error our linear regression makes on a dataset. Although many loss functions exist, all of them essentially penalize us on the distance between the predicted y-value $f(x)$ from a given x and its actual value in our dataset:

$$MSE(f) = \frac{1}{n}\sum_{i}\left(y_i - f(x_i)\right)^2$$

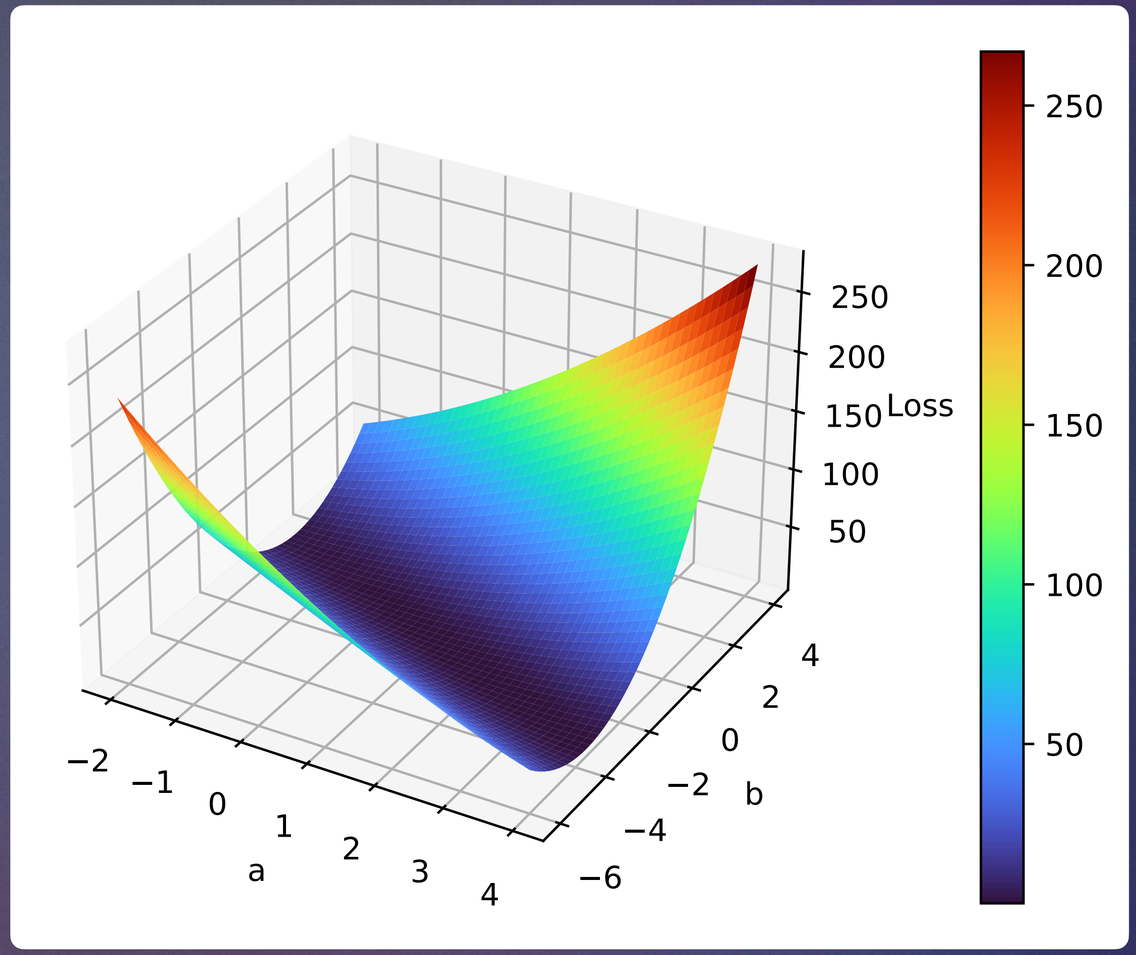
In our case, the above equation can be described as:

$$MSE(f) = \frac{1}{n}\sum_{i}\left(y_i - (ax_i + b)\right)^2$$

https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

Loss Function (cost function)

We can get some intuition if we calculate the MSE for all a and b within some neighborhood and compare them. Consider the figure in which uses a visualizations of the MSE in the range where the slope a is between -2 and 4, and the intercept b is between -6 and 8.

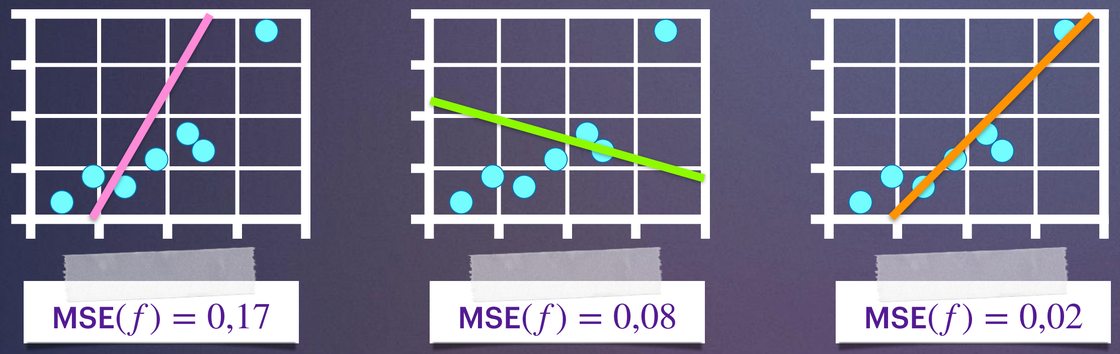


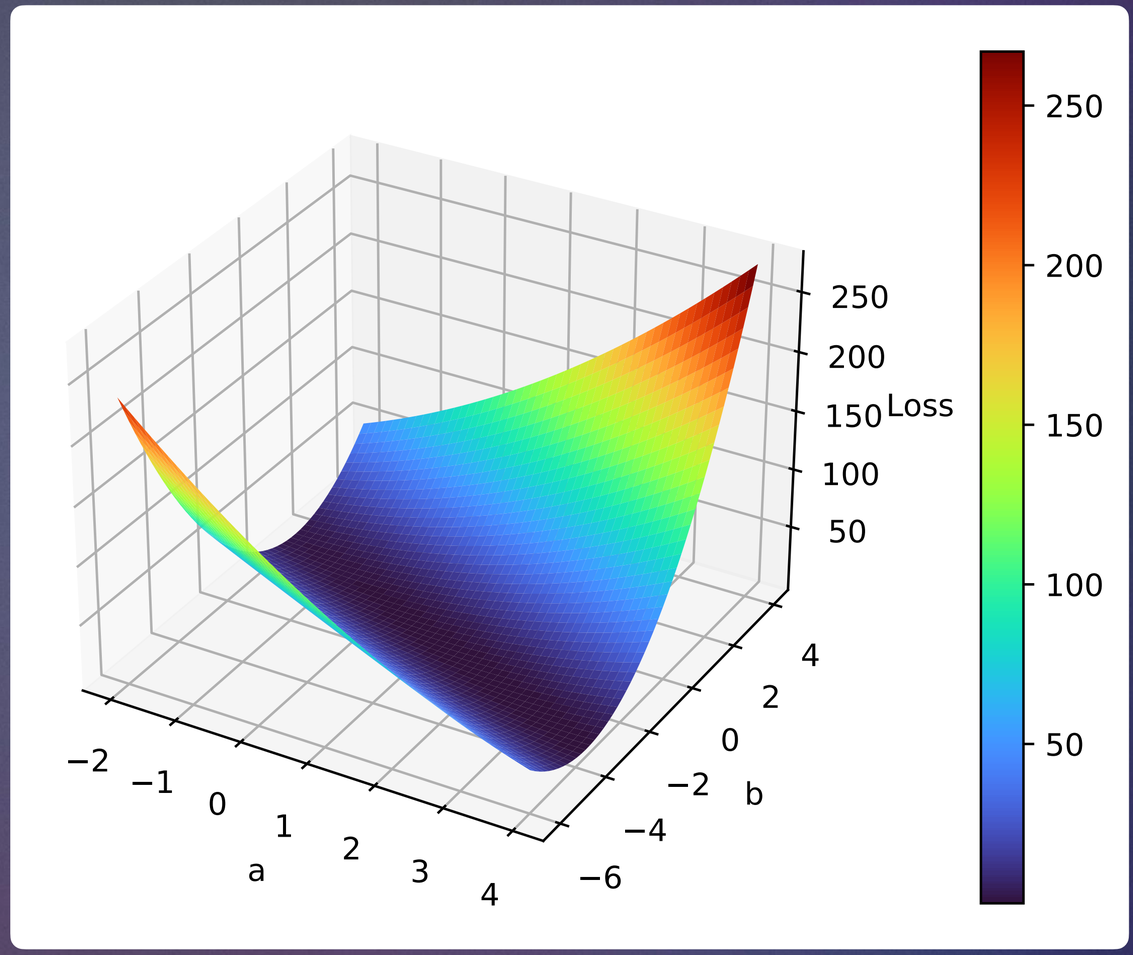
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

Loss Function (cost function)

We can get some intuition if we calculate the MSE for all a and b within some neighborhood and compare them. Consider the figure in which uses a visualizations of the MSE in the range where the slope a is between -2 and 4, and the intercept b is between -6 and 8.





Common Loss Functions in Deep Learning

- Quadratic cost function:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i}\left(y_i - f(x_i)\right)^2.$$

- Cross entropy error function:

$$J(\mathbf{w}) = -\frac{1}{n}\sum_{x}\sum_{i}\left[y_i \log\left(f(x_i)\right) + (1-y_i)\,\log\left(1-f(x_i)\right)\right].$$

- The cross entropy error function depicts the distance between two probability distributions, which is a widely used loss function for classification problems.
- Generally, the mean square error function is used to solve the regression problem, while the cross entropy error function is used to solve binary classification problems.

Gradient Descent

The general problem we've been dealing with – that of finding parameters to satisfy some objective function – is not specific to machine learning. Indeed it is a very general problem found in **mathematical optimization**, known to us for a long time, and encountered in far more scenarios than just neural networks.

Today, many problems in multivariable function optimization – including training neural networks – generally rely on a very effective algorithm called gradient descent to find a good solution much faster than taking random guesses, and more powerful than linear regression.

The Gradient Descent Method

Intuitively, the Gradient Descent works similarly to the mountain climber analogy given at the beginning of the class. First, we start with a random guess at the parameters and start there. We then figure out which direction the loss function steeps downward the most (changing the parameters) and step slightly in that direction. To put it another way, we determine the amounts to tweak all of the parameters such that the loss function goes down by the most considerable amount. We repeat this process over and over until we are satisfied we have found the lowest point.

To figure out which direction the loss steeps downward the most, one must calculate the loss function's gradient concerning all of the parameters. **A gradient is a multidimensional generalization of a derivative**; it is a vector containing each of the partial derivatives of each variable. In other words, it is a vector that contains the slope of the loss function along every axis.

The Gradient Descent Method

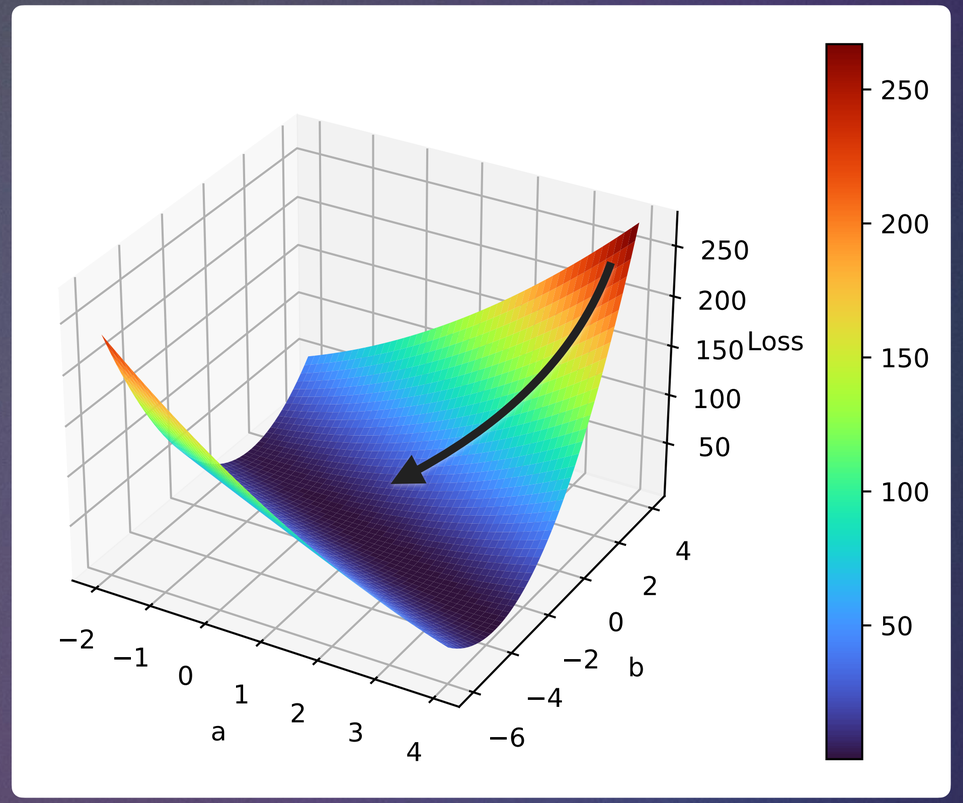
Recall the mean squared error loss we introduced in the previous, which we will denote as:

$$J = \frac{1}{n}\sum_i \left(y_i - (ax_i + b)\right)^2.$$

There are two parameters we are trying to optimize: a and b. Let's calculate the partial derivative of J with respect to each of them:

$$\frac{\partial J}{\partial a} = \frac{2}{n}\sum_i x_i \left(y_i - (ax_i + b)\right), \text{ and}$$

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_i \left(y_i - (ax_i + b)\right).$$



https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

The Gradient Descent Method

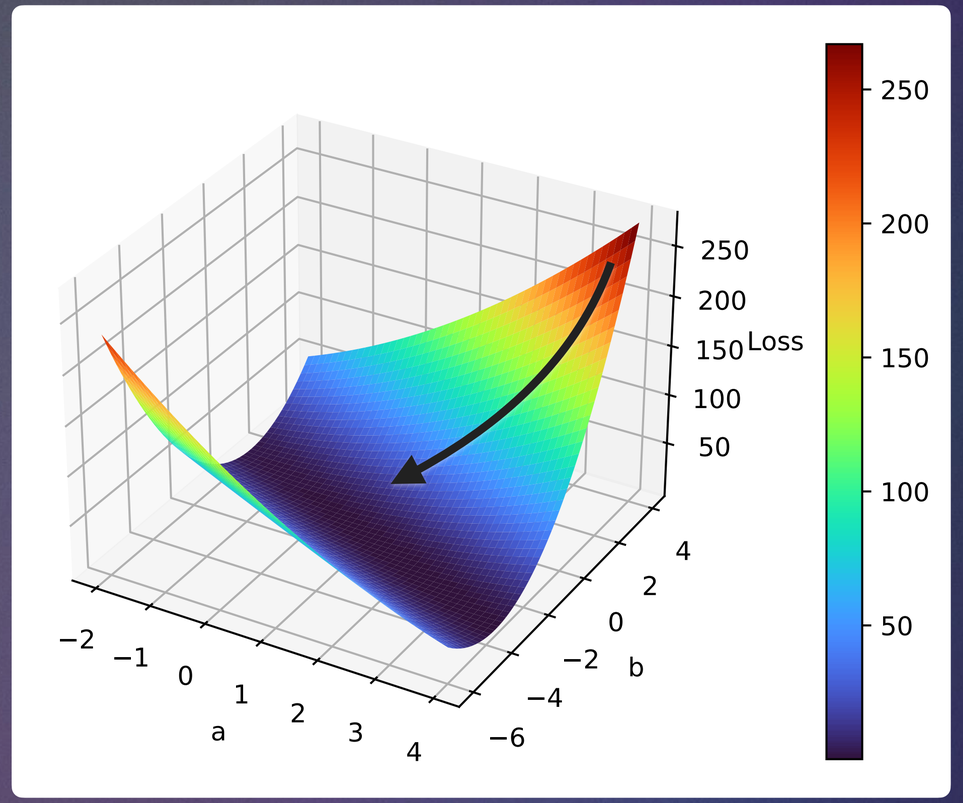
Recall the mean squared error loss we introduced in the previous, which we will denote as:

$$J = \frac{1}{n}\sum_i \left(y_i - (ax_i + b)\right)^2.$$

There are two parameters we are trying to optimize: a and b. Let's calculate the partial derivative of J with respect to each of them:

$$\frac{\partial J}{\partial a} = \frac{2}{n}\sum_i x_i \left(y_i - (ax_i + b)\right), \text{ and}$$

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_i \left(y_i - (ax_i + b)\right).$$



https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

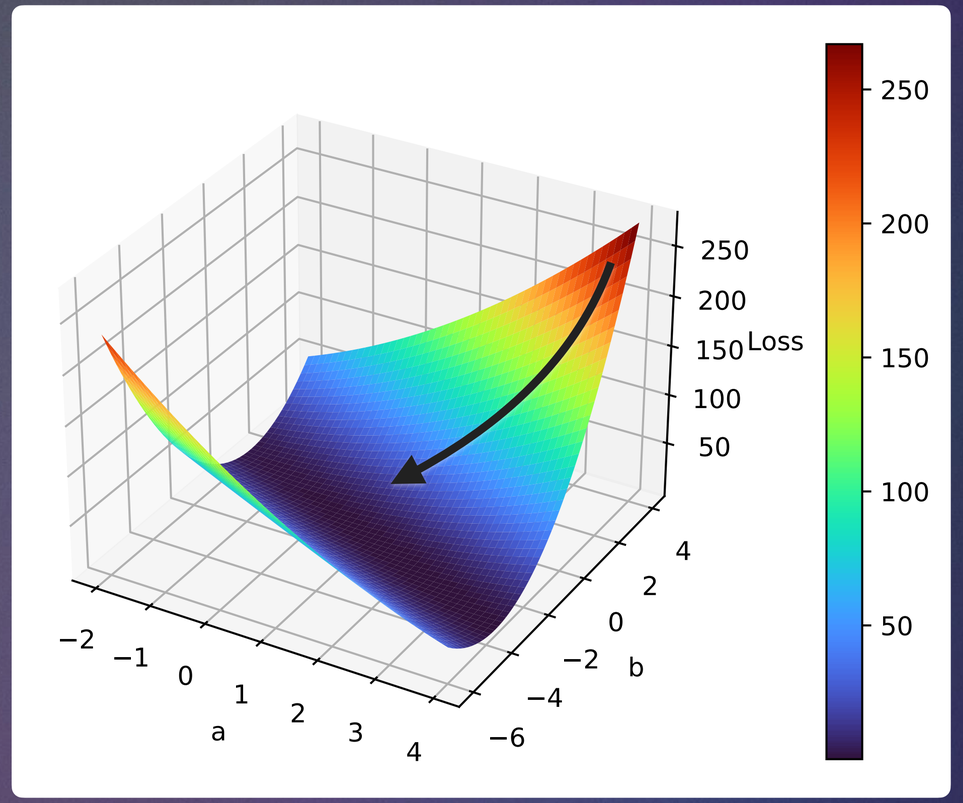
The Gradient Descent Method

How far in that direction should we step? Such a choice turns out to be an important one, and in ordinary gradient descent, this **is left as a hyperparameter to decide manually**. This hyperparameter – known as the learning rate – is generally the most important and sensitive hyperparameter to set and is often denoted as η. If η is set too low, it may take an unacceptably long time to get to the bottom. If η is too high, we may overshoot the correct path or even climb upwards.

Now, we can write the update steps for the two parameters as follows:

$$a^{(t+1)} = a^{(t)} - \eta \frac{\partial J}{\partial a}, \text{ and}$$

$$b^{(t+1)} = b^{(t)} - \eta \frac{\partial J}{\partial b}.$$

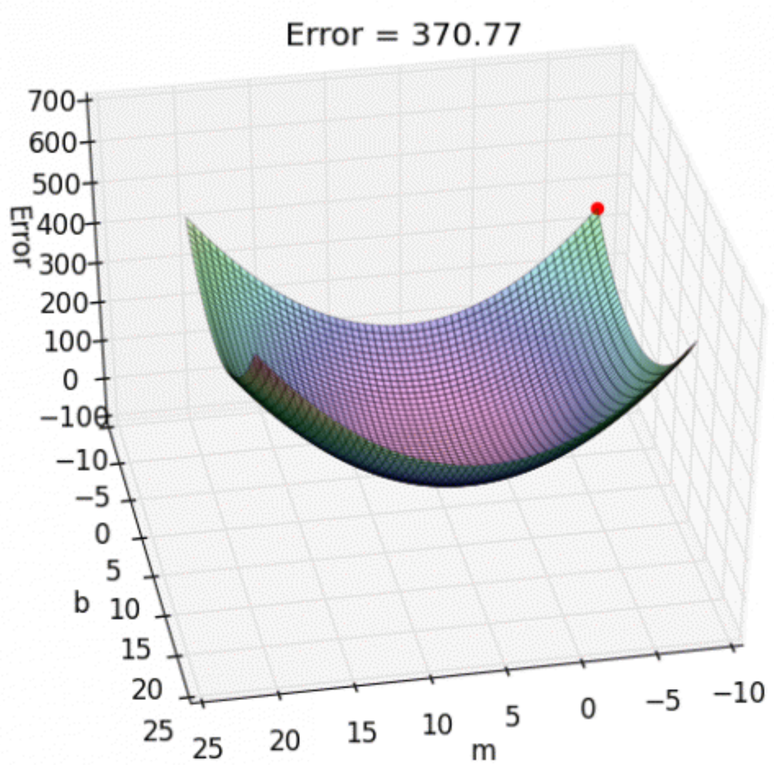


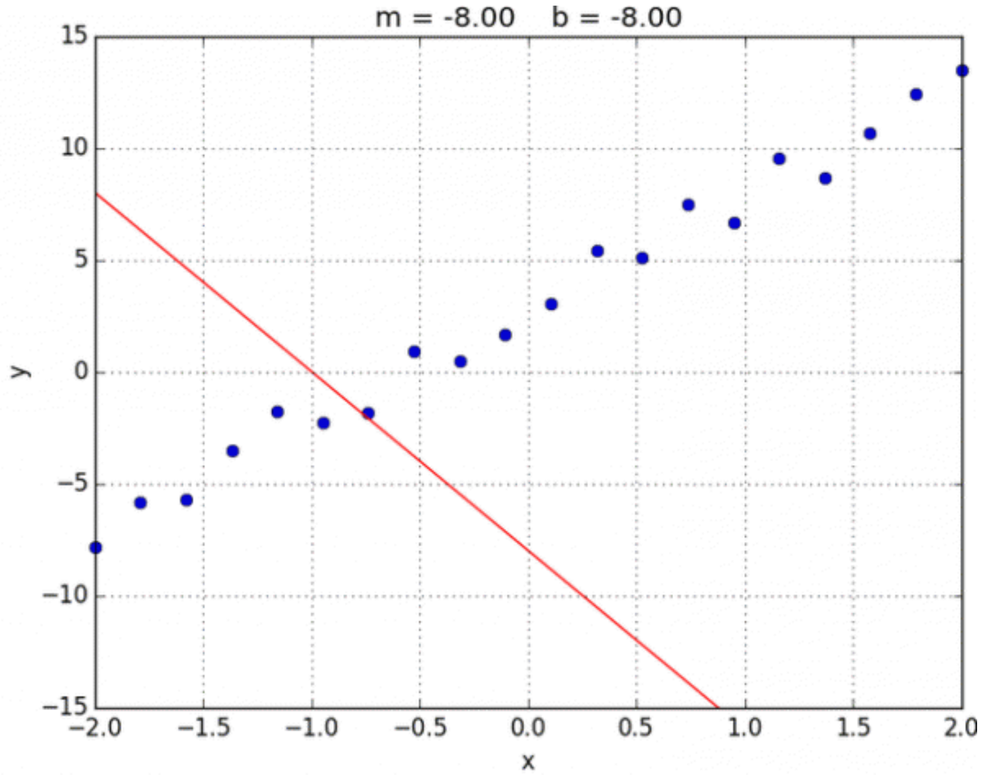
https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

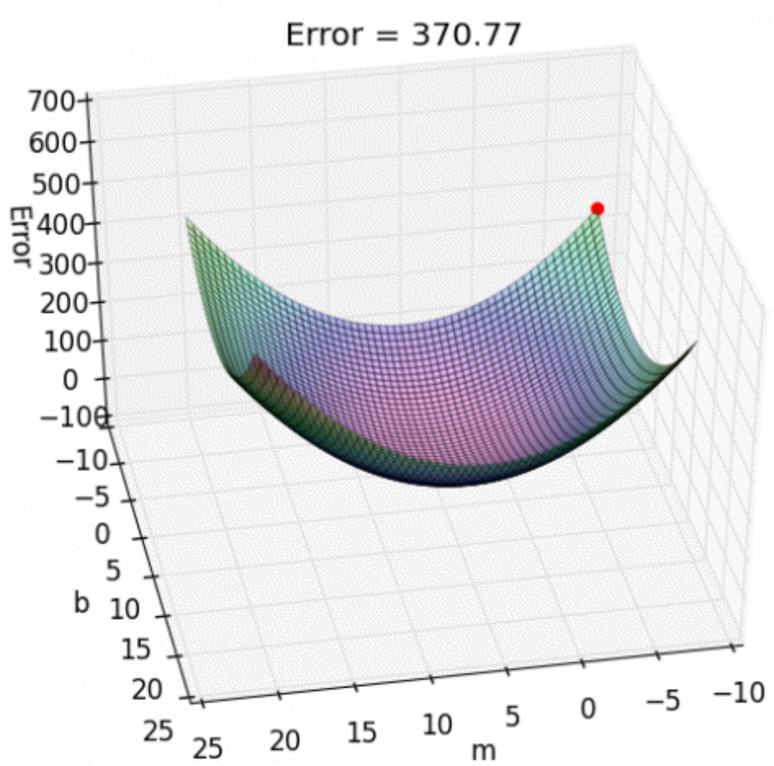
The Gradient Descent in Action!!!!

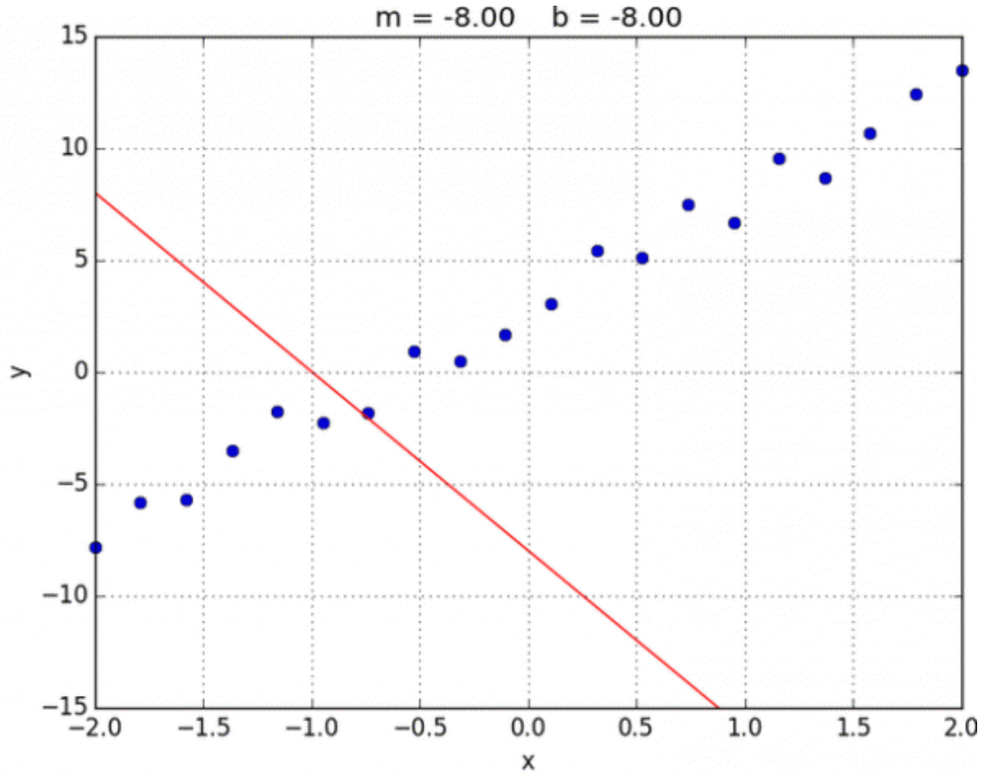




https://gifer.com/en/NqNM

The Gradient Descent in Action!!!!





https://gifer.com/en/NqNM

What about high-dimensional spaces?

And if there are more dimensions? If we denote all of our parameters as w_i, thus giving us the form $f(\mathbf{x}) = b + \mathbf{w}^\intercal \mathbf{x}$, then we can extrapolate the above example to the multidimensional case. This can be written down more succinctly using gradient notation. Recall that the gradient of J, which we will denote as ∇J, is the vector containing each of the partial derivatives. Thus we can represent the above update step as:

$$\nabla J(\mathbf{w}) = \left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n} \right),$$

with the following update rule:

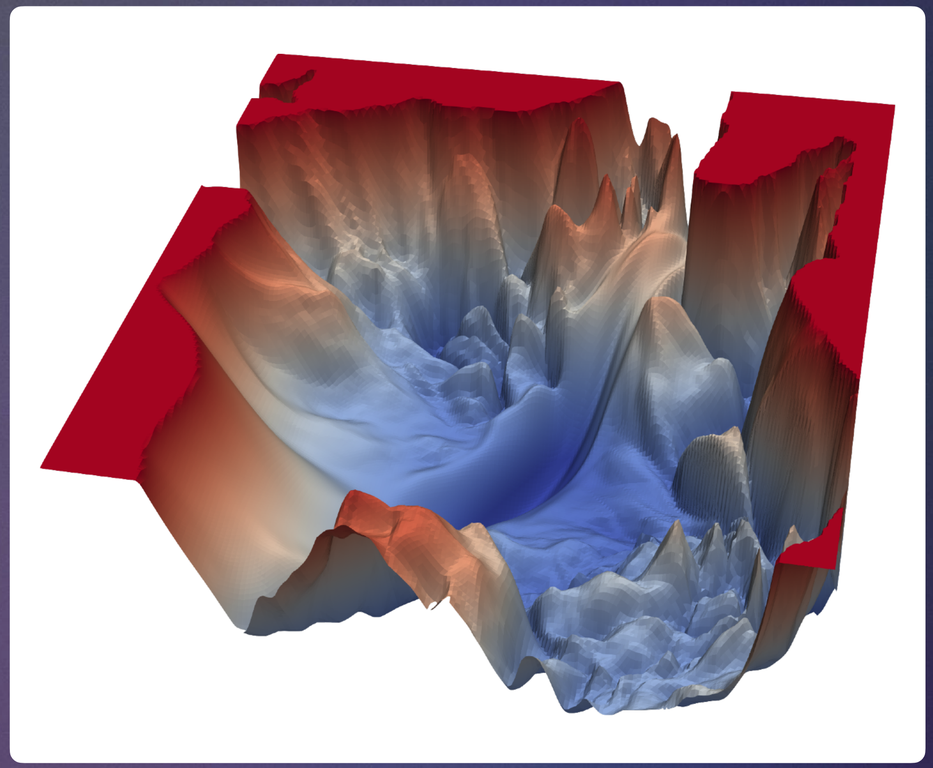
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla J(\mathbf{w}).$$

The above formula is the canonical formula for ordinary gradient descent. It is guaranteed to get you the best set of parameters for a linear regression, or indeed for any linear optimization problem.

Applying gradient descent to neural nets

Due to the nonlinearity introduced by their activation functions, a neural net's loss function is not convex. Instead, its loss function is much more complex, with many hills and valleys and curves and other irregularities.

We will get there later!



LI, Hao et al. Visualizing the loss landscape of neural nets. arXiv preprint arXiv:1712.09913, 2017.

Such a condition means there are many local minima, i.e., parameterizations, where the loss is the lowest in its immediate neighborhood, but not necessarily the absolute minimum (or "global minimum"). This circumstance means that we might accidentally get stuck in a local minimum if we run gradient descent.

https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

Recap: The Gradient Descent Method

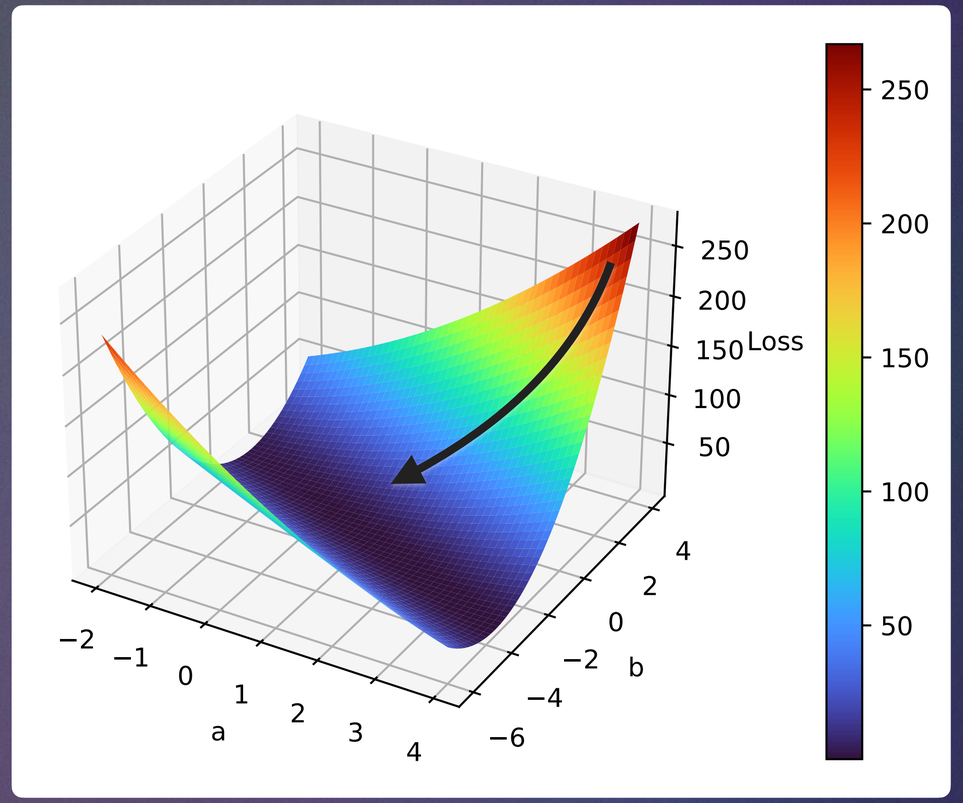
Recall the mean squared error loss we introduced in the previous:

$$J = \frac{1}{n}\sum_i \left(y_i - (ax_i + b)\right)^2.$$

As for two parameters: a and b. The partial derivative of J with respect to each of them are:

$$\frac{\partial J}{\partial a} = \frac{2}{n}\sum_i x_i \left(y_i - (ax_i + b)\right), \text{ and}$$

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_i \left(y_i - (ax_i + b)\right).$$



https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

Recap: The Gradient Descent Method

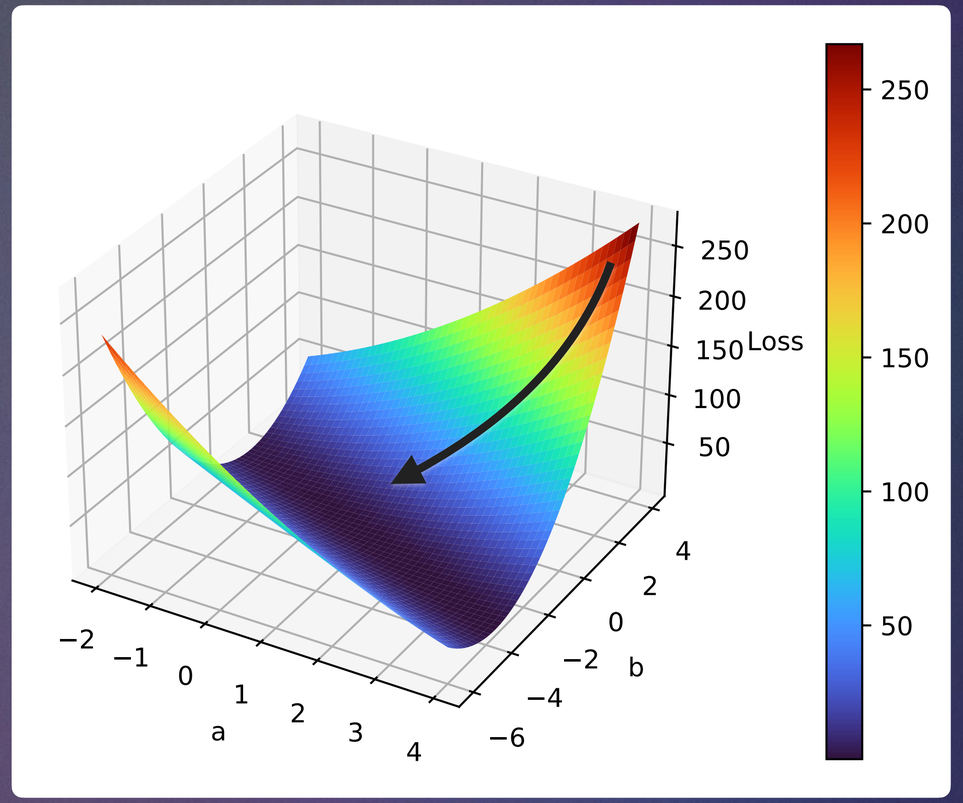
Recall the mean squared error loss we introduced in the previous:

$$J = \frac{1}{n}\sum_i \left(y_i - (ax_i + b)\right)^2.$$

As for two parameters: a and b. The partial derivative of J with respect to each of them are:

$$\frac{\partial J}{\partial a} = \frac{2}{n}\sum_i x_i \left(y_i - (ax_i + b)\right), \text{ and}$$

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_i \left(y_i - (ax_i + b)\right).$$



https://ml4a.github.io/ml4a/how_neural_networks_are_trained/

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

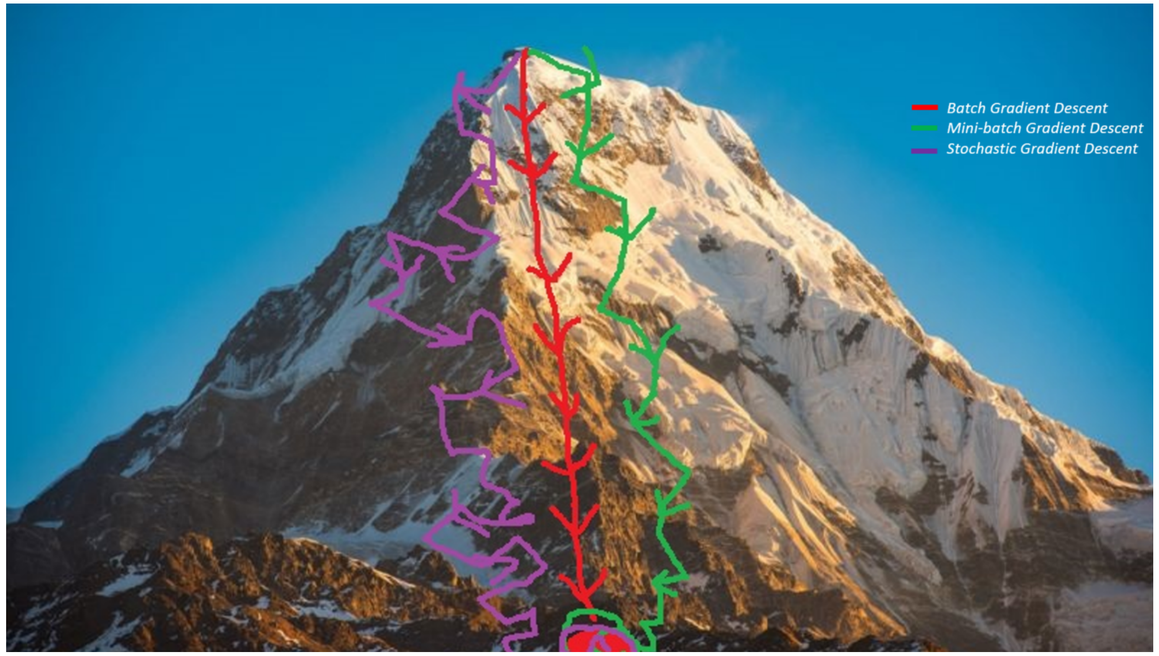
HUAWEI

Stochastic, batch, and mini-batch gradient descent

Besides, the **original gradient descent** has another significant problem for local minima: **it is too slow**. A neural net may have hundreds of millions of parameters; this means a single example from our dataset requires hundreds of millions of operations to evaluate.

Subsequently, gradient descent evaluated over all of our dataset – also known as **batch gradient descent** – is a costly and slow operation. Moreover, because every dataset has inherent redundancy, it can be shown that **a large enough subset of points can approximate the full gradient** in any way, making batch gradient descent unnecessarily expensive to estimate the gradient.

https://ml4a.github.io/ml4a/how_neural_networks_are_trained/



https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Batch gradient descent

THE MAIN ADVANTAGES:

- We can use fixed learning rate during training without worrying about learning rate decay.
- It has straight trajectory towards the minimum and it is guaranteed to converge in theory to the global minimum if the loss function is convex and to a local minimum if the loss function is not convex.
- It has unbiased estimate of gradients. The more the examples, the lower the standard error.

PYTHON: BATCH GRADIENT DESCENT

```
for _ in range(num_epochs):
    grad = compute_gradient(data, params)
    params = params - learning_rate * grad
```

THE MAIN DISADVANTAGES:

- Even though we can use vectorized implementation, it may still be slow to go over all examples especially when we have large datasets.
- Each step of learning happens after going over all examples where some examples may be redundant and don't contribute much to the update.

https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Mini-batch gradient descent

THE MAIN ADVANTAGES:

- Faster than Batch version because it goes through a lot less examples than Batch (all examples).
- Randomly selecting examples will help avoid redundant examples or examples that are very similar that don't contribute much to the learning.
- With batch size < size of training set, it adds noise to the learning process that helps improving generalization error.
- Even though with more examples the estimate would have lower standard error, the return is less than linear compared to the computational burden we incur.

PYTHON: MINI-BATCH GRADIENT DESCENT

```
for _ in range(num_epochs):
    np.random.shuffle(data)
    for batch in random_minibatches(data, batch_size=32):
        grad = compute_gradient(batch, params)
        params = params - learning_rate * grad
```

THE MAIN DISADVANTAGES:

- It won't converge. On each iteration, the learning step may go back and forth due to the noise. Therefore, it wanders around the minimum region but never converges.
- Due to the noise, the learning steps have more oscillations (see figure 4) and requires adding learning-decay to decrease the learning rate as we become closer to the minimum.

https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Stochastic gradient descent

IT SHARES MOST OF THE ADVANTAGES AND THE DISADVANTAGES WITH MINI-BATCH VERSION. BELOW ARE THE ONES THAT ARE SPECIFIC TO SGD:

- It adds even more noise to the learning process than mini-batch that helps improving generalization error. However, this would increase the run time.

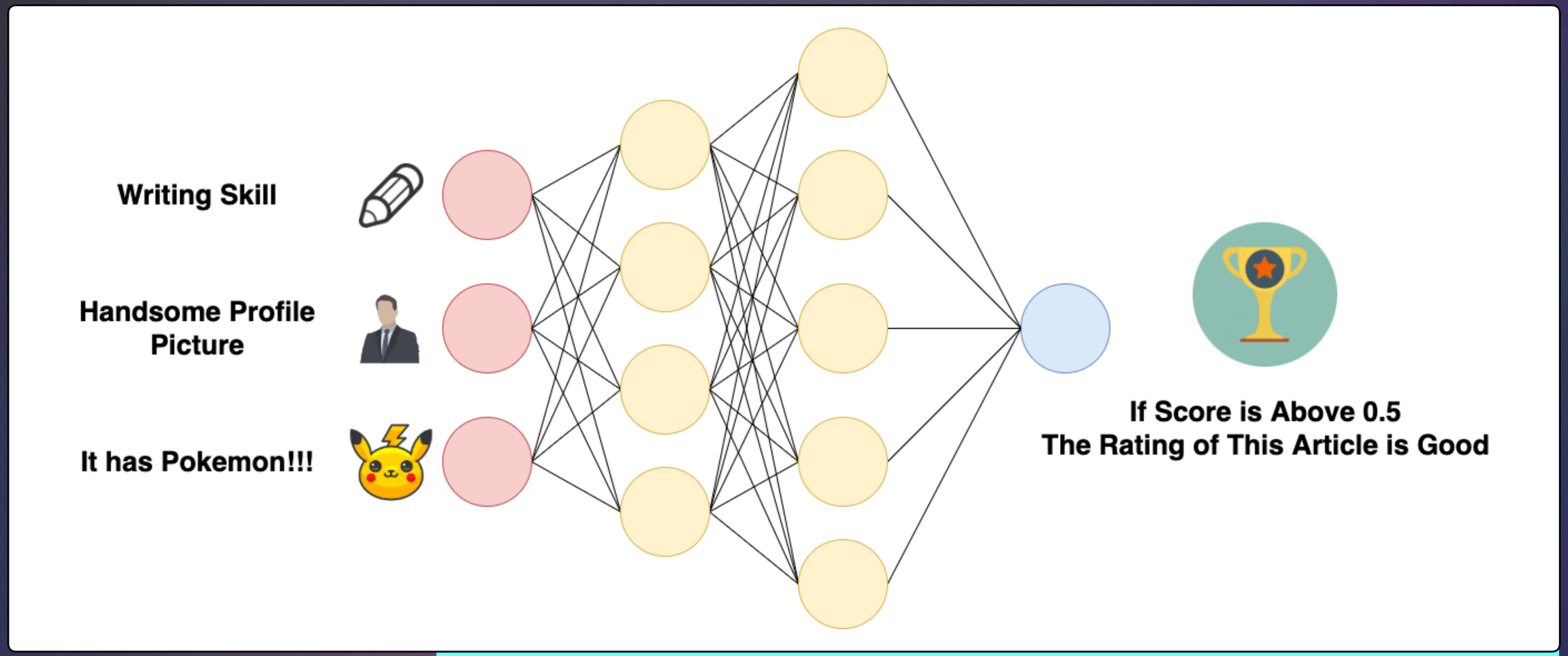
PYTHON: STOCHASTIC GRADIENT DESCENT

```
for _ in range(num_epochs):
    np.random.shuffle(data)
    for sample in data:
        grad = compute_gradient(sample, params)
        params = params - learning_rate * grad
```

- We can't utilize vectorization over one single example and becomes very slow. Also, the variance becomes large since we only use one example for each learning step.

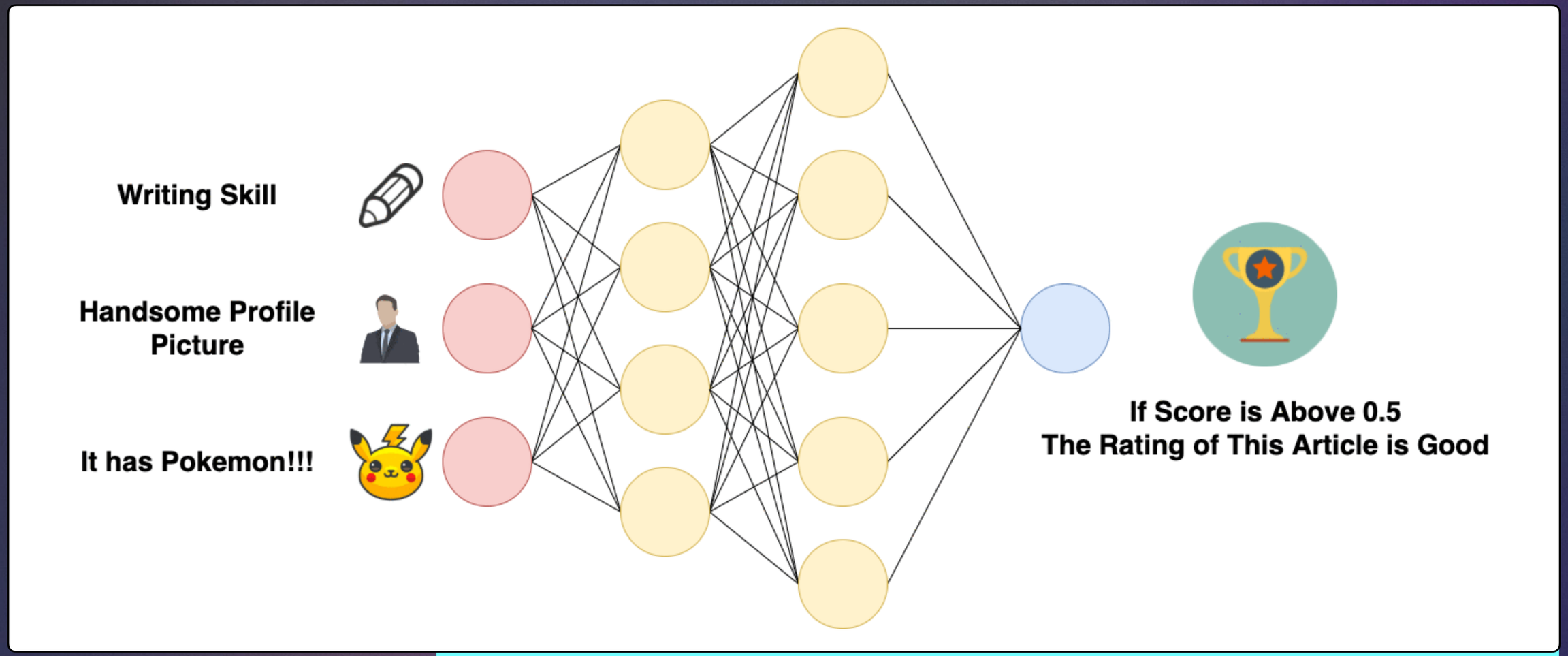
https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Forward and Back/propagations



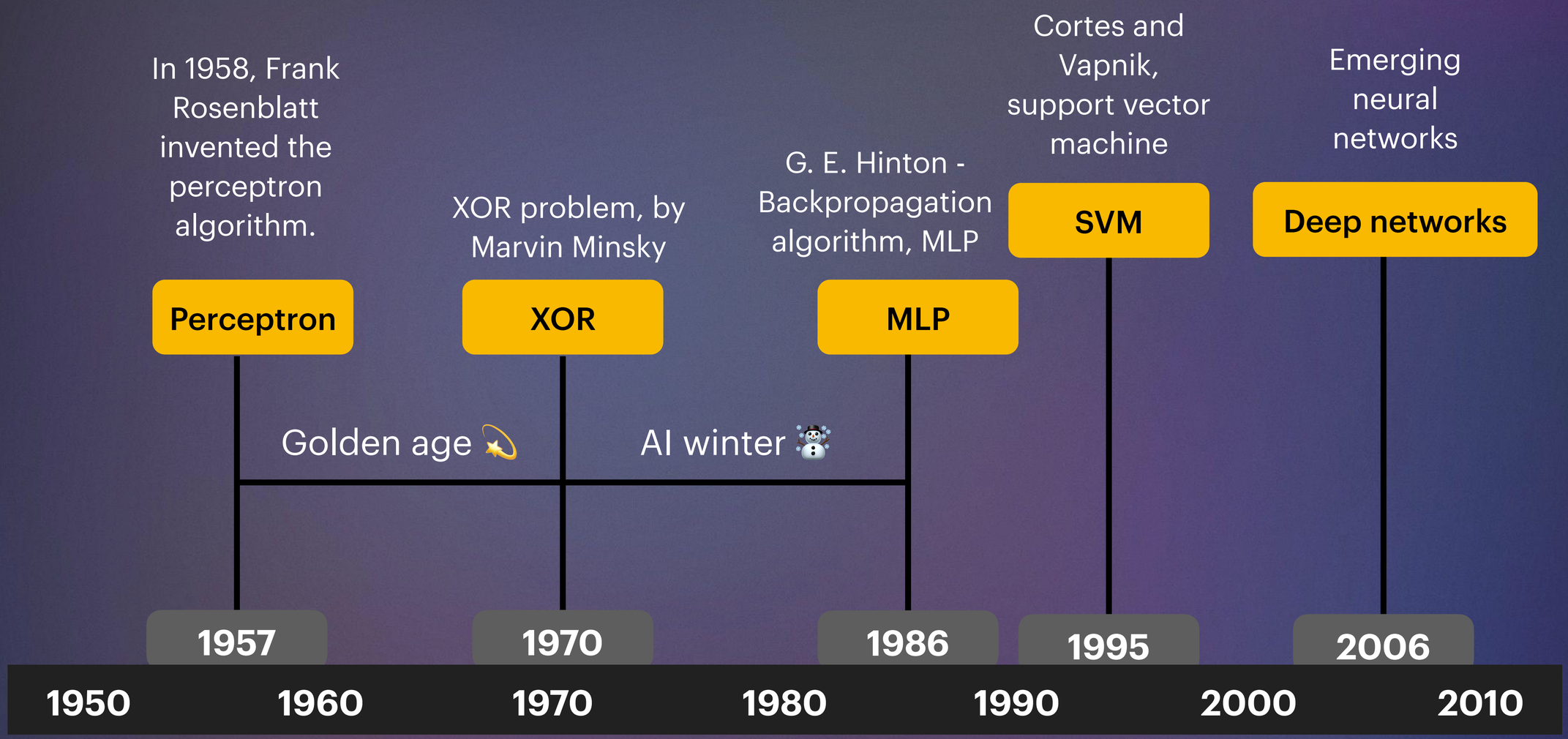
https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae

Forward and Back/propagations

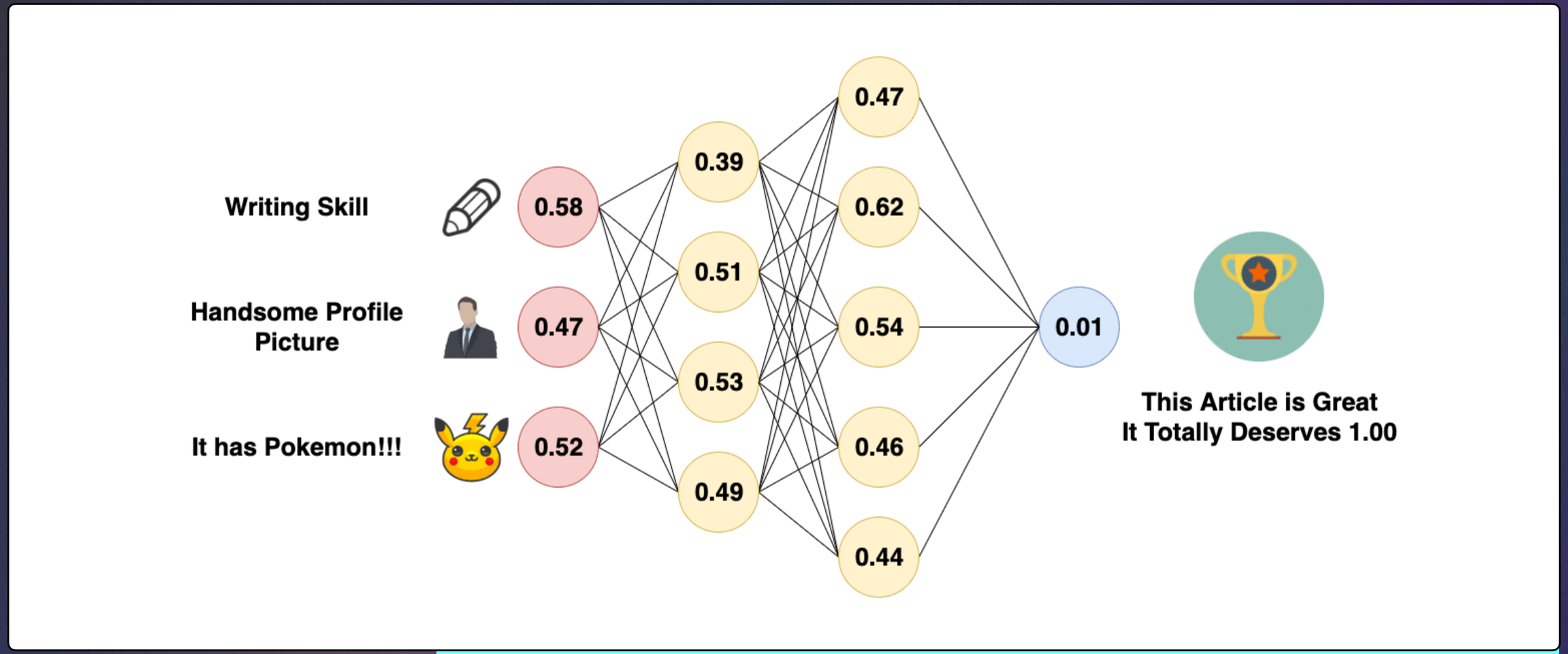


https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae

Development History of Neural Networks

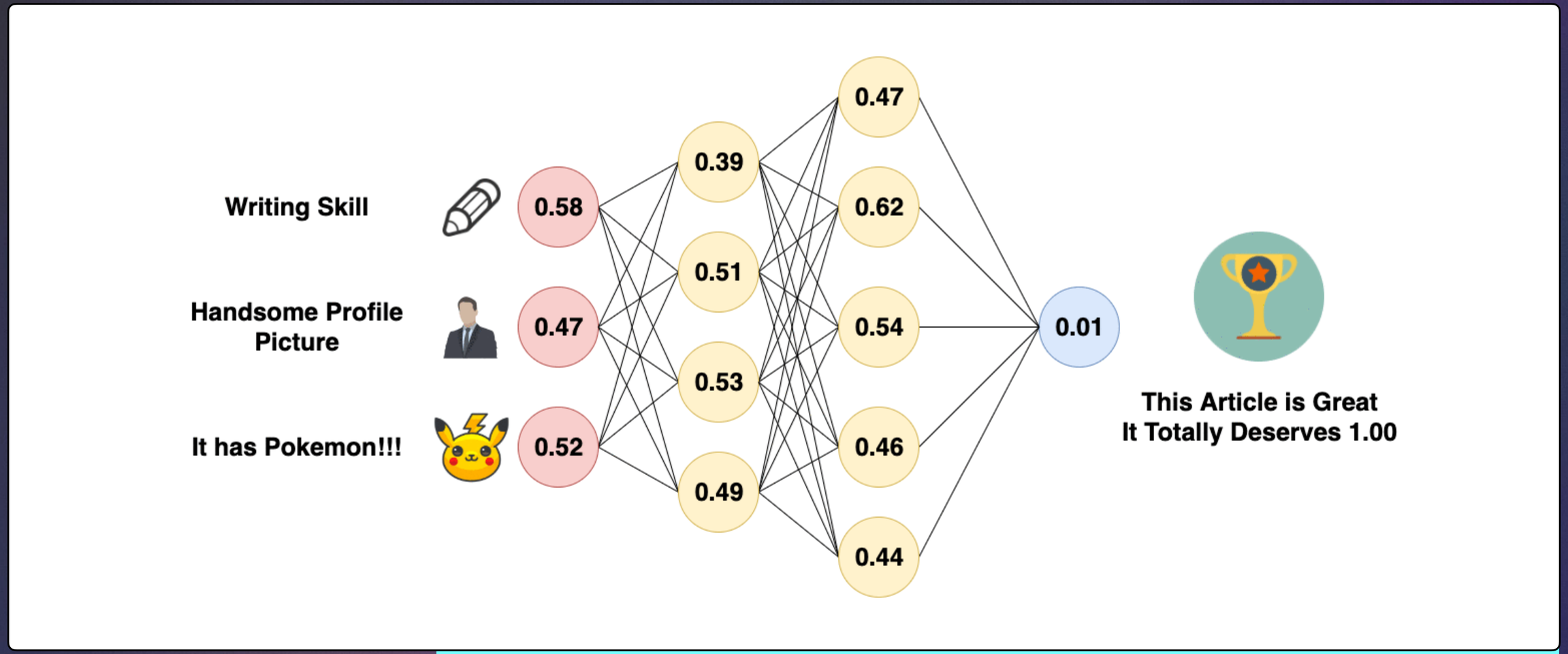


Backpropagation (1)



https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae

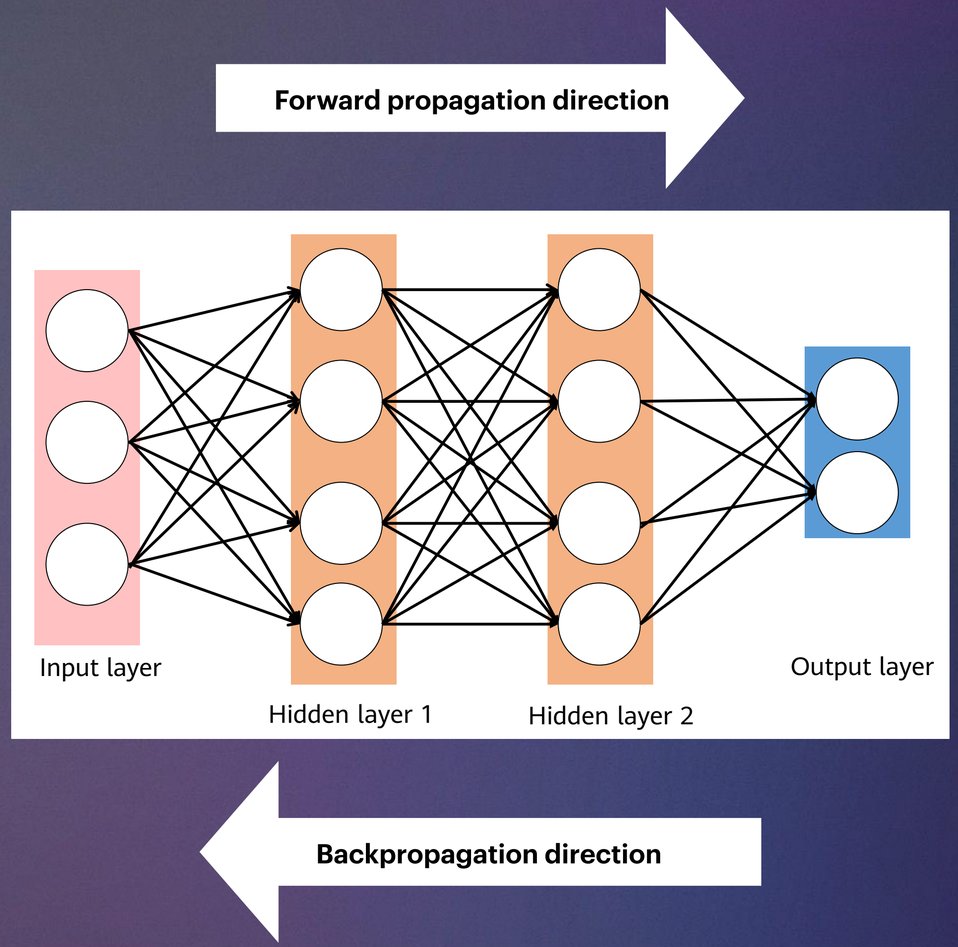
Backpropagation (1)



https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae

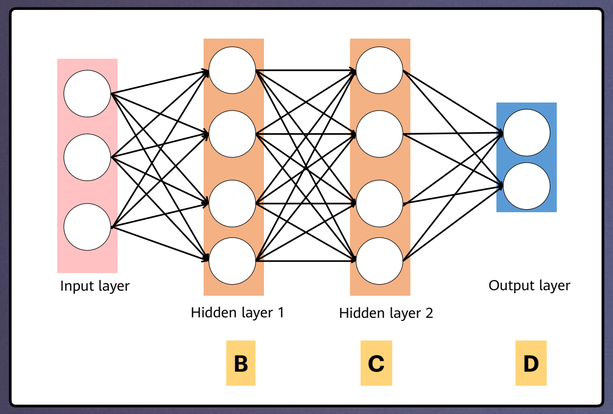
Backpropagation (2)

The basic idea is that backprop makes it possible to compute all the elements of the gradient in a single forward and backward pass through the network, rather than having to do one forward pass for every single parameter, as we'd have to using the naive approach. This is enabled by utilizing the chain rule in calculus, which lets us decompose a derivative as a product of its individual functional parts. By keeping track of the differences in a forward pass along every connection and storing them, we can calculate the gradient by taking the loss term found at the end of the forward pass, and "propagating the error backwards" through each of the layers. This makes a backward pass take roughly the same amount of work as a forwards pass.



Backpropagation (3)

- According to the following formulas, errors in the input, hidden, and output layers are accumulated to generate the error in the loss function.



- Let $\mathbf{w}_c$ be the weight coefficient between the hidden layer and the output layer, while $\mathbf{w}_b$ is the weight coefficient between the input layer and the hidden layer. D is the output layer set, C and B are the hidden layer set and input layer set respectively, and f is the activation function. Assume that the loss function is a quadratic cost function:

- **Output layer error:**

$$E = \frac{1}{2}\sum_{d\in D}\left(y_d - o_d\right)^2 .$$

- **Expanded hidden layer error:**

$$E = \frac{1}{2}\sum_{d\in D}\left(y_d - f(\text{net}_d)\right)^2 = \frac{1}{2}\sum_{d\in D}\left[y_d - f\left(\sum_{c\in C}\mathbf{w}_c y_c\right)\right]^2 .$$

- **Expanded input layer error:**

$$E = \frac{1}{2}\sum_{d\in D}\left[y_d - f\left(\sum_{c\in C}\mathbf{w}_c f(\text{net}_c)\right)\right]^2$$

$$= \frac{1}{2}\sum_{d\in D}\left[y_d - f\left(\sum_{c\in C}\mathbf{w}_c f\left(\sum_{b\in B}\mathbf{w}_b x_b\right)\right)\right]^2 .$$

Backpropagation (4)

- To minimize error E, the gradient descent iterative calculation can be used to solve $\mathbf{w}_c$ and $\mathbf{w}_b$, that is, calculating $\mathbf{w}_c$ and $\mathbf{w}_b$ to minimize error E.

$$\Delta \mathbf{w}_c = -\eta \frac{\partial E}{\partial \mathbf{w}_c}, c \in C$$

$$\Delta \mathbf{w}_b = -\eta \frac{\partial E}{\partial \mathbf{w}_b}, b \in B$$

- If there are multiple hidden layers, chain rules are used to take a derivative for each layer to obtain the optimized parameters by iteration.
- For a neural network with any number of layers, the arranged formula for training is as follows:

$$\Delta w_{j,k}^{l} = -\eta \delta_k^{l+1} f_j(z_j^l)$$

$$\delta_j^l = \begin{cases} f_j'\left(z_j^l\right)\left(y_j - f_j(z_j^l)\right) & \text{if } l \in \text{outputs}, \\ \sum_k \delta_k^{l+1} w_{j,k}^l f_j(z_j^l) & \text{otherwise} \end{cases}.$$

- The BP algorithm is used to train the network as follows:
  - ➡ Takes out the next training sample, inputs it to the network, and obtains the actual output.
  - ➡ Calculates output layer δ according to the output layer error formula.
  - ➡ Calculates δ of each hidden layer from output to input by iteration according to the hidden layer error propagation formula.
  - ➡ According to the δ of each layer, the weight values of all the layer are updated.

References

- Gene Kogan. How neural networks are trained. https://ml4a.github.io/ml4a/how_neural_networks_are_trained/. 2016, Accessed on Feb 2021.
- HUAWEI. Deep Learning Overview. 2020, Accessed on Feb 2021.
- Imad Dabbura. Gradient Descent Algorithm and Its Variants. Available at: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3. Accessed on Feb 2021.
- LI, Hao et al. Visualizing the loss landscape of neural nets. arXiv preprint arXiv:1712.09913, 2017.
- Mohammed Terry-Jack. Introduction to Deep Learning: Feed Forward Neural Networks FFNNs. https://medium.com/@b.terryjack/introduction-to-deep-learning-feed-forward-neural-networks-ffnns-a-k-a-c688d83a309d. 2019, Accessed on Feb 2021.
- Riti Dass. The complete list to make you an AI Pro. https://medium.com/predict/the-complete-list-to-make-you-an-ai-pro-be83448720b8. 2018, Accessed on Feb 2021.
- Shen Huang. Understanding AlphaGo: how AI thinks and learns (Advanced). https://towardsdatascience.com/understanding-alphago-how-ai-thinks-and-learns-advanced-d70780744dae. 2020, Accessed on Feb 2021.

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Ceará

HUAWEI

Thank you for your attention!

🙏☺️

Prof. Me. Saulo A. F. Oliveira
saulo.oliveira@ifce.edu.br