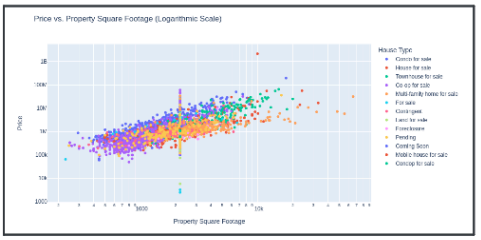
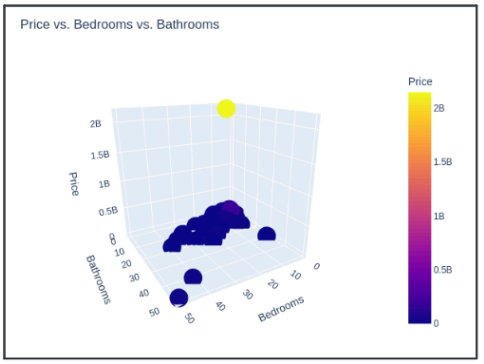
CSCI-561 - Spring 2024 - Foundations of Artificial Intelligence

Homework 3

Due Monday April 15, 2024, 23:59:59 PST





1. Assignment Overview

In this homework assignment, you will implement a multi-layer perceptron (MLP) and use it to solve a classification task on real-world data from the New York housing market. Your algorithm will be implemented from scratch, using no external libraries other than Numpy (or similar); machine learning libraries are **NOT** allowed (e.g., Sklearn, TensorFlow, PyTorch, etc.). This dataset is publicly available, **but we modified it.** So only use our data.

2. Grading

Your final score for this project will be a combination of the following three items:

1. (70%) **Primary prediction task**: the bulk of your homework score will be determined by your model's performance relative to a benchmark model on the central classification task (see Section 4 for task details). In particular, the test accuracy of your model will be compared to the test accuracy of our model, and your grade will be determined by the following rubric:

    - < 50% baseline accuracy ⟶ 0% grade
    - >= 50% baseline accuracy ⟶ 20% grade
    - >= 60% baseline accuracy ⟶ 40% grade
    - >= 70% baseline accuracy ⟶ 60% grade
    - >= 80% baseline accuracy ⟶ 80% grade
    - >= 90% baseline accuracy ⟶ 100% grade

    That is, for each 10% bump in accuracy your model achieves relative to ours, your score for this section (which itself is worth 70% of the HW grade) will increase by 20%, so long as you've achieved 50% of the

baseline accuracy. The scoring is tiered in this way due to the relative difficulties of reaching each performance level; achieving the first 50% of baseline accuracy is about as hard as the final 10%, for instance. Your model will be scored according to the highest category for which it's eligible.

For clarity, the following is an example of how a submission will be graded:

- Ten different train/test splits of the New York Housing dataset are generated. Then for each split:
- Both your submitted model and our baseline model are trained, from scratch, on the training set.
- Both models are then evaluated on the test set, producing the test classification accuracies **A** for your model and **B** for our baseline.
- The percentage of baseline accuracy for your model is then determined by calculating **A / B**. For example, if your model reached 70% classification accuracy and our baseline reached 85%, your relative score is 70 / 85 = 0.824, or 82.4% of baseline accuracy.
- After all ten train/test splits are processed, your average percentage of baseline accuracy is computed.
- Your average relative accuracy is mapped to its corresponding rubric score. In the example, 80 <= 82.4 < 90, so the submission would receive a score of 80% for this grading section (which again is worth 70% of the final HW grade).

Important: feedback prior to deadline. The above grading process describes how we will generate your submission's final score *after the deadline* has passed. Prior to the deadline, we will provide 5 sample train/test splits, in **`resources/asnlib/publicdata/dev/`** on Vocareum. By default, **the first split will be used each time you submit** to Vocareum. You can instruct the Vocareum submission script to use another one by adding the following commented-out line anywhere in your code:

- In Python:
  ```
  # USE_DATASET_SPLIT x
  ```
- In C++ or Java:
  ```
  // USE_DATASET_SPLIT x
  ```

Where **`x`** is a number from 1 to 5 (inclusive). All splits are ~70% training / ~30% test data.

Note regarding double-dipping: There is some double-dipping in the process described above (using some data samples for training or validation that will later also be in the test set). Indeed, because this dataset is publicly available, it is not possible for us to carve out a pure test set that can never be seen prior to the deadline. Although not acceptable if you were to publish a research paper based on your work, this is ok for a homework project. Please see Section 4b for more guidance; in particular, **loading pre-trained weights is strictly prohibited**, you need to train and test from scratch on any given split.

2. (30%) **Hyperparameter report**: you will also receive credit for providing a **`report.pdf`** file in your Vocareum submission that indicates any hyperparameter exploration you have done during development. This file should include the validation scores you received for various network settings you've tried, such as number of layers & nodes per layer (e.g., deep vs wide networks), learning rate, activation function (e.g., RELU, sigmoid, etc). The format is up to you, this will be graded by a human. Typically, you would briefly explain what hyperparameter you explored, and then, for each value, show the obtained results on the 5 data splits. See section 6 for guidance.

3. Data Description

You will train your model on data from the New York housing market. This dataset includes 4801 real estate sales in the region, each with the following 17 attributes:

- **BROKERTITLE**: *Title of the broker*
- **TYPE**: *Type of the house*
- **PRICE**: *Price of the house*
- **BEDS**: *Number of bedrooms*
- **BATH**: *Number of bathrooms*
- **PROPERTYSQFT**: *Square footage of the property*
- **ADDRESS**: *Full address of the house*
- **STATE**: *State of the house*
- **MAIN_ADDRESS**: *Main address information*
- **ADMINISTRATIVE_AREA_LEVEL_2**: *Administrative area level 2 information*
- **LOCALITY**: *Locality information*
- **SUBLOCALITY**: *Sublocality information*
- **STREET_NAME**: *Street name*
- **LONG_NAME**: *Long name*
- **FORMATTED_ADDRESS**: *Formatted address*
- **LATITUDE**: *Latitude coordinate of the house*
- **LONGITUDE**: *Longitude coordinate of the house*

Your goal is to predict the number of bedrooms (the "BEDS" feature) for a given property, using any of the other 16 available features. The dataset will be provided to you in a CSV-like format (see exact format in Section 4), so you can perform local analyses and design feature transformations as needed (e.g., one-hot encoding certain discrete-valued features).

4. Task description

Your task is to implement a multi-layer neural network learner (see Section 5 for additional details), that will do the following:

1. Construct and train a neural network classifier using provided labeled training data,
2. Use the learned classifier to classify unlabeled test inputs,
3. Output the predictions of your classifier on the test data into a file in the ***same*** directory,
4. ***Finish in 5 minutes on Vocareum (for both training your model and making predictions)***.

For step #1, your program will read from the provided `train_data.csv` and `train_label.csv` files in the current directory, providing the input features and output labels, respectively, to use for training. Once your model has finished training, your program must read `test_data.csv` to obtain the test set data points, and produce predictions by evaluating your model for each. Your program will then write a single `output.csv` file containing the corresponding predictions, one per line.

4a. File structure

Your program must be named `homework.py`, `homework.cpp`, or `homework.java` as in previous homeworks. It will be compiled and then invoked without any command-line arguments. Three files will be present in

the current directory when your program is invoked: `train_data.csv` and `train_label.csv`, for the input features and output labels to use for training. After training, your program must read `test_data.csv` to obtain the test set data points, and produce predictions in a new file `output.csv`

The format of `*_data.csv` looks like:

BROKERTITLE, TYPE, PRICE, … (16 column labels)
$x_1^{(1)}$, $x_1^{(2)}$, …
$x_2^{(1)}$, $x_2^{(2)}$, …
…

Where $\mathbf{x}_i^{(j)}$ is the $\mathbf{j}^{th}$ feature of the $\mathbf{i}^{th}$ data point. There will be exactly 16 comma-separated entries in each line, corresponding to the features described in Section 3 except for BEDS (which your network will try to predict). Columns will always be in the same order (see samples on Vocareum). The `train_label.csv` and ***your*** `output.csv` files will look like

BEDS
y_1
y_2
…

where $\mathbf{y}_i$ is the integer-valued label for data point $\mathbf{x}_i$. Thus, there is a single column indicating the predicted class label for each unlabeled sample in the input test file, i.e., the predicted number of bedrooms.

The format of your `output.csv` file is crucial. Your output file must have this ***name and format*** so that it can be parsed correctly to compare with true labels by the auto-grading scripts. This file should be written to your working path.

4b. Implementation constraints

As mentioned above, ***NumPy is the only external library you can use in your implementation (or equivalent numerical computing-only library in non-Python languages).*** By *external* we mean *outside the standard library* (e.g., in Python, `random`, `os`, etc are fine to use). No component of the neural network implementation can leverage a call to an external ML library; you must implement the algorithm yourself, from scratch.

The maximum running time to train and test your model is 5 minutes on Vocareum. The size of the training set will remain roughly fixed across Vocareum evaluations, providing ample opportunity to tune your submission according to its efficiency (e.g., by adjusting the number of training epochs). Here it is particularly important to vectorize your implementation (see Section 6).

Directly loading pre-trained weights of the neural network is prohibited. Your model must be trained from scratch using the data provided to your script's working path. Efficient, well-tuned implementations can achieve high scores using the time constraints and computational resources provided by the Vocareum environment.

Failure to adhere to any of the above constraints will result in your submission receiving no credit.

5. Model description

The model you will implement is a vanilla feed-forward neural network, possibly with many hidden layers (see Figure 2 for a generic depiction). Your network should output a single value, and may have variable input size depending on your design (e.g., whether you transform any of the 16 provided features). Beyond this, there are no constraints on your model's structure; it is up to you to decide what activation function, number of hidden layers, number of nodes per hidden layer, etc, your model should use.

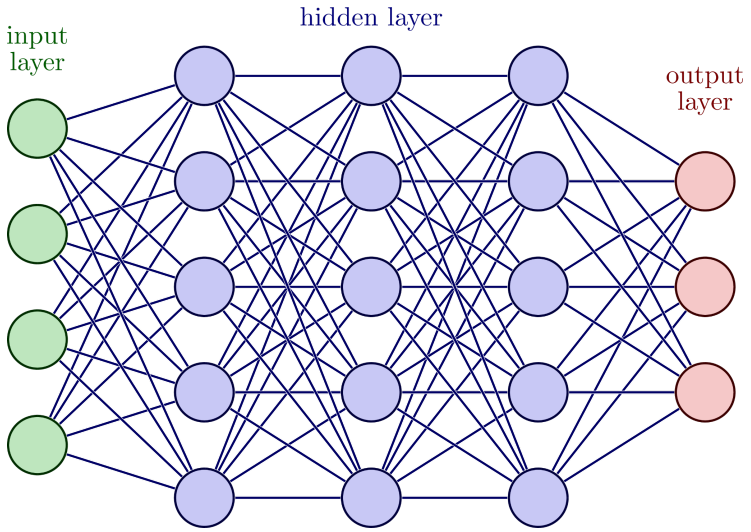


Figure 2: Diagram of an example neural network with 3 hidden layers.

There are many hyperparameters you will likely need to tune to get better performance. These can be hard-coded by you in your program (possibly after structured exploration of your hyperparameter space), or selected through a cross validation process dynamically (in the latter case, be wary of runtime limits). A few example hyperparameters are as follows:

- **Learning rate**: step size to update weights (*e.g. weights = weights - learning * grads*), different optimizers have different ways to use learning rate.
- **Mini-batch size**: number of samples processed each time before the model is updated. The mini-batch size is some value smaller than the size of the dataset that effectively splits it into smaller chunks during training. Using batches to train your network is highly recommended.
- **Number of the epochs**: the number of complete passes through the training dataset (*e.g. if you have 1000 samples, 20 epochs mean you loop through these 1000 samples 20 times).*
- **Number of hidden layers & number of units in each hidden layer**: these settings constitute the overall structure of your model. Unnecessarily deep or wide networks may negatively impact your model's performance given the time constraints. Here you need to find a proper tradeoff between feasible time to convergence and model expressivity.

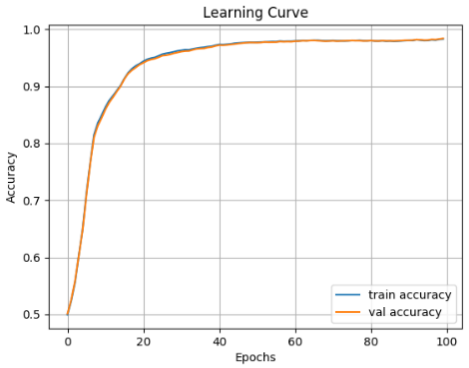
6. Implementation and report.pdf guidance

Here are a few suggestions you might want to consider during your implementation:

1. **Think about how to deal with text data.** In the dataset, some entries are numeric (e.g., PRICE), others are text with only a few alternative choices (e.g., TYPE), and others are more open English text (e.g.,

BROKERTITLE, ADDRESS). You need to think about how to use text-based inputs in your neural network. This will likely require some custom encoding of your choice.

2. **Train your model using mini-batches**: there are many good reasons to use mini-batches to train your model (instead of individual points or the entire dataset at once), including benefits to performance and convergence.
3. **Initialize weights and biases**: employ a proper random initialization scheme for your weights and biases. This can have a large impact on your final model.
4. **Use backpropagation:** you should be using backpropagation along with a gradient descent-based optimization algorithm to update your network's weights during training.
5. **Vectorize your implementation:** vectorizing your implementation can have a large impact on performance. Use vector/matrix operations when possible instead of explicit programmatic loops.
6. **Regularize your model**: leverage regularization techniques to ensure your model doesn't overfit the training data and keeps model complexity in check.
7. **Plot your learning curve:** plotting your train/test accuracy after each epoch is a quick and helpful way to see how your network is performing during training. Here you **are allowed** to use external plotting libraries, but worth noting that you should likely remove them prior to submission for performance reasons. The figure on the right shows a generic example of such a plot; your plot(s) may look different.
8. **Putting it all together**: see Figure 3 on the next page for a basic depiction of an example training pipeline. Note that this diagram lacks detail and is only meant to provide a rough outline for how your training loop might look.



While recommended, the use of these suggestions in your implementation is not explicitly required. Your grade will be determined by your model's performance as described in the grading section.

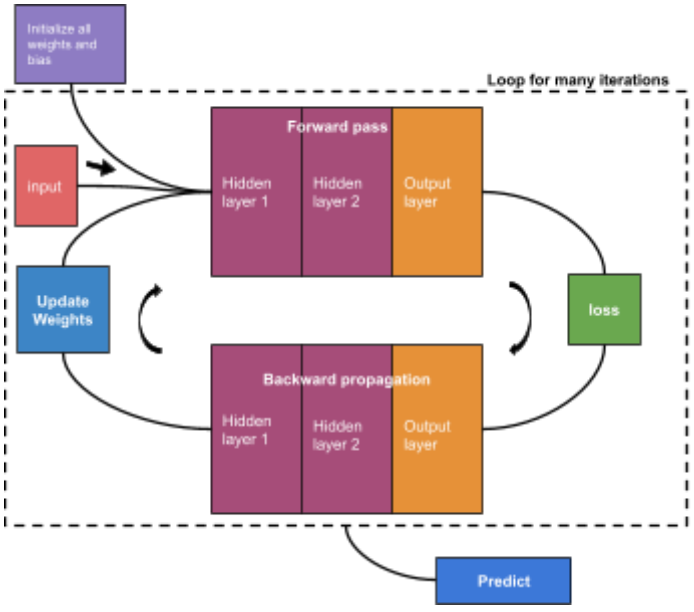


Figure 3: Diagram depicting the basic components of the training process.

7. Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe, you are urged to err on the side of caution. Do not copy work from another student or off the web. Keep in mind that sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

- **Do not copy** code or written material from another student. Even single lines of code should not be copied.
- **Do not collaborate** on this assignment. The assignment is to be solved individually.
- **Do not copy** code off the web. This is easier to detect than you may think.
- **Do not share** any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones that are given.
- **Do not copy** code from past students. We keep copies of past work to check for this. Even though this project differs from those of previous years, do not try to copy from the homework of previous years.
- **Do not ask Piazza about** how to implement some function for this homework, or how to calculate something needed for this homework.
- **Do not post your code on Piazza** asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
- **Do not post test cases on Piazza** asking for what the correct solution should be.
- **Do** ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.
- **DO NOT USE ANY** existing machine learning library such as Tensorflow, Pytorch, Scikit-Learn, etc. Violation will cause a penalty to your credit.