PyTorch Cheatsheet - Part 1

Useful activation function and torch.nn.functional

- Linear function: $y = WX + b$ where W and X are vectors of size N (number of dimensions to the input).

```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

- Sigmoid function: $\frac{1}{1+e^{-z}}$ where z is the logit(s).

```
torch.nn.functional.sigmoid(input)
```

- Softmax function: $p(Y = t|x) = \frac{\exp(w_t^T x)}{\sum_{y \in \{0,\ldots,C-1\}} \exp(w_y^T x)}$

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None)[source]
```

Loss Functions

- Mean squared error: $\ell(x, t; w) = (y - t)^2$

```
torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')
```

- Minimum log-likelihood: $\ell(x, t; w) = \sum_{(x^{(i)}, t^{(i)}) \in \mathcal{D}} - \log p(t|x)$
  - Combined with binary classification: $\ell(x, t; w) = \sum_{(x^{(i)}, t^{(i)}) \in \mathcal{D}} \log(1 + \exp(-t^{(i)} w^T x^{(i)}))$
  - Combined with softmax: $\ell(x, t; w) = \sum_{(x^{(i)}, t^{(i)}) \in \mathcal{D}} \left(-w_{t^{(i)}}^T x + \log \sum_{c \in \{0,\ldots,C-1\}} \exp(w_c^T x)\right)$

```
torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None,
    reduction='mean', label_smoothing=0.0)[source]
```

- Cross Entropy Loss:
  - Linear (SVM formulation): $\ell(x, t; w) = \frac{||W[1:]||}{2} + C \sum \max\left(0, t^{(i)} \cdot Wx^{(i)}\right)^2$
  - Logistic: $\ell(x, t; w) = -t \log y - (1 - t) \log(1 - y)$

Optimizers and torch.optim

In standard gradient descent, the update rule is:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \nabla f(\mathbf{w}_k)$$

In **gradient descent with momentum**, we introduce a **velocity term** v_k:

$$v_{k+1} = \beta v_k - \alpha \nabla f(\mathbf{w}_k)$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k + v_{k+1}$$

where: α is the learning rate, $\beta \in [0, 1]$ is the momentum coefficient, and v_k is the velocity term.

The following are some useful optimizers provided by the torch.optim library including:

- Stochastic gradient descent

```
torch.optim.SGD(params, lr=0.001, momentum=0, dampening=0, weight_decay=0, nesterov=False, *,
    maximize=False, foreach=None, differentiable=False, fused=None)[source]
```

- Adam

```
torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False, *,
     foreach=None, maximize=False, capturable=False, differentiable=False, fused=None)
```

PyTorch datasets

Required functions for dataset class:

- *__init__*: The *__init__* method is the constructor for the new dataset.
- *__len__*: The *__len__* method overrides the *len()* function in Python to determine the length of the dataset.
- *__getitem__*: The *__getitem__* method overloads the use of brackets to index items in a dataset.

There are lots of cool dataloader attributes and methods including:

- *batch_size*: number of examples in each batch or call to the dataloader
- *shuffle*: Boolean option to shuffle dataset each pass or *epoch* through the dataset
- *sampler*: *Sampler* object that specifies how data will be extracted from the dataset. For example, the *SubsetRandomSampler* allows us to specify indices within the larger dataset to sample at random.

Sample Code

Here is a sample, two-dimensional logistic classifier code:

```python
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import SubsetRandomSampler

class LogisticRegression(nn.Module):
    def __init__(self, N):
        super().__init__()
        self.w = nn.Parameter(torch.ones(N))
        self.b = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return 1/(1+torch.exp(-(self.w@x+self.b)))

class TwoClassDataset(Dataset):
    # don't forget the self identifier!
    def __init__(self, N, sigma):
        self.N = N # number of data points per class
        self.sigma = sigma # standard deviation of each class cluster
        self.plus_class = self.sigma*torch.randn(N, 2) + torch.tensor([-1, 1])
        self.negative_class = self.sigma*torch.randn(N, 2) + torch.tensor([1, -1])
        self.data = torch.cat((self.plus_class, self.negative_class), dim=0)
        self.labels = torch.cat((torch.ones(self.N), torch.zeros(self.N)))

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        x = self.data[idx]
        y = self.labels[idx]
        return x, y # return input and output pair

N = 100
sigma = 1.5
dataset = TwoClassDataset(N, sigma)
plus_data = dataset.plus_class
negative_data = dataset.negative_class

# create indices for each split of dataset
N_train = 60
N_val = 20
N_test = 20
indices = np.arange(len(dataset))
np.random.shuffle(indices)
train_indices = indices[:N_train]
val_indices = indices[N_train:N_train+N_val]
test_indices = indices[N_train+N_val:]

# create dataloader for each split
batch_size = 8
train_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(train_indices)
    )
val_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(val_indices))
test_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(test_indices))

criterion = nn.BCELoss(reduction='mean') # binary cross-entropy loss, use mean loss
logreg_model = LogisticRegression(2) # initialize model
optimizer = torch.optim.SGD(logreg_model.parameters()) # initialize optimizer

n_epoch = 200 # number of passes through the training dataset
loss_values, train_accuracies, val_accuracies = [], [], []
for n in range(n_epoch):
    epoch_loss, epoch_acc = 0, 0
    for x_batch, y_batch in train_loader:
        optimizer.zero_grad()
        predictions = logreg_model(x_batch.unsqueeze(-1)).squeeze(-1)
        loss = criterion(predictions, y_batch)
        loss.backward()
        optimizer.step()
```