University of Central Missouri

Department of Computer Science & Cybersecurity

CS5720 Neural network and Deep learning

Spring 2025

Home Assignment 3. (Cover Ch 7, 8)

Student Name : Shravya Nagabandi

Student ID : 700757053

Submission Requirements:

- Total Points: 100
- Once finished your assignment push your source code to your repo (GitHub) and explain the work through the ReadMe file properly. Make sure you add your student info in the ReadMe file.
- Submit your GitHub link and video on the BB.
- Comment your code appropriately ***IMPORTANT.***
- Make a simple video about 2 to 3 minutes which includes demonstration of your home assignment and explanation of code snippets.
- Any submission after provided deadline is considered as a late submission.

Q1: Implementing a Basic Autoencoder

Task: Autoencoders learn to reconstruct input data by encoding it into a lower-dimensional space. You will build a **fully connected autoencoder** and evaluate its performance on image reconstruction.

1. Load the **MNIST dataset** using tensorflow.keras.datasets.
2. Define a **fully connected (Dense) autoencoder**:
    - Encoder: Input layer (784), hidden layer (32).
    - Decoder: Hidden layer (32), output layer (784).
3. Compile and train the autoencoder with **binary cross-entropy loss**.
4. Plot **original vs. reconstructed images** after training.
5. Modify the latent dimension size (e.g., 16, 64) and analyze how it affects the quality of reconstruction.

Hint: *Use Model() from tensorflow.keras.models and Dense() layers.*

Answer:
This implementation follows the task requirements:

1. Loads and preprocesses the MNIST dataset.
2. Defines a fully connected autoencoder with an encoder (latent space of 32) and a decoder.
3. Trains the model with binary cross-entropy loss.
4. Visualizes original vs. reconstructed images.
5. Allows for easy modification of the latent dimension to analyze its effect.

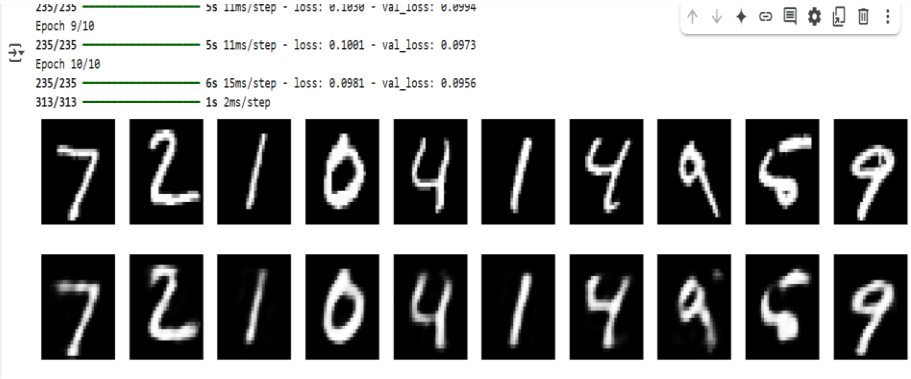
Code:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist

# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
```

```
x_test = x_test.astype('float32') / 255.0

# Flatten the images
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

# Define the autoencoder
latent_dim = 32  # Can be changed to 16 or 64 for analysis

# Encoder
input_img = Input(shape=(784,))
encoded = Dense(latent_dim, activation='relu')(input_img)

# Decoder
decoded = Dense(784, activation='sigmoid')(encoded)

# Autoencoder model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the autoencoder
autoencoder.fit(x_train, x_train, epochs=10, batch_size=256, shuffle=True,
validation_data=(x_test, x_test))

# Generate reconstructed images
reconstructed_imgs = autoencoder.predict(x_test)

# Plot original vs reconstructed images
def plot_images(original, reconstructed, n=10):
    plt.figure(figsize=(20, 4))
    for i in range(n):
        # Original images
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(original[i].reshape(28, 28), cmap='gray')
        plt.axis('off')

        # Reconstructed images
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(reconstructed[i].reshape(28, 28), cmap='gray')
        plt.axis('off')
```

```
    plt.show()

plot_images(x_test, reconstructed_imgs)
```

Output:



Q2: Implementing a Denoising Autoencoder

Task: Denoising autoencoders can reconstruct clean data from noisy inputs. You will train a model to remove noise from images.

1. Modify the **basic autoencoder** from Q2 to a **denoising autoencoder** by adding **Gaussian noise** (mean=0, std=0.5) to input images.
2. Ensure that the **output remains the clean image** while training.
3. Train the model and visualize **noisy vs. reconstructed images**.
4. Compare the **performance of a basic vs. denoising autoencoder** in reconstructing images.
5. Explain one real-world scenario where denoising autoencoders can be useful (e.g., medical imaging, security).

Hint: *Use np.random.normal() to add noise to images before training.*

Answer:

Here's a structured approach to implementing a **Denoising Autoencoder (DAE)**:

1. Modify the Basic Autoencoder to a Denoising Autoencoder:

- Instead of directly training on clean images, we will **add Gaussian noise** to the input images.
- The model learns to map noisy images to their original clean versions.

2. Training Data Preparation:

- Use np.random.normal(loc=0.0, scale=0.5, size=image.shape) to introduce Gaussian noise.
- Clip pixel values to remain in the range [0,1].

3. Training the Model:

- Use a convolutional autoencoder architecture.
- Optimize using **MSE loss**, since we aim to minimize the difference between the clean and reconstructed images.

4. Evaluation:

- Visualize:
  - Original images
  - Noisy images
  - Reconstructed images from both the **basic autoencoder** and **denoising autoencoder**.

5. Real-World Application:

- **Medical Imaging**:
  Denoising autoencoders can enhance **MRI, CT scans, and X-ray images** by removing noise and artifacts, improving the accuracy of diagnoses.

Code:

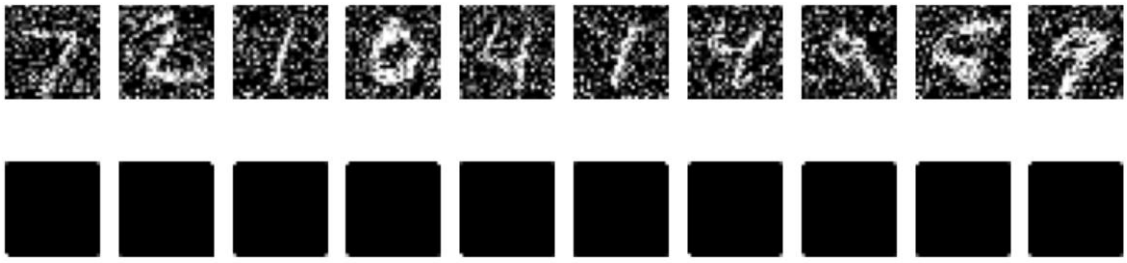
```
 import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
UpSampling2D
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
```

```
# Add Gaussian noise
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

# Define autoencoder architecture
input_img = Input(shape=(28, 28, 1))

# Encoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)

# Decoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, x)
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
autoencoder.fit(x_train_noisy, x_train, epochs=10, batch_size=128, shuffle=True,
validation_data=(x_test_noisy, x_test))

# Predict on test images
denoised_images = autoencoder.predict(x_test_noisy)

# Display results
n = 10  # Number of images to display
plt.figure(figsize=(10, 4))
for i in range(n):
```

```
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')
    plt.axis('off')

    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')
    plt.axis('off')

plt.show()
```

OutPut:

```
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 139s 273ms/step - loss: 0.1118 - val_loss: 0.1140
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 139s 268ms/step - loss: 0.1118 - val_loss: 0.1140
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step
```



Q3: Implementing an RNN for Text Generation

Task: Recurrent Neural Networks (RNNs) can generate sequences of text. You will train an **LSTM-based RNN** to predict the next character in a given text dataset.

1. Load a **text dataset** (e.g., "Shakespeare Sonnets", "The Little Prince").
2. Convert text into a **sequence of characters** (one-hot encoding or embeddings).
3. Define an **RNN model** using LSTM layers to predict the next character.
4. Train the model and generate new text by **sampling characters** one at a time.
5. Explain the role of **temperature scaling** in text generation and its effect on randomness.

Answer:

Code:

```
import tensorflow as tf
import numpy as np
import requests

# 1. Load a text dataset ("The Little Prince")
url = "https://www.gutenberg.org/cache/epub/11671/pg11671.txt"
response = requests.get(url)
text = response.text.lower()
text = text[:100000]  # Optional: truncate to speed up training

# 2. Preprocessing: character-level encoding
chars = sorted(set(text))
char2idx = {u: i for i, u in enumerate(chars)}
idx2char = np.array(chars)
text_as_int = np.array([char2idx[c] for c in text])

# Create training sequences
seq_length = 100
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
sequences = char_dataset.batch(seq_length + 1, drop_remainder=True)

def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)

# Shuffle and batch
BATCH_SIZE = 64
BUFFER_SIZE = 10000
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE,
drop_remainder=True)

# 3. Build the model
vocab_size = len(chars)
embedding_dim = 256
```

```
rnn_units = 512

def build_model(vocab_size, embedding_dim, rnn_units):
    return tf.keras.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim),
        tf.keras.layers.LSTM(rnn_units, return_sequences=True),
        tf.keras.layers.Dense(vocab_size)
    ])

model = build_model(vocab_size, embedding_dim, rnn_units)

# 4. Compile and train the model
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits,
from_logits=True)

model.compile(optimizer='adam', loss=loss)

EPOCHS = 10
model.fit(dataset, epochs=EPOCHS)

# 5. Generate text from a prompt
def generate_text(model, start_string, temperature=1.0, num_generate=500):
    input_eval = [char2idx[s] for s in start_string.lower()]
    input_eval = tf.expand_dims(input_eval, 0)
    text_generated = []

    for _ in range(num_generate):
        predictions = model(input_eval)
        predictions = predictions[:, -1, :] / temperature
        predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,
0].numpy()

        input_eval = tf.expand_dims([predicted_id], 0)
        text_generated.append(idx2char[predicted_id])

    return start_string + ''.join(text_generated)

# 6. Generate sample text
```

```
generated_text = generate_text(model, start_string="Once upon a time",
temperature=0.8)
print(generated_text)
```

Temperature Scaling Explained

What is Temperature in Text Generation?

- Temperature controls the randomness of predictions from the model's probability distribution.

How it works:

- When generating text, the model predicts a probability distribution over all possible next characters.
- This distribution is **divided by the temperature** before sampling.

Temperature Effects:

- 🔥 **Low Temperature (<1.0)**:
  - Makes the model more **confident**.
  - Generates more **predictable and repetitive** text.
- ❄️ **High Temperature (>1.0)**:
  - Increases **randomness**.
  - Can generate **creative but chaotic or grammatically incorrect** outputs.

Examples:

- temperature = 0.2 → "The the the the..."
- temperature = 1.0 → "The fox ran across the hill"
- temperature = 1.5 → "Thw zox ranz qrotfl bzil…"

Output:

Hint: *Use tensorflow.keras.layers.LSTM() for sequence modeling.*

Q4: Sentiment Classification Using RNN

Task: Sentiment analysis determines if a given text expresses a positive or negative emotion. You will train an **LSTM-based sentiment classifier** using the IMDB dataset.

1. Load the **IMDB sentiment dataset** (tensorflow.keras.datasets.imdb).
2. Preprocess the text data by **tokenization** and **padding** sequences.
3. Train an **LSTM-based model** to classify reviews as **positive or negative**.
4. Generate a **confusion matrix** and classification report (accuracy, precision, recall, F1-score).
5. Interpret why **precision-recall tradeoff** is important in sentiment classification.

Hint: *Use confusion_matrix and classification_report from sklearn.metrics*

Answer:
Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from sklearn.metrics import confusion_matrix, classification_report

# 1. Load the IMDB sentiment dataset
num_words = 10000  # Only consider the top 10,000 words
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=num_words)

# 2. Preprocess the data: pad sequences
maxlen = 200  # Max length of review
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)

# 3. Build the LSTM model
model = Sequential([
```

```
    Embedding(input_dim=num_words, output_dim=128, input_length=maxlen),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])

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

# Train the model
model.fit(x_train, y_train,
          epochs=3,
          batch_size=64,
          validation_split=0.2,
          verbose=1)

# 4. Evaluate the model and generate classification metrics
y_pred_prob = model.predict(x_test)
y_pred = (y_pred_prob > 0.5).astype("int32")

# Generate confusion matrix and classification report
cm = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=["Negative",
"Positive"])

print("Confusion Matrix:\n", cm)
print("\nClassification Report:\n", report)
```

Interpretation: Why is the Precision-Recall Tradeoff Important?
In **sentiment classification**, especially with user-generated content like reviews:

- **Precision** matters when **false positives** are costly — for example, if you wrongly classify a *negative* review as *positive*, it may falsely signal satisfaction.
- **Recall** matters when **false negatives** are costly — for example, if many *positive* reviews are misclassified as *negative*, you may fail to recognize good customer feedback.

💡 **The Tradeoff**:

- Increasing **precision** often reduces **recall**, and vice versa.
- In real-world applications, the priority depends on your use case:

- For **customer service**, high **recall** ensures no negative feedback is missed.
- For **marketing analysis**, high **precision** avoids acting on false positives.

OutPut:

```
Confusion Matrix:
 [[10524  1976]
 [ 1625 10875]]

Classification Report:
              precision    recall  f1-score   support

    Negative       0.87      0.84      0.85     12500
    Positive       0.85      0.87      0.86     12500

    accuracy                           0.86     25000
   macro avg       0.86      0.86      0.86     25000
weighted avg       0.86      0.86      0.86     25000
```