PROGRAMMING IN PYTHON I

Unit 12: ML modules in Python: A preview to Programming in Python II

Michael Widrich
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

MACHINE LEARNING IN PYTHON

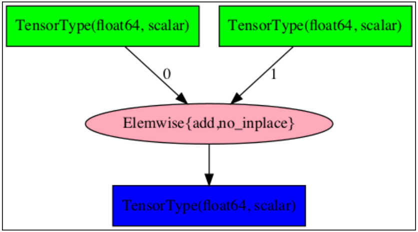
Motivation

- Python is a go-to language for Machine Learning (ML) Deep Learning (DL)
  - Implementation of research and production code is possible
  - Interpreted language with convenient usage supports quick implementation of ideas
  - Dedicated modules allow for fast execution on dedicated hardware
- Now we will tap into modules that allow us to write optimized Python code for ML
  - Usage of dedicated hardware (CPU, GPU, TPUs)
  - Automatic differentiation (e.g. for training of DL networks)
  - Convenience functions for data loading, preprocessing, training, evaluation, . . .
  - TensorFlow, PyTorch, . . .

COMPUTATIONAL GRAPHS

Computational graphs

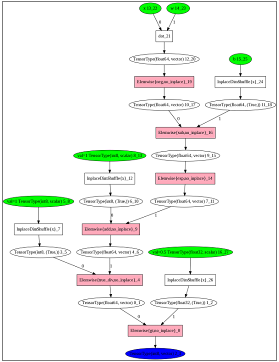
- A large part of the *magic* provided by TensorFlow/PyTorch is based on computational graphs
  - Symbolic graph of computations
  - Defines a pipeline of computations
  - Nodes in the graph can represent functions and placeholders for the data (=tensors)



[Scalar addition as computational graph. Source: https://www.tutorialspoint.com/theano/theano_computational_graph.htm]

Computational graphs: Benefits

- Computational graphs allow for:
  - Compilation of graph with optimization for dedicated devices or datatypes
  - Automatic differentiation



[Scalar addition as computational graph. Source: `https://www.tutorialspoint.com/theano/theano_computational_graph.htm`]

AUTOMATIC DIFFERENTIATION

Automatic differentiation

- When training artificial neural networks (NNs), we typically rely on gradient-based methods to update the weights
  - E.g.: Compute gradient of loss w.r.t. NN parameters to change parameters such that loss decreases
  - For deeper NNs (multiple layers), this relies heavily on the chain rule for differentiation

Automatic differentiation

- When training artificial neural networks (NNs), we typically rely on gradient-based methods to update the weights
  - E.g.: Compute gradient of loss w.r.t. NN parameters to change parameters such that loss decreases
  - For deeper NNs (multiple layers), this relies heavily on the chain rule for differentiation
- Not having to implement the chain rule formulas by hand for every NN makes our lives A LOT easier
  - Combined with modular layer design, this makes NN design and training almost plug-and-play

Automatic differentiation

- When training artificial neural networks (NNs), we typically rely on gradient-based methods to update the weights
  - E.g.: Compute gradient of loss w.r.t. NN parameters to change parameters such that loss decreases
  - For deeper NNs (multiple layers), this relies heavily on the chain rule for differentiation
- Not having to implement the chain rule formulas by hand for every NN makes our lives A LOT easier
  - Combined with modular layer design, this makes NN design and training almost plug-and-play
- Computational graph contains information about functions used to compute result and allows for automatic differentiation (This makes us really happy!)

[Further reading: `https://PyTorch.org/tutorials/beginner/blitz/autograd_tutorial.html`]

PYTHON MODULES

Python modules

- Computational graph created using Python code
  - After creation, data can be fed into the graph to compute output and gradients for updates
- PyTorch (`https://PyTorch.org/`)
  - Numpy-like code to dynamically* create graph
  - Computational graph is evaluated automatically when result is used (* no explicit PyTorch commands required)
- TensorFlow (`https://www.TensorFlow.org/`)
  - Explicit creation of static computational graph using TensorFlow commands in Python code
  - Evaluation of graph explicitly via TensorFlow commands
- PyTorch and TensorFlow both provide many convenience functions for ML