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	**PRATHAMESH S. CHIKANKAR**
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	**PRATHAMESH S. CHIKANKAR**
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	**PRATHAMESH S. CHIKANKAR**
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

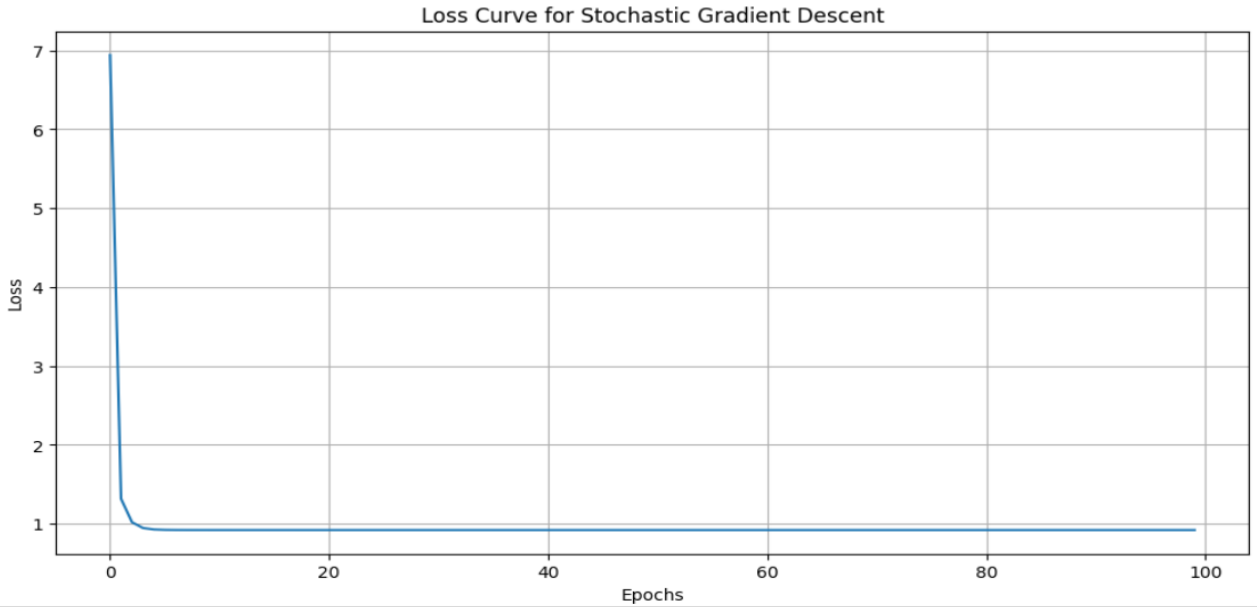
```
[5] def momentum_gd(X, y, W, b, learning_rate, beta, num_epochs):
      velocity_W, velocity_b = 0, 0
      for _ in range(num_epochs):
        for xi, yi in zip(X, y):
          gradient_W, gradient_b = compute_gradient(xi, yi, W, b)
          velocity_W = beta * velocity_W + (1 - beta) * gradient_W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

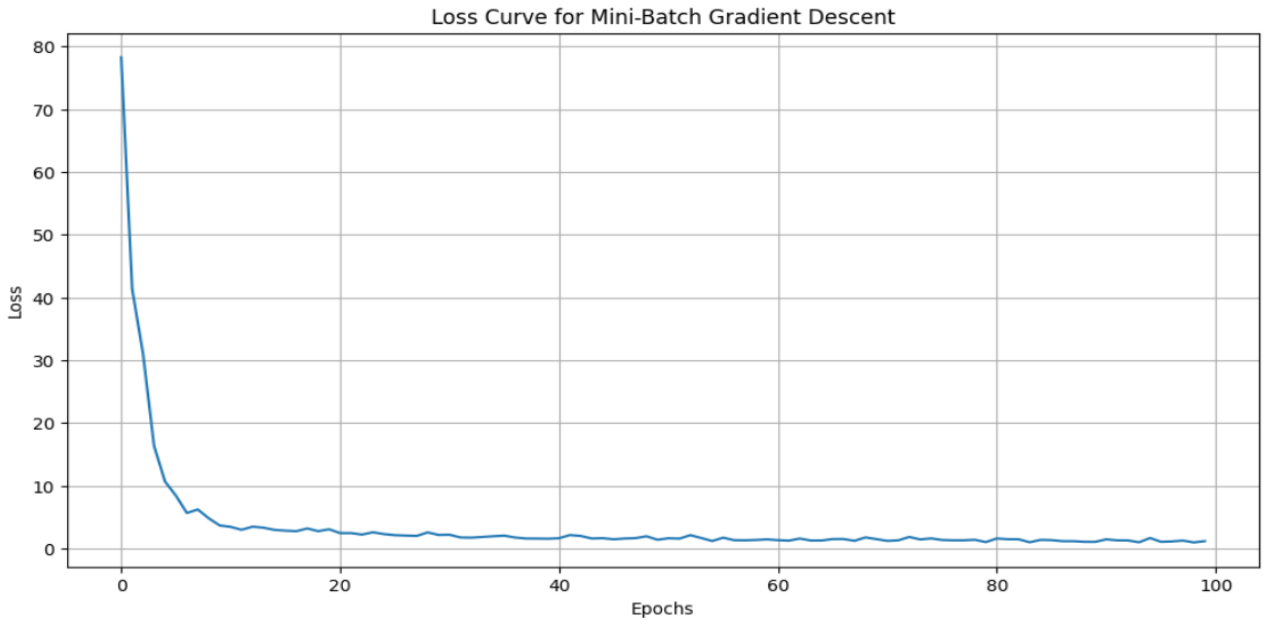
```
Learned Weights (W): [1.28056599 2.51443097 3.64171733]
Learned Bias (b): [2.81177659]
```



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()
```

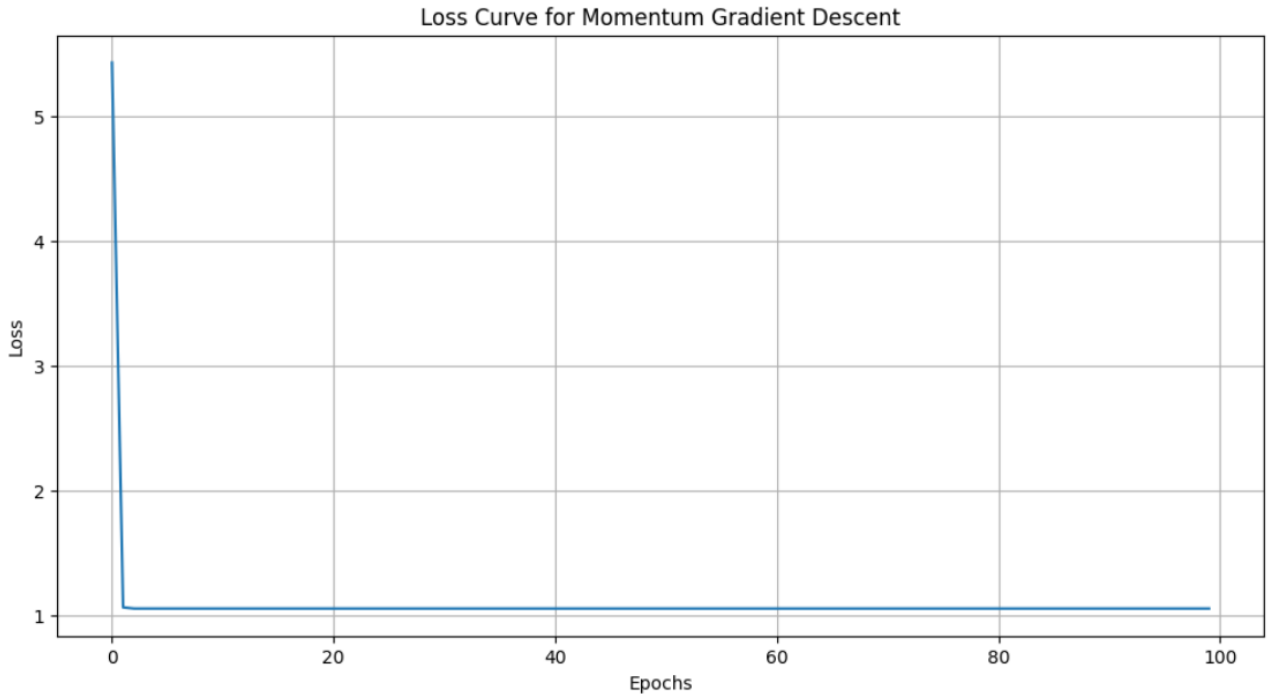
```
Learned Weights (W): [1.466288   2.66537392 3.34412419]
Learned Bias (b): [2.89975528]
```



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	**PRATHAMESH S. CHIKANKAR**
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	**PRATHAMESH S. CHIKANKAR**
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
```

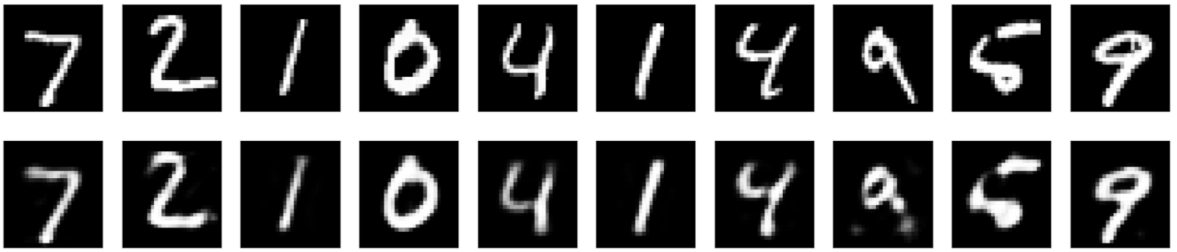
```
Epoch 47/50
235/235 [==============================] - 3s 13ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 48/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 49/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val_loss: 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	**PRATHAMESH S. CHIKANKAR**
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
```

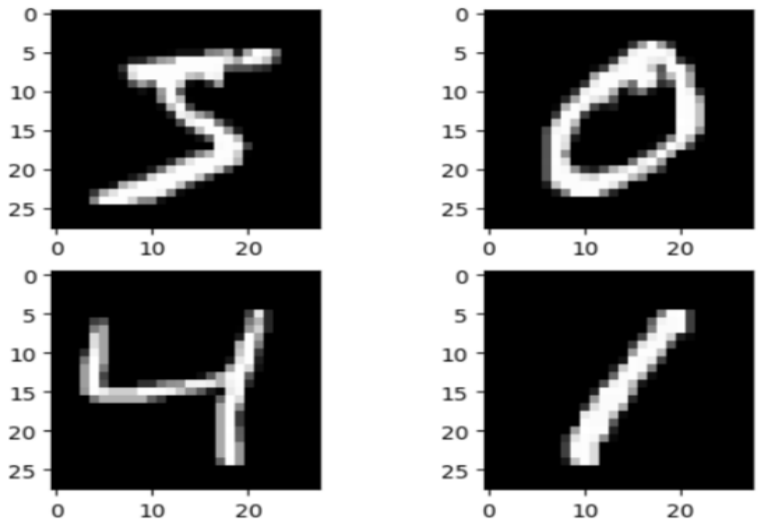
```
(60000, 28, 28)
```

```
[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

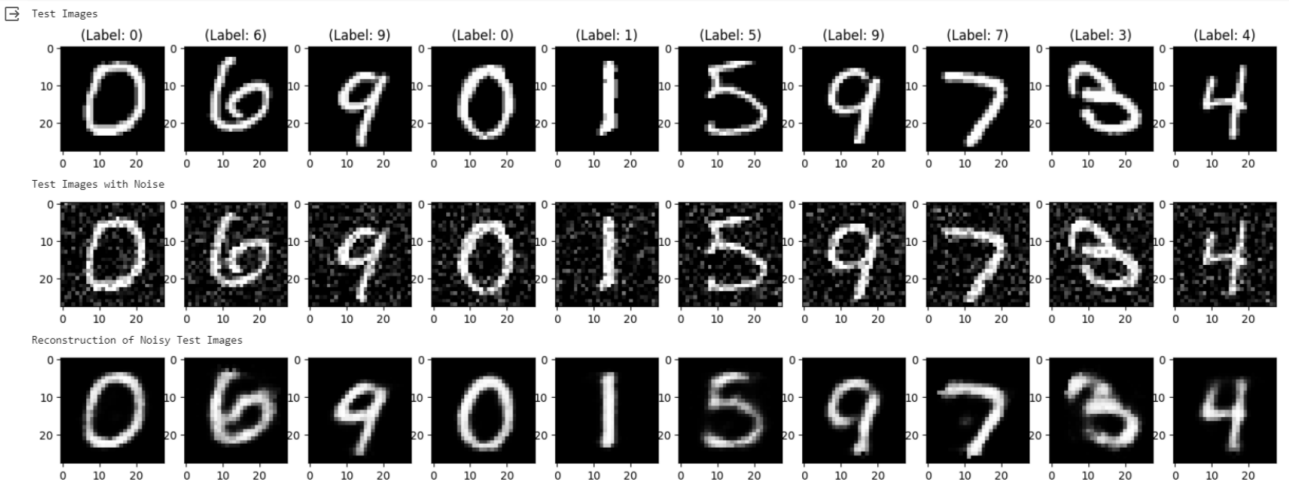
```
[13] # Final evaluation of the model
     pred = model.predict(x_test_noisy)
     pred.shape
```

```
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	**PRATHAMESH S. CHIKANKAR**
BRANCH	**CSE-(AI&ML)**
ROLL NO.	
SUBJECT	**DEEP LEARNING LAB**
COURSE CODE	**CSL701**
PRACTICAL NO.	
DOP	
DOS	

Output:

Import TensorFlow

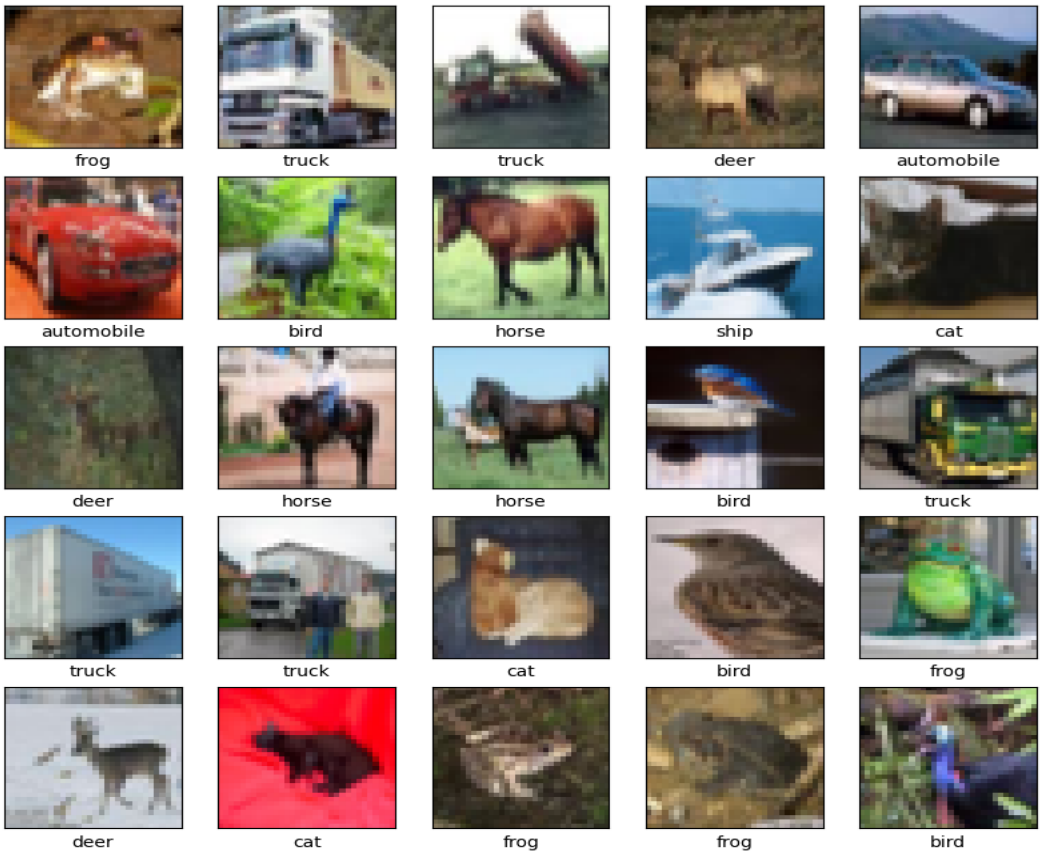
```
[2] import tensorflow as tf
    from tensorflow.keras import datasets, layers, models
    import matplotlib.pyplot as plt
```

```
[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

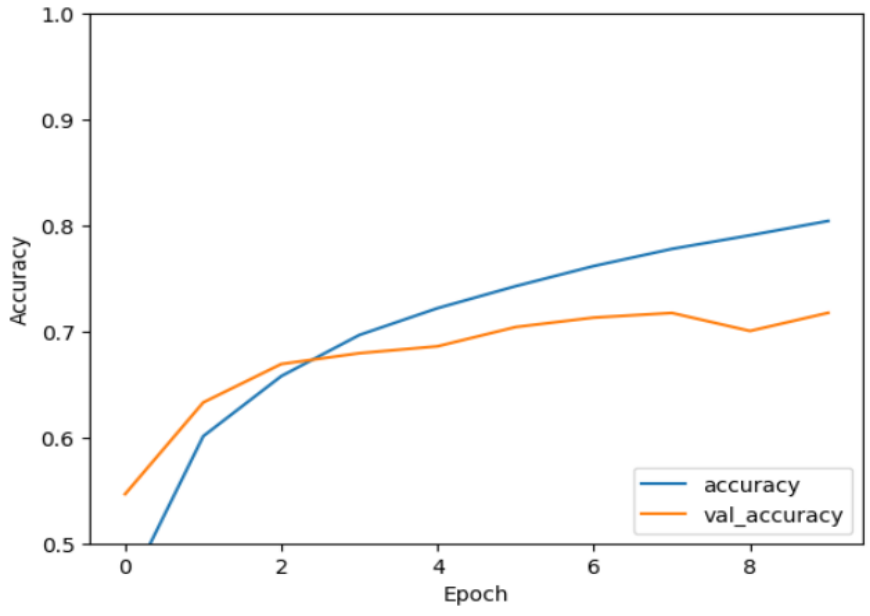
```
[9] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    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	**PRATHAMESH S. CHIKANKAR**
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

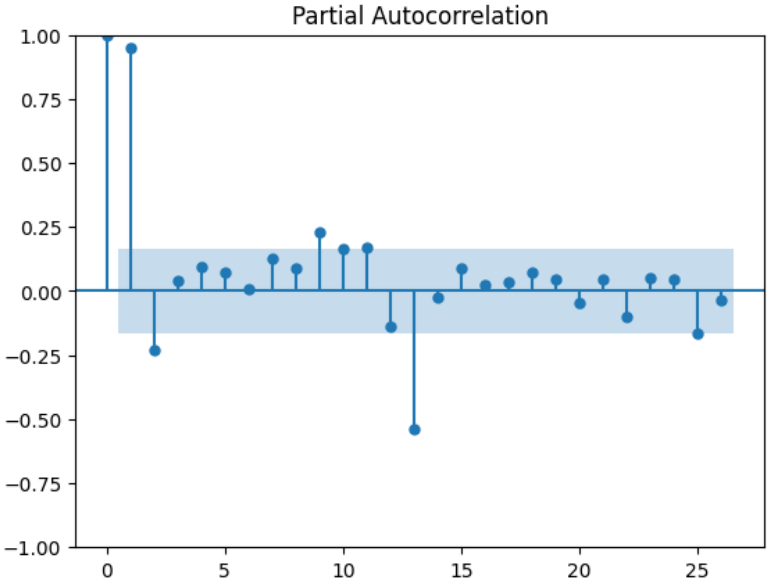
```
[11] import pandas as pd
     import numpy as np
     import pickle
     import seaborn as sns
     import matplotlib.pyplot as plt
     from scalecast.Forecaster import Forecaster

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

Exploratory Data Analysis

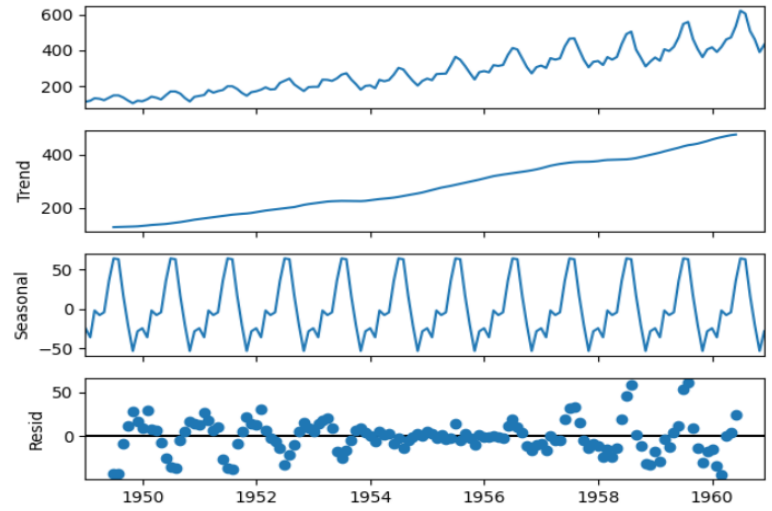
```
[12] f = Forecaster(y=df['#Passengers'],current_dates=df['Month'])
     f
```

```
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
```

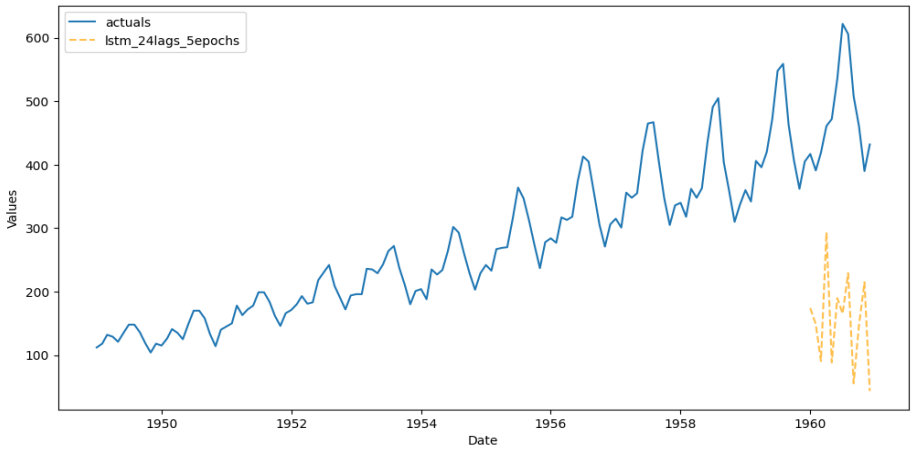
```
[16] pval

     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
```

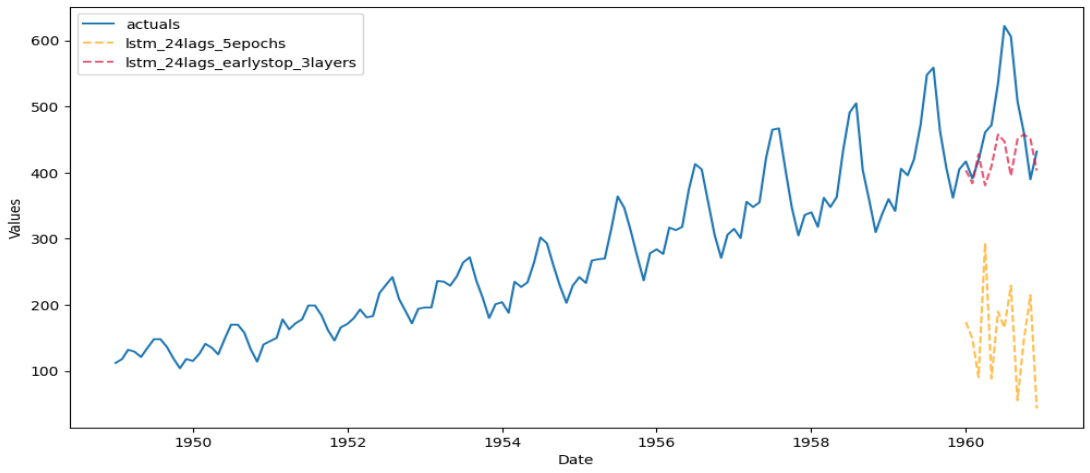
```
[18] f.manual_forecast(
         call_me='lstm_24lags_5epochs',
         lags=24,
         epochs=5,
         validation_split=.2,
         shuffle=True,
     )
     f.plot_test_set(ci=True)
```

```
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
```



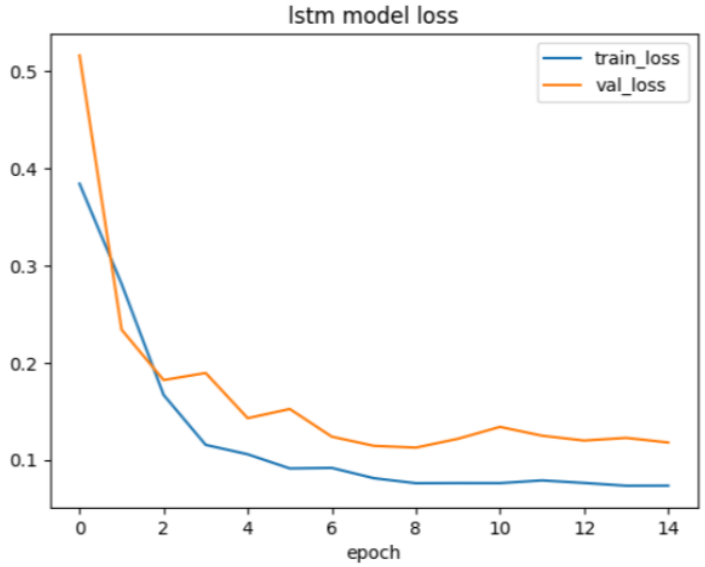
```
[19] from tensorflow.keras.callbacks import EarlyStopping

     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'>
```

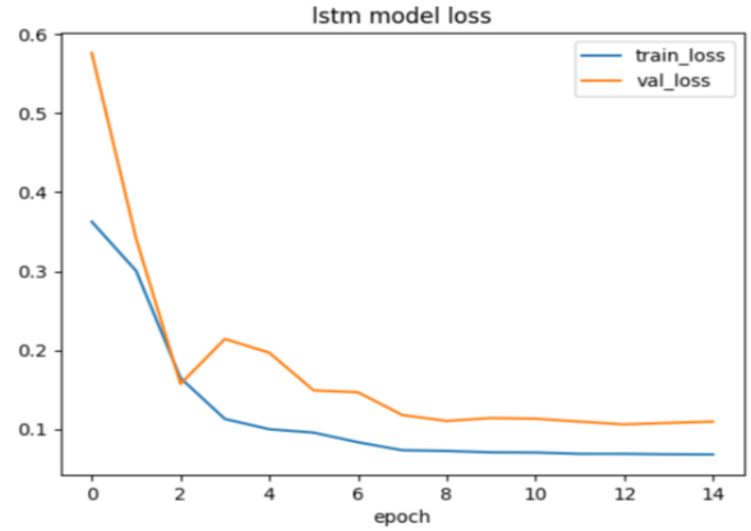



```
[21] f.manual_forecast(
         call_me='lstm_best',
         lags=36,
         batch_size=32,
         epochs=15,
         validation_split=.2,
         shuffle=True,
         activation='tanh',
         optimizer='Adam',
         learning_rate=0.001,
         lstm_layer_sizes=(72,)*4,
       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

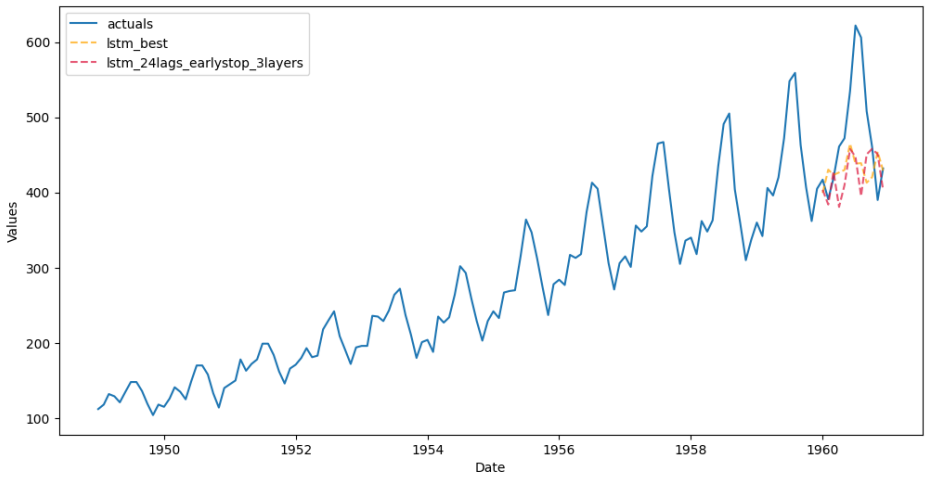
```
Epoch 1/15
3/3 [==============================] - 11s 1s/step - loss: 0.3839 - val_loss: 0.5157
Epoch 2/15
3/3 [==============================] - 0s 97ms/step - loss: 0.2809 - val_loss: 0.2339
Epoch 3/15
3/3 [==============================] - 0s 97ms/step - loss: 0.1668 - val_loss: 0.1821
Epoch 4/15
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
```



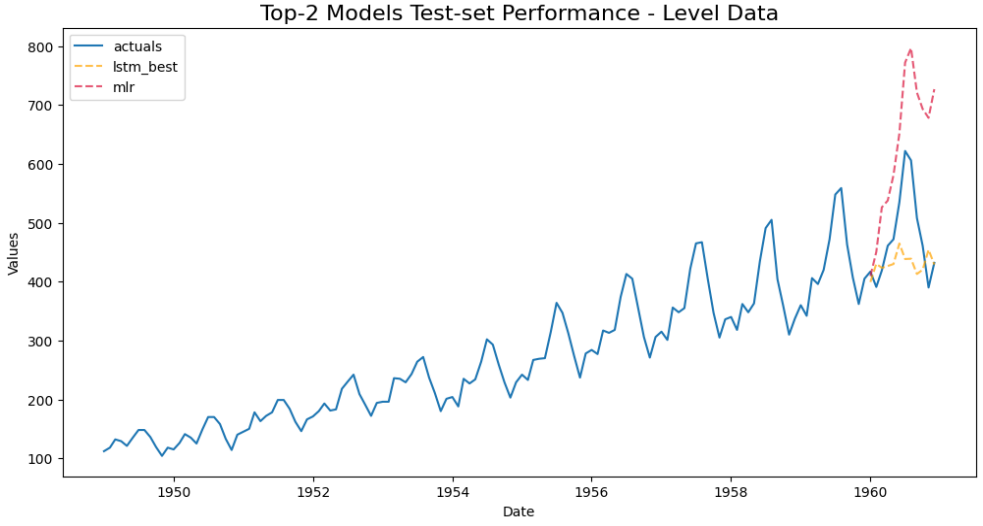

```
/usr/local/lib/python3.10/dist-packages/scalecast/_utils.py:60: Warning: Confidence inte
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/scalecast/_utils.py:60: Warning: Confidence inte
  warnings.warn(
<Axes: xlabel='Date', ylabel='Values'>
```



MLR Forecasting and Model Benchmarking

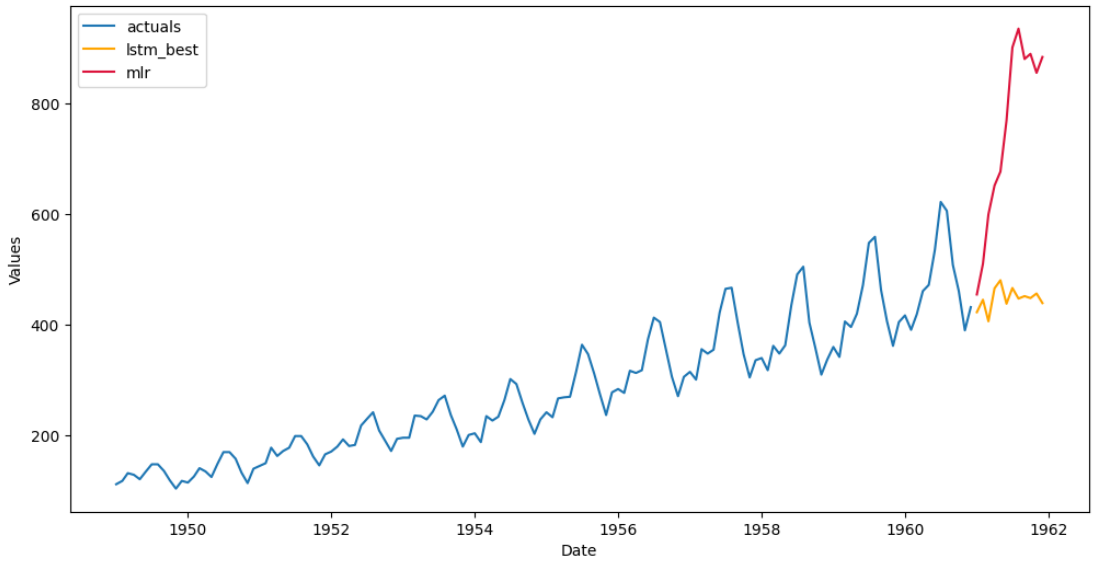
[45]
```
from scalecast.SeriesTransformer import SeriesTransformer

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')
```

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




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

	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 :- Prathamesh S. Chikankar.

Roll No. :- AIML11 Branch :- CSE-(AI&ML)

Year :- BE Subject :- Deep learning [DL]

Topic :- Assignment No. 01

Date :- August '23 Sign :- Prathm

1. What is Deep Learning?

> Deep Learning is a subset of machine Learning that focuses on artificial Neural Networks with multiple layers, also known as Deep Neural Networks.
These Networks are designed to automatically learn and extract hierarchical features from data.

> Explain three different classes of deep Networks.

- (*) feedforward Neural N/w (FFNN):
1) A FFNN is an artificial Neural Network where in connectn blw the Nodes and do not form a cycle.
2) As such it is different from its descendent: recurrent neural N/w. The feed forward neural N/w was the first & simplest type of neural N/w devised.
3) In an artificial feed forward Network (FNN), Infomatn always move in one direction, ie. it never goes backwards.

- (*) Convolution Neural N/w:
1) A CNNs (convolutn Neural N/w) is a type of ANN used in major recognitn. CNNs are a category of Neural N/w that have proven very effective in areas such as image recognitn and classificatn.
2) CNNs are important tools for ML practitioners today. CNN is a deep learning neural N/w designed for processing structured arrays of data such as images.

- (*) Recurrent Neural N/w (RNN):
1) RNN neural N/w is a class of artificial Neural N/w where connectns blw nodes form a directed graph along a temporal sequence.

PTO

This makes it to show temporal dynamic behaviour.

2) 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.

---

2. Explain Multilayer perceptrons (MLPs) and representation power of MLPs.

> 1) Multilayer perceptron is connected dense layers, and that transforms any i/p dimensions to desired dimensions.
A MLP is a Neural N/w that has multiple layers.
MLP are sometimes referred as "Vanilla" Neural N/w.

> 2)

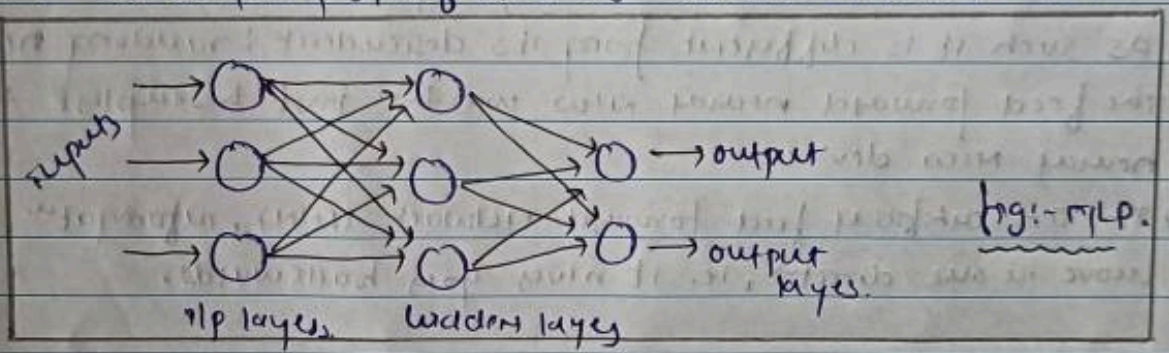


fig:- MLP.

- In above diagram of MLP, there are three i/p & they three i/p nodes and the hidden layer has three nodes.
- The o/p layer gives two o/p, hence there are two o/p nodes. The nodes in the i/p layer take i/p & forward it for further process.
- The hidden layer processes the i/p. & passes it to the o/p layer.

> 3) Representation power: (Activation fn) —
- 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 functions are both sigmoids, and are given as:

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

PTO

- The first is a 'hyperbolic tangent' that ranges from -1 to 1. While the other is 'logistic function' which is similar in shape but ranges from 0 to 1.
- Here y_i is the o/p of i^{th} node & x_i is the weighted sum of i/p connection.
- Alternate activation function has also been mentioned, including the Rectified and softplus function.

---

3. Explain the working of multilayer feedforward neural N/W & backpropogation in detail.

★ Multilayer feed forward N/W:

1) A multilayer feed forward N/W (also known as a feedforward neural N/W or FNN) works by passing data through multiple layers of neurons 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 layers, each consisting of multiple neurons. Each neuron computes a weighted sum of its i/p, applies an activation function (eg:- sigmoid or ReLU) and passes the result to the next layer.
- 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 and biases of neurons are initialized randomly, and the N/W o/p is compared to the true target value using a loss function (eg:- mean squared error for regression or cross-entropy for classification).

PTO

> * Back propogatn:

- 1) Back propogatn is used to adjust the weights & biases to minimize the loss.
- 2) working:
  - forward pass: the i/p data is passed through the NN, and predictions are obtained.
  - Loss computatn: the loss is calculated by comparing predictn to the true labels.
  - Backward pass (Backpropogatn): Errors are propogated backward through the NN. The Gradient of the loss with respect to the weights and biases are computed using the chain rule of calculus.
  - Weight updatn: the weight & biases are updated in the opposite directn of the gradients to minimize the loss. This is typically done using optimizatn algorithms like gradient descent.
  - The process is repeated iteratively until the model convergence to the set of weights & biases that produce accurate predictn.

4. Explain different activatn functn in detail.

1) tan h activatn: Used for neural NNs.

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

the tanh fn became preferred over the sigmoid fn as it gives better performance for multi-layer NNs.

- But it doesn't solve vanishing-gradient problem.

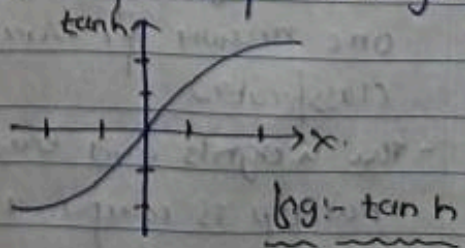


fig:- tan h

2) Logistic activatn fn: Also known as sigmoid fn.

> This fn is used in back-propagation nets and are of two types.

- ① Binary sigmoid fn:
  Also known as unipolar sigmoid fn or logistic sigmoid fn.
  ↳ defined as $f(x) = \frac{1}{1+e^{-\lambda x}}$

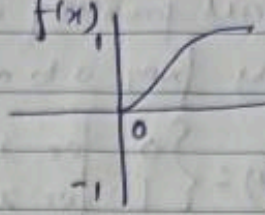
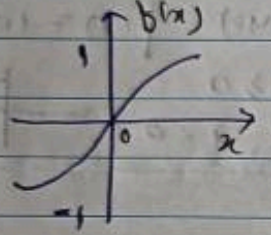


fig. Binary sigmoid fn.

- Range from 0 to 1.
  ∴ λ → steepness parameter.

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



Sigmoid fn are so called as their graph are S-shaped.
Sigmoid fn defined to be odd, are monotonic fn of the activation variable.

★ 3) Linear activation fn:
defined as, $f(x) = x$ for all x.
here i/p = o/p.

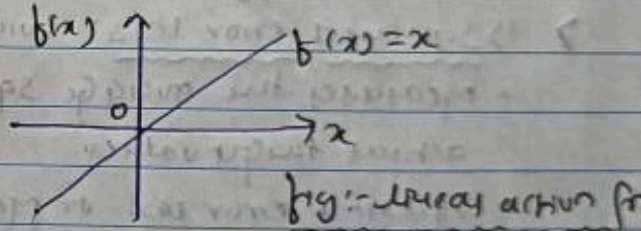


fig:- Linear activation fn.

★ 4) Softmax activation fn:
- also known as softargmax or normalized exponential fn. It converts a vector of K real no. into a probability distribution of K possible outcomes.
- It's a generalization of logistic fn; formula is,
  ∴ σ = softmax ; $\bar{z}$ = i/p vector $\quad \sigma = (\bar{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
  e^{z_i} = stand. exponential fn for i/p vector.
  K = no. of classes in multi-class classifier
  e^{z_j} = standard exponential fn for o/p vector

5) ReLU fn: Rectified linear unit.

Doesn't activate all the neurons at the same time and vanishing gradient problem doesn't occur in ReLU.
range from 0 to ∞, eqn is $f(x) = \begin{cases} 0 & \text{for } x<0 \\ x & \text{for } x \geq 0 \end{cases}$

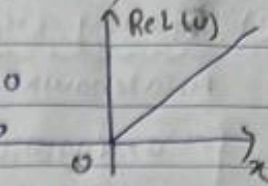


fig:- ReLU activation fn.

ex. $f(x) = \begin{cases} 0 & \text{for } x<0 \\ 1 & \text{for } x \geq 0 \end{cases}$

★ 6) Leaky activation fn:-

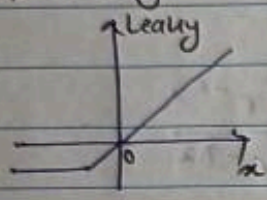


fig:- Leaky activation fn

- It is a improved version of ReLU fn, as in ReLU the negative slope is -ve area, multiplied by 0.1 (not zero) $f(x) = (0.1x, x)$
& given as $g(x) = 1, x \geq 0$
$= 0.1x, x < 0$

5. Explain different loss fn in detail.

> 1) Squared error loss: Widely used loss fn in ML and statistic.
- measures the average squared difference b/w predicted values & the actual target values.
- squared error loss or mean squared error loss is calculated by
- squaring the difference b/w the value by & the predicted value, we
- sum the no. to get a total value & then divide the no. y again.

thus, $MSE = \frac{1}{n} \Sigma (y - \hat{y})^2$ ... this yields a true no.

> 2) Cross-Entropy

- ① Binary cross Entropy: This loss fn is used for binary classification tasks, It measures the dissimilarity b/w predicted probability & the binary labels.
given as:- $\frac{1}{M} \sum_{i=1}^{M} \underbrace{(y_i \log(\hat{y}_i)}_{\text{class 1}} + \underbrace{(1-y_i) \log(1-\hat{y}_i)}_{\text{class 2}}$ ⇒ Binary class.

- ② categorical cross Entropy: for multiple-class classifn & categorical cross entropy is used. It measures the dissimilarity b/w predicted class

PTO

probability and one-hot encoded true labels.
given by : $-\sum_{i=1}^{n} y_i \log(\hat{y}_i)$ → for multi-class..

> 3) Choosing o/p fn and loss fn:

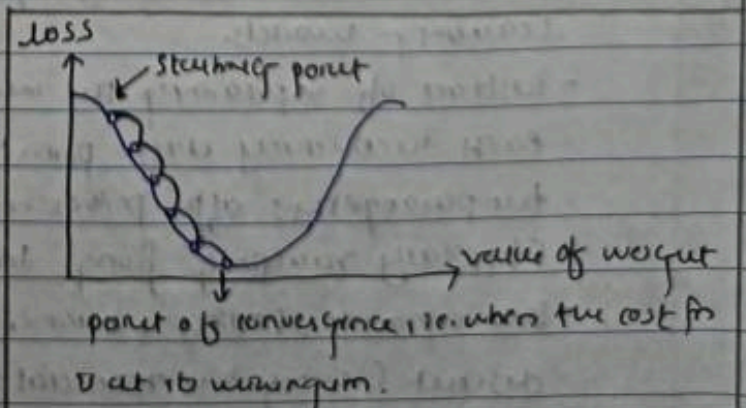
- ① when Numerical value as o/p, which you want to predict.
  - activn fn: linear activation fn. & ReLU
  - loss fn: MAE
- ② when the o/p you trying to predict is binary.
  - activation fn: sigmoid fn
  - loss fn: Binary cross entropy fn.
- ③ when the o/p you trying to predict for multiple class.
  - activation fn: Softmax fn
  - loss fn:- Categorical cross entropy loss fn.

---

6. Explain Gradient Descent, stochastic GD and mini batch GD in detail.

★ Gradient Descent is an optimization algorithm and it is used to train ML models and Neural NW.
Training the data helps these models learn over time and the cost fn with within gradient descent guages its accuracy, with each iteration of parameter updates.

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



point of convergence, i.e. when the cost fn is at its minimum.

fig:- Gradient Descent algorithm.

PTO

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

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

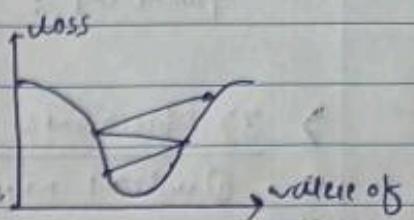


fig: large learning rate.

② a low learning rate may small step sizes. advantage of this method is give more precision. since no. of steps are more..

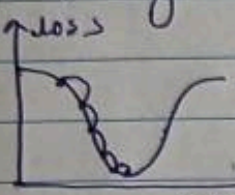


fig:- small learning rate.

* • Stochastic Gradient Descent :- SGD is an optimizer algorithm commonly used in training ML models, including deep learning NNs.
- unlike traditional GD which calculate the average gradient on the entire dataset before updating the model's parameters, SGD update the parameters after processing each individual data point (or a small subset called mini-batch). This introduce randomness into 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) optimizer algorithm used in training deep learning models.
- Instead of updating the models parameters after processing each individual data point (as in SGD), mini-batch GD updates the parameters after processing a small batch of data points (typically ranging from tens to hundreds)
- this approach strikes a balance b/w the efficiency of batch gradient descent (using the entire dataset) and the noise introduced by the stochastic gradient descent (SGD).

7. Write a short note on :

* Momentum based GD - Momentum based Gradient descent is an optimizer algorithm used to train different models.
- The training process consist of an objective fn (or error fn), which finds out error a ML model has on given dataset.
- The momentum term is set to a value b/w 0 and 1 with a higher value resulting in a more stable optimizer process.
- Advantage : momentum is faster than SGD and the training will be faster than SGD.

* Nesterov Accelerated GD - NAG is a modified version of momentum with stronger theoretical convergence and guarantees for convex fn.

- Momentum vs. Nesterov

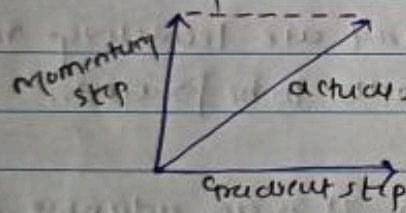


fig. a) Momentum update

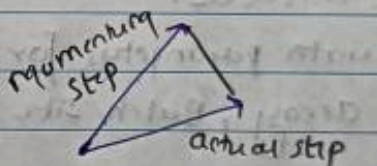


fig. b) Nesterov momentum update.

- In the standard momentum method, the gradient is computed using current parameters (θ_t).
- Nesterov momentum achieves stronger convergence by applying the velocity (v_t) to the parameters in order to compute interim parameters ($\theta = \theta_t + \mu * v_t$), where μ is the decay rate.
- Advantage of NAG is that it allows larger decay rates.

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

PTO

- The weights update formula is given by, ... η = learning rate.

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

... 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 : fastly convergence & more stable.

★ Adam : The name 'Adam' is derived from 'adaptive momentum estimation'. This optimization algorithm is an extension of SGD to update new weights during training.

- Benefits of using Adam on non-convex optimization problems are :
  > straightforward to implement. & > computationally efficient.
- Adam is known for its fastly convergence & ability to work well on noisy and sparse datasets.
- Configuration parameters for Adam required are learning rate, weight decay, Batch size and max epochs. to be formed..

★ RMS prop : Root mean square propagator. 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 NW.

- In RMS prop, each update is done according to eqn described below. This update is done separately for each parameter.

for each parameter wi (we drop superscript i for clarity)

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

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

... η → initial learning rate.

... v_t → Exponential avg of squares of gradients.

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

... g_t → gradient at time t along wi.

8. Explain the term:

★ Parameter sharing: (weight sharing) → refers 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 img having specific statistical properties that are robust for translation.

★ Weight decay: Weight decay is a regularization technique that is used in ML to reduce the complexity of a model & prevent overfitting. The new loss for using weight decay technique is:

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

here $L(w,b)$ represents the original loss for before adding regularization L_2, using (weight decay) term.

★ Dropout: The typical characteristic of CNN is dropout layers. The dropout layer is 'mask that nullify the contribution of some neurons towards the next layer and layer leaves unmodified all others.

★ 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.

★ Data Augmentation: Data Augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data.
techniques: 1) Audio data augmentation
2) Text data augmentation
3) Image augmentation...

PTO

★ Bias-variance tradeoff: The bias-variance tradeoff is a fundamental concept in ML that refers to the balance b/w two sources of error that affect the performance of a predictive models: bias & variance. -finding the right balance b/w these two sources of error is crucial for building a model that generalizes well to unseen data.

Q. What is Regularization?

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

> How does regularization help in reducing overfitting?

- 1) penalizing complexity: Regularization methods penalize complex models by adding a cost to the loss fn based on the complexity of model. This discourages the optimization model from fitting noise in the data.
- 2) Constraining parameters values: Some regularization techniques, like weight decay (L2 regularization), restrict the magnitude of parameter values, preventing them from becoming too large. This encourages the model to have smaller, more manageable weights.
- 3) Encouraging simplicity: Techniques like dropout and early stopping force the model to learn more robust and simpler representations by randomly dropping neurons or stopping training when performance on validation data degrades.

> By imposing these constraints or penalties, regularization techniques bring the model toward simpler solutions that generalize better to unseen data, effectively reducing overfitting.

10. Explain autoencoders. (Autoencoders)

> Autoencoders are neural N/ws used for unsupervised learning and dimensionality reduction. They consist of an encoder and a decoder. The encoder maps input data to a lower-dimensional representation (encoding), and the decoder reconstructs the original data from the encoding.

-Architecture:

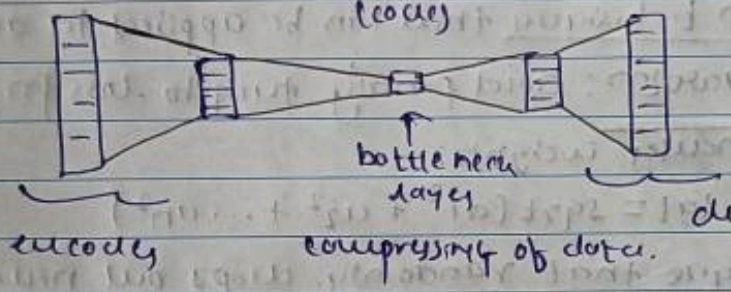


fig:- autoencoders..

> Justify the advantages of autoencoders over principle component analysis for dimensionality reduction..

- ① Non-linearity: Autoencoders can capture complex, non-linear relationship in the data, while PCA is inherently linear.
- ② Representational learning: Autoencoder learn meaningful features directly from the data, whereas PCA relies on orthogonal transformation.
- ③ flexibility: Autoencoders can handle various types of data, including images, text and sequences, while PCA is primarily designed for numerical data.
- ④ 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 two methods depends upon specific problem and data characteristics..

11. Explain regularization in autoencoders & its techniques in detail.

> Regularization in autoencoders is used to prevent overfitting, just like in other neural nets. overfitting in autoencoders can lead to reconstructions that are overly reliant on noise or specific values in training data.

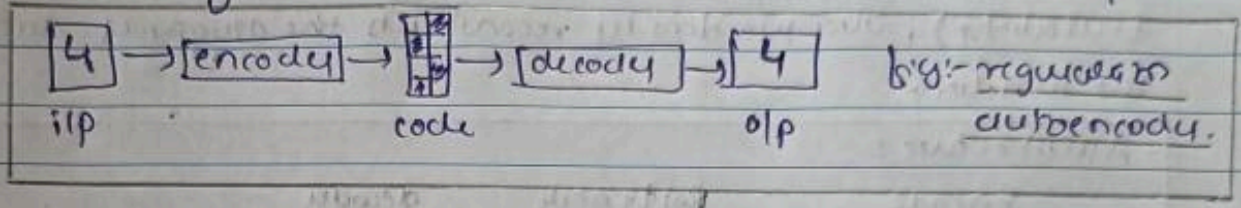


fig:- regularized autoencoder.

> Types of regularization techniques that can be applied to autoencoders:

- 1) L1 and L2 regularization: add penalty term to loss fn, that encourages model to have smaller weights.

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

- 2) Dropout: technique that randomly drops out neurons during training to prevent overfitting.
- 3) 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.
- 4) Data augmentation: Involves applying transformations to i/p data, such as cropping or flipping, to increase size of training set and improve generalization performance, Eg:- for images.
- 5) Geometric transformations: You can randomly flip, crop, rotate or translate images, & this is just the tip of the iceberg.
- 6) Color space transformations: change RGB color channels, intensify any color.
- 7) 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.

Name :- Prathamesh S. Chikankar

Roll No. :- AIML11 Branch :- (SE-(AI&ML)

Year :- BE Subject :- Deep learning (DL)

Topic :- Assignment No. 02

Sign :- Prathy Date :- October '23

11/10/23

1. Explain CNN architecture in detail.

A Convolution Neural Network (CNN), is a NN architecture for DL which learns directly from data. CNN are particularly used for finding patterns in images to recognize objects. They can also be quite effective for classifying non-image data such as audio, time series, and signal data.

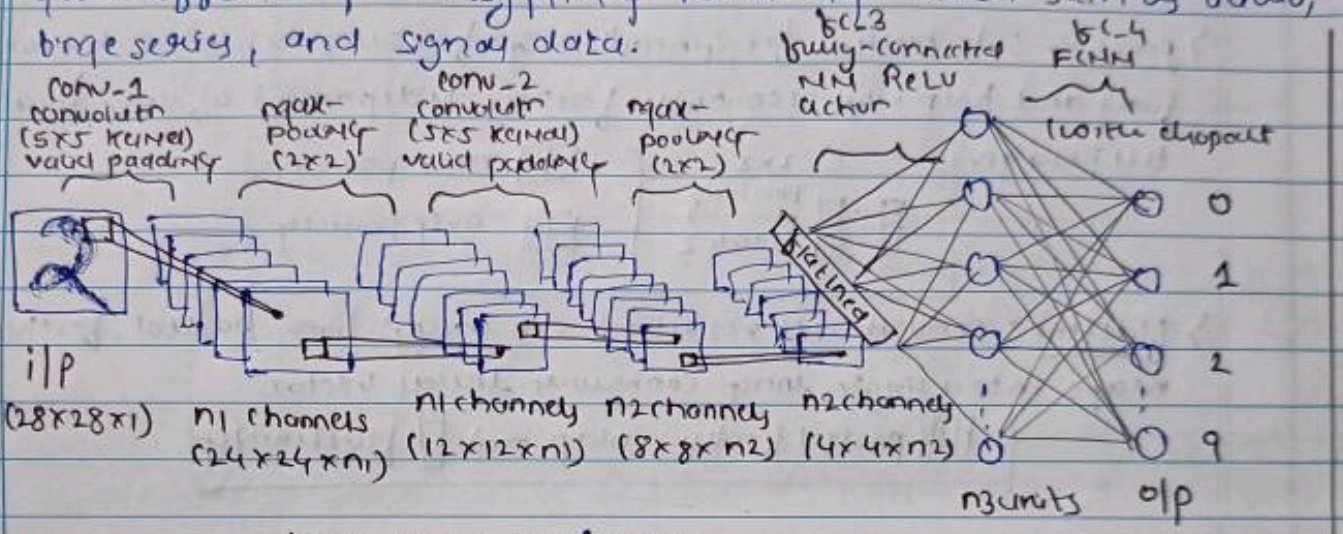


fig:- CNN architecture

1) Kernel or filter or feature Detectors: filter that is used to extract the features from the images. i → size of i/p
formula = [i-k]+1 ... k → size of kernel

i/p → image patch * kernel → o/p
(local receptive field) (filter)

2) stride: modify the amount of movement over the image or video. If stride is 1 then 1 by 1 & if stride is 2 it skips next 2 pixels.
formula = [i-k/s] + 1 ... s → stride

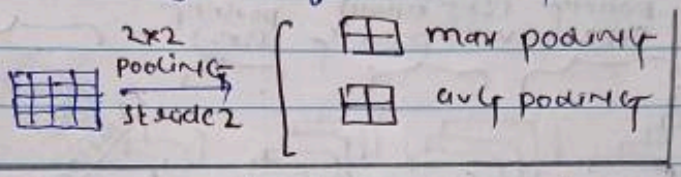
7x7 i/p volume → (stride 1) 5x5 o/p volume
7x7 i/p volume → (stride 2) 3x3 o/p volume

PTO

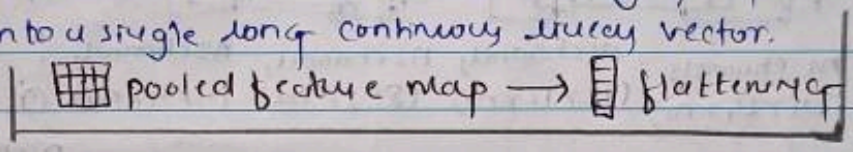
} padding: term relevant to CNN as it refers to the no. of pixels added to an image when it is being processed by kernel of CNN.
formula = [~~(~~i-k+2p/s]+1 ... p → padding

| image x [filter/kernel] = feature |

} pooling: technique for generalizing features extracted by convolutional filters and help NN to recognize features independent of their location in the image.



} flatten: convert all resultant 2-D arrays from pooled feature maps into a single long continuous linear vector.



} layers used to build CNN:
- convolutional layer
- pooling layer
- fully-connected layer (FC)

- convolution layer
$W - F + 2P/S + 1$

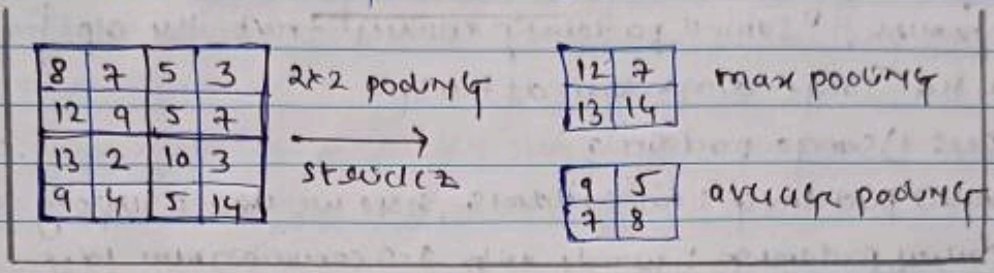
- pooling layer
$W - F/S + 1$

} dropout: Nullifies contribution of some neurons towards next layer and leaves unmodified all others.

} Activation fn: decides whether a neuron should be activated or not. This means that it will decide whether the neuron's i/p to N/w is important or not in process of prediction.
They are several commonly used activation fn such as ReLU, softmax, tanH, and the sigmoid fn.

2. Explain the terms w.r.t. CNN with example.

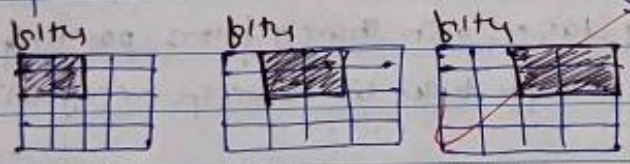
> Pooling: pooling in CNN is a technique for generalizing features extracted by convolutional filters and helping the NN to recognize features independent of their location in the image.



example: selects the maximum value from a group of numbers, if we apply 2×2 max pooling to a 4×4 feature map, o/p size becomes 2×2.

> Stride: Stride is a parameter of NN filters for modifying amount of movement over the image or video. We had stride 1 so it will take one by one. If we give stride 2 then it will take value by skipping the next 2 pixels.

formula = $[i - k(s)] + 1$ ... i → size of i/p, k → size of kernel, S → stride



left image: stride 0
middle image: stride 1
right image: stride 2

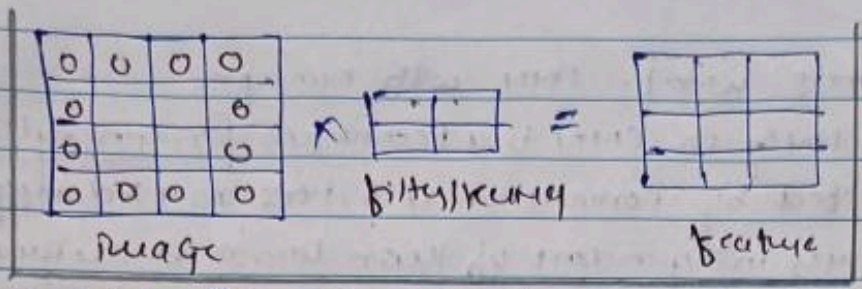
fig:- convolution (stride)

Size of o/p image calculated by,

$$\left[\frac{(n+2p-f+1)}{s}+1\right] * \left[\left(\frac{n+2p-f+1}{s}\right)+1\right]$$

> Padding: padding is a term relevant to CNN as it refers to the no. of pixels added to an image when it is being processed by the kernel of a CNN.

PTO



for example, "same" padding ensures that the o/p feature map has the same dimensions as the i/p.

Types: 1) same padding

2) valid padding: no padding, here we don't apply any padding

3) causal padding: mostly with 1-D convolutional layers.

---

3. Explain how fully connected N/W is different from CNN.

1) Data type:

FCN: for structured data, like spreadsheets or numerical data.

CNN: for grid-like data, especially images & spatially correlated i/p.

2) layer types:

FCN: Each neuron connects to all neurons in next layer. No shared weights

CNN: uses convolutional layers with shared filters, pooling layers, and fully connected layers for tasks like image recognition.

3) Weight sharing:

FCN: no weight sharing; every connection has its own weight.

CNN: uses weight sharing, allowing same filter to be applied at different positn in i/p, which is efficient for images.

4) Spatial Hierarchies:

FCN: doesn't capture spatial hierarchy or local patterns.

CNN: specially designed to detect local patterns & hierarchical features in grid-like data.

5) Translation invariance:

FCN: Lacks inherent translation invariance, treats similar features in

different locations differently.
CNN: inherently possess translation invariance, making it suitable for recognizing patterns regardless of their location in an image.

6) Use cases:
FCN: ideal good for structured data analysis & non-sparsity tasks like text classification.
CNN: ideal for image-related skills tasks, including image classification object detection & image segmentation.

4. Differentiate between

Aspect	LeNet	AlexNet
Year of introduction	1998	2012
Architecture depth	relatively shallow for its time	considered deep at the time of release
Convolution layers	2	5 (including some larger filters)
pooling layers	2 (Avg pooling)	3 (max pooling)
activation fn	Sigmoid	Rectified linear unit (ReLU)
local response normalization (LRN)	not used	used in specific layers to reduce overfitting
dropout	not used	used to prevent overfitting.
fully connected layers	3 (including the o/p layer)	2 (including the o/p layer)
Size of filters	smaller (eg.. 5x5 & 3x3)	larger (eg.. 11x11, 5x5, 3x3)
no. of filters	fewer (eg.. 6 and 16)	more (eg.. 96, 256 & 384)
use of GPU	limited GPU use (less common)	utilized GPUs extensively
winner of ImageNet	No	Yes (won the ImageNet competition.

5. Explain the architecture of ResNet in details with its advantages.

many Residual blocks are stacked together to form a ResNet. we have "skipped connection" which are the major part of ResNet. The following image provided denotes how residual NW works. The idea is to connect the i/p of a layer directly to the o/p layer after skipping a few connection.

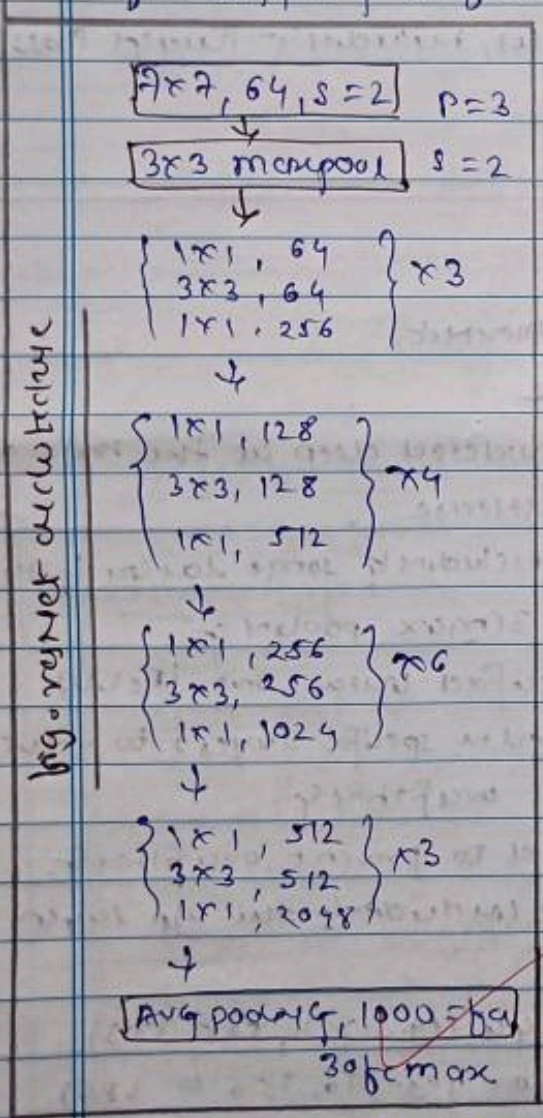
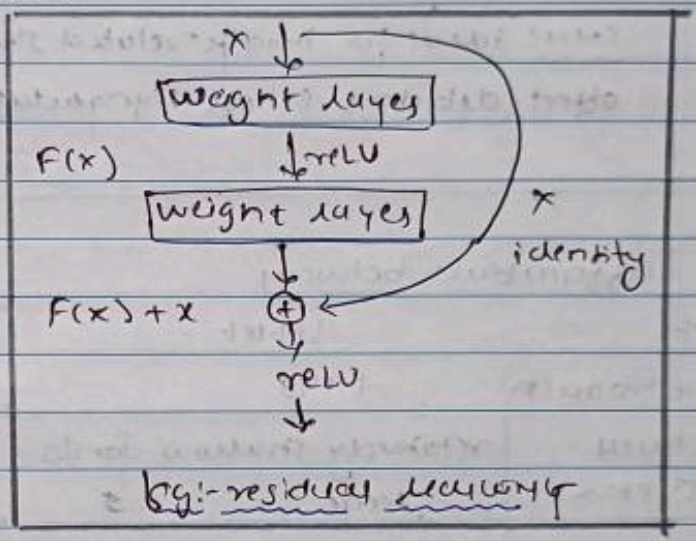


Fig: resnet architecture



- $a^{[l]} = x$
- $z^{[l+1]} = w^{[l+1]} a^{[l]} + b^{[l+1]}$
- $a^{[l+1]} = \text{relu}(z^{[l+1]})$
- $z^{[l+2]} = w^{[l+2]} a^{[l+1]} + b^{[l+2]}$
- $a^{[l+2]} = \text{ReLU}(w^{[l+2]} a^{[l+1]} + b^{[l+2]} + a^{[l]})$
  $= \text{ReLU}(z^{[l+2]} + a^{[l]})$

> we can see here x is the i/p to the layer which we are directly using to connect to layer after skipping two identity connection and if we think the o/p from identity connection to be f(x). then we can say the o/p will be F(x)+x.

- input : n×n
- padding : p
- filter size : f×f
- output : ((n+2p-f)/s+1) × ((n+2p-f)/s+1)

advantages :
- each block augments the data (additional data for learning)
- facilitate training of very deep NW w/o vanishing gradient problem.
- shorter gradient path (maintain)
- achieve state of art performance on various CV tasks.
- modularity
- reduce the need of extremely large datasets.

8. Difference bw

Aspect	Recurrent Neural NW (RNN)	Feed forward Neural NW (FNN)
architecture	Recurrent connection, allowing information to circulate through the NW over time.	No recurrent connection; information flows one way from I/p to o/p.
Data type	Designed for sequential data, such as time series, text, or speech	Typically used for structured data like tabular form, numerical data, & non-sequential data.
Layer type	Consist of recurrent layers, such as Simple RNN, LSTM, or GRU	composed of i/p hidden & o/p layers, with no recurrent connection.
Hidden State	Maintain hidden state that store info. from previous time steps.	No hidden state; each i/p is processed independently.
Temporal processing	Handles sequences by considering the order and temporal dependency in data.	Does not inherently capture temporal information.

PTO

o/p length flexibility	Can handle i/p sequence o/s varying lengths & produce o/p at each time step	Typically produces a fixed-length o/p for a given i/p.
use cases	Time series predict, NLP, Speech recognition, etc..	classification, regression and structured data analysis where sequence processing is not required
vanishing gradient	prone to vanishing gradient problems when processing long sequences.	Less prone to vanishing gradient issues.
Training method	Trained using Backpropogation through time (BPTT) for sequence learning	Training using standard backpropogation.
Example appln	Language translation (eg: with seq2seq models, text Generation sentiment analysis	Image classification, recommendation systems, financial modeling.

9. Explain RNN architecture in detail.

A recurrent neural n/w is a type of deep learning neural net that remembers the i/p sequence, stores it in memory states/cell states, and predicts the future word/sequence. sentences.

Lets consider x11, x12, x13 as i/p & o1, o2, o3 as o/p of hidden layers 1, 2 and 3 respectively. The i/p are sent to the NN at different time intervals, so lets say x11 is sent to the hidden layer 1 at time t1, x12@t2, x13@t3.

Also, lets assume weights are same in forward propogation.

The o/p o3 is dependent on o2 which in turn is dependent on o1.

$o1 = f(x11 * w)$ - where w is weight and f is activation fn.

$o2 = f(o1 + x12 * w)$; $o3 = f(o2 + x13 * w)$

PTO

finally, the o/p of O_3 is the actual o/p indicated by $\hat{y}$.

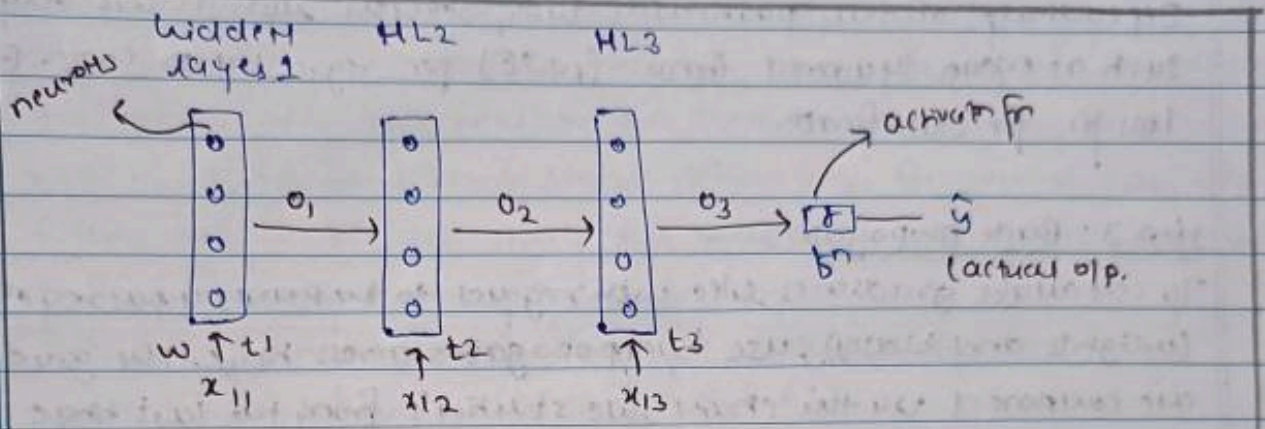


fig:- architecture & working of simple RNN

The predicted value $\hat{y}$ is nothing but O_3

i.e $O_3 = f((x_{11} * w) + (x_{12} * w) + (x_{13} * w))$

applications: Speech recognition (Google voice search)
machine translation (Google Translate)
Time series forecasting
sales forecasting, etc...

10. Explain Backpropagation through time (BPTT) algorithm in detail.

step 1: forward pass

For the whole i/p sequence, unroll the RNN over time, and calculate the forward pass for each time step. The hidden state is updated throughout the forward pass, and the RNN's eqn are used to compute the o/p at each time step.

step 2: calculating the loss

By contrasting the expected o/p (y-t) with the actual target (the ground truth) at each time step, determine the loss at each

time step. The sum of the losses at all time steps is the overall loss. Depending on the particular task, several loss fn are employed such as mean Squared Error (MSE) for regression or Cross-Entropy loss fn. for classification.

Step 3: Back propogation:
To calculate gradients like with regard to the RNNs parameters (weights and biases), use backpropogation across time. The gradient are computed via the chain rule starting from the last time step (T) and going back to the initial time step (t=1). As they share weights, the gradients are accumulated over time steps.

Step 4: Gradient update
using the obtained gradient, modify the RNN's parameters. For this update, you can use the common gradient descent or one of its variant (such as Adam or RMSprop). Based on the particular problem and design, an appropriate technique & learning rate can be selected.

Challenges with BPTT.
- Vanishing / Exploding gradient problem.
- Memory requirements : Greater memory requirement.

Solution to BPTT challenges:
- Long short term memory (LSTM) &
- Gated Recurrent Unit (GRU)
These topologies employ gating methods to regulate the info. flow enabling RNNs to keep track of (store) data across longer sequences.

11. Explain Vanishing Gradient problem in RNN

> As a gradient moves backward in time, its magnitude gets less and smaller until it vanishes entirely. This occur when the recurrent weights are nearly zero or when the activation fn, such as the sigmoid fn utilized in the RNN has a small derivative. Because of this, the model has trouble updating the weights in earlier time steps, which prevents it from discovering long-term dependencies.

consider the RNN update eqn for a single time step t.

$$h_t = f(W_h * h_{t-1} + W_x * x_t + b)$$

... h_t : Hidden state at time step t.

f : Activation fn (eg. tanh or sigmoid)

W_h : weight matrix for hidden state ; W_x : for i/p.

b : bias vector ; x_t : i/p at time step t.

> during backpropagation, the gradient of the loss w.r.t. hidden state ($\partial L / \partial h_t$) is calculated & propagated backward to update the weights (W_h, W_x) and biases (b). The gradient update involves taking the derivative of the activation fn (f) and is multiplied with the chain rule of gradients are propagated through time.

> If the activation fn is the sigmoid fn ($f(z) = 1/(1+\exp^{(-z)})$), its derivative is given by, $f'(z) = f(z) * (1 - f(z))$

as the absolute value of $f(z)$ approaches zero, the gradient vanishes, leading to vanishing gradient problem.

☆ Exploding- Gradients in RNN.

> on the other hand, explosive gradients happen when a gradients magnitude increases dangerously during backpropagation. This may occur if the activation function has a high derivative or the recurrent weights are substantial. The model diverges during training as a

PTO

result of the steep gradients heavy weight updates.

> The recurrent weights (w-h) at each time step, on the other hand are doubled when gradients are propagated backward in time.

> The exploding gradient problem occurs when the gradients expand exponentially as they progress backward through sequence when the recurrent weights are significant (more than 1).

---

12. Explain LSTM in detail.

> Long short-term memory, sometimes known as LSTM, is a sort of sophisticated Recurrent Neural Network (RNN) architecture. The disappearing and ballooning gradient difficulties, as well as the challenge of capturing long-term relationship in sequential data are some of the shortcomings of conventional RNNs that LSTM were created by overcome.

> Simply put, the LSTM introduces a unique memory cell and three gating mechanisms (input gate, forget gate, and output gate) that allow it to selectively keep or forget info. over time. As a result, LSTM can handle long-range relationship and keep crucial info. over lengthy sequences.

- Memory cell: An LSTM's main building block is a memory cell. It is updated using i/p, forget & o/p gates and stores the context or data for each time steps.
- I/p Gate: The gate chooses which data for the most recent concealed state and the current time step should be added to the memory cell.
- Forget Gate: The forget gate makes the decision as to which data in the memory cell should be deleted or forgotten.
- O/p Gate: The o/p gate regulates how much of the data stored in the

PTO

memory cell should be used to make the o/p or hidden state for the current time step.

★ GRU: Gated Recurrent Unit

A modification of the conventional long short-term memory (LSTM) architecture known as the Gated Recurrent Unit (GRU) was created in order to solve some of the LSTM's complexities while preserving its capacity to identify long-term dependencies in sequential data.

> Key components of GRU

- Update Gate (z_t): The update gate determines how much of the previous hidden state to retain (1- forget) and how much of the new candidate activation to consider (update)
- Reset Gate (r_t): the reset gate determines how much of previous hidden state should be forgotten (reset) to account for new i/p.
- Candidate activation (h_tilde_t): the candidate activation represents the new info. that can be added to hidden state.

> Advantages of Gated Recurrent Unit (GRU) over LSTM

- Simpler architecture: fewer gating mechanisms than LSTM, making it easier to use & more efficient computationally.
- Faster training: can frequently be trained more quickly.
- Less prone to overfitting
- Simpler performance to LSTM.

---

13. Explain GAN architecture in detail.

Generative Adversarial NNs don't work with any explicit density estimation, but it is based on game theory approach with an objective to find Nash equilibrium, b/w the two NNs, Generator & Discriminator. The idea is to sample from a simple distribution like Gaussian &

turn learn to transform this noise to data distribution using accuracy for approximators such as NNway NN.

> Generative: To learn generative model, which describes how data is generated in terms of a probabilistic model.

Adversarial: training of model is done in an adversarial setting.

NN: use DNN as AI algorithm for training purpose.

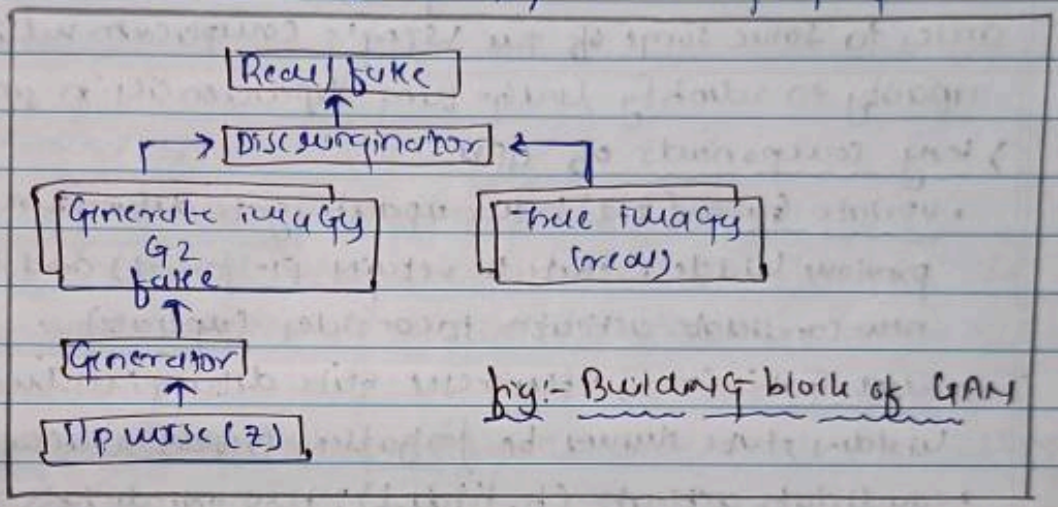


fig:- Building block of GAN

The objective fn of GAN is defined as:

$$\min_G \max_D V(D,G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim P_z(z)}[\log(1 - D(G(z))$$

... where G = Generator ; x = sample from p data (x)

D = Discriminator ; z = sample from p(z)

p(data)(x) = distribution of real data

p(z) = distribution of generator ; G(z) = Generator NN.

D(x) = discriminator NN

applications:

- image Generation: involves the use of two-part NN - a generator and a discriminator. Generator creates fakes imgs while discriminator evaluates.
- deep fake: used to create realistic but fake videos or images by swapping faces or modifying content in an existing video.

6. Given an input image size 28*28*1. Apply 1 convolution layers. First layer - apply 64 filters each of size of 3*3, max pooling - s=2, f=2 calculate the output volume, activation maps & no. of parameters for each layer.

Soln :-

layer	layer Neuron shape	Activation shape	parameters.
INPUT I/p	(28×28×1) - gray scale image	28×28×1 = 784	0
Conv2D (filter = 3×3)	(26×26×64)	43,264	640
maxpooling2D	(13×13×64)	10,816	0
flatten layer	(10816, 1)	(10,816)	0
Dense	(128, 1)	128	13,84,576
Dense	(10, 1)	10	1290 (13 neuron)
	10 Neuron dense layer		

1) i/p : 28×28×1, 64 filters, f=3

① o/p volume = $\frac{28+0-3}{1}+1 = 25+1 \Rightarrow 26$ //

∴ (26, 26, 64) = 43,264

② 64 filters of size 3×3×1 channel $[(3\times3\times1)+1]\times64$

$10\times64 \Rightarrow 640$ //

2) pooling (26, 26, 64), s=2 → o/p vol = $\frac{26-3}{2}+1 = 12.5 = 13$ //

f=2, s=2 if 2×2 pooling $\frac{26-2}{2}+1 \Rightarrow \frac{24}{2}+1 \Rightarrow 13$ //

Conv2D dimensions - pool dimension $+1 = \frac{26-2}{2}+1 \Rightarrow 13$ //

stride

3) Dense : layer 1

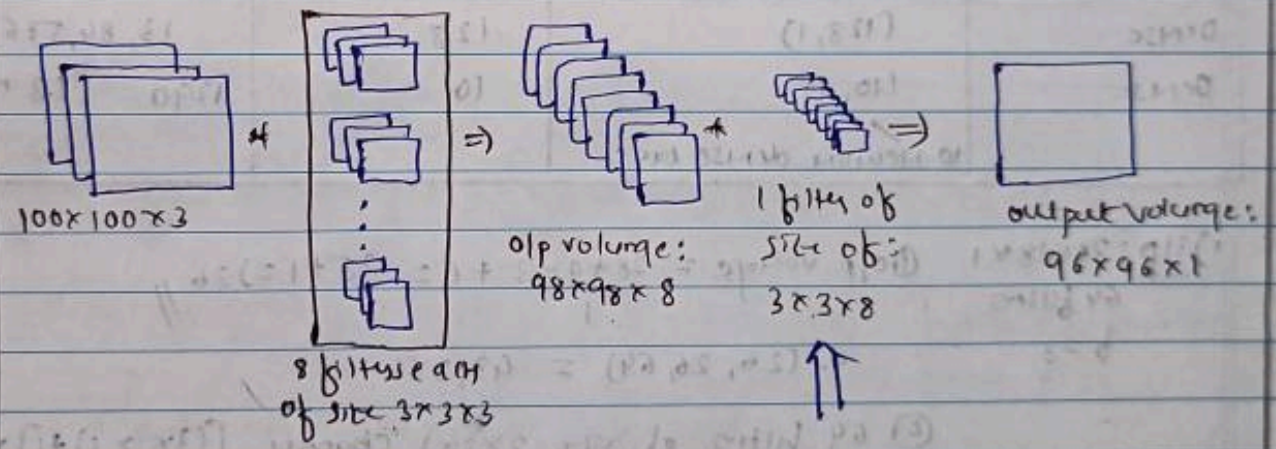
PTO

6. (current layer memory * previous layer memory)
+1 * (current layer memory)
(128 * 10,816) + 1 * 128 ⇒ 13,84,576
(1.3 million)

4) Dense layer 2:
(10 * 128) + 1 × 10 ⇒ 1290.

7. Given an input image volume of dimension 100 * 100 * 3. For first convolution layer, apply 8 filters of size 3 * 3 * 3. For second convolution layer apply 1 filter of size 3 * 3. Calculate the no. of parameters at each convolution layer. (Assume s=1, p=0)

Soln:-



no. of parameters = [(3×3×3) + 1] × 8
= 224

no of parameters = [(3×3×8) + 1] * 1
= 73

CNN architecture