1. Tensorboard

TensorBoard is a powerful visualization toolkit for machine learning experimentation, primarily used with TensorFlow. It helps you track and visualize various metrics and aspects of your machine learning models.

```
import tensorflow as tf

from tensorflow import keras

import datetime

mnist = keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0


model = keras.models.Sequential([

   keras.layers.Flatten(input_shape=(28, 28)),

   keras.layers.Dense(128, activation='relu'),

   keras.layers.Dropout(0.2),

   keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

              loss='sparse_categorical_crossentropy',

              metrics=['accuracy'])


log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test),
```

```
        callbacks=[tensorboard_callback])
```

2. **Layers and their types (dense, convolution, and recurrent)**

Neural networks are organized into layers, where each layer consists of neurons that process inputs and pass the outputs to the next layer. There are different types of layers used for various tasks

```
import tensorflow as tf
from tensorflow import keras


model = keras.models.Sequential()


model.add(keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(28, 28, 1)))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))


model.add(keras.layers.Flatten())
model.add(keras.layers.Reshape((28, 28)))


model.add(keras.layers.LSTM(64, activation='tanh',
return_sequences=False))


model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
model.summary()
```

3. Sequential API

• Structure: This API allows you to build models layer-by-layer in a linear stack.

• Ease of Use: It's straightforward and easy to use, making it ideal for simple models.

• Limitations: It doesn't support models with multiple inputs or outputs, and it can't share layers or create complex architectures

Sequential API

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))

model.add(Dense(10, activation='softmax')
```

4. Functional API

• Structure: This API is more flexible and powerful, allowing you to build complex models with non-linear topology, shared layers, and multiple inputs and outputs.

• Flexibility: You can define models as directed acyclic graphs (DAGs) of layers, which is useful for creating more sophisticated architectures.

• Use Cases: Ideal for models that require branching, merging, or other complex layer configurations

```
import tensorflow as tf

from tensorflow.keras import layers, Model

inputs = tf.keras.Input(shape=(32,), name="input_layer")

x = layers.Dense(64, activation="relu")(inputs)x = layers.Dense(64,
activation="relu")(x)

outputs = layers.Dense(10, activation="softmax", name="output_layer")(x)
```

```
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer="adam",
          loss="sparse_categorical_crossentropy",
          metrics=["accuracy"])

model.summary()
```

5. Adding layers to a model

Adding layers to a model in TensorFlow/Keras can be done in several ways, depending on whether you're using the **Sequential API** or the **Functional API**.

Adding Layers with the Sequential API

With the Sequential API, you can either define all layers at once when creating the model or add them incrementally.

```
from tensorflow.keras.models import Sequential
 from tensorflow.keras.layers import Dense
 model = Sequential()
 model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax')
```

Adding Layers with the Functional API

In the Functional API, you define each layer and its connection to the previous layer explicitly.

```
import tensorflow as tf
from tensorflow.keras import layers, Model
inputs = tf.keras.Input(shape=(32,), name="input_layer")
x = layers.Dense(64, activation="relu")(inputs)x = layers.Dense(64, activation="relu")(x)
```

```
outputs = layers.Dense(10, activation="softmax", name="output_layer")(x)
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer="adam",
          loss="sparse_categorical_crossentropy",
          metrics=["accuracy"])

model.summary()
```

6. Configuring layer parameters (activation functions, number of units, etc.)

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

model = tf.keras.Sequential()

model.add(layers.InputLayer(input_shape=(32,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
          loss='binary_crossentropy',
          metrics=['accuracy'])
model.summary()
```

7. Preparing input data (data preprocessing, normalization, one-hot encoding)

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
data = pd.read_csv('data.csv')
data.fillna(method='ffill', inplace=True)
data.drop_duplicates(inplace=True)

scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(data)
encoder = OneHotEncoder(sparse=False)
encoded_data = encoder.fit_transform(data[['category_column']])
```

8. **Splitting data into training, validation, and test sets**

```
from sklearn.model_selection import train_test_split

data = pd.read_csv('data.csv')
X = data.drop('target', axis=1)
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
 X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25,
random_state=42)
```

9. Recurrent Neural Networks (RNN)

Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other.

```
import tensorflow as tf

from tensorflow.keras import layers


model = tf.keras.Sequential()

model.add(layers.SimpleRNN(64, activation='tanh', input_shape=(10, 1)))

model.add(layers.Dense(1))


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

model.summary()


import numpy as np


x_train = np.random.random((1000, 10, 1))

y_train = np.random.random((1000,))


model.fit(x_train, y_train, epochs=5)
```

10. Cleaning and Normalizing Text Data

Cleaning and normalizing text data is a critical preprocessing step in Natural Language Processing (NLP) to ensure that the text is in a consistent format and free from noise, which improves the performance of NLP models.

```
import re

import nltk

from nltk.corpus import stopwords

from nltk.stem import PorterStemmer, WordNetLemmatizer

from nltk.tokenize import word_tokenize


nltk.download('punkt')

nltk.download('stopwords')

nltk.download('wordnet')


text = "This is an example sentence with a URL: https://example.com, numbers 123, and special characters like @ and #!"


text = text.lower()

text = re.sub(r'[^\w\s]', '', text)

text = re.sub(r'\d+', '', text)

text = re.sub(r'\s+', ' ', text).strip()


stop_words = set(stopwords.words('english'))

tokens = word_tokenize(text)

filtered_tokens = [word for word in tokens if word not in stop_words]


stemmer = PorterStemmer()

stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]


lemmatizer = WordNetLemmatizer()

lemmatized_tokens = [lemmatizer.lemmatize(word) for word in filtered_tokens]
```

```
cleaned_text = ' '.join(lemmatized_tokens)

print(cleaned_text)
```

11. Working with the Tokenizer

Tokenization is the process of dividing a text into smaller units known as tokens. **Tokens** are typically words or sub-words in the context of natural language processing. Tokenization is a critical step in many NLP tasks, including text processing, language modelling, and machine translation. The process involves splitting a string, or text into a list of tokens.

```
import nltk

from nltk.tokenize import word_tokenize, sent_tokenize

import tensorflow as tf


nltk.download('punkt')


text = "Hello world! How are you? Playing football is fun."


word_tokens = word_tokenize(text)

sentence_tokens = sent_tokenize(text)


tokenizer = tf.keras.preprocessing.text.Tokenizer()

tokenizer.fit_on_texts([text])

subword_tokens = tokenizer.texts_to_sequences([text])


print("Word Tokens:", word_tokens)
```

print("Sentence Tokens:", sentence_tokens)

print("Subword Tokens:", subword_tokens)

12.Text to Sequence

Text to Sequence is an NLP preprocessing technique that converts text into numerical sequences, where each word or token is represented by a unique integer. This step is crucial for preparing text data for machine learning models, which require numerical input.

import numpy as np

from tensorflow.keras.preprocessing.text import Tokenizer

texts = ["I love machine learning.", "Deep learning is amazing!", "Natural language processing is fascinating."]

tokenizer = Tokenizer()

tokenizer.fit_on_texts(texts)

sequences = tokenizer.texts_to_sequences(texts)word_index = tokenizer.word_index

print("Sequences:", sequences)

print("Word Index:", word_index)