Assignment 1: Machine Learning Basics
MA-INF 2313: Deep Learning for Visual Recognition

Due Date Theoretical: 23.11.2020
Due Date Programming: 23.11.2020

1 Theoretical Exercises *(20 points)*

a) Bias of an estimator *(5 points)*

The bias of an estimator is defined as

$$\text{Bias}(\hat{\theta}_m) = E(\hat{\theta}_m) - \theta,$$

where $E(\hat{\theta}_m)$ is the expectation over data and θ is the true value. When the difference between the expectation and true value is zero, the estimator is unbiased.

We define two types of sample variances $\hat{\sigma}_m^2$ and $\tilde{\sigma}_m^2$

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2 \qquad \text{and} \qquad \tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2.$$

For a 1-D Gaussian distribution, show whether these two sample variances are biased or unbiased. *Hint:* $\mathbb{E}\left[\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] = (m-1)\cdot\sigma^2$.

b) Bias Variance Trade-off *(5 points)*

The *mean squared error* (MSE) of an estimator is defined as

$$\text{MSE}(\hat{\theta}_m) = \mathbb{E}\left[\left(\hat{\theta} - \theta\right)^2\right].$$

The MSE measures the expected squared "deviation" between the estimator and the true parameter value and is a good measure for the generalization error. Derive the bias variance trade-off from the definition of MSE.

c) MAP for a conditional likelihood *(5 points)*

The *Maximum A Posteriori* (MAP) estimator is defined as

$$\boldsymbol{\theta}_{\text{MAP}} = \underset{\boldsymbol{\theta}}{\text{argmax}}\ p(\boldsymbol{\theta}|\boldsymbol{X}).$$

By treating $\boldsymbol{\theta}$ as a random variable and applying Bayes Rule, we get

$$\boldsymbol{\theta}_{\text{MAP}} = \underset{\boldsymbol{\theta}}{\text{argmax}}\ \left[\sum_{i=1}^{m}\log p(x^{(i)}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta})\right].$$

The MAP estimate can also be applied to the conditional likelihood $p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\theta})$. Derive MAP of conditional likelihood based on Bayes' rule.

d) Derivatives and the chain rule ***(5 points)***
In a binary classification problem, the goal is to assign the correct binary class label $y_i \in \{0, 1\}$ to a vector $x_i \in \mathbb{R}^d$, where i is an index. Consider the parametric linear classifier

$$f_{W,b}(x_i) = \sigma\big(\langle W, x_i\rangle + b\big)$$

where $W = (W_1, ..., W_d)^T \in \mathbb{R}^d$ is called the weight vector, $b \in \mathbb{R}$ is called the bias, $\langle \cdot, \cdot \rangle$ denotes the scalar product and $\sigma(t) = \frac{1}{1+e^{-t}}$ is the Sigmoid function. During training of the classifier, the weight vector W and the bias b are optimized to minimize a squared L_2-loss between the predicted class label and the true class label

$$L(w, b) = \sum_{i=1}^{N} (y_i - f_{W,b}(x_i))^2.$$

Compute the partial derivatives $\frac{\partial}{\partial W_j} L(W, b)$ and $\frac{\partial}{\partial b} L(W, b)$. How does the gradient $\nabla_W L(W, b)$ look like and how can we use the (partial) derivatives to find a local minimum of the function L?

2 Programming Exercises *(15 points)*

In this assignment, you will implement, train, test and report the performance of a k-Nearest Neighbor (kNN) classifier and a linear classifier with the logistic function. You will use the Fashion-MNIST dataset, which is an alternative to the popular but simplistic MNIST dataset set, which in turn has become a benchmark for testing a wide range of classification algorithms. Please download the following 4 files:

- `train-images-idx3-ubyte.gz`
- `train-labels-idx1-ubyte.gz`
- `t10k-images-idx3-ubyte.gz`
- `t10k-labels-idx1-ubyte.gz`

To access the dataset in your python script, you can use the `load_mnist`-function provided in supplementary material to this exercise. You can access the data as follows:

```
from load_mnist import load_mnist
images, label = load_mnist(dataset="training", path=".")
```

The functions returns a tensor with shape $(N, 28, 28)$ which contains N grey-scale images with values in the range of $[0, 255]$ and a tensor with shape (N) which contains labels $y_i \in [0, 9]$. By passing the argument `dataset="training"`, the functions returns the 60,000 training images and labels. The other option is to pass the argument `dataset="testing"` to obtain the 10,000 test images and labels.

a) K-Nearest Neighbors (kNN) Classifier *(5 points)*

K-Nearest Neighbors (kNN) classifiers are very simple machine learning algorithms which are often used as a benchmark for more complex algorithms. kNN requires no work at training time, it only stores all training data and their labels. At test time, for a given test data, it finds the k closest training data points according to some distance metrics and predicts a class label based on majority voting. The kNN algorithm has two hyper-parameters, k and the distance metric. k is the number of closest training data points and the distance metric is a function that is used to compute the similarity between pairs of data points. In this exercise, you will use the Euclidean distance as the distance metric for your kNN classifiers and cross validate the value of k.

1. *(2 points)* Implement a Python class called `KNNClassifier` which **only uses PyTorch functions** to evaluate a kNN-classifier as described above. Furthermore, **your implementation must avoid loops and match the following interface**:

   - The constructor of your class takes an integer value k, which controls the number of closest training data points, a tensor of training images and a tensor of training labels.
   - Your class has to implement a member function `forward(self, x)`, which takes a tensor of images as its input and returns the labels obtained using the kNN-classifier algorithm.

2. *(3 points)* After you implemented your kNN-classifier it is time to test your implementation and to investigate the influence of the hyperparameter k. Use the Fashion-MNIST dataset and implement the following tasks:

   - Load the complete Fashion-MNIST dataset and construct a training set with 1000 examples by randomly selecting from the training set, 100 samples per class. Let y_i^* be a class label of the test set and y_i be the predicted class label. The accuracy of your classifier can be computed as

   $$\frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \mathbb{I}(y_i^*, y_i) \text{ where } \mathbb{I}(y_i^*, y_i) = \begin{cases} 1 & \text{if } y_i^* = y_i \\ 0 & \text{otherwise.} \end{cases}$$

   Implement a function to compute the accuracy and report the accuracy for the complete test set with $k = 1$ and $k = 5$ neighbours on your newly constructed training dataset.

- Visualize 1 and 5 nearest neighbour images for the first 10 test samples. Plot your confusion matrix to see which classes are mixed up with each other. (You can use the sklearn.metrics library to compute the confusion matrix)
- Implement a function which performs a 5-fold cross-validation on your training dataset. Use validation part to test the accuracy of your kNN-classifier. Test your functions for k neighbours $k = 1, 2, ..., 15$ and plot these accuracies. Which k is performing best?

b) Dataloading and Preprocessing *(5 points)*

Efficiently loading and processing the elements of a dataset is a fundamental task when training any kind of data-driven machine learning model, and especially if you are going to train a deep learning model. In this assignment, you will learn to load the Fashion-MNIST dataset using PyTorch's data loading utility called "DataLoader"

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=32,
                                         shuffle=True, drop_last=False)
```

The arguments in the constructor of the "DataLoader"-class have the following meaning:

- `dataset` - An instance of your "Dataset"-class.
- `batch_size` - The number of elements which are included in a single batch.
- `shuffle` - Tells the "dataloader" to randomize the order of the elements.
- `drop_last` - Allows the last batch to include fewer elements than the batch size.

A complete list of all arguments for the "DataLoader"-class can be found here. A key advantage of using the "DataLoader" class compared to a simple loop is that the framework allows for multithreaded prefetching of dataset elements. This ensures that your training is not bottlenecked by IO-operations.

1. *(3 points)* The image and label tensors cannot be passed to the "DataLoader" directly, and instead you have to implement a "Dataset"-Class first. Your dataset class must meet the following requirements:
   - The constructor of your class needs to unpack the **training data** from the Fashion-MNIST. **You dataset-class should only consider dataset elements with label 0 or 1 and discard all other elements.** Furthermore, you should compute the mean and variance of the remaining training images in the constructor.

- Your "Dataset"-class has to implement a function `__len__(self)`, which returns the number of examples with class label 0 or 1 in the dataset.
- Your "Dataset"-class has to implement a function `__getitem__(self, index)`, which takes an index as its input and returns the processed image and label associated with this index. Data normalization is beneficial when addressing a classification problem. The processed image should be shifted and scaled, such that the resulting dataset has zero mean and standard variance. Furthermore, the processing step should ensure, that all pixel values all lie within an interval of $[-1, 1]$.

2. *(2 points)* Now that have you implemented a Dataset-Class to access the elements of the dataset, you can create an instance of the "DataLoader"-class and loop over batches and labels.

c) Linear Classifier with Logistic Function *(5 points)* You already computed the derivatives for a simple linear classifier in the theoretical part of this exercise sheet. In this task you will implement such a classifier in "PyTorch". But before you can get started, you need a short introduction into optimizers and loss functions in "PyTorch". Generally speaking, an optimizer class takes a list of gradients as its input, implements a gradient descent algorithm, and modifies the weights associated with these gradients. You will use a simple stochastic gradient descent optimizer, which you can create with

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.005)
```

where `model` is an instance of a class which inherited from `torch.nn.Module`. For simplicity we will call such a class a "Network"-class.

Every "PyTorch"-optimizer implements the functions `optimizer.zero_grad()`, which resets the gradient to zero and `optimizer.step()` which performs a single optimization step on the parameters. These two functions are called in every training iteration, but we still have to define a loss, and tell the framework which gradient it should compute. As a loss we use the mean squared error, which you can compute by calling

```
loss = torch.nn.functional.mse_loss(predictions, labels, reduction='mean')
```

where `predictions` are obtained by calling an instance of a "Network"-class, and `labels` are obtained by iterating over an instance of the "DataLoader"-class. Recall from the theoretical exercise that we are interested computing the gradient of the loss function. This can be achieved by calling `loss.backward()`.

So let us summarize a single optimization step for a model in PyTorch: First you reset the gradient of your optimizer to zero, then you compute the predicted labels by calling an

instance of your "Network"-class with the current training batch as the input. Afterwards you compute the loss between the predicted label, and the true labels of the training batch, and tell the network to compute the gradient w.r.t to the loss tensor. Finally you call the step function of your optimizer to update the weights of your network.

1. *(2 points)* The first step is to create a "Network"-class which describes the architecture of your network. Your network class has to meet the following requirements:

   - Your "Network"-class has to inherit the class `torch.nn.Module`.
   - The constructor of your class has to call the constructor of its parent class. Furthermore you need to create a single fully-connected layer as a member variable. This linear layer should be able to map a image represented as a vector to a single value. A fully-connected layer is an instance of the class

     ```
     torch.nn.Linear(in_features, out_features)
     ```

     where `in_features` is the number of coefficients in the input vector and `out_features` the number of coefficients in the resulting vector.
   - Your class needs to implement a function called `forward(self, x)`, where x will be a training batch. This functions has to pass the input through the linear layer you defined in the constructor and return the sigmoid of this vector. You can compute the element-wise sigmoid of a vector with `torch.sigmoid`.
     Note : If your class inherited from `torch.nn.Module` and implements `forward(self, x)`, an instance of your class is callable and executes the implementation in `forward`.

2. *(1 point)* Implement a function `train(model, dataloader, optimizer)` which takes an instance of your model, a dataloader instance and an optimizer instance as its input. This function has to iterate over all training batches in the dataloader, and perform an optimization step in every iteration. In addition the function should print the current loss for every 100th training batch.

3. *(1 point)* Now it is time to train your first network. Create an instance of your network with the name "model", an instance of a stochastic gradient descent optimizer with a learning rate of 0.005, an instance of your Dataset-Class and use it to create an iterable instance of the "DataLoader"-class. Finally implement a loop in which you call your function `train`. Every full iteration over the entire dataset is called an epoch. So this outer loop controls the number of training epochs.

4. *(1 point)* Plot your loss function for 25 epochs and report the accuracy of all elements in the test dataset, which have either class label 0 or 1.