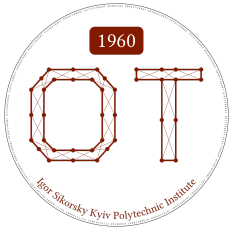


Computer Vision

Lecture 2: Building Neural Networks with Perceptrons

Yuriy Kochura
iuriy.kochura@gmail.com
@y_kochura

Today

- Single-layer Neural Network: Forward propagation
- One-Dimensional Gradient Descent
- Single-layer Neural Network: Backward propagation
- Multi Output Perceptron
- Multilayer Perceptron (feedforward neural network)
- An Example of Applying Neural Networks

The Perceptron: Simplified

Single-layer Neural Network

Logistic Regression

Perceptron

The perceptron (Rosenblatt, 1958)

$$g(z) = \begin{cases} 1 & \text{if } z = \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

This model was originally motivated by biology, with w_i being synaptic weights for inputs x_i and g firing rates.

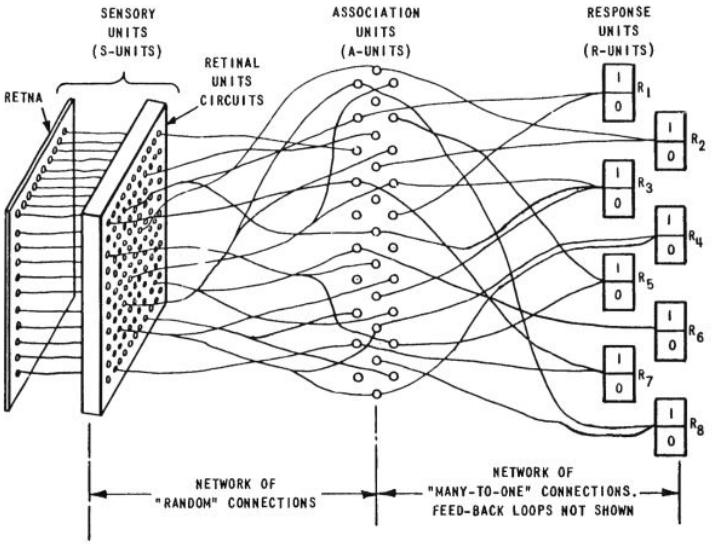
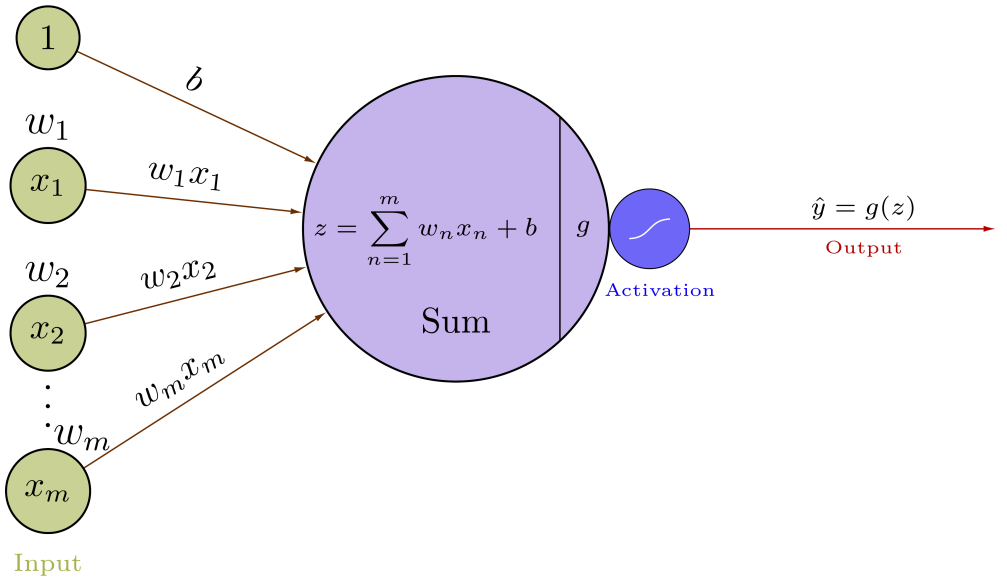


Figure 1 ORGANIZATION OF THE MARK I PERCEPTRON

—

Image source: Frank Rosenblatt, Mark I Perceptron operators' manual, 1960.

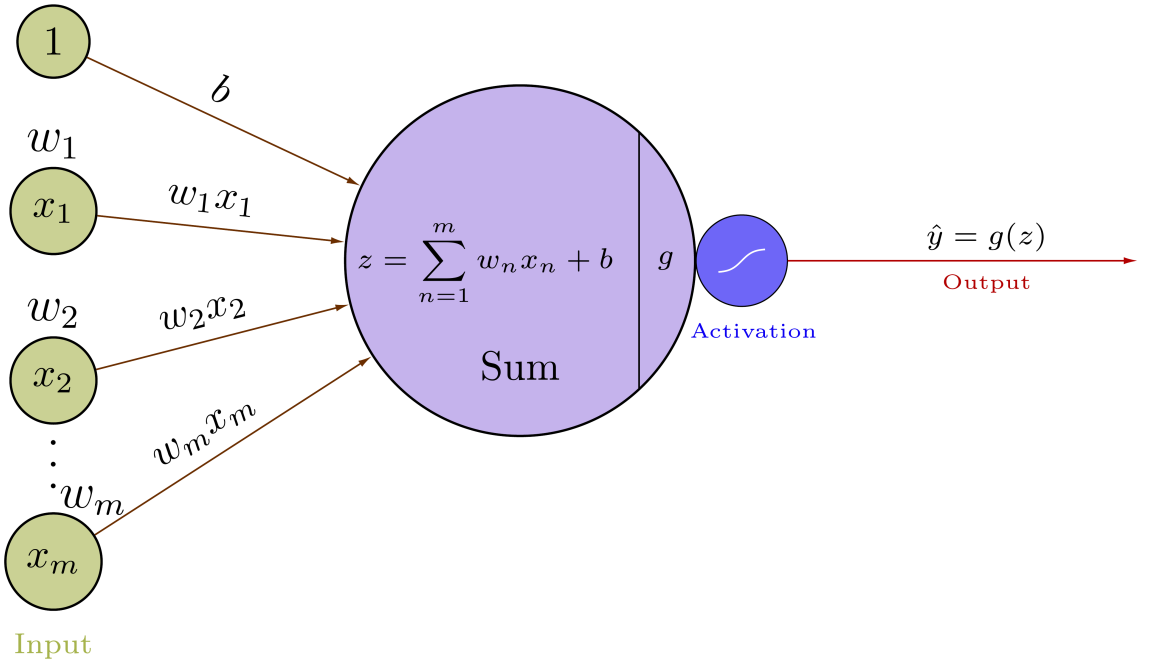


$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \qquad \mathbf{X}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}$$

$$z = \sum_{n=1}^{m} w_n x_n + b = \mathbf{X}^T \cdot \mathbf{W} + b = \mathbf{W}^T \cdot \mathbf{X} + b$$

$$\hat{y} = g(z)$$

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$



Forward propagation

$$z = \sum_{n=1}^{m} w_n x_n + b = \mathbf{X}^T \cdot \mathbf{W} + b = \mathbf{W}^T \cdot \mathbf{X} + b$$

$$\hat{y} = g(z)$$

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Example

Suppose $m = 3$

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -0.1 \\ 0.7 \\ 0.5 \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 2 \end{bmatrix} \qquad b = 0.8$$

$$z = \sum_{n=1}^{3} w_n x_n + b = w_1 x_1 + w_2 x_2 + w_3 x_3 + b =$$
$$= 1 \cdot -0.1 + -2 \cdot 0.7 + 2 \cdot 0.5 + 0.8 = 0.3$$

$$z = \mathbf{X}^T \cdot \mathbf{W} + b = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} + b =$$
$$= w_1 x_1 + w_2 x_2 + w_3 x_3 + b = 0.3$$

$$\hat{y} = g(z) = g(\mathbf{X}^T \cdot \mathbf{W} + b) = \frac{1}{1 + \exp(-z)} = \frac{1}{1 + \exp(-0.3)} \approx 0.57$$

One-Dimensional Gradient Descent

One-Dimensional Gradient Descent

Consider some continuously differentiable real-valued function $f : \mathbb{R} \to \mathbb{R}$. Using a Taylor expansion we obtain:

$$f(x + \varepsilon) = f(x) + \varepsilon f^{'}(x) + \mathcal{O}(\varepsilon^2)$$

To keep things simple let's pick a fixed step size $\alpha > 0$ and choose $\varepsilon = -\alpha f^{'}(x)$. Plugging this into the Taylor expansion above we get:

$$f(x - \alpha f^{'}(x)) = f(x) - \alpha f^{'2}(x) + \mathcal{O}(\alpha^2 f^{'2}(x))$$

If the derivative $f^{'}(x) \neq 0$ does not vanish we make progress since $\alpha f^{'2}(x) > 0$. Moreover, we can always choose α small enough for the higher-order terms to become irrelevant. Hence we arrive at

$$f(x - \alpha f^{'}(x)) \lessapprox f(x)$$

This means that, if we use

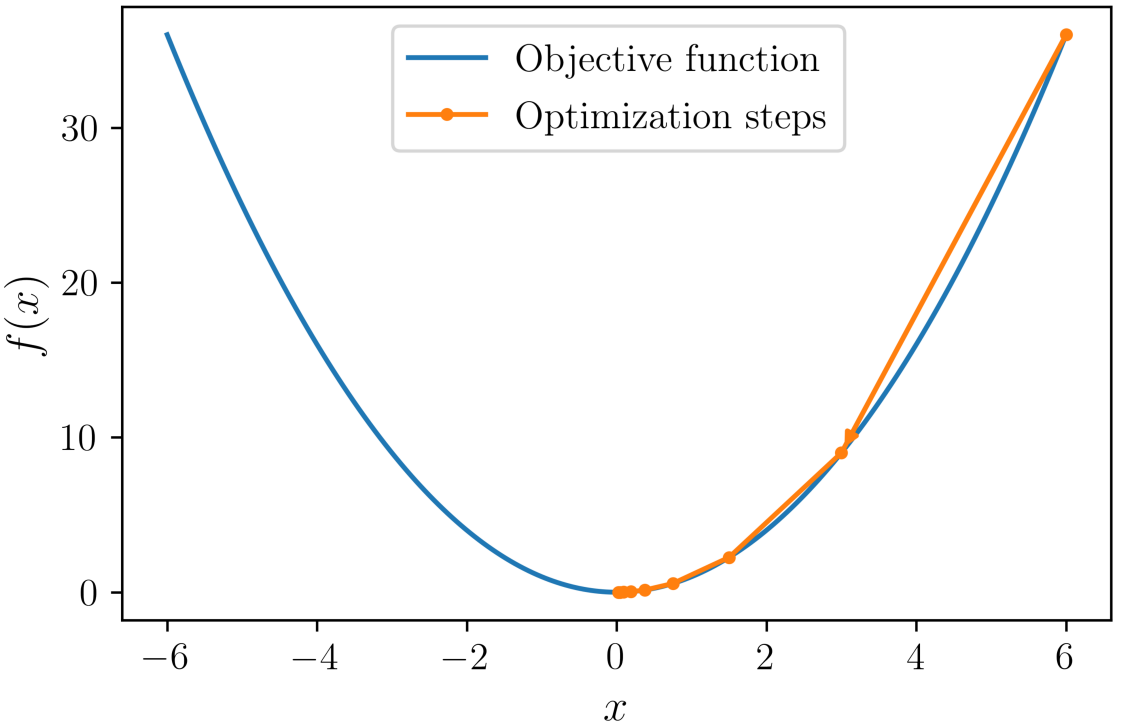
$$x \leftarrow x - \alpha f^{'}(x)$$

to iterate x, the value of function $f(x)$ might decline.

```python
import numpy as np

def f(x):  # Objective function
    return x**2

def f_grad(x):  # Gradient (derivative) of the objective function
    return 2 * x

def bgd(alpha, f_grad):
    x = 6.0         # Initial value of x
    results = [x]
    epoch = 8       # Number of iterations
    for i in range(epoch):
        x -= alpha * f_grad(x)
        results.append(float("%.6f" % x))
    print(f'epoch {epoch}, x: {x:.6f}')
    return results

results = bgd(0.25, f_grad)
print(results)
```

```
epoch 8, x: 0.023438
[6.0, 3.0, 1.5, 0.75, 0.375, 0.1875, 0.09375, 0.046875, 0.023438]
```

The progress of optimizing over x

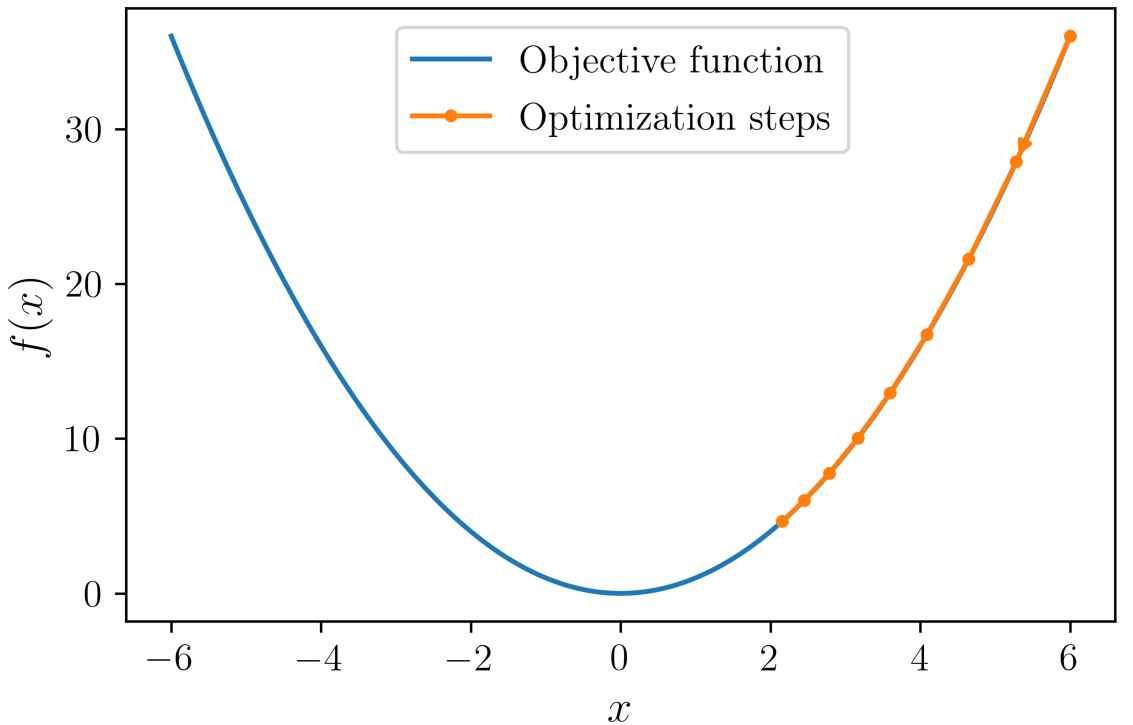


One-Dimensional Gradient Descent ($\alpha = 0.25$)

The progress of optimizing over x

```
results = bgd(0.06, f_grad)
```

```
epoch 8, x: 2.157807
```

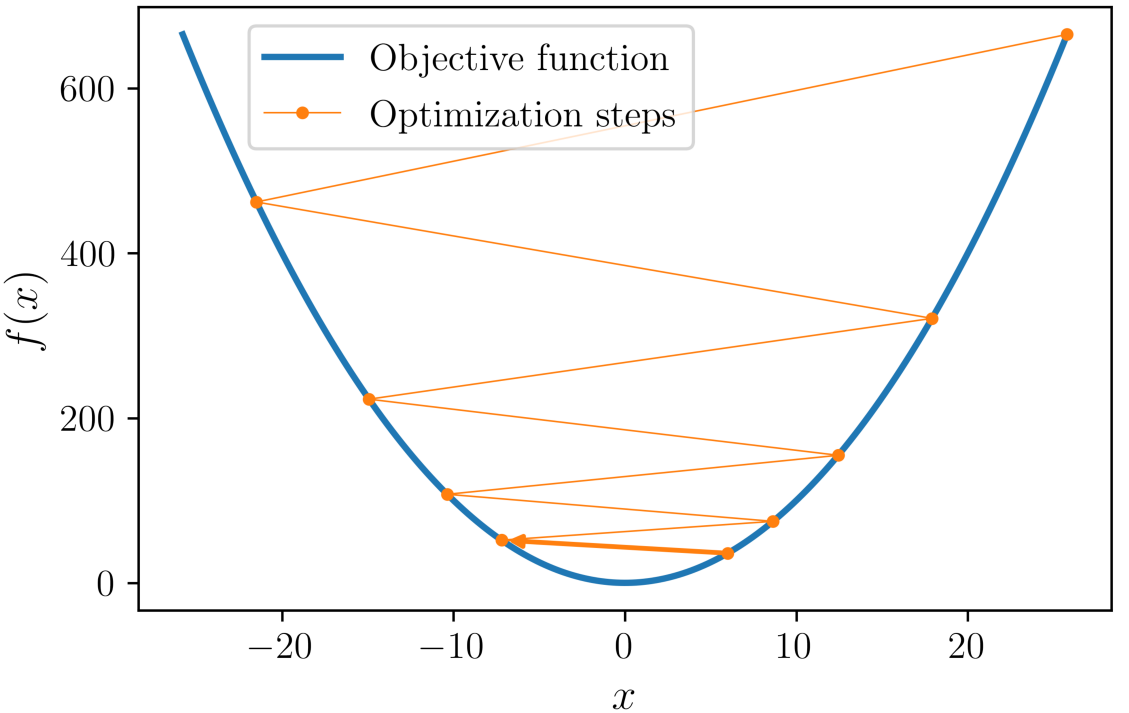


One-Dimensional Gradient Descent ($\alpha = 0.06$)

The progress of optimizing over x

```
results = bgd(1.1, f_grad)
```

```
epoch 8, x: 25.798902
```



One-Dimensional Gradient Descent ($\alpha = 1.1$)

The Perceptron: Backward propagation

In Leibniz notations, the chain rule states that

$$\frac{\partial \ell}{\partial \theta_i} = \sum_{k \in \text{parents}(\ell)} \frac{\partial \ell}{\partial u_k} \underbrace{\frac{\partial u_k}{\partial \theta_i}}_{\text{recursive case}}$$

Backpropagation

- Since a neural network is a composition of differentiable functions, the total derivatives of the loss can be evaluated backward, by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called reverse automatic differentiation or backpropagation.

Forward propagation

$$z = \sum_{n=1}^{m} w_n x_n + b = \mathbf{X}^T \cdot \mathbf{W} + b = \mathbf{W}^T \cdot \mathbf{X} + b$$

$$\hat{y} = g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Backward propagation

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial z} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} = \hat{y} - y$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial \mathbf{W}} = \mathbf{X}^T \cdot (\hat{y} - y)$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial b} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial b} = \hat{y} - y$$

Update the weights and bias

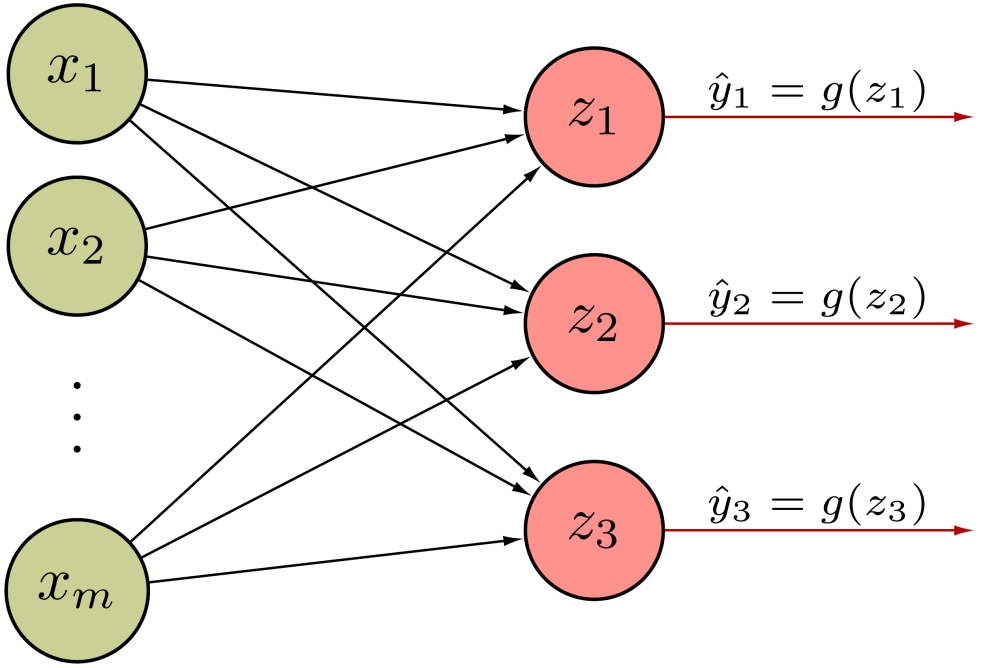
$$\mathbf{W} = \mathbf{W} - \alpha \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \mathbf{W}}$$

$$b = b - \alpha \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial b}$$

Multi Output Perceptron

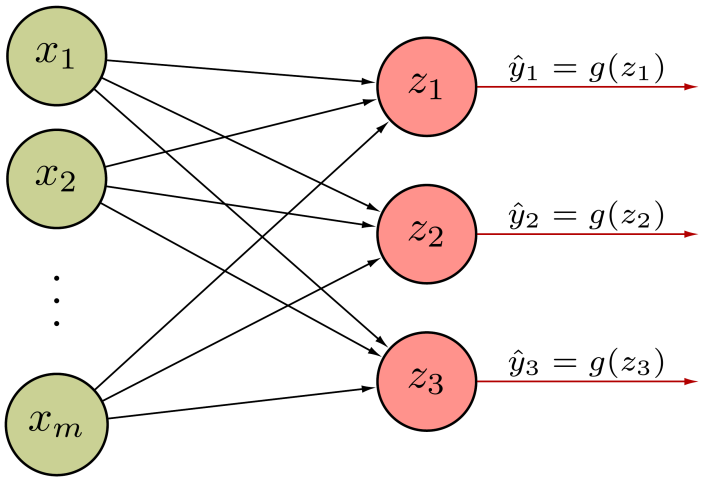
Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called Dense layers



$$z_j = \sum_{n=1}^{m} w_{j,n} x_n + b_j$$

Example



$$\mathbf{X}^{m\times 1} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad \mathbf{W}^{3\times m} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ w_{31} & w_{32} & \cdots & w_{3m} \end{bmatrix} \qquad \mathbf{b}^{3\times 1} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\mathbf{z} = \mathbf{W}\cdot\mathbf{X} + \mathbf{b} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ w_{31} & w_{32} & \cdots & w_{3m} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} =$$

$$= \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m + b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m + b_2 \\ w_{31}x_1 + w_{32}x_2 + \cdots + w_{3m}x_m + b_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

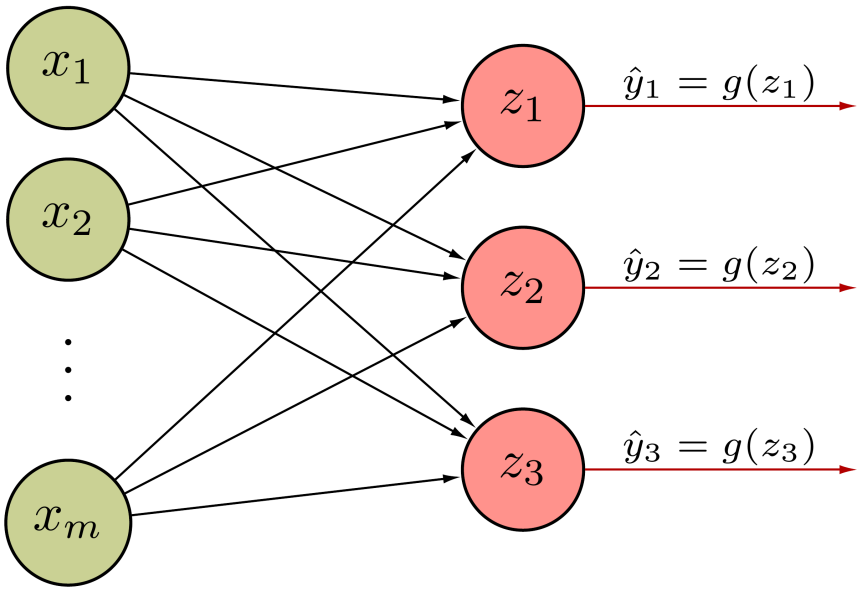
Dense layer from scratch


```
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, input_dim, output_dim):
    super(MyDenseLayer, self).__init__()

    # Initialize weights and bias
    self.W = self.add_weight([input_dim, output_dim])
    self.b = self.add_weight([1, output_dim])

  def call(self, inputs):
    # Forward propagate the inputs
    z = tf.matmul(inputs, self.W) + self.b

    # Feed through a non-linear activation
    output = tf.math.sigmoid(z)

    return output
```

—

Slide source: MIT 6.S191

Because all inputs are densely connected to all outputs, these layers are called Dense layers

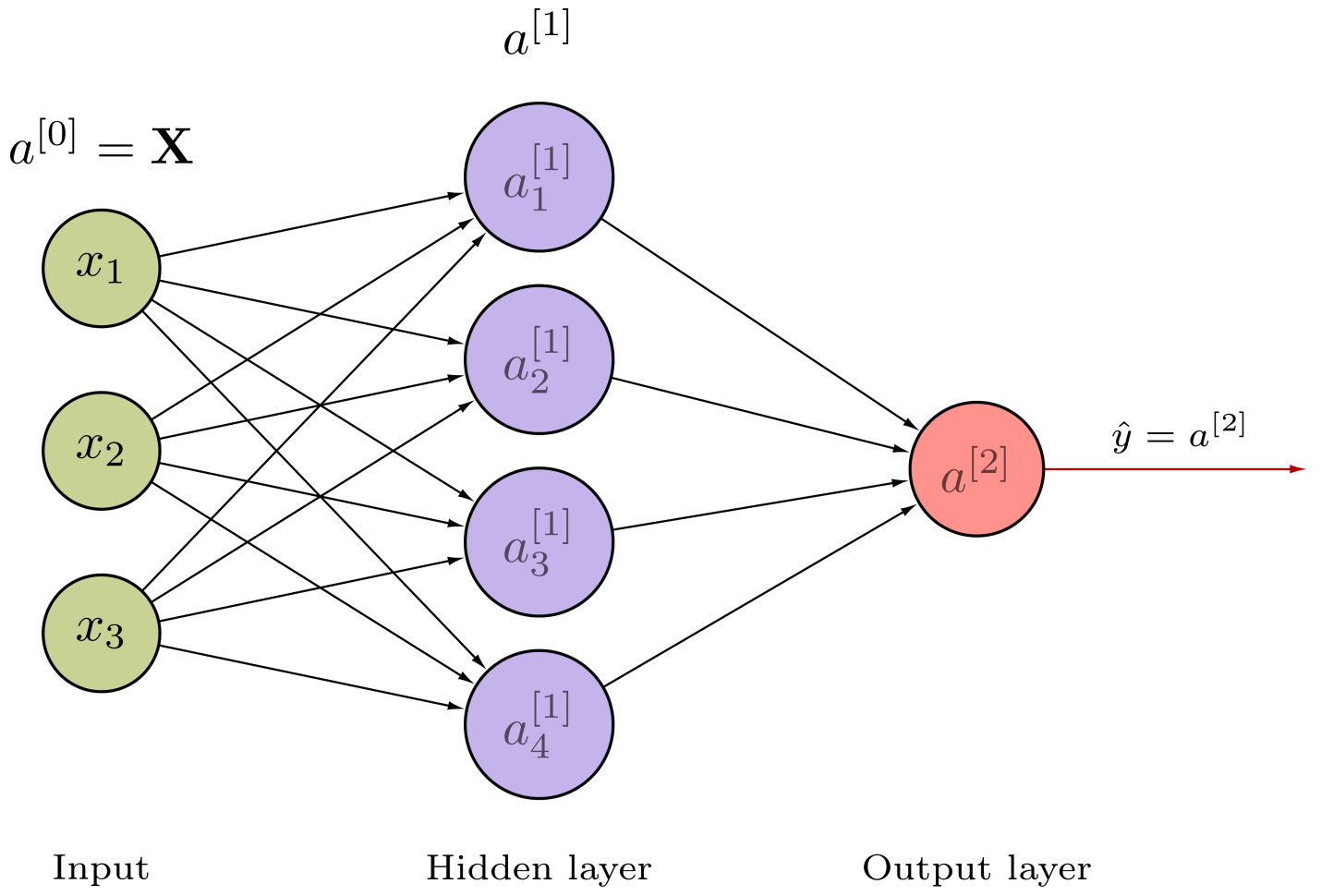



```python
import tensorflow as tf

layer = tf.keras.layers.Dense(
    units=3)
```

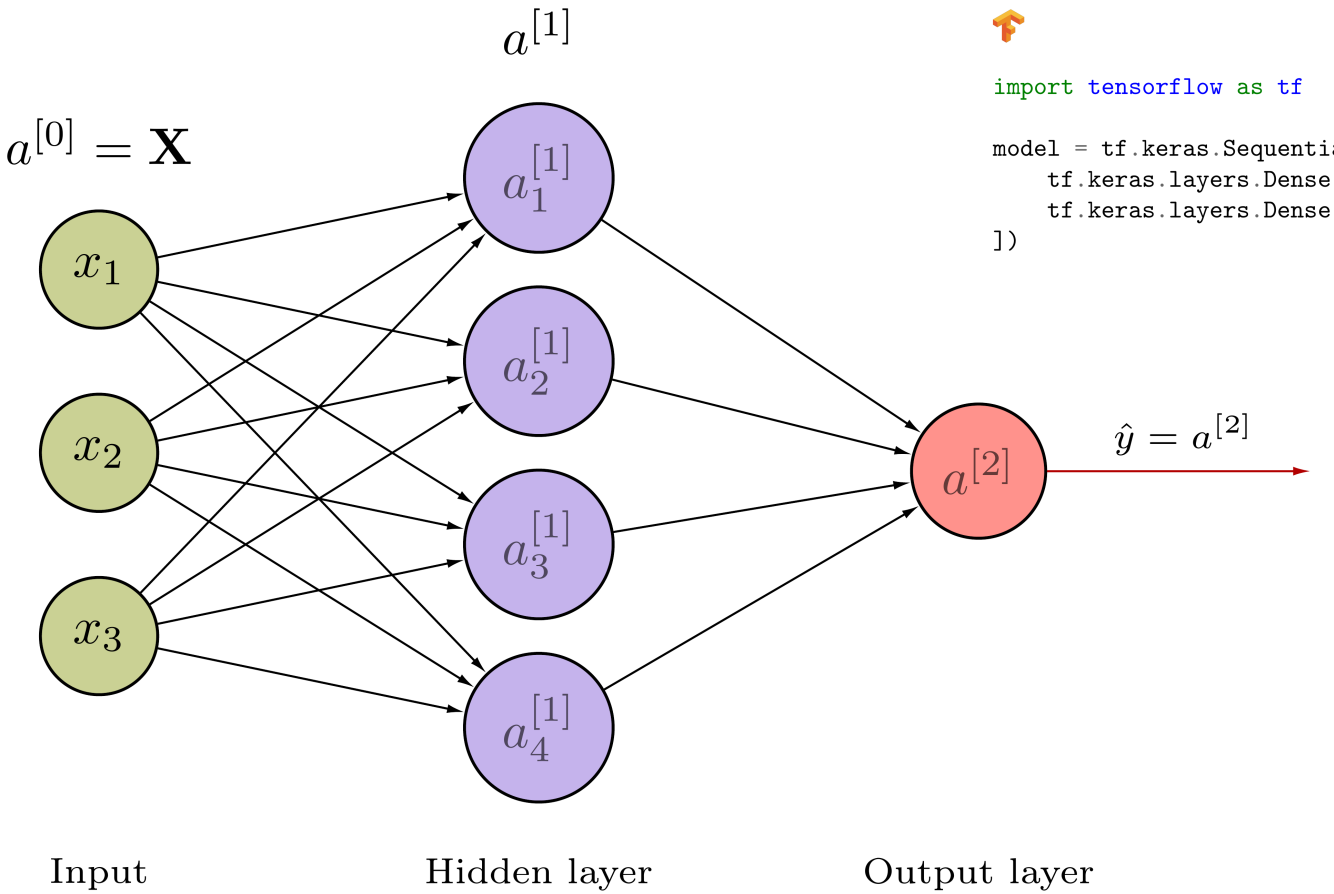
$$z_j = \sum_{n=1}^{m} w_{j,n} x_n + b_j$$

Multilayer Perceptron

One hidden layer Neural Network

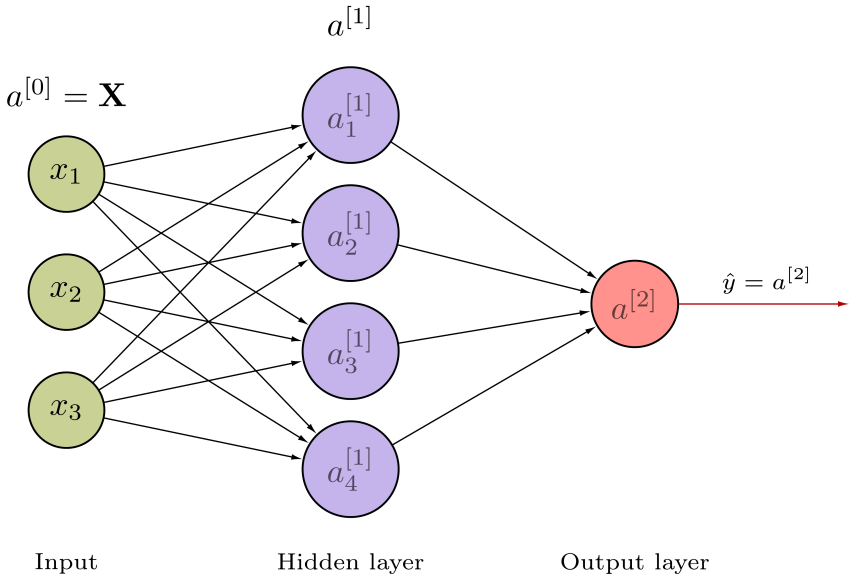


One hidden layer Neural Network




```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(4),
    tf.keras.layers.Dense(1)
])
```

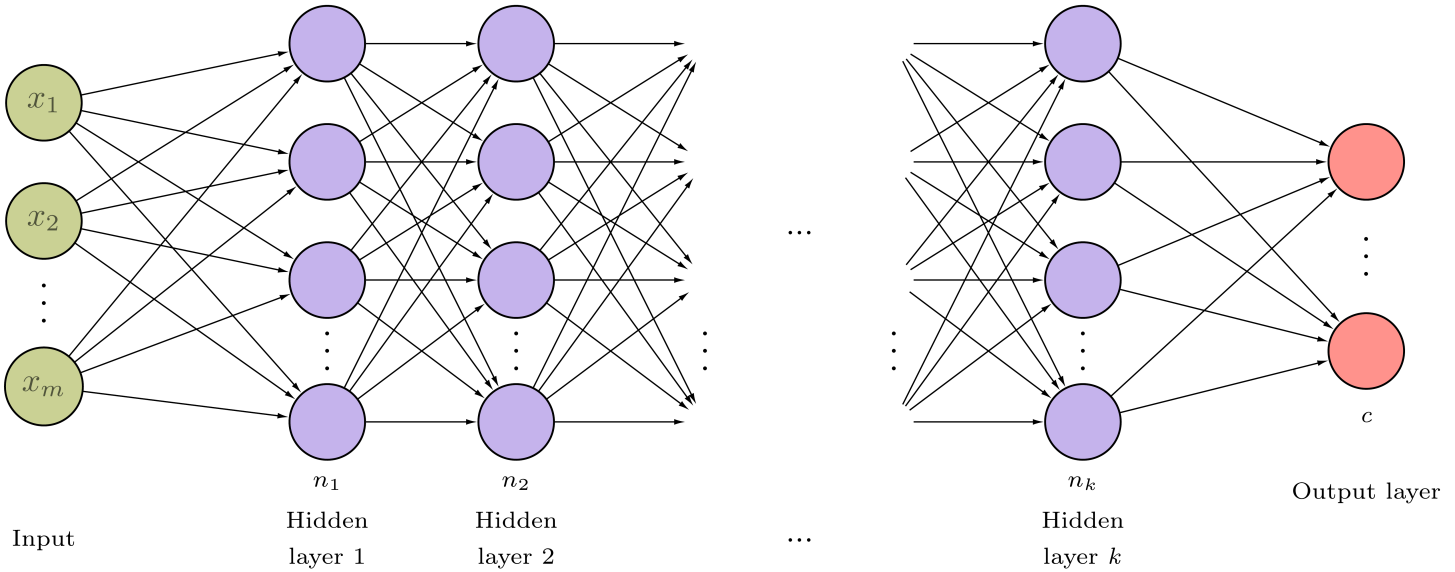
One hidden layer Neural Network



$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \mathbf{W}^{[1]} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \qquad \mathbf{b}^{[1]} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \qquad \mathbf{W}^{[2]} = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix} \qquad b^{[2]} = b$$

$$\begin{aligned} \mathbf{z}^{[1]} &= \mathbf{W}^{[1]} \cdot \mathbf{X} + \mathbf{b}^{[1]} \\ \mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\ z^{[2]} &= \mathbf{W}^{[2]} \cdot \mathbf{a}^{[1]} + b^{[2]} \\ \hat{y} &= a^{[2]} = g^{[2]}(z^{[2]}) \end{aligned}$$

Deep Neural Network




```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n1),
    tf.keras.layers.Dense(n2),
    .
    .
    .

    tf.keras.layers.Dense(nk),
    tf.keras.layers.Dense(c)
])
```

Applying Neural Networks

Example Problem

Will I pass this class?

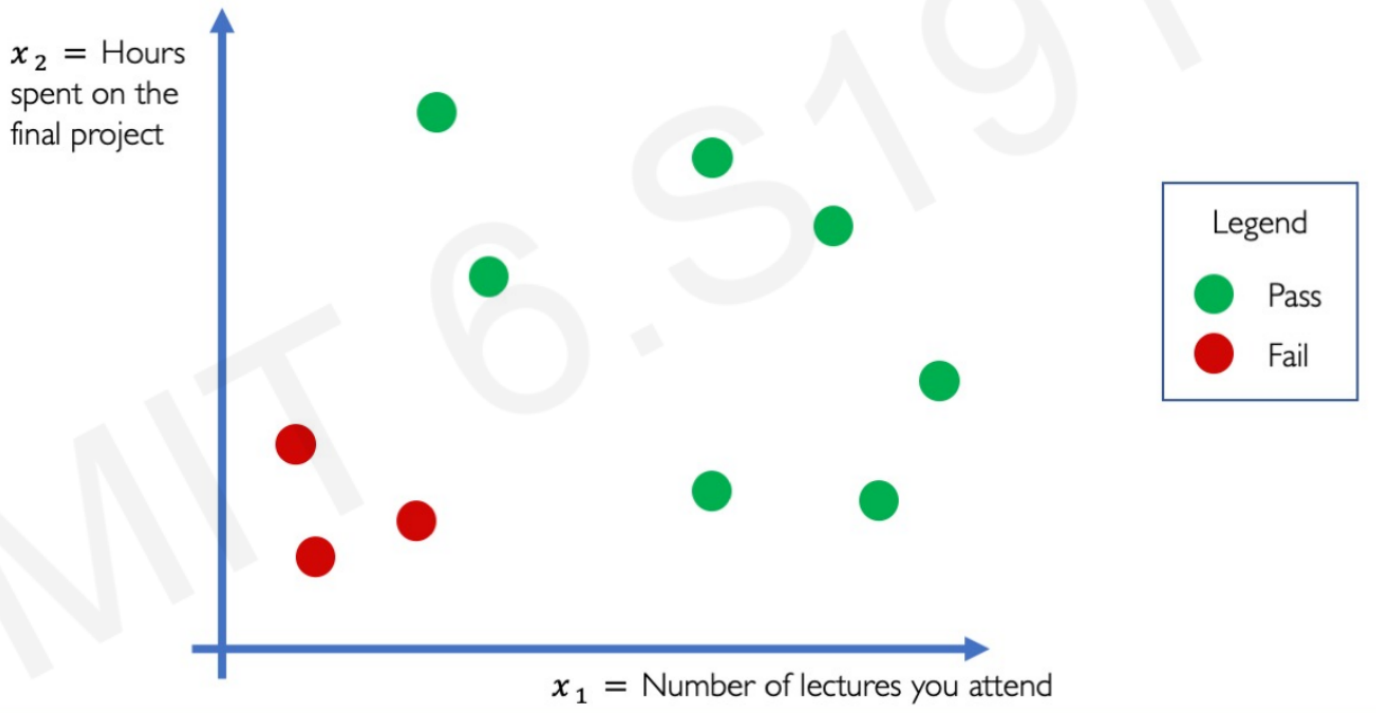
Let's start with a simple two feature model

x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

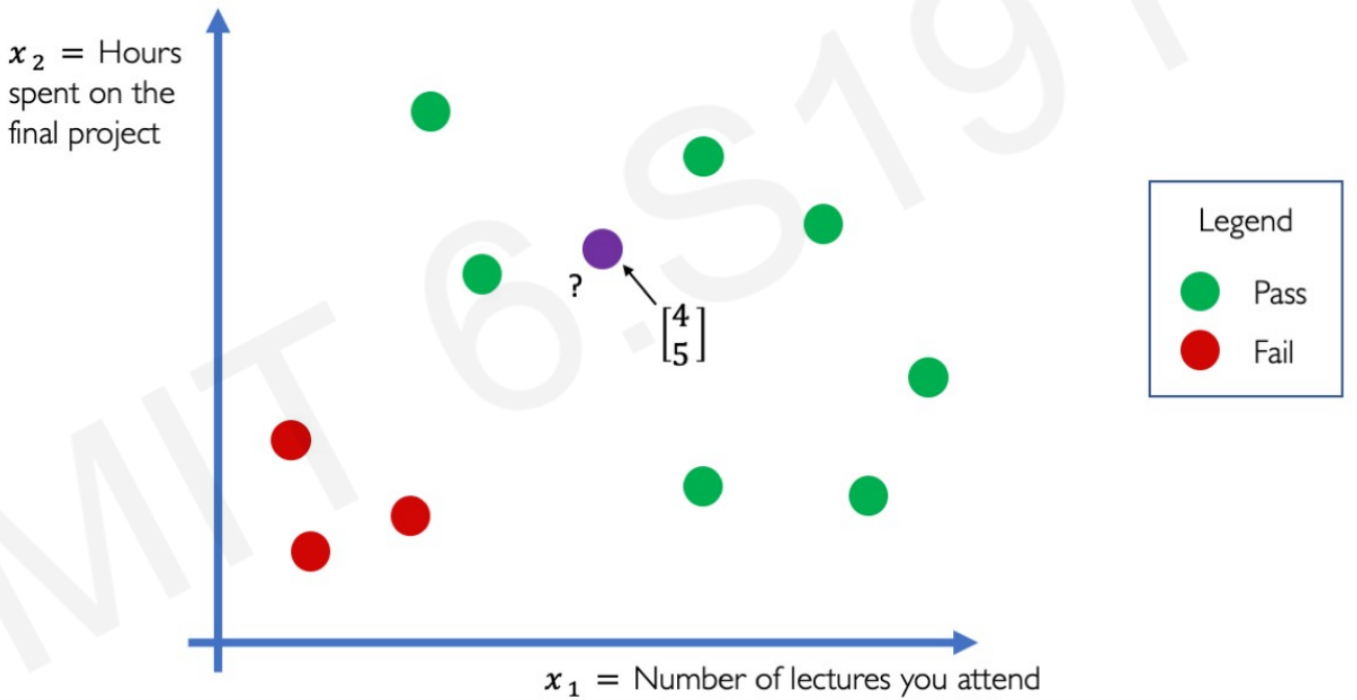
Slide source: MIT 6.S191

Example Problem: Will I pass this class?



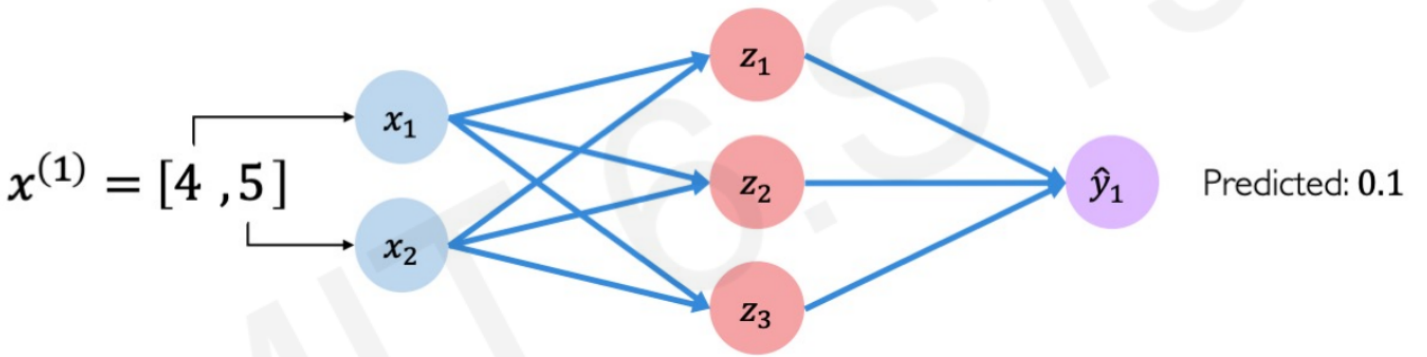
Slide source: MIT 6.S191

Example Problem: Will I pass this class?



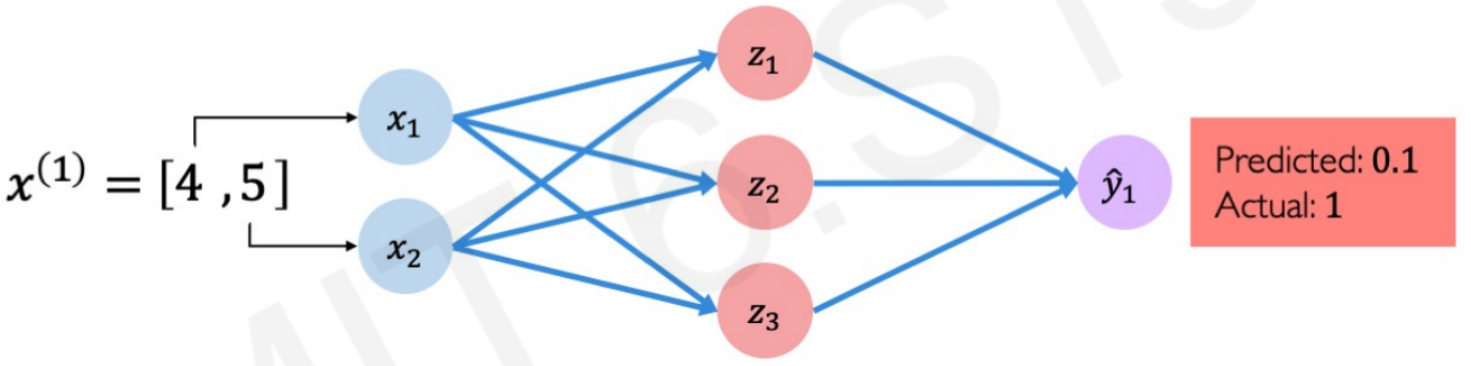
Slide source: MIT 6.S191

Example Problem: Will I pass this class?



Slide source: MIT 6.S191

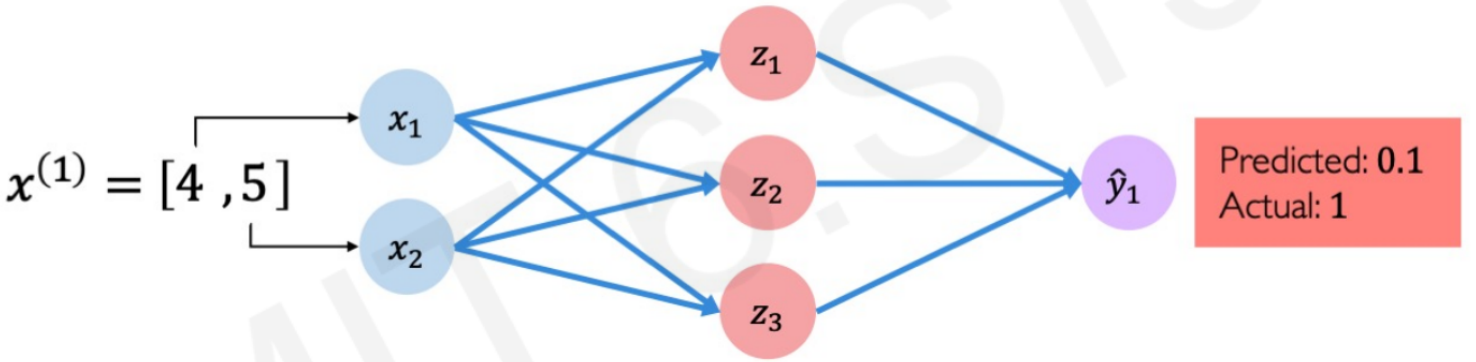
Example Problem: Will I pass this class?



Slide source: MIT 6.S191

Quantifying Loss

*The **loss** of our network measures the cost incurred from incorrect predictions*



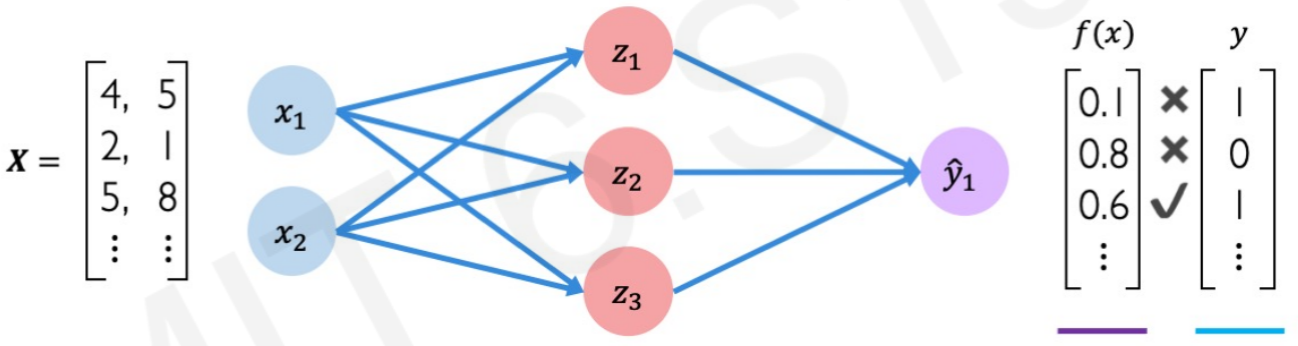
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Slide source: MIT 6.S191

Empirical Loss

*The **empirical loss** measures the total loss over our entire dataset*



$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

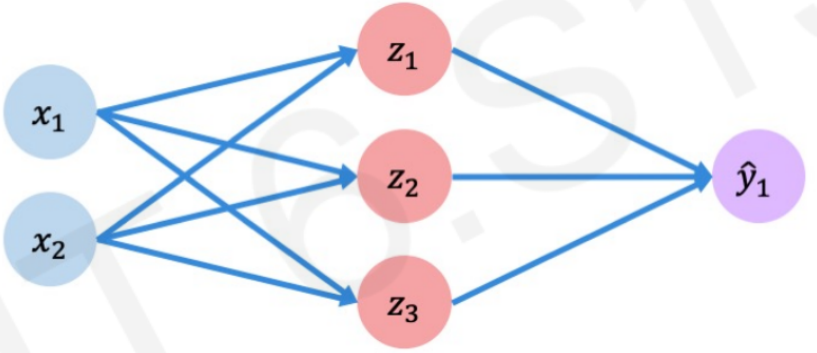
Predicted Actual

Also known as:
- Objective function
- Cost function
- Empirical Risk

Binary Cross Entropy Loss

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

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

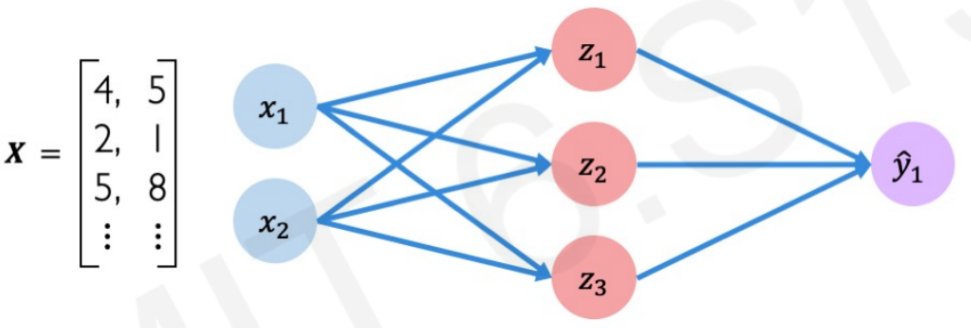
$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log \underbrace{\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right)}_{\text{Predicted}} + \underbrace{(1 - y^{(i)})}_{\text{Actual}} \log \underbrace{\left(1 - f\left(x^{(i)}; \boldsymbol{W}\right)\right)}_{\text{Predicted}}$$

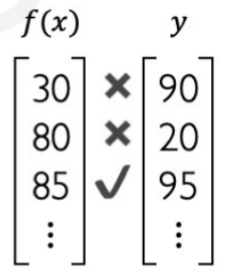
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```


Slide source: MIT 6.S191

Mean Squared Error Loss

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





$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; W\right)\right)^2$$

Actual Predicted

```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
loss = tf.keras.losses.MSE( y, predicted )
```

The end