

Introduction to Machine Learning (67577)

Exercise 7 Neural Networks

Second Semester, 2022

Contents

1	Submission Instructions	2
2	Theoretical Part	2
3	Practical Part	2
3.1	Classifying a 2D linearly inseparable simulated dataset	4
3.2	Classifying MNIST	5

1 Submission Instructions

Please make sure to follow the general submission instructions available on the course website. In addition, for the following assignment, submit a single `ex7_ID.tar` file containing:

- An `Answers.pdf` file with the answers for all theoretical and practical questions (include plotted graphs *in* the PDF file).
- The following python files (without any directories): `nn_simulated_data.py`, `nn_mnist_digit_classification.py`, `neural_network.py`, `neural_networks.modules.py`, `stochastic_gradient_descent.py`, `loss_functions.py`

The `ex7_ID.tar` file must be submitted in the designated Moodle activity prior to the date specified *in the activity*.

- Late submissions will not be accepted and result in a zero mark.
- Plots included as separate files will be considered as not provided.
- Do not forget to answer the Moodle quiz of this assignment.

2 Theoretical Part

1. Let $\mathbf{x}, \mathbf{w} \in \mathbb{R}^d$. Calculate the Jacobian $\frac{\partial \mathbf{x}^\top \mathbf{w}}{\partial \mathbf{w}}$. What are the Jacobian dimensions? (Hint: what are the input/output dimensions of this function?)
2. Let $\mathbf{X} \in \mathbb{R}^{m \times d}$ and $\mathbf{w} \in \mathbb{R}^d$. Calculate the Jacobian $\frac{\partial \mathbf{X}\mathbf{w}}{\partial \mathbf{w}}$. What are the Jacobian dimensions? (Hint: what are the input/output dimensions of this function?)
3. Let $\mathbf{x} \in \mathbb{R}^d$ and $W \in \mathbb{R}^{k \times d}$. Calculate the derivative $\frac{\partial W\mathbf{x}}{\partial W}$. What are the Jacobian dimensions? (Hint: what are the input/output dimensions of this function?)
4. Let $\mathbf{X} \in \mathbb{R}^{m \times d}$ and $W \in \mathbb{R}^{d \times k}$. Calculate the derivative $\frac{\partial \mathbf{X}W}{\partial W}$. What are the Jacobian dimensions? (Hint: what are the input/output dimensions of this function?)
5. Let $\mathbf{v} \in \mathbb{R}^d$, let $S: \mathbb{R}^d \rightarrow (0, 1)^d$ be the softmax function and let $H: [0, 1]^d \times [0, 1]^d \rightarrow \mathbb{R}_+$ be the cross entropy function


$$[S(\mathbf{v})]_i := \frac{e^{v_i}}{\sum_j e^{v_j}}, \quad H(p, q) := -\sum p_i \log(q_i)$$

Find a simple expression for: $\frac{\partial H(e_k, S(\mathbf{v}))}{\partial \mathbf{v}}$

3 Practical Part

In this exercise you will implement a simple feed-forward neural network from scratch. You will then use it to learn both simulated and real-world data. You should use your implementations of Gradient Descent and learning rates previously implemented (i.e. `gradient_descent.py` and

`learning_rate.py`). You would also implement Stochastic Gradient Descent, which should be very similar to your Gradient Descent implementation.

 If you did not do exercise 6, you are allowed to use the GD implementation from the published solution, once it's published.

Implement the following classes as described below. Follow class and function documentations.

- Implement the `cross_entropy` and `softmax` functions in the `metrics.loss_functions.py` file.
- Implement following modules in the file `learners.neural_networks.modules.py`: `FullyConnectedLayer`, `ReLU`, `CrossEntropyLoss`.
- Implement the `NeuralNetwork` class in the `learners.neural_networks.neural_network.py` in the following steps:
 - Implement the `compute_output` method according to the Forward pass in the Back-propagation algorithm. Propagate the input through all network layers including the loss function.
 - Implement the `compute_jacobian` method according to the Backwards pass in the Backpropagation algorithm. Notice the method returns a flattened vector of all weights, for simple usage in `GradientDescent/StochasticGradientDescent`. Flattening can be done using the `flatten_parameters` method.
 - Implement the remaining `NeuralNetwork` methods.
- Implement the class `StochasticGradientDescent`.
- Implement the `confusion_matrix` function in `metrics.py`

Implementation tips and hints:

As this is a non-trivial algorithm to implement, use the following tips and hints for the back-propagation implementation. *Be sure to read these carefully.*

- When initializing a `FullyConnectedLayer` with d inputs and k outputs, each weight should be initialized randomly following the following distribution: $\mathcal{N}(0, \frac{1}{d})$.
- As specified in the back-propagation algorithm, be sure to *save all intermediate calculations* of the forward pass (\mathbf{a}_l and \mathbf{o}_l). These are to be used in the backward pass.
- Before calculating the derivatives with respect to the parameters, make sure to answer the theoretical questions. Specifically, recall that when computing the derivative of a matrix w.r.t to another matrix the results in a 4 dimensional tensor. For every output index i, j we need to compute the derivative with respect to each weight index k, l .

Notice however that in many cases this computation can be simplified in various ways and therefore saving computation time and memory consumption:

- Activation functions are element-wise functions. So, index i, j in the output would only be influenced by the index i, j in the input. Therefore, the derivative can be computed *element-wise* and applied in the same manner.
- As for the matrix multiplication \mathbf{XW} , the derivative of this expression can also be simplified. Recall that the Jacobian of a function is the linear approximation of the

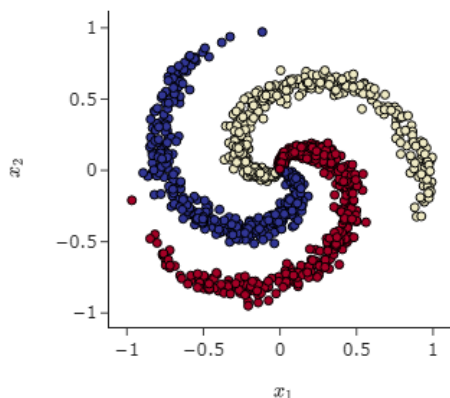
function. Since the function \mathbf{XW} is linear (w.r.t W), it makes sense that the Jacobian would be \mathbf{X} . However, as the multiplication order is \mathbf{XW} and not $W\mathbf{X}$, the derivative is actually \mathbf{X}^\top .

- Notice that in each layer, you need to perform two **different** operations:
 1. Modify the Jacobian to be sent to the preceding layer: calculating Δ_t from Δ_{t-1}
 2. Calculate the gradient w.r.t the current layer weights: $\nabla_{W_{t-1}}$ by using the gradient from the previous layers Δ_{t-1}
- Should the intercept (if added) in layer W_{t-1} influence the calculation of Δ_t ?
- Test your implementation when using unique layer widths - this will make some bugs result in a runtime error (which is much better than a bug without a runtime error).
- Additional helpful material can be found: [here](#) or [here](#) or [here](#)

3.1 Classifying a 2D linearly inseparable simulated dataset

In this part, you will use your neural network implementation to classify a simple 2d dataset as seen below, which is not linearly separable. Since this data is 2d, we can easily visualize the network's decision boundary, and see the piecewise linear behavior of the neural network.

Train Data



In the `nonlinear_decision_boundary.py` file

- Create a simple architecture that consists of *two* fully connected hidden layers, each with 16 neurons, an intercept (not included in the 16 neurons) and ReLU activations.
- Specify the network's loss function to be Softmax-Cross Entropy loss function.
- Specify the network's solver to be your previously implemented gradient descent algorithm set to use a fixed learning rate of $\eta \equiv 0.1$ and to perform up to 5000 iterations

Then, answer the following questions:

1. Fit the network over the train set using the hyperparameters mentioned above. Using the `plot_decision_boundary` function, plot the boundaries learned by the network. Then, evaluate network's performance (accuracy) over the test set and report the results.
2. Remove both hidden layers (only for this question), and repeat the previous question. Explain the results.
3. Rerun the fitting of the network in question 1, this time while specifying a callback function to record network's convergence process. Plot the convergence process. i.e., loss as a function of iteration and well as gradient norm as a function of iteration. In addition, on every 100'th iteration store network's weights. Call the `animate_decision_boundary` function to see how the network's decision boundaries change as fitting progresses. No need to include the outputted animation in your submission.
4. Decrease the hidden layer's width to 6 neurons each (only for this question), and repeat the previous question (both plot and animation). Explain the results.

3.2 Classifying MNIST

Next, we turn to classify a more challenging task - the MNIST image recognition dataset. As this dataset is much larger, using GD is computationally challenging. Therefore, we will use Stochastic Gradient Descent (SGD) in order to improve runtime.

In the `mnist_digit_recognition.py` file

- Create a simple architecture that consists of *two* fully connected hidden layers, each with 64 neurons, an intercept (not included in the 64 neurons) and ReLU activations.
- Specify the network's loss function to be the Softmax-Cross Entropy loss function.
- Specify the network's solver to be your implementation of the SGD algorithm set to use a fixed learning rate of $\eta \equiv 0.1$, perform up to 10,000 iterations and to use batches of 256 samples.
- Specify a callback function that records convergence process: at each call the current loss and the gradient norm.

After completing those, answer the following questions:

5. Train the neural network on the train set using the hyperparameters mentioned above. Evaluate it on the test set and report the test accuracy.
6. Plot the convergence process. i.e., loss as a function of iteration and well as gradient norm as a function of iteration.
7. Plot a confusion matrix between the true- and predicted labels of the test set. What are the two most common confusion? What are the three least common confusions? Do these results

make sense?

8. Remove both hidden layers (only for this question), and repeat question 5. What can be assumed about the data based on the evaluated test accuracy?
9. Recall that the prediction of a network (after Softmax) is a probability vector \mathbf{p} , where p_k describes the probability that the true class is k . We define the *confidence* of a prediction \mathbf{p} as $\max_k p_k$. Notice that if \mathbf{p} is the uniform vector, the confidence is minimal.

Compare the most/least confident images of a specific digit. Filter the test dataset to containing only images where the true digit is 7. Using the `plot_images_grid` function plot the 64 images we are most and least confident about their prediction. Can you identify any differences between these sets?

10. In this question, we show the runtime differences between SGD and GD. Using the same architecture defined at the beginning of this section, train a network twice: once using GD as the solver and one using SGD as the solver.
 - Initialize both solvers to use a fixed learning rate of 10^{-1} , a maximum number of 10,000 iterations and a tolerance of 10^{-10} .
 - When using the SGD solver use batches of 64 samples.
 - Specify a callback function to record at each iteration the network's current loss and time passed from the beginning of the fit.
 - Train the networks over the first 2500 train samples

Then, plot the following:

- For each solver separately, plot a graph showing the running time vs. loss. (i.e. two figures)
- An additional figure with graphs of both solvers one on top of the other as two different scatters (i.e. a single figure, no subplots, two scatters)

Explain the similarities and differences between the fit processes using the two solvers. Consider running times, loss scales and shape of curve.