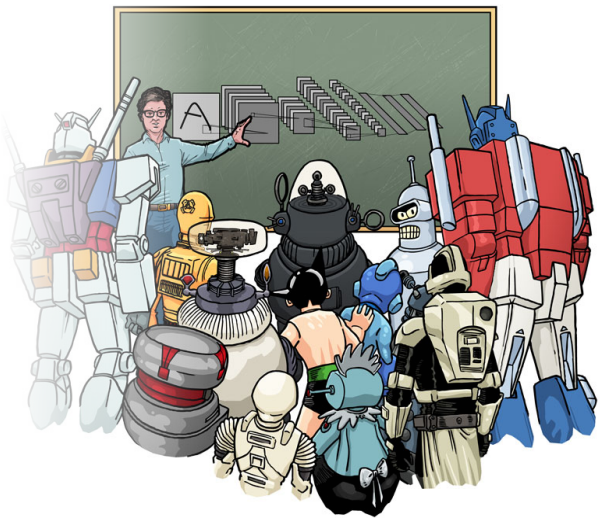
METU EE449
Computational Intelligence

Training Artificial Neural Network



Homework 1 - Training Artificial Neural Network

Due: 23:55, 07/04/2024

Late submissions are welcome, but penalized according to the following policy:

- 1 day late submission: HW will be evaluated out of 70.
- 2 days late submission: HW will be evaluated out of 40.
- 3 or more days late submission: HW will not be evaluated.

You should prepare your homework by yourself alone and you should not share it with other students, otherwise you will be penalized.

Introduction

In this homework, you will perform experiments on artificial neural network (ANN) training and draw conclusions from the experimental results. You will implement and train multi layer perceptron (MLP) and convolutional neural network (CNN) classifiers. The implementations will be in Python language and you will be using PyTorch [1], NumPy [2] and scikit-learn [3]. You can visit the link provided in the references [1–3] to understand the usage of the libraries. To install scikit-learn, you may use `conda install scikit-learn` or `pip install scikit-learn`, depending on your packager.

Homework Task and Deliverables

The homework is composed of 5 parts. In the first part you are to implement an MLP with its forward and backward propagations using NumPy and train it. In the second part, you will implement forward propagation of one Convolutional Layer using NumPy. For the other parts, you will write codes to perform experiments on classification performances under several settings using PyTorch. You will provide the results of the experiments by some visuals and you will interpret the results by your own conclusions.

You should submit a single report in which your answers to the questions, the required experimental results (performance curve plots, visualizations etc.) and your deductions are presented for each part of the homework. Moreover, for all of the parts, you should append your Python codes as **`text`** to the end of the report for each part to generate the results and the visualizations of the experiments. Namely, all the required tasks for a part can be reproduced by running the related code. The codes should be well structured and **well commented**. The non-text submissions (e.g. image) or the submissions lacking

comments will not be evaluated. Similarly answers/results/conclusions written in code as a comment will not be graded. In parts 1-2, you will implement Neural Networks on your own, so **don't use** any NN libraries such as PyTorch or Tensorflow. For parts 3-5, ANNs implemented in an environment **other than** PyTorch will get zero grade.

The report should be in portable document format (pdf) and named as *hw1_name_surname_eXXXXXX* where *name*, *surname* and *X*s are to be replaced by your name, surname and digits of your user ID, respectively. You do not need to send any code files to your course assistant(s), since everything should be in your one pdf file.

Your work will automatically be uploaded to Turnitin and checked for plagiarism alongside that of your peers from both this semester and previous semesters. Any detected misconduct may result in a grade as low as zero for the assignment and may lead to disciplinary action.

Do not include the codes in *utils.py* to the end of your pdf file.

1 Basic Neural Network Construction and Training

1.1 Preliminaries

Using pen and paper, calculate the partial derivatives of following activation functions $\frac{\partial y_k}{\partial x}$. Plot each activation function y_k and corresponding partial derivative $\frac{\partial y_k}{\partial x}$ using matplotlib.

Tanh	Sigmoid	ReLU
$y_1 = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$	$y_2 = \sigma(x) = \frac{1}{1+e^{-x}}$	$y_3 = \max(0, x)$

1.2 Implementation

In this part, you will implement your own MLP (with one hidden layer) using NumPy and simple mathematical operations to solve the XOR problem. For this, firstly you need to call required libraries and functions. You also need to call functions from `utils.py`.

```
import numpy as np
from matplotlib import pyplot as plt
```

Then, you will define an MLP class. You can use the following code directly, but you need to complete missing parts denoted with FILL HERE. You may take activation function and its derivative from 1.1.

```
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))
    def sigmoid(self, x):
        return #FILL HERE
    def sigmoid_derivative(self, x):
        return #FILL HERE
    def forward(self, inputs):
        # Forward pass through the network
        self.hidden_output = #FILL HERE
        self.output = #FILL HERE
        return self.output
    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        output_error = #FILL HERE
        hidden_error = #FILL HERE

        # Compute gradients
        output_delta = #FILL HERE
        hidden_delta = #FILL HERE

        # Update weights and biases
        self.weights_hidden_output = #FILL HERE
        self.bias_output = #FILL HERE
        self.weights_input_hidden = #FILL HERE
        self.bias_hidden = #FILL HERE
```

You should call XOR dataset from `part1CreateDataset` function in `utils.py`. You may generate 1000 training and 100 validation samples.

```
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)
```

Then you may define network parameters, create the network, train and test it.

```
# Define neural network parameters
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.001

# Create neural network
nn = MLP(input_size, hidden_size, output_size)

# Train the neural network
for epoch in range(10000):
    # Forward propagation
    output = #FILL HERE

    # Backpropagation
    #FILL HERE

    # Print the loss (MSE) every 1000 epochs
    if epoch % 1000 == 0:
        loss = #FILL HERE
        print(f'Epoch {epoch}: Loss = {loss}')

# Test the trained neural network
y_predict = #FILL HERE
print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
```

Finally, you should plot your final decision boundary from `part1PlotBoundary` function in `utils.py`.

```
part1PlotBoundary(x_val, y_val, nn)
```

You should repeat the training for sigmoid, tanh and ReLU activations. Once the aforementioned tasks are performed, plot the decision boundaries for each training and put these plots to your report.

1.3 Discussions

1. With and without considering your experimental results, what are the advantages and disadvantages of using each activation function? Which one is useful in which case?
2. What is the XOR problem? Why do we need to use MLP instead of a single layer to solve it?
3. Run your training code a couple of times. Does your decision boundary change with each run or stay the same? Why?

2 Implementing a Convolutional Layer with NumPy

In this part, you will implement your own *torch.conv2d* function *my_conv2d* with default settings (no padding, 1 stride etc.) using NumPy. You are only required to do the forward propagation. You will check whether your implementation is correct by running it in a small batch of MNIST [4] dataset.

2.1 Experimental Work

Download *samples_X.npy* where X is your last digit of your 7-digit student ID as your input, and *kernel.npy* as your kernel file. Load them via following code:

```
import numpy as np

# input shape: [batch size, input_channels, input_height, input_width]
input=np.load('samples_X.npy')
# input shape: [output_channels, input_channels, filter_height, filter width]
kernel=np.load('kernel.npy')

out = my_conv2d(input, kernel)
```

Your *my_conv2d* function should generate an output file *out*. Once the aforementioned tasks are performed, create the output image of your function by using the provided *part2Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Put this output image to your report. Note that you should already have installed PyTorch and Torchvision at this point, since *part2Plots* function needs these libraries for plotting.

2.2 Discussions

After creating the output image, answer the following questions.

1. Why are Convolutional Neural Networks important? Why are they used in image processing?
2. What is a kernel of a Convolutional Layer? What do the sizes of a kernel correspond to?
3. Briefly explain the output image. What happened here?
4. Why are the numbers in the same column look alike to each other, even though they belong to different images?
5. Why are the numbers in the same row do not look alike to each other, even though they belong to same image?
6. What can be deduced about Convolutional Layers from your answers to Questions 4 and 5?

Put your discussions together with output image to your report.

3 Experimenting ANN Architectures

In this part, you will experiment on several ANN architectures for classification task via PyTorch. Use **strides** of 1 for convolutions and 2 for max-pooling operations. Use *valid* **padding** for both convolutions and pooling operations. Use *adaptive moment estimation (Adam)* with default parameters for the optimizer. If your computation power allows, use batch size of 50 samples; reduce batch size accordingly otherwise. Use no weight regularization throughout the all experiments.

Split 10% of the training data as the validation set by randomly taking equal number of samples for each class. Hence, you should have three sets: training, validation and testing.

Dataset Description

The dataset you will work on is Fashion-MNIST dataset [5]. It is composed of 28×28 gray-scale images of 10 clothing classes which are t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. Some samples from the dataset is provided in Fig. 1.

The dataset is provided in *torchvision* module of PyTorch. The dataset is split into two subsets: one for training and one for testing. For training, there are 60000 samples corresponding to 6000 samples for each class and for testing, there are 10000 samples corresponding to 1000 samples for each class.

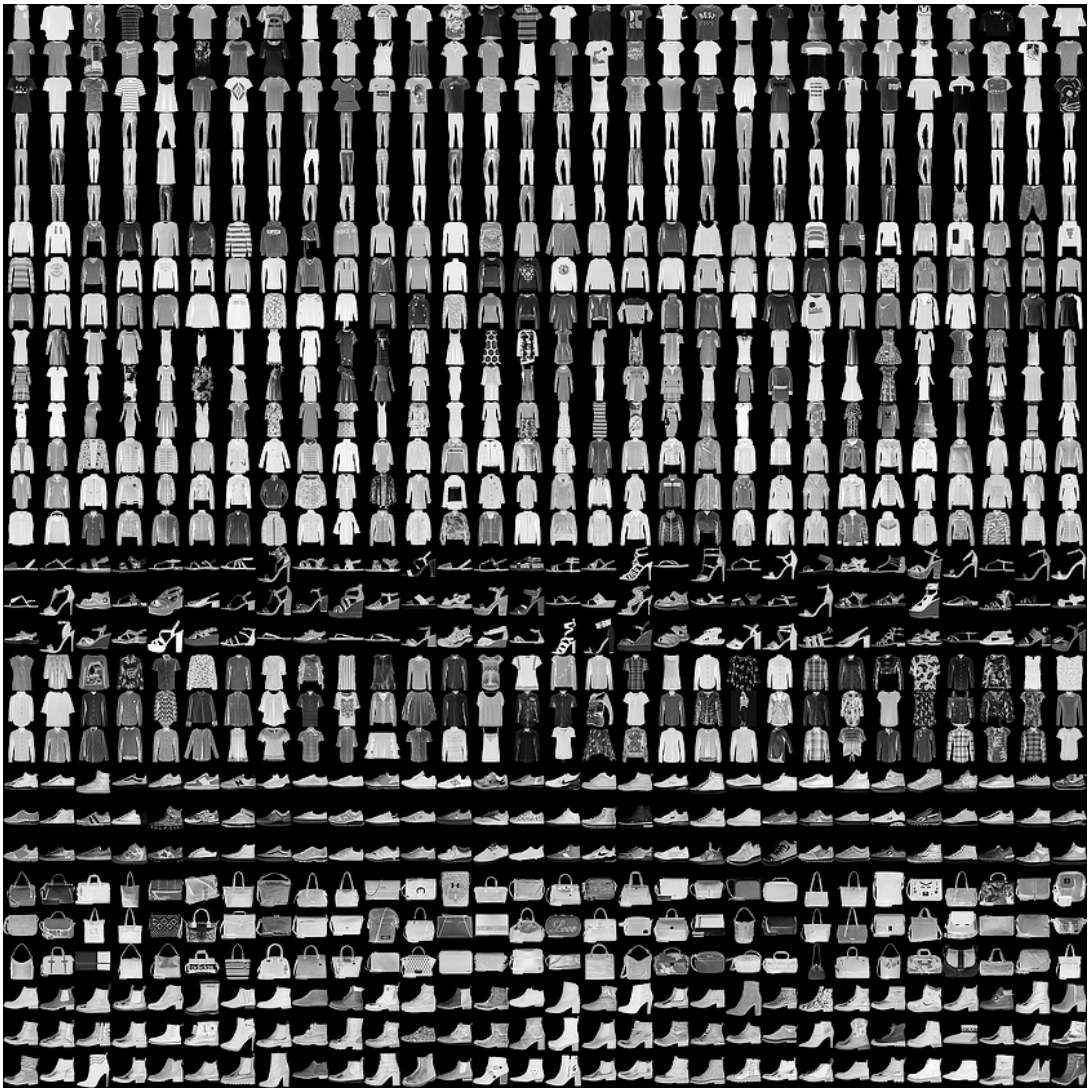


Figure 1: Samples from dataset. The classes are (from top to bottom for every 3 row) t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot.

The images are stored as 1 channel grayscale 28×28 images. One can use them as 2-D directly in a Convolutional Neural Networks (CNN) or they can be flattened to 1-D arrays for Multi Layer Perceptron (MLP) structures.

Each pixel of the image is an integer between 0 and 255. The labels are one-hot-labeled form of integers between 0 and 9 for t-shirt/top (0), trouser (1), pullover (2), dress (3), coat (4), sandal (5), shirt (6), sneaker (7), bag (8) and ankle boot top-wear (9).

In order to download and load Fashion-MNIST dataset, use following command at the beginning of your code.

```
import torchvision

# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
    transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
    transform = torchvision.transforms.ToTensor())
```

You also need to define dataloader in order to train on batches and control your GPU/RAM usage. Select a batch size according to your memory.

```
train_generator = torch.utils.data.DataLoader(train_data, batch_size = 96, shuffle = True)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 96, shuffle = False)
```

PyTorch Library

PyTorch contains many modules and classes for ANN design and training. In the scope of the homework, very limited subset of those modules will suffice. You can simply use module model class (*torch.nn.Module*) together with layers (*torch.nn.Linear, torch.nn.Conv2d* etc.) to create ANN models.

```
import torch
# example mlp classifier
class FullyConnected(torch.nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(FullyConnected, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size)
    self.fc2 = torch.nn.Linear(hidden_size, num_classes)
    self.relu = torch.nn.ReLU()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden = self.fc1(x)
    relu = self.relu(hidden)
    output = self.fc2(relu)
    return output
# initialize your model
model_mlp = FullyConnected(784,128,10)
```

Once you create a model, you can obtain the parameters of desired layer from class attributes.

```
# get the parameters 784x128 layer as numpy array
params_784x128 = model_mlp.fc1.weight.data.numpy()
```

Once you create your classification model, you need to create a loss and an optimizer to compile your model to be trained for the classification task.

```
# create loss: use cross entropy loss)
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.SGD(model_mlp.parameters(), lr = 0.01, momentum = 0.0)
# transfer your model to train mode
model_mlp.train()
# transfer your model to eval mode
model_mlp.eval()
```

Upon compiling a *torch.nn.Module* class for your task, the training and testing operations are simple via calling the class methods. Please refer to the simple explanations of those classes in the web page of the module [1] where you can find examples as well. During training, do not forget optimization steps such as calculating loss, clearing gradients, computing gradients using loss function and taking a gradient descent step using optimizer. **Hint:** Each of these corresponds to one line of code in training. Also, if you want to work in GPU, don't forget to transfer your model and data to CUDA.

3.1 Experimental Work

In the following, FC-N denotes fully connected (dense) layer of size N, Conv-$W\times H\times N$ denotes N many 2-D convolution filters of spatial size $W\times H$ and MaxPool-2×2 denotes max-pooling operation of spatial pool size 2×2.

The name of the ANN architectures to be experimented and their layers are:

'mlp_1' : [FC-32, ReLU] + PredictionLayer

'mlp_2' : [FC-32, ReLU, FC-64(no bias)] + PredictionLayer

'cnn_3' : [Conv-3×3×16, ReLU,
Conv-5×5×8, ReLU, MaxPool-2×2,
Conv-7×7×16, MaxPool-2×2] + PredictionLayer

'cnn_4' : [Conv-3×3×16, ReLU,
Conv-3×3×8, ReLU, Conv-5×5×16, ReLU, MaxPool-2×2,
Conv-5×5×16, ReLU, MaxPool-2×2] + PredictionLayer

'cnn_5' : [Conv-3×3×8, ReLU,
Conv-3×3×16, ReLU, Conv-3×3×8, ReLU, Conv-3×3×16, ReLU, MaxPool-2×2,
Conv-3×3×16, ReLU, Conv-3×3×8, ReLU, MaxPool-2×2] + PredictionLayer

where PredictionLayer = [FC10].

Please note that SoftMax Layer is not required when *torch.nn.CrossEntropyLoss()* is used.

Now, for each architecture, you will perform the following tasks:

1. Using training set, train the ANN for 15 epochs by creating a training loop using your dataloader *train_generator*. Do not forget necessary optimization steps.

   During training,

   - Record the training loss, training accuracy, validation accuracy for every 10 steps to form loss and accuracy curves (**Hint:** You can save training loss with *loss.item()*. You need to write simple codes for calculating training and validation accuracy in *model.eval()* mode;
   - Shuffle training set after each epoch (**Hint:** Your defined dataloader has a shuffle flag).

   After training,

   - Compute test accuracy (**Hint:** Use same code written for calculating validation accuracy);
   - Record the weights of the first layer as numpy array (**Hint:** Call *numpy()* method of the *model.xx.weight.data* attribute where *xx* corresponds to first layer).

2. Now, form a dictionary object with the following key-value pairs as the result of the training experiment for the given architecture:

   - 'name': name of the architecture
   - 'loss_curve': training loss curve of the ANN
   - 'train_acc_curve': training accuracy curve of the ANN
   - 'val_acc_curve': validation accuracy curve of the ANN
   - 'test_acc': test accuracy value of the ANN
   - 'weights': the weight of the first layer of the ANN

3. Save the dictionary object with the filename as the architecture name by prefixing 'part3' in the front(**Hint:** Use *pickle or json* to save dictionary objects to file and load dictionary objects from file).

Once the aforementioned tasks are performed for each architecture, create performance comparison plots by using the provided *part3Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Note that you should pass all the dictionary objects corresponding to results of the experiments as a list to create performance comparison plots. (**Hint:** You can load previously saved results and form a list to be passed to the plot function). Add this plot to your report.

Additionally, for all architectures visualize the weights of the first layer by using the provided *visualizeWeights* function in the *utils.py* file under HW1 folder in ODTUClass course page. Add these visualizations to your report.

3.2 Discussions

Compare the architectures by considering the performances, the number of parameters, architecture structures and the weight visualizations.

1. What is the generalization performance of a classifier?
2. Which plots are informative to inspect generalization performance?
3. Compare the generalization performance of the architectures.
4. How does the number of parameters affect the classification and generalization performance?
5. How does the depth of the architecture affect the classification and generalization performance?
6. Considering the visualizations of the weights, are they interpretable?
7. Can you say whether the units are specialized to specific classes?
8. Weights of which architecture are more interpretable?
9. Considering the architectures, comment on the structures (how they are designed). Can you say that some architecture are akin to each other? Compare the performance of similarly structured architectures and architectures with different structure.
10. Which architecture would you pick for this classification task? Why?

Put your discussions together with performance plots and weight visualizations to your report.

4 Experimenting Activation Functions

In this part, you will compare rectified linear unit (ReLU) function and the logistic sigmoid function. Use SGD for the training method, constant learning rate of 0.01, 0.0 momentum (no momentum), batch size of 50 samples and use no weight regularization throughout the all experiments.

4.1 Experimental Work

Consider the architectures in 3.1, for each architecture create two *torch.nn.Module* objects: one with the ReLU activation function (original archs. in 3.1) and one with the logistic sigmoid activation function (replaces ReLU of archs. in 3.1). Then, perform the following tasks for the two classifiers:

1. Using training set, train the ANN for 15 epochs by creating a training loop using *train_generator*. Do not forget necessary optimization steps.

   During training,

   - Record the training loss and magnitude of the loss gradient with respect to the weights of the first layer at every 10 steps to form loss and gradient magnitude curves (**Hint:** You can save training loss with *loss.item()* and call *numpy()* method of the first item in the *weight.data* attribute to obtain the copies of the weights of the first layer at time steps);
   - Shuffle training set after each epoch (**Hint:** Your defined dataloader has a shuffle flag).

   After training, form a dictionary object with the following key-value pairs as the result of the training experiment for the given architecture:

   - 'name': name of the architecture
   - 'relu_loss_curve': the training loss curve of the ANN with ReLU
   - 'sigmoid_loss_curve': the training loss curve of the ANN with logistic sigmoid
   - 'relu_grad_curve': the curve of the magnitude of the loss gradient of the ANN with ReLU
   - 'sigmoid_grad_curve': the curve of the magnitude of the loss gradient of the ANN with ReLU

2. Save the dictionary object with the filename as the architecture name by prefixing 'part4' in the front (**Hint:** Use *pickle or json* to save dictionary objects to file and load dictionary objects from file).

Once the aforementioned tasks are performed for each architecture, create performance comparison plots by using the provided *part4Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Note that you should pass all the dictionary objects corresponding to results of the experiments as a list to create performance comparison plots. Add this plot to your report.

4.2 Discussions

Compare the architectures by considering the training performances:

1. How is the gradient behavior in different architectures? What happens when depth increases?
2. Why do you think that happens?
3. Is the effect of using different activation functions different or same from Part 1.2?
4. *Bonus:* What might happen if we use inputs in the range $[0, 255]$ instead of $[0.0, 1.0]$?

Put your discussions together with performance plots to your report.

5 Experimenting Learning Rate

In this part, you will examine the effect of the learning rate in SGD method. Use SGD for the optimizer, constant learning rate, ReLU activation function, 0.0 momentum (no momentum), batch size of 50 samples and use no weight regularization throughout the all experiments. You will vary the initial learning rate during the experiments so that each training will be performed with different learning rate.

Split 10% of the training data as the validation set by randomly taking equal number of samples for each class. Hence, you should have three sets: training, validation and testing.

5.1 Experimental Work

Pick your favorite architecture from 3.1, excluding 'mlp_1'. Create three *torch.nn.Module* objects of initial learning rates 0.1, 0.01 and 0.001, respectively. Then, perform the following tasks for the three classifiers:

- Using training set, train the three ANNs for 20 epochs by creating a training loop using *train_generator*. Do not forget necessary optimization steps.

  During training,

- Record the training loss and the validation accuracy for every 10 steps to form loss and accuracy curves (**Hint:** You can save training loss with *loss.item()* and use your previous code to compute validation accuracy);
- Shuffle training set after each epoch (**Hint:** Your defined dataloader has a shuffle flag).

After training, form a dictionary object with the following key-value pairs as the result of the training experiment for the given architecture:

- 'name': name of the architecture
- 'loss_curve_1': the training loss curve of the ANN trained with 0.1 learning rate
- 'loss_curve_01': the training loss curve of the ANN trained with 0.01 learning rate
- 'loss_curve_001': the training loss curve of the ANN trained with 0.001 learning rate
- 'val_acc_curve_1': the validation accuracy curves of the ANN trained with 0.1 learning rate
- 'val_acc_curve_01': the validation accuracy curves of the ANN trained with 0.01 learning rate

- 'val_acc_curve_001': the validation accuracy curves of the ANN trained with 0.001 learning rate

Once the aforementioned tasks are performed for each architecture, create performance comparison plots by using the provided *part5Plots* function in the *utils.py* file under HW1 folder in ODTUClass course page. Note that you should pass a single dictionary objects corresponding to result of the experiment to create performance comparison plots. Add this plot to your report.

Now, you will try to make scheduled learning rate to improve SGD based training.

1. Examine the validation accuracy curve of the ANN trained with 0.1 learning rate. Approximately determine the epoch step where the accuracy stops increasing.
2. Create a *torch.nn.Module* object with the same parameters as above and with initial learning rate of 0.1.
3. Train that classifier until the epoch step that you determined in 1. Then, set the learning rate to 0.01 and continue training until 30 epochs (**Hint:** Create an optimizer with the new learning rate and **retrain** the model with the new optimizer, without **reinitializing** model).
4. Record only the validation accuracy during this training.
5. Now, plot the validation accuracy curve and determine the epoch step where the accuracy stops increasing.
6. Repeat 2 and 3; however, in 3, continue training with 0.01 until the epoch step that you determined in 5. Then, set the learning rate to 0.001 and continue training until 30 epochs.
7. Repeat 4 and once the training ends, record the test accuracy of the trained model and compare it to the same model trained with Adam in 3.1.

Note: You can increase the number of epochs if you are not to observe the steps where training stops improving.

5.2 Discussions

Compare the effect of learning rate by considering the training performances:

1. How does the learning rate affect the convergence speed?
2. How does the learning rate affect the convergence to a better point?
3. Does your scheduled learning rate method work? In what sense?
4. Compare the accuracy and convergence performance of your scheduled learning rate method with Adam.

Put your discussions together with performance plots to your report.

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library." `https://pytorch.org`, 2019.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python." `https://scikit-learn.org/stable/modules/neural_networks_supervised.html`, 2011.

[4] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[5] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.