# 1 Using Exact Multiplier in Convolutional Neural Networks (CNN)

In this exercise, you will explore the application of an exact multiplier within a Convolutional Neural Network (CNN) architecture while working with the popular MNIST dataset. The MNIST dataset consists of a large collection of hand-written digits, and it serves as an ideal starting point for understanding CNNs.

## 1.1 Subtask 1: Importing Essential Packages

In this subtask, we'll introduce some of the critical Python packages we need for our exciting exercise. Let's meet our essential tools!

1. **NumPy (numpy)**: The Wizard of Numbers! NumPy is your trusted companion for numerical operations and versatile array handling. It's your go-to when crunching numbers.
2. **Matplotlib (plt)**: The Artistic Maestro! Matplotlib is here to turn data into art. It's your ticket to creating stunning visualizations and plotting graphs.
3. **TensorFlow (tf)**: The Machine Learning Powerhouse! TensorFlow is your key to building and training machine learning models. It's the muscle behind our neural networks.
4. **SciPy's signal**: The Signal Whisperer! SciPy's signal processing functions, including convolution, are your secret weapon for data processing magic.
5. **TensorFlow's Keras**: The Deep Learning Sorcerer! TensorFlow's Keras is your high-level spellbook for crafting and training deep learning models with ease.

Now, let's get these wizards and tools on board in our Python environment:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from scipy import signal
from tensorflow import keras
```

With this powerful ensemble, we're ready to embark on our data manipulation, visualization, and neural network development journey. Get ready for some magic!

## 1.2 Subtask 2: Loading Data and Pre-trained CNN

In our thrilling journey through the world of deep learning, we've reached a pivotal moment: Subtask 2. Here, we'll embark on an exciting mission to load data and unveil the secrets of a pre-trained Convolutional Neural Network (CNN).

1. **Our Quest Begins: The MNIST Dataset**

   Our adventure starts with the MNIST dataset, a treasure trove of hand-written digits. Imagine pages filled with numbers, each with its unique charm.

   ```
   # Load MNIST Dataset
   mnist = tf.keras.datasets.mnist
   (train_images, train_labels), (test_images, test_labels)
       = mnist.load_data()
   ```

   Our heroes are the training and testing images, along with their trusty companions, the labels. These images, each telling a digit's story, will be our guides through the labyrinth of neural networks.

2. **A Heroic Arrival: The Pre-trained CNN Model**

   But no quest is complete without a seasoned guide. That's where our pre-trained Convolutional Neural Network (CNN) comes into play. This mighty network has journeyed far and wide, learning to recognize patterns and unveil the secrets hidden within images.

```
# Load Pre-trained CNN Model
model = tf.keras.models.load_model('
    my_org_model_top4_quant.h5')
```

Our pre-trained model is like an ancient map, revealing the way to recognizing shapes and features with great precision.

3. **Unveiling Hidden Knowledge: Model Weights**

   As we forge ahead, we uncover the wisdom of the model. Within its depths lie the weights—the encoded knowledge that enables the model to make predictions.

```
# Get Model Weights
a = model.get_weights()
```

   These weights are like the ancient scrolls of a wise sage, filled with the mysteries of the data they've encountered.

4. **The Inner Workings Revealed: Feature Extraction**

   But our quest wouldn't be complete without an understanding of the model's inner workings. We construct an ëxtractor"model, like a magician's glass revealing secrets layer by layer.

```
# Feature Extraction
extractor_in = keras.Model(inputs=model.inputs,
                           outputs=[layer.input for layer
                              in model.layers])
features_in = extractor_in(test_images)
```

   With this extractor, we unveil the hidden feature maps, peering into the magic that unfolds as the model processes images.

   In Subtask 2, we've loaded the tools of our trade, setting the stage for our grand adventure in the realm of deep learning. Our story continues, and with each step, we unravel the mysteries of this fascinating world.

## 1.3 Subtask 3: Implementing Exact Multiplier and Convolution

In Subtask 3, we delve into the precise world of exact multipliers and convolution methods, where accuracy reigns supreme.

1. **Creating an Exact Multiplier Matrix**

   Our journey begins with crafting an exact multiplier matrix, where precision knows no bounds. It's like laying the foundation for a grand structure.

```
Multiplier_Exact = np.zeros([256, 256])

# Populate the Exact Multiplier Matrix
for i in range(-128, 128):
    for j in range(-128, 128):
        Multiplier_Exact[i + 128, j + 128] = i * j
```

   This matrix holds the key to exact multiplication and will serve as our guiding star.

2. **Exact Multiplication: `My_Mult_Exact`**

   In our quest for precision, we've designed an exact multiplication function, `My_Mult_Exact`. It's like wielding a magnifying glass to scrutinize the finest details.

```
def My_Mult_Exact(a, b, t=0):
    a = np.array(a)
    b = np array(b)
    a_shape = np.shape(a)
    b = np.reshape(b, a_shape)
```

```
    res = np.zeros(a_shape)

    if len(a_shape) == 1:
        # Exact multiplication for 1D arrays
        for i in range(np.shape(a)[0]):
            res[i] = Multiplier_Exact[int(a[i]) + 128,
                int(b[i]) + 128]
    if len(a_shape) == 2:
        # Exact multiplication for 2D arrays
        for i in range(a_shape[0]):
            for j in range(a_shape[1]):
                res[i, j] = Multiplier_Exact[int(a[i, j
                    ]) + 128, int(b[i, j]) + 128]

    return res
```

This function allows us to perform exact multiplication with meticulous care.

3. **Exact Matrix Multiplication:** `My_Matmul_LT_Exact`

   Now, we explore exact matrix multiplication. Our `My_Matmul_LT_Exact` function is akin to assembling a complex puzzle piece by piece.

```
def My_Matmul_LT_Exact(a, b, t=0):
    a = np.array(a)
    b = np.array(b)
    a_shape = np.shape(a)
    b_shape = np.shape(b)
    res = np.zeros([a_shape[0], b_shape[1]])

    for i in range(a_shape[0]):
        for j in range(b_shape[1]):
            # Exact matrix multiplication using
                My_Mult_Exact
            res[i, j] = np.sum(My_Mult_Exact(a[i, :], b
                [:, j], t))

    return res
```

   This function handles exact matrix multiplications with unwavering precision.

4. **Exact Look-up Based Convolution:** `My_Conv2d_LT_Exact`

   As our journey nears its apex, we introduce `My_Conv2d_LT_Exact`, our tool for exact look-up-based convolution.

```
def My_Conv2d_LT_Exact(a, b, t=0):
    a = np.array(a)
    b = np.array(b)
    a_shape = np.shape(a)
    b_shape = np.shape(b)
    res_shape1 = np.abs(a_shape[0] - b_shape[0]) + 1
    res_shape2 = np.abs(a_shape[1] - b_shape[1]) + 1
    res = np.zeros([res_shape1, res_shape2])

    for i in range(res_shape1):
        for j in range(res_shape2):
            # Exact convolution using My_Matmul_LT_Exact
```

```
            res[i, j] = np.sum(My_Matmul_LT_Exact(np.
                flip(b), a[i:i + b_shape[0], j:j +
                b_shape[1], t))

    return res
```

With this function, we master exact convolution using a look-up approach, leaving no room for approximation.

In Subtask 3, we've achieved a level of precision that sets the stage for comparison with approximate methods in our ongoing quest. Each function represents a step closer to understanding the heart of convolution and multiplication.

## 1.4 Subtask 4: Implementing the Exact CNN

In Subtask 4, we delve into the meticulous implementation of the Exact CNN. The journey involves quantization, convolution layers, ReLU activations, and fully connected layers, all executed with utmost precision.

1. **Quantization and First Convolution Layer**

   We embark on this journey by quantizing the input data and initiating the first convolution layer:

```
def Exact_CNN(k, t):
    # Quantization of input data
    z1 = np.floor(features_in[0][k] / 2)

    # Initialize the feature map for the first
        convolution layer
    z2 = np.zeros([28, 28, 64])

    # Iterate through each of the 64 output channels
    for i in range(64):
        for j in range(1):
            # Convolve with precision using
                My_Conv2d_LT_Exact
            z2[:, :, i] = z2[:, :, i] +
                My_Conv2d_LT_Exact(
                  np.array(z1[:, :, j]), np.flip(a[0][:,
                      :, j, i]), t
            )

        # Add biases from the first layer
        z2[:, :, i] = z2[:, :, i] + a[1][i]
    # First convolution layer is complete

    # Apply Rectified Linear Unit (ReLU) activation
        function
    z3 = np.maximum(0, z2)

    # Quantize the feature map
    z3 = np.round((z3 / np.max(z3)) * 127)
    # Quantization ensures data precision is maintained
```

   This segment sets the stage with quantization, and the first convolution layer showcases precision. The ReLU activation further enhances the data, and quantization assures data precision is preserved.

2. **Second Convolution Layer**

   Continuing the journey, we navigate the second convolution layer:

```
# Initialize the feature map for the second convolution layer
z4 = np.zeros([28, 28, 32])

# Iterate through each of the 32 output channels
for i in range(32):
    for j in range(64):
        # Convolve the feature map with precision
        z4[:, :, i] = z4[:, :, i] + My_Conv2d_LT_Exact(
            np.array(z3[:, :, j]), np.flip(a[2][:, :, j, i]), t
        )

    # Add biases from the second layer
    z4[:, :, i] = z4[:, :, i] + a[3][i]
# Second convolution layer is complete

# Apply Rectified Linear Unit (ReLU) activation function
z5 = np.maximum(0, z4)

# Quantize the feature map
z5 = np.round((z5 / np.max(z5)) * 127)
# ReLU and quantization contribute to data precision
```

In this segment, precision is maintained while executing the second convolution layer, followed by ReLU activation and quantization.

3. **Remaining Convolution Layers**

   Our journey further explores the remaining convolution layers:

```
# Continue with the remaining Convolution Layers
z6 = np.zeros([28, 28, 16])

# Iterate through each of the 16 output channels
for i in range(16):
    for j in range(32):
        # Convolve the feature map with precision
        z6[:, :, i] = z6[:, :, i] + My_Conv2d_LT_Exact(
            np.array(z5[:, :, j]), np.flip(a[4][:, :, j, i]), t
        )
    # Add biases from the respective layer
    z6[:, :, i] = z6[:, :, i] + a[5][i]

# Apply Rectified Linear Unit (ReLU) activation function
z61 = np.maximum 0, z6)

# Quantize the feature map
z61 = np.round((z61 / np.max(z61)) * 127)

```

```
    # Remaining convolution layers are executed with precision
     z7=np.zeros([26,26,8])
    for i in range(8):
        for j in range(16):
             z7[:,:,i]=z7[:,:,i]+My_Conv2d_LT_Exact(np.array(z61[:,:,j]),np.flip(a[6][:,:,j,i]),t)
        z7[:,:,i]=z7[:,:,i]+a[7][i]
    z8=np.maximum(0,z7)                # ReLU
    z8=np.round((z8/np.max(z8))*127)   # Quantization
    z9=np.zeros([24,24,4])
    for i in range(4):
        for j in range(8):
             z9[:,:,i]=z9[:,:,i]+My_Conv2d_LT_Exact(np.array(z8[:,:,j]),np.flip(a[8][:,:,j,i]),t)
        z9[:,:,i]=z9[:,:,i]+a[9][i]
    z10=np.maximum(0,z9)              # ReLU
    z10=np.round((z10/np.max(z10))*127)  # Quantization
```

In this section, the precision of the data is upheld as we traverse the remaining convolution layers.

4. **Fully Connected Layers and Quantization**

As we approach the final steps, we handle fully connected layers and quantization:

```
    # Fully Connected Layers and Quantization
    z13 = np.reshape(z10, [1, -1])  # Flatten the feature map

    # Perform matrix multiplication with precision
    z14 = My_Matmul_LT_Exact(z13, a[10], t) + a[11]

    # Apply Rectified Linear Unit (ReLU) activation function
    z15 = np.maximum(0, z14)

    # Quantize the feature map
    z15 = np.round((z15 / np.max(z15)) * 127)

    # Further matrix multiplication with precision
    z141 = My_Matmul_LT_Exact(z15, a[12], t) + a[13]

    # Apply Rectified Linear Unit (ReLU) activation function
    z151 = np.maximum(0, z141)

    # Quantize the final feature map
    z151 = np.round((z151 / np.max(z151)) * 127)

    # Execute the last matrix multiplication with precision
    z16 = My_Matmul_LT_Exact(z151, a[14], t) + a[15]

```

```
    # Quantize the final result
    z16 = np.round((z16 / np.max(z16)) * 127)
    # Fully connected layers are executed with utmost precision

    # Return the predicted class and intermediate feature maps
    return np.argmax(z16), z3, z5, z61, z8, z10, z15, z151, z16
```

This segment encompasses the final stages of the Exact CNN, culminating in fully connected layers and quantization. The predicted class and intermediate feature maps are returned with precision.

Subtask 4 is a comprehensive representation of the Exact CNN, emphasizing the critical aspect of maintaining data integrity throughout the process.

$$
\begin{aligned}
\text{Input Data} &\xrightarrow{\text{Quantization}} QuantizedInput \\
&\xrightarrow{\text{1st Convolution Layer}} FeatureMap1 \\
&\xrightarrow{\text{ReLU Activation}} FeatureMap1(Activated) \\
&\xrightarrow{\text{Quantization}} QuantizedFeatureMap1 \\
&\xrightarrow{\text{2nd Convolution Layer}} FeatureMap2 \\
&\xrightarrow{\text{ReLU Activation}} FeatureMap2(Activated) \\
&\xrightarrow{\text{Quantization}} QuantizedFeatureMap2 \\
&\xrightarrow{\text{Remaining Convolution Layers}} \ldots \\
&\xrightarrow{\text{Fully Connected Layers}} FlattenedFeatureMap \\
&\xrightarrow{\text{Matrix Multiplication}} IntermediateFeatureMap1 \\
&\xrightarrow{\text{ReLU Activation}} IntermediateFeatureMap1(Activated) \\
&\xrightarrow{\text{Quantization}} QuantizedIntermediateFeatureMap1 \\
&\xrightarrow{\text{Matrix Multiplication}} IntermediateFeatureMap2 \\
&\xrightarrow{\text{ReLU Activation}} IntermediateFeatureMap2(Activated) \\
&\xrightarrow{\text{Quantization}} QuantizedIntermediateFeatureMap2 \\
&\xrightarrow{\text{Final Matrix Multiplication}} PredictedOutput \\
&\xrightarrow{\text{Quantization}} QuantizedPredictedOutput
\end{aligned}
$$

### 1.5 Subtask 5: Visualization of Convolution Outputs

1. **In this subtask,** we visualize the output of convolutional layers for an input image processed through the Exact CNN model using different multipliers.

   (a) **First,** we generate feature maps for a specific input image using the `Exact_CNN` function with various multipliers.

   (b) **Next,** we select a particular layer (e.g., Layer 1) from the generated feature maps.

   (c) **We create** a 3x3 grid of subplots to display the feature maps.

   (d) **For each** subplot, we check if there is an image to display (i.e., it's not empty), and if so, we plot the feature map and set a title indicating the approximate multiplier used.

   (e) **If there are** fewer feature maps than subplots, the remaining subplots are turned off.

   (f) **To enhance** visualization, we adjust the spacing between subplots for better clarity.

2. **In Python,** the following code accomplishes this visualization:

```
Vis_Mat = []  # List to store feature maps
for i in range(1):
    Vis_Mat.append(Exact_CNN(30, i))  # Generate feature maps

images = []  # List to store selected layer's feature maps
Layer_Number = 1  # Selected layer
for i in range(1):
    images.append(Vis_Mat[i][Layer_Number])  # Select the feature maps for the chosen layer

# Create a 3x3 grid of subplots
fig, axes = plt.subplots(3, 3, figsize=(8, 8))

# Loop through your image data and plot each image on a subplot
for i, ax in enumerate(axes.ravel()):
    # Check if there are more images than subplots
    image_data = np.average(images, axis=-1)
    if i < len(image_data):
        ax.imshow(image_data[i])  # Plot the image
        ax.set_title(f'Approximate Multiplier {i}')  # Set a title for the subplot
    else:
        ax.axis('off')  # Turn off the empty subplots if there are fewer images

# Adjust spacing between subplots for better visualization
plt.tight_layout()

# Display the plot
plt.show()
```

This subtask visualizes the effects of different multipliers on the feature maps generated by the Exact CNN model for a specific input image.

## 1.6 Subtask 6: Model Evaluation and Confusion Matrix

In Subtask 6, we evaluate the performance of the trained model and create a confusion matrix to gain insights into its ability to correctly classify the digit ”5.”

1. **Model Compilation and Evaluation**

   First, we compile the model using the ädamöptimizer and ßparse_categorical_crossentropy”loss function. Then, we evaluate the model's performance on the test data and print the test accuracy.

```
# Compile the model if it's not already compiled
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
```

```

# Print the test accuracy
print(f'Test accuracy: {test_accuracy}')
```

This segment assesses the model's performance and reports its accuracy on the test data.

2. **Confusion Matrix Creation**

   To further understand the model's performance, we create a confusion matrix specifically for the digit "5."We identify the indices of true "5"labels and predicted "5"labels. Then, we visualize the confusion matrix to analyze how well the model distinguishes between "5änd "not 5."

```
predictions = model.predict(test_images)

# Find the indices of true "5" labels and predicted "5"
    labels
true_indices = (test_labels == 5)
predicted_indices = (np.argmax(predictions, axis=1) ==
    5)

# Create a confusion matrix for the digit "5"
confusion = confusion_matrix(true_indices,
    predicted_indices)

# Visualize the confusion matrix
plt.figure(figsize=(6, 6))
sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues"
    , square=True,
            xticklabels=["Not 5", "Is 5"],
            yticklabels=["Not 5", "Is 5"])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title("Confusion Matrix for Digit '5'")
plt.show()
```

   This section generates the confusion matrix, providing a visual representation of the model's performance when distinguishing between "5änd "not 5."

In Subtask 6, we assess the model's accuracy and gain valuable insights into its ability to correctly classify the digit "5"through the visualization of a confusion matrix.