PA - Part I: Basic Vision Models [Experiments with MNIST] (55pt)

Keywords: Multiclass Image Classification, Neural Networks, PyTorch

MNIST

- The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.
- The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image.
- However to speed up computations, we will use a much smaller dataset with size 8×8 images. These images are loaded from `sklearn.datasets`.

Agenda:

- The PA is split into three parts, the first part dealing with miniature models which we will build from scratch, the second part dealing with modern architectures and the bonus third part dealing with training vision models robust to adversarial attacks.
- In this part, we will be performing multiclass classification on the simplified MNIST dataset from scratch.
- We will be applying and analysing different loss functions, optimization techniques and learning methods.
- We will be using PyTorch to do most of the heavylifting for modeling and training.

Note:

- Hardware acceleration (GPU) is recommended but not required for this part.
- A note on working with GPU:
  - Take care that whenever declaring new tensors, set `device=device` in parameters.
  - You can also move a declared torch tensor/model to device using `.to(device)`.
  - To move a torch model/tensor to cpu, use `.to('cpu')`
  - Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU).
- Run all the cells in order.
- Only **add your code** to cells marked with "TODO:" or with "..."
- You should not have to change variable names where provided, but you are free to if required for your implementation.

(a) Utils and pre-processing (5pt)

- Scale the image data between 0 and 1
- One-hot encode the target data
- Convert the data to tensors and move them to the required device

(b) Implement Multi-Class Logistic Regression from scratch (15pt)

In this problem, we will apply multiclass logistic regression from scratch trained using gradient descent (GD) with Mean Squared Error (MSE) loss as the objective.

We will be using a linear model $y^{(i)} = W\mathbf{x}^{(i)}$, with the following notations:

- Weight matrix: $W_{p\times n} = \begin{bmatrix} \leftarrow & \mathbf{w}_1^\top & \rightarrow \\ \leftarrow & \mathbf{w}_2^\top & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{w}_p^\top & \rightarrow \end{bmatrix}$, where p is the number of target classes
- data points: $\mathbf{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \mathbb{R}$, and $X = \begin{bmatrix} \uparrow & \uparrow & \cdots & \uparrow \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \downarrow & \downarrow & \cdots & \downarrow \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$,

  where m is the number of datapoints.

Note: Here we need to define the model prediction. The input matrix is $X_{n\times m}$ where m is the number of examples, and n is the number of features. The linear predictions can be given by: $Y = WX + b$ where W is a $p \times n$ weight matrix and b is a p size bias vector. p is the number of target classes.

#1. Define a function `linear_model`

- This function takes as input a weight matrix (`W`), bias vector (`b`), and input data matrix of size $m \times n$ (`XT`).
- This function should return the predictions $\hat{y}$, which is an $m \times p$ matrix. For each datapoint row which is a $1 \times p$ vector, the prediction can be seen as the scores that the model gives each target class for that datapoint

Note: The loss function that we would be using is the Mean Square Error (L2) Loss: $MSE = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$, where m is the number of examples, $\hat{y}^{(i)}$ is the predicted value of $x^{(i)}$ and $y^{(i)}$ is the ground truth of $x^{(i)}$.

#2. Define a function `mse_loss`

- This function takes as input prediction (`y_pred`) and ground-truth label (`y`), and returns the MSE loss.

#3. Define a function: `initializeWeightsAndBiases`

In this part, we will do some setup required for training (such as initializing weights and biases) and move everything to torch tensors.

- This function returns tuple `(W, b)` , where `W` is a randomly generated torch tensor of size `num_classes x num_features` , and `b` is a randomly generated torch vector of size `num_classes` .
- For both the tensors, set `requires_grad=True` in parameters.

#4 Training code

- Given below is a function: `train_linear_regression_model` that takes as inputs as max number of epochs (`max_epochs`), weights (`W`), biases (`b`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`).
- This function returns a tuple `(W,b,losses)` where `W,b` are the trained weights and biases respectively, and `losses` is a list of tuples of loss logged every 100^{th} epoch.
- You can go through this article for reference.

#5. Initialize parameters and train your own model

- Initialize weights and biases using the `initializeWeightsAndBiases` function that you defined earlier
- Train your model using function `train_linear_regression_model` defined above.
- Use full batch (set `batch_size=len(X_train)` for training (Gradient Descent).
- Also plot the graph of loss vs number of epochs (Recall that values for learning rate (`lr`) and tolerance (`tolerance`) are already defined above).

(c) Use PyTorch for training (10pt)

- In the previous part, we defined the model, loss, and even the gradient update step. We also had to manually set the gradients to zero.
- In this part, we will re-implement the linear model and see how we can directly use Pytorch to do all this for us in a few simple steps.

#1. Define the linear model using PyTorch

- Use the inbuilt PyTorch `torch.nn.Module` to define a model class.
- Use the `torch.nn.Linear` to define the linear layer of the model

#2. A general function for training a PyTorch model.

Define a general training function: `train_torch_model` .

- This function takes as input an initialized torch model (`model`), batch size (`batch_size`), initialized loss (`criterion`), max number of epochs (`max_epochs`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`).
- This function will return a tuple `(model, losses)` , where `model` is the trained model, and `losses` is a list of tuples of loss logged every 100^{th} epoch.

Note: You can reuse a lot of components from the `train_linear_regression_model` function from the previous section

You can go through this article for reference.

(d) Working with different model types (15pt)

- Now, we retrain the above model with `batch_size=64`
- Use Stochastic/Mini-batch Gradient Descent and keep everything else the same.
- Like before, we plot the graph between loss and number of epochs.

#1. MSE Loss and Gradient Descent (GD)

- Use `nn.MSELoss` .
- Use full batch for training (Gradient Descent).
- Also plot the graph of loss vs number of epochs.

#2. MSE Loss and Stochastic Gradient Descent (SGD)

- Use `nn.MSELoss` .
- Use `batch_size=64` for training
- Also plot the graph of loss vs number of epochs.

Cross-Entropy (CE) Loss with Linear Model

- Instead of using MSE Loss, we will use a much more natural loss function for the multi-class logistic regression task which is the Cross Entropy Loss.
- CE loss converts the model scores of each class into a probability.
- This model penalizes both choosing the wrong class as well as uncertainty (choosing the right class with low probability).

- And we will use the same linear model defined in **(c)**.

Note: The Cross Entropy Loss for multiclass classification is the mean of the negative log likelihood of the output logits after softmax:

$$L = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\underbrace{\underbrace{-y^{(i)} log \underbrace{\frac{e^{\hat{y}^{(i)}}}{\sum_{j=1}^{p} e^{\hat{y}^{(j)}}}}_{\text{Softmax}}}_{\text{LogSoftmax}}}_{\text{Negative Log Likelihood (NLL)}}}_{\text{Cross Entropy (CE) Loss}} ,$$

where $y^{(i)}$ is the ground truth, and $\hat{y}^{(k)}$ (also called as *logits*) represent the outputs of the last linear layer of the model.

#3. CE Loss and GD

- Instead of `nn.MSELoss`, train the linear model with `nn.CrossEntropyLoss`.
- Use **full-batch**.
- Also plot the graph between loss and number of epochs.

#4. CE Loss and SGD

- Use a different batch size, `batch_size=64` with the new loss and repeat the previous part #1.
- Also plot the graph of loss vs epochs.

Training a neural network model in PyTorch

- We will train a neural network in pytorch with two hidden layers of sizes 32 and 16 neurons.
- We will use non-linear ReLU activations to effectively make it a non-linear model.
- We will use this neural network model for multi-class classification with Cross Entropy Loss.

Note: The neural network model output can be represented mathematically as below: $\hat{y}^{(i)}_{10\times1} = W^{(3)}_{10\times16}\sigma(W^{(2)}_{16\times32}\sigma(W^{(1)}_{32\times64}\mathbf{x}^{(i)}_{64\times1} + \mathbf{b}^{(1)}_{32\times1}) + \mathbf{b}^{(2)}_{16\times1}) + \mathbf{b}^{(3)}_{10\times1}$, where σ represents ReLU activation, $W^{(i)}$ is the weight of the i^{th} linear layer, and $\mathbf{b}^{(i)}$ is the layer's bias. We use the subscript to denote the dimension for clarity.

#5. Define the 2 hidden-layer Neural Network (NN)

#6. NN with CE Loss and GD

- Use Cross Entropy Loss.
- Use full-batch and plot the graph of loss vs number of epochs.
- Note that you can re-use the training function `train_torch_model` (from part (b)).

#7. NN with CE Loss and SGD

- Re-train the above model with `batch_size=64`.
- Also plot the graph of loss vs epochs.

(e) Analyze the results (10pt)

In the above few examples, we performed several experiments with different batch size and loss functions. Now it's time to analyze our observations from the results.

Recall that we trained the following models in this tutorial:

1. Linear Model - Scratch + MSE Loss + Full Batch (GD)
2. Linear Model - PyTorch + MSE Loss + Full Batch
3. Linear Model - PyTorch + MSE Loss + Mini Batch (SGD)
4. Linear Model - PyTorch + CE Loss + Full Batch
5. Linear Model - PyTorch + CE Loss + Mini Batch
6. NN Model - PyTorch + CE Loss + Full Batch
7. NN Model - PyTorch + CE Loss + Mini Batch

#1.Analysis

Effect of using full vs. batch gradient descent:

Effect of using different loss strategy:

Effect of using linear vs. non-linear models:

Training time per epoch in different cases:

PA - Part II: Advanced Vision Models [Experiments with CIFAR10] (45pt)

Keywords: Multiclass Image Classification, ResNet, Vision Transformers, CLIP

CIFAR10

- The CIFAR10 database is a large database with images of objects like airplane, bird, cat, dog, etc.
- It contains 70,000 labeled images. Each datapoint is a 32×32 pixels RGB image.
- To provide a realistic problem, we will use the images at this standard resolution of 32×32.

Agenda:

- The PA is split into three parts, the first part dealing with miniature models which we will build from scratch, the second part dealing with modern architectures and the bonus third part dealing with training vision models robust to adversarial attacks.
- In this part, we will be moving towards more modern architectures and a more realistic problem, finetuning pretained model of different architectures on the CIFAR10 dataset and comparing its performance with models trained from scratch (initialized with random weights).
- We will also be evaluating CLIP, which uses an architecture which is a backbone or helper model in most modern LLMs for image analysis and even in image generation models.
- We will be using PyTorch and HuggingFace for loading and reusing complex model architecture.

Note:

- Hardware acceleration (GPU) is **highly recommended** for this part as we will be training comparitively larger models!
- A note on working with GPU:
  - Take care that whenever declaring new tensors, set `device=device` in parameters.
  - You can also move a declared torch tensor/model to device using `.to(device)`.
  - To move a torch model/tensor to cpu, use `.to('cpu')`
  - Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU).
- Run all the cells in order.
- Only **add your code** to cells marked with "TODO:" or with "..."
- You should not have to change variable names where provided, but you are free to if required for your implementation.

Pre-training and fine-tuning

- In this part, we want to use modern architectures, namely the **ResNet** and **Vision Transformer**, for classification on the CIFAR10 dataset.
- Instead of trying to train these modern architectures from random weights for our task, we can instead use **pre-trained** models that have already been trained on a large amount of data.
- We can then **fine-tune** these models for our specific tasks.
- ImageNet is one such large-scale dataset (~1.3 million images) that is widely used for pretraining neural network models.
- These pretrained models serve as good initializations for various image-related tasks.
- We compare fine-tuned models and models trained from random initialization to analyze the difference in performance of these approaches.

(a) Training framework for CIFAR10 (10pt)

- For the more challenging CIFAR10 dataset, it is more useful to track the training and testing set accuracies during training.
- However, unlike with MNIST and the simple linear models, it is not always possible to load the entire dataset and pass the full batch through the model to compute the accuracy.
- Hence, we need to compute the accuracy in a mini-batch manner in the function `get_accuracy_cifar10`.
  - It takes as input the `model`, a `dataloader` (either for training or test set), and the `batch_size`.
- Following this, implement the training function `train_torch_model_cifar10` (similar to MNIST) but compute and print the training and testing accuracy after every epoch.
  - It should take as input the `model`, `trainloader`, `testloader`, `batch_size`, a predefined `optimizer`, the loss function `criterion`, maximum number of epochs `max_epochs`, and `tolerance`.
- You should be able to use the `train_torch_model` code from Part I by adding support for dataloaders.

(b) ResNet (10pt)

#1. Training with random initialization

- Using the defined `train_torch_model_cifar10`, we will first train a model using the ResNet50 architecture from scratch.
- Hyperparameters:
  - Batch size of 100
  - SGD optimizer with learning rate 0.001 and momentum 0.9
  - CE Loss

    - Train the model for a maximum of 15 epochs.
- Use the `torchvision.models.resnet50` function to define the model and set the arguments such that pretrained weights are not used.
- The default `resnet50` model has 1000 output neurons (since ImageNet1k has 1000 classes) in the last layer while we need only 10 for CIFAR10.

We will also compute the training and testing set accuracies after every epoch and plot them.

#2. Training with pretrained initialization

- Using the defined `train_torch_model_cifar10`, we will now train a ResNet50 model initialized from an ImageNet pretrained model.
- We will use the same hyperparameters as the previous part #2.1.
    - Batch size of 100
    - SGD optimizer with learning rate 0.001 and momentum 0.9
    - CE Loss
    - Train the model for a maximum of 15 epochs.
- We will use the same `torchvision.models.resnet50` function to define the model, but set the arguments such that ImageNet1k-v2 pretrained weights are used for initialization.

We will also compute and plot the training and testing set accuracies after every epoch.

(c) Vision Transformer (https://arxiv.org/abs/2010.11929) (10pt)

- This architecture was proposed following the popularization of transformer models for language modeling.
- Vision Transformers can be considered a variation of transformers with image inputs.
- The images are divided as 16x16 patches (the size varies with different models) and a sequence of these patches is passed as the input to the transformer model.

First, we modify the shape of the images to be 224x224 as this is what the ViT model we are using expects.

#1. Training with random initialization

- Similarly, now we will use the `train_torch_model_cifar10` function to train a Classifier using the Vision Transformer Architecture
- Hyperparameters:
    - Batch size of 100
    - SGD optimizer with learning rate 0.001 and momentum 0.9
    - CE Loss
    - Train the model for a maximum of 7 epochs as training this model is slower than ResNet50.
- Use the `timm/efficientvit_m1.r224_in1k` model either using the `timm` library. We

choose this model due to its small number of parameters (3M) which make it easy to train and experiment with.
- By setting the `num_classes` argument, we can customise the number of required classes which is 10, in the case of CIFAR10.

We will also compute the training and testing set accuracies after every epoch and plot them.

#2. Training with pretrained initialization

- We will use the same `timm/efficientvit_m1.r224_in1k` model but we will use its pretrained weights on ImageNet using the `pretained` parameter.
- We will use the same hyperparameters as the previous part #3.1.
  - Batch size of 100
  - SGD optimizer with learning rate 0.001 and momentum 0.9
  - CE Loss
  - Train the model for a maximum of 7 epochs.

We will also compute and plot the training and testing set accuracies after every epoch.

(d) CLIP (Contrastive Language-Image Pretraining - Paper) (10pt)

- CLIP jointly trains a text-encoder and image-encoder on images and text captions.
- CLIP generates representations (vector embeddings) of both the text and image such that they lie in the same vector space.
- CLIP optimizes the embeddings such that the image and captions are closer in the vector space if they correspond to each other and farther if they do not.
- The image-encoder is a pluggable model - usually a ResNet or Vision Transformer. We will use both.
- For this section, we will not be doing any pretraining or finetuning, rather just loading and evaluating it on CIFAR10 test-set (zero-shot image classification).

Note: This is a simplified explanation. You can read the paper if you need more details. However, this is not essential for the assignment.

Note: We will use the `open_clip` package to run CLIP. You can refer the documentation at https://github.com/mlfoundations/open_clip

We define a new testset for CLIP since it uses PIL images as inputs and does not require tensor conversion

#1. Predict and accuracy function

- Write the `clip_predict` function that takes in the CLIP `model`, a list of processed `images`, tokenized `texts` and encodes the texts and images.

- The function should then use the encoded features to find and return the predicted label for each of the `images`.
- Complete the `clip_accuracy` function takes in the CLIP `model`, the CLIP image `processor` and text/caption `tokenizer` and torchvision image `dataset` as input
- The function processes the images and texts into vectors in batch-form and uses the `clip_predict` predictions to calculate accuracy over the `dataset`

#2. CLIP Model using Vision Transformer

- Use the `open_clip.create_model_and_transforms` function from the `open_clip` library to load the model and image processor.
- Pass the `ViT-B-32` as the model and `openai` as the pretrained weights to use.
- Use the `open_clip.get_tokenizer` with the same model name to get the tokenizer.

- Call the `clip_accuracy` function with appropriate parameters. Optionally, you can pass in custom images and captions to the `clip_predict` function to see interesting outputs.

#3. CLIP Model using Resnet50

- We are now going to use a ResNet50 based CLIP-model instead of Vision Transformer a
- Use the same code as in #2 except the model name should be `RN50`

- Call the `clip_predict` function with appropriate parameters. Optionally, you can pass in custom captions to see interesting outputs using CLIP.

(e) Analysis (5pt)

Effect of pretraining

Effect of architecture (ResNet v/s ViT v/s CLIP)

PA - Part III: Training a Robust Model - Optional/Bonus (10pt)

Keywords: Adversarial Robustness Training

About the dataset:
The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.
The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image.

Agenda:

- The PA is split into three parts, the first part dealing with miniature models which we will build from scratch, the second part dealing with modern architectures and the bonus third part dealing with training vision models robust to adversarial attacks.
- In this part, you will train a 2-hidden layer neural network which is robust to adversarial attacks.
- You will train models on adversarial examples generated using FGSM and PGD.

Note:

- Hardware acceleration (GPU) is recommended but not required for this part.
- A note on working with GPU:
  - Take care that whenever declaring new tensors, set `device=device` in parameters.
  - You can also move a declared torch tensor/model to device using `.to(device)`.
  - To move a torch model/tensor to cpu, use `.to('cpu')`
  - Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU).
- Run all the cells in order.
- Only **add your code** to cells marked with "TODO:" or with "..."
- You should not have to change variable names where provided, but you are free to if required for your implementation.

Adversarial training:

- To train robust models, the most intuitive strategy is to train on adversarial examples.
- The adversarial (robust) loss function is defined as: $\min_{\theta} \frac{1}{|S|} \sum_{x,y \in S} \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$, where θ are the learnable parameters, S is the set of training examples with x representing

the input example and y the ground truth label, h_θ is the score function (neural network model), δ is the attack perturbation, and ϵ is the attack budget.

- This is also known as the min-max loss function. The gradient descent step now becomes: $\theta := \theta - \frac{\alpha}{|B|} \sum_{x,y \in B} \nabla_\theta \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$,
  where B is the mini-batch and α is the learning rate.
- Now the question becomes how to solve the inner term: $\nabla_\theta \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$ of the gradient descent step.
- For this, we can use **Danskin's Theorem**, which states that to compute the (sub)gradient of a function containing a max term, we need to simply
  1. find the maximum and,
  2. compute the normal gradient evaluated at this point.
- This holds only when you have the exact maximum. Note that it is not possible to solve the inner maximization problem exactly (NP-hard). However, the better job we do of solving the inner maximization problem, the closer it seems that Danskin's theorem starts to hold. That is why we can re-use methods such as FGSM/PGD to find approximate worst case examples.
- In other words, we can perform the attack to find $\delta^* = \arg\max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$, and then compute this term at the perturbed image: $\nabla_\theta \ell(h_\theta(x+\delta^*), y)$.

In summary, we will create an adversarial example for each datapoint in the mini-batch and use the loss corresponding to these adversarial examples to compute the gradient.

We explore two attacks to get the perturbration - Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD)

- In the Fast Gradient Sign Method (FGSM), the perturbation δ on an input example (e.g. input image) X is given by $\epsilon \cdot \text{sign}(g)$.
  - Here, g is the gradient of the loss function $g := \nabla_\delta \ell(h_\theta(x+\delta), y)$.
  - ℓ is the loss function, more precisely `nn.CrossEntropyLoss`. In the first timestep, this value of δ is 0.
- For the Projected Gradient Descent (PGD) attack, you will create an adversarial example by iteratively performing **steepest descent** with a fixed step size α.
  - The update rule is: $\delta := P(\delta + \alpha\,\text{sign}(\nabla_\delta \ell(h_\theta(x+\delta), y)))$.
  - Here δ is the perturbation, θ are the frozen DNN parameters, x and y is the training example and its ground truth label respectively.
  - h_θ is the score function and ℓ denotes the loss function.
  - P denotes the projection onto a norm ball (l_∞, l_1, l_2, etc.) of interest. For l_∞ ball, this just means clamping the value of δ between $-\epsilon$ and ϵ.

(a) Setup (4pt)

- In this part you will create a few adversarial examples using FGSM and PGD attacks. Use an attack budget $\epsilon = 0.05$.

#1. Define the `fgsm` function

- Define a function `fgsm` which takes as input the neural network model (`model`), test examples (`X`), target labels (`y`), and the attack budget (`epsilon`).
- Return the value of the perturbation (δ) after one gradient descent step.

#2. Define the `pgd` function

- Instead of using FGSM, now use Projected Gradient Descent (PGD) with projection on l_∞ ball for the attack.
- Define a function `pgd` that takes as input the neural network model (`model`), training examples (`X`), target labels (`y`), step size (`alpha`), attack budget (`epsilon`), and number of iterations (`num_iter`).
- Return the perturbation (δ) after `num_iter` gradient descent steps.

#3. Define a 2 hidden-layer NN in PyTorch

- Create a 2-hidden-layer neural network model in PyTorch.
- The input should be the size of the flattened MNIST image, and output layer should be of size 10, which is the number of target labels.

- Each of the two hidden layers should be of size 1024 with ReLU activations between each subsequent layer except the last layer.

You can refer to the structure of `NN_Model` from Part 1 (d) #5

#4. Adversarial Training Framework

- Define a function `train_torch_model_adversarial`
  - which takes as input: a PyTorch model (`model`), batch size (`batch_size`), loss function (`criterion`), maximum number of epochs (`max_epochs`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`), adversarial strategy (`adversarial_strategy` : `None/'fgsm'/'pgd'`), and attack budget(`epsilon`).
  - Note: use an SGD optimizer with the given learning rate (`lr`) to update all the model parameters.
- If `adversarial_strategy` is `None` , don't train on adversarial examples.
- This function will return a tuple of `(model, losses)` , where `model` is the trained model, and `losses` are a list of tuple of loss logged every epoch.
- The only difference from the function `train_torch_model` that you wrote in Part 1 (c) #2 is that you find the adversarial noise using an attack (based on `adversarial strategy`) and add it to the input before training with it.

(b) Training and evaluation with different strategies (3 points)

#1. Standard, FGSM-based, and PGD-based training

- Train three models:
  - without adversarial training,
  - with adversarial training using `fgsm`
  - with adversarial training using `pgd` .
- Hyperparameters
  - Use attack budget `epsilon` of 0.05.
  - Use a batch-size 512
  - Train for 20 epochs with learning rate 10^{-2}, and early stopping tolerance of 10^{-6}.
  - For `pgd` , use a step-size `alpha=0.01` and number of iterations `num_iter=40` when training.

  Note: PGD implementation will be slow (1hr+) when using a non-GPU runtime.

#2. Measuring Standard Performance

- Compute and print the accuracy of each of the three trained models on the clean test dataset.

- You can implement a function similar to the `print_accuracies_torch` function from HW1-Q1 (but compute accuracy only for the test set).

#3. Measuring Adversarial Robustness

- Using the same test dataset, perform adversarial attack to compute robust accuracy for each of the three models. Report the robust accuracy of each of the three models for both:
  - FGSM attack
  - PGD attack
- Note: In total, you need to report 6 robust accuracies here (3 models * 2 attacks).
- To create PGD attack examples, use `alpha=0.01`, `num_iter=40`.

(c) Evaluate the robust trained models at different epsilon (1.5 points)

- Report the robust accuracy of each of the three models (standard-trained, FGSM-trained and PGD-trained from **(b)**) using FGSM and PGD attacks at `epsilon = [0, 0.01, 0.02, ..., 0.09, 0.1]`.
- Plot a (single) graph of robust accuracy vs. `epsilon` (i.e. six curves, since we have 3 models and 2 attacks).

(d) Analysis of results (1.5 points)

Describe and analyze the observations from the graph that you plotted in the previous question **(c)**.