Introduction to Machine Learning

Hands-on 3

Classifying Hand-written numbers with convolutional neural networks using Pytorch

DEPARTAMENTO DE INGENIERÍA TELEMÁTICA Y ELECTRÓNICA

UNIVERSIDAD POLITÉCNICA DE MADRID

Alberto Martín Pérez
Miguel Chavarrías
Eduardo Juárez

Index

1. Introduction

a. What is Pytorch?

Pytorch is a machine learning framework used for applications such as computer vision and natural language processing. It was originally developed by Meta AI (Facebook) and is a free and open-source software. It is commonly used for developing and training neural network based deep learning models and is backed by huge companies such as Microsoft, Tesla or Uber. Although other frameworks exist in Python [1] to create deep neural networks such as Tensorflow, Pytorch is picking up fast and is immensely popular in research labs. [2]

b. MNIST database

The MNIST database is a group of handwritten digits images with 60,000 examples to be used for training, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. [8]

c. Structure of a convolutional neural network

Convolutional neural networks (CNNs) have the characteristic of being composed of two main stages: the feature extraction and the classification processes. In the feature extraction block, the convolutions are performed for detecting patterns This feature extraction stage outputs a vector of reduced characteristics that serves as an input for the classification stage, where a series of fully connected layers map a function to separate the data into the desired classes. In Figure 2 we can clearly see these two main stages. Given an input image of a car, the data is passed to the CNN in small groups of pixels from the image. These are often call patches, and after passing through the "hidden layers" the CNN extract some visual features of the image such as the shape of the object. Once the features have been obtained, the final stage is used to classify the given input, which in this case the output is a vector with probabilities for the different possible classes (car, truck, van, bicycle…)

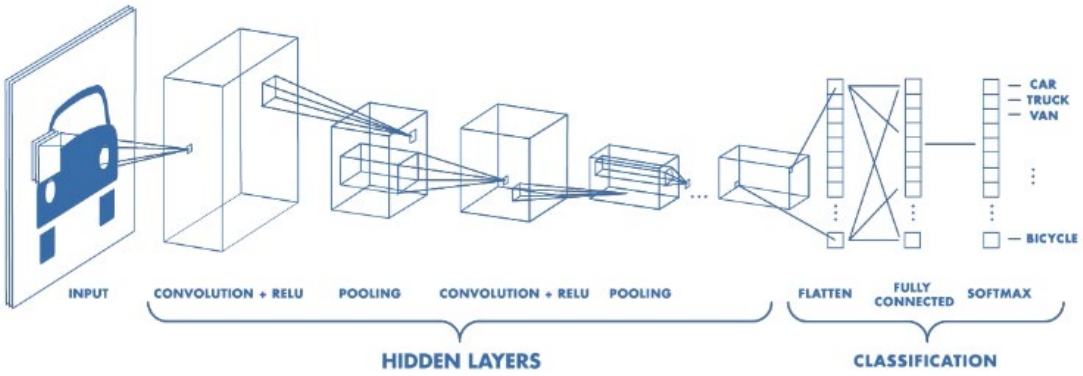


Figure 2: Architecture of a CNN [9]

2. Previous considerations

The exercises proposed in this document consist of a series of small programs written in Python, intended to get familiar with the Pytorch framework. For testing, it is possible to use both

1) Visual Studio IDE, installed in the virtual machine available in the Moodle's course site. The installed package already has Python installed and it can be straightforwardly used,

and

2) Google Colab. Although the use of the previous option is recommended, in case of difficulties with the virtual machine or if you want to make specific tests out of the normal working environment, you can use the Google's online coding tool, Colab [5].

Please note that the Virtual Machine password is: ***imlstudent2022***

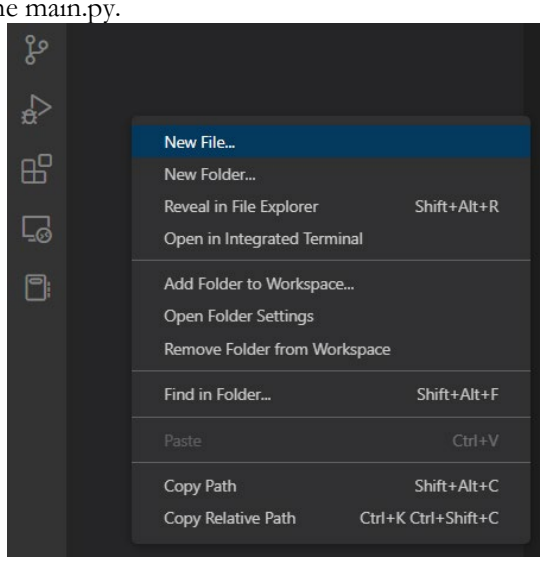
3. Hands-on 3: Classifying hand-written numbers with convolutional neural networks using pytorch

Before continuing, we need to ensure that the **torchvision** Python package is installed. To do so, execute the following code in the command line:

```
pip3 install torchvision
```

a. Loading and understanding data

1. Open Visual Studio Code (or your text editor of preference) to create and open a folder available in your computer (e.g. Desktop/Numbers_CNN/).
2. Download the files available in Moodle in the created folder and then unzip it.
3. Create a new Python file in the Numbers_CNN folder. To do so, right click > "New file..." and give it the name main.py.



4. Before learning how to train our CNN model, we need to download the MNIST dataset. For that purpose, we simply need to download it executing the following code:

```
from torch.utils.data import DataLoader
from torchvision import datasets
```

```
from torchvision.transforms import ToTensor

# Download the training data into the "data" folder
train = datasets.MNIST(root="data", download=True, train=True,
transform=ToTensor())
```

Please note that Pytorch work with **Tensors**, which basically is a data container very similar to a **Numpy array**. Therefore, we must convert the downloaded images to Pytorch tensors. This is easily done by passing the **ToTensor()** class to the *transform* parameter, which basically takes PIL or Numpy arrays of shape (height, width, channels) to a Tensor of shape (channels, height, width). Here channels refer to the number of color channels. If the image is grayscale, then *channels = 1*, if it is a color image, then *channels = 3* (for red, green and blue).

5. Once the data has been downloaded and transformed to Pytorch tensors, we will group the images in what is commonly named as "batches". To do so, we will run the next code to create batches with 32 images each:

```
# Group the training dataset into batches of size 32
train_batches = DataLoader(dataset=train, batch_size=32)
```

6. To better understand what has been saved in *train_batches*, we are going to check its longitude and iterate over it, which can be done with a simple "for" loop:

```
print(f"Number of created batches: {len(train_batches)}")
# Understand the data loaded
for batch in train_batches:

    # Check what is "batch"
    print(f"Type of 'batch': {type(batch)}")

    # Each batch returns the data (the images) and their corresponding labels
    data, labels = batch

    # Check what are "data" and "labels"
    print(f"Type of 'data': {type(data)}")
    print(f"Type of 'labels': {type(labels)}")

    # Check the shape of "data" and "labels"
    print(f"Shape of data Tensor from current batch: {data.shape}")
    print(f"Shape of first image Tensor from current batch: {data[0].shape}")
    print(f"Shape of labels Tensor from current batch: {labels.shape}")
    print(f"Labels included in the 'labels' Tensor from current batch:
{labels}")

    # Use "break" to exit the for loop
```

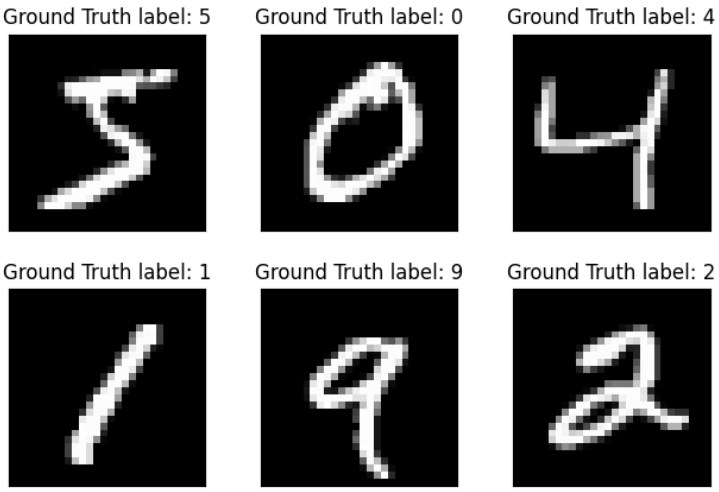
As you can see, each *batch* is a Python list including two Pytorch tensors. These are the *data* images included in the batch and their corresponding *labels* (which can be: 0, 1, 2, …, 9).

Exercise 1: Knowing the number of created batches and how many images each of them contains, how many training images do we have? Does it match the number of training images in MNIST? **Tip**: Check this using reference [8].

7. Until this point, we've numerically check what we are working with. Wouldn't it be better to visualize the images? Let's do that now. First, we will import **matplotlib.pyplot** module, create a figure and show the first 6 images from the first training batch. This can be done by executing the following code:

```
import matplotlib.pyplot as plt

# Create a figure to plot the images
plt.figure()

for batch in train_batches:
    # Get the images and the labels for the current training batch
    data, labels = batch

    # Show the first 6 images in the plot
    for i in range(6):
        plt.subplot(2, 3, i + 1)
        plt.tight_layout()
        plt.imshow(data[i][0], cmap="gray", interpolation="none")
        # Indicate the label provided to the image
        plt.title(f"Ground Truth label: {labels[i]}")
        plt.xticks([])
        plt.yticks([])

    plt.show()
    plt.close()

    # Use "break" to exit the for loop
    break
```

This will provide a figure like this one:



b. Build and train a CNN model

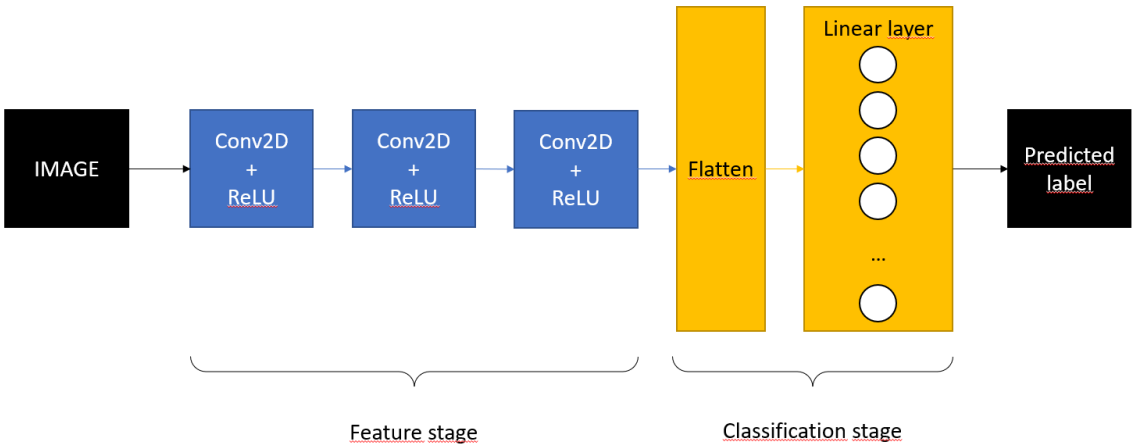
Now we know what we want our CNN to classify. Once it is trained, we want our model to provide us a label with the number of the image. For example, if the new image is a handwritten 9, we want our model to provide a 9 label. To achieve that, we need to build the CNN architecture first and then train it.

1. All Pytorch models should be children of the **torch.nn.Module** class [9]. This will ease the creation of models and will provide many useful functionalities such as GPU acceleration, set the model to training mode or evaluation mode, compute forward and backward propagation and many more. Let's define our module by using the following code:

```
# Define the Convolutional Neural Network architecture
import torch

# CNN MNIST classifier
class DigitClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
```

In case you are not familiar with object oriented programming (OOP) in Python, the way to inherit from a class is by defining our class and include in parenthesis the parent class. Here we define our model class ***"DigitalClassifier"*** to inherit from ***"torch.nn.Module".*** Then, we define the class constructor using the "***def __init__()"*** method. Typically, ***self*** is a reserved word used to indicate the instance of the class, but you can change it as long as you are consistent in the entire class. Finally, to inherit all attributes, properties and methods from ***"torch.nn.Module"***, we need to call our parent constructor inside ours. This is done by using the ***super()*** (which calls the parent class) and calling its constructor method (*"__init__()"*).

2. Although we have defined our model class, we still need to define the architecture of our neural network. One way to so is by defining an attribute storing all layers of the architecture inside the constructor method (*"def __init__()"*). But first, we need to define the architectures and understand the available layers in Pytorch. The architecture we will implement is described in the following figure:



As you can see, each block contains a hidden layer and its activation function. Specifically, the three layer in the feature stage contain a 2D convolutional layer [10] followed by ReLU activation function [11]. The ***"torch.nn.Conv2d"*** class implements the 2D convolutional layer that is applied to the image to extract features (such as the shape of the handwritten digit). This class requires the input and output channels as well as the kernel size. For the

first layer, the input channel (here the color channels) is 1, since the images in MNIST are grayscale images. The output channels can be any number that we want, although we recommend powers of two (16, 32, 64…). Finally, the kernel size is the dimension of the small window that is moved around the images to extract its features. If *kernel_size = (1*, then we check pixel by pixel, which is not useful to extract spatial features of the image. It is recommended to use odd numbers to be able to centre the kernel window in every pixel in the image. In our case, we will set a small kernel with *kernel_size = (3,3)*. If you want to get more details about how the convolutional operation is performed, it is encouraged to visit [14] to visually understand it. Right after each convolution stage, we will employ the ReLU activation function to introduce **non-linearity** in the network, otherwise, our network would essentially be a "sophisticated" linear regression model. We will use the ***"torch.nn.ReLU"*** class from Pytorch. Once the feature stage has been configured, we need to define the classification stage to obtain the classified output from any handwritten image. We will use a linear layer for this purpose, but first, we need to adapt the output of the latest convolutional layer (which will be a 2D matrix) to a vector array (1D) for the linear layer. This can be achieved using the ***"torch.nn.Flatten"*** [12] class implemented in Pytorch. Then, we will define the latest hidden layer with a linear layer using the ***"torch.nn.Linear"*** [13] class. Finally, we will make use of the ***"torch.nn.Sequential"*** [15] class to connect all previously defined layer with each other and add the ***fordward()*** method to be able to perform the forward propagation. The structure of our architecture would look like this in code:

```
# CNN MNIST classifier
class DigitClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model = torch.nn.Sequential(
            # Feature extraction stage
            torch.nn.Conv2d(…),
            torch.nn.ReLU(),
            torch.nn.Conv2d(…),
            torch.nn.ReLU(),
            torch.nn.Conv2d(…),
            torch.nn.ReLU(),
            # Classification stage
            torch.nn.Flatten(),
            torch.nn.Linear(…)
        )

    def forward(self, x):
        return self.model(x)
```

3. Although you might think that the architecture is ready to use, we still need to configure some parameters of the defined layers. This is an important step to guide the behaviour of the CNN. Although we can optimize these hyperparameters to find a suitable configuration, it will take too much time for us. Therefore, the parameters we will configure for the indicated layers are the following ones:
    - **Conv2d:** in_channels, out_channels, kernel_size
    - **Linear:** in_features, out_features

    The other parameter that we will use are the default ones.

Exercise 2: Configure the mentioned parameters for each of the defined layers. To do so, check the documentation provided ([10] and [13]), to select their values. Note that the values you define will influence further parameters along the CNN layers.

Tip 1: The value of each *output_channels* should be the same as next *in_channels* values. For example, if *output_channels=32* in the first convolutional layer, then the *in_channels=32* for the second convolutional layer.

Tip 2: The value of *in_channels* from the first convolutional layer must be 1 since the input images only have 1 channel (recall that they are grayscale images).

Tip 3: The value of *out_features* for the linear layer must be 10 since the images can be any number from 0 to 9.

Tip 4: You need to use the equation below to know the output spatial dimensions of each convolutional layer. This is included in [10] and is necessary to know the value of *in_features* for the *torch.nn.Linear* layer. By default, *padding = 0*, *dilation = 1* and *stride = 1*. This should be applied to the width and heigh dimensions of the images, but since they are squared images, the resulted value can be used for both dimensions.

$$W_{out} = H_{out} = \left\lfloor \frac{H_{in} + 2 \times padding - dilation \times (kernel_{size} - 1) - 1}{stride} + 1 \right\rfloor$$

Tip 5: To calculate *in_features* you must apply the previous equation 3 times, one for each call to ***torch.nn.Conv2d()***. The output of the latest convolution is a 3D array with shape $(out_3, W_{out}, H_{out})$ equivalent to (number of channels, width, heigh). This has to be flattened into a 1D vector. For that purpose, you would need to multiply all dimensions together such as $out_3 \times W_{out} \times H_{out}$

You should end with a code like the one below. Therefore, you need to manually define out_1, out_2 and kernel_size as you wish. Then, use those parameters to compute in_features employing the equation above.

```
# CNN MNIST classifier
class DigitClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model = torch.nn.Sequential(
            # Feature extraction stage
            torch.nn.Conv2d(in_channels=1, out_channels=out_1,
kernel_size=(kernel_dim, kernel_dim)),
            torch.nn.ReLU(),
            torch.nn.Conv2d(in_channels=out_1, out_channels=out_2,
kernel_size=(kernel_dim, kernel_dim)),
            torch.nn.ReLU(),
```

```
            torch.nn.Conv2d(in_channels=out_2, out_channels=out_3,
kernel_size=(kernel_dim, kernel_dim)),
            torch.nn.ReLU(),
            # Classification stage
            torch.nn.Flatten(),
            torch.nn.Linear(in_features=in_features, out_features=10),
        )
```

You will be able to pass to the next point once you are able to execute the following code without encountering the following error:
`RuntimeError: mat1 and mat2 shapes cannot be multiplied…`

```
    # Instance of the neural network
    cnn = DigitClassifier()

    for batch in train_batches:
        X, y = batch
        y_pred = cnn(X)
        exit()
```

4. Once you can execute the previous code, you are ready to define the optimizer during gradient descent and define a loss function to minimize. This will help us upgrade the weights stored inside the CNN to train more robustly and be able to predict better. By using the "*torch.optim.Adam*" [16] and "*torch.nn.CrossEntropyLoss*" [17] classes we will be able to implement the optimizer and the loss function respectively.

```
# TRAINING PROCEDURE
# Instance of the neural network, loss, optimizer
cnn = DigitClassifier()
opt = torch.optim.Adam(cnn.parameters(), lr=1e-3)
loss_fn = torch.nn.CrossEntropyLoss()
```

5. Then, we are ready to train the CNN model and perform forward and backward propagations. The following code computes the necessary operations over every batch as many times as we define with *num_epochs* (first use 1 and if you want to train more in depth, add more epochs):

```
# Number of times to pass through every batch
num_epochs = 1

# Set the CNN to training mode
cnn.train(True)

for epoch in range(num_epochs):

    for batch_idx, (X, y) in enumerate(train_batches):
        # Pass the batch of images to obtain a prediction
        y_pred = cnn(X)

        # Compute the loss comparing the predictions
```

```
        # made by the CNN with the original labels
        loss = loss_fn(input=y_pred, target=y)

        # Perform backpropagation with the computed loss
        # to compute the gradients
        loss.backward()

        # Update the weights with regard to the computed
        # gradients to minimize the loss
        opt.step()

        # In each iteration we want to compute new gradients,
        # that is why we set the gradients to 0
        opt.zero_grad()

        # Print to check the progress
        if batch_idx % 50 == 0:
            print(
                f"Train Epoch: {epoch} [{ batch_idx
*len(data)}/{len(train_batches.dataset)} ({100.0 * batch_idx /
en(train_batches):.0f}%)]\tLoss: {loss.item():.6f}"
            )

    print(f"Epoch: {epoch} loss is {loss.item()}")
```

6. If you want to save the state of your trained CNN model in disk memory or load its state, you can use the following code to do so:

```
from torch import save, load

# Save the model state
with open("./cnn_model_state.pt", "wb") as file:
    save(cnn.state_dict(), file)

# Load the model state
with open("./cnn_model_state.pt", "rb") as file:
    cnn.load_state_dict(load(file))
```

7. To evaluate how well the model performs, we can predict the test images in the MNIST database. For that purpose, we will download the data and create batches as we did with the training data.

```
# Download the test data into the "data" folder
test_data = datasets.MNIST(root="data", download=True, train=False,
transform=ToTensor())

# Group the training dataset into batches of size 32
test_batches = DataLoader(dataset=test_data, batch_size=32)

# Set the model to evaluation mode to predict data
```

```
cnn.eval()

with torch.no_grad():  # Avoid calculating gradients

   # Predict the images in batches
    for images, labels in test_batches:
        # Predict the images with probabilities
        test_output = cnn(images)

        # For each image, take the highest probability
        y_test_pred = torch.max(test_output, 1)[1].data.squeeze()

        # Measure the accuracy
        accuracy = (y_test_pred == labels).sum().item() /
float(labels.size(0))

print("VALIDATION SET ACCURACY: %.2f" % accuracy)
```

c. Predict with the CNN model

To visualize the prediction of the CNN model with a new image, we can use the test dataset that we have downloaded from the MNIST database:

1. Before extracting an image from the test dataset, we need to remember how the CNN model has been trained. If you recall from previous steps, we passed groups of images during the training or what is commonly named as "batches". These batches had the following shape: $(32, 1, 28, 28) \equiv (num_{images}, num_{channels}, img_{height}, img_{width})$. If we take a single image with shape $(28, 28, 1)$ and convert it to a Pytorch tensor to have shape $(1, 28, 28)$, we quickly see that our model has been trained with tensors with 4 dimensions instead of 3. To obtain an image from the test dataset and reshape it so that the CNN can predicted we can use do:

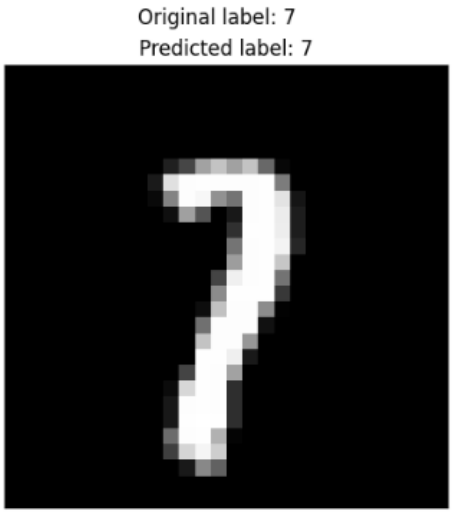
```
# PREDICT A SINGLE IMAGE IN THE DATASET
img_idx = 111 # <-- Change this value to select other image

img_tensor, label = test_dataset[img_idx]
print(f"Shape of tensor: {img_tensor.shape}")

img_tensor = img_tensor.unsqueeze(...) # <-- Add the right value
print(f"Shape of tensor: {img_tensor.shape }")
```

   Exercise 3: How *"unsqueeze()"* works? Try changing the parameter value so that the end shape of *img_tensor* is $(1, 1, 28, 28)$. **Tip**: Read the documentation available in Pytorch to understand better its functionality.

2. Now that we have our single image as a tensor with the desired shape, we need to configure the CNN model to not calculate gradients when a new image is passed. This way we can predict a new image without altering the behaviour of the trained model. Then, we can pass the image and obtain the label predicted. Finally, we will plot the results to visually check the performance of the CNN model. This can be done using the following code:

```
# Predict with the CNN
with torch.no_grad():  # Avoid calculating gradients

    # Generate the prediction
    img_pred = cnn(img_tensor)
    pred_label = torch.argmax(img_pred)

# Plot the result
plt.figure()
plt.suptitle(f"Original label: {label}")
plt.title(f"Predicted label: {pred_label}")

# From the tensor extract only the image
plt.imshow(img_tensor[0, 0, :, :], cmap="gray")

# Remove the numbers in the axes
plt.xticks([])
plt.yticks([])
plt.savefig(f"./prediction_of_test_image_{img_idx}.png")
```

The result would be like:



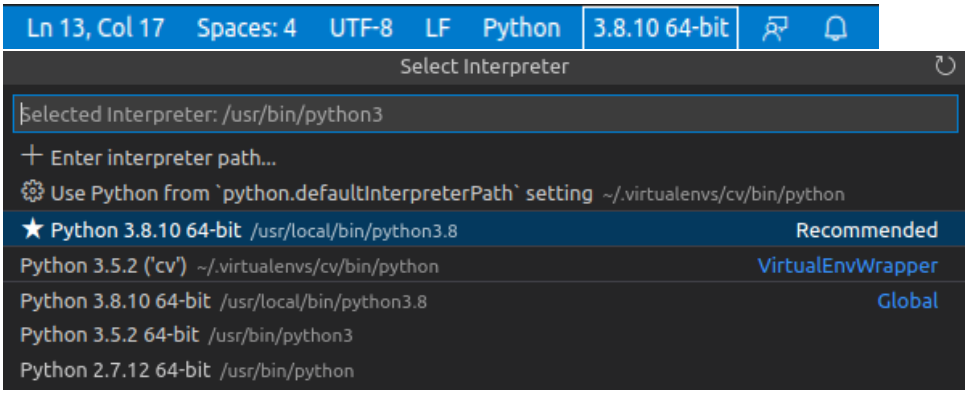
4. Troubleshooting

The most common errors are listed below, and the corresponding suggested workarounds are proposed.

Visual Studio Code says ImportError: No module named 'sklearn' or 'ImportError: cannot import name 'DecisionBoundaryDisplay'

- To solve this issue, you need to ensure that the Python version used to run the program have the 'sklearn' package installed. We recommend to use the already installed Python 3.8.10 version. To do so, make sure that in the bottom right corner of Visual Studio Code the

version used is 3.8.10 64-bit. Changing the version is as simply as clicking in the version and then select the indicated interpreter:



I can't show a figure when running an experiment.

- If you are using the Python 3.8.10 64-bit Python interpreter probably you encounter the "*UserWarning: Matplotlib is currently using agg, which is a non-GUI backend, so cannot show the figure.*" error. If so, this has to do with the line of code "**plt.plot()**". This problem has to do with not having a GUI backend that allows the Matplotlib python package to display a figure. The easiest way to solve this problem is saving the figure instead of plotting it. To do so, just replace the "plt.show()" function call with "**plt.savefig('name_of_my_figure.png')**" call. For more information regarding saving figures with Matplotlib, you can visit the documentation using this link.

5. References

[1] Python, available in URL: https://www.python.org/doc/essays/blurb/

[2] Pytorch, available in URL: https://pytorch.org/

[3] Müller, Andreas C., and Sarah Guido. Introduction to machine learning with Python: a guide for data scientists. " O'Reilly Media, Inc.", 2016.

[4] SVM implementation with scikit-learn, available in URL: https://scikit-learn.org/stable/modules/svm.html

[5] Google Colab, available in URL: https://colab.research.google.com/

[6] Scipy, available in URL: https://scipy.org/

[7] Python dictionaries, available in URL: https://realpython.com/python-dicts/

[8] MNIST database, available in URL: http://yann.lecun.com/exdb/mnist/

[8] Introduction to Deep Learning: What are Convolutional Neural Networks?, Mathworks, available in URL: https://es.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html

[9] Pytorch nn.Module class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.Module.html

[10] Pytorch nn.Conv2d class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

[11] Pytorch nn.ReLU class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

[12] Pytorch nn.Flatten class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.Flatten.html

[13] Pytorch nn.Linear class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

[14] Conv2d: Finally Understand What Happens in the Forward Pass, available in URL: https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148

[15] Pytorch nn.Sequential class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html

[16] Pytorch optim.Adam class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

[17] Pytorch nn.CrossEntropyLoss class documentation, available in URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[18] Pytorch unsqueeze() function, available in URL: https://pytorch.org/docs/stable/generated/torch.unsqueeze.html