# 50.021 – AI

Alex

Week 02/03: SGD for logistic regression in python

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources. ]

## 1 Task1

The goal is to implement SGD for logistic regression. You work in `pytorch_logreg_gdesc_studentversion.py`.

What does one need to run logistic regression in pytorch? The computational flow is as follows:

1. define dataset class and dataloader class
2. define model
3. define loss
4. define an optimizer
5. initialize model parameters, usually when model gets instantiated
6. loop over epochs. every epoch has a train and a validation phase
7. train phase: loop over minibatches of the training data
   (a) set model to train mode
   (b) set model parameters gradients to zero
   (c) fetch input and ground truth tensors, move them to a device
   (d) compute model prediction
   (e) compute loss

   (f) run `loss.backward()` to compute gradients of the loss function with respect to parameters
   (g) run optimizer (here by hand) to apply gradients to update model parameters

8. validation phase: loop over minibatches of the validation data
   (a) set model to evaluation mode
   (b) fetch input and ground truth tensors, move them to a device
   (c) compute model prediction
   (d) compute loss in minibatch
   (e) compute loss averaged/accumulated over all minibatches
   (f) if averaged loss is lower than the best loss so far, save the state dictionary of the model containing the model parameters

9. return best model parameters

Things that need particular attention:

1. a subclass of `torch.utils.data.Dataset` class.

   Its functionality:

   `thisclass[i]` returns a tuple of pytorch tensors belonging to the i-th datasample.

   - We use `torch.utils.data.TensorDataset`.
   - Takes as argument $n$ tensors (e.g. features, labels, filenames):
     `dataset=torch.utils.data.TensorDataset(t0,t1,...tn-1)` .
   - `thisclass[i]` will be then a tuple of $n$ tensors:
     ```
     thisclass[i][0]=t0[i]
     thisclass[i][1]=t1[i]
     ...
     thisclass[i][n-1]=tn-1[i]
     ```
   - we create two datasets: one from the training data numpy array, a second from the validation data numpy array.
   - the numpy arrays must be converted into pytorch tensors before handing them over
   - More about the interfaces of `torch.utils.data.Dataset` in the next lecture

2. a `torch.nnDataloader` class: takes the Dataset as input, and returns a python iterator which produces a minibatch of some batch size every time it is called. Common parameters are batchsize, whether the data should be shuffled (suggested during training), or a sampler class which allows to control how minibatches are created.

3. define a model which takes samples as input, and produces an input. In pytorch it is a class derived from `nn.Module`. A standard neural network module (no matter pytorch or mxnet) needs usually two functions:
   - `def __init__(self,parameter1,parameter2):`
     where you define all layers which have model parameters or you define your parameter variables
   - `def forward(self,x):`
     where the input $x$ is processed until the output of your network.

   `class logreglayer(nn.Module):` is the model where I have predefined $w$ and $bias$, and you have to define `def forward(self,x):`

4. a loss function, used at training phase to measure difference between prediction and ground truth. Possibly, also a loss function at validation phase to measure the quality of your prediction on your validation dataset

5. an optimizer to apply the gradients to model parameters. In this lecture we will not use a pytorch optimizer, but updates weights

What do you have to do? check the #TODO tags.

- define the forward in the model `class logreglayer(nn.Module):`
- use `torch.utils.data.TensorDataset`
- initialize an instance of your model
- define a suitable loss for a binary classification with only one output, check docs on pytorch loss functions
- apply gradient to model parameters