Implementation of Fully Connected Neural Network From Scratch

Ayan Sengupta
16307R005

MODULES	Operation	Function Names
Pre-Processing Data Module: preprocessing.py	Embedding	encode_train encode_class get_labels
Activation **(All were not used, just for testing purpose)** Module: model.py, model3.py, model4.py	Function Evaluation	sigmoid Softmax Relu tanh
	Function Differentiation	sigmoid_backward softmax_backward relu_backward tanh _backward
K-Fold Cross-Validation Module: model.py, model3.py, model4.py	Cross-Validation with k =5	k_fold_cross_validation
Accuracy and Prediction Module: model.py, model3.py, model4.py	Calculation accuracy of each validation set. Prediction of weights for the test data set.	accuracy2 predict
Model Module: model.py, model3.py, model4.py	Forward Propagation, Cost Computation, Backward Propagation, Update Parameters, Update Parameters with momentum, Cross Entropy with Regularizer	stochastic_model

All the functions created have docstrings and adequate comments for understanding the code.

EMBEDDING:

The data given is as follows:

s1 "Suit of card #1" Ordinal (1-4) representing {Hearts, Spades, Diamonds, Clubs}
c1 "Rank of card #1" Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
s2 "Suit of card #2" Ordinal (1-4) representing {Hearts, Spades, Diamonds, Clubs}
c2 "Rank of card #2" Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
s3 "Suit of card #3" Ordinal (1-4) representing {Hearts, Spades, Diamonds, Clubs}
c3 "Rank of card #3" Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
s4 "Suit of card #4" Ordinal (1-4) representing {Hearts, Spades, Diamonds, Clubs}
c4 "Rank of card #4" Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
s5 "Suit of card #5" Ordinal (1-4) representing {Hearts, Spades, Diamonds, Clubs}
c5 "Rank of card 5" Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)

class "Poker Hand" Ordinal (0-9)

0: Nothing in hand; not a recognized poker hand
1: One pair; one pair of equal ranks within five cards
2: Two pairs; two pairs of equal ranks within five cards
3: Three of a kind; three equal ranks within five cards
4: Straight; five cards, sequentially ranked with no gaps
5: Flush; five cards with the same suit
6: Full house; pair + different rank three of a kind
7: Four of a kind; four equal ranks within five cards
8: Straight flush; straight + flush
9: Royal flush; {Ace, King, Queen, Jack, Ten} + flush

In data preprocessing each attribute will be described with binary sets (0 and 1). There are four different card suits (Hearts, Spades, Diamonds, Clubs) so it will be four binary sets to describe them. Different suites would be encoded like

[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]

There are 13 different cards in every suit and they will get their own binary sets with 13 values to describe them. Different card values would be encoded like:

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

So a typical hand [s1, c1, s2, c2, s3, c3, s4, c4, s5, c5] would be a list of length 85
example:
A typical hand like [4, 7, 3, 5, 3, 3, 1, 13, 4, 8, 0] would be encoded to

array([0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
0., 1., 0., 0., 0., 0., 0., 1.])

In this case we will have 85 inputs and 10 outputs. We have 85 inputs because of 5 cards in hand and every card have suit (set of 4 values) and rank(set of 13 values) so 5 * (4 + 13) = 85 inputs and we have 10 outputs as we have set of 10 values for describing a poker hand. There are 10 possible poker hands depending which 5 cards we have so we need to give matching binary set for every poker hand. After encoding each class would look like this:

class 0 after encoding : [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
class 1 after encoding : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
class 2 after encoding : [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
class 3 after encoding : [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
class 4 after encoding : [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
class 5 after encoding : [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
class 6 after encoding : [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
class 7 after encoding : [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
class 8 after encoding : [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
class 9 after encoding : [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

Architecture

The neural network model is a 4 layer model. The Input layer has 85 neurons, output layer has 10 neurons. This model has been created with 2 hidden layers, 3 hidden layers and 4 hidden layers separately.

Number of neurons used in hidden layers for different trials:

2 Hidden Layer:
1st hidden layer: 10 - 85
2nd hidden layer: 10 - 20

3 Hidden Layer:
1st hidden layer: 10 - 85
2nd hidden layer: 10 - 85
3rd hidden layer: 10 -20

4 Hidden Layer:
1st hidden layer : 10 - 85
2nd hidden layer: 10 - 40
3rd hidden layer: 10 -40
4th hidden layer: 10 -20

Hyperparameters used:
K-fold Cross-validation over: K=5
learning_rate = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5]
num_hidden_layers = [2, 3, 4]
regularizer_lambda = [1e+1, 1, 1e-1]
Momentum_factor = 0.4

The activation function attempted for different trials for better results:

1. **Sigmoid** in hidden layers, **Sigmoid** in the output layer. This finally gave the best results and this one is the fastest one also. As we had already done encoding, output layer had 10 neurons, each varying from 0-1. Sigmoid was the most natural option. It is easy to differentiate as well. I hoped to get better results

with softmax so I tried softmax next.

2. **Sigmoid** in hidden layers, **Softmax** in the output layer. Softmax had a general tendency of blowing up. After numerous tries and after the successful implementation of computationally stable softmax, I found accuracy was lower than sigmoid.
3. **Relu** in hidden layers, **Sigmoid** in the output layer. Relu was not differentiable. So I ran into problem immediately, so went and tried leaky relu. Due to unsatisfactory results from softmax, I stuck with Sigmoid as my output layer.

4. **leaky_relu** in hidden layers, **Sigmoid** in the output layer. Leaky relu was showing improvements in accuracy, but it took a lot more iterations than sigmoid as hidden layers. As I was using stochastic model (discussed later), leaky relu was taking unprecedented time for computation.

The Explanation for using momentum, batch, mini-batch and stochastic updates.

Neural Network was designed first with the full matrix, then with batch updates and finally stochastic updates. I got best result for stochastic updates, for my network.

Pros:

1. Cost Entropy loss is less.
2. My results of batch updates showed, it was getting stuck at some local minima, but for stochastic updates, global minima is ensured.

Cons:

1. Takes a long time to converge with respect to batch/ mini batch.
2. Iterating with 800000 data was really slow, so had to use 200000 data for most of my computations.

Results

No	Learning rate	Hidden layers	regularizer	iterations	Train Accuracy	Test Accuracy(kaggle)
1	1e-1	2	1e+1	200	97.07 %	91.13 %
2	1e-2	2	1e+1	200	97.59 %	97.69 %
3	1e-3	2	1e+1	200	97.89 %	97.93 %
4	1e-4	2	1e+1	200	97.95 %	97.98 %
5	1e-5	2	1e+1	200	98.11 %	98.13 %
6	1e-1	2	1	200	97.58 %	97.62 %
7	1e-2	2	1	200	98.25 %	98.21 %
8	1e-3	2	1	200	98.20 %	98.14 %
9	1e-4	2	1	200	98.21 %	98.23 %
10	1e-5	2	1	200	98.56 %	N.A.
11	1e-1	2	1e-1	200	97.61%	N.A.
12	1e-2	2	1e-1	200	98.27 %	N.A.
13	1e-3	2	1e-1	200	98.54 %	N.A.
14	1e-4	2	1e-1	200	98.58 %	N.A.
15	1e-5	2	1e-1	200	98.46 %	N.A.
16	1e-1	3	1e+1	200	93.46 %	N.A.
17	1e-2	3	1e+1	200	93.89 %	N.A.
18	1e-3	3	1e+1	200	94.11 %	N.A.
19	1e-4	3	1e+1	200	94.25 %	N.A.
20	1e-5	3	1e+1	200	95.25 %	N.A.
21	1e-1	3	1	200	Not tried	N.A.
22	1e-2	3	1	200	Not tried	N.A.
23	1e-3	3	1	200	Not tried	N.A.
24	1e-4	3	1	200	Not tried	N.A.

25	1e-5	3	1	200	Not tried	N.A.
26	1e-1	3	1e-1	200	97.32%	N.A.
27	1e-2	3	1e-1	200	97.39%	N.A.
28	1e-3	3	1e-1	200	97.48%	N.A.
29	1e-4	3	1e-1	200	97.71%	N.A.
30	1e-5	3	1e-1	200	97.88%	N.A.

31	1e-1	4	1e+1	200	97.08 %	N.A.
32	1e-2	4	1e+1	200	97.41 %	N.A.
33	1e-3	4	1e+1	200	97.58 %	N.A.
34	1e-4	4	1e+1	200	97.89 %	N.A.
35	1e-5	4	1e+1	200	98.01 %	N.A.
36	1e-1	4	1	200	97.21 %	N.A.
37	1e-2	4	1	200	97.57 %	N.A.
38	1e-3	4	1	200	97.61 %	N.A.
39	1e-4	4	1	200	97.83 %	N.A.
40	1e-5	4	1	200	97.98 %	N.A.
41	1e-1	4	1e-1	200	Not tried	N.A.
42	1e-2	4	1e-1	200	Not tried	N.A.
43	1e-3	4	1e-1	200	Not tried	N.A.
44	1e-4	4	1e-1	200	Not tried	N.A.
45	1e-5	4	1e-1	200	Not tried	N.A.