**A neural network** is a computational model inspired by the structure and functioning of the human brain. It consists of interconnected nodes called neurons, organized in layers. Each neuron receives input signals, performs a computation, and produces an output signal.

Here are the basic components and steps involved in building a neural network from scratch:

* Neuron: A neuron is a fundamental building block of a neural network. It takes multiple inputs, applies a weighted sum, adds a bias term, and applies an activation function to produce an output. The activation function introduces non-linearity into the network, allowing it to model complex relationships.
* Layers: Neurons are organized into layers. A neural network typically consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the raw input data, and the output layer produces the final predictions or outputs of the network. The hidden layers are intermediate layers that perform computations and extract features from the input.
* Weights and Biases: Each connection between neurons has an associated weight and bias. The weights represent the strength of the connections, and the biases act as offsets, allowing neurons to activate even when the inputs are zero. These weights and biases are learnable parameters that the network adjusts during training to improve its performance.
* Forward Propagation: In the forward propagation step, the inputs are passed through the network in a sequential manner, starting from the input layer. Each neuron computes its output based on the weighted sum of the inputs, adds the bias term, and applies the activation function. The outputs of one layer serve as inputs to the next layer until the final output is generated.
* Loss Function: A loss function measures the difference between the network's predictions and the true values. It quantifies the network's performance on a given task. Common loss functions include mean squared error (MSE) for regression problems and categorical cross-entropy for classification problems.
* Backpropagation: Backpropagation is a learning algorithm that calculates the gradients of the loss function with respect to the weights and biases of the network. It propagates these gradients backward through the network, updating the parameters in the opposite direction of the gradient. This step enables the network to adjust its weights and biases to minimize the loss and improve its predictions.
* Training: During the training phase, the network iteratively performs forward propagation to make predictions, calculates the loss, and performs backpropagation to update the weights and biases. This process continues for multiple epochs or until the network's performance converges to a satisfactory level.
* Testing and Evaluation: Once the network is trained, it can be used to make predictions on new, unseen data. The performance of the network is evaluated using various metrics, such as accuracy, precision, recall, or mean squared error, depending on the task.

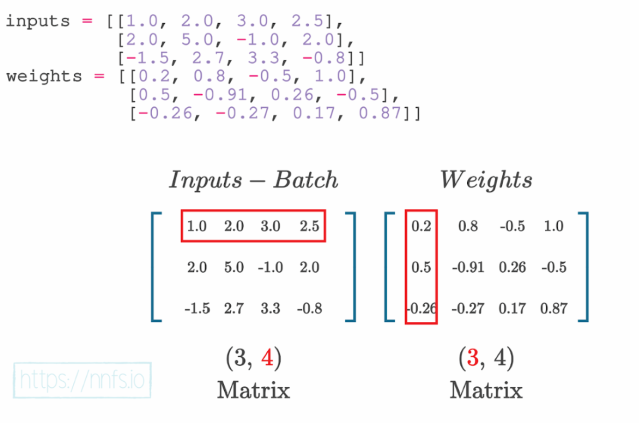
This is a high-level overview of building a neural network from scratch. In practice, there are many variations and advanced techniques that can be applied to improve network performance, such as regularization, different activation functions, and optimization algorithms.

**Building neural network from scratch**

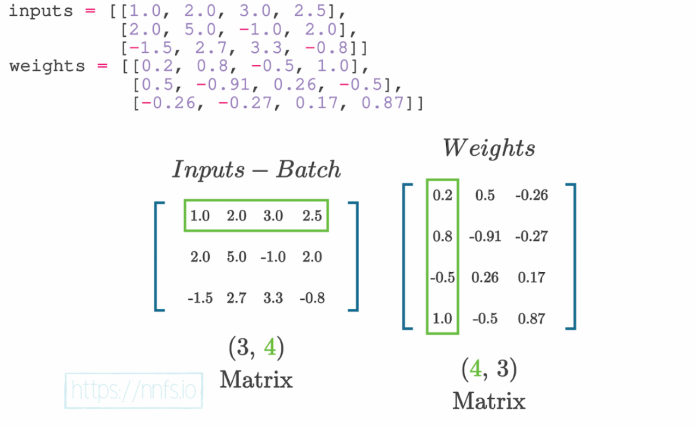
**Simple neural network**: see the Jupiter **ipynb** code

Now, set up more complex neural network design: A Layer of Neurons & Batch of Data w/ NumPy

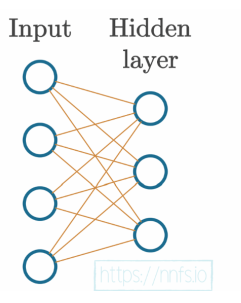
Let’s get back to our inputs and weights —​when covering them, we mentioned that we need to perform dot products on all of the vectors that consist of both input and weight matrices. Can we multiply this?



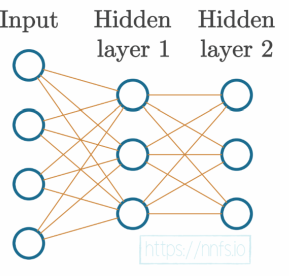
Definitely not, we need to transpose the second matrix and make it to the same size as inputs.



As we have shown how to work on single layer network, now, it’s time to add layers in the network. Before we add another layer, let’s think about what will be coming. In the case of the first layer, we can see that we have an input with 4 features.

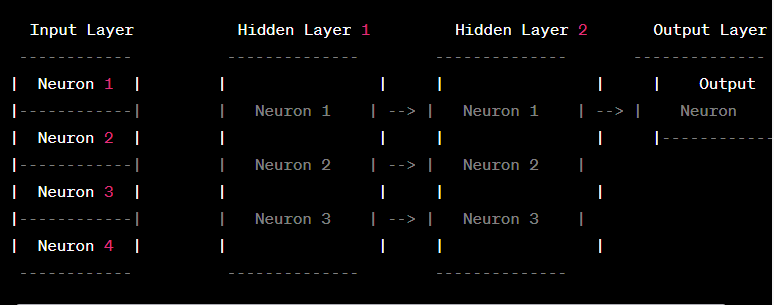


Now, we want to add another layer to make it multi-layer network which looks like this:



**Lab exercise:**

* Design the previous examples of neural network by adjusting weights and biases randomly.
* Design a neural network that has four input neurons, two hidden layers, and one output layer neuron. The activation function in the output payer should be sigmoidal. You should also adjust the weights and biases randomly. Moreover, the input dataset should contain 10 samples. Follow the figure:



Neural network with loss calculation

The choice of the best loss function for a neural network depends on the specific problem you are trying to solve. Different loss functions are designed for different types of problems and have different properties. Here are some commonly used loss functions for different scenarios:

1. Mean Squared Error (MSE):
   * Used for regression problems.
   * Measures the average squared difference between the predicted and target values.
   * Formula: MSE = (1/n) \* Σ(y\_pred - y\_true)^2
2. Binary Cross Entropy:
   * Used for binary classification problems.
   * Measures the dissimilarity between predicted probabilities and true binary labels.
   * Formula: BCE = - (y\_true \* log(y\_pred) + (1 - y\_true) \* log(1 - y\_pred))
3. Categorical Cross Entropy:
   * Used for multi-class classification problems.
   * Calculates the average cross-entropy loss over all classes.
   * Formula: CCE = - Σ(y\_true \* log(y\_pred))
4. Hinge Loss (or SVM loss):
   * Used for support vector machine (SVM) classification problems.
   * Encourages correct classification margins between classes.
   * Formula: Hinge Loss = max(0, 1 - y\_true \* y\_pred)
5. Kullback-Leibler Divergence (KL Divergence):
   * Used in probabilistic models and for measuring the difference between probability distributions.
   * Formula: KL Divergence = Σ(y\_true \* log(y\_true / y\_pred))

**Question**: When do we use which loss function from the above lists?

**Use of activation function:**

The choice of activation function in a neural network depends on the specific problem, network architecture, and desired properties of the model. Here's a general guide on commonly used activation functions and their typical use cases:

1. Sigmoid:
   * Range: (0, 1)
   * Used in: Binary classification problems, where the output represents a probability.
   * Not recommended for: Hidden layers of deep neural networks due to vanishing gradient problem.
2. Tanh (Hyperbolic Tangent):
   * Range: (-1, 1)
   * Used in: Similar cases as sigmoid, but with the advantage of being centered at 0.
   * Not recommended for: Hidden layers of deep neural networks due to vanishing gradient problem.
3. ReLU (Rectified Linear Unit):
   * Range: [0, +∞)
   * Used in: Hidden layers of deep neural networks, where sparse activation is desirable.
   * Not recommended for: Output layer of regression problems, as it is unbounded.
4. Leaky ReLU:
   * Range: (-∞, +∞)
   * Used in: Hidden layers of deep neural networks, similar to ReLU, but addressing dead neurons.
   * Not recommended for: Output layer of regression problems, as it is unbounded.
5. Softmax:
   * Range: (0, 1) and sums to 1 across classes
   * Used in: Multi-class classification problems, where the output represents class probabilities.
   * Not recommended for: Hidden layers or binary classification problems.
6. Linear (Identity):
   * Range: (-∞, +∞)
   * Used in: Regression problems where the output doesn't need to be constrained.
   * Not recommended for: Problems where non-linearity is crucial, such as classification.

It's important to note that these guidelines are not strict rules, and experimentation is often needed to find the most suitable activation function for a particular task. Additionally, advances in neural network architectures, such as the use of batch normalization and skip connections, have mitigated some of the limitations associated with certain activation functions.

**Derivatives, gradient descent, backpropagation and learning rate (chapter-8,9)**

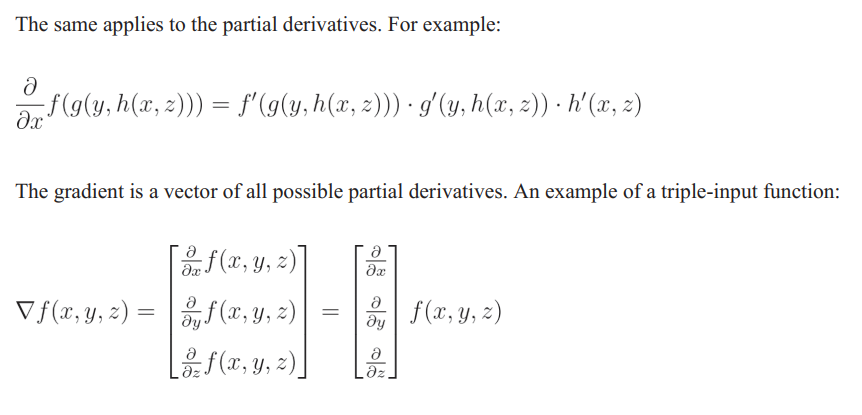
Partial derivative is required to find the weight and bias update simultaneously, i.e.,

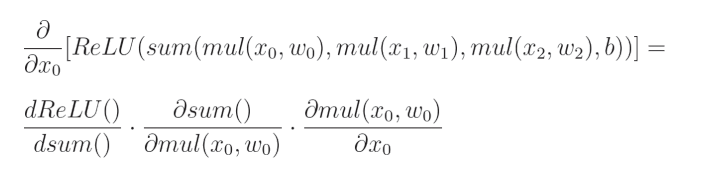
Y=w.x+b

dy/dw=d/dw(w.x)+b

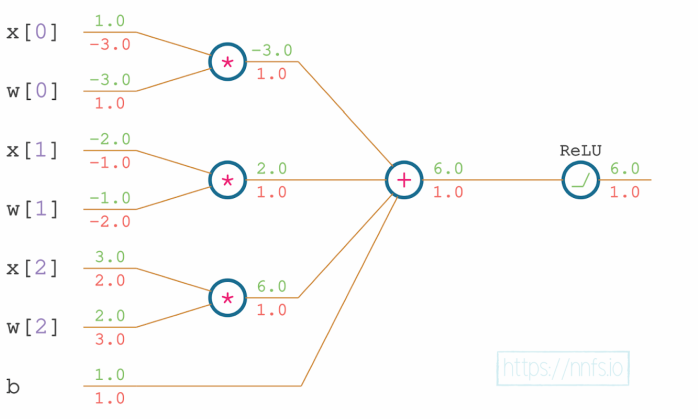
dy/db=w.x+d/db(b)

Now, if we set these derivatives in a vector then is said to be gradient. For example,

Chain rule



Example,



Derivative of sigmoidal:

The simplified equation **sigmoid'(x) = output \* (1 - output)** for the derivative of the sigmoid function can be derived using basic calculus. Here's how:

1. Start with the sigmoid function: sigmoid(x) = 1 / (1 + e^(-x))
2. Let's denote the output of the sigmoid function as "output": output = sigmoid(x) = 1 / (1 + e^(-x))
3. To find the derivative of the sigmoid function, we differentiate the output with respect to x: sigmoid'(x) = d(output) / dx
4. Using the quotient rule, we differentiate the output: sigmoid'(x) = (d(1) / dx) \* (1 + e^(-x)) - (d(1 + e^(-x)) / dx) \* (1) / (1 + e^(-x))^2
5. Since (d(1) / dx) is 0, we can simplify the expression: sigmoid'(x) = -(d(1 + e^(-x)) / dx) / (1 + e^(-x))^2
6. Differentiating (1 + e^(-x)) with respect to x, we get: sigmoid'(x) = -(-e^(-x)) / (1 + e^(-x))^2
7. Simplifying the expression further, we obtain: sigmoid'(x) = e^(-x) / (1 + e^(-x))^2

Now, to obtain the simplified form **sigmoid'(x) = output \* (1 - output)**, we substitute the output value back into the expression:

output = 1 / (1 + e^(-x))

sigmoid'(x) = (e^(-x)) / (1 + e^(-x))^2 = (e^(-x)) / (1 + e^(-x)) \* (1 + e^(-x)) / (1 + e^(-x)) = output \* (1 - output)

Therefore, the simplified equation **sigmoid'(x) = output \* (1 - output)** is derived by substituting the value of the output from the sigmoid function back into the derivative expression.

**How error\*derivative(sigmoidal) equation comes?**

The equation **error \* derivative(sigmoidal)** is derived using the chain rule in calculus. Let's go through the derivation step by step:

1. Start with the error term: **error = predicted\_output - true\_output**
2. The derivative of the sigmoid function with respect to its input (also known as the derivative of the sigmoidal activation function) is **derivative(sigmoidal) = sigmoidal \* (1 - sigmoidal)**
3. Now, we want to calculate the derivative of the error with respect to the input of the sigmoid function. We can do this by applying the chain rule:

**derivative(error) / derivative(input\_of\_sigmoidal) = derivative(error) / derivative(output\_of\_sigmoidal) \* derivative(output\_of\_sigmoidal) / derivative(input\_of\_sigmoidal)**

1. The first part **derivative(error) / derivative(output\_of\_sigmoidal)** is simply the derivative of the loss function with respect to the output of the sigmoidal activation. In the case of mean squared error (MSE) loss, it is calculated as **(predicted\_output - true\_output)**.
2. The second part **derivative(output\_of\_sigmoidal) / derivative(input\_of\_sigmoidal)** is the derivative of the sigmoidal activation function, which we derived earlier as **sigmoidal \* (1 - sigmoidal)**.
3. Combining these parts, we have:

**derivative(error) / derivative(input\_of\_sigmoidal) = (predicted\_output - true\_output) \* (sigmoidal \* (1 - sigmoidal))**

This is equivalent to **error \* derivative(sigmoidal)**.

Therefore, the equation **error \* derivative(sigmoidal)** represents the derivative of the error with respect to the input of the sigmoidal activation function, which is used in backpropagation to update the weights and biases in a neural network.

I hope this clarifies how the equation is derived. Let me know if you have any further questions!

Top of Form

**References**:

1. Neural network basics: <https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9>
2. Solving logic gates using neural network: <https://medium.com/@stanleydukor/neural-representation-of-and-or-not-xor-and-xnor-logic-gates-perceptron-algorithm-b0275375fea1>
3. How Neural Networks Solve the XOR Problem: <https://towardsdatascience.com/how-neural-networks-solve-the-xor-problem-59763136bdd7>
4. Neural network with python: Harrison Kinsley, Daniel Kukieła - Neural Networks from Scratch in Python (2020)
5. <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>