SRM Institute of Science and Technology
Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204
Department of Computer Applications
Circular – 2023-24
BCA DS 5th Sem

Deep Learning with Keras and Tensorflow (UDS23501J)

Lab Manual

Here's a structured lab manual based on the provided list of programs. Since I don't have the specific code for each lab, I'll provide a general structure and example code in Python using Keras and TensorFlow, which are commonly used for deep learning tasks. You can fill in the specific code for each lab exercise.

Lab 1: Learning XOR Problem

Title: Learning XOR Problem

Aim: To implement a neural network to solve the XOR problem.

Procedure:

1. Define the XOR input and output data.
2. Build a neural network model with appropriate layers (e.g., a simple network with one hidden layer).
3. Compile the model with a suitable optimizer and loss function (e.g., 'adam' and 'binary_crossentropy').
4. Train the model on the XOR data.
5. Evaluate the model's performance.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define XOR data
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [0, 1, 1, 0]

# Build the model
model = Sequential([
    Dense(2, input_dim=2, activation='relu'),
    Dense(1, activation='sigmoid')
])

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

# Train the model
model.fit(X, y, epochs=1000, verbose=0)
```

```python
# Evaluate the model
loss, accuracy = model.evaluate(X, y)
print(f"Loss: {loss}, Accuracy: {accuracy}")

# Make predictions
predictions = model.predict(X)
for i in range(len(X)):
    print(f"Input: {X[i]}, Predicted: {predictions[i][0]:.4f}, Actual: {y[i]}")
```

Input:

X: [[0, 0], [0, 1], [1, 0], [1, 1]]

Expected Output:

The model should learn the XOR function, and the predictions should be close to the actual XOR outputs: [0, 1, 1, 0]. The accuracy should be high (close to 1).

Lab 2: Image Classification using CNN

Title: Image Classification using CNN

Aim: To build and train a Convolutional Neural Network (CNN) for image classification.

Procedure:

1. Load and preprocess an image dataset (e.g., MNIST, CIFAR-10).
2. Define the CNN architecture (e.g., convolutional layers, pooling layers, fully connected layers).
3. Compile the model.
4. Train the model on the training data.
5. Evaluate the model on the test data.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
x_train = x_train.reshape(-1, 28, 28, 1) # Add channel dimension
x_test = x_test.reshape(-1, 28, 28, 1)

# Define the CNN model
model = keras.Sequential(
    [
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu",
input_shape=(28, 28, 1)),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(10, activation="softmax"),
    ]
)

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

# Train the model
model.fit(x_train, y_train, batch_size=128, epochs=5, validation_split=0.1)

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Make predictions (example)
predictions = model.predict(x_test[:10])
print(predictions)
```

Input:

MNIST dataset (handwritten digits)

Expected Output:

The model should classify the images into their correct digit categories (0-9) with reasonable accuracy.

Lab 3: Building a Deep Learning Model

Title: Building a Deep Learning Model

Aim: To design and implement a deep learning model for a specific task (e.g., regression, classification).

Procedure:

1. Define the problem and dataset.
2. Choose an appropriate model architecture (e.g., MLP, CNN, RNN).
3. Build the model using Keras/TensorFlow.
4. Compile and train the model.
5. Evaluate and fine-tune the model.

Source Code: (This is a general example; you'll need to adapt it)

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense

# 1. Define the problem and dataset (Example: Regression)
#   Assume you have loaded your data into X_train, y_train, X_test, y_test

# 2. Choose a model architecture (Example: Multi-Layer Perceptron)
model = keras.Sequential([
    Dense(64, activation='relu', input_dim=X_train.shape[1]),  # Adjust
input_dim
    Dense(64, activation='relu'),
    Dense(1)  # Output layer for regression
])

# 3. Compile the model
model.compile(optimizer='adam', loss='mse')  # Mean Squared Error for regression

# 4. Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)

# 5. Evaluate the model
loss = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}")

# Make predictions
predictions = model.predict(X_test)
print(predictions)
```

Input:

Dataset specific to the chosen problem.

Expected Output:

Output will depend on the problem (e.g., predicted values for regression, class probabilities for classification).

Lab 4: Data Augmentation Lab

Title: Data Augmentation Lab

Aim: To apply data augmentation techniques to increase the size and variability of the training data.

Procedure:

1. Load the dataset.
2. Create an ImageDataGenerator object from Keras.
3. Specify augmentation parameters (rotation, zoom, flip, etc.)
4. Apply the data augmentation to the training data.
5. Train a model using the augmented data.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Load dataset (example using CIFAR-10)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Create ImageDataGenerator
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    zoom_range=0.2,
)

# Apply augmentation to training data
datagen.fit(x_train)
train_generator = datagen.flow(x_train, y_train, batch_size=32)

# Define a simple CNN model
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu',
input_shape=x_train.shape[1:]),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(10, activation='softmax')
])

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

# Train the model using the data generator
model.fit(train_generator, epochs=10, validation_data=(x_test, y_test))

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")
```

Input:

6. A dataset of images (e.g., CIFAR-10)

Expected Output:

7. A trained model with improved generalization due to the augmented training data. The validation accuracy should be higher than if no data augmentation was used.

Lab 5: Implementation of RNN

Title: Implementation of RNN

Aim: To implement a Recurrent Neural Network (RNN) for sequence data.

Procedure:

1. Prepare sequence data (e.g., text, time series).
2. Preprocess the data (e.g., tokenization, padding).
3. Build an RNN model (e.g., SimpleRNN, LSTM, GRU).
4. Compile and train the model.
5. Evaluate the model.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import SimpleRNN, Embedding, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Sample text data
sentences = [
    "This is the first sentence.",
    "Here is another sentence.",
    "Sentences are sequences of words."
]

# Tokenize the text
tokenizer = Tokenizer(num_words=100)  # Limit vocabulary size
tokenizer.fit_on_texts(sentences)
sequences = tokenizer.texts_to_sequences(sentences)

# Pad sequences to ensure equal length
padded_sequences = pad_sequences(sequences)

# Build the RNN model
model = keras.Sequential([
    Embedding(input_dim=100, output_dim=32,
input_length=padded_sequences.shape[1]),  #  input_dim is vocab size
    SimpleRNN(32),
    Dense(1, activation='sigmoid')  # For binary classification (example)
])

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

# Train the model (replace with your training data)
import numpy as np
labels = np.array([0, 1, 0]) # Example labels
model.fit(padded_sequences, labels, epochs=10)

# Evaluate the model (replace with your test data)
# loss, accuracy = model.evaluate(test_data, test_labels)
# print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Make predictions
# predictions = model.predict(new_data)
# print(predictions)
```

Input:

Sequence data (e.g., text sentences).

Expected Output:

Output depends on the task (e.g., sentiment classification, next word prediction).

Lab 6: Restricted Boltzmann Machine

Title: Restricted Boltzmann Machine

Aim: To implement a Restricted Boltzmann Machine (RBM).

Procedure:

1. Prepare the input data.
2. Define the RBM architecture (number of visible and hidden units).
3. Initialize weights and biases.
4. Implement the Contrastive Divergence algorithm for training.
5. Train the RBM.
6. Use the trained RBM for tasks like feature extraction or generation.

Source Code: (Note: RBM implementations in TensorFlow/Keras are less common now; this is a conceptual outline. Use a library like torch if needed.)

```
import numpy as np
#import tensorflow as tf # Removed tensorflow
# No direct replacement.  Conceptual outline.

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

class RBM:
    def __init__(self, n_visible, n_hidden):
        self.n_visible = n_visible
        self.n_hidden = n_hidden
        # Initialize weights and biases
        self.W = np.random.randn(n_visible, n_hidden) * 0.01
        self.v_bias = np.zeros(n_visible)
        self.h_bias = np.zeros(n_hidden)

    def forward(self, v):
        """Propagate visible to hidden."""
        return sigmoid(np.dot(v, self.W) + self.h_bias)

    def backward(self, h):
        """Propagate hidden to visible."""
        return sigmoid(np.dot(h, self.W.T) + self.v_bias)

    def sample_hidden(self, v):
        """Sample hidden units given visible units."""
        h_prob = self.forward(v)
        return np.random.binomial(n=1, p=h_prob)  # Changed to numpy

    def sample_visible(self, h):
        """Sample visible units given hidden units."""
        v_prob = self.backward(h)
        return np.random.binomial(n=1, p=v_prob) # Changed to numpy

    def train(self, data, epochs=10, batch_size=10, learning_rate=0.1):
        """Train the RBM using Contrastive Divergence."""
        n_samples = data.shape[0]
        for epoch in range(epochs):
            for start in range(0, n_samples, batch_size):
                end = min(start + batch_size, n_samples)
                batch = data[start:end]
```

```
                # 1. Positive phase
                v0 = batch
                h0 = self.sample_hidden(v0)

                # 2. Negative phase
                v1 = self.sample_visible(h0)
                h1 = self.sample_hidden(v1)

                # 3. Update weights and biases
                positive_grad = np.dot(v0.T, h0)
                negative_grad = np.dot(v1.T, h1)
                self.W += learning_rate * (positive_grad - negative_grad) /
batch_size
                self.v_bias += learning_rate * np.mean(v0 - v1, axis=0)
                self.h_bias += learning_rate * np.mean(h0 - h1, axis=0)

# Example usage (replace with your data)
# Assuming you have binary data
data = np.random.randint(0, 2, size=(100, 10))  # 100 samples, 10 visible units
rbm = RBM(n_visible=10, n_hidden=5)
rbm.train(data, epochs=100, batch_size=10, learning_rate=0.1)

# To use the trained RBM (e.g., for feature extraction):
# hidden_representation = rbm.forward(data)
# print(hidden_representation)
```

Input:

Binary or real-valued data.

Expected Output:

The RBM learns a probabilistic model of the input data. Output can be reconstructed data, hidden unit activations (features), or generated samples.

Lab 7: Generative Adversarial Networks

Title: Generative Adversarial Networks

Aim: To implement a Generative Adversarial Network (GAN) for generating new data.

Procedure:

1. Define the generator and discriminator networks.
2. Define the loss functions for the generator and discriminator.
3. Implement the training loop:
   - Train the discriminator to distinguish between real and generated data.
   - Train the generator to fool the discriminator.
4. Generate new data using the trained generator.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

# Define the generator
def build_generator(latent_dim):
    model = keras.Sequential([
        layers.Dense(7 * 7 * 256, use_bias=False, input_shape=(latent_dim,)),
        layers.BatchNormalization(),
        layers.Reshape((7, 7, 256)),
        layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
use_bias=False, activation='relu'),
        layers.BatchNormalization(),
        layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='relu'),
        layers.BatchNormalization(),
        layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'), # Output
    ])
    return model

# Define the discriminator
def build_discriminator(img_shape):
    model = keras.Sequential([
        layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
input_shape=img_shape, activation='relu'),
        layers.Dropout(0.3),
        layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same',
activation='relu'),
        layers.Dropout(0.3),
        layers.Flatten(),
        layers.Dense(1, activation='sigmoid')  # Output: probability of being
real
    ])
    return model

# Define the GAN model
def build_gan(generator, discriminator):
    discriminator.trainable = False  # Only train generator in the combined
model
    model = keras.Sequential([generator, discriminator])
    return model
```

```
# Hyperparameters
latent_dim = 100
img_shape = (28, 28, 1)  # Example: MNIST image shape

# Build the networks
generator = build_generator(latent_dim)
discriminator = build_discriminator(img_shape)
gan_model = build_gan(generator, discriminator)

# Optimizers
generator_optimizer = keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
discriminator_optimizer = keras.optimizers.Adam(learning_rate=0.0002,
beta_1=0.5)

# Loss functions
cross_entropy = keras.losses.BinaryCrossentropy(from_logits=False)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

# Training function
@tf.function
def train_step(images, batch_size):
    noise = tf.random.normal([batch_size, latent_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
    return gen_loss, disc_loss

def train_gan(dataset, epochs, batch_size):
    for epoch in range(epochs):
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch, batch_size)
        print(f"Epoch {epoch}, Generator Loss: {gen_loss:.4f}, Discriminator
Loss: {disc_loss:.4f}")

# Load and preprocess data (example using MNIST)
(x_train, _), (_, _) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1).astype('float32')
x_train = (x_train - 127.5) / 127.5  # Normalize to [-1, 1]
buffer_size = x_train.shape[0]
batch_size = 128
train_dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(buffer_size).batch(batch_siz
e)
```

```
# Train the GAN
train_gan(train_dataset, epochs=50, batch_size=batch_size)

# Generate images after training
# noise = tf.random.normal([16, latent_dim])
# generated_images = generator(noise)
# ... (display or save generated images)
```

Input:

A dataset of real images (e.g., MNIST).

Expected Output:

The generator learns to produce new images that resemble the training data.

Lab 8: Variational Autoencoder

Title: Variational Autoencoder

Aim: To implement a Variational Autoencoder (VAE) for data generation or dimensionality reduction.

Procedure:

1. Define the encoder network (maps input to a latent distribution).
2. Define the decoder network (maps latent samples to reconstructed data).
3. Define the VAE model, including the reparameterization trick.
4. Define the loss function (reconstruction loss + KL divergence).
5. Train the VAE.
6. Generate new data by sampling from the latent space and passing it through the decoder.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.losses import MeanSquaredError

# Define the encoder
def build_encoder(latent_dim, img_shape):
    encoder_inputs = keras.Input(shape=img_shape)
    x = layers.Conv2D(32, 3, activation="relu", strides=2,
padding="same")(encoder_inputs)
    x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
    x = layers.Flatten()(x)
    x = layers.Dense(16, activation="relu")(x)
    z_mean = layers.Dense(latent_dim, name="z_mean")(x)
    z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
    return keras.Model(encoder_inputs, [z_mean, z_log_var])

# Define the decoder
def build_decoder(latent_dim, img_shape):
    latent_inputs = keras.Input(shape=(latent_dim,))
    input_side = img_shape[0] // 4
    x = layers.Dense(input_side * input_side * 64,
activation="relu")(latent_inputs)
    x = layers.Reshape((input_side, input_side, 64))(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2,
padding="same")(x)
    x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2,
padding="same")(x)
    decoder_outputs = layers.Conv2D(img_shape[-1], 3, activation="sigmoid",
padding="same")(x) # Output
    return keras.Model(latent_inputs, decoder_outputs)

# Define the VAE class
class VAE(keras.Model):
    def __init__(self, latent_dim, encoder, decoder, img_shape):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = encoder
        self.decoder = decoder
        self.img_shape = img_shape
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
```

```
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def sample(self, z_mean, z_log_var):
        """Reparameterization trick."""
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

    def call(self, data):
        z_mean, z_log_var = self.encoder(data)
        z = self.sample(z_mean, z_log_var)
        reconstruction = self.decoder(z)
        reconstruction_loss = tf.reduce_mean(
            tf.reduce_sum(
                keras.losses.binary_crossentropy(data, reconstruction), axis=(1,
2)
            )
        )
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
        kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
        total_loss = reconstruction_loss + kl_loss
        return reconstruction, total_loss, reconstruction_loss, kl_loss

    def train_step(self, data):
        with tf.GradientTape() as tape:
            reconstruction, total_loss, reconstruction_loss, kl_loss =
self(data)
        grads = tape.gradient(total_loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.trainable_variables))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }

# Example usage
img_shape = (28, 28, 1) # MNIST
latent_dim = 2
encoder = build_encoder(latent_dim, img_shape)
decoder = build_decoder(latent_dim, img_shape)
vae = VAE(latent_dim, encoder, decoder, img_shape)
vae.compile(optimizer=keras.optimizers.Adam())

# Load MNIST data
(x_train, _), (_, _) = keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_train = np.reshape(x_train, (-1, 28, 28, 1))

# Train the VAE
vae.fit(x_train, epochs=10, batch_size=128)

# Generate new images
```

```
# z_sample = tf.random.normal(shape=(16, latent_dim))
# generated_images = vae.decoder(z_sample)
# ... (display or save images)
```

Input:

A dataset of images (e.g., MNIST).

Expected Output:

The VAE learns a latent representation of the data, which can be used to generate new samples or for dimensionality reduction.

Lab 9: LSTM

Title: LSTM

Aim: To implement a Long Short-Term Memory (LSTM) network for sequence data.

Procedure:

1. Prepare sequence data.
2. Preprocess the data.
3. Build an LSTM model.
4. Compile and train the model.
5. Evaluate the model.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import LSTM, Embedding, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Sample text data (example)
sentences = [
    "The quick brown fox jumps over the lazy dog.",
    "A journey of a thousand miles begins with a single step.",
    "It is not the strongest of the species that survives, nor the most
intelligent that survives. It is the one that is most adaptable to change."
]

# Tokenize the text
tokenizer = Tokenizer(num_words=200)  # Limit vocabulary size
tokenizer.fit_on_texts(sentences)
sequences = tokenizer.texts_to_sequences(sentences)

# Pad sequences
max_length = max(len(seq) for seq in sequences)
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post')

# Create labels (example: predict the next word)
X, y = padded_sequences[:, :-1], padded_sequences[:, -1]
y = keras.utils.to_categorical(y, num_classes=200) #  num_classes

# Build the LSTM model
model = keras.Sequential([
    Embedding(input_dim=200, output_dim=64, input_length=max_length-1), #
input_dim
    LSTM(64),
    Dense(200, activation='softmax') #  num_classes
])

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

# Train the model
model.fit(X, y, epochs=100, verbose=1)

# Evaluate the model (replace with your test data)
# loss, accuracy = model.evaluate(X_test, y_test)
```

```
# print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Make predictions (example)
# test_sentence = "The quick brown fox jumps over the"
# test_sequence = tokenizer.texts_to_sequences([test_sentence])
# test_padded = pad_sequences(test_sequence, maxlen=max_length-1,
padding='post')
# prediction = model.predict(test_padded)
# predicted_word_index = np.argmax(prediction)
# predicted_word = tokenizer.index_word[predicted_word_index]
# print(f"Predicted next word: {predicted_word}")
```

Input:

Sequence data (e.g., text).

Expected Output:

The model learns to predict the next element in the sequence (e.g., next word in a sentence).

Lab 10: Bidirectional LSTM

Title: Bidirectional LSTM

Aim: To implement a Bidirectional LSTM network for sequence data.

Procedure:

1. Prepare sequence data.
2. Preprocess the data.
3. Build a Bidirectional LSTM model.
4. Compile and train the model.
5. Evaluate the model.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Bidirectional, LSTM, Embedding, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Sample text data
sentences = [
    "The quick brown fox jumps over the lazy dog.",
    "A journey of a thousand miles begins with a single step.",
    "It is not the strongest of the species that survives, nor the most
intelligent that survives. It is the one that is most adaptable to change."
]

# Tokenize the text
tokenizer = Tokenizer(num_words=200)  # Limit vocabulary size
tokenizer.fit_on_texts(sentences)
sequences = tokenizer.texts_to_sequences(sentences)

# Pad sequences
max_length = max(len(seq) for seq in sequences)
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post')

# Create labels (example: predict the next word)
X, y = padded_sequences[:, :-1], padded_sequences[:, -1]
y = keras.utils.to_categorical(y, num_classes=200) # num_classes

# Build the Bidirectional LSTM model
model = keras.Sequential([
    Embedding(input_dim=200, output_dim=64, input_length=max_length-1), #
input_dim
    Bidirectional(LSTM(64)),
    Dense(200, activation='softmax') # num_classes
])

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

# Train the model
model.fit(X, y, epochs=100, verbose=1)

# Evaluate the model (replace with your test data)
# loss, accuracy = model.evaluate(X_test, y_test)
```

```
# print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Make predictions (example)
# test_sentence = "The quick brown fox jumps over the"
# test_sequence = tokenizer.texts_to_sequences([test_sentence])
# test_padded = pad_sequences(test_sequence, maxlen=max_length-1,
padding='post')
# prediction = model.predict(test_padded)
# predicted_word_index = np.argmax(prediction)
# predicted_word = tokenizer.index_word[predicted_word_index]
# print(f"Predicted next word: {predicted_word}")
```

Input:

Sequence data (e.g., text).

Expected Output:

The model learns to predict the next element in the sequence, considering both past and future context.

Lab 11: Data Augmentation Lab I

Title: Data Augmentation Lab I

Aim: To apply a variety of data augmentation techniques.

Procedure:

1. Load image data.
2. Create an ImageDataGenerator.
3. Specify a range of augmentations.
4. Visualize the augmented images.
5. Train a model (optional).

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt

# Load a sample dataset (e.g., a few images from CIFAR-10)
(x_train, _), (_, _) = keras.datasets.cifar10.load_data()
sample_images = x_train[:9]  # Take the first 9 images

# Create an ImageDataGenerator with various augmentations
datagen = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Prepare the data for augmentation
sample_images = sample_images.astype('float32') / 255.0
augmented_images = []
for img in sample_images:
    img = img.reshape((1,) + img.shape) # reshape
    i = 0
    for batch in datagen.flow(img, batch_size=1):
        augmented_images.append(batch[0])
        i += 1
        if i > 0:
            break

# Visualize the original and augmented images
plt.figure(figsize=(12, 6))
for i in range(9):
    plt.subplot(3, 6, i + 1)
    plt.imshow(sample_images[i])
    plt.title("Original")
    plt.axis('off')

    plt.subplot(3, 6, i + 10)
    plt.imshow(augmented_images[i])
    plt.title("Augmented")
    plt.axis('off')
plt.show()
```

Input:

A set of images.

Expected Output:

A visualization showing the original images and their augmented versions, demonstrating the effects of the chosen augmentation techniques.

Lab 12: Data Augmentation Lab II

Title: Data Augmentation Lab II

Aim: To explore advanced data augmentation techniques or libraries.

Procedure:

1. Import necessary libraries (e.g., albumentations).
2. Define an augmentation pipeline.
3. Apply the pipeline to images.
4. Visualize the augmented images.
5. (Optional) Train a model with augmented data.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
import albumentations as A
import matplotlib.pyplot as plt

# Load sample images (e.g., from CIFAR-10)
(x_train, _), (_, _) = keras.datasets.cifar10.load_data()
sample_images = x_train[:9]

# Define an augmentation pipeline using albumentations
transform = A.Compose([
    A.RandomCrop(width=32, height=32),
    A.HorizontalFlip(p=0.5),
    A.RandomBrightnessContrast(p=0.2),
    A.Rotate(limit=30, p=0.3),
    A.Blur(blur_limit=3, p=0.1),
])

# Apply the augmentation pipeline
augmented_images = []
for img in sample_images:
    transformed = transform(image=img)['image']
    augmented_images.append(transformed)

# Visualize the original and augmented images
plt.figure(figsize=(12, 6))
for i in range(9):
    plt.subplot(3, 6, i + 1)
    plt.imshow(sample_images[i])
    plt.title("Original")
    plt.axis('off')

    plt.subplot(3, 6, i + 10)
    plt.imshow(augmented_images[i])
    plt.title("Augmented")
    plt.axis('off')
plt.show()
```

Input:

A set of images.

Expected Output:

A visualization of the original and augmented images, demonstrating the effects of the advanced augmentation pipeline.

Lab 13: Install, Import Tensorflow and Keras. Create a Basic Neural Network with a Few Layers.

Title: Install, Import Tensorflow and Keras. Create a Basic Neural Network with a Few Layers.

Aim: To set up the TensorFlow and Keras environment and build a simple neural network.

Procedure:

1. Install TensorFlow and Keras (if not already installed). (Omitted in the code, but crucial)
2. Import TensorFlow and Keras.
3. Define a simple neural network model.
4. Compile the model.
5. (Optional) Train and evaluate the model (with dummy data).

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense

# Verify TensorFlow installation
print("TensorFlow version:", tf.__version__)

# Create a basic neural network model
model = keras.Sequential([
    Dense(128, activation='relu', input_dim=10),  # Input layer with 10 features
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])

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

# Optional: Train and evaluate with dummy data
import numpy as np
X_train = np.random.rand(100, 10)  # 100 samples, 10 features
y_train = np.random.randint(0, 2, size=(100, 1))  # Binary labels

model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=0) # Suppress
output
loss, accuracy = model.evaluate(X_train, y_train, verbose=0)
print(f"Loss: {loss}, Accuracy: {accuracy}")
```

Input:

(Optional) Dummy input data for training.

Expected Output:

The code should print the TensorFlow version, indicating successful installation. If the optional training is run, it will print the loss and accuracy on the dummy data.

Lab 14: Install, Import Tensorflow and Keras. Create a Basic Neural Network with a Few Layers

Title: Install, Import Tensorflow and Keras. Create a Basic Neural Network with a Few Layers

Aim: This lab has the same aim as Lab 13.

Procedure:

1. Install TensorFlow and Keras (if not already installed).
2. Import TensorFlow and Keras.
3. Define a simple neural network model.
4. Compile the model.
5. (Optional) Train and evaluate the model.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense

# Verify TensorFlow installation
print("TensorFlow version:", tf.__version__)

# Create a basic neural network model
model = keras.Sequential([
    Dense(64, activation='relu', input_dim=20),  # Input layer
    Dense(32, activation='relu'),
    Dense(10, activation='softmax')  # Output layer for multi-class
classification
])

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

# Optional: Train and evaluate with dummy data
import numpy as np
X_train = np.random.rand(100, 20)  # 100 samples, 20 features
y_train = np.random.randint(0, 10, size=(100, 1))  # Multi-class labels (0-9)
y_train = keras.utils.to_categorical(y_train, num_classes=10) # one-hot encode

model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=0)
loss, accuracy = model.evaluate(X_train, y_train, verbose=0)
print(f"Loss: {loss}, Accuracy: {accuracy}")
```

Input:

(Optional) Dummy data

Output:

Prints the TensorFlow version. Optionally prints loss and accuracy.

Lab 15: Neural Networks using Keras

Title: Neural Networks using Keras

Aim: To implement various neural network architectures using Keras.

Procedure:

1. Explore different layer types (Dense, Conv2D, LSTM, etc.).
2. Experiment with different activation functions, optimizers, and loss functions.
3. Build and train several neural network models for different tasks.
4. Evaluate and compare the performance of the models.

Source Code: (This is a general template; you would create multiple models here)

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense, Conv2D, Flatten, LSTM, Embedding
from tensorflow.keras.datasets import mnist
import numpy as np

# Example 1: Multi-Layer Perceptron (MLP) for MNIST
(x_train_mnist, y_train_mnist), (x_test_mnist, y_test_mnist) = mnist.load_data()
x_train_mnist = x_train_mnist.reshape(-1, 28 * 28).astype('float32') / 255.0
x_test_mnist = x_test_mnist.reshape(-1, 28 * 28).astype('float32') / 255.0
y_train_mnist = keras.utils.to_categorical(y_train_mnist, num_classes=10)
y_test_mnist = keras.utils.to_categorical(y_test_mnist, num_classes=10)

mlp_model = keras.Sequential([
    Dense(512, activation='relu', input_dim=28 * 28),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
mlp_model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
mlp_model.fit(x_train_mnist, y_train_mnist, epochs=10, verbose=0)
loss, accuracy = mlp_model.evaluate(x_test_mnist, y_test_mnist, verbose=0)
print(f"MLP - Loss: {loss}, Accuracy: {accuracy}")


# Example 2: Convolutional Neural Network (CNN) for MNIST
(x_train_cnn, y_train_cnn), (x_test_cnn, y_test_cnn) = mnist.load_data()
x_train_cnn = x_train_cnn.reshape(-1, 28, 28, 1).astype('float32') / 255.0
x_test_cnn = x_test_cnn.reshape(-1, 28, 28, 1).astype('float32') / 255.0
y_train_cnn = keras.utils.to_categorical(y_train_cnn, num_classes=10)
y_test_cnn = keras.utils.to_categorical(y_test_cnn, num_classes=10)

cnn_model = keras.Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    Flatten(),
    Dense(10, activation='softmax')
])
cnn_model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
cnn_model.fit(x_train_cnn, y_train_cnn, epochs=10, verbose=0)
loss, accuracy = cnn_model.evaluate(x_test_cnn, y_test_cnn, verbose=0)
print(f"CNN - Loss: {loss}, Accuracy: {accuracy}")
```

```
# Example 3:  LSTM for sequence data (example, you'd need real sequence data)
#  ... (Code for LSTM example, similar to Lab 9, but with potentially different
data)
```

Input:

MNIST dataset, or other appropriate datasets.

Expected Output:

The code will print the loss and accuracy for each of the neural network models (MLP, CNN, and potentially others), allowing you to compare their performance.