CS440/ECE448 Spring 2024

MP05: Neural Networks

This tutorial and its materials are based on and adapted from the PyTorch Official Tutorial and mrdbourke/pytorch-deep-learning, revised to meet the requirements of this assignment.

We encourage you to explore the original tutorials for additional topics not covered in this assignment.

This notebook will walk you through the whole MP, giving you instructions and debugging tips as you go.

If you are already familiar with PyTorch, you can jump to the **implement this** sections.

Goal

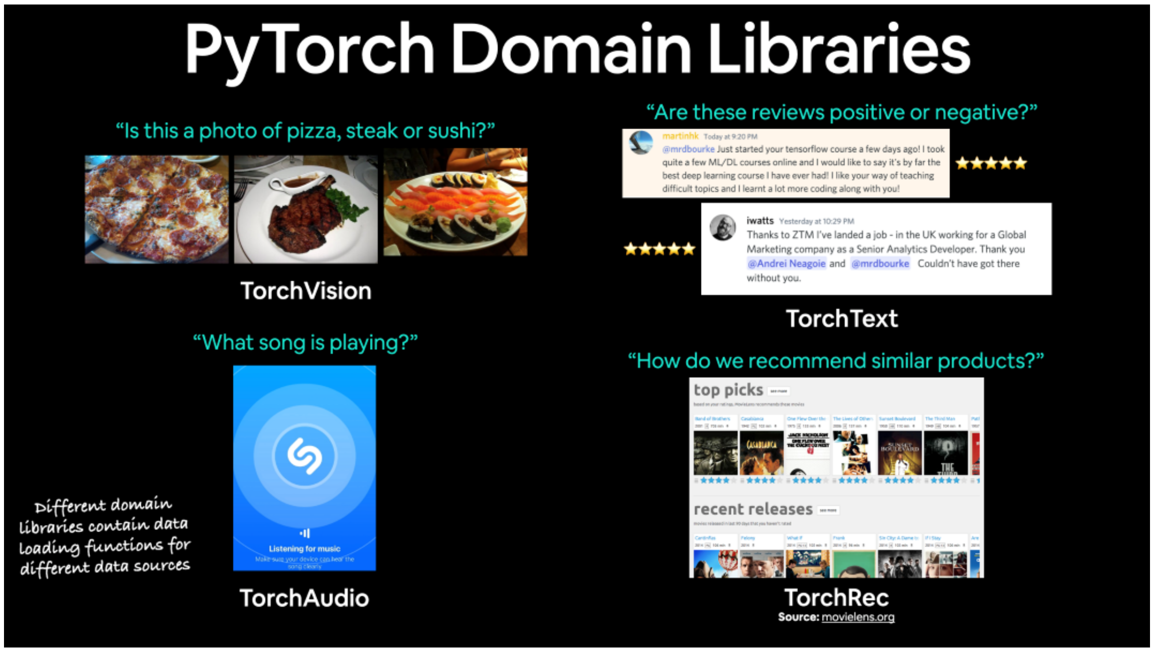
The goal of this assignment is to employ neural networks, nonlinear and multi-layer extensions of the linear perceptron, to classify images into `5 categories`:

- ship (0)
- automobile (1)
- dog (2)
- frog (3)
- horse (4)

Table of Contents

What is *PyTorch*?

PyTorch is an open source machine learning framework that accelerates the path from research prototyping to production deployment.
PyTorch allows you to manipulate and process data and write machine learning algorithms using Python code (user-friendly!).
PyTorch also offers some domain-specific libraries such as TorchText, TorchVision, and TorchAudio.


PyTorch Domain Libraries
"Is this a photo of pizza, steak or sushi?"
TorchVision
"Are these reviews positive or negative?"
TorchText
"What song is playing?"
TorchAudio
"How do we recommend similar products?"
top picks
recent releases
TorchRec
Source: movielens.org
Different domain libraries contain data loading functions for different data sources

You can install *PyTorch* with conda or pip.
The exact installation command varies based on your system setup and desired *PyTorch* version. For example, to install *PyTorch* with pip, you can use the following command:

```
pip install torch
```

For this assignment, **GPU support is not required** since the autograder doesn't have any GPUs. Therefore, you can install *PyTorch* without CUDA support. However, learning how to use *PyTorch* with CUDA might be beneficial for your future projects.

Let's verify the installation by printing *PyTorch* version. The code block below should run without error if *PyTorch* was installed correctly.

In [1]:
```
import torch    # the library name is torch
print("PyTorch version:", torch.__version__)
```

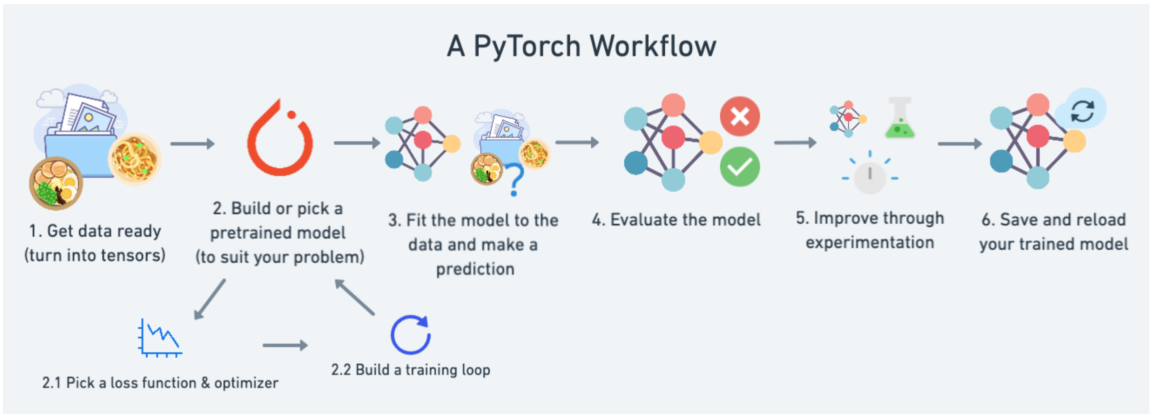
```
PyTorch version: 1.13.1
```

PyTorch Workflow

Source: mrdbourke/pytorch-deep-learning

Machine learning is a game of two parts:

1. Transform your data, whether it's images, text, or any other form, into a numerical format (a representation).
2. Pick or build a model to learn the representation as best as possible.


A PyTorch Workflow
1. Get data ready (turn into tensors)
2. Build or pick a pretrained model (to suit your problem)
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentation
6. Save and reload your trained model
2.1 Pick a loss function & optimizer
2.2 Build a training loop

In this MP, you will mostly work on the second step: ***building a model***.

Datasets

The dataset consists of `3750` 31x31 colored (RGB) images (a modified subset of the CIFAR-10 dataset, provided by Alex Krizhevsky). This set is split for you into `2813` training examples and `937` test examples.

In this MP, data processing is taken care of for you. The **helper.py** is provided as utility to facilitate the data loading process. You can ignore the implementation details in **helper.py**.

To use **helper.py**, you need to have **numpy** and **matplotlib** installed.

The function `Load_dataset()` unpacks the dataset file (*you don't need to call this function in the submitted.py*), returning images and labels for the training and test sets. Note that the images have been flattened, therefore the dimension of one image sample is `2883` (31 x 31 x 3).

In [2]:
```
import helper

filepath = "./data/mp_data"
# load datasets, you don't need to call this function in the MP
train_set, train_labels, test_set, test_labels = helper.Load_dataset(filepat
print("Shape of train set:", train_set.shape)
print("Shape of test set:", test_set.shape)
```

```
Shape of train set: (2813, 2883)
Shape of test set: (937, 2883)
```

In [3]:
```
# Use the helper function to visualize training set.
# The third argument is the index of image to visualize.
# Feel free to change the index. Note that the size of training set is 2813
helper.show_train_image(train_set, train_labels, 2812)
```

In [4]:
```
# Use the helper function to visualize testing set.
# The third argument is the index of image to visualize.
# Feel free to change the index. Note that the size of testing size is 937
helper.show_test_image(test_set, test_labels, 20)
```

Dataloaders

Source: PyTorch Tutorial

Code for processing data samples can get messy and hard to maintain; we ideally want our data processing code to be decoupled from our model training code for better readability and modularity. PyTorch provides two data primitives: `torch.utils.data.DataLoader` and `torch.utils.data.Dataset` that allow you to use pre-loaded datasets as well as your own data. Dataset stores the samples and their corresponding labels, and DataLoader wraps an iterable around the Dataset to enable easy access to the samples.

A dataloader is a way for you to handle loading and transforming data before it enters your network for training or prediction. It lets you write code that looks like you're just looping through the dataset, with the division into batches happening automatically. In this MP, you **don't need to write the dataset and dataloader** part, we have provided one for you, but you need to understand how to use it.

For more information about datasets and dataloaders, please refer to **Source**.

In [5]:
```
# Use provided helper function to generate dataloaders
# You don't need to call these functions in submitted.py

# Preprocess the datasets
```

```
train_set_processed, test_set_processed = helper.Preprocess(train_set, test_

# Generate dataloaders
train_loader, test_loader = helper.Get_DataLoaders(
    train_set_processed,
    train_labels,
    test_set_processed,
    test_labels,
    batch_size=100,
)
```

Each iteration below returns a batch of `train_features` and `train_labels` (in this MP, we set up batch size equal to 100, so each batch contains 100 feature and label tensors respectively). You can pass the feature batch to your neural network, and then compare the label batch with your predictions. Let's iterate over the dataset and see what each batch looks like.

Labels:
`ship: 0, automobile: 1, dog: 2, frog: 3, horse: 4`

In [6]:
```
batch_index = 0

# Iterate over the first 3 batches
for features, labels in train_loader:
    print("Batch #", batch_index)
    print("Features shape:", features.shape)
    print(features, "\n")
    print("Labels shape:", labels.shape)
    print(labels, "\n\n")
    batch_index += 1
    if batch_index == 3:
        break
```

```
Batch # 0
Features shape: torch.Size([100, 2883])
tensor([[ 0.3525,  0.3150, -0.6970,  ...,  1.2380,  1.2221,  1.3589],
        [ 0.8684,  0.9081,  0.9093,  ...,  1.0154,  1.0167,  0.9519],
        [ 0.3932,  0.3839, -0.2262,  ..., -0.6230, -0.5951, -0.5977],
        ...,
        [-0.4756,  0.1494,  0.5631,  ..., -0.6230, -0.7689, -1.1299],
        [ 1.4793,  1.5012,  1.5324,  ...,  0.9199,  0.9219,  0.8737],
        [-0.5027, -0.5816, -0.5309,  ...,  1.0631,  1.0483,  0.9989]])

Labels shape: torch.Size([100])
tensor([0, 0, 0, 3, 1, 1, 0, 1, 4, 4, 4, 2, 1, 0, 3, 4, 1, 1, 3, 1, 1, 4, 1,
1,
        4, 4, 3, 2, 0, 4, 2, 1, 0, 1, 0, 3, 0, 1, 0, 4, 4, 0, 4, 0, 3, 2, 2,
3,
        4, 2, 3, 3, 1, 2, 3, 3, 2, 2, 2, 2, 1, 0, 4, 0, 3, 3, 0, 3, 1, 0, 0,
2,
        1, 3, 0, 0, 2, 4, 3, 2, 3, 2, 2, 3, 3, 2, 0, 3, 4, 0, 1, 0, 3, 1, 0,
0,
        4, 2, 0, 1])


Batch # 1
Features shape: torch.Size([100, 2883])
tensor([[-1.7381, -1.7541, -1.7771,  ..., -1.7523, -1.7487, -1.7403],
        [-0.3127, -0.3333, -0.3647,  ..., -0.2890, -0.3107, -0.3159],
        [-1.0594, -1.0368, -1.4863,  ..., -1.4024, -0.6899, -0.7229],
        ...,
        [ 0.3661,  0.4115,  0.4108,  ...,  0.6018,  0.4320,  0.4510],
        [ 1.7237,  1.7633,  1.7539,  ...,  2.3038,  2.2808,  2.2511],
        [-0.6249, -0.7609, -0.3232,  ..., -0.4003,  0.2898,  1.3276]])

Labels shape: torch.Size([100])
tensor([3, 2, 3, 1, 3, 3, 0, 0, 0, 4, 1, 2, 3, 4, 2, 4, 3, 3, 3, 2, 3, 4, 2,
3,
        4, 4, 2, 1, 4, 3, 3, 2, 2, 1, 2, 2, 0, 2, 3, 3, 2, 0, 0, 3, 1, 0, 4,
1,
        3, 4, 1, 0, 3, 2, 3, 4, 2, 2, 3, 1, 4, 2, 2, 2, 4, 3, 1, 3, 3, 1, 4,
4,
        1, 3, 2, 3, 2, 4, 2, 2, 2, 4, 4, 3, 4, 0, 0, 0, 1, 3, 4, 1, 1, 4, 3,
4,
        0, 0, 3, 3])


Batch # 2
Features shape: torch.Size([100, 2883])
tensor([[ 0.8412,  0.8943,  0.9093,  ...,  1.4766,  1.3643,  1.1084],
        [-0.8557, -0.1954, -0.1016,  ...,  1.1426,  1.1747,  1.1084],
        [ 0.1353,  0.1632,  0.4938,  ..., -0.0027,  0.0844, -0.0499],
        ...,
        [ 1.7237,  1.7081,  1.7262,  ..., -1.1956,  1.0641,  1.8911],
        [ 0.0403,  0.1494,  0.2584,  ..., -0.6389, -0.6741, -0.7542],
        [-0.9507, -1.0230, -1.0294,  ...,  0.7449,  0.2108, -0.1438]])

Labels shape: torch.Size([100])
tensor([2, 4, 4, 2, 1, 4, 2, 3, 0, 3, 3, 1, 2, 0, 4, 3, 4, 2, 2, 0, 0, 3, 4,
3,
        4, 3, 2, 4, 3, 3, 2, 0, 2, 0, 3, 4, 0, 4, 3, 1, 2, 2, 3, 4, 1, 4, 1,
2,
        4, 1, 3, 1, 4, 0, 4, 2, 4, 3, 4, 0, 1, 3, 1, 4, 2, 3, 1, 2, 2, 1, 0,
2,
        3, 2, 2, 4, 3, 1, 0, 2, 2, 2, 4, 3, 3, 1, 1, 0, 2, 2, 4, 3, 4, 1, 0,
3,
        3, 0, 0, 3])
```

Each batch contains features and labels. The features are tensors with shape `(batch_size, feature_size)`, and the labels are tensors with shape `(batch_size)`.

Tensors

Source: mrdbourke/pytorch-deep-learning

Tensors are a specialized data structure that are very similar to numpy arrays and matrices. Their job is to represent data in a numerical way. Tensors are similar to NumPy's arrays, except that tensors can run on GPUs or other hardware accelerators (better performance!). In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

The code cell below may give you some idea of how to use tensors.

In [7]:
```
rand_tensor = torch.rand(2, 3)             # create a random tensor of size
zeros_tensor = torch.zeros(5)              # create a tensor of size 5 that
print("Random Tensor: \n", rand_tensor, "\n")
print("Zeros Tensor: \n", zeros_tensor, "\n")

# explore some of the attributes of a Tensor
tensor = torch.tensor([[7, 7, 5], [1, 3, 0], [2, 2, 1], [9, 4, 8]])
print("My Tensor: \n", tensor)
print("Shape of tensor: ", tensor.shape)
print("Datatype of tensor: ", tensor.dtype)
print("Device tensor is stored on: ", tensor.device, "\n")

# element-wise multiplication
tensor1 = torch.tensor([1, 2, 3])
tensor2 = torch.tensor([2, 3, 4])
print("Element-wise multiplication:")
print(tensor1, "*", tensor2, "=", tensor1 * tensor2)
```

```
Random Tensor:
 tensor([[0.7706, 0.2286, 0.2136],
        [0.6712, 0.4824, 0.2182]])

Zeros Tensor:
 tensor([0., 0., 0., 0., 0.])

My Tensor:
 tensor([[7, 7, 5],
        [1, 3, 0],
        [2, 2, 1],
        [9, 4, 8]])
Shape of tensor:  torch.Size([4, 3])
Datatype of tensor:  torch.int64
Device tensor is stored on:  cpu

Element-wise multiplication:
tensor([1, 2, 3]) * tensor([2, 3, 4]) = tensor([ 2,  6, 12])
```

One of the most common errors you'll run into in deep learning is shape mismatches, because matrix multiplication has a strict rule about what shapes and sizes can be combined.

The code cell below is such an example.

In [8]:
```
# matrix multiplication
tensor_A = torch.tensor([[1, 2],
                         [3, 4],
                         [5, 6]], dtype=torch.float32)

tensor_B = torch.tensor([[7, 10],
                         [8, 11],
                         [9, 12]], dtype=torch.float32)
print("tensor_A, shape =", tensor_A.shape)
print(tensor_A, "\n")
print("tensor_B, shape =", tensor_B.shape)
print(tensor_B)

# torch.matmul() is a built-in matrix multiplication function
torch.matmul(tensor_A, tensor_B)    # this will error because of shape misma
```

```
tensor_A, shape = torch.Size([3, 2])
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])

tensor_B, shape = torch.Size([3, 2])
tensor([[ 7., 10.],
        [ 8., 11.],
        [ 9., 12.]])
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[8], line 15
     12 print(tensor_B)
     14 # torch.matmul() is a built-in matrix multiplication function
---> 15 torch.matmul(tensor_A, tensor_B)    # this will error because of sha
pe mismatch

RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x2 and 3x2)
```

In [9]:
```
# tensor_A and tensor_B cannot be multiplied
# However, multiplying tensor_A with the transpose of tensor_B is legal (3x2
# transpose of a tensor:    tensor.T - where tensor is the desired tensor to
print("tensor_A, shape =", tensor_A.shape)
print(tensor_A, "\n")
print("transpose of tensor_B, shape =", tensor_B.T.shape)
print(tensor_B.T, "\n")         # transpose of tensor_B
print("tensor_A * tensor_B.T equals")
print(torch.matmul(tensor_A, tensor_B.T))
```

```
tensor_A, shape = torch.Size([3, 2])
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])

transpose of tensor_B, shape = torch.Size([2, 3])
tensor([[ 7.,  8.,  9.],
        [10., 11., 12.]])

tensor_A * tensor_B.T equals
tensor([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])
```

Neural Net Layers

So far we have looked into the tensors, their properties and basic operations on tensors. These are especially useful to get familiar with if we are building the layers of our network from scratch. PyTorch also provides some built-in blocks in the torch.nn module.

We can use nn.Linear(in_features, out_features) to create a a linear layer that applies a linear transformation to the incoming data x:

- y = x A^T + b , where A and b are initialized **randomly**.

This will take an input of size (∗, ***in_features***) where ∗ means any number of dimensions including none and ***in_features*** is the size of each input sample. It will yield an output of size (∗, ***out_features***) where all but the last dimension are the same shape as the input and ***out_features*** is the size of each output sample.

In [10]:
```
# Create the input (8 samples, each of size 5)
input = torch.randn(8, 5)
print("Input:")
print(input)
print("Input size:", input.size(), "\n")

# Make a linear layer transforming (*, 5)-dimensinal inputs to (*, 7)-dimens
linear_layer = torch.nn.Linear(5, 7)

# Apply the linear layer
output = linear_layer(input)
print("Output:")
print(output)
print("Output size:", output.size())
```

```
Input:
tensor([[-2.1427, -0.7382,  0.3335,  0.3865, -0.6146],
        [-0.8864,  0.7096,  0.9242, -0.4327, -0.1272],
        [ 0.9784,  0.2195,  0.1283, -2.1508,  0.8675],
        [-0.2716,  0.4867, -0.6735,  2.4885, -0.1850],
        [-0.2660, -0.7077,  2.2901, -0.3677,  0.5883],
        [ 0.3426, -1.5650,  0.0481,  0.4113, -0.2566],
        [-0.4732, -0.3190, -0.8887,  0.5200, -0.9329],
        [-2.9853,  0.1357,  1.3753, -0.4170,  0.5080]])
Input size: torch.Size([8, 5])

Output:
tensor([[-0.2137, -0.0931, -0.0244, -1.2966,  0.0315, -1.2910,  0.6712],
        [-1.0045, -0.5326,  0.5715, -0.5316,  0.6377, -0.5839,  0.2541],
        [-0.6021,  0.0302,  0.0732, -0.1940,  0.4768,  0.5576,  0.6897],
        [ 0.9654, -0.8886, -0.0976, -0.1540, -0.9765, -1.1113, -1.1009],
        [-0.7647,  0.1497,  0.1468,  0.2737,  0.3389, -0.8903,  0.6927],
        [ 1.0874,  0.2011, -0.6012, -0.3632, -0.6953, -0.6893,  0.0779],
        [ 0.7485, -0.4204, -0.1624, -1.0676, -0.3794, -0.5958, -0.2804],
        [-1.7559, -0.0757,  0.4434, -1.0591,  0.7498, -1.3274,  1.4558]],
       grad_fn=<AddmmBackward0>)
Output size: torch.Size([8, 7])
```

We can also use the torch.nn module to apply activations functions to our tensors. Activation functions are used to add non-linearity to our network. Activation functions operate on each element seperately, so the shape of the tensors we get as an output are the same as the ones we pass in. Let's try nn.Sigmoid().

In [11]:
```
# Pass the output of previous layer to the current layer as the input
print("Previous output:")
print(output)
print("Output size:", output.size(), "\n")

# Create a sigmoid function
sigmoid = torch.nn.Sigmoid()

# Apply the activation function
activated_output = sigmoid(output)
print("Activated output:")
print(activated_output)
print("Activated output size:", activated_output.size())
```

```
Previous output:
tensor([[-0.2137, -0.0931, -0.0244, -1.2966,  0.0315, -1.2910,  0.6712],
        [-1.0045, -0.5326,  0.5715, -0.5316,  0.6377, -0.5839,  0.2541],
        [-0.6021,  0.0302,  0.0732, -0.1940,  0.4768,  0.5576,  0.6897],
        [ 0.9654, -0.8886, -0.0976, -0.1540, -0.9765, -1.1113, -1.1009],
        [-0.7647,  0.1497,  0.1468,  0.2737,  0.3389, -0.8903,  0.6927],
        [ 1.0874,  0.2011, -0.6012, -0.3632, -0.6953, -0.6893,  0.0779],
        [ 0.7485, -0.4204, -0.1624, -1.0676, -0.3794, -0.5958, -0.2804],
        [-1.7559, -0.0757,  0.4434, -1.0591,  0.7498, -1.3274,  1.4558]],
       grad_fn=<AddmmBackward0>)
Output size: torch.Size([8, 7])

Activated output:
tensor([[0.4468, 0.4768, 0.4939, 0.2147, 0.5079, 0.2157, 0.6618],
        [0.2680, 0.3699, 0.6391, 0.3701, 0.6542, 0.3580, 0.5632],
        [0.3539, 0.5075, 0.5183, 0.4517, 0.6170, 0.6359, 0.6659],
        [0.7242, 0.2914, 0.4756, 0.4616, 0.2736, 0.2476, 0.2496],
        [0.3176, 0.5374, 0.5366, 0.5680, 0.5839, 0.2911, 0.6666],
        [0.7479, 0.5501, 0.3541, 0.4102, 0.3328, 0.3342, 0.5195],
        [0.6788, 0.3964, 0.4595, 0.2559, 0.4063, 0.3553, 0.4304],
        [0.1473, 0.4811, 0.6091, 0.2575, 0.6791, 0.2096, 0.8109]],
       grad_fn=<SigmoidBackward0>)
Activated output size: torch.Size([8, 7])
```

So far we have seen that we can create layers and pass the output of one as the input of the next. Instead of creating intermediate tensors and passing them around, we can use nn.Sequentual, which does exactly that.

In [12]:
```
# Create a sequential container
block = torch.nn.Sequential(
    torch.nn.Linear(5, 7),
    torch.nn.Sigmoid()
)

print("Input:")
print(input)
print("Input size:", input.size(), "\n")

# Apply the block
block_output = block(input)
print("Block output:")
print(block_output)
print("Block output size:", block_output.size())
```

```
Input:
tensor([[-2.1427, -0.7382,  0.3335,  0.3865, -0.6146],
        [-0.8864,  0.7096,  0.9242, -0.4327, -0.1272],
        [ 0.9784,  0.2195,  0.1283, -2.1508,  0.8675],
        [-0.2716,  0.4867, -0.6735,  2.4885, -0.1850],
        [-0.2660, -0.7077,  2.2901, -0.3677,  0.5883],
        [ 0.3426, -1.5650,  0.0481,  0.4113, -0.2566],
        [-0.4732, -0.3190, -0.8887,  0.5200, -0.9329],
        [-2.9853,  0.1357,  1.3753, -0.4170,  0.5080]])
Input size: torch.Size([8, 5])

Block output:
tensor([[0.3037, 0.7349, 0.1701, 0.5356, 0.8148, 0.1889, 0.6813],
        [0.3476, 0.5693, 0.2838, 0.4702, 0.5560, 0.4595, 0.4932],
        [0.3449, 0.2760, 0.4551, 0.6195, 0.3657, 0.7563, 0.3839],
        [0.5403, 0.7303, 0.6199, 0.3546, 0.7415, 0.1658, 0.5843],
        [0.1345, 0.4733, 0.1670, 0.5443, 0.5537, 0.5599, 0.5445],
        [0.3013, 0.5081, 0.3476, 0.5320, 0.6801, 0.3028, 0.6909],
        [0.5304, 0.6252, 0.4428, 0.4566, 0.7025, 0.2153, 0.6614],
        [0.1900, 0.7365, 0.1062, 0.6096, 0.7908, 0.3465, 0.5088]],
       grad_fn=<SigmoidBackward0>)
Block output size: torch.Size([8, 7])
```

Note the output values produced by the two methods are not the same, because the weights and biases were initialized randomly. But the shape of the output tensors are the same.

You need to implement this (1)

Implement the `create_sequential_layers()` function in the `submitted.py` file. This function should return a `nn.Sequential` object.

Once you have implemented the function, you can run the code cell below to test your implementation.

In [13]:
```
!python -m unittest tests.test_visible.Test.test_sequential_layers
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.040s

OK
```

Build a Model

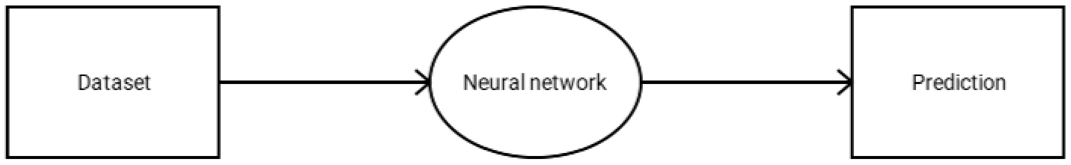
Source: mrdbourke/pytorch-deep-learning and PyTorch Tutorial

Now that we have covered some fundamentals, let's focus on how to build a model. In this MP, you will implement a neural network.

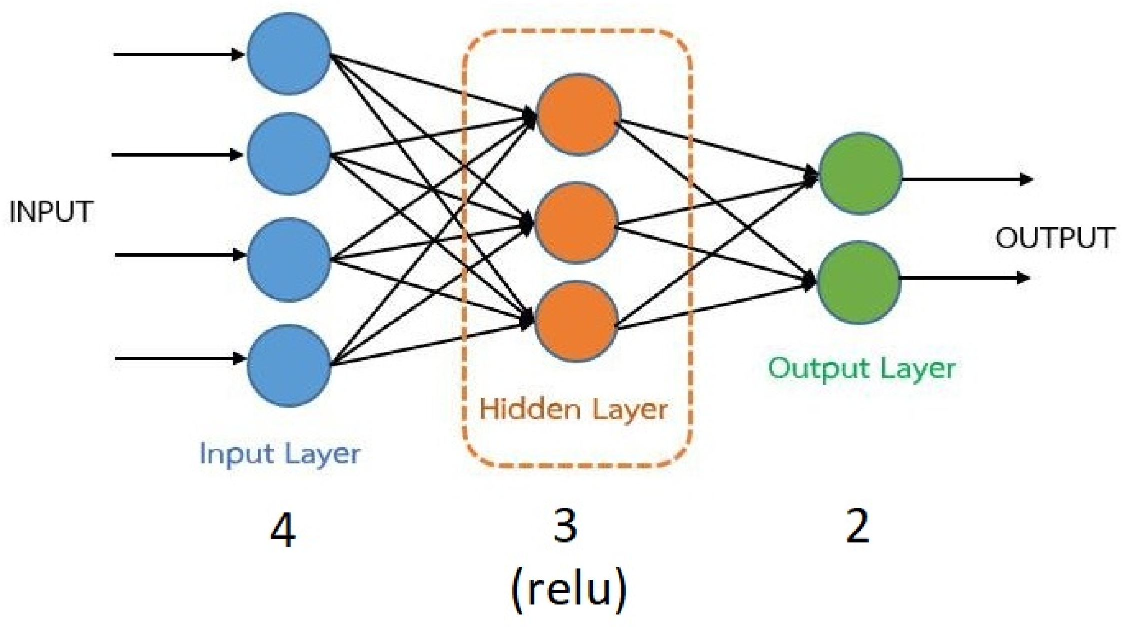
Neural networks comprise of layers/modules that perform operations on data. The torch.nn (a PyTorch module) namespace provides all the building blocks you need to build your own neural network. Every module in PyTorch subclasses the torch.nn.Module (object-oriented programming -- if you are not familar with Python class notation, or if you don't quite understand what it means, you may want to take a look at this). A neural network is a derived class of nn.Module that consists of other modules (layers). This nested structure allows for building and managing complex architectures easily.

The neural network model is defined in two steps:

1. we first specify the parameters (e.g., layers) of the model (in `__init__()`),
2. and then outline how they are applied to the inputs (in `forward()`).



The code snippet provided below is a simple example of how to construct a network and use it to make a prediction. Note that there are ***more than one way*** to construct a network architecture, you can find many examples on the internet.



In [14]:
```
import torch

class SimpleNet(torch.nn.Module):
  def __init__(self):
    """
    In the initialization function we specify the parameters of our network.
```

```
        """
        super().__init__()  # call the initialization function of the base class
        # network architecture, please try to relate the code to the picture
        self.hidden = torch.nn.Linear(4, 3)  # input has 4 values
        self.relu = torch.nn.ReLU()  # activation function
        self.output = torch.nn.Linear(3, 2)  # output has 2 values

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data (the variable x
        We can use Modules defined in the __init__() as well as arbitrary (diffe
        """
        x_temp = self.hidden(x)              # input data x flows through the hid
        x_temp = self.relu(x_temp)           # use relu as the activation functio
        y_pred = self.output(x_temp)         # predicted value
        return y_pred

# Create an instance of the SimpleNet model (this is a subclass of nn.Module
model = SimpleNet()

# Create inputs, here we use a random tensor, but in reality, the input shou
x = torch.rand(3, 4)   # 3 samples, each sample of size 4

# Forward pass: compute predicted y by passing x to the model
# Note that the model is randomly initialized, so this prediction probably c
# We need to train our model and teach it to make reasonable predictions (we
y_pred = model(x)

print("y_pred.shape: ", y_pred.shape)   # since our output layer has 2 value
print(y_pred)
```

```
y_pred.shape:  torch.Size([3, 2])
tensor([[-0.4393,  0.4056],
        [-0.4375,  0.4294],
        [-0.4397,  0.3995]], grad_fn=<AddmmBackward0>)
```

Name	What does it do?
`torch.nn`	Contains all of the building blocks (network layers) for computational graphs (essentially a series of computations executed in a particular way).
`torch.nn.Module`	The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass `nn.Module`. Requires a `forward()` method be implemented.
`__init__()`	`__init__()` is your network's initialization function, where you will initialize the neural network layers.
`forward()`	All `nn.Module` subclasses (e.g., your own network) require a `forward()` method, this defines the computation that will take place on the data passed to the particular `nn.Module`. Simply put, `forward()` should perform a forward pass through your network. Note that you should ***NOT*** directly call the `forward(x)` method, though. To use the model, you should call the whole model itself and pass it the input data, as in `model(x)` to perform a forward pass

Name	What does it do?
	and output predictions. This executes the model's `forward()` automatically.
`torch.optim`	Contains various optimization algorithms (these tell the model parameters stored in `nn.Parameter` how to best change to improve gradient descent and in turn reduce the loss).

Train a Model

Source: PyTorch Tutorial

Training a model is an iterative process; in each iteration the model makes a guess about the output, calculates the error in its guess (loss), collects the derivatives of the error with respect to its parameters, and optimizes these parameters using gradient descent.

Note that the parameters are initialized randomly, and for our model to update its parameters on its own, we'll need to add a ***loss function*** as well as an ***optimizer***.

Loss Function and Optimizer

Function	What does it do?	Where does it live in PyTorch?	Common values
Loss function	Measures how wrong your models predictions (e.g. `y_preds`) are compared to the truth labels (e.g. `y_test`). Lower the better.	PyTorch has plenty of built-in loss functions in `torch.nn`.	Mean absolute error (MAE) for regression problems (`torch.nn.L1Loss()`). Binary cross entropy for binary classification problems (`torch.nn.BCELoss()`). Cross entropy for multi-class classification problems (`torch.nn.CrossEntropyLoss()`)
Optimizer	Tells your model how to update its internal parameters to best lower the loss.	You can find various optimization function implementations in `torch.optim`.	Stochastic gradient descent (`torch.optim.SGD()`). Adam optimizer (`torch.optim.Adam()`).

You need to implement this (2)

Implement the `create_loss_function()` function in the `submitted.py` file. This function should return a loss object that is suitable for the classification problem.

Once you have implemented the function, you can run the code cell below to test your implementation.

In [15]:
```
!python -m unittest tests.test_visible.Test.test_loss_fn
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.039s

OK
```

Gradients

When training neural networks, the most frequently used algorithm is back propagation. In this algorithm, parameters (model weights, biases, ...) are adjusted according to the gradient of the loss function with respect to the given parameter. To compute those gradients, PyTorch has a built-in differentiation engine called `torch.autograd`. It supports automatic computation of gradient for any computational graph. Take a look at this for more information about autogradient in PyTorch (you should be able to understand everything that is covered in the linked tutorial by now).

Time for an example:

In [16]:
```
y_pred = model(x)
print("Predicted values:")
print(y_pred, "\n")

# Create a loss function, e.g., Mean Squared Error
loss_fn = torch.nn.MSELoss()

# Make up some true values, in reality, they should come from a real-world d
y_true = torch.tensor(
    [
        [1, 1],
        [1, 1],
        [1, 1]
    ], dtype=torch.float32)
print("True values:")
print(y_true, "\n")

# Calculate MSE
loss = loss_fn(y_pred, y_true)
print("MSE:", loss, "\n\n\n")       # You can verify the results manually


# Create an optimizer, e.g., SGD optimizer
optimizer = torch.optim.SGD(params=model.parameters(), lr=1)
# params:   parameters of target model to optimize
# lr:       learning rate (how much the optimizer should change parameters a

print("Weights of hidden linear layer, before back propagation:")
print(model.hidden.weight, "\n")

# Preform back propagation
optimizer.zero_grad()   # Clear previous gradients, will see more about this
loss.backward()         # back propagation
# Here we only print the gradients of hidden.weight,
# but backward() updates gradients of all related parameters
print("gradients of weights of the hidden layer:")
print(model.hidden.weight.grad, "\n")

# Update parameters
optimizer.step()
print("Weights of hidden linear layer, after back propagation:")
print(model.hidden.weight) # You can verify the results, after = before - gr
```

```
Predicted values:
tensor([[-0.4393,  0.4056],
        [-0.4375,  0.4294],
        [-0.4397,  0.3995]], grad_fn=<AddmmBackward0>)

True values:
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])

MSE: tensor(1.2084, grad_fn=<MseLossBackward0>)



Weights of hidden linear layer, before back propagation:
Parameter containing:
tensor([[-0.4284,  0.2926, -0.4868, -0.1456],
        [ 0.4332,  0.2243,  0.4980,  0.2780],
        [ 0.1151, -0.1169, -0.1383, -0.2282]], requires_grad=True)

gradients of weights of the hidden layer:
tensor([[ 0.0000,  0.0000,  0.0000,  0.0000],
        [-0.0840, -0.0754, -0.0986, -0.0919],
        [ 0.0000,  0.0000,  0.0000,  0.0000]])

Weights of hidden linear layer, after back propagation:
Parameter containing:
tensor([[-0.4284,  0.2926, -0.4868, -0.1456],
        [ 0.5172,  0.2997,  0.5966,  0.3699],
        [ 0.1151, -0.1169, -0.1383, -0.2282]], requires_grad=True)
```

Hyperparameters

Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates.

We define the following hyperparameters for training:

- ***Number of Epochs*** - the number of times to iterate over the dataset (one epoch = one forward pass and one backward pass of all the training samples)
- ***Batch Size*** - the number of data samples propagated through the network before the parameters are updated
- ***Learning Rate*** - how much to update models parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training.

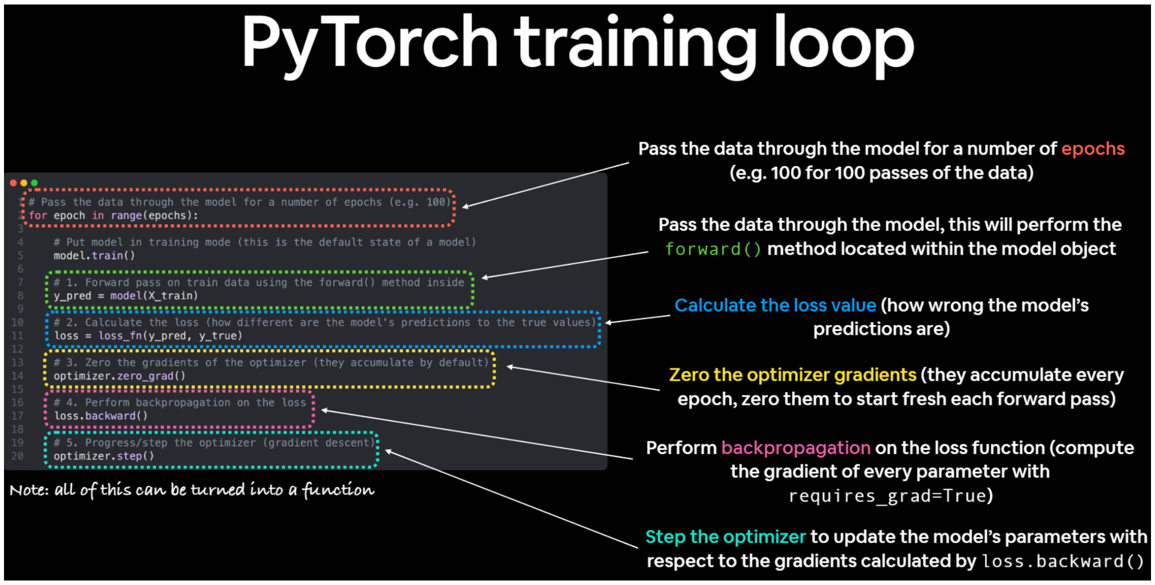
Batch size

Batch size might be confusing if this is your first time hearing it. You can find a wonderful explanation here.

> Let's say you have 1050 training samples and you want to set up a batch_size equal to 100. The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network. Next, it takes the second 100 samples (from 101st to 200th) and trains the network again. We can keep doing this procedure until we have propagated all samples through of the network. Problem might happen with the last set of samples. In our example, we've used 1050 which is not divisible by 100 without remainder. The simplest solution is just to get the final 50 samples and train the network.
>
> Advantages of using a batch size < number of all samples:
>
> - It requires less memory. Since you train the network using fewer samples, the overall training procedure requires less memory. That's especially important if you are not able to fit the whole dataset in your machine's memory.
> - Typically networks train faster with mini-batches. That's because we update the weights after each propagation. In our example we've propagated 11 batches (10 of them had 100 samples and 1 had 50 samples) and after each of them we've updated our network's parameters. If we used all samples during propagation we would make only 1 update for the network's parameter.
>
> Disadvantages of using a batch size < number of all samples:
>
> - The smaller the batch the less accurate the estimate of the gradient will be.

Annotated Code

Please note that the code snippets below do not make use of batch size, instead it uses one single "batch" with all training data. Take a look here for an example with batch size. In this MP, you will write a training loop which operates data in batches i.e., using dataloaders.



PyTorch training loop

```python
# Pass the data through the model for a number of epochs (e.g. 100)
for epoch in range(epochs):

    # Put model in training mode (this is the default state of a model)
    model.train()

    # 1. Forward pass on train data using the forward() method inside
    y_pred = model(X_train)

    # 2. Calculate the loss (how different are the model's predictions to the true values)
    loss = loss_fn(y_pred, y_true)

    # 3. Zero the gradients of the optimizer (they accumulate by default)
    optimizer.zero_grad()

    # 4. Perform backpropagation on the loss
    loss.backward()

    # 5. Progress/step the optimizer (gradient descent)
    optimizer.step()
```

Note: all of this can be turned into a function

Pass the data through the model for a number of epochs (e.g. 100 for 100 passes of the data)

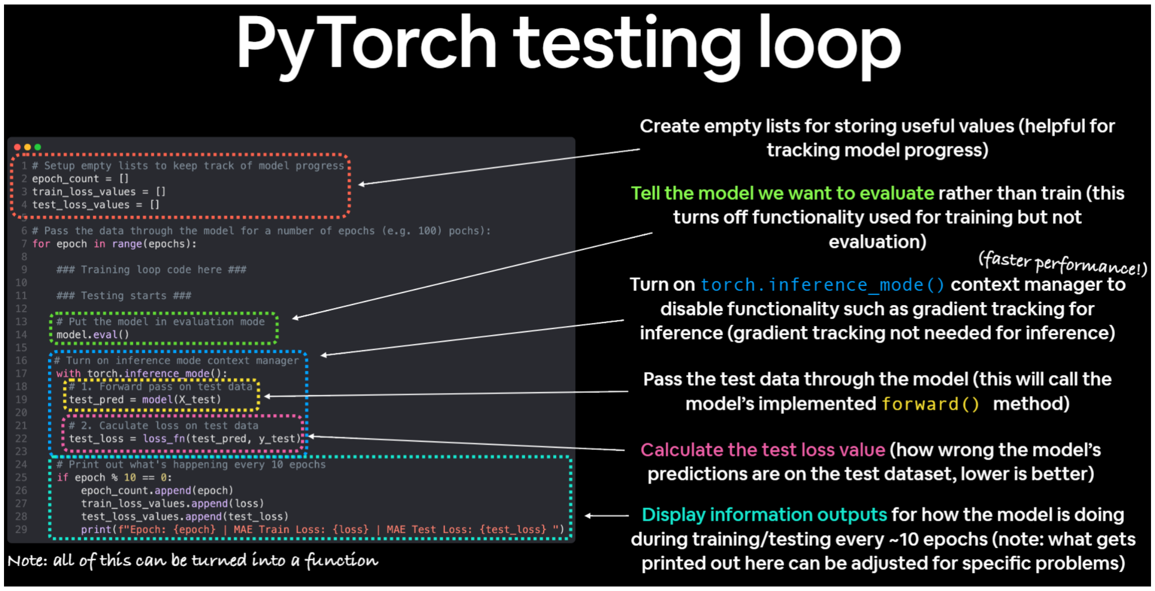
Pass the data through the model, this will perform the `forward()` method located within the model object

Calculate the loss value (how wrong the model's predictions are)

Zero the optimizer gradients (they accumulate every epoch, zero them to start fresh each forward pass)

Perform backpropagation on the loss function (compute the gradient of every parameter with `requires_grad=True`)

Step the optimizer to update the model's parameters with respect to the gradients calculated by `loss.backward()`



PyTorch testing loop

```python
# Setup empty lists to keep track of model progress
epoch_count = []
train_loss_values = []
test_loss_values = []

# Pass the data through the model for a number of epochs (e.g. 100) pochs):
for epoch in range(epochs):

    ### Training loop code here ###

    ### Testing starts ###

    # Put the model in evaluation mode
    model.eval()

    # Turn on inference mode context manager
    with torch.inference_mode():
     # 1. Forward pass on test data
     test_pred = model(X_test)

     # 2. Caculate loss on test data
     test_loss = loss_fn(test_pred, y_test)
    # Print out what's happening every 10 epochs
    if epoch % 10 == 0:
        epoch_count.append(epoch)
        train_loss_values.append(loss)
        test_loss_values.append(test_loss)
        print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss} ")
```

Note: all of this can be turned into a function

Create empty lists for storing useful values (helpful for tracking model progress)

Tell the model we want to evaluate rather than train (this turns off functionality used for training but not evaluation)

(faster performance!)

Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference)

Pass the test data through the model (this will call the model's implemented `forward()` method)

Calculate the test loss value (how wrong the model's predictions are on the test dataset, lower is better)

Display information outputs for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

You need to implement this (3)

Now it's time for you to create a neural network. You will implement a neural network in `Class NeuralNet` and write a training loop in the function `train()`.

In each training iteration, the input of your network is a batch of preprocessed image data of size (batch size, 2883). The number of neurons in the output layer should be equal to the number of categories.
For this assignment, you should use Cross Entropy Loss. Notice that because

PyTorch's CrossEntropyLoss incorporates a softmax function, you do not need to explicitly include an normalization function in the last layer of your network.

To get a full score, the accuracy of your network must be above `0.61 on the visible testing set`, and `above 0.57 on the hidden testing set`. The structure of the neural network is completely up to you. You should be able to get around 0.62 testing-set accuracy with a two-layer network with no more than 200 hidden neurons.

If you are confident about a model you have implemented but are not able to pass the accuracy thresholds on gradescope, try adjusting the learning rate. Be aware, however, that using a very high learning rate may worse performance since the model may begin to oscillate around the optimal parameter settings.

Once you have implemented the function, you can run the code cell below to test your implementation.

In [1]:
```
!python grade.py
```

```
Total number of network parameters:  138677

 Accuracy: 0.6264674493062967

Confusion Matrix =
 [[146.  24.   8.   5.   7.]
 [ 27. 123.  11.  22.  19.]
 [ 13.  13.  87.  45.  28.]
 [  9.   4.  34. 122.  17.]
 [ 14.  10.  23.  17. 109.]]
+5 points for accuracy above 0.15
+5 points for accuracy above 0.25
+5 points for accuracy above 0.48
+5 points for accuracy above 0.55
+5 points for accuracy above 0.57
+5 points for accuracy above 0.61
...
----------------------------------------------------------------------
Ran 3 tests in 2.395s

OK
```

Extra credit

You can earn extra credits worth 10% of this MP if the accuracy of your network is above `0.66`

Some tips:

1. Choose a good activation function.

2. L2 Regularization.
3. Convolutional neural network.

```
In [ ]:
```