Training Day-80 Report:

1. Encoder-Decoder Architecture

The Encoder-Decoder architecture is commonly used for tasks like machine translation, summarization, and image captioning. It transforms an input sequence into a fixed-size context vector (using the encoder) and then generates an output sequence (using the decoder).

Components of Encoder-Decoder Architecture:

1. **Encoder**:
   - Processes the input sequence.
   - Outputs a context vector summarizing the sequence.
   - Typically implemented with RNNs, LSTMs, or GRUs.
2. **Decoder**:
   - Takes the context vector as input.
   - Generates the output sequence one step at a time.
3. **Attention Mechanism** (optional but widely used):
   - Enhances performance by allowing the decoder to focus on relevant parts of the input sequence dynamically.

Implementation: Machine Translation Example

```
import tensorflow as tf

from tensorflow.keras.layers import Input, LSTM, Dense

from tensorflow.keras.models import Model


# Define model parameters

latent_dim = 256  # Latent dimensionality for LSTM layers

num_encoder_tokens = 1000  # Vocabulary size for input

num_decoder_tokens = 1000  # Vocabulary size for output


# Encoder

encoder_inputs = Input(shape=(None, num_encoder_tokens))

encoder_lstm = LSTM(latent_dim, return_state=True)
```

```
encoder_outputs, state_h, state_c = encoder_lstm(encoder_inputs)

# Decoder
decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=[state_h, state_c])
decoder_dense = Dense(num_decoder_tokens, activation="softmax")
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer="adam", loss="categorical_crossentropy")

# Model Summary
model.summary()

# Training would require preprocessed input-output pairs (not shown here)
# model.fit([encoder_input_data, decoder_input_data], decoder_target_data, epochs=10,
batch_size=64)
```

2. Generative Adversarial Networks (GANs)

GANs consist of two networks: a **Generator** and a **Discriminator**, trained adversarially to generate realistic data.

Components of GAN:

1. **Generator**:
   - Takes random noise as input and generates fake data.
2. **Discriminator**:
   - Distinguishes between real and fake data.
3. **Adversarial Training**:
   - The generator aims to fool the discriminator, while the discriminator aims to identify fake data.

Steps for Training GANs:

1. Train the **discriminator**:
   - On real data labeled as 1.
   - On fake data generated by the generator, labeled as 0.
2. Train the **generator**:
   - Generate fake data and pass it to the discriminator.
   - Update the generator to maximize the discriminator's classification error on fake data.

Implementation: GAN for Image Generation

```
import tensorflow as tf

from tensorflow.keras.layers import Dense, LeakyReLU, Reshape, Flatten

from tensorflow.keras.models import Sequential

import numpy as np


# Define generator model

def build_generator(latent_dim):

    model = Sequential([

        Dense(128, activation=LeakyReLU(0.2), input_dim=latent_dim),

        Dense(256, activation=LeakyReLU(0.2)),

        Dense(512, activation=LeakyReLU(0.2)),

        Dense(28 * 28 * 1, activation='tanh'),  # Output: 28x28 image

        Reshape((28, 28, 1))

    ])

    return model


# Define discriminator model

def build_discriminator(input_shape):

    model = Sequential([

        Flatten(input_shape=input_shape),

        Dense(512, activation=LeakyReLU(0.2)),
```

```
        Dense(256, activation=LeakyReLU(0.2)),
        Dense(1, activation='sigmoid')  # Output: Real or Fake
    ])
    return model

# Parameters
latent_dim = 100
image_shape = (28, 28, 1)

# Instantiate generator and discriminator
generator = build_generator(latent_dim)
discriminator = build_discriminator(image_shape)

# Compile discriminator
discriminator.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

# Combined model (for training the generator)
discriminator.trainable = False
gan = Sequential([generator, discriminator])
gan.compile(optimizer="adam", loss="binary_crossentropy")

# Training
def train_gan(generator, discriminator, gan, epochs, batch_size):
    (X_train, _), _ = tf.keras.datasets.mnist.load_data()
    X_train = (X_train.astype("float32") - 127.5) / 127.5  # Normalize to [-1, 1]
    X_train = np.expand_dims(X_train, axis=-1)

    real_labels = np.ones((batch_size, 1))
    fake_labels = np.zeros((batch_size, 1))
```

```
    for epoch in range(epochs):
        # Train discriminator
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        real_images = X_train[idx]
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        fake_images = generator.predict(noise)

        d_loss_real = discriminator.train_on_batch(real_images, real_labels)
        d_loss_fake = discriminator.train_on_batch(fake_images, fake_labels)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train generator
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        g_loss = gan.train_on_batch(noise, real_labels)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: D Loss = {d_loss[0]}, G Loss = {g_loss}")

train_gan(generator, discriminator, gan, epochs=1000, batch_size=64)
```

Key Takeaways:

1. **Encoder-Decoder**:
    - Converts input sequences into context vectors and generates output sequences.
    - Useful for translation, summarization, etc.
2. **GANs**:
    - Generates realistic data by training a generator and discriminator adversarially.
    - Applications: Image generation, style transfer, etc.