Machine Learning in Physics — UCSD PHYS 139/239

Homework 2

Draft version due: Friday, October 24, 2025, 8:00pm

Final version due: Wednesday, October 29, 2025, 8:00pm

Policies

- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- Please submit your report as a single .pdf file to Gradescope under "Homework 2" or "Homework 2 Corrections". **In the report, include any images generated by your code along with your answers to the questions.** For instructions specifically pertaining to the Gradescope submission process, see `https://www.gradescope.com/get_started#student-submission`.
- Please submit your code as a .zip archive to Gradescope under "Homework 2 Code" or "Homework 2 Code Corrections". The .zip file should contain your code files. Submit your code either as Jupyter notebook .ipynb files or .py files.

1 Stochastic Gradient Descent [36 Points]

Stochastic gradient descent (SGD) is an important optimization method in machine learning, used everywhere from logistic regression to training neural networks. In this problem, you will be asked to first implement SGD for linear regression using the squared loss function. Then, you will analyze how several parameters affect the learning process.

Linear regression learns a model of the form:

$$f(x_1, x_2, \cdots, x_d) = \left(\sum_{i=1}^{d} w_i x_i\right) + b$$

Problem A [2 points]: We can make our algebra and coding simpler by writing $f(x_1, x_2, \cdots, x_d) = \mathbf{w}^\intercal \mathbf{x}$ for vectors $\mathbf{w}$ and $\mathbf{x}$. But at first glance, this formulation seems to be missing the bias term b from the equation above. How should we define $\mathbf{x}$ and $\mathbf{w}$ such that the model includes the bias term?

Hint: *Include an additional element in* $\mathbf{w}$ *and* $\mathbf{x}$.

Solution A: *We can include the bias term b by setting $w_0 = b$ and adding a new constant feature to the dataset $x_0 = 1$. Thus, we have*

$$\left(\sum_{i=0}^{d} w_i x_i\right) = \left(\sum_{i=1}^{d} w_i x_i\right) + b \tag{1}$$

Linear regression learns a model by minimizing the squared loss function L, which is the sum across all training data $\{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\}$ of the squared difference between actual and predicted output values:

$$L(f) = \sum_{i=1}^{N} (y_i - \mathbf{w}^\intercal \mathbf{x}_i)^2 \tag{2}$$

Problem B [2 points]: SGD uses the gradient of the loss function to make incremental adjustments to the weight vector $\mathbf{w}$. Derive the gradient of the squared loss function with respect to $\mathbf{w}$ for linear regression.

Solution B: *We can take the derivative of the loss function L with respect to one of the weight vector elements w_k,*

$$\frac{\partial L}{\partial w_k} = \frac{\partial}{\partial w_i} \sum_{i=1}^{N} \left(y_i - \sum_{j=0}^{d} w_j x_{ij}\right)^2 \tag{3}$$

$$= 2 \sum_{i=1}^{N} \left(y_i - \sum_{j=0}^{d} w_j x_{ij}\right)(-x_{ik}) \tag{4}$$

where x_{ij} is the jth feature of the ith data point. We can rewrite this with vectors as

$$\nabla_{\mathbf{w}} L = 2 \sum_{i=1}^{N} (y_i - \mathbf{w}^\intercal \mathbf{x}_i)(-\mathbf{x}_i) \tag{5}$$

The following few problems ask you to work with the first of two provided Jupyter notebooks for this problem, `1_notebook_part1.ipynb`, which includes tools for gradient descent visualization. This notebook utilizes the files `sgd_helper.py` and `sgd_multiopt_helper.py`, but you should not need to modify either of these files.

For your implementation of problems C–E, **do not** consider the bias term.

Problem C [8 points]: Implement the `loss`, `gradient`, and `SGD` functions, defined in the notebook, to perform SGD, using the guidelines below:

- Use a squared loss function.
- Terminate the SGD process after a specified number of *epochs*. Each epoch corresponds to one full pass over the entire dataset. One SGD iteration (weight update) is performed for each point in the dataset. So one epoch is equivalent to N gradient updates, where N is the size of the dataset.
- It is recommended, but not required, that you shuffle the order of the points before each epoch such that you go through the points in a random order. You can use `numpy.random.permutation`.
- Measure the loss after each epoch. Your `SGD` function should output a single vector with the loss after each epoch, and a single matrix of the weights after each epoch (one row per epoch). Note that the weights from all epochs are stored in order to run subsequent visualization code to illustrate SGD.

Solution C: *See code.*

Problem D [2 points]: Run the visualization code in the notebook corresponding to problem D. How does the convergence behavior of SGD change as the starting point varies? How does this differ between datasets 1 and 2? Please answer in 2–3 sentences.

Solution D: *SGD converges to the same point, the global minimum, regardless of the starting point. Because the loss function is convex, the minimum is unique. It also steps fairly directly toward the minimum when viewing the steps after each epoch. However, if the starting point is farther from the global minimum, it may take more iterations.*

Problem E [6 points]: Run the visualization code in the notebook corresponding to problem E. One of the cells—titled "Plotting SGD Convergence"—must be filled in as follows. Perform SGD on dataset 1 for each of the learning rates $\eta \in \{10^{-6}, 5 \times 10^{-6}, 10^{-5}, 3 \times 10^{-5}, 10^{-4}\}$. On a single plot, show the training error vs. number of epochs trained for each of these values of η. What happens as η changes?

Solution F: *The loss vs. number of epochs for each value of the learning rate η is shown in the figure below. When η is smaller, convergence takes more epochs.*

The following problems consider SGD with the larger, higher-dimensional dataset, `sgd_data.csv`. The file has a header denoting which columns correspond to which values. For these problems, use the Jupyter notebook `1_notebook_part2.ipynb`.

For your implementation of problems F–H, **do** consider the bias term using your answer to problem A.

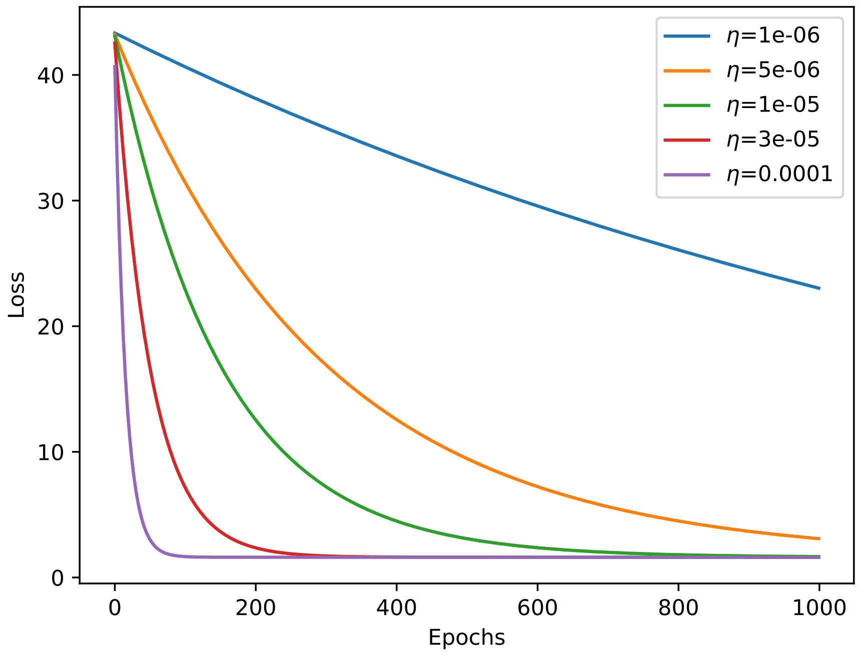

Figure 1: *Loss vs. number of epochs for each value of η for dataset 1.*

Problem F [6 points]: Use your SGD code with the given dataset, and report your final weights. Follow the guidelines below for your implementation:

- Use $\eta = e^{-15}$ as the step size.
- Use $\mathbf{w} = [0.001, 0.001, 0.001, 0.001]$ as the initial weight vector and $b = 0.001$ as the initial bias.
- Use at least 800 epochs.
- You should incorporate the bias term in your implementation of SGD and do so in the vector style of problem A.
- Note that for these problems, it is no longer necessary for the `SGD` function to store the weights after all epochs; you may change your code to only return the final weights.

Solution F: *See code. The final bias is (approximately) $b = -0.227$ and the final weights are (approximately)* $\mathbf{w} = [-5.942, 3.944, -11.724, 8.786]$.

Problem G [2 points]: Perform SGD as in the previous problem for each learning rate η in

$$\{e^{-10}, e^{-11}, e^{-12}, e^{-13}, e^{-14}, e^{-15}\},$$

and calculate the training error at the beginning of each epoch during training. On a single plot, show training error vs. number of epochs trained for each of these values of η. Explain what is happening.

Solution G: *See the code in `1_notebook_part2_sol.ipynb`. The loss vs. number of epochs for each value of the learning rate η is shown in the figure below. When η is smaller, convergence takes more epochs. In fact, we can see that for $\eta = e^{-18}$, 800 epochs is not enough for the algorithm to converge.*

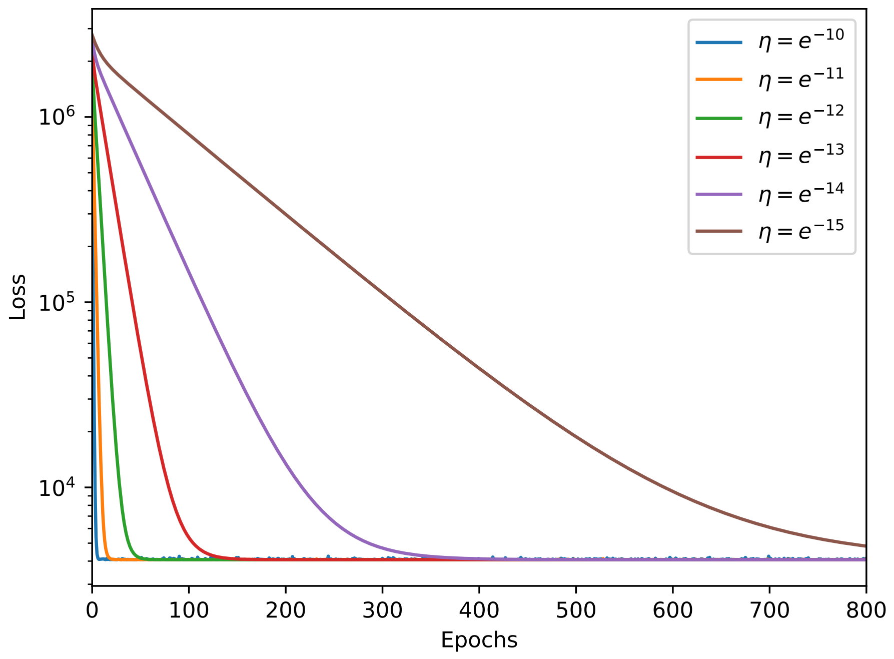

Figure 2: *Loss vs. number of epochs for each value of η.*

Problem H [2 points]: The closed-form solution for linear regression with least squares is

$$\mathbf{w} = \left(\sum_{i=1}^{N} \mathbf{x_i} \mathbf{x_i}^\intercal \right)^{-1} \left(\sum_{i=1}^{N} \mathbf{x_i} y_i \right).$$

Compute this analytical solution. Does the result match up with what you got from SGD?

Solution H: *For the larger learning rates where SGD actually converges, the result matches up reasonably well.*

Answer the remaining questions in 1–2 short sentences.

Problem I [2 points]: Is there any reason to use SGD when a closed-form solution exists?

Solution I: *Yes, it may be more computationally efficient to use SGD compared to the closed-form solution, especially if there are memory constraints and the dataset is extremely large. It is also easier to parallelize SGD*

Problem J [2 points]: Based on the SGD convergence plots that you generated earlier, describe a stopping condition that is more sophisticated than a pre-defined number of epochs.

Solution J: *The SGD convergence plots show that the training loss asymptotically approaches a minimum value. We can set a threshold such that if the improvement in the training loss is less than this threshold, we stop training. Better yet, we can hold out a fraction of the dataset from the training to compute a validation loss, and stop the training if the validation loss has not improved in the last N epochs (i.e. early stopping with a patience of N).*

Problem K [2 points]: How does the convergence behavior of the weight vector differ between the perceptron and SGD algorithms?

Solution K: *For SGD, the process is guaranteed to converge if the loss function is convex. SGD also converges relatively smoothly and directly. For the perceptron algorithm, it is not guaranteed to ever converge, and each update can change the weights quite dramatically.*

2 Neural networks vs. boosted decision trees [45 Points]

In this problem, you will compare the performance of neural networks and boosted decision trees for binary classfication on a tabular dataset, namely the MiniBooNE dataset: `https://archive.ics.uci.edu/ml/datasets/MiniBooNE+particle+identification`.

This dataset is taken from the MiniBooNE experiment and is used to distinguish electron neutrinos (signal) from muon neutrinos (background) The dataset contains 130,065 samples with 50 features and a single binary label. We will randomly split the dataset into training (80%) and testing (20%) subsets.

We will use `2_notebook_part1.ipynb` for parts A and B and `2_notebook_part1.ipynb` for parts C, D, and E.

Problem A [15 points]: Using the MiniBooNE dataset and XGBoost, train a boosted decision tree on the training dataet. Use the Scikit-learn API `xgboost.XGBClassifier`. For an initial choice of hyperparameters use 100 trees (`n_estimators`), maximum tree depth (`max_depth`) of 10, learning rate (`learning_rate`) of 0.1, `colsample_bytree` of 0.8, and `subsample` of 0.8.

Plot the receiver operating characteristic (ROC) curve using the testing dataset. What area under the curve (AUC) and accuracy do you achieve "out of the box"?

Solution A: *See the code in `2_notebook_part1_sol.ipynb`. The ROC curve for the BDT is shown in the figure below. We can achieve an AUC of 98.55% and an accuracy of 94.63% out of the box.*

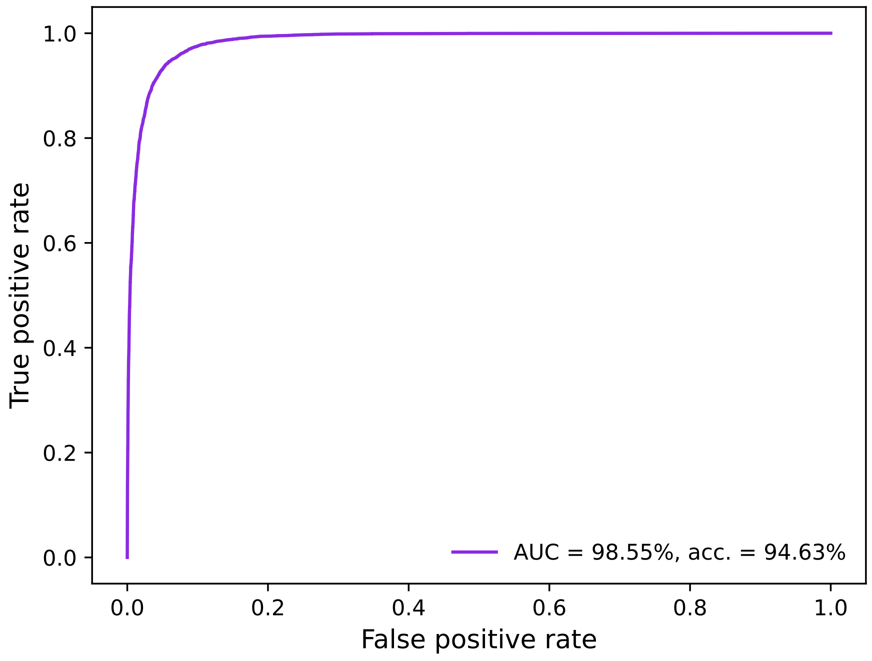

Figure 3: *ROC curve for the BDT.*

Problem B [5 points]: Plot the F-score for all the 10 "most important" features using `xgboost.plot_importance`. Which feature is the most important?

Plot this feature using the testing dataset in a 1D histogram separately for signal and background. For the histogram binning, use 100 bins from the minimum value of this feature to the maximum value of this feature in the testing dataset. What do you notice about this feature?

Solution B: *The F-scores for the top 10 input features (left) and the distributions of the most important feature for signal and background (right) are shown in the figure below. We can see that there are a few outlier values at -999.*

This represents a "null" value meaning this input variable cannot be computed.

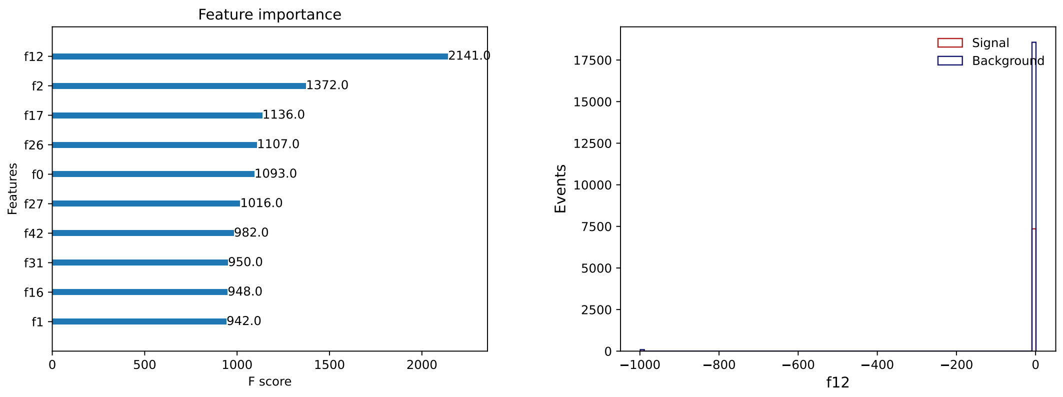

Figure 4: *F-scores for the top 10 input features (left) and the most important feature plotted (right).*

Problem C [15 points]: Using the MiniBooNE dataset and the Keras Model API, train a neural network with 3 hidden layers each with 128 units and `tanh` activations. The final layer should have sigmoid activation. Use the binary crossentropy loss function, the SGD optimizer with a learning rate of 0.01 (which is the default), and a batch size of 128. Train the model for 50 epochs.

Plot the receiver operating characteristic (ROC) curve using the testing dataset. What AUC and accuracy do you achieve "out of the box"?

Solution C: *See the code in `2_notebook_part2_sol.ipynb`. The ROC curve for the neural network is shown in the figure below. We can achieve an AUC of 90.77% and an accuracy of 86.44% out of the box, which is not as good as the BDT.*

Problem D [5 points]: Swap out the `tanh` activations for ReLU activations, while keeping everything else the same. Does the network train effectively? Why or why not?

Solution D: *No, the network does not train effectively. Quite early during the training, the loss becomes "not a number" (NaN), which prevents any further training from occurring. This is most likely related to the fact that the input feature values can be large in magnitude, (e.g. -999) and the weights are not initialized properly to reflect that.*

Problem E [5 points]: Now, we will make two minor changes to the network with ReLU activations: preprocessing and the optimizer.

For the feature preprocessing use `sklearn.preprocessing.StandardScaler` to standardize the input features. Note you should use fit the standard scaler to the training data *only* and apply it to both the

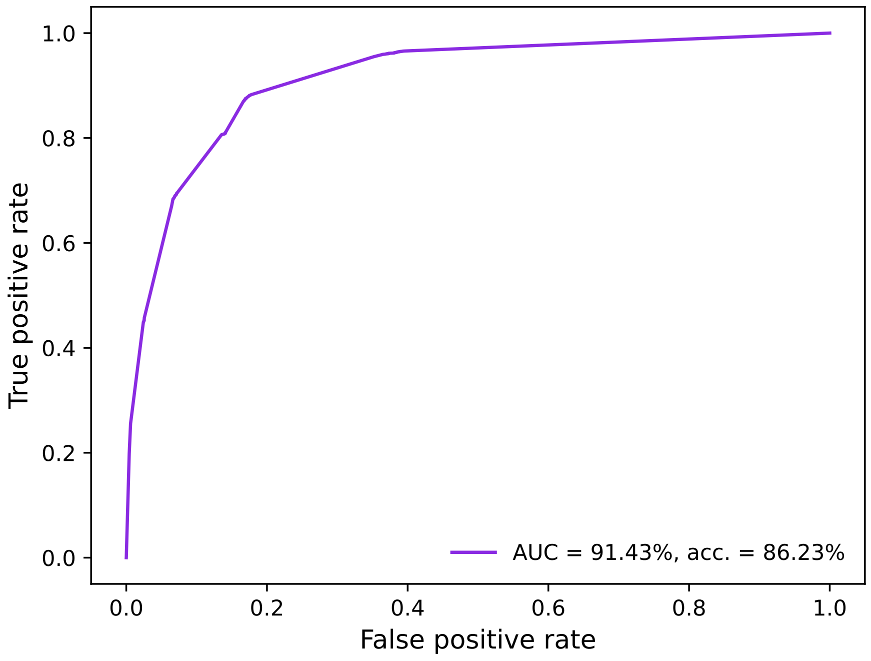

Figure 5: *ROC curve for the NN.*

training and testing data. For the optimizer, use Adam with a learning rate of 0.001 (which is the default) instead of SGD. Train the model for 50 epochs.

Plot the receiver operating characteristic (ROC) curve using the testing dataset. What AUC and accuracy do you achieve now? Is it comparable to the BDT?

Solution C: *The ROC curve for the neural network using the standard scaler and Adam optimizer is shown in the figure below. We can achieve an AUC of 97.88% and an accuracy of 93.08%, which is now much closer to what we achieved with the BDT.*

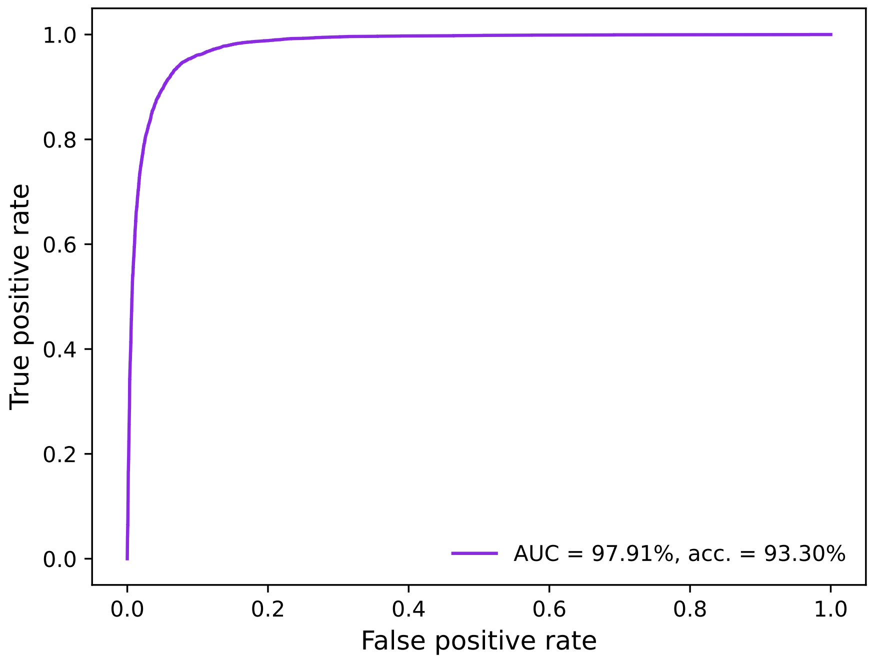

Figure 6: *ROC curve for the NN using the standard scaler and Adam optimizer.*