Project 5

Introduction

For Project 5, you will be implementing different statistical machine learning models. Specifically, you will be implementing logistic regression, linear regression, polynomial regression, perceptrons, and a neural network. Note that you will need to have *numpy* and *matplotlib* installed in order for the supporting py files and the autograder to work.

To ensure that you have the dependencies installed run:

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

You will be modifying the files *models.py.*

Please note no libraries outside of the ones specified above are to be used in this project. They have been written such that you will not need any libraries outside of the listed to complete the project. If you are unsure, please ask the TAs.

Machine Learning

This project is an introduction to the basics of statistical machine learning and neural networks, by first having you implement regression and the idea of a perceptron before moving onto implementing a basic neural net model. The design of this final neural net is up to you, the number of hidden layers, the size of the layers and whatever else you may decide to customize can be used to attempt to achieve the accuracy threshold of 96%.

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: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

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.

Questions

Question 1 (6 points): Logistic Regression

In this question, you will implement logistic regression to do binary classification. The classification labels are either +1 or -1 given a set of features of some dimensionality. The model definition is in models.py under LogisticRegressionModel class. NOTE: Do not use the nn.py modules in this question, it will not work until later.

Given the dimensionality of data as input for initialization, first generate a list of weights that are initialized. Also, set an appropriate learning rate (you will need to tune this once you get the model to work).

Next, complete the DotProduct and sigmoid function definitions. The sigmoid function implements the logistic function (they are identical in the context of machine learning). The output of the logistic function is a number between 0 and 1, which can be interpreted as a probability of being in class +1 (check this yourself). This also makes the logistic regression model a linear classifier (as an exercise for yourself, prove this). The logistic function is shown below.

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}} \,.$$

Next, complete the run and get_prediction functions, which interprets the probability output of the sigmoid function and returns the class membership predicted by the probability.

Finally, complete the train function. This utilizes gradient descent in order to tune the weights appropriately until convergence. To get a single datapoint, use for x,y in dataset.iterate_once(batch_size) as the for loop structure. Furthermore, x and y will be wrapped in an object that is not needed for this question. Thus, to get the x as a list, use x = nn.as_vector(x) and y = nn.as_scalar(y). When an entire pass over the data set is completed without making any mistakes, 100% training accuracy has been achieved, and training can terminate. Furthermore, you will need to compute the derivative of the loss function in order to compute an update step. Try to derive the derivative yourself (Hint: the gradient of the sigmoid/logistic function is a function of itself). The loss function for logistic regression is shown below.

$$\frac{\partial}{\partial w_i} Loss(\mathbf{w}) = \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2$$

Question 1.5 (6 points): Perceptron

Before diving into the question, here is a little information about the supporting files

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. 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.
- nn.Parameter represents a trainable parameter of a perceptron or neural network.
- nn.DotProduct computes a dot product between its inputs.

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): 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.

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 × 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.

First, 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.

Next, 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, dataset) 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. To change the value of a parameter is by calling parameter.update(direction, multiplier), which will perform the update to the weights: weights ← weights + direction · multiplier The direction argument is a Node with the same shape as the parameter, and the multiplier argument is a Python scalar.

Question 2 (8 points): Linear Regression

Before diving into the question, here is a little more information about the supporting files.

- 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.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.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).

For this question, you will be implementing a linear regression model using two Parameter nodes and a single linear node. This is the equivalent of a single neuron of a network. Initialize these parameter nodes. The weight parameter (b1..bn) should take an input of size “dimension” and output a value of dimension 1. The bias parameter (b0) should take an input of size 1 and output a value of size 1. This means that the regression task is to implement a scalar-valued function of the form y = f(x) = b0 + b1*x1 + b2*x2 + … + bn*xn. This is shown below.

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}$$

Next, complete the run function, which uses a linear node to compute a linear transformation and then the bias is added. The output of this is returned.

Next, complete the get_loss function, which returns the loss in order to do gradient descent. The loss function to use for linear regression is shown below (Hint: check the available loss functions in nn.py).

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

Complete the train(self, dataset) function which uses gradient descent to train the linear node and bias node to approximate the linear function. You will need to compute the gradients using nn.gradients and then update the weights and bias nodes using the update function.

It is important to note that linear regression can be solved in closed form via the following formula.

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N$$

Implement this for the 2D case in the closedFormSolution function. Note that this is specific to the 2D case, but there exists a more general closed form solution for any arbitrary dimension of input data.

Question 2.5 (6 points): Polynomial Regression

This question will require you to think creatively. You will need to figure out a way to implement polynomial regression for the 2-dimensional case. This question is simple if you use the right features, which need to be computed in computePolyFeatures. Hint: Think of an equivalent formulation for approximating a polynomial of order n. An example polynomial is shown below, and it should look similar to something you have seen before.

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

Note: there are some necessary unpacking and repacking needed when doing computePolyFeatures. Read the comments to get a jist of what to do.

Question 3 (6 points): MNIST Fashion Classification

Since you will now create a whole network, here is the list of the remaining nn.py functionality at your disposal.

- 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.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.SoftmaxLoss computes a batched softmax loss, used for classification problems.
    - Usage: nn.SoftmaxLoss(logits, labels), where logits and labels both have shape batch_size × 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.

Also, the dataset object offers some additional help if you need it.

- dataset.iterate_forever(batch_size) yields an infinite sequences of batches of examples.
- dataset.get_validation_accuracy() returns the accuracy of your model on the validation set. This can be useful to determine when to stop training.

For this question you will train a network to classify articles of fashion clothing from the MNIST dataset.

Each article of clothing 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.

Complete the implementation of the *FashionClassificationModel* class in *models.py*. The return value from *FashionClassificationModel.run()* should be a batch_size×10 node containing scores, where higher scores indicate a higher probability of a digit belonging to a particular class (0-9). You should use *nn.SoftmaxLoss* as your loss. Do not put a *ReLU* activation after the last layer of the network.

For both this question and Q4, in addition to training data, there is also validation data and a test set. You can use *dataset.get_validation_accuracy()* to compute validation accuracy for your model, which can be useful when deciding whether to stop training. The test set will be used by the autograder.

To receive points for this question, your model should achieve an accuracy of at least 96% on the test set. If you are having trouble reaching this think about what modifications to the neural network you can make.

Submission

Please submit your *models.py* onto gradescope.