Exercise Sheet 7

1. Exercise: Linear Classifier for Multiple Classes (MNIST)

(a) Load the digits dataset provided by the scikit-learn (`sklearn`) library and generate the following subsets. Note that each subset consists of data samples and their corresponding labels.

  (i) `train_data`: The training subset of the datasest makes up 70% of the original dataset. This is the largest portion of the dataset and it's used directly for training the model.

  (ii) `test_data`: The test subset of the datasest makes up 30% of the original dataset. The test dataset is a subset used to provide an unbiased evaluation of a final trained model. It is used used to verify the accuracy of the model built.

(b) Implement the t-SNE algorithm to reduce the data dimensionality down to four dimensions ($f_{\text{t-SNE}} : \mathbb{R}^{64} \to \mathbb{R}^4$). Apply this procedure for the training, test, and validation datasets.

(c) Implement the *One-versus-the-rest* multiclass strategy using K linear dichotomy-classifiers. Write a Python class, called `OneVsRest`, that implements the *One-versus-the-rest* multiclass classifier and should include the following methods:

  (i) `__init__`: This function initializes the classifier given the number K of dichotomy-classifiers.

  (ii) `fit`: This should fit the model to the provided training data.

  (iii) `predict`: This should predict class labels for the provided test data samples.

(d) Utilize the *One-versus-the-rest* classifier in order to classify the data samples of the digit test dataset. Therefore, initialize the class using $K = 10$ and fit the classifier using the dimensionality reduced train data ($\mathbb{R}^4$). Utilize the trained model to make predictions on the labels of the (dimensionality reduced) test dataset.

(e) Evaluate the performance of the implemented classifier. Therefore, write a Python function called `evaluate(y_pred, y_gt)` to determine the accuracy of a classification model's predictions. The function computes the accuracy as the proportion of correct predictions over the total number of predictions.

(f) Apply the t-SNE algorithm to reduce the data dimensionality of the original test dataset down to two dimensions ($f_{\text{t-SNE}} : \mathbb{R}^{64} \to \mathbb{R}^2$). Generate one figure with two distinct subplots that show the dimensionally reduced test dataset, using the predicted labels from the *One-versus-the-rest* classifier and the actual ground truth labels, respectively.

2. Exercise: Softmax Classification

This exercise focuses on the implementation of a softmax classifier using the `PyTorch` library. It provides a way to understand the functionalities and potential applications of `PyTorch`, a popular library in the field of machine learning. To implement and use the classifier, follow these steps:

(a) **Data Preprocessing:**
Use the dataset created in Exercise 1 (a). Transform the target classes of the train and test datasets into one-hot encoded class labels.

(b) **Model Definition and Initialization:**
Using `PyTorch`, define a model by creating a class that inherits from the Module class. To do so, start by importing the `PyTorch` library using:
```
import pytorch as torch
```
and generate the class `SoftmaxClassifier(torch.nn.Module)` with the functions:

(i) `__init__(self, n_feat, n_classes)`: This function defines the layers of the classification network. The model should be composed of one linear layer and one softmax function. To define the layers use:
```
super().__init__()
self.linear = torch.nn.Linear(n_inputs, n_outputs)
self.softmax = softmax()%TODO
```
Implement the `softmax()` function without using built-in functionality from a library, computing

$$P(c_l|\boldsymbol{x}) = \frac{e^{a_l}}{\sum_{k=1}^{K} e^{a_k}}, \tag{1}$$

where $\boldsymbol{a} = \boldsymbol{\theta}_k^{\mathrm{T}} \tilde{\boldsymbol{x}}$ is the output of the linear transformation of data $\tilde{\boldsymbol{x}} = [\boldsymbol{x}^{\mathrm{T}} \quad 1]^{\mathrm{T}}$. Note, that the linear layer `torch.nn.Linear()` intrinsically performs the mapping $\boldsymbol{x} \rightarrow \tilde{\boldsymbol{x}}$.

(ii) `forward(self, x)`: Defines how the data $\boldsymbol{x}$ moves through the layers:
```
a = self.linear(x)
p = self.softmax(a)
return p
```

Initialize the model using
```
model = Softmax(n_feat, n_classes)
```
where n_{feat} is the number of data features and n_{classes} is the number of potential class labels.

(c) **Loss Definition:**
Implement the function `cross_entropy(y_pred, y_gt)` which computes and returns the cross entropy between the prediction $\boldsymbol{y}_{\text{pred}}$ and the (one-hot encoded) ground truth class $\boldsymbol{y}_{\text{gt}}$.

(d) **Optimizer Definition:**
`PyTorch` provides many different built-in optimization algorithms like Stochastic Gradient Descent (SGD) which can be used to update the weights of your model. Define a SGD-optimizer for the parameters of the initialized model with the learning rate $lr = 0.01$ using:
```
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
```

(e) **Train the Model:**
This is done by passing your input data through the model in the forward direction, calculating the loss, and then backpropagating the error by calling `loss.backward()`.
Then, you update the weights using the optimization algorithm by calling `optimizer.step()`.
The complete training algorithmic is:

```python
Loss = []
epochs = 100
len_traindata = #TODO
for epoch in range(epochs):
    for i in range(len_traindata):
        x = torch.from_numpy(X_train[i])
        y = torch.from_numpy(y_train[i])
        optimizer.zero_grad()
        y_pred = model(x)
        loss = cross_entropy(y_pred, y)
        loss.backward()
        optimizer.step()
        Loss.append(loss.detach())
print("Model training is finalized!")
```

Perform the training of the model and plot the `Loss` curve, where the x-axis represents the training steps and the y-axis the according loss values.

(f) **Evaluating the Model:**
Once you have trained the model, the model performance is evaluated using the test data. Therefore, set the model in evaluation mode using: `model.eval()` before doing so. Similar to the training routine, iterate (only once!) over the test data and compute the average cross entropy loss.