METL – TP1
Neural Network Basics

Alexandre Kabbach
alexandre.kabbach@unige.ch

Paola Merlo
paola.merlo@unige.ch

20.02.2020

Evaluation: You are allowed an unlimited number of submissions in order to receive feedback. When you are satisfied with your work, you can ask for it to be graded. You can also ask for it to be graded upon a single submission, without receiving feedback. All your TPs must have been graded and must have received an average grade of at least 4/6 for you to register for the METL exam. Indicative deadline: March 4 2020 (this TP should take you two weeks).

1 Preliminary instructions

Mathematics as used in the field of machine learning is often filled with approximations or underspecifications which can hinder the comprehension of the notions at hand. The purpose of this TP is to make your life easier by first ***making the implicit explicit****. For our first TP, we go over some important mathematical notions regarding neural networks, as usually specified in the field, and rework them as cleanly as possible. To do so, we ask you to be as precise and clean as possible in all your mathematical developments and demonstrations. Notably:*

1. *Always specify the* ***domain*** *of the function at hand;*
2. *Always specify the* ***scope*** *of your variables;*
3. *Always specify the* ***nature*** *of your demonstration (by equivalence, reductio ad absurdum, proof by induction, etc.) and use the relevant mathematical symbols;*
4. *Always keep you demonstrations* ***short*** *and* ***simple****: limit verbosity, write down* ***all*** *the necessary steps, but* ***only*** *the necessary steps.*

2 Softmax

Consider the following (poorly defined) *softmax* function:

$$softmax(x) = \frac{e^x}{\sum_j e^{x_j}} \tag{1}$$

1. Cleanly redefine the softmax function;
2. Prove that softmax is invariant to constant offsets in the input, that is, for any input vector x and any constant c:

$$softmax(x) = softmax(x + c) \tag{2}$$

Note: In practice, we make use of this property and choose $c = -\max_i x_i$ when computing softmax probabilities for numerical stability, that is, we subtract the maximum element from all elements of x. Remember this in your implementation of softmax...

3. Given an input matrix of N rows and d columns, compute the softmax prediction for each row. Write your implementation in `softmax.py` and test your code by running:

   ```
   python3 softmax.py
   ```

   Note: The provided tests are not exhaustive. Later parts of the assignment will reference this code so it is important to have a correct implementation. Your implementation should also be efficient and vectorized whenever possible.

3 Gradient, Sigmoid, Forward, Backward

1. Derive the gradients of the sigmoid function and show that it can be rewritten as a function of the function value (i.e. in some expression where only $\sigma(x)$, but not x, is present). Assume that the input x is a scalar for this question. Recall the (yet again poorly defined) sigmoid function, which you will first properly redefine:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

2. Derive the gradient with respect to the inputs of a softmax function when cross entropy loss is used for evaluation, i.e. find the gradients with respect to the softmax input vector z, when the prediction is made by $\hat{y} = softmax(z)$. Recall the cross entropy function:

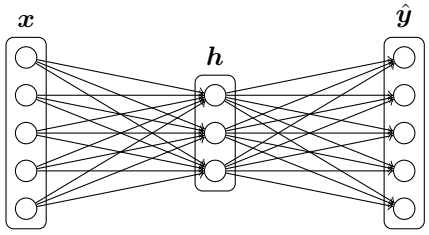
$$CE(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i) \tag{4}$$

   where y is the one-hot label vector and $\hat{y}$ is the predicted probability vector for all classes. *Hint: you may want to consider the fact that many elements of y are zeros, and assume that only the k-th dimension of y is one.*

3. Derive the gradients with respect to the inputs x of a one-hidden-layer neural network. That is, find $\frac{\partial J}{\partial x}$ where J is the cost/loss function of the neural network. Consider a neural network that employs a sigmoid activation function for the hidden layer and a softmax for the output layer. Assume the one-hot label vector to be y and the cost/loss function to be cross entropy. Recall that the forward propagation follows:

$$h = sigmoid(xW_1 + b_1) \qquad \hat{y} = softmax(hW_2 + b_2) \tag{5}$$

   You can denote the sigmoid gradient as $\sigma'(x)$ and define $z_2 = hW_2 + b_2$ and $z_1 = xW_1 + b_1$.



   Note that here we are assuming that the input vector (thus the hidden variables and output probabilities) is a row vector. When we apply the sigmoid function to a vector, we apply it to each of its elements. W_i and b_i $(i = 1, 2)$ are the weights and biases of the two layers.

Careful! Taking the sigmoid of a vector is once again an abuse of notation. In your calculation and code make sure that multiplying σ' on the left side is done using element-wise multiplication $\odot$ (in python, use $*$ instead of *np.dot* in $gradz_1$ of Q3.7).

4. How many parameters are there in this neural network, assuming the input to be D_x-dimensional, the output to be D_y-dimensional and there to be H hidden units?

5. Fill in the implementation for the sigmoid activation function and its gradient in `sigmoid.py` and test your implementation using:

   ```
   python3 sigmoid.py
   ```

   Again, thoroughly test your code as the provided tests may not be exhaustive.

6. To make debugging easier, we will now implement a gradient checker. Fill in the implementation of `gradcheck_naive` in `gradcheck.py` and test your code using:

   ```
   python3 gradcheck.py
   ```

   Hint: Use the centrale difference.[1] *Reminder: Why do we need to check the gradient?*[2]

7. Implement the forward and backward passes for a neural network with one sigmoid hidden layer. Fill in your implementation in `neural.py` and sanity check it by running:

   ```
   python3 neural.py
   ```

   For the backward propagation, consider the following gradients:

   - $grady = \frac{\partial CE(y,\hat{y})}{\partial z_2}$
   - $gradW_2 = \frac{\partial CE(y,\hat{y})}{\partial W_2}$
   - $gradb_2 = \frac{\partial CE(y,\hat{y})}{\partial b_2}$
   - $gradh = \frac{\partial CE(y,\hat{y})}{\partial h}$
   - $gradz_1 = \frac{\partial CE(y,\hat{y})}{\partial z_1}$
   - $gradW_1 = \frac{\partial CE(y,\hat{y})}{\partial W_1}$
   - $gradb_1 = \frac{\partial CE(y,\hat{y})}{\partial b_1}$

   To help you compute those values, rely on the enclosed *NN_foundations_notes.pdf* file summarized in Section 4 below.

4 Notes on gradients and backprop

In the above question (and in the TP2 that will follow) it is asked to compute the gradient of a scalar (the cost function) with respect to a matrix (of weights). This is at the core of the backpropagation algorithm. Recall the backpropagation algorithm where the weights W and the bias b are updated with respect to a cost function J given a learning rate α:

$$W = W - \alpha \frac{\partial J}{\partial W} \tag{6}$$

[1] https://math.stackexchange.com/questions/2120946/why-is-central-difference-preferred-over-backward-and-f
[2] http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization

$$b = b - \alpha \frac{\partial J}{\partial b} \tag{7}$$

The backpropagation algorithm therefore requires the scalar-by-matrix gradient $\frac{\partial J}{\partial W}$ to be of a size consistent with the size of W. Therefore, it is important to understand that the gradient matrix $\frac{\partial J}{\partial W}$ is actually an **abuse of notation** as it is understood as the following matrix, considering W to be a $n \times m$ matrix:

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \cdots & \frac{\partial J}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W_{n1}} & \cdots & \frac{\partial J}{\partial W_{nm}} \end{bmatrix} \tag{8}$$

Now to ease the computation we will introduce a vector $z = xW$ so that:

$$\frac{\partial J}{\partial W} = \frac{\partial z}{\partial W} \frac{\partial J}{\partial z} \tag{9}$$

(Note that the chain rule for the denominator layout notation is in reverse order, and that the aforementioned equation is also somehow an abuse of notation).
You should have shown in Q3.2 in the case where $J = CE(y, \hat{y})$ and $\hat{y} = softmax(hW_2 + b_2)$ and given $z = hW_2 + b_2$:

$$\frac{\partial J}{\partial z} = \frac{\partial CE}{\partial z} = \hat{y} - y \tag{10}$$

Note that if z is a vector of size $1 \times m$, $\hat{y}$ will also be a vector of size $1 \times m$ and so will be $\frac{\partial J}{\partial z}$. All we need to do now is to compute $\frac{\partial z}{\partial W} = \frac{\partial xW}{\partial W}$, which we do for all element of w_{ij} with:

$$z_k = \sum_{l=1}^{n} x_l W_{lk} \tag{11}$$

$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{n} x_l \frac{\partial W_{lk}}{\partial W_{ij}} \tag{12}$$

where z_k is the k^{th} column component of the row vector z and x is a $1 \times n$ row vector and W is a $n \times m$ matrix. Given that:

$$\frac{\partial W_{lk}}{\partial W_{ij}} = \begin{cases} 1 & \text{if } i = l \text{ and } j = k \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Let us detail the calculations:

$$z = [x_1 w_{11} + \ldots + x_n w_{n1}, \cdots, x_1 w_{1m} + \ldots + x_n w_{nm}] \tag{14}$$

$$\begin{aligned} z_1 &= x_1 w_{11} + \ldots + x_n w_{n1} = \sum_{l=1}^{n} x_l W_{l1} \\ &\ldots \\ z_m &= x_1 w_{1m} + \ldots + x_n w_{nm} = \sum_{l=1}^{n} x_l W_{lm} \end{aligned} \tag{15}$$

Try deriving with respect to w_{11} and w_{12}, etc. and you should find that $\frac{\partial z_k}{\partial W_{ij}} = x_i$.
Therefore, we can show that::

$$\forall (i,j) \in [1,n] \times [1,m] \quad \frac{\partial J}{\partial W_{ij}} = \frac{\partial z}{\partial W_{ij}} \frac{\partial J}{\partial z} = \sum_{k=1}^{m} \frac{\partial z_k}{\partial W_{ij}} \Big(\frac{\partial J}{\partial z} \Big)_k = x_i \Big(\frac{\partial J}{\partial z} \Big)_j \tag{16}$$

Recall that $\frac{\partial J}{\partial z}$ is a $1 \times m$ vector and that the chain rule for the denominator layout notation is in reverse order. We conclude that:

$$\frac{\partial J}{\partial W} = x^\top \frac{\partial J}{\partial z} \tag{17}$$

with the aforementioned abuse of notation (see eq. 8). You can double check consistency of dimensions to be sure ($x : 1 \times n$, $\frac{\partial J}{\partial z} : 1 \times m$ and therefore $\frac{\partial J}{\partial W} : n \times m$).