Neural Networks

1 Basic Optimizer

In this course we will implement advanced optimization schemes, but in the first exercise we start with the basic Stochastic Gradient Descent (SGD).

Task:

Implement the class **Sgd** in the file "Optimizers.py" in folder "Optimization".

- The **Sgd** constructor receives the **learning_rate** with data type float.
- Implement the method
  calculate_update(weight_tensor, gradient_tensor) that returns the updated weights according to the basic gradient descent update scheme.

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

2 Fully Connected Layer

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, including the Fully Connected Layer which must be implemented in this task, has to implement two fundamental operations: **forward(input_tensor)**, **backward(error_tensor)**. These operations are the basic steps executed during training and testing. 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)** as well as the property **optimizer**.

- Implement a method **forward(input_tensor)** which returns the **input_tensor** for the next layer. **input_tensor** is a matrix with columns of arbitrary dimensionality **input_size** and rows 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 $[0, 1)$.
- Add a setter and getter property **optimizer** which sets and returns the protected member **_optimizer** for this layer. Properties offer a pythonic way of realizing getters and setters. Please get familiar with this concept if you are not aware of it.
- Implement a method **backward(error_tensor)** which 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 property **gradient_weights** which returns the gradient with respect to the weights, after they have been calculated in the backward-pass.
- Use the method **calculate_update(weight_tensor, gradient_tensor)** of your optimizer in your backward pass, in order to update your weights. **Don't perform an update if the optimizer is not set**.

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

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**

4 Cross Entropy Loss

The cross entropy Loss is often used in classification task, typically in conjunction with SoftMax (or Sigmoid).

Task:

Implement a class **CrossEntropyLoss** in the file: "Loss.py" in folder "Optimization". When forward propagating we now additionally need the argument **label_tensor** for **forward(input_tensor, label_tensor)** and **backward(label_tensor)**.

- Write a constructor for this class, receiving no arguments.
- Implement a method **forward(input_tensor, label_tensor)** which computes the Loss value according the CrossEntropy Loss formula accumulated over the batch.
- Implement a method **backward(label_tensor)** which returns the **error_tensor** for the next layer. The backpropagation starts here, hence no **error_tensor** is needed. Instead, we need the **label_tensor**.
  Hint: the same hint as before applies.

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

5 SoftMax Layer

The SoftMax activation function is used to transform the logits (the output of the network) into a probability distribution. Therefore, SoftMax is typically used for classification tasks.

Task:

Implement a class **SoftMax** in the file: "SoftMax.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 estimated class probabilities for each row representing an element of the batch.
- Implement a method **backward(error_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**

6 Neural Network Skeleton

The Neural Network defines the whole architecture by containing all its layers from the input to the loss layer. This Network manages the testing and the training, that means it calls all forward methods passing the data from the beginning to the end, as well as the optimization by calling all backward passes afterwards.

Task:

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

- Implement five public members. An **optimizer** object received upon construction as the first argument. 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. You do not need to care for filling these members with actual values. They will be set within the unit tests.

- 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.

- Implement the method **append_trainable_layer(layer)** which makes a deep_copy of the neural networks **optimizer** and sets it for the **layer** by using its **optimizer** property. Make sure you append this layer to the list **layers** afterwards.

- 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 propagates the **input_tensor** through the network and returns the prediction of the last layer. For classification tasks we typically query the probabilistic output of the SoftMax layer.

7 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. Make sure you don't forget to upload your submission to StudOn. Use the dispatch tool, which checks all files for completeness and zips the files you need for the upload. Try

```
python3 dispatch.py --help
```

to check out the manual. For dispatching your folder run e.g.

```
python3 dispatch.py -i ./src -o submission.zip
```