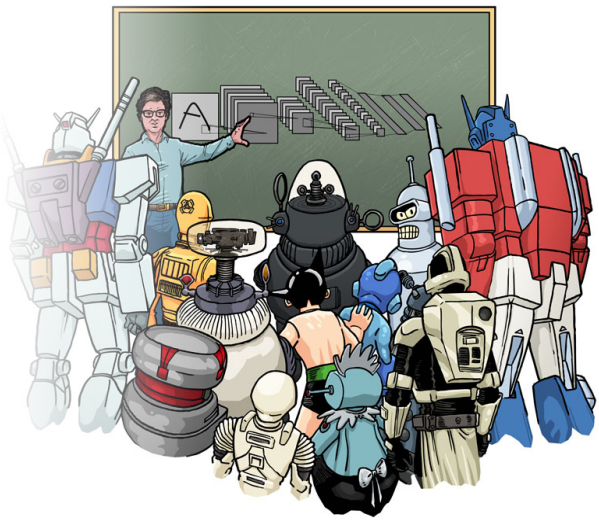

METU EE 496
Introduction to
Computational Intelligence

Training Multi-Layer Perceptron

Homework 1 - Training Multi-Layer Perceptron

Due: 23:55, 31/03/2019

Late submissions are welcome, but penalized according to the following policy:

- 1 day late submission: HW will be evaluated out of 70.
- 2 days late submission: HW will be evaluated out of 50.
- 3 days late submission: HW will be evaluated out of 30.
- 4 or more days late submission: HW will not be evaluated.

You should prepare your homework by yourself alone and you should not share it with other students, otherwise you will be penalized.

Introduction

In this homework, you will perform experiments on multilayer perceptron (MLP) training and draw conclusions from the experimental results. You will train a classifier on a subset of Fashion-MNIST dataset [1]. The implementations will be in Python language and you will be using Neural Network module of Scikit-Learn package [2]. You can visit the link provided in the reference [2] to understand the usage of the module.

Figure 1: Samples from dataset. The classes are (from top to bottom) top-wear, bottom-wear, dress, footwear and bag.

Dataset Description

The dataset you will work on is a subset of Fashion-MNIST dataset [1]. It is composed of 28×28 1-band gray-scale images of 5 clothing classes which are top-wear, bottom-wear, dress, footwear and bag. Some samples from the dataset is provided in Fig. 1.

The dataset is provided in ODTUClass course page under Homework_1 folder. The dataset is split into two subsets: one for training and one for testing. For training, there are 30000 samples corresponding to 6000 samples for each class and for testing, there are 5000 samples corresponding to 1000 samples for each class.

The labels and the images are stored in separate files as NumPy arrays (**.npy*). *load* function of NumPy package can be simply used to read the dataset subsets. The labels are stored as 1-D arrays and the images are stored as 2-D arrays where each row corresponds to a flattened (vectorized) image in row-major order. Thus, one can reshape the flattened image array to 28×28 array to reconstruct the 2-D image. The i^{th} row in the image array is associated to the label which is the i^{th} entity in the label array.

Each pixel of the image is an integer between 0 and 255. The labels are integers between 0 and 4 with the semantic associations of top-wear (0), bottom-wear (1), dress (2), footwear (3) and bag (4).

Neural Network Module of Scikit-Learn

Network module of Scikit-Learn package contains two classes for supervised learning: MLPClassifier and MLPRegressor. Upon constructing a class for your task, the training and testing operations are simple via calling the class methods. Moreover, the learned weights and biases of MLP can be obtained from and modified via class attributes. Please refer to the simple explanations of those classes in the web page of the module [2] where you can find examples as well.

Homework Task and Deliverables

The homework is composed of 5 parts. In the first part you are to answer general questions on MLP and its training. For the other parts, you will write codes to perform experiments on classification performances under several settings. You will provide the results of the experiments by some visuals and you will interpret the results by your own conclusions.

You should submit a report in which your answers to the questions, the required experimental results (performance curve plots, visualizations etc.) and your deductions are presented for each part of the homework. Moreover, for the parts 2-4, you should submit submit a single Python file for each part to generate the results and the visualizations of the experiments. Namely, all the required tasks for a part can be performed by running the related code file. The codes should be well structured and **well commented**. The submissions lacking comments will be simply not evaluated.

The report should be in portable document format (pdf) and named as *report.pdf*. The naming of the code files should be *part_X.py* where X is the part number.

For the homework submission, make sure that you have properly named the deliverables (report and 3 code files) and zip all the deliverables in a single archive named *hw1_name_surname_eXXXXXX* where *name*, *surname* and Xs are to be replaced by your name, surname and digits of your user ID, respectively.

1 Basic Concepts

Answer the following questions.

1.1 Which Function?

MLPs are actually parametric functions which can be used to approximate other functions. What function does the MLP classifier of Scikit-Learn [2] approximates? How is the loss defined to approximate that function? *Bonus: Why?*

1.2 Gradient Computation

You are training a MLP by using stochastic gradient descent (SGD) approach with a learning rate γ. You introduce no weight regularization to the loss and no momentum is used in the gradient updates. Under these settings, you are given weights, (w_k, w_{k+1}), at iterations k and k+1, respectively. How can you compute the gradient of the loss, $\mathcal{L}$, with respect to w at step k? Express the gradient, $\nabla_w \mathcal{L} \mid_{w=w_k}$, in terms of (γ, w_k, w_{k+1}).

1.3 Some Training Parameters and Basic Parameter Calculations

1. What are **batch** and **epoch** in the context of MLP training?
2. Given that the dataset has N samples, what is the number of batches per epoch if the batch size is B?
3. Given that the dataset has N samples, what is the number of SGD iterations if you want to train your MLP for E epochs with the batch size of B?

1.4 Computing Number of Parameters of an MLP Classifier

Consider an MLP classifier of K hidden units where the size of each hidden unit is H_k for k=1, . . . , K. Derive a formula to compute the number of parameters that the MLP has if the input and output dimensions are D_{in} and D_{out}, respectively.

2 Experimenting MLP Architectures

In this part, you will experiment on several MLP architectures for classification task. Use *rectified linear unit* function for the activation function, *adaptive moment estimatation (ADAM)* with default parameters for the training method, batch size of 500 samples and use no weight regularization throughout the all experiments.

Preprocess the train and the test data so that the pixel values are scaled to $[-1.0, 1.0]$. Split 10% of the training data as the validation set by randomly taking equal number of samples for each class. Hence, you should have three sets: training, validation and testing.

2.1 Experimental Work

The name of the architectures to be experimented and their hidden layer sizes are:

1. 'arch_1': (128)
2. 'arch_2': (16, 128)
3. 'arch_3': (16, 128, 16)
4. 'arch_5': (16, 128, 64, 32, 16)
5. 'arch_7': (16, 32, 64, 128, 64, 32, 16)

Now, for each architecture, you will perform the following tasks:

1. Using training set, train the MLP for 100 epochs using *partial_fit* method of *MLPClassifier*. Do not use *fit* method.

   During training,

   - Record the training loss, training accuracy, validation accuracy for every 10 steps to form loss and accuracy curves (**Hint:** Use *loss_* attribute to get the current loss and use *score* method to compute accuracy);
   - Shuffle training set after each epoch (**Hint:** Use *random.permutation* of NumPy package to shuffle both images and labels in the same order).

   After training,

   - Compute test accuracy (**Hint:** Use *score* method);
   - Record the weights of the first hidden layer (**Hint:** Copy the first item in the *coefs_* attribute).

2. Repeat 1 for at least 10 times and
   - Take the average of the resultant loss and accuracy curves;
   - Record the best test accuracy among all the runs;
   - Record the weights of the first hidden layer of the trained MLP that has the best test performance.

3. Now, form a dictionary object with the following key-value pairs as the result of the training experiment for the given architecture:
   - 'name': name of the architecture
   - 'loss_curve': average of the training loss curves from all runs
   - 'train_acc_curve': average of the training accuracy curves from all runs
   - 'val_acc_curve': average of the validation accuracy curves from all runs
   - 'test_acc': the best test accuracy value from all runs
   - 'weights': the weights of the first hidden layer of the trained MLP with the best test performance

4. Save the dictionary object with the filename as the architecture name by prefixing 'part2' in the front(**Hint:** Use *pickle* to save dictionary objects to file and load dictionary objects from file).

Once the aforementioned tasks are performed for each architecture, create performance comparison plots by using the provided *part2Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Note that you should pass all the dictionary objects corresponding to results of the experiments as a list to create performance comparison plots. (**Hint:** You can load previously saved results and form a list to be passed to the plot function). Add this plot to your report.

Additionally, for all architectures except for 'arch_1', visualize the weights of the first hidden layer by using the provided *visualizeWeights* function in the *utils.py* file under HW1 folder in ODTUClass course page. Add these visualizations to your report.

2.2 Discussions

Compare the architectures by considering the performances, the number of parameters, architecture structures and the weight visualizations.

1. What is the generalization performance of a classifier?
2. Which plots are informative to inspect generalization performance?
3. Compare the generalization performance of the architectures.
4. How does the number of parameters affect the classification and generalization performance?

5. How does the depth of the architecture affect the classification and generalization performance?
6. Considering the visualizations of the weights, are they interpretable?
7. Can you say whether the units are specialized to specific classes?
8. Weights of which architecture are more interpretable?
9. Considering the architectures, comment on the structures (how they are designed). Can you say that some architecture are akin to each other? Compare the performance of similarly structured architectures and architectures with different structure.
10. Which architecture would you pick for this classification task? Why?

Put your discussions together with performance plots and weight visualizations to your report. Submit a single Python file named *part_2.py* to generate the results and the visualizations of the experiments. Namely, all the required tasks for part 2 can be performed by running *part_2.py*.

3 Experimenting Activation Functions

In this part, you will compare rectified linear unit (ReLU) function and the logistic sigmoid function. Use SGD for the training method, constant learning rate of 0.01, 0.0 momentum (no momentum), batch size of 500 samples and use no weight regularization throughout the all experiments.

Preprocess the train and the test data so that the pixel values are scaled to $[-1.0, 1.0]$.

3.1 Experimental Work

Consider the architectures in 2.1, for each architecture create two MLPClassifier objects: one with the ReLU activation function and one with the logistic sigmoid activation function. Then, perform the following tasks for the two classifiers:

1. Using training set, train the two MLPs for 100 epochs using *partial_fit* method of *MLPClassifier*. Do not use *fit* method.

   During training,

   - Record the training loss and magnitude of the loss gradient with respect to the weights of the first hidden layer at every 10 steps to form loss and gradient magnitude curves (**Hint:** Use *loss_* attribute to get the current loss and use copy of the first item in the *coefs_* attribute to obtain the copies of the weights of the first hidden layer at time steps);
   - Shuffle training set after each epoch (**Hint:** Use *random.permutation* of NumPy package to shuffle both images and labels in the same order).

   After training, form a dictionary object with the following key-value pairs as the result of the training experiment for the given architecture:

   - 'name': name of the architecture
   - 'relu_loss_curve': the training loss curve of the MLP with ReLU
   - 'sigmoid_loss_curve': the training loss curve of the MLP with logistic sigmoid
   - 'relu_grad_curve': the curve of the magnitude of the loss gradient of the MLP with ReLU
   - 'sigmoid_grad_curve': the curve of the magnitude of the loss gradient of the MLP with ReLU
2. Save the dictionary object with the filename as the architecture name by prefixing 'part3' in the front (**Hint:** Use *pickle* to save dictionary objects to file and load dictionary objects from file).

Once the aforementioned tasks are performed for each architecture, create performance comparison plots by using the provided *part3Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Note that you should pass all the dictionary objects corresponding to results of the experiments as a list to create performance comparison plots. Add this plot to your report.

3.2 Discussions

Compare the architectures by considering the training performances:

1. How is the gradient behavior in different architectures? What happens when depth increases?
2. Why do you think that happens?
3. *Bonus:* What might happen if we do not scale the inputs to the range $[-1.0, 1.0]$?

Put your discussions together with performance plots and weight visualizations to your report. Submit a single Python file named 'part_3.py' to generate the results and the visualizations of the experiments. Namely, all the required tasks for part 3 can be performed by running 'part_3.py'.

4 Experimenting Learning Rate

In this part, you will examine the effect of the learning rate in SGD method. Use SGD for the training method, constant learning rate, ReLU activation function, 0.0 momentum (no momentum), batch size of 500 samples and use no weight regularization throughout the all experiments. You will vary the iniital learning rate during the experiments so that each training will be performed with different learning rate.

Preprocess the train and the test data so that the pixel values are scaled to $[-1.0, 1.0]$. Split 10% of the training data as the validation set by randomly taking equal number of samples for each class. Hence, you should have three sets: training, validation and testing.

4.1 Experimental Work

Pick your favorite architecture from 2.1, excluding 'arch_1'. Create three MLPClassifier objects of initial learning rates 0.1, 0.01 and 0.001, respectively. Then, perform the following tasks for the three classifiers:

- Record the training loss, validation accuracy for every 10 steps to form loss and accuracy curves (**Hint:** Use *loss_* attribute to get the current loss and use *score* method to compute accuracy);
- Shuffle training set after each epoch (**Hint:** Use *random.permutation* of NumPy package to shuffle both images and labels in the same order).

After training, form a dictionary object with the following key-value pairs as the result of the training experiment for the given architecture:

- 'name': name of the architecture
- 'loss_curve_1': the training loss curve of the MLP trained with 0.1 learning rate
- 'loss_curve_01': the training loss curve of the MLP trained with 0.01 learning rate
- 'loss_curve_001': the training loss curve of the MLP trained with 0.001 learning rate
- 'val_acc_curve_1': the validation accuracy curves of the MLP trained with 0.1 learning rate
- 'val_acc_curve_01': the validation accuracy curves of the MLP trained with 0.01 learning rate
- 'val_acc_curve_001': the validation accuracy curves of the MLP trained with 0.001 learning rate

Once the aforementioned tasks are performed for each architecture, create performance comparison plots by using the provided *part4Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Note that you should pass a single dictionary objects corresponding to result of the experiment to create performance comparison plots. Add this plot to your report.

Now, you will try to make scheduled learning rate to improve SGD based training.

1. Examine the validation accuracy curve of the MLP trained with 0.1 learning rate. Approximately determine the epoch step where the accuracy stops increasing.
2. Create an MLPClassifier object with the same parameters as above and with initial learning rate of 0.1.

3. Train that classifier until the epoch step that you determined in 1. Then, set the learning rate to 0.01 and continue training until 200 epochs (**Hint:** Use *set_params* method).
4. Record only the validation accuracy during this training.
5. Now, plot the validation accuracy curve and determine the epoch step where the accuracy stops increasing.
6. Repeat 2 and 3; however, in 3, continue training with 0.01 until the epoch step that you determined in 5. Then, set the learning rate to 0.001 and continue training until 200 epochs.
7. Repeat 4 and once the training ends, record the test accuracy of the trained model and compare it to the same model trained with ADAM in 2.1.

Note: You can increase the number of epochs if you are not to observe the steps where training stops improving.

4.2 Discussions

Compare the effect of learning rate by considering the training performances:

1. How does the learning rate affect the convergence speed?
2. How does the learning rate affect the convergence to a better point?
3. Does your scheduled learning rate method work? In what sense?
4. Compare the accuracy and convergence performance of your scheduled learning rate method with ADAM.

Put your discussions together with performance plots and weight visualizations to your report. Submit a single Python file named 'part_4.py' to generate the results and the visualizations of the experiments. Namely, all the required tasks for part 4 can be performed by running 'part_4.py'.

References

[1] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python." `https://scikit-learn.org/stable/modules/neural_networks_supervised.html`, 2011.