Convolutional Neural Network (CNN) with TensorFlow - Code and Explanation

This document provides a simple example of building and training a Convolutional Neural Network (CNN) using TensorFlow and Keras. The model will be trained on the MNIST dataset (handwritten digits), and the code will be explained line by line to help beginners understand how CNNs work. Feel free to modify the code to explore different configurations and observe how they affect the model's performance.

Step-by-Step CNN Example Code:

Here is the full code:

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Load the dataset (MNIST - handwritten digits)
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()

# Preprocess the data
# Reshape the images to include channel dimension and normalize pixel values to [0, 1]
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1)).astype('float32') /
255
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1)).astype('float32') / 255

# Define the CNN model
model = models.Sequential()

# Add convolutional layers
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Add fully connected (Dense) layers
```

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))  # Output layer for 10 classes

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

# Train the model
history = model.fit(train_images, train_labels, epochs=5, validation_data=(test_images,
test_labels))

# Evaluate the model on test data
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print("\nTest accuracy:", test_acc)

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()
```

Line-by-Line Explanation:

1. **Import Libraries**: TensorFlow is the core library, Keras helps to define layers, and matplotlib is for plotting.

2. **Load the Dataset**: Loads the MNIST dataset, containing handwritten digits (0-9).

3. **Preprocess the Data**: Reshapes the images and normalizes the pixel values to [0, 1].

4. **Define the CNN Model**: Sequential model is created, and layers are added.

5. **Convolutional Layer (32 filters)**: First convolutional layer with 32 filters of size 3x3.

6. **MaxPooling Layer**: Max pooling reduces spatial dimensions, focusing on important features.

7. **Second Conv Layer (64 filters)**: Increases the depth to capture more complex features.

8. **Second MaxPooling Layer**: Further reduces spatial dimensions to simplify computation.

9. **Third Conv Layer (64 filters)**: More filters to learn higher-level features.

10. **Flatten Layer**: Flattens the data from 2D to 1D for fully connected layers.

11. **Dense Layer (64 units)**: Fully connected layer to combine features.

12. **Output Layer (Softmax)**: Outputs 10 probabilities for 10 classes.

13. **Compile the Model**: Adam optimizer and cross-entropy loss are used.

14. **Train the Model**: Model is trained for 5 epochs and validated on the test set.

15. **Evaluate and Plot**: Accuracy and loss are plotted to see the model performance.

Suggested Modifications:

1. **Change the Kernel Size**: Modify the kernel size in the convolutional layers.

2. **Add More Filters**: Increase the number of filters in the convolutional layers.

3. **Change the Dense Layer Size**: Adjust the number of neurons in the Dense layer.

4. **Increase Epochs**: Train for more epochs to see how performance changes.

5. **Use Different Activation Functions**: Try tanh or sigmoid instead of ReLU.

Introduction to MobileNetV3

MobileNetV3 is an efficient convolutional neural network optimized for mobile and edge devices. It uses advanced techniques like Squeeze-and-Excitation (SE) blocks and network architecture search (NASNet) to further improve performance over MobileNetV2.

What is MobileNetV3?

MobileNetV3 integrates SE blocks to focus on important features, making it more accurate while maintaining efficiency. It also uses inverted residuals and linear bottlenecks from MobileNetV2, improving speed and accuracy.

MobileNetV3 Small vs Large

- **MobileNetV3 Small**: Smaller and more efficient, designed for devices with very limited resources.

- **MobileNetV3 Large**: Larger and more accurate, but requires more computational resources.

Here is an example code snippet to load MobileNetV3 in TensorFlow:

```
from tensorflow.keras.applications import MobileNetV3Small, MobileNetV3Large

# Load MobileNetV3 (pre-trained on ImageNet)
model_small = MobileNetV3Small(weights='imagenet', input_shape=(224, 224, 3))
model_large = MobileNetV3Large(weights='imagenet', input_shape=(224, 224, 3))

# Display the model summary
model_small.summary()
model_large.summary()
```

Final Note

MobileNetV3 balances speed and accuracy, making it a great choice for mobile applications. Experiment with both MobileNetV3 Small and Large, and compare their trade-offs between performance and efficiency.