Module 2: Deep Learning, Opening the Machine

In our previous module, we embarked on an introductory journey through the basics of deep learning. As we progress in our exploration, it's time to dive deeper into the core building blocks of these systems. This module gives a closer look at neurons, weights, biases, and activation functions that play pivotal roles in models. We use the example of image recognition using a relatively small neural network. We'll tie it together with a practical demonstration in a Jupyter Notebook. Fasten your… flippers?

Module 2 Course Objectives:

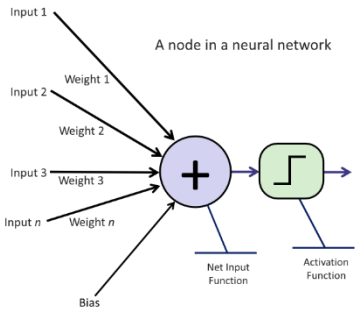
By the end of this module, you will be able to:

1. Describe the basis of a neural network (neuron).
2. Identify and describe an artificial neuron (perceptron).
3. Discuss bias and weights.
4. Describe and identify activation functions.
5. Describe and simulate image processing in a small neural network.
6. Implement and train a perceptron using TensorFlow.

Neural Network Neurons

Neurons, often called nodes, are a neural network's fundamental building blocks. Drawing inspiration from biological neurons in the human brain, these computational units are designed to process information in a way that mimics how our brains function.

At its core, a neuron consists of:



1. **Inputs**: Each neuron receives one or more inputs. These can be raw data values (in the input layer) or the outputs from neurons in previous layers.
2. **Weights**: Associated with each input is a weight. Weights are crucial numerical parameters in neural networks and determine the importance or influence of a particular input on the neuron's output. They are adjusted during the training process (specifically, during **back-propagation**) to minimize the error in the network's predictions. Back-propagation and training in general are explained in more detail later in this module. The inputs and weights are multiplied and added together.
3. **Bias**: This numerical parameter is added to the sum of all the input by weight products. The bias allows the neuron to shift its sum, enabling more complex representations. It is like an intercept in linear regression.
4. **Activation Function**: After summing the weighted inputs and the bias, the result is passed through an activation function. This function can introduce non-linearity to the model, allowing the network to capture complex relationships in the data.

Neuron Function

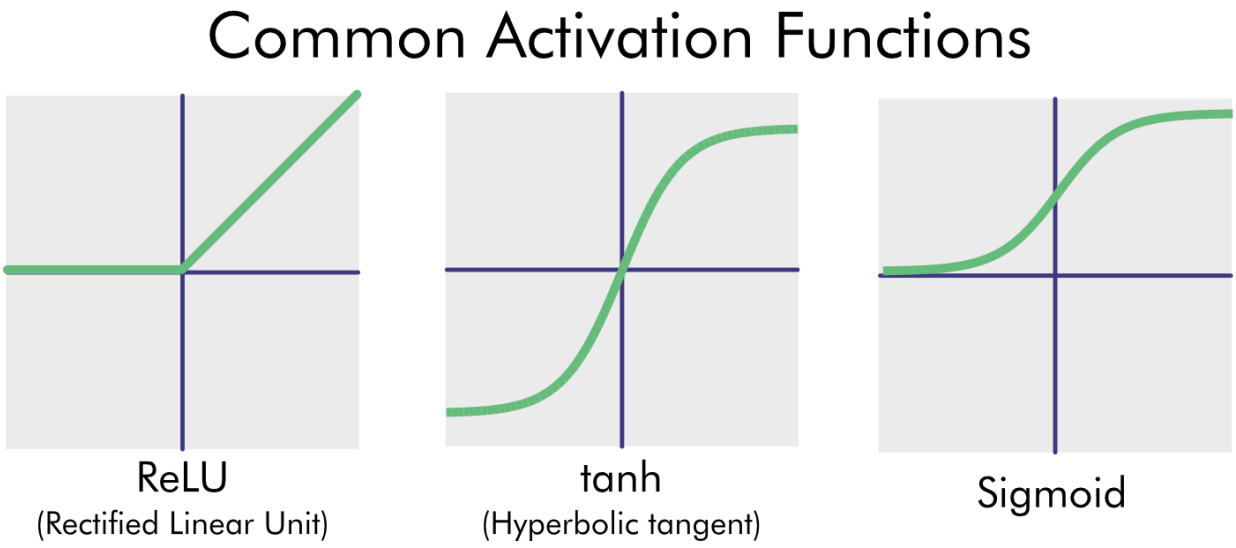
A neuron performs a two-step process:

1. **Linear Transformation:** It calculates a weighted sum of its inputs and adds the bias term. Mathematically, this can be represented as:

$$Sum = w_1 \times x_1 + w_2 \times x_2 + \cdots + w_n \times x_n + bias$$

Where w_i are the weights and x_i are the inputs.

2. **Activation**: The result from the above step is then passed through an activation function, producing the neuron's output. Common activation functions include the sigmoid, hyperbolic tangent (tanh), and Rectified Linear Unit (ReLU). Graphs of these are below. The activation function to use is one hyperparameter. Empirically, the ReLU has become popular because it works well in many situations and is computationally quick to calculate.



Functions, ACTIVATE!

As mentioned above, activation functions introduce non-linearity to the model, allowing neural networks to capture complex relationships and patterns in the data. Without them, no matter how deep the network, it would behave like a linear model, limiting its predictive power. As an analogy, imagine building a complex structure out of blocks. If you only use straight pieces, you're limited in what shapes you can create. But introduce some curved or angled pieces (akin to activation functions), and suddenly, you can build intricate and varied designs. Here are some examples of some of the more popular activation functions.

1. **Step Function:** Our first activation function is the simple step function. When its input is less than or equal to zero, the perceptron outputs 0. If the input becomes positive – even by a tiny amount – the perceptron outputs a 1. But this sudden and extreme transition is not ideal for training. Essentially, the neuron has no finesse – it's either yelling or silent. Step functions only work for modeling single, straight lines like those in binary classifiers and are rarely used in deep learning.
2. **ReLU (Rectified Linear Unit) Function:** The ReLU (rectified linear unit) function outputs 0 for all negative inputs; otherwise, the output is the input. The ReLU activation function has recently become quite popular because it's simple and trains exceptionally well. The angles it outputs

allow it to model linear functions and make excellent approximations of curves if there is enough capacity in the network.

3. **Sigmoid (or Logistic) Function:** The sigmoid function – also known as the logistic function – returns a 0 value for extremely negative inputs and a value of 1 for extremely positive inputs. This function is called sigmoid because it resembles the curve of an S shape. Because of its binary output, this function works well for binary classifiers that are separated by a curved line rather than a straight one.
4. **tanh Function**: The output of the tanh is a hyperbolic tangent and looks like the sigmoid function. It has that same S-shape. But there are differences. For example, the TANH function returns a value of −1 for extremely negative inputs.

Where Do Weights and Bias Come From?

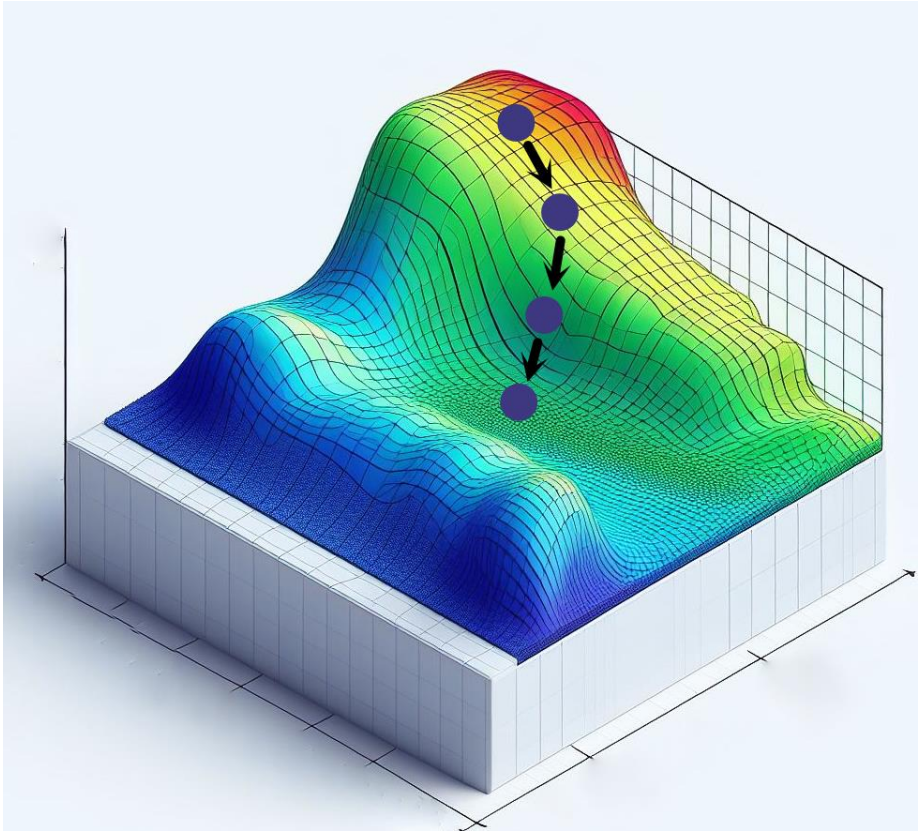
We make them up! At first, at least. Weights and biases are initialized, often with some form of randomness, and then iteratively adjusted using the training data to minimize the **network's prediction error**. This adjustment process uses the **gradients** derived from the **loss function** to guide the updates.

1. **Random Initialization**: Weights are typically initialized with small random values. This breaks the symmetry, ensuring that each neuron learns something different from the start. If all weights started with the same value, all neurons in each layer would update identically, making them redundant.
2. **Zeroes Initialization:** Setting all initial weights to zero is also an option. Choosing to initialize with all zeroes doesn't work with certain activation functions, such as ReLU as it passes all zeroes forward as... zero, effectively "killing" your nodes.
3. **Bias Initialization**: Biases can be initialized to zero or small values. Starting with zero is often acceptable since the random weights already introduce the asymmetry.
4. **Advanced Initialization Techniques:** There are methods for initializing starting weights and biases that aren't random, but we won't go into that much here. Later, we will explore the powerful concept of **Transfer Learning** that uses this concept.

The Training Process

1. **Forward Pass:** Input data is passed through the network layer by layer, using the current weights and biases to produce a prediction. In the last module, we had a model that could predict the class of a picture fed to it. During the training process, the forward pass was feeding images into the model and letting it guess what it "thought" it was "seeing."
2. **Loss Calculation:** The network's prediction is compared to the actual target values using a loss function, quantifying how far off the predictions are. In the pizza recognizer example, if the model guessed that a picture of a pizza was a "cat," the loss function would tell the model it needed to adjust its weights and biases and try again.
3. **Backward Pass (Backpropagation):** The **gradient** of the loss function concerning each weight and bias is computed. We aren't going to dive into the details, but this uses partial differential equations (from that Calculus class you may have taken long ago) to find the gradients (or tangents). These gradients indicate the direction and magnitude to adjust each parameter (weights and biases) to minimize the loss calculated by the loss function.

4. **Optimization:** Weights and biases are adjusted to decrease the loss. The learning rate, a hyperparameter, determines the size of the adjustments. This step is the secret sauce for how networks learn: With each backward pass, the network (hopefully) gets a little more accurate with its predictions.
5. **Iterative Process:** The forward pass, loss calculation, backpropagation, and optimization steps are repeated multiple times (epochs) on the training data until the network converges to an optimal set of weights and biases or until a set number of epochs is reached.



The 3D surface in this figure represents the loss surface for different sets of weights. While we draw this in 3D, this surface is highly dimensional, with as many dimensions as there are parameters (millions!). At each epoch, the loss is calculated. Each blue dot is the position on the loss surface at an epoch. The gradient and learning rate are used to calculate the direction and amount each parameter should be adjusted. The training data are re-evaluated, and the new loss is calculated. This process iterates repeatedly until the loss reaches a satisfactory level.

Model Deployment

Once a model is trained with its parameters adjusted, it can be **deployed**. The model is made available in a production environment to make predictions on new, unseen data, fed as input to the trained

model. This could be on a server in a Jupyter Notebook, a cloud platform, a mobile app, or any system where real-time or batch predictions are needed based on the model.

Why Do We Use Neurons?

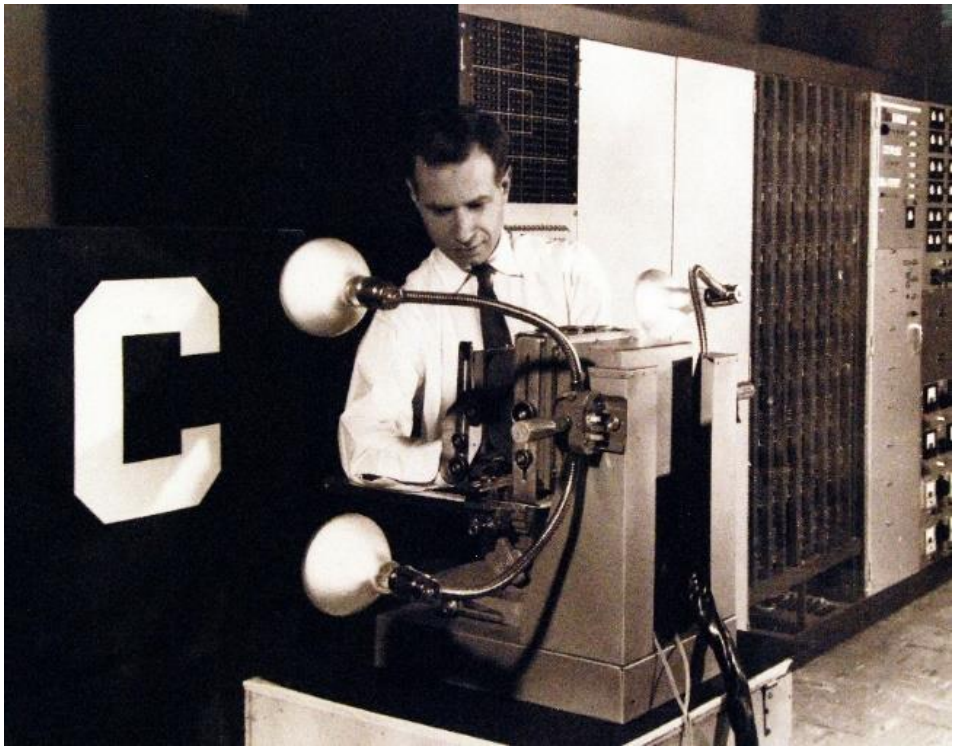
Individual neurons' simplicity and adaptability make them incredibly powerful when combined in layers to form neural networks. One of the most remarkable properties of neural networks is their ability to serve as "**universal approximators**." Theoretically, a neural network (even with a single hidden layer, given enough neurons) can approximate *any* continuous function to a desired level of accuracy. In other words, they can model and represent a vast range of intricate patterns, behaviors, and relationships in data.

The concept of universal approximation underscores the potential and versatility of neural networks. By adjusting the weights and biases during training, a neural network doesn't just adapt—it can model complex, non-linear relationships in the data, making it a tool of choice for numerous applications ranging from image recognition to financial forecasting.

While the theory assures us of this capability, creating and training such networks efficiently for complex tasks in practice often requires deeper architectures (deep learning) and sophisticated training techniques.

The Perceptron

As we mentioned in our Getting Started with AI course, the perceptron was developed by Frank Rosenblatt in the late 1950s and is one of the simplest forms of a neural network. It can be considered the building block or the "atom" of more complex neural networks. It’s essentially a single-layer neural network. Functionally, perceptrons work as binary classifiers, meaning they can categorize inputs into one of two classes. This is because, with a single neuron, perceptrons can't handle data that can’t be



separated by a single straight line (or hyperplane in higher dimensions). This limitation in **network capacity** inspired the creation of linking perceptrons together, creating networks... Neural networks! In this module's exercise, we will create our own perceptron and take it for a spin!

From Wikimedia Commons:

This photo shows the Mark I Perceptron, an experimental machine which can be trained to automatically identify objects or patterns, such as letters of the alphabet. Originated by Dr. Frank Rosenblatt, a psychologist who is in charge of the program at Cornell Aeronautical Laboratory, Buffalo, New York, Mark I is an electromechanical device consisting basically of a "sensory unit" of photocells which contain the machine's memory, and response units which visually display the machine's pattern recognition response. The machine is generally trained by placing the test patterns, which could be letters of the alphabet of a single-type face, in the view field of the perceptron's photoelectric cell "eye." When the machine incorrectly identifies a pattern or letter, the trainer forces it to respond correctly by means of an electrical control. When the training is completed the letters of the particular type face can then be shown to the machine's eye, and it will correctly identify the letters without error. When the recognition problem has been complicated by adding letters of a different type face, the machine has been correct 85 percent of the time. The perceptron has particular potential use in the processing of non-numerical information for the solution of scientific, engineering, and military problems. The perceptron research program is sponsored by the Office of Naval Research with the assistance from Rome Air Development Center, Rome, New York, of the Air Research and Development Command. Photograph released June 24, 1960.

Coding a perceptron

The perceptron is relatively simple—only one neuron to worry about! If this simple network seems complicated to code, don't worry! We are doing most of the coding from scratch in this exercise. This will help you see how neurons work and follow individual components through the training process. After this notebook though, we'll leave the details to the AI frameworks and use APIs that make all of this much easier!

Work through notebook **02.1_code_a_perceptron.ipynb**.

Network Capacity

Network capacity refers to the complexity or flexibility of a neural network to capture patterns in data. A network with higher capacity can model more intricate relationships but is also more prone to **overfitting**, meaning it might perform well on training data but poorly on unseen data. Factors affecting capacity include:

- The number of layers

- The number of neurons in each layer
- The type of activation functions used

Image Processing

Due to the magic of universal approximation, the tools that let us program a perceptron can be expanded to recognize images! In the following Jupyter Notebook, we'll go more in-depth with how to create a model from layers and train it!

Work through notebook **02.2_mnist_classifier.ipynb.**