L Mill N Ravikumar
T Würfl K Breininger
S Gündel M Hoffmann
F Thamm F Denzinger

Regularization

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 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" folder "Optimization". Both have to provide the methods **calculate(weights)**. These will perform a (sub-)gradient update on the weights. Additionally they have to provide a method **norm(weights)**.

- Implement the two schemes. The constructor of each receives an argument **alpha** representing the regularization weight.
- Refactor the optimizers to be derived from a common base-class providing the method **add_regularizer(regularizer)** storing the **regularizer**.
- Refactor the optimizers to apply the new **regularizer** if it is set.
- Refactor the **Neural Network** class to add the regularization loss to the data loss. Hint: It will be necessary to refactor more classes to get the necessary information. Make use of base-classes.

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

2 Dropout

Dropout is a special regualizer typical 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. Also implement a base-class for the layers in the file "Base.py" in folder "Layers.

- Refactor all layers using a base-class to contain an enum named **Phase** specifying **train**, **test** or **validation** phase.
- Add a method **set_phase(phase)** to the **NeuralNetwork** class setting each layer's phase. Use this method to set the phase in **train** and **test**.
- Implement the constructor for this layer receiving the argument **probability** determining the fraction of dropped units.
- Implement the Dropout methods **forward(input_tensor)** and **backward(error_tensor)** for the training phase.
- 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**.

3 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.
- Implement the Batch Normalization methods **forward(input_tensor)** and **backward(error_tensor)** with independent activations for the training phase.
- Implement the online 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.
- Implement the convolutional variant. The layer should change behaviour depending on a argument **channels** to the constructor, which has a default value of zero. If it's higher than zero it should perform the convolutional variant.

Hint: Since you do not want the weights and bias to have an impact at the beginning of the training you should initialise the bias as zeros and all weights as ones.
You can verify your implementation using the provided testsuite by providing the commandline parameter **TestBatchNorm**.

4 LeNet

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.
- 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^{-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"**.

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.