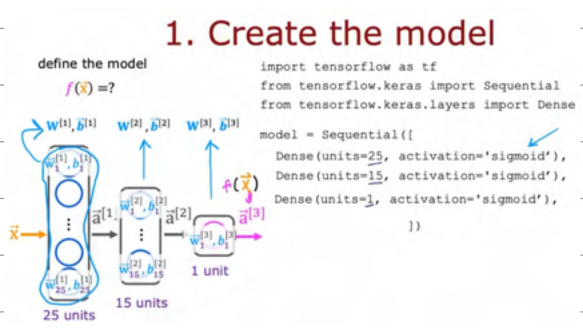
Week 2

o Steps to train a Neural Network model

1. Create the Model :-

This is done using the Keras library of tensorflow.


define the model
$f(\vec{x})$ =?
$\mathbf{w}^{[1]}, \vec{b}^{[1]}$
$\mathbf{w}^{[2]}, \vec{b}^{[2]}$
$\mathbf{w}^{[3]}, \vec{b}^{[3]}$
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_{25}^{[1]}, b_{25}^{[1]}$
$\vec{w}_1^{[2]}, b_1^{[2]}$
$\vec{w}_{15}^{[2]}, b_{15}^{[2]}$
$\vec{w}_1^{[3]}, b_1^{[3]}$
$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
$f(\vec{x})$
25 units
15 units
1 unit

1. Create the model

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(units=25, activation='sigmoid'),
    Dense(units=15, activation='sigmoid'),
    Dense(units=1, activation='sigmoid'),
                    ])
```

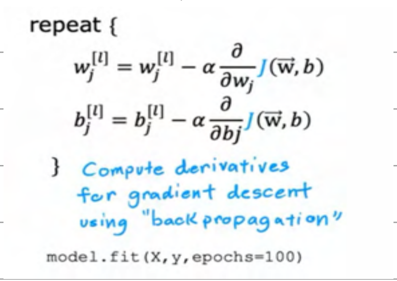
2. Loss and Cost Function :-

We use binary classification to reduce the average loss and we do this with the help of BinaryCrossentropy () which is aka logistic loss.

* model. compile (loss = BinaryCrossentropy())

3. Applying Gradient Decent :-

This step is same as logistic regression where we take small steps to reach the minimum value while reducing the parameters.



repeat {

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

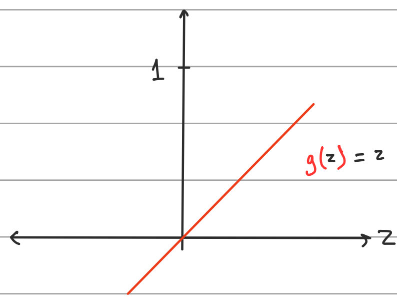
$$b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial bj} J(\vec{w}, b)$$

} Compute derivatives for gradient descent using "back propagation"

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

o Types of Activation Functions

1. Linear Activation (No activation)

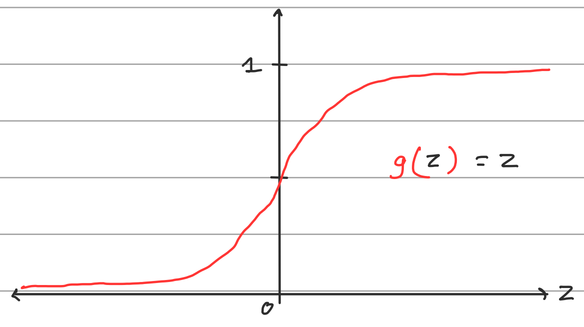


as

activation $a = g(z) = z$

hence no activation

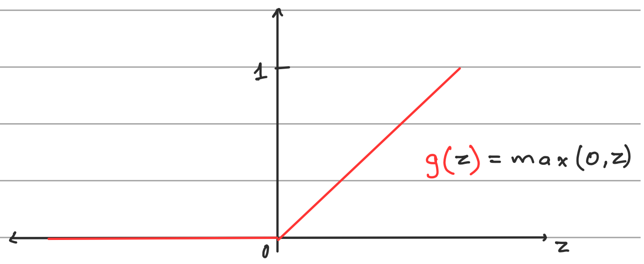
2. Sigmoid Function :-



$0 < g(z) < 1$

predicts a value between 0 & 1

3. Rectified Linear Unit (ReLU)



Outputs only a positive value.

o How to choose Activation Functions?

A. Output Layer :-

As this layer calculates the final prediction, we can use all the above mentioned activations depending upon the desired output.

⋆ For $y = 0/1$
use Binary Classifier aka Sigmoid

⋆ For $y = +/-$
use Regression aka Linear Function

⋆ For $y = 0$ or $+$
use Rectified Linear unit ReLU

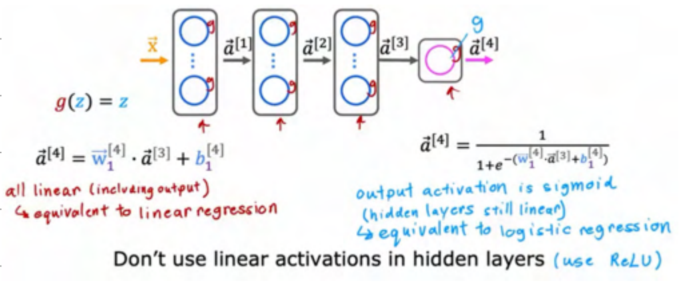
B. Hidden Layer :-

Our goal for hidden layers is to increase efficiency by making algorithm faster, hence it is recommended to use ReLU

- It has only 1 flat as compared to sigmoid's two.
- It is faster than sigmoid as the ReLU function is less computational.

o Why does Neural Networks require Activation?

It turns out that if we use Linear activation function or even the sigmoid function, the output of a Neural Network is same as the output of a Regression or a Classification Model.



$g(z) = z$

$\vec{a}^{[4]} = \vec{w}_1^{[4]} \cdot \vec{a}^{[3]} + b_1^{[4]}$

all linear (including output)
↳ equivalent to linear regression

$$\vec{a}^{[4]} = \frac{1}{1+e^{-(\vec{w}_1^{[4]}\cdot\vec{a}^{[3]}+b_1^{[4]})}}$$

output activation is sigmoid
(hidden layers still linear)
↳ equivalent to logistic regression

Don't use linear activations in hidden layers (use ReLU)

o Multiclass Classification

Just like logistic regression which was a binary classification algorithm, we have a way to classify N possible outputs.

o Softmax Regression

This a generalized logistic regression Here we take multiple z values to predict the class.

$$z_1 = \vec{w}_1 \cdot \vec{x} + b_1 = P(y=1|\vec{x})$$

$$z_2 = \vec{w}_2 \cdot \vec{x} + b_2 = P(y=2|\vec{x})$$

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

→ This model predicts if $y = 1, 2, 3$

Hence the general function for Softmax regression is :-

$$\boxed{z_j = \vec{w}_j \cdot \vec{x} + b_j \qquad j = 1 \ldots N}$$

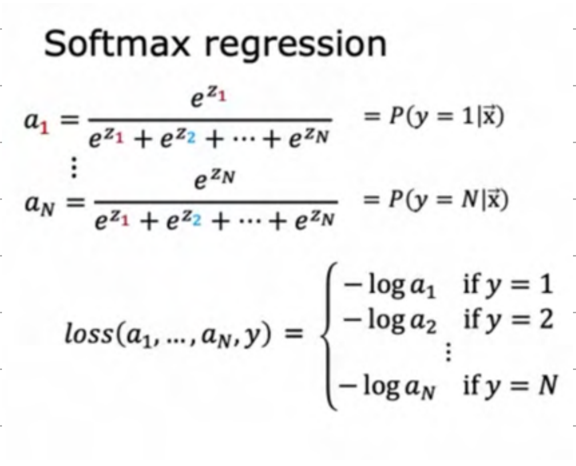
activation,

$$\boxed{a_j = \frac{e^{z_j}}{\sum_{K=1}^{N} e^{z_K}} = P(y = j|\vec{x})}$$

N = no. of possible outputs
j, K = basically equals N

☆ $a_1 + a_2 + \ldots\ldots + a_N = 1$

○ Cost Function for Softmax



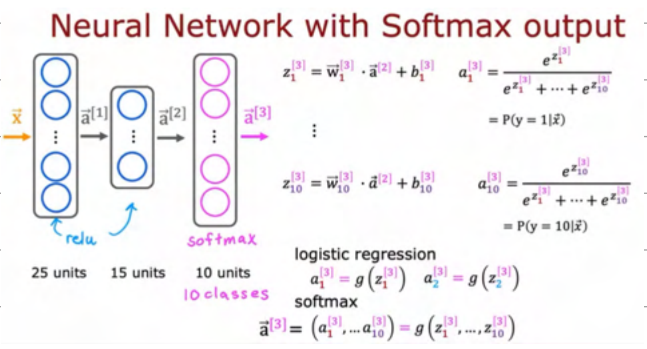
Softmax regression

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} = P(y = 1|\vec{x})$$
$$\vdots$$
$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} = P(y = N|\vec{x})$$

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

○ Neural Network with Softmax Output.

- In order to perform Multiclass classification, we follow the exact same steps but there's a catch. In the output layer we add Softmax which classifies our result.



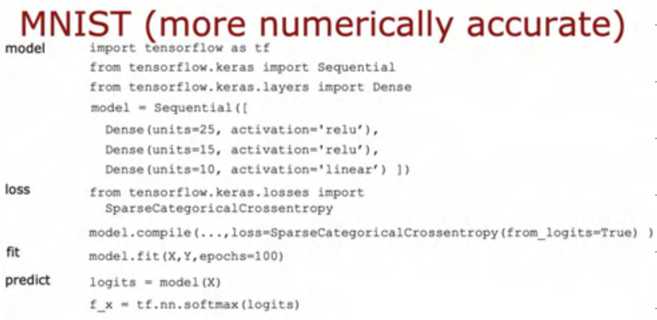
$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \qquad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y = 1|\vec{x})$$
$$\vdots$$
$$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]} \qquad a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y = 10|\vec{x})$$

logistic regression
$$a_1^{[3]} = g(z_1^{[3]}) \qquad a_2^{[3]} = g(z_2^{[3]})$$
softmax
$$\vec{a}^{[3]} = (a_1^{[3]}, \ldots a_{10}^{[3]}) = g(z_1^{[3]}, \ldots, z_{10}^{[3]})$$

- We can later select the class which has the highest probability.

○ Implementation of Softmax

- In the implementation you'll notice that we've not using softmax in the output layer
- It's due to a change in the type of loss function which improves accuracy of the model



MNIST (more numerically accurate)

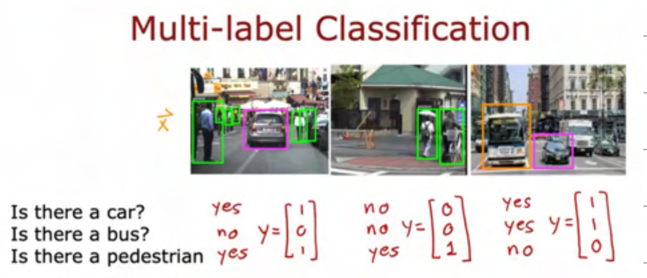
```
model     import tensorflow as tf
          from tensorflow.keras import Sequential
          from tensorflow.keras.layers import Dense
          model = Sequential([
            Dense(units=25, activation='relu'),
            Dense(units=15, activation='relu'),
            Dense(units=10, activation='linear') ])
loss      from tensorflow.keras.losses import
            SparseCategoricalCrossentropy
          model.compile(...,loss=SparseCategoricalCrossentropy(from_logits=True) )
fit       model.fit(X,Y,epochs=100)
predict   logits = model(X)
          f_x = tf.nn.softmax(logits)
```

○ Multi-Label Classification

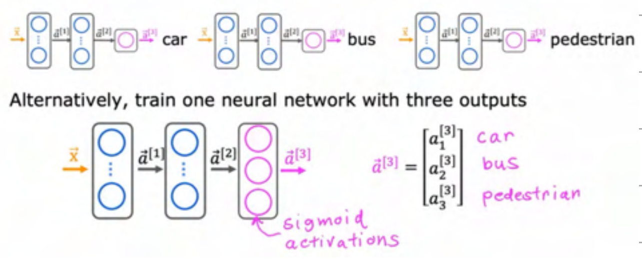
- In Multiclass classification :-
  we had multiple possible outputs
  But only 1 actual output

- But in Multi-Label we have multiple actual output out of multiple possible outputs

- We can easily understand it with this example :-


Multi-label Classification
$\vec{x}$
Is there a car?
Is there a bus?
Is there a pedestrian
yes
no
yes
$y=\begin{bmatrix}1\\0\\1\end{bmatrix}$
no
no
yes
$y=\begin{bmatrix}0\\0\\1\end{bmatrix}$
yes
yes
no
$y=\begin{bmatrix}1\\1\\0\end{bmatrix}$

- This is how it works under the hood.


$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$a^{[3]}$
car
bus
pedestrian
Alternatively, train one neural network with three outputs
$\vec{a}^{[3]}$
$\vec{a}^{[3]}=\begin{bmatrix}a_1^{[3]}\\a_2^{[3]}\\a_3^{[3]}\end{bmatrix}$
car
bus
pedestrian
sigmoid activations

○ Optimised Gradient Descent

- Efficiency of gradient Descent completely depends upon 'α' the Learning rate.
- Is there a way to take such a value of α that the step is not too big neither too small?

○ Adaptive Moment Estimation

$\underbrace{\text{Adaptive}}_{\text{Ad}}\ \underbrace{\text{Moment}}_{\text{am}}$ Estimation

Adam Algorithm

- Here, instead of One α for all the parameters, we take different learning rates for each & every w & b

$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$

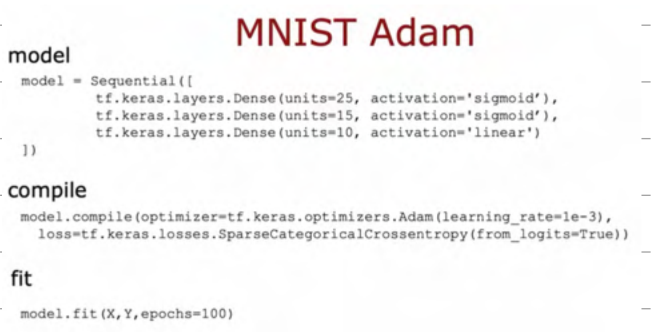
$$w_2 = w_2 - \alpha_2 \frac{\partial}{\partial w_2} J(\vec{w}, b)$$

$$\vdots$$

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

$$b = b - \alpha_{j+1} \frac{\partial}{\partial b} J(\vec{w}, b)$$

- This is how its implemented in code using TensorFlow:-



MNIST Adam

model

```
model = Sequential([
        tf.keras.layers.Dense(units=25, activation='sigmoid'),
        tf.keras.layers.Dense(units=15, activation='sigmoid'),
        tf.keras.layers.Dense(units=10, activation='linear')
])
```

compile

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

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

○ Convolutional Layer

- It is an optimised approach for Neural Networks.
- Here each neuron looks only at a particular part of the previous layers output.

★ How does it help?

1. Speeds up computation
2. Requires less training data
3. less prone to overfitting

○ How does derivatives help in Gradient descent.

- Our goal is to minimize the value of w and derivatives helps us in doing that.

- Small change in w, makes a K times small change in $J(w)$

as $w \downarrow \varepsilon$ $\quad J(w) \downarrow K \cdot \varepsilon$

○ Computational Graph

- It helps frameworks like tensorflow to compute derivatives.
- For normal calculations we use forward prop whereas for calculus usually backward prop is used.

★ Lets see how it works

Forward prop

Backward prop

$J = \frac{1}{2}\left[f(w,b) - y\right]^2$

as $f(w,b) = wx + b$

we start with wx

then we add $wx + b$

To this sum we reduce $(wx+b) - y$

Then we compute the square & divide it by 2

That's how we get J

$$\begin{tikzcd}
w \arrow[d] \arrow[d, bend right=60] \\
c = wx \arrow[d] \arrow[d, bend right=60] \arrow[u, bend right=60] \\
a = c + b \arrow[d] \arrow[d, bend right=60] \arrow[u, bend right=60] \\
d = a - y \arrow[d] \arrow[d, bend right=60] \arrow[u, bend right=60] \\
J = \frac{1}{2} d^2 \arrow[d] \arrow[d, bend right=60] \arrow[u, bend right=60] \\
J \arrow[u, bend right=60]
\end{tikzcd}$$

$\frac{\partial J}{\partial w}$

$\frac{\partial J}{\partial c}$

$\frac{\partial J}{\partial a}$

$\frac{\partial J}{\partial d}$

- If there are N nodes & P parameters the the computation takes roughly

$N + P$ steps rather than $N \times P$

- Backward prop is efficient as it reduces the number of computations.