PyTorch-Dev

March 3, 2026

1 Machine Learning with PyTorch

1.1 Table of contents

1.2 Introduction

So far, we have used the Scikit Learn library for our machine learning (ML) work. While this provides a wide range of algorithms in a user-friendly package, it doesn't implement all the types of neural network we might want to use. We will therefore start to look at PyTorch, which is probably the most widely used ML library for academic work. After discussing how to install PyTorch, we introduce some fundamental concepts important for the understanding of neural networks in general and convolutional neural networks in particular. These include convolution (or cross-correlation), padding, pooling, regularising (using L1 or L2 regularisation or dropout) and some comments on loss functions. We mention the quirks of reading images in PyTorch and then show how networks incorporating CNNs can be contructed and how these can be applied in image analysis. Note that PyTorch is a complex package, so we can only briefly touch on some of the things PyTorch can do in this lecture!

1.3 Installing PyTorch

Here, we will install PyTorch in the Anaconda3 framework. If you haven't already done so, you can download Anaconda3 from the webiste linked here and install it as per usual on your computer. This will provide you with a `base` environment for your day-to-day work.

I suggest you install PyTorch in a new conda environment to ensure it doesn't break your `base` environment. To do this, open an Anaconda Prompt terminal (e.g. from the Anaconda3 menu in the Windows Start GUI or Apple's Finder). This will open in your `base` environment: the prompt will include the string `(base)`. Then enter:

```
(base) PS C:\Users\green> conda create -n pytorch python <Enter>
```

(I'll stop writing the prompt `(base) PS C:\Users\green>` and indicating that you have to press `<Enter>` after each line from now on!)

You can run PyTorch using your computers CPUs or, if you have the the right procesors, its CPUs and GPUs. The latter can speed up calculations significantly in some cases. To use the GPUs, you must check that your processors are of the right type. For Nvidia GPUs, look here, and for Intel GPUs here. If your GPUs are suitable and you want to use them, you will need to install some software. What you need depends on what kind of GPUs your computer has.

CPUs
No additional software needed.

Nvidia GPUs
See here for information on using PyTorch with CUDA, the software that allows you to exploit your GPUs for PyTorch calculation.

Make sure your Nvidia graphics driver is up to date.

Install Microsoft Visual Studio 2022 Community (or MS Visual Studio 2019 Community). You can get this from the Microssoft Store. During the installation, tick the following options:
* Python development * Nodejs development * Data science and analytical applications
If you already have Visual Studio, use the Visual Studio Installer in the Windows Start menu to check that you have these options installed and install them if you haven't.

Install the CUDA toolkit. You can download the CUDA package from here.

Intel GPUs
See here for information on what you must install.

Make sure the Intel driver for your GPUs is up to date.

Install Microsoft Visual Studio 2022 Community (or MS Visual Studio 2019 Community). You can get this from the Microsoft Store. During the installation, tick the following options: * Python development * Desktop development with C++ * Data science and analytical applications
If you already have Visual Studio, use the Visual Studio Installer in the Windows Start menu to check that you have these options installed and install them if you haven't.

Install Intel deep learning essentials.

Pytorch install

Once all the prerequisites are in place:

Activate your `pytorch` environment:

```
conda activate pytorch
```

Install PyTorch and some of its associated packages. (Choose the appropriate line below depending on whether you want to run PyTorch on your CPU, on Nvidia GPUs or on Intel GPUs.)

CPUs

```
pip3 install torch torchvision torchaudio
```

Nvidia GPUs
Assuming you have installed CUDA 12.6, run:

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu126
```

If you have installed a different version of CUDA (e.g. CUDA 11.8) then you have to change the above to:

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
```

Intel GPU

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/xpu
```

Note that we have to use `pip` to install PyTorch: the `conda` version of PyTorch is outdated and it seems that it is no longer maintained.

Get any other packages you want. For example,

```
conda install matplotlib pandas scikit-learn pandoc nodejs
```

To move back to your `base` environment, run

```
conda deactivate
```

If you want to be able to use Jupyter Lab and Jupyter Notebook in your pytorch environment without reinstalling them, in your `base` environment run:

```
conda install nb_conda_kernels
ipython kernel install --user --name=pytorch
```

When you run Jupyter Lab, you will then have the option to select the `pytorch` kernel, which implies you are working in the `pytorch` environment.

While you can update your `base` environment, using `conda update --all`, I would recommend that you don't do this in the `pytorch` environment. It has been established using a mix of the `conda` and `pip` package managers. While these will ensure that the inital set-up is consistent (the installed versions of all the packages are compatible) this is not 100% guaranteed if either `conda` or `pip` is used to update the environment. (See here for more information.)

If you are using **Intel GPUs**, you must run the following commands once every time you start a session, before you do anything else, to set up some environment variables. (More information is avialable in the Intel documentation.)

[1]:
```
comString = "C:/Program Files (x86)/Intel/oneAPI/compiler/2025.0/env/vars.bat"
outString = get_ipython().getoutput(comString)
```

```
if len(outString) > 0:
    print(outString)
```

[2]:
```
comString = "C:/Program Files (x86)/Intel/oneAPI/ocloc/2025.0/env/vars.bat"
outString = get_ipython().getoutput(comString)
if len(outString) > 0:
    print(outString)
```

You can now test your PyTorch installation. Activate `pytorch`:

```
conda activate pytorch
```

Start Jupyter Lab (or Notebook if you prefer)

```
jupyter lab
```

Run the cell below. You should see an array (or, in PyTorch terminology, a tensor) with five rows and three columns, full of random numbers.

[3]:
```
import numpy as np
import torch
import torch.nn as nn
#
print(" ")
print("Torch version",torch.__version__)
print(" ")
x = torch.rand(5, 3)
print(x)
```

```
Torch version 2.10.0+xpu

tensor([[0.5931, 0.2599, 0.8637],
        [0.7849, 0.9496, 0.3743],
        [0.7638, 0.4200, 0.4212],
        [0.8979, 0.3462, 0.7842],
        [0.0674, 0.4300, 0.6737]])
```

You can test whether your computer has Nvidia GPUs that can be used to speed up PyTorch.

[4]:
```
torch.cuda.is_available()
```

[4]:
```
False
```

My computer doesn't have Nvidia GPUs which can be programmed using CUDA, hence the `False` above, but it does have Intel GPUs.

[5]:
```
torch.xpu.is_available()
```

[5]:
```
True
```

These can be used provided the correct version of PyTorch is intalled.

[6]:
```
torch.__version__
```

[6]:
```
'2.10.0+xpu'
```

1.4 Convolutions

1.4.1 Convolutions in 1D

Convolutions (in the context of this lecture) are operations that are performed on (parts of) images. Consider a one dimensional "image" $x = [1, 2, 1, 3]$. If we convolve this using the kernel $w = [1, 2, 3]$, we must form the dot product of elements in the image with the kernel, centred at each element of the image in turn. For the second element in the image (the "2"), this will give:

$$1 \times 1 + 2 \times 2 + 3 \times 1 = 8.$$

For the third element (the second "1") it gives:

$$1 \times 2 + 2 \times 1 + 3 \times 3 = 13.$$

As the kernel must be centred on the image elements, it must have an odd number of elements.

A health warning There is a disagreement about terminology here. The above procedure is what CNNs call convolution. In the mathematical definition however, the kernel is first flipped before the dot products are formed, i.e. the vector used in forming the dot product is $[3, 2, 1]$. What CNNs call convolution is what mathematicians refer to as cross-correlation!

We can't apply the convolution to the first and last elements of the above image, as there aren't any elements before the index element (in the first case) and after the index element (in the second). In order to deal with this, we introduce the idea of padding.

1.4.2 Padding in 1D

Padding is the process of adding p zeros before and after an image array. For example $x = [1, 2, 1, 3]$ becomes $x_p = [0, 1, 2, 1, 3, 0]$ when padded with $p = 1$. If this is done, then the kernel $w = [1, 2, 3]$ can be applied to the image to give:

$$[1 \times 0 + 2 \times 1 + 3 \times 2, 1 \times 1 + 2 \times 2 + 3 \times 1, 1 \times 2 + 2 \times 1 + 3 \times 3, 1 \times 1 + 2 \times 3 + 3 \times 0] = [8, 8, 13, 7].$$

Padding with $p = 1$ will allow a kernel of length three to be applied to an image and will give a result of the same size as the image. A kernel of length five will require that padding is applied with $p = 2$ if the image size is to be preserved. In general, if the kernel has n elements, padding with $p = (n - 1)/2$ will be needed to maintain the image size.

1.4.3 Examples in 1D

Need to set the environment variable `KMP_DUPLICATE_LIB_OK` to ensure matlplotlib works with pytorch!

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

[7]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
#
import torch
import torchvision
import numpy as np
```

[20]:
```
def conv1d(x, w, p = 0, math = False):
    '''
    Given 1D image x and kernel w, pad image with p zeros then convolve with␣
 ↪kernel
    '''
    #
    # Mathematical version
    if math:
        w_r = np.array(w[::-1])
    #
    # CNN version
    else:
        w_r = np.array(w[:])
    #
    len_xp = 2*p + len(x)
    x_padded = np.zeros(len_xp)
    x_padded[p:len(x) + p] = x[:]
    result = np.zeros(len_xp - len(w_r) + 1)
    for n in range(0, len_xp - len(w_r) + 1):
        result[n] = np.sum(x_padded[n:n + w_r.shape[0]]*w_r)
    return result
#
# Testing:
x = [1, 2, 1, 3]
w = [1, 2, 3]
#
math = False
print(" ")
print("Math mode =", math)
print("Input image:", x)
print("Kernel:", w)
print("Conv1d:",conv1d(x, w, p = 1, math = math))
if math:
    print("Numpy convolve:",np.convolve(x, w, mode = 'same'))
else:
    print("Numpy correlate:",np.correlate(x, w, mode = 'same'))
#
```

```
x = [1, 2, 1, 3, 7, 8]
w = [1, 2, 3, 2, 3]
#
print(" ")
print('Input image:', x)
print('Conv1d:', conv1d(x, w, p = 2, math = math))
if math:
    print('Numpy convolve:', np.convolve(x, w, mode = 'same'))
else:
    print('Numpy correlate:', np.correlate(x, w, mode = 'same'))
print(" ")
```

```
Math mode = False
Input image: [1, 2, 1, 3]
Kernel: [1, 2, 3]
Conv1d: [ 8.  8. 13.  7.]
Numpy correlate: [ 8  8 13  7]

Input image: [1, 2, 1, 3, 7, 8]
Conv1d: [10. 19. 35. 51. 44. 41.]
Numpy correlate: [10 19 35 51 44 41]
```

Note, the following NumPy code is used above to flip the `w` array!

[21]:
```
w = np.array([1, 2, 3])
w_s = w[:]
w_r = w[::-1]
print(" ")
print("Input kernel:\n",w)
print(" ")
print("Copied kernel:\n",w_s)
print(" ")
print("Flipped kernel:\n",w_r)
print(" ")
```

```
Input kernel:
 [1 2 3]

Copied kernel:
 [1 2 3]

Flipped kernel:
 [3 2 1]
```

1.4.4 Convolutions in more dimensions

The above extends to two and more dimensions.

This (mathematical!) convolution operation is referred to as the Frobenius inner product.

1.4.5 Examples in 2D

[22]:
```
import scipy.signal
#
def conv2d(X, W, p = (0, 0), math = False):
    '''
    Given 2D image X and kernel W, pad image with p[0] zeros (rows) and p[1]␣
 ↪zeros (cols)
    then convolve with kernel
    '''
    # Mathematical version
    if math:
        w_r = np.array(W)[::-1, ::-1]
    #
    # Cross correlation
    else:
        w_r = np.array(W)
    #
    X_orig = np.array(X)
    nr = X_orig.shape[0] + 2*p[0]
    nc = X_orig.shape[1] + 2*p[1]
    X_padded = np.zeros(shape = (nr, nc))
    X_padded[p[0]:p[0] + X_orig.shape[0], p[1]:p[1] + X_orig.shape[1]] = X_orig
    #
    result = np.zeros((X_padded.shape[0] - w_r.shape[0] + 1,
                       X_padded.shape[1] - w_r.shape[1] + 1))
    for r in range(0, X_padded.shape[0] - w_r.shape[0] + 1):
        for c in range(0, X_padded.shape[1] - w_r.shape[1] + 1):
            X_sub = X_padded[r:r + w_r.shape[0], c:c + w_r.shape[1]]
            result[r, c] = np.sum(X_sub*w_r)
        #
    return result
#
X = np.array([[1, 3, 2, 4],
              [5, 6, 1, 3],
              [1, 2, 0, 2],
              [3, 4, 3, 2]])
W = np.array([[1, 0, 3],
              [1, 2, 1],
              [0, 1, 1]])
#
math = False
```

```
print(" ")
print("Math mode =", math)
print("Input image: \n", X)
print(" ")
print("Kernel: \n", W)
print(" ")
print('Conv2d:\n', conv2d(X, W, p = (1, 1), math = math))
#
print(" ")
if math:
    print('SciPy convolve:\n',
    scipy.signal.convolve2d(X, W, mode = 'same'))
else:
    print('SciPy cross correlate:\n',
    scipy.signal.correlate2d(X, W, mode = 'same'))
print(" ")
```

```
Math mode = False
Input image:
 [[1 3 2 4]
 [5 6 1 3]
 [1 2 0 2]
 [3 4 3 2]]

Kernel:
 [[1 0 3]
 [1 2 1]
 [0 1 1]]

Conv2d:
 [[16. 16. 15. 13.]
 [28. 27. 28. 11.]
 [29. 20. 24.  7.]
 [16. 15. 20.  7.]]

SciPy cross correlate:
 [[16 16 15 13]
 [28 27 28 11]
 [29 20 24  7]
 [16 15 20  7]]
```

Again, the `W` array is reversed using the code

[23]:
```
W = np.array([[1, 0, 3],
              [1, 2, 1],
              [0, 1, 1]])
```

```
W_c = W[:]
W_r = W[::-1, ::-1]
print(" ")
print("Input array: \n",W)
print(" ")
print("Copied array: \n",W_c)
print(" ")
print("Flipped array:\n",W_r)
print(" ")
```

```
Input array:
 [[1 0 3]
 [1 2 1]
 [0 1 1]]

Copied array:
 [[1 0 3]
 [1 2 1]
 [0 1 1]]

Flipped array:
 [[1 1 0]
 [1 2 1]
 [3 0 1]]
```

Efficient algorithms for computing convolutions (in the sense of CNNs!) are described in Fast Algorithms for Convolutional Neural Networks..

1.5 Pooling

Pooling is the process of taking a block of pixels in the input image and using the values to calculate a single output. Typical choices are the maximum value in the block or the average value of the pixels in the block. In one dimension, pooling the image $[1, 2, 2, 5, 1]$ in blocks of three, taking the maximum value, would result in the image $[2, 5, 5]$.

1.6 Reading an image file with pytorch

Another health warning

The keyword `Image` is used in two contexts here:

`from IPython.display import Image`

and

`from PIL import Image`

Make sure the required version is used or there will be problems!

`Imshow` in PIL expects images to be in the format height (number of pixel rows), width (number of pixel columns), channels (1 if black and white, 3 if RBG or YMC, 4 if RBG or YMC with opacity).

[25]:
```
from IPython.display import Image
Image(filename = 'Jasper.jpg')
```

[25]:

[26]:
```
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline
#
img_pil = Image.open("Jasper.jpg")
img_pil_arr = np.asarray(img_pil)
print('Image shape:', img_pil_arr.shape)
plt.imshow(img_pil)
plt.show()
```

```
Image shape: (3888, 5184, 3)
```

Health warning continued!

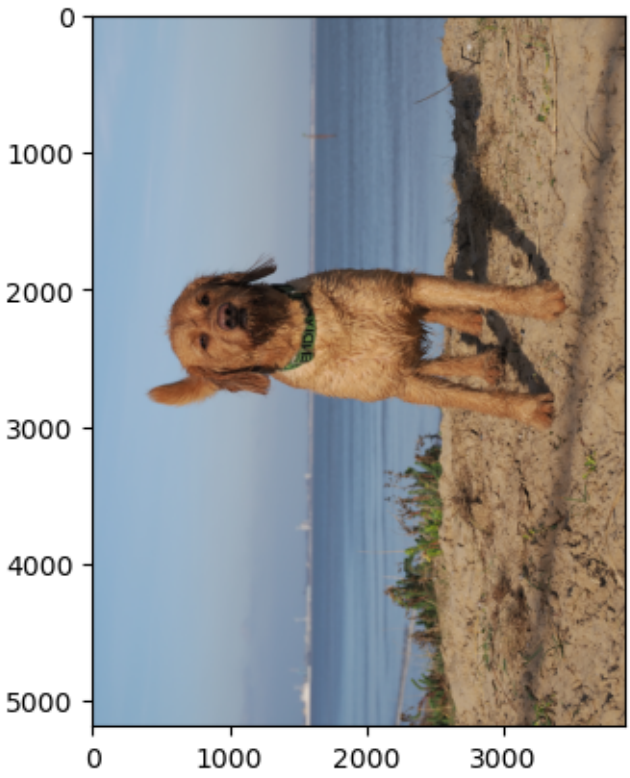
In contrast, `read_image` from `torchvision.io` expects images in the format channel, height, width, so if an image is read using `read_image`, it must be permuted before it can be displayed using `imshow`.

[27]:
```
import torch
from torchvision.io import read_image
#
img = read_image('Jasper.jpg')
#
print('Image shape:', img.shape)
print('Number of channels:', img.shape[0])
print('Image data type:', img.dtype)
#
plt.imshow(img.permute(1, 2, 0))
plt.show()
#
print(" ")
print("Just to illustrate what permute does!")
#
plt.imshow(img.permute(2, 1, 0))
```

```
plt.show()
```

```
Image shape: torch.Size([3, 3888, 5184])
Number of channels: 3
Image data type: torch.uint8
```

Just to illustrate what permute does!



1.7 Loss functions for classification problems

Binary cross-entropy is the appropriate loss function for binary classification (with a single output unit), while categorical cross-entropy is the relevant loss function for multiclass classification (with one output unit for each class). Examples of calculations using these loss functions are given below.

Note that `logits` in this context are the vectors of ouputs from the final layer of a classification network, before application of the sigmoid (or other) node function. The `probas` are the result of applying the output node function to the `logits`. (They aren't necessarily or even usually real probabilities, despite the terminology!) For the more formal defininition, see here.

[31]:
```
import torch.nn as nn
#
# Binary cross-entropy
logits = torch.tensor([0.8])
probas = torch.sigmoid(logits)
target = torch.tensor([1.0])
print(f" ")
print(f"logits = {logits}")
print(f"probas = {probas}")
print(f"target = {target}")
#
```

```
bce_loss_fn = nn.BCELoss()
bce_logits_loss_fn = nn.BCEWithLogitsLoss()
#
print(f" ")
print(f"BCE(probas, target) = {bce_loss_fn(probas, target):.4f}")
print(f"BCE(logits, target) = {bce_logits_loss_fn(logits, target):.4f}")
#
# Categorical cross-entropy (3 outputs)
logits = torch.tensor([[1.5, 0.8, 2.1]])
probas = torch.softmax(logits, dim = 1)
target = torch.tensor([2])
print(" ")
print(f"logits = {logits}")
print(f"probas = {probas}")
print(f"target = {target}")
#
cce_loss_fn = nn.NLLLoss()
cce_logits_loss_fn = nn.CrossEntropyLoss()
#
print(f" ")
print(f"CCE(logits, target) = {cce_logits_loss_fn(logits, target):.4f}")
print(f"CCE(log(probas), target) = {cce_loss_fn(torch.log(probas), target):.
 ↪4f}")
print(f" ")
```

```
logits = tensor([0.8000])
probas = tensor([0.6900])
target = tensor([1.])

BCE(probas, target) = 0.3711
BCE(logits, target) = 0.3711

logits = tensor([[1.5000, 0.8000, 2.1000]])
probas = tensor([[0.3013, 0.1496, 0.5490]])
target = tensor([2])

CCE(logits, target) = 0.5996
CCE(log(probas), target) = 0.5996
```

1.8 Regularising a neural network

There is as yet no way of choosing the optimal size of a network: too small and it may not be able to fit the data (*underfitting*), too large and it may perform very well on the training dataset by learning the features of individual instances of the data rather than its general properties, and therefore perform poorly on new data (*overfitting*). A typical approach is to use a network that is expected (from experience) to be a little too large, but then tackle the problem of overfitting by

introducing some form of regularisation.

1.8.1 L1 and L2 regularisation

L1 regularisation adds the sum of the absolute value of the weights (multiplied by some constant) to the loss function, while L2 regularisation adds the sum of the squares of the weights (multiplied by a constant).

1.8.2 Dropout

In recent years, dropout has emerged as a popular technique for regularizing (deep) NNs (see Dropout: A Simple Way to Prevent Neural Networks from Overfitting). Dropout is usually applied to the hidden units of networks with several layers and works as follows: during the training phase of, a fraction of the unit in a hidden layer is randomly dropped at every iteration with probability p_{drop} (or keep probability $p_{keep} = 1 - p_{drop}$). Typically $p_{drop} = 0.5$. The weights associated with the remaining neurons are rescaled to ensure the overall output levels from the layerare the same despite the dropped neurons. All neurons are used in the calculation of predictions (nothing is dropped). This is similar to training a set of models with different sets of neurons then taking the average of the models to determine the final result.

1.9 Implementing a CNN using PyTorch

Here we will look at the MNIST (Modified National Institute of Standards and Technology) dataset of images of handwritten digits from 0…9 and try and identify which digit the images represent. (This is a much-used benchmark in computer vision studies, and there are many solutions to this problem online. The one shown below is based on that in the book "Machine Learning with PyTorch and Scikit-Learn" by Raschka, Liu and Mirjalili.)

1.9.1 Loading and preprocessing the data

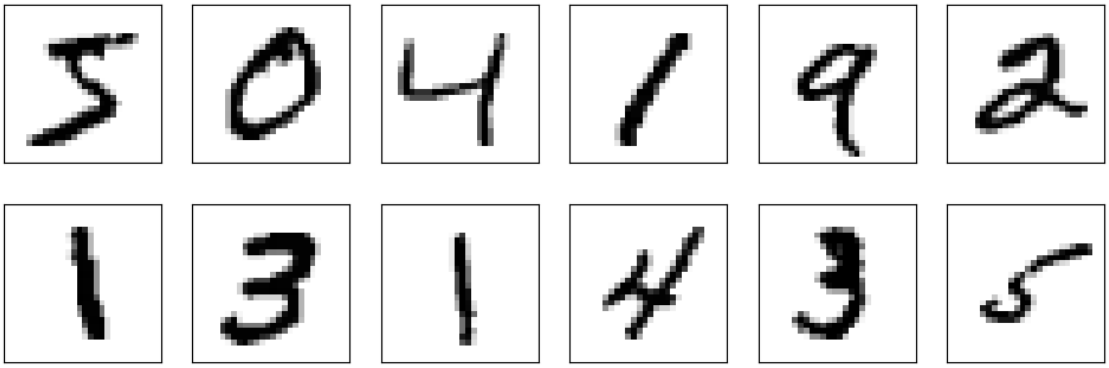
Get the training dataset.

[28]:
```
from torchvision import transforms
#
# Set the local image location
image_path = './'
#
# Set transform to convert images to tensors
transform = transforms.Compose([transforms.ToTensor()])
#
# Get the data,
mnist_dataset = torchvision.datasets.MNIST(root = image_path,      # Look for␣
 ↪data here
                                            train = True,          # Specify␣
 ↪this is training data
                                            transform = transform, # Convert to␣
 ↪tensor
                                            download = False)      # Don't get␣
 ↪from internet
```

```
                                                                # if not here
```

Here are some example images:

[29]:
```
import matplotlib.pyplot as plt
#
fig = plt.figure(figsize = (12, 4))
for i in range(12):
    ax = fig.add_subplot(2, 6, i + 1)
    ax.set_xticks([])
    ax.set_yticks([])
    img = mnist_dataset[i][0][0, :, :]
    ax.imshow(img, cmap = 'gray_r')
plt.show()
```



Split the training data into subsets for validation and training and get the test data.

[30]:
```
#
# Get routine for partitioning the data into subsets
from torch.utils.data import Subset
#
# Create a validation data subset
mnist_valid_dataset = Subset(mnist_dataset, torch.arange(10000)) # Use first␣
 ↪10000 images
#
# Create a training data subset
mnist_train_dataset = Subset(mnist_dataset, torch.arange(10000,␣
 ↪len(mnist_dataset))) # Use images
                                                                              ␣
 ↪     # from 10000
                                                                              ␣
 ↪     # to end
#
# Get the test data
```

```
mnist_test_dataset = torchvision.datasets.MNIST(root = image_path,
                                                train = False,         # Specify␣
 ↪this is test data
                                                transform = transform,
                                                download = False)      # Don't␣
 ↪get from internet
```

Use the DataLoader to create batches of training and validation data.

[31]:
```
from torch.utils.data import DataLoader
#
# Set up data for training
# Specify size of batches of features and labels that are used for each␣
 ↪training epoch
batch_size = 64
#
# Set random seed
torch.manual_seed(1)
#
if torch.cuda.is_available():
    #device = torch.device("cuda:0")
    device = torch.device("cuda")
elif torch.xpu.is_available():
    #device = torch.device("xpu:0")
    device = torch.device("xpu")
else:
    #device = torch.device("cpu:0")
    device = torch.device("cpu")
#
# Set up training and validation dataloaders, specify whether batches should be
# reshuffled for each epoch
train_dl = DataLoader(mnist_train_dataset, batch_size, shuffle = True)
valid_dl = DataLoader(mnist_valid_dataset, batch_size, shuffle = False)
```

1.9.2 Constructing the CNN

Figure 1 is a diagram of the network we want to build.

Figure 1 Network including convolution and pooling elements, flattening and fully connected layers.

In figure 1:
The inputs are 28×28 grey-scale images, i.e. there is 1 channel and the images' dimensions are $28 \times 28 \times 1$.
The image goes through a first convolutional layer with kernels of size 5×5 and padding of 2×2 which produces "images" of size 28×28. There are 32 kernels, so the output has dimensions $28 \times 28 \times 32$. There follows a max-pooling operation with size 2×2, producing dimensions of $14 \times 14 \times 32$.
The second convolutional layer uses kernels of size 5×5 with padding of 2×2 and produces "images" of size 14×14. A total of 64 kernels are used, so 64 output features are produced. The output has

dimensions $14 \times 14 \times 64$.
The second max-pooling operation has size 2×2, producing dimensions of $7 \times 7 \times 64$.
The flattened output from the second max-pooling operation, dimension $7 \times 7 \times 64 = 3136$, forms the input to a fully connected (perceptron) layer with 1024 outputs. The number of weights in this part of the net is 3136×1024.
These outputs form the inputs to a second fully connected (perceptron) layer with 10 outputs, one for each of the digits 0…9. The number of weights required here is 1024×10.

The dimensions of the tensors at each step are then:
* Input: *batchsize* $\times 28 \times 28 \times 1$.
* First convolutional layer: *batchsize* $\times 28 \times 28 \times 32$. * First max-pooling: *batchsize* $\times 14 \times 14 \times 32$.
* Second convolutional layer: *batchsize* $\times 14 \times 14 \times 64$.
* Second max-pooling: *batchsize* $\times 7 \times 7 \times 64$.
* First fully connected layer: *batchsize* $\times 3136 \times 1024$.
* Second fully connected layer: *batchsize* $\times 1024 \times 10$.

Here's one way we can "sequentially" build the convolutional and pooling parts of this model in PyTorch.

[41]:
```
import torch.nn as nn
#
model = nn.Sequential()
model.add_module('conv1', nn.Conv2d(in_channels = 1,
                                    out_channels = 32,
                                    kernel_size = 5,
                                    padding = 2))
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(kernel_size = 2))
model.add_module('conv2', nn.Conv2d(in_channels = 32,
                                    out_channels = 64,
                                    kernel_size = 5,
                                    padding = 2))
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(kernel_size = 2))
#
# Here's an example input, assuming a batchsize of 64, 1 channel and a 28 x 28␣
 ↪pixel image.
x = torch.ones((64, 1, 28, 28))
#
# After going through the model so far, the shape is
# (batch size, number of "images", "image" rows, "image" cols)
model(x).shape
```

[41]:
```
torch.Size([64, 64, 7, 7])
```

Now flatten the output and check the shape.

[43]:
```
model.add_module('flatten', nn.Flatten())
#
```

```
# After flattening, the shape is (batch size, number of nodes)
model(x).shape
```

[43]:
```
torch.Size([64, 3136])
```

The batchsize is 64 and the dimension of the input to the first fully connected layer is 3136.

[44]:
```
model.add_module('fc1', nn.Linear(3136, 1024))
model.add_module('relu3', nn.ReLU())
model(x).shape
```

[44]:
```
torch.Size([64, 1024])
```

Add the second fully connected layer and allow for dropouts between the two fully connected layers.

[45]:
```
model.add_module('dropout', nn.Dropout(p = 0.5))
model.add_module('fc2', nn.Linear(1024, 10))
model(x).shape
```

[45]:
```
torch.Size([64, 10])
```

If we have Nvidia GPUs, can use CUDA, if have Intel GPUs, can use XPU, otherwise use CPU.

[46]:
```
if torch.cuda.is_available():
    device = torch.device("cuda")
elif torch.xpu.is_available():
    device = torch.device("xpu")
else:
    device = torch.device("cpu")
#
# Write the model to the device (CPU or GPU) you are using
model = model.to(device)
```

Create a function to steer the training and validation, saving plots to illustrate progress. Note that we must write the data to the GPUs if we want to use them for calculation!

[47]:
```
#
# Choose the loss function
loss_fn = nn.CrossEntropyLoss()
#
# Choose the optimiser and set the learning rate
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
#
def train(model, num_epochs, train_dl, valid_dl):
    '''
    Train the model for num_epochs epochs using the train_dl data batches,
    validate using the valid_dl data batches.
    '''
    #
```

```
    # Creat lists for plots
    loss_plot_train = [0]*num_epochs
    accuracy_plot_train = [0]*num_epochs
    loss_plot_valid = [0]*num_epochs
    accuracy_plot_valid = [0]*num_epochs
    #
    # Do training using training data
    #
    print(f"Epoch\t Loss (training data)  Loss (validation data) ",
          f"Accuracy (training data)  Accuracy (validation data)")
    for epoch in range(num_epochs):
        model.train()
        #
        # For batches of features (images) and targets...
        for x_batch, y_batch in train_dl:
            #
            # Write data to GPU/CPU
            x_batch = x_batch.to(device)
            y_batch = y_batch.to(device)
            #
            # Calculate model predictions
            pred = model(x_batch)
            #
            # Calculate loss
            loss = loss_fn(pred, y_batch)
            #
            # Do back propagation to determine changes to weights and biasses
            loss.backward()
            #
            # Update weights and biasses
            optimizer.step()
            #
            # Set gradients to zero for next epoch
            optimizer.zero_grad()
            #
            # Add results for this epoch to training plots
            loss_plot_train[epoch] += loss.item()*y_batch.size(0)
            is_correct = (torch.argmax(pred, dim = 1) == y_batch).float()
            accuracy_plot_train[epoch] += is_correct.sum().cpu()
            #
        # Normalise training plots
        loss_plot_train[epoch] /= len(train_dl.dataset)
        accuracy_plot_train[epoch] /= len(train_dl.dataset)
        #
        # Evaluate model using validation data
        model.eval()
        #
```

```
        # Don't need to calculate and save gradients when doing validation
        with torch.no_grad():
            #
            # For features (images) and targets...
            for x_batch, y_batch in valid_dl:
                #
                # Write data to CPU/GPU
                x_batch = x_batch.to(device)
                y_batch = y_batch.to(device)
                #
                # Calculate predictions
                pred = model(x_batch)
                #
                # Calculate loss
                loss = loss_fn(pred, y_batch)
                #
                # Make validation plots
                loss_plot_valid[epoch] += loss.item()*y_batch.size(0)
                is_correct = (torch.argmax(pred, dim = 1) == y_batch).float()
                accuracy_plot_valid[epoch] += is_correct.sum().cpu()
                #
            #
        # Normalise validation plots
        loss_plot_valid[epoch] /= len(valid_dl.dataset)
        accuracy_plot_valid[epoch] /= len(valid_dl.dataset)
        #
        print(f"{epoch + 1}\t",
              f"{loss_plot_train[epoch]:.4f}\t",
              f"{loss_plot_valid[epoch]:20.4f}\t",
              f"{accuracy_plot_train[epoch]:20.4f}\t",
              f"{accuracy_plot_valid[epoch]:22.4f}")
        #
    return loss_plot_train, loss_plot_valid, accuracy_plot_train,␣
↪accuracy_plot_valid
```

Do the training.

[48]:
```
import datetime
now = datetime.datetime.now()
print("Date and time",str(now))
#
torch.manual_seed(1)
num_epochs = 20
print(" ")
loss_train, loss_valid, acc_train, acc_valid = train(model, num_epochs,␣
↪train_dl, valid_dl)
#
```

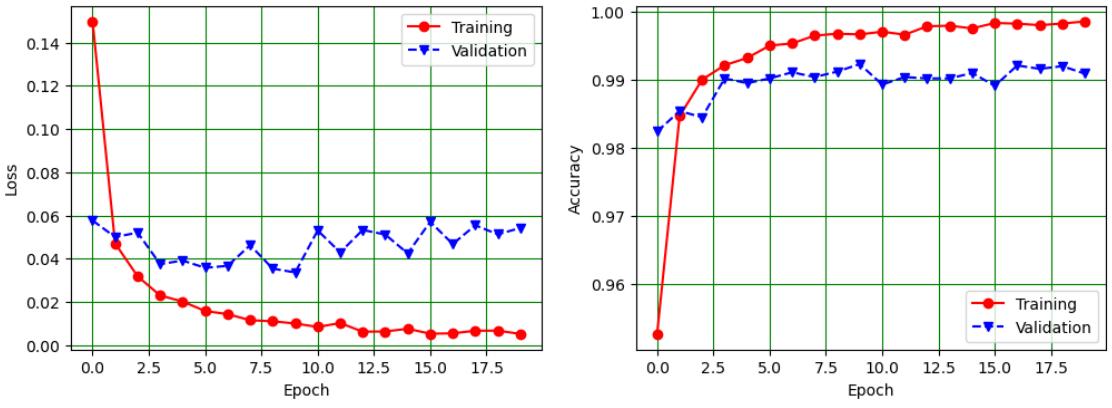
```
then = now
now = datetime.datetime.now()
print("\nDate and time",str(now))
print("Time since last check is",str(now - then))
```

```
Date and time 2026-03-03 20:25:30.635452

Epoch    Loss (training data)  Loss (validation data)  Accuracy (training data)
Accuracy (validation data)
1        0.1498                0.0577                 0.9527
0.9824
2        0.0471                0.0500                 0.9847
0.9854
3        0.0318                0.0520                 0.9900
0.9845
4        0.0229                0.0374                 0.9921
0.9902
5        0.0201                0.0391                 0.9932
0.9895
6        0.0157                0.0357                 0.9950
0.9902
7        0.0143                0.0367                 0.9953
0.9911
8        0.0115                0.0462                 0.9965
0.9904
9        0.0110                0.0354                 0.9968
0.9912
10       0.0100                0.0336                 0.9967
0.9923
11       0.0084                0.0531                 0.9971
0.9893
12       0.0102                0.0429                 0.9966
0.9904
13       0.0062                0.0533                 0.9979
0.9902
14       0.0063                0.0512                 0.9979
0.9902
15       0.0075                0.0424                 0.9975
0.9910
16       0.0052                0.0571                 0.9983
0.9891
17       0.0054                0.0468                 0.9982
0.9921
18       0.0066                0.0554                 0.9980
0.9916
19       0.0067                0.0513                 0.9983
0.9920
20       0.0051                0.0542                 0.9986
```

```
0.9909

Date and time 2026-03-03 20:30:05.965846
Time since last check is 0:04:35.330394
```

Look at plots illustrating training process.

[49]:
```
#
x_arr = np.arange(num_epochs)
#
fig = plt.figure(figsize = (12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, loss_train, marker = 'o', linestyle = '-', label = 'Training',␣
 ↪color = 'r')
ax.plot(x_arr, loss_valid, marker = 'v', linestyle = '--', label =␣
 ↪'Validation', color = 'b')
ax.legend()
ax.set_xlabel('Epoch')
ax.set_ylabel('Loss')
ax.grid(color = 'g')
#
ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, acc_train, marker = 'o', linestyle = '-', label = 'Training',␣
 ↪color = 'r')
ax.plot(x_arr, acc_valid, marker = 'v', linestyle = '--', label = 'Validation',␣
 ↪color = 'b')
ax.legend()
ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy')
ax.grid(color = 'g')
#
plt.show()
```



Calculate accuracy achieved using test data.

[50]:
```
#
# Calulate predictions
test_data = mnist_test_dataset.data.unsqueeze(1)/255.
test_data = test_data.to(device)
pred = model(test_data)
#
# Determine how many are correct
test_targets = mnist_test_dataset.targets
test_targets = test_targets.to(device)
is_correct = (torch.argmax(pred, dim = 1) == test_targets).float()
print(" ")
print(f"Test accuracy: {is_correct.mean():.4f}")
print(" ")
```

```
Test accuracy: 0.9921
```

Note that the function `unsqueeze` adds a dimension to a tensor. It's the PyTorch equivalent of Numpy's `expand_dims` function. It's used here to ensure that the input to the model (the neural net) is in the format the net expects.

[53]:
```
import numpy as np
#
x_arr = np.array([1, 2, 3])
x_arr_exp_0 = np.expand_dims(x_arr, axis = 0)
x_arr_exp_1 = np.expand_dims(x_arr, axis = 1)
print(" ")
print("x_arr: \n",x_arr)
print("x_arr.shape =",x_arr.shape)
print("x_arr[1]: \n",x_arr[1])
print(" ")
print("x_arr_exp_0: \n",x_arr_exp_0)
print("x_arr_exp_0.shape =",x_arr_exp_0.shape)
print("x_arr_exp_0[0, 1]: \n",x_arr_exp_0[0, 1])
print(" ")
print("x_arr_exp_1: \n",x_arr_exp_1)
print("x_arr_exp_1.shape =",x_arr_exp_1.shape)
print("x_arr_exp_1[1, 0]: \n",x_arr_exp_1[1, 0])
```

```
x_arr:
 [1 2 3]
x_arr.shape = (3,)
x_arr[1]:
 2
```

```
x_arr_exp_0:
 [[1 2 3]]
x_arr_exp_0.shape = (1, 3)
x_arr_exp_0[0, 1]:
 2

x_arr_exp_1:
 [[1]
 [2]
 [3]]
x_arr_exp_1.shape = (3, 1)
x_arr_exp_1[1, 0]:
 2
```

There is a corresponding function **squeeze**, which removes a dimension of size 1, in both PyTorch and Numpy.

[56]:
```
x_arr = np.array([[11],
                  [12],
                  [13]])
x_arr_sq_1 = np.squeeze(x_arr, axis = 1)
print(" ")
print("x_arr: \n",x_arr)
print("x_arr.shape",x_arr.shape)
print("x_arr_sq_1: \n",x_arr_sq_1)
print("x_arr_sq_1.shape",x_arr_sq_1.shape)
```

```
x_arr:
 [[11]
 [12]
 [13]]
x_arr.shape (3, 1)
x_arr_sq_1:
 [11 12 13]
x_arr_sq_1.shape (3,)
```

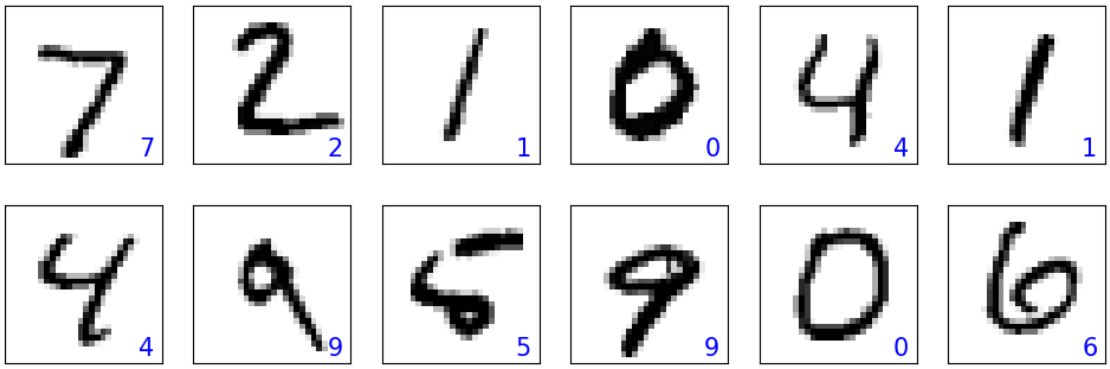
[57]:
```
x_arr = np.array([[11, 21, 31]])
x_arr_sq_0 = np.squeeze(x_arr, axis = 0)
print(" ")
print("x_arr: \n",x_arr)
print("x_arr.shape",x_arr.shape)
print("x_arr_sq_0: \n",x_arr_sq_0)
print("x_arr_sq_0.shape",x_arr_sq_0.shape)
```

```
x_arr:
 [[11 21 31]]
x_arr.shape (1, 3)
```

```
x_arr_sq_0:
 [11 21 31]
x_arr_sq_0.shape (3,)
```

Here are some images of digits with their classifications.

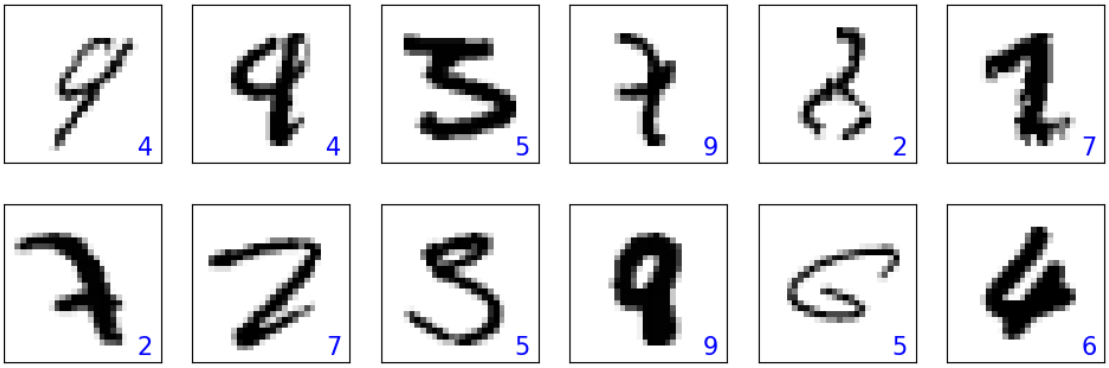
[58]:
```
fig = plt.figure(figsize=(12, 4))
#
for n in range(12):
    ax = fig.add_subplot(2, 6, n + 1)
    ax.set_xticks([])
    ax.set_yticks([])
    img = mnist_test_dataset[n][0][0, :, :]
    img_dev = img.to(device)
    pred = model(img_dev.unsqueeze(0).unsqueeze(1))
    y_pred = torch.argmax(pred)
    ax.imshow(img, cmap = 'gray_r')
    ax.text(0.9, 0.1, y_pred.item(),
            size = 15, color = 'b',
            horizontalalignment = 'center',
            verticalalignment = 'center',
            transform = ax.transAxes)
plt.show()
```



Here are some images that were mis-classified.

[59]:
```
n_plot = 0
max_plots = 12
n_try = 0
fig = plt.figure(figsize = (12, 4))
#
while n_plot < max_plots and n_try < len(mnist_test_dataset):
    img = mnist_test_dataset[n_try][0][0, :, :]
    img_dev = img.to(device)
    pred = model(img_dev.unsqueeze(0).unsqueeze(1))
```

```
    y_pred = torch.argmax(pred)
    target =  mnist_test_dataset.targets[n_try]
    is_correct = (torch.argmax(pred, dim = 1) == mnist_test_dataset.
 ↪targets[n_try])
    n_try += 1
    if is_correct:
        continue
    ax = fig.add_subplot(max_plots//6 + int(max_plots%6 > 0), 6, n_plot + 1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(img, cmap = 'gray_r')
    ax.text(0.9, 0.1, y_pred.item(),
            size = 15, color = 'b',
            horizontalalignment = 'center',
            verticalalignment = 'center',
            transform = ax.transAxes)
    n_plot += 1
plt.show()
```



1.10 Summary

We have had a quick look at how PyTorch can be installed, introduced convolution (or cross-correlation), padding, pooling, regularising (using L1 or L2 regularisation or dropout) and mentioned the loss functions relevant for one and multi-dimensional classification problems. We have pointed out the quirks of reading images in PyTorch and we have constructed a network using convolutional neural networks and a multilayer perceptron. We used this to identify handwritten digits, a job which the network performed pretty successfully!

[ ]: