Deep Learning Cheatsheet

torch.nn Functions

- **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)
```

torch.nn Layers

- **Linear layer:** $y = WX + b$ 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)
```

- **Convolutional layer:** In the simplest case, assuming a input size of (N, C_{in}, H, W), the output is sized$(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ where N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1,
    bias=True, padding_mode='zeros', device=None, dtype=None)
```

(1d/2d/3d variations available as well)

- **Pooling layer:** Applies a #D (1d/2d/3d variations available) pooling over an input signal composed of several input planes. There are two flavors

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode
    =False)
```

```
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True,
    divisor_override=None)
```

- **BatchNorm layer:** normalizes the data over a batch using the formula $y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$ where γ and β are trainable parameters:

```
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True,
    device=None, dtype=None)
```

Principal Component Analysis

Steps to calculate principal components:

- Normalize data, $\frac{1}{N}\sum_{n=1}^{N} \mathbf{x}^{(n)} \left(\mathbf{x}^{(n)}\right)^T = \mathbf{\Sigma}$ and calculate covariance matrix: $\mathbf{\Sigma} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}^{(n)} \left(\mathbf{x}^{(n)}\right)^T = \frac{1}{N} X^T X$
- Find the D eigenvectors with the largest eigenvalues:

```
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
# Use torch.pca_lowrank to compute the top 2 principal components
U, S, V = torch.pca_lowrank(X, q=2)
# Project the data onto the principal components
X_pca = X @ V
```

K-means clustering

Steps to calculate k-means clusters. First, we got to choose how many (K) clusters we want to break the data up into. Randomly assign data points into clusters.

- Update centroid of clusters based on current data point assignment.
- Update datapoint assignment based off cluster centroids.

Loss function: $J\left(C_1, \ldots, C_k, \mu_1, \ldots, \mu_k\right) = \Sigma_{k=1}^{K} \Sigma_{i \in C_k} ||x^{(i)} - \mu_k||^2$

Gaussian mixture models

Model the data set as a combination of Gaussian curves: $p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x \mid \mu_k, \Sigma_k)$ where $\mathcal{N}(x \mid \mu_k, \Sigma_k)$: Gaussian density for the k-th component, $\mathcal{N}(x \mid \mu_k, \Sigma_k)$: Gaussian density for the k-th component, and K: total number of components.

- **Expectation step:** compute the "responsibilities" or the posterior probabilities that a data point $x^{(i)}$ belongs to each Gaussian component k: $\gamma(z_k^{(i)}) = \frac{\pi_k \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x^{(i)} \mid \mu_j, \Sigma_j)}$
- **Maximization step:** parameters of the GMM (i.e., the mixing coefficients, means, and covariances) are updated to maximize the expected complete-data log-likelihood ($\sum_{i=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k) \right)$) computed during the E-Step.
  - Update mixing coefficient: $\pi_k = \frac{1}{N} \sum_{i=1}^{N} \gamma(z_k^{(i)})$
  - Update means: $\mu_k = \frac{\sum_{i=1}^{N} \gamma(z_k^{(i)})\, x^{(i)}}{\sum_{i=1}^{N} \gamma(z_k^{(i)})}$
  - Update covariance matrices: $\Sigma_k = \frac{\sum_{i=1}^{N} \gamma(z_k^{(i)})\, (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^\top}{\sum_{i=1}^{N} \gamma(z_k^{(i)})}$

Generative adversarial networks

Assume a generator ($G_\theta(z)$) and discriminator ($D_w(x) = p(y = 1|x)$). The loss function for the discriminator is: $\mathcal{J}_D = -\Sigma_x \log D_w(x) - \Sigma_z \log(1 - D_w(G_\theta(z)))$.
For the generator, we have the formulation: $\mathcal{J}_G = -\mathcal{J}_D = const + \Sigma_z \log(1 - D_w(G_\theta(z)))$, but the question is how to optimize:

- min-max formulation: $\max_\theta \min_w \Sigma_x \log D_w(x) + \Sigma_z \log(1 - D_w(G_\theta(z)))$
- non-saturating formulation: $\min_\theta -\Sigma_z \log(D_w(G_\theta(z)))$

Image processing

- For bounding box problems, we optimize intersection over union.
- Let $x_{ij}^p \in \{0, 1\}$ be an indicator of matching default box i to ground-truth box j from class p, c be the class of the bounding box, l be the predicted bounding box, g be the ground-truth bounding box, and d be the matched default box. The loss function $\mathcal{L}$ is given as $\mathcal{L}(x, c, l, g) = \frac{1}{N}(\mathcal{L}_{\text{cls}}(x, c) + \alpha \mathcal{L}_{\text{loc}}(x, l, g))$, where N is the number of matched default boxes for the given image

Accuracy measures:

- $\text{Precision} = \frac{TP}{TP+FP} \in [0, 1]$, $\text{Recall} = \frac{TP}{TP+FN} \in [0, 1]$
- Average precision (AP) is area under precision recall curve.
- mAP score is AP at multiple IoU thresholds.

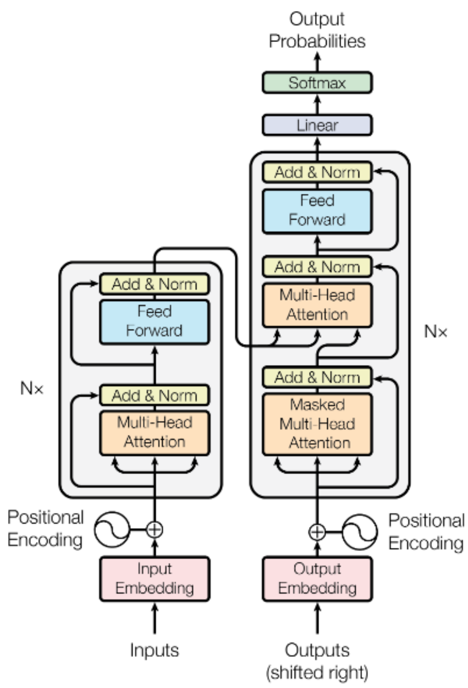

Figure 1: The Transformer - model architecture.

Transformer model

Position encoding:

- $\text{PE}^0_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$
- $\text{PE}^1_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$

where pos is the position in the sequence, i is the embedding dimension index is the d_{model} = total embedding size (e.g., 512) **Attention types:**

- $\text{SelfAttention}(Q_{\text{encoder}}, K_{\text{encoder}}, V_{\text{encoder}}) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$
- $\text{CrossAttention}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}}) == \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$
- $\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right) V$ where M is a mask of $-\infty$ values.

Attention heads are calculated by: $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ and $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O$

Sample Code

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

```
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()
```