IFT6135 – Homework #1

Multilayer perceptrons - programming part

Iban Harlouchet 1034545

Jean-Philippe Reid 701300

February, 19 2018

The reader should note that we implemented the complete assignment in both Keras et Pytorch. We present in this document the results from Pytorch and include those from Keras in a separate document.

Problem 1

1.1 Building the model

In the following sub-questions, please specify the model architecture (number of hidden units per layer, and the total number of parameters), the nonlinearity chosen as neuron activation, learning rate, mini-batch size.

- Data preparation : The data was imported from the linked provided. The train, valid and test sets sizes are 50K, 10K and 10K respectively. The data was flatten from a 28×28 matrix to a 1×784 vector.
- The architecture is composed of two hidden layers with ReLU neuron activations,

```
model = torch.nn.Sequential(
          torch.nn.Linear(h0,h1),
          torch.nn.ReLU(),
          torch.nn.Linear(h1,h2),
          torch.nn.ReLU(),
          torch.nn.Linear(h2,h3))
```

  We use the cross entropy loss as training criterion and stochastic gradient descent to optimize the model parameters,

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

where `lr = 0.1` is defined as the learning rate, initialized to 0.1. It is important to note that the learning rate decreasing with the number of epoch, following an exponential decay (See Annexe B.2).

- The dimensions of the three fully connected layers are $\mathbb{R}^{h^0 \times h^1}$, $\mathbb{R}^{h^1 \times h^2}$ and $\mathbb{R}^{h^2 \times h^3}$ respectively. We initialize $h_1 = b_1 = 500$ and $h_2 = b_2 = 300$, the total number of parameters is given by 0.54581 M. We kept constant the input and output dimensions to $h_0 = 784$ and $h_3 = 10$ respectively.

```
# important information
cuda = False
batch_size = 100
num_epochs = 100
lr0 = 0.1

h0 = 28*28    #784
h1 = b1 = 500
h2 = b2 = 300
h3 = b3 = 10

print('Number of parameters = '+str((h0*h1 + h1*h2 + h2*h3 + b1 + b2 + b3)/1E6)+'M')

# we can also get this number directly in Pytorch
model = MLPLinear([h0,h1,h2,h3])
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
print('Number of parameters = '+str(params/1e6)+'M')

#Output :
Number of parameters = 0.54581M
Number of parameters = 0.54581M
```

- Finally, the mini-batch size and the number of epochs are initially set to 100 and 20 respectively.

```
batch_size = 100
num_epochs = 20
```

1.2 Initialization

In this sub-question, we consider different initial values for the weight parameters. Set the biases to be zeros, and consider the following settings for the weight parameters: none, zero, normal and glorot.

For this section, we implemented more general modules which build the model using different initialization.

See the class model `MLPLinear` and the modules `linear_ini` in annexes A and B

- *Train the model for 10 epochs using the initialization methods above and record the average loss measured on the training data at the end of each epoch (10 values for each setup).*

  We show in Fig. 1 the mean loss for each initialization.

  Comments :

  - The model with the **zero** initialization does not better than classifying randomly the MNIST digits, as indicated by the accuracy of both the training and validation sets,

    $$\text{acc}_v = 10.64\%$$
    $$\text{acc}_t = 11.36\%.$$

  - The model with the **normal** initialization does learn properly the training set, but fails to generalize as indicated by the large mean loss (small accuracy) for the validation set.

    $$\text{acc}_v = 93.13\%$$
    $$\text{acc}_t = 99.23\%.$$

  - The model with the **glorot** initialization has the best performance. Both the mean loss is the lowest and the accuracy the largest for both the training and validation sets. This comes with no surprise as this method is designed to set the weights variance such that the signal stays in a reasonable range through many layers. Specifically, the glorot initialization is such that the variance of the input and output are equal. See problem 7 of the theoretical part of the assignment.

    $$\text{acc}_v = 96.06\%$$
    $$\text{acc}_t = 96.06\%.$$

    We observed (see programming in Keras), that Normal initialization with standard deviation equal to 0.05 (the default Keras value) instead of to 1 provides equivalent results as Glorot initialization (learning curves coincide, and numerical values are very close).

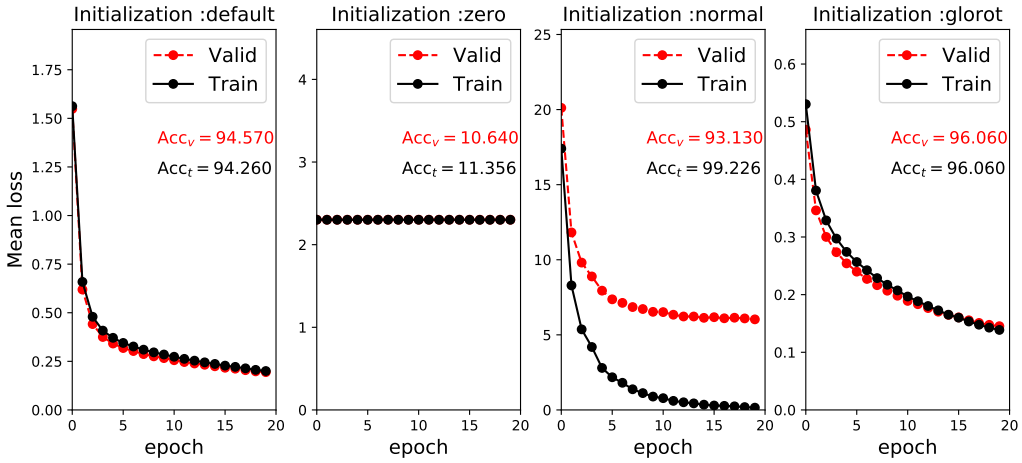

Figure 1: Loss as a function of epochs for four different initialization, i.e. default, zero, normal and glorot.

.

- The model with the **default** initialization is reasonably good compare to other techniques. It's unclear though which default initialization is used for a linear layer as it depends on the size of the layer [1].
- General note : the accuracy levels indicated above were computed after 10 epochs.

1.3 Learning curves

1. *Find out a combination of hyper-parameters (model architecture, learning rate, non-linearity, etc.) such that the average accuracy rate on the validation set (r^{valid}) is at least 97%.*

   We used a systematic approach in order to find the ideal hyperparameters. Specifically, we sampled random values for h_1, h_2, the batch size, the initial learning rate and the initialization method. We then searched for the hypeparameters that had the best accuracy.

   We obtained :

   - $h_1 = 787$
   - $h_2 = 437$
   - Batch size = 77

[1] https://discuss.pytorch.org/t/whats-the-default-initialization-methods-for-layers/3157

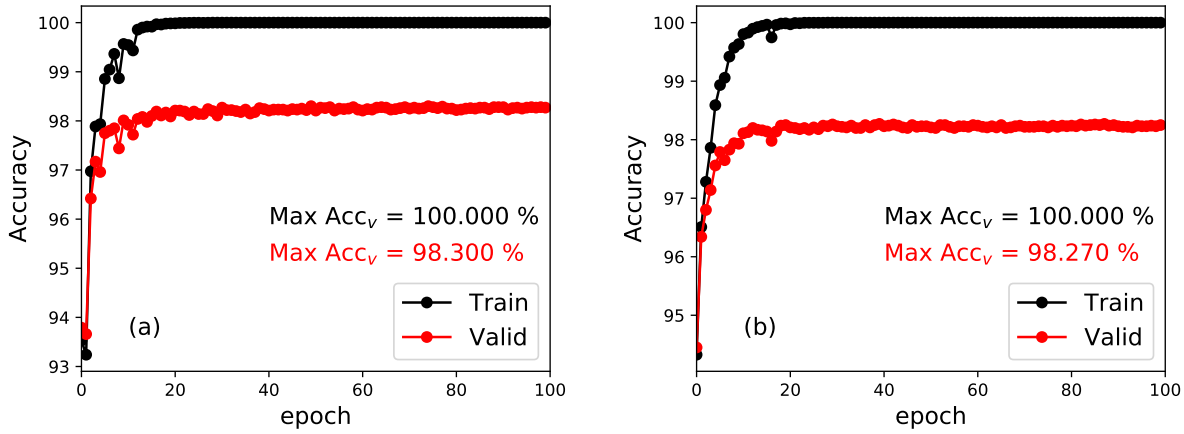

Figure 2: (a): Accuracy as a function of epoch for the problem in hand. We used the glorot weights initialization method. The chosen hyper-parameters are $h_1 = 787$, $h_2 = 437$, batch-size $= 77$, $l_r = 0.0866$ (with an exponential decay). The maximum accuracies for the training and validation are indicated. (b): Same as (a), but for twice the number of parameters.

.

- $lr_0 = 0.0886$, with an exponential decay.
- Initialization method = 'glorot'

2. *Plot a figure of learning curves with the x-axis being the training time and the y-axis being the accuracy. The plot should include the accuracy measured on both training and validation sets at the end of each epoch.*

   We show in 2 the learning curves for the optimal hyper-parameters for which the average accuracy rate is at least 98.3%.

3. Train the model for 100 epochs.

   –

4. *Train a model for 100 epochs with a capacity twice superior as that the for the previous measurements, i.e.the model that has generated the results in Fig. 2(a).*

   –

5. Doubling the number of parameters increases the capacity of the model. The main effects are: (1) the bias decreases and (2) the variance increases. As presented in fig.2(b), the result is almost identical as that the result in presented in fig.2(a). The maximum accuracy is slightly smaller for the model with a double capacity. It is important to note that we trained a model with 3x times the the validation accuracy was about 98.5%.

Table 1: Generalization gap for train dataset of various size N_{train}.

N_{train}	500	1000	2500	5000	50000
	14.90	11.50	8.10	5.83	1.83
	14.73	11.67	8.14	5.84	1.68
	14.78	11.69	8.05	5.96	1.74
	14.72	11.60	8.26	5.86	1.76
	14.98	11.60	8.23	5.77	1.73
Average	14.82	11.60	8.16	5.85	1.75
Standard deviation	0.10	0.073	0.078	0.062	0.048

1.4 Training Set Size, Generalization Gap, and Standard Error

We present in the table 1 the generalization gap defined as

$$G_a = r^{(train)} - r^{(test)}, \tag{1.1}$$

where $r^{(train)}$ and $r^{(test)}$ are evaluated at the epoch for which $r^{(valid)}$ is maximal.

Our main results may be explained as follow : The generalization gap means and standard deviations decrease with the size the training data set (D_{train}).

Changing the size of the D_{train} has two implications

1. Larger D_{train} is, the more likely the solution will approach the true distribution of digits in MNIST dataset that is, *i.e.* to reduce the *bias*.

   $$\begin{aligned} a = 0.01 \rightarrow D_{\text{train,valid,test}} &: \ 20500 \text{ data} \\ a = 0.1 \rightarrow D_{\text{train,valid,test}} &: \ 25000 \text{ data} \\ a = 1 \rightarrow D_{\text{train,valid,test}} &: \ 70000 \text{ data.} \end{aligned}$$

   To emphasize our argument, the reader should appreciate that, for $a = 0.01$, only 500 data points were used for training in order obtain the best representation for the 10 000 data points in the test data set. Therefore, it comes with no surprise that the generalization gap mean decreases with D_{train}.

   Further, having small D_{train} implicates that the given data subsets may not represent the same distribution of digits. This explains why the standard deviation decreases with D_{train}.

2. As the batch size is set constant in our experiment, the size of D_{train} dictates the number of batches and, consequently, of the model parameter updates. Thus, the gradient descent in the parameter space is more optimal for larger D_{train} and fixed

batch size. In fact, at each epoch, the model *estimates* the true loss before updating the parameters. The estimate thus obtained will approach asymptotically the true loss as the the number of batches increases. This is a fundamental consequence of the central limit theorem.

Problem 2

2.1 Building the Model

For the three processing procedures listed below, do a quick hyper-parameter search for the learning rate. Plot the accuracy on the training and test set for 20 epochs.

- **No preprocessing**
- **tf-idf**
- **Standardization**

We used a systematic approach to estimate the optimal set of hyperparameters, namely the `batch_size` and the `learning_rate`. We created a set of hyperparameters sampled randomly in the interval of `batch_size` $= [20, 300]$ and `lr_0` $= [0.0001, 0.01]$ for all three preprocessing procedures. We show in Fig. 3 the random set of these hyperparameters color coded. We could have searched for narrower intervals close to the optimal solutions, but we believe it is just enough for a 'quick hyper-parameter search'.

We used these hyperparameters to compare with the other preprocessing methods described above. We present our results in the Fig.4, where the accuracy is plotted with respect to the number of epochs.

1. *Briefly discuss the results[...]*

   As presented in Fig.4, the `tf_idf` preprocessing method surpasses the others. This is also true for any set of hyperparameters.

$$\texttt{No processing} = 82.4\%$$
$$\texttt{TF IDF} = 89.9\%$$
$$\texttt{Standardization} = 85.4\%$$

2. *[...] and answer the following points in a few lines each:*

   (a) *Could the same learning rate could be used for all of the models? Why, or why not?*

   The challenge in optimizing an objective function with sparse data is that each intermediate solution generated during the stochastic gradient descent will also be sparse. Therefore, this method will have a high variance due to heterogeneity in feature sparsity and the optimization problem will certainly be sensitive to the learning rate.

   We present in Fig. 5 the accuracy as a function of epochs for various learning rates. The batch size was set constant to 150 for all three methods. Our results suggest

that it is possible to use the same learning rate for all three models. However, the optimal learning rate is not the same for the preprocessing methods.

For a small learning rate, the validation accuracy increases monotonically, but very slowly which is expected. There is clearly an ideal learning rate for the three methods, that is around $\mathrm{lr}_0 \simeq 3 \times 10^{-3}$, where the validation accuracy in the maximum. As the learning rate increases, the validation accuracy becomes unstable, which is also to be expected.

It is important to note that we gradually decreased the learning rate after each epoch. See AnnexeB.2 for more details.

(b) *What might have happened if $\epsilon = 0$ for the Standardization preprocessing? Why? Besides modifying $\epsilon = 0$, how else could we deal with this problem?*

Setting $\epsilon = 0$ increases the risk for underflowing, that is rounding to zero numbers near zero. In fact, the variance of sparse data can be very small and, therefore, trigger underflowing.

Another way to get around this problem is to work in the log space instead, that is to compute the log of the standardized data. More precisely, we have that

$$\begin{aligned} d_j^i &= \frac{d_j^i - \mu_j}{\sigma_j} \\ \log d_j^i &= \log\left[\frac{d_j^i - \mu_j}{\sigma_j}\right] \\ \log d_j^i &= \log\left(d_j^i - \mu_j\right) - \log\left(\sigma_j\right) \\ d_j^i &= \exp\left[\log\left(d_j^i - \mu_j\right) - \log\left(\sigma_j\right)\right] \end{aligned} \tag{2.1}$$

(c) *What advantage does tf-idf has over basic word count approach that could help the model learn better?*

The term frequency (TF) rewards words with high occurence in a document, i.e. complexe or rare words. This is in strong contrast with the inverse document frequecy term (IDF) which penalize words appearing many times in a document such as 'and', 'or', 'is', 'the', etc. Therefore, by combining TF and IDF enables to select discriminative words in a document for proper classification.

2.2 Variance in training

To investigate the variance of the training loss with respect to the batch size, we took few precautions in order to properly make the comparison:

- As advised, we record the training loss after each parameter update,
- The learning rate was set to 0.2 for both models with batch size of 1 and 100.

We present in Fig. 6 the training loss and accuracy plotted with the gradient updates for batch size of 1 and 100.

For the model with batch size = 100, the training loss decreases with the number of updates. Clearly, the model learns properly. This is in strong contrast with the model with batch size of 1, which does learn very little. See below for more details.

Answer the following questions:

1. *Given that the two models had updates of the same magnitude, how do you explain the variance and the difference in the loss?*

   For the stochastic gradient descent, the parameter updates are defined as

   $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \left[\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f\left(\mathrm{x}^{(i)}; \boldsymbol{\theta} \right), \mathrm{y}^{(i)} \right) \right] \tag{2.2}$$

   where ϵ is the learning rate which is multiplied by the gradient of the average loss of the m samples of a given batch. The average loss of a batch correspond to an unbiased **estimate** of the gradient. Of course, large batches provide a more accurate estimate of the gradient.

   Suppose x_i is an i.i.d sample of N random variables of expectation μ and variance σ^2. Then, the expectation and the variance of the sample mean are given by,

   $$\begin{aligned} \mathbb{E}\left(\bar{\mathrm{x}}\right) &= \mu \\ \mathrm{Var}\left(\bar{\mathrm{x}}\right) &= \sigma^2 / N. \end{aligned} \tag{2.3}$$

   Therefore, the variance of the empirical risk for a batch size of 1 is 100 times larger than that of a batch size of 100.

   The accuracy for the model with batch size of 1 is about 6.8%[2], which means that the classification made by our model is just slightly better than random. This means that the model hardly learns in one epoch, as the parameter updates of the model is determined with too much variability. This is in strong contrast with the model of batch size of 100 for which the accuracy is $\simeq 99.9\%$ after 45 epochs.

[2]We obtained 10% with Keras.

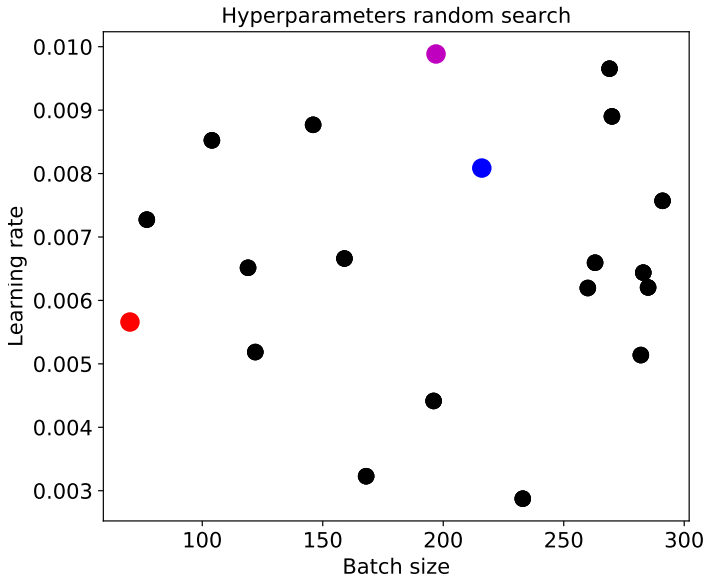

Figure 3: Random set of hyperparameters. We computed the validation accuracy for the three preprocessing method. The red, blue and magenta circles represent the optimal solutions for the no processing, tf-idf and standardization preprocessing methods respectively. Those solutions are plot in Fig. 4.

.

2. *If we could only have a minibatch of size 1, what technique could we use to reduce the variance during training? Explain the intuition behind it.*

   In this case, we would set a relatively small learning rate in order to maintain the stability of the training process. This is to counterbalance the effect of having an estimated gradient with a large variance. Of course, we should also use a large number of epochs to ensure convergence.

   The intuition behind is that, for batch size of 1 the weights are updated after each instance. This implies that some updates may be wrong, but at least the next update will be taken from that new place.

   Note : We refer the reader to section 8.1.3, p. 279 : *Small batches can offer a regularizing effect (Wilson and Martinez, 2003),perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batchsize might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient. The total runtime can be very high as a result of the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.*

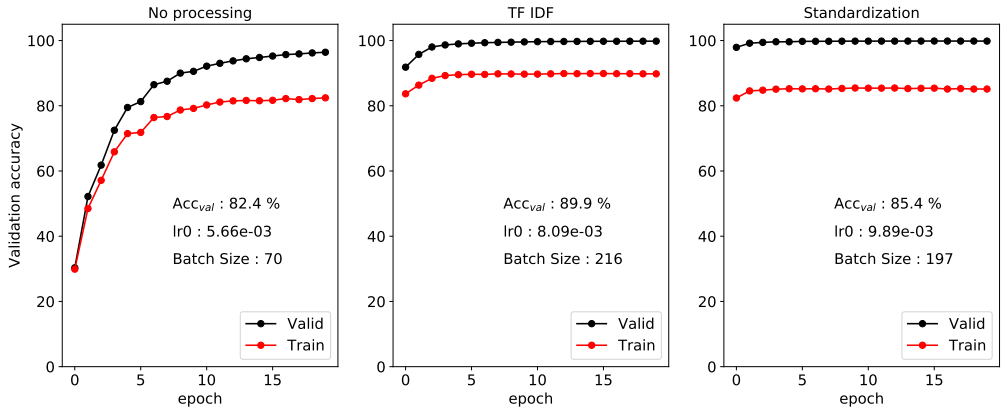

Figure 4: Validation accuracy as a function of epoch for the problem in hand. We used the `glorot` initialization method for the linear layer. The hyper-parameters for each preprocessing technic are indicated in the panels.

.

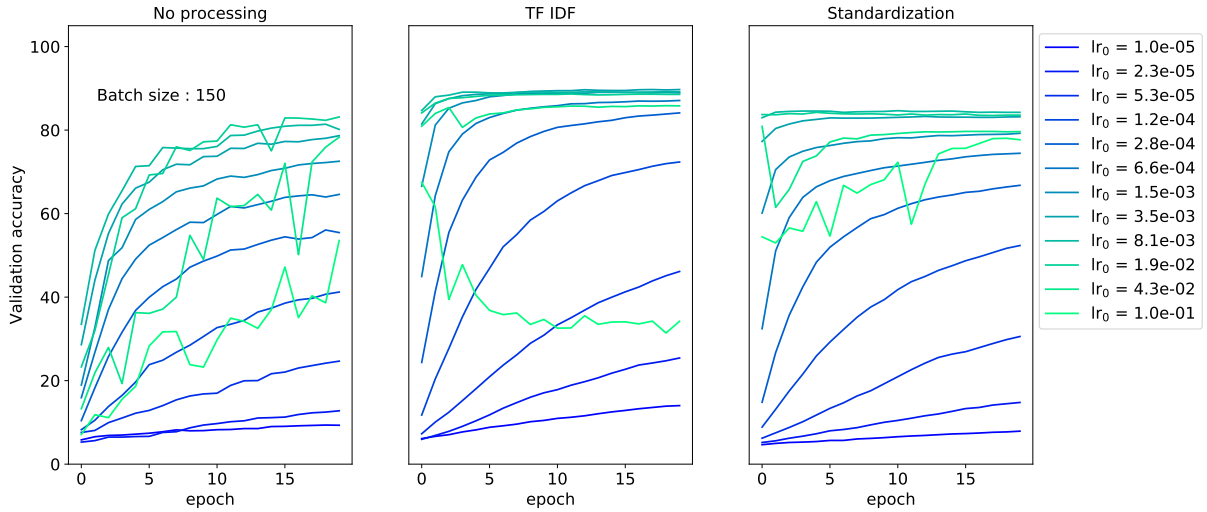

Figure 5: Validation accuracy as a function of epoch for various learning rates. The batch size was set constant, *i.e.* `batch_size` = 150.

.

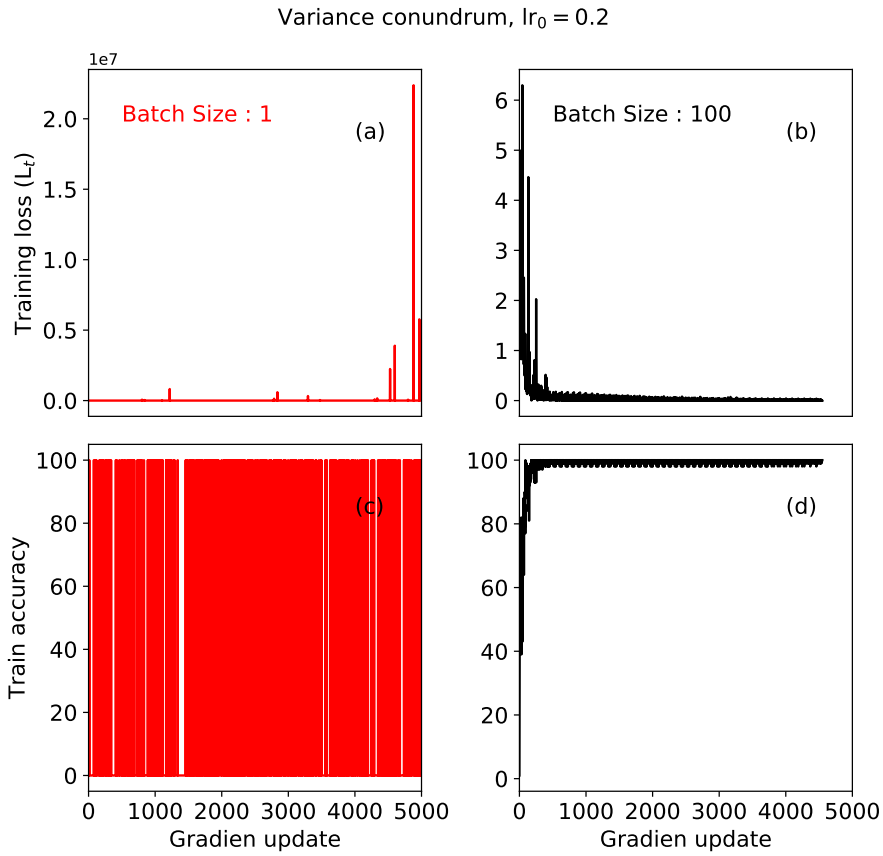

Figure 6: (a-b) Training loss and accuracy as a function of gradient updates.

.

Appendice A : Model - code

```
class MLPLinear(nn.Module):
    def __init__(self, dimensions, cuda):
        super(MLPLinear, self).__init__()
        self.h0 = dimensions[0]
        self.h1 = dimensions[1]
        self.h2 = dimensions[2]

        self.fc1 = torch.nn.Linear(self.h0,self.h1)
        self.fc2 = torch.nn.Linear(self.h1,self.h2)
        self.relu = nn.ReLU()

        self.criterion = nn.CrossEntropyLoss()

    def initialization(self,method):
        self.fc1 = linear_ini(self.fc1,method)
        self.fc2 = linear_ini(self.fc2,method)

    def input(self,x,y):
        if cuda :
            x = Variable(x.cuda())
            y = Variable(y.cuda())
        else:
            x = Variable(x)
            y = Variable(y)
        return x,y
    def preprocessing():
        pass

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return  out
```

Appendice B : various modules - code

B.1 Linear layer initialization

```
def linear_ini(LL,initialization):
    '''
    inputs : linear layer (LL) and the initialization
    output : linear layer with the chosen initialization
    '''
    if initialization == 'zero':
        LL.weight.data = nn.init.constant(LL.weight.data, 0)
        LL.bias.data = nn.init.constant(LL.bias.data, 0)

    if initialization == 'normal':
        LL.weight.data = nn.init.normal(LL.weight.data, 0,1)
        LL.bias.data = nn.init.constant(LL.bias.data, 0)

    if initialization == 'glorot':
        LL.weight.data = nn.init.xavier_uniform(LL.weight.data, gain=1)
        # that is important, see paper.
        LL.bias.data = nn.init.constant(LL.bias.data, 0)
    if initialization == 'default':
        pass
    return LL
```

B.2 Learning rate - exponential decay

```
def adjust_lr(optimizer, epoch, total_epochs):
    lr = lr0 * (0.36 ** (epoch / float(total_epochs)))
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
    return lr
```

Annexe C : Training - code

```
batch_size = 100
num_epochs = 20

initialization_method = ['none','zero','normal','glorot']

n = len(initialization_method)
m = num_epochs
loss_train = np.empty([n,m])
loss_valid = np.empty([n,m])
acc_train = np.empty([n,m])
acc_valid = np.empty([n,m])

train_batch = torch.utils.data.DataLoader(data_utils.TensorDataset(train_data,
    train_labels), batch_size=batch_size, shuffle=True)
valid_batch = torch.utils.data.DataLoader(data_utils.TensorDataset(valid_data,
    valid_labels), batch_size=batch_size, shuffle=False)
test_batch = torch.utils.data.DataLoader(data_utils.TensorDataset(test_data,
    test_labels), batch_size=batch_size, shuffle=False)

for i, method in enumerate(initialization_method):
    model = model_architecture([h0,h1,h2,h3],method)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    print('Initialization method : '+method)
    print('--------------------------------')
    for j in range(num_epochs):
        model_loss = 0
        for batch_idx, (x,y) in enumerate(train_batch):
            yy = y
            x = Variable(x.view(-1,784))
            y = Variable(y)

            optimizer.zero_grad()
            loss = criterion(model(x), y)
            model_loss += loss.data[0]
            loss.backward()
            optimizer.step()

        loss_valid[i,j] = batch_loss(valid_batch,model,criterion)
        loss_train[i,j] = batch_loss(train_batch,model,criterion)
        acc_valid[i,j]  = prediction(valid_batch,model)
        acc_train[i,j]  = prediction(train_batch,model)
        print('Epoch #'+str(j)+', Train loss = '+str(loss_train[i,j])+', Valid loss =
    '+str(loss_valid[i,j]))

    print('Train accuracy = '+str(prediction(train_batch,model))+'%')

print('done!')
```