ECE 285 – MLIP – Assignment 1 Backpropogation

Written by Sneha Gupta, Shobhit Trehan and Charles Deledalle. Last updated: April 2, 2019.

This assignment focuses on multiclass classification on the MNIST Dataset. You will learn to develop a simple implementation for artificial neural networks and backpropagation algorithm. The MNIST Dataset consists of labeled images of handwritten digits from 0 to 9. Each image is 28 by 28 pixels (784 pixels). Read about the dataset here http://yann.lecun.com/exdb/mnist/.

1 Artificial Neural Networks (ANNs) and backpropagation?

We will classify images into ten digits by using an artificial neural network. Let $\boldsymbol{x} \in \mathbb{R}^{784 \times N}$ be a collection of N input images (stacked in columns) and $\boldsymbol{y} \in [0,1]^{10 \times N}$ be the collection of our class predictions made for all images and encoded by so-called one-hot codes meant to approximate class probabilities, *i.e.*:

$$y_{i,j} \approx \mathbb{P}(\text{the } j\text{-th image in } \boldsymbol{x} \text{ represents the digit } i), \quad \text{for } 0 \leq i \leq 10 \text{ and } 0 \leq j \leq N.$$

A neural network makes the prediction $\boldsymbol{y}$ from the collection $\boldsymbol{x}$ of images by forward propagation which is defined recursively as follows

$$\boldsymbol{y} = g_k(\boldsymbol{a}_k) \quad \text{with} \quad \boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k \quad \text{with} \quad \boldsymbol{h}_0 = \boldsymbol{x}$$

with k the layer index, g_k a so-called activation function, $\boldsymbol{h}_k$ an array of N hidden feature vectors (also stacked in columns), $\boldsymbol{a}_k$ an array of N vectors of activations (also called potentials), $\boldsymbol{W}_k$ a matrix of synaptic weights and $\boldsymbol{b}_k$ a vector of biases. We will use a shallow neural network meaning that we will consider $k = 2$. The hidden layer will use `ReLU` activation function and the output layer will use `Softmax` (see definitions in Section 4).

The matrices $\boldsymbol{W}_k$ and vectors $\boldsymbol{b}_k$ will be learned/estimated from the MNIST training set by using logistic regression. This consists in minimizing the cross-entropy loss, between the collection $\boldsymbol{d}$ of desired one-hot codes ($d_{ij} = 1$ if the j-th image represents the digit i, 0 otherwise) and the prediction $\boldsymbol{y}$, which is defined as follows

$$E = -\sum_{j=1}^{N} \sum_{i=1}^{10} d_{ij} \log y_{ij} \ .$$

The optimization will be performed by gradient descent with backpropagation that can be implemented iteratively, for $t = 0, 1, \ldots$, and some initializations $\boldsymbol{W}_k^0$ and $\boldsymbol{b}_k^0$, as:

$$\begin{aligned}
& \boldsymbol{W}_k^{t+1} = \boldsymbol{W}_k^t - \gamma \nabla_{\boldsymbol{W}_k} E^t \\
& \boldsymbol{b}_k^{t+1} = \boldsymbol{b}_k^t - \gamma \nabla_{\boldsymbol{b}_k} E^t \\
\text{with} \quad & \nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T \\
\text{and} \quad & \nabla_{\boldsymbol{b}_k} E = \boldsymbol{\delta}_k \boldsymbol{1}_N \\
\text{and} \quad & \boldsymbol{\delta}_k = \left[\frac{\partial g_k(\boldsymbol{a}_k)}{\partial \boldsymbol{a}_k} \right]^T \times \boldsymbol{e}_k \quad \text{where} \quad \boldsymbol{e}_k = \begin{cases} \nabla_{\boldsymbol{y}} E & \text{if } k \text{ is an output layer} \\ \boldsymbol{W}_{k+1}^T \boldsymbol{\delta}_{k+1} & \text{otherwise} \end{cases}
\end{aligned}$$

where the notations are:

- t: iteration index of gradient descent,
- γ: learning rate of gradient descent,
- $\boldsymbol{W}_k$: matrix of weights at layer k,
- $\boldsymbol{b}_k$: vector of biases at layer k,
- $\boldsymbol{x}$: matrix with the N training input vectors in columns,
- $\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$: matrix with all hidden outputs at layer k in columns,
- $\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$: matrix with all weighted sums at layer k in columns,
- g_k: activation function at layer k,
- $\mathbf{1}_N$: column vector of size N containing only ones.

In the following, for simplicity, we will drop the indices k and t.

This was a very short description but neural networks and backpropagation will be studied more in depth during the class (Chapter 2).

2 Getting started

First of all, connect to DSMLP (`ieng6` server), start a *pod* and connect to your Jupyter Notebook from your web browser (go back to Assignemnt 0 for more details). Create a new notebook `assignment1.ipynb` and import

```
import numpy as np
from matplotlib import pyplot
```

For the following questions, please write your code and answers directly in your notebook. Organize your notebook with headings, markdown and code cells (following the numbering of the questions).

3 Read MNIST Data

The MNIST dataset is already installed on DSMLP in the directory `/datasets/MNIST/`. The data can be loaded using a small dedicated package `/datasets/ee285s-public/MNISTtools.py`. From your terminal, create a symbolic link on this package into your working/current directory

```
$ ln -s /datasets/ee285s-public/MNISTtools.py .
```

Back to your Jupyter Notebook, you can now import its functions as:

```
import MNISTtools
```

This package contains two functions: `load` and `show`. You can display their description using the Python command `help`:

```
help(MNISTtools.load)
help(MNISTtools.show)
```

You can also type `MNISTtools.load` and press `Shift + Tab`.

1. Using `MNISTtools.load`, store the images and labels from the training datasets into two variables, respectively, `xtrain` and `ltrain`. What are the shapes of both variables? What is the size of the training dataset? What is the feature dimension?

2. Display the image of index 42 and check that its content corresponds to its label.

3. What is the range of `xtrain` (minimum and maximum values)? What is the type of `xtrain`?

4. Create a function

   ```python
   def normalize_MNIST_images(x):
   ```

   that takes a collection of images (such as `xtrain`) and returns a modified version in the range $[-1, 1]$ of type `float32`. Update `xtrain` accordingly.

   Hint: convert a Numpy array `x` from `int8` to `float32` using `x.astype(np.float32)`.

5. Using integer array indexing, complete the following function

   ```python
   def label2onehot(lbl):
       d = np.zeros((lbl.max() + 1, lbl.size))
       d[COMPLETE, np.arange(lbl.size)] = 1
       return d
   ```

   such that `dtrain = label2onehot(ltrain)` will create a Numpy array `dtrain` of one-hot codes stacked in columns (with shape `(10, 60000)`). Make sure that the one-hot code `dtrain[:,42]` corresponds to `ltrain[42]`.

6. Complete the following function

   ```python
   def onehot2label(d):
       lbl = d.argmax(axis=COMPLETE)
       return lbl
   ```

   such that `ltrain == onehot2label(dtrain)`.

4 Activation functions

For our digit multiclass classification, we will use the **softmax** activation function for the output layer (with 10 units). Given a vector $\boldsymbol{a} \in \mathbb{R}^{10}$ (obtained by forward propagation), `softmax` will output a vector $\boldsymbol{y} \in [0,1]^{10}$, where each element y_i represents the probability that $\boldsymbol{x} \in \mathbb{R}^{784}$ is in class i. The relation between $\boldsymbol{a}$ and $\boldsymbol{y}$ is given for all $i \in [1, 10]$ by

$$y_i = g(\boldsymbol{a})_i = \frac{\exp(a_i)}{\sum_{j=1}^{10} \exp(a_j)}$$

7. When using exponential functions, one should make sure that the input won't be too large or you may observe numerical issues (apparition of `Inf` and subsequently of `NaN`). A simple trick to get rid of this problem is based on the following observation

$$y_i = g(\boldsymbol{a})_i = \frac{\exp(a_i - M)}{\sum_{j=1}^{10} \exp(a_j - M)} \quad \text{where} \quad M = \max_{j=1\ldots10} a_j$$

As all inputs of the exponentials will be smaller than 0 and one of them will be exactly 0, you won't encounter numerical issues. Based on this trick, create a function

```python
def softmax(a):
```

that returns an array whose columns are the 60,000 predictions $\boldsymbol{y}$ from an array whose columns are the 60,000 vectors $\boldsymbol{a}$. *Hint: use Broadcasting and the methods `.max(axis=0)` and `.sum(axis=0)`.*

8. Show that $\dfrac{\partial g(\boldsymbol{a})_i}{\partial a_i} = g(\boldsymbol{a})_i(1 - g(\boldsymbol{a})_i)$.

9. Show that $\dfrac{\partial g(\boldsymbol{a})_i}{\partial a_j} = -g(\boldsymbol{a})_i g(\boldsymbol{a})_j$ for $j \neq i$.

10. Given a vector $\boldsymbol{e} \in \mathbb{R}^{10}$ (obtained during backward propagation), backprop algorithm have to compute

$$\boldsymbol{\delta} = \left[\frac{\partial g(\boldsymbol{a})}{\partial \boldsymbol{a}}\right]^T \times \boldsymbol{e}$$

where $\times$ denotes here the matrix vector product. The matrix $\frac{\partial g(\boldsymbol{a})}{\partial \boldsymbol{a}}$ is the Jacobian matrix defined as

$$\frac{\partial g(\boldsymbol{a})}{\partial \boldsymbol{a}} = \begin{pmatrix} \frac{\partial g(\boldsymbol{a})_1}{\partial a_1} & \frac{\partial g(\boldsymbol{a})_1}{\partial a_2} & \dots & \frac{\partial g(\boldsymbol{a})_1}{\partial a_{10}} \\ \frac{\partial g(\boldsymbol{a})_2}{\partial a_1} & \frac{\partial g(\boldsymbol{a})_2}{\partial a_2} & \dots & \frac{\partial g(\boldsymbol{a})_2}{\partial a_{10}} \\ \vdots & & & \\ \frac{\partial g(\boldsymbol{a})_{10}}{\partial a_1} & \frac{\partial g(\boldsymbol{a})_{10}}{\partial a_2} & \dots & \frac{\partial g(\boldsymbol{a})_{10}}{\partial a_{10}} \end{pmatrix}$$

When the activation is element-wise ($g(\boldsymbol{a})_i = g(a_i)$), the Jacobian is diagonal and the update of $\boldsymbol{\delta}$ is simply an element wise product between $g'(a_i)$ and e_i (see Chapter 2). But `softmax` is not an element-wise function and its Jacobian is not diagonal. Nevertheless, it enjoys some interesting properties. From the previous question, deduce that the Jacobian of `softmax` is symmetric and that

$$\boldsymbol{\delta} = g(\boldsymbol{a}) \otimes \boldsymbol{e} - \langle g(\boldsymbol{a}), \boldsymbol{e} \rangle \, g(\boldsymbol{a})$$

where $\otimes$ is the element-wise product. (*Remark that the expression is not that much different from the derivative of the sigmoid logistic function $g'(a) = g(a)(1 - g(a))$. This is because softmax is a multivariate generalization of the sigmoid logistic function.*)

Based on this formula, write a function

```python
def softmaxp(a, e):
```

that, given an array whose columns are the vectors $\boldsymbol{a}$ and an array whose columns are vectors $\boldsymbol{e}$, returns an array whose columns are the vectors $\boldsymbol{\delta}$.

Hint: call `softmax(a)`.

11. Since the Jacobian is symmetric, the update of $\boldsymbol{\delta}$ corresponds to the directional derivative of g at point $\boldsymbol{a}$ in the direction $\boldsymbol{e}$:

$$\boldsymbol{\delta} = \frac{\partial g(\boldsymbol{a})}{\partial \boldsymbol{a}} \times \boldsymbol{e} = \lim_{\epsilon \to 0} \frac{g(\boldsymbol{a} + \epsilon \boldsymbol{e}) - g(\boldsymbol{a})}{\epsilon}$$

Based on this formula, you can check your implementation by numerical approximations (finite difference). Complete the following script to check your function `softmaxp` as follows

```
eps        = 1e-6                       # finite difference step
a          = np.random.randn(10, 200)   # random inputs
e          = np.random.randn(10, 200)   # random directions
diff       = softmaxp(a, e)
diff_approx = COMPLETE
rel_error   = np.abs(diff - diff_approx).mean() / np.abs(diff_approx).mean()
print(rel_error, 'should be smaller than 1e-6')
```

12. For the hidden layers, we will be using $\texttt{ReLU}(\boldsymbol{a})_i = \max(a_i, 0)$. Write two functions:

```
def relu(a):
   COMPLETE

def relup(a, e):
   COMPLETE
```

implementing ReLU and its directional derivative. Check your implementation based on numerical approximations.

5 Backpropagation

We are now ready to implement our shallow network. The (single) hidden layer between the input and output will consist of $N_h = 64$ units with the `relu` activation function. The output layer will consist of $N_o = 10$ units with the `softmax` activation function. The input layer is of dimension $N_i = 784$.

13. Use the following function to create/initialize your shallow network as follows

```
def init_shallow(Ni, Nh, No):
    b1 = np.random.randn(Nh, 1)   / np.sqrt((Ni+1.)/2.)
    W1 = np.random.randn(Nh, Ni)  / np.sqrt((Ni+1.)/2.)
    b2 = np.random.randn(No, 1)   / np.sqrt((Nh+1.))
    W2 = np.random.randn(No, Nh)  / np.sqrt((Nh+1.))
    return W1, b1, W2, b2

Ni = xtrain.shape[0]
Nh = 64
No = dtrain.shape[0]
netinit = init_shallow(Ni, Nh, No)
```

These types of initializations are called He and Xavier initializations, respectively, and will be explained later during the class.

14. Complete the function `forwardprop_shallow` to evaluate the prediction of our initial network:

```
def forwardprop_shallow(x, net):
    W1 = net[0]
    b1 = net[1]
    W2 = net[2]
    b2 = net[3]
    a1 = W1.dot(x) + b1
    COMPLETE

    return y

yinit = forwardprop_shallow(xtrain, netinit)
```

that produces an array whose columns are vectors $\boldsymbol{y}$ corresponding to the predictions obtained for each column $\boldsymbol{x}$ given in arguments.

15. Complete the function `eval_loss`:

```
def eval_loss(y, d):
    COMPLETE

print(eval_loss(yinit, dtrain), 'should be around .26')
```

that given your predictions $\boldsymbol{y}$ and the desired one-hot codes $\boldsymbol{d}$, computes the average cross-entropy loss (averaged over both the training samples and the vector dimension). Recall that the cross-entropy for $K = 10$ classes of a vector $\boldsymbol{y}$ against a vector $\boldsymbol{d}$ is

$$E = -\sum_{i=1}^{10} d_i \log y_i$$

16. Complete the function `eval_perfs`:

```
def eval_perfs(y, lbl):
    COMPLETE

print(eval_perfs(yinit, ltrain))
```

that given your predictions $\boldsymbol{y}$ and the desired labels $\boldsymbol{lbl}$, computes the percentage of misclassified samples. Interpret the result.

Hint: use the function `onehot2label` and do not use loops.

17. Complete the following function `update_shallow`

```
def update_shallow(x, d, net, gamma=.05):
    W1 = net[0]
    b1 = net[1]
    W2 = net[2]
    b2 = net[3]
    Ni = W1.shape[1]
    Nh = W1.shape[0]
    No = W2.shape[0]

    gamma = gamma / x.shape[1] # normalized by the training dataset size

    COMPLETE
    return W1, b1, W2, b2
```

such that it performs one backpropagation update for your shallow neural network. Recall that the inputs $\boldsymbol{x}$ and $\boldsymbol{d}$ are going to be arrays with 60,000 columns corresponding to the vectors of images and the vectors of one-hot codes respectively. Show that

$$(\nabla_{\boldsymbol{y}} E)_i = -\frac{d_i}{y_i}$$

Hint: Takes inspiration of the code described in Chapter 2, slide 67. Use the functions `softmax`*,* `softmaxp` `relu` *and* `relup`*.*

18. Using `update_shallow`, complete the function `backprop_shallow`

```
def backprop_shallow(x, d, net, T, gamma=.05):
    lbl = onehot2label(d)
    for t in range(T):
        COMPLETE TO UPDATE NET
        COMPLETE TO DISPLAY LOSS AND PERFS
    return net
```

that performs T updates of the network and `print` the loss and the percentage of training errors at each iteration of backprop. Start testing it using $T = 2$ iterations

```
nettrain = backprop_shallow(xtrain, dtrain, netinit, 2)
```

When your code starts working, increase the number of iterations to $T = 5$, and if it is still working try $T = 20$. The loss and training errors should decrease (with some fluctuations).

What percentage of training errors do you reach? (Feel free to increase even more the number of iterations: with $T = 100$ you should reach about 13% of training errors)

19. Load the testing dataset into two variables `xtest` and `ltest`. What is the size of the testing set? Evaluate the performance of your network on the testing dataset.

20. We will now implement a variant of backpropagation based on stochastic/minibatch gradient descent (we will explain later this variant during the class). The algorithm is very similar to backprop but instead of updating the weights based on the 60,000 data points at once, we will partition our training sets on random blocks of size $B = 100$ (called minibatches) and update successively the weights and biases based on the error of each minibatch. The number of updates is then TN/B. An epoch corresponds to a succession of updates where all minibatches have been processed. The parameter T is then referred to as the number of epochs. Note that at each epoch we randomly resample new mini-batches. Using `update_shallow`, interpret each instruction and complete the function `backprop_minibatch_shallow`:

```
def backprop_minibatch_shallow(x, d, net, T, B=100, gamma=.05):
    N = x.shape[1]
    NB = int((N+B-1)/B)
    lbl = onehot2label(d)
    for t in range(T):
        shuffled_indices = np.random.permutation(range(N))
        for l in range(NB):
            minibatch_indices = shuffled_indices[B*l:min(B*(l+1), N)]
            COMPLETE TO UPDATE NET
        y = forwardprop_shallow(x, net)
        COMPLETE TO DISPLAY LOSS AND PERFS
    return net

netminibatch = backprop_minibatch_shallow(xtrain, dtrain, netinit, 5, B=100)
```

Hint: use integer array indexing.

21. Run `backprop_minibatch_shallow` for 5 epochs. Compare the performance of this new network on the testing dataset.

6 Bonus (optional and ungraded)

22. Play with the number of hidden units N_h, try different step sizes γ and minibatches of sizes B. Look at the training and testing errors. Interpret the results.

23. Write generic functions that can deal with an arbitrary number L of hidden layers.

24. Increase the number of hidden layers, e.g., use two hidden layers instead of one. Create a new architecture that uses two hidden layers of equal size and has approximately the same number of parameters as the previous network with one hidden layer. By that, we mean it should have roughly the same total number of weights and biases. Study the loss, training and testing errors vs. the number of epochs. Repeat with four, eight, sixteen, . . . layers.