1. TensorBoard:

- A visualization toolkit for tracking and visualizing machine learning metrics.
- **Example**:

```
import tensorflow as tf
from tensorflow import keras
import datetime

(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = keras.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. Layer Types:

- **Dense**, **Convolutional**, **Recurrent** layers for various tasks.

3. Sequential API:

- Build models layer-by-layer in a linear stack.
- **Example**:

```
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:

- Create complex architectures with shared layers.
- **Example**:

```
from tensorflow.keras import layers, Model
```

```
inputs = tf.keras.Input(shape=(32,))
x = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(x)
model = Model(inputs=inputs, outputs=outputs)
```

5. Adding Layers:

- Incrementally add layers using both APIs:

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

6. Configuring Layer Parameters:

- Set activation functions and units:

```
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

7. Data Preprocessing:

- **Normalization** & **One-Hot Encoding**:

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

data =
pd.read_csv('data.csv').fillna(method='ffill').drop_duplicates()
normalized_data = MinMaxScaler().fit_transform(data)
encoded_data =
OneHotEncoder(sparse=False).fit_transform(data[['category_column']])
```

8. Splitting Data:

- Split data into training, validation, and test sets:

```
from sklearn.model_selection import train_test_split

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)
```

9. Recurrent Neural Networks (RNN):

- RNNs process sequences where outputs depend on previous inputs.
- **Example**:

```
model = tf.keras.Sequential()
model.add(layers.SimpleRNN(64, activation='tanh', input_shape=(10,
1)))
model.add(layers.Dense(1))
```

10. Cleaning Text Data:

- Normalize and clean text for NLP:

```
import re
import nltk

text = "This is an example sentence!"
text = re.sub(r'[^\w\s]', '', text.lower()).strip()
```

11. Tokenization:

- Process of dividing text into tokens:

```
from nltk.tokenize import word_tokenize, sent_tokenize

text = "Hello world!"
word_tokens = word_tokenize(text)
```

12. Text to Sequence:

- Convert text to numerical sequences:

```
from tensorflow.keras.preprocessing.text import Tokenizer

texts = ["I love machine learning."]
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

13. Training Neural Networks with Backpropagation
Backpropagation is the process of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch.

```
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_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))
```

---

14. Evaluating Model Performance
Model performance can be evaluated using metrics like accuracy, precision, recall, and F1-score.

```
from sklearn.metrics import classification_report
```

```
y_pred = model.predict_classes(X_test)
print(classification_report(y_test, y_pred))
```

---

15. Hyperparameter Tuning
You can improve the model by tuning hyperparameters like learning rate, batch size, and number of epochs.

```
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(learning_rate=0.001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=32, epochs=20,
validation_data=(X_val, y_val))
```

---

16. Early Stopping in Neural Networks
Early stopping is used to stop training when the model's performance on the validation set stops improving.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss', patience=3)
model.fit(X_train, y_train, epochs=50, validation_data=(X_val, y_val),
callbacks=[early_stopping])
```

---

17. Batch Normalization
Batch normalization is used to speed up training and make the model more stable.

```
from tensorflow.keras.layers import BatchNormalization

model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(10, activation='softmax'))
```

---

18. Dropout Regularization
Dropout is a regularization technique that prevents overfitting by randomly dropping neurons during training.

```
from tensorflow.keras.layers import Dropout

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

---

19. Transfer Learning
Transfer learning involves using a pre-trained model and adapting it to your task.

```
from tensorflow.keras.applications import VGG16
```

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 
224, 3))
base_model.trainable = False
model = Sequential([base_model, Dense(10, activation='softmax')])
```

---

20. Custom Loss Function
You can define your custom loss function to handle specific problems.

```
import tensorflow as tf

def custom_loss(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))

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