IST 597 Foundations of Deep Learning
Homework 2: Back-Propagation of Errors and Multilayer Perceptrons

This assignment is worth 15% of your grade for this class.

1 Introduction

In this assignment, you will implement and apply a simple variant of back-propagation of errors that you can use to learn multilayer perceptrons. Before starting, we strongly encourage to read Chapter 6 of the class textbook, *Deep Learning* [2]. While the chapter gives a full treatment of automatic differentiation, which is essentially what back-propagation of errors is, you will focusing on implementing reverse-mode differentiation for a chain-like computation graph, or feedforward network.

It is important that you understood the basic ideas presented in the previous assignment, "Regression & Gradient Descent", as you will effectively be extending what you originally wrote to now train more complex models (w.r.t. parameters as well as number of output/target variables). After developing your gradient-calculation algorithm and combining it with gradient descent, you will apply your work to the classical XOR problem. Finally, you will then attempt to fit a one and two hidden layer multilayer perceptron (MLP) to a version of the IRIS dataset.

The starter code and data for this assignment can be found at the following Github repository:

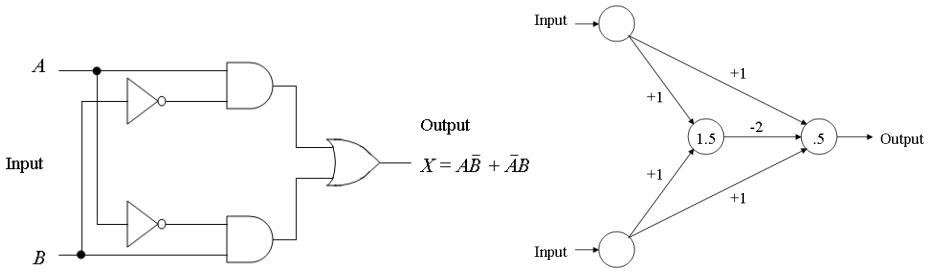
`https://github.com/ago109/IST597-Deep-Learning-Foundations.git`

Specifically, you will want to download the files found under `/problems/HW2/` directory. The data to be used for this assignment can be found inside the `/problems/HW2/data/` folder. Please consult HW #1 for directions on the necessary Python setup as well pointers to Python tutorials.

Make sure to add a comment to each script (or modify the file's header comment) that contains your name. You will notice that there is an empty folder, `/problems/HW2/out/`, which is where you will need to store your program outputs for each part of this assignment (i.e., plots).

For the questions you will be asked to answer/comment on in each section, please maintain a separate document (e.g., such as Microsoft Word) and record yours answers/thoughts there. Furthermore, while the Python scripts will allow you to save your plots to disk, please copy/insert the generated plots into your document again mapped to the correct problem number. For answers to various questions throughout, please copy the question(s) being asked and write your answer underneath (as well as the problem number).

Make sure you look for all of the `WRITEME` and `TODO` comments, as these will be the places you will need to write code in order to complete the assignment successfully (note that you may have to write more code than just in-place of the annotated comments, such as code for plot/figure generation, etc.). If you find yourself being clever and finding better ways to write the routines required, you must still replace the `WRITEME` comments with at least an explanation as to why you chose to not follow the design approach implied by the assignment. A general requirement for all sub-problems is that you will need to write code for the main training loop (whether in batch or mini-batch form). Do not also forget that you can also experiment with the initial conditions for all parameters (the parameter initializations) as well.



(a) The exclusive-or (XOR) digital circuit.

(b) An XOR perceptron gate.

Figure 1: In Figure 1a we see the full XOR gate's circuit diagram and in Figure 1b we see the same gate implemented as a hard-coded multilayer perceptron. The center-most unit of the perceptron-gate is hidden from outside influence and is connected only to input or output units. The value of 1.5 for the threshold of the internal unit ensures that it will be turned on only when both input units are on. The value of 0.5 for the output unit ensures that it will turn on only when the net positive input is greater than 0.5. The weight of -2 from the hidden unit to the output unit ensures that the output unit will *not* turn on when both inputs are on.

2 Problem #1: Back-Propagation of Errors (50 points)

In order to learn a parametrized model, as we saw in homework #1, we need a method for calculating partial derivatives of our loss with respect to model parameters and an update rule that alters model parameters using the computed partial derivatives. For this problem, you will implement back-propagation of errors to calculate the gradients needed. You will then use the gradients you calculate to update parameters via mini-batch gradient descent.

The logical exclusive-OR (XOR) problem is a classical problem that is linearly inseparable and useful for motivating the need for hidden processing units. It is fully specified via the following truth table:

$\mathbf{in}_1$	$\mathbf{in}_2$	**out**
0	0	0
0	1	1
1	0	1
1	1	0

You will work with this classical problem (and solve it) as you develop your basic softmax regressor and MLP models. In Figure 1, you can see the classical circuit depicted as well as a non-learnable/hard-coded MLP that can directly solve the problem.

2.1 Problem #1a: Softmax Regression & the XOR Problem

Implementing back-propagation, or the application of the chain rule (from differential calculus) to a computation graph, is rather straightforward once a loss function (as well as some architectural details, like the choice of activation function for MLPs) has been defined. We will focus on a simple computation graph, particularly one with no branches (or a chain of successive operations following from one input to one output). To begin, we will first design the simplest possible MLP–one with no hidden units, or simply a perceptron-based classifier.

Following up to the last assignment, you will implement a simple multi-class regression model known as softmax regression (or multinoulli regression[1]). Eventually, we will extend this model to sport hidden units, building a deeper model (or MLP). Our parameters are housed in the construct $\Theta = \{W, b\}$ (where W is a weight matrix of dimensions $(d \times k)$ and b is a bias vector of dimensions $(1 \times k)$, where d is the number of observed variables and k is the number of classes we want to predict). Our dataset is $\mathbf{X}$ which is of dimension $(N \times d)$ (N is the number of samples) and y (which

[1]The multinoulli distribution (also referred to as the categorical distribution) is a generalization of the Bernoulli distribution. If you perform an experiment that has *more* than two outcomes, or *k* outcomes, you have a multinoulli distributed random variable.

is a column vector of class indices, dimension $(N \times 1)$, though one could formulate this as matrix of one-hot encodings instead).

We will learn parameters by minimizing the negative log likelihood of our model's predictive posterior. If we say that $\mathbf{p}$ is our model's vector of normalized output probabilities, we define the negative log loss (cost) as follows:

$$\mathbf{p}_k = \frac{e^{\mathbf{f}_k}}{\sum_j e^{\mathbf{f}_j}}, \qquad \mathcal{J} = -\frac{1}{N}\sum_i \Big(\log(\mathbf{p}_{y_i})\Big) + \lambda \sum_d \sum_k (W_{d,k})^2 \tag{1}$$

noting that we have written a regularized loss (imposing a Gaussian prior distribution over the input-to-hidden parameters W).

We seek $\partial \mathcal{J}_i / \partial \mathbf{f}_k$, where the loss is computed from $\mathbf{p}$ which itself depends on $\mathbf{f}$. Implementing the gradient calculation for softmax regression turns out to be quite simple once one works through the derivations (by judiciously applying the chain rule of calculus[2]). It turns out that most things cancel out in the derivation, yielding the very elegant expression:

$$\frac{\partial \mathcal{J}_i}{\partial \mathbf{f}_k} = \mathbf{p}_k - \mathbf{1}_{y_i=k} \tag{2}$$

which directly gives us the partial derivatives of our loss with respect to the output unit pre-activities.[3]

We can think of this as simply subtracting one from the probability indexed at the correct class (or your probability vector minus the one-hot encoding of your class label). To obtain the partial derivatives of the loss with respect to parameters, we will need to take the derivative above and "back-prop" it through, yielding the simple gradients:

$$\frac{\partial \mathcal{J}}{\partial W} = \mathbf{X}^T * \Big(\frac{1}{N}\frac{\partial \mathcal{J}}{\partial \mathbf{f}_k}\Big) + \lambda(W) = \mathbf{X}^T * \Big(\frac{1}{N}(\mathbf{p}_k - \mathbf{1}_{y_i=k})\Big) + \lambda(W) \tag{3}$$

$$\frac{\partial \mathcal{J}}{\partial b} = \sum_{n=0}^{N} \frac{1}{N}\frac{\partial \mathcal{J}}{\partial \mathbf{f}_k} = \sum_{n=0}^{N} \frac{1}{N}(\mathbf{p}_k - \mathbf{1}_{y_i=k}) \tag{4}$$

where we also see that the gradient of the Gaussian prior over the weights W is equally straightforward (as discussed in HW#1, we do not regularize the biases). Note that the notation used here has been massaged a bit to be more implementation-oriented (for a language like Python).

Gradient-checking the code you have written for the partial derivatives is absolutely essential. Oftentimes, it is possible that your model will appear to "learn" even with a buggy implementation, however, you will find that it never quite behaves the way literature claims it should. While for the last homework assignment, gradient-checking was optional (though suggested), you are required to do so in this assignment to verify your learning algorithm's correctness.

As discussed in class, one can approximate the partial derivatives desired by instead invoking a secant approximation in what is otherwise known as the method of finite-differences. To calculate the numerical gradient, all we require is the prediction routine and the cost function routine to be correctly written. Specifically, we may estimate the gradient by simply recording how the loss function's value changes each time we perturb a parameter in a certain direction, by a certain amount, ϵ. The secant approximation to the partial derivatives we seek can be found from the definition of a limit:

$$\frac{\partial \mathcal{J}(\Theta)}{\partial \Theta} = \lim_{\epsilon \to 0} \frac{\mathcal{J}(\Theta + \epsilon) - \mathcal{J}(\Theta - \epsilon)}{2\epsilon} \tag{5}$$

where now our derivatives can be approximated by simply using the above equation and setting ϵ to a reasonably small value (i.e., 10^{-4}–do not set this to too small a value, else you will encounter numerical instabilities). The essence of implementing the numerical gradient calculation involves

[2]This is a rewarding derivation to work through but is left as an optional, ungraded exercise to the reader.

[3]Recall that the pre-activation, or "pre-activities", is simply a vector where each element is each neuron's integration of the incoming weighted inputs, or summation of the incoming signals. Post-activation, or activation, simply refers to the, usually non-linear, element-wise function we apply to the pre-activation values.

iterating over each and every single parameter scalar value (for example, each value inside the W matrix). As an example, let us say we operating with the bias vector b (containing j elements) of your regressor. To find the numerical gradient, you will loop through the parameter vector and do the following to each scalar value (b_j) within it (for notational simplicity, we can think of $\Theta = \{b\}$, since W is fixed:

1. Add ϵ to b_j, put b back into Θ, giving us $(\Theta + \epsilon)$ at j
2. Given perturbed b_j (all the rest of the values in Θ are fixed), calculate $\mathcal{J}(\Theta + \epsilon)$
3. Reset b to its original state
4. Subtract ϵ from b_j
5. Given perturbed b_j, calculate $\mathcal{J}(\Theta - \epsilon)$
6. Estimate the scalar derivative, $\left(\frac{\partial \mathcal{J}(\Theta)}{\partial \Theta}\right)_j$ (for b_j) using Equation 5
7. Reset b to its original state
8. Repeat Steps 1-7 for each b_j in b

After running through the following sketch of your numerical gradient algorithm, you will obtain an approximation of $\frac{\partial \mathcal{J}(\Theta)}{\partial \Theta}$ for b. Once you have obtained the full set of numerical partial derivatives, you will then want to compare them against your own routine's calculation of the exact derivatives.

You should observe agreement between your analytic solution and the numerical solution, up to a reasonable amount of decimal points in order for this part of the problem to be counted correct. Make sure you check your gradients before you proceed with the rest of this problem. We have provided you with a simple bit of code that will then use your numerical gradient and your analytic gradient to determine if your code passes the gradient-check. Make sure the program returns "CORRECT" for each parameter matrix/vector before focusing on tuning your model on the XOR data.

Once you have finished gradient-checking, fit your multinoulli regressor to the XOR dataset, found in the file `/problems/HW2/data/xor.dat`. Record what tried for your training process and any observations (such as any of the model's ability to fit the data) as well as your accuracy.

2.2 Problem #1b: Softmax Regression & the Spiral Problem

Now it is time to fit a model to a multi-class problem (generally defined as $k > 2$, or beyond binary classification). The spirals dataset (which we artificially generated for you offline to simulate an "unknown" data generating process) represents an excellent toy problem for observing the differences between various classification models, especially between multinoulli regression and the MLP.

Instead of the XOR data, you will now load in the spiral dataset file, `/problems/HW2/data/spiral_traindat`. In the starter script we have provided for you `/problems/HW2/data/prob1b.py`, we have written code for you again to plot the decision boundary of your trained multinoulli regressors on the spirals dataset (as we did in the end of HW #1). Similar code will also be used to plot the MLPs you will trained later for this problem. Please save and update your answer document with the generated plot once you have fit the softmax regression model as best you can to the data. Make sure you comment on your observations and describe any insights/steps taken in tuning the model. You will certainly want to re-use the code you wrote for the last sub-problem, such as code you wrote for *predict(·)*, *computeCost(·)*, and *computeGrad(·)*. Do NOT forget to report your accuracy.

2.3 Problem #1c: MLPs & the XOR Problem

Extending our softmax regression model to an MLP is not difficult, especially once we have decided on the activation function we would like to use for the hidden units. In this assignment, you will use the linear rectifier unit (*relu*), which has gained a lot of attention in modern neural network research due to the impressive performance it offers in training deep networks. The MLP architecture[4], with only a single hidden layer of h rectifier units, is defined as follows:

[4] Also known as a sparse rectifier network [1].

$$\mathbf{h}_{pre} = W_1 * \mathbf{X} + b_1, \qquad \mathbf{h} = \phi(\mathbf{h}_{pre}) \tag{6}$$

$$\mathbf{f} = W_2 * \mathbf{h} + b_2, \qquad \mathbf{p} = softmax(\mathbf{f}) \tag{7}$$

where $\Theta = \{W_1, b_1, W_2, b_2\}$ houses the MLP's parameters and $\phi(\cdot)$ is the *relu* activation function, or $\phi(x) = max(0, x)$. Note that $softmax(\cdot)$ is the softmax activation function, or the soft winner-take-all function presented in Equation 1 (leftmost expression). The full loss for this MLP is simply:

$$\mathcal{J} = -\frac{1}{N}\sum_i \Big(\log(\mathbf{p}_{y_i})\Big) + \lambda\Big(\sum_h \sum_k (W_{2,(h,k)})^2 + \sum_d \sum_h (W_{1,(d,h)})^2\Big) \tag{8}$$

To calculate the gradients for the rest of the MLP, in addition to the gradients we derived in Problem #1a (just swap out the names W with W_2 and b with b_2, since the softmax model sits on top of the *relu* units), we need to further back-propagate the derivatives we calculated at the pre-activations of the output units, $\mathbf{f}$, back to the pre-activations of the hidden units, $\mathbf{h}_{pre}$. This can be done simply as follows:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{h}} = \Big(\frac{1}{N}(\mathbf{p}_k - \mathbf{1}_{y_i=k})\Big) * W_2^T \tag{9}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{h}_{pre}} = \frac{\partial \mathcal{J}}{\partial \mathbf{h}} \otimes \mathbf{1}_{\mathbf{h}_{pre}>0} = \Big(\Big(\frac{1}{N}(\mathbf{p}_k - \mathbf{1}_{y_i=k})\Big) * W_2^T\Big) \otimes \mathbf{1}_{\mathbf{h}_{pre}>0}. \tag{10}$$

where you can think of the term $\mathbf{1}_{\mathbf{h}_{pre}>0}$ (the first-order derivative of our activation function, which is an indicator function) as simply a gate that lets the gradient pass backwards (by multiplying it by one) if it is positive (greater than zero) and mutes it completely (by multiplying it by zero) if it is negative (less than zero). Continuing from the derivative of the loss with respect to the hidden layer pre-activities, we can now compute the gradients of all of the parameters of the MLP as follows:

$$\frac{\partial \mathcal{J}}{\partial W_2} = \Big(\mathbf{h}^T * \Big(\frac{1}{N}\frac{\partial \mathcal{J}}{\partial \mathbf{f}_k}\Big)\Big) + \lambda(W_2) \tag{11}$$

$$\frac{\partial \mathcal{J}}{\partial b_2} = \sum_{n=0}^{N} \frac{1}{N}\frac{\partial \mathcal{J}}{\partial \mathbf{f}_k} \tag{12}$$

$$\frac{\partial \mathcal{J}}{\partial W_1} = \Big(\mathbf{X}^T * \Big(\frac{\partial \mathcal{J}}{\partial \mathbf{h}_{pre}}\Big)\Big) + \lambda(W_1) \tag{13}$$

$$\frac{\partial \mathcal{J}}{\partial b_1} = \sum_{n=0}^{N} \frac{\partial \mathcal{J}}{\partial \mathbf{h}_{pre}}. \tag{14}$$

Armed with these gradients, you can now implement your update rule and traverse the error landscape to fit your MLP to data. Much as you did in assignment one, you will perform multiple steps of gradient descent to update your parameters, following the recipe:

$$W_1 = W_1 - \alpha\frac{\partial \mathcal{J}}{\partial W_1}, \qquad b_1 = b_1 - \alpha\frac{\partial \mathcal{J}}{\partial b_1} \tag{15}$$

$$W_2 = W_2 - \alpha\frac{\partial \mathcal{J}}{\partial W_2}, \qquad b_2 = b_2 - \alpha\frac{\partial \mathcal{J}}{\partial b_2} \tag{16}$$

Just like you did for softmax regression, you will want to make sure you gradients pass the finite-difference check. Go ahead and use (or extend) your gradient-check routine from Problem #1a to ensure that your MLP gradients are indeed correct. Once you have passed the gradient-check, fit your MLP to the XOR problem data. Report your accuracy as well as record your loss as a function of epochs (present its evolution during training either through a plot or a screenshot of the program output that should appear in the terminal). Was your MLP model better able to fit the XOR data? Why would this be the case, when compared to your softmax regressor?

2.4 Problem #1d: MLPs & the Spiral Problem

Much as you did for Problem #1b, you are now to fit your MLP model to the spiral dataset and plot its decision boundary. Use the dataset file, `/problems/HW2/data/spiral_traindat`, and

write your code in the starter script `/problems/HW2/prob1d.py`. Again, you will most likely want to re-use/copy over the code you gradient-checked for the last sub-problem. Make sure you document what you did to tune the MLP, including what settings of the hyper-parameters you explored (such as learning rate/step-size, number of hidden units, number of epochs, value of regularization coefficient, etc.). Do not forget to write your observations and thoughts/insights.

Generate the decision boundary plot of your tuned MLP and paste it into your answer document. Comment (in your answer document) on the decision boundary plot: What is different between the MLP's decision boundary and the multinoulli regressor's decision boundary? Does the MLP decision boundary accurately capture the data distribution? How did the regularization coefficient affect your model's decision boundary? Do NOT forget to report your accuracy.

3 Problem #2: Fitting MLPs to Data

You will now re-appropriate your MLP code to a more standard machine learning problem, where a separate validation set is given in addition to the training set. In this scenario, you are no longer concerned directly with pure optimization (as we discussed in class), but with generalization (or out-of-sample performance). As such, you will have two quantities to track – the training loss and the validation loss.

3.1 Problem #2a: 1-Layer MLP for IRIS

Fit/tune your single hidden layer MLP to the IRIS dataset, `/problems/HW2/data/iris_train.dat`. Write your code in `/problems/HW2/prob2a.py` (you will most certainly want to re-use your code from the previous problem). Note that now you are concerned with out-of-sample performance, and must estimate it by using the validation/development dataset we have also provided for you, `/problems/HW2/data/iris_test.dat`.

Note that for Problem #2, we will be utilizing mini-batches instead of the full batch of data to calculate gradients. As a result, you will have to write your own mini-batch creation function (as indicated in the comment-annotated routine *create_mini_batch(·)*). This will be necessary in order to write a useful/successful training loop. Note that you are free to modify the design of *create_mini_batch(·)* as you see fit (though the suggested design allows or a simple implementation).

Besides recording your accuracy for both your training and development sets, track your loss as a function of epoch. Create a plot with both of these curves superimposed. Furthermore, consider the following questions: What ultimately happens as you train your MLP for more epochs? What phenomenon are you observing and why does this happen? What are two ways you can control for this?

3.2 Problem #2b: 2-Layer MLP for IRIS

Extend your MLP code from the last problem to support an additional hidden layer of h_2 hidden units (the first hidden layer could be considered as having h_1 units). This means that the parameter construct for your MLP is defined as $\Theta = \{W_1, b_1, W_2, b_2, W_3, b_3\}$. Write this in the starter script `/problems/HW2/prob2b.py`.

Fit/tune your deeper MLP to the IRIS dataset and record your training/validation accuracies. Furthermore, create plots of your loss-versus-epoch curves as you did in Problem #1a. Put all of this in your answer document and write down any thoughts/comments of your tuning process and the differences observed between training a single and two-hidden layer MLP.

References

[1] GLOROT, X., BORDES, A., AND BENGIO, Y. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (2011), pp. 315–323.

[2] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.