Convolutional Neural Networks

We will extend our framework to include the building blocks for modern Convolutional Neural Networks (CNNs). To this end, we will add initialization schemes improving our results, advanced optimizers and the two iconic layers making up CNNs, the convolutional layer and the max-pooling layer. To ensure compatibility between fully connected and convolutional layers, we will further implement a flatten layer.

1 Initializers

Initialization is critical for non-convex optimization problems. Depending on the application and network, different initialization strategies are requires. A popular initialization scheme is named Xavier or Glorot initialization. Later an improved scheme specifically targeting ReLU activation functions was proposed by Kaiming He.

Task:

Implement four classes **Constant**, **UniformRandom**, **Xavier** and **He** in the file "Initializers.py" in folder "Layers". Each of those has to provide the method **initialize(weights_shape, fan_in, fan_out)** which returns an initialized tensor of the desired shape.

- Implement all four initialization schemes. Use the original formulation of the Xavier and the He initialization.
- Add a method **initialize(weights_initializer, bias_initializer)** to the class **FullyConnected** reinitializing its weights. Initialize the bias separately with the **bias_initializer**.
- Refactor the class **NeuralNetwork** to receive a **weights_initializer** and a **bias_initializer** upon construction.
- Add a method **append_trainable_layer(layer)** to the class **NeuralNetwork** initializing the layer with the stored **initializers**.

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

2 Advanced Optimizers

More advanced optimization schemes can increase speed of convergence. We implement a popular per-parameter adaptive scheme named ADAM and a common scheme improving stochastic gradient descent called momentum.

Task:

Implement the classes **Sgd**, **SgdWithMomentum** and **Adam** in the file "Optimizers.py" in folder "Optimization". These classes all have to provide the method **calculate_update(weight_tensor, gradient_tensor)**.

- Implement all three schemes. The constructor of each optimizer receives its typical parameters as floats. The constructor for **SGD** receives only the **learning_rate**. The **SgdWithMomentum** constructor receives the **learning_rate** and the **momentum_rate** in this order. The **Adam** constructor receives the **learning_rate**, **beta1** and **beta2**, again exactly in this order.
- Add a method **set_optimizer(optimizer)** to the class **FullyConnected**, storing the **optimizer** for this layer.
- Refactor the class **FullyConnected** to use its **optimizer** to update its parameters. **Don't perform an update if the optimizer is unset**.
- Refactor the class **NeuralNetwork** to receive an **optimizer** upon construction as the first argument.
- Refactor the method **append_trainable_layer(layer)** of the class **NeuralNetwork** providing the **layer** with a deep_copy of the optimizer.

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

3 Flatten Layer

Flatten layers bring the multi-dimensional input to one dimension only. This is useful especially when connecting a convolutional or pooling layer with a fully connected layer.

Task:

Implement a class **Flatten** in the file "Flatten.py" in folder "Layers". This class 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 reshapes and returns the **input_tensor**.
- Implement a method **backward(error_tensor)** which reshapes and returns the **error_tensor**.

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

4 Convolutional Layer

We will extend our framework from last session to deep neural networks. The convolutional layer is the backbone of these. It reduces overfitting and memory consumption by restricting the layers parameters to local receptive fields.

Task:

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

- Write a constructor for this class, receiving the arguments **stride_shape**, **convolution_shape** and **num_kernels** defining the operation. The input layout for 2D is defined in b, c, y, x order, where b stands for the batch, c represents the channels and x, y represent the spatial dimensions. The argument **stride_shape** can be a single value or a tuple to allow for different strides in the spatial dimensions. Initialize the parameters of this layer uniformly random in the range $[0, 1)$.

- To be able to test the gradients with respect to the weights: The members for weights and biases should be named **weights** and **bias**. Additionally provide two methods: **get_gradient_weights** and **get_gradient_bias**, which return the gradient with respect to the weights and bias, after they have been calculated in the backward-pass.

- Implement a method **forward(input_tensor)** which returns the **input_tensor** for the next layer. The output shape is calculated in the forward process based on the **input_tensor**. Use zero-padding to achieve the same spatial dimensions for input and output ("same"-padding). You can choose which implementation approach to follow. Either use the convolution and correlation implementations of scipy, use the im2col matrix multiplication approach or roll your own inner products. Efficiency tradeoffs will be necessary and are expected in this scope. For example stride may be implemented wastefully as subsampling. However make sure 1x1-convolutions and 1-dimensional convolutions are handled correctly.
  Hint: Using correlation in the forward and convolution/correlation in the backward pass might help with the flipping of kernels.
  Hint 2: The scipy package features a n-dimensional convolution/correlation.

- Implement a method **backward(error_tensor)** which updates the parameters using the **optimizer** and returns the **error_tensor** for the next layer. You can again choose any of the mentioned implementation strategies.

- Implement a method **set_optimizer(optimizer)** storing the optimizer for this layer. Note that you might need two copies of the optimizer object if you handle the bias separate from the other weights.
- Implement a method **initialize(weights_initializer, bias_initializer)** which reinitializes the weights by using the provided initializer objects.

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

5 Pooling Layer

Pooling layers are typically used in conjunction with the convolutional layer. They reduce the dimensionality of the input and therefore also decreases memory consumption. Additionally they reduce overfitting by introducing a degree of scale and translation invariance. We will implement max-pooling as the most classical form of pooling.

Task:

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

- Write a constructor receiving the arguments **stride_shape** and **pooling_shape**, with same ordering specified in the convolutional layer.
- Implement a method **forward(input_tensor)** which returns the **input_tensor** for the next layer. Hint: Keep in mind to store the correct information necessary for the backward pass.
- Implement a method **backward(error_tensor)** which returns the **error_tensor** for the next layer.

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

6 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.