L Mill N Ravikumar
T Würfl K Breininger
Y Xu F Denzinger
June 11, 2018

Recurrent Neural Networks

1 Recurrent Layer

Many prediction 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. Return values as in the implementation of the ReLU.

2 Activation functions

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

Task:

Implement two classes **TanH** and **Sigmoid** in files: "TanH.py" and "Sigmoid.py" in the folder "Layers" and add additional accessors to Conv, FullyConnected and BatchNormalization.

- Implement the operations **forward(input_tensor)**, **backward(error_tensor)** for the **TanH** 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).
- Implement the operations **forward(input_tensor)**, **backward(error_tensor)** for the **Sigmoid** activation function.
- Specifically store *activations* for the dynamic programming component, instead of the **input_tensor**.
- Refactor *every layer* using trainable *weights* to expose their *weights* with two methods **set_weights** and **get_weights**.

L Mill N Ravikumar
T Würfl K Breininger
Y Xu F Denzinger

3 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 methods required by a trainable layer for our framework.

- Write a constructor, receiving the arguments **(input_size, hidden_size, output_size, bptt_length)**. Here **input_size** denotes the dimension of the *input vector* while **hidden_size** denotes the dimension of the *hidden state*. The **bptt_length** controls how many steps backwards are considered in the calculation of the gradient with respect to the weights.
- Add a method **toggle_memory()** which switches a boolean value representing whether the RNN regards subsequent sequences as a belonging to the same long sequence. This is required to switch form BPTT to TBPTT.
- Implement a method **forward(input_tensor)** which returns the **input_tensor** for the next layer. Consider the *batch* dimension as the *time* dimension of a sequence where the recurrence is performed. The first *hidden_state* is either all zero or restored from a previous iteration depending on the boolean memory. 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. Truncate the calculation of gradients with respect to the weights after the *steps* specified by **bptt_length**. Remember that **optimizers** are decoupled from our *layers.*
- Implement the accessor methods **get_gradient_weights()**, **get_weights()** and **set_weights(weights)**. Here the **weights** are not uniquely defined. Return all the weights involved in calculating the *hidden_state.*
- To be able to reuse all regularizers, add the methods to add an optimizer as **set_optimizer(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 *intializers.*

L Mill N Ravikumar
T Würfl K Breininger
Y Xu F Denzinger

4 Long Short-Term Memory (LSTM)

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 of such a cell is the LSTM cell.

Task:

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

- Write a constructor, receiving the arguments **(input_size, hidden_size, output_size, bptt_length)**. Here **input_size** denotes the dimension of the *input vector* while **hidden_size** denotes the dimension of the *hidden state*. The **bptt_length** controls how many steps backwards are considered in the calculation of the gradient with respect to the weights.

- Add a method **toggle_memory()** which switches a boolean value representing whether the RNN regards subsequent sequences as a belonging to the same long sequence. This is required to switch form BPTT to TBPTT.

- Implement a method **forward(input_tensor)** which returns 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* is either all zero or restored from a previous iteration depending on the boolean memory. 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 the **error_tensor** for the next layer. Truncate the calculation of gradients with respect to the weights after the *steps* specified by **bptt_length**. Remember that **optimizers** are decoupled from our *layers*.

- Implement the accessor methods **get_gradient_weights()**, **get_weights()** and **set_weights(weights)**. Return all the weights involved in calculating the *hidden_state*.

- To be able to reuse all regularizers, add the methods to add an optimizer as **set_optimizer(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 *intializers*.

L Mill N Ravikumar
T Würfl K Breininger
Y Xu 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.