Training Day-76 Report:

1. Window Padding

Padding is a technique in convolutional neural networks (CNNs) to control the output size of the convolution operation. It adds extra borders (usually filled with zeros) to the input data to ensure certain properties in the output dimensions.

Types of Padding:

1. **Valid Padding (No Padding):**
    - No extra padding is added.
    - Output size decreases with each convolution.
    - Formula for output size:
      Output Size=Input Size−Kernel SizeStride+1\text{Output Size} = \frac{\text{Input Size} - \text{Kernel Size}}{\text{Stride}} + 1
2. **Same Padding:**
    - Padding is added to ensure the output size is the same as the input size.
    - Formula for padding size: P=⌈Kernel Size−12⌉P = \left\lceil \frac{\text{Kernel Size} - 1}{2} \right\rceil

Example:

For a 5×55 \times 5 input and a 3×33 \times 3 kernel with stride 1:

- **Valid Padding**: 3×33 \times 3 output.
- **Same Padding**: 5×55 \times 5 output.

TensorFlow Example:

```
import tensorflow as tf

# Dummy input data
input_data = tf.constant([[[[1], [2], [3]], [[4], [5], [6]], [[7], [8], [9]]]], dtype=tf.float32)

# Convolution layer with 'valid' padding
conv_valid = tf.keras.layers.Conv2D(1, (3, 3), strides=(1, 1), padding='valid')
output_valid = conv_valid(input_data)

# Convolution layer with 'same' padding
conv_same = tf.keras.layers.Conv2D(1, (3, 3), strides=(1, 1), padding='same')
output_same = conv_same(input_data)

print("Valid Padding Output Shape:", output_valid.shape)
```

```
print("Same Padding Output Shape:", output_same.shape)
```

2. Image Classification Using CNN

Image classification involves assigning a label to an image based on its content.

Steps to Implement Image Classification:

1. **Load and Preprocess Data**: Use datasets like CIFAR-10, MNIST, or custom datasets.
2. **Define a CNN Model**: Use convolutional, pooling, and dense layers.
3. **Train the Model**: Use labeled data to optimize the model's performance.
4. **Evaluate and Predict**: Test the model on unseen data.

Example Implementation:

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load and preprocess MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1) / 255.0  # Normalize and reshape
x_test = x_test.reshape(-1, 28, 28, 1) / 255.0

# Define CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # Output layer for 10 classes
])
```

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


# Train the model
model.fit(x_train, y_train, epochs=10, validation_split=0.1)


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


# Predict on new data
predictions = model.predict(x_test[:5])
print("Predictions:", tf.argmax(predictions, axis=1).numpy())
```

Key Points:

- **Padding** ensures control over output dimensions, especially in deep networks.
- **Image Classification** involves training CNNs to recognize patterns in images and assign labels.
- Use libraries like TensorFlow and datasets like MNIST for practice.

Training Day-77 Report:

1. Convolutional Neural Networks (CNNs)

CNNs are primarily used for processing grid-like data such as images. They excel at feature extraction, learning hierarchical patterns like edges, textures, and shapes, making them ideal for tasks like image classification, object detection, and more.

Key Components of CNNs:

1. **Convolutional Layer**:
    - Applies filters (kernels) to the input data.
    - Outputs feature maps highlighting specific features.
    - Formula for output size: O=I−K+2PS+1O = \frac{I - K + 2P}{S} + 1 Where:
        - OO: Output size
        - II: Input size
        - KK: Kernel size
        - PP: Padding
        - SS: Stride
2. **Activation Function**:
    - Introduces non-linearity, e.g., ReLU (max⁡(0,x)\max(0, x)).
3. **Pooling Layer**:
    - Reduces spatial dimensions using techniques like MaxPooling.
4. **Fully Connected Layer**:
    - Connects all neurons from the previous layer for classification or regression.

Example CNN in TensorFlow:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Define a CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
```

```
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # Output for 10 classes
])

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

# Summary
model.summary()
```

2. Recurrent Neural Networks (RNNs)

RNNs are designed for sequential data like time series, text, and audio. Unlike feedforward networks, RNNs retain a "memory" of past inputs, making them effective for handling temporal dependencies.

How RNNs Work:

1. **Input Sequence**: Data is processed one time step at a time.
2. **Hidden State**: The hidden state captures information about previous time steps, allowing the network to maintain a memory of past events.
3. **Output**: Each time step generates an output based on the current input and the hidden state.

Challenges:

- **Vanishing Gradient Problem**: Difficulty in learning long-term dependencies.
- **Solutions**: Use variants like Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU).

Example RNN in TensorFlow:

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

```
from tensorflow.keras.layers import SimpleRNN, Dense

# Dummy sequential data
X = tf.random.normal([100, 10, 8])  # 100 samples, 10 timesteps, 8 features
y = tf.random.uniform([100], maxval=2, dtype=tf.int32)  # Binary labels

# Define an RNN model
model = Sequential([
    SimpleRNN(32, activation='tanh', input_shape=(10, 8)),
    Dense(1, activation='sigmoid')
])

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

# Train the model
model.fit(X, y, epochs=5, batch_size=16)

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

Key Differences: CNNs vs. RNNs

Feature	CNN	RNN
Input Type	Spatial data (e.g., images)	Sequential data (e.g., text, time series)
Architecture	Filters and pooling layers	Recurrent connections with memory
Use Case	Image classification, object detection	Language modeling, time series prediction

---

Training Day-78 Report:

1. Long Short-Term Memory (LSTM) Architecture

LSTMs are an advanced type of Recurrent Neural Network (RNN) designed to overcome the vanishing gradient problem and handle long-term dependencies in sequences. They achieve this with a memory cell and gating mechanisms.

LSTM Components:

1. **Cell State** (CtC_t):
    - The "memory" of the network.
    - Modified by input, forget, and output gates.
2. **Gates**:
    - **Forget Gate**: Decides what information to discard. ft=σ(Wf·[ht−1,xt]+bf)f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)
    - **Input Gate**: Decides what new information to store. it=σ(Wi·[ht−1,xt]+bi)i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) C~t=tanh⁡(WC·[ht−1,xt]+bC)\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) Update the cell state: Ct=ft·Ct−1+it·C~tC_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t
    - **Output Gate**: Determines what part of the cell state contributes to the output. ot=σ(Wo·[ht−1,xt]+bo)o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) ht=ot·tanh⁡(Ct)h_t = o_t \cdot \tanh(C_t)
3. **Hidden State** (hth_t):
    - The output at time tt, influenced by the cell state and output gate.

2. Building a Story Writer using Character-Level RNN

A character-level RNN generates text one character at a time by predicting the next character given the current sequence.

Steps:

1. **Prepare the Data**: Convert text into a sequence of integers, each representing a character.
2. **Build the Model**: Use an LSTM layer to process the character sequences.
3. **Train the Model**: Train the RNN to predict the next character in a sequence.

4. **Generate Text**: Use the trained model to generate text by sampling predictions iteratively.

Implementation: Character-Level RNN with LSTM

```
import tensorflow as tf
import numpy as np

# Sample text data
text = "Once upon a time in a faraway land, there was a little girl named Red Riding Hood."
chars = sorted(set(text))  # Unique characters
char_to_idx = {char: idx for idx, char in enumerate(chars)}
idx_to_char = {idx: char for char, idx in char_to_idx.items()}

# Convert text to integer sequence
encoded_text = np.array([char_to_idx[char] for char in text])

# Prepare input-output pairs
seq_length = 40
input_sequences = []
output_sequences = []
for i in range(len(encoded_text) - seq_length):
    input_sequences.append(encoded_text[i:i + seq_length])
    output_sequences.append(encoded_text[i + seq_length])

input_sequences = np.array(input_sequences)
output_sequences = np.array(output_sequences)

# One-hot encode the input and output
vocab_size = len(chars)
X = tf.keras.utils.to_categorical(input_sequences, num_classes=vocab_size)
y = tf.keras.utils.to_categorical(output_sequences, num_classes=vocab_size)
```

```
# Build the LSTM model
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(256, input_shape=(seq_length, vocab_size), return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.LSTM(256),
    tf.keras.layers.Dense(vocab_size, activation='softmax')
])

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

# Train the model
model.fit(X, y, epochs=20, batch_size=64)

# Generate text
def generate_text(model, start_string, gen_length=100):
    input_seq = [char_to_idx[char] for char in start_string]
    input_seq = tf.keras.utils.to_categorical(input_seq, num_classes=vocab_size).reshape(1, -1, vocab_size)

    generated_text = start_string
    for _ in range(gen_length):
        pred = model.predict(input_seq, verbose=0)
        next_char = idx_to_char[np.argmax(pred)]
        generated_text += next_char

        # Update input sequence
        next_input = tf.keras.utils.to_categorical([np.argmax(pred)], num_classes=vocab_size)
```

```
    input_seq = np.concatenate((input_seq[:, 1:, :], next_input.reshape(1, 1, vocab_size)), axis=1)

  return generated_text

# Generate a story snippet
start = "Once upon a time"
print(generate_text(model, start))
```

Key Points:

1. **LSTM**:
    - Manages long-term dependencies effectively using gates.
2. **Character-Level RNN**:
    - Generates text by learning patterns in sequences of characters.
3. **Applications**:
    - Chatbots, story generation, code autocompletion, and more.

Training Day-79 Report:

1. Sentiment Analysis Hands-On

Sentiment Analysis is a common NLP task where the goal is to determine the sentiment (e.g., positive, negative, neutral) of a given text.

Steps for Sentiment Analysis:

1. **Data Preparation**: Use a dataset like IMDb Movie Reviews or any labeled sentiment dataset.
2. **Preprocessing**:
    - Tokenization.
    - Padding sequences to a fixed length.
    - Encoding labels.
3. **Building the Model**: Use an Embedding layer followed by LSTM/GRU or CNN for feature extraction.
4. **Training**: Train the model to classify sentiments.
5. **Evaluation**: Test the model on unseen data.

Implementation in TensorFlow:

```
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, Dropout


# Load IMDb dataset

vocab_size = 10000

max_len = 200

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)


# Pad sequences to ensure uniform input length

x_train = pad_sequences(x_train, maxlen=max_len)
```

```
x_test = pad_sequences(x_test, maxlen=max_len)

# Build the sentiment analysis model
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len),  # Embedding layer
    LSTM(64, return_sequences=False),                    # LSTM layer
    Dropout(0.5),                                        # Dropout for regularization
    Dense(1, activation='sigmoid')                       # Output layer for binary classification
])

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

# Train the model
model.fit(x_train, y_train, epochs=3, batch_size=64, validation_data=(x_test, y_test))

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

# Make predictions
sample_review = "The movie was fantastic and very engaging!"
encoded_review = [1] + [word for word in sample_review.split() if word in imdb.get_word_index()]
padded_review = pad_sequences([encoded_review], maxlen=max_len)
prediction = model.predict(padded_review)
print(f"Sentiment: {'Positive' if prediction[0][0] > 0.5 else 'Negative'}")
```

2. Seq-to-Seq Model

Sequence-to-Sequence (Seq2Seq) models are used for tasks like machine translation, summarization, and chatbot applications. These models take a sequence as input and generate another sequence as output.

Seq-to-Seq Architecture:

1. **Encoder**:
    - Encodes the input sequence into a fixed-length context vector.
    - Typically uses RNNs, LSTMs, or GRUs.
2. **Decoder**:
    - Takes the context vector as input and generates the output sequence.
3. **Attention Mechanism**:
    - Enhances performance by allowing the decoder to focus on specific parts of the input sequence at each time step.

Seq-to-Seq Implementation for Machine Translation:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, LSTM, Dense
from tensorflow.keras.models import Model
import numpy as np

# Sample data
input_texts = ["hello", "how are you", "good morning"]
target_texts = ["hola", "cómo estás", "buenos días"]

# Tokenize data
input_tokenizer = tf.keras.preprocessing.text.Tokenizer()
target_tokenizer = tf.keras.preprocessing.text.Tokenizer()

input_tokenizer.fit_on_texts(input_texts)
target_tokenizer.fit_on_texts(target_texts)

input_sequences = input_tokenizer.texts_to_sequences(input_texts)
```

```
target_sequences = target_tokenizer.texts_to_sequences(target_texts)

input_data = tf.keras.preprocessing.sequence.pad_sequences(input_sequences, padding='post')
target_data = tf.keras.preprocessing.sequence.pad_sequences(target_sequences, padding='post')

# Define model parameters
num_encoder_tokens = len(input_tokenizer.word_index) + 1
num_decoder_tokens = len(target_tokenizer.word_index) + 1
latent_dim = 256

# Encoder
encoder_inputs = Input(shape=(None,))
encoder_embedding = Dense(64, activation='relu')(encoder_inputs)
encoder_lstm = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(encoder_embedding)

# Decoder
decoder_inputs = Input(shape=(None,))
decoder_embedding = Dense(64, activation='relu')(decoder_inputs)
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=[state_h, state_c])
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# Define model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')

# Training
```

```
model.fit([input_data, target_data[:, :-1]], target_data[:, 1:], batch_size=64, epochs=50)


# Inference
def translate(input_seq):
    states = encoder_lstm.predict(input_seq)
    translated_seq = decoder_lstm.predict(states)
    return translated_seq


print("Translation:", translate(input_data[0:1]))
```

Summary:

1. **Sentiment Analysis**:
    - Used LSTM for sequence-based binary classification.
    - Applied text preprocessing, tokenization, and padding.
2. **Seq-to-Seq**:
    - Demonstrated architecture for sequence translation tasks.
    - Encoder-decoder structure handles input-to-output sequence transformation.

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.