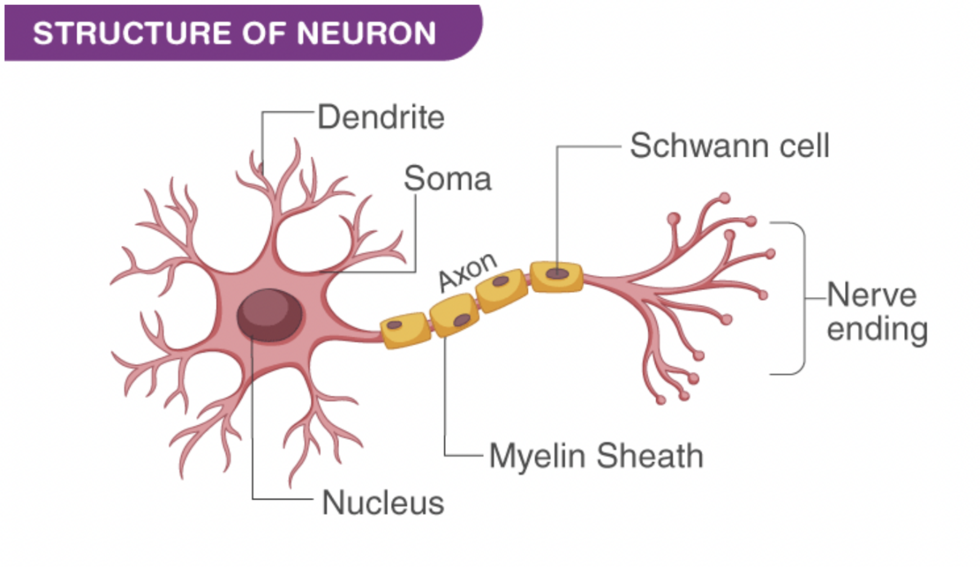
Neural Networks

1. What are Neurons?

Neurons are the building blocks of the nervous system. They receive and transmit signals to different parts of the body. This is carried out in both physical and electrical forms. There are several different types of neurons that facilitate the transmission of information.

In [1]:
```
from IPython.display import Image
Image("image.png",width="500")
```

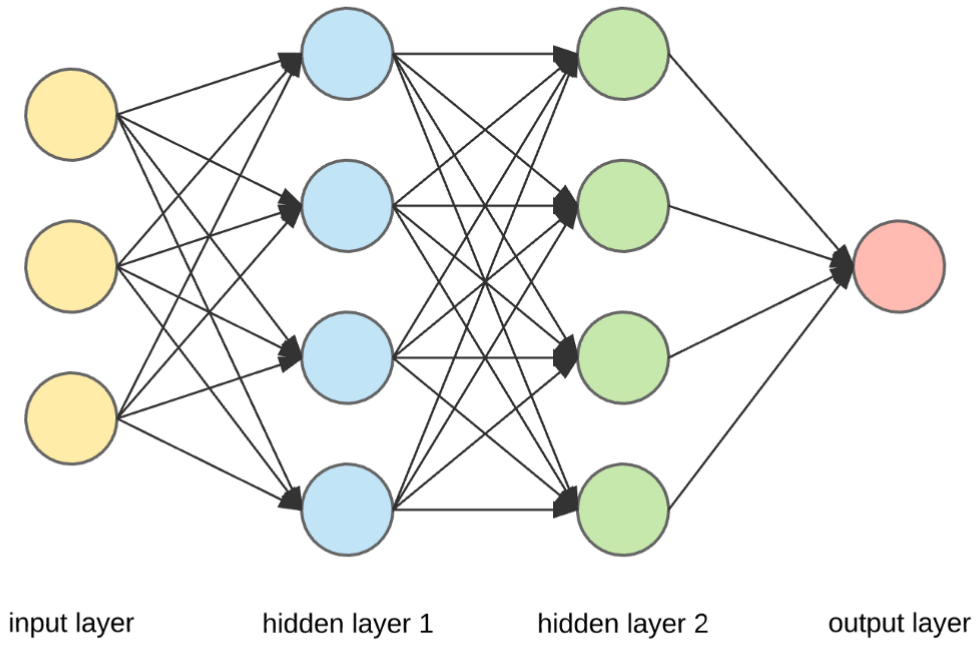
Out[1]:



2. Explain the architecture of a neural network

In [2]:
```
Image("1.png",width="500")
```

Out[2]:



A neuron is the basic unit of a neural network.

The outermost yellow layer is the input layer.

They receive input from an external source or other nodes. Each node is connected with another node from the next layer, and each such connection has a particular weight. Weights are assigned to a neuron based on its relative importance against other inputs.

When all the node values from the yellow layer are multiplied (along with their weight) and summarized, it generates a value for the first hidden layer. Based on the summarized value, the blue layer has a predefined “activation” function that determines whether or not this node will be “activated” and how “active” it will be.

The layer or layers hidden between the input and output layer is known as the hidden layer. It is called the hidden layer since it is always hidden from the external world. The main computation of a Neural Network takes place in the hidden layers. So, the hidden layer takes all the inputs from the input layer and performs the necessary calculation to generate a result.

This result is then forwarded to the output layer so that the user can view the result of the computation.

3. How many parameters are there in a neural network with 4 hidden layers having 30,25,20,15 neurons each and the input and output layer having 50 and 1 neuron respectively

Input = 50
Layer1 = 30
Layer2 = 25
Layer3 = 20

Layer4 = 15

Output = 1

Parameter calculation:

For the weights: 50 x 30 + 30 x 25 + 25 x 20 + 20 x 15 + 15 x 1 = 3065

For the bias components: 30 + 25 + 20 + 15 + 1 = 91

The amount of parameters in this neural network is 3156.

In [41]:
```
def total_param(l=[]):
    s=0
    for i in range(len(l)-1):
        s=s+l[i]*l[i+1]+l[i+1]
    return s
```

In [42]:
```
total_param([50,30,25,20,15,1])
```

Out[42]:
```
3156
```

4. Explain gradient descent

Gradient Descent is a process that occurs in the backpropagation phase where the goal is to continuously resample the gradient of the model's parameter in the opposite direction based on the weight w, updating consistently until we reach the global minimum of function J(w).

In [3]:
```
Image("2.png",width="500")
```

Out[3]:

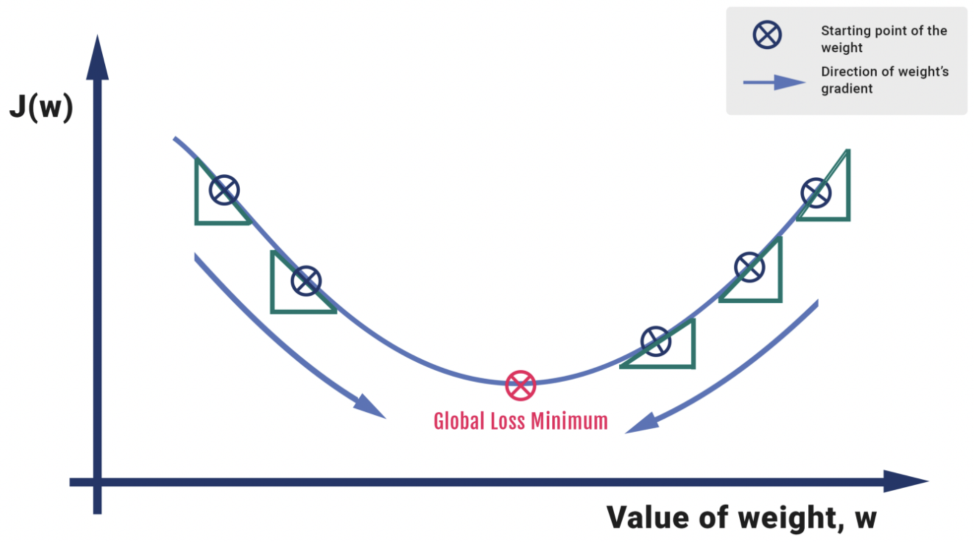
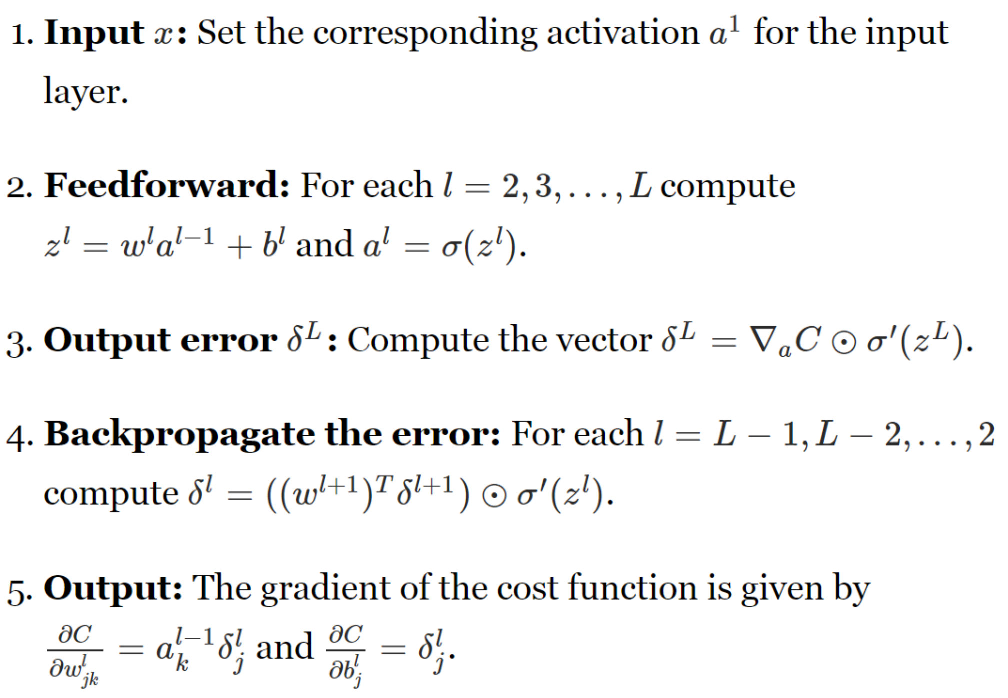


Fig 1: How Gradient Descent works for one parameter, w

The gradient descent process is exhibited in the form of the backpropagation step where we compute the error vectors δ backward, starting from the final layer. Depending upon the activation function, we identify how much change is required by much change is required by taking the partial derivative of the function with respect to w. The change value gets multiplied by the learning rate. As part of the output, we subtract this value from the previous output to get the updated value. We continue this till we reach convergence.

In [4]:
```
Image("3.png",width="500")
```

Out[4]:


1. **Input x:** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

There are a number of gradient descent algorithms: Batch Gradient Descent Stochastic Gradient Descent Mini-batch Gradient Descent

Exploding & Vanishing Gradients In deep networks or recurrent neural networks, there are two known issues explained in a paper by Pascanu et al (1994) — of exploding and vanishing gradients. This happens when are doing back propagation as we iterate through the code, there is a chance that the normal of the weight matrices going beyond 1. If this happens, the gradient explodes but if the normal is below 1, the gradient vanishes.

If we want to visualize exploding gradients, you will encounter at least one of the problems:

1. The model will output 'Nan' values
2. The model will display very large changes upon each step
3. The error gradient values are consistently above 1.0 for each node in the training layer.

Solution: Gradient Clipping

The solution to the exploding and vanishing gradient problem, we introduce gradient clipping, where we 'clip' the gradients if it goes over a certain threshold represented by a maximum absolute value. Hence, we keep the neural network stable as the weight values never reach the point that it returns a 'Nan'. In a coded implementation removing the clipped gradients leads to 'Nan' values or infinite in the losses and fails to run further.

5. What is categorical cross entropy

It is the default loss function to use for multi-class classification problems where each class is assigned a unique integer value from 0 to (num_classes – 1).

It will calculate the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0.

In [5]:

```
Image("4.png",width="300")
```

Out[5]:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\\text{size}}} y_i \cdot \log \hat{y}_i$$

The target need to be one-hot encoded this makes them directly appropriate to use with the categorical cross-entropy loss function.

The output layer is configured with n nodes (one for each class), for example in MNIST case, 10 nodes, and a “softmax” activation in order to predict the probability for each class.

6. Explain in short the following terms: Relu, Sigmoid, softmax

ReLU

It stands for Rectified Linear Units.
The formula: $max(0, z)$.
Despite its name and appearance, it’s not linear and provides the same benefits as Sigmoid (i.e. the ability to learn nonlinear functions), but with better performance.

Pros :

1. It avoids and rectifies vanishing gradient problem.
2. ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Cons :

1. One of its limitations is that it should only be used within hidden layers of a neural network model.
2. Some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. In other words, ReLu can result in dead neurons.
3. In another words, For activations in the region (x<0) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which

go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called the dying ReLu problem.

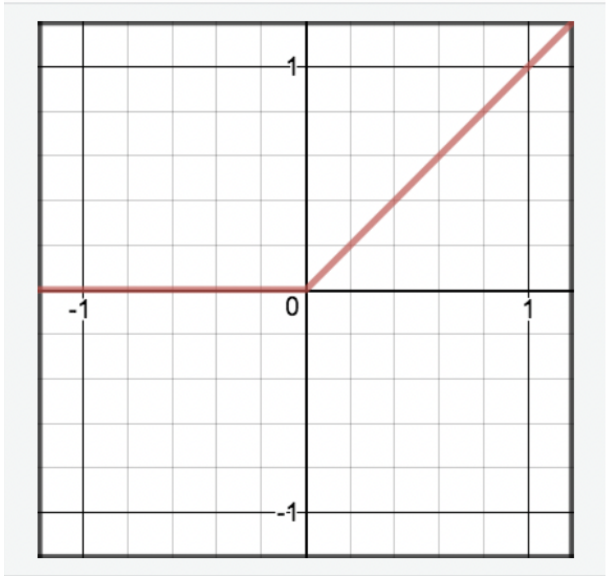
4. The range of ReLu is [0,∞). This means it can blow up the activation.

In [7]:
```
Image("5.png",width="300")
```

Out[7]:

Function

$$R(z) = \begin{Bmatrix} z & z > 0 \\ 0 & z <= 0 \end{Bmatrix}$$



Sigmoid

Sigmoid takes a real value as input and outputs another value between 0 and 1. It's easy to work with and has all the nice properties of activation functions: it's non-linear, continuously differentiable, monotonic, and has a fixed output range.

Pros :

1. It is nonlinear in nature. Combinations of this function are also nonlinear!
2. It will give an analog activation unlike step function.
3. It has a smooth gradient too.
4. It's good for a classifier.
5. The output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range. Nice, it won't blow up the activations then.

Cons :

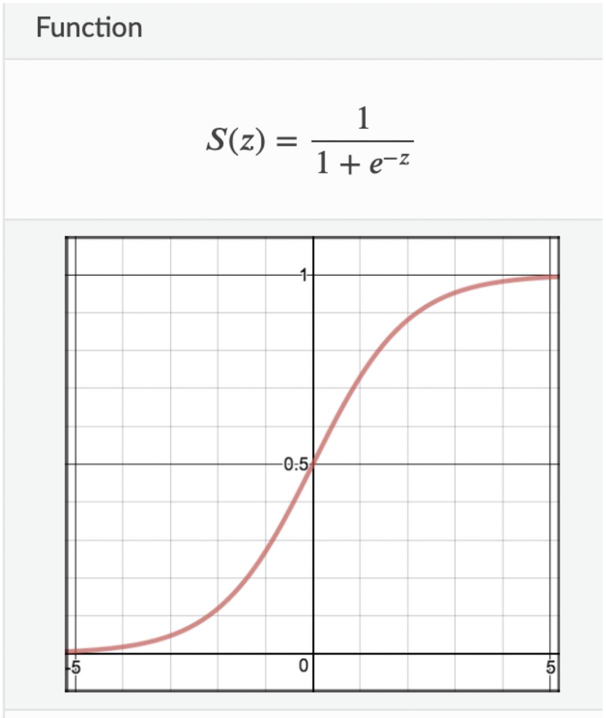
1. Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.
2. It gives rise to a problem of "vanishing gradients".

3. Its output isn't zero centered. It makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder.
4. Sigmoids saturate and kill gradients.
5. The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits).

In [8]:

```
Image("6.png",width="300")
```

Out[8]:



Softmax

Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

Pros:

The main advantage of using Softmax is the output probabilities range. The range will 0 to 1, and the sum of all the probabilities will be equal to one. If the softmax function used for multi-classification model it returns the probabilities of each class and the target class will have the high probability.

7. Load the mnist dataset

In [46]:

```
# Importing required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
```

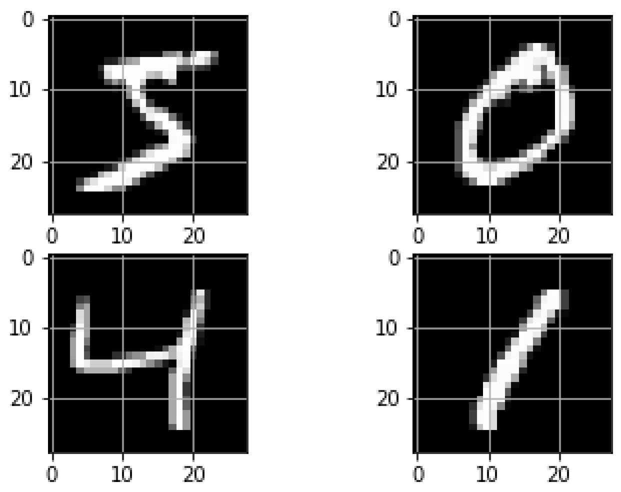
In [47]:
```
import keras
import tensorflow as tf
```

In [48]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.datasets import mnist
```

In [49]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

8. plot some sample images

In [50]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.grid('off')
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.grid('off')
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
plt.grid('off')
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
plt.grid('off')
plt.show()
```



9. pre process the target variable to make it binary

In [51]:
```
#preprocessing data
num_classes = 10
```

```
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

10. construct a neural network

In [52]:
```
image_size = 784
num_classes = 10
nn_model = Sequential()
```

In [53]:
```
nn_model.add(Flatten(input_shape=(28,28)))
nn_model.add(Dense(units=10, activation='sigmoid'))
nn_model.add(Dense(units=num_classes, activation='softmax'))
nn_model.compile(optimizer="adam", loss='categorical_crossentropy', metrics=['ac
```

11. plot the network (visualise/summarise)

In [54]:
```
nn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_2 (Dense)             (None, 10)                7850

 dense_3 (Dense)             (None, 10)                110

=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

12. train the network

In [55]:
```
nn_model.fit(X_train, y_train, epochs=5, batch_size=128, validation_split=.15, v
```

Out[55]:
```
<keras.callbacks.History at 0x7fee75f7f6d0>
```

In [56]:
```
pred = nn_model.predict(X_test)
```

In [57]:
```
loss, accuracy  = nn_model.evaluate(X_test, y_test, verbose=False)
print('loss: ', loss)
print('accuracy: ', accuracy)
```

```
loss:  0.5618279576301575
accuracy:  0.8510000109672546
```

13. plot epoch vs accuracy curve

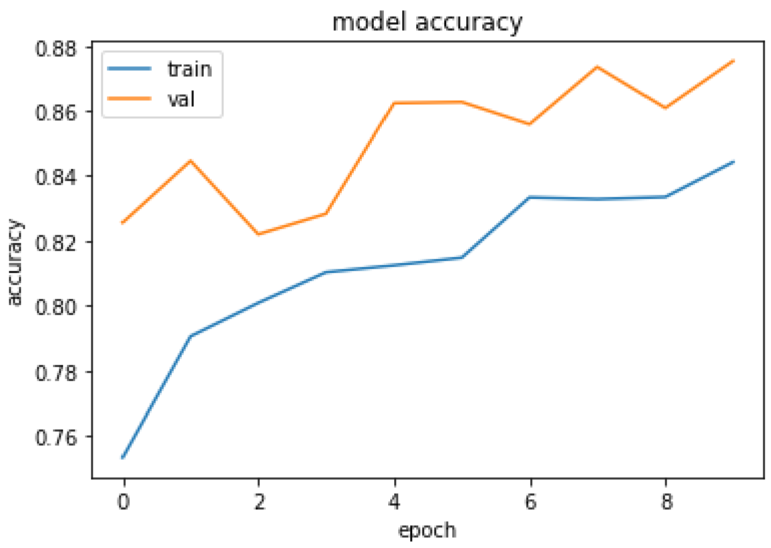
In [58]:

```
import keras
from matplotlib import pyplot as plt
history = nn_model.fit(X_train, y_train,validation_split = 0.1, epochs=10, batch
```

```
Epoch 1/10
13500/13500 [==============================] - 37s 3ms/step - loss: 0.7662 - acc
uracy: 0.7533 - val_loss: 0.5587 - val_accuracy: 0.8257
Epoch 2/10
13500/13500 [==============================] - 38s 3ms/step - loss: 0.6798 - acc
uracy: 0.7905 - val_loss: 0.5427 - val_accuracy: 0.8447
Epoch 3/10
13500/13500 [==============================] - 38s 3ms/step - loss: 0.6366 - acc
uracy: 0.8008 - val_loss: 0.5476 - val_accuracy: 0.8220
Epoch 4/10
13500/13500 [==============================] - 37s 3ms/step - loss: 0.6050 - acc
uracy: 0.8103 - val_loss: 0.5251 - val_accuracy: 0.8283
Epoch 5/10
13500/13500 [==============================] - 38s 3ms/step - loss: 0.6041 - acc
uracy: 0.8124 - val_loss: 0.4776 - val_accuracy: 0.8623
Epoch 6/10
13500/13500 [==============================] - 36s 3ms/step - loss: 0.6016 - acc
uracy: 0.8148 - val_loss: 0.4437 - val_accuracy: 0.8627
Epoch 7/10
13500/13500 [==============================] - 37s 3ms/step - loss: 0.5565 - acc
uracy: 0.8334 - val_loss: 0.4967 - val_accuracy: 0.8558
Epoch 8/10
13500/13500 [==============================] - 36s 3ms/step - loss: 0.5483 - acc
uracy: 0.8329 - val_loss: 0.4300 - val_accuracy: 0.8735
Epoch 9/10
13500/13500 [==============================] - 39s 3ms/step - loss: 0.5467 - acc
uracy: 0.8335 - val_loss: 0.4395 - val_accuracy: 0.8608
Epoch 10/10
13500/13500 [==============================] - 38s 3ms/step - loss: 0.5240 - acc
uracy: 0.8443 - val_loss: 0.4245 - val_accuracy: 0.8753
```

In [59]:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



14. print the results

In [60]:

```
y_test_arg=np.argmax(y_test,axis=1)
Y_pred = np.argmax(nn_model.predict(X_test),axis=1)
print('Confusion Matrix')
print(confusion_matrix(y_test_arg, Y_pred))
```

```
Confusion Matrix
[[ 916    0    4   10    1   23   20    2    4    0]
 [    0 1113    5    6    0    0    3    1    6    1]
 [   21    4  892   33   15    2   19   22   22    2]
 [    2   16   49  849    1   40    1   24   19    9]
 [    2    7    5    0  840    1   11    2   11  103]
 [   19    2    9  102   19  638   15   16   58   14]
 [   24    8   16    1   19   11  872    1    6    0]
 [    5   20   23   15    8    1    1  902    9   44]
 [    8   36   21   63   30   67   12   23  689   25]
 [    7    3    2   15   65    7    3   23    7  877]]
```

References:

https://towardsdatascience.com/gradient-descent-3a7db7520711

https://androidkt.com/choose-cross-entropy-loss-function-in-keras/

https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

https://medium.com/yottabytes/everything-you-need-to-know-about-gradient-descent-applied-to-neural-networks-d70f85e0cc14