الغاية من الجلسة :

بناء شبكة عصبونية باستخدام Tensorflow and Keras

Building your First Neural Network on a Structured Dataset (using Keras)

Before starting, I would like to give an overview of how to structure any deep learning project

1-**Preprocess and load data**- As we have already discussed data is the key for the working of neural network and we need to process it before feeding to the neural network. In this step, we will also visualize data which will help us to gain insight into the data.

2-**Define model**- Now we need a neural network model. This means we need to specify the number of hidden layers in the neural network and their size, the input and output size.

3-**Loss and optimizer**- Now we need to define the loss function according to our task. We also need to specify the optimizer to use with learning rate and other hyperparameters of the optimizer.

4-**Fit model**- This is the training step of the neural network. Here we need to define the number of epochs for which we need to train the neural network

After fitting model, we can test it on test data to check whether the case of overfitting. We can save the weights of the model and use it later whenever required.

Data processing We will use simple data of mobile price range classifier. The dataset consists of 20 features and we need to predict the price range in which phone lies. These ranges are divided into 4 classes. The features of our dataset include

'battery_power', 'blue', 'clock_speed', 'dual_sim', 'fc', 'four_g', 'int_memory', 'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_height','px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'three_g','touch_screen', 'wifi'

```
#Dependencies
import numpy as np
import pandas as pd
#dataset import
dataset = pd.read_csv("train.csv") #You need to change #directory accordingly
dataset.head(10) #Return 10 rows of data
```

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	...	px_height	px_width	ram	sc_h	sc_w	talk_time	th
0	842	0	2.2	0	1	0	7	0.6	188	2	...	20	756	2549	9	7	19	
1	1021	1	0.5	1	0	1	53	0.7	136	3	...	905	1988	2631	17	3	7	
2	563	1	0.5	1	2	1	41	0.9	145	5	...	1263	1716	2603	11	2	9	
3	615	1	2.5	0	0	0	10	0.8	131	6	...	1216	1786	2769	16	8	11	
4	1821	1	1.2	0	13	1	44	0.6	141	2	...	1208	1212	1411	8	2	15	
5	1859	0	0.5	1	3	0	22	0.7	164	1	...	1004	1654	1067	17	1	10	
6	1821	0	1.7	0	4	1	10	0.8	139	8	...	381	1018	3220	13	8	18	
7	1954	0	0.5	1	0	0	24	0.8	187	4	...	512	1149	700	16	3	5	
8	1445	1	0.5	0	0	0	53	0.7	174	7	...	386	836	1099	17	1	20	
9	509	1	0.6	1	2	1	9	0.1	93	5	...	1137	1224	513	19	10	12	

10 rows × 21 columns

```
#Changing pandas dataframe to numpy array
X = dataset.iloc[:,:20].values
y = dataset.iloc[:,20:21].values
```

This code as discussed in python module will make two arrays X and y. X contains features and y will contain classes.

```
#Normalizing the data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
```

This step is used to normalize the data. Normalization is a technique used to change the values of an array to a common scale, without distorting differences in the ranges of values. It is an important step and you can check the difference in accuracies on our dataset by removing this step. It is mainly required in case the dataset features vary a lot as in our case the value of battery power is in the 1000's and clock speed is less than 3. So if we feed unnormalized data to the neural network, the gradients will change differently for every column and thus the learning will oscillate.

The X will now be changed to this form:

```
Normalized data:
[-0.90259726 -0.9900495   0.83077942 -1.01918398 -0.76249466 -1.04396559
 -1.38064353  0.34073951  1.34924881 -1.10197128 -1.3057501  -1.40894856
 -1.14678403  0.39170341 -0.78498329  0.2831028   1.46249332 -1.78686097
 -1.00601811  0.98609664]
```

```
  File "<ipython-input-8-fd11b8c5c399>", line 1
    Normalized data:
                   ^
SyntaxError: invalid syntax
```

Next step is to one hot encode the classes. One hot encoding is a process to convert integer classes into binary values. Consider an example, let's say there are 3 classes in our dataset namely 1,2 and 3. Now we cannot directly feed this to neural network so we convert it in the form:

1- 1 0 0

2- 0 1 0

3- 0 0 1

Now there is one unique binary value for the class. The new array formed will be of shape (n, number of classes), where n is the number of samples in our dataset. We can do this using simple function by sklearn:

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
y = ohe.fit_transform(y).toarray()
```

Our dataset has 4 classes so our new label array will look like this:

One hot encoded array: [[0. 1. 0. 0.] [0. 0. 1. 0.] [0. 0. 1. 0.] [0. 0. 1. 0.] [0. 1. 0. 0.]]

Generally, it is better to split data into training and testing data. Training data is the data on which we will train our neural network. Test data is used to check our trained neural network. This data is totally new for our neural network and if the neural network performs well on this dataset,

it shows that there is no overfitting

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.1)
```

This will split our dataset into training and testing. Training data will have 90% samples and test data will have 10% samples. This is specified by the test_size argument.

Building Neural Network

Keras is a simple tool for constructing a neural network. It is a high-level framework based on tensorflow, theano or cntk backends.

In our dataset, the input is of 20 values and output is of 4 values. So the input and output layer is of 20 and 4 dimensions respectively.

```
#Dependencies
import keras
from keras.models import Sequential
from keras.layers import Dense
# Neural network
model = Sequential()
model.add(Dense(16, input_dim=20, activation="relu"))
model.add(Dense(12, activation="relu"))
model.add(Dense(4, activation="softmax"))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. Whe
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In our neural network, we are using two hidden layers of 16 and 12 dimension.

Now I will explain the code line by line.

Sequential specifies to keras that we are creating model sequentially and the output of each layer we add is input to the next layer we specify.

model.add is used to add a layer to our neural network. We need to specify as an argument what type of layer we want. The Dense is used to specify the fully connected layer. The arguments of Dense are output dimension which is 16 in the first case, input dimension which is 20 for input dimension and the activation function to be used which is relu in this case. The second layer is similar, we dont need to specify input dimension as we have defined the model to be sequential so keras will automatically consider input dimension to be same as the output of last layer i.e 16. In the third layer(output layer) the output dimension is 4(number of classes). Now as we have discussed earlier, the output layer takes different activation functions and for the case of multiclass classification, it is softmax.

Now we need to specify the loss function and the optimizer. It is done using compile function in keras.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Here loss is cross entropy loss as discussed earlier. Categorical_crossentropy specifies that we have multiple classes. The optimizer is Adam. Metrics is used to specify the way we want to judge the performance of our neural network. Here we have specified it to accuracy.

Now we are done with building a neural network and we will train it.

Training model

Training step is simple in keras. model.fit is used to train it.

```
history = model.fit(X_train, y_train, epochs=100, batch_size=64)
```

```
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9878 - loss: 0.0593
Epoch 85/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9913 - loss: 0.0548
Epoch 86/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9901 - loss: 0.0617
Epoch 87/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9905 - loss: 0.0541
Epoch 88/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9893 - loss: 0.0565
Epoch 89/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9944 - loss: 0.0470
Epoch 90/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9919 - loss: 0.0532
Epoch 91/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9939 - loss: 0.0483
Epoch 92/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9898 - loss: 0.0518
Epoch 93/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9919 - loss: 0.0496
Epoch 94/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9930 - loss: 0.0477
Epoch 95/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9903 - loss: 0.0495
Epoch 96/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9887 - loss: 0.0501
Epoch 97/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9919 - loss: 0.0516
Epoch 98/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9903 - loss: 0.0472
Epoch 99/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9945 - loss: 0.0425
Epoch 100/100
29/29 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9923 - loss: 0.0439
```

Here we need to specify the input data-> X_train, labels-> y_train, number of epochs(iterations), and batch size. It returns the history of model training. History consists of model accuracy and losses after each epoch. We will visualize it later.

Usually, the dataset is very big and we cannot fit complete data at once so we use batch size. This divides our data into batches each of size equal to batch_size. Now only this number of samples will be loaded into memory and processed. Once we are done with one batch it is flushed from memory and the next batch will be processed.

Now we have started the training of our neural network.

it will take around a minute to train. And after 100 epochs the neural network will be trained. The training accuracy is reached 99.5 % so our model is trained

Now we can check the model's performance on test data:

```
y_pred = model.predict(X_test)
#Converting predictions to label
pred = list()
for i in range(len(y_pred)):
    pred.append(np.argmax(y_pred[i]))
#Converting one hot encoded test label to label
test = list()
for i in range(len(y_test)):
    test.append(np.argmax(y_test[i]))
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
```

This step is inverse one hot encoding process. We will get integer labels using this step. We can predict on test data using a simple method of keras, model.predict(). It will take the test data as input and will return the prediction outputs as softmax.

```
from sklearn.metrics import accuracy_score
a = accuracy_score(pred,test)
print('Accuracy is:', a*100)
```

```
Accuracy is: 94.5
```

We can use test data as validation data and can check the accuracies after every epoch. This will give us an insight into overfitting at the time of training only and we can take steps before the completion of all epochs. We can do this by changing fit function as:

```
history = model.fit(X_train, y_train,validation_data = (X_test,y_test), epochs=100, batch_size=64)
```