CS725 : Assignment

Predicting Song Release Year Using Timbre based Audio Feautures

Goda Nagakalyani

Implement a Feedforward Neural Network using Python

This assignment will familiarize you with training and evaluating feedforward neural networks. You will work on a regression task where you will have to predict the release year of a song from a set of timbre-based audio features extracted from the song. This consists of a year range between 1922 to 2011.

Dataset Information

- The dataset contains three files `train.csv`, `dev.csv`, and `test.csv`
- Each row in the `__.csv` file contains timbre-based audio features extracted from a song.
- The dataset has 90 features: 12 timbre average values and 78 timbre covariance values. Each column denotes a feature.
- `train.csv` and `dev.csv` contains following columns:

```
1. label - Year of release of the song in the range [1922, 2011]
2. TimbreAvg1
3. TimbreAvg2
.
.
13. TimbreAvg12
14. TimbreCovariance1
15. TimbreCovariance2
.
.
91. TimbreCovariance78
```

- `test.csv` contains same features except `label`

Implement the neural network, train it using the data in `train.csv` and report its performance on dev data in `dev.csv`.
Implement the functions definitions given in [nn.py](nn.py) to create and train a neural network. Run mini-batch gradient descent on Mean Squared Error (MSE) loss function. We have provided a helper file helper.md to give step-by-step introduction to the [nn.py](nn.py) file

```
- Seed for numpy: 42
- Use ReLU activation function for ALL HIDDEN layers.
```

Initialize weights and biases using the uniform distribution in the range \[-1, 1\].

Part 2 (10 points)
In This Part , you will evaluate your network's performance on test data given in `test.csv`.

In this part, there is no restriction on any hyper-parameter values. You are also allowed to explore various hyper-parameter tuning and cross-validation techniques.

Part 3 (5 points)
Feature selection and create a new feature set based on the original feature set consisting of 90 features. The size of the new feature set should be strictly smaller than 90, and performance using this subset of features should ideally be no worse than what you get with using the complete feature set. Note that each feature in the new feature set could be a combination of features in the original feature set. You are free to use any technique to identify potentially useful features and create any wrapper functions over given functions in [nn.py](nn.py).

Dataset Information
- The dataset contains three files `train.csv`, `dev.csv`, and `test.csv`
- Each row in the `__.csv` file contains timbre-based audio features extracted from a song.
- The dataset has 90 features: 12 timbre average values and 78 timbre covariance values. Each column denotes a feature.
- `train.csv` and `dev.csv` contains following columns:
```
1. label - "Very Old", "Old", "New" and "Recent", based on when it was released
2. TimbreAvg1
3. TimbreAvg2
.
.
13. TimbreAvg12
14. TimbreCovariance1
15. TimbreCovariance2
.
.
91. TimbreCovariance78
```
- `test.csv` contains same features except `label`

Step-by-step Instructions

Initialization
First, create weights and biases in the __init__(..) function of the code. This can be done several ways. We provide two options here:
```python
biases = []
weights = []
for i in range(num_layers):
```

```
 if i==0:
   # Input layer
   weights.append(np.random.uniform(-1, 1, size=(NUM_FEATS, num_units)))
 else:
   # Hidden layer
   weights.append(np.random.uniform(-1, 1, size=(num_units, num_units)))

 biases.append(np.random.uniform(-1, 1, size=(num_units, 1)))

# Output layer
biases.append(np.random.uniform(-1, 1, size=(1, 1)))
weights.append(np.random.uniform(-1, 1, size=(num_units, 1)))
```
We have provided one example in the code itself. You can access the weights and biases using list indices.
For example, weights of input layer are in the first position of the list. Note that python lists indices are zero-based, i.e. first position is at index 0.
```python
input_layer_weights = weights[0]
```
Similarly, weights of first hidden layer are at index 1.
```python
first_hidden_layer_weights = weights[1]
```
In case you need to access weights in reverse order, python lists provide negative indexing. For example, weights of output layer can be accessed using index -1 and weights of last hidden layer can be accessed using index -2.
```python
output_layer_weights = weights[-1]
last_hidden_layer_weights = weights[-2]
```
Few more useful indexing tricks are:
```python
weights[1:] # weights of all layers except input layer
weights[:-1] # weights of all layers except output layer
weights[1:-1] # weights of all layers except input and output layer i.e. weights of all HIDDEN layers
```

Another way to implement weights is:
```python
# Assuming 2 hidden layers
weights = OrderedDict(
```

```
 0: np.random.uniform(-1, 1, size=(NUM_FEATS, num_units))
 1: np.random.uniform(-1, 1, size=(num_units, num_units))
 2: np.random.uniform(-1, 1, size=(num_units, num_units))
 3: weights.append(np.random.uniform(-1, 1, size=(num_units, 1)))
)
```
Note that OrderedDict is a dictionary structure which remembers the order in which entries were added to the dictionary. More about OrderedDict [here](https://www.geeksforgeeks.org/ordereddict-in-python/)

Forward Pass
Once the weights and biases are initialized, you can easily forward propogate throught the network. Do not bother about the loss or error at this stage. Pseudocode for forward pass can be written as:
```python
# Given: Input to network, X
h = X
for w, b in weights, biases:
 h = w*h + b
 h = activation(h)

output = h
return output
```
Please note that this is just a pseudocode, and copy-pasting it won't work. Please understand the underlying logic.

Matrix Dimensions
Let us consider an example of data with `NUM_FEATS=3` and `batch_size=5`, 2-hidden layers and `num_units=4`.
```python
NUM_FEATS = 3
batch_size = 5

num_layers = 2
num_units = 4

X = np.random.normal(size=(batch_size, NUM_FEATS))
target = np.random.normal(size=(batch_size, 1))

```
If we consider a 2-hidden layer network, we can initialize weights and biases as given [here](#Initialization).

Now we can write the forward pass as:
```python
a = X
for i, (w, b) in enumerate(zip(weights, biases)):

 h = np.dot(a, w) + b.T

 if i < len(weights)-1:
   a = relu(h)
 else: # No activation for the output layer
   a = h

pred = a
```
Let us first understand how `zip(..)` and `enumerate(..)` work.
```python

# First Create two lists
l_1 = [10,20,30,40]
l_2 = ['a','b','c','d']

# zip(..) function on two lists works as follows:
for i,j in zip(l_1, l_2):
 print(i,j)
```
Output:
```
10 a
20 b
30 c
40 d
```
So, `zip(l_1, l_2)` returns a list of tuples in which `i`-th tuple is `(l_1[i], l_2[i])`.

Now let's see how `enumerate(..)` works:
```python
# enumerate is used as follows:
for i,j in enumerate(l_2):
 print(i,j)
```
Output:
```
0 a
```

```
1 b
2 c
3 d
```
`enumerate(l_2)` returns a list of tuples in which `i`-th tuple is `(i, l_2[i])`.

Next we will parse the line `h = np.dot(h, w) + b.T`.

Note the `transpose` operation on bias `b`. For `i=0`, shapes of `h` and `w` are `[5,3]` and `[3,4]` respectively.
`np.dot(h, w)` computes the matrix multiplication of `h` and `w`. Hence shape of `np.dot(h, w)` is `[5,4].`

Next we add bias to `np.dot(h, w)`. Note that the shape of bias vector at `i=0` is `[4,1]`. In order to add bias vector to all examples in the batch, we use something called [broadcasting](https://numpy.org/doc/stable/user/basics.broadcasting.html).
Shape of transpose of `b` is `[1,4]`. The addition of two matrices with shapes `[5,4]` and `[1,4]` returns a matrix with shape `[5,4]`. The addition is performed by copying the bias vector 5 times. Hence we use the transpose here. Transpose of vector `b` is denoted as `b.T`.

Gradient of Loss w.r.t. Output Layer Weights

Continuing above example, let us try to compute the gradient of the loss function with respect to weights of the output layer.
For that, let's first write a loss function.
We will use mean squared error (mse) loss function.
```python
loss = (target - pred)**2 # Sum of squared errors.
```
Note that `loss` is a numpy array of shape `[5,1]`. Let us expand the `pred` variable. For that, let's track few more variables from the forward pass as follows:
```python
a = X
h_states = []
a_states = []
for i, (w, b) in enumerate(zip(weights, biases)):

 if i==0:
   h_states.append(a) # For input layer, both h and a are same
 else:
   h_states.append(h)
 a_states.append(a)
```

```
 h = np.dot(a, w) + b.T

 if i < len(weights)-1:
   a = relu(h)
 else: # No activation for the output layer
   a = h

pred = a
```

In our example, weights of the output layer are denoted as `weights[-1]`.
Now, From the forward pass, we know that `pred` is expressed as
```python
pred = np.dot(a_states[-1], weights[-1]) + biases[-1].T # Convince yourself that this is true.
```
Hence we can express `loss` as
```python
loss = (target - (np.dot(a_states[-1], weights[-1]) + biases[-1].T) )**2 # Sum of squared errors.
```
Now we want to compute gradient of `loss` with respect to weights of the output layer i.e. weights[-1]. Gradient of `loss` w.r.t. weights of the output layer
can be written as
```
loss_gradient = a_states[-1] * (pred - target) # batch_size x num_units
update_gradient = 1./batch_size * np.sum(loss_gradient, axis=0) # num_units
```
Please refer to [this file](gradients.pdf) for a detailed explanation of above expression. `update_gradient` can be used to update the `weights[-1]` as `weights[-1] = weights - learning_rate*update_gradient`.

General Guidelines
1. Make sure your code is vectorized, otherwise it will be very slow. Like we have computed gradients w.r.t. weights of output layer for all examples in a single expression, design gradients of all weight vectors in the same way.
2. Do not get hung up over multiplications and divisions by constants such as 2 or m. These constants do not affect the final performance of the network.
3. For dev, there is no need to have batches, entire dev data can be passed as a single batch.

Submitted solution to the assignment in file [submission.ipynb]

- Created a neural Network using the **NumPy** library, which inputs music data as mentioned in [README.md] and predicts the year it was released.
- We used **MSE loss** along with **L2-Regularization** to train the model. We also used learning rate decay and input normalization to preprocess the data.
- The model achieved an **RMSE** of **9.87962** on the provided test data.