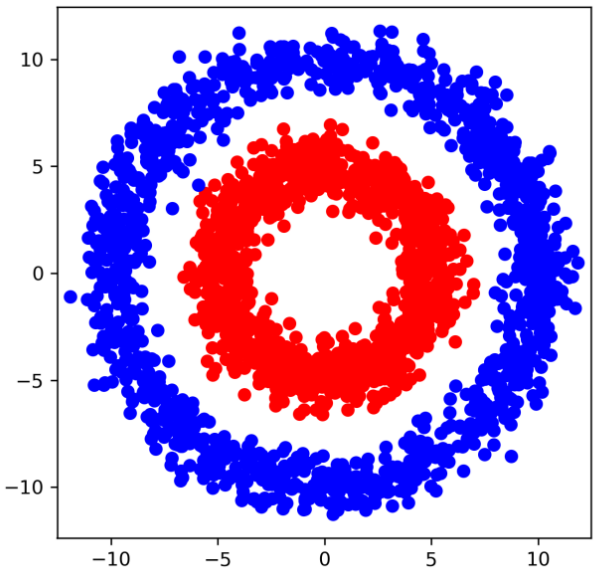
Assignment 1

Computer Vision ETH, Fall 2021
Quentin Guignard
20th October, 2021

2D Classifier

Figure 1: training data for the 2D classifier



2.1 Dataset *Start by filling in the Simple2DDataset class from lib/dataset.py. First, in the class constructor init , you will need to read the right npz file from disk based on the split parameter and then save the contents to the associated class members. Second, in the getitem method, you will need to recover a single data sample and its annotation based on its index idx.*

Listing 1: Implementation of the Simple2DDataSet class in dataset.py

```
"""function that reads the correct data set"""
def read_npz(split):
    data_directory_path = os.path.join("data")
    train_path = data_directory_path + "/train.npz"
    valid_path = data_directory_path + "/valid.npz"
```

```
    assert os.path.exists(train_path), "train.npz must exist in the data directory"
    assert os.path.exists(valid_path), "valid.npz must exist in the data directory"

    path = train_path if (split == 'train') else valid_path

    data_npz = np.load(path)

    samples = data_npz['samples']
    annotations = data_npz['annotations']

    data_npz.close()

    return (samples, annotations)

class Simple2DDataset(Dataset):
    def __init__(self, split='train'):
        super().__init__()
        assert split in ['train', 'valid'], f'Split parameters "{split}" must be either "train" or "valid".'
        # Read either train or validation data from disk based on split parameter using np.load.
        # Data is located in the folder "data".
        # Hint: you can use os.path.join to obtain a path in a subfolder.
        # Save samples and annotations to class members self.samples and self.annotations respectively.
        # Samples should be an Nx2 numpy array. Annotations should be Nx1.
        self.samples, self.annotations = read_npz(split)
        #raise NotImplementedError()

    def __len__(self):
        # Returns the number of samples in the dataset.
        return self.samples.shape[0]

    def __getitem__(self, idx):
        # Returns the sample and annotation with index idx.
        #raise NotImplementedError()
        sample = self.samples[idx]
        annotation = self.annotations[idx]

        # Convert to tensor.
        return {
            'input': torch.from_numpy(sample).float(),
            'annotation': torch.from_numpy(annotation[np.newaxis]).float()
        }
```

2.2 Linear classifier *Next step is defining your first network by filling in the LinearClassifier class from lib/networks.py. Add a single linear layer nn.Linear inside the nn.Sequential call. The output should be a single value, corresponding to the probability of a given 2D point being part of cluster 1.*

Listing 2: Implementation of a single Linear classifier Note: dim_input can be either 2 or 3

```
class LinearClassifier(nn.Module):
    def __init__(self, dim_input):
        super().__init__()

        self.codename = 'linear'

        # Define the network layers in order.
        # Input is 2D.
```

```
        # Output is a single value.
        # Single linear layer.

        #raise NotImplementedError()
        # TODO
        self.layers = nn.Sequential(
            nn.Linear(dim_input, 1)
        )

    def forward(self, batch):
        # Process batch using the layers.
        x = self.layers(batch)
        # Final sigmoid activation to obtain a probability.
        return torch.sigmoid(x)
```

2.3 Training loop *Finally, finish implementing the training loop in the run training epoch function from train.py. There are four important steps during each training step and respecting the order is essential: 1. Clear the gradients from the previous step using optimizer.zero grad() 2. Forward pass of the network to obtain the predictions 3. Compute the loss (binary cross entropy) 4. Backwards pass on the loss using loss.backward() to obtain the gradients followed by one step of gradient descent (optimization) using using optimizer.step() Once this is done, you should be able to run the train.py script. What accuracy do you achieve with the linear classifier? Is this an expected result? Justify your answer.*

Listing 3: Implementation of the training loop

```
def run_training_epoch(net, optimizer, dataloader):
    loss_aggregator = UpdatingMean()
    # Put the network in training mode.
    net.train()
    # Loop over batches.
    for batch in dataloader:
        #raise NotImplementedError()

        # Reset gradients.
        # TODO
        optimizer.zero_grad()

        # Forward pass.
        output = net.forward(batch['input'])

        # Compute the loss - binary cross entropy.
        # Documentation https://pytorch.org/docs/stable/generated/torch.nn.functional.binary_cross_entropy.html.
        loss = F.binary_cross_entropy(output, batch['annotation'])

        # Backwards pass.
        # TODO
        loss.backward()
        optimizer.step()

        # Save loss value in the aggregator.
        loss_aggregator.add(loss.item())
    return loss_aggregator.mean()
```

Here are the results we achieved with the Linear classifier.

```
[Epoch 01] Loss: 1.0745
[Epoch 01] Acc.: 48.0159%
[Epoch 02] Loss: 0.7637
[Epoch 02] Acc.: 48.4127%
[Epoch 03] Loss: 0.6979
[Epoch 03] Acc.: 49.0079%
[Epoch 04] Loss: 0.6932
[Epoch 04] Acc.: 49.0079%
[Epoch 05] Loss: 0.6933
[Epoch 05] Acc.: 48.2143%
[Epoch 06] Loss: 0.6931
[Epoch 06] Acc.: 48.4127%
[Epoch 07] Loss: 0.6930
[Epoch 07] Acc.: 48.2143%
[Epoch 08] Loss: 0.6930
[Epoch 08] Acc.: 48.6111%
[Epoch 09] Loss: 0.6931
[Epoch 09] Acc.: 49.0079%
[Epoch 10] Loss: 0.6932
[Epoch 10] Acc.: 48.6111%
```

The accuracy is of 48.6 percents. The classifier is about as good as chance and this is to be expected. The data is not linearly separable. As we can see on the plot, both classes are defined by two nested circles. Therefore, we cannot imagine trying to fit a line in the 2D plane that splits the data. Indeed, given a new point, we cannot predict its class accurately as in a best case scenario half of the points of same class would still lie on the other boundary side. The functions that would be needed to be approximated in order to split the data would a circle equation which is non linear.

2.4 Multi-layer perceptron *Now let us try a multi-layer classifier with non-linearities by filling in the MLPClassifier class from lib/networks.py. Implement an MLP with 2 hidden layer and one final linear prediction layer. The hidden layers should have 16 neurons and be followed by a nn.ReLU non-linearity. Switch to the new network by uncommenting L83 in train.py. What accuracy does this network obtain? Why are the results better compared to the previous classifier?*

Listing 4: Implementation of a MLP with a 2D input, 1 hidden layer of 16 neurons and 1D output

```
class MLPClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.codename = 'mlp'

        # Define the network layers in order.
        # Input is 2D.
        # Output is a single value.
        # Multiple linear layers each followed by a ReLU non-linearity (apart from the last).
        #raise NotImplementedError()
        # TODO
```

```
        self.layers = nn.Sequential(
            nn.Linear(2, 16), #input layer
            nn.ReLU(),
            nn.Linear(16, 16), #1st hidden layer
            nn.ReLU(),
            nn.Linear(16, 1) #output layer
        )

    def forward(self, batch):
        # Process batch using the layers.
        x = self.layers(batch)
        # Final sigmoid activation to obtain a probability.
        return torch.sigmoid(x)
```

Here are the results we achieved with the MLP.

```
[Epoch 01] Loss: 0.6052
[Epoch 01] Acc.: 77.7778%
[Epoch 02] Loss: 0.3481
[Epoch 02] Acc.: 99.0079%
[Epoch 03] Loss: 0.1183
[Epoch 03] Acc.: 100.0000%
[Epoch 04] Loss: 0.0474
[Epoch 04] Acc.: 100.0000%
[Epoch 05] Loss: 0.0270
[Epoch 05] Acc.: 100.0000%
[Epoch 06] Loss: 0.0180
[Epoch 06] Acc.: 99.8016%
[Epoch 07] Loss: 0.0128
[Epoch 07] Acc.: 99.8016%
[Epoch 08] Loss: 0.0095
[Epoch 08] Acc.: 100.0000%
[Epoch 09] Loss: 0.0079
[Epoch 09] Acc.: 100.0000%
[Epoch 10] Loss: 0.0065
[Epoch 10] Acc.: 99.8016%
```

The accuracy is of 99.8 percents. It is much better as we now can also approximate non-linear functions because of the ReLU layers. We also have more parameters for our model which helps in this case increasing the accuracy.

2.5 Feature transform *One common way to improve the performance of a classifier is feature engineering. The easiest type of feature engineering is a coordinate system change. Think of a coordinate system that renders the two classes linearly separable and justify your choice.*

Start by filling in the Simple2DTransformDataset class from lib/dataset.py as you did above. Next, update the transform function to change to your proposed coordinate system. Verify the hypothesis by training a linear classifier on the new representation. You will have to uncomment L65 and L75 in train.py.

The data can be linearly separable if mapped on a surface in higher dimension. For example, we could map our data to the 3D surface:

$$x^2 + y^2 = z$$

This surface represents a bowl in 3D. An instinctive way to see why it would work is that we can draw a 2D bowl in 3D by taking a 2D circle and lifting it at the same time as expanding its radius from zero to some arbitrary value. This way we can visualize that our two data-sets will not lie at the same height on the bowl. Their x,y coordinates will not change, only their z coordinate. We could express our mapping as follows:

$$(x, y) \longrightarrow (x, y, f(x, y)) = (x, y, x^2 + y^2)$$

.

From now on, we will only need to find a 3D plane that splits our data instead of a 2D line. A plane can equation is linear and can be approximated by the Linear classifier.

Listing 5: Transformation of the data

```
class Simple2DTransformDataset(Dataset):

    ...

    def __getitem__(self, idx):
        ...

        # Transform the sample to a different coordinate system.
        sample = transform(sample)

        # Convert to tensor.
        return {
            'input': torch.from_numpy(sample).float(),
            'annotation': torch.from_numpy(annotation[np.newaxis]).float()
        }

def transform(sample):
    #raise NotImplementedError()
    new_sample = np.array([sample[0], sample[1], sample[0]**2 + sample[1]**2])
    return new_sample
```

As we can see below our hypothesis is verified as we get a final accuracy of 99.6 percents. It indeed proves that our data was linearly separable on a higher dimension manifold. Moreover, it is worth to note that we achieved a similar accuracy as with the MLP but with the use of less parameters to train.

```
[Epoch 01] Loss: 0.9054
[Epoch 01] Acc.: 64.4841%
[Epoch 02] Loss: 0.6072
[Epoch 02] Acc.: 62.6984%
[Epoch 03] Loss: 0.4898
```

```
[Epoch 03] Acc.: 57.7381%
[Epoch 04] Loss: 0.4399
[Epoch 04] Acc.: 77.5794%
[Epoch 05] Loss: 0.3995
[Epoch 05] Acc.: 88.4921%
[Epoch 06] Loss: 0.3620
[Epoch 06] Acc.: 96.4286%
[Epoch 07] Loss: 0.3295
[Epoch 07] Acc.: 93.0556%
[Epoch 08] Loss: 0.2991
[Epoch 08] Acc.: 98.2143%
[Epoch 09] Loss: 0.2716
[Epoch 09] Acc.: 97.6190%
[Epoch 10] Loss: 0.2485
[Epoch 10] Acc.: 99.6032%
```

3 MNIST Classifier

3.1 Data normalization *Gray-scale images are generally stored in memory as uint8 arrays. This means that each pixel can take values between 0 and 255. For better convergence of neural networks, it is generally common practice to normalize the values to the range [-1, 1]. Update the normalize function in lib/dataset.py accordingly.*

Listing 6: MNIST Dataset for normalization of the data

```
class MNISTDataset(Dataset):

    ...

    def __getitem__(self, idx):
        # Returns the sample with index idx from the dataset as a torch tensor.
        sample = self.data[idx]
        annotation = self.annotations[idx]

        # Images are generally represented as uint8 matrices ([0 .. 255]).
        # Normalize the data between -1 and 1!
        sample = normalize(sample)

        return {
            'input': torch.from_numpy(sample).float(),
            'annotation': torch.tensor(annotation).long()
        }


def normalize(sample):
    new_sample = 2 * (sample / 255.0) - 1.0
    return new_sample
```

3.2 Training loop *As above, finish implementing the training loop in the run training epoch function from train.py. The main difference is the loss which should be cross entropy instead of the binary version.*

Listing 7: Implementation of the training loop for the MNIST classifier

```
def run_training_epoch(net, optimizer, dataloader):
    loss_aggregator = UpdatingMean()
    # Put the network in training mode.
    net.train()
    # Loop over batches.
    for batch in tqdm(dataloader):
```

```
        #raise NotImplementedError()
        # Reset gradients.
        # TODO
        optimizer.zero_grad()

        # Forward pass.
        output = net.forward(batch['input'])

        # Compute the loss - cross entropy.
        # Documentation https://pytorch.org/docs/stable/generated/torch.nn.functional.cross_entropy.html.
        loss = F.cross_entropy(output, batch['annotation'])

        # Backwards pass.
        # TODO
        loss.backward()
        optimizer.step()

        # Save loss value in the aggregator.
        loss_aggregator.add(loss.item())
    return loss_aggregator.mean()
```

3.3 Multi-layer perceptron *Implement a linear classifier by filling in the MLPClassifier class from lib/networks.py. What performance do you obtain? Next, use an MLP with one hidden layer of dimension 32 followed by ReLU and then the final linear prediction layer. What is the new testing accuracy?*

Accuracy with the Linear classifier.

```
[Epoch 01] Loss: 0.4121
[Epoch 01] Acc.: 91.1500%
[Epoch 02] Loss: 0.3328
[Epoch 02] Acc.: 91.3400%
[Epoch 03] Loss: 0.3213
[Epoch 03] Acc.: 90.6300%
[Epoch 04] Loss: 0.3147
[Epoch 04] Acc.: 91.7100%
[Epoch 05] Loss: 0.3094
[Epoch 05] Acc.: 91.1100%
```

The accuracy with the Linear classifier is of 91.1 percents. This is not so surprising as if we look at the input images as vectors, there is a very high chance for the vectors to be structured. Instinctively, the zero should have zero elements in around the middle of the vector in a striped pattern. Instinctively, there should exists a linear way of splitting digits in higher dimension.

Accuracy with the MLP with 1 hidden layer.

```
[Epoch 01] Loss: 0.4133
[Epoch 01] Acc.: 90.3600%
[Epoch 02] Loss: 0.2851
[Epoch 02] Acc.: 91.8300%
[Epoch 03] Loss: 0.2405
[Epoch 03] Acc.: 93.0800%
[Epoch 04] Loss: 0.2188
[Epoch 04] Acc.: 92.9800%
[Epoch 05] Loss: 0.2059
[Epoch 05] Acc.: 94.0900%
```

The accuracy is of 94 percents and is a bit better than with the Linear classifier. As said before, even if the data is sort of linearly separable in higher dimensions, there still exists some non-linearities. For example, instinctively with the digits that are lookalike and have rounded shapes. Since the MLP can approximate non-linear functions and so it do better at splitting the data.

Listing 8: Implementation of the MLP

```
class MLPClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.codename = 'mlp'

        #explanation of how to build the architecture

        # Define the network layers in order.
        # Input is 28 * 28.
        # Output is 10 values (one per class).
        # Multiple linear layers each followed by a ReLU non-linearity (apart from the last).
        #raise NotImplementedError()

        #if using the single Linear Classifier
        #self.layers = nn.Sequential(
        # nn.Linear(28 * 28, 10)
        #)

        self.layers = nn.Sequential(
            nn.Linear(28 * 28, 32), #input -> hidden
            nn.ReLU(), #hidden
            nn.Linear(32, 10) # hidden -> output
        )


    def forward(self, batch):
        # Flatten the batch for MLP.
        b = batch.size(0)
        batch = batch.view(b, -1)
        # Process batch using the layers.
        x = self.layers(batch)
        return x
```

3.4 Convolutional network *Implement a convolutional network by filling in the ConvClassifier class from lib/networks.py. The network should be as follows: 1. Convolutional layer nn.Conv2d with 3 × 3 kernel and 8 channels followed by ReLU and 2 × 2 max pooling nn.MaxPool2d with stride 2. 2. Convolutional layer nn.Conv2d with 3 × 3 kernel and 16 channels followed by ReLU and 2 × 2 max pooling nn.MaxPool2d with stride 2. 3. Convolutional layer nn.Conv2d with 3 × 3 kernel and 32 channels followed by ReLU and the already defined adaptive max pooling Finally, the classifier should be a simple linear prediction layer. What testing accuracy do you obtain with this architecture?*

Listing 9: Architecture definition of the Convolutional neural network

```
class ConvClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.codename = 'conv'

        # Define the network layers in order.
        # Input is 28x28, with one channel.
        # Multiple Conv2d and MaxPool2d layers each followed by a ReLU non-linearity (apart from the last).
        # Needs to end with AdaptiveMaxPool2d(1) to reduce everything to a 1x1 image.
        #raise NotImplementedError()
        self.layers = nn.Sequential(

            nn.Conv2d(1, 8, (3, 3)), # input layer - 1st hidden layer
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2),

            nn.Conv2d(8, 16, (3, 3)), # 1st hidden - 2nd hidden
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2),

            nn.Conv2d(16, 32, (3, 3)), # 2nd hidden - 3rd hidden
            nn.ReLU(),
            nn.AdaptiveMaxPool2d(1), # output 32-D signal for 1 st linear layer

        )

        # Linear classification layer.
        # Output is 10 values (one per class).
        self.classifier = nn.Sequential(
            nn.Linear(32, 10) # 1 st linear layer - output layer
        )

    def forward(self, batch):
        # Add channel dimension for conv.
        b = batch.size(0)
        batch = batch.unsqueeze(1)
        # Process batch using the layers.
        x = self.layers(batch)
        x = self.classifier(x.view(b, -1))
        return x
```

Accuracy of the Convolutional network.

```
[Epoch 01] Loss: 0.2886
[Epoch 01] Acc.: 96.6000%
```

```
[Epoch 02] Loss: 0.0970
[Epoch 02] Acc.: 97.7600%
[Epoch 03] Loss: 0.0713
[Epoch 03] Acc.: 97.8400%
[Epoch 04] Loss: 0.0608
[Epoch 04] Acc.: 97.8700%
[Epoch 05] Loss: 0.0525
[Epoch 05] Acc.: 98.0900%
```

The accuracy of the Convolutional network is of 98.1 percents. It is good at finding local patterns in some input. Our inputs are digits which do have a lots of local features that helps differentiates them. It is worth to note this is know that convolutional networks are good at performing recognition in images so the accuracy was to be expected.

3.5 Comparison of number of parameters *Compute the number of parameters of the MLP with one hidden layer. Compute the number of parameters of the convolutional network. You should count both the weights and biases. Provide detailed explanations and computations*

For the MLP we counted 25450 parameters and 6218 parameters for the convolutional neural network. Please, refer to the parameters_computations_details.pdf for the details.

3.6 Confusion matrix *The confusion matrix Mi,j is a useful tool for understanding the performance of a classifier. It is defined as follows: Mi,j is the number of test samples for which the network predicted i, but the ground-truth label was j. Ideally, in the case of 100run validation epoch function from train.py, update plot confusion matrix.py to compute the confusion matrix. Provide the plot in the report and comment the results.*

Listing 10: Computation of the confusion matrix

```
# Create the validation dataset and dataloader.
valid_dataset = MNISTDataset(split='test')
valid_dataloader = DataLoader(
    valid_dataset,
    batch_size=BATCH_SIZE,
    num_workers=NUM_WORKERS
)

# Create the network.
net = MLPClassifier()
#net = ConvClassifier()

# Load best checkpoint.
net.load_state_dict(torch.load(f'best-{net.codename}.pth')['net'])
# Put the network in evaluation mode.
```

```
net.eval()

# Create the optimizer.
optimizer = Adam(net.parameters())

# Based on run_validation_epoch, write code for computing the 10x10 confusion matrix.
confusion_matrix = np.zeros([10, 10])

for batch in valid_dataloader:

    output = net(batch['input'])
    expected = batch['annotation']

    # for each batch element, for each of the output we need to retain
    # the predicated class, the one with the higher value among the
    # predicted classes. We also need to retain the class that is the ground
    # truth

    for prediction, ground_truth in zip(output, expected):
        predicated_class = torch.argmax(prediction)
        #print("predicted : ", predicated_class, " and is : ", ground_truth)
        i = predicated_class
        j = ground_truth

        confusion_matrix[i][j] += 1
```

A perfect and utopian classifier would yield a perfectly diagonal confusion matrix for any data we feed to it. However, here it is not the case. We have 1000 samples for every digits, therefore, the more every diagonal elements are near to 1000, the more accurate is the model in regards to this data (neglecting some possible overfitting). Bellow, we see that it is the case for the convolutional network. All digits are fairly well recognized (bright color means high value) and the diagonal is well visible. Same for the MLP, it is indeed almost as good as compared to the convolution network with the exception of the digit 9 which was always miss-predicted as either 8, 7 or more often 4. In fact, other than these miss-predictions, the results look similar than those obtained with the convolutional neural network. We can observe that the accuracy of the MLP was 94 percents as opposed to 98 for the convolutional network. Therefore, we could make the hypothesis that differentiating 9 from 4 (and sometimes 8 or 7) relies on some little local features details that appear more clearly when the input is filtered through the convolutional network's layers and were hard to catch with the MLP simpler architecture. Hence, the accuracy is a bit better on the convolutional network even if both classifiers performed around the same for the rest of the digits.

As next questions to explore, we maybe could apply the same trick as in the first part and map the 784-D vectors to a higher dimension manifold that would allow splitting linearly the digits and so allow a better accuracy with the MLP? Maybe a deeper MLP architecture would yield in a better accuracy over the 9 digit?

Figure 2: Confusion matrix plot for the MLP classifier

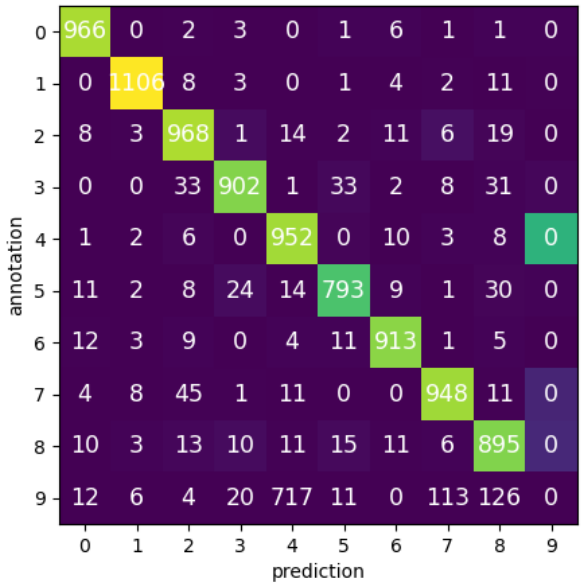


Figure 3: Confusion matrix plot for the Convolution network