Université de Genève

Deep Learning

14X050

Mini Projects

Auteur : Antoine Branca

E-mail : Antoine.Branca@etu.unige.ch

Auteur : Deniz Sungurtekin

E-mail : Deniz.Sungurtekin@etu.unige.ch

Auteur : Sajaendra Thevamanoharan

E-mail : sajaendra.thevamanoharan@etu.unige.ch

Github

17 December 2021

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Département d'informatique

Project 1 – Classification, weight sharing, auxiliary losses

The aim of this project is to implement a deep network such that, given a set of 2×14×14 tensors corresponding to pairs of 14×14 grayscale images, it predicts for each pair whether the first digit is less or greater than the second. The dataset is MNIST.

Dataset

We use the function **generate_pair_sets(N)** defined in the file **loading_functions.py** to generate the dataset. This function is almost the same as the one we can find in the file **dlc_pratical_prologue.py** with some modifications that take away pairs with the same digit so that each pair contains always one digit that is bigger than the other. This function returns six tensors :

- train_input : N x 2 x 14 x 14
- train_target : N
- train_classes : N x 2
- test_input : N x 2 x 14 x 14
- test_target : N
- test_classes : N x 2

We can visualize the data set.

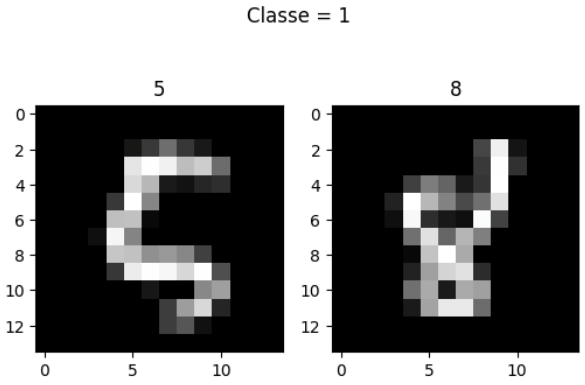

Figure 1: Sample of classe 1

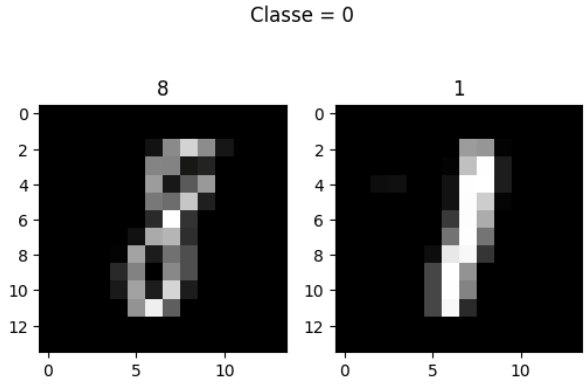

Figure 2: Sample of classe 0

We define the classe number as the index of the large number among the two layers. The training and test set should be 1000 pairs each. In other words, N = 1000.

The goal of the project is to compare different architectures and evaluate the performance improvement that can be achieved through weight sharing or using auxiliary losses. We implement two models for each architecture. Two models to check if weight sharing is more efficient. The same is true for auxiliary losses.

Weight Sharing

We know that the input data is a tensor of size N x 2 x 14 x 14, where N = 1000 in our case. The classical method of feature extraction is to apply a convolutional layer to each layer to extract information, which is then passed to the fully connected layers.

First, we implement a simple model as follows :

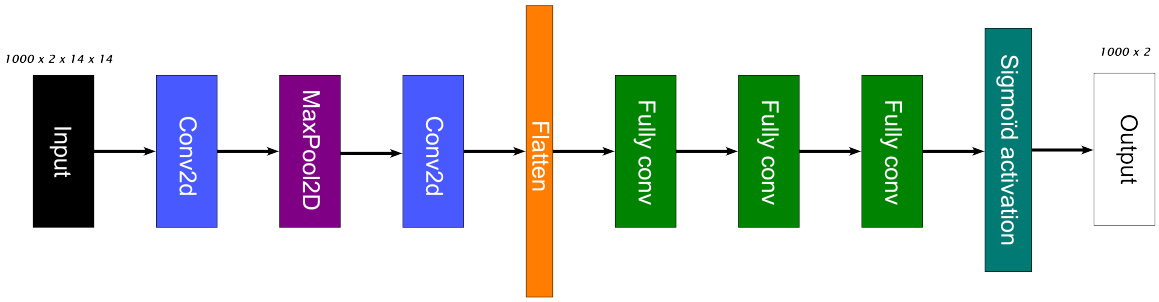

Figure 3: A simple model

The weight sharing comes into play when the Conv2d layer is the same for each layer of the input. The input consists of two layers of size 14x14, which again is an image of a handwritten digit. To compare the effects of weight sharing in a Convoluitonal Neural Network, we implement two models, one without weight sharing and the other with weight sharing.

Model 1

This model is implemented wihtout weight sharing. The two channels of the input are passed through two separate convolutional networks (**cf** Net1()). They are concatenated into one channel as the input of size 2x14x14. The concatenated input is passed through three fully connected layers.

Model 2

This model is implemented wiht weight sharing. The two channels of the input are passed through a single convolutional network (**cf** Net2()). They are concatenated into one channel as the input of size 2x14x14. The concatenated input is passed through three fully connected layers.

Performance evaluation

In this part, we will evaluate these two models. We run each model for 30 rounds of 25 epochs each, with a batch size of 50. We consider the performance of these two models as the average of the thirty rounds.

The performance of a model is also based on the amount of time spent. We will also study the time taken by these models during runtime. The evaluation metrics are the average and standard deviation of these measurements.

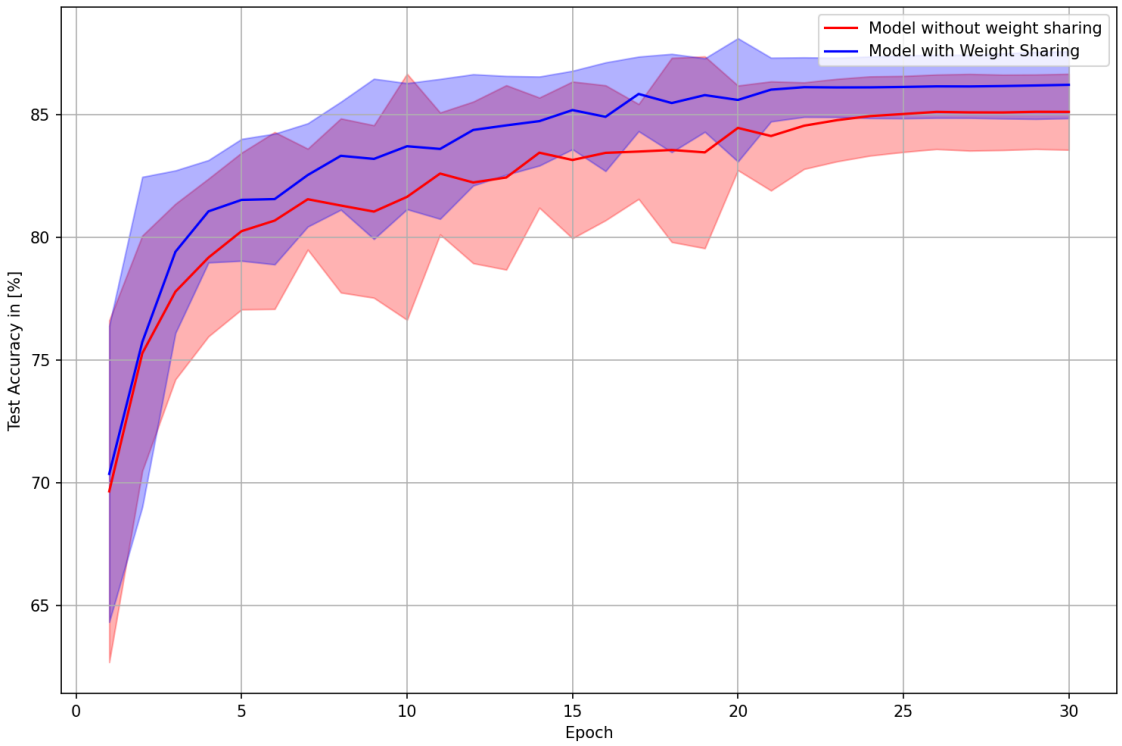

Figure 4: Mean and standard deviation per epoch over 30 rounds

Discussion

We can see that introducing the weight sharing into the model slightly improves the performance of the model. For the first model, we obtain a final test accuracy of $85 \pm 1.54\%$. For the second model, we obtain a final test accuray of $86 \pm 1.36\%$. Now we focus on the time consumption of these two models to make a better statement about their performances. The first model took an average of 9.8 seconds per round, while the second model took an average of 9.7 seconds. The advantage of performing weight sharing is that the algorithm takes less time. We run it here only for a small number of epochs. For a large number of epochs, the time difference between these two models would be larger. The model with weight sharing is more efficient than the model without. This is because the two layers of input have the same distribution (handwritten digits). Compared to other cases, there would be a different distribution for each layer in a color image (R,G,B). With the same distribution, the model is able to calculate the correct weights to predict the appropriate class in a short period of time.

Auxiliary losses

In this part we are going to see if we can improve the performance of our simple model (model 1, without weight sharing) through the introduction of auxiliary losses as this technic has been proven to give good performances in very deep model since it helps to propagate the gradient to the early layers of the network.

Model 3 (*Net1 & Net_ aux in AuxiliaryLoss.ipynb*)

The first model implementing auxiliary classifiers is a slight modification of model 1 with the addition of two auxiliary classifiers as represented below:

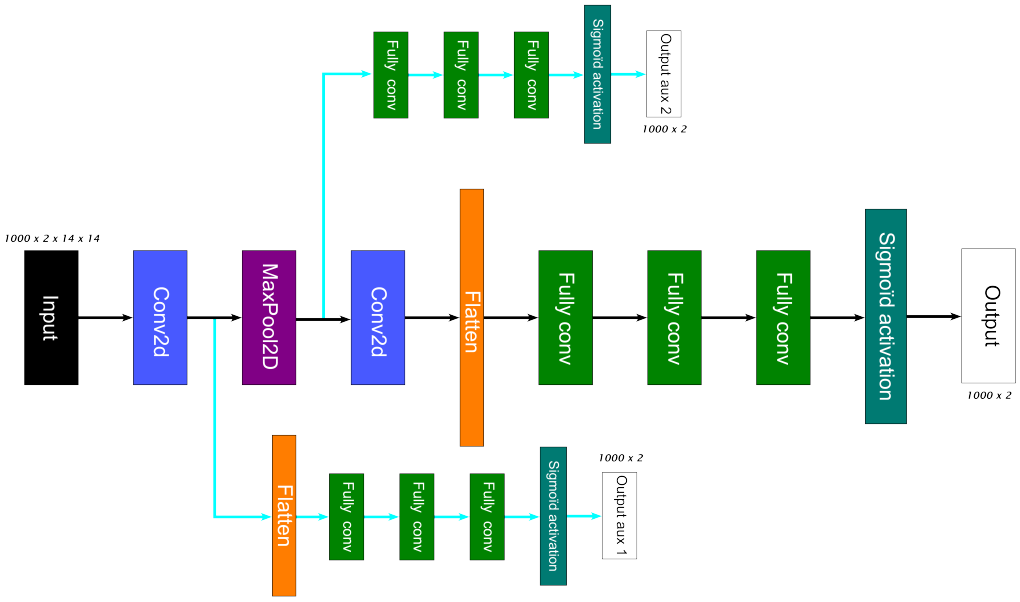

Figure 5: model 1 with two auxiliary classifiers

As for weight sharing we compared this model against the simple one (model 1) by averaging the prediction accuracy of each epoch over 50 rounds. Here is what we get.

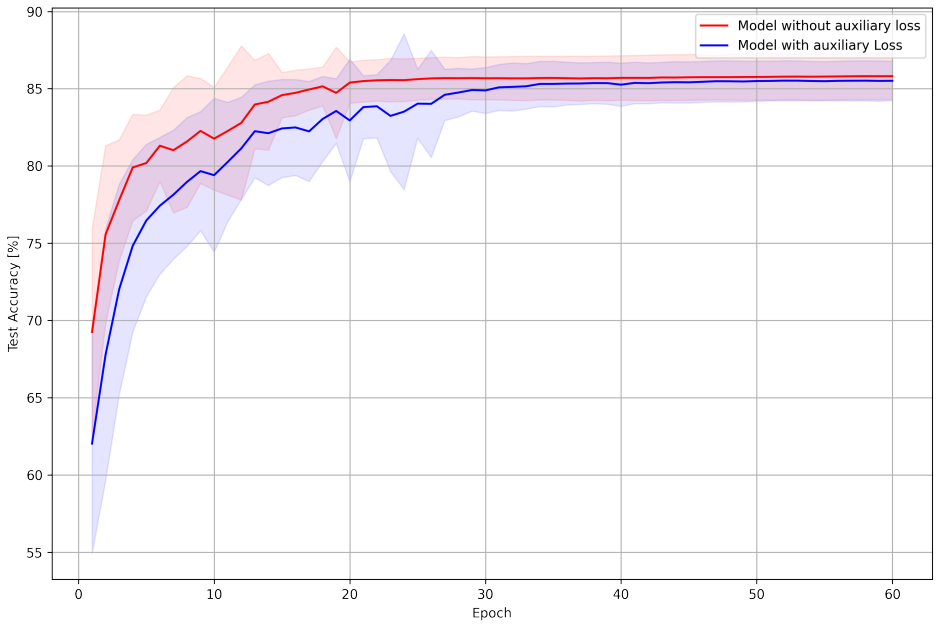

Figure 6: Performance comparison of a basic model

We got an average of $85.81 \pm 1.50\%$ for the final test accuracy for model 1 and $85.51 \pm 1.27\%$ with auxliary classifiers. As we can see there is no improvements of permormance by adding these two auxiliary classifiers. Plus, we can see a drastic reduction of computational performance since model 1 took an average of 11.25 seconds per round while model 3 took an average of 14.16 seconds per round. This is 26 % much slower for no improvements.
We can conclude that for such simple model there is no need to require additionnal complexity by introducing

auxiliary classifiers. This makes sense since the gradient has no time to vanish with a network of only two convolutional and three fully connected layers.

Model 4 (*Net1_deep & Net1_deep_aux in AuxiliaryLoss.ipynb*)

In order to see improvements due to the addition of auxiliary classifiers, we need a deeper model. Therefore, we implemented a network inspired by an AlexNet network. This fourth model has 5 convolutionnal layers and 3 fully connected layers. For the version with auxiliary losses we added three auxiliary classifiers. We can see a scheme of this model below (turquoise arrows represent auxiliary classifiers).

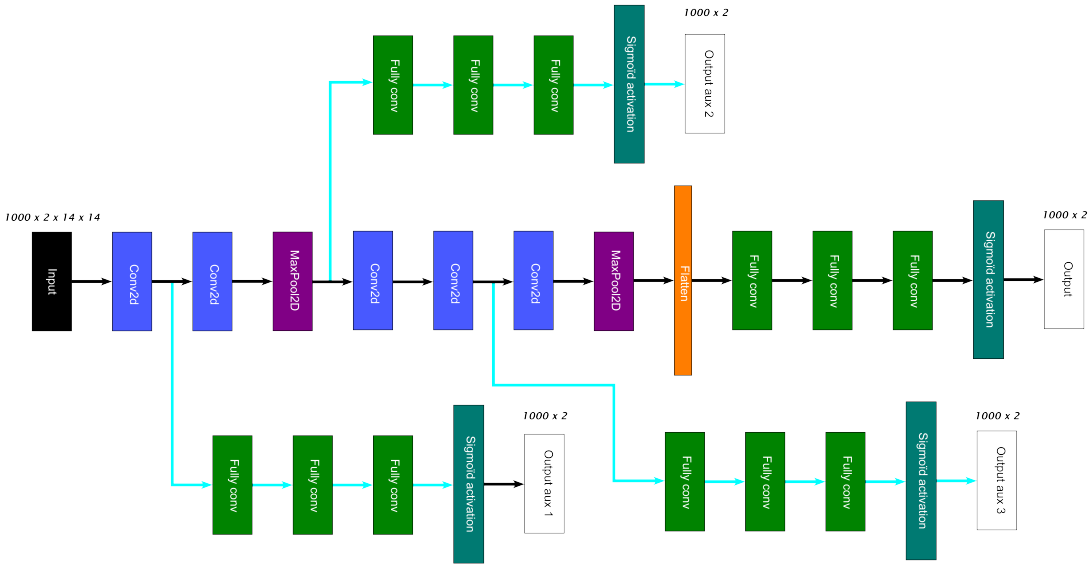

Figure 7: Deeper model (and auxiliary classifiers)

Let's compare the performance of both models (with and without classifiers). This time the network with auxiliary classifiers needed a smaller learning rate than the version without. It also took more time to converge (no improvements in learning) during the training phase. Therefore it needed more epochs. Here is the results we get (average of 50 rounds):

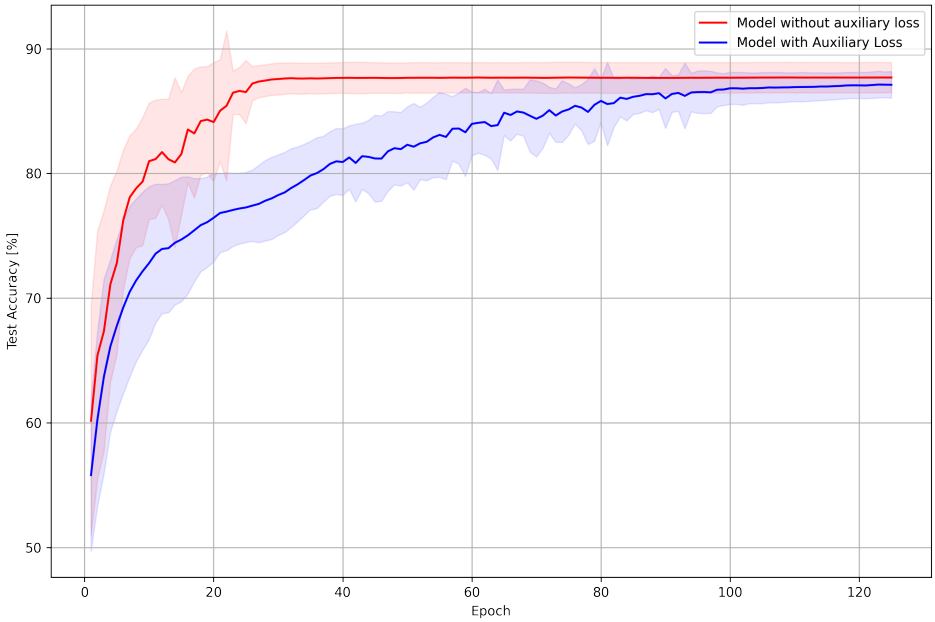

Figure 8: Test accuracy - with/without auxiliary classifier

We got a final test accuracy of $87.70 \pm 1.21\%$ without auxiliary classifiers and $87.11 \pm 1.07\%$ with auxiliary classifiers showing no improvements won by introducing additionnal complexity. In addition to that each training round of the model with auxiliary classifiers took $\approx 30\%$ more time than the model without (43.89 seconds against 33.65 seconds).

Model 5 (*Net1_deep & Net1_deep_classes in AuxiliaryLoss.ipynb*)

The last model we are going to study is going to take advantage of another piece of information that was hidden before: the class of each digit of the pairs. We are going to insert two auxiliary classifiers just before the last fully connect layer that is going to return two times a vector of dimension 1000 x 10. This time during training we are going to use the training classes to compute the auxiliary loss and the training target (0 if first digit is bigger and 1 if second is bigger) for the main output. A scheme of this network is represented on the figure below:

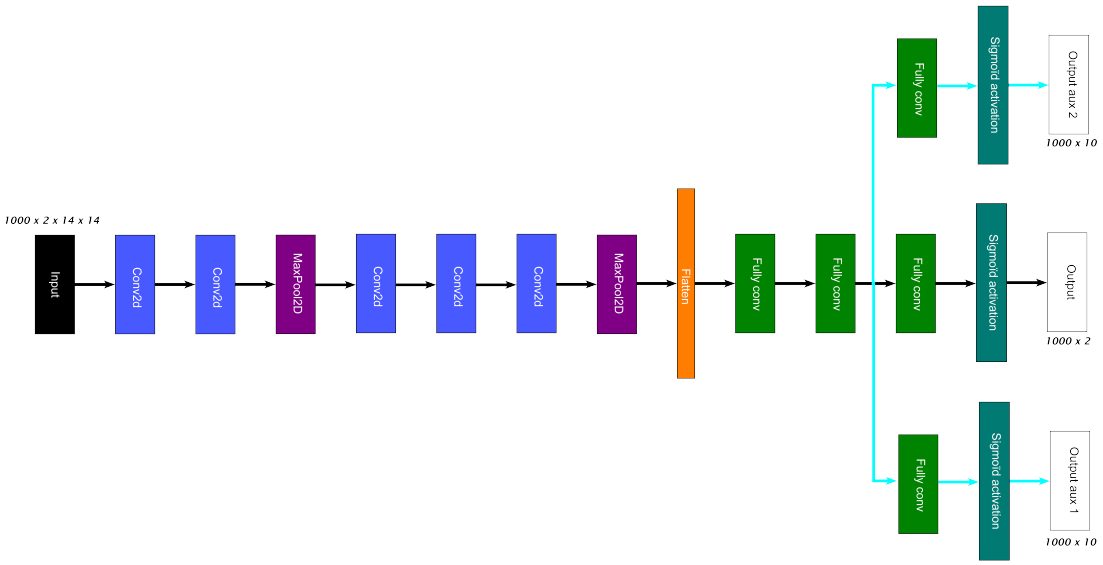

Figure 9: Network using training classes and training targets

We did the same test as before but this time again decreasing the learning rate a lot since the two new auxiliary losses made the whole network having a gradient of the loss more important than before (we can see a lot of fluctuactions during training if we let the same learning rate as before). Therefore, we had to increase significantly the number of epochs to about 400 against 60 for the model without auxiliary classifiers to achieve convergence during the training phase. Since the model without auxiliary loss only need 50 epochs to converge we did not compute the extra epochs. On the plot, we only kept the 50 last epochs to show the convergence value of the test accuracy. Here are our results.

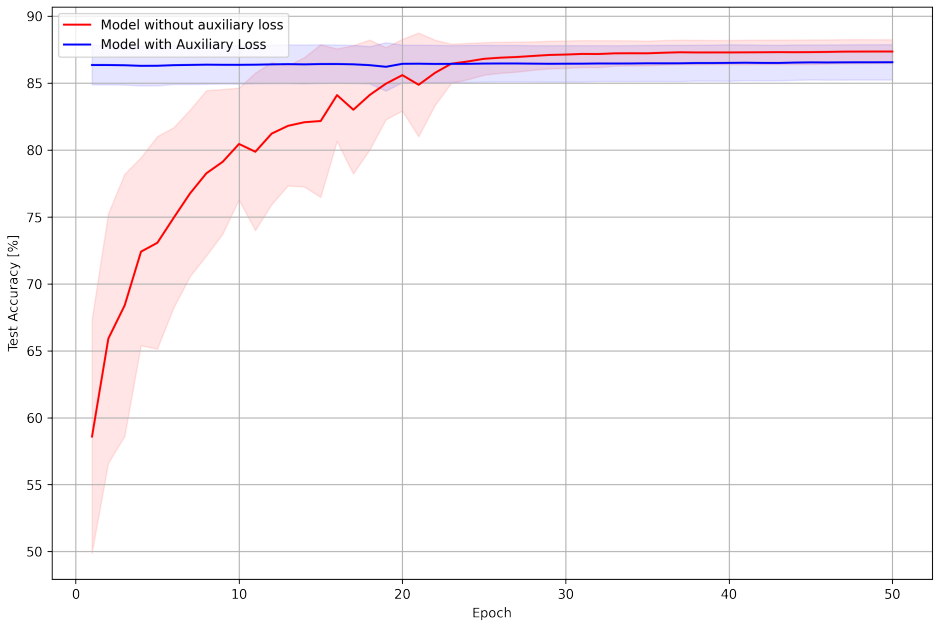

Figure 10: Test accuracy - with/without auxiliary classifier

We got a final test accuracy of $87.36 \pm 0.88\%$ without auxiliary classifiers and $86.56 \pm 1.32\%$ with auxiliary

classifiers.

Discussion

Despite all our models we were not able to improve our results by adding auxiliary classifiers. We only improved them by creating a deeper model ($\approx 2\%$ more accurate). The reason might be that the input of our model is two simple (only two channels of 14 x 14 pixels) and so the complexity added with auxiliary class does not bring any additionnal information.
Also, the models we built are also not that deep compared to GooLeNet (https://arxiv.org/abs/1409.4842) where auxiliary classifiers were successfully used.

Project 2 – Mini deep-learning framework

The objective of this framework is to provide the necessary tools to:

- Build networks combining fully connected layers, TanH and ReLU.
- Run the forward and backward passes.
- Optimize parameters with stochastic gradient descent for MSE.

Obviously, we will only use pytorch tensor operations and the standard math library to build this framework without using pre-existing neural-network python toolbox or autograd function.

Structure

First, we implemented the general structure of our framework. In order to be able to build neural networks, we begin by declaring a class with only three attributes and three subclasses:

```
class NeuralNetwork(object):

    def __init__(self):
        self.operations = []
        self.memory = []
        self.fls = [] #pile

    class Linear():
        def __init__(self,in_features,out_features,bias=True):
            self.in_features = in_features
            self.out_features = out_features
            self.bias = bias
            self.bool = False # indicate if bias has been broadcasted
            self.type = 1 # int to know which operation we use during bac

            if self.bias:
                self.bias = t.tensor([1/in_features]) # pytorch initiali
            else:
                self.bias = t.tensor([0])

            self.weights = t.rand((out_features,in_features))

        def evaluation(self,input):
            mul = t.matmul(self.weights,input)
            if not self.bool:
                self.bias = t.full(mul.size(),self.bias[0])
                self.bool = True

            output = t.add(mul,self.bias) # Wx' + b = output
            return output

    def add(self,*operation):
        for op in operation:
            self.operations.append(op)
```

```
class reLU():
    def __init__(self):
        self.type = 2

    def evaluation(self,input):
        return input.apply_(lambda x: (max(0, x)))

    def mapFunction(self,x): #used in derivative to derive Relu
        res = 0
        if x > 0:
            res = 1
        elif x <= 0:
            res = 0
        return res

    def derivative(self,input):
        return input.apply_(lambda x: (self.mapFunction(x)))

class tanH():
    def __init__(self):
        self.type = 2

    def evaluation(self,input):
        return (t.exp(input)-t.exp(-input) )/ (t.exp(input) + t.exp(-input))

    def derivative(self,input):
        return (1 - (self.evaluation(input)**2))
```

Figure 11: General structure

We will describe later in details the use of the attributes "operations", "memory" and "fls" which will be used to compute the backpropagation.

As regards the Linear class, it's an object taking 3 parameters for initialization (similar to pytorch):

1. $in\_features$, the number of features from our input value.
2. $out\_features$, the number of features from our output value.
3. $bias$, a boolean which specify if we must add bias or not. (By default set to True, if false the bias is set to zero and isn't update during SGD.)

With theses parameters, we can easily initialize the associated weights matrix "W" of dimension ($out\_features$,$in\_features$) and the bias "b". In addition, we add a "type" attribute which specify if the class is an activation function or not (1 for fully connected layer and 2 for activation function).
In order to implement the forward pass, we add an evaluation function which broadcast the dimension of the bias and computes Wx' + b.

For the classes TanH and reLU, we implement an evaluation function which returns the output of the activation function and a derivative function which return the derivative.

We also need a function to add layers to our neural network:

```
def add(self,*operation):
    for op in operation:
        self.operations.append(op)
```

```
nn.add(nn.Linear(2,25),nn.tanH(),nn.Linear(25,25),nn.reLU(),nn.Linear(25,1),nn.reLU())
```

Figure 12: Construction of a neural network

So, with this function, we keep the layers specified by the users in a list "operations" and we can implement the forward pass:

```
def forward(self,input):#add y_train to see precision at each step
    input = t.transpose(input,0,1) #transpose
    self.memory = [] # reset memory for each FF # In example = [x0,W1x0,x1,w2x1]
    self.fls = []
    for operation in self.operations:

        self.memory.append(input) #Store x before FL or z before activation function
        if operation.type == 1:
            self.fls.append(operation)
        input = operation.evaluation(input)

    size = input.size() # Just to check if we dont have a zeros tensor as output because with reLu its mean the neural is dead -> relu(0) = 0, relu'(0) = 0
    zero = t.zeros(size)
    if t.equal(input,zero):
        print("Warning: your output is zero and might not be learning with reLU -> Try to lower your learning step.")

    return input
```

Figure 13: Forward pass

Here, we just transpose the input (to match the dimension y = Wx'+b) and call the evaluation function of each class in the operation list. Besides, we will keep in memory the evaluation of the input at each step in the stack "memory" and the fully connected layers in the stack "fls" (To have an easy access to the parameters). We will use these two stacks to compute the gradients during the backpropagation.

Now that we have our forward pass, we want to optimize our parameters (weights and bias of the fully connected layers). For this, we need to compute the derivative of the loss (MSE in our case) w.r.t the weights and bias of the system. So, to do this we need to use the chain rule by beginning to compute the derivative of our loss function w.r.t the output:

$$Loss = \frac{1}{N}\sum_{i=1}^{N}(y_i - output_i)^2 = E$$

$$\Leftrightarrow \frac{\partial E}{\partial output} = \frac{-2}{N}(y - output)$$

where N is the number of samples.

For example, let's assume we have built a neural network with two fully connected layers (FL) and two activation functions (σ_i) where the output of each pair i of a fully connected layer and an activation function are noted x_i (for example here the final output is x_2):

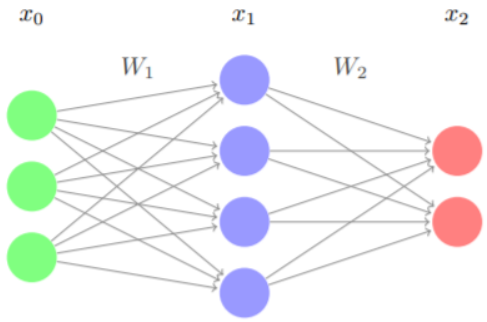

Figure 14: Example

$$FL - \sigma_1 - FL - \sigma_2$$

We will have:

$$\frac{\partial E}{\partial W_2} = \frac{-2}{N}(y - x_2)\frac{\partial(\sigma_2(W_2 x_1)}{\partial W_2} = [\frac{-2}{N}(y - x_2) * \sigma_2'(W_2 x_1)]x_1'$$

where * is the element-wise multiplication.

To simplify the computation, we can define:

$$\delta_2 = \frac{-2}{N}(y - x_2) * \sigma_2'(W_2 x_1)$$

So we obtain:

$$\frac{\partial E}{\partial W_2} = \delta_2 x_1'$$

Let's continue to the next weight matrix:

$$\frac{\partial E}{\partial W_1} = \frac{-2}{N}(y - x_2)\frac{\partial(\sigma_2(W_2 x_1)}{\partial W_1} = \delta_2 \frac{\partial W_2 x_1}{\partial W_1} = W_2'\delta_2 \frac{\partial x_1}{\partial W_1}$$

$$\frac{\partial E}{\partial W_1} = W_2'\delta_2 \frac{\partial(\sigma_1(W_1 x_0))}{\partial W_1} = [W_2'\delta_2 * \sigma_1'(W_1 x_0)]\frac{\partial W_1 x_0}{\partial W_1}$$

Again, we simplify:

$$\delta_1 = W_2'\delta_2 * \sigma_1'(W_1 x_0)$$

And we obtain:

$$\frac{\partial E}{\partial W_1} = \delta_1 \frac{\partial W_1 x_0}{\partial W_1} = \delta_1 x_0'$$

We didn't show here the development of for the partial derivative of the bias since it's the same reasoning where the inner derivative is 1 and not x_{i-1}.

This simple example, allows us to understand that we can generalize this for any layer where the only difference is the delta for the last layer:

$$\delta_L = (x_L - y) * \sigma_L'(W_L x_{L-1})$$

And the delta for other layer is given by:

$$\delta_i = W_{i+1}'\delta_{i+1} * \sigma_i'(W_i x_{i-1})$$

So the gradient for every weight matrix and bias are given by:

$$\frac{\partial E}{\partial W_i} = \delta_i x'_{i-1}$$

and

$$\frac{\partial E}{\partial b_i} = \delta_i$$

So now that we understand the computation of the gradient we can implement our backward function:

```
def backward(self,output,operation,y_train): # cf nn2.pdf
    gradiants = []
    deltas = []
    derivative = self.MSE_derivative(y_train,output)
    i = 0
    operations = c.deepcopy(operation)
    operations = operations[::-1] # Inverse list order

    for op in operations:
        if op.type == 2:
            if i == 0:
                derivative = t.mul(derivative, op.derivative(self.memory.pop()))  # compute of delta if i == 0
                deltas.append(derivative)
            else:
                w=self.fls.pop().weights #taking the weight of next layer to compute derivative c.f report
                derivative = t.mul(t.matmul(t.transpose(w,0,1),deltas.pop()),op.derivative(self.memory.pop()))
                deltas.append(derivative)

        if op.type == 1:
            gradiant = {}
            gradiant['b'] = derivative # doesnt change because if we derive wrt b we obtain 1 -> last computed d
            gradiant['w'] = t.matmul(derivative,t.transpose(self.memory.pop(),0,1)) #delta2 x_i-1 ' , in example
            gradiants.append(gradiant)
        i += 1
    return gradiants[::-1] #reverse to have the gradiants in the good order
```

Figure 15: Backward

Here, we will apply this generalization by looking over the operations in the reverse order to compute the gradients. We will apply the chain rule at each step by updating the value of the variable "derivative" and keeping the deltas value in a stack.

As we keep the values inside our "memory", we can pop the required $W_i x_{i-1}$ and x_i in the right order to compute the gradients.

Now that we have our gradients we can update our parameters at each training step by accumulating the gradient (or not). For that we will add a SGD function and a train function to our module:

```
def SGD(self,operations,gradiants,step,bias): #update weights
    j = 0  # j is an index corresponding to the ith gradiants (1/2
    for i in range(len(operations)):
        if operations[i].type == 1:
            operations[i].weights -= step * gradiants[j]["w"]
            if bias:
                operations[i].bias -= step * gradiants[j]["b"]
            j+=1
```

```
def train(self,x_train,y_train,batch_size,epochs,training_step=0.01,grad_accumulate=0,bias=True):
    count = grad_accumulate
    N = x_train.size()[0]
    indexs = [i for i in range(0,N,batch_size)]
    batchs = [x_train[i:i+batch_size] for i in indexs]
    y_trains = [y_train[i:i+batch_size] for i in indexs]
    countfirst = True
    for i in range(epochs):
        if i%5000 == 0:
            print("Start of epochs ",i)
        for batch,y_train in zip(batchs,y_trains):
            output = self.forward(batch)
            gradiants = self.backward(output,self.operations,y_train)
            if grad_accumulate == 0:
                self.SGD(self.operations,gradiants,training_step,bias)
                grad_accumulate = count
            else:
                if countfirst: # The first iteration we initialize the accumulate gradient
                    acc_grads = gradiants
                    countfirst = False
                else:
                    for acc_grad,grad in zip(acc_grads,gradiants):
                        acc_grad["w"] += grad["w"]
                        acc_grad["b"] += grad["b"]
                grad_accumulate -= 1
```

Figure 16: Training

The SGD function will simply update the parameters at iteration k by followings this rule:

$$W_{ik+1} = W_{ik} - \alpha \nabla W_{ik}$$

$$b_{ik+1} = b_{ik} - \alpha \nabla b_{ik}$$

Where α is the learning step and ∇ the corresponding gradient.

The train function will take as parameters:

- x_train: the training dataset.
- y_train: the correspondings labels.
- batch_size: the size of each batch.
- epochs: the number of epochs.
- training_step: the value of the training step.(By default 0.01)
- grad_accumulate: the number of batch, we accumulate the gradients before updating our parameters. (By default 0)
- bias: A boolean which specify if we want to add and train bias.(By default True)

Thus, we have a function which perfoms the forward pass with each batch in the dataset, then do the backward pass to compute the gradient and update the parameters (Or accumulate it), this process will be repeated epochs number of times. Finally, the user can choose to add a bias or not and specifies the value of the learning step to train a model.

Additional features

Because of the way of computing the backward step, the neural network need to have a structure where there is always an activation function after a fully connected layer, so I added a "no activation" function which doesn't change the input and have a derivative of 1. This allows the user to only use fully connected layer to build a neural network. In addition, it's very easy to add others activation function in the framework just by defining a new class with an evaluation method returning the desired output and a derivative method (I added sigmoid as an example).

An example:

```
class noActivation(): # because the backpropagation supose that after the line
    def __init__(self):
        self.type = 2

    def evaluation(self,input):
        return input

    def derivative(self,input):
        size = input.size()
        return t.ones(size)

class sigmoid():
    def __init__(self):
        self.type = 2

    def evaluation(self,input):
        return 1/(1+t.exp(-input))

    def derivative(self,input):
        size = input.size()
        ones = t.ones(size)
        return t.mul(self.evaluation(input),(ones-self.evaluation(input)))
```

Figure 17: Activation function

We also added a "computeAcc" method to allow the user to easily compute the accuracy of their model:

```
def computeAcc(self,x_test,y_test,batch_size):
    warnings.filterwarnings("ignore")
    N = y_test.size()[0]
    y_preds = []
    indexs = [i for i in range(0, N, batch_size)]
    for i in indexs:
        y_pred = self.round(self.forward(x_test[i:i+batch_size]))
        y_preds.append(y_pred)

    #flatten the list (list of list of list)
    y_preds = [x for sub in y_preds for x in sub]
    y_preds = [x for sub in y_preds for x in sub]
    y_preds = t.FloatTensor(y_preds)

    prediction = (t.tensor(y_preds).flatten() == y_test.flatten()).tolist()
    true_positif = prediction.count(True)
    print("Accuracy = ", true_positif / N, "%")
    return true_positif / N
```

Figure 18: Compute accuracy

Thus, if the users are happy with their result, they can save/load a model to use it later or to continue the training with the updated parameters:

```
def saveModel(self,model,name): #Sa
    string = "models/"
    string += name
    file = open(string,"wb")
    pickle.dump(model,file)
    file.close()

def loadModel(self, name):
    string = "models/"
    string += name
    file = open(string, "rb")
    load_model = pickle.load(file)
    file.close()
    return load_model
```

Figure 19: Save and load a framework

The model will be saved in a folder named "models" with the specified file name.

Testing our framework

To test our framework we will:

- Implement a test.ipyn that imports our framework to train and save a model.
- Implement a test executable named test.py that imports our framework to load a model and compute the final accuracy.
- Generate a training and a test set of 1,000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if outside the disk centered at (0.5, 0.5) of radius $1/\sqrt{2\pi}$, and 1 inside.
- Builds a network with two input units, one output unit, three hidden layers of 25 units.

First, let's define a function generating our datasets:

```
def generateData():
    # Generating x test and train set
    x_train = t.rand((1000,2))
    x_test = t.rand((1000,2))

    # Generate y test and y train set
    radius = 1/((2*m.pi)**(1/2)) # radius of the circle
    limit_up = 0.5+radius # Limit up and down of the circle
    limit_down = 0.5-radius

    y_train = t.tensor([0 if ((pair[0]>limit_up or pair[1]>limit_up) or (pair[0]<limit_down or pair[1]<limit_down)) else 1 for pair in x_train])
    y_train = y_train.view(y_train.size()[0],1) #training y must be in size (n,1) and not (n)
    y_test = t.tensor([0 if ((pair[0]>limit_up or pair[1]>limit_up) or (pair[0]<limit_down or pair[1]<limit_down)) else 1 for pair in x_test])
    y_test = y_test.view(y_train.size()[0],1)

    return x_train,y_train,x_test,y_test

x_train,y_train,x_test,y_test = generateData()
```

Figure 20: Generating the datasets

Now, we can train our model on jupyter notebook, compute the train accuracy and save our model:

```
Entrée [ ]: #Example of initialisation

            import framework as fw
            import torch as t
            import test as test

            x_train,y_train,x_test,y_test = test.generateData() #Generate the data

Entrée [ ]: nn = fw.NeuralNetwork()  #Initialize and train your model

            batch_size = 50
            epochs = 50000
            l_rate = 0.0001

            nn.add(nn.Linear(2,25),nn.tanH(),nn.Linear(25,25),nn.reLU(),nn.Linear(25,1),nn.reLU())
            nn.train(x_train,y_train,batch_size,epochs,l_rate)

Entrée [12]: print("Train accuracy: ")
             nn.computeAcc(x_train,y_train,batch_size)

             Train accuracy:
             Accuracy =  0.896

Out[12]: 0.896

Entrée [ ]: nn.saveModel(nn,"MonModel") # Save your model
```

Figure 21: Training and saving our model

Let's load our model with the test.py and compute our final test accuracy:

```
x_train,y_train,x_test,y_test = generateData()
# Load a model
nn = fw.NeuralNetwork()
myNN = nn.loadModel("MonModel")

# Compute accuracy
batch_size = 50 #Needed because the model was trained with this batch size and the dimension of the bias depend on this batch size

print("Test accuracy: ")
myNN.computeAcc(x_test,y_test,batch_size)# A batch size is specified but the accuracy is computed on the entier dataset


Test accuracy:
Accuracy =  0.851
```

Figure 22: Final accuracy

Conclusion

In conclusion, the main difficulty on the implementation of this framework is the design of the backpropagation which requires to fully understand the mathematical part. At the start, the idea was to make a kind of replication of autograd function from pytorch which use graph to keep in memory the operations done on a tensor and compute the derivative. However, this technique is not easy to implement for a one month project and almost need another framework to make it work. In addition, we found very interesting to implement a function taking advantage of a generalized formula which only need to keep in memory the evolution of the input during the forward pass. Of course, the implementation of an autograd module could be an improvement for this framework by making it more flexible.