Artificial Neural Networks

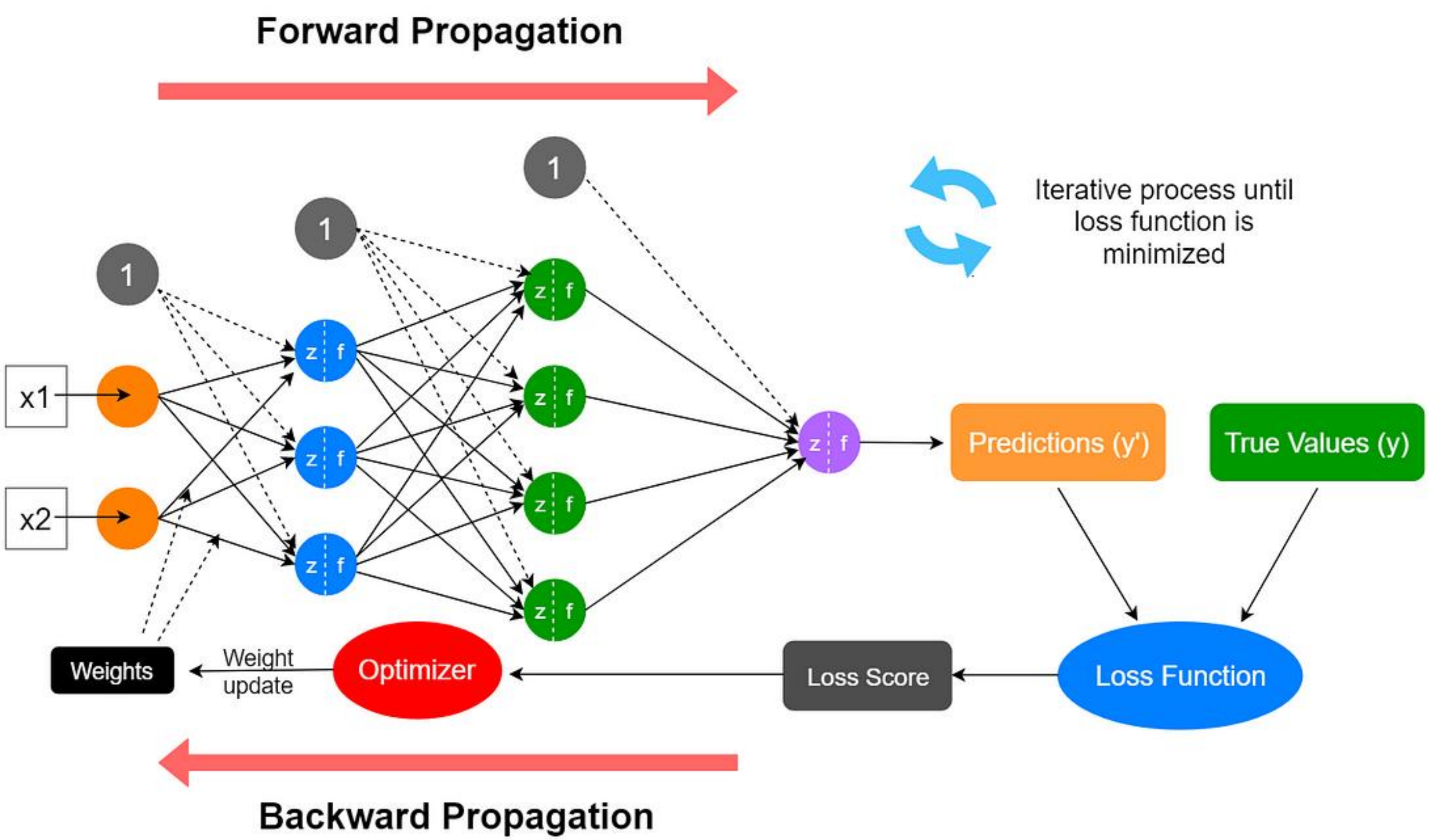


Image source

Outline

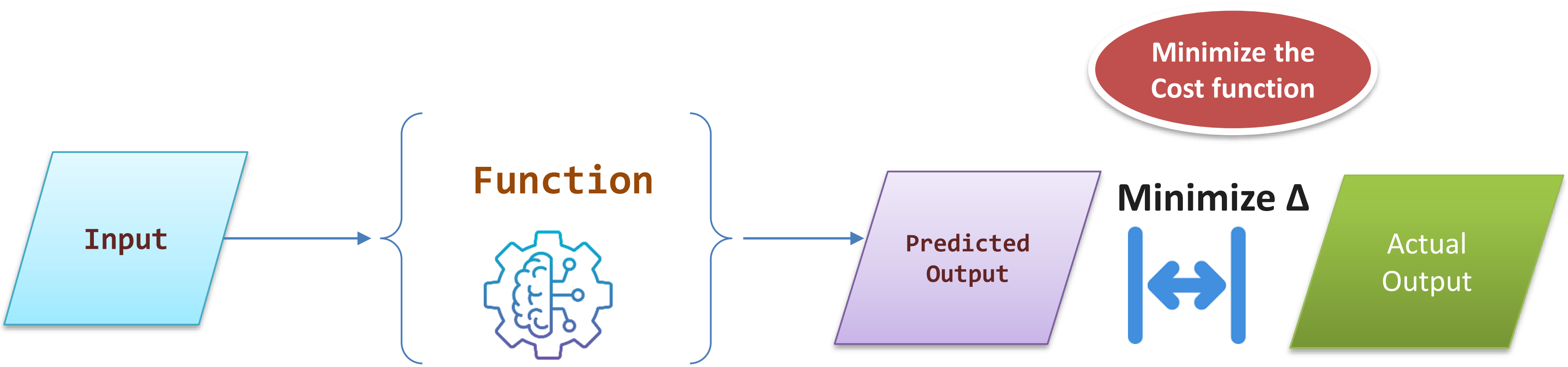
- Introduction to Artificial Neural Networks
- NN Architectures
- Neuron: The structural building block of NN
- Building a NN with Neurons
- Training a NN
- Training in Practice

Acknowledgement : some slides are adapted from Lecture of MIT intro deep learning

Introduction to Artificial Neural Networks

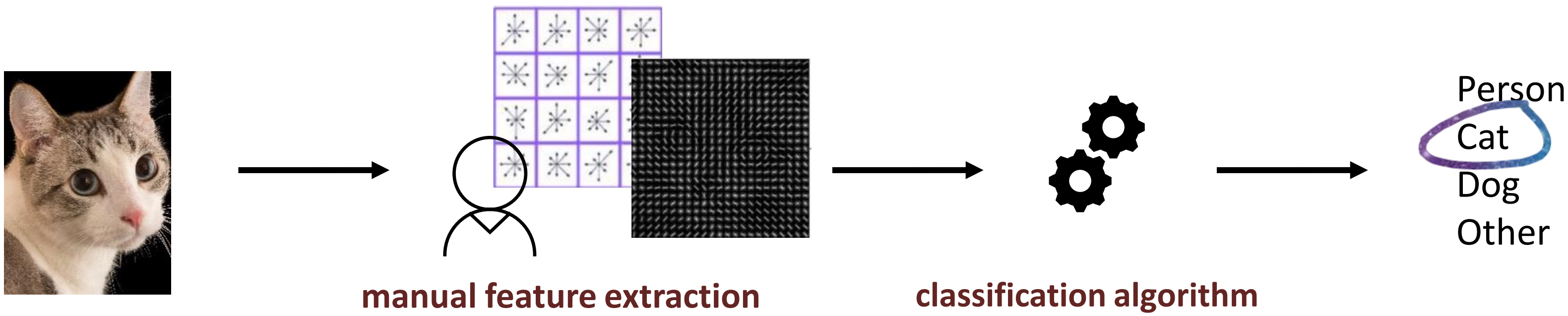
ML: learn a Function that minimizes the cost

- Start with random function parameters
- Repeat intelligent guessing/approximation of the Function parameters such that the difference between the Predicted Output the Actual Output is reduced
  - i.e., minimize a Cost function a.k.a loss, or error function

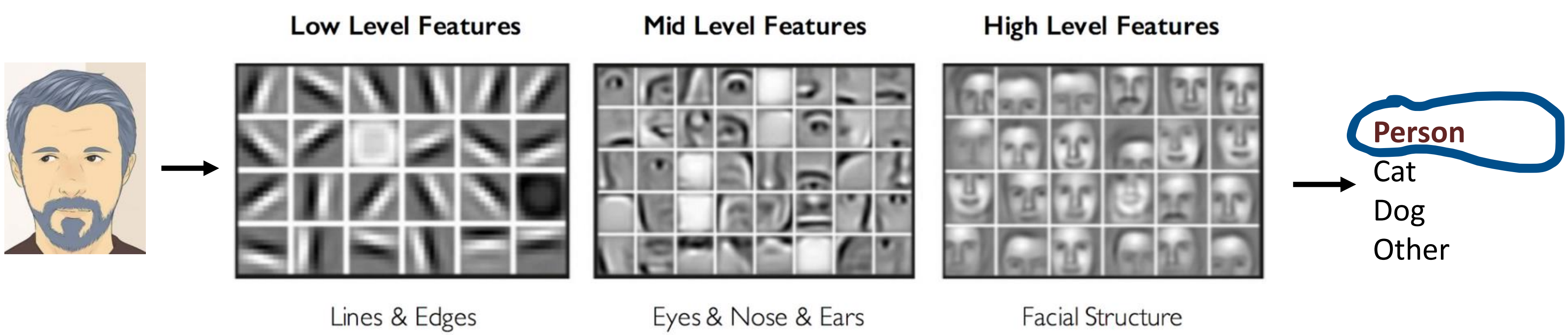


Why Deep learning?

In traditional **Machine Learning (ML)**, hand engineered features are time consuming, brittle, and not scalable in practice

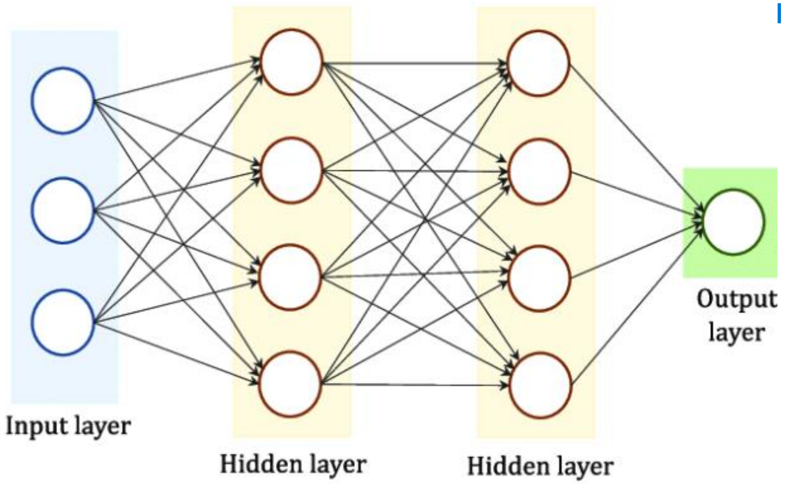


Deep Learning (DL) enables learning the underlying features directly from data using many layers of abstraction



feature learning + classification

Characteristics of Deep Learning

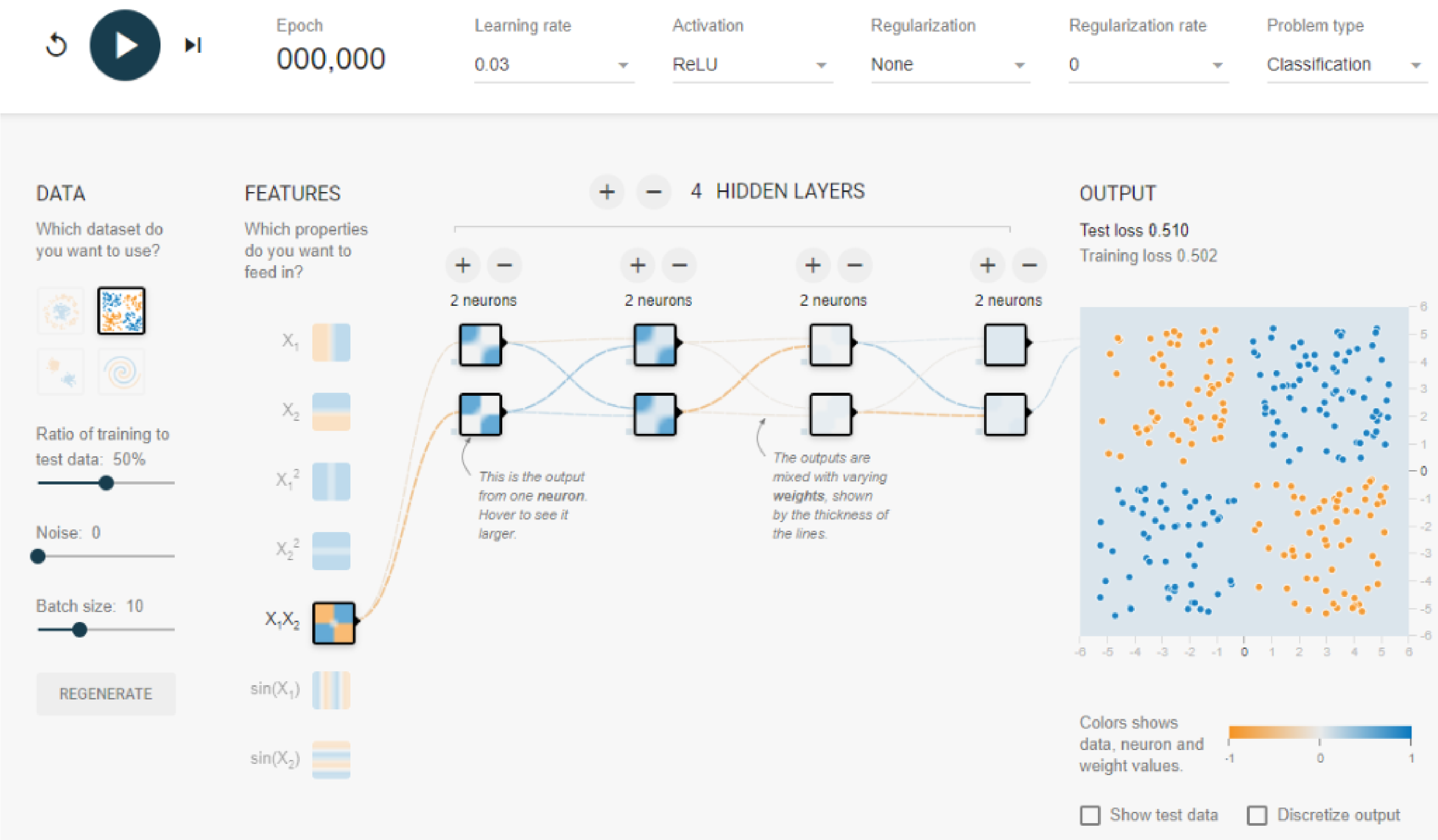


- Massive amounts of training data
- Excels with raw, unstructured data
- Automatic feature extraction
- Computationally expensive

Biological Inspirations

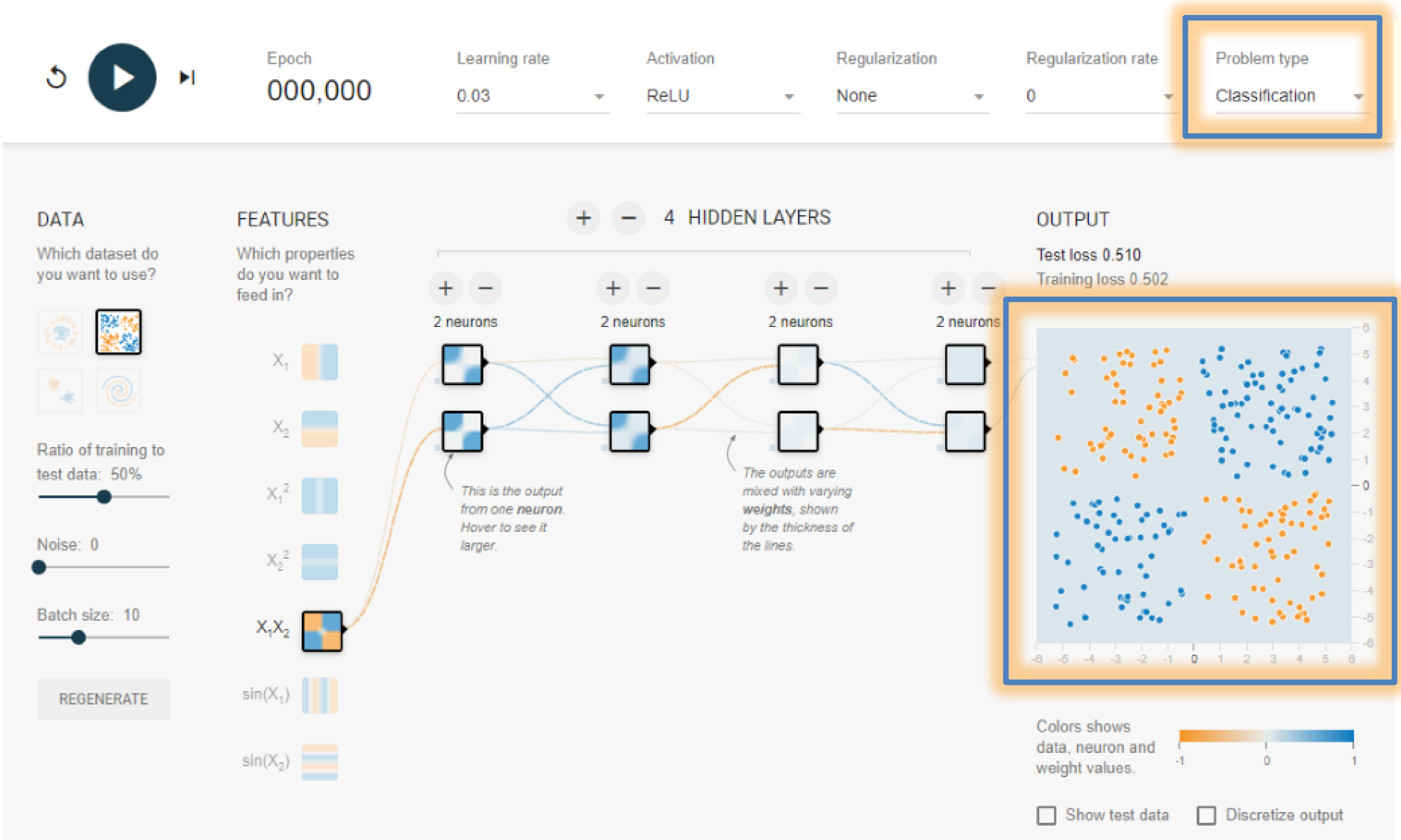
- An artificial neural network (ANN), or simply neural network (NN), serves as a computational model inspired by the workings of the human brain
  - The human brain comprises a densely interconnected network of nerve cells, known as neurons, which serve as the fundamental units for processing information
  - With nearly 10 billion neurons and an astonishing 60 trillion synapses linking them, the human brain exemplifies a remarkable level of interconnectivity
  - Through the simultaneous activation of multiple neurons, the brain achieves computational tasks at a speed that surpasses even the most advanced computers available today
  - Within the brain, various types of neurons exist, including motor neurons and visual cells, each characterized by unique branching structures
- A NN is composed of a multitude of elementary processors known as neurons, akin to the biological neurons found in the brain
- These neurons are interconnected through **weighted links**, facilitating the transmission of signals from one neuron to another

Fundamental Concepts - Visualizing a Neural Network



https://playground.tensorflow.org/

Understand the data and the goal

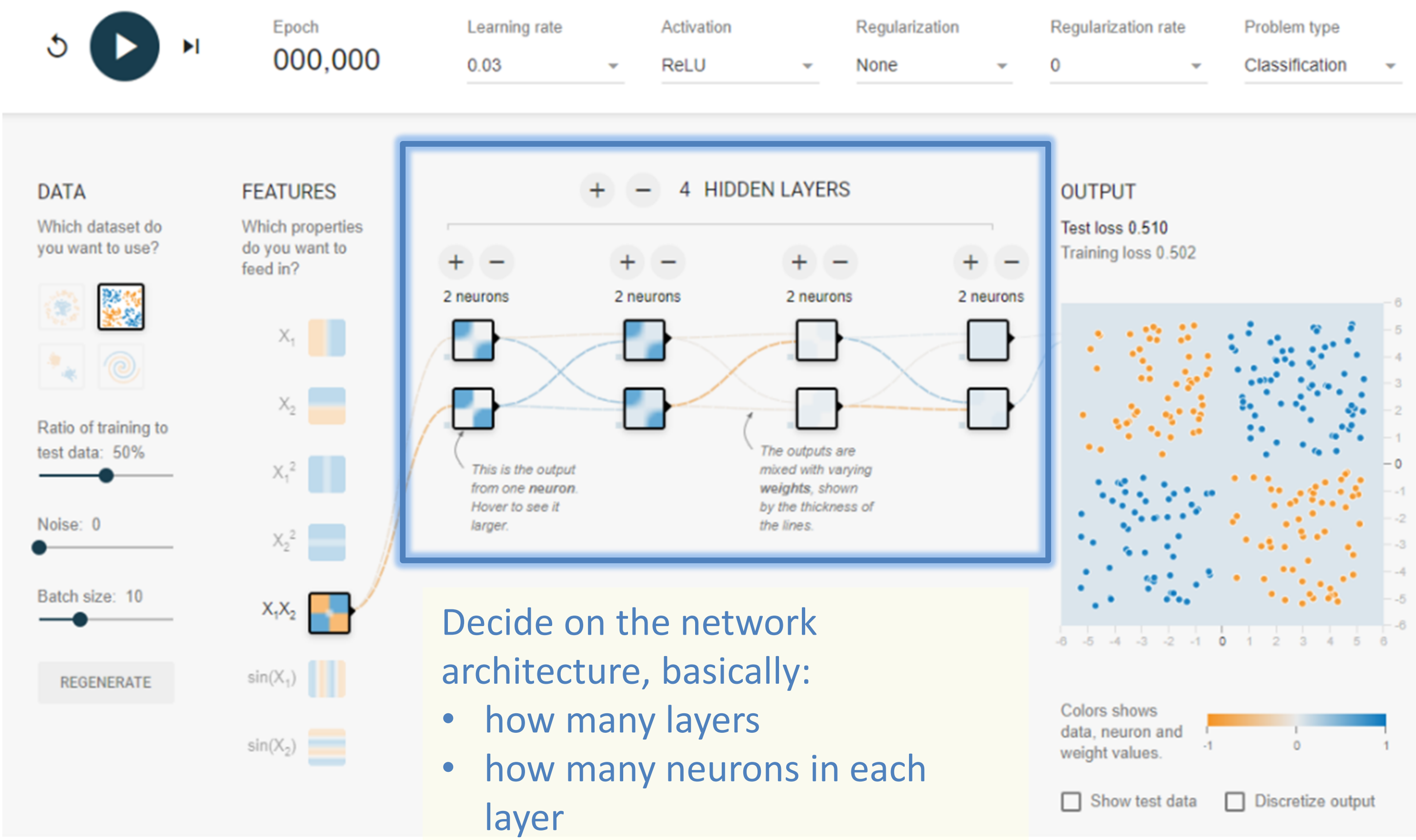


Classification or Regression

We have two classes of data (**orange** and blue).

We *want* to train a model that given some input data, it can classify it as **orange (-1)** or **blue (+1)**

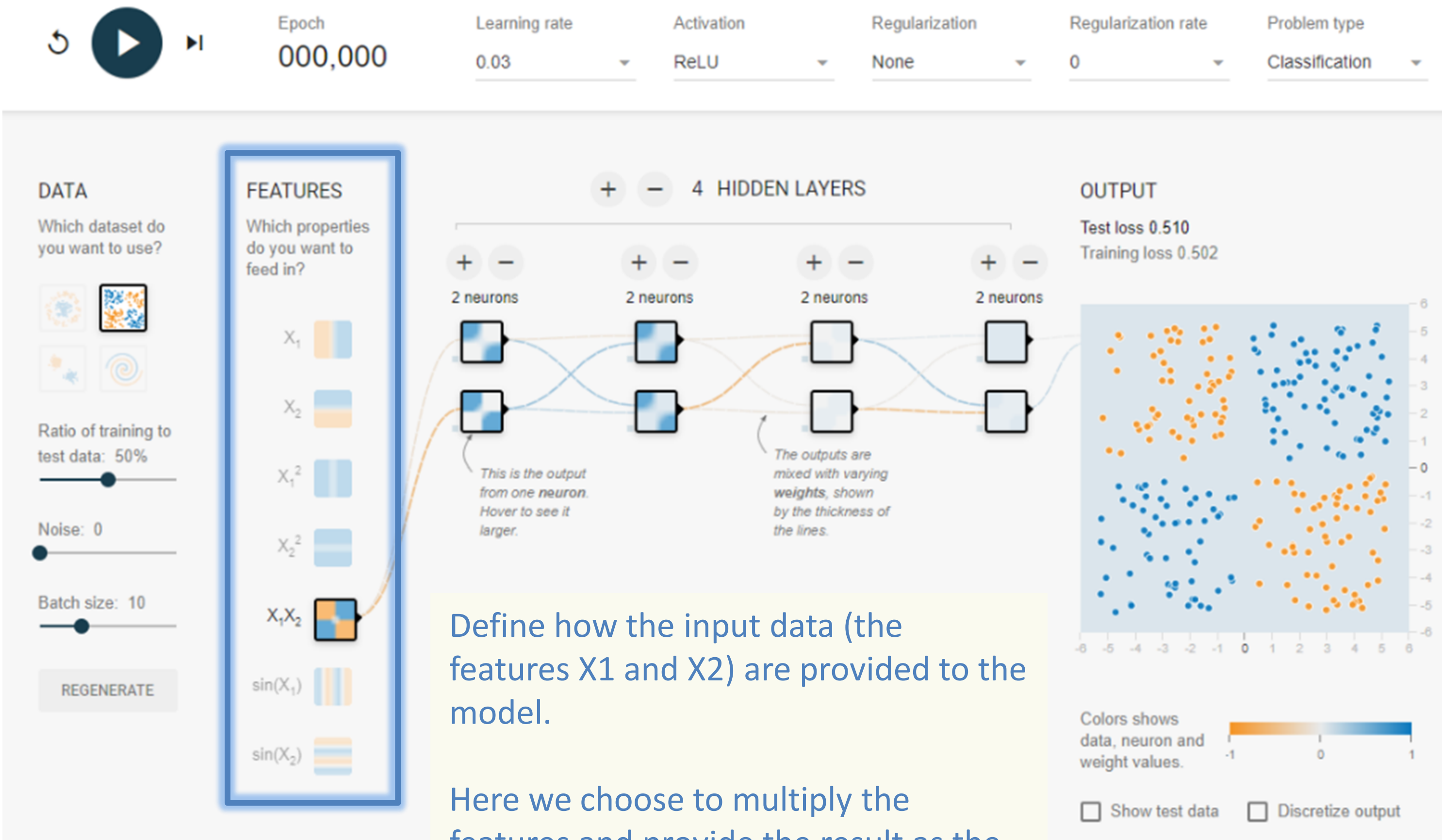
Define a network architecture



Decide on the network architecture, basically:

- how many layers
- how many neurons in each layer
- how the neurons in each layer are connected (fully or partially)

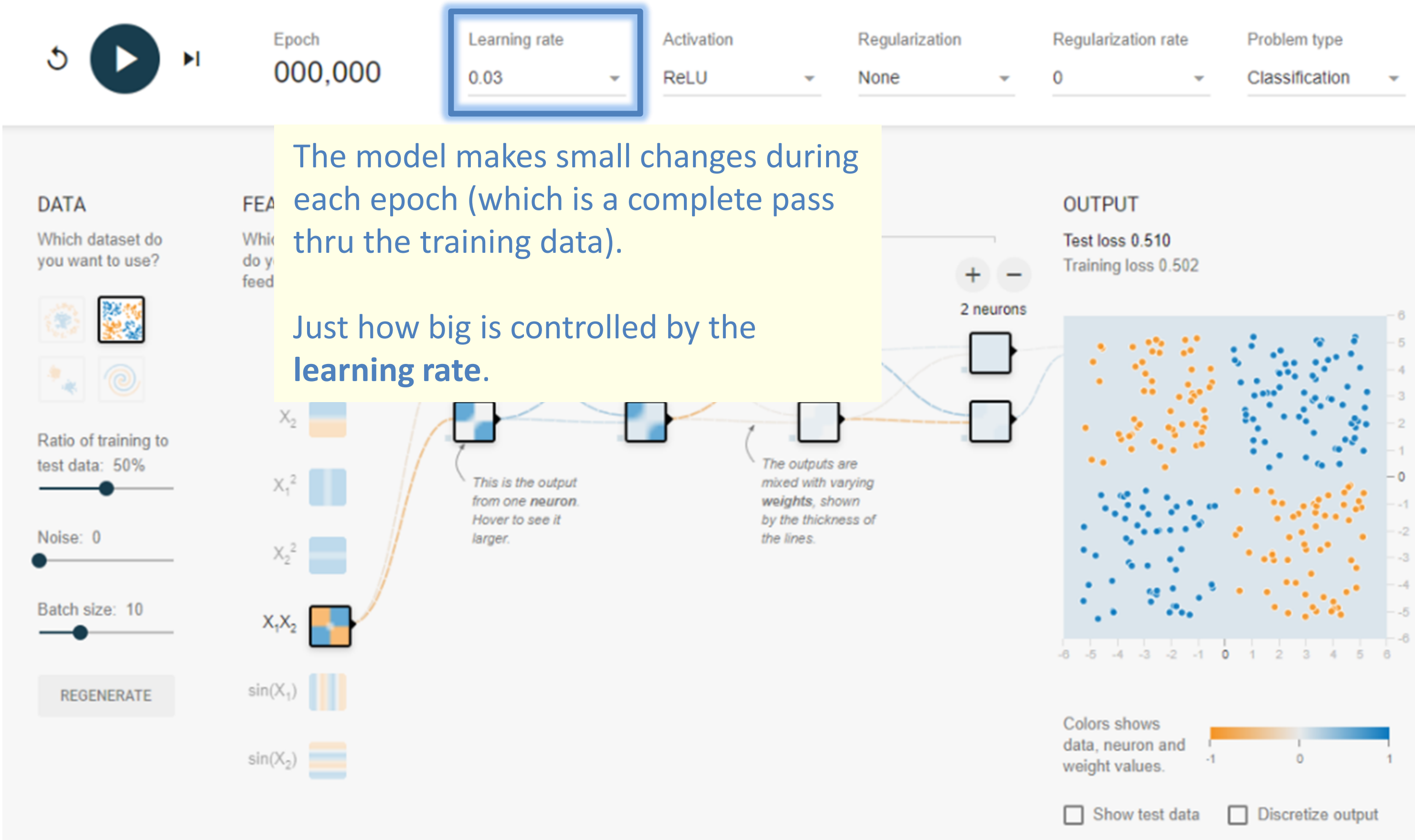
Define shape of input data



Define how the input data (the features X1 and X2) are provided to the model.

Here we choose to multiply the features and provide the result as the input.

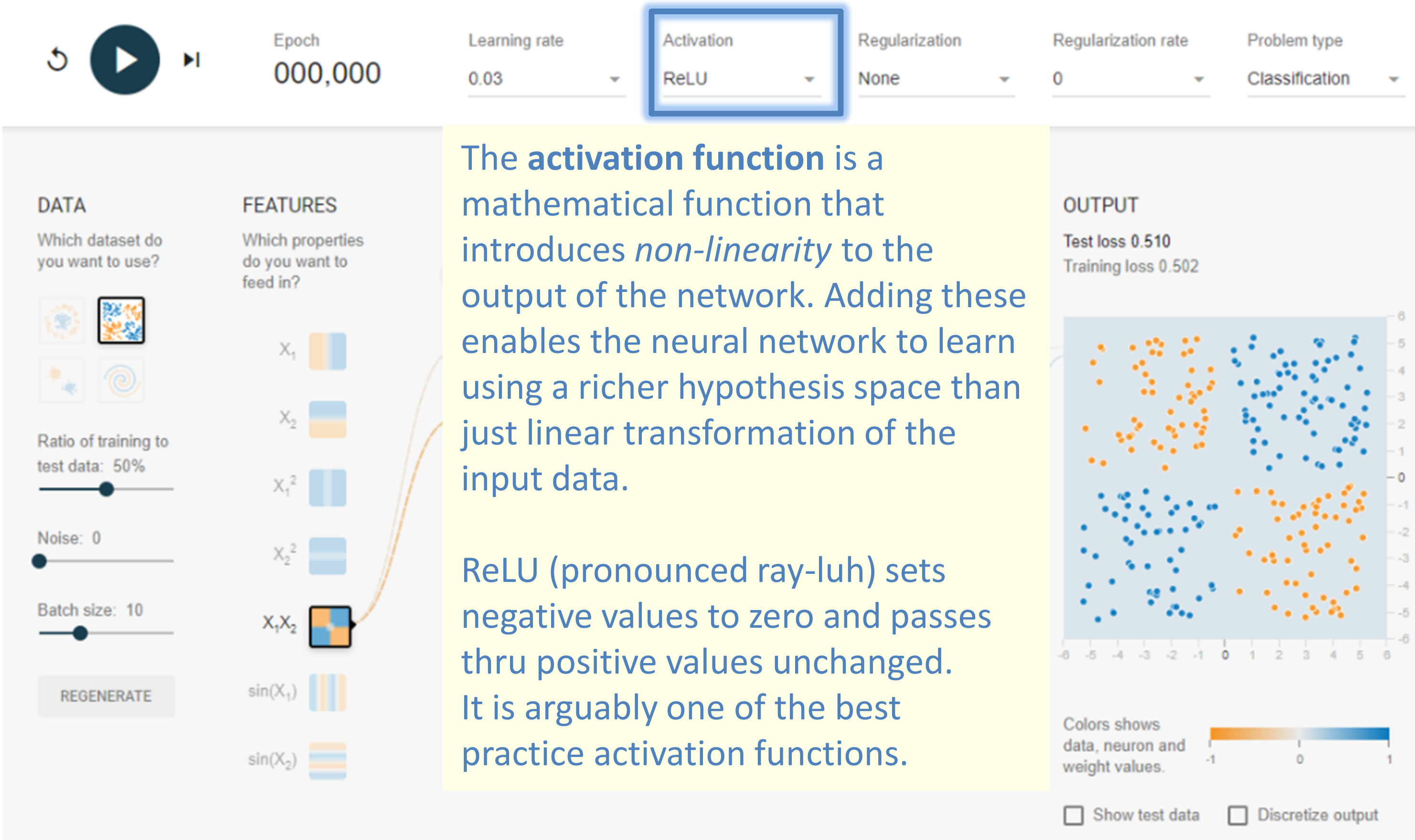
Set parameters – Learning rate



The model makes small changes during each epoch (which is a complete pass thru the training data).

Just how big is controlled by the **learning rate**.

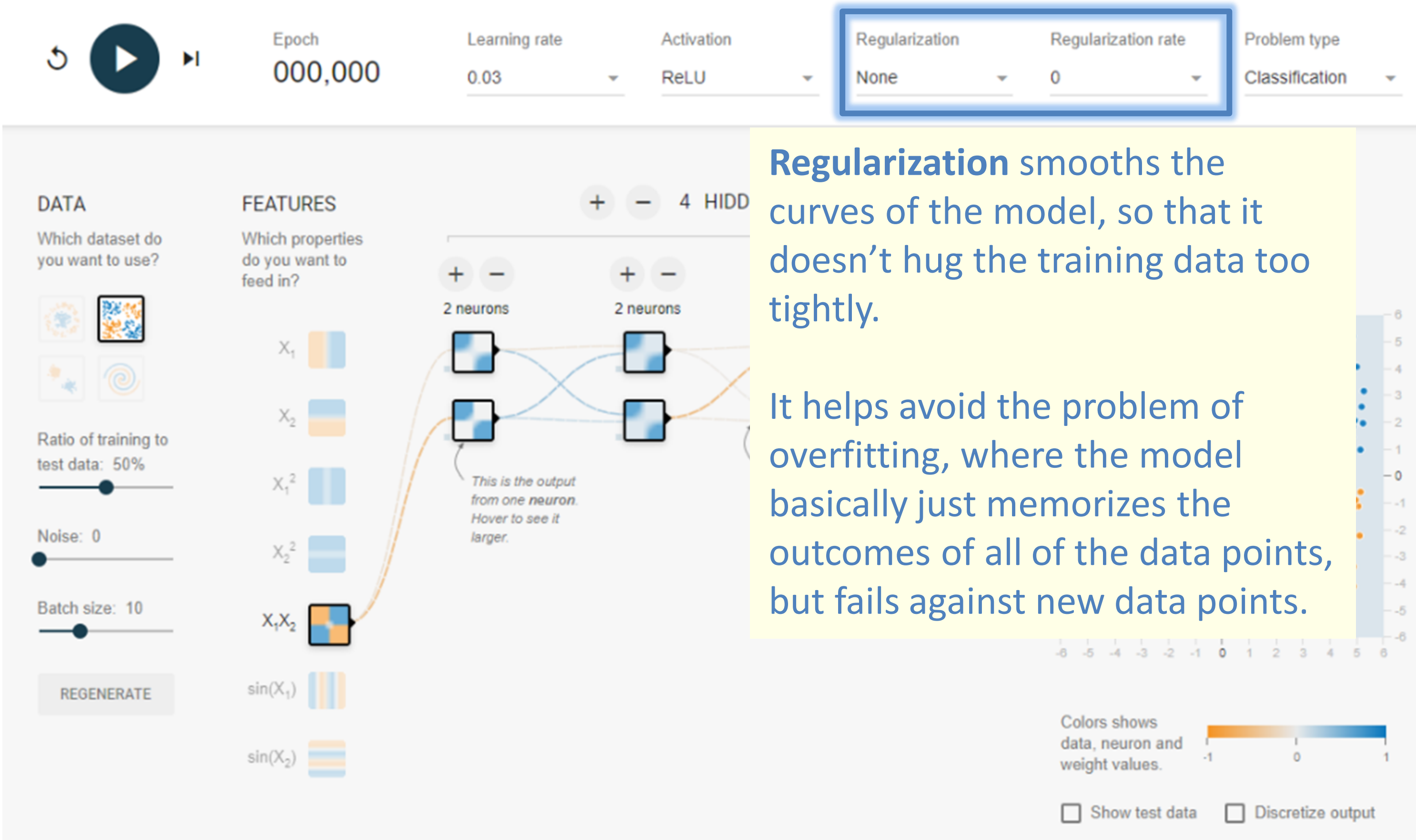
Set parameters – Activation function



The **activation function** is a mathematical function that introduces *non-linearity* to the output of the network. Adding these enables the neural network to learn using a richer hypothesis space than just linear transformation of the input data.

ReLU (pronounced ray-luh) sets negative values to zero and passes thru positive values unchanged. It is arguably one of the best practice activation functions.

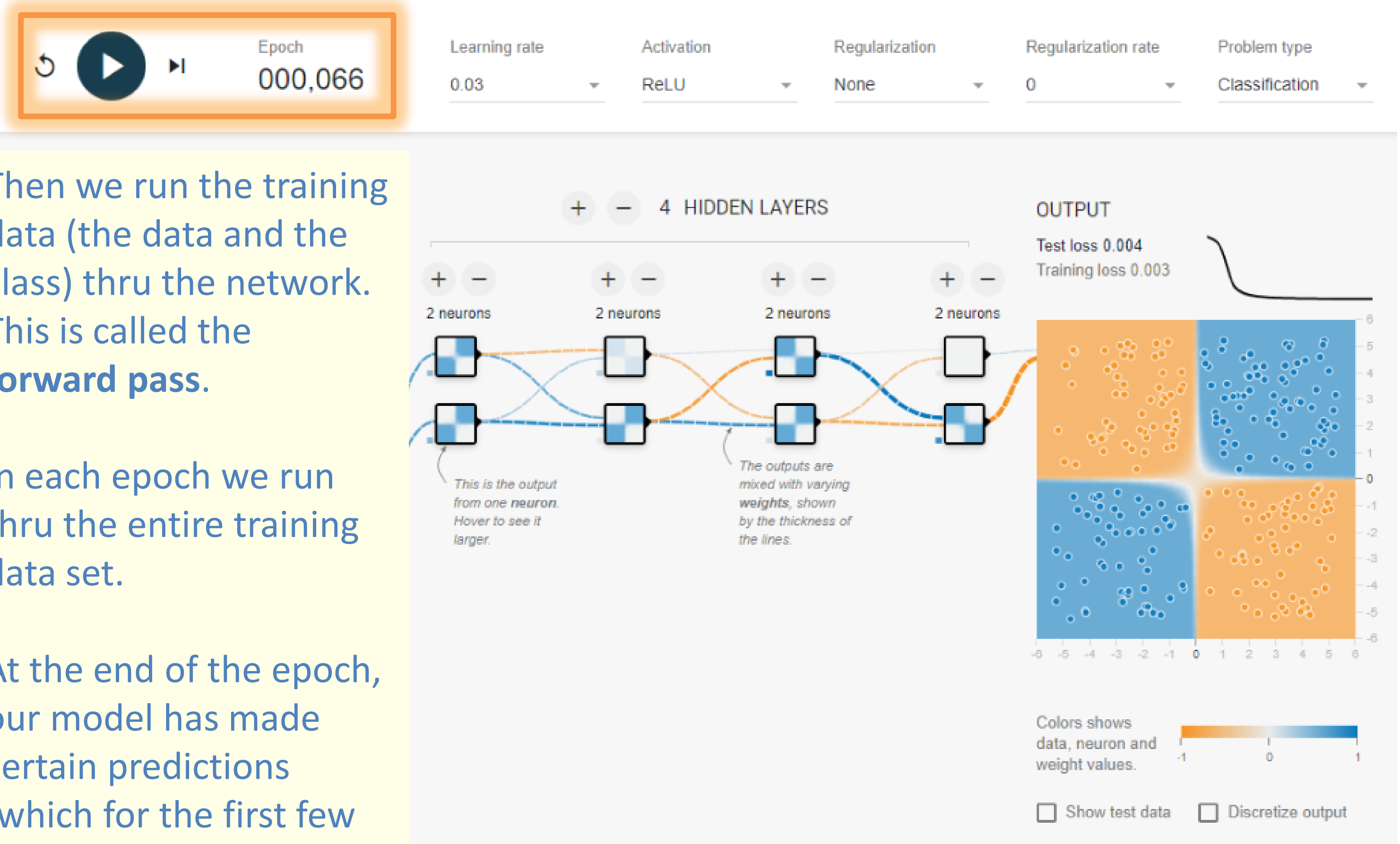
Set parameters - Regularization



Regularization smooths the curves of the model, so that it doesn't hug the training data too tightly.

It helps avoid the problem of overfitting, where the model basically just memorizes the outcomes of all of the data points, but fails against new data points.

Training – The forward pass

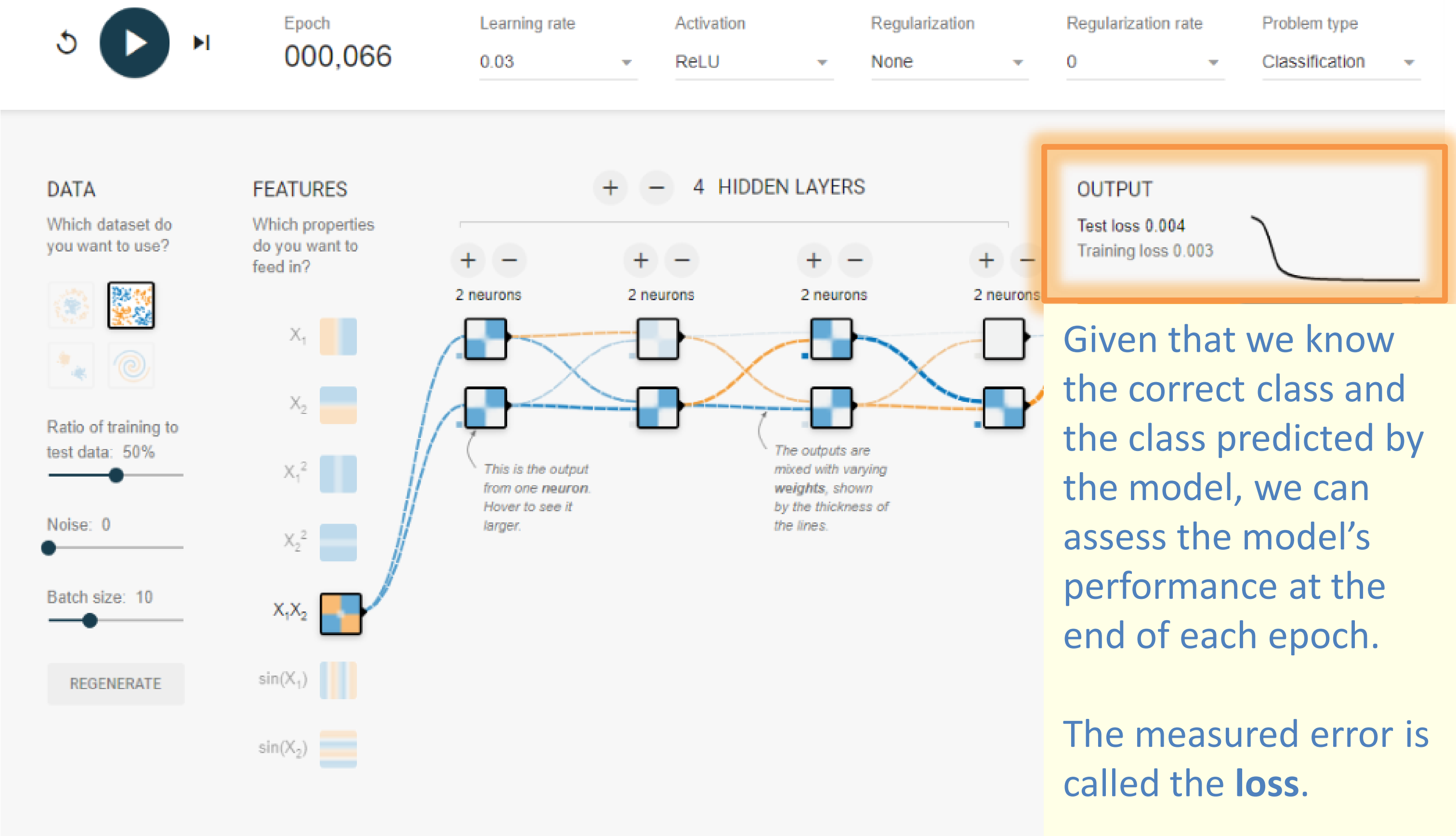


Then we run the training data (the data and the class) thru the network. This is called the **forward pass**.

In each epoch we run thru the entire training data set.

At the end of the epoch, our model has made certain predictions (which for the first few epochs are almost certainly wrong because they are random).

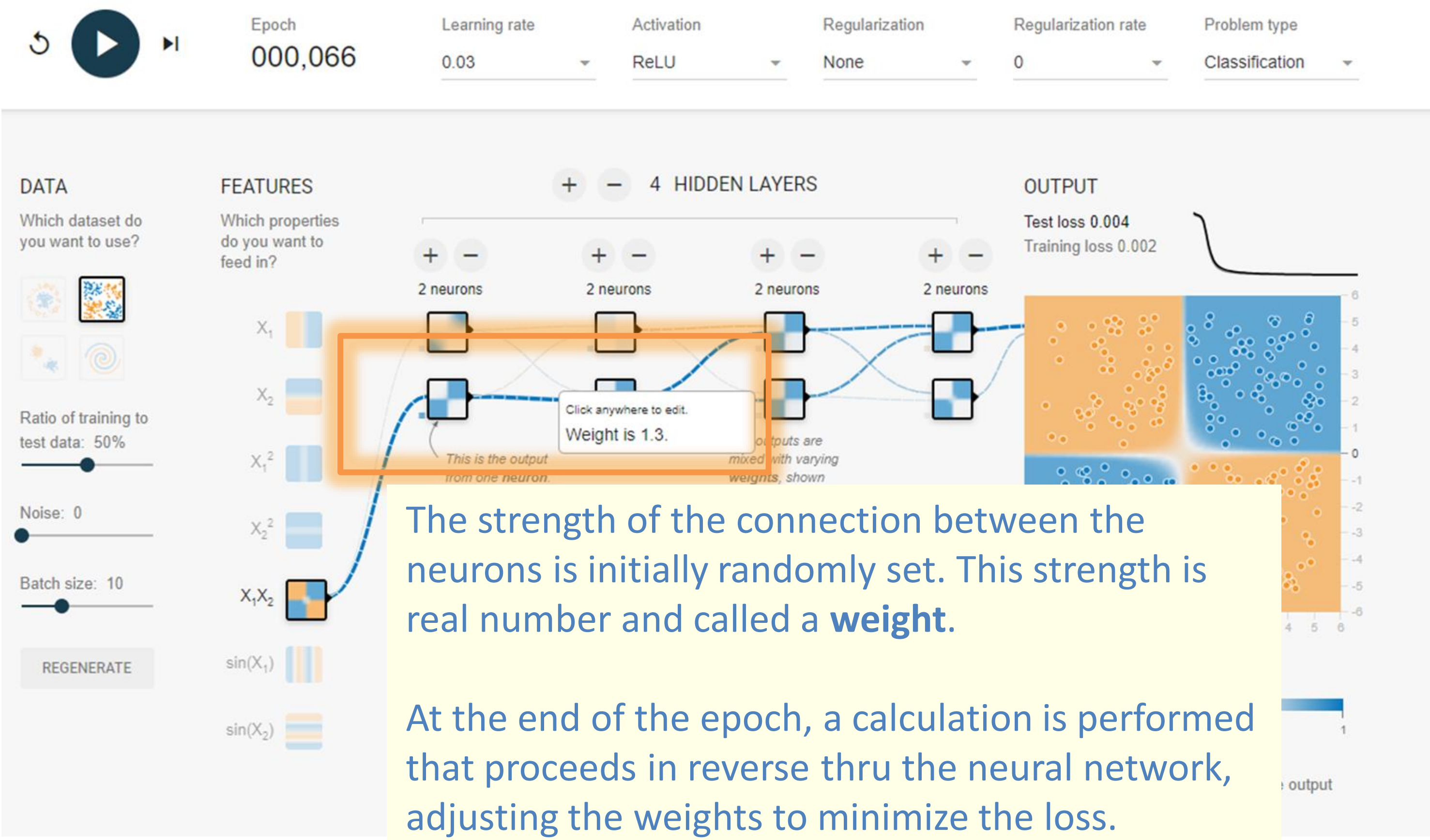
Training – Evaluate loss



Given that we know the correct class and the class predicted by the model, we can assess the model's performance at the end of each epoch.

The measured error is called the **loss**.

Training – The backwards pass



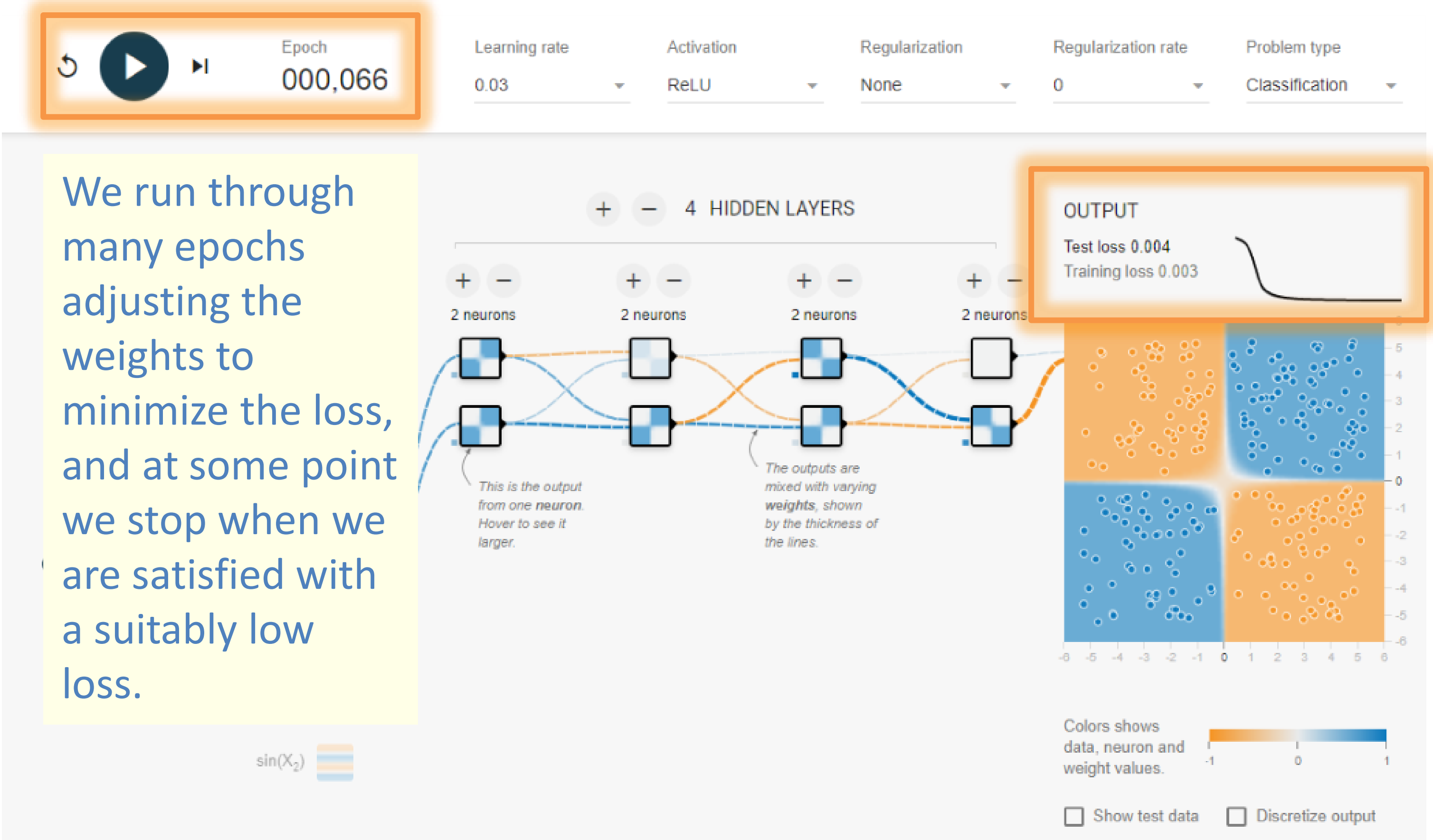
The strength of the connection between the neurons is initially randomly set. This strength is real number and called a **weight**.

At the end of the epoch, a calculation is performed that proceeds in reverse thru the neural network, adjusting the weights to minimize the loss.

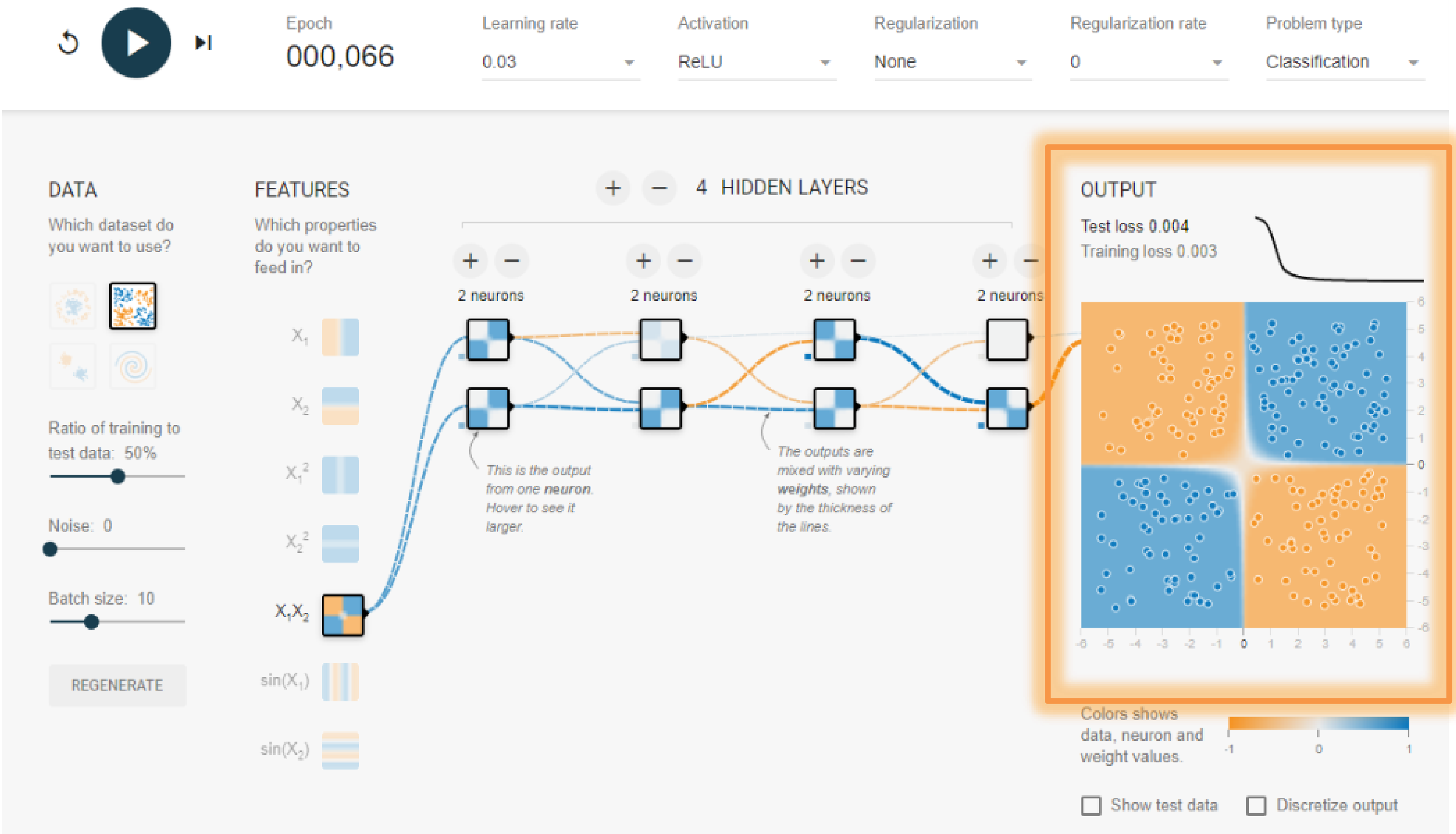
This is called the **backwards pass**.

Training – Run *lots* of epochs

We run through many epochs adjusting the weights to minimize the loss, and at some point we stop when we are satisfied with a suitably low loss.



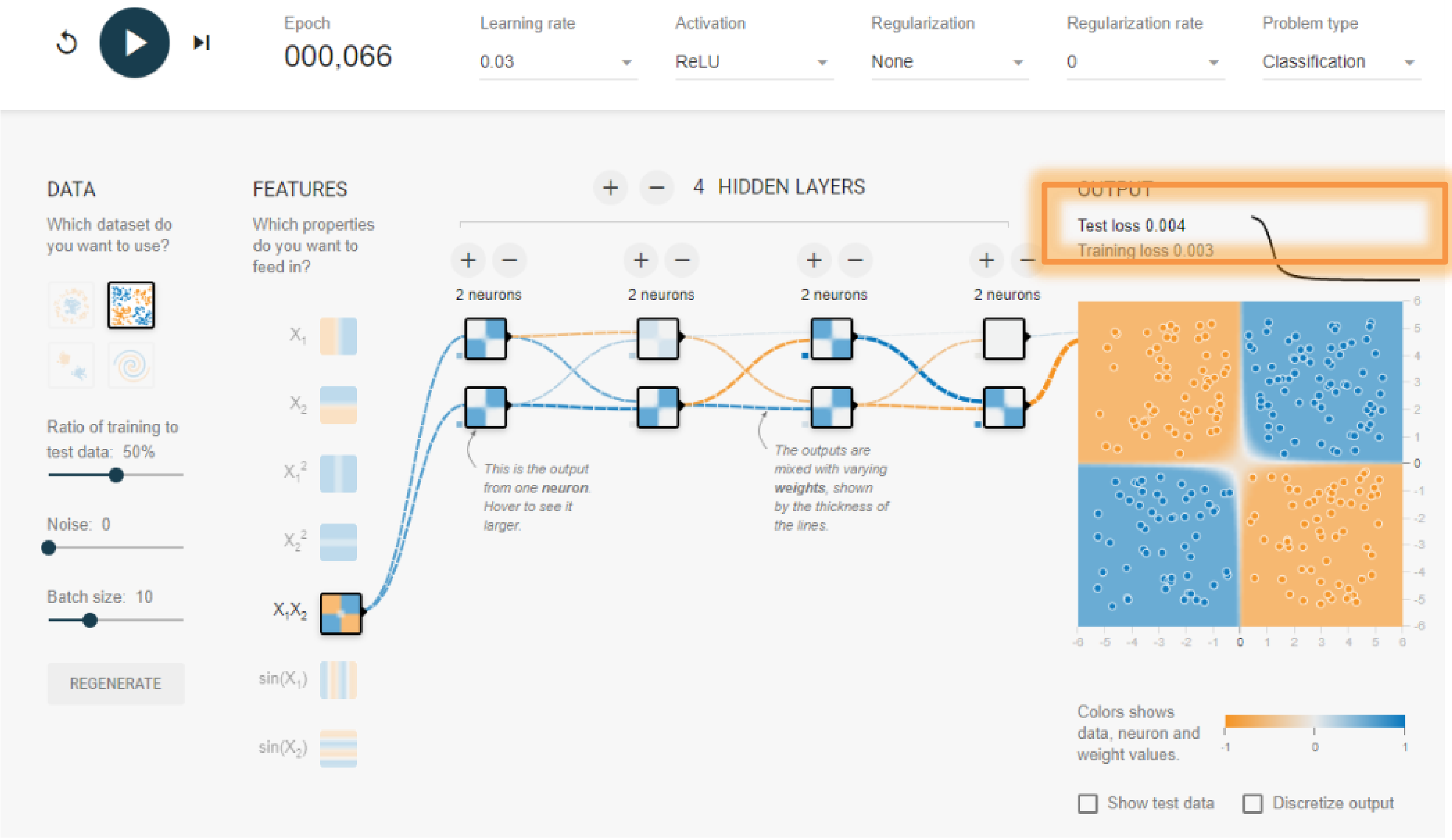
Training – Model complete!



Now we have a trained model that can classify the training data reasonably well.

But how well does it perform against data it has not seen?

Model Evaluation



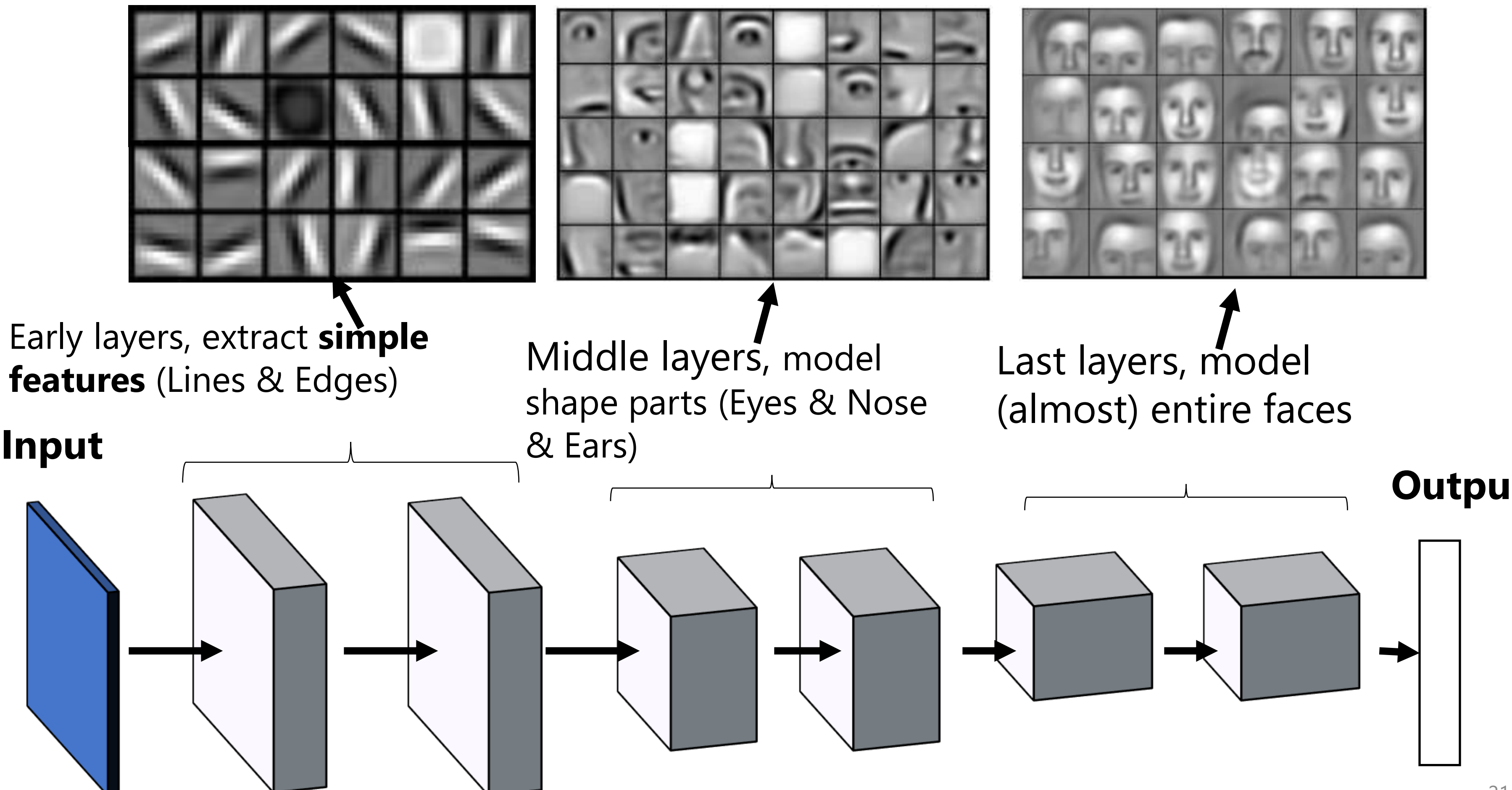
That's were the test data set comes in.

We run all of the test data thru the model and measure the result.

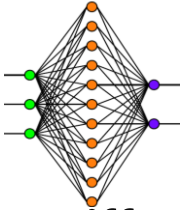
The **Test Loss** tells us the error of the model against this unseen data.

NN Visualization

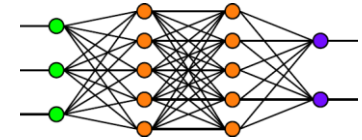
- NN **extract patterns** from data
- Visualization of a deep neural network trained with face images:
  - Credit for feature visualizations: Honglak Lee, Roger Grosse, Rajesh Ranganath and Andrew Y. Ng., ICML 2009



NN Architectures



NN Architectures (1 of 2)



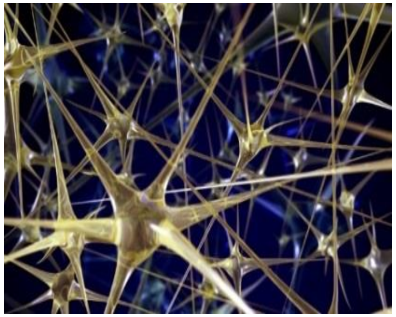
Different NN architectures are just different ways to connect neurons

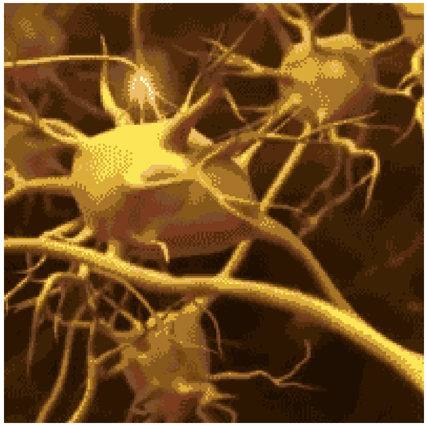
- **Feedforward Neural Networks (FNN) a.k.a. Multilayer Perceptrons (MLP)**
  - **Architecture**: consist of multiple layers of neurons, each connected to the next layer without any cycles
  - **Use Cases**:
    - Classification tasks like image recognition, sentiment analysis, and spam detection
    - Regression tasks such as predicting house prices, stock prices, etc.
- **Convolutional Neural Networks (CNN)**
  - **Architecture**: Specifically designed for processing grid-like data, such as images or videos
  - **Use Cases**:
    - Medical image analysis for disease detection, like identifying tumors in MRI scans
    - Image recognition tasks, object detection, facial recognition, and video analysis
- **Recurrent Neural Networks (RNN)**
  - **Architecture**: Designed to work with sequential data by maintaining a "memory" of previous inputs
  - **Use Cases**:
    - Natural Language Processing (NLP) tasks such as language translation, sentiment analysis, and speech recognition
    - Time-series analysis for tasks like stock market prediction, and weather forecasting

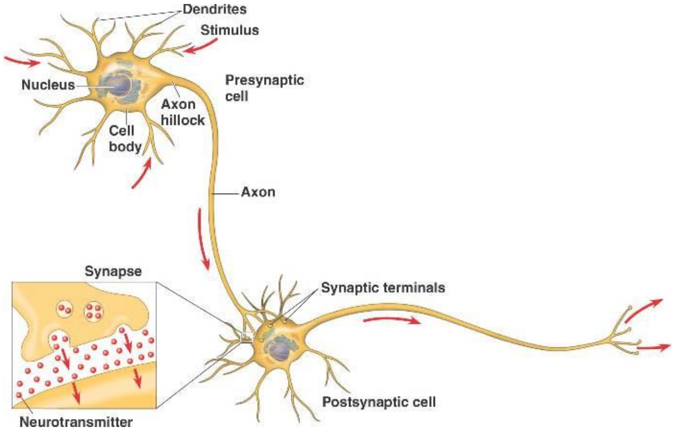
NN Architectures (2 of 2)

- **Generative Adversarial Networks (GAN)**
  - **Architecture**: Composed of two neural networks, a generator and a discriminator, trained together in a game-theoretic setup
  - **Use Cases**:
    - Image generation, creating new and realistic images from scratch
    - Data augmentation when more training data is needed, like in medical imaging or art generation
- **Autoencoders**
  - **Architecture**: Consists of an encoder that compresses the input and a decoder that reconstructs the input from this representation
  - **Use Cases**:
    - Dimensionality reduction to capture the most important features of the data
    - Anomaly detection by comparing the input and the reconstructed output
- **Transformer Networks**
  - **Architecture**: Used for NLP tasks
  - **Use Cases**:
    - Machine translation
    - Language modeling, summarization, and question answering systems like chatbots

Neuron: The structural building block of NN





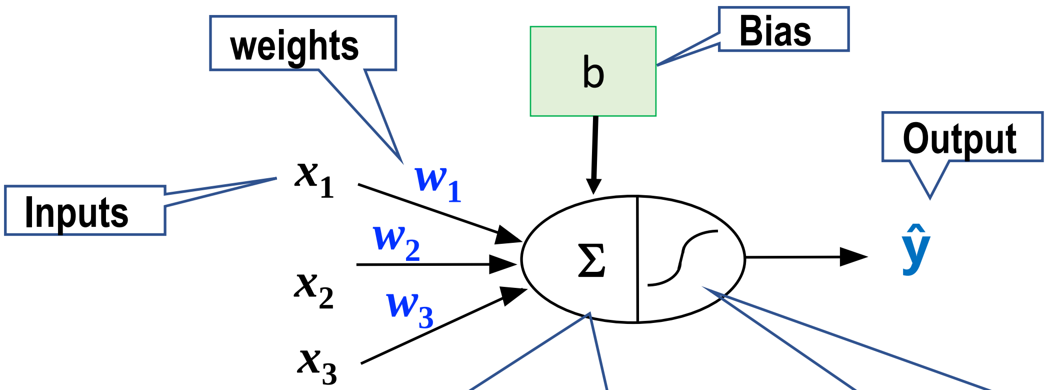


Neuron

- A neuron is the basic computational unit of NN
  - A neuron **computes a weighted sum of its inputs**, **adds a bias** term, and **applies an activation function** to produce the output
- Neurons are typically organized in layers within a neural network, where each layer consists of multiple neurons that process information in parallel
- Neurons in the input layer receive input data, while neurons in subsequent layers (hidden layers and output layer) perform transformations and generate predictions or classifications

Neuron: the structural building block of NN

Neuron is modeled by a unit j connected by **weighted links** $\boldsymbol{w_{ij}}$ to other units i



Sum function (linear combination of inputs):

$$z = x^T w + \mathbf{b} = x_1 * w_1 + x_2 * w_2 + x_3 * w_3 + \mathbf{b}$$

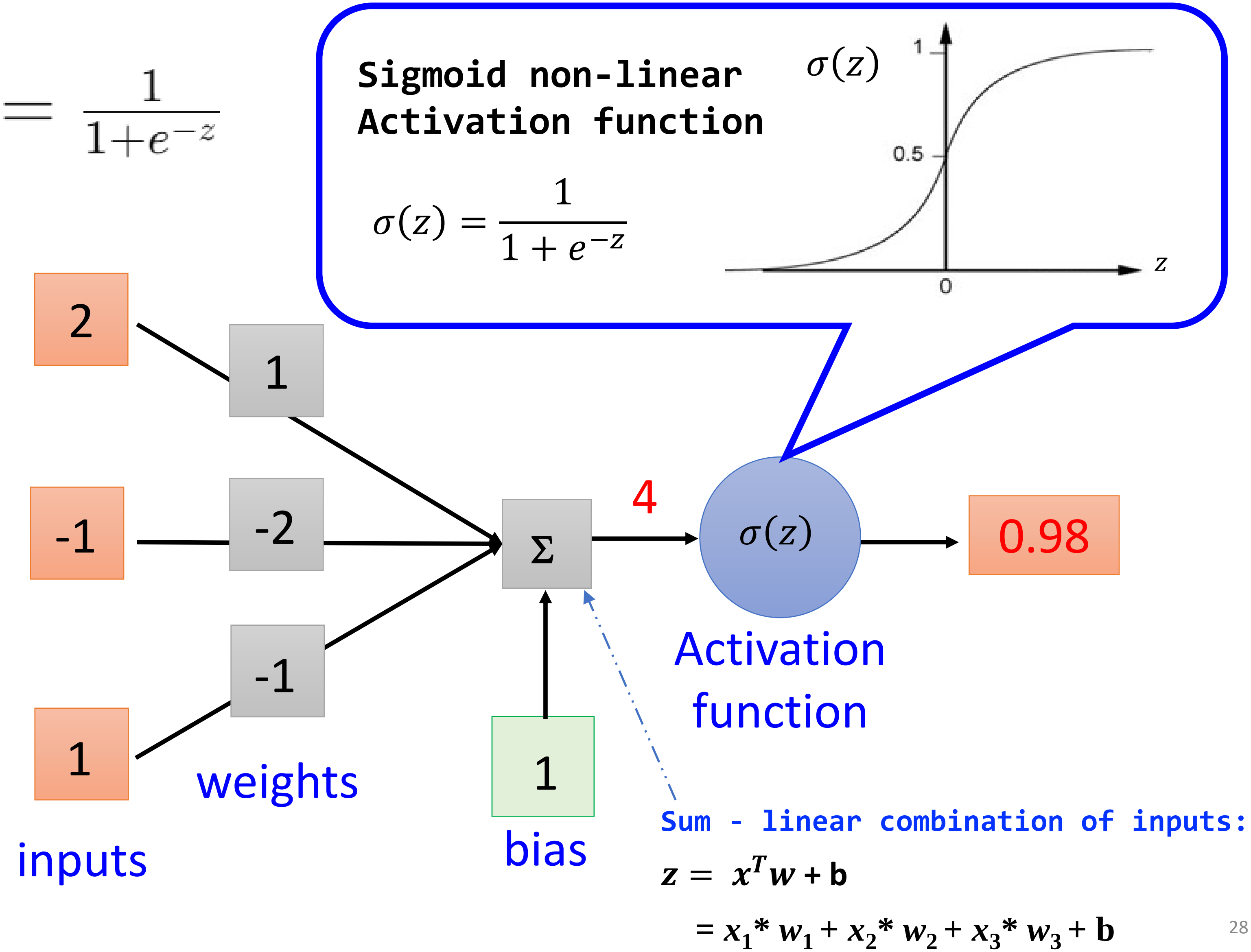
Non-linear Activation function (e.g. sigmoid)

Where:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

Perceptron model for Logistic regression

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

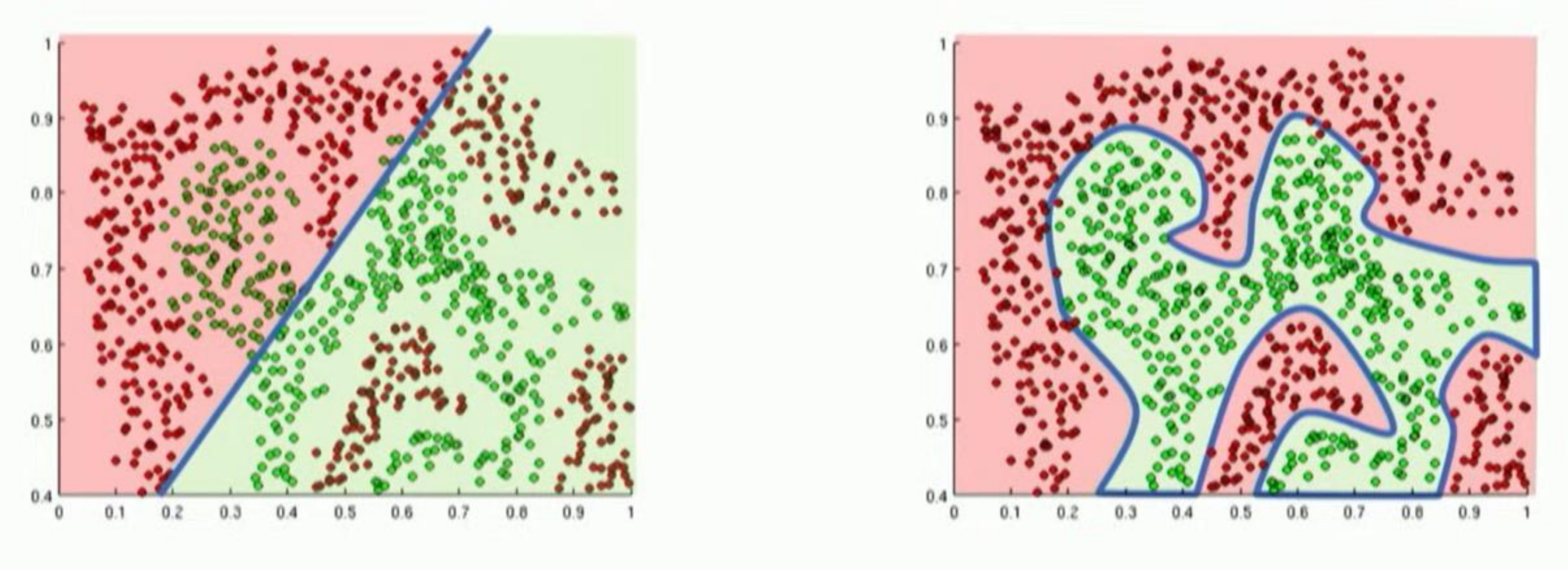


Activation function

- Activation functions bring **nonlinearity** into hidden layers, which increases the capacity of the model to capture complex patterns
- Good activation functions should be **differentiable** for optimization purpose
- An activation function $f(\cdot)$ in the output layer can control the nature of the output (e.g., applying sigmoid activation function to output layer produced a probability value in [0, 1])

Importance of Activation Functions

- The purpose of activation functions is to introduce non-linearities into the network



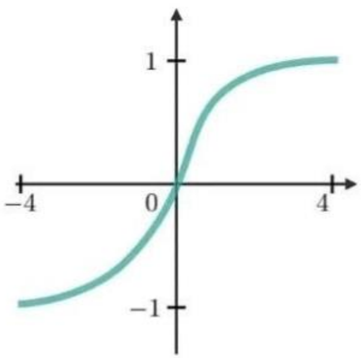
Linear activation functions produce linear decisions no matter the network size

Non-linearities allow us to approximate arbitrarily complex functions

Activation functions of a neuron

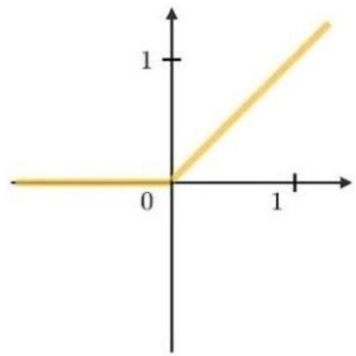
Hyperbolic Tangent (Tanh)

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



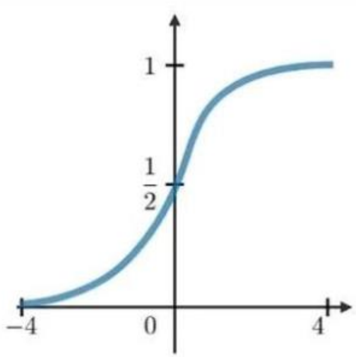
Rectified Linear Unit (ReLU)

$$g(z) = \max(0, z)$$



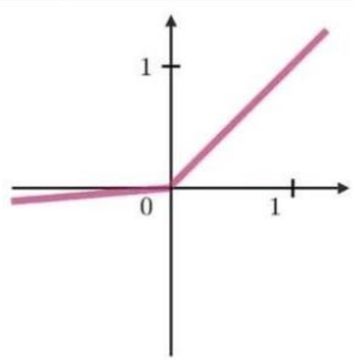
Sigmoid

$$g(z) = \frac{1}{1 + e^{-z}}$$



Leaky ReLU

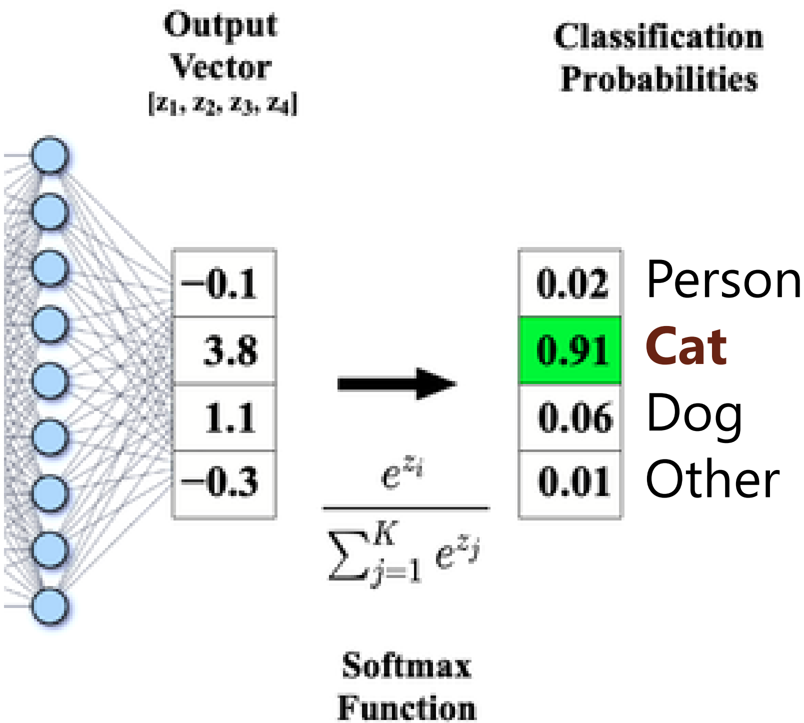
$$g(z) = \max(\epsilon z, z)$$
with $\epsilon \ll 1$



The optimal activation function may vary depending on the task and dataset

`10.nn\02_activation_functions.ipynb`

Softmax activation function



Given a vector of scores $z=(z_1, z_2,\ldots, z_n)$, the softmax function computes the probabilities $p=(p_1,p_2,\ldots,p_n)$ for each class i as:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

The denominator sums up the exponentials of all scores across all classes, ensuring that the resulting probabilities sum up to 1

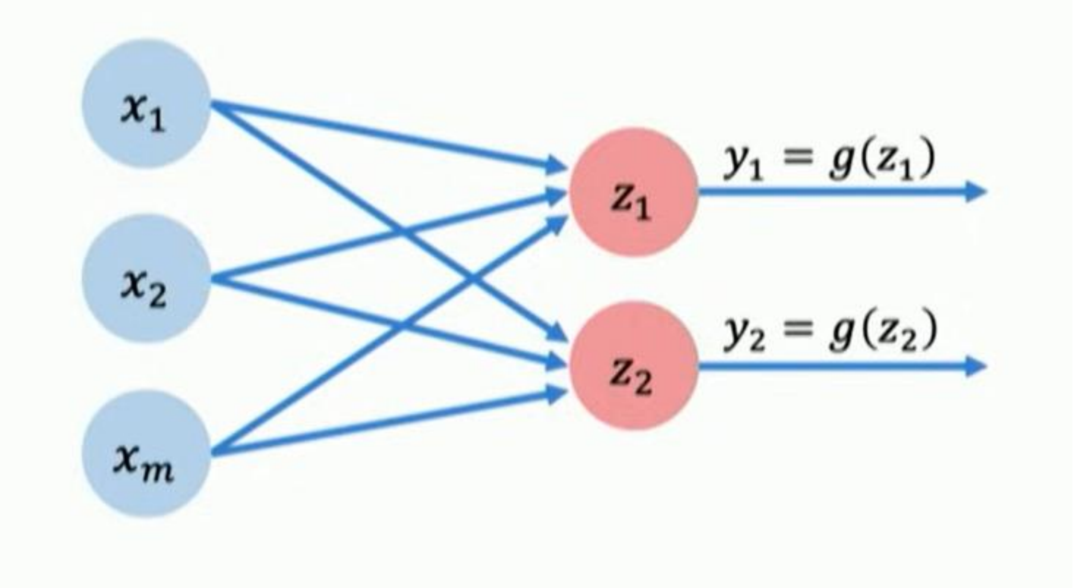
Choosing the right last layer activation & loss function (Pytorch)

Problem Type	Last Layer Activation	Loss Function
Binary classification	Sigmoid	binary_crossentropy
Multiclass classification	Softmax	categorical_crossentropy
Regression to arbitrary value	[None]	mse
Regression to values between 0 and 1	Sigmoid	mse or binary_crossentropy

Building a NN with Neurons

Multi Output NN

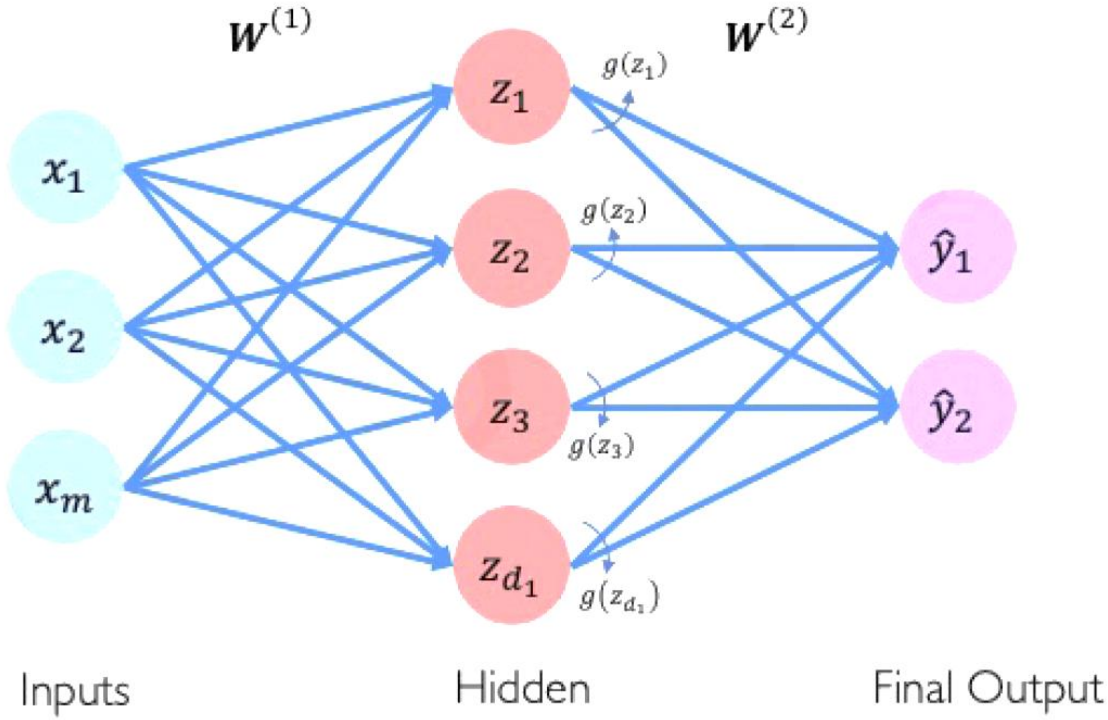
- NN with a **dense layers**
  - all inputs are densely connected to all outputs



$$z_i = \mathbf{b}_i + \sum_{j=1}^{m} x_j \, w_{j,i}$$

10.nn\01_logistic-regression-nn.ipynb

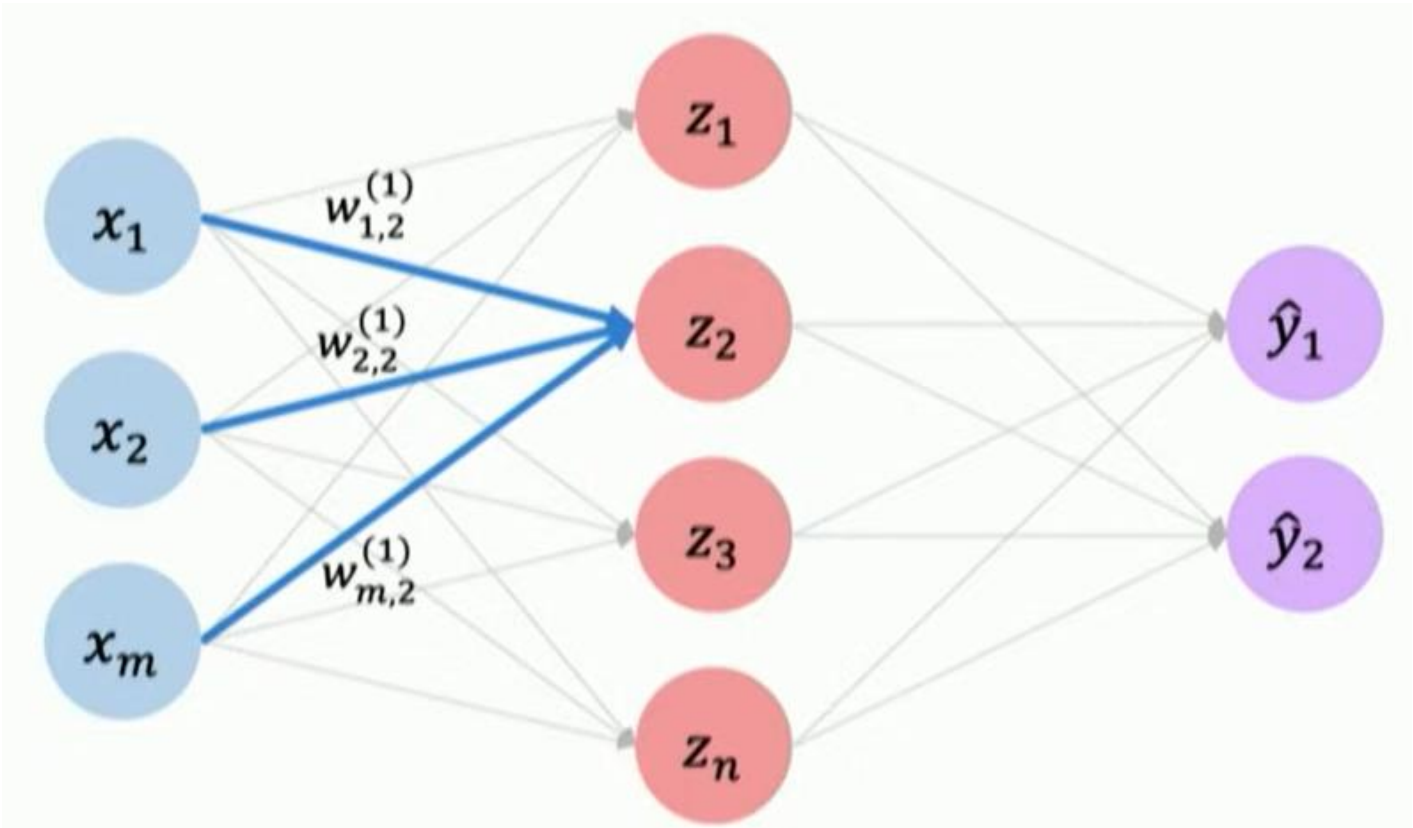
Single Layer Neural Network



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)} \right)$$

w_0 refers to the bias term

Single Layer Neural Network

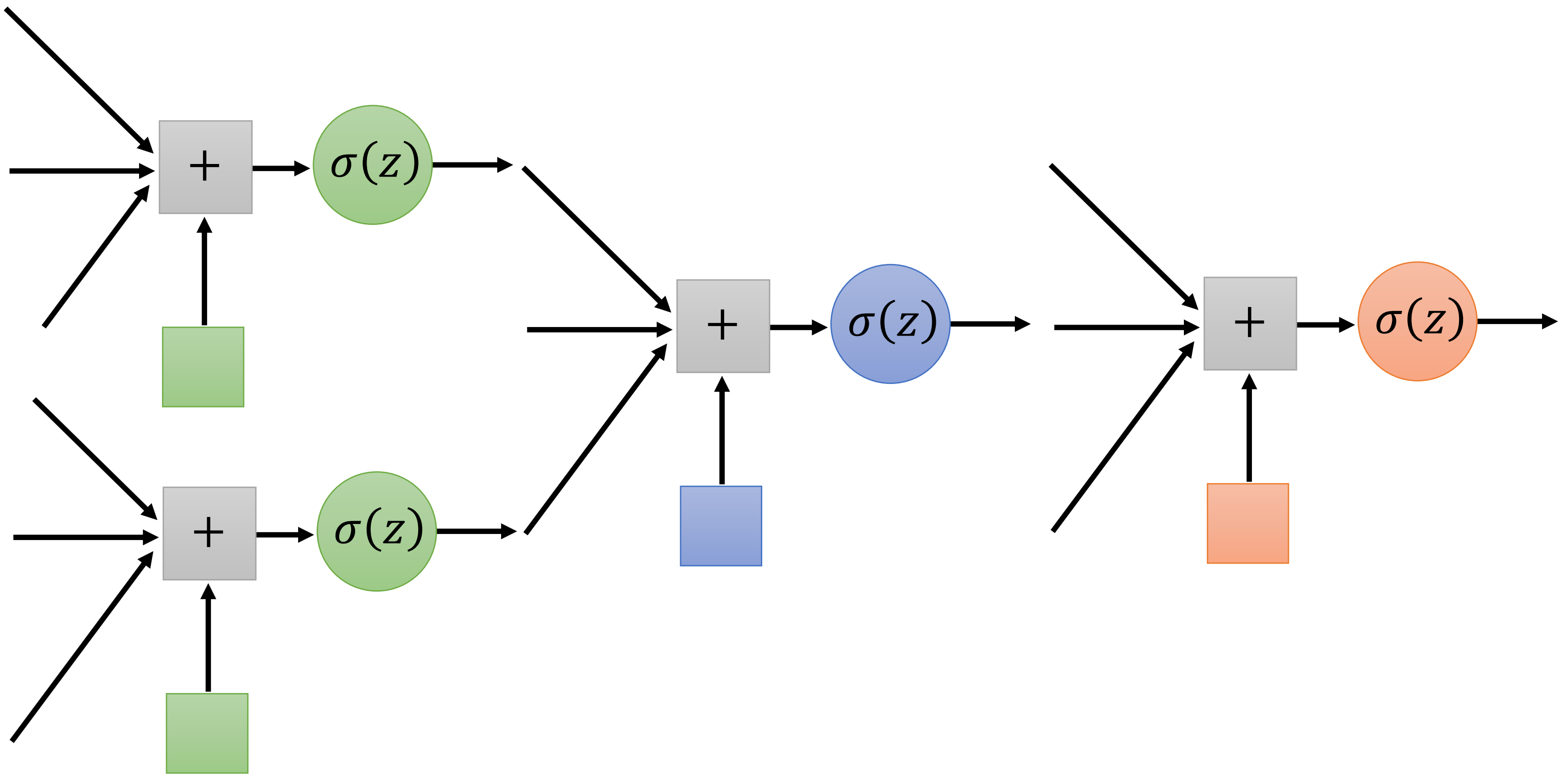


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)} \right)$$

w_0 refers to the bias term

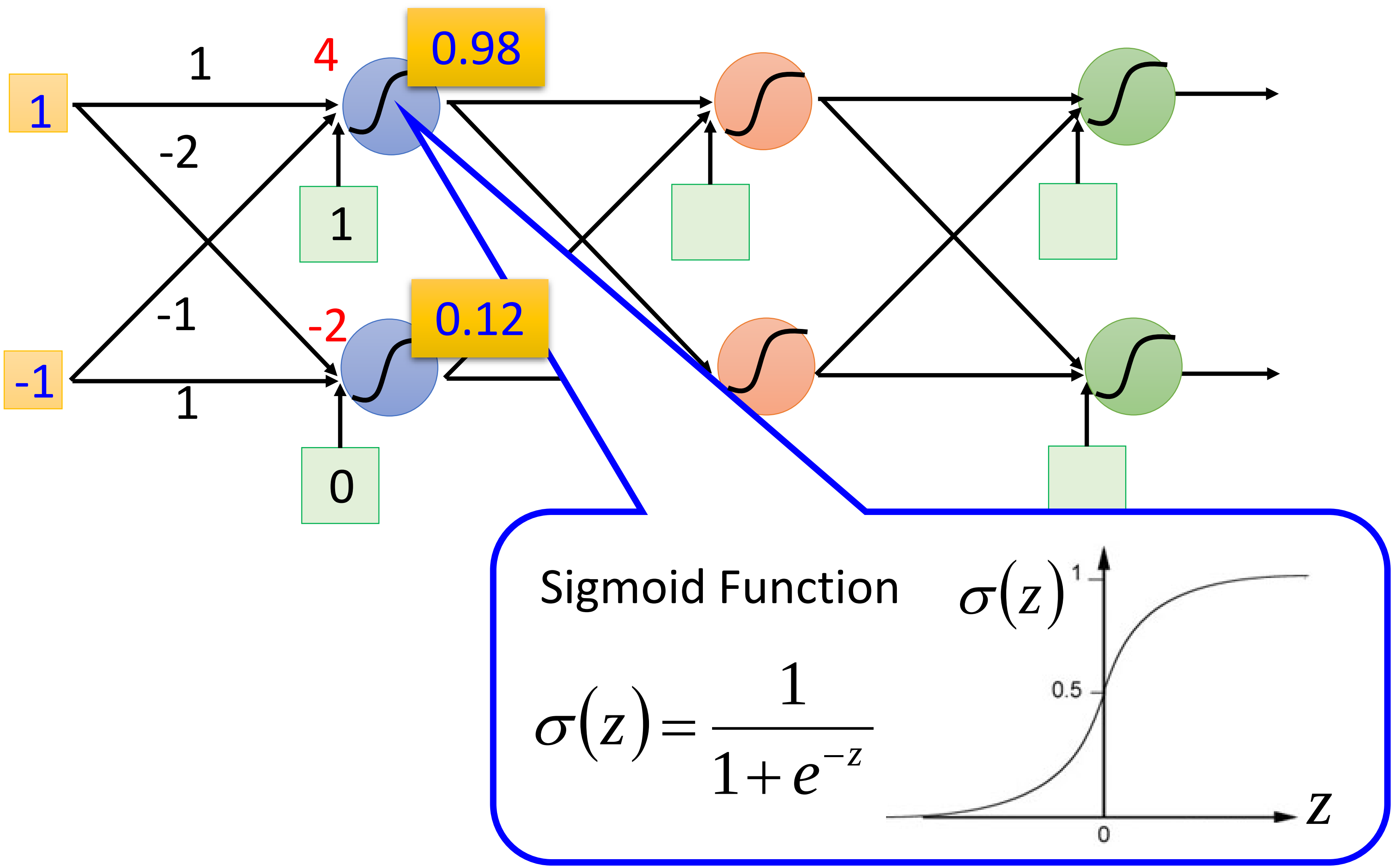
Neural Network

- The neurons have different values of weights and biases (learned from data)

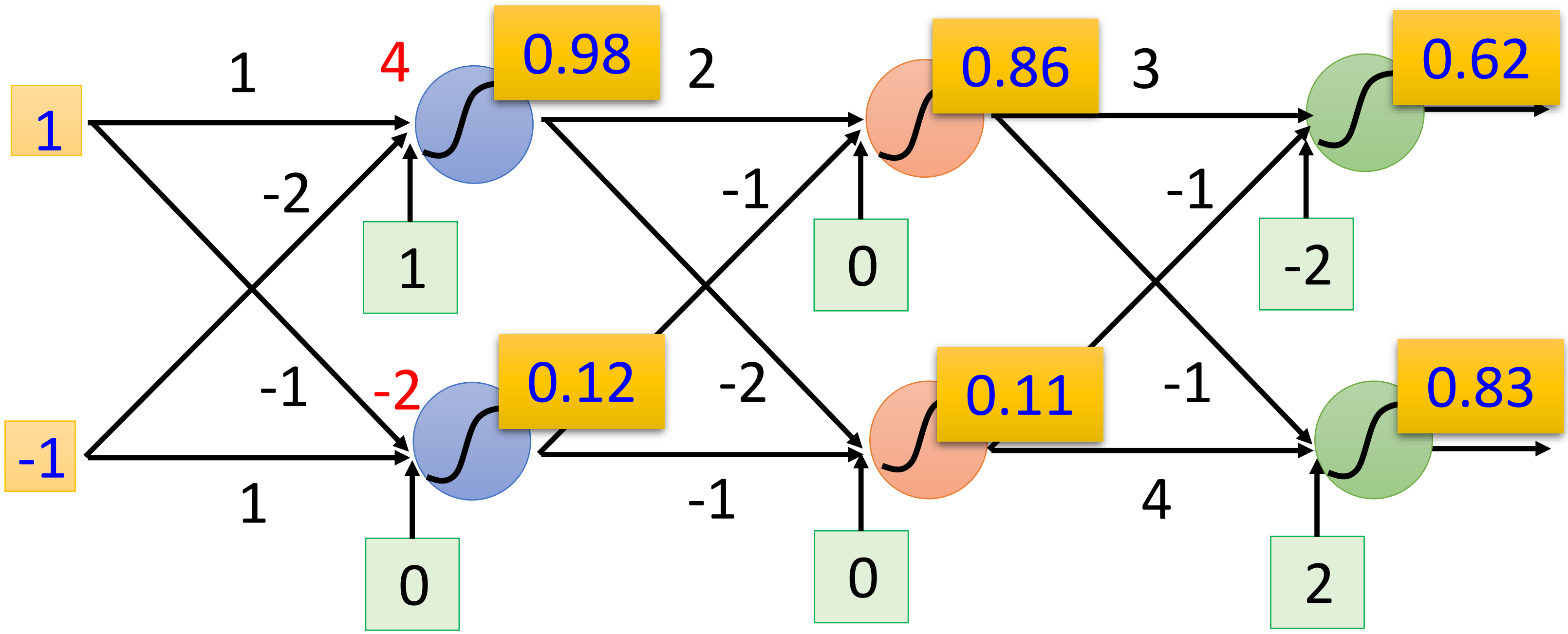


Weights and biases are called network parameters

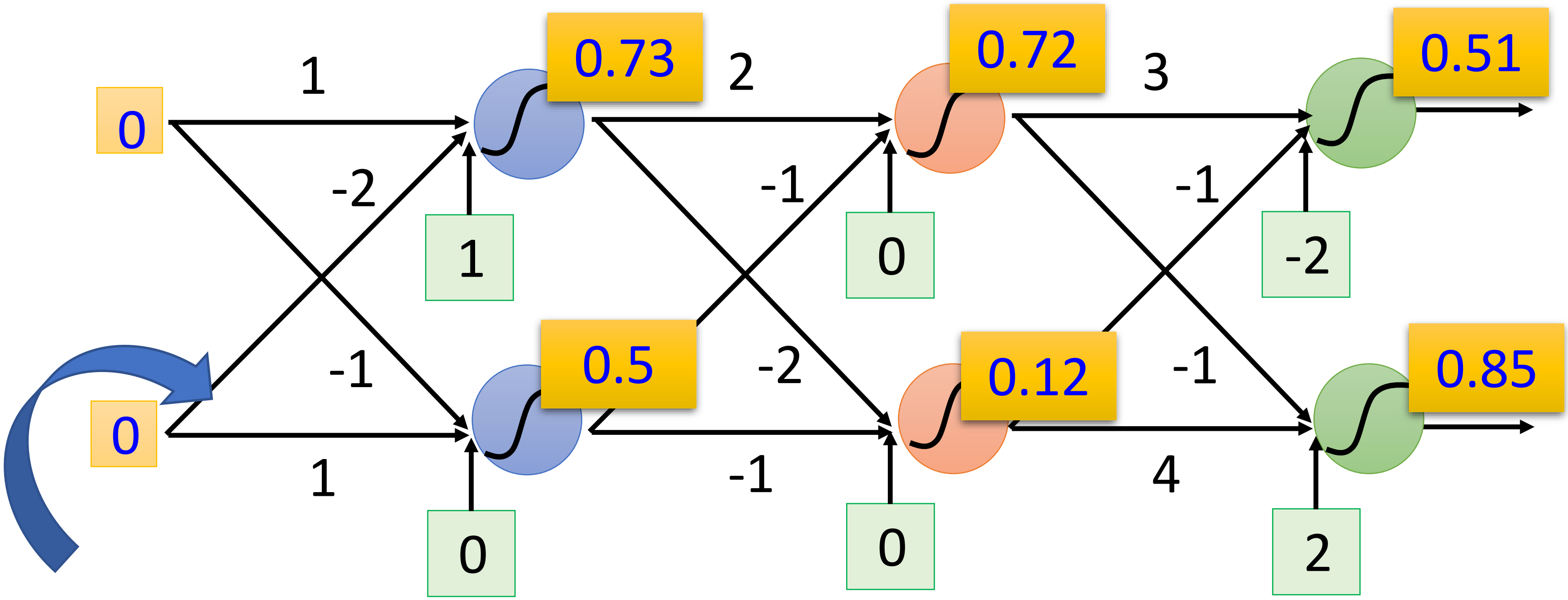
Fully-connected Feedforward Network



Fully-connected Feedforward Network
(with input 1,-1)



Fully-connected Feedforward Network
(with input 0, 0)



This is a function

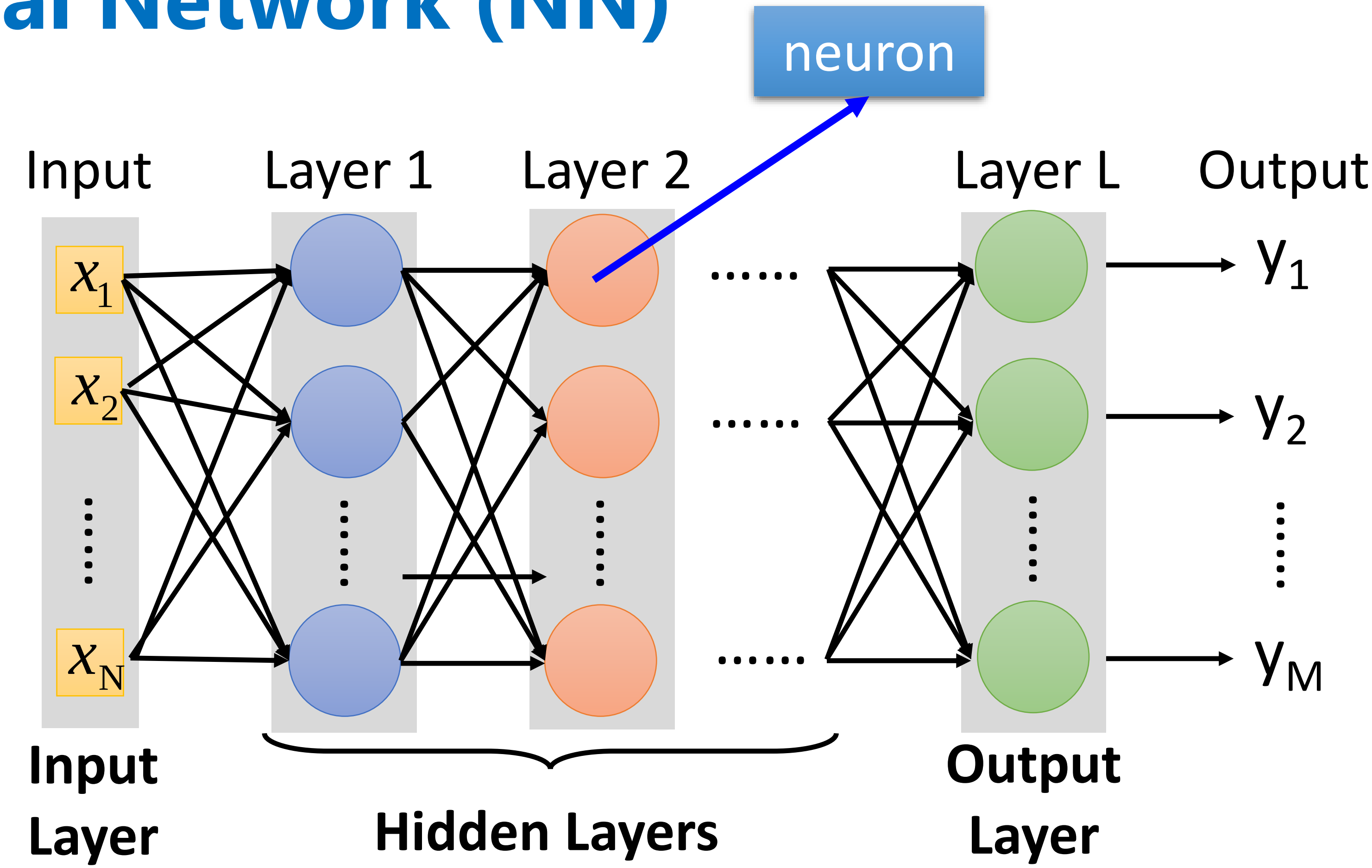
Input vector, output vector

$$f\left(\begin{bmatrix}1\\-1\end{bmatrix}\right)=\begin{bmatrix}0.62\\0.83\end{bmatrix}\quad f\left(\begin{bmatrix}0\\0\end{bmatrix}\right)=\begin{bmatrix}0.51\\0.85\end{bmatrix}$$

Given parameters, define a function

Given network structure, define ***a function set***

Neural Network (NN)



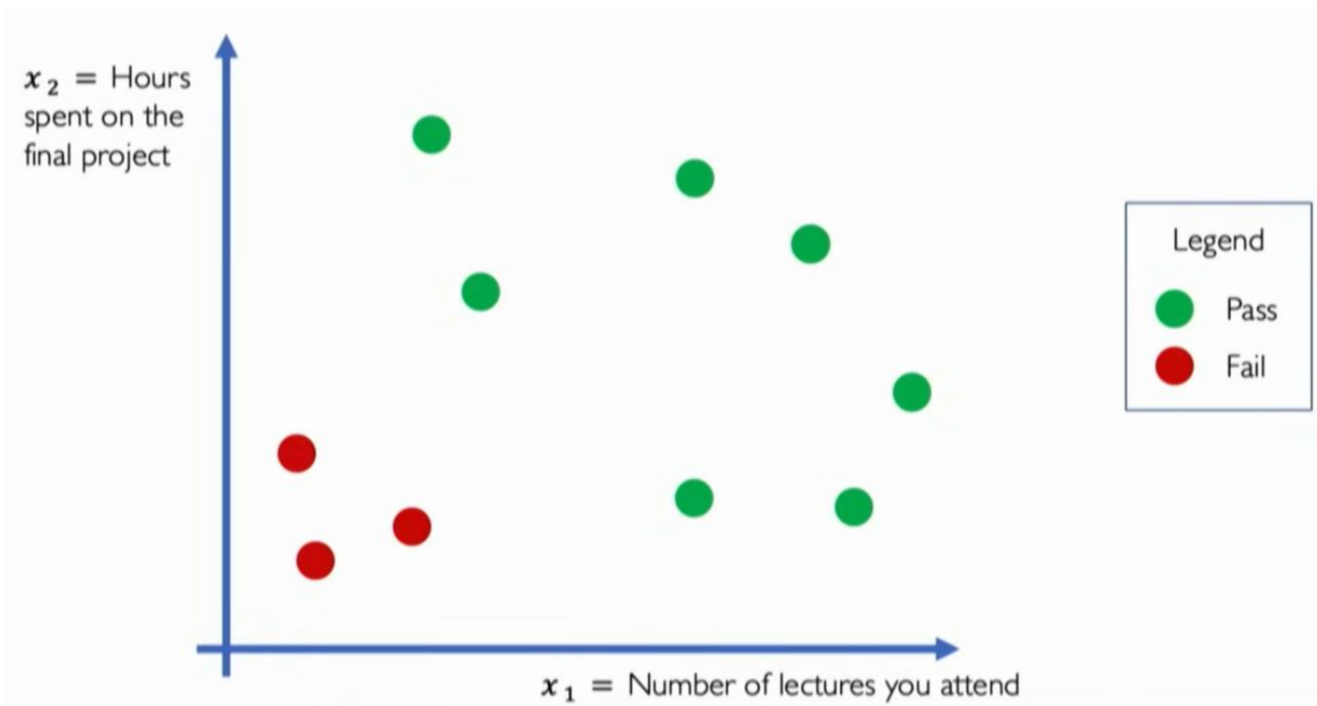
Has 2 passes:

1. **Forward pass**: compute output based on input of training data

2. **Backward pass**: Adjust weights/biases to reduce loss

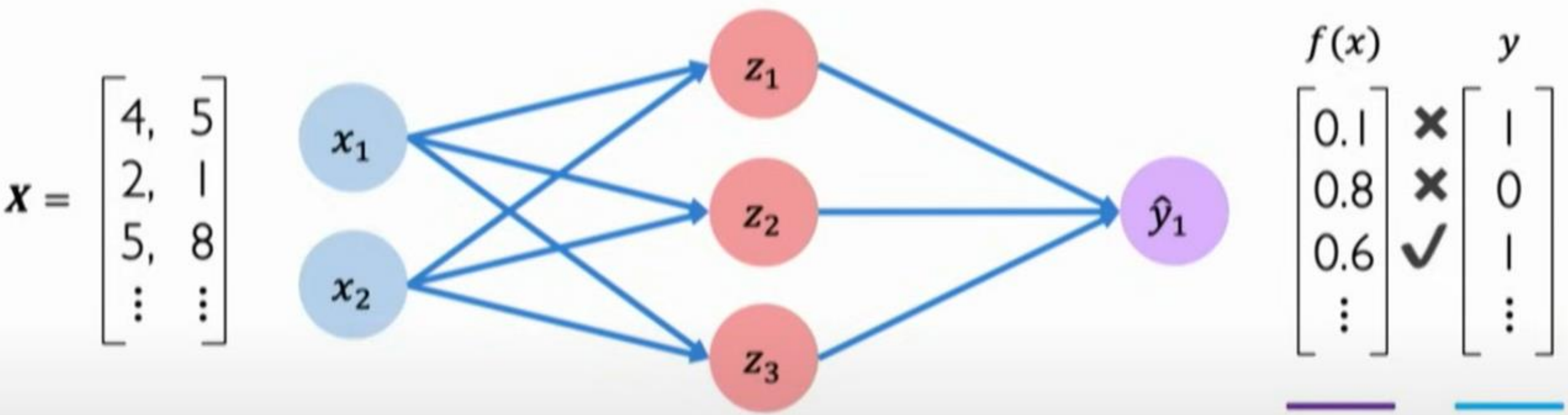
Example Problem

- Will I pass this class?
- Let's start with a simple two feature model
  - x_1 = Number of lectures you attend
  - x_2 = Hours spent on the final project



Quantifying Loss

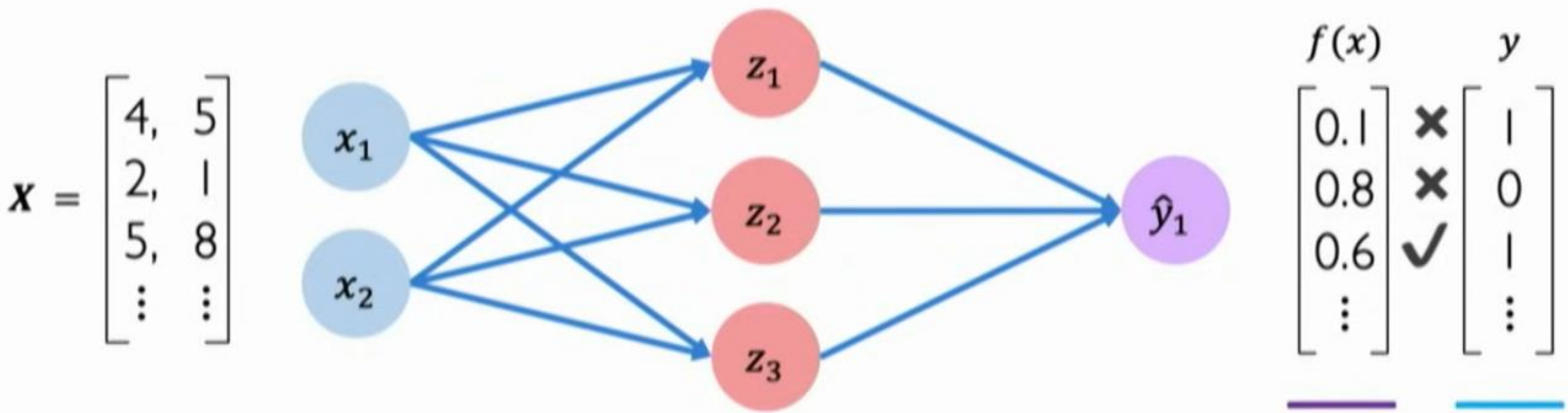
- The loss (cost function) of our network measures the cost incurred from incorrect predictions over our entire dataset



$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\big)$$

Binary Cross Entropy Loss

- **Cross entropy loss** can be used with models that output a probability between 0 and 1



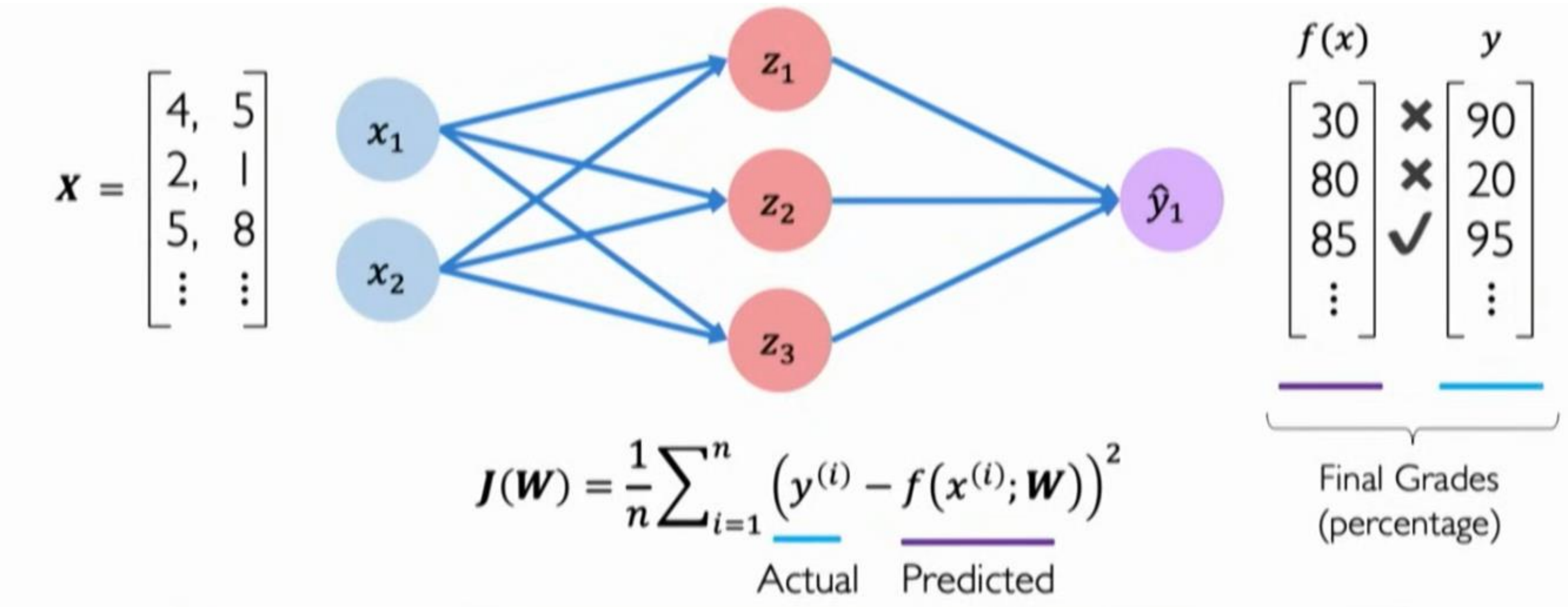
$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)}; W\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)}; W\big)}_{\text{Predicted}}\Big)$$

PyTorch

```
lossfun = nn.BCELoss()
```

Mean Squared Error Loss

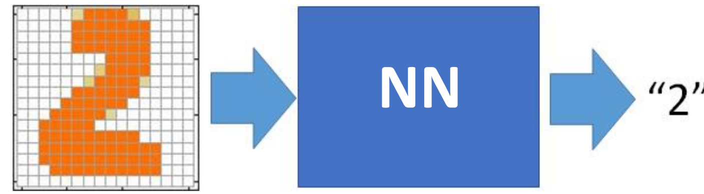
- Mean squared error loss can be used with regression models that output continuous real numbers



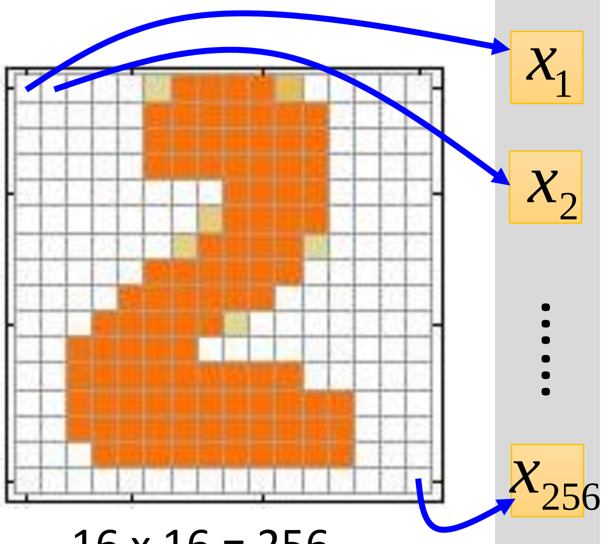
PyTorch

```
lossfun = nn.MSELoss()
```

Example Application



Input

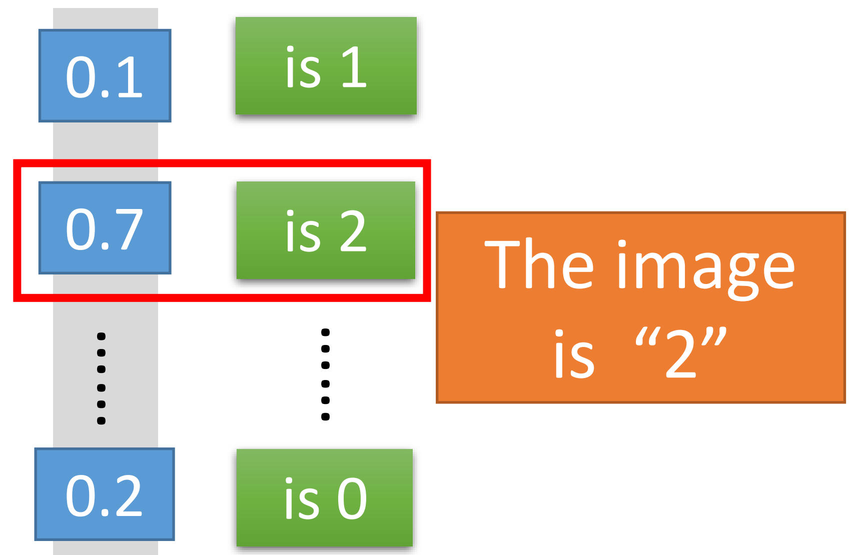


16 x 16 = 256

Ink → 1

No ink → 0

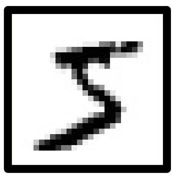
Output

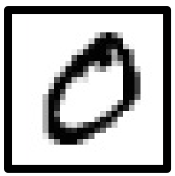


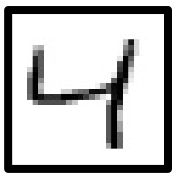
Each output represents the confidence of a digit

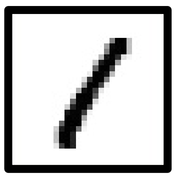
Training Data

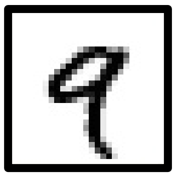
- Preparing training data: images and their labels

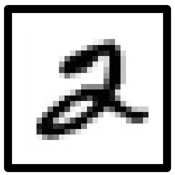

“5”

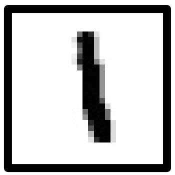

“0”

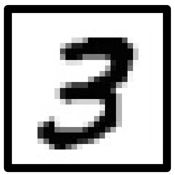

“4”


“1”

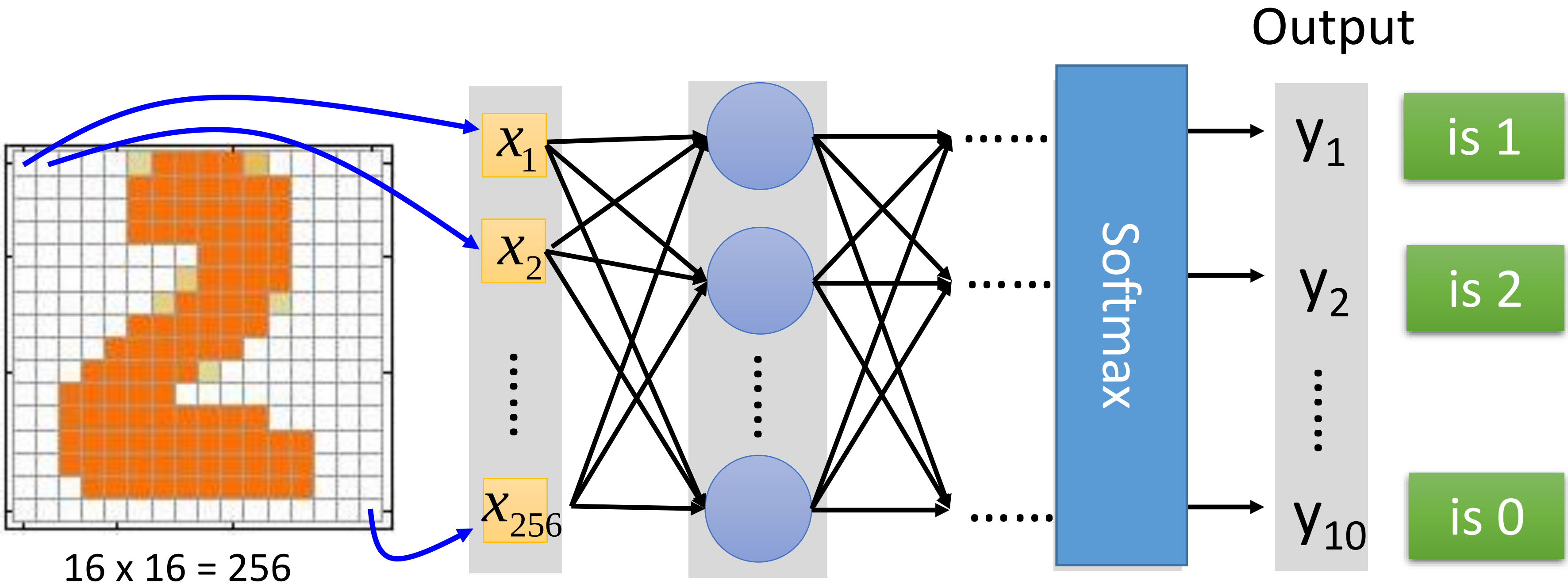

“9”


“2”


“1”


“3”

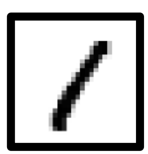
What is a good function?



Ink → 1

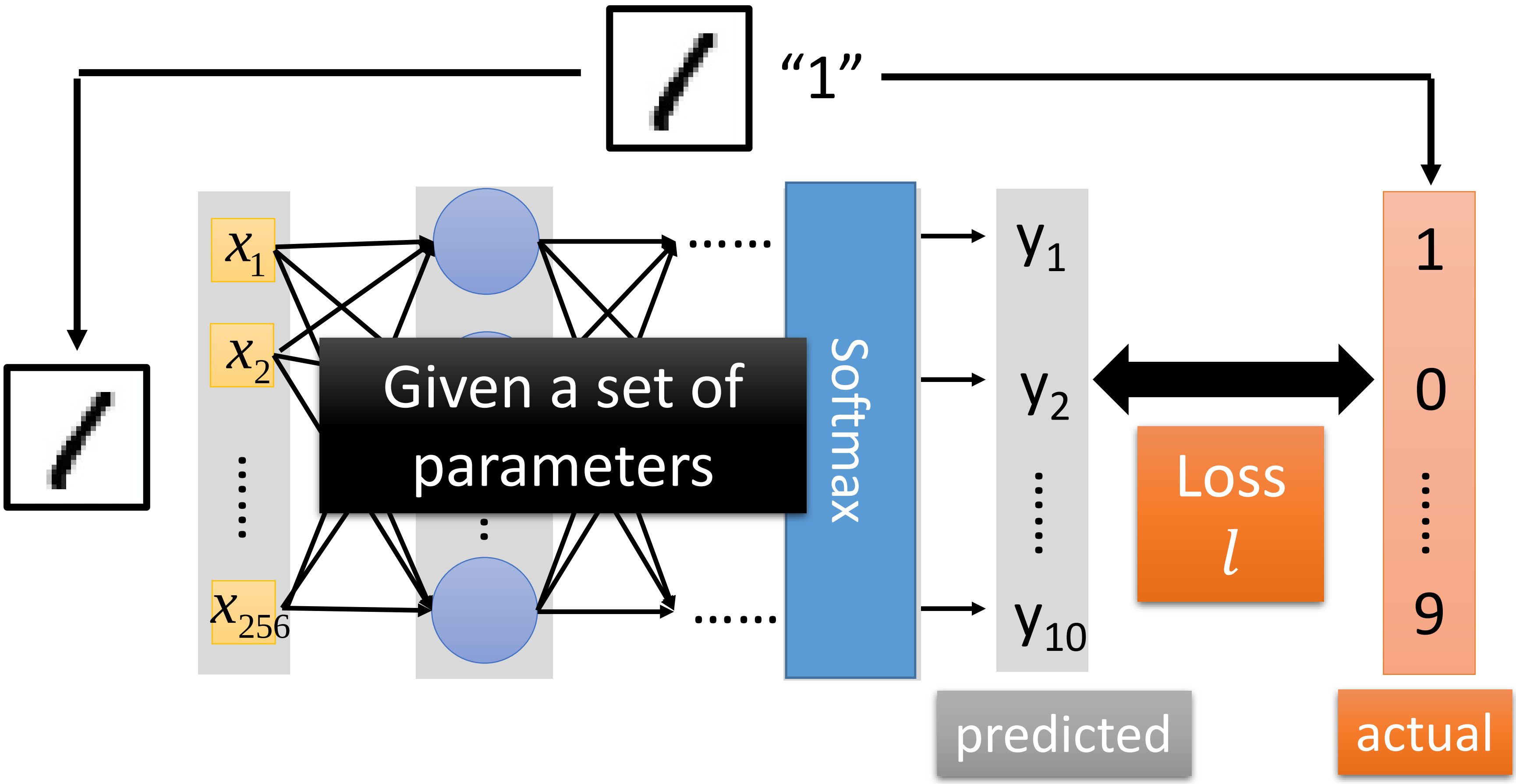
No ink → 0

A good function should ......

Input:  y_1 has the maximum output value

Input:  y_2 has the maximum output value

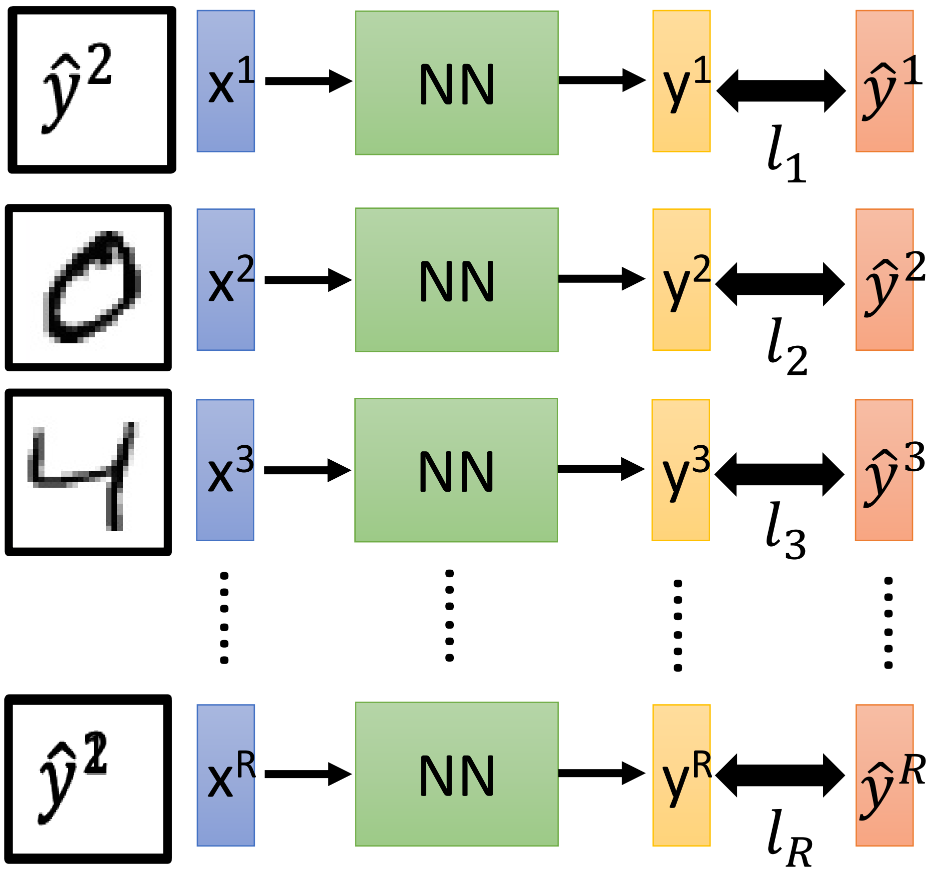
What is a good function?



- Loss can be **square error** or **cross entropy** between the network output and the actual label

Total Loss

For all training data …



Total Loss:

$$\boxed{L = \sum_{r=1}^{R} l_r}$$

As small as possible

Find ***the network parameters*** that minimize total loss L

Training a NN

Loss Optimization

- We want to **find the network weights** (and biases) that achieve the lowest loss

$$W^* = \underset{W}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; W\right), y^{(i)}\right)$$

$$W^* = \underset{W}{\operatorname{argmin}}\, J(W)$$

Remember:

$$W = \left\{W^{(0)}, W^{(1)}, \cdots\right\}$$

Gradient Descent

Algorithm

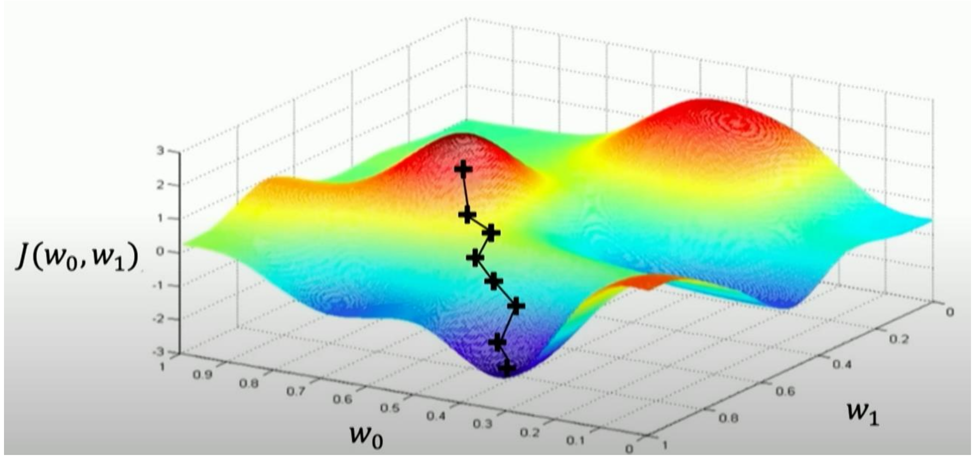
1. Initialize weights randomly
2. Loop until convergence:

   Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

   Learning Rate

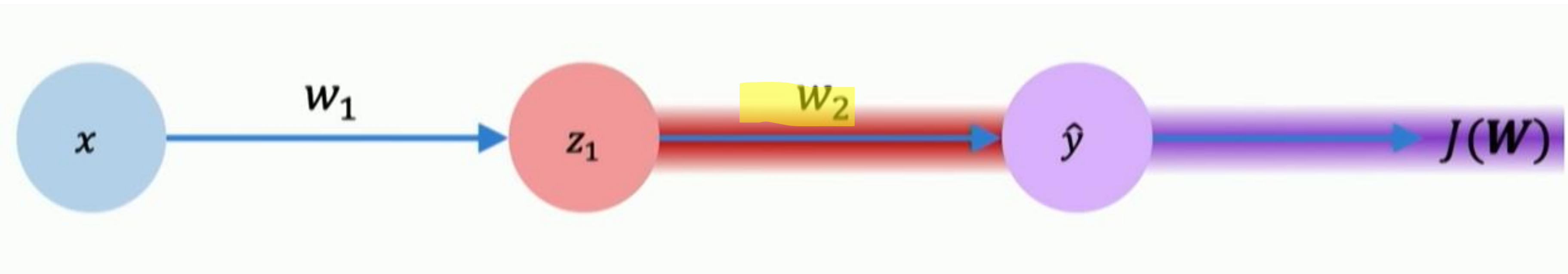
   Update weights, $\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

At each step, the weight vector is modified in the direction that produces the **steepest descent** along the error surface



Computing Gradients: Backpropagation

- How does a small change in one weight (e.g. $\mathbf{w_2}$) affects the final loss $\mathbf{J(W)}$?

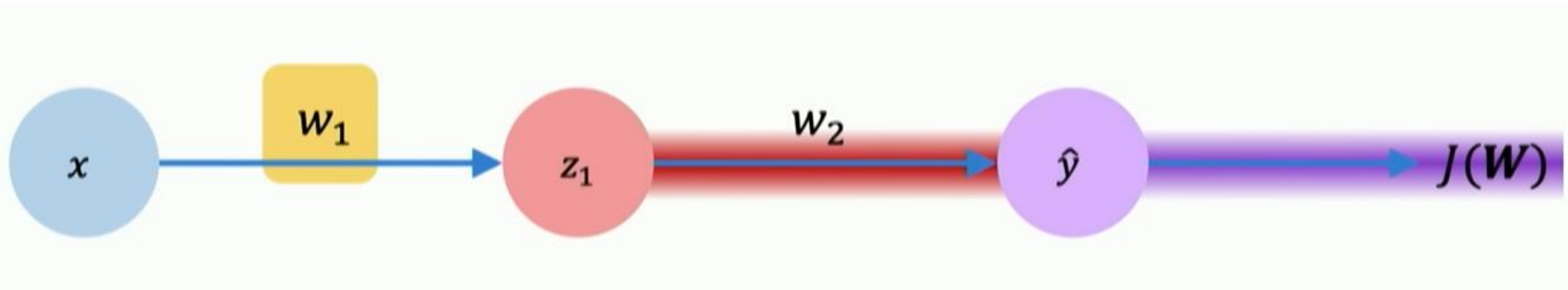


$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

Apply chain rule!

Computing Gradients: Backpropagation

- How does a small change in one weight (e.g. $\mathbf{w_1}$) affects the final loss $\mathbf{J(W)}$?



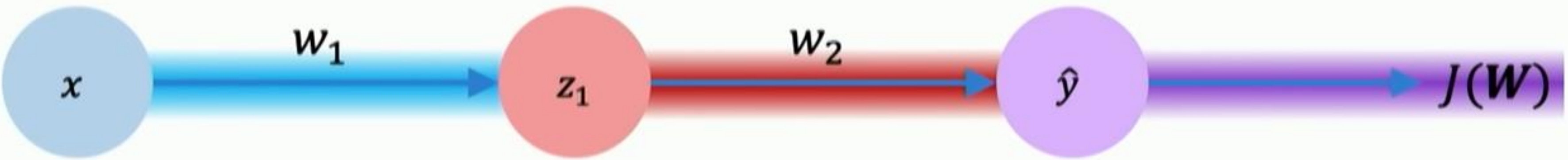
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

Computing Gradients: Backpropagation

- How does a small change in one weight (e.g. $\mathbf{w_1}$) affects the final loss $\mathbf{J(W)}$?



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Apply chain rule!

Repeat this for **every weight in the network** using gradients from the outputs to the inputs

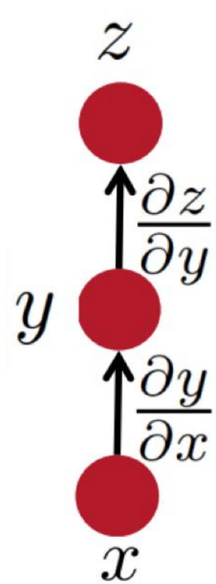
Chain Rule

$$\frac{d}{dx}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right)g'(x)$$

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

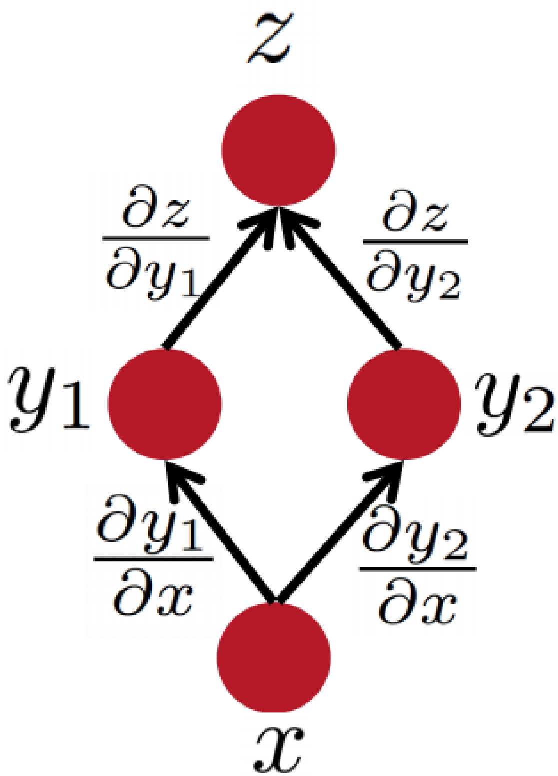
Simple chain rule

- If z is a function of y, and y is a function of x
  - Then z is a function of x, as well
- Question: how to find $\frac{\partial z}{\partial x}$



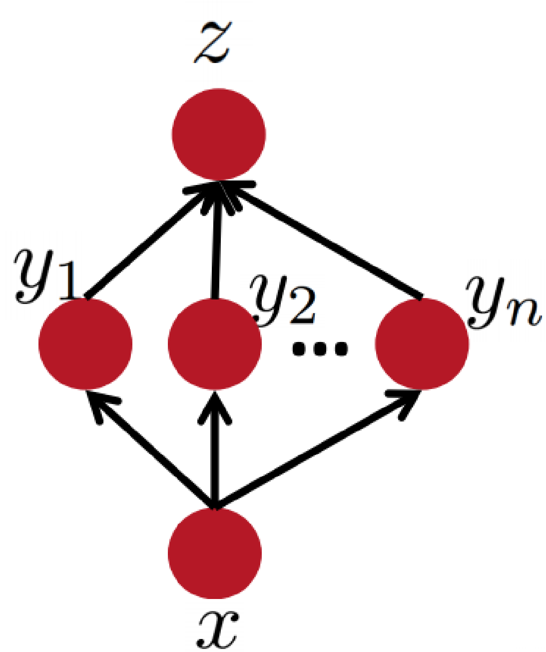
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Multiple path chain rule



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

In general:



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

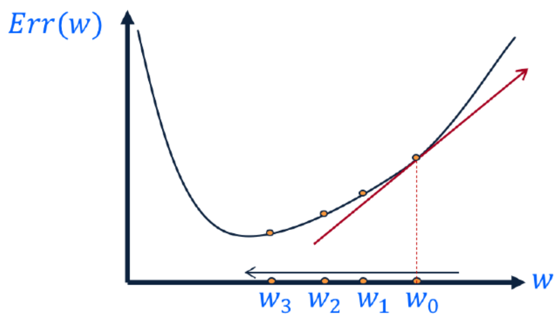
Gradient Descent Algorithms

Algorithm	**Reference**
SGD	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function" 1952
Adam	Kingma et al. “Adam: A Method for Stochastic Optimization" 2014
Adadelta	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method" 2012
Adagrad	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization" 2011
…	

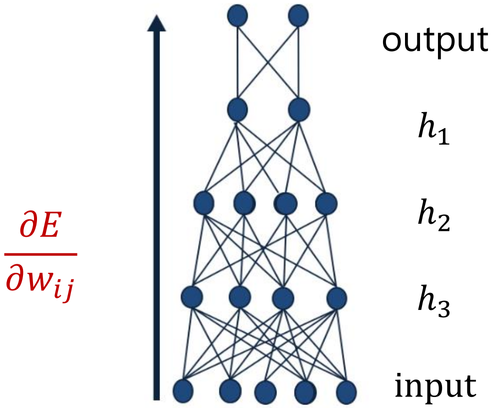
PyTorch https://pytorch.org/docs/stable/optim.html#algorithms

Key intuitions of Backpropagation

- Gradient Descent
  - Change the weights in the direction of gradient to minimize the error function

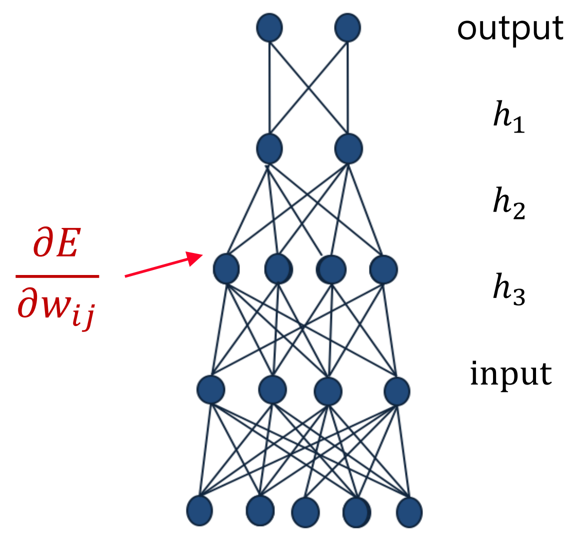


- Chain Rule
  - Use the chain rule to calculate the weights of the intermediate weights



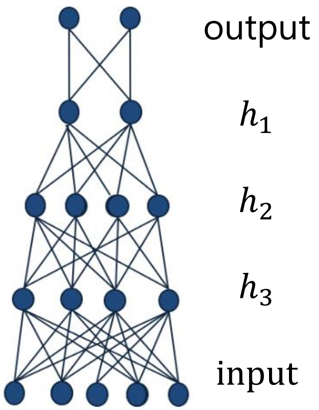
Backpropagation: the big picture

- Loop over instances:
  1. **The forward step**
     - Given the input, make predictions layer-by-layer, starting from the input layer)
  2. **The backward step**
     - Calculate the error in the output
     - Update the weights layer-by-layer, starting from the output layer



Quiz time!

- Given a neural network, how can we make predictions?
  - Do a forward pass: given input, calculate the output of each layer (starting from the input layer), until you get to the output



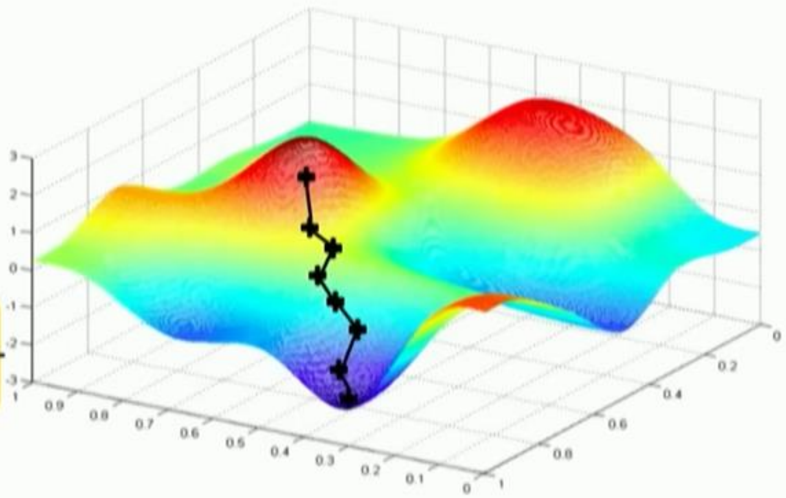
- What is the purpose of backward step?
  - To update the weights, given an output error
- Why do we use the chain rule?
  - To calculate gradient in the intermediate layers
- How to make backpropagation more efficient?
  - Make it parallelized

Training in Practice

Batch Gradient Descent

Algorithm

1. Initialize weights randomly
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

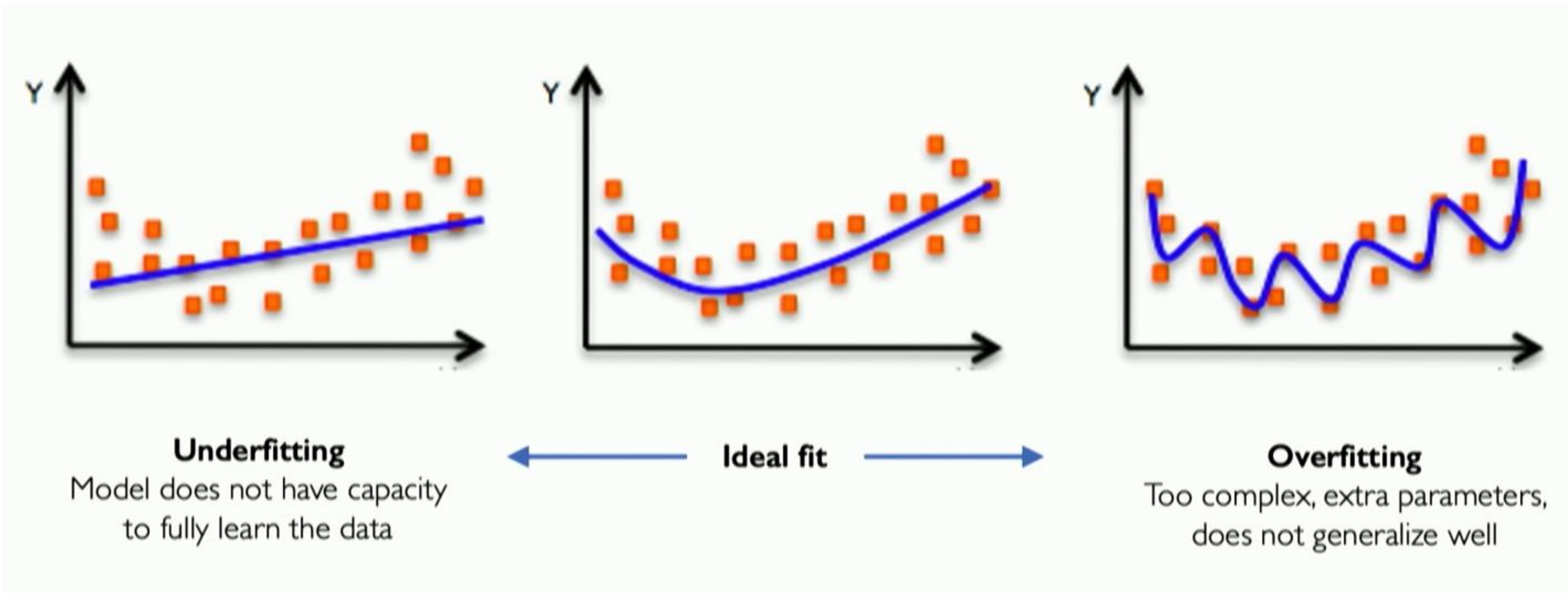


Fast to compute and a much better estimate of the true gradient!

Training in Practice

- **No guarantee of convergence:** neural networks form non-convex functions with multiple local minima
  - To **avoid local minima**: several trials with different random initial weights
- In practice, many large networks can be trained on large amounts of data for realistic problems
- **Many epochs** (tens of thousands) may be needed for adequate training. Large data sets may require many hours of CPU
- **Termination criteria:** Number of epochs; Threshold on training set error; No decrease in error; Increased error on a validation set

The Problem of Overfitting



Over-fitting prevention

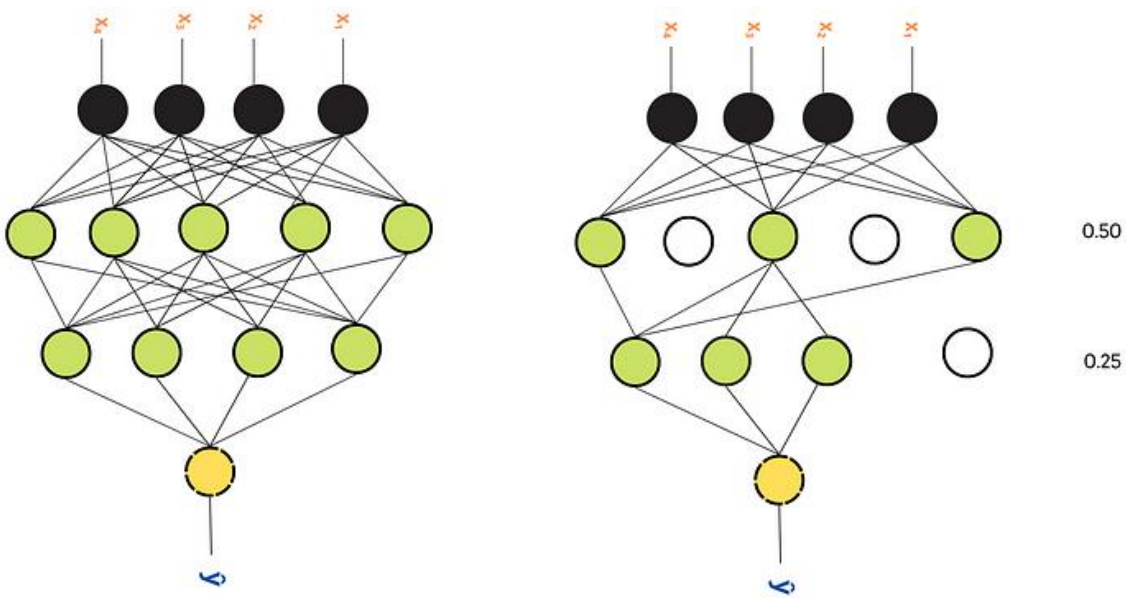
- **Too few hidden layers** prevent the system from adequately fitting the data and learning the patterns
- Using **too many hidden layers** leads to over-fitting
- Cross-validation method can be used to determine an appropriate number of hidden layers
- Approaches to prevent over-fitting includes *Dropout training* and *Early Stopping*

Regularization 1: Dropout training

- During dropout training, randomly selected neurons are temporarily "dropped out" or ignored during the forward and backward passes of training
  - This means their activations and gradients are not propagated through the network for that particular iteration

PyTorch

```
nn.Dropout(p=0.2)
```

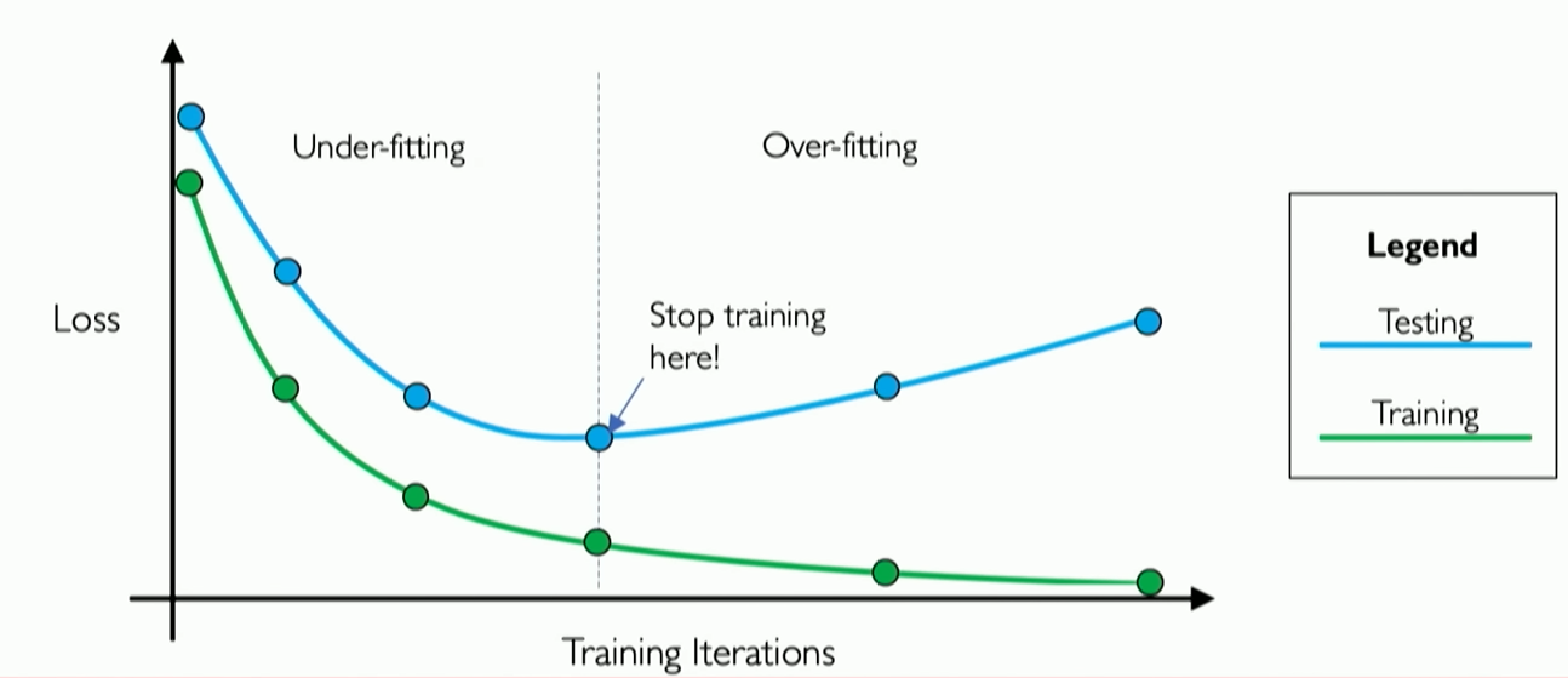


(a) Standard Neural Network

(b) After applying dropout

Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Summary

Neuron	Neural Networks	Training in Practice
• Structural building block of NN • Nonlinear activation functions	• Stacking Neurons to form neural networks • Optimization through backpropagation	• Batching • Regularization (e.g., Early Stopping, Dropout)