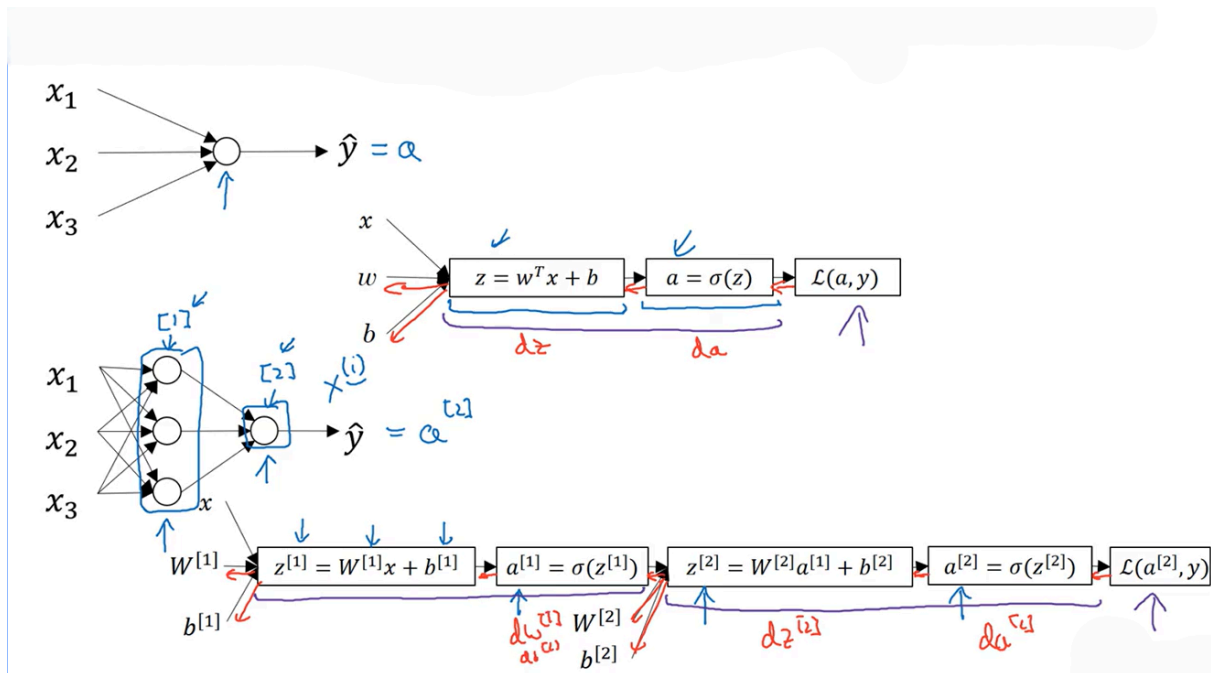


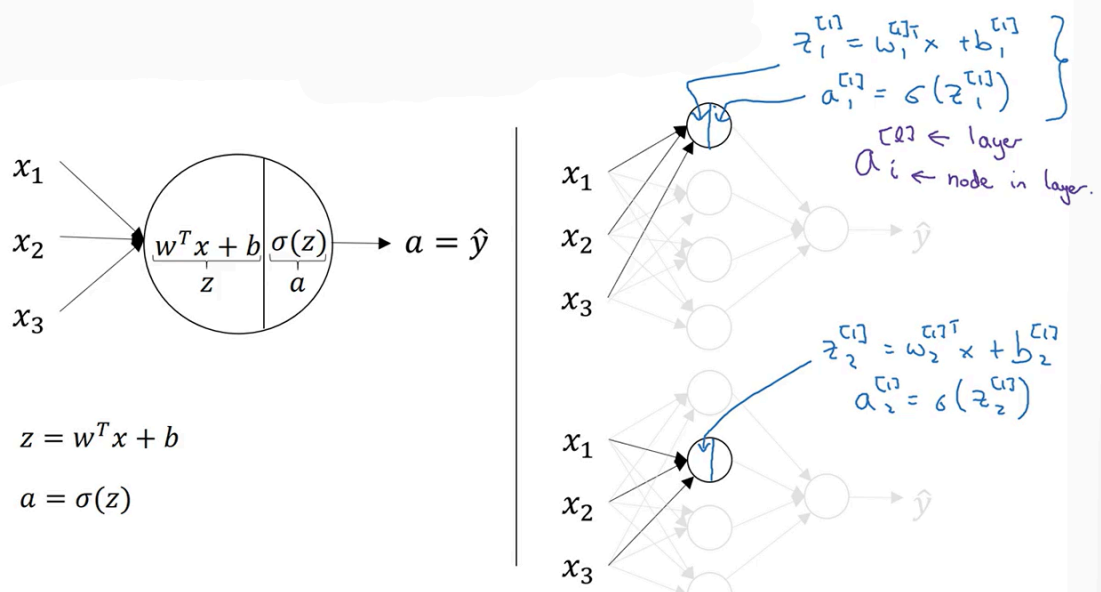
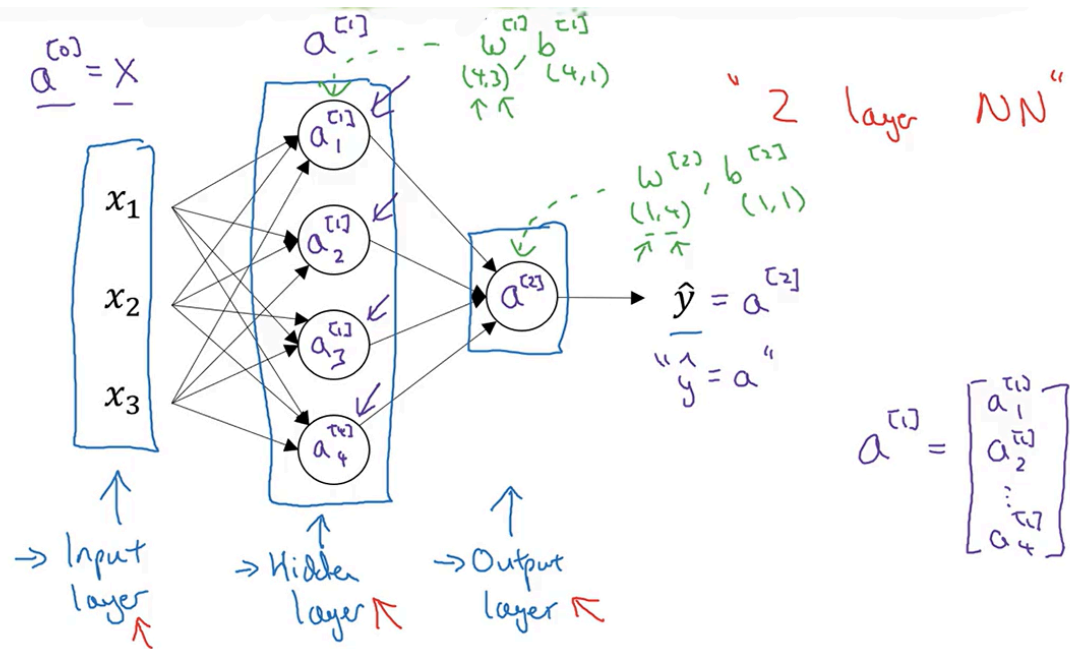
Week 3: One hidden layer Neural Networks

Neural Network Overview: Week2



we have seen single layer neural network earlier(week2), now dive into 2-layer NN.

Two-Layer Neural Network

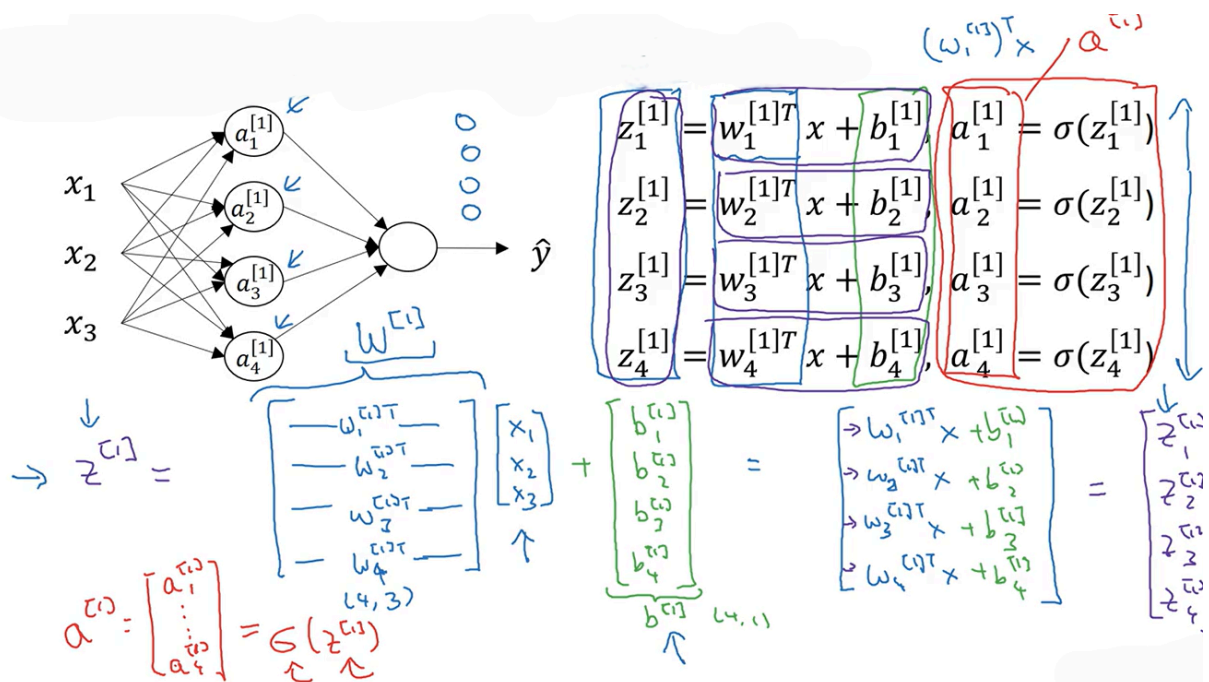


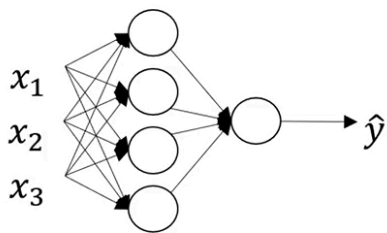
$$z_1^{[1]} = \underline{w_1^{[1]T}} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$



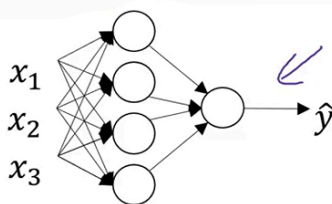


$$\begin{aligned}
 X &\rightarrow a^{[2]} = \hat{y} \\
 x^{(1)} &\rightarrow a^{[2](1)} = \hat{y}^{(1)} \\
 x^{(2)} &\rightarrow a^{2} = \hat{y}^{(2)} \\
 &\vdots \\
 x^{(m)} &\rightarrow a^{[2](m)} = \hat{y}^{(m)}
 \end{aligned}$$

$a^{[2](i)}$ ← example i layer 2

$$\left\{ \begin{aligned} z^{[1]} &= W^{[1]}x + b^{[1]} \\ a^{[1]} &= \sigma(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \end{aligned} \right\} \leftarrow$$

$$\begin{aligned}
 &\rightarrow \text{for } i = 1 \text{ to } m, \\
 &\quad z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]} \\
 &\quad a^{[1](i)} = \sigma(z^{[1](i)}) \\
 &\quad z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]} \\
 &\quad a^{[2](i)} = \sigma(z^{[2](i)})
 \end{aligned}$$



$$X = \begin{bmatrix} | & | & \dots & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & \dots & | \end{bmatrix} \leftarrow$$

$$A^{[1]} = \begin{bmatrix} | & | & \dots & | \\ a^{1} & a^{[1](2)} & \dots & a^{[1](m)} \\ | & | & \dots & | \end{bmatrix} \checkmark$$

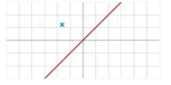
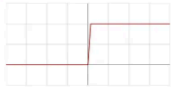
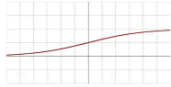
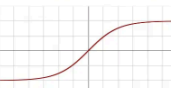

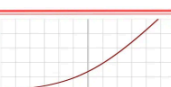


for $i = 1$ to m

$$\left\{ \begin{aligned} &z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]} \\ &\rightarrow a^{[1](i)} = \sigma(z^{[1](i)}) \\ &\rightarrow z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]} \\ &\rightarrow a^{[2](i)} = \sigma(z^{[2](i)}) \end{aligned} \right\}$$

$$\left\{ \begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \leftarrow W^{[1]}A^{[0]} + b^{[1]} \\ A^{[1]} &= \sigma(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(Z^{[2]}) \end{aligned} \right\}$$

$x = a^{[0]}$ $x^{(i)} = a^{[0](i)}$

Activation Functions:

ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Softplus		$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-1, 1)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$[0, \infty)$

<https://iq.opengenus.org/linear-activation-function/>

NOTE:

1. The natural choice for binary classification is Sigmoid Activation function almost never use this except for the final output layer.
2. For all other trending activation function is ReLU, and Tanh Activation function

Why do you need Non-Linear Activation Functions?

- Without non-linearity, deep networks would behave like a single layer perceptron
- Linear activation functions make neural networks unable to learn complex patterns

- Non-linear functions allow networks to approximate any function (universal approximation theorem)
- They enable the network to learn hierarchical features and representations
- Non-linear activations help models solve classification problems with non-linearly separable data
- They prevent the vanishing/exploding gradient problem (particularly ReLU and its variants)

we can use linear activation functions at output layers like house price prediction

Derivation of Activation Functions:

- mentioned above diagram see

Gradient Descent for Neural Networks:

The image shows handwritten mathematical derivations for forward and back propagation in a two-layer neural network. The left side details forward propagation, and the right side details back propagation with associated dimensions and array structures.

Forward propagation:

$$z^{[1]} = w^{[1]}x + b^{[1]}$$

$$A^{[1]} = g^{[1]}(z^{[1]}) \leftarrow$$

$$z^{[2]} = w^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(z^{[2]}) = \underline{\sigma}(z^{[2]})$$

Back propagation:

$$dz^{[2]} = A^{[2]} - Y \leftarrow$$

where $Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(n)}]$

$$dw^{[2]} = \frac{1}{n} dz^{[2]} A^{[1]T}$$

Dimensions: $(n, 1) \leftarrow (n, 1) \leftarrow (n^{[2]}, 1)$

$$db^{[2]} = \frac{1}{n} \text{np.sum}(dz^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$$

$$dz^{[1]} = \underbrace{w^{[2]T}}_{(n^{[1]}, m)} dz^{[2]} \times \underbrace{g^{[1]'}(z^{[1]})}_{\text{element-wise product}} \quad (n^{[1]}, m)$$

$$dw^{[1]} = \frac{1}{n} dz^{[1]} x^T$$

$$db^{[1]} = \frac{1}{n} \text{np.sum}(dz^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$$

Dimensions: $(n^{[1]}, 1) \quad (n^{[1]}, 1) \quad \text{axis} \uparrow$

Forward Propagation (left side)

The network consists of two layers ($[1]$ = first layer, $[2]$ = second layer):

Layer 1

```
Z1 = W1 @ X + b1    # Linear step (matrix multiplication + bias)
A1 = g1(Z1)         # Apply activation function g1 (e.g., ReLU, tanh, sigmoid)
```

- **W1**: weights of shape `(n_h, n_x)`
- **X**: input data `(n_x, m)`
- **b1**: bias `(n_h, 1)`
- **A1**: activation output `(n_h, m)`

Layer 2

```
Z2 = W2 @ A1 + b2    # Linear step for layer 2
A2 = g2(Z2)          # Output activation (often sigmoid for binary classification)
```

- **W2**: weights of shape `(n_y, n_h)`
- **A1**: previous activation `(n_h, m)`
- **b2**: bias `(n_y, 1)`
- **A2**: final output (predictions) `(n_y, m)`

Backward Propagation (right side)

Step 1: Output layer error

```
dZ2 = A2 - Y
```

- For logistic regression with sigmoid, `dZ2` simply represents the prediction error.
- **Y**: true labels `(n_y, m)`

Step 2: Gradients for W2, b2

```
dW2 = (1/m) * (dZ2 @ A1.T)
db2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)
```

- Shape of `dW2`: `(n_y, n_h)`
 - Shape of `db2`: `(n_y, 1)`
-

Step 3: Propagate error to layer 1

```
dZ1 = (W2.T @ dZ2) * g1_prime(Z1)
```

- `W2.T @ dZ2` propagates error contribution from layer 2
 - `g1_prime(Z1)` applies element-wise derivative of activation g1
-

Step 4: Gradients for W1, b1

```
dW1 = (1/m) * (dZ1 @ X.T)
db1 = (1/m) * np.sum(dZ1, axis=1, keepdims=True)
```

- Shape of `dW1`: `(n_h, n_x)`
 - Shape of `db1`: `(n_h, 1)`
-

Key Implementation Notes

- `keepdims=True` preserves bias shapes as `(n,1)` instead of collapsing to `(n,)`.
 - Shapes in purple ensure correct broadcasting.
 - `axis=1` performs summation across training examples.
-

```
import numpy as np

# ----- 1. Sigmoid Activation -----
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ----- 2. Forward + Backward Propagation -----
def propagate(W, b, X, Y):
    """
    W: weights (n_features, 1)
    b: bias (scalar)
```

```

X: input data (n_features, m_examples)
Y: labels (1, m_examples)
"""

m = X.shape[1] # number of examples

# ---- Forward propagation ----
Z = W.T @ X + b          # Linear part
A = sigmoid(Z)           # Activation
cost = -(1/m) * np.sum(Y*np.log(A) + (1-Y)*np.log(1-A)) # Loss

# ---- Backward propagation ----
dZ = A - Y               # Error
dW = (1/m) * (X @ dZ.T)   # Gradient w.r.t weights
db = (1/m) * np.sum(dZ)    # Gradient w.r.t bias

grads = {"dW": dW, "db": db}

return grads, cost

# ----- 3. Optimization -----
def optimize(W, b, X, Y, learning_rate=0.01, num_iterations=1000):
    costs = []
    for i in range(num_iterations):
        grads, cost = propagate(W, b, X, Y)

        # Update parameters
        W -= learning_rate * grads["dW"]
        b -= learning_rate * grads["db"]

        # Store cost every 100 iterations
        if i % 100 == 0:
            costs.append(cost)
            print(f"Iteration {i}: Cost = {cost:.4f}")

    return W, b, costs

# ----- 4. Prediction -----
def predict(W, b, X):

```

```

A = sigmoid(W.T @ X + b)
return (A > 0.5).astype(int)

# ----- 5. Example run -----
# Fake dataset (2 features, 4 examples)
X_train = np.array([[0.5, 1.0, -1.5, 2.0],
                    [1.0, -1.0, 2.0, 0.5]]) # Shape (2,4)

Y_train = np.array([[1, 0, 1, 0]]) # Shape (1,4)

# Initialize weights
W = np.zeros((X_train.shape[0], 1))
b = 0

# Train model
W, b, costs = optimize(W, b, X_train, Y_train, learning_rate=0.1, num_iteratio
ns=1000)

# Predict
preds = predict(W, b, X_train)
print("\nPredictions:", preds)

```