Building a CNN for MNIST Handwritten Digit Classification

⌄ Introduction

Welcome! In this assignment, you will build a Convolutional Neural Network (CNN) to classify handwritten digits from the famous MNIST dataset. This dataset is a classic in the field of computer vision and provides a great starting point for understanding image classification with deep learning.

This notebook is structured to guide you step-by-step through the process. You will load the data, preprocess it, define a CNN model, train it, and evaluate its performance. Throughout the assignment, you will have opportunities to experiment and deepen your understanding of the concepts.

Remember to:

- **Read all instructions carefully.**
- **Execute the code cells in order.**
- **Fill in the missing code sections marked as "Students: Fill in the blanks".**
- **Answer the reflection questions in the designated Markdown cell.**
- **Experiment and explore!** Change parameters, layers, and observe the effects.

Let's get started and build our MNIST digit classifier!

Section 1: Setting Up - Imports

Before we dive into building our CNN, we need to import the necessary libraries. These libraries provide pre-built tools and functions that will make our work much easier.

Instructions:

1. **Carefully review the code cell below.** It imports libraries from TensorFlow and Keras, which are powerful frameworks for building and training neural networks.
2. **Execute the code cell by selecting it and pressing [Shift + Enter] (or the "Run" button).**
3. **Ensure there are no error messages after running the cell.** If you encounter errors, double-check that you have TensorFlow and Keras installed in your environment.

```
# Cell 1: Imports
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
```

Explanation of Imports:

- `tensorflow as tf` **and** `keras`: TensorFlow is the main deep learning framework, and Keras is its high-level API that simplifies building and training models. We import TensorFlow as `tf` and Keras directly for easy access to their functionalities.
- `from tensorflow.keras import layers`: This imports the `layers` module from Keras, which provides various layers for building neural networks (like convolutional layers, dense layers, etc.).
- `from tensorflow.keras.datasets import mnist`: This imports the MNIST dataset directly from Keras datasets. This is very convenient for loading and using the MNIST data.
- `from tensorflow.keras.utils import to_categorical`: This imports the `to_categorical` function, which we will use to perform one-hot encoding of our labels.

⌄ Section 2: Data Loading and Preprocessing

In this section, we will load the MNIST dataset and prepare it for training our CNN model. Preprocessing steps are crucial to ensure our data is in the right format for the model to learn effectively.

Instructions:

1. **Read through the code in the cell below.** Understand how it loads the MNIST dataset and what preprocessing steps are applied.
2. **Execute the code cell.**
3. **Examine the comments in the code** to understand each preprocessing step in detail.

```
# Cell 2: Data Loading and Preprocessing
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Add a channel dimension (for grayscale images, it's 1)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# One-hot encode the labels
num_classes = 10
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

⤵ Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
   11490434/11490434 ━━━━━━━━━━━━━━━━━━ 0s 0us/step

Explanation of Data Preprocessing:

- **Loading the MNIST dataset:** `mnist.load_data()` loads the MNIST dataset, which is already split into training and testing sets
  (`(x_train, y_train), (x_test, y_test)`). `x_train` and `x_test` contain the images (pixel data), and `y_train` and `y_test` contain the
  corresponding labels (digits 0-9).
- **Normalization:** `x_train = x_train.astype("float32") / 255.0` and `x_test = x_test.astype("float32") / 255.0` normalize the pixel
  values. Pixel values in images are typically in the range 0-255. Dividing by 255 scales them to the range 0-1. This normalization helps
  the neural network train faster and more effectively.
- **Adding Channel Dimension:** `x_train = x_train.reshape(-1, 28, 28, 1)` and `x_test = x_test.reshape(-1, 28, 28, 1)` reshape the
  data to add a channel dimension. Even though MNIST images are grayscale (single channel), CNNs in Keras expect input data to have a
  channel dimension. We reshape from `(number_of_images, 28, 28)` to `(number_of_images, 28, 28, 1)`. The `-1` in `reshape` means
  "infer the dimension based on the size of the array."
- **One-Hot Encoding:** `y_train = to_categorical(y_train, num_classes)` and `y_test = to_categorical(y_test, num_classes)` perform
  one-hot encoding on the labels. Instead of representing the digit '3' as a single number, one-hot encoding converts it into a vector `[0,
  0, 0, 1, 0, 0, 0, 0, 0, 0]`, where the 4th position (index 3) is 'hot' (value 1), and all other positions are 'cold' (value 0). This is a
  standard way to represent categorical labels for neural networks in multi-class classification problems. `num_classes = 10` specifies
  that we have 10 classes (digits 0-9).

∨ Section 3: Model Definition - Building the CNN

Now we will define the architecture of our Convolutional Neural Network (CNN). You will be building a sequential model using Keras layers.

Instructions:

1. **Carefully examine the code in the cell below.** Notice the structure of the `keras.Sequential` model.
2. **Fill in the missing parts** marked with `# Students: Fill in the blanks` to complete the model definition.
3. **Experiment!** You are encouraged to try different configurations for the layers, such as changing the number of filters in the
   convolutional layers, or adding more layers.

```
# Cell 3: Model Definition
# Build the CNN model.  Students: Fill in the missing parts!
model = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),  # Input layer
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"), # Convolutional layer 1
        layers.MaxPooling2D(pool_size=(2, 2)), # Max pooling layer 1
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"), # Convolutional layer 2
        # Students: Add another Conv2D layer here.  Experiment with the number of filters!
        # layers.Conv2D(____, kernel_size=(____, ____), activation="____"),  # Convolutional layer 2
        layers.Conv2D(128, kernel_size=(5, 5), activation="relu"), #convolutional layer 3
        layers.MaxPooling2D(pool_size=(2, 2)), # Max pooling layer 2
        layers.MaxPooling2D(pool_size=(2, 2)), #max pooling layer 2
        # Students: Add another MaxPooling2D layer here if needed.
        # layers.MaxPooling2D(pool_size=(____, ____)),  # Max pooling layer 2
        layers.Flatten(),  # Flatten layer
        layers.Dropout(0.5),  # Dropout layer
        layers.Dense(num_classes, activation="softmax"),  # Output layer
    ]
)
```

Explanation of Layers:

- `keras.Input(shape=(28, 28, 1))`: This is the input layer of our model. It specifies the shape of the input images, which are 28x28 pixels with 1 channel (grayscale).
- `layers.Conv2D(32, kernel_size=(3, 3), activation="relu")`: This is a 2D Convolutional layer.
  - `32`: This is the number of filters (also called kernels). Each filter learns to detect specific features in the input image.
  - `kernel_size=(3, 3)`: This defines the size of the convolutional filter as 3x3 pixels.
  - `activation="relu"`: ReLU (Rectified Linear Unit) is the activation function. It introduces non-linearity into the model, allowing it to learn complex patterns.
- `layers.MaxPooling2D(pool_size=(2, 2))`: This is a Max Pooling layer.
  - `pool_size=(2, 2)`: It reduces the spatial dimensions of the feature maps by taking the maximum value within each 2x2 window. This helps to reduce the number of parameters, control overfitting, and make the model more robust to small shifts and distortions in the input.
- `layers.Flatten()`: This layer flattens the 2D feature maps from the convolutional and pooling layers into a 1D vector. This is necessary to connect the convolutional part of the network to the fully connected (Dense) layers.
- `layers.Dropout(0.5)`: This is a Dropout layer.
  - `0.5`: This sets the dropout rate to 50%. During training, this layer randomly sets 50% of the input units to 0 at each update. This is a regularization technique that helps to prevent overfitting.
- `layers.Dense(num_classes, activation="softmax")`: This is the output Dense (fully connected) layer.
  - `num_classes`: This is set to 10 because we have 10 classes (digits 0-9).
  - `activation="softmax"`: Softmax activation ensures that the output values are probabilities, and they sum up to 1 across all classes. The output will be a vector of 10 probabilities, where each probability represents the model's confidence that the input image belongs to that specific digit class.

˅   Section 4: Model Compilation - Choosing Loss and Optimizer

Before we can train our model, we need to compile it. Compilation involves choosing an optimizer, a loss function, and metrics to evaluate the model's performance.

Instructions:

1. **Examine the code cell below.** You need to fill in the blanks for the `loss` and `optimizer` parameters in `model.compile()`.
2. **Choose an appropriate loss function and optimizer** for this multi-class classification problem.
3. **In the Markdown cell after the code, explain your choices.** Why are these choices suitable for this task?

```
# Cell 4: Model Compilation
# Students: Choose an appropriate loss function and optimizer.  Why did you choose these?
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"]) #Students: Fill in the blanks
```

Explanation of Choices (To be filled by students in the reflection section):

- **Loss Function:** You need to choose a loss function that is appropriate for multi-class classification. Think about what kind of error we are trying to minimize when classifying digits into 10 categories.
- **Optimizer:** You need to choose an optimizer that will efficiently update the model's weights to minimize the loss function. Consider common optimizers used in deep learning.
- **Metrics:** We are using "accuracy" as a metric to evaluate the model's performance. Accuracy is a common metric for classification tasks, representing the percentage of correctly classified images.

˅   Section 5: Model Training - Fitting the Model to the Data

Now it's time to train our CNN model using the training data. Training involves feeding the training data to the model and adjusting its weights to minimize the loss function.

Instructions:

1. **Examine the code cell below.** You need to fill in the blanks for `batch_size` and `epochs` in `model.fit()`.
2. **Choose appropriate values for `batch_size` and `epochs`.**
3. **Run the code cell to start training.** Observe the training progress, especially the loss and accuracy on both the training and validation sets.

4. **Experiment!** Change the `batch_size` and `epochs` and see how it affects the training process and the final performance.

```
# Cell 5: Model Training
# Students: Adjust the batch size and number of epochs.  What happens if you change them?
model.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1) #Students: Fill in the blanks
```

```
Epoch 1/15
422/422 ──────────────── 113s 263ms/step - accuracy: 0.7576 - loss: 0.7567 - val_accuracy: 0.9712 - val_loss: 0.1031
Epoch 2/15
422/422 ──────────────── 111s 262ms/step - accuracy: 0.9468 - loss: 0.1793 - val_accuracy: 0.9812 - val_loss: 0.0680
Epoch 3/15
422/422 ──────────────── 113s 267ms/step - accuracy: 0.9644 - loss: 0.1235 - val_accuracy: 0.9840 - val_loss: 0.0598
Epoch 4/15
422/422 ──────────────── 140s 262ms/step - accuracy: 0.9709 - loss: 0.0997 - val_accuracy: 0.9877 - val_loss: 0.0493
Epoch 5/15
422/422 ──────────────── 110s 261ms/step - accuracy: 0.9753 - loss: 0.0828 - val_accuracy: 0.9882 - val_loss: 0.0413
Epoch 6/15
422/422 ──────────────── 143s 262ms/step - accuracy: 0.9784 - loss: 0.0728 - val_accuracy: 0.9890 - val_loss: 0.0403
Epoch 7/15
422/422 ──────────────── 112s 264ms/step - accuracy: 0.9831 - loss: 0.0608 - val_accuracy: 0.9897 - val_loss: 0.0398
Epoch 8/15
422/422 ──────────────── 139s 258ms/step - accuracy: 0.9820 - loss: 0.0582 - val_accuracy: 0.9902 - val_loss: 0.0369
Epoch 9/15
422/422 ──────────────── 146s 268ms/step - accuracy: 0.9853 - loss: 0.0496 - val_accuracy: 0.9888 - val_loss: 0.0372
Epoch 10/15
422/422 ──────────────── 139s 261ms/step - accuracy: 0.9853 - loss: 0.0481 - val_accuracy: 0.9905 - val_loss: 0.0341
Epoch 11/15
422/422 ──────────────── 141s 259ms/step - accuracy: 0.9877 - loss: 0.0406 - val_accuracy: 0.9913 - val_loss: 0.0369
Epoch 12/15
422/422 ──────────────── 144s 263ms/step - accuracy: 0.9883 - loss: 0.0380 - val_accuracy: 0.9912 - val_loss: 0.0345
Epoch 13/15
422/422 ──────────────── 141s 260ms/step - accuracy: 0.9885 - loss: 0.0377 - val_accuracy: 0.9897 - val_loss: 0.0403
Epoch 14/15
422/422 ──────────────── 140s 256ms/step - accuracy: 0.9885 - loss: 0.0378 - val_accuracy: 0.9925 - val_loss: 0.0329
Epoch 15/15
422/422 ──────────────── 145s 264ms/step - accuracy: 0.9905 - loss: 0.0304 - val_accuracy: 0.9918 - val_loss: 0.0339
<keras.src.callbacks.history.History at 0x7aa8882dab90>
```

Explanation of Training Parameters:

- `batch_size`: This determines the number of training samples processed in each mini-batch during training. A larger batch size can speed up training but might require more memory. A smaller batch size can lead to more noisy updates but might generalize better.
- `epochs`: One epoch represents one complete pass through the entire training dataset. More epochs can potentially lead to better training but also increase the risk of overfitting, where the model learns the training data too well and performs poorly on unseen data.
- `validation_split=0.1`: This reserves 10% of the training data as a validation set. During training, the model's performance is evaluated on this validation set after each epoch. This helps to monitor for overfitting and tune hyperparameters.

˅ Section 6: Model Evaluation - Assessing Performance on Test Data

After training, we need to evaluate our model's performance on the test dataset. This gives us an estimate of how well the model generalizes to unseen data.

Instructions:

1. **Run the code cell below.**
2. **Observe the output.** It will print the test loss and test accuracy.
3. **Think about the results.** Is the test accuracy satisfactory? How does it compare to the training and validation accuracy you observed during training?

```
# Cell 6: Model Evaluation
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test loss: {loss:.4f}")
print(f"Test accuracy: {accuracy:.4f}")
```

```
Test loss: 0.0343
Test accuracy: 0.9896
```

Section 7: Reflection and Answers to Questions

This is an important section! Take some time to reflect on what you have learned and answer the following questions in detail. Your thoughtful answers will demonstrate your understanding of the concepts covered in this assignment.

Reflection Questions:

1. **Conv2D Layer:** What is the role of the Conv2D layer? How do the `kernel_size` and the number of filters affect the learning process?
   Hint: Experiment by changing these values in Cell 3.

Answer: The Conv2D layer in a Convolutional Neural Network (CNN) is responsible for detecting various features in an input image. It applies a set of learnable filters (or kernels) to the input image, performing convolution operations that result in feature maps. These feature maps help the network identify patterns such as edges, textures, and shapes at different levels of abstraction.

I changed the kernal size from (3,3) to (5,5) and the filter size from 64 to 128. The kernal size defines the dimensions of the filter that slides over the input image and by changing it to (5,5) the filter covers a larger area of the image at once. Larger kernals can capture more complex and broader features. By changing the filter size from 64 to 128 I allowed the network to learn a richer and more diverse set of features that could have potentially lead to better performance.

2. **MaxPooling2D Layer:** What is the purpose of the MaxPooling2D layer? How does it contribute to the model's performance? *Hint: Try removing or adding a MaxPooling2D layer and see what happens.*

Answer: The purpose of the MaxPooling2D layer is to reduce the spatiial dimensions of the input feature maps. It helps the model to become more invariant to small translations or distortions in the input images and the model can better recognize objects irrespective of their position in the image.

How it contributes to the models performance is by reducing the number of parameters to help prevent overfitting. It also helps concentrating on the most important features detected by the convolutional layers for more accurate predictions. When I removed the MaxPooling layer I noteiced the output features map output retained its larger size and when I added extra MaxPooling2D layer the spatial dimensions of hte features map reduced more aggressively and lead to loss information.

3. **One-Hot Encoding:** Why do we use one-hot encoding for the labels?

Answer: One hot encoding helps provide a clear, and compatible way for models to process categorical labels, and improve the accuracy and performance of classification task.

4. **Flatten Layer:** Why do we need the Flatten layer before the Dense layer?

Answer: We need to flatten the layer before the dense layer to enable layers to establish connections with every node in the proceding layer. This helps in learning complex patterns in data. Without the flattend layer there would be a mismatch in the dimensions of the data which will prevent the dense layers from processing the features map effectively.

5. **Optimizer and Loss Function:** What optimizer and loss function did you choose in Cell 4? Explain your choices. Why is categorical cross-entropy a suitable loss function for this task? Why is Adam a good choice of optimiser?

Answer In Cell 4, I initially chose Sparse Categorical Cross Entropy as the loss function and Stochastic Gradient Descent (SGD) as the optimizer. However, I later switched to Categorical Cross Entropy and Adam when I attempted to run cell 5 and received an error message.

Categorical Cross Entropy is suitable for this task because we are dealing with multi-class classification, where the goal is to classify images of handwritten digits into one of 10 categories (0-9).

Adam (Adaptive Moment Estimation) is an excellent choice of optimizer because it combines the benefits of two other popular optimizers, AdaGrad and RMSProp. Adam adapts the learning rate for each parameter based on the first and second moments of the gradients, which results in more efficient and robust updates.

6. **Batch Size and Epochs:** How did you choose the batch size and number of epochs in Cell 5? What are the effects of changing these parameters? *Hint: Experiment!*

Answer: In Cell 5, I randomly chose a batch size of 64. A smaller batch size allows the model to learn and update its weights more frequently. I later switched to 128 and that helped in stabilizing the learning process and reducing the noise in gradient updates. For the number of epochs, I started with 10 epochs to get a quick understanding of the model's performance. Then increased it to 15 and allowed the model to learn for a longer period and achieve better accuracy. It provided more opportunities for the model to adjust its weights and improve its generalization to the test data.

7. **Dropout:** Why is the Dropout layer included in the model?

Answer The dropout layer is included to prevent overfitting and improve generalization.

8. **Model Architecture:** Describe the overall architecture of your CNN. How many convolutional layers did you use? How many max pooling layers? What is the final dense layer doing?

Answer: The overal architecture of the CNN is a image that is 28x28 pixels with 1 grayscale channel. I used 128 filters, each of size 5x5 to detect features in the input image and the ReLu activation function was used to introduce non linearity and allwed the model to learn complex patterns. I used a MaxPooling size of 2x2 and The final Dense layer has 10 units, corresponding to the 10 classes (digits 0-9). This architecture is designed to effectively extract features from the input images, reduce dimensionality to manage computation, and make accurate predictions.

9. **Performance:** What accuracy did you achieve on the test set? Are you happy with the result? Why or why not? If you're not happy, what could you try to improve the performance?

Answer: I achieved a test accuracy of 98.96% and a test loss of 0.0343. Yes, I am happy with the result. Achieving a test accuracy of 98.96% on the MNIST dataset indicates that the model is performing very well in classifying handwritten digits. This high level of accuracy suggests that the model has effectively learned to recognize and distinguish between the different digit classes.

Tips and Explanations:

- **Normalization:** Dividing the pixel values by 255 normalizes them to the range [0, 1]. This is important for training neural networks.
- **Reshaping:** The `reshape` operation adds a channel dimension to the images. For grayscale images, the channel dimension is 1.
- **One-Hot Encoding:** `to_categorical` converts the class labels (0-9) into one-hot encoded vectors.
- **Conv2D Parameters:** The `kernel_size` determines the size of the convolutional filter (e.g., 3x3). The number of filters determines how many different features are learned.
- **MaxPooling2D Parameters:** The `pool_size` determines the size of the pooling window (e.g., 2x2).
- **Optimizer:** The optimizer is the algorithm used to update the model's weights during training.
- **Loss Function:** The loss function measures the error between the model's predictions and the true labels.
- **Batch Size:** The batch size is the number of samples processed in each training iteration.
- **Epochs:** An epoch is one complete pass through the entire training dataset.
- **Dropout:** Dropout is a regularization technique that helps prevent overfitting.

Remember to run each cell to see its output. Experiment with the code and try to understand how different parameters affect the model's performance. Good luck! """

Conclusion and Submission

Congratulations on completing this notebook assignment! You have successfully built and trained a Convolutional Neural Network to classify handwritten digits from the MNIST dataset. You've explored key concepts like convolutional layers, pooling layers, activation functions, optimizers, loss functions, and training procedures. To further solidify your understanding, consider the following:

- **Review your notebook:** Go back through each section, reread the explanations, and make sure you understand the code and the concepts.
- **Experiment further:** Try different CNN architectures, add more layers, change hyperparameters, and see how it affects the performance. Explore other optimizers or loss functions.
- **Reflect on your learning:** Think about the challenges you faced and how you overcame them. What were the most important takeaways for you from this assignment?

Submission Instructions

To submit your assignment:

1. **Save your notebook:** Ensure all your work, including code cells, outputs, and answers to reflection questions, is saved in the notebook.
2. **Print the notebook as a `.pdf` file** and submit it to Canvas.

Deadline: February, 12th