CS 182: Introduction to Machine Learning, Fall 2022
Homework 5

(Due on Wednesday, Dec. 20 at 11:59pm (CST))

Notice:

- Please submit your assignments via Gradescope. The entry code is G2V63D.
- Please make sure you select your answer to the corresponding question when submitting your assignments.
- Each person has a total of five days to be late without penalty for all the assignments. Each late delivery less than one day will be counted as one day.

1. [10 points] *[Deep Learning Models]*

   (a) Consider a 3D convolution layer. Suppose the input size is $32 \times 32 \times 3$ (width, height, depth) and we use ten 5×5 (width, height) kernels to convolve with it. Set stride $= 1$ and pad $= 2$. What is the output size? Let the bias for each kernel be a scalar, how many parameters do we have in this layer? [5 points]

   (b) The convolution layer is followed by a max pooling layer with 2×2 (width, height) filter and stride $= 2$. What is the output size of the pooling layer? How many parameters do we have in the pooling layer? [5 points]

   Solution:

   (a) If only one such kernel is used to convolve the input, the output is of the size:

   $$\begin{aligned} &width = height = (32 + 2 \times 2 - 5) \div 1 + 1 = 32, \\ &depth = 1, \end{aligned}$$

   i.e., $32 \times 32 \times 1$. Note that the width, height, and depth all follow the same operation rule as in 2D case, and the depth is 1 because the convolution acts as first elementwise multiplication and then addition (which returns only 1 number). Therefore, when one kernel is sliding through the input along the width and the height axes, the depth of the output is 1.
   Still, for one kernel, the number of parameters is

   $$5 \times 5 \times 3 + 1 = 76,$$

   where 1 in the equation above denotes the number of bias.
   Now we have 10 such kernels, so the size of the output is $32 \times 32 \times 10$, and the number of parameters is $76 \times 10 = 760$.

   (b) For the max pooling layer, its input is the output from (a), i.e., the input size is $32 \times 32 \times 10$. So the output size is

   $$\begin{aligned} &width = height = (32 - 2) \div 2 + 1 = 16, \\ &depth = 10, \end{aligned}$$

   i.e., $16 \times 16 \times 10$.
   The max pooling layer has no parameters.

2. [30 points] *[Deep Learning Models]* Principal component analysis (PCA) and autoencoders are popular tools for dimension reduction in machine learning. In this problem, we look into the relation between PCA and the linear autoencoders.

   (a) Given a sample matrix $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_M] \in \mathbb{R}^{d\times M}$ where each column denotes a d-dimensional zero-mean sample. The goal of PCA is to find an orthogonal matrix (transformation) $\mathbf{W} \in \mathbb{R}^{d\times r}$ $(r \leq d)$ which is the solution to

$$\underset{\mathbf{W}}{\text{maximize}} \ \text{Tr}(\mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W}) \quad \text{s.t.} \ \ \mathbf{W}^T\mathbf{W} = \mathbf{I}_r,$$

   where $\text{Tr}(\cdot)$ denotes the trace and $\mathbf{I}_r$ is an $r \times r$ identity matrix.
   Show that

   i. when $r = 1$, $\mathbf{W}$ is exactly the eigenvector of $\mathbf{X}\mathbf{X}^T$ corresponding to its largest eigenvalue. [7 points]

   ii. $\mathbf{W}$ is also the solution to

$$\underset{\mathbf{W}}{\text{minimize}} \ \|\mathbf{X} - \mathbf{W}\mathbf{W}^T\mathbf{X}\|_F^2 \quad \text{s.t.} \ \ \mathbf{W}^T\mathbf{W} = \mathbf{I}_r,$$

   where $\|\cdot\|_F$ is the Frobenius norm, i.e., $\|\mathbf{P}\|_F = \sqrt{\text{Tr}(\mathbf{P}^T\mathbf{P})}$. [7 points]

   (b) Consider a linear autoencoder (as a neural network) with a single hidden layer structure:

$$\mathbf{H} = \mathbf{A}_1\mathbf{X} + \mathbf{b}_1\mathbf{1}^T$$
$$\hat{\mathbf{X}} = \mathbf{A}_2\mathbf{H} + \mathbf{b}_2\mathbf{1}^T,$$

   where $\mathbf{A}_1 \in \mathbb{R}^{r\times d}$ $(\mathbf{A}_2 \in \mathbb{R}^{d\times r})$ and $\mathbf{b}_1 \in \mathbb{R}^{r\times 1}$ $(\mathbf{b}_2 \in \mathbb{R}^{d\times 1})$ are respectively the weight matrix and the bias vector of the layer in the encoder (decoder), and $\mathbf{1} = [1, ..., 1]^T \in \mathbb{R}^M$. One trains the linear autoencoder by minimizing the reconstruction error:

$$\{\mathbf{A}_1^\star, \mathbf{b}_1^\star, \mathbf{A}_2^\star, \mathbf{b}_2^\star\} = \underset{\mathbf{A}_1,\mathbf{b}_1,\mathbf{A}_2,\mathbf{b}_2}{\text{argmin}} \ \|\mathbf{X} - \hat{\mathbf{X}}\|_F^2.$$

   Show that

   i. $\mathbf{A}_2^\star$ can be solved from:

$$\underset{\mathbf{A}_2}{\text{minimize}} \ \|\mathbf{X} - \mathbf{A}_2\mathbf{A}_2^\dagger\mathbf{X}\|_F^2,$$

   where $\mathbf{A}_2^\dagger = (\mathbf{A}_2^T\mathbf{A}_2)^{-1}\mathbf{A}_2^T$ is the Moore-Penrose pseudoinverse of $\mathbf{A}_2$. [8 points]
   (*Hint:* use the derivative of Frobenius norm and the fact that $\mathbf{A}_2^\dagger = \underset{\mathbf{A}_1}{\text{argmin}}\|\mathbf{X} - \mathbf{A}_2\mathbf{A}_1\mathbf{X}\|_F^2$)

   ii. the solution $\mathbf{W}$ from (a) can be taken as the same as $\mathbf{A}_2^\star$. [8 points]
   (*Hint:* you may prove it by showing the equivalence of the column spaces spanned by these two matrices)

Solution:

(a) Proof:

i. When $r = 1$, it is easy to show that $\mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{W}$ are scalars. Therefore, one can reformulate the problem as follows:

$$\underset{\mathbf{W}}{\text{maximize}} \ \mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W} \quad \text{s.t.} \ \ \mathbf{W}^T\mathbf{W} = 1.$$

The Lagrangian is

$$L(\mathbf{W}, \lambda) = \mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W} + \lambda(1 - \mathbf{W}^T\mathbf{W}),$$

where λ is the Lagrangian multiplier. Then, consider the stationarity condition in KKT conditions

$$\frac{\partial L}{\partial \mathbf{W}} = 2\mathbf{X}\mathbf{X}^T\mathbf{W} - \lambda\mathbf{W} = 0,$$

which implies that $\mathbf{X}\mathbf{X}^T\mathbf{W} = \lambda\mathbf{W}$. It means that the optimal $\mathbf{W}$ must be an eigevector of $\mathbf{X}\mathbf{X}^T$. Substituting $\mathbf{X}\mathbf{X}^T\mathbf{W} = \lambda\mathbf{W}$ into the original objective, maximizing $\mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W}$ is equivalent to maximizing λ. Hence, the optimal $\mathbf{W}$ is the eigenvector of $\mathbf{X}\mathbf{X}^T$ corresponding to its largest eigenvalue.

ii. Using the definition of Frobenius norm and the properties of trace, the following derivations hold:

$$\begin{aligned}\|\mathbf{X}-\mathbf{W}\mathbf{W}^T\mathbf{X}\|_F^2 &= \mathrm{Tr}\left((\mathbf{X}-\mathbf{W}\mathbf{W}^T\mathbf{X})^T(\mathbf{X}-\mathbf{W}\mathbf{W}^T\mathbf{X})\right)\\ &= \mathrm{Tr}\left(\mathbf{X}^T\mathbf{X}-2\mathbf{X}^T\mathbf{W}\mathbf{W}^T\mathbf{X}+\mathbf{X}^T\mathbf{W}\mathbf{W}^T\mathbf{W}\mathbf{W}^T\mathbf{X}\right)\\ &= \mathrm{Tr}\left(\mathbf{X}^T\mathbf{X}-2\mathbf{X}^T\mathbf{W}\mathbf{W}^T\mathbf{X}+\mathbf{X}^T\mathbf{W}\mathbf{W}^T\mathbf{X}\right)\\ &= \mathrm{Tr}\left(\mathbf{X}^T\mathbf{X}-\mathbf{X}^T\mathbf{W}\mathbf{W}^T\mathbf{X}\right)\\ &= \mathrm{Tr}\left(\mathbf{X}^T\mathbf{X}\right)-\mathrm{Tr}\left(\mathbf{X}^T\mathbf{W}\mathbf{W}^T\mathbf{X}\right)\\ &= \mathrm{Tr}\left(\mathbf{X}^T\mathbf{X}\right)-\mathrm{Tr}\left(\mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W}\right).\end{aligned}$$

Note that, given $\mathbf{X}$, $\mathrm{Tr}\left(\mathbf{X}^T\mathbf{X}\right)$ is a constant that is irrelevant with $\mathbf{W}$. Hence, minimizing $\|\mathbf{X}-\mathbf{W}\mathbf{W}^T\mathbf{X}\|_F^2$ is equivalent to maximizing $\mathrm{Tr}\left(\mathbf{W}^T\mathbf{X}\mathbf{X}^T\mathbf{W}\right)$. It implies that the solutions of these two optimization problems are same.

(b) Proof:

i. Let $\frac{\partial\|\mathbf{X}-\hat{\mathbf{X}}\|_F^2}{\partial\mathbf{b}_2}=\mathbf{0}$, the following condition holds:

$$-2(\mathbf{X}-\mathbf{A}_2\mathbf{A}_1\mathbf{X}-\mathbf{A}_2\mathbf{b}_1\mathbf{1}^T-\mathbf{b}_2\mathbf{1}^T)\mathbf{1}=\mathbf{0}.$$

Since each $\mathbf{X}$ represents a zero-mean sample matrix, one has $\mathbf{X}\mathbf{1}=\mathbf{0}$. Substituting it into the condition above gives

$$\mathbf{b}_2=\mathbf{A}_2\mathbf{b}_1.$$

Therefore, one can simplify $\|\mathbf{X}-\hat{\mathbf{X}}\|_F^2$ as $\|\mathbf{X}-\mathbf{A}_2\mathbf{A}_1\mathbf{X}\|_F^2$. Then using the fact that $\mathbf{A}_2^\dagger=\underset{\mathbf{A}_1}{\mathrm{argmin}}\|\mathbf{X}-\mathbf{A}_2\mathbf{A}_1\mathbf{X}\|_F^2$ gives

$$\mathbf{A}_2^\star\in\underset{\mathbf{A}_2}{\mathrm{argmin}}\ \|\mathbf{X}-\mathbf{A}_2\mathbf{A}_2^\dagger\mathbf{X}\|_F^2.$$

ii. Denote $\mathbf{W}^\star$ as the optimal solution in (a)i. Since $\mathbf{W}^T\mathbf{W}=\mathbf{I}_r$, we have

$$\mathbf{W}^\star\in\underset{\mathbf{W}}{\mathrm{argmin}}\ \|\mathbf{X}-\mathbf{W}\mathbf{W}^T\mathbf{X}\|_F^2=\underset{\mathbf{W}}{\mathrm{argmin}}\ \|\mathbf{X}-\mathbf{W}(\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{X}\|_F^2.$$

And from (b)i, we know

$$\mathbf{A}_2^\star\in\underset{\mathbf{A}_2}{\mathrm{argmin}}\ \|\mathbf{X}-\mathbf{A}_2(\mathbf{A}_2^T\mathbf{A}_2)^{-1}\mathbf{A}_2^T\mathbf{X}\|_F^2.$$

The key difference between $\mathbf{W}^\star$ and $\mathbf{A}_2^\star$ is that $\mathbf{W}^\star$ is orthogonal while $\mathbf{A}_2^\star$ is not. But they share a same column space. Hence, we can set $\mathbf{A}_2^\star$ as orthogonal to make $\mathbf{W}^\star=\mathbf{A}_2^\star$.

3. [15 points] *[Ensemble Learning]* Suppose there are L independent two-class classifiers used for simple voting and the output of classifier j ($j = 1, \ldots, L$) is denoted as d_j. From the point of view that the mean squared error of an estimator can be decomposed into the bias part and the variance part, explain why increasing L can lead to the increase of the classification accuracy.

Solution:

Let $\mathbb{E}[d_j]$ and $\mathrm{Var}(d_j)$ are the expected value and variance of d_j for classifier j, where d_j is the output of the jth classifier. Expected value and variance of output for independent classifiers:

$$\mathbb{E}[y] = \mathbb{E}\left[\sum_j \frac{1}{L} d_j\right] \geq \frac{1}{L} L \min_j \{\mathbb{E}[d_j]\} = \min_j \{\mathbb{E}[d_j]\}$$

$$\mathrm{Var}(y) = \mathrm{Var}\left(\sum_j \frac{1}{L} d_j\right) = \frac{1}{L^2} \mathrm{Var}\left(\sum_j d_j\right) \leq \frac{1}{L^2} L \max_j \{\mathrm{Var}(d_j)\} = \frac{1}{L} \max_j \{\mathrm{Var}(d_j)\}$$

As L increases, the expected value (and hence the bias) does not change but the variance decreases, and hence the mean squared error of the estimator y decreases, leading to an increase in accuracy.

4. [15 points] *[Model Assessment and Selection]* Suppose we carry out a K-fold cross-validation on a dataset and obtain the classification error rates $\{p_i\}_{i=1}^K$, describe the steps of a one-sided t test on testing the null hypothesis H_0 that the classifier has error percentage p_0 or less at a significance level α.

Solution:

Let

$$m = \frac{\sum_{i=1}^K p_i}{K}, \quad S^2 = \frac{\sum_{i=1}^K (p_i - m)^2}{K-1},$$

we have

$$\sqrt{K}\frac{(m - p_0)}{S} \sim \tau_{K-1},$$

and hence we will fail to reject at significance level α if

$$\sqrt{K}\frac{m - p_0}{S} \in (-\infty, t_{\alpha,K-1}).$$

5. [30 points] *[Coding: CNN]* Complete "HW5-Coding.ipynb". After completion, you should convert your notebook to PDF, and concatenate the writing part and the coding part into one PDF which is the file to submit.

Homework 5: Convolutional neural network (20 points)

In this part, you need to implement and train a convolutional neural network on the CIFAR-10 dataset with PyTorch.

What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

How can I learn PyTorch?

Justin Johnson has made an excellent tutorial (https://github.com/jcjohnson/pytorch-examples) for PyTorch.

You can also find the detailed API doc (http://pytorch.org/docs/stable/index.html) here. If you have other questions that are not addressed by the API docs, the PyTorch forum (https://discuss.pytorch.org/) is a much better place to ask than StackOverflow.

Install PyTorch and Skorch.

In [ ]:

```
!pip install -q torch skorch torchvision torchtext
```

```
     |████████████████████████████████████████| 112kB 4.8MB/s eta 0:00:01
```

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import skorch
import sklearn
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:144: FutureWarning: The sklearn.metrics.scorer module is
deprecated in version 0.22 and will be removed in version 0.24. The corresponding classes / functions should instead be import
ed from sklearn.metrics. Anything that cannot be imported from sklearn.metrics is now part of the private API.
  warnings.warn(message, FutureWarning)
```

0. Tensor Operations

Tensor operations are important in deep learning models. In this part, you are recommended to get famaliar to some common tensor operations in PyTorch. This part is not attached to points.

1) Tensor squeezing, unsqueezing and viewing

Tensor squeezing, unsqueezing and viewing are important methods to change the dimension of a Tensor, and the corresponding functions are torch.squeeze (https://pytorch.org/docs/stable/torch.html#torch.squeeze), torch.unsqueeze (https://pytorch.org/docs/stable/torch.html#torch.unsqueeze) and torch.Tensor.view (https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view). Please read the documents of the functions, and finish the following practice.

In [ ]:

```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2],
                  [3, 4],
                  [5, 6]])
x.shape
# Add two new dimensions to x by using the function torch.unsqueeze, so that the size of x becomes (3, 1, 2, 1).
x = torch.unsqueeze(torch.unsqueeze(x,1),-1)
print(x.shape)
# Remove the two dimensions justed added by using the function torch.squeeze, and change the size of x back to (3, 2).
x = torch.squeeze(torch.squeeze(x,-1),1)
print(x.shape)
# x is now a two-dimensional tensor, or in other words a matrix. Now use the function torch.Tensor.view and change x to a one-dimensional vect
x = torch.Tensor.view(x,6)
print(x.shape)
```

```
torch.Size([3, 1, 2, 1])
torch.Size([3, 2])
torch.Size([6])
```

2) Tensor concatenation and stack

Tensor concatenation and stack are operations to combine small tensors into big tensors. The corresponding functions are torch.cat (https://pytorch.org/docs/stable/torch.html#torch.cat) and torch.stack (https://pytorch.org/docs/stable/torch.html#torch.stack). Please read the documents of the functions, and finish the following practice.

In [ ]:

```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])

# y is a tensor with size being (3, 2)
y = torch.Tensor([[-1, -2], [-3, -4], [-5, -6]])

# Our goal is to generate a tensor z with size as (2, 3, 2), and z[0,:,:] = x, z[1,:,:] = y.

# Use torch.stack to generate such a z
z = torch.stack((x,y),0)
print(z[0,:,:])
# Use torch.cat and torch.unsqueeze to generate such a z
z = torch.cat((torch.unsqueeze(x,0),torch.unsqueeze(y,0)))
print(z[1,:,:])
```

Out[5]:

```
tensor([[-1., -2.],
        [-3., -4.],
        [-5., -6.]])
```

3) Tensor expansion

Tensor expansion is to expand a tensor into a larger tensor along singleton dimensions. The corresponding functions are torch.Tensor.expand (https://pytorch.org/docs/stable/tensors.html#torch.Tensor.expand) and torch.Tensor.expand_as (https://pytorch.org/docs/stable/tensors.html#torch.Tensor.expand_as). Please read the documents of the functions, and finish the following practice.

In [ ]:

```
# x is a tensor with size being (3)
x = torch.Tensor([1, 2, 3])

# Our goal is to generate a tensor z with size (2, 3), so that z[0,:,:] = x, z[1,:,:] = x.

# [TO DO]
# Change the size of x into (1, 3) by using torch.unsqueeze.
x = torch.unsqueeze(x,0)
print(x.shape)

# [TO DO]
# Then expand the new tensor to the target tensor by using torch.Tensor.expand.
z = x.expand(2,-1)
print(z.shape)
```

```
torch.Size([1, 3])
torch.Size([2, 3])
tensor([1., 2., 3.])
```

4) Tensor reduction in a given dimension

In deep learning, we often need to compute the mean/sum/max/min value in a given dimension of a tensor. Please read the document of torch.mean (https://pytorch.org/docs/stable/torch.html#torch.mean), torch.sum (https://pytorch.org/docs/stable/torch.html#torch.sum), torch.max (https://pytorch.org/docs/stable/torch.html#torch.max), torch.min (https://pytorch.org/docs/stable/torch.html#torch.min), torch.topk (https://pytorch.org/docs/stable/torch.html#torch.topk), and finish the following practice.

In [ ]:

```
# x is a random tensor with size being (10, 50)
x = torch.randn(10, 50)

# Compute the mean value for each row of x.
# You need to generate a tensor x_mean of size (10), and x_mean[k, :] is the mean value of the k-th row of x.
x_mean = torch.mean(x,1)
print(x_mean[3, ])

# Compute the sum value for each row of x.
# You need to generate a tensor x_sum of size (10).
x_sum = torch.sum(x,1)
print(x_sum.shape)

# Compute the max value for each row of x.
# You need to generate a tensor x_max of size (10).
x_max = torch.max(x,1)[0]
print(x_max.shape)

# Compute the min value for each row of x.
# You need to generate a tensor x_min of size (10).
x_min = torch.min(x,1)[0]
print(x_min.shape)

# Compute the top-5 values for each row of x.
# You need to generate a tensor x_mean of size (10, 5), and x_top[k, :] is the top-5 values of each row in x.
x_mean_xtop = x.topk(5,1).values
print((x_mean_xtop.shape))
```

```
torch.Size([10])
torch.Size([10])
torch.Size([10])
torch.Size([10, 5])
```

Convolutional Neural Networks

Implement a convolutional neural network for image classification on CIFAR-10 dataset.

CIFAR-10 is an image dataset of 10 categories. Each image has a size of 32x32 pixels. The following code will download the dataset, and split it into `train` and `test`. For this question, we use the default validation split generated by Skorch.

In [ ]:

```
train = torchvision.datasets.CIFAR10("./data", train=True, download=True)
test = torchvision.datasets.CIFAR10("./data", train=False, download=True)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)
to ./data/cifar-10-python.tar.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

The following code visualizes some samples in the dataset. You may use it to debug your model if necessary.

In [ ]:

```
def plot(data, labels=None, num_sample=5):
  n = min(len(data), num_sample)
  for i in range(n):
    plt.subplot(1, n, i+1)
    plt.imshow(data[i], cmap="gray")
    plt.xticks([])
    plt.yticks([])
    if labels is not None:
      plt.title(labels[i])

train.labels = [train.classes[target] for target in train.targets]
plot(train.data, train.labels)
```

1) Basic CNN implementation

Consider a basic CNN model

- It has 3 convolutional layers, followed by a linear layer.
- Each convolutional layer has a kernel size of 3, a padding of 1.
- ReLU activation is applied on every hidden layer.

Please implement this model in the following section. The hyperparameters is then be tuned and you need to fill the results in the table.

a) Implement convolutional layers (10 Points)

Implement the initialization function and the forward function of the CNN.

In [ ]:

```
class CNN(nn.Module):
  def __init__(self, channels):
    super(CNN, self).__init__()
    # implement parameter definitions here
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    self.conv1 = nn.Conv2d(3,channels,3,padding=1)
    self.conv2 = nn.Conv2d(channels,channels,3,padding=1)
    self.conv3 = nn.Conv2d(channels,channels,3,padding=1)
    self.fc1 = nn.Linear(channels*32*32,10)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  def forward(self, images):
    # implement the forward function here
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    images = images.float()
    images = F.relu(self.conv1(images))
    images = F.relu(self.conv2(images))
    images = F.relu(self.conv3(images))
    images = images.view(images.size(0), -1)
    images = self.fc1(images)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return images
```

b) Tune hyperparameters

Train the CNN model on CIFAR-10 dataset. Tune the number of channels, optimizer, learning rate and the number of epochs for best validation accuracy.

In [ ]:

```
# implement hyperparameters here

learning_rate = [1e-5, 1e-4, 1e-3, 1e-2]
optimize = [torch.optim.SGD, torch.optim.Adam]
channel = [128, 256, 512]


train_data_normalized = torch.Tensor(train.data/255)
train_data_normalized = train_data_normalized.permute(0,3,1,2)

for l in learning_rate:
  for o in optimize:
    for c in channel:
      print(f'The channel was {c}, the learning rate was {l} and the optimizer was {str(o)}')

      cnn = CNN(channels = c)

      model = skorch.NeuralNetClassifier(cnn, criterion=torch.nn.CrossEntropyLoss,
                                         device="cuda",
                                         optimizer=o,
                                        # optimizer__momentum=0.90,
                                         lr=l,
                                         max_epochs=25,
                                         batch_size=64,
                                         callbacks=[skorch.callbacks.EarlyStopping(lower_is_better=True)])
      # implement input normalization & type cast here
      model.fit(train_data_normalized, np.asarray(train.targets))
```

```
The channel was 128, the learning rate was 1e-05 and the optimizer was <class 'torch.optim.sgd.SGD'>
  epoch    train_loss    valid_acc    valid_loss      dur
-------  ------------  -----------  ------------  -------
      1        2.3032       0.0999        2.3027  18.5060
      2        2.3021       0.0990        2.3016  18.4268
      3        2.3010       0.1131        2.3005  18.4448
      4        2.3000       0.1135        2.2995  18.4798
      5        2.2990       0.1079        2.2985  18.4404
      6        2.2981       0.1052        2.2975  18.4569
      7        2.2971       0.1037        2.2965  18.4508
      8        2.2961       0.1036        2.2956  18.4712
      9        2.2952       0.1029        2.2946  18.4494
     10        2.2942       0.1029        2.2936  18.4175
     11        2.2933       0.1035        2.2926  18.4708
     12        2.2923       0.1039        2.2916  18.4207
     13        2.2913       0.1053        2.2906  18.4349
     14        2.2903       0.1067        2.2895  18.4755
     15        2.2893       0.1078        2.2884  18.4174
     16        2.2882       0.1100        2.2874  18.3775
```

Write down **validation accuracy** of your model under different hyperparameter settings. Note the validation set is automatically split by Skorch during `model.fit()` .

Hint: You may need more epochs for SGD than Adam.

#channel for each layer \ optimizer	SGD	Adam
(128, 128, 128)	64.13	65.91
(256, 256, 256)	63.35	64.89
(512, 512, 512)	63.93	66.12

2) Full CNN implementation (10 points)

Based on the CNN in the previous question, implement a full CNN model with max pooling layer.

- Add a max pooling layer after each convolutional layer.
- Each max pooling layer has a kernel size of 2 and a stride of 2.

Please implement this model in the following section. The hyperparameters is then be tuned and fill the results in the table. You are also required to complete the questions.

a) Implement max pooling layers

Copy the CNN implementation in previous question. Implement max pooling layers.

In [ ]:

```
class CNN_MaxPool(nn.Module):
  def __init__(self):
    super(CNN_MaxPool, self).__init__()
    # implement parameter definitions here
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    self.conv1 = nn.Conv2d(3,512,3,padding=1)
    self.conv2 = nn.Conv2d(512,512,3,padding=1)
    self.conv3 = nn.Conv2d(512,512,3,padding=1)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    self.fc1 = nn.Linear(512*4*4,10)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****



  def forward(self, images):
    # implement the forward function here
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    images = self.pool(F.relu(self.conv1(images)))
    images = self.pool(F.relu(self.conv2(images)))
    images = self.pool(F.relu(self.conv3(images)))
    images = images.view(images.size(0), -1)
    images = self.fc1(images)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return images
```

b) Tune hyperparameters

Based on best optimizer you found in the previous problem, tune the number of channels and learning rate for best validation accuracy.

In [ ]:

```
model = skorch.NeuralNetClassifier(CNN_MaxPool, criterion=torch.nn.CrossEntropyLoss,
                                   device="cuda",
                                   optimizer=torch.optim.Adam,
                                   lr=0.0001,
                                   max_epochs=100,
                                   batch_size=64,
                                  callbacks=[skorch.callbacks.EarlyStopping(lower_is_better=True)],)

# implement input normalization & type cast here
train_data_normalized = torch.Tensor(train.data/255)
train_data_normalized = train_data_normalized.permute(0,3,1,2)

model.fit(train_data_normalized, np.asarray(train.targets))
```

```
  epoch    train_loss    valid_acc    valid_loss      dur
-------  ------------  -----------  ------------  -------
      1        1.6548       0.5282        1.3343  16.7617
      2        1.2446       0.6112        1.1179  16.7871
      3        1.0674       0.6559        0.9982  16.7890
      4        0.9457       0.6755        0.9350  16.7998
      5        0.8547       0.6916        0.8925  16.7792
      6        0.7818       0.7005        0.8618  16.7560
      7        0.7190       0.7089        0.8419  16.8500
      8        0.6625       0.7167        0.8247  16.8303
      9        0.6106       0.7237        0.8100  16.8161
     10        0.5605       0.7308        0.7962  16.8390
     11        0.5138       0.7336        0.7851  16.8361
     12        0.4679       0.7400        0.7769  16.8542
     13        0.4230       0.7416        0.7835  16.7921
     14        0.3821       0.7402        0.7988  16.8363
     15        0.3428       0.7375        0.8234  16.8182
     16        0.3061       0.7376        0.8576  16.8274
Stopping since valid_loss has not improved in the last 5 epochs.
```

Out[89]:

```
<class 'skorch.classifier.NeuralNetClassifier'>[initialized](
  module_=CNN_MaxPool(
    (conv1): Conv2d(3, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=8192, out_features=10, bias=True)
  ),
)
```

Write down the **validation accuracy** of your model under different hyperparameter settings.

#channel for each layer	validation accuracy
(128, 128, 128)	72.62
(128, 256, 512)	74.77
(256, 256, 256)	73.88
(256, 512, 1024)	74.13
(512, 512, 512)	74.76
(512, 1024, 2048)	74.70

For the best model you have, test it on the test set.

It is fine if you found some hyperparameter combination better than those listed in the tables.

In [ ]:

```
# implement the same input normalization & type cast here
test_data_normalized = torch.Tensor(test.data/255)
test_data_normalized = test_data_normalized.permute(0,3,1,2)
test.predictions = model.predict(test_data_normalized)
sklearn.metrics.accuracy_score(test.targets, test.predictions)
```

Out[27]:

0.7081

How much **test accuracy** do you get?

Your Answer: I get 70.81% of test accuracy

What can you conclude for the design of CNN structure?

Your Answer: Regarding the first simple CNN structure, we get almost the same accuracy (around 64%) for the different hyperparameters settings. We can see that as mentionned above, the SGD optimizer required more epoches than Adam optimizer to converge. We can also see that the increase in the dimension of the channel for each layer was not beneficial in this case. Indeed, we got 65.91% validation accuracy for the channels (128,128,128) with a duration of 18 secondes per epoch, compared to 66.12% validation accuracy for the channels (512, 512, 512) with a duration of 84 secondes per epoch. Meaning much more training time for almost the same validation accuracy!

Regarding the second CNN model, we added 3 MaxPooling layers (one after each convolutionnal layer) that had 2 impressive benefits. The first one is that we increased our validation accuracy by 10%, going from 64% to 74.77% (in the best cases). The second one is that we decreased our training time significantly from 84 secondes to 16 secondes per epoch for the channels configuration (512, 512, 512). We can see that by down-sampling the input dimension and by reducing its dimensionality by keeping the max values, we were able to improve our model accuracy and decrease the training time! In this case, we can conclude that adding layers and pooling parameters we were able to improve the model accuracy.