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. Of course we want to continue implementing the layers ourselves and the usage of machine learning libraries is still not allowed.

1 Initializers

Initialization is critical for non-convex optimization problems. Depending on the application and network, different initialization strategies are required. 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 them 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. Note the following:
  - The **Constant** class has a member that determines the constant value used for weight initialization. The value can be passed as a constructor argument, with a default of 0.1.
  - The support of the uniform distribution is the interval $[0, 1)$.
  - Have a look at the exercise slides for more information on Xavier and He initializers.
- Add a method **initialize(weights_initializer, bias_initializer)** to the class **FullyConnected** reinitializing its weights. Initialize the bias separately with the **bias_initializer**. Remember that the bias is usually also stored in the weights matrix.
- Refactor the class **NeuralNetwork** to receive a **weights_initializer** and a **bias_initializer** upon construction.
- Extend the method **append_layer(layer)** in the class **NeuralNetwork** such that it initializes trainable layers with the stored **initializers**.

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 **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)**.

- The **SgdWithMomentum** constructor receives the **learning_rate** and the **momentum_rate** in this order.
- The **Adam** constructor receives the **learning_rate**, **mu** and **rho**, exactly in this order. In literature **mu** is often referred as β_1 and **rho** as β_2.
- Implement for both optimizers the method **calculate_update(weight_tensor, gradient_tensor)** as it was done with the basic SGD Optimizer.

3 Flatten Layer

Flatten layers reshapes the multi-dimensional input to a one dimensional feature vector. 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**.

4 Convolutional Layer

While fully connected layers are theoretically well suited to approximate any function they struggle to efficiently classify images due to extensive memory consumption and overfitting. Using convolutional layers, these problems can be circumvented by restricting the layer's 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. Note the following:
  - this layer has trainable parameters, so set the inherited member **trainable** accordingly.
  - **stride_shape** can be a single value or a tuple. The latter allows for different strides in the spatial dimensions.
  - **convolution_shape** determines whether this object provides a 1D or a 2D convolution layer. For 1D, it has the shape [c, m], whereas for 2D, it has the shape [c, m, n], where c represents the number of input channels, and m, n represent the spatial extent of the filter kernel.
  - **num_kernels** is an integer value.

  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 properties: **gradient_weights** and **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 a tensor that serves as the **input_tensor** for the next layer. Note the following:
  - The input layout for 1D is defined in b, c, y order, for 2D in b, c, y, x order. Here, b stands for the batch, c represents the channels and x, y represent the spatial dimensions.
  - You can calculate the output shape in the beginning based on the **input_tensor** and the **stride_shape**.
  - Use zero-padding for convolutions/correlations ("same" padding). This allows input and output to have the same spatial shape for a stride of 1.

Make sure that 1×1-convolutions and 1D 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.

Hint 3: Efficiency trade-offs will be necessary in this scope. For example, striding may be implemented wastefully as subsampling *after* convolution/correlation.

- Implement a property **optimizer** storing the optimizer for this layer. Note that you need two copies of the optimizer object if you handle the bias separately from the other weights.
- Implement a method **backward(error_tensor)** which updates the parameters using the **optimizer** (if available) and returns the **error_tensor** which returns a tensor that servers as **error_tensor** for the next layer.
- Implement a method **initialize(weights_initializer, bias_initializer)** which reinitializes the weights by using the provided initializer objects.

5 Pooling Layer

Pooling layers are typically used in conjunction with the convolutional layer. They reduce the dimensionality of the input and therefore also decrease memory consumption. Additionally, they reduce overfitting by introducing a degree of scale and translation invariance. We will implement max-pooling as the most common 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 the same ordering as specified in the convolutional layer.
- Implement a method **forward(input_tensor)** which returns a tensor that serves as the **input_tensor** for the next layer. Hint: Keep in mind to store the correct information necessary for the backward pass.
  - Different to the convolutional layer, the pooling layer must be implemented only for the 2D case.
  - Use "valid"-padding for the pooling layer. This means, unlike to the convolutional layer, don't apply any "zero"-padding. This may discard border elements of the input tensor. Take it into account when creating your output tensor.
- Implement a method **backward(error_tensor)** which returns a tensor that serves as the **error_tensor** for the next layer.