1 Deep Learning

Lab Objective: *Deep Learning is a popular method for machine learning tasks that have large amounts of data, including image recognition, voice recognition, and natural language processing. In this lab, we use PyTorch to write a convolution neural net to classify images. We also look at one of the challenges of deep learning by performing an adversarial attack on our model.*

Intro to Neural Networks

The brain, a biological neural network, is composed of neurons that are connected, either chemically or electrically, via synapses. A synapse transfers a signal from one neuron to another until it reaches a target cell, which acts on the signal. For instance, when you see a ball coming towards you, it's a collection of synapses traveling through the brain that trigger your arm to catch it. Every movement of your body is the result of these connections, even the unconscious ones like breathing. Synapses also play a role in developing or losing memory, and are therefore instrumental in how a human learns.

An artificial neural network, simply called a neural net in this lab, takes this idea of neurons passing information through synapses to learn a pattern. The network is composed of layers of neurons, usually called nodes, that are connected. Each connection has a weight based on its importance, and information is passed through the network from one layer to the next. For example, in Figure 1.1, the yellow input is passed to the first layer, blue, then the green layer, and then to the final output layer. In the example of raising your hand to catch a ball, the only thing that you're aware of is the input, the incoming ball, and the output, the hand being raised. All of the synapses happen unconsciously. Similarly, the middle layers for an artificial neural net are hidden because they're not visible.

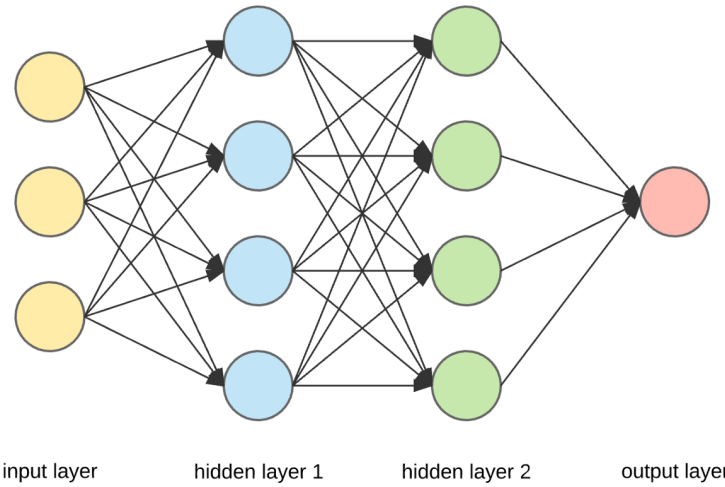

Figure 1.1: A high level diagram of an artificial neural network.

In a neural net, the input is often images, text, or sounds. This is transformed into tensors of real numbers. A tensor is a data structure similar to a numpy array and is compatible with GPUs. It has a shape and data type and can be multi-dimensional. The input tensors are sent to the first hidden layer and multiplied by the weights of their respective edges to get a measure of importance. An activation function takes the measures of importance and determines whether to pass the data onto the next layer via some threshold value. The activation functions are nonlinear, which allows the model to learn complex transformations between the input and output. This whole process is repeated many times, with the weights and thresholds being adjusted until the labels of the training data match the outputs.

Intro to PyTorch

PyTorch is an open source machine learning library. Developed by Facebook AI Research, it has two main capabilities, deep neural networks and GPU tensor computing. Deep neural networks, a neural net with many hidden layers, are possible through automatic differentiation, in which computers calculate derivatives automatically, accurately, and quickly, a necessary feature of computing the gradient with many parameters. A GPU can compute thousands of operations at once and is necessary for parallel computing. With the amount of data used in neural networks, GPUs significantly speed up the process. For more information and documentation on PyTorch, visit `https://pytorch.org/`

We will be working in Google's Colaboratory, `https://colab.research.google.com/notebooks/intro.ipynb`. Colab notebooks use Google's cloud servers, which have a built-in GPU. To enable the GPU in a Colab notebook, select the *Runtime* tab and then *Change runtime type*. This will open a popup called `Notebook Settings`. Under *Hardware Settings*, select GPU.

We can verify that GPU is enabled by the following code:

```
>>> import torch
>>> assert torch.cuda.is_available()
```

If the GPU is not enabled, there will be an Assertion Error when the assertion is run. This means that the a GPU is not available, and the code will be run on the CPU.

CUDA is a parallel computing platform developed by Nvidia for GPU computing. `torch.cuda` is the PyTorch package that supports CUDA and is what allows code to be run on the GPU. In order to run on the GPU, variables, including the data and the model must be stored on the GPU. We can set individual variables to be on the GPU by setting them directly using `variable.cuda()`. However, if the GPU is not available, this will cause an error. For flexibility, we create a `device` variable that allows the code to run on either the CPU or the GPU, depending on what is available. `cuda:0` means that the device running is the default GPU. If you a running on machine that has more than one GPU, you can set the device to be a specific GPU by changing the number appropriately. In Colab, you only need to use `cuda:0`. You can check which device a variable is on by displaying it. If it is on a GPU, it will list which number it is.

```
>>> x = torch.tensor([3., 4.])                  # Create tensor on CPU
>>> y = torch.tensor([1., 2.]).cuda()           # Create tensor on GPU

# Create the device, choosing GPU if available
>>> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
>>> z = torch.tensor([1., 2.]).to(device)       # Create tensor on device

>>> x
tensor([3., 4.])                                    # Check location of x (CPU)

>>> x = x.to(device)                                # Move x to GPU
>>> x                                           # Check location of x (GPU 0)
tensor([3., 4.], device='cuda:0')
```

ACHTUNG!

Cross-GPU operations are not allowed. This means that the model and data must all be on the same device. If the model is called on data that is on a different device, say the model is located on the CPU and the data is on the GPU, you will get the following runtime exception:

```
RuntimeError: Input type (torch.FloatTensor) and weight type (torch.cuda.↩
    FloatTensor) should be the same.
```

If you get this error, you will need to move one the variables so that they are all on the same device.

Data

For this lab, we will be using the CIFAR10 dataset, [Kri09]. It consists of $60,000$, 32×32 colored images: $50,000$ in the training set and $10,000$ in the test set. Each image is a $3 \times 32 \times 32$ matrix, with the 3 channels describing RGB color levels. The images are evenly split into ten classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, represented by the numbers $0 - 9$.

The CIFAR10 dataset is built into PyTorch's torchvision package, which makes getting the data simple. To use the data, we must first transform it to a PyTorch tensor. When using PyTorch, all of the data must be in tensor form so that it can be moved to a GPU. The next step is to normalize it.

While this is not necessary, it generally improves the result. We will normalize the values from $[0, 1]$ to $[-1, 1]$. Torchvision has a handy model called Transforms, which allows us to convert the data to a tensor and perform any other modification such as normalization, cropping, or rotating when loading the data.

To download the CIFAR10 set, call the dataset and pass in the location you want the data saved. There are three other important parameters. `train` indicates whether to get the training data or the test data. Torchvision conveniently splits this data. `download` indicates whether to download the data or not. If you have already downloaded the data in your current workspace, you can pass in `False`, but otherwise, you will need to download it. The `transform` parameter applies the given transform when loading the data. You should always have a transform that converts the data to tensors when loading a torchvision dataset.

The data can then be accessed using indexing. Each data point is a tuple consisting of the $3x32x32$ image and its class. You can also see the specs of the dataset by calling it without an index.

```
>>> from torchvision import datasets, transforms

# Normalize data and transform it into a tensor
>>> transform = transforms.Compose(
                [transforms.ToTensor(),
                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Download the CIFAR10 training data to ../data
>>> train_data = datasets.CIFAR10('../data', train=True, download=True, ↩
    transform=transform)

# Get the specs of the CIFAR10 training set
>>> train_data
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: ../data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
           )

# Get the first training data point
>>> train_data[0]
(tensor([[[-0.5373, ...,  0.1608],
          [-0.8745, ..., -0.0431],
          ...,
          [ 0.4118, ..., -0.3490],
          [ 0.3882, ..., -0.0353]],

         [[-0.5137, ..., -0.0275],
          [-0.8431, ..., -0.3176],
          ...,
          [ 0.0902, ..., -0.5843],
```

```
          [ 0.1294, ..., -0.2784]],

         [[-0.5059, ..., -0.1922],
          [-0.8431, ..., -0.5529],
          ...,
          [-0.2471, ..., -0.7333],
          [-0.0902, ..., -0.4353]]]), 6)

# Get the class of the first training data point
>>> train_data[0][1]
6
```

Once the data is loaded, PyTorch has a special class, DataLoader, that splits the data into batches for easy manipulation. Batches are small groups of data that are loaded onto the GPU individually before training. This allows us to run large amounts of data through our model while not running out of space on the GPU. A smaller batch size will improve accuracy but increase the number of iterations. Using larger batch sizes allows us to take advantage of GPUs, speeding up the training time. Depending on the size of the model, data, and the GPU, too large of a batch size can cause out of memory issues. Typical batch sizes are powers of 2: 32, 64, 128, 256.

Other parameters to the DataLoader include `shuffle`, defaulted to False, and `num_worker`, an integer which creates multiple processes. Accessing the data is done in two ways. To get a single batch of images, use the `iter` method. The other method is to loop through the batches in the loader, which we will do in the training loop later. The code so far demonstrates how to download and load the training set, as well as loop through the DataLoader for training. Loading the test set is done in the same way with `train=False`.

```
>>> from torch.utils.data import DataLoader

# Create a DataLoader from the shuffled training data
>>> train_loader = DataLoader(train_data,batch_size=36,shuffle=True)

# Get the 36 images of size 3x32x32 and labels in the first batch
>>> dataiter = iter(train_loader)
>>> images, labels = dataiter.next()
>>> images.size()
torch.Size([36, 3, 32, 32])

>>> images[0].size()
torch.Size([3, 32, 32])

>>> labels[0]
tensor(3)

# Loop through the data for training
>>> for batch, (x, y_truth) in enumerate(train_loader):
>>>       x, y_truth = x.to(device), y_truth.to(device)
```

Problem 1. Set the device as indicated above. Download the CIFAR10 training and test datasets. Transform them into tensors, normalize them as described above in the code, and create DataLoaders for each one. For the training set, use a batch size of 32, and for the test set, use a batch size of 1.

Convolution Neural Network

A convolution neural network, CNN, is a neural net that includes sequences of convolution layers, activation functions, and pooling layers. A convolution layer takes a two-dimensional array of weights called a kernel (sometimes called a filter) and multiplies it by the input at all possible locations, sliding around the input. This allows the model to retain some information from the model, hopefully important information, and ignore the rest. Consider the following two-dimensional input image of size 5×5 and 3×3 kernel.

2	4	7	6	2
9	7	1	2	1
8	3	4	5	8
4	3	3	1	2
5	2	1	5	3

5×5 Input Image

1	0	-1
1	0	-1
1	0	-1

3×3 Kernel

The stride of a convolution is how much the kernel slides as it passes over the input. If it slides one spot over, there will be 9 different submatrices inside the input image that match the kernel size. A stride of 2 means that on the first row, we'd only multiply the kernel by the image twice, once for each of the following submatrices.

2	4	7
9	7	1
8	3	4

Top left hand 3×3 square

7	6	2
1	2	1
4	5	8

Top right hand 3×3 square

Notice that as the kernel slides around the image, the inside values are used in more multiplication than the outside value, causing us to lose information, especially about the corners. If we want to keep more infomation about the edges of the image, or if we want the output to be the same size as the input, we introduce padding. Padding refers to a border added around the input. It is usually filled in with zeros.

2	4	7	6	2
9	7	1	2	1
8	3	4	5	8
4	3	3	1	2
5	2	1	5	3

5×5 input image

0	0	0	0	0	0	0
0	2	4	7	6	2	0
0	9	7	1	2	1	0
0	8	3	4	5	8	0
0	4	3	3	1	2	0
0	5	2	1	5	3	0
0	0	0	0	0	0	0

5×5 input image padded with 0

The size of the output for a convolution layer is calculated as follows:

(input size - kernel size + 2× Padding size)/stride + 1

In our example with stride 1 and no padding, the output size is $(5-3)+1=3$. To get each value in the output, the kernel is multiplied element-wise by every $3x3$ square inside the input and summed. For the top left square, the output is

$$2\times 1+4\times 0+7\times(-1)+9\times 1+7\times 0+1\times(-1)+8\times 1+3\times 0+4\times(-1)=7.$$

The 7 represents a feature of the 3×3 block in the top left corner. In an image, these are things like lines, curves, and colors, or even objects like a nose.

2× 1	4× 0	7× (-1)	6	2
9× 1	7×0	1× (-1)	2	1
8× 1	3× 0	4× (-1)	5	8
4	3	3	1	2
5	2	1	5	3

5×5 Input Images

7		

3×3 Kernel

After each convolution layer, we need to determine which features are important enough to pass to the next layer. We do this with an activation function. As mentioned earlier, activation functions are nonliner functions. Convolutions are linear, so the nonlinear activation functions are needed so that the model can learn nonlinear relationships. ReLU is a common activation layer in which any features with negative values are changed to 0. Since ReLU only decides whether or not to keep a feature and doesn't perform any multiplication or addition, it does not change the shape of its input. In practice, it converges faster and is more computationally efficient than other activation functions.

ReLU(x) = max(0,x)

Pooling layers are used to reduce the size of the data while retaining some information. Often, pooling layers reduce the data by 2, halving it. Max pooling does this by taking the maximum of a block while average pooling takes the mean of the values in the block. In PyTorch, the pooling functions take in a `kernel_size` parameter which is the size of the block, either an integer if the block is square or a tuple of the length and width. This size is usually 2 and is not related to the kernel size in a convolution layer.

4	7	6	2
7	1	2	1
3	3	1	2
2	1	5	3

4×4 Input Image

7	6
3	5

2×2 output after max pooling

The final layer in a CNN is a fully connected layer that computes the final score for each class. It takes the output from the previous layer, representing high level features and decides which features are most important for each class. It then predicts which class of the image based on the amount and type of features in the image. The final layer has the same number of outputs as the number of classes; with the CIFAR10 data, this is 10. The index of the largest entry will be the predicted class.

1.0.1 CNN in PyTorch

PyTorch has several classes and modules that contain many of the functions needed for deep learning. `torch.nn` contains the base class for all network models, `nn.Module`[1]. This is where the layers of a deep learning algorithm are defined. Each module has a forward method that returns the output, or the predicted y-values, on the input. Calling the module runs the `forward` method so `forward()` does not need to be specifically called. Note that `model` is a variable, so it should be moved to the GPU using `to(device)` once it is instantiated. We demonstrate how to create a basic convolution model below.

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F

class ConvolutionModel(nn.Module):

    def __init__(self):
        super(ConvolutionModel, self).__init__()

        # Initialize the layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5) #6↩
            x28x28
        self.relu = nn.ReLU()# 6x28x28
        self.maxpool = nn.MaxPool2d(kernel_size=2)    # 6x14x14
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=10, kernel_size=3)  ↩
            # 10x12x12
        self.flatten = nn.Flatten() # 1x(10*12*12)
        self.linear = nn.Linear(in_features=10 * 12 * 12, out_features=10) # 1↩
            x10

    # Call the layers on an image
    def forward(self, x):
        output = self.relu(self.conv1(x))
        output = self.maxpool(output)
```

[1]For all of the layers, activation functions, and loss function, see `https://pytorch.org/docs/stable/nn.html`

```
        output = self.relu(self.conv2(output))
        output = self.flatten(output)
        return self.linear(output)
```

The above example describes a CNN with one convolution layer as the first layer. `nn.Conv2d` applies 2D convolutions over several channels of input. The images in the CIFAR10 dataset are $3 \times 32 \times 32$ so we have 3 channels of 32×32 tensors. The `in_channels` is this number of channels. The `out_channels` is the number of channels we want to output and typically increases in size. The convolution layer is always followed by the activation function. In this model, we apply a max pool layer to cut the size of the data in half. In order to use the fully connected linear layer at the end, we use `nn.Flatten` to flatten the tensor to one-dimension. We then apply a second convolution layer. Its input is the output of the maxpool layer. The parameters for the `Linear` layer are the input size and output size respectively. The sizes after applying each layer are indicated in the comments to help see how the data changes. Each layer takes as input the previous layer's output, creating a chain that passes the data through the network. Notice that this model cannot be used on images of different sizes because the layers have specific input and output sizes.

The model accepts an $n \times 3 \times 32 \times 32$ tensor, where n is the number of images in our batch. If the model is run on only one image, it must be converted from a $3 \times 32 \times 32$ tensor into a $1 \times 3 \times 32 \times 32$ by using `unsqueeze(i)`. The integer in `unsqueeze` is the dimension to expand.

```
# Instantiate model
>>> model = Model()

# Set model to the GPU if available
>>> model = model.to(device)

# Apply the model to a single image
>>> m = model(images[0].unsqueeze(0))
tensor([[0.0000, 0.0000, 0.0000, 1.1688, 0.1514, 0.4041, 0.0000, 2.1056, ↩
    0.0000,
         2.3942]], device='cuda:0', grad_fn=<ReluBackward0>)

# Since the maximum value, 2.39, occurs in the final index, the predicted class↩
     is 10.
```

Parameters

Often we want to know how many learnable parameters, or weights, our model has. This number of parameters dictates how much space we need to able to train the model, and gives us an idea of the size of the model. We can calculate the number of parameters in each convolution layer with equation 1.1 where the kernel size is $(m \times n)$. Using our example above, $m = n = 3$. l is the number of channels in the previous layer. In PyTorch, this in the `in_channels` number. k is the number of channels that layer is outputting, or the `out_channels`

$$\text{number of parameters in convolution layer} = ((l * m * n) + 1) * k) \tag{1.1}$$

The MaxPool2d does not have any parameters since it does not have weights; there is no multiplication.

To calculate the parameters in the linear layer, use

$$(\text{number of inputs } +1) \times \text{number of outputs.}$$

Each layer in the ConvolutionModel above has the following parameters, giving a total of $15,416$ parameters.

Layer	Number of Parameters
self.conv1	((3*5*5*)+1)*6 = 456
self.maxpool	0
self.conv2	((6*3*3)+1)*10 = 550
self.linear	(10x12x12+1)*10 = 14410

Problem 2. Create a convolution model class convolves an image of size $3 \times 32 \times 32$ into a 1D tensor that represents the 10 classes. The model should have at least three convolution layers, each followed by an activation function, and include at least two linear layers and one maxpool layer. Select channels and kernel sizes so that you get at least $50,000$ parameters.

Calculate and print the number of learnable parameters in your model.

Initialize your model and move it to the device. Run the model on a single image to make sure there are no errors.

Hint: When creating your model, it's helpful to keep track of the shape of the image, `image.shape`, in each step.

Training

Now that we have data and a model, we need to train the model on the data. We do this by iterating through the DataLoader, calling the model, and optimizing the parameters. To optimize, we use a loss function to calculate the difference between the model's prediction and the actual classification, called label. A common classification loss function is Cross Entropy Loss, Eq 1.2. For each data point i, it calculates the log of the Softmax probability p_i and multiplies by the label. PyTorch's `nn.CrossEntropyLoss()` handles all of this.

$$L_{CE} = -\sum_{i}^{n} a_i log(p_i) \tag{1.2}$$

Once the loss is calculated, we can use it to determine how to change the weights to make the loss smaller. This is done by calculating partial derivatives of the loss function with respect to each weight. Then gradient descent is performed to update the weights. This process of calculating loss and performing gradient descent to update weights is called backpropagation. PyTorch has several predefined options that do this for you, and we'll use the popular Adam algorithm. PyTorch accumulates gradients when doing backpropagation. There are situations where this is good, but in our case, it would cause the loss to increase. To prevent this, we need to zero out the gradients before we perform backpropagation.

```
>>> objective = nn.CrossEntropyLoss()
>>> optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

```
# The following steps should be performed each iteration.
>>> optimizer.zero_grad()             # Zero out the the gradients
>>> outputs = model(inputs)           # Run the model
>>> loss = objective(outputs, labels) # Calculate loss
>>> loss.backward()                   # Compute gradients
>>> optimizer.step()                  # Optimize and update the weights
```

To improve accuracy, we loop through our data multiple times. Each of these loops is called an epoch. Large neural nets are often trained for many epochs. A good guideline to how many epochs to run is to stop training once the loss is no longer decreasing.

TQDM is a python package that shows a progress meter inside loops. Initialize tqdm outside of the loop, then update it inside the loop. This will display a progress bar showing the number of iterations. When running many iterations, tqdm can be helpful to estimate the time remaining.

```
>>> from tqdm import tqdm
>>> loop = tqdm(total=len(train_loader), position=0)

>>> for i in range(10):
>>>     loop.set_description('loss:{:.4f}'.format(loss.item()))
>>>     loop.update()

loss:1.8585: : 1402it [00:17, 79.69it/s]
```

Problem 3. Train the model by looping through the training data. Inside the loop, you should

1. Zero out the gradients.
2. Run the model on the inputs.
3. Calculate the loss on the model output and the actual label.
4. Backpropogate the error.
5. Optimize.

Run the loop for 10 epochs. At the end of each epoch, calculate the mean loss of the training data for that epoch. Then calculate the accuracy of the model on the test data. Since the model is no longer training, it needs to be set to evaluation mode using `model.eval()`. To resume training at the beginning of the next epoch, set the model to training mode using `model.train()`.

You should have around 50% accuracy at the end of 10 epochs.

Plot the epochs v. mean training loss each epoch and the epochs v. accuracy. Examples of the plots are in Figure 1.2.

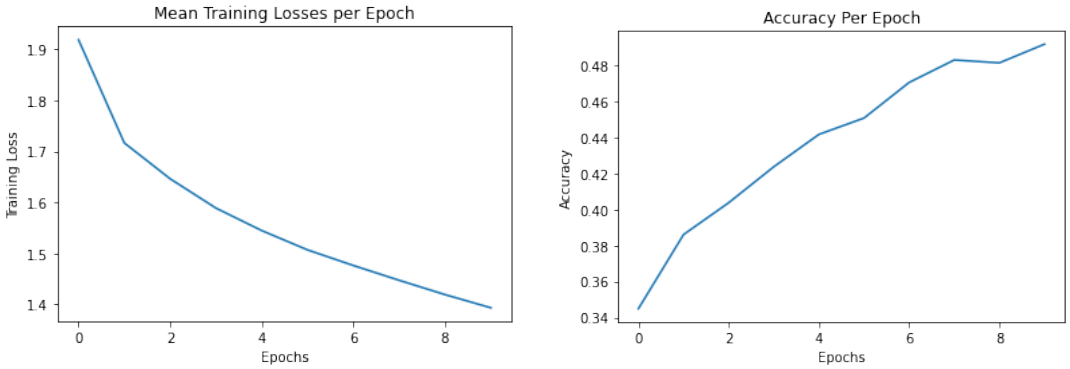

Figure 1.2: Training Loss and Accuracy for a CNN on CIFAR10.

Adversarial Networks

Just like any algorithm or software, deep learning is susceptible to attacks. When designing a machine learning model, accounting for security vulnerabilities is as important as speed and accuracy. Examples of attacks, or adversarial networks, range from adding a small amount of noise to a picture of a pandas, resulting in a gibbon classification [GSS15] to fooling facial recognition by printing a pair of eyeglasses [GKB17].

In the famous panda example mentioned above, some researchers found that by adding a small amount of noise to a picture of a panda, the algorithm incorrectly identified it as a Gibbon, with 99% accuracy!

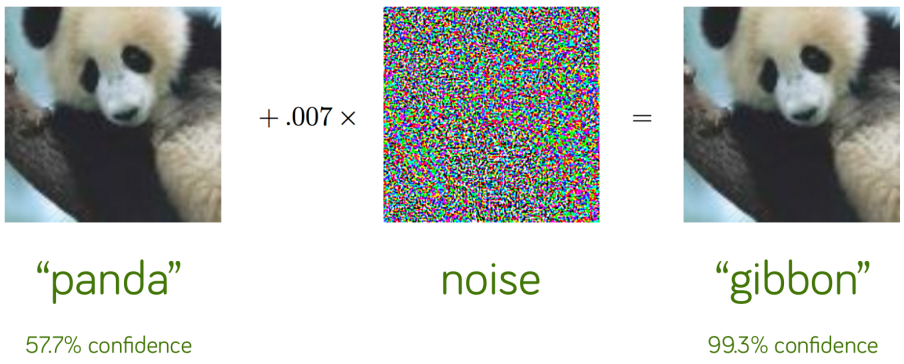

Figure 1.3: Panda: 57.7% confidence

This is done via Fast Gradient Sign Attack, FGSM. FGSM is a white-box attack, meaning that the attacker has access to the model. During model training, gradients are used to adjust the model weights so that loss is minimized. In FGSM however, the input data uses the gradients to maximize loss and perturb the image slightly, using the following equation.

$$perturbed_x = x + \varepsilon \times (\bigtriangledown_x Loss(\theta, x, y))$$

where x is the input, y is the label, and θ is the model parameters.

We can perform this perturbation in PyTorch using the built-in gradients `x.grad.data`. The following function `fgsm_attack`, takes in an image, or batch of images, a perturbation amount ε, and the gradient data. It then creates a modified image by adjusting each pixel slightly in the direction of the gradient. Since we normalized the images to be between $[-1, 1]$, we clamp the final image to stay in that range.

```
# FGSM attack code
def fgsm_attack(image, epsilon, data_grad):
    # Collect the element-wise sign of the data gradient
    sign_data_grad = data_grad.sign()

    # Create the perturbed image by adjusting each pixel of the input image
    perturbed_image = image + epsilon*sign_data_grad

    # Return the perturbed image adding clipping to maintain [-1,1] range
    return torch.clamp(perturbed_image, -1, 1)

perturbed_data = fgsm_attack(data, epsilon, data_grad)
output = model(perturbed_data)
```

Using the `fgsm_attack` function, we perturb the data that has been classified correctly. The process is described in Algorithm 1.1.

Algorithm 1.1 Advesarial Attack. This algorithm accepts a trained neural network model, data, and perturbation value ε. It iterates through each data point and if the model is correct, modifies the data based on ε. The algorithm returns the accuracy of the model after the attack.

procedure ADVESARIAL ATTACK(*model*, *data*$_l$*oader*, ε)
  for x,y **in** test_loader **do** ▷ Loop through test data
    x. requires_grad ← **True**
    output ← model(x)
    if output ! = y **then** ▷ Only modify correct images
      continue
    Update loss
    Zero out gradients
    Backwards step
    data_grad ← x.grad.data
    perturbed_data ← fgsm_attack(data, ε, data_grad)
    output ← model(perturbed_data)
  calculate accuracy

The algorithm is similar to the training loop done in Problem 3. However, before calling the model on the input, we need to set `data.requires_grad` attribute to `True` so that we can calculate `data.grad.data.sign()` in `fgsm_attack`. If the model is incorrect, we skip the rest of the steps. If the model is correct, we perform the attack to modify the image slightly, with the goal of tricking the model into predicting an incorrect label. The accuracy of the model is the percentage of modified images that still match their labels over the total number of images. Notice that since we are evaluating the test set, the model should be set to `eval()`, and we do not need to optimize.

Problem 4. Write a function that loops through the test data, modifying the images as described in Algorithm 1.1, using your trained model from Problem 3.

Run your function for each epsilon in $[0, .05, .1, .15, .2, .25, .3]$, and plot epsilon v. accuracy.

Display the perturbed version of the first image in the test data for each epsilon, using the following code. Your figure should look similar to Figure 1.4.

```
# Move the image to cpu and convert to numpy array
>>> perturbed_data.squeeze().detach().cpu().numpy()

# Plot the image
>>> img = ex/ 2 + 0.5     # unnormalize
>>> plt.imshow(np.transpose(img, (1, 2, 0)))
```

Figure 1.4: The first modified image for different εs.

Additional Materials

TensorBoard

TensorBoard is a visualization toolkit. Originally built for Tensorflow, TensorBoard can now be used with PyTorch. Some of the big features of TensorBoard include visualizing the model, dimenionality reduction, tracking and visualizing metrics, and displaying data.

To create a tensorboard, run the following code.

```
>>> %load_ext tensorboard
>>> logs_base_dir = "runs"
>>> os.makedirs(logs_base_dir, exist_ok=True)
>>> %tensorboard --logdir {logs_base_dir}
```

The TensorBoard homepage will show up inline.

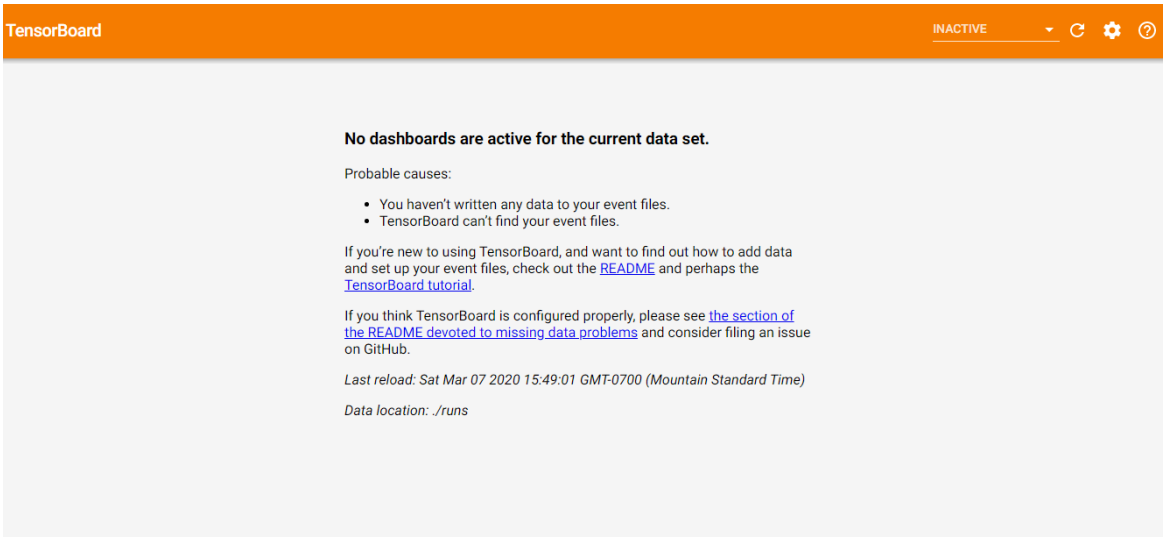

Figure 1.5: The home page of an empty TensorBoard.

We write to TensorBoard using `SummaryWriter`. It writes to files in the `logs_base_dir` that are used by TensorBoard to display information. You can view the `logs_base_dir` directory by selecting the file icon on the far left of the page. For example, we can create an interactive graph of our model.

```
>>> tb = SummaryWriter()
>>> tb.add_images("Image", images)
>>> tb.add_graph(model, images)
>>> tb.close()
```

This updates our TensorBoard with a `GRAPHS` tab, which describes the model. If it doesn't show up automatically, press the refresh button in the top right corner of the TensorBoard. You can explore the model by clicking on the components.

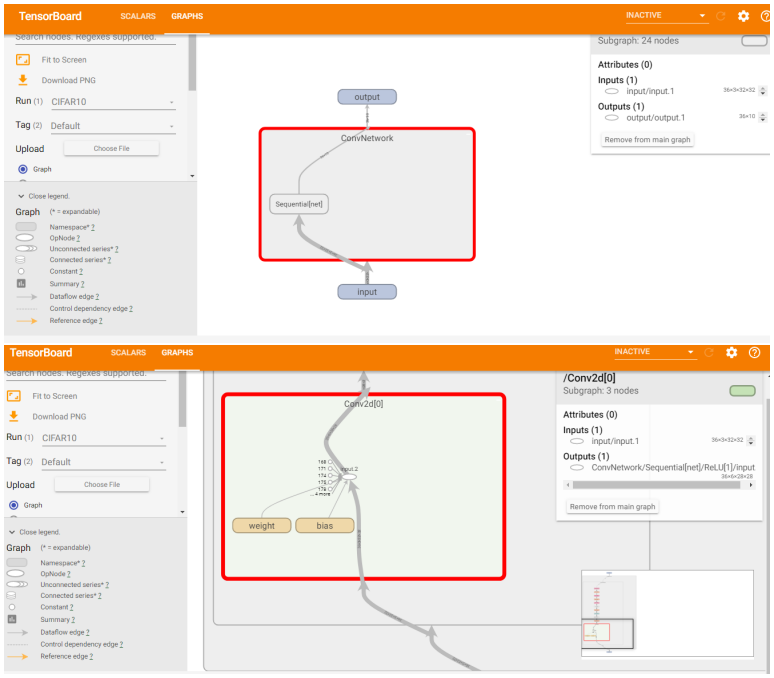

Figure 1.6: Examples of TensorBoard Graph Tab.

The following items can be added to TensorBoard, with more information at `https://pytorch.org/docs/stable/tensorboard.html`.

- `add_scalar/s`
- `add_image/s`
- `add_figure`
- `add_text`
- `add_graph`
- `add_hparams`

To save the training loss, write a function that returns a matplotlib figure of the training loss plot. Then use `tb.add_figure(figure_name, plot_loss())`.

```
writer.add_figure('Training Loss',plot_loss())
```

Problem 5. Create a TensorBoard for this project that includes the network, a plot of iterations versus training loss and a plot of iterations versus test accuracy from the training done in Problem 3.

Bibliography

[GKB17] Ian J. Goodfellow, Alexey Kurakin, and Samy Bengio. Adversarial examples in the physical world. 2017. [12]

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. 2015. [12]

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. [3]