L Mill N Ravikumar
T Würfl K Breininger
S Gündel M Hoffmann
F Thamm F Denzinger
October 25, 2018

LME

Neural Networks

1 Neural Network Skeleton

We will realize a small layer oriented Deep Learning framework in this exercise. Layer oriented frameworks represent a higher level of abstraction to their users than graph oriented frameworks. This approach limits flexibility but enables easy experimentation using conventional architectures. Examples are Keras or Caffe.
Every layer in these architectures has to implement two fundamental operations: **forward(input_tensor)**, **backward(error_tensor)**. These operations are the basic steps executed during training and testing.

Task:

Implement a class **NeuralNetwork** in the file: "NeuralNetwork.py" in the same folder as "NeuralNetworkTests.py".

- Implement four public members. A list **loss** which will contain the loss value for each iteration after calling **train**. A list **layers** which will hold the architecture, a member **data_layer**, which will provide input data and labels and a member **loss_layer** referring to the special layer providing loss and prediction.
- Implement a method **forward** using input from the **data_layer** and passing it through all layers of the network. Note that the **data_layer** provides an **input_tensor** and a **label_tensor** upon calling **forward()** on it.
- Implement a method **backward** starting from the **loss_layer** passing it the **label_tensor** for the current input and propagating it back through the network.
- Additionally implement a convenience method **train(iterations)**, which trains the network for **iterations** and stores the loss for each iteration.
- Finally implement a convenience method **test(input_tensor)** which implements prediction using the method **predict(activation_tensor)** of the **loss_layer**. The desired output are the class probabilities of the **predict** method.

L Mill N Ravikumar
T Würfl K Breininger
S Gündel M Hoffmann
F Thamm F Denzinger

2 Fully Connected Layer

The Fully Connected(FC) layer is the theoretic backbone of layer oriented architectures.

Task:

Implement a class **FullyConnected** in the file: "FullyConnected.py" in folder "Layers". This class has to provide the methods **forward(input_tensor)** and **backward(error_tensor)**.

- Implement a method **forward(input_tensor)** which returns the **input_tensor** for the next layer. **input_tensor** is a matrix with rows of arbitrary dimensionality **input_size** and columns of size **batch_size** representing the number of inputs processed simultaneously. The **output_size** is a parameter of the layer specifying the row dimensionality of the output.
- Write a constructor for this class, receiving the arguments **(input_size, output_size)**. Initialize the parameters of this layer uniformly random in the range $[0, 1)$ and add a public member **delta** representing the individual learning rate of this layer with a default of one.
- Implement a method **backward(error_tensor)** which updates the parameters and returns the **error_tensor** for the next layer. Hint: if you discover that you need something here which is no longer available to you, think about storing it at the appropriate time.
- To be able to test the gradients with respect to the weights: The member for the weights and biases should be named **weights**. Additionally provide a method: **_get_gradient_weights** which returns the gradient with respect to the weights, after they have been calculated in the backward-pass.

You can verify your implementation using the provided testsuite by providing the commandline parameter **TestFullyConnected**

L Mill N Ravikumar
T Würfl K Breininger
S Gündel M Hoffmann
F Thamm F Denzinger

3 Rectified Linear Unit

The Rectified Linear Unit is the standard activation function in Deep Learning nowadays. It has revolutionized Neural Networks because it improves on the "vanishing gradient" problem.

Task:

Implement a class **ReLU** in the file: "ReLU.py" in folder "Layers". This class also has to provide the methods **forward(input_tensor)** and **backward(error_tensor)**.

- Write a constructor for this class, receiving no arguments.
- Implement a method **forward(input_tensor)** which returns the **input_tensor** for the next layer.
- Implement a method **backward(error_tensor)** which returns the **error_tensor** for the next layer. Hint: the same hint as before applies.

You can verify your implementation using the provided testsuite by providing the commandline parameter **TestReLU**

L Mill N Ravikumar
T Würfl K Breininger
S Gündel M Hoffmann
F Thamm F Denzinger

4 SoftMax Layer

The SoftMax activation function is actually a activation function (SoftMax) with Maximum Likelihood estimation of the parameters using a cross-entropy loss.

Task:

Implement a class **SoftMax** in the file: "SoftMax.py" in folder "Layers". This class now has to provide three methods. When forward propagating we now additionally need the argument **label_tensor**: **forward(input_tensor, label_tensor)**. Additionally we need a method **predict(input_tensor)** which we use during testing when we don't have any labels to output our classification result. Finally we need a method: **backward(label_tensor)** which starts the backward passing of our error.

- Write a constructor for this class, receiving no arguments.
- Implement a method **forward(input_tensor, label_tensor)** which returns the loss accumulated over the batch. The **label_tensor** consists of **batch_size** rows of the one-hot encoded target vectors.
- Implement a method **predict(input_tensor)** which returns the estimated class probabilities for each row representing an element of the batch.
- Implement a method **backward(label_tensor)** which returns the **error_tensor** for the next layer. Hint: again the same hint as before applies.

You can verify your implementation using the provided testsuite by providing the commandline parameter **TestSoftMax**

L Mill N Ravikumar
T Würfl K Breininger
S Gündel M Hoffmann
F Thamm F Denzinger

5 Test, Debug and Finish

Now we implemented everything.

Task:

Debug your implementation until every test in the suite passes. You can run all tests by providing no commandline parameter.