# Neural Networks

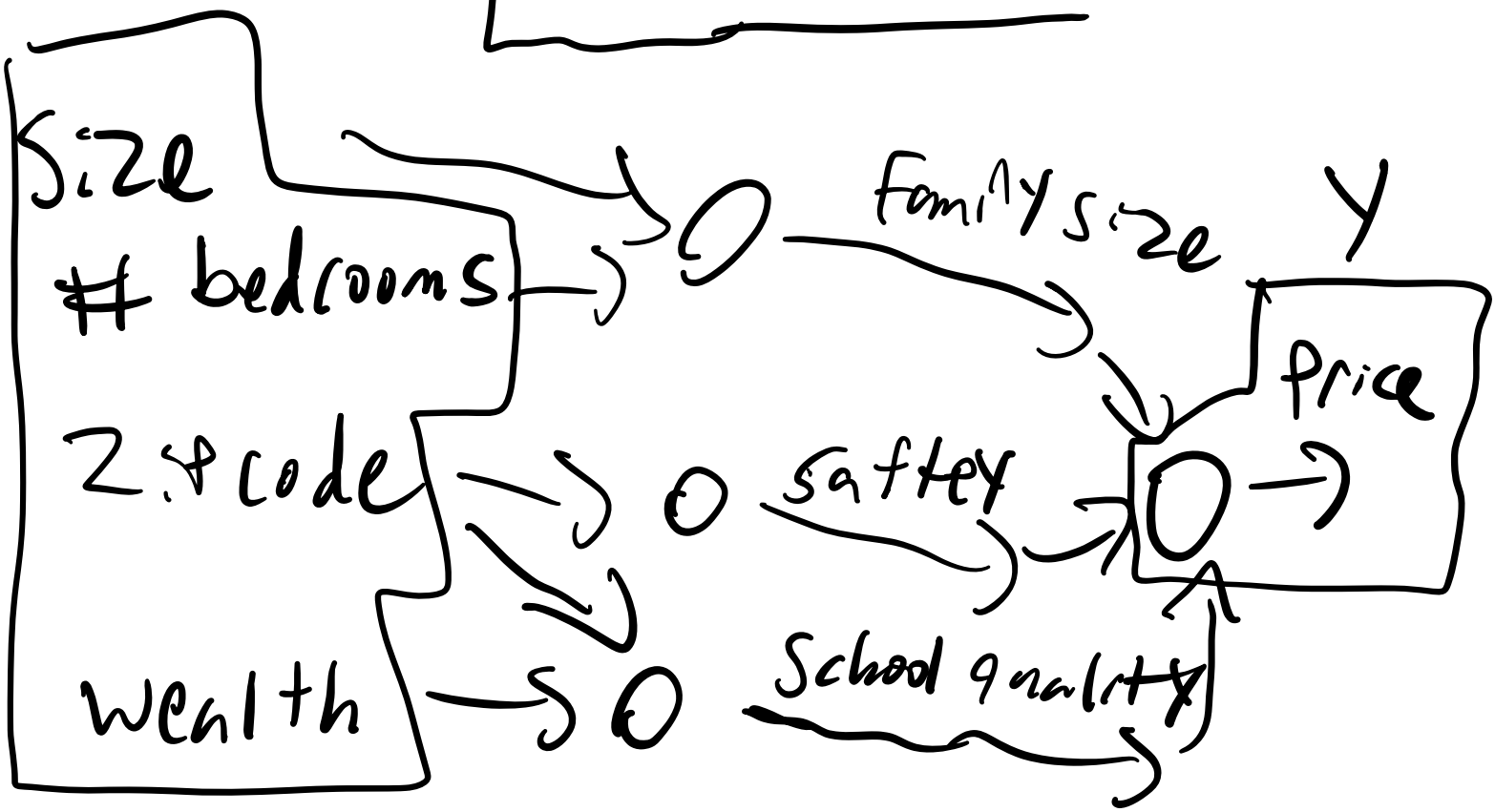

Size → ○ → Price
x ↑ y
"neuron"

* inputs size
* computes linear function
* takes max of zero
* outputs price

(function)

ReLU
Rectified (linear unit)
max of zero

x



↑

Neural Network will decide what this will be, all you need x and y

# Supervised learning

Application

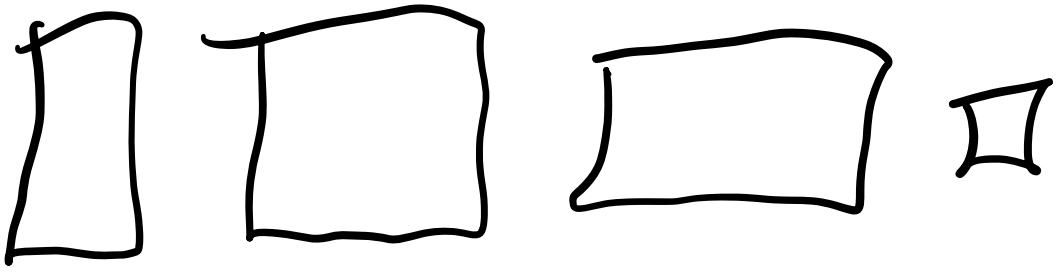
Standard NN -> Real estate, online ads

CNN -> photo tagging

RNN -> speech recognition, machine translation

Custom hybrid -> Autonomous driving

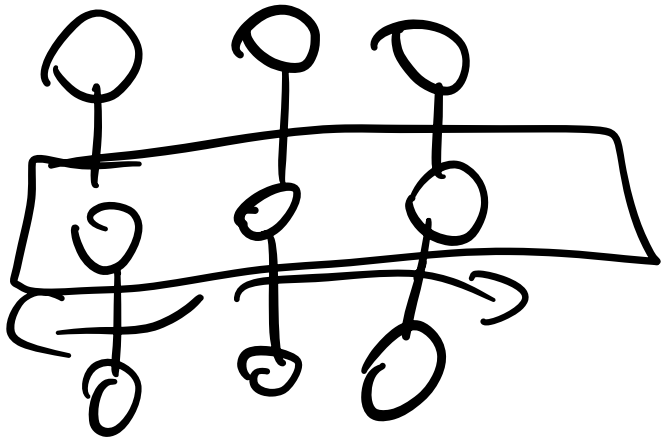
CNN ->



Image

RNNs

one dimensional sequence data



Structured data

vs

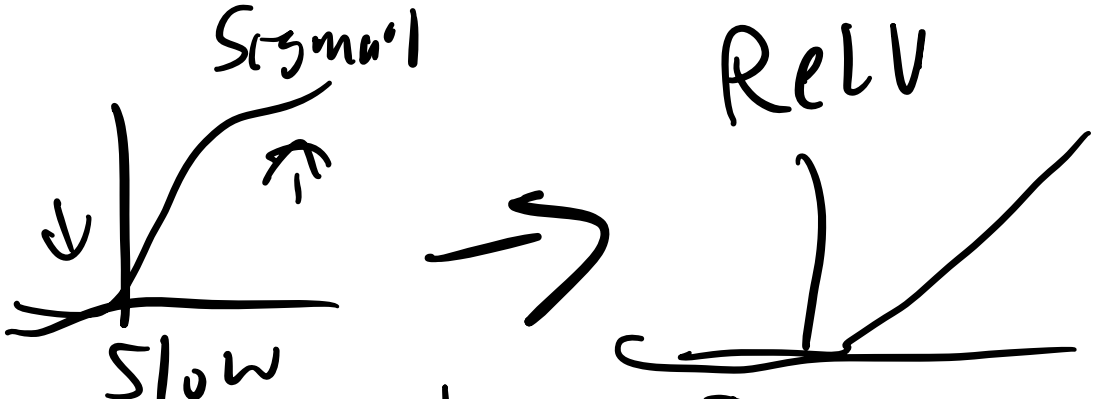
Unstructured data

well defined features such as a number representing user age

Audio and images or random text depending on more recognition

(pixels, individual words

# Scale drives deep learning progress*




Data
computation
Algorithms


makes
⇧ faster
⇒ "convergence"
less
steps
epoch

(m) = size of training set

---

# Binary Classification

Picture

of
Cat

→ 1 (cat)
0 (non cat)


matrices
(how computer stores image)

$\in$ unravel values

$$X = \begin{bmatrix} \text{all} \\ \text{pixels} \end{bmatrix}$$

$64 \times 64 \times 3 = 12288$

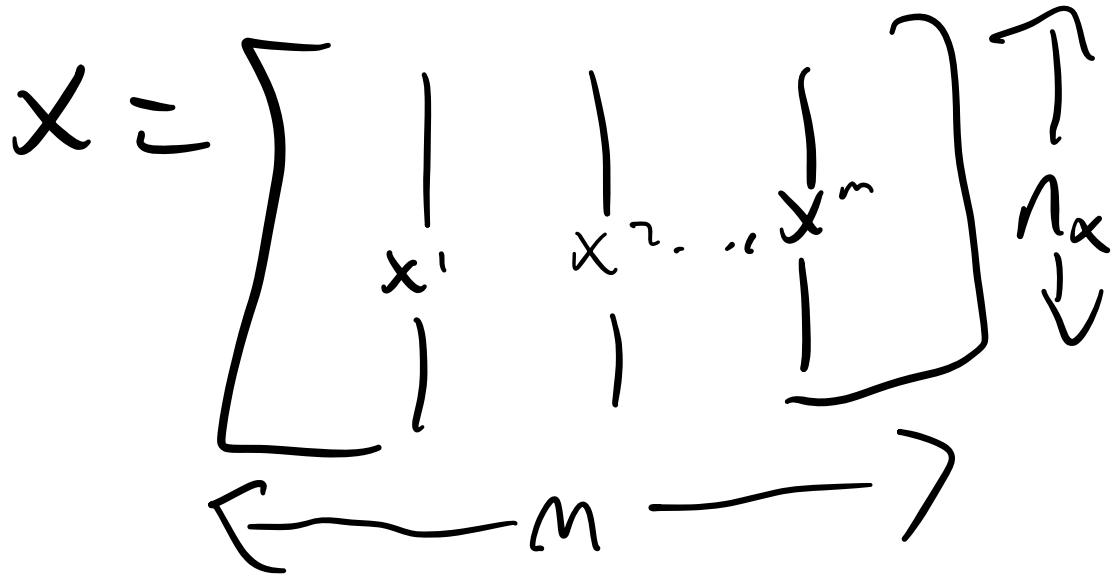
$n = n_x = 12288$

$X \rightarrow Y$

$(x, y)$ $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$

$m$ training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})\}$

$m = m_{train}$ $m_{test} =$ test




$X \in \mathbb{R}^{n_x \times m}$ X. Shape $= (n_x, m)$

$Y = [y^1, y^2, \ldots y^m]$

$Y \in \mathbb{R}^{1 \times m}$

Y Shape $= (1, m)$

$\mathbb{R}$ real Num

# Logistic Regression

predicts 1 or 0 values

Given X want $\hat{y} = P(y=1|x)$

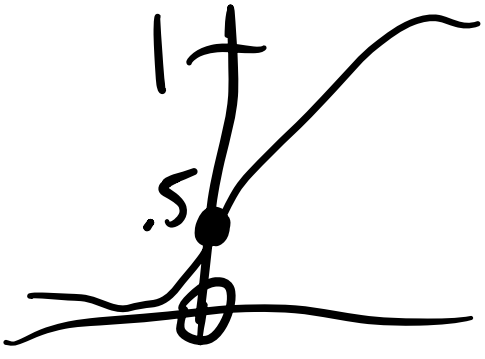
$X \in \mathbb{R}^{n_x}$ $0 \leq \hat{y} \leq 1$

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output: $\hat{y} = \sigma(w^T x + b)$

Sigmoid $z$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$




If $z$ is large

$$\sigma(z) \approx \frac{1}{1+0} = 1$$

If $z$ is large negative

$$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{big num}} \approx 0$$

---

# Logistic Regression Cost function

$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$ where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

Given $\{(x^1, y^1), \ldots, (x^m, y^m)\}$ want $\hat{y}^{(i)} \approx y^{(i)}$

$z^{(i)} = w^T x^{(i)} + b$ $x^{(i)}, y^{(i)}, z^{(i)}$ : ith example

Loss (error) function:

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

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$

want $\hat{y}$ to be as big as possible so $\log \hat{y}$ is big

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$

want $\log 1-\hat{y}$ to be large, want $\hat{y}$ small

Cost function:

$$J(w, b) = \frac{1}{m} \sum \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) =$$

$\sum_{i=1}^{m}$ ↑ ↓

Prediction output

$$-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)}+(1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

want to find w and b that

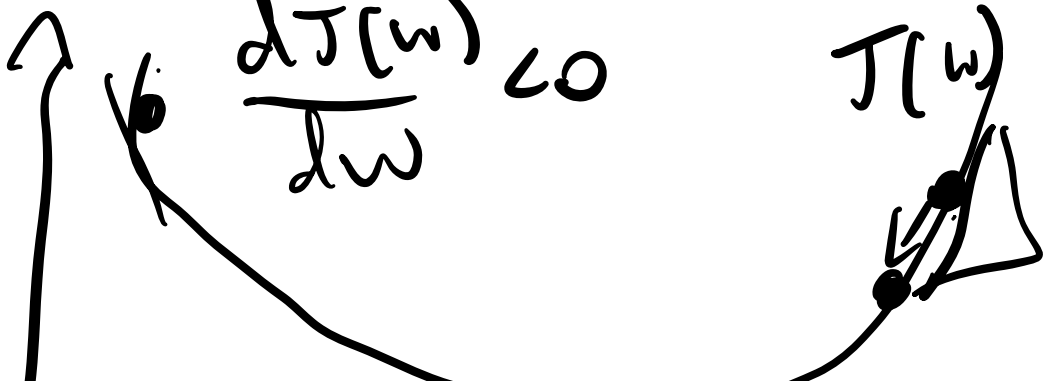
Minimize J(Cost function)

Loss - single training example

Cost - average loss entire data set

---

# Gradient Descent




Repeat {

$$w := w - \alpha \frac{dJ(w)}{dw}$$

→ derivative ("update")

learning rate — controls how big a step you take

"dw" — slope of function

}

$$w := w - \alpha dw$$

If you start off with large value of w, gradient will make you decrease slowly and positive slope

If you start out with small number and negative slope it increases

Both of these ideally come to middle which is best cost

$J(w,b)$ $w := w - \alpha \frac{dJ(w,b)}{dw}$

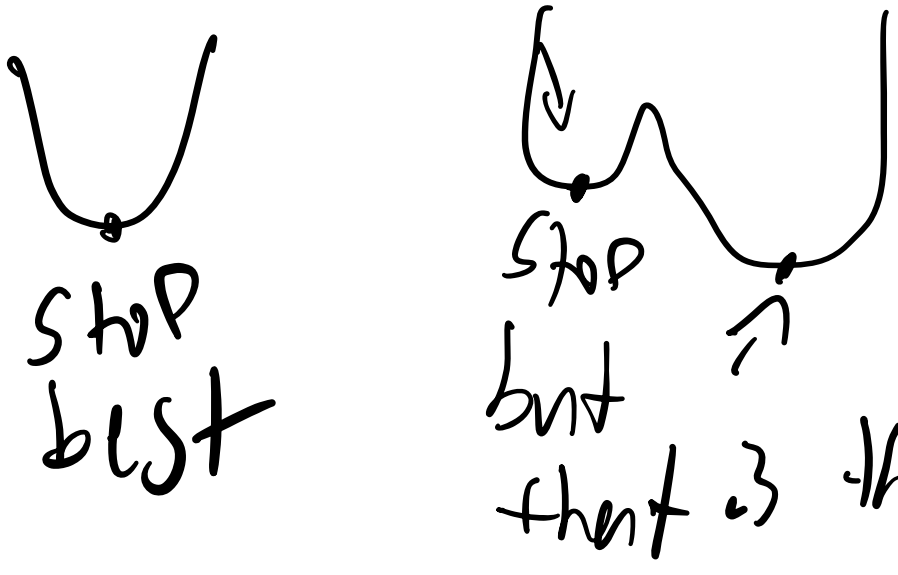
$b := b - \alpha \frac{dJ(w,b)}{db}$

$w = dw$
$B = db$

incode

A convex function does not always have multiple local optima.*

# Ada vs Adam




---

## Derivatives

ie: "this goes one"

$$\frac{d\,f(a)}{da} = 3 = \frac{d}{da} f(a)$$

"this goes 3 times"

$$\frac{d}{da} a^2 = 2a \qquad 0.001$$

$$(2a) \times .0001$$

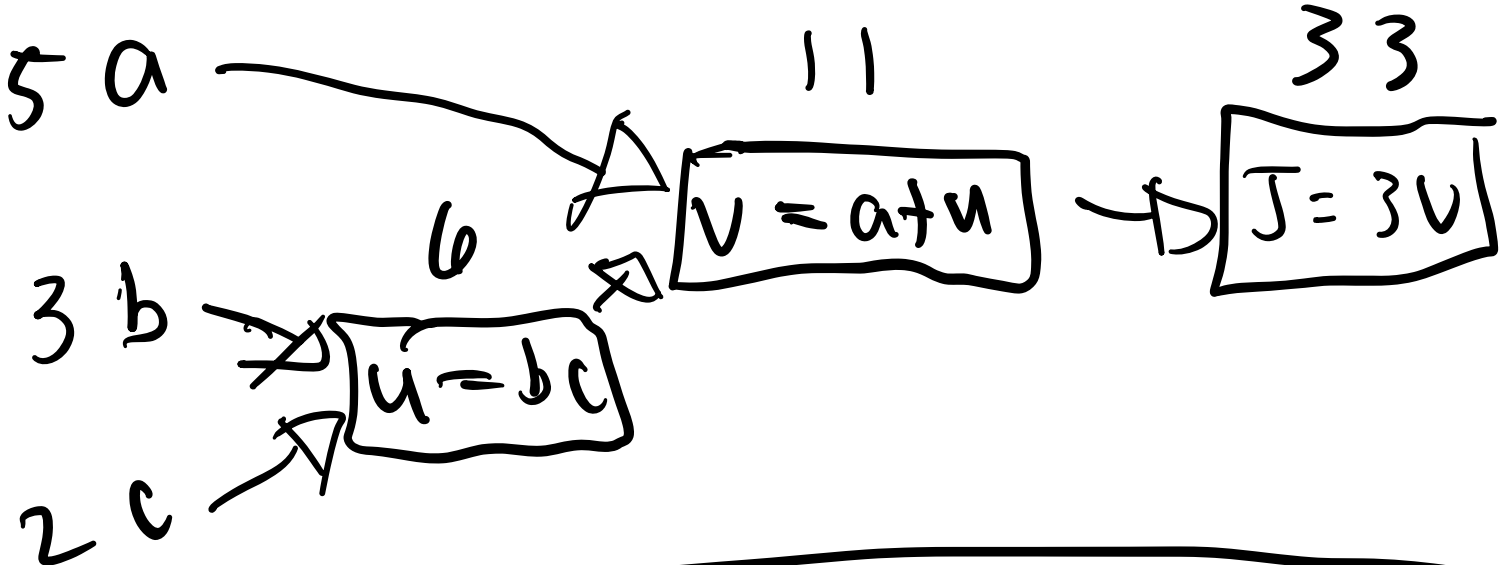
# Computation Graph

$J(a,b,c) = 3(a+bc)$

$u = bc$

$v = a+u$

$J = 3v$




# Combining Last few things

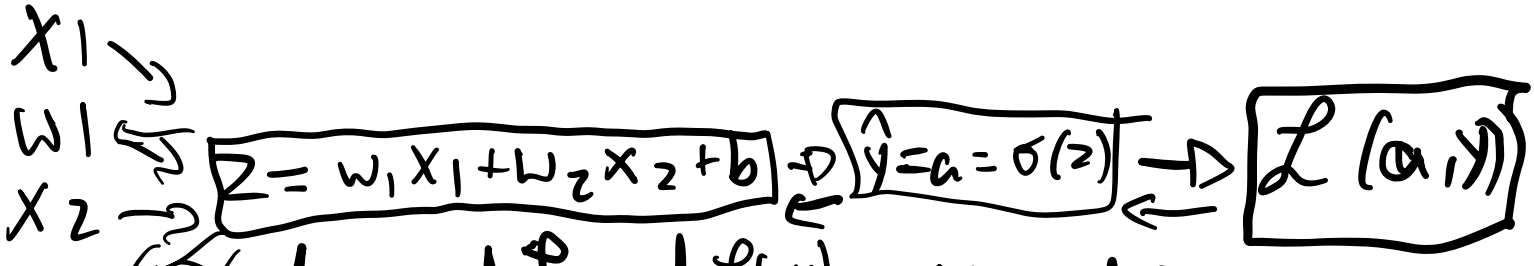
## Linear Regression

$z = w^T x + b$

$z = w^T x + b$

$\hat{y} = a = \sigma(z)$

$\mathcal{L}(a, y) = -(y \log(a) + (1-y)\log(1-a))$




$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$ "$da$" $= \frac{d\mathcal{L}(a,y)}{da}$

$= a - y$ $= -\frac{y}{a} + \frac{1-y}{1-a}$

$\frac{2\mathcal{L}}{2w_1} =$ "$dw_1$" $= x_1 \cdot dz$

"$dw_2$" $= x_2 \cdot dz$

"$db$" $= dz$

$w_1 = w_1 - \alpha dw_1$

$w_2 = w_2 - \alpha dw_2$

$b = b - \alpha db$

---

$J = 0 \quad dw = 0 \quad dw_2 = 0 \quad db = 0$

for $i = 1$ to $m$

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J += [y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 += x_1^{(i)} dz^{(i)}$ ⟷ $n = 2$

$dw_2 += x_2^{(i)} dz^{(i)}$

$db += dz^{(i)}$

$J /= m \quad dw_1 /= m \quad dw_2 /= m \quad db /= m$

---

Vectorization (getting rid of for loops)

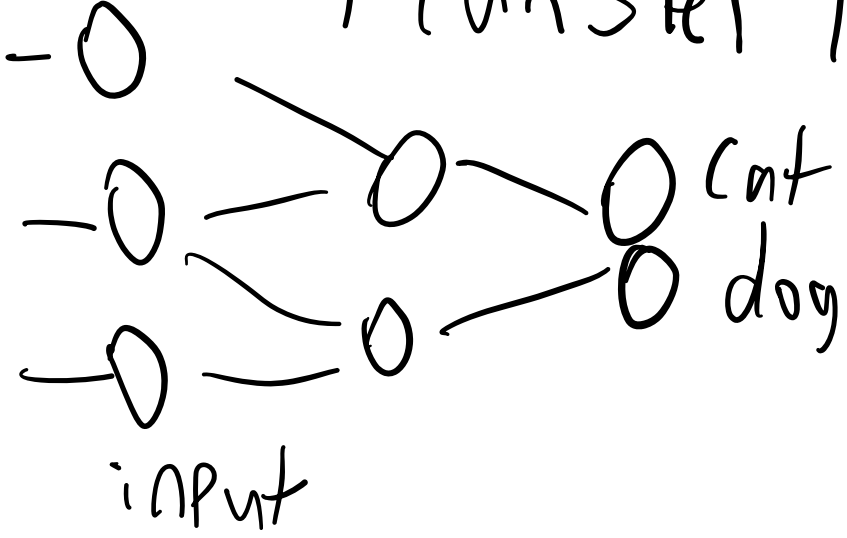
Back Propagation

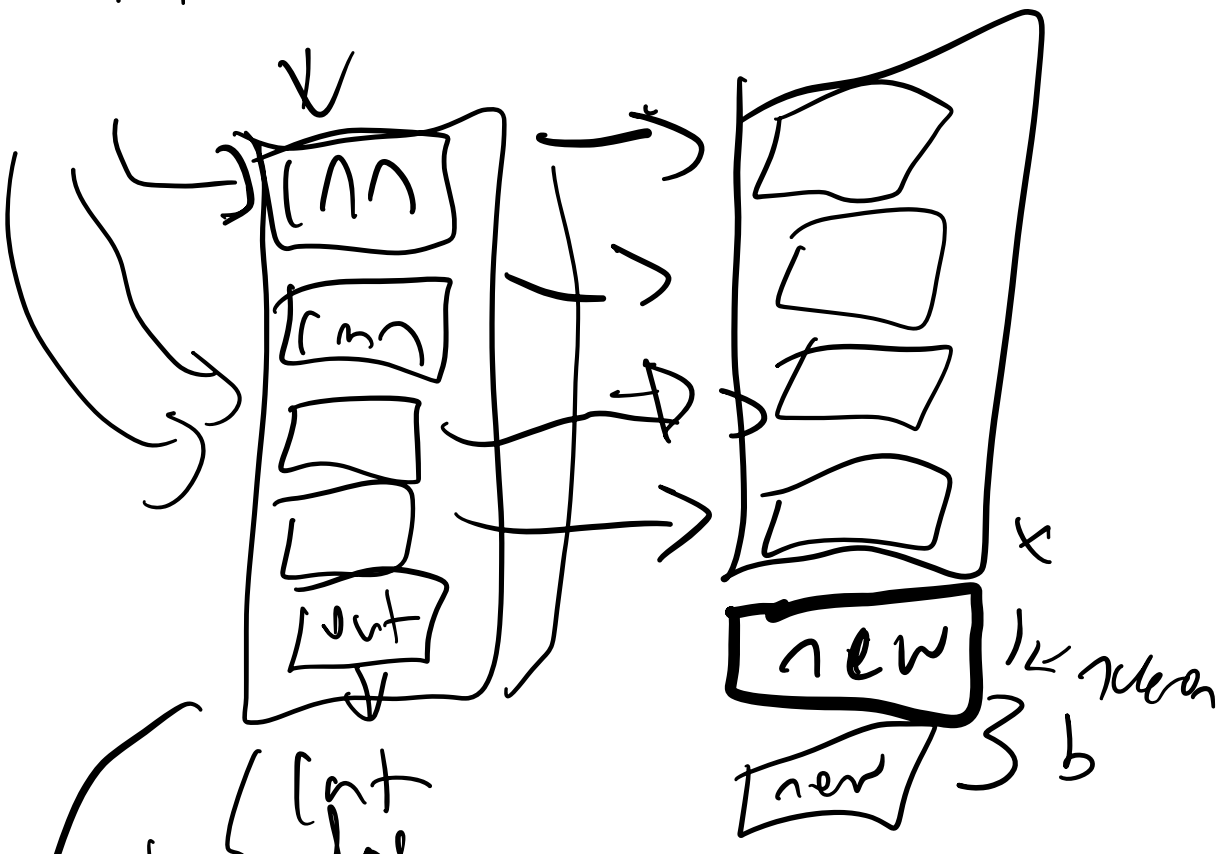
using chain rule from gradient

call derivative

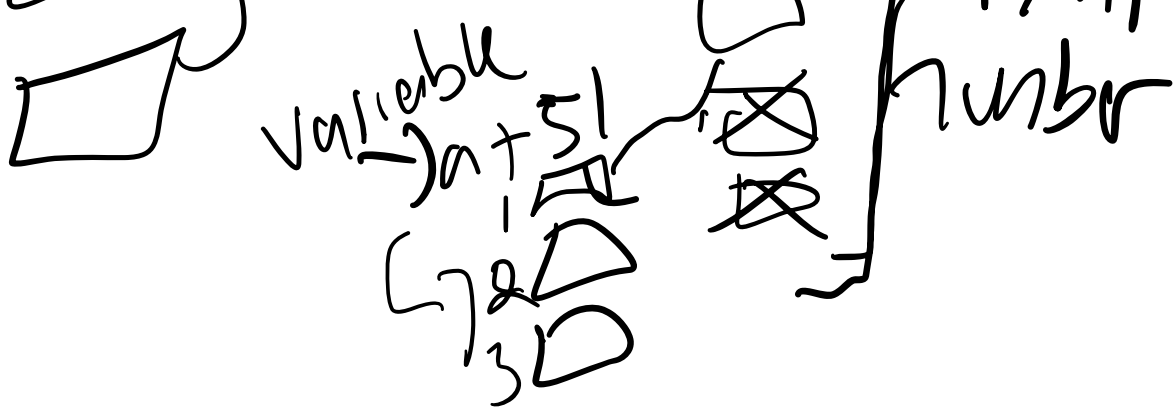
---

Transfer learning




ResNet
PyTorch




Sample classification
Tensorflow Sample codes
Transfer learning.

---

Data Augmentation
ie
Rotated image of
Cats workstop

---

Wights Pretrained set

# Sigmoid - logistic

$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$ regression function

Use "np.whatever()"

if $x = np.array([1,2,3])$ then $np.exp(x)$

does exp for every val in array

same if its print(x+3) -> [4,5,6]

For $x \in R^n$, $\text{Sigmoid}(x) = \text{Sigmoid}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \frac{1}{1+e^{-x_1}} \\ \frac{1}{1+e^{-x_2}} \\ \vdots \\ \frac{1}{1+e^{-x_n}} \end{pmatrix}$

Coding Implementation

```
def Sigmoid(x):
    s = 1/(1+np.exp(-x))
    return s
```

# Sigmoid Gradient

optimization using back Propagation

**Sigmoid derivative:**

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Coding Implementation:

```
def sigmoid_derivative(x):
    s = sigmoid(x)
    ds = s * (1-s)
return ds
```

# Reshaping

NP.shape - tells you shape

NP.reshape - this is how you turn a for example image input of (length, height, width) into a 1D array

So it simple for the AI training

use reshape(-1,1) -1 just takes everything ie for images it takes length, hiesht, and width and multiplies.

Coding Implementation:

```
def image2vector(image):
    v = v.reshape(-1,1).T
return v
```

.T is transpose which is a needed to make the training example a long

# Normalizing

keep gradients (derivatives) more consistent which helps it converge faster during training

$$X \to \frac{X}{\|X\|} \qquad \|X\| = \sqrt{X_1^2 + X_2^2 + \dots + X_n^2}$$

ie $X = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \end{bmatrix}$

$\|X\|$ = np.linalg.norm(X, axis=1, keepdims=True)

$= \begin{bmatrix} 5 \\ \sqrt{56} \end{bmatrix}$ X_normalized $= \begin{bmatrix} 0 & \frac{3}{5} & \frac{4}{5} \\ \frac{2}{\sqrt{56}} & \frac{6}{\sqrt{56}} & \frac{4}{\sqrt{56}} \end{bmatrix}$

keepdims = True: keeps it a 2D array so it can be broadcasted back to original matrix during division.

axis=1: Calculates norm row by row

Coding Implementation:

```
def normalize_rows(x):
    norm = np.linalg.norm(x, axis=1, keepdims=True)
    x = x / norm
return x
```

---

# Softmax (Broadcasting)

evidence

```
def softmax(x):
    x_exp = np.exp(x)
    x_sum = np.sum(x, axis=1, keepdims=True)
    s = x_exp/x_sum
return s
```

---

# Vectorization

## Dot Product

`np.dot(x1, x2)` $\sum_{i=1}^{n} x1[:] \cdot x2[:]$

Could be used for weighted sum of inputs

$x1 = [1, 2, 3]$
$x2 = [4, 5, 6]$ → $dot = (1 \times 4) + (2 \times 5) + (3 \times 6)$

## Outer Product

$outer(x1, x2)[i, j] = x1[i] \cdot x2[j]$

`np.outer(x1, x2)`

Used for constructing matrices from vectors

$x1 = [1, 2]$
$x2 = [3, 4]$ → $\begin{bmatrix} 1\times3 & 1\times4 \\ 2\times3 & 2\times4 \end{bmatrix}$ (outer) → $\begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}$ result

## Element wise multiplication

`np.multiply(x1, x2)` $elementwise(x1, x2)[i] = x1[i] \cdot x2[i]$

used for feature wide scaling

$x1 = [1, 2, 3]$ → $[1\times4, 2\times5, 3\times6]$ Mult

x2 = [4, 5, 6] [4, 10, 18] result

## General Dot Product

np.dot(W, x1) gdot(W, x1)[i] = $\sum_j W[i,j] \cdot x1[j]$

Core operation in Linear Regression

Calculating weighted sums in hidden layers

W = ( [1, 2], [3, 4], [5, 6] )

[1×5 + 2×6, 3×5 + 4×6] → dot → [17, 39] result

x1 = ([5, 6])

---

# Loss functions

sparse data

heavy errors

less sensitive to outliers

```
def L1(Yhat, Y):
    "Yhat = Predicted"
    "Y = true"
    loss = np.sum(Y-Yhat)
```

$L_1(\vec{Y}, Y) = \sum_{i=0}^{N-1} |y[i] - \hat{y}[i]|$

return loss

---

def L2(yhat, y):

$$L_1(\hat{y}, y) = \sum_{i=0}^{m-1} (y^{(i)} - \hat{y}^{(i)})^2$$

loss = np.sum(np.square(y - yhat))

return loss

minimize lugeerror
Balanced data
more sensitive
to outliers

---

# CNN Logistic Regression example

We have

1* train_set_x_orig which is numpy array of shape (M_train, num_px, num_px, 3)

* train_set_x_orig[0] which is M_train is the number of training examples

* train_set_x_orig[1] which is num_px is

the height of the image

2* Flatten it for the model to process

train_set_x_flatten = tran_set_x_orig.reshape(
transet_x_orig.shape[0], -1).T

-1 : flattens all other dimensions

.T : transposes to a column required by model

3* Standardize the dataset

train_set_x = tran_set_flatten/255

# Steps to build Newal Network

0. preprocess data

1. Define model structure (ie: # of input features)

2. Initialize model's parameters

3. Loop

* Calculate current loss (foward propogation)
* Calculate current gradient (backward propogation)
* update parameters (gradient descent)

## Helper function

Sigmoid(z):

```
s = 1/(1+np.exp(-z))
return s
```

---

Initialize Parameters

```
initialize_with_zeros(dim):
    w = np.zeros((dim,1)) # column vector of zeros
    b = 0.0 # bias scalar
return w, b
```

---

Forward + Backward Propogation

```
Propogate(w,b,X,Y):
#cost function & gradient
#Arguments
"""
W - weights (numpy array of size (num_px*
num_px*3,1)
b - bias scalar
X - data of size (num_px * num_px* 3, # of examples)
Y - true label vector (0: if non cat 1: if cat)
```

# Return
grads - dictionary containing gradients
(dw - gradient of loss with respect to w)
(db - gradient of loss with respect to b)
cost - negative log-likelihood cost for logistic regression
'''
m = X.shape[1]
#forward propagation
#Sigmoid activation function
A = sigmoid(np.dot(w.T, X) + b)
#inside of here is linear function
#now comes log loss function
cost = -(1/m) * np.sum(Y * np.log(A) + (1-Y) * np.log(1-A))
#Backward propagation

dw = (1/m) * np.dot(X, (A-Y).T)

# gradient of w

# gradient of b

db = (1/m) * np.sum(A-Y)

cost = np.squeeze(np.array(cost))

grad = {"dw": dw, "db": db}

return grads, cost

---

Optimization

# runs a gradient descent algorithm

def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009, print_cost=False)

# updating original ones

```
w = copy.deepcopy(w)
b = copy.deepcopy(b)
# list all costs during optimization
to plot learning curve
costs = []
for i in range(num_iterations):
#cost & gradient calculation
    grad, cost = propagate(w, b, X, Y)
    #get derivatives
     dw = grads["dw"]
     db = grads["db"]
     #update rule
      w = w - learning_rate * dw
      b = b - learning_rate * db
```

```
        #record cost
        if i % 100 == 0:
            costs.append(cost)
        #Print cost every 100 training
        #iterations
        if print_cost:
            print("Cost after iteration
                  %i: %f" %(i, cost))

    params = {"w": w, "b": b}
    grads = {"dw": dw, "db": db}
    return params, grads, costs
```

Predict whether label is 0 or 1

```
def predict(w, b, X):
    m = X.shape[1]
```

```
Y_prediction = np.zeros((1, m))
# initialization
#make sure w is right shape
w = w.reshape(X.shape[0], 1)
# activation
A = sigmoid((np.dot(w.T, X) + b))
#prediction
for i in range(A.shape[1]):
    if A[0, i] > 0.5:
        Y_prediction[0, i] = 1
    else:
        Y_prediction[0, i] = 0
return Y_prediction
```

Merge all functions into model

```
def model(X_train, Y_train, X_test,
Y_test, num_iterations=2000, learning_
rate=0.5, print_cost=false):
#initialize
W, b=initialize_with_zeros(X_train.shape[0]
#gradient descent
params, grads, costs = optimize(W, b,
X_train, Y_train, num_iterations, learning_
rate, print_cost)
# retrieve parameters
W = params["w"]
b = params["b"]
#predict on training set
Y_prediction_train = predict(W, b, X_train)
#predict on test set
Y_prediction_test = predict(W, b, X_test)
```

```
# Print errors
print("train accuracy: {} %".format(100 - np.mean(
np.abs(Y_prediction_train - Y_train)) * 100))
print("test accuracy: {} %".format(100 - np.mean(
np.abs(Y_prediction_test - Y_test)) * 100))
# Store relevant outputs
d = {
    "costs": cost
    "Y_prediction_test": Y_prediction_test
    "Y_prediction_train": Y_prediction_train
     "w": w
     "b": b
     "learning_rate": learning_rate,
     "num_iterations": num_iterations }

return d
```

# Neural Network

How to tell Shape: numpyarray.shape
# of trainy examples: m = nparray.Shape[1]

Methodology:

1. Define neural network structure (# of inputs, # of hidden units, etc.)
2. Initialize models parameters
3. Loop
   - Implement forward Propogation
   - compute loss
   - Implement backward Propogation (get gradients)
   - update parameters (gradient descent)

---

## 1. Define structure

```
def layer_sizes(X,Y):
# X - data set size (inputsize, # of examples)
# Y - labels of shape (outputsize, # of examples)
```

```
n_x = X.shape[0] # input layer size
n_h = 4 # hidden layer size
n_y = Y.shape[0] # output layer size

return n_x, n_h, n_y
```

## 2. Initialize parameters

```
def initialize_parameters(n_x, n_h, n_y):
#returns: params (dictionary)
    W1 - weight matrix of shape (n_h, n_x)
    b1 - bias vector of shape (n_h, 1)
    W2 - weight matrix of shape (n_y, n_h)
    b2 - bias vector of shape (n_y, 1)
W1 = np.random.rand(n_h, n_x) * .01
b1 = np.zeros((n_h, 1))
W2 = np.random.rand(n_y, n_h) * .01
b2 = np.zeros((n_y, 1))
Parameters = {"W1"
                etc.

return parameters
```

# 3. The loop

```
def forward_propagation(X, parameters):
#X input data of size (n_x, m)
W1 = parameters["W1"]
etc. W2, b1, b2
Z1 = np.dot(W1, X) + b1
A1 = np.tanh(Z1)
Z2 = np.dot(W2, A1) + b2
A2 = sigmoid(Z2)
assert(A2.shape == (1, X.shape[1]))
cache = {"Z1": Z1
         etc.
return A2, cache
```

```
def compute_cost(AL, Y):
    m = Y.shape[1] # of examples
    # cross entropy cost
    logprobs = np.multiply(np.log(AL), Y) +
    np.multiply(np.log(1 - AL), (1 - Y))
    # APPLY full cost function formula
    cost = -np.sum(logprobs) / m
    cost = float(np.squeeze(cost))
    # makes sure dimension is how
    we expect eg: [[17]] -> 17
    return cost
```

```
def back_propagation(parameters, cache, X, Y):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    A1 = cache["A1"]
    A2 = cache["A2"]
    dZ2 = A2 - Y # compute error at output layer
    dW2 = (1/m) * np.dot(dZ2, A1.T) #Gradient of W2
    #gradient of b2
    db2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)
    #Back Propagate through tanh activation
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW2 = (1/m) * np.dot(dZ1, X.T) #Gradient of W1
    #gradient of b1
```

db2 = (1/m) * np.sum(dZ1, axis=1, keepdims=True)

grads = {dW1, db1, dW2, db2}

return grads

---

def update_parameters(parameters, grads,
learning_rate = 1.2):

W1 = copy.deepcopy(parameters["W1"])

b1, W2, b2 (//)

# deepcopy of parameters

# retrieve each gradient

dW1 = grads["dW1"]

db1, dW2, db2 (//)

# update rule

W1 = W1 - learning_rate * dW1

b1, W2, b2 (//)

parameters = {W1, b1, W2, b2}

return parameters

Integration into model

```
def nn_model(X, Y, n_h, num_iterations=10000, print_cost=False):
    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]
    # initialize parameters
    parameters = initialize_parameters(n_x, n_h, n_y)
    # gradient descent loop
    for i in range(num_iterations):
        # forward propagation
        A2, cache = forward_propagation(X, parameters)
        # compute cost
        cost = compute_cost(A2, Y)
        # back propagation
        grads = backward_propagation(parameters, cache, X, Y)
        # update parameters
```

```
# Update parameters
Parameters = update_parameters(parameters, grads)
    if print_cost and i%1000 == 0:
        print("cost after iteration i, cost)
return parameters
```

# Test model

```
def predict(parameters, X):
# forward propagation to get prediction
A2, cache = forward_propagation(X, parameters)
#convert probabilities into binary predictions
predictions = (A2>.5).astype(int)
return predictions
```