TP3: NN and CNN with `Pytorch`

Kim ANTUNEZ, Isabelle BERNARD (Group : Mr Denis)

1 Introduction

> **Question 1:** Solve the problem $\min_{w\in\mathbb{R}^5}(1-x^\top w)^2$ with $x=(1,\dots,1)^\top\in\mathbb{R}^5$ analytically and compare to the result of the Gradient Descent.

If $x=(1,\dots,1)^\top\in\mathbb{R}^5$ and $w=(w_1,\dots,w_5)^\top\in\mathbb{R}^5$

Let be $f(w)=f(w_1,\dots w_5)=(1-x^\top w)^2=(1-w_1-w_2-w_3-w_4-w_5)^2$

As f is a positive function, we clearly see that the minimum of f is 0 and is reached for all the points w such as $\sum_{k=1}^5 w_k=1$

In particular the point $w=(0.2,0.2,0.2,0.2,0.2)$ which is the result of the previous Gradient Descent is one of the minima of the function. We can note that the result of the gradient descent would have been different if we had chosen another $w_0\neq(0,0,0,0,0)$

> **Question 2:** Recalling the theory of numerical optimization, what is the learning rate `lr` that we need to set to ensure the fastest convergence?

The learning rate η is a hyperparameter that determines how fast the algorithm learns. It controls how much to change the model in response to the estimated error each time the model weights are updated following this formula : $w_k=w_{k-1}-\eta\nabla L(w_{k-1})$. The learning rate is one of the most important hyperparameter when configuring our neural network. Choosing it is challenging because :

- **if η is too large**, the model learns too much (rapid learning) and may result in learning a sub-optimal set of weights or an unstable training process (unable to effectively gradually decrease our loss). The gradient descent can overshoot the local lowest value. It may fail to converge to a good local minimum or may even diverge.
- **if η is too small**, the model learns too little (slow learning) and it may take too long to converge or would even get stuck and unable to converge to a good local minima

➔ **To conclude, we must choose a learning rate which is larger enough to converge fast enough but not to large to prevent from rapid learning (being unable to converge to a good minimum). To choose it, we have to test different hyperparameters and define which is the best compromise ("hyperparameter tuning").**

Source : here and here

> **Question 3:** Explain the connection of `loss.backward()` and the backpropagation for feedforward neural nets.

Backpropagation is a short form for "backward propagation of errors". Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer. The backpropagation computes the gradient of the loss function with respect to the weights of the network. This helps to update weights to minimize loss. There are many update rules for updating the weights : mainly Gradient descent, Stochastic gradient descent, RMSProp, Adam.

`loss.backward()` computes gradient of loss (dloss/dw) with respect to all the parameters w in loss for which `requires_grad = True`. It stores them in the parameter.grad (`w.grad`) attribute for every parameter w.

Here, the output values are compared with the correct answer to compute the value of a predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small. In this case, one would say that the network has learned a certain target function. To adjust weights properly, one applies a general method for non-linear optimization that is called gradient descent. For this, the network calculates the derivative of the error function with respect to the network weights, and changes the weights such that the error decreases (thus going downhill on the surface of the error function). For this reason, back-propagation can only be applied on networks with differentiable activation functions.

Source : here

2 Multi layer perceptron

Question 4: Run the above block several times. Is it plotting the same number all the time? If not, why?

No, because `torch.utils.data.random_split` randomly splits the dataset all_train into non-overlapping new datasets (trainset and valset). We can optionally fix the generator for reproducible results, e.g.:

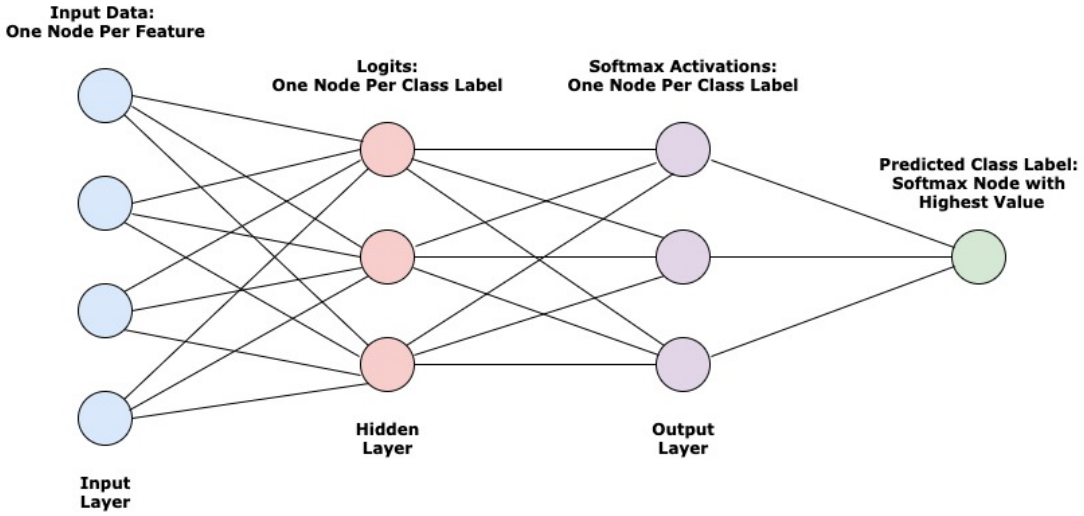
```
trainset, valset =  torch.utils.data.random_split(all_train, [num_train,  len(all_train) - num_train],
generator=torch.Generator().manual_seed(42))
```

3 Problem 1: Logistic regression via pytorch

1. Mathematical description of logistic regression

Logistic Regression can be thought of as a simple, fully-connected neural network with one hidden layer.

In the following network (see diagram), the **forward pass** are the three steps from the input to the output layer.



1. The input data in blue are the n features ($n = 4$ in the example)
2. In the hidden layer in red are the resulting matrix Z of multiplying each m rows (training observations) of the dataset a weight matrix with n rows and k columns, with k the number of unique classes you want to predict. We also add a bias to the result. The equation is $Z_{[m,k]} = X_{[m,n]}W_{[n,k]} + b_{[m,k]}$. $z_{i,j}$ is called the logit for the j^{th} label of the i^{th} training example

*Remark : To represent multinomial classes, we use **one-hot encoding** as we saw for TP2 : a simple transformation of a 1-dimensional vector of length m into a binary tensor of shape (m,k), where k is the number of unique classes. Each column in the new tensor represents a specific class label and for every row there is exactly one column with a 1, everything else is zero. But the PyTorch's loss function that we use (`nn.CrossEntropyLoss`) take directly class labels as their targets so we don't need to convert targets into onehot vectors.*

3. For the output layer in green, every logit of the matrix Z are passed through an activation function called Softmax (equation below) and the results are numbers between 0 and 1. $a_{i,j}$ will correspond to the probability that observation i is of the type j.

$$\sigma_i(z_{i,j}) = \frac{e^{z_{i,j}}}{\sum_{n=0}^{k} e^{z_{i,n}}} = a_{i,j}$$

➔ Finally, we pick the node in the output layer that has the highest probability and choose that as the predicted class label.

2. Mathematical description of optimization algorithm that you use

At the end of each forward pass in the training process, we use the activations to determine the performance of the model using the cost called **Cross Entropy Loss** : $Cross\,Entropy\,Loss = -\frac{1}{m}\sum_{i=0}^{m}\sum_{j=0}^{k} y_{i,j} \cdot \log(a_{i,j})$

Remark : To prevent the model from overfitting, we can also regularize the regression by simply adding a term to the cost function intended to penalize model complexity. We use : L2 Regularized Loss $= -\frac{1}{m}\sum_{i=0}^{m}\sum_{j=0}^{k} y_{i,j}\cdot\log(a_{i,j}) + \lambda\sum_{i=0}^{n}\sum_{j=0}^{k} w_{i,j}^2$ *with* $\lambda \leq 0$ *a hyperparameter to be tuned. We don't do it here because it is not specifically asked.*

Finally, during the **backward pass**, go back through the network and make adjustments to every hidden layer's parameters. The goal is to reduce the loss in the next training iteration. In this particular case of Logistic Regression, there's only one layer of parameters that will get adjusted by a method called **Gradient Descent** :

1. We first get the gradient of each model parameter using the backpropagation algorithm.

$$\nabla(W) = \begin{bmatrix} \nabla(w_{0,0}) & \nabla(w_{0,1}) & \cdots & \nabla(w_{0,k}) \\ \nabla(w_{1,0}) & \nabla(w_{1,1}) & \cdots & \nabla(w_{1,k}) \\ \vdots & \vdots & \cdots & \vdots \\ \nabla(w_{n,0}) & \nabla(w_{n,1}) & \cdots & \nabla(w_{n,k}) \end{bmatrix} \quad \text{where} \quad \nabla(w_{f,l}) = [\frac{1}{m}\sum_{i=0}^{m}(x_{i,f} * (y_{i,l} - a_{i,l}))] + [2\lambda * w_{f,l}]$$

$$\nabla(b) = \begin{bmatrix} \nabla(b_0) & \nabla(b_1) & \cdots & \nabla(b_k) \end{bmatrix} \quad \text{where} \quad \nabla(b_l) = \frac{1}{m}\sum_{i=0}^{m}(y_{i,l} - a_{i,l})$$

2. We update each model parameter in the opposite direction of its gradient.

Source : here and here

3. High level idea of how to implement logisitic regression with pytorch

Preliminary remark: In the function `train` constructed by the teacher, there was a little mistake that we had to correct so that the following code works. The following line…

```
#careful : the following line was a mistake in the teacher's proposition !
#outputs = net(inputs) # mistake we need to replace net by the parameter "model"
```

had been replaced by…

```
outputs = model(inputs) # Forwards stage (prediction with current weights)
```

```
# Add multinomial logistic regression
class MultinomialLogisticRegression(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.classifier = torch.nn.Linear(input_size, output_size)
        # We don't use Softmax here because the CrossEntropyLoss function computes softmax before the CE.

    def forward(self, x):
        x = x.reshape(-1, input_size)
        x = self.classifier(x)
        return x
```

```
# initializing our model/loss/optimizer
MLR = MultinomialLogisticRegression(input_size, output_size)
criterion = nn.CrossEntropyLoss() # /!\ computes softmax and then the cross entropy
optimizer = optim.SGD(MLR.parameters(), lr=lr)
```

```
# num_epochs indicates the number of passes over the data
for epoch in range(num_epochs):

    # makes one pass over the train data and updates weights
    train(MLR, trainloader, criterion, optimizer, epoch, num_epochs)

    # makes one pass over validation data and provides validation statistics
    val_loss, val_acc = validation(MLR, valloader, criterion)
```

4. Report classification accuracy on test data.

➲ The accuracy is 83 % with 2 epochs on test data.

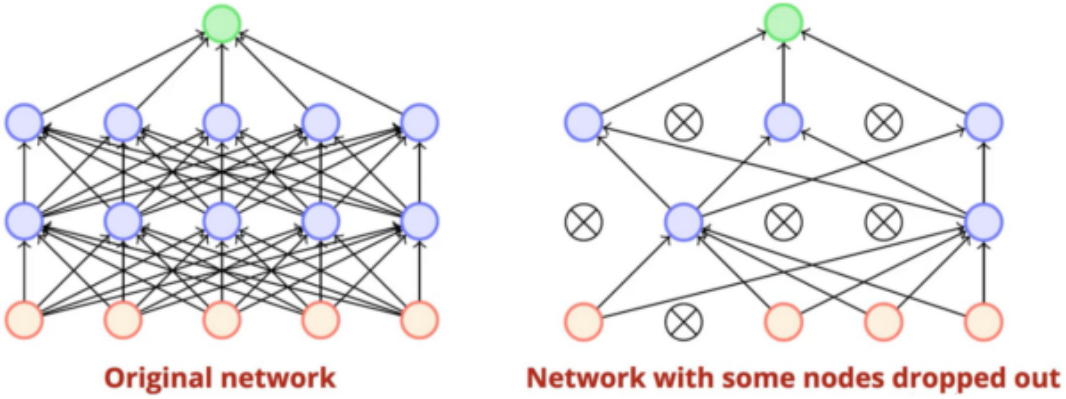
```
# Let us evaluate our net on the test set that we have never seen!
testset = datasets.MNIST('data/',
                         download=True,
                         train=False,
                         transform=transforms.ToTensor())
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=True)

test_loss, test_acc = validation(MLR, testloader, criterion)
print(f'Test accuracy: {test_acc} | Test loss: {test_loss}')
```

```
Test accuracy: 0.83 | Test loss: 0.9829374759674072
```

4 Problem 2: Dropout

1. High level description of the dropout

The idea of "Dropout" is to prevent from overfitting in deep neural network with large parameters on the data. As you can see in the following figure, "Dropout" deactivates the neurons randomly at each training step instead of training the data on the original network, we train the data on the network with dropped out nodes. In the next iteration of the training step, the hidden neurons which are deactivated by dropout changes because of its probabilistic behavior. In this way, by applying dropout i.e. . . deactivating certain individual nodes at random during training we can simulate an ensemble of neural network with different architectures.



In Pytorch, we simply need to introduce `nn.Dropout` layers specifying the rate at which to drop (i.e. zero) units. Learning a neural network with dropout is usually slower than without dropout.

Source : here

2. High level description of your architechture

We read many articles including the one recommanded by this subject or this one and implemented **3 different architectures**, still on two epochs (using more epochs was too long to compute). See the codes below.

1. For the first architecture `ConvNet2`, we simply implementes the previous architecture `ConvNet` but we add a 50 % probability dropout before the penultimate linear layer. The forward pass does not change.

➲ The accuracy of `ConvNet2` is the same as the previous model (98,3 % which is still very high) and thus does not improve the model. However, the model is slower because, as we said before, a neural network with dropout is usually slower.

2. For the second architecture `ConvNet3`, we use a more complicated network. Instead of using only 1 Conv2d-ReLU-MaxPool2d sequence, we use 3 of them. Then, we use 2 Linear-ReLU sequences instead of 1 and use before them a 50 % probability dropout (as in `ConvNet3`). We also include a dropout in the forward pass. The more precise architecture is:
   - The input layer
   - 3 consecutive blocks with Conv(5×5, strides=1, padding=2)-Relu-MaxPooling(2×2, strides=2). It takes 16 filters for the first Conv layer, 32 filters for the second and 64 for the third one.

- Two dropout-linear-relu blocks of size 128 and 256, with 0.5 dropout probability
- One dense layer with 10 units (linear but softmax is then embedded in the loss function).

➲ We notice a real improvement in the accuracy of the model which is now 98,9 % for `ConvNet3`. However, the model was slow to compute (around 30 min. on our computer for only 2 epochs).

3. Finally, for our third and last architecture, we use a very complex architecture, using not only `Conv2d`, `ReLu` and `MaxPool2d` layers but also **`BatchNorm2d`** and **`AvgPool2d`** in addition to Droupout:

   - `BatchNorm2d` applies Batch Normalization as described in this paper. The fact that the parameters of the previous layers change changes the distribution of each layer's inputs and slows down the training. The goal is here to accelerate deep network training by reducing internal covariate shift.
   - `AvgPool2d` applies an average pooling over an input signal composed of several input planes. Here, with global average pooling, we basically average pooling with a kernel of the same size (width,height) as the feature map.

   If 64C3s1 denotes a convolutional layer with 64 kernels, of size 3×3, with stride 1, with zero padding, the precise architecture can be described as:

   - The input layer
   - 64C3s1-BatchNorm-Relu-64C3s1-BatchNorm-Relu - MaxPool2s2
   - Droupout 40% - 128C3s1-BatchNorm-Relu-128C3s1-BatchNorm-Relu - MaxPool2s2
   - Droupout 40% - 256C3s1-BatchNorm-Relu-256C3s1-BatchNorm-Relu - GlobalAverage(kernel size of 7)
   - One dense layer with 10 units (linear but softmax is then embedded in the loss function).

   Indeed, it is recently suggested in the litterature to use small convolutional kernels (3×3 or 5×5) because it leads to less parameters (and more non linearity). The number of filters is usually increased as we go deeper in the network (because we expect the low level layers to extract basic features that are combined in the deeper layers). Finally, Lin, Chen, & Yan, 2013 suggested that we can completly remove the final fully connected layers and replace them by global average pooling. The network is thus less likely to overfit and you end up with much less parameters for a network of a given depth (this is the fully connected layers that usually contain most of your parameters).

➲ We noticed another improvement in the accuracy of the model which is now 99,1 % for `ConvNet4` but it took 40 minutes for 2 epochs.

```
net = ConvNet4() #test the model of our choice ConvNetX (2 to 5)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

for epoch in range(num_epochs):

    # makes one pass over the train data and updates weights
    train(net, trainloader, criterion, optimizer, epoch, num_epochs)

    # makes one pass over validation data and provides validation statistics
    val_loss, val_acc = validation(net, valloader, criterion)
```

```
test_loss, test_acc = validation(net, testloader, criterion)
print(f'Test accuracy: {test_acc} | Test loss: {test_loss}')
```

```
Test accuracy: 0.9912 | Test loss: 0.027332775755273177
```

A Code of Problem 2

```
class ConvNet2(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=5, stride=[1, 1], padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.classifier = nn.Sequential(
            nn.Dropout(0.5), #NEW DROPOUT = 50 % probability
            nn.Linear(14 * 14 * 8, 500),
            nn.ReLU(),
            nn.Linear(500, 10)
        )

    def forward(self, x):
        out = self.layer1(x)
        out = out.reshape(out.size(0), -1)
        out = self.classifier(out)
        return out
```

```
class ConvNet3(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Sequential( #more complicated network
            nn.Conv2d(1, 16,kernel_size=5,stride=1,padding=int((5-1)/2), bias=True),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(16, 32,kernel_size=5,stride=1,padding=int((5-1)/2), bias=True),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 64,kernel_size=5,stride=1,padding=int((5-1)/2), bias=True),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2)
        )

        probe_tensor = torch.zeros((1,1,28,28))
        out_features = self.layer1(probe_tensor).view(-1)
        num_classes = 10

        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(out_features.shape[0], 128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(128, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        out = self.layer1(x)
        out = out.reshape(out.size(0), -1)
        out = self.classifier(out)
        return out
```

```
def batch_relu(in_channels, out_channels, ks):
    return [nn.Conv2d(in_channels, out_channels,
                  kernel_size=ks,
                  stride=1,
                  padding=int((ks-1)/2), bias=True),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)]

class ConvNet4(nn.Module):

    def __init__(self):
        super().__init__()
        num_classes = 10
        base_n = 64
        self.features = nn.Sequential(
            *batch_relu(1, base_n, 3),
            *batch_relu(base_n, base_n, 3),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.4),
            *batch_relu(base_n, 2*base_n, 3),
            *batch_relu(2*base_n, 2*base_n, 3),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.4),
            *batch_relu(2*base_n, 4*base_n, 3),
            *batch_relu(4*base_n, 4*base_n, 3),
            nn.AvgPool2d(kernel_size=7)
        )

        self.lin1 = nn.Linear(4*base_n, num_classes)

    def forward(self, x):
        x  = self.features(x)
        x = x.view(x.size()[0], -1)
        y = self.lin1(nn.functional.dropout(x, 0.5, self.training, inplace=True))
        return y
```