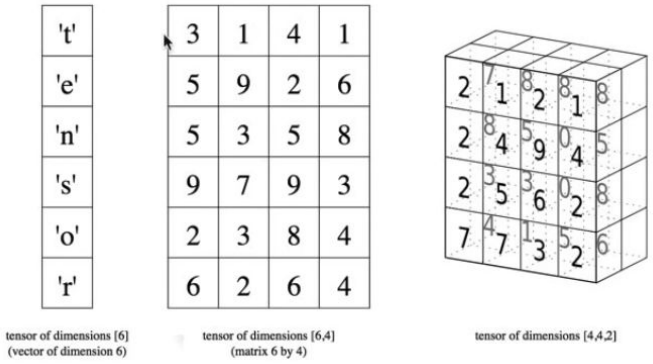
Lesson 05: Introduction to PyTorch

Notes by: Jvinniec

Note that some of the notes in this section are derived from the Jupyter notebooks associated with this lesson.

Tensors

Tensors are arrays of data and can come in many shapes and sizes:

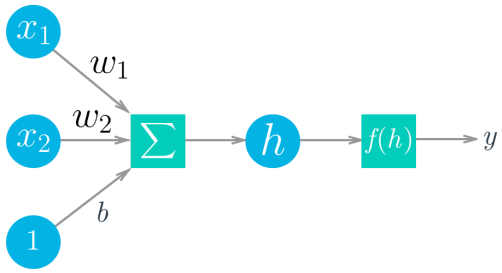


Here are some useful functions in the Python 'torch' module:

- `import torch`: This is the code for loading the 'torch' module:.
- `torch.exp(x)`: Computes the exponent of the specified value or tensor
- `torch.sum(x)`: Computes the sum of the elements in x.
- `torch.randn(shape)`: Computes a tensor of random variables with a given shape. Values are generated from a normal (i.e Gaussian) distribution with mean=0 and σ=1.
- `torch.randn_like(tensor)`: Computes a tensor of random values with a shape similar to another tensor.

Building a Neural Network in PyTorch

Recall that a simple neuron has the following form:



We can represent this mathematically as:

$$y = f(w_1 x_1 + w_2 x_2 + b)$$

Or for a n-feature network:

$$y = f\left(\sum_i^n w_i x_i + b\right)$$

Or with vectors vectors:

$$\boldsymbol{h} = [\,\boldsymbol{x}_1\,\boldsymbol{x}_2 \cdots \boldsymbol{x}_n\,] \cdot \begin{bmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \vdots \\ \boldsymbol{w}_n \end{bmatrix}$$

Note that the features have a shape of (1-row, n-columns) while the weights have a shape of (n-rows, 1-column). Now we can assemble all of this together by doing the following:

```
y = activation(torch.sum(features*weights) + bias)
```

but PyTorch provides some functions to make the multiplication of matrices a bit faster.

Matrix multiplication in PyTorch

There are two functions for matrix multiplication in PyTorch:

- torch.mm(tensor1, tensor2): Simple matrix multiplication. Requires matrices have the correct shape.
- torch.matmul(tensor1, tensor2): Somewhat more complicated & supports broadcasting.

Note that for most cases, `torch.mm()` will suffice.

Note that `torch.mm()` requires that the matrices have the correct shape. This means that if the features are (n x m) then our weights need to be (m x n). In order to get the transposed version of a torch array:

- `weights.reshape(a, b)` will return a new tensor with the same data as `weights` with size `(a, b)` sometimes, and sometimes a clone, as in it copies the data to another part of memory.
- `weights.resize_(a, b)` returns the same tensor with a different shape. However, if the new shape results in fewer elements than the original tensor, some elements will be removed from the tensor (but not from memory). If the new shape results in more elements than the original tensor, new elements will be uninitialized in memory. Here I should note that the underscore at the end of the method denotes that this method is performed **in-place**. Here is a great forum thread to read more about in-place operations in PyTorch: https://discuss.pytorch.org/t/what-is-in-place-operation/16244
- `weights.view(a, b)` will return a new tensor with the same data as `weights` with size `(a, b)`.

So in general it looks like the safest method to use is 'weights.view'. This method leaves the underlying data untouched, but simply changes the shape of the tensor.

Using this new information we can now compute the output in the following way:

```
y = activation(torch.mm(features, weights.view(5,1)) + bias)
```

Multi-layer Networks

Building a multi-layer neural network is fairly straight forward. The output from each layer in a network simply serves as the input to the next layer:

$$y = f_2(f_1(x \cdot W_1)W_2)$$

Or in using PyTorch:

```
h = activation(torch.mm(features, W1) + B1)
y = activation(torch.mm(h, W2) + B2)
```

Converting between NumPy arrays and PyTorch tensors

PyTorch tensors and NumPy arrays are interchangeable, meaning you can easily change one to the other:

```
a = np.random.rand(4,3)
b = torch.from_numpy(a)
```

Note that 'a' and 'b' will be the same. Also, a,b use the same memory, so any change to one will be reflected in the other.

Multi-layer Neural Networks: A more complex example

One of the basic tests of a neural network model is the **MNIST** dataset. MNIST is a collection of handwritten digits. In this section we will discuss how to generate a multi-layer neural network model designed around classifying these digits.

Importing Datasets

The `torchvision` Python module provides the ability to download a variety of datasets from the network. We can use this library to download the data that we want for the MNIST dataset:

```
from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                                ])
# Download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

What this does:

1. Import useful modules
    a. **datasets**: Provides access to many datasets that can be used for machine learning.
    b. **transforms**: Provides useful transformation operations that can be applied to tensors.
2. Compose a series of transformations that we would like to apply to the MNIST dataset
3. Create a loader (‘trainloader’) for the training dataset.

We can iterate through the training datasets by creating an iterator:

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

Building the network: Raw Tensors

So far we've only used fully-connected networks (i.e. every node in a layer is connected to every node in the following layer). **In fully-connected networks, each layer *MUST* be represented as a 1-dimensional vector**. This is a problem, since the MNIST data is 2-dimensional (they're images afterall). To fix this we simply use the '.view()' method we covered above to convert the training dataset from a 64x28x28 array (64 images that are 28x28 pixes each) into a 64x784. This operation, going from n-dimensions to 1-dimension, is called **Flattening**:

```
# Flatten the images
inputs = images.view(images.shape[0], 784)
# or equivalently...
inputs = images.view(images.shape[0], -1)
```

Note that in the second method didn't require us to know the explicit shape of the data, which is very useful.

You can build the respective elements of the model (weights and biases) in the same way as above:

```
# Create parameters
w1 = torch.randn(784, 256)
b1 = torch.randn(256)

w2 = torch.randn(256, 10)
b2 = torch.randn(10)

h = activation(torch.mm(inputs, w1) + b1)

out = torch.mm(h, w2) + b2
```

A few key notes:

- The first hidden layer has 256 nodes, so we need at least this many bias terms
- The final output layer has 10 nodes, one for each of our classifications.

Assign Probabilities to outputs: Softmax

The outputs will have some value depending on the weights and biases of the second layer. We need to turn these values into probabilities for each of the 10 classes in the output layer. For this we can use the 'softmax' function as our activation:

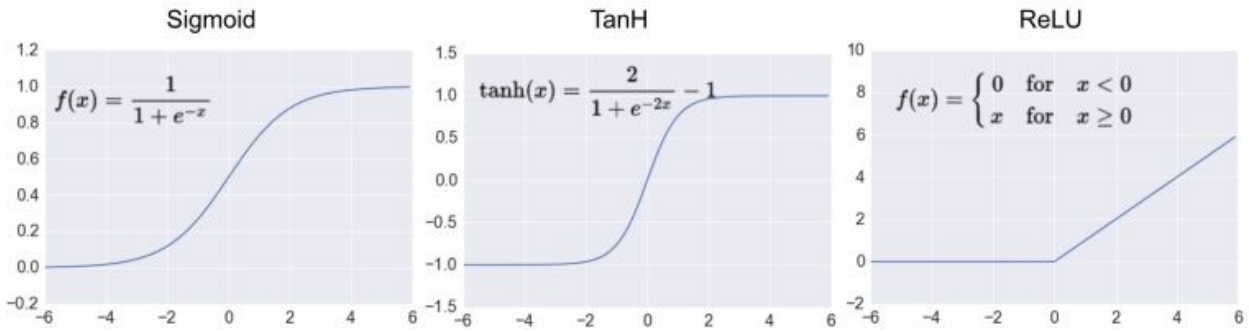
$$softmax(x_i) = \frac{e^{x_i}}{\sum_{n}^{N} e^{x_n}}$$

```
def softmax(x):
    return torch.exp(x)/torch.sum(torch.exp(x), dim=1).view(-1, 1)
```

where ‘N’ is the number of output classes that we have.

Other Activation functions

So far we’ve used the ‘sigmoid’ and ‘softmax’ activation functions, but there are many others. The only requirement to fit to non-linear data is that the activation function itself must be non-linear.



TanH

The hyperbolic tangent function.

ReLU

ReLU is the most common activation function used in neural networks. It has the benefit that the gradient of the function is always 1 (i.e. it doesn’t go to 0 for large inputs).

Building Networks in PyTorch

PyTorch provides functionality to assist with creating multi-layer networks. We can use the ‘torch.nn’ and ‘torch.nn.functional’ modules:

Using torch.nn

We can create a model class using the 'torch.nn' module as follows:

```
from torch import nn
class Network(nn.Module):
    def __init__(self):
        super().__init__()

        # Inputs to hidden layer linear transformation
        self.hidden = nn.Linear(784, 256)
        # Output layer, 10 units - one for each digit
        self.output = nn.Linear(256, 10)

        # Define sigmoid activation and softmax output
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # Pass the input tensor through each of our operations
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)

        return x
```

Notes about key parts:

- Our network class MUST inherit from the `nn.Module` class.
- Calling `super().__init__()` In our `__init__` function ensures all the necessary PyTorch internals are appropriately setup for our network.
- `nn.Linear(x, y)` defines a linear neural network (including weights and biases) with `x` inputs and `y` outputs.
- `nn.Sigmoid()` and `nn.Softmax(dim=1)` define the sigmoid and softmax functions. The 'dim=1' means softmax will calculate the normalization across each row (single image), rather than for all images combined.
- The `forward(...)` method will be called to forward-pass our data through the network and compute the result. **THIS METHOD IS MANDATORY!**

Using torch.nn.functional

The ‘functional’ module also provides some helper functions for creating more concise networks.

```
import torch.nn.functional as F
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        # Inputs to hidden layer linear transformation
        self.hidden = nn.Linear(784, 256)
        # Output layer, 10 units - one for each digit
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        # Hidden layer with sigmoid activation
        x = F.sigmoid(self.hidden(x))
        # Output layer with softmax activation
        x = F.softmax(self.output(x), dim=1)

        return x
```

Note here we don’t define the sigmoid and softmax functions at initialization time.

A note on the ‘dim’ parameter in ‘nn.Softmax()’ and ‘nn.LogSoftmax()’:

The output of our model (the ***logits***) will have the following shape:

```
# Forward pass, get our logits
logits = model(images)
print(logits.shape)
print(logits)
```

```
torch.Size([64, 10])
tensor([[-2.4929, -2.1632, -2.4207, -2.3949, -2.4264, -2.4367, -2.0594, -2.1978, -2.4533, -2.1042],
        [-2.4460, -2.3163, -2.3987, -2.3688, -2.3358, -2.2922, -2.2383, -2.1215, -2.3321, -2.2171],
        [-2.4587, -2.1802, -2.3335, -2.2939, -2.4069, -2.3237, -2.1682, -2.1931, -2.5049, -2.2259],
        [-2.3480, -2.2525, -2.4108, -2.3648, -2.3844, -2.3907, -2.1779, -2.1161, -2.4280, -2.2084],
… (and on and on)
```

The shape of the resulting tensor is 64 rows (1 for each input image) with 10 columns per row (1 column for each possible label).

- ‘**dim=1**’ means we want to normalize across the columns in each row so that the sum of all the probabilities for all labels for a single image is 1.
- ‘**dim=0**’ means we want to normalize across the rows in each column so that the sum of all the probabilities for a single label for all images is 1.

Using nn.Sequential

The ‘nn.Sequential’ module allows construction of a model by sequentially identifying how we want to pass our data through the network:

```
# Hyperparameters for our network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

# Build a feed-forward network
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.Softmax(dim=1))
```

‘nn.Sequential’ provides a builtin ‘forward()’ method, so we don’t have to define this ourselves. We access the individual models by their index (e.g. `model[0]` for first hidden layer). This approach makes accessing the components a little difficult, so instead we could use an ‘OrderedDict’:

```
from collections import OrderedDict
model = nn.Sequential(OrderedDict([
                      ('fc1', nn.Linear(input_size, hidden_sizes[0])),
                      ('relu1', nn.ReLU()),
                      ('fc2', nn.Linear(hidden_sizes[0], hidden_sizes[1])),
                      ('relu2', nn.ReLU()),
                      ('output', nn.Linear(hidden_sizes[1], output_size)),
                      ('softmax', nn.Softmax(dim=1))]))
```

We can now access our models by index, or by name (e.g. ‘model[0]’ and ‘model.fc1’ are identical).

NOTE:
It’s important to name your layers in such a way that makes it obvious what they are doing (e.g. ‘fc1’ and ‘fc2’ for the first and second fully connected layers).

Initializing Weights and Biases

To initialize the weights and Biases we can do the following (‘fc1’ is our first fully connected layer):

```
# Set biases to all zeros
model.fc1.bias.data.fill_(0)
# sample from random normal with standard dev = 0.01
model.fc1.weight.data.normal_(std=0.01)
```

This sets the biases to 0 and fills the weights with values sampled from a Gaussian distribution with standard deviation=0.01 centered at 0.

Forward Pass

In order to pass our data forward through our network and get the predictions, we can do:

```
# Grab some data
dataiter = iter(trainloader)
images, labels = dataiter.next()

# Resize images into a 1D vector, new shape is (batch size, color channels, image
pixels)
images.resize_(64, 1, 784)
# or images.resize_(images.shape[0], 1, 784) to automatically get batch size

# Forward pass through the network
img_idx = 0
ps = model.forward(images[img_idx,:]) # This is our probabilities for each class
```

Training Neural Networks in PyTorch

Now that we've built the models, we need to train them. During training we use a loss function to evaluate how well our model is doing.

$$l = \frac{1}{2n}\sum_{i}^{n}(y_i - \widehat{y}_i)^2$$

where '*n*' is the number of training samples. Reminder: To find the best value of our weights and biases we want to minimize the value of the loss function by gradient descent (this is done through backpropagation, see lesson 3 notes).

Computing the Loss

Pytorch has multiple loss formulas available:

- Cross-entropy (`nn.CrossEntropyLoss`): Often used with classification problems (such as MNIST) where you have a discrete set of outputs (1's and 0's, not continuous).

NOTE: `nn.CrossEntropyLoss` applies the `nn.LogSoftmax` function as well, so we want to pass it the raw output from our network (i.e. the ***logits***).

Computing the loss in PyTorch requires defining the criterion, computing the logits of our training data, and comparing them to the expected labels of the training sets:

```
# Define the loss
criterion = nn.CrossEntropyLoss()

# Forward pass, get our logits
logits = model(images)
# Calculate the loss with the logits and the labels
loss = criterion(logits, labels)
```

Computing the Gradients

PyTorch uses the **'autograd**' module to compute the gradients of the network. Autograd works by tracking operations performed on tensors, then going backwards through those operations to compute the gradients. To ensure the operations on a tensor are tracked we need to call one of two methods:

- Set '`requires_grad = True`' on a tensor at creation time.
- Call '`tensor.requires_grad_(True)`' on a tensor.

We can turn off gradients for a block of code:

```
x = torch.zeros(1, requires_grad=True)
>>> with torch.no_grad():
...     y = x * 2
>>> y.requires_grad
False
```

Or all together:

```
torch.set_grad_enabled(True|False)
```

Gradients are then computed with respect to some variable `z` with `z.backward()`. For example:

```
x = torch.randn(2,2, requires_grad=True)
y = x**2
z = y.mean()
```

Now that we have an output variable (i.e. 'z'), we can compute the gradients of the tensor 'x' with respect to it:

```
z.backward() # Propagate the values back through the network
X.grad       # Print the gradients
```

Typically, we will want to know the gradients with respect to the loss. Once we have the gradients we can iterate the gradient descent by one step.

Loss and Autograd Together

When we create a network all the parameters are initialized with 'requires_grad=True', so the gradients can be computed. A full example looks like this:

```
# Build a feed-forward network
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1)

# Compute 'loss' and back-propagate
logits = model(images)
loss = criterion(logits, labels)
loss.backward()

# Compute the gradients of the weights
model[0].weight.grad
```

Optimizer

The 'optimizer' is used to compute the gradients of the network. In PyTorch we get these from the 'optim' module. One example that we can use is the SGD optimizer:

```
from torch import optim
# Optimizers require the parameters to optimize and a learning rate
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Training the Network

Now we have the pieces to train our network. A single iteration in the training process involves the following 4 steps:

1. Make a forward pass through the network
2. Use the network output to calculate the loss
3. Perform a backward pass through the network with loss.backward() to calculate the gradients

4. Take a step with the optimizer to update the weights

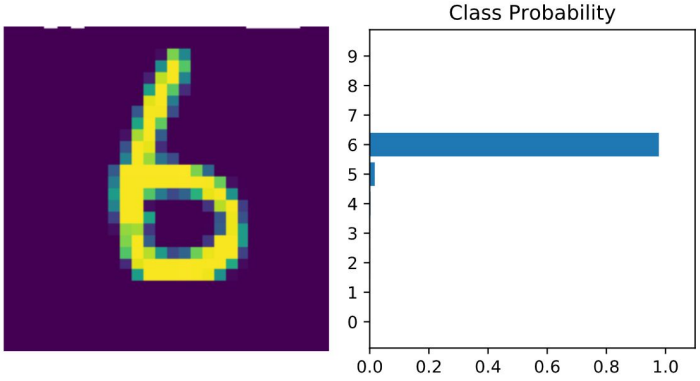
Note that in PyTorch the optimizers keep track of the gradients from all iterations, so it is important to clear these on each iteration:

```
# Clear the gradients, do this because gradients are accumulated
optimizer.zero_grad()
```

So programmatically, the training step looks like:

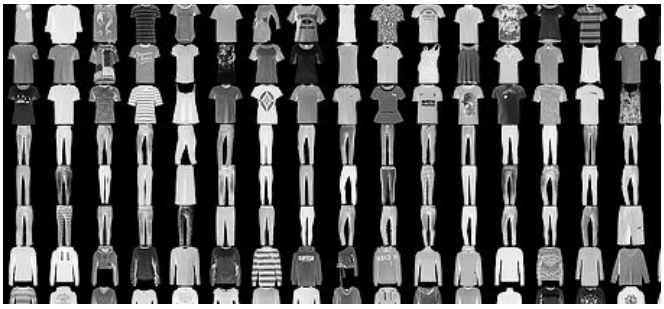
```
# Clear the gradients, do this because gradients are accumulated
optimizer.zero_grad()
# Forward pass, then backward pass, then update weights
output = model(images)
loss = criterion(output, labels)
loss.backward()
# Take an update step and few the new weights
optimizer.step()
```

When we do this and re-evaluate the resulting probabilities for an image we see that now the probability is much higher for the correct class and much lower for the incorrect classes:



Testing on Fashion-MNIST

Fashion-MNIST is a slightly more complicated dataset than the MNIST dataset. It contains images of different types of clothing (shirts, pants, shoes, etc…). Here, we are going to try to generate a model, train that model, and test its accuracy.



For this I'm going to start from where we left off in the previous MNIST example and go from there.

My solution:
https://github.com/Jvinniec/udacity_bertelsmann_ai_track/blob/master/Lesson05/scripts/part-04_classifying_fashion_mnist/Part4-Fashion_MNIST.ipynb

Inference and Validation

Overfitting is a topic covered in lesson 3. Simply, it happens when our model has become too specific to our training dataset and doesn't generalize well to data it hasn't seen before. We can test for overfitting by measuring the model's performance on data *NOT* in the training sample. This is called the **validation set**. There are ways that we can prevent overfitting (also covered in the previous lesson) such as dropout.

Measuring Performance

In PyTorch we can get the most likely class for a given data point (or image) using torch's '.topk':

```
ps = <model predictions tensor>
top_p, top_class = torch.topk(ps, 1, dim=1)
# or
top_p, top_class = ps.topk(1, dim=1)
```

'torch.topk()' returns (1) a vector of the top 'k' values and (2) vector of the top 'k' indices. By passing a value of '1' we are extracting values for the single most probable class. Note that the indices of top_class and labels are 0-based tensors.

We can now compute whether the classifications are correct and the accuracy:

```
equals = top_class == labels.view(*top_class.shape)
accuracy = torch.mean(equals.type(torch.FloatTensor))
print(f'Accuracy: {accuracy.item()*100}%')
```

Note that 'equals' is of type boolean, so in order to compute the mean we need to convert it to 'float'. Also, 'accuracy' is returned as a tensor, so we need to call 'accuracy.item()' to get the actual value.

Accuracy

Accuracy simply measures the number of classifications that were correct vs. the total number of classifications.

Precision

Recall

Top-5 Error Rate

Dropout in PyTorch

Dropout is simply another component that is added to a layer *after* the activation function.

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

# New model with dropout
class Classifier_withdropout(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 64)
        self.fc2 = nn.Linear(64, 10)

        # Define the dropout rate
        self.dropout = nn.Dropout(p=0.2)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        x = self.dropout(F.relu(self.fc1(x)))
        x = F.log_softmax(self.fc2(x), dim=1)

        return x
```

We use the `nn.Dropout' or 'F.dropout' function, the parameter 'p' represents the fraction of nodes to dropout during training.

Saving the models

One of the key aspects of training our models is to be able to deploy them somewhere else to actually run the classifications. So we need a method for saving the models, which is called 'torch.save'.

We typically save the model as a Python 'dict'. To get the dict associated with our model we do the following:

```
model.state_dict()
```

To ensure that we have all the information about the model, we can also put it information on the number of expected inputs, outputs, and hidden layer sizes into a dict along with the state_dict:

```
checkpoint = {'input_size': 784,
              'output_size': 10,
              'hidden_layers': [each.out_features for each in model.hidden_layers],
              'state_dict': model.state_dict()}

torch.save(checkpoint, 'checkpoint.pth')
```

Loading the models

Now we can use this information to reconstruct our model. This requires a dedicated function. Something like:

```
from torch import nn
def load_checkpoint(filepath):
    # Load the checkpoint dictionary
    checkpoint = torch.load(filepath)
    # Add the input layer
    model_list = nn.Linear(checkpoint['input_size'],
                           checkpoint['hidden_layers'][0])
    model_list.append(nn.ReLU())
    model_list.append(nn.Dropout(p=0.2))

    # Construct the hidden layers
    for i in range(len(checkpoint['hidden_layers'][:-1])):
        model_list.append(nn.Linear(checkpoint['hidden_layers'][i],
                                    checkpoint['hidden_layers'][i+1]))
```

```
            model_list.append(nn.ReLU())
            model_list.append(nn.Dropout(p=0.2))

        # Construct the output layer
        model_list.append(nn.Linear(checkpoint['hidden_layers'][-1],
                                    checkpoint['outputs']))
        model_list.append(nn.LogSoftmax(dim=1))

        # Construct the full model
        model = nn.Sequential(*model_list)

        # Set model values from the loaded state
        model.load_state_dict(checkpoint['state_dict'])

        return model
```

Ideally we would put our model construction into a class with a dedicated loading function to ensure the model is correctly constructed the same way every time.

Loading Image Data

The image datasets used so far in the lesson have been artificial and heavily processed to make using them to learn PyTorch easier. In reality the images would be much more messy.

datasets.ImageFolder

The easiest way to load image data in PyTorch is the 'datasets.ImageFolder' method. Using this method looks like this:

```
data = datasets.ImageFolder('path/to/data', transform=transforms)
```

Here:

- 'path/to/data' is the full path to the directory containing the images. Inside this directory should be a set of additional directories for each classification. For example, if we want to classify dog and cat images we would have 2 directories called 'dog' (containing our dog images) and 'cat' (containing our cat images).
- `'transform'` is the list of transforms to use for processing the images into a format usable by the analysis code (see the next section).

Transforms

Raw images usually come in all different shapes and sizes. However, for training we need to ensure that the images are all the same size. To format the images, we use 'transforms'.

Transforms apply a series of transformations to the images to get them into a standard size and shape. Some available transforms:

- transforms.Resize: Resize an image
- transforms.CenterCrop: Crop an image at the center
- transforms.RandomResizedCrop: Crop the given Image to random size and aspect ratio.
- transforms.ColorJitter: Randomly change the brightness, contrast and saturation of an image.
- transforms.RandomRotation: Rotate the image by angle.

Documentation on PyTorch transforms where we can find additional transforms (there are many).

We typically combine several transforms into a pipeline using transforms.Compose:

```
transform = transforms.Compose([transforms.Resize(255),
                                transforms.CenterCrop(224),
                                transforms.ToTensor()])
```

We will cover 'Transforms' more in Data Augmentation.

Data Loaders

Once we've defined the 'ImageFolder' object, we pass it to a 'DataLoader'. This takes the data and allows us to access the images along with their associated labels:

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=True)
```

Here we set:

- **batch_size=32**: Each time we loop over the images we will pull in batches of 32 images.
- **shuffle=True**: Each epoch will shuffle the images so that we never look at the same batch of images.

'DataLoader' is a generator, which means we can turn it into an iterator (via 'iter(DataLoader)') and call 'next()' to get subsequent batches. The following are equivalent ways to loop through the data:

```
# Looping through it, get a batch on each loop
for images, labels in dataloader:
    pass
```

or:

```
# Get one batch
images, labels = next(iter(dataloader))
```

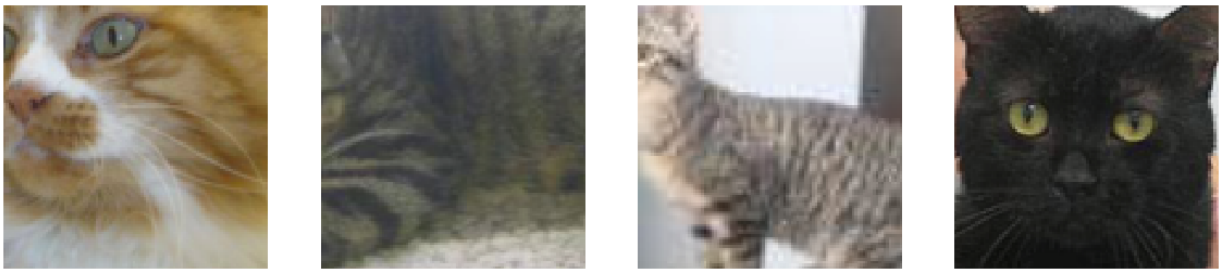
Data Augmentation

One strategy for training NNs is to introduce randomness in the input data. This includes: rotating, mirroring, scaling, cropping, and adjusting brightness. The following applies a random rotation, scaling/cropping, and flipping:

```
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
                                                            [0.5, 0.5, 0.5])])
```

At the end we apply a normalization to the image by passing a list of means and standard deviations. Typically we use values that scale the images to be centered at 0 in the range -1,1.

FOR TESTING we don't typically want to apply flips or rotations. Typically we just apply scalings, crops, and the same normalization. The training images should look something like the following:



Transfer Learning

Transfer learning is the process of taking a pre-trained neural network and applying it to a completely new set of data. PyTorch has many models available. See here:

https://pytorch.org/docs/0.3.0/torchvision/models.html#torchvision-models

Note, typically the number in the model name implies the number of layers.

Typical models in PyTorch are very large, so much so that training them on a CPU is not feasible. Instead models are often trained on GPUs using CUDA. To turn on evaluation of the models on the GPU we would do the following:

```
model.to('cuda')
# Reassign necessary
images = images.to('cuda')
labels = labels.to('cuda')
```

And to move them back to the CPU we would call:

```
model.to('cpu')
# Reassign necessary
images = images.to('cpu')
labels = labels.to('cpu')
```

We can also test if a cuda resource is available before hand. The following creates a ‘device’ variable that identifies which device we want to run on:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

By default we choose ‘cpu’, but choose ‘cuda’ if a cuda device is available.

Tips and Tricks

Making sure the shapes are correct

Make sure shapes are correct by monitoring the ‘.shape’ parameter of our tensors.

If network isn’t training properly

Check these things:

- Clear the gradients in each training loop with ‘optimizer.zero_grad()’
- Use ‘evaluation mode with ‘model.eval()’ and training mode with ‘model.train()’

CUDA Errors

Sometimes we see the following error:

```
RuntimeError: Expected object of type torch.FloatTensor but
found type torch.cuda.FloatTensor for argument #1 ‘mat1’
```

This usually happens when we are trying to do operations that involve multiple tensors, but where the tensors on different devices. Check to make sure that the tensors have been moved to the same device.