Instructor: Alina Vereshchaka

Assignment 2

Building Neural Networks and CNNs

Checkpoint: October 23, Thu, 11:59pm

Due Date: November 6, Thu, 11:59pm

Description

Welcome to our second assignment. This assignment focuses on building fully connected neural networks (NN) and convolutional neural networks (CNN). It consists of four parts where you practice dealing with various datasets and implement, train, and adjust neural networks.

The first part consists of performing data analysis and building a basic NN. In the second part, we learn how to optimize and improve your NN using various techniques. In the third part, we will implement a basic CNN and apply optimization and in part 4 we will implement the VGG-11 model. There is also a bonus task focusing on implementing more complex architecture.

Notes:

- **Deep learning framework:** this assignment should be completed using PyTorch. You can refer to the Pytorch tutorials on how to get started.
- **Libraries:** any pre-trained or pre-built NN or CNN architectures cannot be used (e.g. torchvision.models). This time you can use scikit-learn for data preprocessing.
- **GPU:** UB provides access to CCR, refer to piazza on how to get access. You can use your local device to train the models. You can also explore free options provided by Google Colab, Kaggle, AWS Sagemaker and others.
- **Large files:** any file that is larger then 10MB (e.g. your model weights) should be uploaded to UBbox and you have to provide a link to them. A penalty of -10pts will be applied towards any file submitted that are bigger than 10MB.

Background

Let's consider a generic structure for defining a neural network in Pytorch. Click here for more details.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__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

model = NeuralNetwork().to(device)
print(model)


X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

This code defines a neural network with three layers:

- Flatten layer that flattens the input image tensor
- Two fully connected layers with 512 hidden units and ReLU activation functions
- Final fully connected layer with 10 output units (corresponding to the 10 possible classes in the MNIST dataset)

The forward method specifies how input data is passed through the layers of the network.

It defines a neural network using PyTorch's `nn.Module` class and the `nn.Sequential` module, and then uses the network to make a prediction.

`nn.Module` is a base class in the PyTorch module that provides a framework for building and organizing complex neural network architectures.

When defining a neural network module in PyTorch, you usually create a class that inherits from `nn.Module`. The `nn.Module` class provides a set of useful methods and attributes that help in building, training and evaluating the neural network.

Some of the key methods provided by `nn.Module` are:

`__init__()`: This is the constructor method that initializes the different layers and parameters of the neural network.

`forward(self, x)`: This method defines the forward pass of the neural network. It takes the input tensor (`x`) and returns the predicted probabilities for each class.

`nn.Linear` is a PyTorch module that applies a linear transformation to the input data. It is one of the most used modules in deep learning for building neural networks.

The `nn.Linear` module takes two arguments: `in_features` and `out_features` - the number of input/output features. When an input tensor is passed through an `nn.Linear` module, it is first flattened into a 1D tensor and then multiplied by a weight matrix of size `out_features x in_features`. A bias term of size `out_features` is also added to the output.

The output of an `nn.Linear` module is given by the following equation:

$$output = w^T x + b$$

```
model = NeuralNetwork().to(device)
print(model)
```

Here we create an instance of the NeuralNetwork class and moves it to the device specified by the device variable (which should be set to 'cuda' for GPU or 'cpu' for CPU). It is also good practice to print the summary of the architecture of the NN.

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

This code generates a random input image tensor of size 1x28x28 (representing a single 28x28 grayscale image) and passes it through the neural network using the model instance. The output of the network is a tensor of size 1x10, representing the predicted probabilities for each of the 10 possible classes.

`nn.Softmax` module is used to convert these probabilities to a valid probability distribution, and the argmax method is used to obtain the class with the highest probability.

Part I: Building a Basic NN [30 points]

In this part, you will implement a neural network using the PyTorch framework. You will train the network on the provided dataset, which consists of seven features and a binary target variable. Your objective is to accurately predict a target based on the input features.

Step 1: Loading the Dataset and main statistics

1. Load the dataset (dataset.csv). It is provided on UBlearns > Assignments. You can use the pandas library to load the dataset into a DataFrame
2. Analyze the dataset, e.g., return the main statistics.

3. Handle invalid character entries, if any. Hint: for each features you can use unique() and replace() functions to identify and replace invalid values.
4. Provide at least 3 visualization graphs with a short description.

Note: You can reuse your code from A0 or A1 with a proper citation.

Step 2: Preprocessing and Splitting the Dataset

1. Preprocess the dataset before we use it to train the neural network.

   Preprocessing typically involves converting categorical variables to numerical variables, scaling numerical variables, etc.

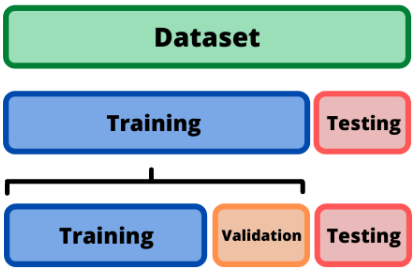
   For this dataset, you can use the following preprocessing steps:

   - Scale numerical variables or normalize the data. You can use StandardScaler from scikit-learn or Normalize from PyTorch
   - [Optional] Address class imbalance in the target column. Possible solutions: oversampling; undersampling; data augmentation techniques for the minority class; assign higher weights to the minority class and lower weights to the majority class, etc.
   - [If needed] Convert categorical variables to numerical variables using one-hot encoding. You can use OneHotEncoder from scikit-learn
   - Provide brief details on how you preprocessed the dataset.

2. Split the dataset into training, testing and validation sets.

   You can use train_test_split from scikit-learn

   Hint: first you can split the dataset into 'training' and 'testing' batches. Then take the 'training' batch and split it again for 'training' and 'validation'.



Why do we need to split into training, testing and validation?

- Training set: used to train the model to learn the patterns or features in the data. It is important to have a large and representative training set so that the model can learn well.

- Validation set: used to tune the model hyperparameters and to prevent overfitting. Overfitting is when the model learns the training data too well and cannot generalize to new data. Hyperparameters are parameters that are not learned from the data, but are set by the user. By tuning the hyperparameters, we can improve the model's performance on the validation set.
- Test set: used to evaluate the final model's performance on unseen data. This gives us a good idea of how well the model will perform in the real world.

The commonly used splits of 70:15:15 or 80:10:10 are good starting points, but the optimal split ratio will vary depending on the size and characteristics of the dataset.

3. PyTorch models expect input data in the form of tensors, thus convert the DataFrame to PyTorch tensor (torch.tensor()). Check more detail here

Step 3: Defining the Neural Network

Now, we need to define the NN that we will use to make predictions on the dataset. For this part, you can define a simple NN and improve it further in the next steps.

1. Decide your NN architecture:
   - How many input neurons are there?
   - How many output neurons are there?
   - What activation function is used for the hidden layers?
     - Suggestion: try ReLU, ELU, Sigmoid, click here for the full list
   - What activation function is used for the output layer?
   - What is the number of hidden layers?
     - Suggestion: start with a small network, e.g., 2 or 3 hidden layers
   - What is the size of each hidden layer?
     - Suggestion: try 64 or 128 nodes for each layer
   - Do you include Dropout? (details)
2. Define your NN architecture using PyTorch (basic building blocks in PyTorch).
3. Return the summary of your model. You can use Torchinfo package

Step 4: Training the Neural Network

1. Define the loss function (loss_function) that will be used to compute the error between the predicted output and the true labels of the training data. List of loss functions (PyTorch).

   For binary classification problems, a commonly used loss function is Binary Cross Entropy Loss (Details).

2. Choose an optimizer and a learning rate. It will update the weights of the NN during training. SGD is one of the commonly used, you can also explore other optimizers like Adam or RMSProp.
   Check a list of optimizers (PyTorch)
3. Set up the training loop:
   a. Create a loop that iterates over the training data for a specified number of epochs.
   b. Iterate over the training data in batches.
   c. Forward pass: pass the input data through the neural network.

```
outputs = model(inputs)
```

   d. Compute loss: calculate the loss using the defined loss function.

```
loss = loss_function(outputs, labels)
```

   e. Backpropagation: zero the gradients, compute gradients, and perform backpropagation.

```
optimizer.zero_grad()    #Clear gradients
loss.backward()          #Backpropagation
```

   f. Update the model's weights using the optimizer.

```
optimizer.step()         #Update weights
```

   g. Validation phase: set the model to evaluation mode.

```
model.eval()             #Set to evaluation mode
```

   During the validation phase, there will be no backward propagation to calculate the gradients and no optimizer step to update the steps.

```
with torch.no_grad(): # Disable gradients
      for val_inputs, val_labels in val_loader:
            val_outputs = model(val_inputs)
            val_loss = loss_function(val_outputs, val_labels)
```

4. Train the neural network. Run the training loop and train the neural network on the training data. Select the number of epochs and batch size. Monitor the training loss and the validation loss at each epoch to ensure that the model is not overfitting. Estimate the time it takes to train the model, e.g. using time.time().
5. Save the weights of the trained neural network that returns the best results.
   Saving and loading models in PyTorch
6. Evaluate the performance of the model on the testing data. Suggested metrics:
   - Accuracy (PyTorch functions)
   - Precision, recall and F1 score (more details). You can use sklearn.metrics.precision_recall_fscore_support

- In case you need to convert the output of sigmoid() to 0 or 1, you can use torch.round() function

Note: The expected accuracy on the testing dataset for this task is > 75%.

7. Visualize the results. Include the following graphs:
    a. A graph that compares training, validation and test accuracy on the same plot with clear labeling. Include your analysis of the results.
    b. A graph that compares training, validation and test loss on the same plot with a clear labeling. Include your analysis of the results.

       Note: your test accuracy and loss are obtained during the prediction phase and it is a single value. To plot it on the same graph, you can repeat this value for the same number of epochs as training, to get a straight line and plot all of them in the same graph.
    c. Confusion matrices on the test data (seaborn.heatmap() or PyTorch). Include your analysis of the results.
    d. ROC curve (receiver operating characteristic curve). Include your analysis of the results.
        - Details about ROC
        - PyTorch ROC function
8. References. Include details on all the resources used to complete this part, e.g. links to datasets, research papers or articles, code examples or tutorials you referred.
9. [For teams of 2] Include a contribution summary for each team member in the format:

Step	Teammate	Contribution %

Part II: Optimizing NN [20 points]

Based on your NN model defined in Part I, tune the hyperparameters and apply various tools and technics to increase the accuracy.

STEPS:

1. Choose one hyperparameter to modify (e.g., Dropout). Fix the NN structure and all other parameters based on the model defined in Part I and change values only for your chosen hyperparameter.
    - Provide the results in the form of a table:

"NAME OF THE HYPERPARAMETER" Tuning

	Value	Test Accuracy
Setup #1		
Setup #2		
Setup #3		

2. Choose another hyperparameter and go to Step 1. Do the same procedure for three hyperparameters. Notes:
    - You can consider adjusting initialization, number of layers, dropout, batch size, optimizer, activation function, etc.
    - After this step, you need to provide three NN models with one changed hyperparameter that returns the best results. E.g., three different NN models, that have slight changes.
    - You can improve your model sequentially, e.g. adjusting a dropout rate and using it as a base model to tune other parameters. Or you can make tuning solely on the base model defined in Part I.
3. After completing Step 2, choose a model with tuned hyperparameters that returns the best test accuracy and use it as a 'base' model. Provide your analysis on how various hyperparameter values influence the accuracy.
4. Further improve your model. There are a few methods which can help increase the training speed, accuracy, etc. Find and try at least four different methods (e.g. earlystopping, k-fold, learning rate scheduler, batch normalization, data augmentation, gradient accumulation, more details on performance tuning)
    a. Choose a method that can improve the performance of the model. Add it to your 'base model, finalized in Step 3.
    b. Train the model with a new method. Provide a graph that compares test accuracy for a 'base' model and an improved version. You can also provide a comparison w.r.t training time and other parameters.
    c. Go to Step 4a. Try four various methods or tools that aim to improve the performance of your model.
    d. After you explore various methods, finalize your NN model that returns the best results, named as 'best model'.
    e. Discuss all the methods you used that help to improve the accuracy or the training time.
5. For your 'best model' defined in Step 4:
    a. Save the weights of the trained neural network (refer to Part I, Step 5)
    b. Evaluate the performance of the model on the testing data (refer to Part I, Step 6)
    c. Visualize the results (refer to Part I, Step 7)

   d. Provide a detailed description of your 'best model' that returns the best results. Discuss the performance.
6. References. Include details on all the resources used to complete this part, e.g. links to datasets, research papers or articles, code examples or tutorials you referred.
7. [For teams of 2] Include a contribution summary for each team member in the format:

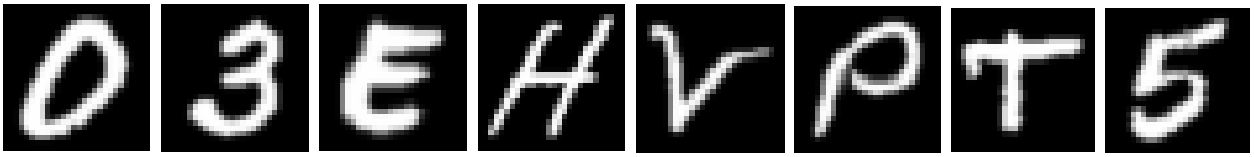
Step	Teammate	Contribution %

Note: The test accuracy for all submitted setups in Part 2 should be greater than 75%.

Part III: Building a CNN [30 points]

In this part we will work on a dataset containing multiple classes.

CNN Dataset



For this part our dataset consists of 36 categories and 2800 examples for each category, thus in total 100,800 samples. Each example is a 28x28 image. A version of the dataset for this assignment can be downloaded from UBlearns (cnn_dataset.zip). Note: The images in the CNN dataset may have either 1 or 3 channels. If an image has 1 channel (grayscale), you can still work with it without any issues.

Read more about the dataset (paper)

STEPS:

1. Load, preprocess, analyze, visualize the dataset and make it ready for training. You can reuse your code from Part I.

   To load the data, there are few options:

   - torchvision.datasets.ImageFolder

- dataloader. In this case, you can obtain the labels by extracting them based on the folder structure. Each folder name corresponds to a specific label.

2. Build and train a basic CNN (with max 10 hidden layers). Decide your CNN architecture:
   - How many input neurons are there?
   - How many output neurons are there?
   - What activation function is used for the hidden layers?
     - Suggestion: try ReLU, ELU, Sigmoid, click here for the full list
   - What activation function is used for the output layer?
   - What is the number of hidden layers?
   - What is the size of each hidden layer?
   - Do you include Dropout? (details)
3. Define your CNN architecture using PyTorch (basic building blocks in PyTorch).
4. Return the summary of your model. You can use Torchinfo package
5. Train your model. You can refer to the code from your Part I implementation.
6. Add at least three improvement methods that you tried for "Part II - Step 4" of this assignment, that are applicable to CNN architecture.
7. Save the weights of the trained neural network that returns the best results. Saving and loading models in PyTorch Describe your CNN architecture choice.
8. Evaluate the performance of the model on the testing data. Suggested metrics:
   - Time to train (e.g. using time.time())
   - Accuracy (PyTorch functions)
   - Precision, recall and F1 score (more details). You can use sklearn.metrics.precision_recall_fscore_support
   - In case you need to convert the output of sigmoid() to 0 or 1, you can use torch.round() function

   Note: The expected accuracy on the testing dataset for this task is > 85%.
9. Visualize the results. Include the following graphs:
   a. A graph that compares training, validation and test accuracy on the same plot with clear labeling. Include your analysis of the results.
   b. A graph that compares training, validation and test loss on the same plot with a clear labeling. Include your analysis of the results.

      Note: your test accuracy and loss are obtained during the prediction phase and it is a single value. To plot it on the same graph, you can repeat this value for the same number of epochs as training, to get a straight line and plot all of them in the same graph.

c. Confusion matrices on the test data (seaborn.heatmap() or PyTorch). Include your analysis of the results.

d. ROC curve (receiver operating characteristic curve). Include your analysis of the results.
   - Details about ROC
   - PyTorch ROC function

10. References. Include details on all the resources used to complete this part, e.g. links to datasets, research papers or articles, code examples or tutorials you referred.

11. [For teams of 2] Include a contribution summary for each team member in the format:

Step	Teammate	Contribution %

Part IV: VGG-13 Implementation [20 points]

VGG is one of the commonly used CNN architectures. It has different versions. In this part we implement the VGG-13 (Version B) and apply it to solve our task. The expected accuracy on the testing dataset for this task is > 85%.

STEPS:

1. Implement the VGG-13 (Version B) architecture following the proposed architecture and adjusting it to fit our dataset. Refer to the original VGG paper for more details.

2. Adjustments based on your dataset:

   a. The input size for the original VGG is 224x224. Here are a few options:

      i. You can remove a few max pooling layers.

      ii. Add padding to your convolutional layers to maintain more spatial dimensions.

      iii. Resize your input size

   b. The last layer of the original VGG contains 1000 classes. You can modify it to fit the number of your classes (e.g. 36).

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

3. Use the dropout and learning rate scheduler, as in the VGG paper (Chapter 3.1 Training)
4. Train the model on a dataset used in Part III (EMNIST).
5. Evaluate the performance of the model on the testing data and visualize the results. Follow Part III, Steps 6&7
6. References. Include details on all the resources used to complete this part, e.g. links to datasets, research papers or articles, code examples or tutorials you referred.
7. [For teams of 2] Include a contribution summary for each team member in the format:

Step	Teammate	Contribution %

Bonus points [10 points]

The bonus section is optional but encouraged if you want to explore deeper neural architectures or interpretability techniques. You may complete one or both parts.

Implement ResNet-34 [5 points]

Implement and train a 34-layer Residual Network (ResNet-34) from scratch using PyTorch.

STEPS:

1. Model implementation:
   - Build the ResNet-34 architecture following the configuration described in the original paper: configuration 34-layer (refer to the paper, page 774)
   - Use modular PyTorch code (e.g., BasicBlock, ResNet class).
2. Apply your ResNet-34 model to the Flower Dataset. Perform standard preprocessing and data augmentation (resize, normalization, etc.).
3. Training and evaluation:
   - Train the model using various hyperparameters (learning rate, batch size, epochs).
   - Evaluate on a validation/test set and report the accuracy. Reuse the evaluation metrics from the previous part in the assignment.
   - **Target goal: Accuracy ≥85%.**
4. Visualization and analysis
   - Include visualizations such as training/validation accuracy curves, confusion matrix, or sample predictions.
   - Discuss observed challenges or key findings.
5. Save the trained model weights that return the best results (.pickle or .h5).

Model's Interpretability [5 points]

Add the interpretability of the ResNet-34 model by visualizing what different layers learn.

STEPS:

1. Convolutional layer outputs:
    - Select at least three convolutional layers for visualization (e.g., early, middle, and late layers).
    - Extract the feature maps using hooks. Ideas
    - Visualize the feature maps and analyze differences between layers.
2. Activation layer outputs:
    - Display the activation maps from selected layers after applying activation functions (e.g., ReLU).
    - Use hooks to capture the outputs after activation. Analyze how these activations change with different inputs.
    - Provide visualizations that illustrate which features are emphasized or suppressed in the network. Discuss your observations.
3. Weight visualization:
    - Extract and visualize the weights (kernels) from the early convolutional layers
    - Use `model.conv1.weight.data` to access the weights and visualize
    - Discuss the types of features (e.g., edges, textures) that the network learns from these weights

Bonus part submission:

- Create a separate Jupyter Notebook (.ipynb) named as a2_bonus_TEAMMATE1_TEAMMATE2.ipynb
  e.g., a2_bonus_ avereshc_ pinazmoh.ipynb
- You can duplicate code from your other parts if needed
- Submit Jupyter Notebook (.ipynb) with saved outputs
- Saved weights that generate the best results for your model (.pickle or h5).
- Report is not required; include all the analysis as part of your Jupyter Notebook.
- Submit your Bonus related files as part of your Final Submission.

ASSIGNMENT STEPS

1. Register your team

You may work individually or in a team of up to 2 people. The evaluation will be the same for a team of any size. Teammates for A1, A2 & A3 should be different.

Register your team at UBLearns > Groups. In case you joined the wrong group, make a private post on piazza.

2. Submit checkpoint (October 23)

- Complete Part I and Part II of the assignment.
- Include all the references at the end of each part.
- Your Jupyter notebook should be saved with the outputs.
- If you are working in a team, we expect equal contribution for the assignment. Each team member is expected to make a code-related contribution. Provide a contribution summary by each team member in the form of a table below. If the contribution is highly skewed, then the scores of the team members may be scaled w.r.t the contribution.

Step	Teammate	Contribution (%)

- Datasets should NOT be submitted. Do not include it as part of the submission.
- Saved weights that generate the best results for your model, named as a2_part#_TEAMMATE1_ TEAMMATE2.pkl or .pickle or .h5

  e.g. a2_part1_avereshc_ manasira.pickle
- Combine all files in a single zip folder that will be submitted on UBlearns, named as assignment_2_checkpoint_YOUR_UBIT.zip
- Submit to UBLearns > Assignments
- Suggested file structure:

  assignment_2_checkpoint_TEAMMATE1_ TEAMMATE2.zip
  - a2_part_1_ TEAMMATE1_ TEAMMATE2.ipynb
  - a2_part_2_ TEAMMATE1_ TEAMMATE2.ipynb
  - a2_part_1_weights_TEAMMATE1_ TEAMMATE2.pkl
  - a2_part_2_weights_TEAMMATE1_ TEAMMATE2.pkl

4. Submit final results (November 4)

- Complete all parts of the assignment (I-IV, plus Bonus if attempted).
- Add all your assignment files in a zip folder including ipynb files for Part I, Part II, Part III, Part IV & Bonus part (optional) and txt file with a link to UBbox with links to weights
- Jupyter Notebooks must contain saved outputs and plots visible.
- Include all references at the end of each part.
- You may make unlimited submissions, only the latest will be evaluated.
- Dataset should NOT be submitted. Don't include as part of the submission
- Name zip folder with all the files as assignment_2_final_ TEAMMATE1_ TEAMMATE2.zip
  e.g. assignment_2_final_avereshc_manasira.zip
- Submit to UBLearns > Assignments
- Your Jupyter notebook should be saved with the outputs.
- Include all the references at the end of each part that have been used to complete that part.
- You can make unlimited number of submissions and only the latest will be evaluated
- If you are working in a team, we expect equal contribution for the assignment. Each team member is expected to make a code-related contribution. Provide a contribution summary by each team member in the form of a table below. If the contribution is highly skewed, then the scores of the team members may be scaled w.r.t the contribution.

Step	Teammate	Contribution (%)

- Suggested file structure (bonus part is optional):

  assignment_2_final_TEAMMATE1_ TEAMMATE1.zip
  - a2_part_1_ TEAMMATE1_ TEAMMATE2.ipynb
  - a2_part_2_ TEAMMATE1_ TEAMMATE2.ipynb
  - a2_part_3_ TEAMMATE1_ TEAMMATE2.ipynb
  - a2_part_4_ TEAMMATE1_ TEAMMATE2.ipynb
  - a2_part_1_weights_TEAMMATE1_ TEAMMATE2.pkl
  - a2_part_2_weights_TEAMMATE1_ TEAMMATE2.pkl

- a2_part_3_weights_TEAMMATE1_ TEAMMATE2.pkl
- a2_part_4_weights_TEAMMATE1_ TEAMMATE2.pkl
- a2_bonus_resnet_TEAMMATE1_ TEAMMATE2.ipynb
- a2_bonus_interpretability_TEAMMATE1_ TEAMMATE2.ipynb

Notes:

- Ensure your code is well-organized and includes comments explaining key functions and attributes. Also ensure that all the section headings for your solutions corresponding to this assignment are displayed. After running the Jupyter Notebooks, all results and plots used in your report should be generated and clearly displayed.
- You can submit multiple files, but they all need to be labeled with a clear name.
- Recheck the submitted files, e.g. download and open them, once submitted and verify that they open correctly
- Large files: any individual file, except .ipynb file, that is larger then 20MB (e.g. your model weights) should be uploaded to UBbox and you have to provide a link to them. A penalty of -10pts will be applied towards the assignment if there is a file submitted that is bigger than 20MB.
- It's ok if your final combined zip folder is larger then 20MB.
- The zip folder should include all relevant files, clearly named as specified.
- Only files uploaded on UBlearns and a submitted link to UBbox for saved datasets are considered for evaluation.
-

Academic Integrity

The standing policy of the Department is that all students involved in any academic integrity violation (e.g., plagiarism in any way, shape, or form) will receive an F grade for the course. The catalog describes plagiarism as "Copying or receiving material from any source and submitting that material as one's own, without acknowledging and citing the particular debts to the source, or in any other manner representing the work of another as one's own.". Refer to the Office of Academic Integrity for more details.

Important Information

This assignment can be done individually or in a team of up to two students. The grading rubrics are the same for a team of any size.

- No collaboration, cheating, and plagiarism is allowed in assignments, quizzes, the midterms or final project.

- All the submissions will be checked using MOSS as well as other tools. MOSS is based on the submitted works for the past semesters as well the current submissions. We can see all the sources, so you don't need to worry if there is a high similarity with your Checkpoint submission.
- The submissions should include all the references. Kindly note that referencing the source does not mean you can copy/paste it fully and submit as your original work. Updating the hyperparameters or modifying the existing code is a subject to plagiarism. Your work has to be original. If you have any doubts, send a private post on piazza to confirm.
- All parties involved in any suspicious cases will be officially reported using the Academic Dishonesty Report form. Please refer to the Academic Integrity Policy for more details.

Late Days Policy

You can use up to 7 late days throughout the course that can be applied to any assignment-related due dates. You do not have to inform the instructor, as the late submission will be tracked in UBLearns.

If you work in teams, the late days used will be subtracted from both partners. In other words, you have 4 late days, and your partner has 3 late days left. If you submit one day after the due date, you will have 3 days, and your partner will have 2 late days left.

Important Dates

October 23, Thursday - Checkpoint is Due

November 6, Thursday - Final Submission is Due