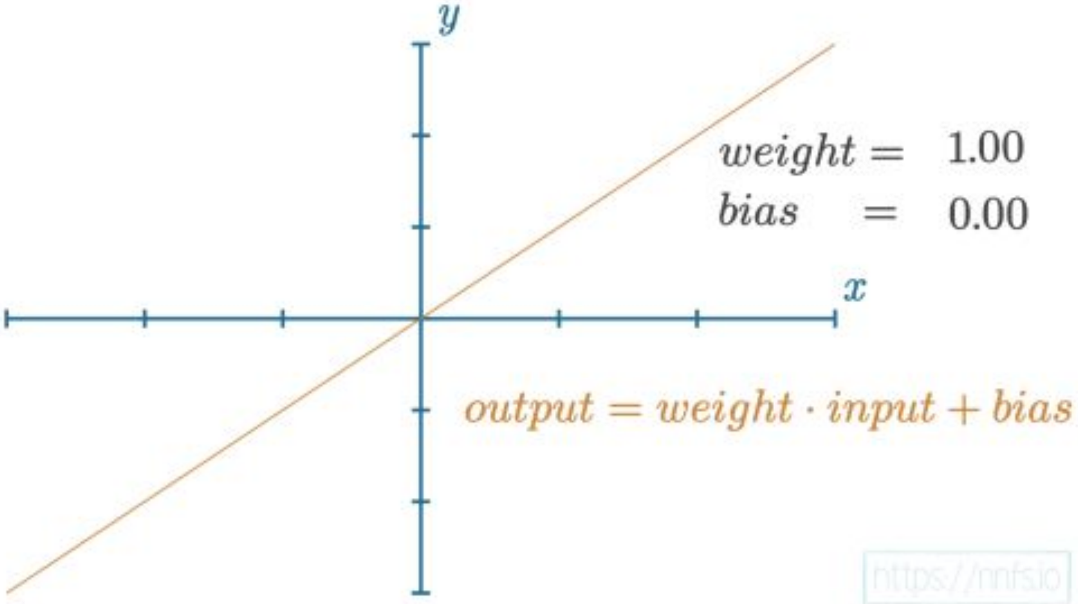
Building a Neural Net from Scratch

Tina Tran

Weights & Biases

- Input multiplied by weight
- Bias added to input



Neurons / Layer(s) of Neurons

- Dot Product (np,dot)
  - Vector Addition
- Matrix Product (np.dot(np.array))

layer1_outputs = np.dot(inputs, np.array(weights).T) + biases

layer2_outputs = np.dot(layer1_outputs, np.array(weights2).T) + biases2

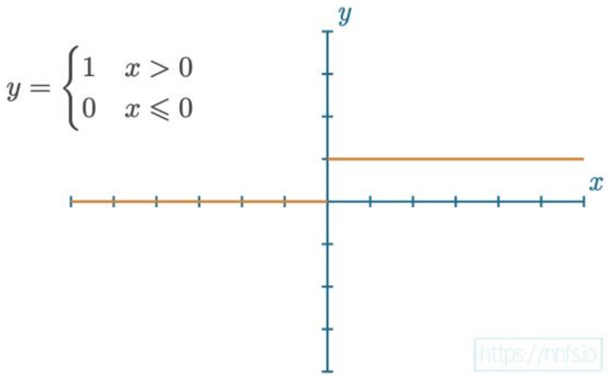
Dense Layer Class

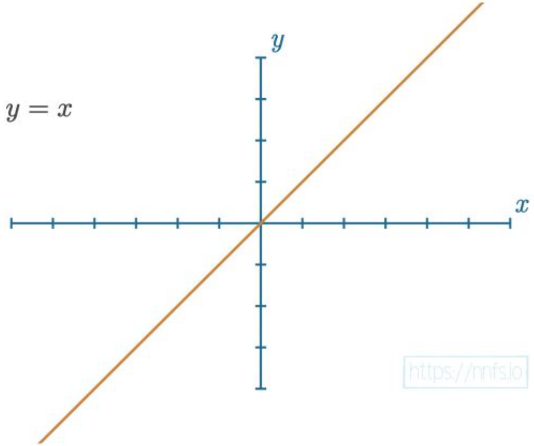
```python
# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases
```

Activation Functions

- Step Activation Function

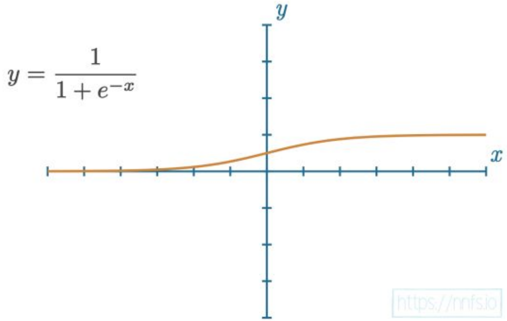
- Linear Activation Function





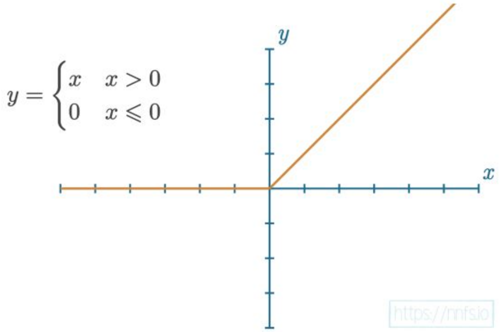
Activation Functions 2

- Sigmoid Activation Function



- Rectified Linear Activation Function (ReLU)

  np.maximum(0, inputs)



Activation Functions 3

- Softmax Activation Function

```
exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True)
probabilities = exp_values / np.sum(exp_values, axis=1, keepdims=True)
```

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}$$

Loss / Cost Function

Loss = -np.log(activation_function_output)

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j})$$

$$L_i = -\log\left(\hat{y}_{i,k}\right) \quad where\ \mathbf{k}\ is\ an\ index\ of\ "true"\ probability$$

Backpropagation

- Calculates partial derivatives to see role in minimizing loss
- Add “backwards pass” to neuron layer and activation functions

Accuracy and Optimization

Optimizers

- Stochastic Gradient Descent (SGD)
  - Learning rate decay
  - Momentum (update_params)
  - AdaGrad (adaptive gradient)
  - RMSProp
- Adam (Adaptive Momentum)

Regularization

- L1 & L2
- Dropout layer

Regression

Determine specific value (predictions) based on input

- Linear Activation
- Mean Squared Error Loss
- Mean Absolute Error Loss