Project 4: Machine Learning

CSCI 360 Fall 2022

Released: April 11, 2022
Due: April 29, 2022

Contents

In this project you will build a neural network to classify digits and more!

Introduction

This project will be an introduction to machine learning.

The code for this project contains the following files, available from vocareum.

Files you will edit:	
`models.py`	Perceptron and neural network models for a variety of applications
`gridSearch.py`	A file to implement grid search.
`mnist_config.py`	A file to set the configuration for the digit classification task
`evaluation.py`	A file to to implement evaluation metrics
`survey_hyperparameters.py`	A file to set the hyperparameters to search over for the survey data
`analysis.txt`	Your analysis for the final question.
Files you should read but NOT edit:	
`nn.py`	Neural network mini-library
Files you can ignore:	
`autograder.py`	Project autograder
`backend.py`	Backend code for various machine learning tasks
`data`	Datasets for digit classification and language identification
`submission_autograder.py`	Submission autograder (generates tokens for submission)

Files to Edit and Submit: You will need to upload all files designated "Files you will edit" above. Please do not change the other files in this distribution.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: You know the drill. Don't copy your classmates' codes. We will know.

Proper Dataset Use: Part of your score for this project will depend on how well the models you train perform on the test set included with the autograder. We do not provide any APIs for you to access the test set directly. Any attempts to bypass this separation or to use the testing data during training will be considered **cheating**.

Credits

This project was adapted from a 2019 CS188 project from Berkeley.[1]

[1] https://inst.eecs.berkeley.edu/ cs188/sp19/project5.html

Setup

For this project, you will need to install the following libraries (make sure your conda environment is activated first):

numpy, which provides support for large multi-dimensional arrays

```
conda install numpy
```

matplotlib, a 2D plotting library

```
conda install matplotlib
```

pandas, a data analysis and manipulation tool,

```
conda install pandas
```

sklearn, a machine learning tool for python, also contains some data processing tools,

```
conda install scikit-learn
```

To test that everything has been installed, run:

```
python autograder.py --check-dependencies
```

If numpy and matplotlib are installed correctly, you should see a window pop up where a line segment spins in a circle.

Provided Code (Part I)

For this project, you have been provided with a neural network mini-library (`nn.py`) and a collection of datasets (`backend.py`).

The library in `nn.py` defines a collection of node objects. Each node represents a real number or a matrix of real numbers. Operations on Node objects are optimized to work faster than using Python's built-in types (such as lists).

Here are a few of the provided node types:

- `nn.Constant` represents a matrix (2D array) of floating point numbers. It is typically used to represent input features or target outputs/labels. Instances of this type will be provided to you by other functions in the API; you will not need to construct them directly
- `nn.Parameter` represents a trainable parameter of a perceptron or neural network
- `nn.DotProduct` computes a dot product between its inputs
- `nn.Sigmoid` computes the element-wise sigmoid function of its inputs: $\frac{1}{1+e^{-z}}$
- `nn.ScalarMatrixMultiply` multiplies every element in the matrix by a scalar.

Additional provided functions:

- `nn.as_scalar` can extract a Python floating-point number from a node.

When training a perceptron or neural network, you will be passed a dataset object. You can retrieve batches of training examples by calling `dataset.iterate_once(batch_size)`:

```
for x, y in dataset.iterate_once(batch\_size):
...
```

For example, let's extract a batch of size 1 (i.e. a single training example) from the perceptron training data:

```
>>> batch_size = 1
>>> for x, y in dataset.iterate_once(batch_size):
...     print(x)
...     print(y)
...     break
...
<Constant shape=1x3 at 0x11a8856a0>
<Constant shape=1x1 at 0x11a89efd0>
```

The input features `x` and the correct label `y` are provided in the form of `nn.Constant` nodes. The shape of `x` will be `batch_size x num_features`, and the shape of `y` is `batch_size x num_outputs`. Here is an example of computing a dot product of `x` with itself, first as a node and then as a Python number.

```
>>> nn.DotProduct(x, x)
<DotProduct shape=1x1 at 0x11a89edd8>
>>> nn.as_scalar(nn.DotProduct(x, x))
1.9756581717465536
```

Question 1 (6 points): Perceptron

Before starting this part, be sure you have `numpy` and `matplotlib` installed!

In this part, you will implement a binary perceptron. Your task will be to complete the implementation of the PerceptronModel class in `models.py`.

For the perceptron, the output labels will be either 1 or -1, meaning that data points `(x, y)` from the dataset will have `y` be a `nn.Constant` node that contains either 1 or -1 as its entries.

We have already initialized the perceptron weights self.w to be a 1 x *dimensions* parameter node. The provided code will include a bias feature inside x when needed, so you will not need a separate parameter for the bias.

Your tasks are to:

Implement the `run(self, x)` method. This should compute the dot product of the stored weight vector and the given input, returning an `nn.DotProduct` object. Implement `get_prediction(self, x)`,

which should return 1 if the dot product is non-negative or -1 otherwise. You should use `nn.as_scalar` to convert a scalar Node into a Python floating-point number. Write the `train(self)` method. This should repeatedly loop over the data set and make updates on examples that are misclassified. Use the update method of the `nn.Parameter` class to update the weights. When an entire pass over the data set is completed without making any mistakes, 100% training accuracy has been achieved, and training can terminate. In this project, the only way to change the value of a parameter is by calling `parameter.update(direction, multiplier)`, which will perform the update to the weights:

`weights` $\leftarrow$ `weights` $+$ `direction` $\times$ `multiplier`

The direction argument is a `Node` with the same shape as the parameter, and the multiplier argument is a Python scalar.

To test your implementation, run the autograder:

```
python autograder.py -q q1
```

Note: the autograder should take at most 20 seconds or so to run for a correct implementation. If the autograder is taking forever to run, your code probably has a bug.

Question 2 (6 points): Logistic Regression

For this question, you will implement a logistic regression model

You will need to complete the implementation of the LogisticRegressionModel class in `models.py`.

Your tasks are to:

- Implement `LogisticRegressionModel.__init__` with any needed initialization
- Implement `LogisticRegressionModel.run` to return a batch_size x 1 node that represents your model's prediction.
- Implement `LogisticRegressionModel.get_prediction` to return the class (1 or 0) for a given input
- Implement `LogisticRegressionModel.calculate_log_likelihood_gradient` to return the gradient of the log likelihood of a single datapoint with respect to the input parameters
- Implement `LogisticRegressionModel.train`, should train your model to maximize the likelihood of the data using stochastic gradient ascent.

Implement stochastic gradient ascent (1 random sample at a time) for the logistic regression. The logistic regression slides and discussion section will help you out here.

We provide you with a `nn.Sigmoid` function that returns the element-wise sigmoid of a matrix (meaning it returns a matrix of the same shape, where each element has been transformed using the sigmoid function).

Jesse's logistic regression discussion notes and video may be helpful for this section. The gradient of the log likelihood for the full dataset is:

$$\sum_{i=1}^{N} \left(y_i - \phi(w \cdot x_i)\right) x_i \tag{1}$$

How does this equation change for a single datapoint?

We provide a `dataset.pick_random(batch_size)` function that returns a random data sample from the dataset.

You will be required to fit the logistic regression to a randomly generated data. For which the log-likelihood of the data given your model should be at least -60.0. You can run the autograder using:

```
python autograder.py -q q2
```

It is possible that your model may be correct but gets just under -60.0. If your implementation is correct, it should do this pretty rarely so you can just run the autograder again.

Note: For the logistic regression, you are not required to include a bias term in the calculation of Z. $Z = w \cdot x$ is good enough. You do not need $Z = w \cdot x + \beta$. You are welcome to include it if you understand how to incorporate it into training.

Neural Network Tips

In the remaining parts of the project, you will implement the following models:

Q3: Handwritten Digit Classification

Q5: Mental Health Prediction Network

Building Neural Nets

Throughout the applications portion of the project, you'll use the framework provided in `nn.py` to create neural networks to solve a variety of machine learning problems. A simple neural network has layers, where each layer performs a linear operation (just like perceptron). Layers are separated by a non-linearity, which allows the network to approximate general functions. We'll use the ReLU operation for our non-linearity, defined as

$$relu(x) = max(x, 0) \tag{2}$$

For example, a simple two-layer neural network for mapping an input row vector `x` to an output vector `f(x)` would be given by the function:

$$f(x) = relu(xW_1 + b_1)W_2 + b_2 \tag{3}$$

where we have parameter matrices W_1 and W_2 and parameter vectors b_1 and b_2 to learn during gradient descent. W_1 will be an $i \times h$ matrix, where i is the dimension of our input vectors x, and h is the hidden layer size. b_1 will be a size h vector. We are free to choose any value we want for the hidden size (we will just need to make sure the dimensions of the other matrices and vectors agree so that we can perform the operations). Using a larger hidden size will usually make the network more powerful (able to fit more training data), but can make the network harder to train (since it adds more parameters to all the matrices and vectors we need to learn), or can lead to overfitting on the training data.

We can also create deeper networks by adding more layers, for example a three-layer net:

$$f(x) = relu(relu(x \times W_1 + b_1)W_2 + b_2)W3 + b3 \tag{4}$$

Note on Batching: For efficiency, you will be required to process whole batches of data at once rather than a single example at a time. This means that instead of a single input row vector x with size i, you will

be presented with a batch of b inputs represented as a $b \times i$ matrix X. We provide an example for linear regression to demonstrate how a linear layer can be implemented in the batched setting.

Note on Randomness: The parameters of your neural network will be randomly initialized, and data in some tasks will be presented in shuffled order. Due to this randomness, it's possible that you will still occasionally fail some tasks even with a strong architecture – this is the problem of local optima! This should happen very rarely, though – if when testing your code you fail the autograder twice in a row for a question, you should explore other architectures.

Practical tips: Designing neural nets can take some trial and error. Here are some tips to help you along the way:

- Be systematic. Keep a log of every architecture you've tried, what the hyperparameters (layer sizes, learning rate, etc.) were, and what the resulting performance was. As you try more things, you can start seeing patterns about which parameters matter. If you find a bug in your code, be sure to cross out past results that are invalid due to the bug.
- Start with a shallow network (just two layers, i.e. one non-linearity). Deeper networks have exponentially more hyperparameter combinations, and getting even a single one wrong can ruin your performance. Use the small network to find a good learning rate and layer size; afterwards you can consider adding more layers of similar size.
- If your learning rate is wrong, none of your other hyperparameter choices matter. You can take a state-of-the-art model from a research paper, and change the learning rate such that it performs no better than random. A learning rate too low will result in the model learning too slowly, and a learning rate too high may cause loss to diverge to infinity. Begin by trying different learning rates while looking at how the loss decreases over time.
- Smaller batches require lower learning rates. When experimenting with different batch sizes, be aware that the best learning rate may be different depending on the batch size. Refrain from making the network too wide (hidden layer sizes too large). If you keep making the network wider accuracy will gradually decline, and computation time will increase quadratically in the layer size – you're likely to give up due to excessive slowness long before the accuracy falls too much. The full autograder for all parts of the project takes 2-12 minutes to run with staff solutions; if your code is taking much longer you should check it for efficiency.
- If your model is returning Infinity or NaN, your learning rate is probably too high for your current architecture.
- Recommended values for your hyperparameters:

  Hidden layer sizes: between 10 and 512

  Batch size: between 1 and the size of the dataset. For Q3, we require that total size of the dataset be evenly divisible by the batch size.

  Learning rate: between 0.001 and 1.0

  Number of layers: between 2 and 4

Provided Code (Part II)

Here is a full list of nodes available in `nn.py`. You will make use of these in the remaining parts of the assignment:

- `nn.Constant` represents a matrix (2D array) of floating point numbers. It is typically used to represent input features or target outputs/labels. Instances of this type will be provided to you by other functions in the API; you will not need to construct them directly
- `nn.Parameter` represents a trainable parameter of a perceptron or neural network. All parameters must be 2-dimensional.
  Usage: `nn.Parameter(n, m)` constructs a parameter with shape `n` × `m`.
- `nn.Add` adds matrices element-wise
  Usage: `nn.Add(x, y)` accepts two nodes of shape `batch_size` × `num_features` and constructs a node that also has shape `batch_size` × `num_features`.
- `nn.AddBias` adds a bias vector to each feature vector
  Usage: `nn.AddBias(features, bias)` accepts features of shape `batch_size` × `num_features` and bias of shape 1 × `num_features`, and constructs a node that has shape `batch_size` × `num_features`.
- `nn.Linear` applies a linear transformation (matrix multiplication) to the input
  Usage: `nn.Linear(features, weights)` accepts features of shape `batch_size` × `num_input_features` and weights of shape `num_input_features` × `num_output_features`, and constructs a node that has shape `batch_size` × `num_output_features`.
- `nn.ReLU` applies the element-wise Rectified Linear Unit nonlinearity `relu(x)=max(x,0)`. This nonlinearity replaces all negative entries in its input with zeros.
  Usage: `nn.ReLU(features)`, which returns a node with the same shape as the input.
- `nn.SquareLoss` computes a batched square loss, used for regression problems
  Usage: `nn.SquareLoss(a, b)`, where `a` and `b` both have shape `batch_size`×`num_outputs`.
- `nn.SoftmaxLoss` computes a batched softmax loss, used for classification problems
  Usage: `nn.SoftmaxLoss(logits, labels)`, where logits and labels both have shape `batch_size` x `num_classes`. The term "logits" refers to scores produced by a model, where each entry can be an arbitrary real number. The labels, however, must be non-negative and have each row sum to 1. Be sure not to swap the order of the arguments! Do not use nn.DotProduct for any model other than the perceptron.

The following methods are available in `nn.py`:

- `nn.gradients` computes gradients of a loss with respect to provided parameters.
  Usage: `nn.gradients(loss, [parameter_1, parameter_2, ..., parameter_n])` will return a list `[gradient_1, gradient_2, ..., gradient_n]`, where each element is an `nn.Constant` containing the gradient of the loss with respect to a parameter.
- `nn.as_scalar` can extract a Python floating-point number from a loss node. This can be useful to determine when to stop training. Usage: `nn.as_scalar(node)`, where node is either a loss node or has shape 1 × 1.

Example: Linear Regression

As an example of how the neural network framework works, let's fit a line to a set of data points. We'll start four points of training data constructed using the function $y = 7x_0 + 8x_1 + 3$. In batched form, our data is:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 3 \\ 11 \\ 10 \\ 18 \end{bmatrix}$$

Suppose the data is provided to us in the form of `nn.Constant` nodes:

```
>>> x
<Constant shape=4x2 at 0x10a30fe80>
>>> y
<Constant shape=4x1 at 0x10a30fef0>
```

Let's construct and train a model of the form $f(x) = x_0 \times m_0 + x_1 \times m_1 + b$. If done correctly, we should be able to learn than $m_0 = 7$, $m_1 = 8$, and $b = 3$.

First, we create our trainable parameters. In matrix form, these are:

$$M = \begin{bmatrix} m_0 \\ m_1 \end{bmatrix} \text{ and } B = \begin{bmatrix} b \end{bmatrix}$$

Which corresponds to the following code:

```
m = nn.Parameter(2, 1)
b = nn.Parameter(1, 1)
```

Printing them gives:

```
>>> m
<Parameter shape=2x1 at 0x112b8b_208>
>>> b
<Parameter shape=1x1 at 0x112b8beb8>
```

Next, we compute our model's predictions for y:

```
xm = nn.Linear(x, m)
predicted_y = nn.AddBias(xm, b)
```

Our goal is to have the predicted y-values match the provided data. In linear regression we do this by minimizing the square loss:

$L = \frac{1}{N} \sum_{(x,y)} (y - f(x))^2$

We construct a loss node:

```
loss = nn.SquareLoss(predicted_y, y)
```

In our framework, we provide a method that will return the gradients of the loss with respect to the parameters:

```
grad_wrt_m, grad_wrt_b = nn.gradients(loss, [m, b])
```

Printing the nodes used gives:

```
>>> xm
<Linear shape=4x1 at 0x11a869588>
>>> predicted_y
<AddBias shape=4x1 at 0x11c23aa90>
>>> loss
<SquareLoss shape=() at 0x11c23a240>
>>> grad_wrt_m
<Constant shape=2x1 at 0x11a8cb_160>
>>> grad_wrt_b
<Constant shape=1x1 at 0x11a8cb588>
```

We can then use the update method to update our parameters. Here is an update for m, assuming we have already initialized a multiplier variable based on a suitable learning rate of our choosing:

```
m.update(grad_wrt_m, multiplier)
```

If we also include an update for b and add a loop to repeatedly perform gradient updates, we will have the full training procedure for linear regression.

Question 3 (6 points): Neural Networks and Digit Classification

For this question, you will implement a generic neural network class and then train it to classify hand-drawn digits from the MNIST dataset.

- Complete the implementation of the ClassificationModel class in `models.py`. You will design this class to instantiate a network with the parameters matching those given to the constructor.
- The return value from `ClassificationModel.run` should be a batch_size x output_dim node containing scores, where higher scores indicate a higher probability of the class being the correct one. Do not put a ReLU activation after the last layer of the network.
- `ClassificationModel.get_loss` should return the loss node computed for a batch of x and y datapoints. It should use `nn.SoftmaxLoss` as the loss function. Remember your model is trying to minimize the loss.
- `ClassificationModel.train` should train the model with the given arguments for the hyperparameters.
- Set the training hyperparameters in `mnist_config.py`

The dataset to use contains digit images. Each digit is of size 28 × 28 pixels, the values of which are stored in a 784 -dimensional vector of floating point numbers. Each output we provide is a 10-dimensional vector which has zeros in all positions, except for a one in the position corresponding to the correct class of the digit. The output will indicate the score for a digit belonging to a particular class (0-9).

In the file `mnist_config.py`, you will set the hyperparameters in `student_config` to those you want to use for mnist digit classification.

To receive points for this question, your model should achieve an accuracy of at least 97% on the test set. Note that the test grades you on test accuracy, while you only have access to validation accuracy - so if your validation accuracy meets the 97% threshold, you may still fail the test if your test accuracy does not meet the threshold. Therefore, it may help to set a slightly higher stopping threshold on validation accuracy, such as 97.5% or 98%.

To test your implementation, run the autograder:

```
python autograder.py -q q3
```

Question 4: Grid Search (4 points)

Grid search is a way of identifying the best hyperparameters in an organized fashion. Under grid search, you define a set of possible choices for each hyperparameter, and then evaluate every combination of those parameters, selecting the one that gives the best validation score.

		batch-size		
		16	32	64
learning-rate	1.0	0.1	0.3	0.3
	0.1	0.1	0.7	0.6
	0.01	0.1	0.1	0.1

Table 1: Table showing the results of running grid search. Colored cells represent the validation score, the green cell indicates the best score and therefore the best hyperparameters are batch size = 32 and learning rate = 0.1

For this question, you will modify the code in `gridSearch.py` to run grid search. You should modify

- `GridSearch.grid_search_configurations` to return a list of hyperparameter configurations to run
- `GridSearch.train_and_eval` to instantiate a ClassicationNeuralNetwork model based on the passed in hyperparameters config, train the model based on the hyperparameters config and the training dataset, and evaluate the trained model using `ClassicationNeuralNetwork.eval` with the validation dataset.
- `GridSearch.perform_grid_search` to call `self.train_and_eval` with each hyperparameter configuration. It should return the best model, score, and the configuration that led to it.

To test your implementation, run the autograder:

```
python autograder.py -q q4
```

Question 5: Evaluation Metrics (2 points)

For this question you will be implementing an evaluation function that returns three evaluation metrics: accuracy, precision, and recall.

In the file `evaluation.py`, implement the function `get_eval_metrics`.

Remember that accuracy is $\frac{\#correct}{\#total}$

Precision is $\frac{\#true\ positive}{\#true\ positive + \#false\ positive}$

Recall is $\frac{\#true\ positive}{\#truepositive + \#false\ negative}$.

You can test your implementation with:

```
python autograder.py -q q5
```

Question 6: Mental Health Treatment Model (2 points)

In this section we will have you work on an AI for Social Good problem. We will give you access to data derived from a mental health in tech survey with the goal of developing a model that can predict if a tech worker should seek mental health treatment/be guided towards such resources.

One of the pieces of data is whether an employee actually did seek treatment for mental health issues. We will be using this data as a proxy for whether a worker *should* seek mental health treatment. You will be training a neural network using grid search to do this task using treatment data as the label and other data from the survey as features. (We provide the dataset class).

This will use your previous implementations of classification model, grid search, and evaluation so make sure those are correct before doing this question.

Your job is to

- Set the hyperparameter search space in `survey_hyperparameters.py`.

Note: In Q3, you implemented a generic classification model that could do multi-class classification. The current problem is what is called a binary classification task (there are only two classes, the positive class and negative class). This can be implemented with a single score (like the perceptron and logistic regression you previously implemented) or with a separate score for the positive and negative class and the max score indicates the predicted class. That is what you will do in this problem in order to re-use the previous network design.

To pass this section your model should have a test accuracy > 78%.

You can evaluate your implementation with:

```
python autograder.py -q q6
```

Submission

To submit your code, upload all the files you edited to Vocareum, and **CLICK THE BLUE SUBMIT BUTTON**.