SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

COURSE CODE: DJ19DSC501   **DATE:**

COURSE NAME: Machine Learning - II   **CLASS: AY 2023-24**

LAB EXPERIMENT NO.3

NAME: Bhuvi Ghosh   **SAP ID: 60009210191**   **BATCH: D22**

AIM :

Evaluate and analyze Prediction performance using appropriate optimizers for deep learning models.

THEORY:

Optimizers: Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. It finds the value of parameters(weights) that minimize the error when mapping inputs to outputs. These optimization algorithms or optimizers widely affect the accuracy of deep learning model and the speed of training of the model.

Types:

1. **Gradient Descent** - most basic but most used optimization algorithm. Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.

2. **Stochastic Gradient Descent** - variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example. So, if the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent

3. **Stochastic Gradient descent with momentum** - Momentum was invented for reducing high variance in SGD and softens the convergence. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction.

4. **Mini-Batch Gradient Descent** - best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model

parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.

5. **Adagrad** - This optimizer changes the learning rate - uses different learning rates for each iteration. It changes the learning rate 'η' for each parameter and at every time step 't'. It's a type second order optimization algorithm. It works on the derivative of an error function.

6. **RMSProp** - The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minima. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

7. **AdaDelta** - It is an extension of AdaGrad which tends to remove the decaying learning Rate problem of it. Instead of accumulating all previously squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size w. In this exponentially moving average is used rather than the sum of all the gradients.

8. **Adam** - adaptive moment estimation - adam optimizer updates the learning rate for each network weight individually. Adam optimizers inherit the features of both Adagrad and RMS prop algorithms. The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, Adam also keeps an exponentially decaying average of past gradients M(t).

Tasks to be performed:

    a) **Take the MNIST dataset**
    b) **Initialize a neural network basic layers with random weights.**
    c) **Perform practical analysis of optimizers on MNIST dataset keeping batch size, and epochs same but with different optimizers.**
    d) **Compare the results by choosing 8 different optimizers on a simple neural network**

[gradient desecent, Stochastic Gradient Descent, Stochastic Gradient descent with momentum, Mini-Batch Gradient Descent, Adagrad, RMSProp, AdaDelta, Adam]

    e) **List Advantages and Disadvantages of each Optimizer.**

```python
import pandas as pd
import numpy as np

np.set_printoptions(suppress=True)

CarPricesDataNumeric=pd.read_pickle('CarPricesData.pkl')
CarPricesDataNumeric.head()
```

	Age	KM	Weight	HP	MetColor	CC	Doors	Price
0	23.0	46986	1165.0	90	1	2000.0	3	13500
1	23.0	72937	1165.0	90	1	2000.0	3	13750
2	24.0	41711	1165.0	90	1	2000.0	3	13950
3	26.0	48000	1165.0	90	0	2000.0	3	14950
4	30.0	38500	1170.0	90	0	2000.0	3	13750

```python
TargetVariable=['Price']
Predictors=['Age', 'KM', 'Weight', 'HP', 'MetColor', 'CC', 'Doors']

X=CarPricesDataNumeric[Predictors].values
y=CarPricesDataNumeric[TargetVariable].values

from sklearn.preprocessing import StandardScaler
PredictorScaler=StandardScaler()
TargetVarScaler=StandardScaler()

PredictorScalerFit=PredictorScaler.fit(X)
TargetVarScalerFit=TargetVarScaler.fit(y)

X=PredictorScalerFit.transform(X)
y=TargetVarScalerFit.transform(y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(1004, 7)
(1004, 1)
(431, 7)
(431, 1)
```

```python
!pip install tensorflow
!pip install keras
```

```python
from keras.models import Sequential
from keras.layers import Dense, Dropout

model = Sequential()

model.add(Dense(units=5, input_dim=7, kernel_initializer='normal', activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(units=5, kernel_initializer='normal', activation='tanh'))
model.add(Dropout(0.2))

model.add(Dense(1, kernel_initializer='normal'))

model.compile(loss='mean_squared_error',metrics=['mae','mse'])# optimizer='adam'

history = model.fit(X_train, y_train ,validation_split=0.33,batch_size = 20, epochs = 10, verbose=1)
```
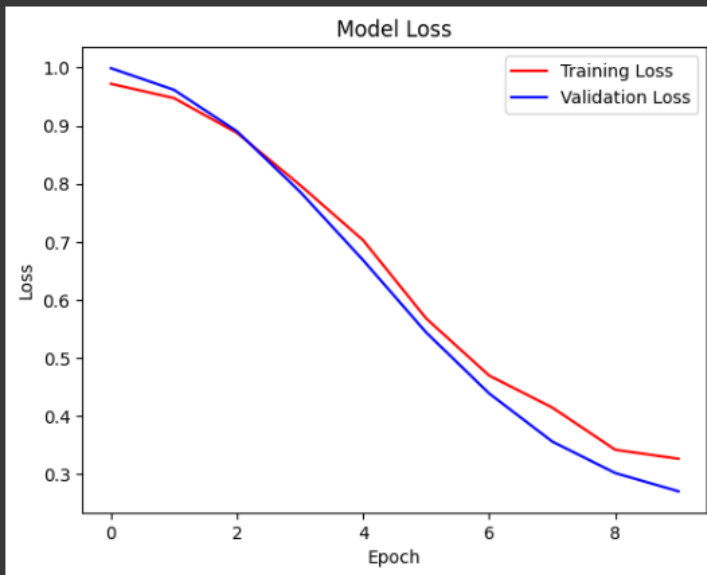
```
Epoch 1/10
34/34 [==============================] - 1s 9ms/step - loss: 0.9719 - mae: 0.7161 - mse: 0.9719 - val_loss: 0.9988 - val_mae: 0.7351 - val_mse: 0.9988
Epoch 2/10
34/34 [==============================] - 0s 5ms/step - loss: 0.9473 - mae: 0.7074 - mse: 0.9473 - val_loss: 0.9615 - val_mae: 0.7242 - val_mse: 0.9615
Epoch 3/10
34/34 [==============================] - 0s 5ms/step - loss: 0.8874 - mae: 0.6843 - mse: 0.8874 - val_loss: 0.8901 - val_mae: 0.6983 - val_mse: 0.8901
Epoch 4/10
34/34 [==============================] - 0s 9ms/step - loss: 0.7975 - mae: 0.6454 - mse: 0.7975 - val_loss: 0.7860 - val_mae: 0.6557 - val_mse: 0.7860
Epoch 5/10
34/34 [==============================] - 0s 4ms/step - loss: 0.7028 - mae: 0.5967 - mse: 0.7028 - val_loss: 0.6684 - val_mae: 0.5990 - val_mse: 0.6684
Epoch 6/10
34/34 [==============================] - 0s 5ms/step - loss: 0.5681 - mae: 0.5334 - mse: 0.5681 - val_loss: 0.5440 - val_mae: 0.5311 - val_mse: 0.5440
Epoch 7/10
34/34 [==============================] - 0s 3ms/step - loss: 0.4697 - mae: 0.4694 - mse: 0.4697 - val_loss: 0.4389 - val_mae: 0.4662 - val_mse: 0.4389
Epoch 8/10
34/34 [==============================] - 0s 5ms/step - loss: 0.4148 - mae: 0.4310 - mse: 0.4148 - val_loss: 0.3562 - val_mae: 0.4115 - val_mse: 0.3562
Epoch 9/10
34/34 [==============================] - 0s 4ms/step - loss: 0.3420 - mae: 0.4038 - mse: 0.3420 - val_loss: 0.3019 - val_mae: 0.3775 - val_mse: 0.3019
Epoch 10/10
34/34 [==============================] - 0s 5ms/step - loss: 0.3266 - mae: 0.4012 - mse: 0.3266 - val_loss: 0.2706 - val_mae: 0.3616 - val_mse: 0.2706
```

```
plt.plot(history.history['loss'],'r', label='Training Loss')
#plt.plot(history.history['val_loss'],'b', label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
y_pred = model.predict(X_test)
```

```
14/14 [==============================] - 0s 2ms/step
```

```python
from sklearn.metrics import mean_squared_error as mse
mse(y_test,y_pred)
```

```
0.29196312439766425
```

```python
import matplotlib.pyplot as plt
```

```python
test_predictions=model.predict(X_test)
test_predictions
```

```
[ 0.32102752],
[-0.6519463 ],
[-0.7491701 ],
[-0.63339514],
[ 0.68644005],
[-0.356234  ],
[ 0.21778   ],
[-0.63017887],
[-0.7659074 ],
[-0.50250554],
[ 0.19420567],
[-0.4399767 ],
[-0.4314459 ],
[ 1.5954922 ],
[ 0.28106818],
[ 0.8787774 ],
[ 0.15963063],
[ 0.35279262],
[-0.47928053],
[ 0.01693684],
[ 1.2178272 ],
[-0.13147369],
[-0.6100435 ],
[ 1.2193887 ],
[-0.6035351 ],
```

```python
loss,mae,mse=model.evaluate(X_test,y_test,verbose=0)
print('Testing mean squared error:MPG\t', format(mse))
print('Testing mean absolute error:MPG\t', format(mae))
print('Testing loss:MPG\t          ', format(loss))
```

```
Testing mean squared error:MPG    0.291963130235672
Testing mean absolute error:MPG   0.3501091003417969
Testing loss:MPG                  0.291963130235672
```