

PROGRAMMING IN PYTHON II

Neural Network Implementation: Inference



Michael Widrich
Institute for Machine Learning

Copyright statement:

This material, no matter whether in printed or electronic form, may be used for personal and non-commercial educational use only. Any reproduction of this material, no matter whether as a whole or in parts, no matter whether in printed or in electronic form, requires explicit prior acceptance of the authors.

Outline

1. Motivation
2. PyTorch for building neural networks
3. Quick guide: Designing your neural network
4. Python II Project
5. Further reading

Motivation

- In order to design our neural networks efficiently, we would like to have
 - modular structure of layers and building blocks
 - consistent interfaces between modules
 - free combination of modules (not only sequential but also tree-like structures)
 - automated optimization of computations
 - automated deployment to CPU, GPU, or other dedicated hardware
 - automated computation of gradients for training
- PyTorch provides these features!

torch.nn.Module

- torch.nn.Module is the base class for all neural network modules
- All custom networks/layers/modules should be derived from this class
- Can contain and utilize other modules
- Automated registration of trainable parameters and parameters in submodules

torch.nn.Module: Usage

- torch.nn.Module usage is simple:
 1. Create class that inherits from torch.nn.Module
 2. Define a `.forward()` method
 3. Overwrite the `.init()` method
 4. Create an instance of your class and apply it to input

torch.nn.Module: Details

- The `.forward()` method will be executed when your class instance is applied to an input
- Using PyTorch (sub)module instances as module attributes, will register the submodule automatically
- Using PyTorch parameter instances as module attributes, will register the parameter automatically
 - `torch.nn.Parameter` will create trainable tensors, e.g. for NN weights
- PyTorch modules behave a lot like PyTorch tensors
 - Can be sent to devices using `.to(device=...)`
 - Can be converted to datatype using `.to(dtype=...)`

Predifined PyTorch modules

- Many predefined PyTorch modules exist
- Sometimes include additional optimization
 - E.g. PyTorch LSTM with specialized CUDA support but less flexible design
- Should be preferred over custom modules unless special functionality is desired

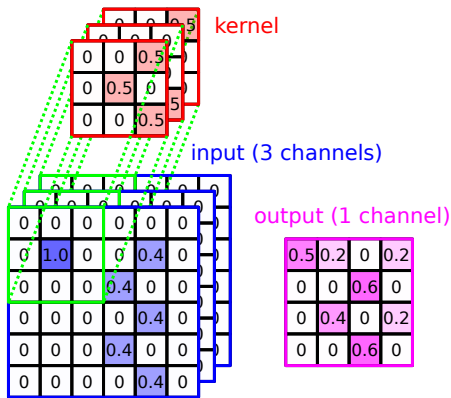
NN types: FFNN

- Fully-connected feed-forward NN (FFNN):
 - High complexity, high number of weights
 - Not utilizing order in feature vectors
 - Multi-dimensional inputs are typically flattened (reshaped to flat vector as input)

NN types: CNN (1)

■ Convolutional NN (CNN):

- Weight kernels are slid along (=convolved) an input tensor



NN types: CNN (2)

■ Convolutional NN (CNN):

- 1D CNNs: Kernels convolved over one dimension (e.g. time dimension in a time-series)
- 2D CNNs: Kernels convolved over 2 dimensions (e.g. spatial dimensions in image)
- nD CNNs: Possible but uncommon at the moment (expensive/lacking optimization)
- Less complexity, fixed kernel size to incorporate information about adjacent features
- Depth of network increases field of vision
- Typically employ pooling operations to pool adjacent features (e.g. max-pooling)
- Outputs sometimes flattend and then fed into FFNN or max-pooled to 1D feature vector

NN types: RNN

■ Recurrent NN (RNN):

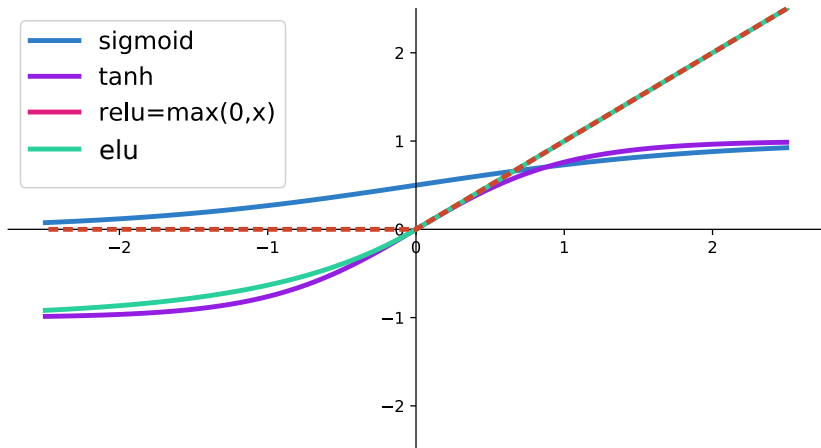
- ☐ NN applied to a list of inputs, using the same weights for each input
- ☐ NN has access to output/hidden state from previous input (=recurrent)
- ☐ Turing complete (but need to be trained somehow)
- ☐ 1D: Sequence (=list of feature vectors) as input
- ☐ nD: Multidimensional RNN variants for matrices or tree-like structure
- ☐ Most popular, since no **vanishing gradient**: LSTM
- ☐ Alternative to RNNs: Transformer NNs

Common activation functions

- Pay attention to recommended weight initialization and input normalization for the specific activation functions
- Scaled Exponential Linear Unit (SELU)
 - Self-normalizing properties benefit learning and design
 - Common for FFNN but also successes in CNN
- Rectified Linear Unit (ReLU)
 - Very common in CNNs
 - Units can “die” once they get stuck in 0-activation (=no gradients)
- Sigmoid activation function
 - Common in (older) NNs
 - Introduce vanishing gradients
 - Outputs in range $[0, 1]$

Common activation functions

Activation functions



Python II Project: NN Design

- We deal with image data → CNN is a natural choice
- Hyperparameters (see Unit 07): Number of kernels, size of kernels, pooling operations, activation function, skip connections, and number of layers
- Additional data can be fed to CNN by creating a new feature channel (see Assignment 2)
- CNN could be combined with FFNN but probably not necessary
- SELU or ReLU activation functions are good candidates

Further reading

- ÖAW AI summer school slide-set:
https://github.com/ml-jku/oeaw_ai_summer_school
- Courses and lecture materials in AI-study (Hands-on AI, Machine Learning: Supervised Techniques, LSTM and Recurrent Neural Nets, ...)
- *Pattern Recognition and Machine Learning* (C. Bishop)
- *Dive into Deep Learning* (A. Zhang, Z. Lipton, M. Li, A. Smola): <https://d2l.ai/>
- <https://pytorch.org/tutorials/index.html>
- https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html