Advanced Natural Language Processing

Assignment 2: Intent Classification with Feedforward Neural Network

Credits:

---

1 Introduction

In this assignment, you will implement a Feedforward Neural Network and train it with backpropagation to classify intent from the provided dataset. For the purpose of understanding the learning process, the whole dataset is used as both training and test data. (What does that mean for your results?). The dataset is available here: `https://github.com/sonos/nlu-benchmark/tree/master/2017-06-custom-intent-engines` under *Creative Commons Zero v1.0 Universal* licence. The provided file contains a randomly generated sample of the original dataset.

You should implement all parts of this exercise using only python + standard libraries + NumPy. (That is, no specialised machine learning libraries are allowed.) Here is a list of NumPy functions that may or may not be useful for this task: `np.array()`, `np.eye()`, `np.reshape()`, `np.ones()`, `np.zeros()`, `np.dot()`, `np.concatenate()`, `np.maximum()`, `np.argmax()`, `np.sum()`, `np.uniform()`.

A more comprehensive introduction to NumPy can be found here:
`https://sites.engineering.ucsb.edu/~shell/che210d/numpy.pdf` .

2 Getting started

You should start by unpacking the archive `assignment2.zip`. This results in a directory `assignment2` with the following structure:

```
- data/
--- dataset.csv
- model/
--- __init__.py
--- ffnn.py
--- model_utils.py
- assignment2.py
- utils.py
- helper.py
```

3 Intent Data

The available utility function will take care of the data loading for you. If you are interested in the datasets, you can take a look at the `data` directory. Each item in our data consists of a sentence and its intent (represented as a string). We can look at several examples:

```
what movie times are at bow tie cinemas, SearchScreeningEvent
can you put this xandee, AddToPlaylist
```

The `load_dataset()` function will return a tuple of sentence list, intent list and set of unique intent that is available in the dataset.

4 Starter Code

Let's look at the main program in `assignment2.py`. It is supposed to ...

1. reads in the intent classification dataset using `load_dataset()`,
2. converts the dataset into a bag-of-words representation matrix and label matrix,
3. initializes a `Feedforward Neural Network` model,
4. trains the model with backpropagation and then uses the model to evaluate the dataset.

However, there are several missing features that need to be implemented. Let's implement them one by one!

4.1 Bag-of-Words Representation [15 pts]

The first thing you're being asked to do is to convert the text into a bag-of-words representation matrix where the dimension of the matrix is $V \times M$ (M: number of examples, V: vocabulary size) and the label to a matrix of dimension $K \times M$ where K is number of classes. You can implement them using `bag_of_words_matrix()` and `labels_matrix()` under `model/model_utils.py`. Add a preprocessing step where words that occur less than 2 times are replaced by an `<UNK>` token.

4.2 Activation Functions [10 pts]

For the classification task, the softmax activation function for the output layer with K classes is given by:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

The activation function of the hidden neurons is a non-linear function. We have seen tanh being used in class, but more common these days are for example ReLU or sigmoid, given by:

$$\text{ReLU}(z) = \max(0, z)$$
$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

You need to implement softmax, ReLU and derivatives of ReLU in `model/model_utils.py` so that we can train our neural network. Additionally, use Jupyter Notebook to plot the ReLU and sigmoid functions, as well as their derivatives. Observe the plot and **discuss briefly what the advantages and disadvantages** of the ReLU and sigmoid activation function might be. Remember that your implementation should be able to accept NumPy array and matrices.

4.3 Feedforward Neural Network [35 pts]

Now that you have created the input matrix, we can implement our neural network and perform a forward propagation to classify intent in `model/ffnn.py`. To perform the forward propagation, you should compute z^l and pass it through the activation function for each layer, given by:

$$z^l = W^l a^{l-1} + b^l$$
$$a^l = g(z^l)$$

where W^l is a weight matrix between layer l and $l+1$, z^l is value of the hidden layer at layer l before activation, a^l is value of the hidden layer at layer l after activation, and b^l is bias term for layer l. You should implement the forward pass using `forward()` method of `NeuralNetwork` class.

Your implementation of feedforward computation should computes $\hat{y_i}$ for every example i. The neural network has 3 layers - an input layer, a hidden layer and an output layer, where the hidden layer has 150 neurons. Don't forget to include the bias term. Use ReLU as the activation function for the hidden layer and softmax for the output layer. When initializing the `NeuralNetwork` class, you should also initialize the parameters using random values from a uniform distribution in the range (-1,1). Provide a seed value to the random number generator during initialization, to make the results reproducible. The purpose of using this kind of initialisation is to break symmetry and ensure that different neurons can learn different non-linear functions. (Hint: use vectorization methods instead of a for loop for speedup.)

Use this neural network to predict the intent and calculate the accuracy of the classifier in `batch_train()` under `helper.py`. (Should you be expecting high numbers yet?). Be aware that the output of `forward()` method is a $K \times M$ matrix containing the *probability distribution* of each class. In order to compare this with the ground truth matrix, you should take the `argmax` to obtain the predicted class of each example. Do this using the `predict()` method.

4.4 Backpropagation [40 pts]

You will now implement the backpropagation algorithm to compute the gradient of the cost function with respect to the neural network weights' and bias term (using the `backward()` method). First of all, implement the cross entropy loss function `compute_loss()` to monitor if your model is actually learning. Remember that in backpropagation we want to propagate the error signal to measure how much each neuron in the hidden layer contributes to the error in the output layer. It is more or less similar to forward propagation but in a reverse direction. For the output layer, set δ for cross entropy loss:

$$\delta^L = \hat{y} - y$$

where L is the output layer and $\hat{y}$ is the predicted probability distribution for y (check the derivation of the gradient for softmax and cross-entropy for details on where this δ comes from).

For the remaining hidden layer l, set:

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot g'(z^l)$$

where $\odot$ is an element-wise product of matrices (Hadamard product), g' is the derivative of the activation function.

The derivative of ReLU is given by:

$$\mathrm{ReLU}'(z) = \begin{cases} 1, & \text{if} \quad z > 0 \\ 0, & \text{otherwise} \end{cases}$$

By calculating the error term for each layer, you can then use the error term to calculate the partial derivatives $\frac{\partial \mathcal{L}}{\partial W^l} = \delta^l (a^{l-1})^T$ and $\frac{\partial \mathcal{L}}{\partial b^l} = \delta^l$ and perform batch gradient descent to update the parameter. (Batch gradient descent = run through all training instances and compute the gradient, then make the weight update.) Make sure that you accumulate the gradients for all the training samples and divide it by number of samples before doing the update.

Here is some simple pseudocode to help with the training procedure:

Algorithm 1 Training procedure with batch gradient descent

```
for $N$ epochs do
    define gradient accumulator $\Delta w = 0$, $\Delta b = 0$ for each weight and bias term
    define cost accumulator $\Delta \mathcal{L} = 0$ for the loss
    for $i$-th training example do
        perform forward pass
        calculate loss on example $i$, $\mathcal{L}_i$
        $\Delta \mathcal{L} \leftarrow \Delta \mathcal{L} + \mathcal{L}_i$

        perform backpropagation
        $\Delta w \leftarrow \Delta w + \frac{\partial \mathcal{L}}{\partial W}$ for each weight
        $\Delta b \leftarrow \Delta b + \frac{\partial \mathcal{L}}{\partial b}$ for each bias term
    end for
    calculate the cost, which is just the average loss, Cost $= \frac{1}{m} \Delta \mathcal{L}$
    $w \leftarrow w - \frac{\alpha}{m} \Delta w$ for each weight
    $b \leftarrow b - \frac{\alpha}{m} \Delta b$ for bias term
end for
```

Run the training for 1000 epochs using learning rate = 0.005 and use this neural network to predict the intent and calculate the accuracy of the classifier. (Hint: the dimension of δ^l should match the dimension of a^l, and the dimension of $\frac{\partial \mathcal{L}}{\partial W^l}$ and $\frac{\partial \mathcal{L}}{\partial b^l}$ should match the dimension of W^l and b^l, respectively). Plot the cost function for each iteration and compare the results after training with results before training. **Discuss what you observe**! (Don't forget to use the flag `--train` when you are ready to train the model. Extend this part of the assignment from the previous section in `batch_train()`)

4.5 Bonus: Mini-batch and Stochastic Gradient Descent [15 pts]

As a bonus, train the neural network using mini-batch gradient descent with batch size = 64 and stochastic gradient descent (i.e., batch size = 1) for 1000 epochs using learning rate = 0.005. Plot the cost vs iteration for both cases and briefly **discuss your observations**! (In order to do this, use the flag `--minibatch` when running `assignment2.py`. Implement the complete functionality in `minibatch_train()` under `helper.py`)

5 Submission

Upload your `assignment2_LASTNAME.zip` file on Moodle.