PROGRAMMING IN PYTHON II

Neural Network Implementation: Training

Michael Widrich
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Outline

1. **Gradient based methods**
2. **PyTorch for training neural networks**
3. **Loss functions**
4. **More on training**
5. **Python II Project**

GRADIENT BASED METHODS

Gradient based methods

- We want to choose our NN weights such that our model output is equal to our target
- We can define a loss function
  - Computes a loss given our model output and the target
  - The higher the loss, the farther away our output is to our target
  - We want to minimize the loss of our model

Gradient based methods

- We can compute the derivative (gradient) of the model loss w.r.t. the current model weights
  - Direction of the gradient is in the same direction as the steepest ascent
  - We can compute the negative gradient, change the weights a little bit (learning rate) into the direction of the steepest decent, and repeat this procedure
  - If we (would) have a convex problem (no local minima), this leads us to the global minimum...
  - ...but often a local minimum is good enough anyway :)

PYTORCH FOR TRAINING NEURAL NETWORKS

Autograd

- Computing all gradients by hand is tedious
- Since we have a computational graph of our operations, the gradients can be computed automatically (using the autograd method)
  - See Theano, TensorFlow, and PyTorch in Programming in Python I
- In PyTorch, the gradients are typically computed using `.backward()` on a tensor
  - Computed gradients are accumulated automatically
  - Autograd can be used explicitly too (for 2nd order methods, meta learning, etc.)

Optimizers

- Different variations of gradient based optimizers exist
- Prominent examples:
  - Stochastic gradient descent (`torch.optim.SGD`)
    - Simple gradient descent with learning rate
    - With optional momentum
    - Very common, good baseline
    - Learning rate and momentum as hyperparameters
  - Adam optimizer (`torch.optim.Adam`)
    - Gradient based optimizer with adaptive learning rate for each parameter and momentum
    - Very common, robust, sometimes doesn’t work
    - Learning rate as hyperparameter

Performing a weigth update

1. Define optimizer `optimizer`
2. Compute loss `loss`
3. Reset gradients `optimizer.zero_grad()`
4. Compute gradients `loss.backward()`
5. Perform weight update `optimizer.step()`
6. Repeat until end of training

LOSS FUNCTIONS

Loss functions (1)

- Different loss functions for different tasks
  - Different theoretical justifications
  - Not every loss function is suitable for every task
  - Choice of loss function depends on data, task, and model class

Loss functions (2)

- Common loss functions:
  - Regression (numerical target value): Mean squared error
    - Typically no output activation function
    - `torch.nn.MSELoss()`
  - Classification (target class): Cross entropy
    - Sigmoid or softmax output activation function
    - `torch.nn.BCEWithLogitsLoss()`
  - Classification (focus on classification border): Hinge loss

MORE ON TRAINING

Training schemes

- Training can be done for a fixed number of updates or epochs
  - Epoch: One iteration over all training samples
  - Update: One weight update
  - Number of updates/epochs is a hyperparameter
- Early stopping
  - Check model loss on validation set every n updates/epochs
  - Continue training but save model with best validation loss
  - After training, choose saved model with best validation loss as final model (least over-fitting)

Regularization

- Regularization can be used to counter over-fitting
- Prominent examples:
  - Dropout: Dropping out features or inputs randomly
  - Weight penalty terms: Add additional term to training loss
    - l1 penalty: Add sum of absolute weight values to loss
    - l2 penalty: Add sum of squared weight values to loss
  - Noise: Add random noise to inputs or features

Monitoring

- Always monitor your model during training!
- Handy for development but lossy: Tensorboard
  - For final evaluation use e.g. `.csv` files
  - Always save the trained model parameters!
- Histograms: Weights, gradients, activations
- Line-plots: Loss, regularization terms (for training and validation set)

Practical aspects

- 16bit float: Adam stability parameter
- Not learning? Check gradients and weights - do they change? are they sane values?
- Check the documentations of the functions
  - `torch.nn.BCEWithLogitsLoss()` expects the raw network output as input and adds sigmoid activation during computation for numerical stability
- Gradient clipping can help to stabilize training
- First find a model that over-fits on training set, then make it smaller/add regularization
- Prefer smaller/simpler models

PYTHON II PROJECT

Python II Project: Training

- We want to predict pixel values (=regression setting)
- If input is normalized, NN output has be de-normalized
- Regularization might help, e.g. l2 penalty with factor 10^{-5}
- First try to find a model that can over-fit on the training data
- Define your training, validation, and test set and keep them separated
- We will see data augmentation methods later