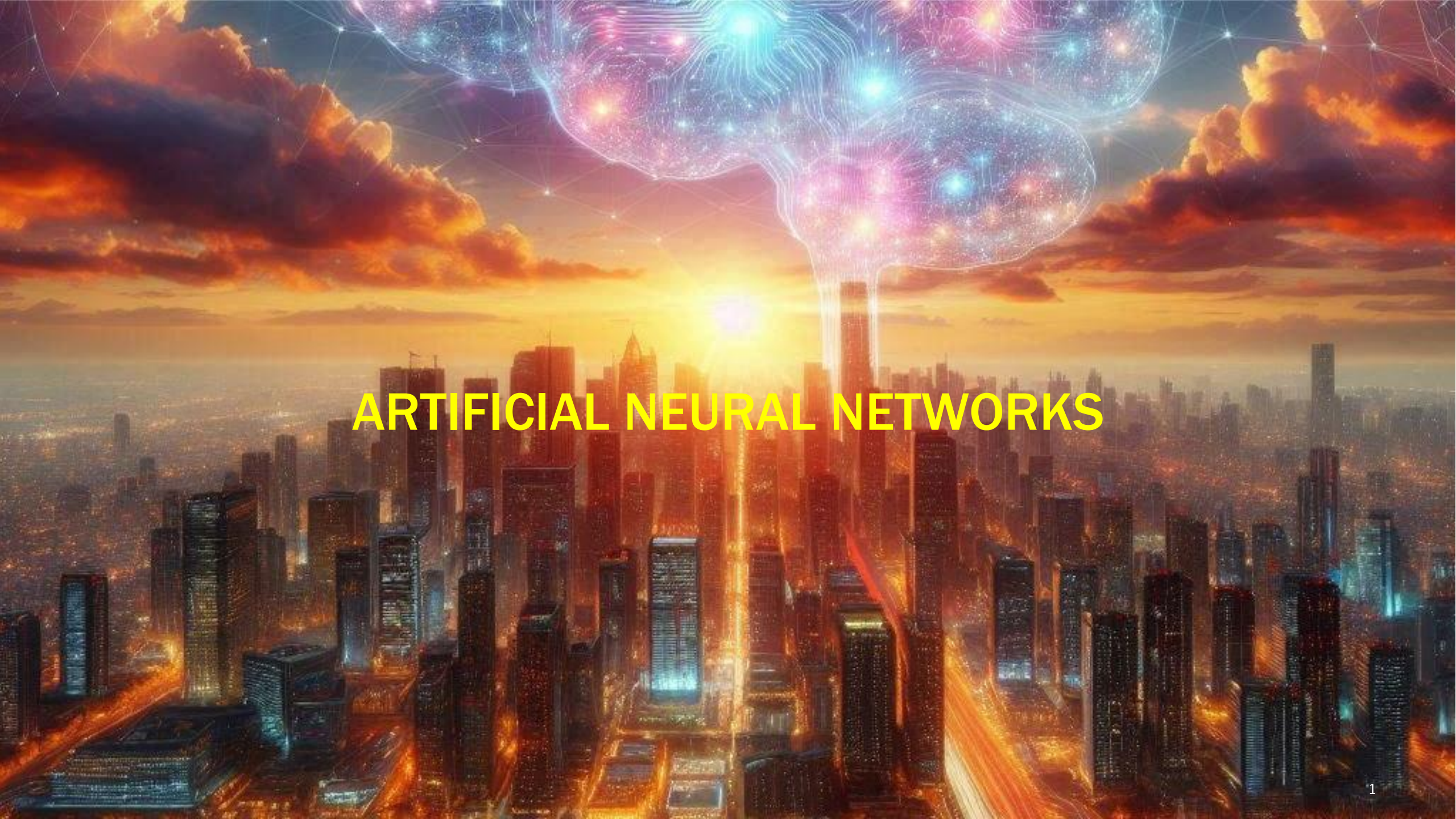

ARTIFICIAL NEURAL NETWORKS

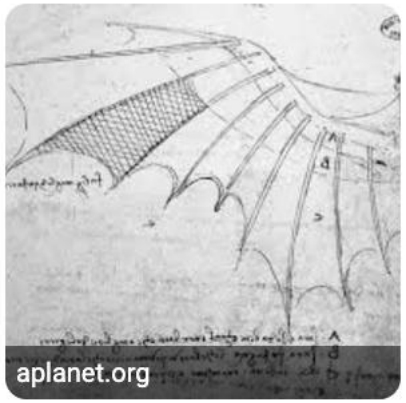
INSPIRATION FROM NATURE

Velcro

Whale wind turbines

Gecko skin

Flight



Sonar

Shark skin

BYU

INSPIRATION: THE BRAIN

- Many machine learning methods inspired by biology, e.g., the (human) brain
- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to ~

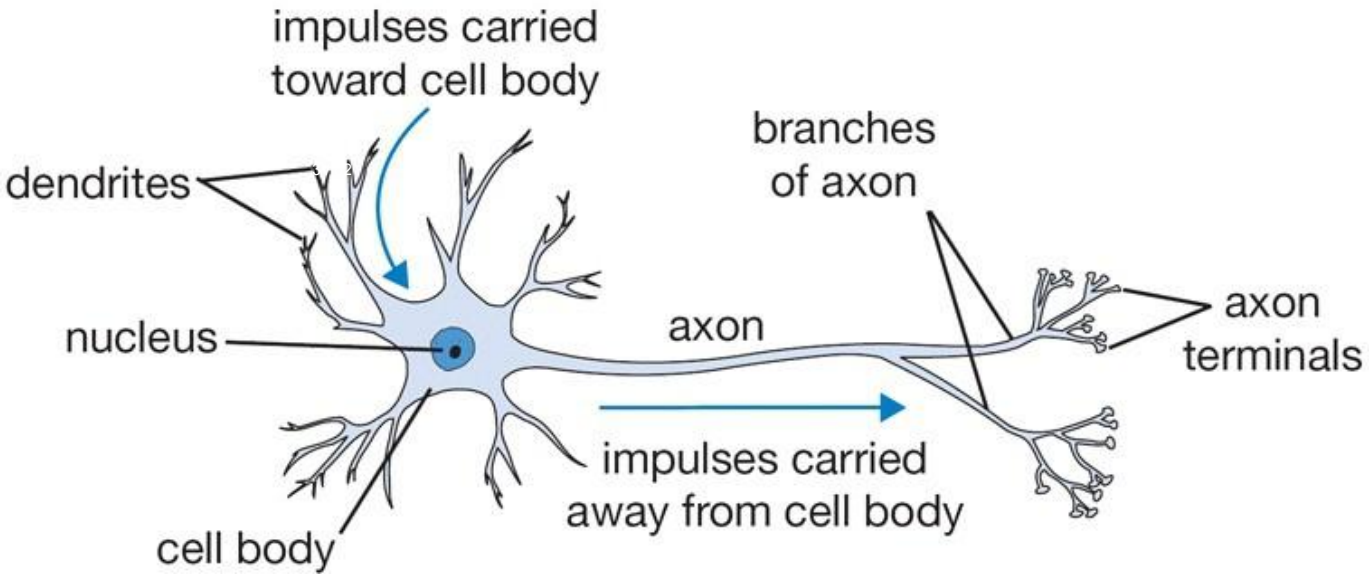


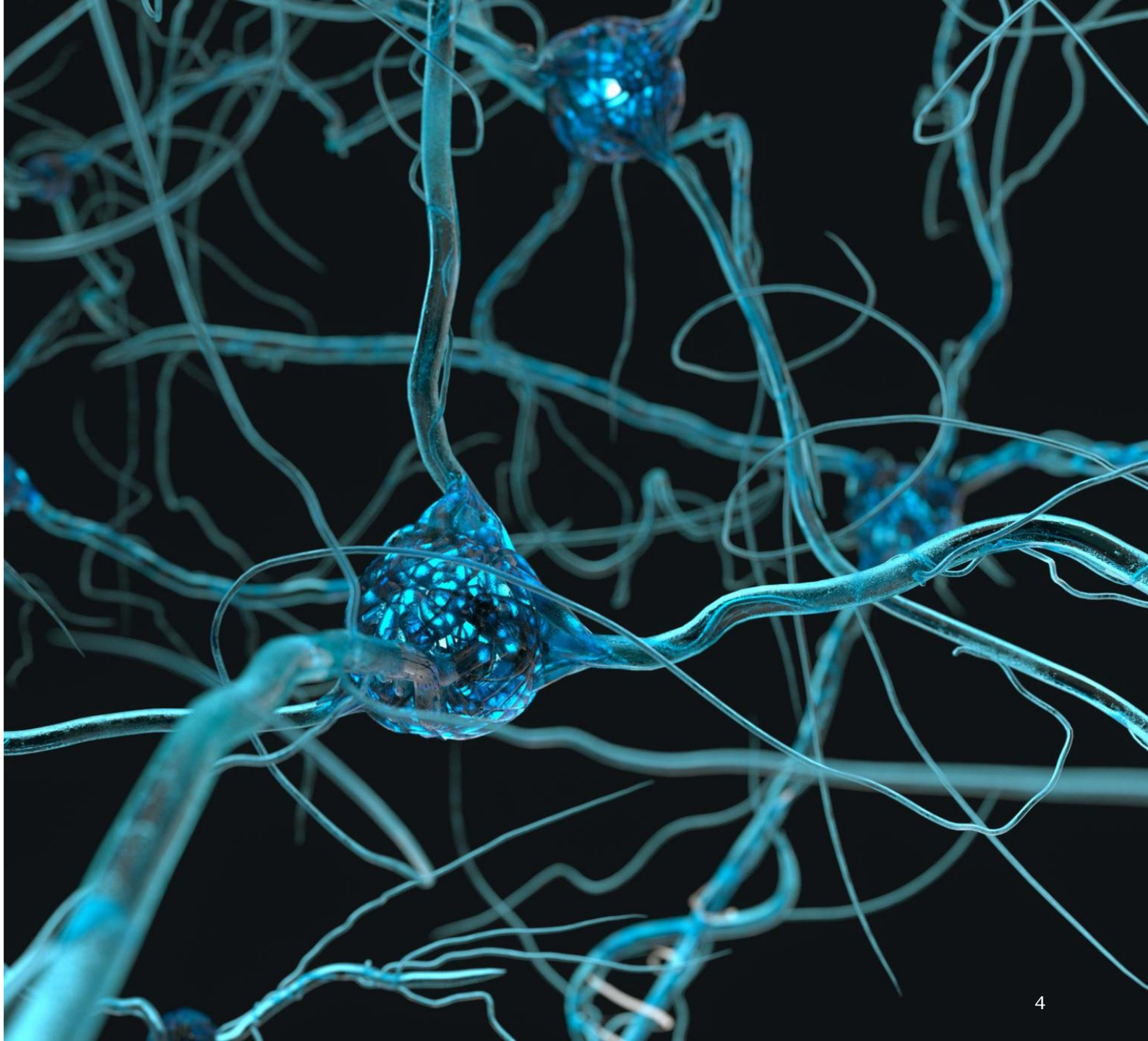
Figure : The basic computational unit of the brain: Neuron

[Pic credit: http://cs231n.github.io/neural-networks-1/]

BYU

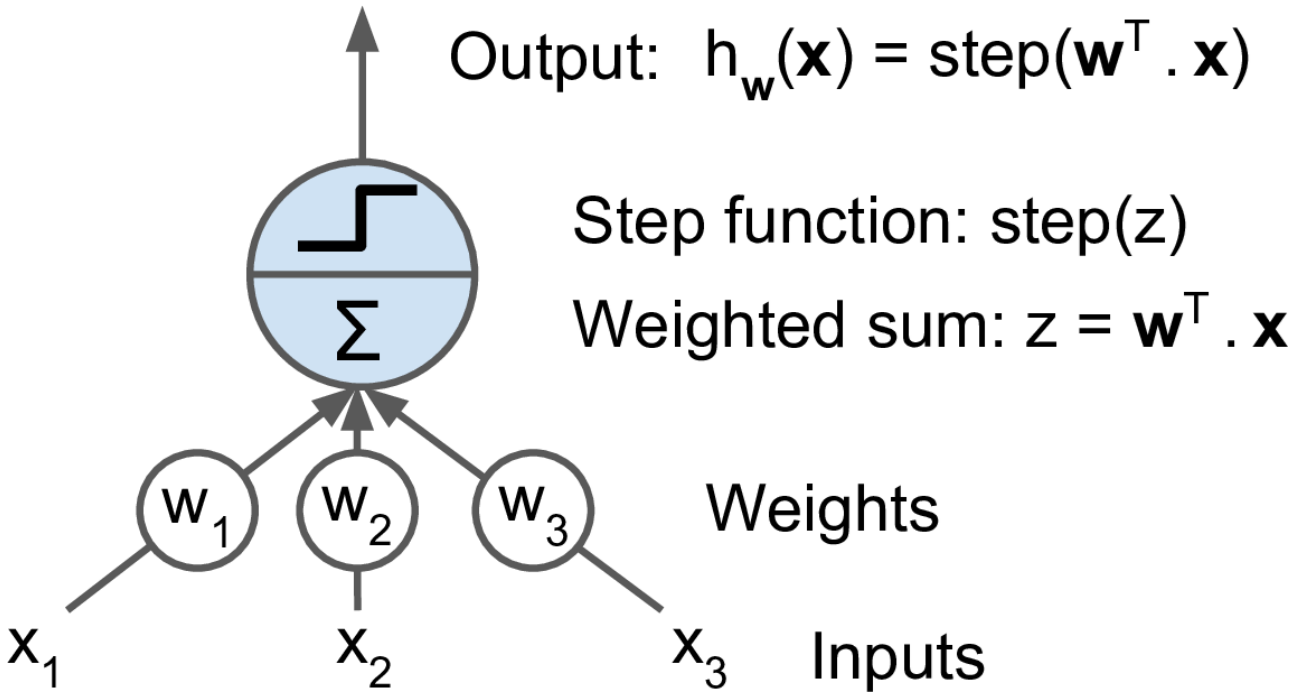
ARTIFICIAL NEURAL NETWORKS

- **Artificial Neural Networks (ANNs)** are computational models inspired by biological neural networks.
- Widely used in machine learning for **classification, regression, clustering, and generative modeling**.
- The earliest models used simple linear functions.



PERCEPTRONS

- **Perceptron** — one of the earliest ANN models (Rosenblatt, 1957).
- Based on a **Linear Threshold Unit (LTU)**.
- No hidden layers.
- LTU computes a weighted sum $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$ ($z = \mathbf{w}^T \mathbf{x}$) and applies a **step function**.
- Often includes a bias term (w_0)
- Works only for **binary, linearly separable** problems.
- Restricted to binary classification of linearly separable problems.



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

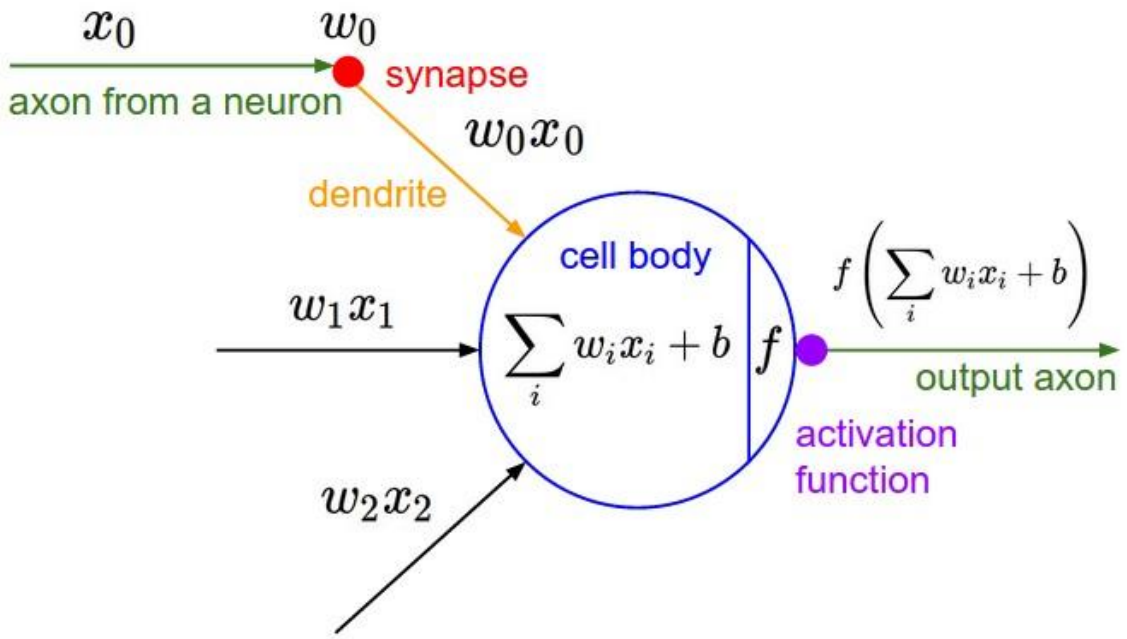
$$\text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

BYU

ARTIFICIAL NEURON STRUCTURE

- **Inputs**: The features or data fed into the neuron.
- **Weights**: Indicate the importance of each input.
- **Bias**: Shifts the weighted sum for flexibility.
- **Activation Function:** Computes output from weighted sum.
- **Output**: The result of applying the activation.
- The formula for the output is:

$$y = f\left(\sum_{i}^{n} w_i x_i + b\right)$$



Example: Making a Class Decision with a Perceptron

You're trying to decide whether to take a class based on the following features:
1. **Is the class required?** (1 = yes, 0 = no)
2. **Is it offered before 9:00 a.m.?** (1 = yes, 0 = no)
3. **Is my girlfriend taking the class?** (1 = yes, 0 = no)

Perceptron Model:

- **x** = input vector = [x1, x2, x3]
- **w** = weights = [w1, w2, w3]
- **b** = bias

$$y = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

This predicts 1 if you decide to take the class, otherwise 0.

BYU

Decision Rule

1. **Is the class required?** (1 = yes, 0 = no)
2. **Is it offered before 9:00 a.m.?** (1 = yes, 0 = no)
3. **Is my significant other taking the class?** (1 = yes, 0 = no)

$$2x_1 - 1.5x_2 + x_3 - 0.5 > 0$$

- If true, take the class
- Else: don't take the class

BYU

Example: Making a Class Decision with a Perceptron (Case 1)

- **Required class:** Yes (1)
- **Before 9 a.m.:** Yes (1)
- **Significant other is in the class:** Yes (1)

$$z = 2(1) - 1.5(1) + 1(1) - 0.5 = 2 - 1.5 + 1 - 0.5 = 1.0 \Rightarrow y = 1 \Rightarrow \textit{Take the class}$$

BYU

Example: Making a Class Decision with a Perceptron (Case 2)

- **Required:** No (0)
- **Before 9 a.m.:** Yes (1)
- **Significant other:** No (0)

$$z = 2(0) - 1.5(1) + 1(0) - 0.5 = -1.5 - 0.5 = -2.0 \Rightarrow y = 0 \Rightarrow \textit{Don't take the class}$$

BYU

Example: Making a Class Decision with a Perceptron (Case 3)

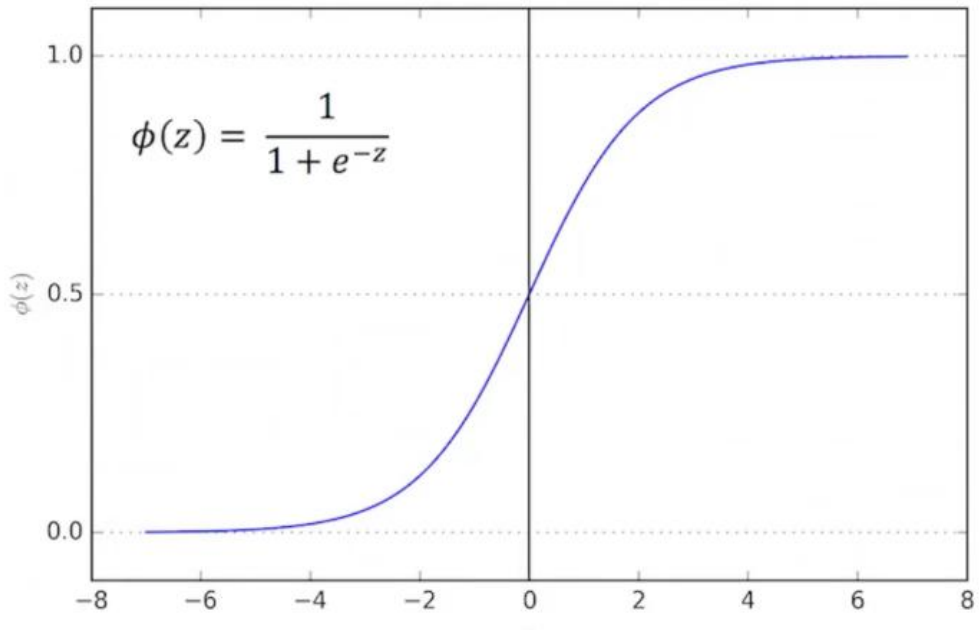
- **Required:** Yes (1)
- **Before 9 a.m.:** No (0)
- **Significant other:** No (0)

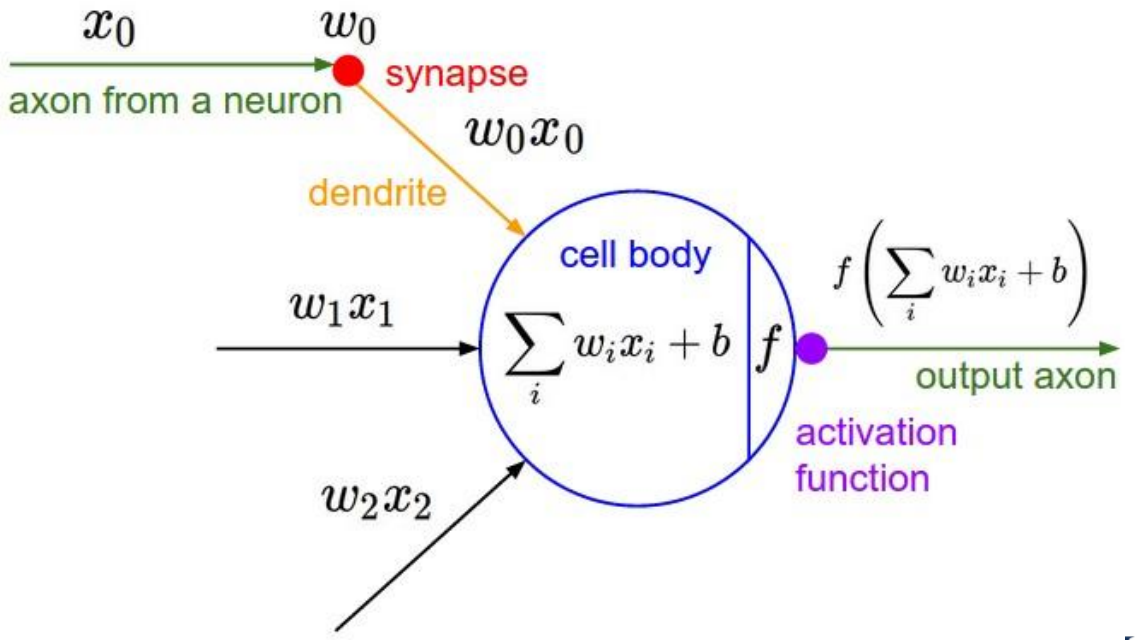
$$z = 2(1) - 1.5(0) + 1(0) - 0.5 = 2 - 0 - 0.5 = 1.5 \Rightarrow y = 1 \Rightarrow \textit{Take the class}$$

BYU

Logistic Regression and MLPs

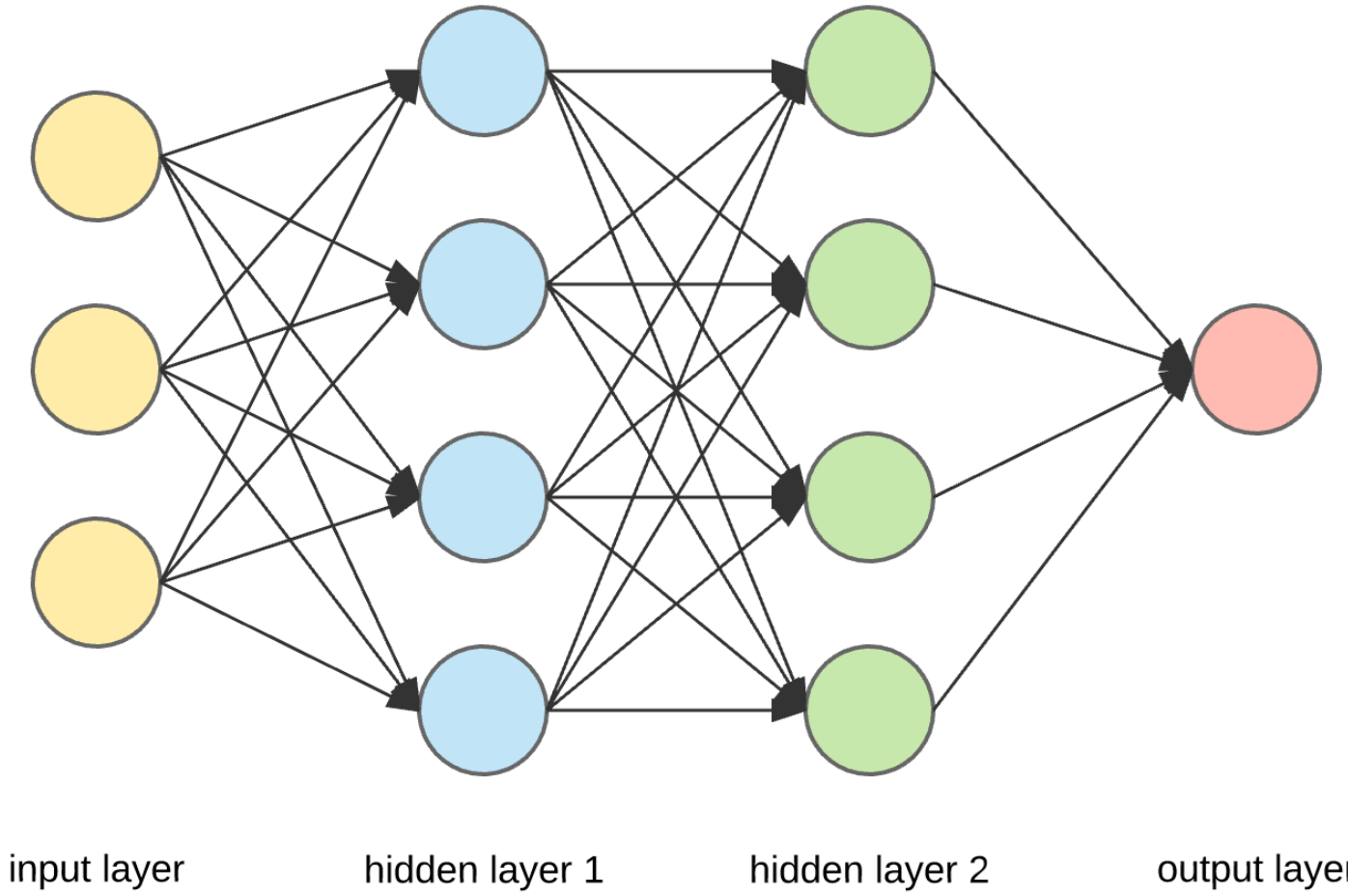
- Multilayer perceptron: add more layers to the perceptron with activations.
- Logistic regression = **no hidden layers**, linear decision boundary with sigmoid (logistic) activation.
- MLPs add **hidden layers** with nonlinear activations.
- Same core idea: weighted inputs → activation → **output**.





NEURAL NETWORK ARCHITECTURE

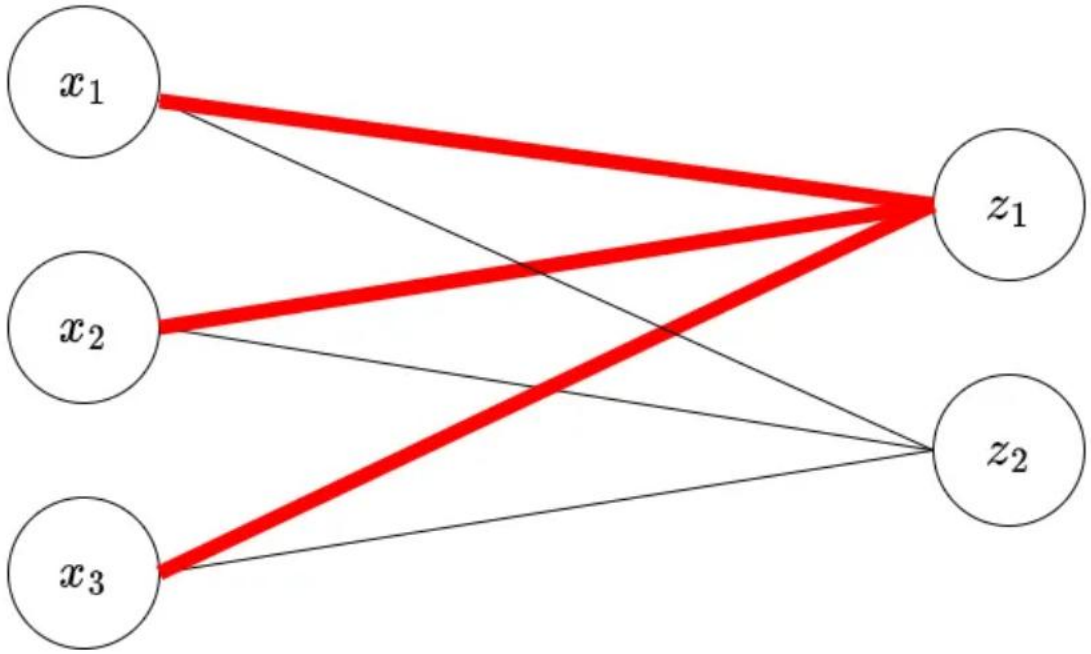
- ANNs are structured into layers:
  - **Input Layer**: Takes the features of the data.
  - **Hidden Layers**: Learn complex patterns.
  - **Output Layer**: Produces the final predictions.
- The number of layers and neurons per layer influences the network's capacity to learn.



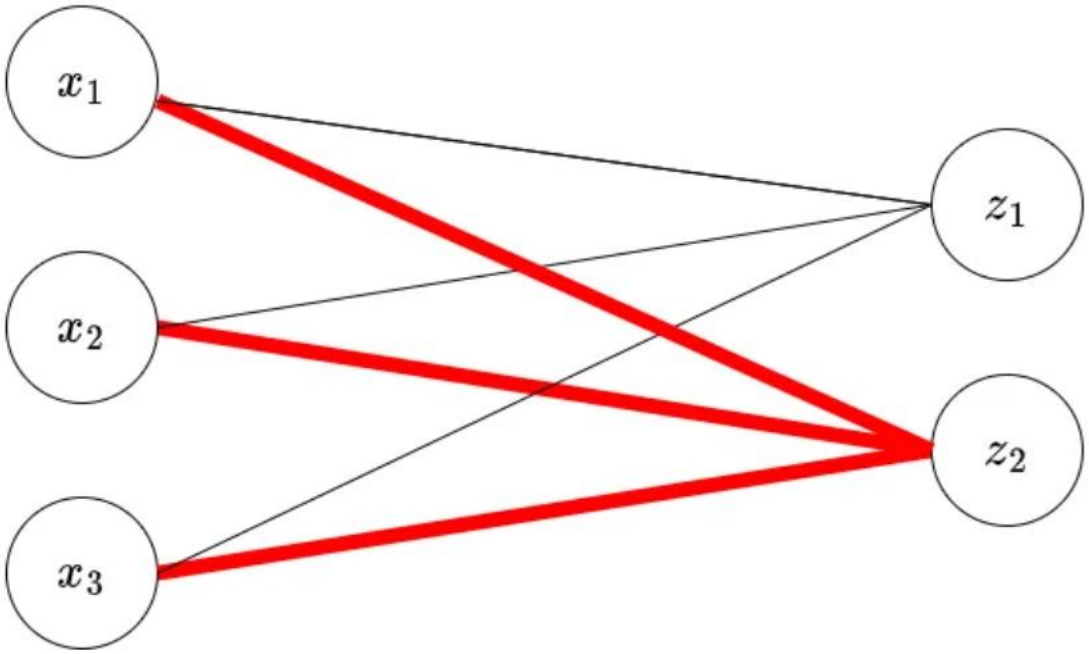
BYU

Weights Represented as Matrices

$$(x_1 \quad x_2 \quad x_3)\begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{pmatrix} = (z_1 \quad z_2)$$



$$(x_1 \quad x_2 \quad x_3)\begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{pmatrix} = (z_1 \quad z_2)$$

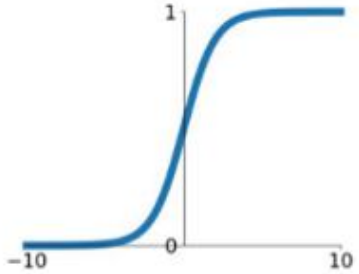


https://medium.com/@funcry/using-matrix-to-represent-fully-connected-layer-and-its-gradient-35604d99d138

ACTIVATION FUNCTIONS

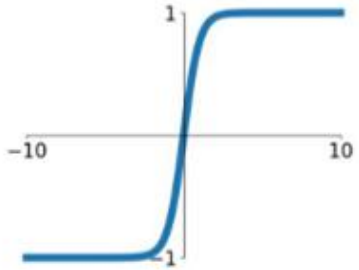
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



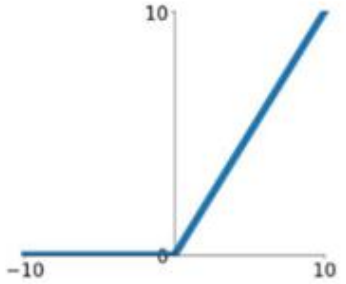
tanh

$$\tanh(x)$$



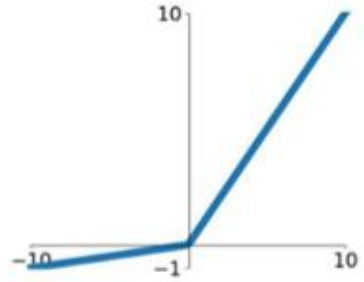
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

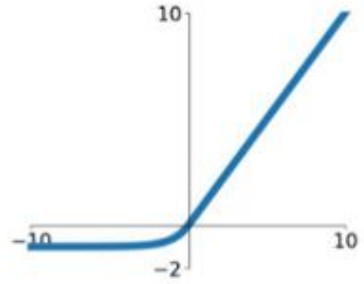


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



BYU

TRAINING PROCESS

1. Forward propagate the input.

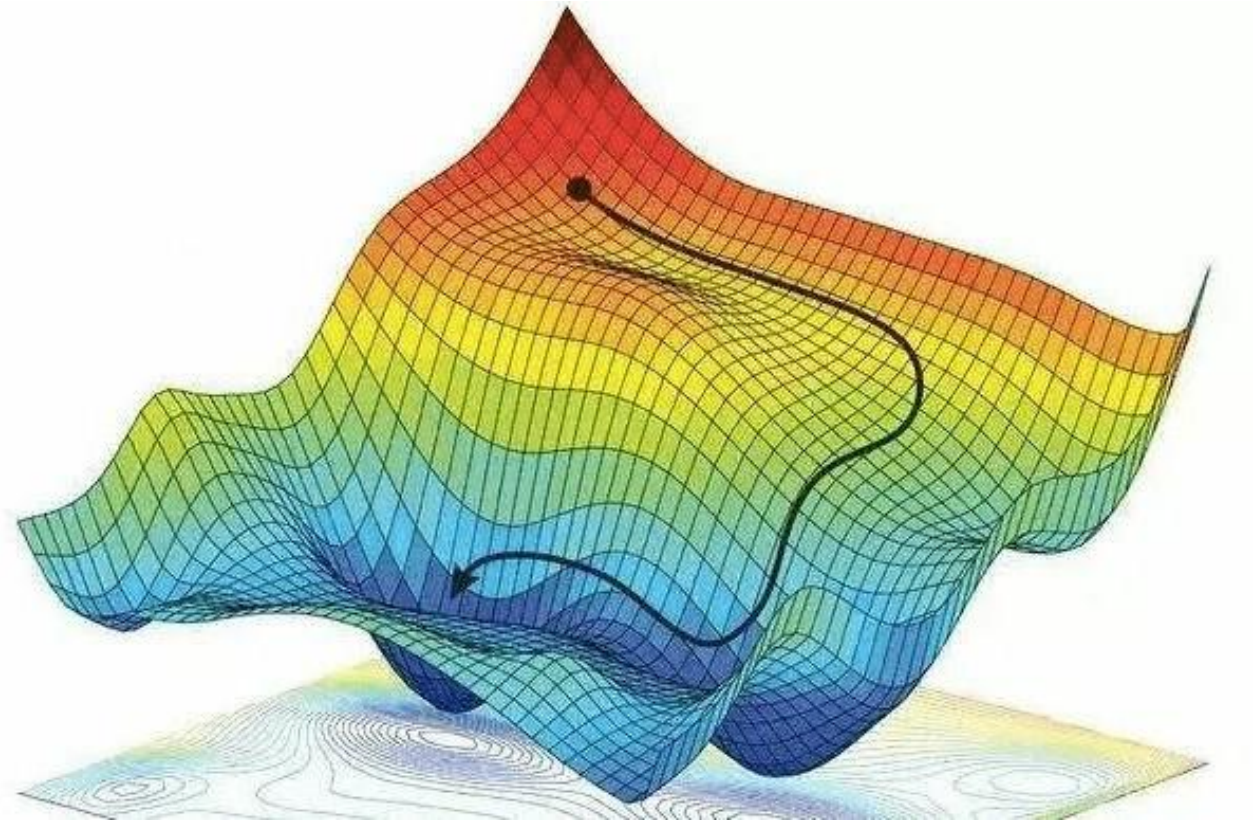
2. Calculate the loss.

3. Backpropagate the error to update weights and biases.

4. Repeat for multiple epochs until loss below threshold.

Weights and biases are updated to minimize a cost function using an optimization process (e.g., gradient descent)

Gradient descent can be visualized as finding the most direct path to the bottom of a canyon (see figure)



BYU

FORWARD PROPAGATION

The process of passing input data through the network to compute the output.

Steps:

1. Multiply inputs with weights and add biases.
2. Apply activation functions at each neuron.
3. Pass the outputs to the next layer.

```python
def forward_propagation(inputs, weights, biases, activation_function):
    layer_output = inputs
    for w, b in zip(weights, biases):
        layer_output = activation_function(np.dot(layer_output, w) + b)
    return layer_output

# Example with 2 layers
weights = [np.array([[0.2, 0.8], [-0.5, 0.3]]), np.array([[0.7], [-1.2]])]
biases = [np.array([0.1, -0.3]), np.array([0.5])]
inputs = np.array([1.0, -0.5])

output = forward_propagation(inputs, weights, biases, sigmoid)
print(f"Network output: {output}")
```

```
Network output: [0.55970227]
```

BYU

BACKPROPAGATION

- Backpropagation trains neural networks by reducing the difference between predicted and actual outputs.
- It works by calculating how much each weight and bias affects the error.
- Errors are sent backward through the network to update weights and biases.
- Uses the chain rule from calculus to compute gradients efficiently.

Steps

1. **Error Calculation**: Compute loss using a function like Mean Squared Error (MSE).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{actual} - y_{pred})^2$$

2. **Gradient Calculation**: Use the chain rule to calculate gradients of weights and biases.

- $$\frac{\partial Loss}{\partial w_k} = \frac{\partial Loss}{\partial z} * \frac{\partial z}{\partial w_k}$$

Where:

$$z = \sum w_i x_i + b$$

3. **Weight Update**: Update parameters using gradient descent:

$$w = w - \eta \frac{\partial\, Loss}{\partial w}$$

where η is the learning rate.

PYTHON EXAMPLE

Explanation of Key Parameters

- hidden_layer_sizes: Tuple specifying the number of neurons in each hidden layer. In this case, two layers with 10 neurons each.
- activation: Activation function for the neurons.
- Options: 'identity', 'logistic', 'tanh', 'relu'.
- solver: Optimization algorithm.
  - Options: 'lbfgs', 'sgd', 'adam'.
- max_iter: Maximum number of iterations for training.

```
# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Standardize the data (important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize the MLPClassifier
mlp = MLPClassifier(
    hidden_layer_sizes=(10, 10), # Two hidden layers with 10 neurons each
    activation='relu',           # Activation function: ReLU
    solver='adam',               # Optimization algorithm: Adam
    max_iter=500,                # Maximum number of iterations
    random_state=42
)

# Train the model
mlp.fit(X_train, y_train)

# Make predictions on the test set
y_pred = mlp.predict(X_test)
```

https://colab.research.google.com/drive/18zuW6teHF4of6dtXNiVG7eoshursaifN?usp=sharing

BYU

Scikit Learn Optimizers (MLP)

Aspect	Gradient Descent (GD)	L-BFGS	Adam
Type	First-order (uses gradient only)	Quasi-Newton (approximates 2nd order)	Adaptive first-order
Update Rule	$w_{t+1} = w_t - \alpha \nabla f(w_t)$	$w_{t+1} = w_t - \alpha H_t^{-1} \nabla f(w_t)$	Uses moving averages of grad & grad²
Batch Type	Full-batch	Full-batch	Mini-batch
Learning Rate Sensitivity	High	Low (uses line search)	Moderate (auto-adjusted)
Speed per Iteration	Fast	Slower (costly updates)	Fast
Convergence Quality	Moderate	Very precise (near optimum)	Fast but less precise
Best For	Simple convex problems	Small/smooth problems	Large or deep networks
Memory Use	Low	Moderate	Low

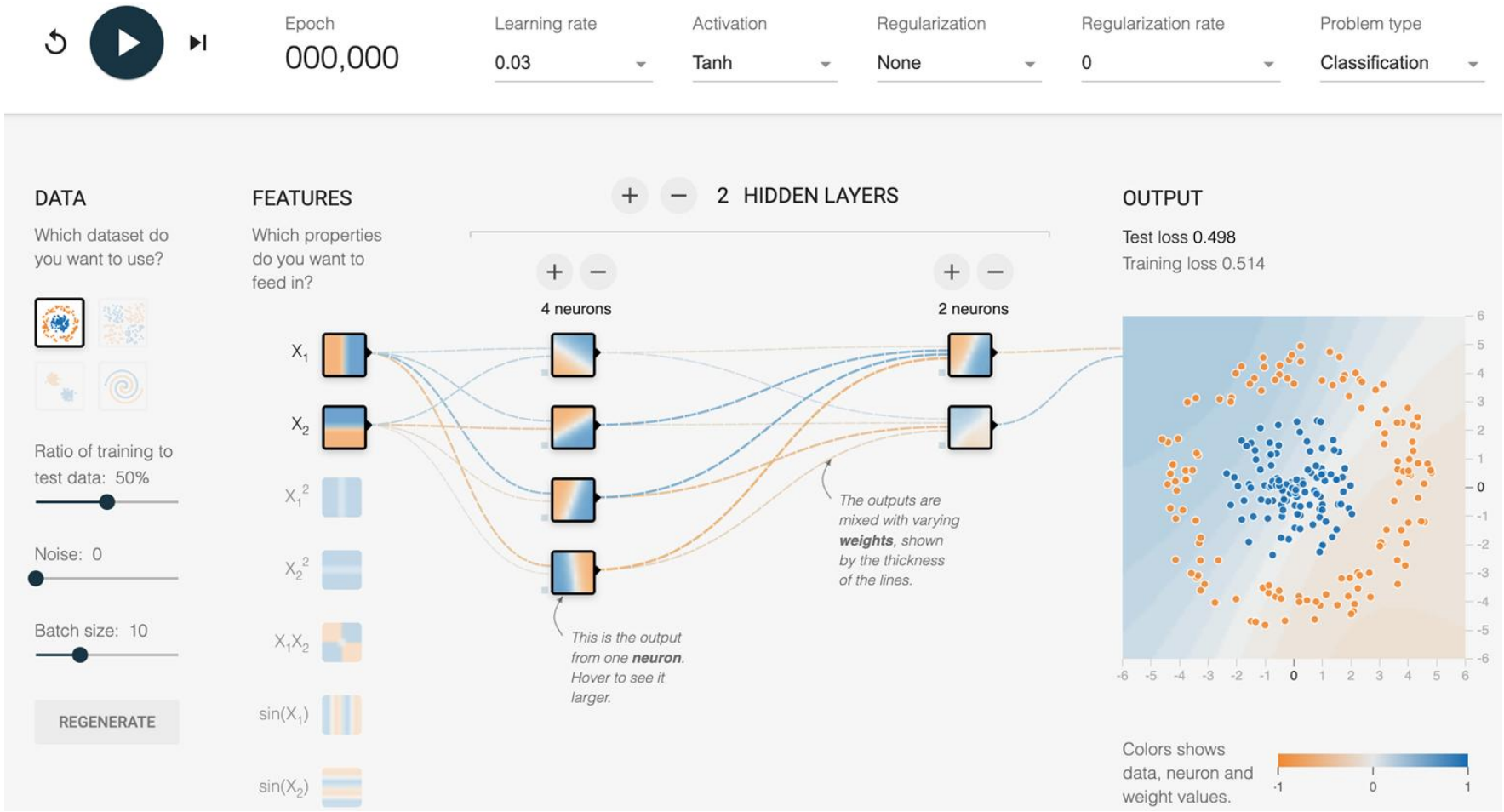
BYU

Activation Function Tables (Scikit Learn Options)

Activation	Formula / Range	Shape & Properties	Pros	Cons	When to Use
Identity	$(f(x) = x)$ $Range: ((-\infty, \infty))$	Linear	Simple, interpretable	No nonlinearity → can't learn complex patterns	Linear regression, output layer for regression
Logistic (Sigmoid)	$\left(f(x) = \frac{1}{1+e^{-x}}\right)$ $Range: ((0,1))$	S-shaped, saturating	Probabilistic output, smooth	Vanishing gradients, not zero-centered	Binary classification output layer
Tanh	$(f(x) = \tanh(x))$ $Range: ((-1,1))$	S-shaped, zero-centered	Better gradient flow than sigmoid	Still saturates, costly to compute	Hidden layers in small networks
ReLU	$(f(x) = \max(0, x))$ $Range: ([0,\infty))$	Piecewise linear	Fast, sparse activations, avoids vanishing gradients	Dying ReLU problem (neurons stuck at 0)	Default for deep networks, image tasks

BYU

EXPERIMENT WITH NEURAL NETWORK PLAYGROUND



https://playground.tensorflow.org/

UNIVERSAL APPROXIMATION THEOREM

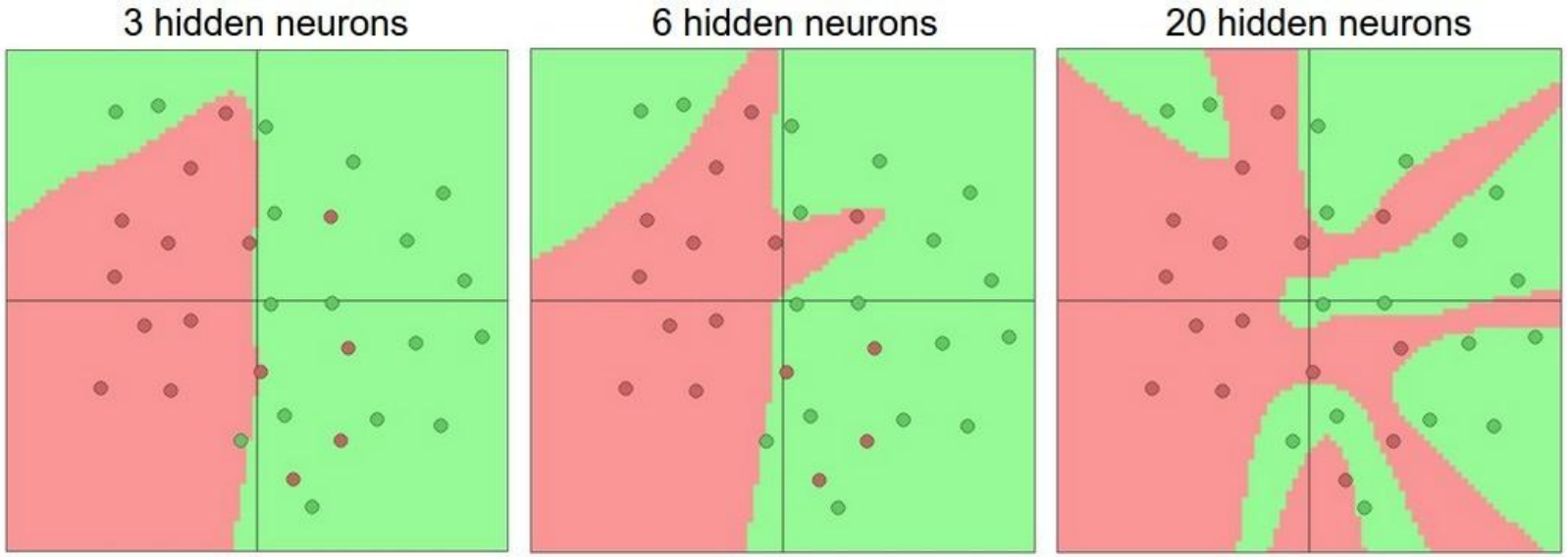
- A neural network with one hidden layer and non-linear activations can approximate any continuous function.
- **Single Hidden Layer:**
  - One hidden layer is enough for approximation.
  - May need many neurons → can be inefficient.
- **Practical Notes:**
  - The theorem doesn't say how many neurons are needed.
  - Deeper networks are often more efficient.
  - Doesn't specify how to find the approximator.
- **Limitations:**
  - **Size:** May need too many neurons → inefficient or overfit.
  - **Training:** Assumes perfect training, which is hard in practice.
  - **Discontinuities:** Only applies to continuous functions.

BYU

REPRESENTATIONAL POWER

Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper



The capacity of the network increases with more hidden units and more hidden layers

Why go deeper? Read e.g.,: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana, Paper: paper]

[http://cs231n.github.io/neural-networks-1/]

BYU

GLOSSARY

Activation Function: A mathematical function that introduces non-linearity into a neural network.

Backpropagation: The central algorithm for training artificial neural networks. It uses gradient descent to minimize the difference between the network's predictions and the actual target values.

Bias: A value added to the weighted sum of inputs for each neuron. Biases help shift the activation function, allowing for more flexibility in the network's learning.

Cost Function: Also known as the loss function, this function measures the difference between the network's predictions and the target values.

Gradient Descent: An optimization algorithm that iteratively adjusts the weights and biases of a neural network to minimize the cost function. It calculates the gradient of the cost function with respect to each parameter and updates the parameters in the direction of the negative gradient.

Hidden Layers: Layers in a neural network between the input and output layers.

Input Layer: The first layer in a neural network that receives the initial data.

Logistic Function/Curve: Also known as the Sigmoid function, this function squashes the output of a neuron to a value between 0 and 1. It's often used in the output layer for binary classification tasks.

Multi-layer Perceptron (MLP): A type of feedforward neural network with one or more hidden layers.

Neuron: The basic building block of a neural network. Each neuron receives input from other neurons, performs a weighted sum of its inputs, applies an activation function, and passes the output to other neurons.

Output Layer: The final layer in a neural network that produces the network's predictions or classifications.

ReLU (Rectified Linear Unit): An activation function that outputs 0 for negative inputs and the input value for positive inputs. It's a popular choice for its computational efficiency and effectiveness in many applications.

Softmax Function: An activation function that outputs a probability distribution over multiple classes. Typically used in the output layer for multi-class classification tasks.

Tanh (Hyperbolic Tangent): An activation function similar to the Sigmoid function but squashes the output to a value between -1 and 1.

Weights: Numerical values that determine the strength of connections between neurons. They are adjusted during the learning process to minimize the cost function.

REFERENCES

- 3blue1brown Overview of Neural Networks: https://www.3blue1brown.com/lessons/neural-networks
- https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- Understanding Backpropagation: https://towardsdatascience.com/understanding-backpropagation-abcc509ca9d0
- Backpropagation – A Visual Walkthrough: https://www.youtube.com/watch?v=9d2fwGjyb4M
- Multilayer Perceptron Classifier (MLPClassier): https://michael-fuchs-python.netlify.app/2021/02/03/nn-multi-layer-perceptron-classifier-mlpclassifier/

BYU