Recap from previous Lab

- We explored PyTorch (PT) tensor support and drew parallelisms w.r.t. NumPy `ndarray`
- We experimented with some basic Machine Learning (ML) building blocks in PT
- We built our first, very simple Artificial Neural Network (ANN), a MultiLayer Perceptron (MLP)

Agenda for today

- Today we will be training a MLP using SGD and backpropagation
  - we will not be using the synthetic dataset we saw last lecture
  - we will be training our MLP on the MNIST dataset, which is a simple (in modern terms) dataset on which to train our ANNs. Due to this reputation, this problem is also called the *Hello World of Deep Learning*

Intro to MNIST

MNIST is a dataset for **handwritten digit recognition**.

- The dataset is composed of 60,000 grayscale images
  - by default, the dataset is already split into a training set of 50,000 images, while the remaining 10,000 images make up the test/validation set
- Each image is composed of 28x28 pixel
- Only one digit is present in each image
  - thus, we will be classifying digits from 0 to 9 (10 classes)
- The digit is centered within the image

Downloading the data

Since we're not covering the handling of data in this specific tutorial, I have prepared an external script which will download the data and "pack" it into DataLoaders. You just need to know that DataLoaders and Datasets are two different entities; namely, DataLoaders are build on top of Datasets and handle the creation of the mini-batches that will later be fed into the MLP for the training and testing phase.

The script returns the DataLoaders for both the training and the testing splits.

In [1]:
```
import torch
from scripts import mnist
from matplotlib import pyplot as plt

minibatch_size_train = 256 #usually: 32, 64, 128,256
minibatch_size_test = 512 #usually 512, 1024

trainloader, testloader = mnist.get_data(batch_size_train=minibatch_size_test, batch_size_test=minibatch_size_test)
```

```
0.1%
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to datasets/MNIST/raw/train-images-idx3-uby
te.gz
100.0%
Extracting datasets/MNIST/raw/train-images-idx3-ubyte.gz to datasets/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to datasets/MNIST/raw/train-labels-idx1-uby
te.gz
102.8%
0.3%
Extracting datasets/MNIST/raw/train-labels-idx1-ubyte.gz to datasets/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to datasets/MNIST/raw/t10k-images-idx3-ubyt
e.gz
100.0%
Extracting datasets/MNIST/raw/t10k-images-idx3-ubyte.gz to datasets/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to datasets/MNIST/raw/t10k-labels-idx1-ubyt
e.gz
112.7%
Extracting datasets/MNIST/raw/t10k-labels-idx1-ubyte.gz to datasets/MNIST/raw
```

Let us quickly view our data.

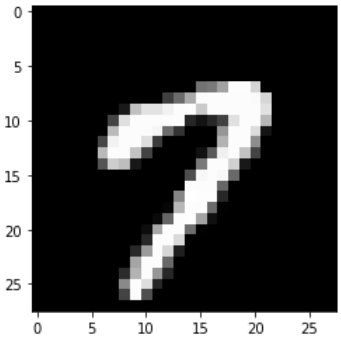
The dataset is an attribute of the dataloader. From here, we can access all the images of our dataset (at least for what concerns small datasets like MNIST; larger datasets are treated a little differently, we'll see more on that in the future labs).

The images are stored within the `data` attribute, while the labels lay in the `targets` attribute.

In [2]:
```
plt.imshow(trainloader.dataset.data[15].numpy(), cmap="gray") #plt.imshow requires numpy form

print("The label is", trainloader.dataset.targets[15].item()) #dataset.targets are the labels of the images
```

```
The label is 7
```



By applying the `Tensor` skills we learned during the previous lab, we may also plot multiple images within the same plot.

In [3]:
```
multi_img = trainloader.dataset.data[25:30] #from 25 to 29
multi_img.shape
```

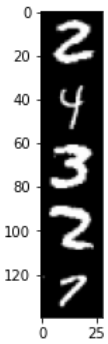
Out[3]:
```
torch.Size([5, 28, 28])
```

We have a (5 x 28 x 28) `Tensor`, we can't plot it with `plt.imshow` because, for grayscale images, it needs a matrix as input.

Q: how can we solve this problem? Think about the methods we saw during the last lab

In [4]:
```
# solve here: metti insime le 5 img in una sola
# shape 5*28 x 28
multi_img_reshaped = multi_img.reshape(5*28, 28)
plt.imshow(multi_img_reshaped.numpy(), cmap="gray")
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x7f6fcb1f28e0>
```



Designing our MLP

Let us design our MLP.

We need to think about:

1. How wide is the output layer? 10 digits
2. How wide is the input layer and how is it structured?
3. How many hidden layers and how wide need they be?
4. What activation functions will we be using?

You may try answering questions 1 and 2 by yourself as there's only one specific answer. Q. 3 and 4, instead, are a choice operated by the designer (the data scientist) and there is no straight definite answer, as usually architectural features are treated as hyperparameters to be tuned as part of the training process.

For our MLP, then, we will be designing an MLP with

- 3 hidden layers
  - the first hidden layer has 16 neurons
  - the second hidden layer has 32 neurons
  - the third hidden layer has 24 neurons
- ReLU as activation function for the hidden layers ($ReLU(x) = \max(0, x)$)
- Cross Entropy loss (or, equivalently, Negative LogLikelihood loss)**

** *Notice that, if you use NLLLoss, PyTorch wants as output of the model the **log** softmax (i.e., the **log probabilities**) of assignment to the classes, not the regular probabilities. This is for stability issues.*

In [8]:
```
# write here the class for your MLP
class MLP(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.flat = torch.nn.Flatten() # X comes in as a n x 1 x 28 x 28 -> we need n 784-size vectors (or, a n x 7
        self.layers = torch.nn.Sequential(
            torch.nn.Linear(784, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 32),
            torch.nn.ReLU(),
            torch.nn.Linear(32, 24),
            torch.nn.ReLU(),
            torch.nn.Linear(24, 10),
            #torch.nn.LogSoftMax(), # we will use cross entropy, so don't use it! <- this only if I plan on using N
        )# your code here

    def forward(self, X):
        # your code here
        out = self.flat(X)
        out = self.layers(out)
        return out
```

The training loop

We now have to train the network using mini-batch Stochastic Gradient Descent (which we'll indicate as SGD).

The mini-batch part is already handled by the `DataLoader`.

We just need to call

```
for X, y in dataloader:
    # do something
```

`X` and `y` represent our mini-batch (images and ground truth respectively).

- What do we need to do inside this loop?
  1. do the forward pass
  2. calculate the loss
  3. do the backward pass and update the parameters
- What objects do we need inside this loop?
  - the model (to do the forward pass)
  - the dataloader (to get `X, y`)
  - the loss function (to calculate the value of the loss and backpropagate)
  - the optimizer (to upgrade the value of the parameters given the gradients obtained via backprop)
- What do we need to do outside this loop?
  1. re-shuffle the mini-batches (NB: already taken care of by the `DataLoader`)
  2. repeat the loop for each epoch

Next, we are going to implement this loop (which we'll call `train_epoch`) in a minimalistic setting, i.e., with no excessive feature or embellishment.

In [9]:
```
def train_epoch(model, dataloader, loss_fn, optimizer):
    for X, y in dataloader:
        # 1. reset the gradients previously accumulated by the optimizer
        #    this will avoid re-using gradients from previous loops
        optimizer.zero_grad() # or model.zero_grad()
        # It tells the optimizer to cancel/reset all the gradients: to prevent gradient accumulation
        # Infact PT allows for maximal flexibility.

        # 2. get the predictions from the current state of the model
        #    this is the forward pass
        y_hat = model(X)
        # 3. calculate the loss on the current mini-batch
        loss = loss_fn(y_hat, y)
        # 4. execute the backward pass given the current loss
        loss.backward()
        # 5. update the value of the params
        optimizer.step()
```

Question (from last year's students): there's also a `model.zero_grad()` functionality. Is it the same as `optimizer.zero_grad()`?

Quick answer: usually yes. I have always seen `optimizer.zero_grad()` but some people prefer the former. See here for discussion.

To work, we just need to wrap this routine inside another loop which will repeat `train_epoch` for each epoch.

In [10]:
```
def train_model(model, dataloader, loss_fn, optimizer, num_epochs):
    # this is a useful switch that lets us pass to training phase to network evaluation
    # we will see in a future lab why it is necessary - for now it does nothing in particular
    model.train()
```

```
    for epoch in range(num_epochs):
        train_epoch(model, dataloader, loss_fn, optimizer)
        print(f"Epoch {epoch+1} completed.") # this is just so we have an idea on where we are during the training
```

We will use vanilla SGD with a learning rate of 0.001 and we'll run the training for 3 epochs. Moreover, we will use the Negative Log-Likelihood (NLL) loss function.

In [11]:
```
learn_rate = 0.1 # for SGD
num_epochs = 3
```

In [12]:
```
model = MLP()
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate)
```

Training on GPU

Training on GPU requires three variables to reside on the same GPU:

1. the data
2. the labels
3. the parameters

they all can be achieved by calling the `.to(<device>)` method, which exists for both tensors and modules. In the case for tensors, it does not operate in-place, but in the case of modules, the function moves all of the parameters without needing assignment.

In [14]:
```
device = "cuda:0" if torch.cuda.is_available() else "cpu"

print(f"We will use {device}")
```

```
We will use cuda:0
```

In [15]:
```
def train_epoch(model, dataloader, loss_fn, optimizer, device):
    for X, y in dataloader:
        ##### MOVE DATA AND LABELS TO THE DESIRED DEVICE #####
        X = X.to(device) # gpu <-> cpu
        y = y.to(device) # gpu <-> cpu

        # OSS: the param of the modle are still in cpu!
        # PT wants data and param in the same device
        # OSS: to applied to module: operates in place x.to(device)
        #                 to tensor: returns a copy

        # 1. reset the gradients previously accumulated by the optimizer
        #    this will avoid re-using gradients from previous loops
        optimizer.zero_grad()
        # 2. get the predictions from the current state of the model
        #    this is the forward pass
        y_hat = model(X)
        # 3. calculate the loss on the current mini-batch
        loss = loss_fn(y_hat, y)
        # 4. execute the backward pass given the current loss
        loss.backward()
        # 5. update the value of the params
        optimizer.step()
```

In [18]:
```
def train_model(model, dataloader, loss_fn, optimizer, num_epochs, device):
    # this is a useful switch that lets us pass to training phase to network evaluation
    # we will see in a future lab why it is necessary - for now it does nothing in particular
    model.train() #in our case we don't have a module/model that depends on param that depend on dataset (we have n
    model.to(device)
    for epoch in range(num_epochs):
        train_epoch(model, dataloader, loss_fn, optimizer, device)
        print(f"Epoch {epoch+1} completed.") # this is just so we have an idea on where we are during the training
```

In [20]:
```
train_model(model, trainloader, loss_fn, optimizer, num_epochs, device)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb Cell 25' in <cell line: 1>()
----> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch00000
24?line=0'>1</a> train_model(model, trainloader, loss_fn, optimizer, num_epochs, device)

/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb Cell 29' in train_model(model, dataloader,
 loss_fn, optimizer, num_epochs, device)
     <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch000002
8?line=23'>24</a> for epoch in range(num_epochs):
     <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch000002
8?line=24'>25</a>     loss_meter = AverageMeter()
---> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch000002
8?line=25'>26</a>     train_epoch(model, dataloader, loss_fn, optimizer, device, loss_meter)
     <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch000002
8?line=26'>27</a>     print(f"Epoch {epoch+1} completed. Training loss: {loss_meter.avg}")

TypeError: train_epoch() takes 5 positional arguments but 6 were given
```

The problem with this setting is that we have no idea how the network is faring, either on the training or the test set.

We need to devise some ways to assess the model.

Three ideas:

1. Accumulate the loss for each epoch
   - Alternatively, use the average loss per instance
2. Show the mean accuracy for each epoch
3. 1 and 2, but calculated on the test set

1 is easily implementable, just introduce a variable to accumulate the loss during the epoch and print it at the end of the training.

The average loss becomes a bit harder to implement. Without reinventing the wheel, what is usually done is to make use of an auxiliary structure, which is called `AverageMeter`, to keep track of the running metrics and quickly summarize them. Let' see how it's implemented:

In [20]:
```
class AverageMeter(object):
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0 # val holds the current stat
        self.avg = 0 # avg holds the cumulative average
        self.sum = 0 # sum holds the cumulative value
        self.count = 0 # count holds the number of instances seen

    def update(self, val, n=1):
        self.val = val
        self.count += n
        self.sum += val * n
        self.avg = self.sum / self.count
```

at the start of each epoch, we are going to re-instantiate one `AverageMeter` for keeping track of the loss.

In [21]:
```
def train_epoch(model, dataloader, loss_fn, optimizer, device, loss_meter):
    for X, y in dataloader:
        ##### MOVE DATA AND LABELS TO THE DESIRED DEVICE #####
        X = X.to(device)
        y = y.to(device)
        # 1. reset the gradients previously accumulated by the optimizer
        #    this will avoid re-using gradients from previous loops
        optimizer.zero_grad()
        # 2. get the predictions from the current state of the model
        #    this is the forward pass
        y_hat = model(X)
        # 3. calculate the loss on the current mini-batch
        loss = loss_fn(y_hat, y)
        # 4. execute the backward pass given the current loss
        loss.backward()
        # 5. update the value of the params
        optimizer.step()
        # 6. update the loss meter: passing current value of the loss and number of instances that the network has
        loss_meter.update(val=loss.item(), n=X.shape[0]) # or n=y.shape[0] since it's the same

def train_model(model, dataloader, loss_fn, optimizer, num_epochs, device):
    model.train()
    model.to(device)
    for epoch in range(num_epochs):
        loss_meter = AverageMeter()
        train_epoch(model, dataloader, loss_fn, optimizer, device, loss_meter)
        print(f"Epoch {epoch+1} completed. Training loss: {loss_meter.avg}")
```

Assessing accuracy

Usually, $\text{loss} \in [0; +\infty)$, so it is hard to assess the effectiveness of the ANN based solely on such value.

Alongside a loss, we can assess the performance via a much more interpretable index as the **accuracy**

$$\text{accuracy} = \frac{\#\text{ correctly identified items}}{\#\text{ all items}}$$

To keep track of the accuracy, we can re-use the AverageMeter defined above, although we need to define a function for it---since PT does not implement it by default.

In [27]:
```
def accuracy(y_hat, y):
    '''
    y_hat is the model output - a Tensor of shape (n x num_classes)
    y is the ground truth

    How can we implement this function?
    '''
    classes_prediction = y_hat.argmax(dim=1)
    match_ground_truth = classes_prediction == y # -> tensor of booleans
    correct_matches = match_ground_truth.sum()
    return (correct_matches / y_hat.shape[0]).item()
```

In [22]:
```
def train_epoch(model, dataloader, loss_fn, optimizer, device, loss_meter, accuracy_meter):
    for X, y in dataloader:
        ##### MOVE DATA AND LABELS TO THE DESIRED DEVICE #####
        X = X.to(device)
        y = y.to(device)
        # 1. reset the gradients previously accumulated by the optimizer
        #    this will avoid re-using gradients from previous loops
        optimizer.zero_grad()
        # 2. get the predictions from the current state of the model
        #    this is the forward pass
        y_hat = model(X)
        # 3. calculate the loss on the current mini-batch
        loss = loss_fn(y_hat, y)
        # 4. execute the backward pass given the current loss
        loss.backward()
        # 5. update the value of the params
        optimizer.step()
        # 6. calculate the accuracy for this mini-batch
        acc = accuracy(y_hat, y)
        # 7. update the loss and accuracy AverageMeter
        loss_meter.update(val=loss.item(), n=X.shape[0])
        accuracy_meter.update(val=acc, n=X.shape[0])

def train_model(model, dataloader, loss_fn, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        loss_meter = AverageMeter()
        accuracy_meter = AverageMeter()
        train_epoch(model, dataloader, loss_fn, optimizer, loss_meter, accuracy_meter)
        # now with loss meter we can print both the cumulative value and the average value
        print(f"Epoch {epoch+1} completed. Loss - total: {loss_meter.sum} - average: {loss_meter.avg}; Accuracy: {a
    # we also return the stats for the final epoch of training
    return loss_meter.sum, accuracy_meter.avg
```

Additional observation: we might wanna pass the `accuracy` as an additional `performance` parameter (analogous to `loss_fn`) since we might wanna evaluate the performance on a metric different than accuracy.

Now, we can train the network and examine its performance as we're running the training

In [25]:
```
# reset the network and optimizer
model = MLP()
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate)

loss, acc = train_model(model, trainloader, loss_fn, optimizer, device, num_epochs)
print(f"Training completed - final accuracy {acc} and loss {loss}")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb Cell 34' in <cell line: 5>()
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch00000
33?line=1'>2</a> model = MLP()
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch00000
33?line=2'>3</a> optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate)
----> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch00000
33?line=4'>5</a> loss, acc = train_model(model, trainloader, loss_fn, optimizer,device, num_epochs)
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/03-sgd-training.ipynb#ch00000
33?line=5'>6</a> print(f"Training completed - final accuracy {acc} and loss {loss}")

TypeError: train_model() takes 5 positional arguments but 6 were given
```

Recall that we wanted to assess also the network's performance on the test set. We need an additional testing routine for it.

For testing, we need only

1. the network
2. the test-set `DataLoader`
3. the loss function / the performance metrics

we don't need the optimizer as we're not updating the weights now

In [23]:
```
def test_model(model, dataloader, device, loss_fn=None,): # None since I decided to use the accuracy as a performan
    if loss_fn is not None:
        loss_meter = AverageMeter()
    acc_meter = AverageMeter()
    model.eval() # this is the contrary of model.train() we saw before - equivalent to model.train(False)
    model.to(device)
    with torch.no_grad(): # in this computation you don't need to calculate the gradient (save_for_backwward): so i
        for X, y in dataloader:
            X = X.to(device)
            y = y.to(device)
            y_hat = model(X)
            loss = loss_fn(y_hat, y) if loss_fn is not None else None
            acc = accuracy(y_hat, y)
            if loss_fn is not None:
                loss_meter.update(loss.item(), X.shape[0])
            acc_meter.update(acc, X.shape[0])
    print(f"TESTING - loss {loss_meter.sum if loss_fn is not None else '--'} - accuracy {acc_meter.avg}")
```

In [28]:
```
test_model(model, testloader, device)
```

```
TESTING - loss -- - accuracy 0.9212833333969116
```

Now that we have nice-enough-looking training and testing routines and that we have trained and tested our model we might want to save it so we can re-use it in the future.

The model can be easily saved using the `state_dict()` method

We can save it using PT built-in `torch.save`

In [ ]:
```
import os
folder_save = "models/mlp_mnist"
os.makedirs(folder_save, exist_ok=True)
filename = os.path.join(folder_save, "model.pt")
torch.save(model.state_dict(), filename)
```

Disclaimer: **NEVER** save the model with `torch.save(model)`. See reason here and here.

Let's suppose we wish to reload the state_dict. We just need one line of code:

In [ ]:
```
model.load_state_dict(torch.load(filename))
```

Saving a model mid-training

Let's suppose we further need to train our model. Often times, storing only the weights may not be enough.

Till now, we have only seen examples in which the training hyperparameters are static. Modern techniques, though, require dynamic hyperparameters to ensure good performance or generalization. You may have heard of optimizers such as ADAM or techniques such as learning rate annealing. Thus, in order to restore the training, we need to save a **checkpoint** so that we're able to re-start training at the exact same conditions we were in when it first ended^.

For now, a checkpoint will be composed of the following objects:

1. The `state_dict` of the model
2. The state of the optimizer (which is also obtainable via a `state_dict` method---see code)
3. The epochs trained, so that we can select the epoch to restart with
4. (optionally) the iterations within the epoch, if we interrupted training mid-epoch (can be useful for very large models)
5. If you're using a learning rate schedule (we'll see in a few labs what this is), also its state dict.

^ *Note that, due to the intrinsic stochasticity of SGD, we may not actually be able to reproduce the exact same situation as before.*

In [ ]:
```
checkpoint_dict = {
    "parameters": model.state_dict(),
    "optimizer": optimizer.state_dict(),
    "epoch": ...,
    "iteration": ...,
    "lr_scheduler": ...
}
```

In [ ]:
```
# save
filename = os.path.join(folder_save, "checkpoint.pt")
torch.save(checkpoint_dict, filename)
```

In [ ]:
```
# restore checkpoint
checkpoint = torch.load(filename)
model.load_state_dict(checkpoint["parameters"])
optimizer.load_state_dict(checkpoint["optimizer"])
lr_scheduler.load_state_dict(checkpoint["lr_scheduler"])
```

In a setting where you wish to restart training, you may modify the `train_model` function by adding the argument `epoch_restart` and looping through epochs like this

```
for epoch in range(epoch_restart, num_epochs):
    # train loop
```

if training broken during a random iteration:

```
for X, y in trainloader:
    # what do I write here?
```

Training a model on custom loss functions: an extension of the Mean Square Error for classification

We might want to fit our MLP using custom losses. Despite being the *de facto* choice for multiclass ML problems, Cross Entropy is not the single loss we can use.

A very simple loss we may consider is the Quadratic Loss (QL) 1:

$$QL(\hat{y}, y) = \frac{1}{2}\mathrm{E}[||y^{(\text{one-hot})} - \hat{y}||^2] = \frac{1}{2n}\sum_{i=1}^{n}(||y_i^{(\text{one-hot})} - \hat{y}_i||)^2 = \frac{1}{2n}\sum_{i=1}^{n}\sum_{j=1}^{d}(y_{i,j}^{(\text{one-hot})} - \hat{y}_{i,j})^2$$

Where:

- the one-hot encoding for $y = c$, where c is a possible class out of C classes, is the column vector whose elements are all 0 beside the c-th entry, which is 1;
- with $\hat{y}$ we mean the output of the model, i.e. a vector representing a probability distribution that a data point be assigned to each class.

For instance, given a data point whose ANN output is $\hat{y} = [0.3, 0.2, 0.4, 0.1]^\top$ and the ground truth is $y = 3$, the QL for this point is

$$0.5 \cdot ||[0, 0, 1, 0]^\top - [0.3, 0.2, 0.4, 0.1]^\top||^2 = 0.5 \cdot ||[-0.3, -0.2, 0.6, -0.1]^\top||^2 = 0.5 \cdot (0.09 + 0.04 + 0.36 + 0.01) = 0.5 \cdot 0.5 = 0.25$$

Let us implement this loss function in PT

OSS: it doesn't work that good as cross entropy

In [33]:
```
y = torch.Tensor([0,1,2,3,0,0]).long()# .long(): otherwise it gets an error
y_onehot = torch.nn.functional.one_hot(y, num_classes=5)
print(y_onehot)
print("Shape", y_onehot.shape)
```

```
tensor([[1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0],
        [1, 0, 0, 0, 0],
        [1, 0, 0, 0, 0]])
Shape torch.Size([6, 5])
```

In [ ]:
```
def quadratic_loss(y_hat, y):
    '''
    y_hat is a matrix of dimension (n x C),
    where C is the number of classes, and n is the number of datapoints
    y is a vector of classes (shape (C))
    '''
    # convert y to onehot
    # y_onehot = torch.nn.functional.one_hot(y.long())
    # y_hat must be a vector of probabilities:
    # y_hat = torch.nn.softmax(y_hat) ##HERE WE MUST DO SOMETHING WITH OUR DATA DEPENDING ON THE FINAL ACTIVATION F
    # norm = (y_onehot - y_hat.exp()).norm(dim=1)**2
    y_onehot = torch.nn.functional.one_hot(y.long())
    # If loss is CrossEntropyLoss, apply softmax to render y_hat a simplex
    y_hat = torch.nn.softmax(y_hat)
    # If loss is NLLLoss, exponentiate y_hat (because PyTorch wants LogSoftmax as final activation)
    # y_hat = y_hat.exp()
    norm = (y_onehot - y_hat).norm(dim=1)**2
    norm_sum = norm.sum()
    return norm_sum / (2 * y_hat.shape[0])
```

Let's be more PyTorch-ian:

In [ ]:
```
class QLoss(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, y_hat, y):
        y_onehot = torch.nn.functional.one_hot(y.long())
        norm_square = (y_onehot - y_hat.exp()).norm(dim=1)**2
        norm_sum = norm_square.sum()
        return norm_sum / (2*y_hat.shape[0])
```

In [ ]:
```
learn_rate = 0.1
num_epochs = 3

model = MLP()
loss_fn = quadratic_loss
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate)
```

In [ ]:
```
train_model(model, trainloader, loss_fn, optimizer, num_epochs)
```

Custom activation functions

Let us suppose we wish to add a custom activation function to our network, for example, the sine function.

We'll quickly discover that we can't just add a functional representation of it to our `Sequential`. Let's try it

In [29]:
```
class SineNN(torch.nn.Module): #used in image compression and reconstruction
    def __init__(self):
        super().__init__()
        self.structure = torch.nn.Sequential(
            torch.nn.Linear(2, 2),
            #torch.sin # this won't work! I need to put a module: the class for the activation function as a MODULE
            #: torch.sin is a function
```

```
            torch.sin # this won't work! <- must be a module!
        )

    def forward(self, X):
        return self.structure(X)

nn = SineNN()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_27776/898093294.py in <module>
     10         return self.structure(X)
     11
---> 12 nn = SineNN()

/tmp/ipykernel_27776/898093294.py in __init__(self)
      2     def __init__(self):
      3         super().__init__()
----> 4         self.structure = torch.nn.Sequential(
      5             torch.nn.Linear(2, 2),
      6             torch.sin # this won't work! <- must be a module!

~/miniconda3/envs/lot/lib/python3.8/site-packages/torch/nn/modules/container.py in __init__(self, *args)
     89         else:
     90             for idx, module in enumerate(args):
---> 91                 self.add_module(str(idx), module)
     92
     93     def _get_item_by_idx(self, iterator, idx) -> T:

~/miniconda3/envs/lot/lib/python3.8/site-packages/torch/nn/modules/module.py in add_module(self, name, module)
    375         """
    376         if not isinstance(module, Module) and module is not None:
--> 377             raise TypeError("{} is not a Module subclass".format(
    378                 torch.typename(module)))
    379         elif not isinstance(name, torch._six.string_classes):

TypeError: _VariableFunctionsClass.sin is not a Module subclass
```

In fact, PyTorch is complaining that the sin module is not inheriting from `torch.nn.Module`, which it needs to do. So, we only need to wrap `torch.sin` in a class like so.

In [31]:
```
class Sin(torch.nn.Module):
    # it needs to have at least a constructor and a forward method, just like an MLP
    # I could not wrtite the constructor:
    def __init__(self):
        super().__init__()
        # no additional stuff to do here as Sin has no additional parameters to set

    def forward(self, X):
        return torch.sin(X)
```

In [32]:
```
# it will not produce an exeption now
class SineNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.structure = torch.nn.Sequential(
            torch.nn.Linear(2, 2),
            Sin()
        )

    def forward(self, X):
        return self.structure(X) # will call forward on the linear and then on the sin

nn = SineNN()
```

Homework *(not compulsory)*

- Reconstruct in PyTorch the first experiment in Learning representations by back-propagating errors with learning rule in eq.8 (gradient descent without momentum---you can re-use `torch.optim.SGD` with the appropriate keyword)
  - Try to be as close as possible to the original protocol, except for what regards the learning rule
  - Read the paper, if you did not do it yet (don't worry if you don't understand the other experiments in detail)
  - Create the data, the model and everything is needed (do not use dataloaders if you don't know how yet how they work)
  - Train the model
  - Inspect the weights you obtained and check if they provide a solution to the problem
  - Compare the solution to the solution reported in the paper Tip: don't expect to get a fully working implementation!

Additional tips for implementing ANNs in PyTorch (will not be part of the exam): PyTorch common mistakes, by Aladdin Person

References

1 A. Demirkaya, J. Chen and S. Oymak, "Exploring the Role of Loss Functions in Multiclass Classification," 2020 54th Annual Conference on Information Sciences and Systems (CISS), Princeton, NJ, USA, 2020, pp. 1-5, doi: 10.1109/CISS48834.2020.1570627167.