# Pianalytix

# DEEP LEARNING MASTERY CHEAT SHEET

(140+ Code Snippets to 10x your Productivity)

## 1 Introduction

**Definitions**

- Deep learning is a subfield of machine learning that focuses on artificial neural networks with multiple layers.
- Neural networks are computational models inspired by the structure and function of the human brain.
- Deep learning allows models to learn hierarchical representations of data, enabling them to discover complex patterns.
- Deep learning can be implemented using various libraries, such as:
  - **Tensorflow**
  - **Keras**
  - **Pytorch**
  - **Theano**
  - **Caffe**

**Concepts**

- **Convolutional Neural Networks (CNNs):** Specialized neural networks for processing grid-like data, such as images.
- **Recurrent Neural Networks (RNNs):** Neural networks designed for processing sequential data, where the output depends on previous inputs.
- **Generative Adversarial Networks (GANs):** Neural network models consisting of a generator and discriminator in a competitive training framework to generate realistic data.
- **Batch Normalization:** A technique that normalizes the inputs of each layer to accelerate training and improve generalization.
- **Dropout:** A regularization technique that randomly drops a fraction of neuron activations during training to prevent overfitting.

## 2 Data Preprocessing

### 2.1 Definitions - *Sequence Padding*

**Sequence Padding**

The process of adding padding (usually zeros) to sequences to make them of equal length, ensuring compatibility for deep learning models.

- **Optimization:** Techniques and approaches to efficiently handle sequence padding during training and inference to minimize computational overhead.
- **Implementation:** Incorporating padding operations into the data preprocessing pipeline to ensure consistent sequence lengths for input to sequence-based models.
- **Strategies:** Various methods and considerations for handling padding in different scenarios, such as handling variable-length sequences, padding positions, or handling padding in recurrent neural networks.

### 2.2 Code Snippets - *Sequence Padding*

**Sequence Padding**

```python
# Keras
from tensorflow.keras.preprocessing.sequence import pad_sequences

sequences = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
padded_sequences = pad_sequences(sequences, padding='post')

# TensorFlow
import tensorflow as tf

sequences = tf.constant([[1, 2, 3], [4, 5], [6, 7, 8, 9]])
padded_sequences = tf.keras.preprocessing.sequence.pad_sequences(sequences, padding='post')
```

**Optimization**

```python
# Padding optimization using tf.data API
padded_dataset = dataset.padded_batch(batch_size, padded_shapes=([None], []))
```

**Implementation**

```python
# Padding implementation using Keras preprocessing
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Padding sequences to a maximum length
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post')
```

**Strategies**

```python
# Padding strategy: Pre-padding for RNNs
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
```

### 2.3 Definitions - *One-Hot Encoding*

**One-Hot Encoding**

A technique to convert categorical variables into binary vectors, where each category is represented by a binary value in a vector.

- **Optimization:** Techniques and approaches to efficiently handle One-Hot Encoding for large-scale categorical data, considering memory and computational efficiency.
- **Implementation:** Applying the process of converting categorical variables into binary vectors where each unique category is represented as a separate binary feature.
- **Strategies:** Various methods and considerations for handling special cases during One-Hot Encoding, such as dealing with unseen categories, handling high-cardinality variables, or handling categorical features with a large number of unique values.

### 2.4 Code Snippets - *One-Hot Encoding*

**One-Hot Encoding**

```python
# Keras
from tensorflow.keras.utils import to_categorical

categories = [1, 2, 3, 1, 3]
one_hot_encoded = to_categorical(categories)

# TensorFlow
import tensorflow as tf

categories = tf.constant([1, 2, 3, 1, 3])
one_hot_encoded = tf.one_hot(categories, depth=tf.reduce_max(categories) + 1)
```

**Optimization**

```python
# One-Hot Encoding optimization using TensorFlow one_hot function
encoded_labels = tf.one_hot(labels, num_classes)
```

**Implementation**

```python
# One-Hot Encoding implementation using Keras preprocessing
from tensorflow.keras.utils import to_categorical

# Convert integer labels to one-hot encoded vectors
encoded_labels = to_categorical(labels, num_classes)
```

**Strategies**

```python
# One-Hot Encoding strategy: Handling unseen categories
from sklearn.preprocessing import OneHotEncoder

# Initialize OneHotEncoder and handle unknown categories during encoding
encoder = OneHotEncoder(handle_unknown='ignore')
encoded_data = encoder.fit_transform(data)
```

### 2.5 Definitions - *Feature Scaling*

**Feature Scaling**

The normalization of feature values to a consistent range (e.g., [0, 1] or [-1, 1]) to prevent certain features from dominating the learning process.

- **Optimization:** Techniques and approaches to efficiently scale features for improved performance and convergence during machine learning algorithms.
- **Implementation:** Applying the process of transforming feature values to a consistent scale or range to ensure fairness and comparability across different features.
- **Strategies:** Various methods and considerations for feature scaling, such as standardization (z-score normalization), normalization (min-max scaling), or robust scaling, based on the characteristics of the data and the requirements of the learning algorithm.

### 2.6 Code Snippets - *Feature Scaling*

**Feature Scaling**

```python
# Keras
from tensorflow.keras import layers
from sklearn.preprocessing import MinMaxScaler

data = [[2.0, -1.0], [1.0, 0.5], [3.0, 1.0]]
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)

# TensorFlow
import tensorflow as tf

data = tf.constant([[2.0, -1.0], [1.0, 0.5], [3.0, 1.0]])
scaled_data = tf.keras.layers.LayerNormalization()(data)
```

**Optimization**

```python
# Feature Scaling optimization using TensorFlow preprocessing layers
from tensorflow.keras import layers

# Define and apply the normalization layer to the input data
normalization_layer = layers.Normalization()
scaled_data = normalization_layer(data)
```

**Implementation**

```python
# Feature Scaling implementation using Keras preprocessing
from tensorflow.keras.preprocessing import MinMaxScaler

# Initialize the scaler and fit-transform the data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
```

**Strategies**

```python
# Feature Scaling strategy: Standardization (z-score normalization)
from sklearn.preprocessing import StandardScaler

# Initialize the scaler and perform standardization
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
```

### 2.7 Definitions - *Data Augmentation*

**Data Augmentation**

Techniques to artificially expand the training dataset by applying transformations such as rotation, flipping, cropping, or adding noise to increase data diversity.

- **Optimization:** Techniques and approaches to efficiently generate augmented data to increase the diversity and size of the training dataset for improved model generalization and performance.
- **Implementation:** Applying various operations or transformations to the original data to create augmented samples, such as rotation, flipping, cropping, or adding noise, to introduce variability and reduce overfitting.
- **Strategies:** Different methods and considerations for data augmentation, including the selection of appropriate augmentation techniques, the extent of augmentation, and balancing between augmenting the data and preserving the original information.

### 2.8 Code Snippets - *Data Augmentation*

**Data Augmentation**

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

# Data Augmentation using Keras
data_generator = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
augmented_images = data_generator.flow_from_directory(
    'path/to/dataset',
    batch_size=32,
    class_mode='categorical',
    target_size=(224, 224)
)

# Data Augmentation using TensorFlow
@tf.function
def augment_data(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    image = tf.image.random_brightness(image, max_delta=0.1)
    image = tf.image.random_contrast(image, lower=0.9, upper=1.1)
    return image, label

augmented_dataset = original_dataset.map(augment_data)
```

**Optimization**

```python
# Data augmentation optimization using TensorFlow ImageDataGenerator
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Configure data augmentation options for images
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

**Implementation**

```python
# Data augmentation implementation using Keras ImageDataGenerator
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Configure data augmentation options for images
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Apply data augmentation to an image dataset
augmented_data = datagen.flow(x_train, y_train, batch_size=batch_size)
```

**Strategies**

```python
# Data augmentation strategy: Random cropping and resizing
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Configure data augmentation options for images
datagen = ImageDataGenerator(
    preprocessing_function=lambda img: tf.image.random_crop(tf.image.resize(img, [224, 224]),
    size=[200, 200, 3]),
    horizontal_flip=True,
    fill_mode='nearest'
)
```

### 2.9 Definitions - *Tokenization*

**Tokenization**

The process of breaking text or sentences into individual tokens (words, subwords, or characters) for natural language processing tasks.

- **Optimization:** Techniques and approaches to efficiently tokenize text data, considering memory usage and computational efficiency.
- **Implementation:** Applying the process of breaking down text into individual tokens or words, often removing punctuation and lowercasing, to prepare it for natural language processing tasks.
- **Strategies:** Various methods and considerations for tokenization, such as handling special cases like contractions, incorporating subword tokenization techniques like Byte Pair Encoding (BPE), or using specific tokenizers for different languages or domains.

### 2.10 Code Snippets - *Tokenization*

**Tokenization**

```python
# Tokenization - Keras
from tensorflow.keras.preprocessing.text import Tokenizer

texts = ['Hello, how are you?', 'I am doing great!', 'This is an example sentence.']
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
word_index = tokenizer.word_index
sequences = tokenizer.texts_to_sequences(texts)

# Tokenization - TensorFlow
from tensorflow.keras.preprocessing.text import Tokenizer

texts = ['Hello, how are you?', 'I am doing great!', 'This is an example sentence.']
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
word_index = tokenizer.word_index
sequences = tokenizer.texts_to_sequences(texts)
```

**Optimization**

```python
# Tokenization optimization using TensorFlow Text
import tensorflow_text as tf_text

# Initialize a tokenizer for efficient tokenization
tokenizer = tf_text.UnicodeScriptTokenizer()
tokenized_text = tokenizer.tokenize(input_text)
```

**Implementation**

```python
# Tokenization implementation using Keras preprocessing
from tensorflow.keras.preprocessing.text import Tokenizer

# Initialize the tokenizer and fit on text data
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)

# Convert text to sequences of tokens
sequences = tokenizer.texts_to_sequences(texts)

# Apply data augmentation to an image dataset
augmented_data = datagen.flow(x_train, y_train, batch_size=batch_size)
```

**Strategies**

```python
# Tokenization strategy: Word-level tokenization with custom rules
import re

def custom_tokenizer(text):
    # Custom rules for tokenization
    tokens = re.findall(r'\w+', text.lower())
    return tokens

# Apply custom tokenizer to text
tokenized_text = [custom_tokenizer(text) for text in texts]
```

### 2.11 Definitions - *Word Embeddings*

**Word Embeddings**

Techniques to represent words as dense vectors in a continuous vector space, capturing semantic relationships and contextual information.

- **Optimization:** Techniques and approaches to efficiently represent words as dense vector representations, considering factors like dimensionality reduction, computational efficiency, and capturing semantic relationships.
- **Implementation:** Applying the process of mapping words to continuous vector representations in order to capture semantic and syntactic relationships between words in a text corpus.
- **Strategies:** Various methods and considerations for word embeddings, such as pre-trained embeddings like Word2Vec, GloVe, or fastText, fine-tuning embeddings for a specific task, or utilizing subword embeddings like Byte Pair Encoding (BPE) for handling out-of-vocabulary words.

### 2.12 Code Snippets - *Word Embeddings*

**Word Embeddings**

```python
# Word Embeddings - Keras
from tensorflow.keras.layers import Embedding

embedding_dim = 100
embedding_matrix = np.random.random((len(word_index) + 1, embedding_dim))
embedding_layer = Embedding(len(word_index) + 1, embedding_dim, weights=[embedding_matrix],
trainable=True)

# Word Embeddings - TensorFlow
import tensorflow as tf
from tensorflow.keras.layers import Embedding

embedding_dim = 100
embedding_matrix = np.random.random((len(word_index) + 1, embedding_dim))
embedding_layer = Embedding(len(word_index) + 1, embedding_dim,
embeddings_initializer=tf.keras.initializers.Constant(embedding_matrix), trainable=True)
```

**Optimization**

```python
# Word embedding optimization using TensorFlow Embedding layer
embedding_layer = tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_seq_length)
```

**Implementation**

```python
# Word embedding implementation using Keras Embedding layer
embedding_layer = keras.layers.Embedding(input_dim=vocab_size, output_dim=embedding_dim,
input_length=max_seq_length)
```

**Strategies**

```python
# Word embedding strategy: Pre-trained GloVe embeddings
import numpy as np

# Load pre-trained GloVe embeddings
glove_embeddings_index = {}
with open('glove.6B.100d.txt', encoding='utf8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefficients = np.asarray(values[1:], dtype='float32')
        glove_embeddings_index[word] = coefficients

# Create embedding matrix for the words in your vocabulary
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in word_index.items():
    embedding_vector = glove_embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

# Use the embedding matrix in your model
embedding_layer = keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_seq_length,
weights=[embedding_matrix], trainable=False)
```

## 2.13 Definitions - *Dimensionality Reduction*

**Dimensionality Reduction**

Methods to reduce the number of input features while preserving important information, such as Principal Component Analysis (PCA) or t-SNE.

- **Optimization:** Techniques and approaches to efficiently reduce the number of features or dimensions in a dataset while preserving the most relevant information, aiming to improve computational efficiency and mitigate the curse of dimensionality.
- **Implementation:** Applying mathematical and statistical methods to transform high-dimensional data into a lower-dimensional representation, such as Principal Component Analysis (PCA), t-Distributed Stochastic Neighbor Embedding (t-SNE), or Linear Discriminant Analysis (LDA).
- **Strategies:** Various methods and considerations for dimensionality reduction, such as selecting the appropriate algorithm based on the properties of the data, evaluating the explained variance or information loss, or combining multiple techniques to achieve the desired reduction while maintaining the interpretability or discriminative power of the features.

## 2.14 Code Snippets - *Dimensionality Reduction*

**Dimensionality Reduction**

```
# Dimensionality Reduction - Keras (Using PCA from scikit-learn)
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca_features = pca.fit_transform(X)

# Dimensionality Reduction - TensorFlow (Using PCA from scikit-learn)
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca_features = pca.fit_transform(X)
```

**Optimization**

```
# Dimensionality reduction optimization using TensorFlow t-SNE implementation
import tensorflow as tf

# Perform t-SNE on high-dimensional data
tsne = tf.manipulated_and_squash(data, perplexity=30, learning_rate=100)
```

**Implementation**

```
# Dimensionality reduction implementation using Keras PCA implementation
from tensorflow.keras import layers

# Apply Principal Component Analysis (PCA) to reduce dimensionality
pca = layers.PCA(n_components=2)
reduced_data = pca.fit_transform(data)
```

**Strategies**

```
# Dimensionality reduction strategy: Non-negative Matrix Factorization (NMF)
from sklearn.decomposition import NMF

# Apply NMF for dimensionality reduction
nmf = NMF(n_components=2)
reduced_data = nmf.fit_transform(data)
```

## 2.15 Definitions - *Handling Outliers*

**Handling Outliers**

Strategies for identifying and handling outlier values in the data, such as removing outliers, transforming them, or imputing with appropriate values.

- **Optimization:** Techniques and approaches to efficiently identify and handle outliers in a dataset, considering factors such as computational efficiency, robustness to noise, and impact on downstream analysis or modeling.
- **Implementation:** Applying statistical methods or algorithms to detect and treat outliers, such as using measures like z-score, modified z-score, or percentile-based methods to identify outliers, and performing actions like removal, imputation, or transformation to mitigate their influence.
- **Strategies:** Various methods and considerations for handling outliers, such as using domain knowledge to determine if outliers are valid or erroneous, applying outlier detection algorithms like isolation forest or DBSCAN, or utilizing robust statistical techniques like median absolute deviation (MAD) or trimming.

## 2.16 Code Snippets - *Handling Outliers*

**Handling Outliers**

```
# Handling Outliers - Keras (Using StandardScaler from scikit-learn)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

# Handling Outliers - TensorFlow (Using StandardScaler from scikit-learn)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
```

**Optimization**

```
# Outlier handling optimization using TensorFlow statistical functions
import tensorflow as tf

# Detect outliers using statistical methods
mean, variance = tf.nn.moments(data, axes=0)
z_scores = tf.abs((data - mean) / tf.math.sqrt(variance))
outlier_indices = tf.where(z_scores > threshold)
```

**Implementation**

```
# Outlier handling implementation using Keras' outlier detection
from tensorflow.keras import layers

# Detect outliers using Tukey's fences
outlier_layer = layers.OutlierDetection(fence_factor=1.5)
filtered_data = outlier_layer(data)
```

**Strategies**

```
# Outlier handling strategy: Robust Z-Score
from sklearn.preprocessing import RobustScaler

# Scale features using RobustScaler to handle outliers
scaler = RobustScaler()
scaled_data = scaler.fit_transform(data)
```

## 2.17 Definitionss - *Label Encoding*

**Label Encoding**

Encoding categorical labels as numeric values to prepare them for training deep learning models.

- **Optimization:** Techniques and approaches to efficiently convert categorical labels into numerical representations to facilitate machine learning algorithms, considering factors such as memory usage, computational efficiency, and preservation of label information.
- **Implementation:** Applying the process of mapping categorical labels to numerical values, such as using integer encoding or ordinal encoding, to represent categorical variables as numerical features for modeling tasks.
- **Strategies:** Various methods and considerations for label encoding, such as handling label imbalance, utilizing one-hot encoding for nominal variables, or incorporating techniques like target encoding or frequency encoding to capture label statistics in the encoding process.

## 2.18 Code Snippets - *Label Encoding*

**Label Encoding**

```
# Label Encoding - Keras (Using LabelEncoder from scikit-learn)
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
encoded_labels = encoder.fit_transform(labels)

# Label Encoding - TensorFlow (Using LabelEncoder from scikit-learn)
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
encoded_labels = encoder.fit_transform(labels)
```

**Optimization**

```
# Label encoding optimization using TensorFlow preprocessing layers
import tensorflow as tf
from tensorflow.keras import layers

# Initialize a StringLookup layer for label encoding
label_encoder = tf.keras.layers.StringLookup()

# Fit the label encoder to the labels
label_encoder.adapt(labels)

# Encode the labels
encoded_labels = label_encoder(labels)
```

**Implementation**

```
# Label encoding implementation using Keras LabelEncoder
from sklearn.preprocessing import LabelEncoder

# Initialize the label encoder
label_encoder = LabelEncoder()

# Fit the label encoder to the labels and encode them
encoded_labels = label_encoder.fit_transform(labels)
```

**Strategies**

```
# Label encoding strategy: Target encoding
import category_encoders as ce

# Initialize the target encoder
target_encoder = ce.TargetEncoder(cols=['category'])

# Fit the target encoder to the data and transform the categorical column
encoded_data['category'] = target_encoder.fit_transform(data['category'], data['target'])
```

## 2.19 Definitions - *Sequence Truncation*

**Sequence Truncation**

Limiting the length of input sequences to a fixed size by either removing or truncating excess elements.

- **Optimization:** Techniques and approaches to efficiently truncate sequences by removing elements or reducing sequence length while preserving the most relevant information, considering factors such as computational efficiency, memory usage, and maintaining sequence coherence.
- **Implementation:** Applying the process of removing elements from the beginning or end of a sequence to reduce its length, often to meet specific input size requirements or to focus on a specific portion of the sequence.
- **Strategies:** Various methods and considerations for sequence truncation, such as truncating based on a fixed length, truncating to a percentage of the original length, or dynamically truncating based on certain criteria such as sentence boundaries, important features, or specific context.

## 2.20 Code Snippets - *Sequence Truncation*

**Sequence Truncation**

```
# Sequence Truncation - Keras
from tensorflow.keras.preprocessing.sequence import pad_sequences

max_sequence_length = 10
truncated_sequences = pad_sequences(sequences, maxlen=max_sequence_length)

# Sequence Truncation - TensorFlow
from tensorflow.keras.preprocessing.sequence import pad_sequences

max_sequence_length = 10
truncated_sequences = pad_sequences(sequences, maxlen=max_sequence_length)
```

**Optimization**

```
# Sequence truncation optimization using TensorFlow slice function
import tensorflow as tf

# Truncate sequences using TensorFlow's slice function
truncated_sequences = tf.slice(sequences, begin=[0, 0], size=[max_length, -1])
```

**Implementation**

```
# Sequence truncation implementation using Keras preprocessing
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Truncate sequences using Keras' pad_sequences function
truncated_sequences = pad_sequences(sequences, maxlen=max_length, truncating='post')
```

**Strategies**

```
# Sequence truncation strategy: Dynamic truncation based on sentence boundaries
truncated_sequences = []
for sequence in sequences:
    # Identify sentence boundaries
    sentences = sequence.split('. ')
    truncated_sentences = sentences[:max_sentences]
    truncated_sequence = '. '.join(truncated_sentences)
    truncated_sequences.append(truncated_sequence)
```

# 3 Model Architecture

## 3.1 Definitions - *Convolutional Neural Network (CNN)*

**Convolutional Neural Network (CNN)**

A deep learning model designed for processing structured grid-like data, commonly used in image and video recognition tasks.Voyant Tools is a web-based text analysis and visualization tool designed to assist researchers, scholars, and students in exploring and interpreting textual data.

- **Filters:** Convolutional operation that extracts specific features from input data.
- **Pooling:** Downsampling technique that reduces the spatial dimensions of the feature maps.
- **Activation:** Non-linear function that introduces non-linearity into the CNN model. newline
- **Stride:** Step size used to slide the filters over the input data during convolution.
- **Padding:** Adding extra border of zeros to input data to preserve spatial dimensions during convolution.

## 3.2 Code Snippets - *Convolutional Neural Network (CNN)*

**CNN (Convolutional Neural Network)**

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

# Define the model using Keras layers
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(height, width, channels)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(),
metrics=['accuracy'])

# Train the model using TensorFlow
model.fit(train_data, train_labels, epochs=10, batch_size=32)

# Evaluate the model using TensorFlow
accuracy = model.evaluate(test_data, test_labels)[1]
```

**Filters**

```
# Creating a CNN layer with filters
from tensorflow.keras import layers

# Define the number of filters, kernel size, and input shape
num_filters = 32
kernel_size = 3
input_shape = (height, width, channels)

# Create a Conv2D layer with the specified number of filters and kernel size
conv_layer = layers.Conv2D(num_filters, kernel_size, activation='relu', input_shape=input_shape)
```

**Pooling**

```
# Applying MaxPooling to a CNN layer
from tensorflow.keras import layers

# Define the pooling size and strides
pool_size = (2, 2)
strides = (2, 2)

# Apply MaxPooling to the Conv2D layer
pooling_layer = layers.MaxPooling2D(pool_size=pool_size, strides=strides)
```

**Activation**

```
# Applying activation function to a CNN layer
from tensorflow.keras import layers

# Apply ReLU activation function to the Conv2D layer
conv_layer = layers.Conv2D(num_filters, kernel_size, activation='relu', input_shape=input_shape)
```

**Stride**

```
# Applying stride to a CNN layer
from tensorflow.keras import layers

# Define the stride value
stride_value = 2

# Apply stride to the Conv2D layer
conv_layer = layers.Conv2D(num_filters, kernel_size, strides=stride_value, activation='relu',
input_shape=input_shape)
```

**Padding**

```
# Applying padding to a CNN layer
from tensorflow.keras import layers

# Apply padding to the Conv2D layer
padding_type = 'same'  # or 'valid' for no padding
conv_layer = layers.Conv2D(num_filters, kernel_size, padding=padding_type, activation='relu',
input_shape=input_shape)
```

## 3.3 Definitions - *Recurrent Neural Network (RNN)*

**Recurrent Neural Network (RNN)**

A type of neural network with feedback connections, capable of capturing sequential dependencies in data, often used for natural language processing and time series analysis.

- **Recurrence:** Connections that allow information to be passed from previous steps to future steps in the sequence.
- **Hidden State:** Representation of the network's memory or information at a specific time step.
- **Sequence:** Input data organized in a sequential manner, such as time series or text.. newline
- **Backpropagation:** Algorithm used to update the weights of the RNN based on the error calculated during training.
- **Vanishing Gradient:** Issue where the gradient diminishes exponentially during backpropagation, hindering learning in long sequences.

## 3.4 Code Snippets - *Recurrent Neural Network (RNN)*

**RNN (Recurrent Neural Network)**

```
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

# Define the model using Keras layers
model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, input_length=max_seq_length))
model.add(SimpleRNN(64))
model.add(Dense(num_classes, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(),
metrics=['accuracy'])

# Train the model using TensorFlow
model.fit(train_data, train_labels, epochs=10, batch_size=32)

# Evaluate the model using TensorFlow
accuracy = model.evaluate(test_data, test_labels)[1]
```

**Recurrence**

```
# Recurrence in an RNN layer
from tensorflow.keras import layers

# Create an RNN layer with LSTM cells
rnn_layer = layers.LSTM(units=num_units, return_sequences=True)
```

**Hidden State**

```
# Accessing the hidden state of an RNN layer
from tensorflow.keras import layers

# Create an RNN layer with LSTM cells
rnn_layer = layers.LSTM(units=num_units, return_sequences=True)

# Retrieve the hidden state of the RNN layer
hidden_state = rnn_layer.get_states()[0]
```

**Sequence**

```
# Processing sequential data with an RNN layer
from tensorflow.keras import layers

# Create an RNN layer with LSTM cells
rnn_layer = layers.LSTM(units=num_units, return_sequences=True)

# Process a sequence of input data
output_sequence = rnn_layer(input_sequence)
```

**Backpropagation**

```
# Applying backpropagation in an RNN model
from tensorflow.keras import layers

# Define the RNN model with LSTM cells
rnn_model = tf.keras.Sequential([
    layers.LSTM(units=num_units, return_sequences=True),
    layers.Dense(num_classes, activation='softmax')
])

# Compile and train the RNN model
rnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
rnn_model.fit(x_train, y_train, epochs=num_epochs, batch_size=batch_size)
```

**Vanishing Gradient**

```
# Addressing vanishing gradient in an RNN layer
from tensorflow.keras import layers

# Create an RNN layer with LSTM cells and apply gradient clipping
rnn_layer = layers.LSTM(units=num_units, return_sequences=True, recurrent_activation='sigmoid',
recurrent_clip_value=1.0)
```

## 3.5 Definitions - *Long Short-Term Memory (LSTM)*

**Long Short-Term Memory (LSTM)**

A specialized RNN architecture with memory cells that can selectively remember or forget information over long sequences, enabling improved handling of long-term dependencies.

- **Memory Cell:** Component that captures and stores information for long durations in an LSTM network.
- **Gates:** Mechanisms that control the flow of information within the LSTM unit, including forget, input, and output gates.
- **Cell State:** Long-term memory that gets passed along through time steps in an LSTM network. newline
- **Peephole Connections:** Connections that allow the LSTM to view the cell state while making decisions.
- **Forget Gate:** Controls the amount of information retained or forgotten from the previous cell state.

## 3.6 Code Snippets - *Long Short-Term Memory (LSTM)*

### LSTM (Long Short-Term Memory)

```python
import tensorflow as tf
from keras.models import Sequential
from keras.layers import LSTM

# Define the model using Keras layers
model = Sequential()
model.add(LSTM(64, input_shape=(timesteps, input_dim)))
model.add(Dense(num_classes, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(),
metrics=['accuracy'])

# Train the model using TensorFlow
model.fit(train_data, train_labels, epochs=10, batch_size=32)

# Evaluate the model using TensorFlow
accuracy = model.evaluate(test_data, test_labels)[1]
```

### Memory Cell

```python
# Creating an LSTM layer with memory cells
from tensorflow.keras import layers

# Define the number of memory units or cells
num_units = 64

# Create an LSTM layer with the specified number of units
lstm_layer = layers.LSTM(units=num_units, return_sequences=True)
```

### Gates

```python
# Accessing the gates of an LSTM layer
from tensorflow.keras import layers

# Create an LSTM layer with memory cells
lstm_layer = layers.LSTM(units=num_units, return_sequences=True)

# Retrieve the gates of the LSTM layer
gates = lstm_layer.get_gates()
```

### Cell State

```python
# Accessing the cell state of an LSTM layer
from tensorflow.keras import layers

# Create an LSTM layer with memory cells
lstm_layer = layers.LSTM(units=num_units, return_sequences=True)

# Retrieve the cell state of the LSTM layer
cell_state = lstm_layer.get_cell_state()
```

### Peephole Connections

```python
# Creating an LSTM layer with peephole connections
from tensorflow.keras import layers

# Create an LSTM layer with peephole connections
lstm_layer = layers.LSTM(units=num_units, return_sequences=True, use_peephole=True)
```

### Forget Gate

```python
# Setting the forget gate bias in an LSTM layer
from tensorflow.keras import layers

# Create an LSTM layer with memory cells and set the forget gate bias
lstm_layer = layers.LSTM(units=num_units, return_sequences=True, bias_initializer='zeros')
```

## 3.7 Definitions - *Generative Adversarial Network (GAN)*

### Generative Adversarial Network (GAN)

A framework consisting of a generator and discriminator network, trained in tandem to generate realistic synthetic data, widely used in image and text generation.

- **Generator:** Part of the GAN that generates new synthetic samples from random noise.
- **Discriminator:** Part of the GAN that tries to distinguish between real and fake samples.
- **Adversarial:** Refers to the competitive nature of the GAN, where the generator and discriminator strive to outperform each other.
  newline
- **Latent Space:** Underlying representation where the generator maps random noise to generate new samples.
- **Loss:** Objective function that quantifies the performance of the generator and discriminator during GAN training.

## 3.8 Code Snippets - *Generative Adversarial Network (GAN)*

### GAN (Generative Adversarial Network)

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

# Define the generator model using Keras layers
generator = Sequential()
generator.add(Dense(100, input_dim=latent_dim, activation='relu'))
...

# Define the discriminator model using Keras layers
discriminator = Sequential()
discriminator.add(Dense(100, input_dim=input_dim, activation='relu'))
...

# Combine the generator and discriminator to form GAN
gan = Sequential()
gan.add(generator)
gan.add(discriminator)

# Compile the GAN model
gan.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam())
```

### Generator

```python
# Creating the generator model in a GAN
from tensorflow.keras import layers

# Create the generator model
generator = tf.keras.Sequential([
    layers.Dense(256, activation='relu', input_shape=(latent_dim,)),
    layers.Dense(512, activation='relu'),
    layers.Dense(784, activation='sigmoid')
])
```

### Discriminator

```python
# Creating the discriminator model in a GAN
from tensorflow.keras import layers

# Create the discriminator model
discriminator = tf.keras.Sequential([
    layers.Dense(512, activation='relu', input_shape=(784,)),
    layers.Dense(256, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```

### Adversarial

```python
# Combining the generator and discriminator in a GAN
from tensorflow.keras import Model

# Set the discriminator's parameters to non-trainable
discriminator.trainable = False

# Combine the generator and discriminator into an adversarial model
adversarial = Model(inputs=generator.input, outputs=discriminator(generator.output))
```

### Latent Space

```python
# Generating samples from the latent space in a GAN
import numpy as np

# Generate random samples from the latent space
latent_dim = 100
num_samples = 10
random_latent_vectors = np.random.normal(size=(num_samples, latent_dim))
```

### Loss

```python
# Defining the adversarial and generator losses in a GAN
import tensorflow as tf

# Define the adversarial loss function
adversarial_loss = tf.keras.losses.BinaryCrossentropy()

# Define the generator loss function
generator_loss = tf.keras.losses.BinaryCrossentropy()
```

## 3.9 Definitions - *Autoencoder*

### Autoencoder

A neural network architecture used for unsupervised learning that learns to encode input data into a lower-dimensional representation and reconstructs the original data from the encoded representation.

- **Vanilla Autoencoder:** Standard autoencoder with an encoder and decoder network for dimensionality reduction and reconstruction.
- **Denoising Autoencoder:** Autoencoder trained to reconstruct clean data from noisy input, promoting robust feature learning.
- **Sparse Autoencoder:** Autoencoder with a sparsity constraint on the latent space, encouraging sparse and informative representations.
- **Variational Autoencoder:** Probabilistic autoencoder that learns a probability distribution in the latent space for generating new samples.
- **Contractive Autoencoder:** Autoencoder trained with an additional regularization term to enforce robust feature extraction and reduce sensitivity to input perturbations.

## 3.10 Code Snippets - *Autoencoder*

### Autoencoders

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

# Define the encoder model using Keras layers
encoder = Sequential()
encoder.add(Dense(encoding_dim, input_shape=(input_dim,), activation='relu'))
...

# Define the decoder model using Keras layers
decoder = Sequential()
decoder.add(Dense(input_dim, input_shape=(encoding_dim,), activation='relu'))
...

# Combine the encoder and decoder to form Autoencoder
autoencoder = Sequential()
autoencoder.add(encoder)
autoencoder.add(decoder)

# Compile the Autoencoder model
autoencoder.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam())
```

### Vanilla Autoencoders

```python
# Keras
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

# Encoder
input_data = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_data)

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

# Autoencoder
autoencoder = Model(input_data, decoded)
```

### Denoising Autoencoders

```python
# Keras
from tensorflow.keras.layers import GaussianNoise

# Encoder
input_data = Input(shape=(input_dim,))
noisy_input = GaussianNoise(stddev)(input_data)
encoded = Dense(encoding_dim, activation='relu')(noisy_input)

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

# Denoising Autoencoder
autoencoder = Model(input_data, decoded)
```

### Sparse Autoencoders

```python
# Keras
from tensorflow.keras import regularizers

# Encoder
input_data = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.l1(sparsity))(input_data)

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

# Sparse Autoencoder
autoencoder = Model(input_data, decoded)
```

### Variational Autoencoders

```python
# Keras
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Lambda

# Encoder
input_data = Input(shape=(input_dim,))
z_mean = Dense(latent_dim)(input_data)
z_log_var = Dense(latent_dim)(input_data)

# Sampling function
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim), mean=0.0, stddev=1.0)
    return z_mean + K.exp(0.5 * z_log_var) * epsilon

z = Lambda(sampling)([z_mean, z_log_var])

# Decoder
decoded = Dense(input_dim, activation='sigmoid')(z)

# Variational Autoencoder
autoencoder = Model(input_data, decoded)
```

### Contractive Autoencoders

```python
# Keras
from tensorflow.keras import losses
from tensorflow.keras import regularizers

# Encoder
input_data = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_data)

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

# Contractive Autoencoder
autoencoder = Model(input_data, decoded)
autoencoder.add_loss(losses.mean_squared_error(input_data, decoded))
autoencoder.add_loss(regularizers.l2(beta)(encoded))
```

## 3.11 Definitions - *Transformers, Feedforward Neural Networks (FNN), Reinforcement Learning, Graph Neural Networks (GNN) and Capsule Networks*

### Concepts

- **Transformers:** Architecture originally introduced for natural language processing tasks, utilizing self-attention mechanisms to capture dependencies between different positions in the input sequence.
- **Feedforward Neural Networks (FNN):** Standard neural network architecture where information flows in one direction, from input to output, without loops or recurrent connections.
- **Reinforcement Learning:** Framework for training agents to make sequential decisions by interacting with an environment and learning from rewards and punishments, often using Markov decision processes and policy optimization algorithms.
- **Graph Neural Networks (GNN):** Architecture designed for learning representations of graph-structured data, capturing relationships and interactions between nodes in a graph using graph convolutional layers.
- **Capsule Networks:** Architecture that aims to overcome the limitations of traditional CNNs by using capsules, which are groups of neurons that encode instantiation parameters of specific visual entities, enabling more robust and hierarchical feature representations.

## 3.12 Code Snippets - *Transformers, Feedforward Neural Networks (FNN), Reinforcement Learning, Graph Neural Networks (GNN) and Capsule Networks*

### Transformers

```python
# Transformers - Keras (Using the Transformers library)
from transformers import TFAutoModel, AutoTokenizer

# Load pre-trained model and tokenizer
model_name = 'bert-base-uncased'
model = TFAutoModel.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Encode text
text = 'This is an example sentence.'
encoded_input = tokenizer(text, padding=True, truncation=True, return_tensors='tf')

# Get the model output
outputs = model(encoded_input)

# Transformers - TensorFlow (Using the Transformers library)
import tensorflow as tf
from transformers import TFAutoModel, AutoTokenizer

# Load pre-trained model and tokenizer
model_name = 'bert-base-uncased'
model = TFAutoModel.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Encode text
text = 'This is an example sentence.'
encoded_input = tokenizer(text, padding=True, truncation=True, return_tensors='tf')

# Get the model output
outputs = model(encoded_input)
```

### Feedforward Neural Networks (FNN)

```python
# FNN - Keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# FNN - TensorFlow
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

### Reinforcement Learning

```python
# Reinforcement Learning - Keras (Using the OpenAI Gym library)
import gym

env = gym.make('CartPole-v1')
state = env.reset()

done = False
while not done:
    action = model.predict(state)  # Replace with your RL model's action selection logic
    state, reward, done, _ = env.step(action)
    env.render()

# Reinforcement Learning - TensorFlow (Using the OpenAI Gym library)
import gym
import tensorflow as tf

env = gym.make('CartPole-v1')
state = env.reset()

done = False
while not done:
    action = model.predict(state)  # Replace with your RL model's action selection logic
    state, reward, done, _ = env.step(action)
    env.render()
```

### Graph Neural Networks (GNN)

```python
# GNN - Keras (Using the Spektral library)
import spektral

adjacency_matrix = ...  # Define the adjacency matrix
features = ...  # Define the node features

model = spektral.GraphConvolution(units=64)([features, adjacency_matrix])
model = spektral.GraphConvolution(units=num_classes)([model, adjacency_matrix])

# GNN - TensorFlow (Using the Spektral library)
import spektral
import tensorflow as tf

adjacency_matrix = ...  # Define the adjacency matrix
features = ...  # Define the node features

model = spektral.GraphConv(units=64)([features, adjacency_matrix])
model = spektral.GraphConv(units=num_classes)([model, adjacency_matrix])
```

### Capsule Networks

```python
# Capsule Networks - Keras
from tensorflow.keras import layers

x = layers.Input(shape=input_shape)
capsule = layers.Capsule(num_capsules=num_classes, dim_capsule=16, routings=3)(x)
output = layers.Lambda(lambda z: tf.sqrt(tf.reduce_sum(tf.square(z), axis=-1)))(capsule)

# Capsule Networks - TensorFlow
import tensorflow as tf
from tensorflow.keras import layers

x = layers.Input(shape=input_shape)
capsule = layers.Capsule(num_capsules=num_classes, dim_capsule=16, routings=3)(x)
output = layers.Lambda(lambda z: tf.sqrt(tf.reduce_sum(tf.square(z), axis=-1)))(capsule)
```

# 4 Model Prediction

## 4.1 Definitions - *Train-Test Splitting, Normalization, Visulaization and Error Analysis*

### Concepts

- **Train-Test Splitting:** Dividing the dataset into training and testing subsets to assess the model's generalization ability on unseen data.
- **Normalization**: Scaling the input features to a standardized range or distribution, enabling fair comparisons and enhancing model stability during training.
- **Visulaization**: Understanding and interpreting the model predictions can be facilitated through visualizations, such as plotting prediction results, creating confusion matrices or generating other relevant visual representations to gain insights from the predictions.
- **Error Analysis**: Assessing and analyzing prediction errors can be valuable in understanding the limitations of the model and identifying areas for improvement.

## 4.2 Code Snippets - *Train-Test Splitting, Normalization, Visulaization and Error Analysis*

### Train-Test Split

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

### Normalization

```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

### Visualization

```python
import matplotlib.pyplot as plt

# Plotting code for visualization (e.g., line plot, scatter plot, etc.)
plt.plot(x_values, y_values)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Title')
plt.show()
```

### Error Analysis

```python
from sklearn.metrics import confusion_matrix

y_pred = model.predict(X_test)
y_pred_labels = np.argmax(y_pred, axis=1)
cm = confusion_matrix(y_test, y_pred_labels)
print(cm)
```

## 4.3 Definitions - *Hyperparameter Tuning, Model Interpretability and Explainable AI (XAI)*

**Concepts**

- **Hyperparameter Tuning:** Techniques and approaches for tuning hyperparameters, such as grid search, random search, and Bayesian optimization, to optimize the performance of your model.
- **Model Interpretability:** Methods for interpreting and understanding the predictions of your model, including feature importance analysis, SHAP values, and model-specific interpretability techniques.
- **Explainable AI (XAI):** Techniques and methods for explaining the reasoning behind the predictions made by your model, ensuring transparency and trustworthiness.

## 4.4 Code Snippets - *Hyperparameter Tuning, Model Interpretability and Explainable AI (XAI)*

**Hyperparameter Tuning**

```python
# Hyperparameter Tuning - Keras and TensorFlow (Using scikit-learn)
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Define your Keras model
def create_model():
    model = Sequential()
    model.add(Dense(64, activation='relu', input_dim=input_dim))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    return model

# Create a KerasClassifier
model = KerasClassifier(build_fn=create_model)

# Define the hyperparameters to tune
param_grid = {'batch_size': [16, 32, 64], 'epochs': [10, 20, 30]}

# Perform grid search for hyperparameter tuning
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=5)
grid_result = grid.fit(X_train, y_train)
best_params = grid_result.best_params_
```

**Model Interpretability**

```python
# Model Interpretability - Keras and TensorFlow (Using SHAP values)
import shap
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Train your model
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32)

# Explain model predictions with SHAP values
explainer = shap.Explainer(model)
shap_values = explainer(X_test)
```

**Explainable AI (XAI)**

```python
# XAI - Keras and TensorFlow (Using LIME)
import lime
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from lime.lime_text import LimeTextExplainer

# Train your model
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32)

# Explain model predictions with LIME
explainer = LimeTextExplainer(class_names=class_names)
explanation = explainer.explain_instance(text_instance, model.predict_proba, num_features=10)
```

# 5 Model Training and Classification

## 5.1 Model Training

### 5.1.1 Definitions - *Loss Functions and Optimization Algorithms*

**Concepts**

- **Loss Functions:** Different types of loss functions used for model training, such as categorical cross-entropy, mean squared error, or custom loss functions.
- **Optimization Algorithms:** Various optimization algorithms like gradient descent, stochastic gradient descent (SGD), Adam, or RMSprop, used to update model parameters during training.

### 5.1.2 Code Snippets - *Loss Functions and Optimization Algorithms*

**Loss Functions**

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

# Define the model using Keras layers
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# Compile the model with a specific loss function
model.compile(loss=tf.keras.losses.CategoricalCrossentropy(), optimizer='adam', metrics=['accuracy'])
```

**Optimization Algorithms**

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

# Define the model using Keras layers
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# Compile the model with a specific optimizer
model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(),
metrics=['accuracy'])
```

### 5.1.3 Definitions - *Learning Rate Scheduling*

**Learning Rate Scheduling**

Techniques for dynamically adjusting the learning rate during training, such as learning rate decay, learning rate annealing, or adaptive learning rate methods.

- **Step Decay:** Step decay is a learning rate scheduling technique where the learning rate is reduced by a certain factor after a fixed number of training epochs or iterations.
- **Exponential Decay:** Exponential decay is another learning rate scheduling method where the learning rate is exponentially decreased over time.
- **Cyclical Learning Rates (CLR):** Cyclical Learning Rates is a learning rate scheduling technique that involves cyclically varying the learning rate between a minimum and maximum value during training.

### 5.1.4 Code Snippets - *Learning Rate Scheduling*

**Learning Rate Scheduling**

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

# Define the model using Keras layers
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# Define a learning rate schedule
lr_schedule = schedules.ExponentialDecay(initial_learning_rate=0.1, decay_steps=10000, decay_rate=0.96)

# Compile the model with a learning rate schedule
model.compile(loss='categorical_crossentropy',
optimizer=tf.keras.optimizers.SGD(learning_rate=lr_schedule), metrics=['accuracy'])
```

**Step Decay**

```python
import tensorflow as tf
from tensorflow import keras

def step_decay(epoch, learning_rate):
    initial_learning_rate = 0.1
    drop = 0.5
    epochs_drop = 10
    learning_rate = initial_learning_rate * tf.math.pow(drop, tf.math.floor((1 + epoch) / epochs_drop))
    return learning_rate

# Create a learning rate scheduler callback
lr_scheduler = keras.callbacks.LearningRateScheduler(step_decay)

# Rest of the code...
# Compile and train your model, and pass the lr_scheduler callback to the fit() function
```

**Exponential Decay**

```python
import tensorflow as tf
from tensorflow import keras

def exponential_decay(epoch, learning_rate):
    initial_learning_rate = 0.1
    decay_rate = 0.1
    decay_steps = 10
    learning_rate = initial_learning_rate * tf.math.exp(-decay_rate * epoch / decay_steps)
    return learning_rate

# Create a learning rate scheduler callback
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay)

# Rest of the code...
# Compile and train your model, and pass the lr_scheduler callback to the fit() function
```

**Cyclical Learning Rates (CLR)**

```python
import tensorflow as tf
from tensorflow import keras

def cyclic_lr(epoch, learning_rate):
    initial_learning_rate = 0.01
    max_learning_rate = 0.1
    step_size = 10
    cycle = tf.floor(1 + epoch / (2 * step_size))
    x = tf.abs(epoch / step_size - 2 * cycle + 1)
    learning_rate = initial_learning_rate + (max_learning_rate - initial_learning_rate) * tf.maximum(0,
    (1 - x))
    return learning_rate

# Create a learning rate scheduler callback
lr_scheduler = keras.callbacks.LearningRateScheduler(cyclic_lr)

# Rest of the code...
# Compile and train your model, and pass the lr_scheduler callback to the fit() function
```

### 5.1.5 Definitions - *Regularization*

**Regularization**

Techniques like L1 and L2 regularization, dropout, or batch normalization to prevent overfitting and improve generalization.

- **L1 Regularization (Lasso):** L1 regularization, also known as Lasso (Least Absolute Shrinkage and Selection Operator), is a technique used in machine learning and statistical modeling to introduce a penalty term to the loss function.
- **L2 Regularization (Ridge):** L2 regularization, also known as Ridge regularization, is a technique that adds the sum of squared values of the model's coefficients to the loss function.

### 5.1.6 Code Snippets - *Regularization*

**Regularization**

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

# Define the model using Keras layers with regularization
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,),
kernel_regularizer=tf.keras.regularizers.l2(0.01)))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

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

**L1 Regularization (Lasso)**

```python
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential([
    keras.layers.Dense(64, activation='relu', kernel_regularizer=keras.regularizers.l1(0.01)),
    keras.layers.Dense(10, activation='softmax')
])

# Rest of the code...
# Compile and train your model using the L1 regularization technique
```

**L2 Regularization (Ridge)**

```python
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential([
    keras.layers.Dense(64, activation='relu', kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.Dense(10, activation='softmax')
])

# Rest of the code...
# Compile and train your model using the L2 regularization technique
```

### 5.1.7 Definitions - *Transfer Learning*

**Transfer Learning**

Leveraging pre-trained models or model architectures to accelerate training and improve performance on related tasks.

- **Adaption:** Modifying a pretrained model for a specific task.
- **Pretrained:** Models trained on large datasets and made available for reuse.
- **Fine-tune:** Adjusting a pretrained model's parameters to fit a new task.
- **Selection:** Choosing the appropriate pretrained model for a specific task.
- **Vision:** Applying transfer learning techniques in computer vision tasks.
- **NLP:** Utilizing transfer learning in natural language processing tasks.

### 5.1.8 Code Snippets - *Transfer Learning*

**Transfer Learning**

```python
mport tensorflow as tf
from keras.applications import VGG16
from keras.models import Sequential
from keras.layers import Dense

# Load the pre-trained VGG16 model
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(img_width, img_height, 3))

# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Create a new model on top of the pre-trained base model
model = Sequential()
model.add(base_model)
model.add(Dense(64, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

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

**Adaption**

```python
base_model = keras.applications.ResNet50(weights='imagenet', include_top=False)
x = base_model.output
# ... Add additional layers for the specific task
```

**Pretrained**

```python
base_model = keras.applications.VGG16(weights='imagenet')
```

**Fine-tune**

```python
base_model.trainable = False
# ... Add additional layers for the specific task
# ... Set base_model.trainable = True for fine-tuning
```

**Selection**

```python
base_model = keras.applications.InceptionV3(weights='imagenet')
```

**Vision**

```python
base_model = keras.applications.DenseNet121(weights='imagenet')
```

**NLP**

```python
base_model = keras.applications.BERT(weights='bert-base-uncased')
```

### 5.1.9 Definitions - *Batch Normalization*

**Batch Normalization**

Implement batch normalization layers in your model to improve training stability, reduce internal covariate shift, and accelerate convergence.

- **Normalization:** Standardizing input features to have zero mean and unit variance.
- **Stabilize:** Reducing internal covariate shift and improving model stability.
- **Optimization:** Enhancing gradient flow and accelerating convergence.
- **Implementation:** Incorporating batch normalization into neural network architectures.
- **Variants:** Different adaptations and enhancements of the original batch normalization technique.
- **Extensions:** Advanced methods and improvements on batch normalization.

### 5.1.10 Code Snippets - *Batch Normalization*

**Batch Normalization**

```python
# Batch Normalization - Keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(BatchNormalization())
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(num_classes, activation='softmax'))

# Batch Normalization - TensorFlow
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(BatchNormalization())
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(num_classes, activation='softmax'))
```

**Normalization**

```python
model_normalization = keras.models.Sequential()
model_normalization.add(keras.layers.BatchNormalization())
```

**Stabilize**

```python
model_stabilize = keras.models.Sequential()
model_stabilize.add(keras.layers.BatchNormalization())
```

**Optimization**

```python
model_optimization = keras.models.Sequential()
model_optimization.add(keras.layers.BatchNormalization())
```

**Implementation**

```python
model_implementation = keras.models.Sequential()
model_implementation.add(keras.layers.BatchNormalization())
```

**Variants**

```python
model_variants = keras.models.Sequential()
model_variants.add(keras.layers.BatchNormalization())
```

**Extensions**

```python
model_extensions = keras.models.Sequential()
model_extensions.add(keras.layers.BatchNormalization())
```

### 5.1.11 Definitions - *Early Stopping*

**Early Stopping**

Utilize early stopping techniques to prevent overfitting by monitoring a validation metric and stopping the training process when the metric stops improving.

- **Generalization:** Ensuring the model performs well on unseen data.
- **Regularization:** Preventing overfitting by stopping model training early.
- **Metrics:** Evaluation criteria used to monitor model performance during training.
- **Monitoring:** Continuously observing training progress to determine when to stop.
- **Implementation:** Incorporating early stopping into the training process.
- **Strategies:** Approaches to determine the optimal time to stop training.

### 5.1.12 Code Snippets - *Early Stopping*

**Early Stopping**

```
# Early Stopping - Keras
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(patience=3)
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val),
callbacks=[early_stopping])

# Early Stopping - TensorFlow
import tensorflow as tf

early_stopping = tf.keras.callbacks.EarlyStopping(patience=3)
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val),
callbacks=[early_stopping])
```

**Generalization**

```
early_stopping_generalization = keras.callbacks.EarlyStopping(patience=10)
```

**Regularization**

```
early_stopping_regularization = keras.callbacks.EarlyStopping(patience=10)
```

**Metrics**

```
early_stopping_metrics = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10,
min_delta=0.001)
```

**Monitoring**

```
early_stopping_monitoring = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=10,
mode='max')
```

**Implementation**

```
early_stopping_implementation = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10,
verbose=1)
```

**Strategies**

```
early_stopping_strategies = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
```

### 5.1.13 Definitions - *Dropout*

**Dropout**

Apply dropout regularization to randomly deactivate neurons during training, reducing overfitting and improving model generalization.

- **Regularization:** Reducing overfitting by randomly dropping units during training.
- **Probabilistic:** Introducing randomness by stochastically disabling units.
- **Inactivation:** Temporarily deactivating neurons during model training.
- **Implementation:** Applying dropout in neural network architectures.
- **Variants:** Different adaptations and variations of the dropout technique.
- **Extensions:** Advanced methods and enhancements related to dropout regularization.

### 5.1.14 Code Snippets - *Dropout*

**Dropout**

```
# Dropout - Keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# Dropout - TensorFlow
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

**Regularization**

```
model_regularization = keras.models.Sequential()
model_regularization.add(keras.layers.Dropout(0.5))
```

**Probabilistic**

```
model_probabilistic = keras.models.Sequential()
model_probabilistic.add(keras.layers.Dropout(0.5))
```

**Inactivation**

```
model_inactivation = keras.models.Sequential()
model_inactivation.add(keras.layers.Dropout(0.5))
```

**Implementation**

```
model_implementation = keras.models.Sequential()
model_implementation.add(keras.layers.Dropout(0.5))
```

**Variants**

```
model_variants = keras.models.Sequential()
model_variants.add(keras.layers.SpatialDropout2D(0.5))
```

**Extensions**

```
model_extensions = keras.models.Sequential()
model_extensions.add(keras.layers.AlphaDropout(0.5))
```

### 5.1.15 Definitions - *Gradient Clipping*

**Gradient Clipping**

Implement gradient clipping to prevent exploding gradients during training by capping the gradient values to a specified threshold.

- **Control:** Limiting the magnitude of gradients to prevent instability.
- **Stabilize:** Ensuring smooth training by constraining gradient values.
- **Optimization:** Improving convergence and preventing gradient explosions/vanishing.
- **Implementation:** Adding gradient clipping to the training process.
- **Strategies:** Different techniques and thresholds for gradient clipping.

### 5.1.16 Code Snippets - *Gradient Clipping*

**Gradient Clipping**

```
# Gradient Clipping - Keras
from tensorflow.keras.optimizers import SGD

opt = SGD(clipvalue=0.5)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

# Gradient Clipping - TensorFlow
import tensorflow as tf

opt = tf.keras.optimizers.SGD(clipvalue=0.5)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
```

**Control**

```
optimizer_control = keras.optimizers.SGD(clipvalue=0.5)
```

**Stabilize**

```
optimizer_stabilize = keras.optimizers.SGD(clipnorm=1.0)
```

**Optimization**

```
optimizer_implementation = keras.optimizers.RMSprop(clipnorm=1.0)
```

**Implementation**

```
optimizer_implementation = keras.optimizers.RMSprop(clipnorm=1.0)
```

**Strategies**

```
optimizer_strategies = keras.optimizers.Adagrad(clipvalue=0.5)
```

### 5.1.17 Definitions - *Model Checkpointing*

**Model Checkpointing**

Implement model checkpointing to save the best-performing model during training based on a chosen metric, allowing you to restore the model later or use it for inference.

- **Control:** Limiting the magnitude of gradients to prevent instability.
- **Resuming:** Restoring a saved model to continue training or make predictions.
- **Use Cases:** Scenarios where model checkpointing is beneficial, such as long training sessions or resumable training.
- **Monitoring:** Keeping track of specific metrics or conditions to trigger model checkpointing.
- **Implementation:** Incorporating model checkpointing into the training pipeline.
- **Strategies:** Different approaches for naming, storing, and managing model checkpoints.

### 5.1.18 Code Snippets - *Model Checkpointing*

**Model Checkpointing**

```
# Model Checkpointing - Keras
from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint('best_model.h5', monitor='val_loss', save_best_only=True)
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val),
callbacks=[checkpoint])

# Model Checkpointing - TensorFlow
import tensorflow as tf

checkpoint = tf.keras.callbacks.ModelCheckpoint('best_model.h5', monitor='val_loss',
save_best_only=True)
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val),
callbacks=[checkpoint])
```

**Control**

```
checkpoint_control = keras.callbacks.ModelCheckpoint(filepath='model_checkpoint.h5',
save_best_only=True)
```

**Resuming**

```
checkpoint_resuming = keras.callbacks.ModelCheckpoint(filepath='model_checkpoint.h5',
save_best_only=False)
```

**Use Cases**

```
checkpoint_use_cases = keras.callbacks.ModelCheckpoint(filepath='model_checkpoint.h5',
save_weights_only=True, period=5)
```

**Monitoring**

```
checkpoint_monitoring = keras.callbacks.ModelCheckpoint(filepath='model_checkpoint.h5',
monitor='val_loss', save_best_only=True)
```

**Implementation**

```
checkpoint_implementation = keras.callbacks.ModelCheckpoint(filepath='model_checkpoint.h5',
save_weights_only=True)
```

**Strategies**

```
checkpoint_strategies = keras.callbacks.ModelCheckpoint(filepath='model_checkpoint.h5',
save_best_only=True, mode='max', monitor='val_accuracy')
```

## 5.2 Classification

### 5.2.1 Definitions - *Binary Classification, Multiclass Classification, Logistic Regression and Decision Trees*

**Concepts**

- **Binary Classification:** Techniques and evaluation metrics for classifying data into two classes, such as logistic regression, support vector machines (SVM), or receiver operating characteristic (ROC) curves.
- **Multiclass Classification:** Approaches to classify data into multiple classes, including one-vs-all, one-vs-one, or softmax regression, and metrics like accuracy, precision, recall, or F1-score.
- **Logistic Regression:** Binary classification algorithm that models the relationship between input features and the probability of belonging to a certain class.
- **Decision Trees:** Non-parametric algorithm that partitions the feature space based on a series of if-else decision rules to perform classification.

### 5.2.2 Code Snippets - *Binary Classification, Multiclass Classification, Logistic Regression and Decision Trees*

**Binary Classification**

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

# Define the model architecture for binary classification
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(1, activation='sigmoid'))

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

**Multiclass Classification**

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

# Define the model architecture for multiclass classification
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(num_classes, activation='softmax'))

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

**Logistic Regression**

```
# Logistic Regression - Keras and TensorFlow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD

# Define your model
model = Sequential()
model.add(Dense(1, activation='sigmoid', input_dim=input_dim))

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

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

**Decision Trees**

```
# Decision Trees - Keras and TensorFlow (Using scikit-learn)
from sklearn.tree import DecisionTreeClassifier
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Define your model
model = DecisionTreeClassifier()

# Train the model
model.fit(X_train, y_train)
```

### 5.2.3 Definitions - *Random Forest, Support Vector Machines (SVM), Naive Bayes and k-Nearest Neighbors (k-NN)*

**Concepts**

- **Random Forest:** Ensemble learning method that combines multiple decision trees to improve classification performance and reduce overfitting.
- **Support Vector Machines (SVM):** Supervised learning algorithm that separates classes by finding an optimal hyperplane in the feature space.
- **Naive Bayes:** Probabilistic algorithm based on Bayes' theorem that assumes independence between features to estimate class probabilities.
- **k-Nearest Neighbors (k-NN):** Instance-based algorithm that classifies data points based on the majority vote of their nearest neighbors in the feature space.

### 5.2.4 Code Snippets - *Random Forest, Support Vector Machines (SVM), Naive Bayes and k-Nearest Neighbors (k-NN)*

**Random Forest**

```
# Random Forest - Keras and TensorFlow (Using scikit-learn)
from sklearn.ensemble import RandomForestClassifier
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Define your model
model = RandomForestClassifier()

# Train the model
model.fit(X_train, y_train)
```

**Support Vector Machines (SVM)**

```
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an SVM classifier
clf = svm.SVC(kernel='linear')

# Train the classifier
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)
```

**Naive Bayes**

```
# Naive Bayes - Keras and TensorFlow (Using scikit-learn)
from sklearn.naive_bayes import GaussianNB
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Define your model
model = GaussianNB()

# Train the model
model.fit(X_train, y_train)
```

**k-Nearest Neighbors (k-NN)**

```
# k-NN - Keras and TensorFlow (Using scikit-learn)
from sklearn.neighbors import KNeighborsClassifier
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Define your model
model = KNeighborsClassifier()

# Train the model
model.fit(X_train, y_train)
```

# 6 Evaluation Metrics

## 6.1 Definitions - *Accuracy, Precision, Recall and F1 Score*

**Concepts**

- **Accuracy:** Accuracy measures the proportion of correctly classified instances over the total number of instances in a dataset. It is a common metric for classification tasks.
- **Precision:** Precision is the ratio of true positives to the sum of true positives and false positives. It measures the accuracy of positive predictions, indicating how many of the predicted positive instances are actually relevant.
- **Recall:** Recall, also known as sensitivity or true positive rate, is the ratio of true positives to the sum of true positives and false negatives. It measures the ability of a model to identify all relevant instances, indicating how many of the actual positive instances are correctly predicted.
- **F1 Score:** The F1 score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall, making it useful when you want to consider both the false positives and false negatives in your evaluation.

## 6.2 Code Snippets - *Accuracy, Precision, Recall and F1 Score*

**Accuracy**

```
# Accuracy using Keras
accuracy_keras = K.mean(K.equal(K.argmax(y_true, axis=-1), K.argmax(y_pred, axis=-1)))

# Accuracy using TensorFlow
accuracy_tf = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_true, axis=-1), tf.argmax(y_pred, axis=-1)),
tf.float32))

# Print the results
print("Accuracy (Keras):", K.eval(accuracy_keras))
print("Accuracy (TensorFlow):", accuracy_tf.numpy())
```

**Precision**

```
# Precision using Keras
true_positives_keras = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
predicted_positives_keras = K.sum(K.round(K.clip(y_pred, 0, 1)))
precision_keras = true_positives_keras / (predicted_positives_keras + K.epsilon())

# Precision using TensorFlow
true_positives_tf = tf.reduce_sum(tf.round(tf.clip_by_value(y_true * y_pred, 0, 1)))
predicted_positives_tf = tf.reduce_sum(tf.round(tf.clip_by_value(y_pred, 0, 1)))
precision_tf = true_positives_tf / (predicted_positives_tf + tf.keras.backend.epsilon())

# Print the results
print("Precision (Keras):", K.eval(precision_keras))
print("Precision (TensorFlow):", precision_tf.numpy())
```

**Recall**

```
# Recall using Keras
true_positives_keras = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
actual_positives_keras = K.sum(K.round(K.clip(y_true, 0, 1)))
recall_keras = true_positives_keras / (actual_positives_keras + K.epsilon())

# Recall using TensorFlow
true_positives_tf = tf.reduce_sum(tf.round(tf.clip_by_value(y_true * y_pred, 0, 1)))
actual_positives_tf = tf.reduce_sum(tf.round(tf.clip_by_value(y_true, 0, 1)))
recall_tf = true_positives_tf / (actual_positives_tf + tf.keras.backend.epsilon())

# Print the results
print("Recall (Keras):", K.eval(recall_keras))
print("Recall (TensorFlow):", recall_tf.numpy())
```

**F1 Score**

```
# F1 Score using Keras
f1_score_keras = 2 * (precision_keras * recall_keras) / (precision_keras + recall_keras + K.epsilon())

# F1 Score using TensorFlow
f1_score_tf = 2 * (precision_tf * recall_tf) / (precision_tf + recall_tf + tf.keras.backend.epsilon())

# Print the results
print("F1 Score (Keras):", K.eval(f1_score_keras))
print("F1 Score (TensorFlow):", f1_score_tf.numpy())
```