6.S083 Fall 2019: Problem set 5

Submission deadline: Wednesday December 11, 2019 at 11:59pm.

Submit a Jupyter notebook with your solutions to `sandersd@mit.edu`. Make sure the name of the file contains your first name and last name.

Neural networks

In this problem set, we will implement a simple neural network to classify real-world data, consisting of photos of fruits from https://github.com/Horea94/Fruit-Images-Dataset. These have been pre-processed into a few features, in particular the average amount of red, green and blue.

Note that the implementation is aimed at understanding and will not use the most efficient methods.

Exercise 1: Loading data

1. Install and load the `CSV` and `DataFrames` packages.
2. Read in the `fruit.dat` data file using `CSV.read`. This will return a `DataFrame` object. Call it `data`.

   You can access columns of it as e.g. `data.red`.

   The `type` column indicates if the data corresponds to an apple (1), banana (2) or grape (3) image.
3. Plot the green column against the blue column, separating and labelling the data according to the type of fruit.
4. Plot the amount of green against the output 0 for banana and 1 for non-banana.

Exercise 2: Single neuron

1. Define a `Neuron` type that will contain a vector of weights `w` and a number of parameters `num_params`. Include the bias as the final weight. For simplicity, leave the field `w` *untyped* (even though this will lower performance).
2. Define a method for the constructor of `Neuron` that accepts a vector of weights and automatically fills in `num_params` as the length of the vector.
3. We can make objects behave like functions with the following syntax:

```
function (n::Neuron)(x)
    ...
end
```

Fill this in so that the neuron output is $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$. Here b is the last element of the **w** vector, which should be 1 longer than than the input **x**.

You will need to define the sigmoid function σ. You may use $\cdot$ from the `LinearAlgebra` package. You may assume that the lengths of vectors are correct, but you can add error handling if you prefer.

4. Define a neuron `n` with 2 random parameters. Check that you can do `n(x)`, where `x` is a vector of length 1.

Exercise 3: Predicting using a single neuron

We will now use our simple neuron to predict the banana-ness from the amount of green.

1. Define a function `stochastic_gradient_descent` that does stochastic gradient descent on the parameters w. It should accept a loss function `L` and two vectors `xs` and `ys`, as well as a number of steps `N`.

   At each step, it chooses a random observation `x` from the vector `xs` of green values, and the corresponding output `y`. It calculates the gradient of L *with respect to the parameters* w at the current value of w using those x and y. It then moves w in the appropriate direction for gradient descent.

   You may use `ForwardDiff.gradient` or your own multi-dimensional gradient function. `ForwardDiff.gradient(f, x)` calculates the gradient of a function f at x, where f takes a *vector* as argument.

2. Define a partial distance-squared loss function $L(w, x, y)$. This measures the distance between "the output of the neuron when the input is x" and "the desired output y", when the parameters are w. It must thus create a new neuron each time with those parameters.

   "Partial" here means that the loss function is for a *single* data point.

   Check that your loss function works by applying it with a random w to the first input point. What type of object should it return?

3. Use this to train a neuron to predict banana-ness from green-ness (as in [1.4]) by passing the data to `stochastic_gradient_descent`.

   Make an interactive visualization of the training process, showing the predicted output curve compared to the input data. To do so, you will need to return the history of w vectors from `stochastic_gradient_descent`.

   To plot the predicted output curve you should use a range of x values in the required range.

   You will need to use $N = 10^5$ or even 10^6. You can try a step size $\eta = 0.1$, but it may need to be smaller.

4. Run the training again using only 90% of the data, retaining the last 10% as a test set. Plot the total training loss (mean of L over all data points) and test loss (mean loss over test set) as a function of time.

Exercise 4: Single neuron with more inputs

Let's use a single neuron to try and separate the data in 2D.

1. Use stochastic gradient descent to train a neuron with 2 inputs and 1 output. The input data should be vectors `[g, b]` of green–blue pairs, and the output should be banana-ness.

   Note that the neuron now needs 3 parameters. You will need to modify `stochastic_gradient_descent` so that it correctly calculates the length of the parameter vector w.

2. Visualize the training interactively by plotting the data and using a heatmap of the neuron output as a function of x and y. Plot the contour level 0.5.

   `heatmap` takes x and y ranges and a function. The function must be of the form `(x, y) -> ...`, i.e. it takes two arguments and returns a number.

   To plot a contour, use `contour`. It takes x and y ranges and the function, and an argument `levels=[0.5]` to specify the level where the output equals `0.5`.

3. Plot the training loss and test loss as a function of time.

Exercise 5: Single layer

We will train a single neural network layer to try to classify the 3 fruits in our data set.

1. Write a function `one_hot` that accepts i and n and returns a one-hot vector of length n with 1 in the ith place.

2. Make a vector `ys` corresponding to the one-hot encoding of the labels of the data points.

3. Write a partial loss function that returns a scalar.

4. Define a `Layer` type that contains a vector of neurons and a total parameter count.

   Define a constructor that accepts a vector of neurons and fills in the total parameter count (the sum of the neurons' parameter counts).

   Check that this works by defining a layer with 2 neurons with 3 weights each.

5. Define a constructor of `Layer` that accepts a matrix (of type `Matrix`). It constructs a layer consisting of neurons whose weights are the rows of the matrix.

   Check that it works by defining a layer using a random 3×4 matrix.

6. Make `Layer` into a callable object using the same method as for `Neuron`. It should return a vector of outputs.

7. Make a version of `stochastic_gradient_descent` to train a layer.

8. Use it to train a layer on the (green, blue) input data, with outputs given by the one-hot vectors.

   Be careful to choose the correct number of neurons to get the correct number of outputs and the length of the weight vector for each neuron to correspond to the inputs.

   You may wish to sample only some of the data points of each type to make this training faster.

9. Make an interactive visualization showing the decision boundaries (0.5 contours) for each neuron separately.

10. Why is the layer not able to correctly classify all of the fruit types?

Exercise 6: Multiple layers

Finally we will train a neural network with one "hidden layer" to predict fruit types.

1. Define a `NeuralNetwork` type, containing a vector of layers and a total number of parameters.

   Make a constructor that takes a vector of layers and sums up their parameter counts.

2. Make a function `make_network`. It accepts a "dimension vector", i.e. a vector of integers like `[1, 3, 2]` that give the number of inputs / outputs. E.g. here, 1 is the number of inputs to the network, 2 is the number of outputs, and 3 is the number of neurons in the intermediate "hidden" layer,.

   Use this to make a `NeuralNetwork` by constructing layers of the correct sizes. Use variables `num_inputs` and `num_outputs` and be very careful about creating layers of the correct sizes.

   Use random weights for each layer.

3. Make `NeuralNetwork` into a callable object. It should run the input through each layer in turn. Finally it should run the resulting vector through `softmax` to give a probability vector as output.

   You will need to write a `softmax` function.

4. Make a method of `NeuralNetwork` that accepts a dimension vector and a vector of weights, and constructs the weight matrices for each layer by taking weights from that weight vector.

5. Make a version of `stochastic_gradient_descent` that accepts a dimension vector. It should initially create a random network to work out the correct length of weight vector.

6. Train the network on 90% of the data. Plot the training and test loss as a function of time.

7. Make an interactive visualization of the learning process. Use a probability of $1/3$ as the separating surface for each component of the output.