Regularization and the Recurrent Layer

1 Regularizers

This time we will extend our framework to include common regularization strategies. As neural networks are extremely flexible statistical models they tend to show more variance and less bias. However this also means they are very prone to overfitting. Therefore, regularization is a very important part of deep learning. Throughout this exercise you will have to use base-classes, which implement default members and methods accessible to each class derived from the respective base-class.

1.1 Refactoring

Since the behaviour of some layers changes depending on whether the network is currently training or testing, we need to extend our base-layer and refactor our neural network a little bit. Moreover, we choose to introduce a "base-optimizer", which provides some basic functionality, in order to enable the use of regularizers.

Task:

yes • Add a boolean member **testing_phase** to the **BaseLayer**. Set it to `False` by default.

yes • Implement a property **phase** in the **NeuralNetwork** class setting the phase of each of its layers accordingly. Use this method to set the phase in the **train** and **test** methods.

yes • Create a base-class **Optimizer** for optimizers in the "Optimizers.py" file. Make all optimizers inherit from this "base-optimizer".

yes • The class **Optimizer** should have a method **add_regularizer(regularizer)** and a member storing the **regularizer**.

1.2 Optimization Constraints

Equality and inequality constraints on the norm of the weights are well known in the field of Machine Learning. They enforce the prior assumption on the model of small weights in case of L2-regularization and sparsity in case of L1-regularization.

Task:

Implement classes **L2_Regularizer** and **L1_Regularizer** in the file "Constraints.py" in the folder "Optimization". Both have to provide the methods **calculate_gradient(weights)** that calculates a (sub-)gradient on the weights needed for the optimizer. Additionally they have to provide a method **norm(weights)**, which is used to calculate the norm enhanced loss.

yes • Implement the two schemes. The constructor of each receives an argument **alpha** representing the regularization weight.

yes • Refactor the optimizers to apply the new **regularizer** if it is set using the **calculate_gradient(weights)** method.

??? • Refactor the **NeuralNetwork** class to add the regularization loss to the data loss. Use the method **norm(weights)** to get the regularization loss inside all layers (Fully Connected, Convolution and RNN) and sum it up.

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

1.3 Dropout

Dropout is a typical regularizer method for Deep Learning. It's most often used to regularize fully connected layers. It enforces independent weights, reducing the effect of co-adaptation.

Task:

Implement a class **Dropout** in the file "Dropout.py" in folder "Layers". This class has to provide the methods **forward(input_tensor)** and **backward(error_tensor)**. This layer has no adjustable parameters. We choose to implement inverted dropout.

yes • Implement the constructor for this layer receiving the argument **probability** determining the fraction units to keep.

yes • Implement the Dropout methods **forward(input_tensor)** and **backward(error_tensor)** for the training phase.

yes • Modify the Dropout method **forward(input_tensor)** for the testing phase.

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

1.4 Batch Normalization

Batch Normalization is a regularization technique which is conceptually very well known in Machine Learning but specially adapted to Deep Learning.

Task:

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

- Implement the constructor for this layer which receives the argument **channels**. **channels** denotes the number of channels of the **input_tensor** in both, the vector and the image-case. Initialize the bias β and the weights γ according to the **channels**-size using the method **initialize**. This layer has trainable parameters, so remember to set the inherited member **trainable** accordingly.
- Implement the method **initialize** which initializes the weights γ and the biases β. **initialize** ignores any assigned initializer and initializes always the weights γ with ones and the biases β with zeros, since you do not want the weights γ and bias β to have an impact at the beginning of the training. Make sure you optimize the weights and bias in the backward pass, but only if optimizers are defined.
- Implement the Batch Normalization methods **forward(input_tensor)** and **backward(error_tensor)** with independent activations for the training phase. Make sure you use an ϵ smaller than 1e-10.
- Hint: In Helpers.py we provide a function **compute_bn_gradients(error_tensor, input_tensor, weights, mean, var)** for the computation of the gradient w.r.t. inputs. Note that this function does not compute the gradient w.r.t. the weights.
- Implement the **moving average estimation** of training set mean and variance.
- Modify the Batch Normalization method **forward(input_tensor)** for the testing phase. Use an online estimation of the mean and variance. Initialize mean and variance with the batch mean and the batch standard deviation of the first batch used for training.
- Implement the convolutional variant. The layer should change behaviour depending on the shape of the **input_tensor**.
- Implement a method **reformat(tensor)** which receives the tensor that must be reshaped. Depending on the shape of the tensor, the method reformats the image-like tensor (with

4 dimension) into its vector-like variant (with 2 dimensions), and the same method reformats the vector-like tensor into its image-like tensor variant. Use this in the forward and the backward pass.

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

1.5 LeNet (optional)

By now our framework contains all the essential parts to build a convolutional neural network. However not every architecture can be trained to high performance. Therefore, successful architectures also play an important role. We implement a variant of a well known architecture called LeNet.

Task:

We implement a modification of a classic architecture called LeNet in file "LeNet.py" in folder "Models".

- Add two functions (This means they should not be part of the class.) **save(filename, net)** and **load(filename, data_layer)** to the **NeuralNetwork** file. Those two methods should use python's pickle to save and load networks. After loading the network the **data_layer** should be set again.
- Exclude the **data_layer** from saving by implementing the **__getstate__()** and **__setstate__(state)** methods. The **__setstate__(state)** method should initialize the dropped members with None. This needs to be done, since the **data_layer** is a generator-object, which cannot be processed by pickle.
- Implement the LeNet architecture as a function **build()** returning a network. Use a SoftMax layer and ReLU activation functions. Also ignore the fact, that the original architecture did not use padding for the convolution operation.
- Not typical for Deep Learning but useful, we also store the optimizer, initializer and their settings. Use the ADAM optimizer with a learning rate of $5 \cdot 10^{-4}$ and a L_2 regularizer of weight $4 \cdot 10^{-4}$.

You can train this LeNet-variant on MNIST using the script **"TrainLeNet.py"**.

2 Recurrent Layers

Many machine learning problems require memorization capabilities along a particular dimension. Typical examples are time series prediction problems. A common method to address those are Recurrent Neural Networks (RNNs). To implement RNNs we can reuse most algorithms of our framework. The only necessary new components are new layers implementing the specific RNN cells.

2.1 Activation functions

Two common activation functions which we didn't implement so far will come in handy: The hyperbolic tangent (Tangens Hyperbolicus) and the Sigmoid.

Task:

Implement two classes **TanH** and **Sigmoid** in files: "TanH.py" and "Sigmoid.py" in the folder "Layers".

- Implement the operations **forward(input_tensor)**, **backward(error_tensor)** for the **TanH** and the **Sigmoid** activation function.
- Specifically store *activations* for the dynamic programming component, instead of the **input_tensor**. This is possible because the gradient involves only activations instead of the input (see slides).

2.2 Elman Recurrent Neural Network (RNN)

The type of recursive neural networks known as Elman network consists of the simplest RNN cells. They can be modularly implemented as layers.

Task:

Implement a class **RNN** in the file: "RNN.py" in folder "Layers". This class has to provide all the functionalities required by a trainable layer for our framework.

- Write a constructor, receiving the arguments **(input_size, hidden_size, output_size)**. Here **input_size** denotes the dimension of the *input vector* while **hidden_size** denotes the dimension of the **hidden state**. Initialize the hidden state with all zeros.
- Add a property **memorize** which sets the boolean state representing whether the RNN regards subsequent sequences as a belonging to the same long sequence. In the constructor, initialize this member with `False`. Hint: think about how to transfer the 'memory' from this sequence to the next sequence.
- Implement a method **forward(input_tensor)** which returns a tensor that serves as the **input_tensor** for the next layer. Consider the "batch" dimension as the "time" dimension of a sequence over which the recurrence is performed. The first **hidden_state** for this iteration is all zero if the boolean member variable is `False`, otherwise restore the hidden state from the last iteration. You can choose to compose parts of the RNN from other *layers* you already implemented.
- Implement a method **backward(error_tensor)** which updates the parameters and returns the **error_tensor** for the next layer. Remember that **optimizers** are decoupled from our *layers*. For the gradient calculation of some layers we need the input of the last forward pass at the respective point in time. Be sure to save those values in the forward pass and set them when backpropagating through time.
- Implement the accessor property **gradient_weights**. Here the **weights** are defined as the weights which are involved in calculating the **hidden_state** as a stacked tensor. E.g. if the **hidden_state** is computed with a single Fully Connected layer, which receives a stack of the **hidden_state** and the **input_tensor**, the weights of this particular Fully Connected Layer, are the weights considered to be weights for the whole class. In order to provide access to the **weights** of the RNN layer, implement a getter and a setter with a property for the **weights** member.
- To be able to reuse all regularizers, add the property to add an optimizer as **optimizer** and to calculate the loss caused by *regularization* as **calculate_regularization_loss()**

as introduced in the regularization exercise. Finally add the method **initialize(weights_initializer, bias_initializer)** to use our *initializers*.

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

2.3 Long Short-Term Memory (LSTM) (optional)

Elman networks severely suffer from the vanishing gradient problem. A common method to remedy this is to use more complicated RNN cells. The classical example is the LSTM cell. The unit tests for this class will only run when you create a file LSTM.py in your Layers folder.

Task:

Implement a class **LSTM** in the file: "LSTM.py" in folder "Layers". This class has to provide all the functionalities required by a trainable layer for our framework.

- Write a constructor, receiving the arguments **(input_size, hidden_size, output_size)**. Here **input_size** denotes the dimension of the *input vector* while **hidden_size** denotes the dimension of the *hidden state*. Initialize the hidden state with all zeros.
- Add a property **memorize** which sets the boolean state representing whether the RNN regards subsequent sequences as a belonging to the same long sequence. In the constructor, initialize this member with `False`. That means if this state is `True`, the *hidden_state* of one sequence processed in one forward call is carried over to next forward call as an initial state for the *hidden_state*. Otherwise the *hidden_state* is initialized with zeros every time a new sequence is processed. This is required to switch form BPTT to TBPTT.
- Implement a method **forward(input_tensor)** which returns a tensor that serves as the **input_tensor** for the next layer. Consider the *batch* dimension as the *time* dimension of a sequence over which the recurrence is performed. The first *hidden_state* for this iteration is all zero if the boolean member variable is `False`, otherwise restore the hidden state from the last iteration. You can choose to compose parts of the LSTM from other *layers* you already implemented.
- Implement a method **backward(error_tensor)** which updates the parameters and returns a tensor which serves as **error_tensor** for the next layer. Remember that **optimizers** are decoupled from our *layers*.
- Implement the accessor property **gradient_weights**. Here the **weights** are defined as the weights which are involved in calculating the **hidden_state** as a stacked tensor. E.g. if the **hidden_state** is computed with a single Fully Connected layer, which receives a stack of the **hidden_state** and the **input_tensor**, the weights of this particular Fully Connected Layer, are the weights considered to be weights for the whole class. In order to provide access to the **weights** of the LSTM layer, implement a getter and a setter with a property decorator for the **weights** member.

- To be able to reuse all regularizers, add the property to add an optimizer as **optimizer** and to calculate the loss caused by *regularization* as **calculate_regularization_loss()** as introduced in the regularization exercise. Finally add the method **initialize(weights_initializer, bias_initializer)** to use our *initializers*.

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

3 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. To run the unittests you can either execute them with python in the terminal or with the dedicated unittest environment of PyCharm. We recommend the latter one, as it provides a better overview of all tests. For the automated computation of the bonus points achieved in one exercise, run the unittests with the bonus flag in a terminal, with

```
python3 NeuralNetworkTests.py Bonus
```

or set in PyCharm a new "Python" configuration with *Bonus* as "Parameters". Notice, in some cases you need to set your src folder as "Working Directory". More information about PyCharm configurations can be found here [1].
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_to_implement -o submission.zip
```

and upload the .zip file to StudOn.

[1] https://www.jetbrains.com/help/pycharm/creating-and-editing-run-debug-configurations.html