Machine Learning and Email Classification

Mahmut Ozge Karakaya

ING

Agenda

- Linear Regression
- Neural Networks
- Convolutional Neural Networks and Reinforcement Learning (Deep Learning)
- Email Classification

Linear Regression

	Living Area (x1)	Build Year (x2)	Fireplaces (x3)	Price (y)
House 1	158	1961	0	181500
House 2	117	1958	1	223500
House 3	165	1998	1	?

- θ1 * 158 + θ2 * 1961 + θ3 * 0 + b = 181500
- θ1 * 117 + θ2 * 1958 + θ3 * 1 + b = 223500
- θ1 * 165 + θ2 * 1998 + θ3 * 1 + b = ?

- θ = weights (w)
- b = bias
- We want to learn θ and b

Matrix Operations

- Transpose matrix
  - It switches the row and column indices of the matrix A by producing another matrix denoted by A^T

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

- Dot product
  - Dot product of N X M matrix and M X K matrix is another matrix with size N X K where N, M, and K are the dimensions.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{bmatrix}$$

Linear Regression (SGD)

```python
import numpy as np

X =np.array([[2, 3, 3], [4, 3, 3], [4, 3, 7], [4, 3, 6]])
y = np.array([2, 7, 1, 3])


def linear_regression(X, y, epochs=100000, learning_rate=0.0001):
    m = float(len(y))
    theta = np.random.random(X.shape[1])
    b = np.random.random()
    for i in range(epochs):
        y_predicted = np.dot(X, theta.T) + b
        loss = y_predicted - y
        for j in range(X.shape[1]):
            theta_gradient = (1 / m) * sum(X[:, j] * loss)
            theta[j] = theta[j] - (learning_rate * theta_gradient)
        b_gradient = (1 / m) * sum(loss)
        b = b - (learning_rate * b_gradient)

    return theta, b


theta, b = linear_regression(X, y)

print("theta=", theta)
print("b=", b)
print(np.dot(theta.T, X[0]) + b)
print(np.dot(theta.T, X[1]) + b)
print(np.dot(theta.T, X[2]) + b)
print(np.dot(theta.T, X[3]) + b)
print('test item', np.dot(theta.T, [1, 3, 2]) + b)
```

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j := 0...n}$$

}

$h_\theta = \text{dot}(x, \theta^T) + b$

```
('theta=', array([ 2.48033737,  0.28643932, -1.45215078]))
('b=', 0.612233930953655 86)
2.07577430071
7.03644903734
1.22784591866
2.67999669833
('test item', 1.0475877120677661)
```

- Can ML learn random data?
- What if y = [Apartment, Not Apartment] ?
  → Apply sigmoid to h_θ and change loss function.
- What if y = [Apartment, Farm House, Castle] ?
  → Apply one-vs-all classification
- Does this algorithm work for house price prediction?

ING

ML Preprocessing

- Min-max scaling
- One-hot Encoding
  - Clay = {0, 0, 1}
  - Metal = {0, 1, 0}
  - Wood = {1, 0, 0}

- In feature engineering, we create new features. New features may improve the results.
  - log(living area) → difference between 75-80 may be more important than the difference between 140-145
  - T-shirt size living area
  - Square(any x)
  - Distance to city center (if you have coordinates)

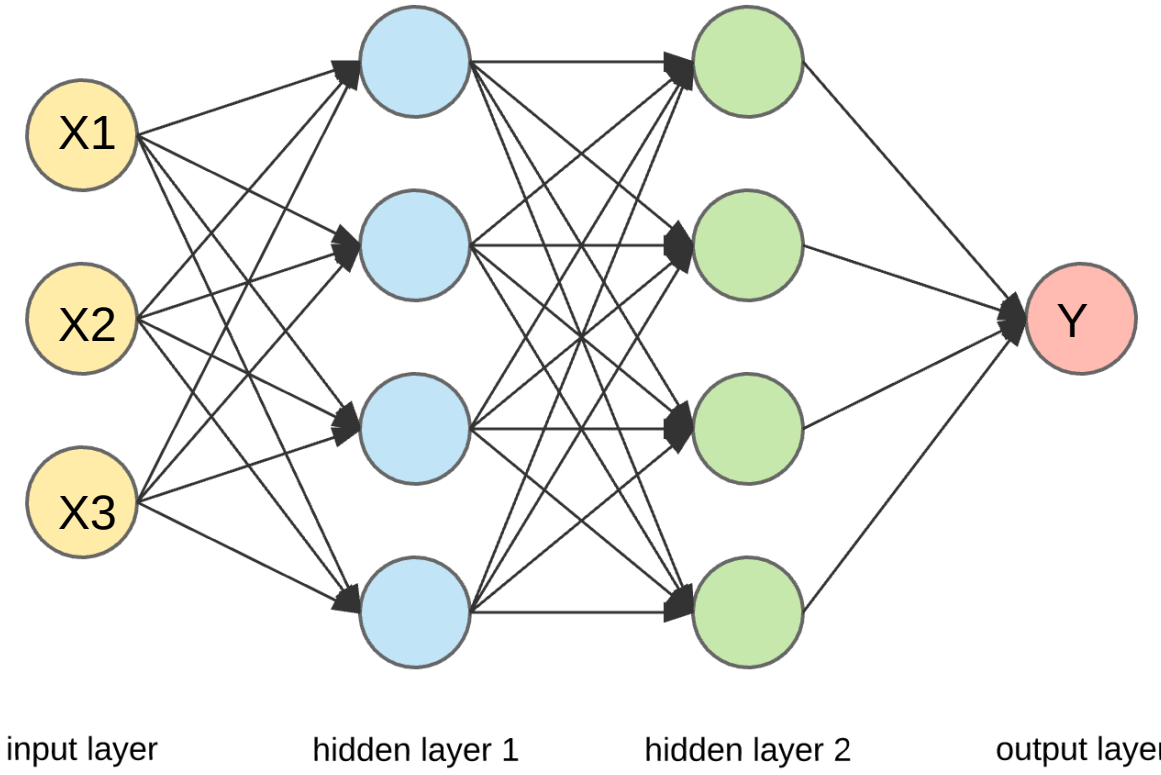
$$X_{sc} = \frac{X - X_{min}}{X_{max} - X_{min}}.$$

Roof Material (x4)
Clay
Metal
Wood

	is_wood (x5)	is_metal (x6)	is_clay (x7)
House 1	0	0	1
House 2	0	1	0
House 3	1	0	0

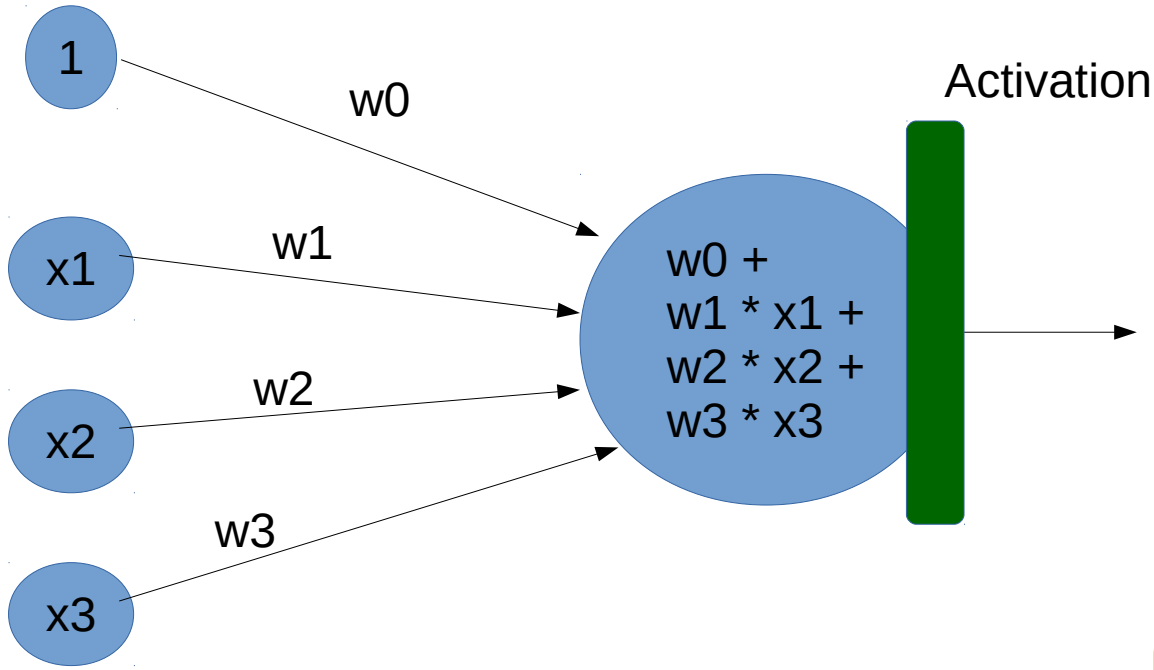
Neural Networks

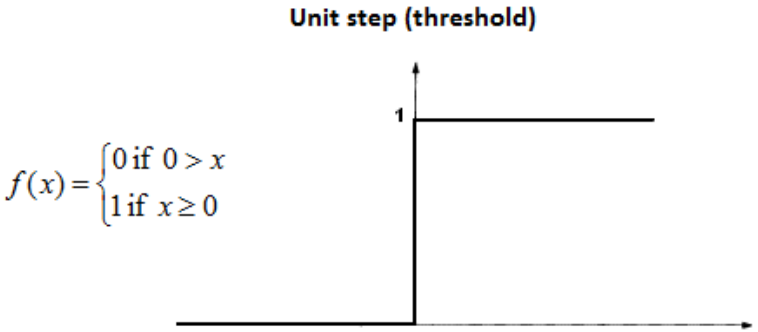
- NN has 3 type of layers.
- They are building blocks of Deep Learning algorithms, and they can be used by their own.
- Nodes are connected to each other with some weights, and we want to learn the all weight values.
- We first predict y values (feed forward), and then we update the weights using the error (backpropagation)
- If all nodes in a layer are connected to all nodes in previous layer, the layer is called fully connected (dense layer)
- The output is a single node if the problem is a regression problem.
- Number of nodes in output layer equal to number of classes, if the problem is a classification problem. (one-vs-all is also possible but costly)



Perceptron

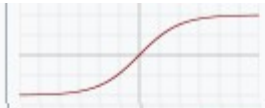
- Perceptons are building blocks of neural networks
- Input values or One input layer
- Weights and Bias
- Net sum (dot product)
- Activation Function
- Output



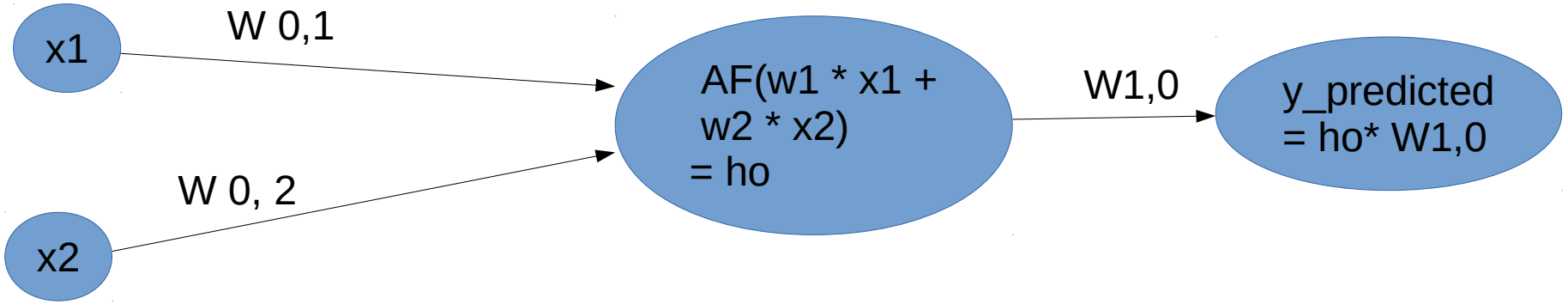


More Activation Functions

- Activation functions are always nonlinear. NNs are nonlinear because of the activation functions.
- Linear AF doesn’t work with NN.
- Sigmoid functions and their combinations usually work better for classification techniques.
- ReLU is a widely used activation function and yields beter results compared to Sigmoid and Tanh.
- Leaky ReLU is a solution for a dead neuron problem.

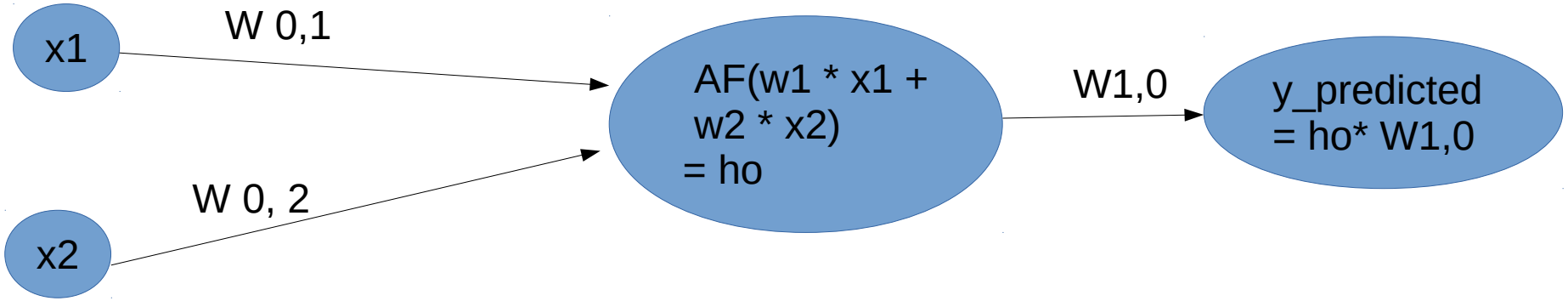
TanH		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$f'(x) = 1 - f(x)^2$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
ReLU		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

Neural Networks – Feed forward



- ho = hidden output
- AF = activation function
- In feed forward process we calculate y values.
- If we had more hidden nodes, we'd get the dot product to calculate y

Neural Networks – Back propagation



- In back propagation, we update the weights (W)
- error = y - y_predicted
- W1, 0 = W1, 0 + learning_rate * (error * ho)
- W0, 1 = W0,1 + learning_rate * (error * W1,0 * AF`(w1 * x1 + w2 * x2) * x1)
- W0, 2 = W0,2 + learning_rate * (error * W1,0 * AF`(w1 * x1 + w2 * x2) * x2)

Neural Network - Implementation

```
import numpy as np


class NeuralNetwork(object):
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        self.input_nodes = input_nodes # 3
        self.hidden_nodes = hidden_nodes # 3
        self.output_nodes = output_nodes # 1

        # input_nodes X hidden_nodes matrix
        self.weights_input_to_hidden = np.random.normal(size=(self.input_nodes, self.hidden_nodes))
        # hidden_nodes X output_nodes matrix
        self.weights_hidden_to_output = np.random.normal(size=(self.hidden_nodes, self.output_nodes))
        self.lr = learning_rate
        self.activation_function = lambda x: 1 / (1 + np.exp(-x))  # sigmoid.

    def train(self, features, targets):
        n_records = features.shape[0]
        delta_weights_i_h = np.zeros(self.weights_input_to_hidden.shape)
        delta_weights_h_o = np.zeros(self.weights_hidden_to_output.shape)
        for X, y in zip(features, targets):
            final_outputs, hidden_outputs = self.forward_pass_train(X)
            delta_weights_i_h, delta_weights_h_o = self.backpropagation(final_outputs, hidden_outputs, X, y,
                                                                        delta_weights_i_h, delta_weights_h_o)
        self.update_weights(delta_weights_i_h, delta_weights_h_o, n_records)
```

Neural Network - Implementation

```
def forward_pass_train(self, X):
    hidden_inputs = np.dot(X, self.weights_input_to_hidden)
    hidden_outputs = self.activation_function(hidden_inputs)
    final_outputs = np.dot(hidden_outputs, self.weights_hidden_to_output)

    return final_outputs, hidden_outputs

def backpropagation(self, final_outputs, hidden_outputs, X, y, delta_weights_i_h, delta_weights_h_o):

    output_error_term = y - final_outputs

    hidden_error = np.dot(output_error_term, self.weights_hidden_to_output.T)
    # derivative of sigmoid function is f(x) * (1 - f(x))
    hidden_error_term = hidden_error * hidden_outputs * (1 - hidden_outputs)

    delta_weights_i_h += hidden_error_term * X[:, None]
    delta_weights_h_o += output_error_term * hidden_outputs[:, None]

    return delta_weights_i_h, delta_weights_h_o

def update_weights(self, delta_weights_i_h, delta_weights_h_o, n_records):
    self.weights_hidden_to_output += self.lr * delta_weights_h_o / n_records
    self.weights_input_to_hidden += self.lr * delta_weights_i_h / n_records
```

Neural Network - Implementation

```
input_nodes = 3
hidden_nodes = 3
output_nodes = 1
iterations = 100000
learning_rate = 0.3

X =np.array([[2, 3, 3], [4, 3, 3], [4, 3, 7], [4, 3, 6]])
y = np.array([2, 7, 1, 3])

nn = NeuralNetwork(input_nodes, hidden_nodes, output_nodes, learning_rate)
for i in range(iterations):
    nn.train(X, y)

print(nn.run(X))
```

```
[[ 2.]
 [ 7.]
 [ 1.]
 [ 3.]]
```

LR, NN implementation will be available at;
https://github.com/mokarakaya/machine-learning-presentation

Playground Tensorflow

- Circular Data (Relatively Simple)
  - One layer one neuron is linear
  - One layer two neurons is non-linear but not enough
  - One layer 3 neurons will learn
  - Does the model work with linear activation function?
- Spiral Data (Relatively Complex)
  - One layer 3 neurons is not enough
  - 8 X 8 X 4 model will learn (L2 = 0.001)
  - Nodes in the first hidden layer are always linear?
  - Nonlinear feature combinations
  - Nodes in the first hidden layer are always linear?

Try the statements above at https://playground.tensorflow.org

Convolutional Neural Networks (CNN)

7 as an image:

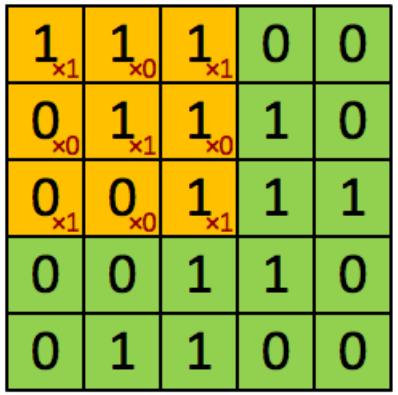
0 **1** **1 1 1 1**
0 0 0 0 0 **1**
0 0 0 0 **1** 0
0 0 0 **1** 0 0
0 0 **1** 0 0 0

Filter:

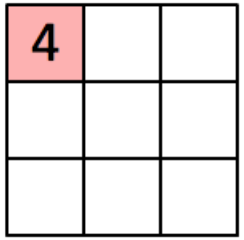
-1	-1	1
-1	1	-1
1	-1	-1

Convolutional Layer:

0	0	0
0	0	3
0	3	0



Image



Convolved Feature

Calculation for CL(0, 0):

ReLU (SUM (-1 * 0 + -1 * 1 + 1 * 1 +
-1 * 0 + 1 * 0 + -1 * 0 +
1 * 0 + -1 * 0 + -1 * 0))
= 0

Convolutional Neural Networks (CNN)

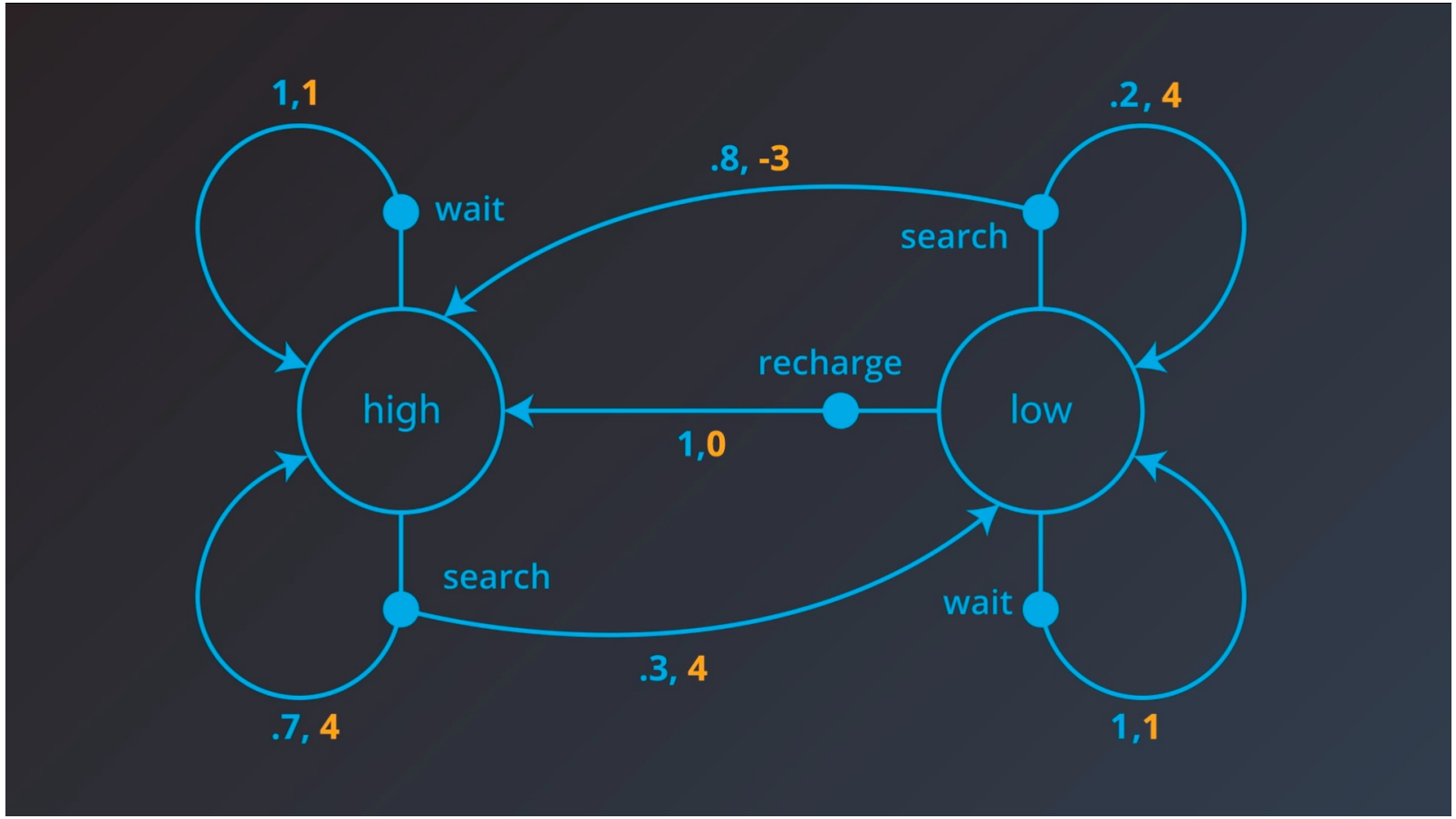
```python
import tensorflow as tf


def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
```

- Strides = height and weight of the convolution window (Generally square)
- The window will shift 1 in all directions.
- Padding SAME will extend the image with zeros if the window exceeds the image.

- In transfer learning, a model developed for a task that was reused as the starting point for a model on a second task
- First task may be predicting dog breeds and the second task may be predicting car models.
- Since first layers of NN consist of simple equations and they keep simple information (lines, edges, etc), reusing these layers in other task yields better results.
- Transfer learning is not limited to visual data.

Reinforcement learning (RL) - Markov Decision Process (MDP)

- The goal of the agent is to maximize expected cumulative reward.
- RL algorithms depends on MDP
- S = {high, low}
- A = {wait, search, recharge}
- R = labeled with yellow



TF-IDF

	Text in email (x1)	Class (y)
Email 1	Hello, sepa payment	1
Email 2	Hello, international payment	2
Email 3	Hello, sepa tax	3
Email 4	Hello, international tax	?

- tf(t,d) = 1 if t occurs in d and 0 otherwise
- idf(t, D) = log(N / | {d ∈ D : t ∈ d} |)
- tf-idf(t, d, D) = tf(t, d) * idf(t, D)

- D = all documents
- N = size of D
- d = a specific document
- t = term

TF-IDF

	Hello (x1)	Sepa (x2)	Payment (x3)	International (x4)	Tax (x5)	Class (y)
Email 1	1 * log(3/3) = 0.0	1 * log(3/2) = 0.1	1 * log(3/2) = 0.1	0 * log(3/1) = 0.0	0 * log(3/1) = 0.0	C 1
Email 2	1 * log(3/3) = 0.0	0 * log(3/2) = 0.0	1 * log(3/2) = 0.1	1 * log(3/1) = 0.4	0 * log(3/1) = 0.0	C 2
Email 3	1 * log(3/3) = 0.0	1 * log(3/2) = 0.1	0 * log(3/2) = 0.0	0 * log(3/1) = 0.0	1 * log(3/1) = 0.4	C 3

- Email 1 = Hello, sepa payment
- Email 2 = Hello, international payment
- Email 3 = Hello, sepa tax
- N-gram(1, 1) = 10848
- N-gram(1, 6) = 254062

- log(1) = 0
- log(3/2) = 0.17
- log(3) = 0.4