DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Experiment No. 01

Aim :- To explore python libraries for deep learning e.g. Theano, TensorFlow etc.

Theory :-

1. **TensorFlow :** TensorFlow is a popular open-source library for deep learning developed by Google. It provides a flexible ecosystem for building and deploying machine learning models. TensorFlow uses a computational graph approach, where you define a graph of operations and execute them within a TensorFlow session**.** It offers a high-level API called Keras that simplifies the process of building and training deep learning models.
   TensorFlow supports distributed computing, allowing you to train models on multiple machines or GPUs simultaneously.
   TensorFlow offers hardware acceleration through its integration with specialized hardware like GPUs and TPUs (Tensor Processing Units).
   - **Eager Execution:** TensorFlow 2.x introduced Eager Execution, which allows immediate evaluation of operations, making TensorFlow behave more like standard Python code.
   - **tf.data API:** TensorFlow provides the tf.data API, which enables efficient and flexible data input pipelines for large datasets.
   - **SavedModel:** TensorFlow's SavedModel format allows you to save the entire model, including the architecture, weights, and training configuration.
   - **Custom Training Loops:** TensorFlow enables you to build custom training loops, providing greater control over the training process.

```
[16] import tensorflow as tf

     # Define the input placeholders
     a = tf.keras.Input(shape=(2,), dtype=tf.float32)
     b = tf.keras.Input(shape=(2,), dtype=tf.float32)

     # Define the matrix multiplication operation
     c = tf.matmul(a, tf.transpose(b))

     # Create a TensorFlow model
     model = tf.keras.Model(inputs=[a, b], outputs=c)

     # Perform matrix multiplication
     x = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
     y = tf.constant([[5, 6], [7, 8]], dtype=tf.float32)
     result = model.predict([x, y])

     print(result)

     1/1 [==============================] - 0s 105ms/step
     [[17. 23.]
      [39. 53.]]
```

2. **Keras (built on top of TensorFlow) :** Keras is a high-level neural networks API that provides a user-friendly interface for building and training deep learning models.
Keras supports multiple backends, including TensorFlow, Theano, and CNTK. In this example, we'll use TensorFlow as the backend.
Essentials/features -

>High-Level API >Model Subclassing >Callbacks
>Pre-Trained Models >Transfer Learning

```python
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Define the training data
X_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_train = np.array([0, 1, 1, 0])

# Build the model
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=1)

# Get the final predictions
predictions = model.predict(X_train)
predictions = np.round(predictions)

print("Final Predictions:")
for i in range(len(predictions)):
    print(f"Input: {X_train[i]}, Predicted: {predictions[i][0]}")
```

```
Epoch 1/100
4/4 [==============================] - 2s 6ms/step - loss: 0.6858 - accuracy: 0.2500
Epoch 2/100
4/4 [==============================] - 0s 6ms/step - loss: 0.6838 - accuracy: 0.2500
Epoch 3/100
4/4 [==============================] - 0s 6ms/step - loss: 0.6827 - accuracy: 0.2500
Epoch 4/100
4/4 [==============================] - 0s 7ms/step - loss: 0.6814 - accuracy: 0.2500
Epoch 5/100
4/4 [==============================] - 0s 8ms/step - loss: 0.6809 - accuracy: 0.2500
Epoch 6/100
4/4 [==============================] - 0s 7ms/step - loss: 0.6798 - accuracy: 0.2500
Epoch 7/100
4/4 [==============================] - 0s 5ms/step - loss: 0.6795 - accuracy: 0.2500
Epoch 8/100
4/4 [==============================] - 0s 17ms/step - loss: 0.6785 - accuracy: 0.2500
Epoch 9/100
4/4 [==============================] - 0s 10ms/step - loss: 0.6777 - accuracy: 0.2500
```

✓ 5s completed at 14:34

```
Epoch 98/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6072 - accuracy: 1.0000
Epoch 99/100
4/4 [==============================] - 0s 7ms/step - loss: 0.6068 - accuracy: 1.0000
Epoch 100/100
4/4 [==============================] - 0s 5ms/step - loss: 0.6058 - accuracy: 1.0000
1/1 [==============================] - 0s 104ms/step
Final Predictions:
Input: [0 0], Predicted: 0.0
Input: [0 1], Predicted: 1.0
Input: [1 0], Predicted: 1.0
Input: [1 1], Predicted: 0.0
```

3. **Theano :** Theano is a popular library for numerical computation and building deep learning models. It allows efficient computation on both CPUs and GPUs.
   Theano provides a symbolic math library that enables you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.
   Theano operates on the concept of symbolic computation. It builds a computational graph that represents the mathematical expressions as a series of operations.
   Theano was designed with deep learning in mind and has been used as a foundational library by other frameworks such as Keras.
   Theano's development has been discontinued, and its successor, JAX, provides similar functionality with additional features.
   - **Symbolic Computation:** One of the key features of Theano is its symbolic computation approach.
   - **Automatic Differentiation:** Theano provides automatic differentiation, which is essential for training neural networks through backpropagation.
   - **GPU Acceleration:** Theano has built-in support for GPU acceleration, allowing you to perform computations on compatible NVIDIA GPUs.
   - **Shared Variables:** Theano introduces shared variables, which allow you to allocate memory on the GPU or CPU and share it between different functions.
   - **Convolution and Pooling Operations:** Theano provides built-in functions for common operations used in deep learning, including convolution and pooling operations, which are fundamental in convolutional neural networks (CNNs).
   - **Speed and Efficiency:** Theano was designed to provide efficient computation, making it a suitable choice for complex mathematical operations often involved in deep learning models.
   - **Deprecated Status:** As of September 2020, Theano is no longer actively developed and maintained by its original developers.

```
[ ]  import theano
     import theano.tensor as T

     # Define the symbolic variables
     a = T.matrix('a')
     b = T.matrix('b')

     # Define the symbolic expression
     c = T.dot(a, b)

     # Compile the function
     matmul = theano.function([a, b], c)

     # Perform matrix multiplication
     x = [[1, 2], [3, 4]]
     y = [[5, 6], [7, 8]]
     result = matmul(x, y)

     print(result)
```

Conclusion :- We had successfully explored python libraries for deep learning such as Keras, TensorFlow, Theano and PyTorch.

3. **PyTorch :** PyTorch is an open-source deep learning library developed by Facebook's AI Research lab (FAIR). It provides a flexible and efficient framework for building and training neural networks. PyTorch is widely used by researchers and practitioners for various machine learning tasks due to its ease of use, dynamic computation graph, and excellent support for automatic differentiation, which is essential for training deep learning models.

   Key Concepts in PyTorch :

   - **Tensors:** Tensors are multi-dimensional arrays similar to NumPy arrays but with additional capabilities for GPU acceleration..
   - **Autograd:** PyTorch's automatic differentiation engine. It automatically tracks and computes gradients for tensors.
   - **Neural Network Module (nn.Module):** PyTorch provides a base class, nn.Module, that allows you to define neural network architectures as a series of layers or functions.
   - **Loss Functions:** PyTorch includes a variety of loss functions for different tasks, such as classification (cross-entropy loss) and regression (mean squared error).
   - **Optimizers:** PyTorch offers various optimization algorithms like stochastic gradient descent (SGD), Adam, RMSprop, etc., to update the model parameters during training.

```
[11] import torch
     import torch.nn as nn
     import torch.optim as optim
     from sklearn.datasets import load_iris
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import OneHotEncoder

     iris = load_iris()
     X, y = iris.data, iris.target
     encoder = OneHotEncoder()
     y_onehot = encoder.fit_transform(y.reshape(-1, 1)).toarray()
     X_tensor = torch.tensor(X, dtype=torch.float32)
     y_tensor = torch.tensor(y, dtype=torch.long)

     class SimpleNN(nn.Module):
         def __init__(self):
             super(SimpleNN, self).__init__()
             self.fc1 = nn.Linear(4, 10)
             self.fc2 = nn.Linear(10, 3)

         def forward(self, x):
             x = torch.relu(self.fc1(x))
             x = torch.softmax(self.fc2(x), dim=1)
             return x

     model = SimpleNN()
     criterion = nn.CrossEntropyLoss()
     optimizer = optim.Adam(model.parameters(), lr=0.001)

     for _ in range(50):
         optimizer.zero_grad()
         outputs = model(X_tensor)
         loss = criterion(outputs, y_tensor)
         loss.backward()
         optimizer.step()

     accuracy = (model(X_tensor).argmax(1) == y_tensor).float().mean().item()
     print("Accuracy:", accuracy)

     Accuracy: 0.6600000262260437
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Program - Output :

In [2]:
```
import torch
import torch.nn as nn
import torch.optim as optim

# XOR gate inputs and outputs
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(2, 4)  # 2 input features, 4 hidden units in the first layer
        self.fc2 = nn.Linear(4, 4)  # 4 hidden units, 4 hidden units in the second layer
        self.fc3 = nn.Linear(4, 1)  # 4 hidden units, 1 output

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))
        return x

model = MLP()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Training the model
num_epochs = 10000
for epoch in range(num_epochs):
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()

# Evaluating the model
with torch.no_grad():
    predicted = model(X)
    predicted = torch.round(predicted)  # Round predictions to 0 or 1
    print("Predicted:")
    print(predicted)
```

```
Predicted:
tensor([[0.],
        [1.],
        [0.],
        [1.]])
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

DLL_EXP3.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[1] import numpy as np
```

```
# Simulated data for linear regression
np.random.seed(42)
X = np.random.rand(50, 1)
y = 2 * X + 1 + 0.1 * np.random.randn(50, 1)

# Helper function to compute gradient for linear regression
def compute_gradient(xi, yi, W, b):
  error = (W * xi + b - yi)
  gradient_W = 2 * xi * error
  gradient_b = 2 * error
  return gradient_W, gradient_b
```

Stochastic Gradient Descent

```
# Optimization algorithms
def sgd(X, y, W, b, learning_rate, num_epochs):
  for _ in range(num_epochs):
    for xi, yi in zip(X, y):
      gradient_W, gradient_b = compute_gradient(xi, yi, W, b)
      W -= learning_rate * gradient_W
      b -= learning_rate * gradient_b
      return W, b

# Hyperparameters
learning_rate = 0.01
batch_size = 10
num_epochs = 50
beta = 0.9

# Initial parameters
W_initial, b_initial = np.random.randn(), np.random.randn()

# Apply optimization algorithms
W_sgd, b_sgd = sgd(X,y,W_initial,b_initial,learning_rate,num_epochs)
print("SGD - Final Parameters: W =", W_sgd, "b =", b_sgd)
```

```
SGD - Final Parameters: W = [0.10320444] b = [0.98500344]
```

Mini-Batch Gradient Descent

```
[4] def mini_batch_gd(X, y, W, b, learning_rate, batch_size, num_epochs):
      num_samples = len(X)
      for _ in range(num_epochs):
        for i in range(0, num_samples, batch_size):
          X_batch, y_batch = X[i:i+batch_size], y[i:i+batch_size]
          gradient_W_avg = np.mean([compute_gradient(xi, yi, W, b)[0] for xi, yi in zip(X_batch, y_batch)])
          gradient_b_avg = np.mean([compute_gradient(xi, yi, W, b)[1] for xi, yi in zip(X_batch, y_batch)])
          W -= learning_rate * gradient_W_avg
          b -= learning_rate * gradient_b_avg
      return W, b

    # Hyperparameters
    learning_rate = 0.01
    batch_size = 10
    num_epochs = 50
    beta = 0.9

    # Initial parameters
    W_initial, b_initial = np.random.randn(), np.random.randn()

    # Apply optimization algorithms
    W_mini_batch, b_mini_batch = mini_batch_gd(X,y,W_initial,b_initial, learning_rate,batch_size,num_epochs)
    print("Mini Batch GD - Final Parameters: W =", W_mini_batch, "b =", b_mini_batch)
```

```
Mini Batch GD - Final Parameters: W = 0.7911081086681657 b = 1.5708653450938141
```

Momentum Gradient Descent

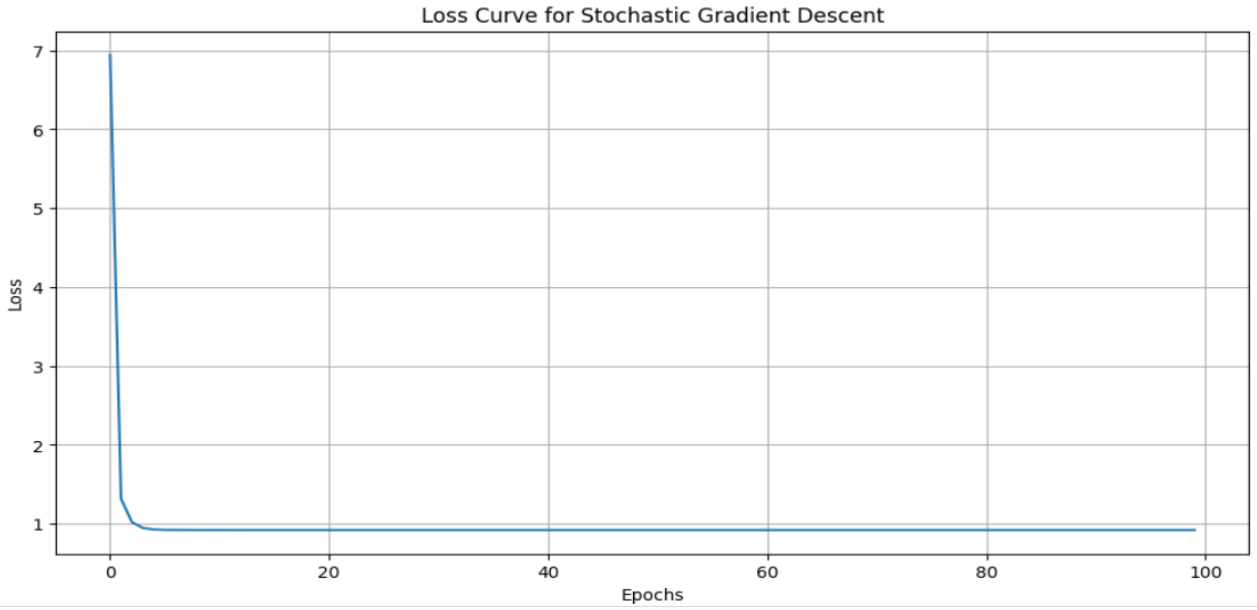
```
[5] def momentum_gd(X, y, W, b, learning_rate, beta, num_epochs):
      velocity_W, velocity_b = 0, 0
      for _ in range(num_epochs):
        for xi, yi in zip(X, y):
          gradient_W, gradient_b = compute_gradient(xi, yi, W, b)
          velocity_W = beta * velocity_W + (1 - beta) * gradient_W
          velocity_b = beta * velocity_b + (1 - beta) * gradient_b
          W -= learning_rate * velocity_W
          b -= learning_rate * velocity_b
      return W, b

    # Hyperparameters
    learning_rate = 0.01
    batch_size = 10
    num_epochs = 50
    beta = 0.9

    # Initial parameters
    W_initial, b_initial = np.random.randn(), np.random.randn()

    # Apply optimization algorithms
    W_momentum, b_momentum = momentum_gd(X,y,W_initial,b_initial, learning_rate,beta,num_epochs)
    print("Momentum GD - Final Parameters: W =", W_momentum, "b =", b_momentum)
```

```
Momentum GD - Final Parameters: W = [1.94629625] b = [1.02554776]
```

Stochastic Gradient Descent (SGD)

```
# Plot the loss curve
plt.figure(figsize=(12, 6))
plt.plot(range(epochs), loss_history, label='Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve for Stochastic Gradient Descent')
plt.grid()
plt.show()
```

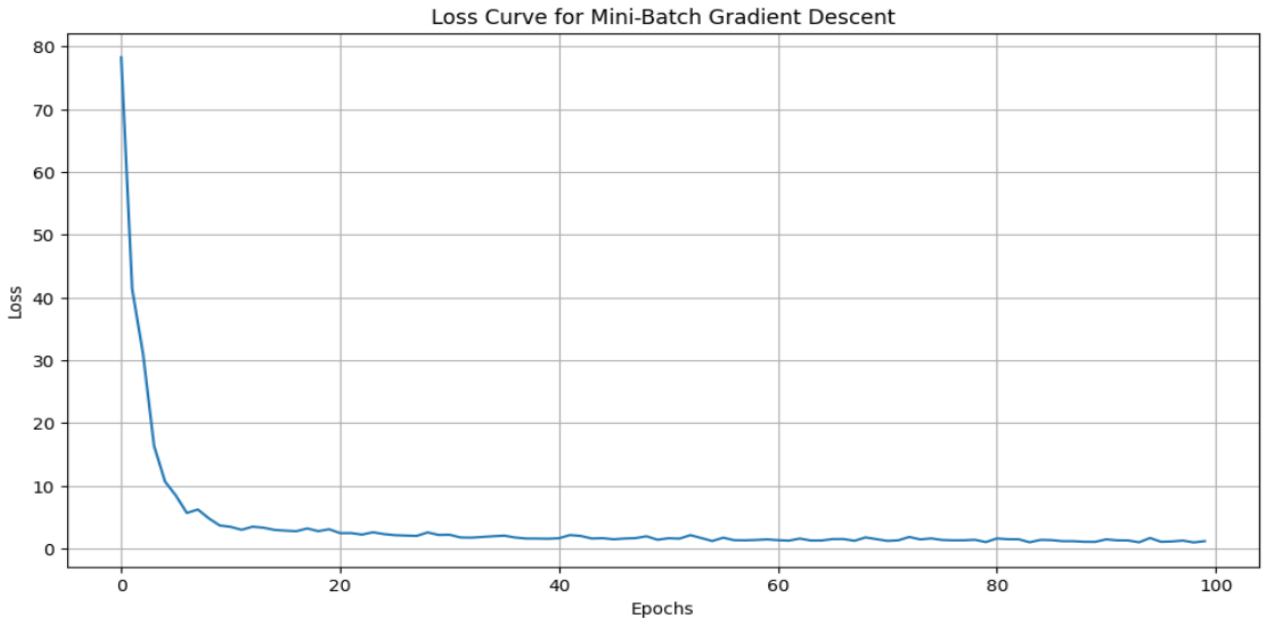
```
Learned Weights (W): [1.28056599 2.51443097 3.64171733]
Learned Bias (b): [2.81177659]
```



Mini-Batch Gradient Descent

```
# Plot the loss curve
plt.figure(figsize=(12, 6))
plt.plot(range(epochs), loss_history, label='Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve for Mini-Batch Gradient Descent')
plt.grid()
plt.show()
```

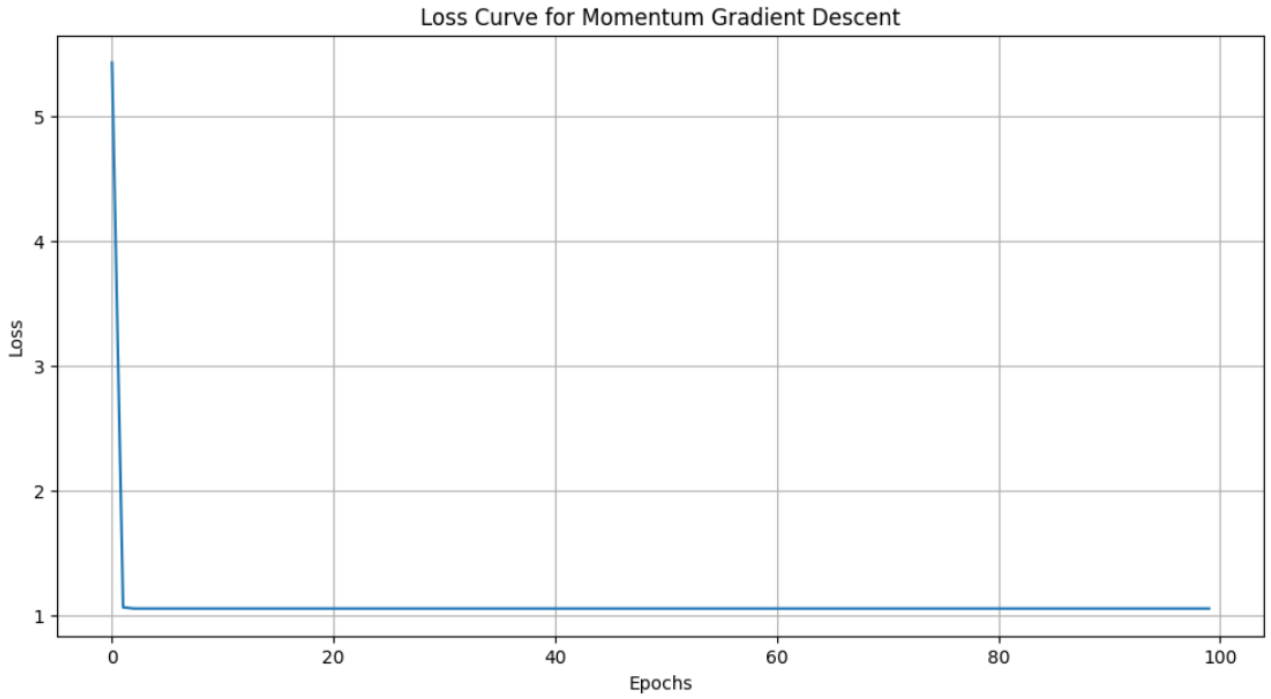
```
Learned Weights (W): [1.466288   2.66537392 3.34412419]
Learned Bias (b): [2.89975528]
```



Momentum Gradient Descent

```
# Plot the loss curve
plt.figure(figsize=(12, 6))
plt.plot(range(epochs), loss_history, label='Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve for Momentum Gradient Descent')
plt.grid()
plt.show()
```

```
Learned Weights (W): [1.53135691 2.41461102 3.45436068]
Learned Bias (b): [2.82780426]
```



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output :-

DLL_EXP4.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[9]  import numpy as np

     # Define a simple dataset for XOR
     X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
     y = np.array([[0], [1], [1], [0]])

     # Define the neural network architecture
     input_size, hidden_size1, hidden_size2, output_size = 2, 4, 3, 1
     learning_rate, epochs = 0.1, 10000

     # Initialize weights and biases
     weights = [np.random.uniform(size=(input_size, hidden_size1)),
                np.random.uniform(size=(hidden_size1, hidden_size2)),
                np.random.uniform(size=(hidden_size2, output_size))]
     biases = [np.zeros((1, hidden_size1)),
               np.zeros((1, hidden_size2)),
               np.zeros((1, output_size))]

     # Sigmoid activation function and its derivative
     sigmoid = lambda x: 1 / (1 + np.exp(-x))
     sigmoid_derivative = lambda x: x * (1 - x)
```

```
     # Training loop
     for epoch in range(epochs):
         # Forward pass
         layers = [X]
         for w, b in zip(weights, biases):
             layers.append(sigmoid(np.dot(layers[-1], w) + b))
         output = layers[-1]

         # Backpropagation
         delta = [y - output]
         for i in range(len(weights) - 1, 0, -1):
             delta.append(delta[-1].dot(weights[i].T) * sigmoid_derivative(layers[i]))

         # Update weights and biases
         for i in range(len(weights)):
             weights[i] += layers[i].T.dot(delta[-1 - i]) * learning_rate
             biases[i] += np.sum(delta[-1 - i], axis=0, keepdims=True) * learning_rate

         if epoch % 1000 == 0:
             loss = np.mean(0.5 * (y - output) ** 2)
             print(f"Epoch {epoch}: Loss {loss}")

     # Print final predictions
     print("Final Predictions:")
     print(output)
```

```
Epoch 0: Loss 0.14173703603991358
Epoch 1000: Loss 0.123212508461408
Epoch 2000: Loss 0.08778770918127633
Epoch 3000: Loss 0.07191910297219416
Epoch 4000: Loss 0.00022262241418865048
Epoch 5000: Loss 4.912702322947456e-05
Epoch 6000: Loss 2.0789432111937717e-05
Epoch 7000: Loss 1.1392281376228497e-05
Epoch 8000: Loss 7.174674996616704e-06
Epoch 9000: Loss 4.928971892945479e-06
Final Predictions:
[[0.00407558]
 [0.99782183]
 [0.99781978]
 [0.00162783]]
```

Output :-

```
[1]  import numpy as np

     # Define the neural network architecture
     input_size = 2
     hidden_size1 = 3
     hidden_size2 = 2
     output_size = 1

     # Initialize weights and biases for each layer
     weights_input_hidden1 = np.random.randn(input_size, hidden_size1)
     biases_hidden1 = np.zeros((1, hidden_size1))

     weights_hidden1_hidden2 = np.random.randn(hidden_size1, hidden_size2)
     biases_hidden2 = np.zeros((1, hidden_size2))

     weights_hidden2_output = np.random.randn(hidden_size2, output_size)
     biases_output = np.zeros((1, output_size))

     # Sample input data
     X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
     # Sample target output
     y = np.array([[0], [1], [1], [0]])

     # Define the learning rate
     learning_rate = 0.01

     # Training loop
     epochs = 10000
     for epoch in range(epochs):
         # Forward pass
         hidden1_input = np.dot(X, weights_input_hidden1) + biases_hidden1
         hidden1_output = 1 / (1 + np.exp(-hidden1_input))

         hidden2_input = np.dot(hidden1_output, weights_hidden1_hidden2) + biases_hidden2
         hidden2_output = 1 / (1 + np.exp(-hidden2_input))

         output = np.dot(hidden2_output, weights_hidden2_output) + biases_output
```

```
         # Calculate the loss
         loss = 0.5 * np.mean((output - y) ** 2)

         # Backpropagation
         dloss_doutput = output - y
         doutput_dhidden2_output = hidden2_output * (1 - hidden2_output)
         dloss_dhidden2_output = dloss_doutput.dot(weights_hidden2_output.T) * doutput_dhidden2_output

         dhidden2_output_dhidden1_output = hidden1_output * (1 - hidden1_output)
         dloss_dhidden1_output = dloss_dhidden2_output.dot(weights_hidden1_hidden2.T) * dhidden2_output_dhidden1_output

         # Update weights and biases
         weights_hidden2_output -= learning_rate * hidden2_output.T.dot(dloss_doutput)
         biases_output -= learning_rate * np.sum(dloss_doutput, axis=0, keepdims=True)

         weights_hidden1_hidden2 -= learning_rate * hidden1_output.T.dot(dloss_dhidden2_output)
         biases_hidden2 -= learning_rate * np.sum(dloss_dhidden2_output, axis=0, keepdims=True)

         weights_input_hidden1 -= learning_rate * X.T.dot(dloss_dhidden1_output)
         biases_hidden1 -= learning_rate * np.sum(dloss_dhidden1_output, axis=0, keepdims=True)

         if epoch % 1000 == 0:
             print(f'Epoch {epoch}, Loss: {loss}')
```

```
# Print final weights and biases
print("Final weights and biases:")
print("Weights Input to Hidden Layer 1:")
print(weights_input_hidden1)
print("Biases Hidden Layer 1:")
print(biases_hidden1)
print("Weights Hidden Layer 1 to Hidden Layer 2:")
print(weights_hidden1_hidden2)
print("Biases Hidden Layer 2:")
print(biases_hidden2)
print("Weights Hidden Layer 2 to Output Layer:")
print(weights_hidden2_output)
print("Biases Output Layer:")
print(biases_output)

print("Final Predictions")
print(output)
```

```
Epoch 0, Loss: 0.23812480513179063
Epoch 1000, Loss: 0.12593376620949448
Epoch 2000, Loss: 0.12568081295462774
Epoch 3000, Loss: 0.125501159986058
Epoch 4000, Loss: 0.12536576775687908
Epoch 5000, Loss: 0.12525762591421308
Epoch 6000, Loss: 0.12516583328803305
Epoch 7000, Loss: 0.12508264353810714
Epoch 8000, Loss: 0.1250018043609171
Epoch 9000, Loss: 0.12491744872159947
Final weights and biases:
Weights Input to Hidden Layer 1:
[[-0.67151782  0.75026953  0.39560969]
 [ 0.41462775 -0.21458599  0.91839141]]
Biases Hidden Layer 1:
[[-0.02962185 -0.04466992 -0.12989401]]
Weights Hidden Layer 1 to Hidden Layer 2:
[[-1.22731923 -1.82450893]
 [ 0.74629927  0.95909885]
 [-0.26582114  0.13230438]]
Biases Hidden Layer 2:
[[-0.21635442  0.19683121]]
Weights Hidden Layer 2 to Output Layer:
[[ 0.78343265]
 [-0.49679063]]
Biases Output Layer:
[[0.4538795]]
Final Predictions
[[0.50782372]
 [0.49428846]
 [0.50578433]
 [0.49061973]]
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

```
[1] import keras
    from keras import layers

    # This is the size of our encoded representations
    encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

    # This is our input image
    input_img = keras.Input(shape=(784,))

    # "encoded" is the encoded representation of the input
    encoded = layers.Dense(encoding_dim, activation='relu')(input_img)

    # "decoded" is the lossy reconstruction of the input
    decoded = layers.Dense(784, activation='sigmoid')(encoded)

    # This model maps an input to its reconstruction
    autoencoder = keras.Model(input_img, decoded)

    # This model maps an input to its encoded representation
    encoder = keras.Model(input_img, encoded)

    # This is our encoded (32-dimensional) input
    encoded_input = keras.Input(shape=(encoding_dim,))

    # Retrieve the last layer of the autoencoder model
    decoder_layer = autoencoder.layers[-1]

    # Create the decoder model
    decoder = keras.Model(encoded_input, decoder_layer(encoded_input))
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

```
[2] from keras.datasets import mnist
    import numpy as np
    (x_train, _), (x_test, _) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
```

```
[3] x_train = x_train.astype('float32') / 255.
    x_test = x_test.astype('float32') / 255.
    x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
    x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
    print(x_train.shape)
    print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```
[4] autoencoder.fit(x_train, x_train,
                    epochs=50,
                    batch_size=256,
                    shuffle=True,
                    validation_data=(x_test, x_test))
```

```
Epoch 1/50
235/235 [==============================] - 6s 21ms/step - loss: 0.2768 - val_loss: 0.1897
Epoch 2/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1708 - val_loss: 0.1542
Epoch 3/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1453 - val_loss: 0.1353
Epoch 4/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1292 - val_loss: 0.1214
Epoch 5/50
```

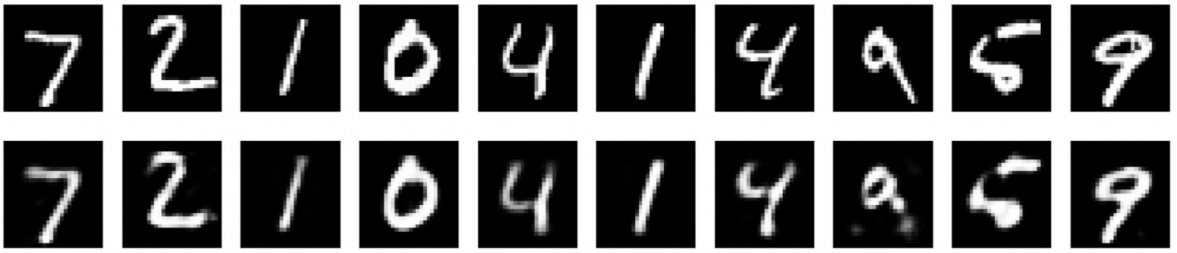
```
Epoch 47/50
235/235 [==============================] - 3s 13ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 48/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 49/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 50/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val_loss: 0.0915
<keras.src.callbacks.History at 0x7c36bd54ca60>
```

✓ 0s

```
[5]  # Encode and decode some digits
     # Note that we take them from the *test* set
     encoded_imgs = encoder.predict(x_test)
     decoded_imgs = decoder.predict(encoded_imgs)
```

```
313/313 [==============================] - 0s 1ms/step
313/313 [==============================] - 0s 1ms/step
```

```
[6]  # Use Matplotlib (don't ask)
     import matplotlib.pyplot as plt
     n = 10  # How many digits we will display
     plt.figure(figsize=(20, 4))
     for i in range(n):
         # Display original
         ax = plt.subplot(2, n, i + 1)
         plt.imshow(x_test[i].reshape(28, 28))
         plt.gray()
         ax.get_xaxis().set_visible(False)
         ax.get_yaxis().set_visible(False)
         # Display reconstruction
         ax = plt.subplot(2, n, i + 1 + n)
         plt.imshow(decoded_imgs[i].reshape(28, 28))
         plt.gray()
         ax.get_xaxis().set_visible(False)
         ax.get_yaxis().set_visible(False)
     plt.show()
```



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

DL_EXP5-denoising.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

Load the necessary Libraries

```
[1]  import numpy
     import matplotlib.pyplot as plt
     from keras.models import Sequential
     from keras.layers import Dense
     from keras.datasets import mnist
```

Load Dataset in Numpy Format

```
[2]  (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
[3]  X_train.shape
```

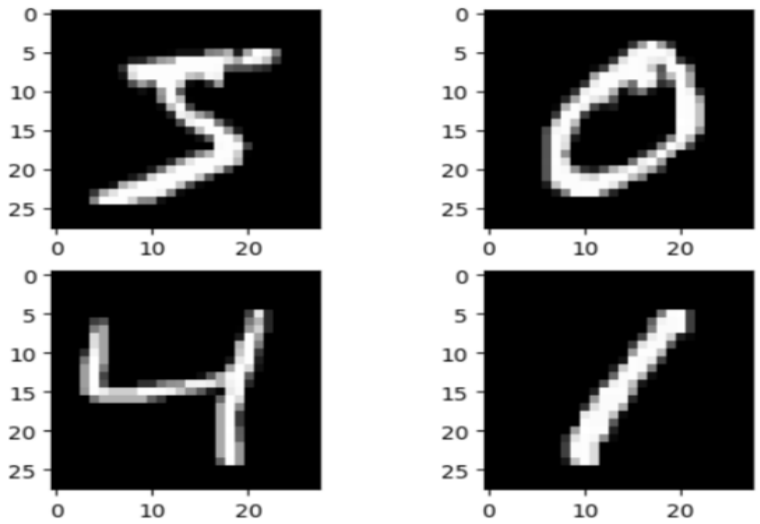
```
(60000, 28, 28)
```

```
[4]  X_test.shape
```

```
(10000, 28, 28)
```

Plot Images as a Grey Scale Image

```
[5]  plt.subplot(221)
     plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
     plt.subplot(222)
     plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
     plt.subplot(223)
     plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
     plt.subplot(224)
     plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
     # show the plot
     plt.show()
```



Formatting Data for Keras

```
[6]  num_pixels = X_train.shape[1] * X_train.shape[2]
     X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
     X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
     X_train = X_train / 255
     X_test = X_test / 255
```

```
[7]  X_train.shape
```

```
(60000, 784)
```

```
[8]  X_test.shape
```

```
(10000, 784)
```

Adding Noise to Images

```
[9]  noise_factor = 0.2
     x_train_noisy = X_train + noise_factor * numpy.random.normal(loc=0.0, scale=1.0, size=X_train.shape)
     x_test_noisy = X_test + noise_factor * numpy.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
     x_train_noisy = numpy.clip(x_train_noisy, 0., 1.)
     x_test_noisy = numpy.clip(x_test_noisy, 0., 1.)
```

Defining an Encoder-Decoder network

```
[10] # create model
     model = Sequential()
     model.add(Dense(500, input_dim=num_pixels, activation='relu'))
     model.add(Dense(300, activation='relu'))
     model.add(Dense(100, activation='relu'))
     model.add(Dense(300, activation='relu'))
     model.add(Dense(500, activation='relu'))
     model.add(Dense(784, activation='sigmoid'))
```

Compiling the model

```
[11] # Compile the model
     model.compile(loss='mean_squared_error', optimizer='adam')
```

Training or Fitting the model

```
[12] # Training model
     model.fit(x_train_noisy, X_train, validation_data=(x_test_noisy, X_test), epochs=2, batch_size=200)
```

```
Epoch 1/2
300/300 [==============================] - 10s 29ms/step - loss: 0.0422 - val_loss: 0.0203
Epoch 2/2
300/300 [==============================] - 11s 35ms/step - loss: 0.0169 - val_loss: 0.0139
<keras.src.callbacks.History at 0x793b0027d270>
```

Evaluating the model

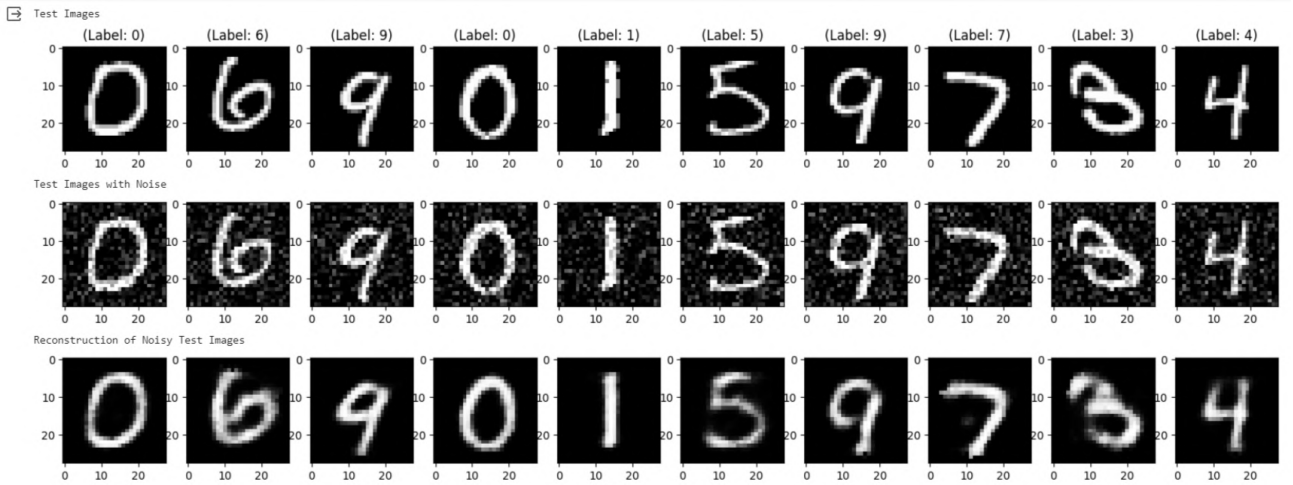
```
[13] # Final evaluation of the model
     pred = model.predict(x_test_noisy)
     pred.shape
```

```
313/313 [==============================] - 1s 4ms/step
(10000, 784)
```

```
[14] X_test.shape
```

```
(10000, 784)
```

```
[15] X_test = numpy.reshape(X_test, (10000,28,28)) *255
     pred = numpy.reshape(pred, (10000,28,28)) *255
     x_test_noisy = numpy.reshape(x_test_noisy, (-1,28,28)) *255
     plt.figure(figsize=(20, 4))
     print("Test Images")
     for i in range(10,20,1):
         plt.subplot(2, 10, i+1)
         plt.imshow(X_test[i,:,:], cmap='gray')
         curr_lbl = y_test[i]
         plt.title("(Label: " + str(curr_lbl) + ")")
     plt.show()
     plt.figure(figsize=(20, 4))
     print("Test Images with Noise")
     for i in range(10,20,1):
         plt.subplot(2, 10, i+1)
         plt.imshow(x_test_noisy[i,:,:], cmap='gray')
     plt.show()
     plt.figure(figsize=(20, 4))
     print("Reconstruction of Noisy Test Images")
     for i in range(10,20,1):
         plt.subplot(2, 10, i+1)
         plt.imshow(pred[i,:,:], cmap='gray')
     plt.show()
```



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

Import TensorFlow

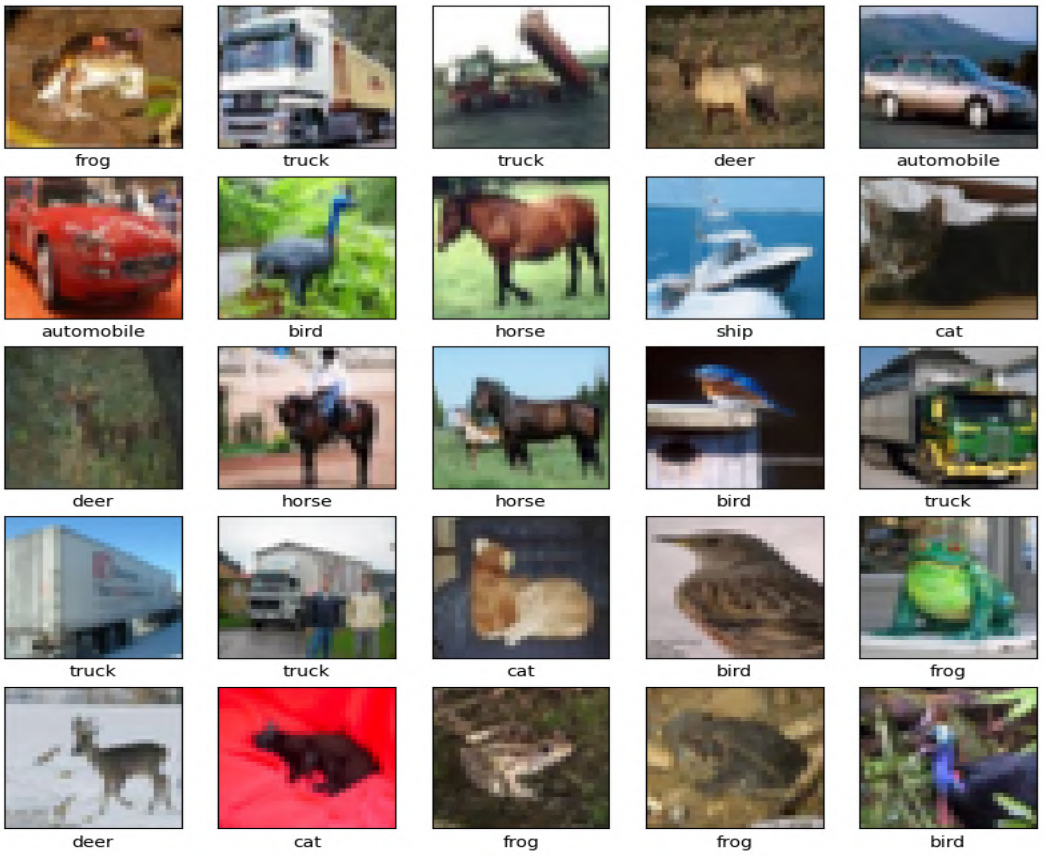
```
[2] import tensorflow as tf
    from tensorflow.keras import datasets, layers, models
    import matplotlib.pyplot as plt
```

```
[3] (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
    # Normalize pixel values to be between 0 and 1
    train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 14s 0us/step
```

Verify the data

```
[4] class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                   'dog', 'frog', 'horse', 'ship', 'truck']

    plt.figure(figsize=(10,10))
    for i in range(25):
        plt.subplot(5,5,i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(train_images[i])
        # The CIFAR labels happen to be arrays,
        # which is why you need the extra index
        plt.xlabel(class_names[train_labels[i][0]])
    plt.show()
```



Create the convolutional base

```
[5]  model = models.Sequential()
     model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Let's display the architecture of your model so far:

```
[6]  model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

=================================================================
Total params: 56320 (220.00 KB)
Trainable params: 56320 (220.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
[7]  model.add(layers.Flatten())
     model.add(layers.Dense(64, activation='relu'))
     model.add(layers.Dense(10))
```

Here's the complete architecture of model:

```
[8]  model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122570 (478.79 KB)
Trainable params: 122570 (478.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Compile and train the model

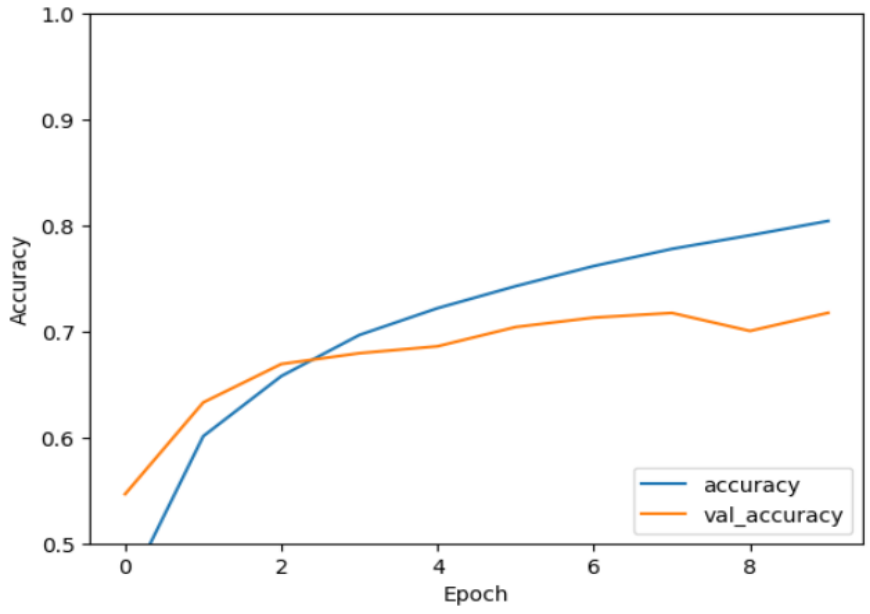
```
[9] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=10,
                        validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 [==============================] - 23s 6ms/step - loss: 1.5035 - accuracy: 0.4517 - val_loss: 1.2647 - val_accuracy: 0.5465
Epoch 2/10
1563/1563 [==============================] - 9s 6ms/step - loss: 1.1228 - accuracy: 0.6011 - val_loss: 1.0465 - val_accuracy: 0.6330
Epoch 3/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.9661 - accuracy: 0.6579 - val_loss: 0.9493 - val_accuracy: 0.6693
Epoch 4/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.8638 - accuracy: 0.6967 - val_loss: 0.9379 - val_accuracy: 0.6795
Epoch 5/10
1563/1563 [==============================] - 9s 6ms/step - loss: 0.7929 - accuracy: 0.7220 - val_loss: 0.9127 - val_accuracy: 0.6860
Epoch 6/10
1563/1563 [==============================] - 9s 6ms/step - loss: 0.7316 - accuracy: 0.7428 - val_loss: 0.8524 - val_accuracy: 0.7042
Epoch 7/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.6782 - accuracy: 0.7618 - val_loss: 0.8613 - val_accuracy: 0.7131
Epoch 8/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.6359 - accuracy: 0.7779 - val_loss: 0.8538 - val_accuracy: 0.7175
Epoch 9/10
1563/1563 [==============================] - 9s 6ms/step - loss: 0.5925 - accuracy: 0.7908 - val_loss: 0.9277 - val_accuracy: 0.7005
Epoch 10/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.5552 - accuracy: 0.8043 - val_loss: 0.8984 - val_accuracy: 0.7176
```

Evaluate the model

```
[10] plt.plot(history.history['accuracy'], label='accuracy')
     plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
     plt.xlabel('Epoch')
     plt.ylabel('Accuracy')
     plt.ylim([0.5, 1])
     plt.legend(loc='lower right')

     test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
313/313 - 1s - loss: 0.8984 - accuracy: 0.7176 - 687ms/epoch - 2ms/step
```




```
[11] print(test_acc)
```

```
0.7175999879837036
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

!pip install scalecast --upgrade

!pip install tensorflow

Data Preprocessing

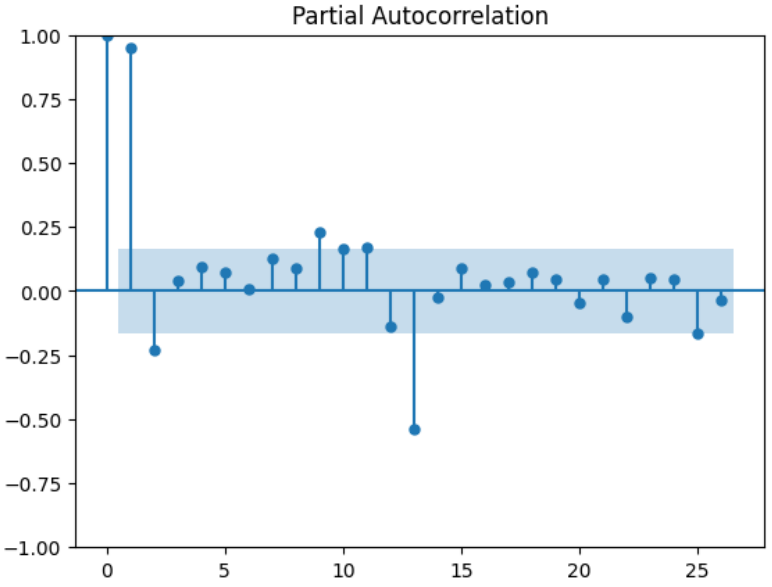
```
[11] import pandas as pd
     import numpy as np
     import pickle
     import seaborn as sns
     import matplotlib.pyplot as plt
     from scalecast.Forecaster import Forecaster

     df = pd.read_csv('AirPassengers.csv',parse_dates=['Month'])
```

Exploratory Data Analysis

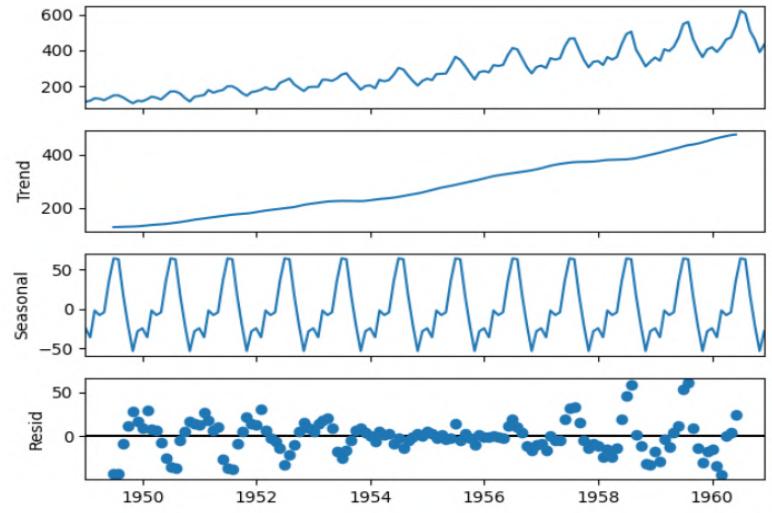
```
[12] f = Forecaster(y=df['#Passengers'],current_dates=df['Month'])
     f
```

```
Forecaster(
    DateStartActuals=1949-01-01T00:00:00.000000000
    DateEndActuals=1960-12-01T00:00:00.000000000
    Freq=MS
    N_actuals=144
    ForecastLength=0
    Xvars=[]
    TestLength=0
    ValidationMetric=rmse
    ForecastsEvaluated=[]
    CILevel=None
    CurrentEstimator=mlr
    GridsFile=Grids
)
```

```
[13] f.plot_pacf(lags=26)
     plt.show()
```



```
[14] f.seasonal_decompose().plot()
     plt.show()
```




```
[15] stat, pval, _, _, _, _ = f.adf_test(full_res=True)
     stat

     0.8153688792060498
```

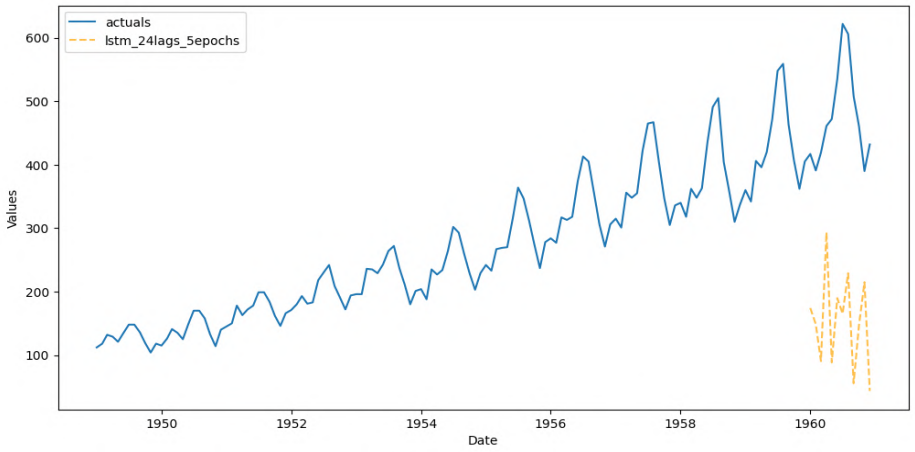
```
[16] pval

     0.991880243437641
```

LSTM Forecasting

```
[17] f.set_test_length(12)       # 1. 12 observations to test the results
     f.generate_future_dates(12) # 2. 12 future points to forecast
     f.set_estimator('lstm')     # 3. LSTM neural network
```

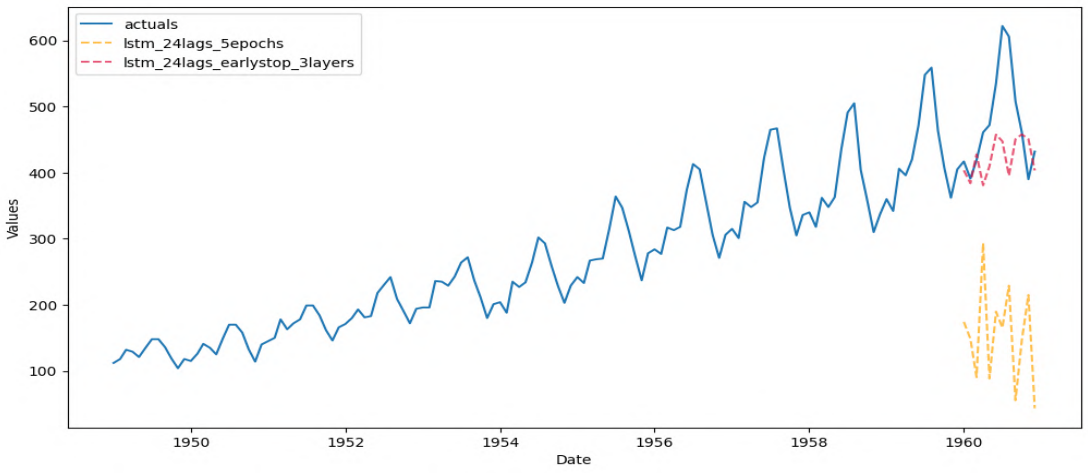
```
[18] f.manual_forecast(
         call_me='lstm_24lags_5epochs',
         lags=24,
         epochs=5,
         validation_split=.2,
         shuffle=True,
     )
     f.plot_test_set(ci=True)
```

```
Epoch 2/5
3/3 [==============================] - 0s 98ms/step - loss: 0.3250 - val_loss: 0.5714
Epoch 3/5
3/3 [==============================] - 0s 100ms/step - loss: 0.3181 - val_loss: 0.5611
Epoch 4/5
3/3 [==============================] - 0s 82ms/step - loss: 0.3109 - val_loss: 0.5505
Epoch 5/5
3/3 [==============================] - 0s 70ms/step - loss: 0.3035 - val_loss: 0.5393
```



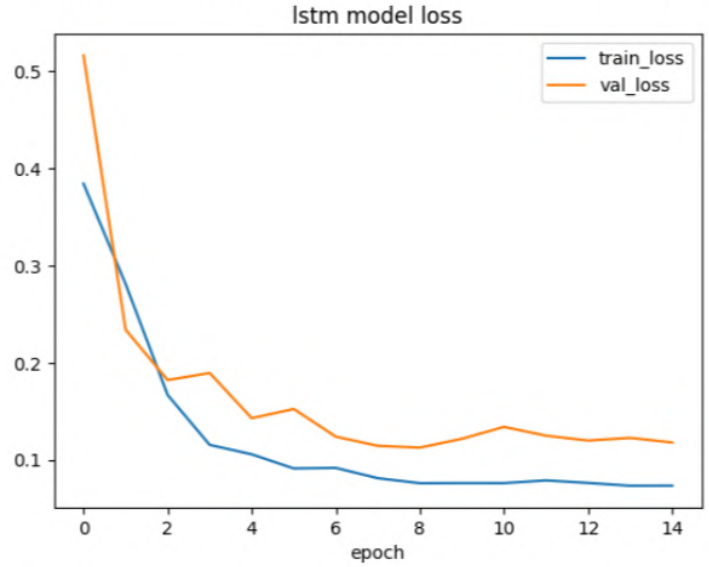
```
[19] from tensorflow.keras.callbacks import EarlyStopping

     f.manual_forecast(
         call_me='lstm_24lags_earlystop_3layers',
         lags=24,
         epochs=25,
         validation_split=.2,
         shuffle=True,
         callbacks=EarlyStopping(
             monitor='val_loss',
             patience=5,
         ),
         lstm_layer_sizes=(16,16,16),
         dropout=(0,0,0),
     )

     f.plot_test_set(ci=True)
```

```
Epoch 1/25
3/3 [==============================] - 10s 922ms/step - loss: 0.3553 - val_loss: 0.6296
Epoch 2/25
3/3 [==============================] - 0s 63ms/step - loss: 0.3400 - val_loss: 0.6073
Epoch 3/25
3/3 [==============================] - 0s 80ms/step - loss: 0.3245 - val_loss: 0.5831
Epoch 4/25
3/3 [==============================] - 0s 86ms/step - loss: 0.3071 - val_loss: 0.5544
Epoch 5/25
3/3 [==============================] - 0s 78ms/step - loss: 0.2859 - val_loss: 0.5174
Epoch 6/25
3/3 [==============================] - 0s 68ms/step - loss: 0.2583 - val_loss: 0.4663
Epoch 7/25
3/3 [==============================] - 0s 54ms/step - loss: 0.0824 - val_loss: 0.1556
Epoch 24/25
3/3 [==============================] - 0s 50ms/step - loss: 0.0781 - val_loss: 0.1520
Epoch 25/25
3/3 [==============================] - 0s 53ms/step - loss: 0.0749 - val_loss: 0.1470
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_predict_function.<locals>.predict_function at 0x797362aa36d0>
1/1 [==============================] - 1s 1s/step
4/4 [==============================] - 0s 9ms/step
/usr/local/lib/python3.10/dist-packages/scalecast/_utils.py:60: Warning: Confidence intervals not found for lstm_24lags_5epochs. To
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/scalecast/_utils.py:60: Warning: Confidence intervals not found for lstm_24lags_earlystop_3
  warnings.warn(
<Axes: xlabel='Date', ylabel='Values'>
```

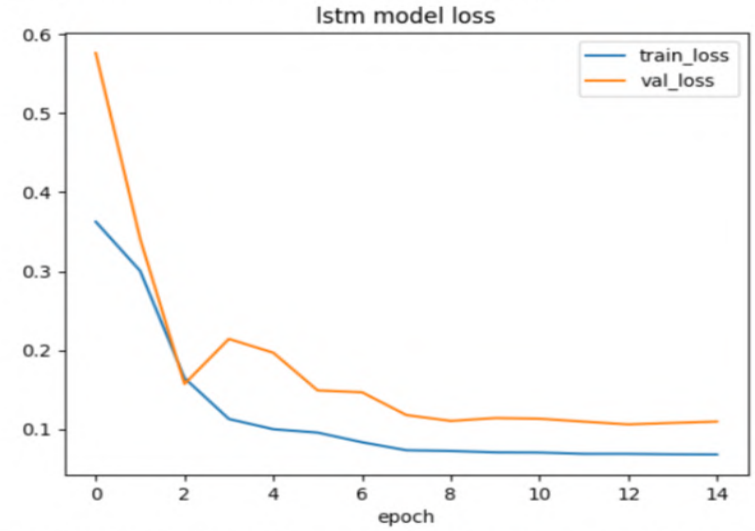



```
[21] f.manual_forecast(
         call_me='lstm_best',
         lags=36,
         batch_size=32,
         epochs=15,
         validation_split=.2,
         shuffle=True,
         activation='tanh',
         optimizer='Adam',
         learning_rate=0.001,
         lstm_layer_sizes=(72,)*4,
         dropout=(0,)*4,
         plot_loss=True
     )
     f.plot_test_set(order_by='TestSetMAPE', models='top_2', ci=True)
```

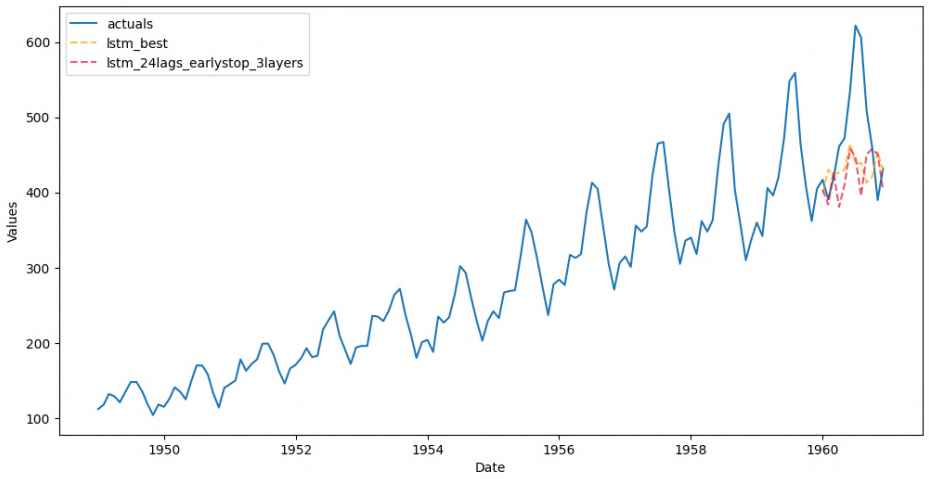
```
Epoch 1/15
3/3 [==============================] - 11s 1s/step - loss: 0.3839 - val_loss: 0.5157
Epoch 2/15
3/3 [==============================] - 0s 97ms/step - loss: 0.2809 - val_loss: 0.2339
Epoch 3/15
3/3 [==============================] - 0s 97ms/step - loss: 0.1668 - val_loss: 0.1821
Epoch 4/15
3/3 [==============================] - 0s 99ms/step - loss: 0.1155 - val_loss: 0.1894
...
Epoch 14/15
3/3 [==============================] - 0s 98ms/step - loss: 0.0735 - val_loss: 0.1226
Epoch 15/15
3/3 [==============================] - 0s 95ms/step - loss: 0.0736 - val_loss: 0.1179
1/1 [==============================] - 1s 1s/step
```




```
Epoch 1/15
3/3 [==============================] - 10s 864ms/step - loss: 0.3627 - val_loss: 0.5764
Epoch 2/15
3/3 [==============================] - 0s 106ms/step - loss: 0.3004 - val_loss: 0.3410
Epoch 3/15
3/3 [==============================] - 0s 103ms/step - loss: 0.1646 - val_loss: 0.1572
Epoch 4/15
...
Epoch 14/15
3/3 [==============================] - 0s 112ms/step - loss: 0.0680 - val_loss: 0.1078
Epoch 15/15
3/3 [==============================] - 0s 106ms/step - loss: 0.0678 - val_loss: 0.1094
1/1 [==============================] - 2s 2s/step
3/3 [==============================] - 0s 39ms/step
```



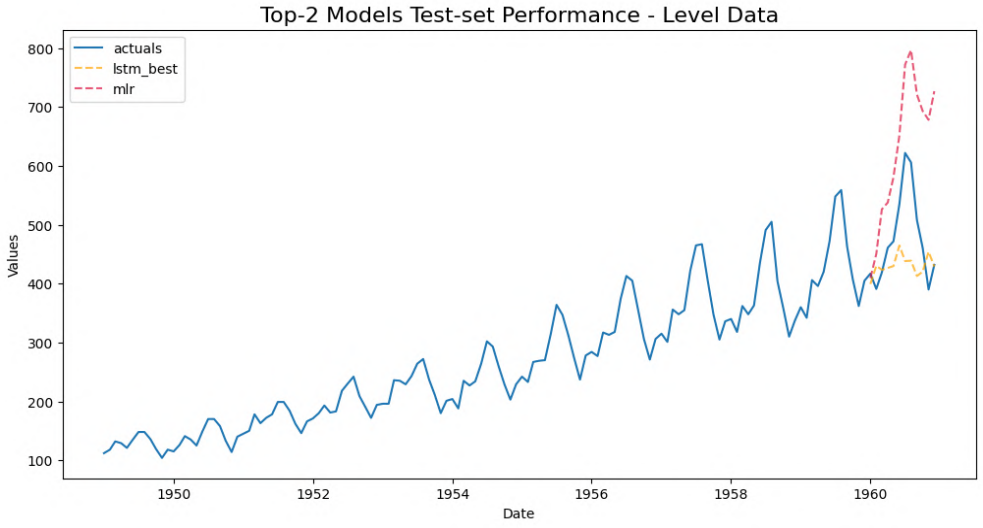

```
/usr/local/lib/python3.10/dist-packages/scalecast/_utils.py:60: Warning: Confidence inte
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/scalecast/_utils.py:60: Warning: Confidence inte
  warnings.warn(
<Axes: xlabel='Date', ylabel='Values'>
```



MLR Forecasting and Model Benchmarking

[45]
```
from scalecast.SeriesTransformer import SeriesTransformer

transformer = SeriesTransformer(f)
f = transformer.DiffTransform()

f.add_ar_terms(24)
f.add_seasonal_regressors('month','quarter',dummy=True)
f.add_seasonal_regressors('year')
f.add_time_trend()
```

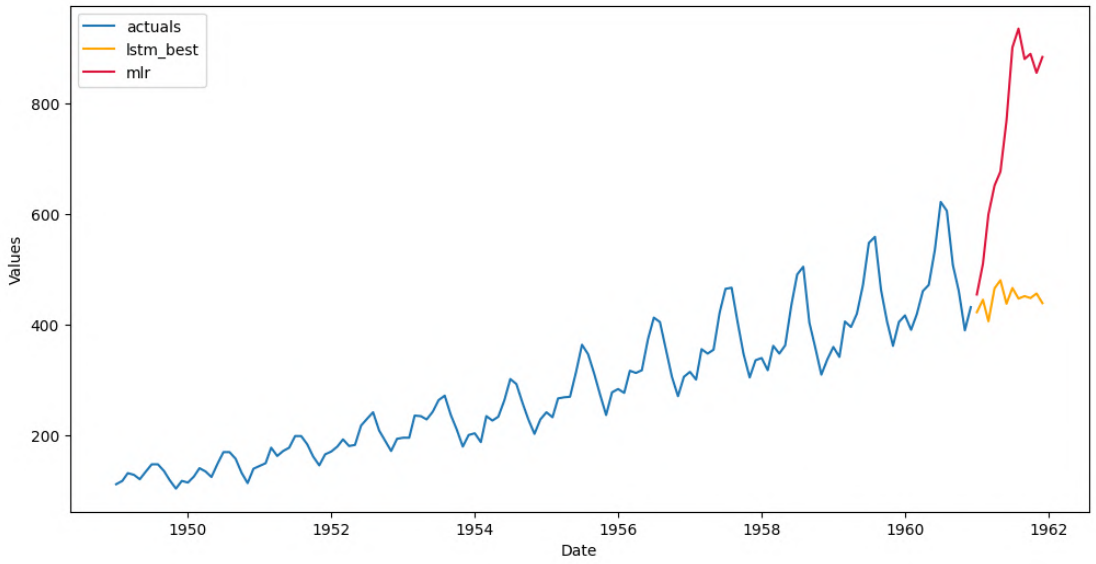
[46]
```
f.set_estimator('mlr')
f.manual_forecast()

f = transformer.DiffRevert(
    exclude_models = [m for m in f.history if m != 'mlr']
) # exclude all lstm models from the revert

f.plot_test_set(order_by='TestSetMAPE',models=['lstm_best','mlr'])
plt.title('Top-2 Models Test-set Performance - Level Data',size=16)
plt.show()
```



```
[57] f.plot(models=['mlr', 'lstm_best'], order_by='TestSetMAPE')
```

2s

```
<Axes: xlabel='Date', ylabel='Values'>
```




```
[65] f.export('model_summaries',determine_best_by='InSampleRMSE')[['ModelNickname','InSampleR2','TestSetRMSE', 'TestSetMAPE', 'TestSetMAE', 'TestSetR2','best_model']]
```

0s

	ModelNickname	InSampleR2	TestSetRMSE	TestSetMAPE	TestSetMAE	TestSetR2	best_model
0	mlr	0.970799	176.765019	0.328659	153.879150	-4.640586	True
1	lstm_best	0.704907	84.740822	0.122145	63.453901	-0.296334	False
2	lstm_24lags_earlystop_3layers	0.643474	90.941170	0.124324	65.348639	-0.492976	False
3	lstm_24lags_5epochs	-0.816249	336.169279	0.673811	322.878162	-19.400822	False

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

B.E / SEM VII / REV 2019 'C SCHEME' / CSE-(AI&ML)
Academic Year: 2023-24

NAME	**SINGH SUDHAM DHARMENDRA**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

!pip install tensorflow

```
[19] import numpy as np
     import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import SimpleRNN, Dense, Embedding
     from tensorflow.keras.preprocessing.text import Tokenizer
     from tensorflow.keras.preprocessing.sequence import pad_sequences

     # Sample data for character-level translation
     input_texts = ['hello', 'world', 'good', 'bye', 'prathmesh', 'op', 'sudhamesh']
     target_texts = ['olleh', 'dlrow', 'doog', 'eyb', 'hsemhtarp', 'po', 'hsemahdus']

     # Tokenize input and target texts
     tokenizer = Tokenizer(char_level=True)
     tokenizer.fit_on_texts(input_texts + target_texts)
     input_sequences = tokenizer.texts_to_sequences(input_texts)
     target_sequences = tokenizer.texts_to_sequences(target_texts)

     # Determine the maximum sequence length
     max_seq_length = max(len(seq) for seq in input_sequences)

     # Pad sequences to the maximum length
     input_sequences = pad_sequences(input_sequences, maxlen=max_seq_length)
     target_sequences = pad_sequences(target_sequences, maxlen=max_seq_length)

     # Define the RNN model
     model = Sequential()
     model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=8, input_length=max_seq_length))
     model.add(SimpleRNN(200, return_sequences=True))
     model.add(Dense(len(tokenizer.word_index) + 1, activation='softmax'))

     # Compile the model
     model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')

     # Train the model
     model.fit(input_sequences, target_sequences, epochs=50)
```

```
     # Translate new input sequences
     new_input_texts = ['prathmesh','sudhamesh']
     new_input_sequences = tokenizer.texts_to_sequences(new_input_texts)
     new_input_sequences = pad_sequences(new_input_sequences, maxlen=max_seq_length)
     predicted_sequences = model.predict(new_input_sequences)
     predicted_sequences = np.argmax(predicted_sequences, axis=-1)
     for i, text in enumerate(new_input_texts):
       input_seq = new_input_sequences[i]
       predicted_seq = predicted_sequences[i]
       decoded_text = tokenizer.sequences_to_texts([predicted_seq])[0]
       print(f'Input: {text}, Predicted: {decoded_text}')
```

```
Epoch 1/50
1/1 [==============================] - 2s 2s/step - loss: 2.8332
Epoch 2/50
1/1 [==============================] - 0s 17ms/step - loss: 2.8066
Epoch 3/50
1/1 [==============================] - 0s 13ms/step - loss: 2.7785
Epoch 4/50
Epoch 47/50
1/1 [==============================] - 0s 17ms/step - loss: 1.0980
Epoch 48/50
1/1 [==============================] - 0s 13ms/step - loss: 1.0764
Epoch 49/50
1/1 [==============================] - 0s 16ms/step - loss: 1.0530
Epoch 50/50
1/1 [==============================] - 0s 14ms/step - loss: 1.0269
1/1 [==============================] - 0s 192ms/step
Input: prathmesh, Predicted: m h t a r p
Input: sudhamesh, Predicted: m a h d u s
```

Name :- Singh Sudham Dharmendra

Branch :- CSE [AIML]

Roll no :- AIML57

Subject :- Deep Learning

Topic :- Assignment No :- 1

Date of submission :- 27/08/2023

Signature :-

Q.1] What is Deep learning ? Explain three different classes of deep networks.

→ Deep Learning is a subset of Machine learning that focuses on artificial Neural Networks with Multiple layers, also known as Deep Neural Network.

These Networks are designed to automatically learn and extract hierarchical features from data.

(i) Feedforward Neural N/w [FFNN] :-

- A FFNN is an artificial neural network where connection between the nodes and do not form a cycle.
- As such it is different from its descendent : random neural network. The feed forward neural network was the first & simplest type of neural network device.
- In an artificial feed forward Network [FFNN], information always move in one direction, i.e, it never goes backwards.

(ii) Convolution Neural N/w [CNN]

- A CNN [Convolution Neural N/w] is a type of ANN used in major recognition. CNN are a category of Neural N/w that have proven very effective in areas such as image recognition & classification.
- CNNs are important tools for ML practioners today. CNN is a deep learning Neural N/w designed for processing structured arrays of data such as images

(iii) Recurrent Neural N/w [RNN] :-

- RNN [Recurrent Neural N/w] is a class of artificial Neural N/w where connection blw nodes form a directed graphs along a temporal sequence This makes it to show temporal : dynamic behaviour.
- It uses sequential data or time series data. These data are commonly used for ordinal or temporal problems, such as language translation, NLP, speech recognition & image captioning.

Q.2] Explain Multilayer Perceptrons [MLPs] and representation power of MLPs.

→ ① Multilayer perceptron is connected dense layers, and that transforms any i/p dimension to desired dimensional. A MLP is a neural N/w that has Multiple layers.

②

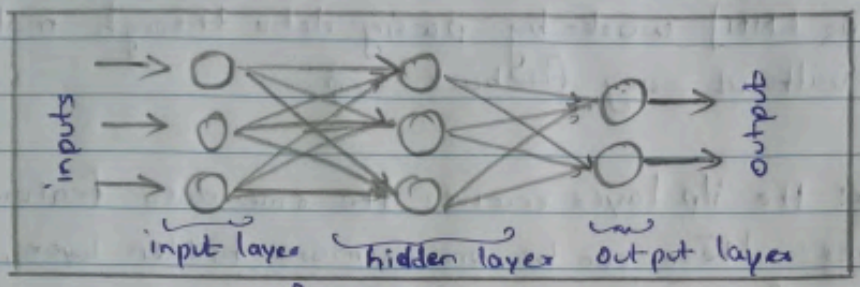


fig MLP

- In above diagram of MLP, there are three i/p & thus three i/p nodes and the hidden layer has three nodes.
- The o/p layers given two o/p, hence there are two o/p nodes. The nodes ~~i/p~~ in the i/p takes i/p & forward it for further process.
- The hidden layer process the information & passes it to the o/p layer.

③ Representational power [Activation function]:-

- Except for the i/p nodes, each node is a neuron that uses a non-linear 'activation function'. MLP utilizes a chain rule based 'supervised learning' technique called back propogation for training.
- The two common activation function are both sigmoids & are given as:

$y(x_i) = \tanh(x_i)$ & $y(x_i) = (1+e^{-x_i}) = \frac{1}{1+e^{-x_i}}$

- The first is a 'hyperbolic tangent' that ranges from -1 to 1 where the other is 'logistic function' which is similar in shape but ranges from 0 to 1.
- Here Y_i is the o/p & x_i is the weighted sum.
  - ↳ ith node
  - ↳ of i/p connected for.
- Alternate activation function have also been mentioned including the 'rectified' & 'softplus' function.

Q. 3] Explain the working of Multilayer feedforward neural network & backpropagation in detail.

→ * Multilayer feedforward neural network :-

① A multilayer feed forward N/w [also known as a feed foodward neural Network or FNN] works by passing data through multiple layers of neuron without any feedback loop.

② Working :-

- input layer: the i/p layer receives the raw data features.
- Hidden layer :- There can be one or more hidden layer, each containing of multiple neurons. Each neuron computs a weighted sum of its i/p; applies an activation function. [Eg :- Sigmoid & Relu] and passes the result to the next layer.
- ~~Output layer :- the o/p layer depends on the surface~~
- Output layer :- The o/p layer produces the final prediction or classification the no. of neurons in this layer depends on the specific task [Eg; one neuron for binary classification & Multiple neurons for multi-class classification].
- The weights & biases of neurons are initialized randomly, and then N/w output is compared to the true target values using a loss function [Eg :- Mean squared error for regression or cross-entropy for classification]

* Back Propagation :-

① Back Propagation is used to adjust the weights and biases to minimize the loss.

② Working :-

- Forward pass :- the i/p data is passed through the N/w, and predictions are obtained.
- loss computation :- the loss is calculated by computing prediction to the true labels.

- Backward pass [Back propagate] :- Errors are propagated backward through the N/w. The Gradient of the loss with respect to the weight and biases are computed using the chain rule of calculus.
- Weight update :- The weight & bias are updated in the opposite direction of the gradient to minimize the loss. This is typically done using optimization algorithm like gradient descent.
- The process is repeated iteratively until the model convergence to the set of weights & biases that produces accurate prediction.

Q.4] Explain different activation functions in detail.

→ (i) tanh activation :- Used for neural n/w

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \tanh(x)$$

the tanh function become preferred over the sigmoid function as it gives better performance for multilayer Neural Network.
But it doesn't solve vanishing gradient problem.

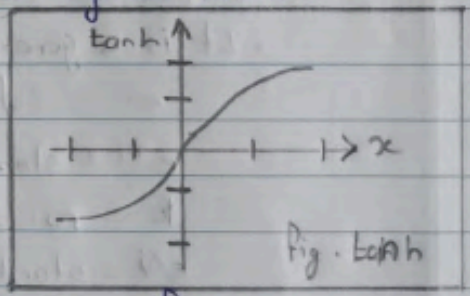


Fig. tanh

(ii) Logistic activation function :- Also known as sigmoid function
This function is back-propagation nets are of two types.

- Binary sigmoidal function :-
Also known as unipolar sigmoid function or logistic sigmoid function.
& defined as $f(x) = \frac{1}{1+e^{-\lambda x}}$
Ranges from 0 to 1

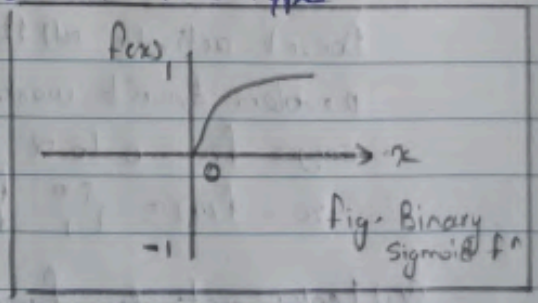


Fig. Binary Sigmoid f^n

- Bipolar sigmoid function :-
Given as :- $f(x) = \frac{2}{1+e^{-\lambda x}} - 1 = \frac{2 - 1 - e^{-\lambda x}}{1+e^{-\lambda x}} = \frac{1 - e^{-\lambda x}}{1+e^{-\lambda x}}$

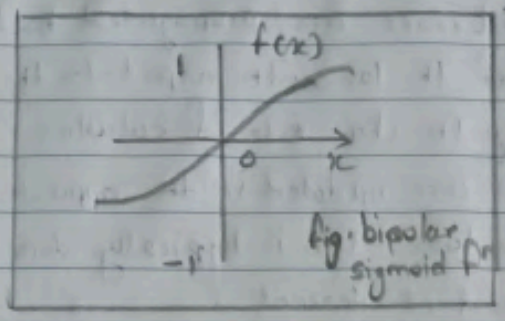


fig. bipolar sigmoid fn

Sigmoid function are so called as because the graph are S-shaped. Sigmoid function defined to be odd, are monotonic function of one variable

(iii) Linear activation function:-
defined as, $f(x) = x$ ∴ for all x
here i/p = o/p

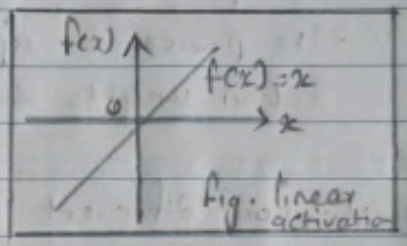


fig. linear activation

(iv) Softmax activation function:-
- also known as soft as Gaumbo normalized exponential function. It converts a vector of k real ~~nos~~ no. into a probability distribution of k possible outcomes.
- It is a generalization of logistic function, formula is;

$$\therefore \sigma = (\vec{z})_i = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_j}}$$

where σ = softmax, $\vec{z}$ = i/p vector
e^{z_i} = standard exponential function for i/p vector
k = no. of classes in multi-class classification
e^{z_j} = standard exponential function for o/p vector

(v) Relu function:- Rectified Linear Unit
Doesn't activate all the neurons at the same time & vanishing gradient problem doesn't work in Relu
ranges from 0 to α, eqn is $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
or $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

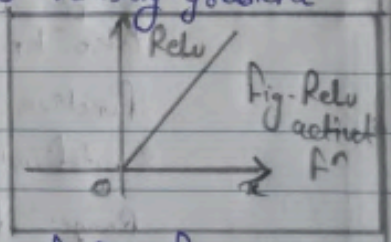


fig. Relu activation fn

(vi) Leaky activation function:- It is a improved version of Relu function as in Relu has small +ve slope in -ve area
Multiplied by 0.1 (not zero)
& Given as $g(x) = \begin{cases} 1, & x \geq 0 \\ 0.1x, & x < 0 \end{cases}$

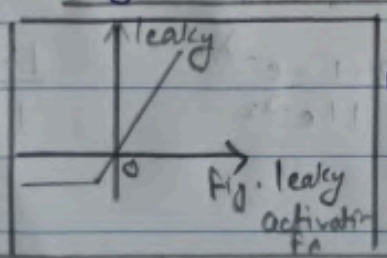


fig. leaky activation fn

$f(x) = (0.1, x)$

Q.5] Explain different loss function in detail.

→ ① Squared error loss :- widely used loss function in ML & statistic
- measured the average squared difference between predicted values & the actual target values.
- squared error loss or mean squared error loss is calculated by squaring the difference between the value by & the predicted value, we sum the no. to get a total value & then decide the no. y again.

Thus, $MSE = \frac{1}{n} \sum (y - \hat{y})^2$ ... this yields a +ve no.

② Cross - Entropy :-

(i) Binary cross entropy :- This loss function is used for binary classification tasks, It measures the dissimilarity b/w predicted probabilities & ~~true~~ ~~it measures the~~ binary labels.

given as :- $\sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i)]$ => binary class

class 1 class 2

(ii) Categorical cross entropy :- for multiple - class classification & Categorical cross entropy is used. It measures the dissimilarity b/w predicted class probabilities and one - hot encoded true labels

Given as :- $\sum_{i=1}^{k} y_i \log(\hat{y}_i)$ → for multi-class

③ Choosing o/p function & loss function

(i) When Numerical value as o/p, which you want to predict
- activation function :- linear activation function & Relu
- loss function :- MAE

(ii) When the o/p you trying to predict is binary.
- activation function - Sigmoid function
- loss function - Binary cross entropy function

(iii) When the o/p you trying to predict form multiple class.
- activation function :- Softmax function.
- loss function :- Categorical cross entropy loss function.

Q.6] Explain Gradient Descent, Stochastic GD, & Minibatch GD in detail

→ Gradient Descent is an optimization algorithm and it is used to train ML models & neural network.

Training the data helps these models learn over time and the cost function within gradient descent gauges it accuracy, with each iteration of parameter updates.

- The aim is Gradient Descent is to minimize the cost function or the error the blw predicted & actual value.
- To do this, we require two data points :- a direction & a learning rate.

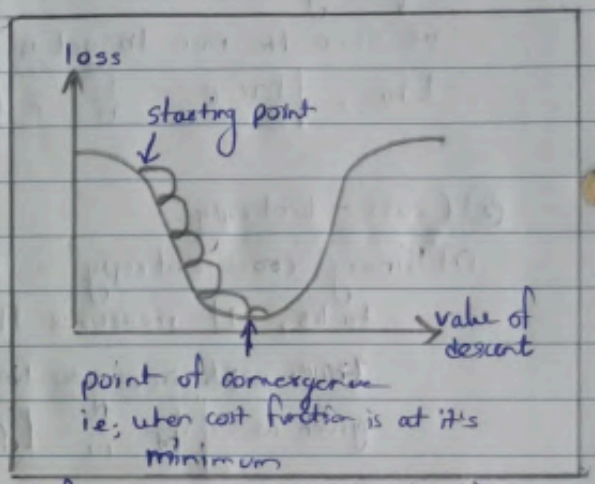


fig. Gradient descent algorithm

learning rate : also called as step size or alpha. This is the size of the steps to reach minimum.

① large learning rate: Result in large steps but the risk involved is minimum its totally depend on the behaviour of cost function.

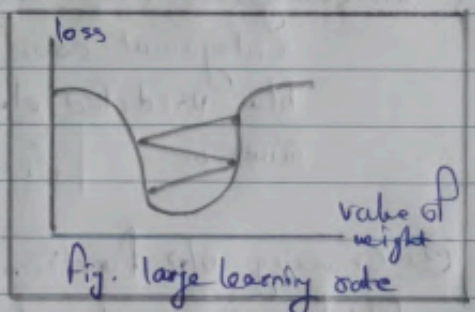


fig. large learning rate

②

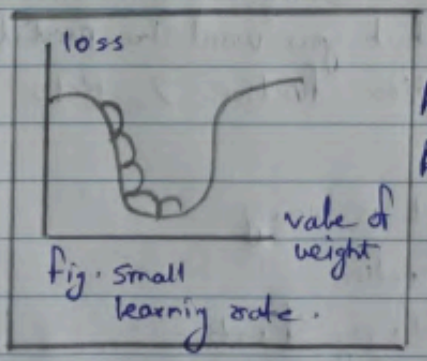


fig. Small learning rate.

A low learning rate has small step size. Advantage of this method is give more precision. since no. of iteration are more.

- Stochastic Gradient Descent:- SGD is an optimization algorithm commonly used in training ML models, including deep learning N/w.
- Unlike traditional GD which calculates the average gradient are

the entire dataset before updating the model's parameters. SGD updates the parameter after processing each individual data point [or a sampled subset called mini batch]. This introduce randomness in updates, which help escape local minima & make training process faster.

- Mini-Batch Gradient Descent:- Mini-Batch GD is a variation of the stochastic Gradient Descent [SGD] optimization algorithm used in training deep learning models.
  - Instead of updating the models parameter after processing each individual data point [as in SGD], mini-batch GD update the parameters after processing a small batch of data points [typically ranging from tens to hundred]
  - this approach strikes a balance between the efficiency of batch Gradient descent [using the entire dataset] and the noise introduced by the stochastic gradient descent [SGD].

Q.7] Write a short note on :-

→ (i) Momentum based GD :- Momentum based Gradient descent is an optimization algorithm used to train different models.
- The training process consist of an objective function [or error function], which finds out error a ML model has an given dataset.
- The momentum term is set to a value between 0 & 1 with a higher value resulting in a more stable optimization process.
- Advantage :- Momentum is faster than SGD and the training will be faster than SGD.

(ii) Nesterov Accelerated GD :- NAG is a modified version of momentum with stronger theoritical convergence and Guarantee for convex function
- In the standard momentum method, the Gradient is computed using current parameters (θ_t)

- Momentum Vs. Nesterov.

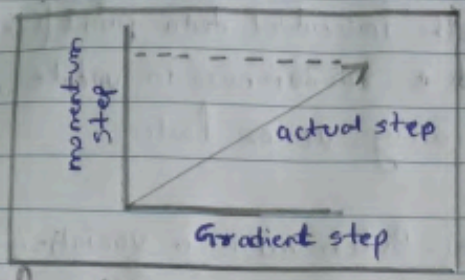


fig. a) Momentum update

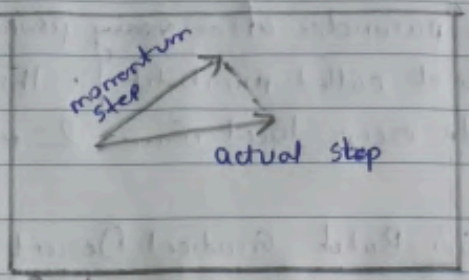


fig. b) Nesterov momentum update

- Nesterov Momentum achieve stronger convergence by applying the velocity [vt] to the parameters in order to compute interm parameters [$\theta = \theta t + \mu * vt$], where μ is the decay rate. Advantage of NAG is that it allow large decay rated.

(iii) **AdaGrad** :- Adagrad stands for 'Adaptive Gradient Optimizer'. The optimizers like Gradient Descent, stochastic GD, mini-batch SGD are used to reduce the loss function with respect to the weights.

- The weights update formula is given by,

$$(w)_{new} = w_{(old)} - \eta \frac{\partial L}{\partial w(old)}$$

--- η = learning rate
--- w(t) = value of w at current iteration

for iteration, the formula, $w_t = w_{t-1} - \eta \frac{\partial L}{\partial w(t-1)}$ --- w(t-1) at previous iteration

- advantage :- Fastest convergence & more reliable.

(iv) **Adam** :- The name 'Adam' is derived from 'adaptive moment estimation'. This optimization algorithm is an extension of SGD to update N/w weights during training.

- Benefits of using Adam on non-convex optimization problem are:
  - Straight forward to implement
  - computationally efficient.
- Adam is known for its fastest convergence & ability to work well on noisy and sparse dataset.

- configuration parameters for Adam required are learning rate, weight decay, Batch size and max epochs to be focused.

(v) RMS Prop :- 'Root Mean Square Propagation'. It is an adaptive learning rate optimization algorithm. Extension of popular Adaptive Gradient algorithm, and is designed to dramatically reduce the amount of computational effort used in training neural network.

- In RMS Prop, each update is done according to equation described below. This update is done separately for each parameters.

For each parameter w^j [we drop superscript j for clarity]

$$v_t = \rho v_{t-1} + (1-\rho) * g_t^2 \quad \text{--- eqn (1)}$$

$$\Delta w_t = \frac{-\eta}{\sqrt{v_t + \epsilon}} \cdot g_t \quad \text{--- eqn (2)}$$

$$w_{t+1} = w_t + \Delta w_t \quad \text{--- eqn (3)}$$

- η - initial learning rate
- v_ϵ → Exponential avg of square of gradients
- g_t - Gradient at time t along w_j

(7.8] Explain the terms:

→ (i) Parameter Sharing :- [weight sharing] - makes set of parameters to be similar as we interpret various models or model components as sharing a unique set of parameters. We only store a subset of memory. Eg:- Natural image having specific statistical properties that are robust for translation.

(ii) Weight decay :- Weight decay is a regularization techniques that is used in ML to reduce the complexity of a model & prevent overfitting.

The new loss function using weight decay technique is:-

$$L(w,b) + \lambda/2 \, \|w\|^2 .$$

Here $L(w,b)$ → represents the original loss function before adding regularization L_2 norm [weight decay] term.

(iii) Dropout :- The typical characteristic of CNN is dropout layer. The dropout layer is "mask that nullifies the contribution of some neurons towards the next layer and layer leaves unmodified all others.

(iv) Early stopping :- In ML, "early stopping" is a form of regularization used to avoid overfitting while training a learner with an iterative method, such as gradient descent.

(vi) Data Augmentation :- Data Augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data

techniques :-
- Audio data augmentation
- Text data augmentation
- Image augmentation

(vii) Bias - Variance Tradeoff :- The bias - variance tradeoff is a fundamental concept in ML that to the balance b/w two sources of error that affect the performance of predictive models: bias & variance. Finding the right balance between these two source of error is crucial for building a model that generalizes well to unseen data.

Q[29] What is Regularization? How does regularization help in reducing overfitting?

→ Regularization is a set of technique used in ML to prevent overfitting, which occurs when a model learns to fit the training data too closely, capturing noise & making it perform poorly on unseen data.

To Reduce Overfitting :-

(i) Penalizing complexity :- Regularization methods penalize complex models by adding a cost to the loss function based on the complexity of models. This discourage the momentum model from fitting noise in the data.

(ii) Constraining parameters values :- Some regularization techniques, like weight decay [L2 regularization], restrict the magnitude of parameter values, preventing them from becoming too large. This encourage the model to have similar smaller more manageable weights.

(iii) Encouraging Simplicity :- Techniques like dropout and early stopping force the model to learn more robust and similar representation by randomly dropping neurons or stopping training when performance on validation data degrades.

By imposing these constraints or penalizes, regularization techniques bias the model toward similar solution that generalizes better to unseen data effectively reducing overfitting.

Q.10] Explain Autoencoders. Justify the advantage of autoencoder over pircipal component analysis for dimensionality reduction.

→ Autoencoder are neural Nlw used for unsupervised and dimensionally reduction. They consist of an encoder & a decoder. The encoder wraps input data to a lower-dimensional representation [encoding], and the decoder reconstructs the orginal data from the encoding.

- Architecture :-

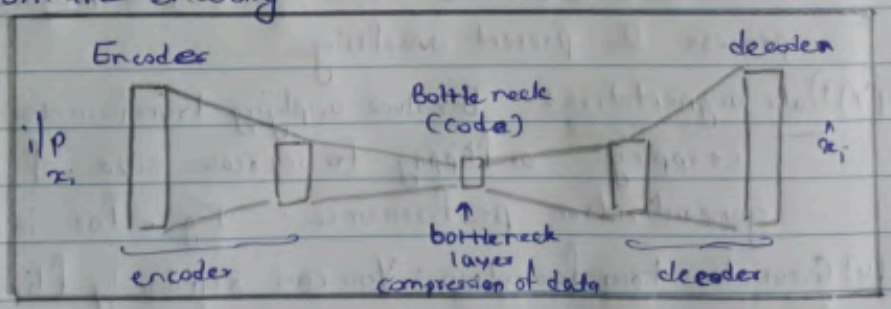


fig. autoencoders

Advantage :- autoencoder over PCA

(i) Non-linearity :- Autoencoder can capture complex, non-linear relationship in the data, where PCA is inherently linear.

(ii) Representational learning :- Autoencoders learns meaningful features directly from the data, whereas PCA relies on orthogonal transformation.

(iii) Flexibility :- Autoencoders can handles various types of data, including images, text and sequences, while PCA is primarily designed for numerical data.

(iv) Robustness:- Autoencoders can handle noisy data & perform well when data has missing values, which can be challenging for PCA

Autoencoders are typically more computationally intensive & require large amounts of data for training compared to PCA. The choice b/w methods depends upon problem

Q.11] Explain regularization in autoencoders & it's techniques in detail.

→ Regularization in autoencoders is used to prevent overfitting, just like in other neural N/w. Overfitting in autoencoders can lead to reconstruction that are overly reliant on noise or small variation in training data.

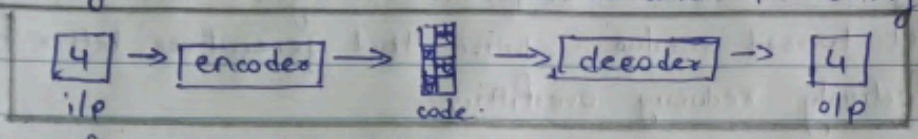


fig. regularization autoencoder.

- Types of regularization technique that can be applied to autoencoders:-

(i) L1 & L2 regularization :- add penalty term to loss function, that encoding model to have smaller weights.

L1 norm : $|a| = \text{sqrt}(a_1^2 + a_2^2 + \cdots a_n^2)$

(ii) Drop out :- technique that randomly drops out neurons during training to prevent overfitting

(iii) Batch normalization :- Technique that normalizes the i/p to each layer to have zero mean & unit variance. This can help to improve stability of training process & prevent overfitting.

(iv) Data augmentation :- Involves applying transformation to i/p data such as cropping or flipping to increase size of training set and improve generalization performance. Eg :- for images.

(v) Geometric transformation : You can randomly flip, crop, rotate or translate images, & this is just the tip of the iceberg.

(vi) Color of space transformation :- change RGB color channels, intensify any color.

(vii) Kernel filters :- sharpen or blur an image

Regularization techniques can be effective way to improve performance of autoencoders & prevent overfitting. The choice of regularization technique will depend on specific problem & data set being considered.