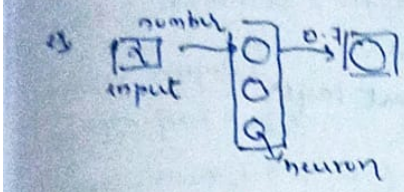
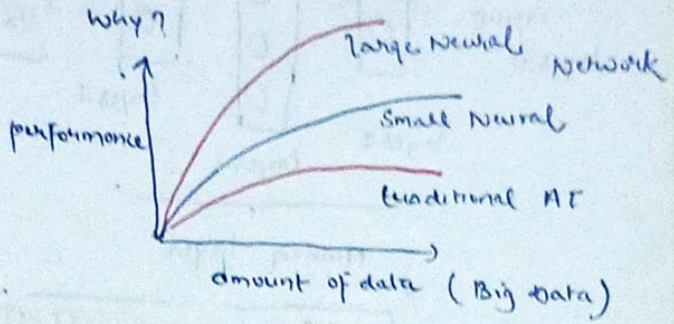
Week_2 Machine learning

Advanced learning Algorithms

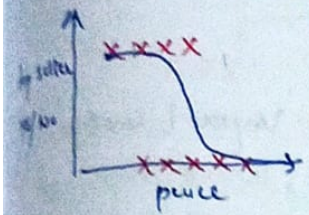
Neural Networks

↳ Algorithms that try to mimic the brain.





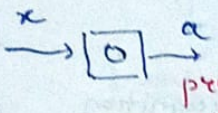
Demand Prediction



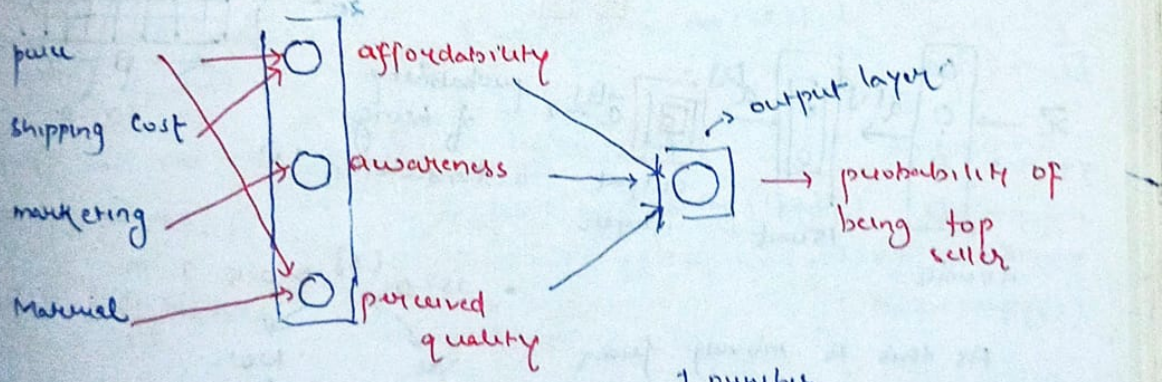
x = price input

$a = f(x) = \frac{1}{1+e^{-(wx+b)}}$ output

↳ activation



more features

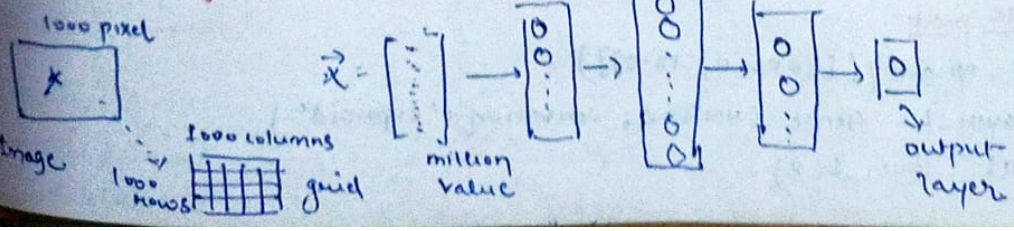


4 numbers (Input layer)

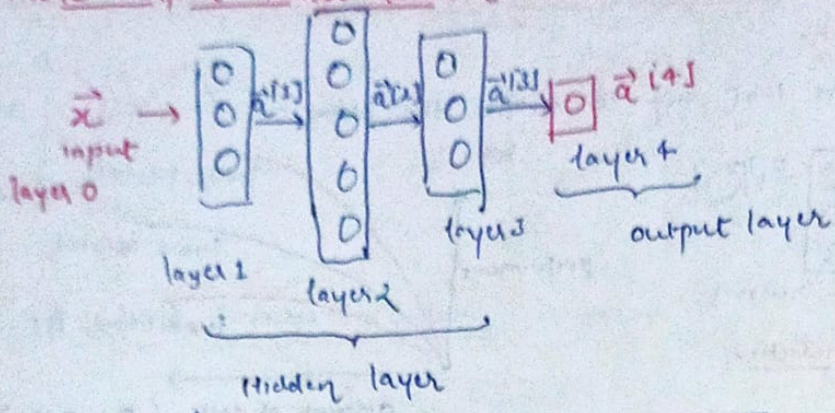
"activations" 3 numbers (hidden layer)

1 number

Face recognition



More complex neural network



$$a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$$

Activation value of layer l

Parameter w & b of layer l unit j

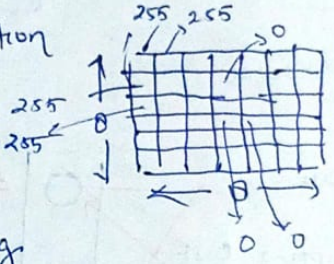
{ sigmoid "activation function" }

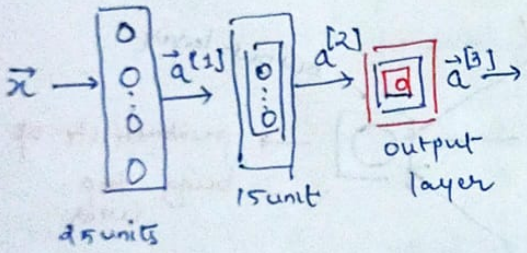
Forward Propagation.

ex Handwritten digit recognition

Digit image

0 1





probability of being a handwritten '1'

- is $a_1^{[3]} \geq 0.5$?
  - yes → $\hat{y} = 1$
  - no → $\hat{y} = 0$

As this is moving from $a^{[1]} \rightarrow a^{[2]} \rightarrow a^{[3]}$

∴ It is known as forward propagation.

output.

Code part:

```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
```

Building a neural network architecture

```
layer_1 = Dense(units=3, activation="sigmoid")
layer_2 = Dense(units=1, activation="sigmoid")
model = Sequential([layer_1, layer_2])
```

Syntax:

```
model.compile()
model.fit(X,Y)
model.predict(X_new)
```

AI

- ANI (artificial narrow intelligence): smart speaker, self driving car, web search, AI in farming and factories
- AGI (artificial general intelligence): do anything a human can do

Model Training Steps:

1. specify how to compute output given input X and parameters w, b (define model) $f_{w,b}(\vec{x}) = ?$
2. specify loss and cost
   $L(f_{\vec{w},b}(\vec{x}), y)$
   $J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$
3. Train on data to minimize cost function

logistic regression

$z = np.dot(w, x) + b$

$f_x = 1/(1+np.exp(-z))$

logistic loss

$loss = -y * np.log(f_x) - (1-y) * np.log(1-f_x)$

neural network

```
model = Sequential([Dense(....)
                    Dense(....)
                    Dense(....)])
```

Binary cross Entropy

```
model.compile(loss=BinaryCrossentropy())
```

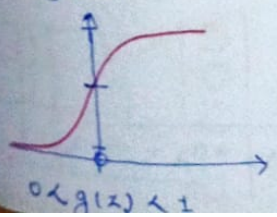
$w = w - alpha * dj\_dw$

$b = b - alpha * dj\_db$

```
model.fit(X,Y, epochs=100)
```

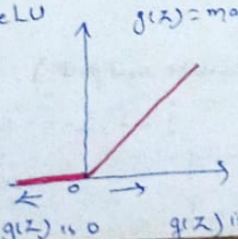
Alternatives to the Sigmoid Activation

Sigmoid



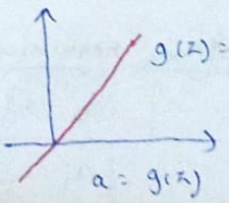
$0 < g(z) < 1$

ReLU $g(z) = max(0,z)$



$g(z)$ is 0 $g(z)$ is z

Linear Activation



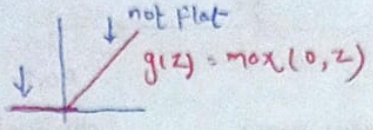
$a = g(z)$

How to choose activation function for output layer or hidden layer

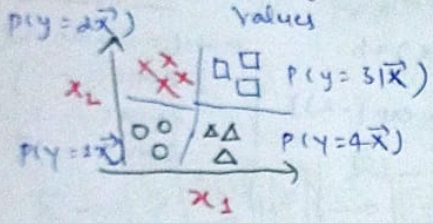
- Binary classification → {0, 1} problem ⇒ Sigmoid
- Regression problem
  for eg tomorrow stock price +/− ⇒ Linear
- for eg House prediction but it will only have positive value ⇒ ReLU

} output layer based on y prediction

- **Hidden Layer:** most common choice ReLU {default}
  - It is faster
  - It has sepreate region of flat or not
  rather than sigmoid
  ∴ It learns pretty fast.



- **Multi class classification**
  → target y can take more than just two 0 and 1 values



- **Softmax**

Softmax regression (4 possible output)
y = 1, 2, 3, 4

⊗ $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$ $\quad a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=1|\vec{x})$

○ $z_2 = \vec{w}_2 \cdot \vec{x} + b_2$ $\quad a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=2|\vec{x})$

□ $z_3 = \vec{w}_3 \cdot \vec{x} + b_3$ $\quad = P(y=3|\vec{x})$

△ $z_4 = \vec{w}_4 \cdot \vec{x} + b_4$ $\quad = P(y=4|\vec{x})$

Softmax regression (N possible output)

$z_j = \vec{w}_j \cdot \vec{x} + b_j \quad j = 1, 2 \cdots N$

$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$

Note: $a_1 + a_2 + \cdots a_N = 1$

Cost

Logistic regression

$z = \vec{w} \cdot \vec{x} + b$

$a_1 = g(z) = \frac{1}{1+e^{-z}} \Rightarrow P(y=1|\vec{x})$

$a_2 = 1 - a_1 = P(y=0|\vec{x})$

$loss = -y \log a_1 - (1-y) \log(1-a_1)$

$J(\vec{w}, b)$ = average loss

predict → f_x = tf.nn.sigmoid(logit)

Something similar like this

Softmax regression

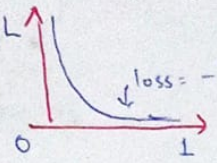
$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} \cdots e^{z_N}}$

$\vdots$

a_N

Crossentropy loss

$$loss(a_1 \cdots a_N) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & y=2 \\ \vdots & \end{cases}$$



* model.compile(loss = SparseCategoricalCrossentropy())

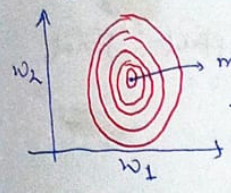
Rest is same except change in outer layer.

Adam Algorithm.

It increases the learning rate α when it sees that gradient descent taking very minute steps to reach minima

- It also decreases alpha when learning rate is too fast

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$



Adam: Adaptive Moment estimation

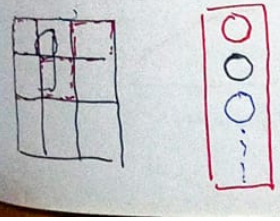
It adjust α automatically.

Code:

```
model.compile(optimizer = tf.keras.optimizers.Adam(
                learning_rate = 1e-3)
                ↘ α = 10^-3
```

Additional Layer Types

Convolutional Layer



Each neuron only look at part of the previous layer's output {each square in grid}

→ Faster computation

→ Need less training data.

How Back Propagation Work {Calculus Part}

- **Derivative** ?

Cost function $J(w) = w^2$

Say $w = 3$ $\quad J(w) = 3^2 = 9$

If we increase w by a tiny amount $\varepsilon = 0.001$ how does $J(w)$ changes?

$w = 3 + 0.001$

$J(w) = w^2 = 9.006001$

If $w \uparrow$ 0.001

$J(w) \uparrow \quad 6 \times 0.001 \quad 6 \times \varepsilon$

$\frac{\partial}{\partial w} J(w) = 6$

* If $w \uparrow \varepsilon$ cause $J(w) \uparrow k \times \varepsilon$ then

$\frac{\partial}{\partial w} J(w) = k$

gradient descent

repeat { $w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$

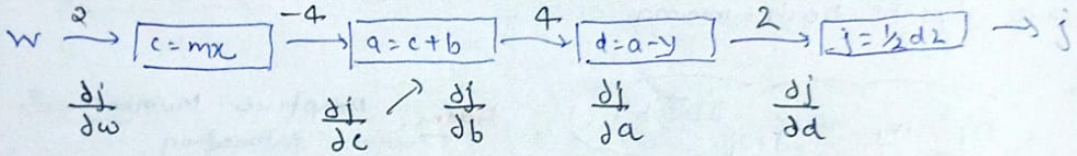
}

If derivative is small, then this update step will make a small update on w_j

If large → so large update

Q. Why we use backpropagation to calculate derivatives

* Backprop is an efficient way to compute derivatives



$\frac{\partial j}{\partial w}$ $\quad \frac{\partial j}{\partial c}$ $\quad \frac{\partial j}{\partial b}$ $\quad \frac{\partial j}{\partial a}$ $\quad \frac{\partial j}{\partial d}$

By drawing right to left we can see how these quantities are changing to effect final output j