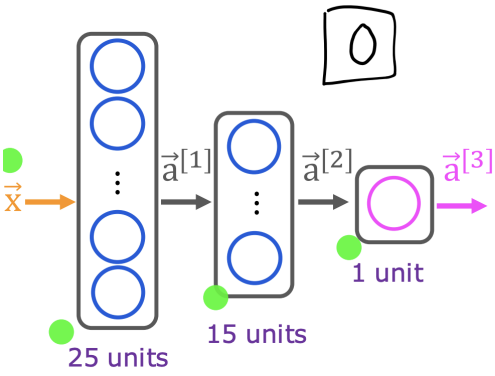
Neural Network

Train a Neural Network in TensorFlow




```
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')
                          )]
from tensorflow.keras.losses import
BinaryCrossentropy
   model.compile(loss=BinaryCrossentropy())
   model.fit(X,Y,epochs=100)
```

① ② ③

epochs: number of steps in gradient descent

Given set of (x,y) examples

How to build and train this in code?

Training details in TensorFlow

Model Training Steps TensorFlow

① specify how to compute output given input x and parameters w,b (define model)

$f_{\vec{w},b}(\vec{x})$ =?

② specify loss and cost

$L\left(f_{\vec{w},b}(\vec{x}), y\right)$ 1 example

$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)$

③ Train on data to minimize $J(\vec{w}, b)$

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)
```

```
w = w - alpha * dj_dw
b = b - alpha * dj_db
```

neural network

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

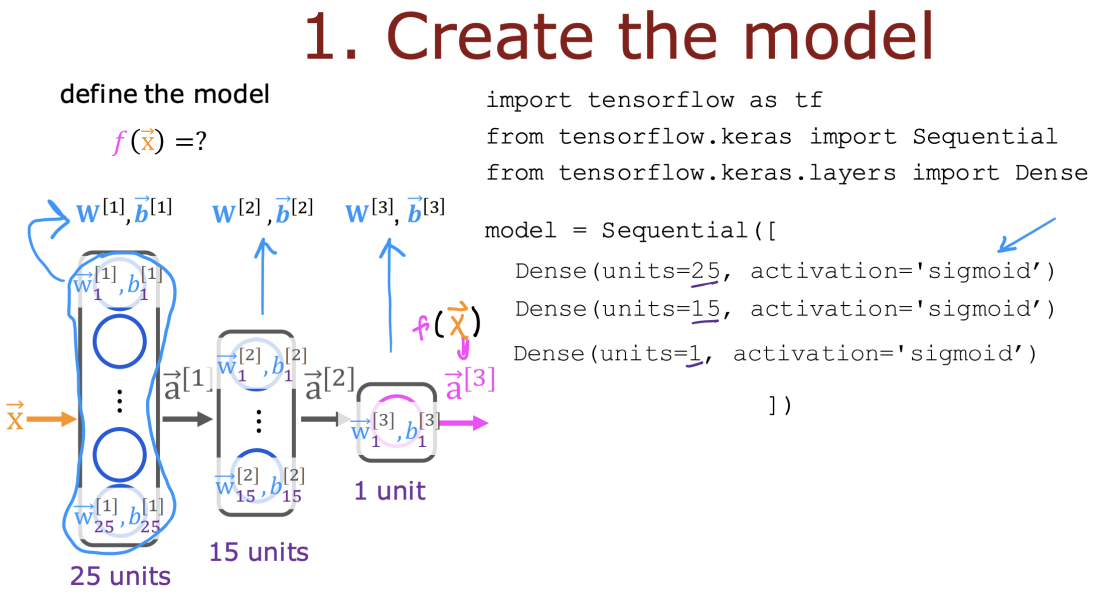
binary cross entropy

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

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

Step 1: Specifying the Model Architecture

In TensorFlow, specifying the model architecture involves defining the layers, their connections, and activation functions. This step is crucial as it outlines the structure through which data will flow and be transformed.

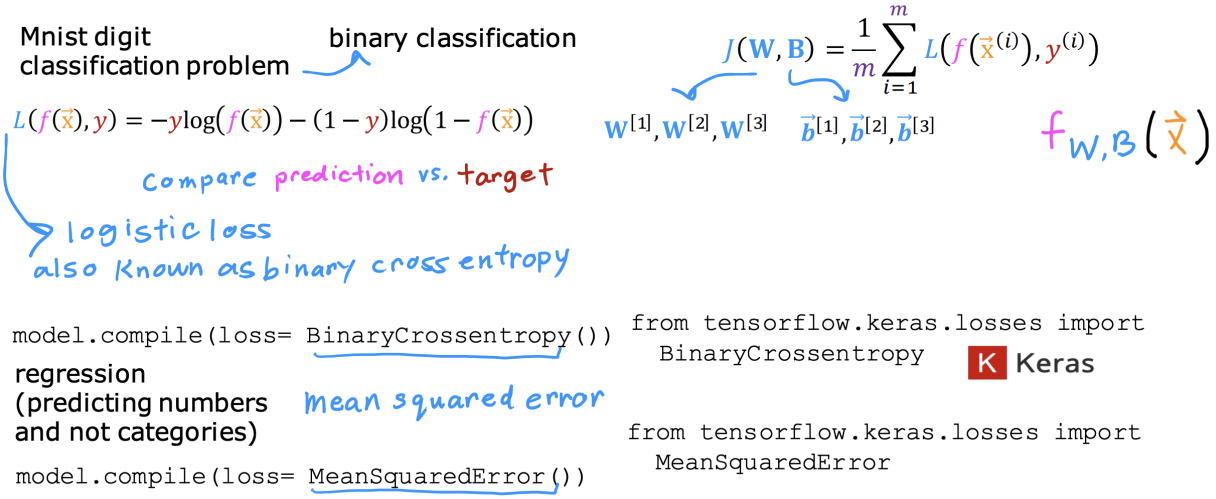


This code snippet effectively sets up the forward propagation path, determining how the input (x) is processed to produce the output (f(x)) through the network layers and their parameters ((W) and (b)).

Step 2: Compiling the Model with a Loss Function

Compiling the model involves specifying the loss function, optimiser, and metrics for evaluation. The loss function quantifies the difference between the predicted outputs and the actual labels, guiding the training process by indicating how well the model is performing.

2. Loss and cost functions

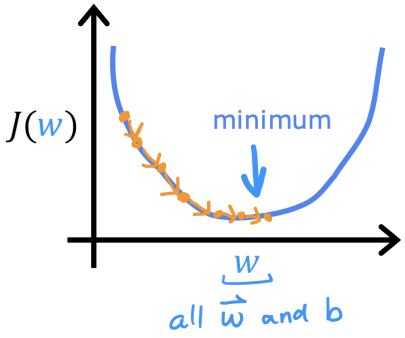


This step also implicitly defines the cost function, which is the average of the loss function over all training examples, providing a single value that summarises the model's performance on the entire training set.

Step 3: Training the Model

Training the model with TensorFlow's `fit` method involves adjusting the model's parameters (\(W) and (b)) to minimise the cost function. This process uses back-propagation and an optimisation algorithm (like gradient descent or its variants) to compute gradients and update the parameters.

3. Gradient descent



repeat {

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

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

} Compute derivatives for gradient descent using "back propagation"

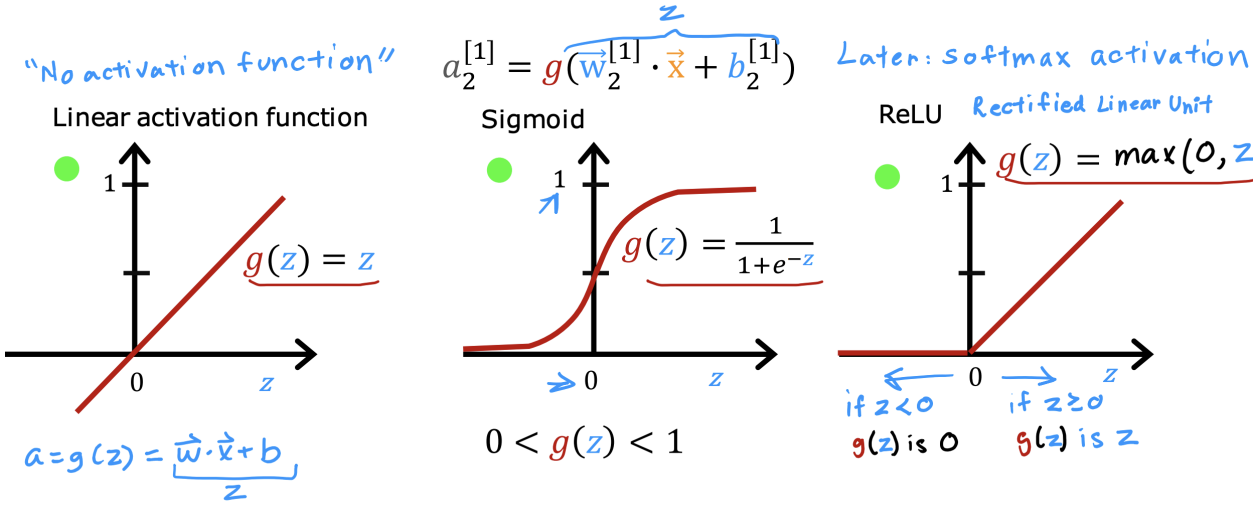
```
model.fit(X,y,epochs=100)
```

This step encapsulates the iterative process of forward propagation (to compute the loss), back propagation (to compute gradients), and parameter updates (to minimise the cost).

Activation Functions

Activation functions determine the output of a neural network node given an input or set of inputs. They introduce non-linearity into the network, enabling it to learn complex patterns.

Examples of Activation Functions



▼ Sigmoid Activation Function

The sigmoid function is defined as (g(z) = {1}/{1 + e^{-z}}). It outputs a value between 0 and 1, making it particularly useful for binary classification problems, such as predicting whether an image is a cat or not. However, it has fallen out of favor for hidden layers due to issues like vanishing gradients, where gradients become very small, impeding the network's learning.

▼ ReLU (Rectified Linear Unit)

Defined as (g(z) = max(0, z)), ReLU has become the default activation function for many types of neural networks. ReLU is linear (identity) for all positive values, and zero for all negative values. This simplicity leads to faster computation and alleviates the vanishing gradient problem, allowing models to learn faster and perform better. However, ReLU units can be fragile during training and can "die," meaning they stop outputting anything other than 0 if a large gradient flows through them.

▼ Linear Activation Function

A linear activation function is simply (g(z) = z). It implies that the output is proportional to the input. It's often used in the output layer for regression problems, where the goal is to predict a continuous value. However, using linear activation functions in hidden layers makes the neural network

essentially a linear model, which limits its ability to capture complex patterns in the data.

Choosing activation functions

Output Layer Activation Functions

- **Binary Classification Problems**: For tasks where the goal is to predict one of two possible outcomes (e.g., spam or not spam), the **sigmoid** activation function is the natural choice. It outputs a probability score between 0 and 1, which can be interpreted as the likelihood of the input belonging to the positive class.
- **Regression Problems**: When predicting continuous values (e.g., stock prices, temperatures), the **linear** activation function is appropriate. It allows the model to output a range of values, both positive and negative, without restriction.
- **Non-negative Output Problems**: For tasks where the output is always non-negative (e.g., predicting prices, quantities), the **ReLU** (Rectified Linear Unit) activation function is suitable. It ensures that the model's predictions are either zero or positive.

Hidden Layer Activation Functions

- **ReLU** has become the default choice for most applications due to its computational efficiency and its ability to mitigate the vanishing gradient problem, which can hinder learning in deep networks. ReLU is defined as (g(z) = max(0, z)), meaning it outputs zero for any negative input and outputs the input itself for any positive input.

Implementation in TensorFlow

Here's how you can implement these choices in TensorFlow:

```
from tensorflow.keras.layers import Dense

# For hidden layers, use ReLU
model.add(Dense(64, activation='relu'))
```

```
model.add(Dense(64, activation='relu'))

# For the output layer, choose based on the problem type
# Binary classification
model.add(Dense(1, activation='sigmoid'))

# Regression
model.add(Dense(1, activation='linear'))

# Non-negative outputs
model.add(Dense(1, activation='relu'))
```

Why Activation Functions Are Necessary

Activation functions introduce non-linearity into the network, enabling it to learn complex patterns. Without activation functions, a neural network, regardless of its depth, would behave just like a single-layer linear model, severely limiting its ability to capture and model complex relationships in the data.