Machine Vision - Assignment 2: Gradient Descent and Neural Networks in PyTorch

Prof. Dr. Markus Enzweiler, Esslingen University of Applied Sciences

markus.enzweiler@hs-esslingen.de

This is the second assignment for the "Machine Vision" lecture.

It covers:

- polynomial regression using gradient descent
- getting started with PyTorch
- training multilayer perceptrons for traffic sign recognition
- working with public benchmark datasets (German Traffic Sign Recognition Benchmark)

Make sure that "GPU" is selected in Runtime -> Change runtime type

To successfully complete this assignment, it is assumed that you already have some experience in Python and numpy. You can either use Google Colab for free with a private (dedicated) Google account (recommended) or a local Jupyter installation.

Exercise 1 - PyTorch Tutorial (10 points)

Introduction to PyTorch

Work through the "Introduction to PyTorch" tutorial consisting of nine topics:

- Learn the Basics
- Quickstart
- Tensors
- Datasets & DataLoaders
- Transforms
- Build Model
- Autograd
- Optimization
- Save & Load Model

You can run each part in Colab and inspect / modify the code and data. There are certainly some details that you do not yet understand. Don't let that stop you, but try to get a good overall view on working with PyTorch.

Please answer the questions below (directly in the notebook):

Question 1 (3 points)

Why does PyTorch have a dedicated tensor data structure (`torch.tensor`) and does not use NumPy multidimensional arrays exclusively?

Your Answer:

Tensors can run on GPU or other hardware accelarators, which NumPy arrays can not

Question 2 (4 points)

In the "Build Model" section of the tutorial, there is a definition of a neural network, as follows:

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
```

```
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Explain the different parameters of the `nn.Linear()` layers. How are the numeric values, e.g. `28*28`, `512`, `10`, determined?

Your Answer:

28*28 = Image Size of the dataset - so from each pixel one input

10 = output of the neural network, dataset has 10 labels. So for each label we have 1 output

512 = choose of hidden layers. Can be determined by the user. Its common to use numbers by the power of 2. You can test with which number the model performs best

Question 3 (3 points)

In the "Optimization" section of the tutorial, an example of a training loop is given:

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Explain the effect of the function calls `optimizer.zero_grad()` and `optimizer.step()`.

Your Answer:

optimizer.zero_grad() = reset the gradients, so we have no doulbe counting

optimizer.step() = adjust the parameters by the collected gradients

Preparations and Imports

Package Path

```
# Package Path (this needs to be adapted)
packagePath = "./" # local
if 'google.colab' in str(get_ipython()):
  packagePath = "/content/drive/My Drive/MachineVis2/template"   # Colab
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Import important libraries (you should probably start with these lines all the time ...)

```
# os, glob, time, logging
import os, glob, time, logging

# NumPy
import numpy as np

# OpenCV
import cv2

# Matplotlib
import matplotlib.pyplot as plt
# make sure we show all plots directly below each cell
%matplotlib inline

# PyTorch
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Some Colab specific packages
if 'google.colab' in str(get_ipython()):
  # image display
  from google.colab.patches import cv2_imshow
```

Some helper functions that we will need

```
def my_imshow(image, windowTitle=None, size=20, depth=3):
  '''
  Displays an image and differentiates between Google Colab and a local Python installation.

  Args:
    image: The image to be displayed

  Returns:
    -
  '''

  if 'google.colab' in str(get_ipython()):
    print(windowTitle)
    cv2_imshow(image)
  else:
    if (size):
      (h, w) = image.shape[:2]
      aspectRatio = float(h)/w
      figsize=(size, size * aspectRatio)
      plt.figure(figsize=figsize)

    if (windowTitle):
      plt.title(windowTitle)

    if (depth == 1):
      plt.imshow(image, cmap='gray', vmin=0, vmax=255)
    elif (depth == 3):
      plt.imshow(cv2.cvtColor(image, cv2.COLOR_RGB2BGR))
    else:
      plt.imshow(image)
    plt.axis('off')
    plt.show()
```

In Google Colab only:

Mount the Google Drive associated with your Google account. You will have to click the authorization link and enter the obtained authorization code here in Colab.

```
# Mount Google Drive
if 'google.colab' in str(get_ipython()):
  from google.colab import drive
  drive.mount('/content/drive', force_remount=True)
```

PyTorch Trainer and Test Class

In the project package, I have provided a Python file `torchHelpers.py` that contains two classes `Trainer` and `Tester`. These classes contain the neural network training loop and test code, similar to what you have already seen in the tutorial.

The classes can be used as follows (see the documentation of the individual classes):

```
# Train a neural network model
# create a trainer
trainer = Trainer(model, lossFunction, optimizer, device, logLevel=logging.INFO)
# train the model
trainer.train(trainLoader, valLoader, numberOfEpochs)

# Test a neural network model
# create a tester
tester = Tester(model, device, logLevel=logging.INFO)
# test the model
tester.test(testLoader)
```

Error and accuracy metrics are available after training / testing via `trainer.metrics` and `tester.metrics`.

```
import sys
sys.path.append(packagePath)
print("If the import does not work, most likely your 'packagePath' is not set correctly!")
print(f'packagePath: {packagePath}')

from torchHelpers import Trainer, Tester

help(Trainer)
```

```
If the import does not work, most likely your 'packagePath' is not set correctly!
packagePath: /content/drive/My Drive/MachineVis2/template
Help on class Trainer in module torchHelpers:

class Trainer(builtins.object)
 |  Trainer(model, lossFunction, optimizer, device, logLevel=20)
 |
 |  Trainer class for training a neural network model in PyTorch.
 |
 |
 |  Parameters
 |  ----------
 |
 |  model : The neural network model to train.
 |
 |  lossFunction : The loss function to use for training.
 |
 |  optimizer : The optimizer to use for training.
 |
 |  device : The device to use for training. Can be either 'cpu', 'mps' or 'cuda'.
 |
 |  logLevel :  The log level to use for logging. Can be one of the following:
 |      logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL
 |
 |  Usage
 |  -----
 |
 |  # create a trainer
 |  trainer = Trainer(model, lossFunction, optimizer, device, logLevel=logging.INFO)
 |  # train the model
 |  trainer.train(trainLoader, valLoader, numberOfEpochs)
 |
 |  Author
 |  ------
 |  Markus Enzweiler (markus.enzweiler@hs-esslingen.de)
 |
 |  Methods defined here:
 |
 |  __init__(self, model, lossFunction, optimizer, device, logLevel=20)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  train(self, trainLoader, valLoader, numEpochs)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Exercise 2 - Polynomial regression (10 points)

This exercise involves fitting a polynomial model to given data using gradient descent. It is similar to the `linear_regression` example from https://github.com/menzHSE/cv-ml-lecture-notebooks that we have discussed in the lecture, but with a polynomial model instead of the linear model.

The polynomial model is given by:

$$y = f(x) = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$$

Your task is to estimate a, b, c and d using gradient descent using mean squared error loss between the predictions $\hat{y}_i$ of our model and true values y_i:

$$L(a,b,c,d)\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - (a \cdot x_i^3 + b \cdot x_i^2 + c \cdot x_i + d)\right)^2$$

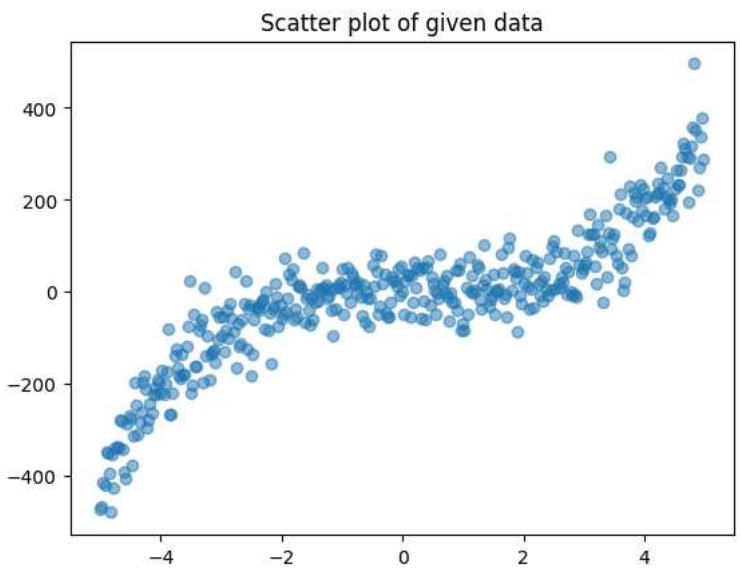
Tasks

- Derive the gradient descent update rules to estimate a, b, c, and d
- Implement gradient descent to estimate a, b, c, and d using your derived update rules

Load and visualize the data

```
data_path = os.path.join(packagePath, "data", "poly.csv")

# load data from csv
data = np.loadtxt(data_path, delimiter=",")
X = torch.Tensor(data[:, 0])
Y = torch.Tensor(data[:, 1])
# Visualize
plt.scatter(X, Y, alpha=0.5)
plt.title("Scatter plot of given data")
plt.show()
```



Initialize parameters

```
a = torch.randn(1, dtype=torch.float32)
b = torch.randn(1, dtype=torch.float32)
c = torch.randn(1, dtype=torch.float32)
d = torch.randn(1, dtype=torch.float32)

print(f"Initial params : {a,b,c,d}")
```

```
Initial params : (tensor([1.0805]), tensor([1.4996]), tensor([-0.1652]), tensor([-0.7082]))
```

Estimate a, b, c, d using gradient descent

Use the following hyperparameters:

- 100000 iterations of gradient descent
- learning rate 1e-5

```
def poly_model(a, b, c, d, x):
  return a * x**3 + b * x**2 + c * x + d
def loss_msl(y_pred, y):
  return torch.mean(torch.square(y-y_pred))
def loss_a(x,y,a,b,c,d):
  return torch.mean(-2 * (y - poly_model(a,b,c,d,x))*x**3)
def loss_b(x,y,a,b,c,d):
  return torch.mean(-2 * (y - poly_model(a,b,c,d,x))*x**2)
def loss_c(x,y,a,b,c,d):
  return torch.mean(-2 * (y - poly_model(a,b,c,d,x))*x)
def loss_d(x,y,a,b,c,d):
  return torch.mean(-2 * (y - poly_model(a,b,c,d,x)))


#Hyperparameters

num_iteration = 10000
learning_rate = 1e-5

for iteration in range (num_iteration):
  y_pred = poly_model(a,b,c,d,X)

  loss = loss_msl(y_pred, Y)

  grad_a = loss_a(X,Y,a,b,c,d)
  grad_b = loss_b(X,Y,a,b,c,d)
  grad_c = loss_c(X,Y,a,b,c,d)
  grad_d = loss_d(X,Y,a,b,c,d)

  with torch.no_grad():
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

  if iteration % 500 == 0 or iteration == 0:
    print(f"Iteration {iteration} | Loss {loss.item()} | a {a.item()} | b{b.item()} | c {c.item()} | d {d.item()}")

print(f"Final| a {a.item()} | b{b.item()} | c {c.item()} | d {d.item()}")
```

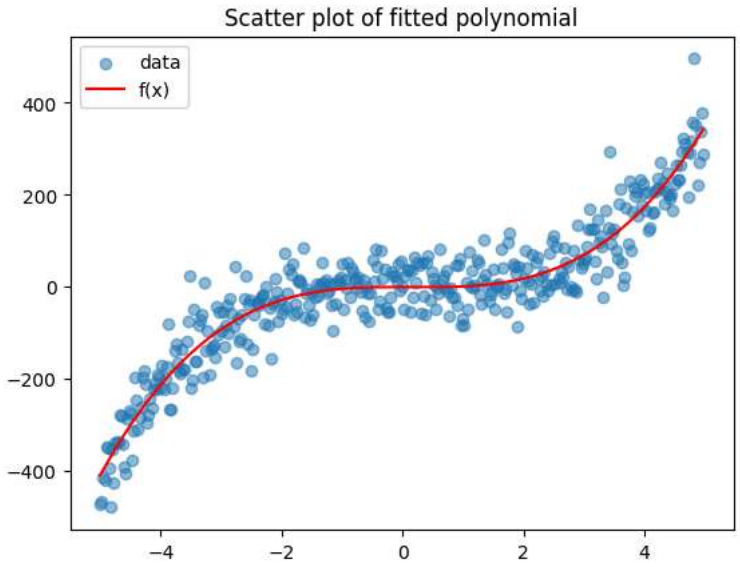
```
Iteration 0 | Loss 11872.83203125 | a 1.1680244207382202 | b1.4925888776779175 | c -0.1603938490152359 | d -0.7086163759231567
Iteration 500 | Loss 2463.67919921875 | a 3.0289556980133057 | b-0.42868828773498535 | c -0.08508336544036865 | d -0.81432640552520
Iteration 1000 | Loss 2396.03369140625 | a 3.0284597873687744 | b-0.9744368195533752 | c -0.11232896149158478 | d -0.82827919721603
Iteration 1500 | Loss 2390.33837890625 | a 3.0293636322021484 | b-1.130768060684204 | c -0.13875004649162292 | d -0.8162769079208374
Iteration 2000 | Loss 2389.66259765625 | a 3.030648946762085 | b-1.1762936115264893 | c -0.16468918323516846 | d -0.7969613075256348
Iteration 2500 | Loss 2389.3974609375 | a 3.032034158706665 | b-1.1902873516082764 | c -0.19024717807769775 | d -0.7756356000900269
Iteration 3000 | Loss 2389.16943359375 | a 3.0334312915802 | b-1.1953057050704956 | c -0.21545696258544922 | d -0.7538080811500549
Iteration 3500 | Loss 2388.9482421875 | a 3.034818649291992 | b-1.1977653503417969 | c -0.24032968282699585 | d -0.7319062948226929
Iteration 4000 | Loss 2388.73095703125 | a 3.0361897945404053 | b-1.1994935274124146 | c -0.2648715078830719 | d -0.710056424140930
Iteration 4500 | Loss 2388.51806640625 | a 3.037543773651123 | b-1.2010068893432617 | c -0.2890875041484833 | d -0.6882895827293396
Iteration 5000 | Loss 2388.309326171875 | a 3.038879871368408 | b-1.202445149421692 | c -0.3129819333553314 | d -0.6666155457496643
Iteration 5500 | Loss 2388.10400390625 | a 3.04019832611084 | b-1.2038756608963013 | c -0.3365591764450073 | d -0.6450376510620117
Iteration 6000 | Loss 2387.902587890625 | a 3.041504144668579 | b-1.2053061723709106 | c -0.3598272502422333 | d -0.6235554218292236
Iteration 6500 | Loss 2387.705078125 | a 3.042788028717041 | b-1.2067108154296875 | c -0.3827890157699585 | d -0.6021679639816284
Iteration 7000 | Loss 2387.510986328125 | a 3.0440549850463867 | b-1.208107590675354 | c -0.40544596314430237 | d -0.580875754356384
Iteration 7500 | Loss 2387.320556640625 | a 3.045305013656616 | b-1.2094985246658325 | c -0.427802175283432 | d -0.5596783757209778
Iteration 8000 | Loss 2387.133544921875 | a 3.0465383529663086 | b-1.210883617401123 | c -0.44986164569854736 | d -0.538575530052185
Iteration 8500 | Loss 2386.94970703125 | a 3.047755479812622 | b-1.2122623920440674 | c -0.4716280996799469 | d -0.5175681114196777
Iteration 9000 | Loss 2386.769287109375 | a 3.0489563941955566 | b-1.2136355638504028 | c -0.49310576915740967 | d -0.4966541528701
Iteration 9500 | Loss 2386.592041015625 | a 3.0501463413238525 | b-1.2150064706802368 | c -0.5143011808395386 | d -0.475833535194397
Final| a 3.0513134002685547 | b-1.2163746356964111 | c -0.5351768136024475 | d -0.45514658093452454
```

Visualize the polynomial fit

```
# Visualize
plt.scatter(X, Y, alpha=0.5)
plt.title("Scatter plot of fitted polynomial")

# Plot the recovered line
Y_model = poly_model(a, b, c, d, X)
plt.plot(X.tolist(), Y_model.tolist(), color="red")

plt.legend(["data", f"f(x)"])
plt.show()
```



Exercise 3 - Traffic Sign Classification using Multilayer Perceptrons in PyTorch (10 points)

In this exercise you will train a multilayer perception neural network using PyTorch on the German Traffic Sign Recognition Benchmark dataset. There will be no previous feature transform, i.e. the raw pixel values are the input to the neural network.

Automatically select the best available device (**PROVIDED**)

In Colab: Make sure that "GPU" is selected in Runtime -> Change runtime type

You should have a GPU device available, e.g.:

```
Using device: cuda
Tesla T4
```

We will transfer our model and data to this device later on in the Assignment. PyTorch takes care of all the particular device handling automatically, i.e. we will not have to explicitly deal with CUDA / MPS.

```
# Check the devices that we have available and prefer CUDA over MPS and CPU
def autoselectDevice(verbose=1):

    # default: CPU
    device = torch.device('cpu')

    if torch.cuda.is_available():
        # CUDA
        device = torch.device('cuda')
    elif torch.backends.mps.is_available() and torch.backends.mps.is_built():
        # MPS (acceleration on Apple M-series SoC)
        device = torch.device('mps')

    if verbose:
        print('Using device:', device)

    # Additional Info when using cuda
    if verbose and device.type == 'cuda':
        print(torch.cuda.get_device_name(0))

    return device

# We transfer our model and data later to this device. If this is a GPU
# PyTorch will take care of everything automatically.
device = autoselectDevice(verbose=1)

```

```
Using device: cuda
Tesla T4
```

Getting familiar with the GTSRB dataset (**PROVIDED**)

```
# GTSRB is available as standard dataset in PyTorch. Nice :)

# Data is loaded and processed in batches of 'batchSize' images
batchSize = 24

# We can add a chain of transforms that is applied to the original data, e.g.
#    resize all images to the same dimensions, e.g. 64x64 pixels
#    convert (batch of) images to a tensor
#    normalize pixel values (to 0-1)

transform = transforms.Compose(
    [transforms.Resize((64, 64)), # resize to 64x64 pixels
     transforms.ToTensor()        # convert to tensor. This will also normalize pixels to 0-1
     ])

# We construct several DataLoaders that take care of loading, storing, caching, pre-fetching the dataset.
# We will have one DataLoader for training, validation and test data.

# Training data
trainSet = torchvision.datasets.GTSRB(root='./data', split='train',
                                      download=True, transform=transform)
trainLoader = torch.utils.data.DataLoader(trainSet, batch_size=batchSize,
                                          shuffle=True, pin_memory=True, num_workers=2)
numTrainSamples = len(trainSet)

# Validation and test data
# GTSRB only provides a single test set. To create a validation and test set,
# we split the original GTSRB test set into two parts. The validation set is
# used to tune performance during training. The test set is only used AFTER
# training to evaluation the final performance.

gtsrbTestSet = torchvision.datasets.GTSRB(root='./data', split='test',
                                          download=True, transform=transform)

# Split the original GTSRB test data into 75% used for validation and 25% used for testing
# We do not need to shuffle the data, as we are processing every validation / test image exactly once
length75Percent = int(0.75 * len(gtsrbTestSet))
length25Percent = len(gtsrbTestSet) - length75Percent
lengths = [length75Percent, length25Percent]
valSet, testSet = torch.utils.data.random_split(gtsrbTestSet, lengths)
valLoader = torch.utils.data.DataLoader(valSet, batch_size=batchSize,
                                        shuffle=False, pin_memory=True, num_workers=2)
numValSamples = len(valSet)

testLoader = torch.utils.data.DataLoader(testSet, batch_size=batchSize,
                                         shuffle=False, pin_memory=True, num_workers=2)
numTestSamples = len(testSet)

# Available traffic sign classes in the dataset
classes = [
    "Speed limit (20km/h)",
    "Speed limit (30km/h)",
    "Speed limit (50km/h)",
    "Speed limit (60km/h)",
    "Speed limit (70km/h)",
    "Speed limit (80km/h)",
    "End of speed limit (80km/h)",
    "Speed limit (100km/h)",
    "Speed limit (120km/h)",
    "No passing",
    "No passing for vehicles over 3.5 metric tons",
    "Right-of-way at the next intersection",
    "Priority road",
    "Yield",
    "Stop",
    "No vehicles",
    "Vehicles over 3.5 metric tons prohibited",
    "No entry",
    "General caution",
    "Dangerous curve to the left",
    "Dangerous curve to the right",
    "Double curve",
    "Bumpy road",
    "Slippery road",
    "Road narrows on the right",
    "Road work",
    "Traffic signals",
    "Pedestrians",
    "Children crossing",
    "Bicycles crossing",
    "Beware of ice/snow",
    "Wild animals crossing",
```

```
    "End of all speed and passing limits",
    "Turn right ahead",
    "Turn left ahead",
    "Ahead only",
    "Go straight or right",
    "Go straight or left",
    "Keep right",
    "Keep left",
    "Roundabout mandatory",
    "End of no passing",
    "End of no passing by vehicles over 3.5 metric tons",
]

numClasses = len(classes)
```

```
Downloading https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB-Training_fixed.zip to data/gtsrb/GTSRB-Train
100%|██████████| 187490228/187490228 [00:11<00:00, 16836512.71it/s]
Extracting data/gtsrb/GTSRB-Training_fixed.zip to data/gtsrb
Downloading https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Test_Images.zip to data/gtsrb/GTSRB_Fi
100%|██████████| 88978620/88978620 [00:06<00:00, 13669099.65it/s]
Extracting data/gtsrb/GTSRB_Final_Test_Images.zip to data/gtsrb
Downloading https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Test_GT.zip to data/gtsrb/GTSRB_Final_
100%|██████████| 99620/99620 [00:00<00:00, 208536.04it/s]
Extracting data/gtsrb/GTSRB_Final_Test_GT.zip to data/gtsrb
```

Print dataset statistics (**PROVIDED**)

```
print("Dataset Statistics")
print(f"  # of training samples:   {numTrainSamples}")
print(f"  # of validation samples: {numValSamples}")
print(f"  # of test samples:       {numTestSamples}")
print(f"  # of different classes:  {numClasses}")
```

```
Dataset Statistics
  # of training samples:   26640
  # of validation samples: 9472
  # of test samples:       3158
  # of different classes:  43
```

Visualize the data (**PROVIDED**)

```
# Visualize a random batch of data from the data set

def imshow(img):
    npimg = img.cpu().numpy() # make sure image is in host memory

    # normalize to 0-1 for visualization
    minPixel = np.min(npimg)
    maxPixel = np.max(npimg)
    npimg = npimg - minPixel
    npimg = npimg / (maxPixel-minPixel)

    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis("off")
    plt.show()


numRows = 8

# get a single random batch of training images
dataIter = iter(trainLoader)
images, labels = next(dataIter)

# print labels
for i in range( batchSize // numRows ):
    print('\n'.join(f'Image {j:2d}: {classes[labels[j]]:5s}' for j in range((i*numRows), (i*numRows)+numRows)))

# show images
imshow(torchvision.utils.make_grid(images, nrow=numRows))
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
Image  0: Road work
Image  1: Speed limit (50km/h)
Image  2: End of speed limit (80km/h)
Image  3: Right-of-way at the next intersection
Image  4: Children crossing
Image  5: Keep right
Image  6: Ahead only
Image  7: End of all speed and passing limits
Image  8: Pedestrians
Image  9: Speed limit (70km/h)
Image 10: Right-of-way at the next intersection
Image 11: Speed limit (80km/h)
Image 12: No passing for vehicles over 3.5 metric tons
Image 13: Speed limit (50km/h)
Image 14: Turn left ahead
Image 15: Keep left
Image 16: No entry
Image 17: Speed limit (70km/h)
Image 18: Speed limit (120km/h)
Image 19: Road work
Image 20: Speed limit (30km/h)
Image 21: Road work
Image 22: Speed limit (120km/h)
Image 23: Speed limit (100km/h)
```

Neural Network Model Definition (**add your code here**)

We want to design a standard "feed-forward" multilayer perceptron as seen in the tutorial. In PyTorch terms, this is a Python class that subclasses torch.nn.Module and contains `__init__()` and `forward()`. Note that you do not have to use `nn.Sequential` as seen in the tutorial. This only groups layers together but is not necessary. You can also treat each layer separately, e.g. as shown in this tutorial.

We will need the following layers (input to output):

- The input will be the raw pixel values, i.e. 12288 values (images of 64x64x3 flattened into a single vector)
- 4 `torch.nn.Linear` layers with 512, 256, 128, 64 neurons each
- The output layer is a `torch.nn.Linear` layer with `numClasses` neurons, one neuron per class
- All neurons except for neurons in the output layer should have `torch.nn.functional.leaky_relu` activation functions
- **Important: The output layer must not have any activation function. It will be automatically applied in the loss computation (softmax activation for CrossEntropy loss, as seen in the lecture).**

```
## Define the layers of the MLP network model

###### YOUR CODE GOES HERE ######
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
      super().__init__()
      self.flatten = nn.Flatten()
      self.fc1 = nn.Linear(64*64*3, 512)
      self.fc2 = nn.Linear(512, 256)
      self.fc3 = nn.Linear(256, 128)
      self.fc4 = nn.Linear(128, 64)
      self.fc5 = nn.Linear(64, numClasses)

    def forward(self, x):
      x = self.flatten(x)
      x = F.leaky_relu(self.fc1(x))
      x = F.leaky_relu(self.fc2(x))
      x = F.leaky_relu(self.fc3(x))
      x = F.leaky_relu(self.fc4(x))
      x = self.fc5(x)
      return x
##################################

```

Print a summary of the structure of our network using the `torchinfo` package. (**PROVIDED**)

The result should look similar to:

```
==========================================================================================
Layer (type (var_name))                  Output Shape              Param #
==========================================================================================
Net (Net)                                [1, 43]                   --
├─Linear (fc1)                           [1, 512]                  6,291,968
├─Linear (fc2)                           [1, 256]                  131,328
├─Linear (fc3)                           [1, 128]                  32,896
├─Linear (fc4)                           [1, 64]                   8,256
├─Linear (fc5)                           [1, 43]                   2,795
==========================================================================================
Total params: 6,467,243
Trainable params: 6,467,243
Non-trainable params: 0
Total mult-adds (M): 6.47
==========================================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 0.01
Params size (MB): 25.87
Estimated Total Size (MB): 25.93
==========================================================================================
```

```
# Instatiate our neural network
network = Net()

# Print a summary of the net
%pip install torchinfo

from torchinfo import summary
summary(network, input_size=(1, 3, 64, 64), row_settings=["var_names"])
```

```
Requirement already satisfied: torchinfo in /usr/local/lib/python3.10/dist-packages (1.8.0)
==========================================================================================
Layer (type (var_name))                  Output Shape              Param #
==========================================================================================
Net (Net)                                [1, 43]                   --
├─Flatten (flatten)                      [1, 12288]                --
├─Linear (fc1)                           [1, 512]                  6,291,968
├─Linear (fc2)                           [1, 256]                  131,328
├─Linear (fc3)                           [1, 128]                  32,896
├─Linear (fc4)                           [1, 64]                   8,256
├─Linear (fc5)                           [1, 43]                   2,795
==========================================================================================
Total params: 6,467,243
Trainable params: 6,467,243
```

```
Non-trainable params: 0
Total mult-adds (M): 6.47
==========================================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 0.01
Params size (MB): 25.87
Estimated Total Size (MB): 25.93
==========================================================================================
```

Question

Why is the number of learnable parameters of the first hidden layer `6,291,968`?

Your Answer:

Neural Network Training (**add your code here**)

To train our network, we will first have to define a loss function, an optimizer and hyperparameters that control the training process:

- `torch.optim.AdamW` is used as an optimizer with default parameters except for the learning rate which is set to `lr=3e-4`.
- `torch.nn.CrossEntropyLoss` is used as loss function. Note, that the softmax activation is applied during loss computation, as stated in the documentation
- The number of training epochs is 15

Train your multilayer perceptron network using the `Trainer` class and provide `trainLoader` as the DataLoader for training data and `valLoader` as the DataLoader for validation data.

The overall training should take about 20 seconds per epoch (**on a GPU**, depending on what GPU is assigned). Reported accuracies on the training (validation) data should be approx. 95% (81%) after 15 training epochs.

```
##### YOUR CODE GOES HERE ######
lr = 3e-4
model = network
param_optimizer = model.parameters()
lossFunction = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(param_optimizer, lr=lr)

trainer = Trainer(model, lossFunction, optimizer, device)
trainer.train(trainLoader, valLoader, 15)
################################
```

```
[Epoch   0] :  /usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was called. os.fork() is incompatible
  self.pid = os.fork()
...../usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was called. os.fork() is incompatible wi
  self.pid = os.fork()
 done (1110 batches)
[Epoch   0] : | time: 22.640s | trainLoss: 2.182  | trainAccuracy: 0.393  | valLoss: 1.745  | valAccuracy: 0.503 | Throughput: 9287
[Epoch   1] :  ..... done (1110 batches)
[Epoch   1] : | time: 18.498s | trainLoss: 0.971  | trainAccuracy: 0.716  | valLoss: 1.217  | valAccuracy: 0.707 | Throughput: 9454
[Epoch   2] :  ..... done (1110 batches)
[Epoch   2] : | time: 21.095s | trainLoss: 0.676  | trainAccuracy: 0.807  | valLoss: 1.386  | valAccuracy: 0.679 | Throughput: 8347
[Epoch   3] :  ..... done (1110 batches)
[Epoch   3] : | time: 18.408s | trainLoss: 0.537  | trainAccuracy: 0.845  | valLoss: 1.144  | valAccuracy: 0.739 | Throughput: 9103
[Epoch   4] :  ..... done (1110 batches)
[Epoch   4] : | time: 18.387s | trainLoss: 0.432  | trainAccuracy: 0.875  | valLoss: 1.370  | valAccuracy: 0.701 | Throughput: 9333
[Epoch   5] :  ..... done (1110 batches)
[Epoch   5] : | time: 19.287s | trainLoss: 0.379  | trainAccuracy: 0.888  | valLoss: 1.122  | valAccuracy: 0.778 | Throughput: 8880
[Epoch   6] :  ..... done (1110 batches)
[Epoch   6] : | time: 19.994s | trainLoss: 0.337  | trainAccuracy: 0.899  | valLoss: 1.051  | valAccuracy: 0.795 | Throughput: 9177
[Epoch   7] :  ..... done (1110 batches)
[Epoch   7] : | time: 18.020s | trainLoss: 0.286  | trainAccuracy: 0.915  | valLoss: 1.159  | valAccuracy: 0.800 | Throughput: 9114
[Epoch   8] :  ..... done (1110 batches)
[Epoch   8] : | time: 18.374s | trainLoss: 0.254  | trainAccuracy: 0.924  | valLoss: 1.125  | valAccuracy: 0.818 | Throughput: 9261
[Epoch   9] :  ..... done (1110 batches)
[Epoch   9] : | time: 18.450s | trainLoss: 0.239  | trainAccuracy: 0.930  | valLoss: 1.348  | valAccuracy: 0.745 | Throughput: 9106
[Epoch  10] :  ..... done (1110 batches)
[Epoch  10] : | time: 19.112s | trainLoss: 0.224  | trainAccuracy: 0.933  | valLoss: 1.115  | valAccuracy: 0.814 | Throughput: 8823
[Epoch  11] :  ..... done (1110 batches)
[Epoch  11] : | time: 20.097s | trainLoss: 0.181  | trainAccuracy: 0.946  | valLoss: 1.234  | valAccuracy: 0.798 | Throughput: 9051
[Epoch  12] :  ..... done (1110 batches)
[Epoch  12] : | time: 20.916s | trainLoss: 0.190  | trainAccuracy: 0.942  | valLoss: 1.277  | valAccuracy: 0.800 | Throughput: 9063
[Epoch  13] :  ..... done (1110 batches)
[Epoch  13] : | time: 18.404s | trainLoss: 0.163  | trainAccuracy: 0.953  | valLoss: 1.295  | valAccuracy: 0.815 | Throughput: 9376
[Epoch  14] :  ..... done (1110 batches)
[Epoch  14] : | time: 18.398s | trainLoss: 0.160  | trainAccuracy: 0.953  | valLoss: 1.405  | valAccuracy: 0.799 | Throughput: 9478
Training finished in 0:04:50.302472 hh:mm:ss.ms
```

Visualize the behavior of the loss and accuracy (**add your code here**)

Using the data available in `trainer.metrics`, create the following two plots:

- Training loss and validation loss as a function of epochs.
- Training accuracy and validation accuracy as a function of epochs.

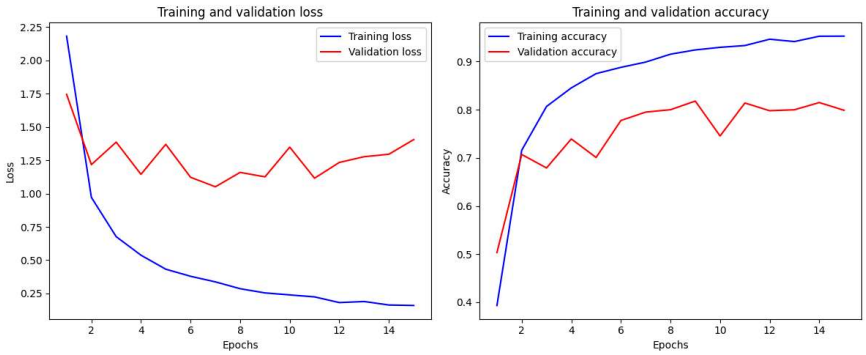
```
##### YOUR CODE GOES HERE ######
trainLoss = trainer.metrics["epochTrainLoss"]
valLoss = trainer.metrics["epochValLoss"]
trainacc = trainer.metrics["epochTrainAccuracy"]
trainValacc = trainer.metrics["epochValAccuracy"]

# Number of epochs
epochs = range(1, len(trainLoss) + 1)

# Plotting training and validation loss
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs, trainLoss, 'b', label='Training loss')
plt.plot(epochs, valLoss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plotting training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(epochs, trainacc, 'b', label='Training accuracy')
plt.plot(epochs, trainValacc, 'r', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
################################
```



Run your network on some images to get predictions (**PROVIDED**)

```
# Run some batches of unseen test data through the network and visualize its predictions
numBatches = 4
numRows = 8

 # Iterator through the test DataLoader
dataIter = iter(testLoader)

# for each batch
for batch in range(numBatches):

    # get images and ground truth labels
    images, labels = next(dataIter)

    # push to the device used
    images, labels = images.to(device), labels.to(device)

    # forward pass of the batch of images
    outputs = network(images)

    # find the index of the class with the largest output
    _, predictedLabels         = torch.max(outputs, 1)

    # print labels and outputs
    countCorrect = 0
    for i in range( batchSize // numRows ):
        for j in range((i*numRows), (i*numRows)+numRows):
            print(f'Image {j:2d} - Label: {classes[labels[j]]:5s} | Prediction: {classes[predictedLabels[j]]:5s}')
            if labels[j] == predictedLabels[j]:
                countCorrect = countCorrect + 1

    print(f"\n{(countCorrect / batchSize) * 100.0:.2f}% of test images correctly classified")

    # show images
    imshow(torchvision.utils.make_grid(images))
```

```
Image  7 - Label: Road work | Prediction: Speed limit (50km/h)
Image  8 - Label: Road work | Prediction: Wild animals crossing
Image  9 - Label: Keep right | Prediction: Keep right
Image 10 - Label: Keep right | Prediction: Keep right
Image 11 - Label: No passing | Prediction: No passing
Image 12 - Label: No entry | Prediction: No entry
Image 13 - Label: No passing | Prediction: No passing
Image 14 - Label: Speed limit (100km/h) | Prediction: Speed limit (100km/h)
Image 15 - Label: Right-of-way at the next intersection | Prediction: Right-of-way
Image 16 - Label: Bumpy road | Prediction: Bumpy road
Image 17 - Label: Speed limit (50km/h) | Prediction: Speed limit (50km/h)
Image 18 - Label: Speed limit (60km/h) | Prediction: Speed limit (60km/h)
Image 19 - Label: End of no passing by vehicles over 3.5 metric tons | Prediction:
Image 20 - Label: Right-of-way at the next intersection | Prediction: Right-of-way
Image 21 - Label: Wild animals crossing | Prediction: Wild animals crossing
Image 22 - Label: Speed limit (120km/h) | Prediction: Speed limit (120km/h)
Image 23 - Label: Double curve | Prediction: Dangerous curve to the left

83.33% of test images correctly classified
```

```
Image  0 - Label: Keep right | Prediction: Keep right
Image  1 - Label: Keep right | Prediction: Go straight or right
Image  2 - Label: Keep right | Prediction: Go straight or right
Image  3 - Label: Speed limit (50km/h) | Prediction: Speed limit (50km/h)
Image  4 - Label: No passing | Prediction: No passing
Image  5 - Label: Traffic signals | Prediction: Traffic signals
Image  6 - Label: Beware of ice/snow | Prediction: Beware of ice/snow
Image  7 - Label: Right-of-way at the next intersection | Prediction: Bicycles cros
Image  8 - Label: No passing | Prediction: Speed limit (60km/h)
Image  9 - Label: End of no passing by vehicles over 3.5 metric tons | Prediction:
Image 10 - Label: Keep right | Prediction: Go straight or right
Image 11 - Label: End of speed limit (80km/h) | Prediction: End of no passing
Image 12 - Label: Wild animals crossing | Prediction: Wild animals crossing
Image 13 - Label: Speed limit (100km/h) | Prediction: Speed limit (100km/h)
Image 14 - Label: General caution | Prediction: General caution
Image 15 - Label: Right-of-way at the next intersection | Prediction: Right-of-way
Image 16 - Label: Traffic signals | Prediction: Traffic signals
Image 17 - Label: Keep right | Prediction: Keep right
Image 18 - Label: No passing | Prediction: No passing for vehicles over 3.5 metric
Image 19 - Label: Speed limit (80km/h) | Prediction: Speed limit (80km/h)
Image 20 - Label: Speed limit (30km/h) | Prediction: Speed limit (30km/h)
Image 21 - Label: Vehicles over 3.5 metric tons prohibited | Prediction: Speed limi
Image 22 - Label: Road work | Prediction: Slippery road
Image 23 - Label: Road work | Prediction: Road work

62.50% of test images correctly classified
```

```
Image  0 - Label: Speed limit (50km/h) | Prediction: Speed limit (30km/h)
Image  1 - Label: General caution | Prediction: Right-of-way at the next intersecti
Image  2 - Label: Priority road | Prediction: Priority road
Image  3 - Label: Vehicles over 3.5 metric tons prohibited | Prediction: Vehicles o
Image  4 - Label: No entry | Prediction: No entry
Image  5 - Label: Speed limit (30km/h) | Prediction: Speed limit (30km/h)
Image  6 - Label: No passing | Prediction: No passing
Image  7 - Label: Keep right | Prediction: Keep right
Image  8 - Label: Keep right | Prediction: Keep right
```