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

An *artificial neural network* is a machine learning tool inspired by the idea of neurons passing information between each other to learn. The network is composed of layers of neurons, usually called *nodes*, that are connected in various ways. Each connection has a *weight* based on its importance, which is used as 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.

The middle layers for a neural network are considered "hidden" because they're not directly viewable to the outside world. You can view them but they will be a mess of seemingly random numbers, not the helpful classification labels you would get from an end layer.

In a neural network, the input is often images, text, or sounds represented as a vector of real numbers. In order to evaluate a network, these values are then multiplied by the weight of each pertinent edge, and all respective values are added up to create the node value. A vector called the *bias* of the layer is added to each respective node, finalizing the linear combination step. An *activation function* takes these node values and applies a nonlinear transformation to them, which allows the model to learn complex nonlinear transformations between the input and output. Without these activation functions, the network would be linear and exactly the same as a network with no hidden layers. Mathematically, a typical neural network looks like the nested function composition

$$f_N(\mathbf{x}) = \mathbf{a}(W_k\mathbf{a}(\ldots\mathbf{a}(W_2\mathbf{a}(W_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2)\ldots)+\mathbf{b}_k)$$

where $\mathbf{a}$ is the activation function, and W_i and $\mathbf{b}_i$ are the weights and biases of each layer of the neural network. The model is trained by adjusting the weights and biases, typically using a variant of gradient descent, until the model output accurately matches the training labels.

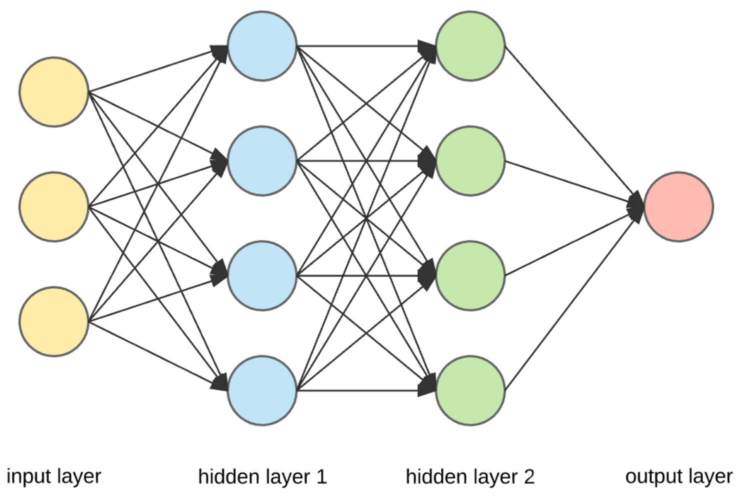


Figure 1.1: A high level diagram of an artificial neural network.

Intro to PyTorch

PyTorch is an open source machine learning library developed by Facebook AI Research. It's mainly used for fast GPU processing of deep neural networks (neural networks with many hidden layers). GPUs (graphics processing units) are designed to compute thousands of operations at once and are vital for parallelizing the operations used by neural networks, which is why they are used here. 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 calling `torch.cuda.is_available()`. If this function returns `False`, a GPU is not available, and the code will be run on the CPU.

CUDA is a parallel computing platform for GPU computing. The PyTorch package `torch.cuda` interfaces with this package and allows code to be run on the GPU.

PyTorch represents vectors and arrays using *tensors*. A tensor is a data structure similar to a numpy array that is designed to be compatible with GPUs. Like a numpy array, it has a shape, data type, and can be multi-dimensional.

In order for a tensor to be used by the GPU, it must be stored on the GPU. A tensor can be sent directly to the GPU using `variable.cuda()`; however, if the GPU is not available, this will cause an error. A more flexible approach is store which deivce we are doing computations on as a variable `device`. We will prefer to use a GPU if available, but will use the CPU if a GPU is not available. Then, we can send our variables to the correct location in both cases by using `variable.to(device)`:

```
>>> import torch

>>> 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
```

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. `cuda:0` means that the device running is the default GPU. If you are using a machine that has multiple GPUs, you can set the device to be a specific GPU by changing the number. In Colab, only `cuda:0` is available.

```
>>> 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 GPU and the data is on the CPU, 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. It consists of $60,000$ images of size 32×32, represented as a $3 \times 32 \times 32$ matrix, where the 3 channels describe the colors using RGB. The images are evenly split into ten classes represented by the numbers $0 - 9$: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

For convenience, the dataset is already split into a training and a testing set; however, we will also add a *validation* set.

Using a train-validate-test split is good practice in general, because usually we will want to test and compare different models and hyperparameter choices iteratively until we arrive at one that works well for our problem. Once we are done training, we would like to use the test set to determine how well our model fits the data. However, if we use the test set to compare how well each of our models performs, it effectively becomes a second train set that we are learning by trying different models, and using the test set to determine how well our model works on the whole dataset is no longer really valid. As such, it is better practice to use a three-way split. We train each model on the train set, use the validation set to compare the models, and use the test set to determine if our final model appears to fit the data well.

The CIFAR10 dataset is split into a train set of 50,000 images and a test set of 10,000 images. We will split the original train set into a new train set of 40,000 images and a validation set of

10,000 images. To use the data, we must transform it into PyTorch tensors. We also will normalize the data, as this generally improves the results. We will normalize the values to have mean 0 and standard deviation 1 for each component. Finally, we will split the dataset and place it inside a `torch.utils.data.DataLoader` class for easier manipulation.

To load the dataset, we use the `torchvision.datasets.CIFAR10` function, which accepts a folder for the data to be stored in. Some important keyword arguments are listed in Table 1.1.

Parameter	Explanation
`train`	Whether to get the training data or the test data.
`download`	Whether to download the data. You usually only need this the first time you access the dataset. Note that restarting Google Colab will require re-downloading the data, however.
`transform`	Applies the given `transform` when loading the data. This transform always should convert the data into a PyTorch tensor.

Table 1.1: Parameters of the `datasets.CIFAR10` loading function

We can use the `transform` parameter in particular to easily normalize our data. PyTorch has a module `torchvision.transforms` to make creating these transformations easier. In this case, we want to use `transforms.ToTensor` to convert the data into tensors, and then `transforms.Normalize` to normalize the data. The `Normalize` object accepts the desired mean and standard deviation after normalization. We can use `transforms.Compose` to combine these together into a single transform object:

```
>>> from torchvision import transforms

# Transform data into a tensor and normalize
>>> transform = transforms.Compose([
...         transforms.ToTensor(),
...         transforms.Normalize((0.0, 0.0, 0.0), (1.0, 1.0, 1.0))
...])
```

We can then load the data:

```
>>> from torchvision import datasets

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

The data can then be accessed using indexing. Each data point is a tuple consisting of the $3 \times 32 \times 32$ image and its class. You can also see the specs of the dataset by calling it without an index.

```
# Get the first training data point
>>> train_data[0]
(tensor([[[ 0.2314, ..., 0.5804],
          [ 0.0627, ..., 0.4784],
          ...,
          [ 0.7059, ..., 0.3255],
          [ 0.6941, ..., 0.4824]],

         [[ 0.2431, ..., 0.4863],
          [ 0.0784, ..., 0.3412],
          ...,
          [ 0.5451, ..., 0.2078],
          [ 0.5647, ..., 0.3608]],

         [[ 0.2471, ..., 0.4039],
          [ 0.0784, ..., 0.2235],
          ...,
          [ 0.3765, ..., 0.1333],
          [ 0.4549, ..., 0.2824]]]), 6)

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

# 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=(0.0, 0.0, 0.0), std=(1.0, 1.0, 1.0))
           )
```

Problem 1. Create the `device` variable as indicated above. Download the CIFAR10 training and test datasets, transform them into tensors, and normalize them as described above.

PyTorch has a special class `DataLoader` that splits the data into batches for easy manipulation. Sending individual data points to the GPU one at a time to be processed by our model is very inefficient, as it makes it impossible for the GPU to parallelize the computations. Instead, we use *batches*, and send multiple data points together. Using larger batch sizes allows us to take advantage of GPUs, speeding up the training time. Storing all of the data on the GPU is, however, generally impossible due to memory constraints. Using too large of a batch size will cause out of memory issues, and tends to reduce the effectiveness of training. Typical batch sizes are powers of 2: 32, 64, 128, 256.

The `DataLoader` class accepts the dataset as its first argument. The dataset can be a dataset object like the one we created above, a list containing the data points, or any iterable. For the train set, we will first split the loaded data into two lists to create the actual train and validation sets. The dataset object does *not* support fancy indexing, so this step should be done with list comprehension:

```
>>> actual_train_data = [train_data[i] for i in range(40_000)]

>>> from torch.utils.data import DataLoader

# Create a DataLoader from the shuffled training data
>>> train_loader = DataLoader(actual_train_data, batch_size=36, shuffle=True)
# and similarly for the validation set
```

The data is not ordered by its classes, so directly indexing like this will put a good mixture of all of the classes into both sets. For the test set, we can just directly pass the dataset object into the `DataLoader`. Some other useful parameters of the `DataLoader` class are listed in Table 1.2.

Parameter	Explanation
`batch_size`	The size of batch to use
`shuffle`	Whether to shuffle the data
`num_worker`	The number of processes to use, in order to load the data in parallel

Table 1.2: Parameters of the `DataLoader` object

Once we have the data in the `DataLoader` class, we can iterate through it to get data points. We can turn it into an iterator using the `iter` method, and then get batches one-at-a-time using the `next` method:

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

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

>>> labels[0]
tensor(8)
```

This method is particularly useful if we just need a few images. We can also directly iterate through all of the images using a `for` loop:

```
>>> for batch, (x, y_truth) in enumerate(train_loader):
...     # Move to the GPU
...     x, y_truth = x.to(device), y_truth.to(device)
...     # ...
```

This will be a more convenient method for training.

Problem 2. Split the data into train, validate, and test sets, and create DataLoaders for each one. The train set should have 40,000 data points and the test and validate sets should each have 10,000 data points. Use a batch size of 32 for the training set and 1 for the validation and test sets. Specify `shuffle=True` for the training set, and `shuffle=False` for the validation and test sets (this is common practice in deep learning).

Neural Networks in PyTorch

Before creating a good model for this dataset, we will start with a simple model to illustrate how to set up a neural network in PyTorch. This model will use only fully-connected linear layers and activation functions. First, we need to import the `nn` module, which contains all of the classes we need for this:

```
from torch import nn
```

Simple layers

A linear layer takes an input vector x and outputs $Ax + b$ for a learned weight matrix A and bias vector b. This is implemented in Pytorch as `nn.Linear(in_features, out_features)`, where `in_features` is the length of the input vector and `out_features` is the desired length of the output vector. This is called a *fully-connected* layer, because every entry of A and b are allowed to be nonzero.

After each layer, we want to pass the values through an *activation function*. This allows the model to be nonlinear, allowing it to learn much more complicated behaviors than it would otherwise. The most commonly used activation function is the Rectified Linear Unit (ReLU) function:

$$\text{ReLu}(x) = \max(0, x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

This activation function avoids many issues that other activation functions have, and is used almost universally. For the final activation function, however, we will use a different activation function: the *softmax* function

$$\text{Softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}} \right).$$

The components of the output of the softmax function are all non-negative and sum to 1. This allows the output of the final layer to be interpreted as probabilities, which is useful for classification. The component with the highest probability will be the neural network's prediction for the input image. This also enables the use of cross-entropy as a very natural loss function, which will be discussed later. These two activation functions are available as `nn.ReLU` and `nn.Softmax`.

Creating a model

To create a neural network in PyTorch, we begin by creating a class that inherits from `nn.Module`:

```
class NNExample(nn.Module):
```

The class `nn.Module` handles internals so that training is simpler, as well as providing a variety of useful methods. In the initializer of our class, we need to call the *superconstructor* `super().__init__()` to initialize the `nn.Module` itself. Then, we initialize all of the layers we want to use in our model. For this example, we will use two fully-connected linear layers with activation functions after each. Since our inputs are $3 \times 32 \times 32$ tensors and linear layers only work with vectors, we will also include an `nn.Flatten` layer.

```
    def __init__(self):
        # Initialize nn.Module
        super().__init__()

        # Create our layers
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(in_features=3*32*32, out_features=100)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(in_features=100, out_features=10)
        self.softmax = nn.Softmax(dim=1)
```

We need to set all of these layers as members of our class in order for them to be properly detected in the training process. Notice that the input dimension of the first layer (`linear1`) is equal to the dimension of the (flattened) input image ($3 \times 32 \times 32$), but from there, the output dimension is chosen arbitrarily to be 100. The second layer (`linear2`) must then have input dimension equal to the output dimension of `linear1`, but its output dimension is chosen to be 10, which is the number of classes possible in the CIFAR10 dataset.

Lastly, we define the `forward()` method, which calls all of the layers on an input image to give us the output:

```
    def forward(self, x):
        x_flat = self.flatten(x)
        x_layer1 = self.relu(self.linear1(x_flat))
        output = self.softmax(self.linear2(x_layer1))
        return output
```

Even though each layer is really a class (note how we initialize them in `__init__()`), we can call them as if they are functions. Any layer that contains *learnable parameters* (for example, the weights present in linear layers), must be called individually in the `forward()` method, as otherwise this would force it to reuse the parameters and reduce training effectiveness. However, for layers that do not have learnable parameters, such as activation functions, we can safely reuse them and call them multiple times in the `forward()` method. Hence, `nn.ReLU()` only needs to be defined once in the `__init__()` method, even when it may be called multiple times in the `forward()` method.

This neural network would likely perform very poorly on the CIFAR10 dataset, however; only using fully-connected linear layers does not work well for images. For a better method, we will turn to *convolutional neural networks*.

Convolutional neural networks (CNNs) are a type of neural network that use *convolution layers*. They also commonly use *pooling layers* They are particularly well-suited to working with images, such as the CIFAR10 dataset. We now discuss these components and how to use them in PyTorch.

Convolution 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. It is particularly useful when working with images, as it preserves and extracts spacial structures, unlike fully-connected linear layers.

Consider the following 5×5 input image 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

To get each value in the output, the kernel is multiplied element-wise by $3x3$ squares inside the input and summed. For the top left square in this example, the output is

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

$2 \cdot 1$	$4 \cdot 0$	$7 \cdot (-1)$	6	2
$9 \cdot 1$	$7 \cdot 0$	$1 \cdot (-1)$	2	1
$8 \cdot 1$	$3 \cdot 0$	$4 \cdot (-1)$	5	8
4	3	3	1	2
5	2	1	5	3

5×5 Input Images

7	$\cdots$	
$\vdots$		

3×3 Output

The 7 represents a feature of the 3×3 block in the top left corner. With a trained network applied to the image, these features can represent things such lines, curves, and colors, or even more complicated objects like a nose.

The *stride* of a convolution is how much the kernel slides at a time as it passes over the input. With out example, if the kernel slides one spot over (has a stride of 1), there will be 9 submatrices inside the input image that will be used, and we would get a 3×3 matrix as output. If we used a stride of 2 instead, only the top-left, top-right, bottom-left, and bottom-right submatrices would be used.

Notice that as the kernel slides around the image, the inside values are used in more multiplications 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, we can use *padding.* Padding consists of adding a border around the input, usually filled with zeros. This can also allow the output of the layer to be the same size as the input.

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

Each dimension of the output for a convolution layer is calculated as follows:

$$\frac{\text{input size} - \text{kernel size} + 2 \cdot \text{padding size}}{\text{stride}} + 1$$

In our example with stride 1, kernel size 3, and no padding, the output size is $(5-3+2\cdot 0)/1+1=3$. It is good to ensure that the stride always divides the numerator, as otherwise the kernel will be applied asymetrically to the image. Calculating the output dimension of a layer is necessary since the following layer will have its *input* dimension size equal to the previous *output* dimension size.

One last feature that we need to discuss is *channels*. Image data (including ours) typically has three channels, representing the red, green, and blue contents of pixels. Channels act like different "layers" of the image or of the convolutional output. In a convolutional layer, there is one kernel for each pair of input channel and output channel. If we have n input and m output channels, then we will have mn total kernels that are learned individually. Each kernel is applied to its corresponding input channel as described above. Then, each output channel is determined as the sum of the results of the convolutions of all of its kernels, plus a learned bias value.

Convolutional layers are represented in PyTorch with `nn.Conv2d`. The constructor of this class requires three parameters `in_channels`, `out_channels`, and `kernel_size`. It also has optional arguments `stride` (default 1) and `padding` (default 0). Note that for each of the kernel size, stride, and padding, only an integer needs to be specified, and it will be used for both the x and y directions. The following creates a convolutional layer that accepts an image with 3 channels, output 8 channels, and uses a 3×3 kernel with stride 1 and no padding:

```
layer = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3)
```

Pooling layers

Pooling layers are used to reduce the size of the image while retaining important information. The input image is broken into small pieces, called *pools*, each of which is condensed to a single number. The most common form of pooling is *max pooling*, where the output of each pool is just the maximum of its inputs.

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

Layer type	Number of Parameters
Linear	$(\texttt{in\_features} + 1) * \texttt{out\_features}$
Convolutional	$\left(\texttt{in\_channels} \cdot \texttt{kernel\_size}^2 + 1\right) * \texttt{out\_channels}$
Pooling	No parameters
Flatten	No parameters
Activation functions	No parameters

Table 1.3: Parameter counts for layer types we use in this lab

Max pooling has the particularly nice property of making the output remain similar if the input image is shifted slightly.

Max pooling layers are represented in PyTorch as `nn.MaxPool2d`. They accept a single parameter `kernel_size`; this is the size of each of the pools. Using 2×2 pools is the most common. The following creates a pooling layer that uses a 2×2 pool size:

```
layer = nn.MaxPool2d(kernel_size=2)
```

Parameters

When working with neural networks, it can be useful to know how many learnable parameters our model has. This particularly dictates the amount of space needed to store the model. The number of parameters in each layer depends on the type of layer and its input and output sizes. Table 1.3 lists how to calculate this number for the layer types we have discussed.

For example, the example neural network above would have

$$\begin{aligned} &\text{(first linear layer)} && (3 \cdot 32 \cdot 32 + 1) \cdot 100 \\ &\text{(second linear layer)} && + (100 + 1) \cdot 10 = 308310 \text{ parameters,} \end{aligned}$$

and the example convolutional layer would have

$$(3 \cdot 3^2 + 1) \cdot 8 = 224 \text{ parameters.}$$

Problem 3. Create a class for a convolutional neural network that accepts images as $3 \times 32 \times 32$ tensors and returns 1D tensors of length 10, representing its predicted probabilities of each class. Include at least the following:

- Three convolutional layers, each followed by an activation function
- A max pooling layer
- Two linear layers

Be sure that your final activation function is the softmax function. Choose the size of the layers so that your model has at least 50,000 parameters (use Table 1.3), and print out this calculation in the Jupyter notebook file. In practice, specifications of model architecture (i.e. number of layers, layer sizes, etc.) are chosen quite arbitrarily until something works. As such, you may customize your model architecture to your liking, provided your model meets the requirements specified above.

Hint: It can be very helpful to keep track of the size of the image after each step, so you know what the input size should be for the next step. The max pooling layer should occur immediately after a convolutional layer is passed through an activation function. Additionally, you will need a `nn.Flatten` layer after the convolutional layers and before the linear layers. You can check that your model works correctly by passing an (unsqueezed) image through it as demonstrated in the following code:

```
>>> model = NNExample()
>>> model(images[0].unsqueeze(0))
tensor([[0.0952, 0.1120, 0.1019, 0.0992, 0.0955, 0.0984, 0.0886, 0.1326, ↩
    0.0966, 0.0800]], grad_fn=<SoftmaxBackward0>)
```

Note that your neural network will predict different probabilities for each of the categories.

Training the Model

Now that we have data and a model all set up, we need to train the model on the data. We do this by iterating through the DataLoader, calling the model on the data, determining how well the model classified the data, and then optimizing the model weights. We use a loss function, called the *objective*, to calculate the loss, which is the difference between the model's predicted labels and the actual labels of the data. A common classification loss function is Cross Entropy Loss

$$L_{CE} = -\sum_i a_i log(p_i),$$

where i represents each data point, p_i is the Softmax probability for each data point, and a_i is the label for each data point. PyTorch's `nn.CrossEntropyLoss()` conveniently handles all of this.

Once the loss is calculated by the objective, we can use it to optimize the model weights to make the loss smaller. This can be done through *backpropagation*, which calculates the partial derivatives of the loss function with respect to each weight, and then uses gradient descent to update every weight. PyTorch has several predefined methods for optimization, but we'll use the popular `Adam` algorithm. PyTorch accumulates gradients when backpropagating, which is sometimes desireable, but in our case it would cause the loss to increase. To prevent this, we need to zero out the gradients before we perform each backpropagation. PyTorch streamlines this entire training sequence in a very clean way, as shown in the following:

```
>>> import torch.optim as optim

# Define the objective and optimizer
>>> objective = nn.CrossEntropyLoss()
>>> optimizer = optim.Adam(model.parameters(), lr=1e-4)


# For each iteration of the DataLoader, do the following
>>> optimizer.zero_grad()             # Zero out the gradients
>>> y_hat = model(x)                  # Predict labels
>>> loss = objective(y_hat, y_truth)  # Calculate loss
>>> loss.backward()                   # Backpropagate to compute gradients
>>> optimizer.step()                  # Optimize and update the weights
```

An *epoch* is a complete training sequence that trains over the entire DataLoader. To improve the model's accuracy, we can train over many epochs. A good guideline is to train the model for the number of epochs it takes for the loss to stop decreasing.

The loss is calculated using the training data, but at the end of each epoch we also want to know how well the model performs with the validation data. Before we determine the validation accuracy, we need to switch our model to evaluation mode so it doesn't continue training. This is done by the simple command `model.eval()`, but it's important to switch the model back to training mode at the start of each epoch using `model.train()` The validation accuracy is determined by simply iterating through the validation DataLoader, and seeing if the model can correctly predict each data point. The validation accuracy is then computed by dividing the number of correct predictions by the total number of data points in the validation DataLoader.

```
>>> model.eval()                            # switch to evaluation mode
>>> validation_score = 0
>>> for x, y_truth in validation_loader:
>>>     x, y_truth = x.to(device), y_truth.to(device)
>>>     y_hat = model(x)
>>>     if y_truth == y_hat.argmax(1):  # compare with greatest probability
>>>         validation_score += 1
>>> validation_accuracy = validation_score / len(validation_loader)
```

The validation accuracy does not determine the model's final accuracy. The final accuracy is computed in the same way as the validation accuracy, but this time using the testing data, and it's only computed one time, when the model finishes training entirely.

TQDM is a python package that displays the progress of a for-loop, which can help estimate the remaining time. TQDM is initialized outside the loop, then updated inside the loop, as follows:

```
>>> from tqdm import tqdm

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

>>> for epoch in range(num_epochs):
>>>     loop.set_description('epoch:{}, loss:{:.4f}'.format(epoch,loss.item()))
>>>     loop.update()

>>> loop.close()

epoch:1 loss:1.8585: : 1402it [00:17, 79.69it/s]
```

Problem 4. Send your model to the device and instantiate the objective and optimizer. Train your model with a TQDM display, and calculate the Validation Accuracy after each epoch. Begin by initializing your TQDM loop, then for each epoch, do the following:

1. Set your model to training mode (`model.train()`)

2. Instantiate an empty `loss_list`
3. For each batch in `train_loader`:
   (a) Send `x` and `y_truth` to device
   (b) Zero out the gradients
   (c) Use model to predict labels of `x`
   (d) Calculate loss between predicted labels and `y_truth`
   (e) Append loss (`loss.item()`) to `loss_list` (the `.item()` feature extracts the element from a tensor with only one element)
   (f) Update TQDM loop
   (g) Backpropagate to compute gradients
   (h) Optimize and update the weights
4. Save the loss mean as the mean of the losses in `loss_list`
5. Set your model to evaluation mode (`model.eval()`)
6. Calculate and save validation accuracy

Finish the training by closing your TQDM loop. Train for 10 epochs, saving the mean loss and validation accuracy for each epoch. Plot the mean losses and validation accuracies, which should resemble Figure 1.2. Lastly, print the final test score using the testing data, as described above.

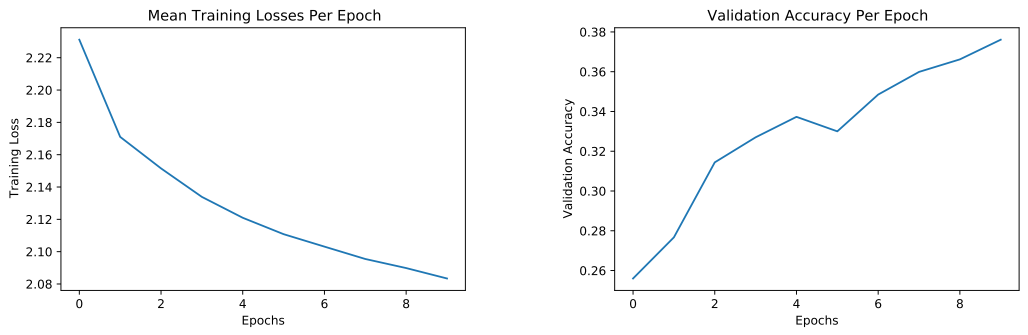


Figure 1.2: Training Loss and Validation Accuracy for a CNN on CIFAR10.

Adversarial Attacks

Just like any algorithm or software, deep learning is susceptible to attacks. For deep learning models, this vulnerability most often manifests in the model being extremely sensitive to certain types of changes in the input that really should not matter. This results in the model giving nonsensical results, which, while amusing, can cause major problems. Examples of adversarial attacks against neural networks range from adding a small amount of noise to a picture of a panda, resulting in the model classifying the image as a gibbon with 99% confidence [?] to fooling facial recognition by printing a pair of eyeglasses [?]. When designing machine learning models, it is important to be aware of these issues so that their impact can be mitigated.

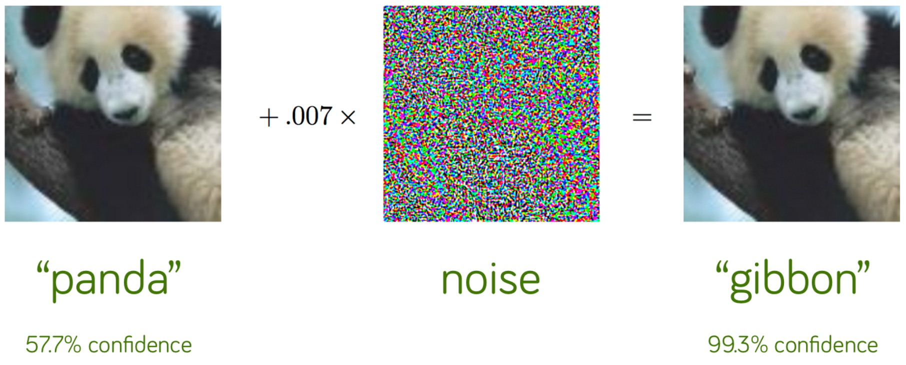


Figure 1.3: A slight modification to a correctly-classified image of a panda results in the model confidently classifying it as a gibbon, despite the image not having changed in any substantial way.

The example of modifying the image of a panda is an attack called the *Fast Gradient Sign Method (FGSM)*. FGSM is a *white-box attack*, meaning that the attacker has access to the model; this is in contrast with a *black-box attack* where the attacker only has access to the model's inputs and outputs.

During model training, gradients are used to adjust the model weights so that loss is minimized. In FGSM, the gradient is instead used to perturb the input image in a direction that *maximizes* the loss, using the following equation:

$$x_{\text{perturbed}} = x + \varepsilon \,\text{Sign}(\nabla_x \text{Loss}(\theta, x, y))$$

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

We can calculate this perturbation in PyTorch as follows. To calculate the gradient of the model with respect to a piece of data `x`, we first need to set `x.requires_grad = True` and call `x.retain_grad()`. Then, we zero out the optimizer's gradient, run `x` through the model, and compute the loss, similar to training. After this, the gradient of the output with respect to `x` can be obtained using the attribute `x.grad.data`.

The following function `fgsm_attack` accepts a model and an image and performs the FGSM attack, returning the perturbed image:

```
def fgsm_attack(model, optimizer, objective, x, y, eps):
    """
    Performs the FGSM attack on the given model and data point x with label y.
    Returns the perturbed data point.
    """
    # Calculate the gradient
    x.requires_grad = True
    x.retain_grad()
    optimizer.zero_grad()
    output = model(x)
    loss = objective(output, y)
    loss.backward()
    data_grad = x.grad.data
    # Perturb the images
    x_perturbed = x + eps * data_grad.sign()

    return x_perturbed
```

We will use this function to explore this type of adversarial attack on our neural network.

Problem 5. Write a function that loops through the test data using the function `fgsm_attack` to perturb the images and using your trained model from Problem 4.

Run your function for each epsilon in `[0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]`, and plot epsilon against the model's 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
>>> ex = perturbed_data.squeeze().detach().cpu().numpy()

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

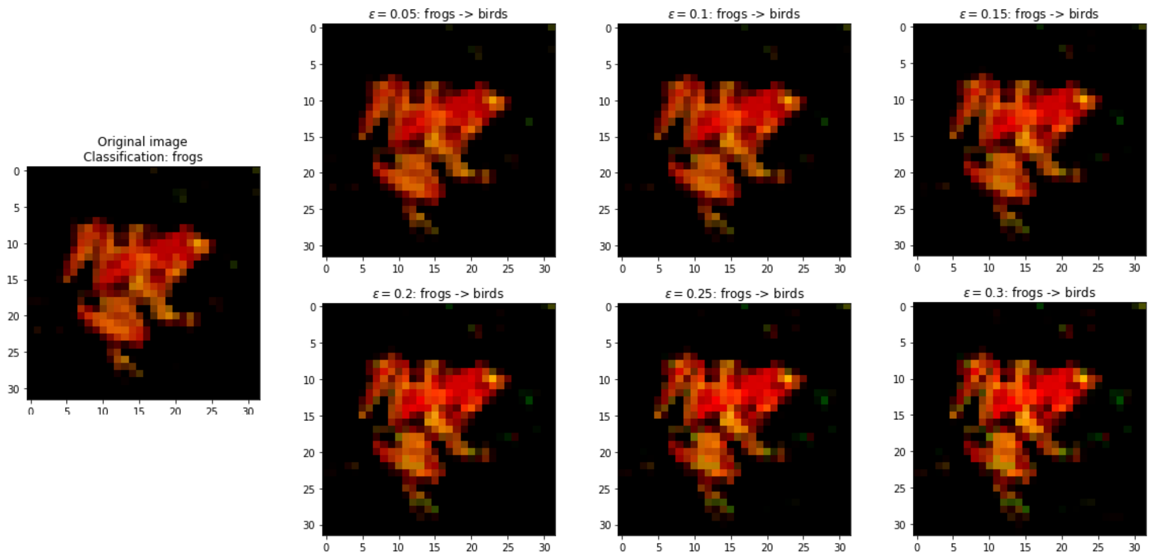


Figure 1.4: The first modified image for different values of epsilon.

Additional Materials

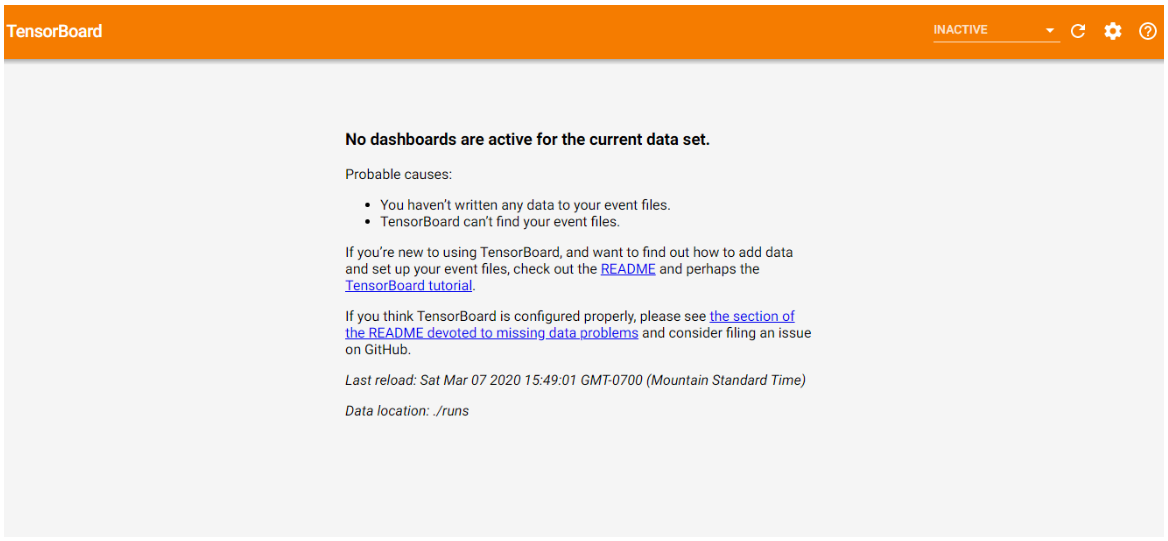
TensorBoard

TensorBoard is a visualization toolkit for neural networks. It was originally built for Tensorflow, but also can be used with PyTorch. The main features of TensorBoard include model visualization, 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:



No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Sat Mar 07 2020 15:49:01 GMT-0700 (Mountain Standard Time)

Data location: ./runs

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.

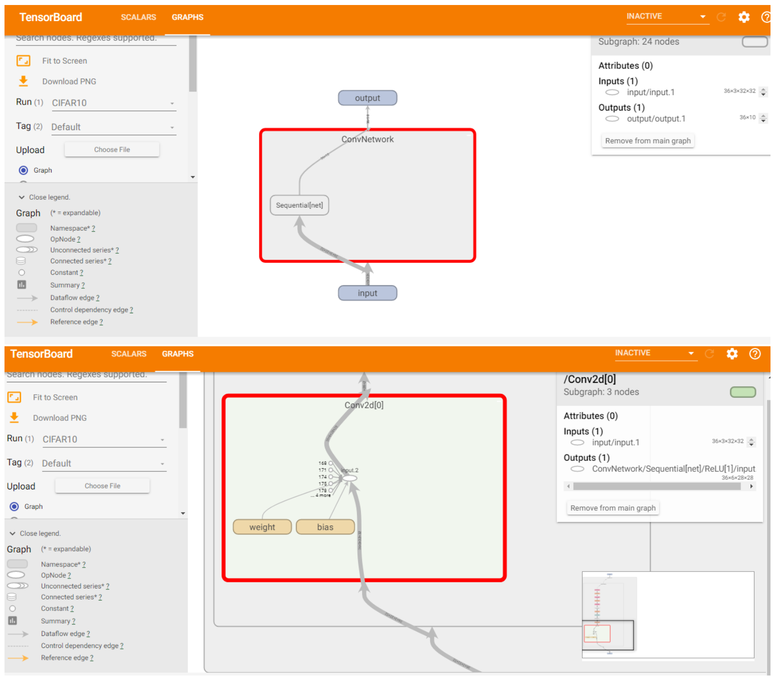


Figure 1.6: Examples of TensorBoard Graph Tab.

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 6. 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 4.