APRIL 10, 2018

Convolutional Neural Nets in PyTorch

Many of the exciting applications in Machine Learning have to do with images, which means they're likely built using Convolutional Neural Networks (or CNNs). This type of algorithm has been shown to achieve impressive results in many computer vision tasks and is a must-have part of any developer's or data scientist's modern toolkit.

This tutorial will walk through the basics of the architecture of a Convolutional Neural Network (CNN), explain why it works as well as it does, and step through the necessary code piece by piece. You should finish this with a good starting point for developing your own more complex architecture and applying CNNs to problems that intrigue you.

Thanks is due to Ujjwal Karn for the intuitive explanation of CNNs.

What Is A Convolutional Neural Net, Anyway?

CNNs are a subset of the field of computer vision, which is all about applying computational techniques to visual content. For more information about how computer vision works and the kinds of problems businesses are tackling with it, check out our introduction here.

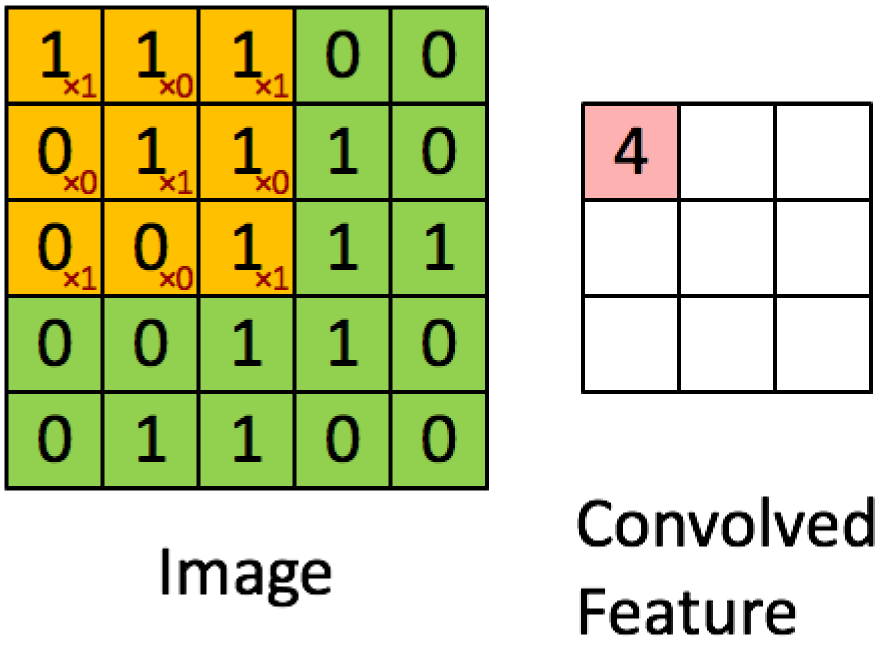
People often refer to a CNN as a type of algorithm but it's actually a combination of different algorithms that work well together. What differentiates a CNN from your run-of-the-mill neural net is the preprocessing or the stuff that you do to your data before passing it into the neural net itself. That means CNNs have two major pieces:

1. *Feature engineering / preprocessing* – turn our images into something that the algorithm can more efficiently interpret
2. *Classification* – train the algorithm to map our images to the given classes and understand the underlying relationship

Preprocessing in CNNs is aimed at turning your input images into a set of features that is more informative to the neural net. It can get complicated, but as long as you remember that there are only two sections and the goals of each, you won't get lost in the weeds.

Convolution

The CNN gets its name from the process of Convolution, which is the first filter applied as part of the feature-engineering step. Think of convolution as applying a filter to our image. We pass over a mini image, usually called a kernel, and output the resulting, filtered subset of our image.



Source: Stanford Deep Learning

Since an image is just a bunch of pixel values, in practice this means multiplying small parts of our input images by the filter. There are a few parameters that get adjusted here:

- *Kernel Size* – the size of the filter.
- *Kernel Type* – the values of the actual filter. Some examples include identity, edge detection, and sharpen.
- *Stride* – the rate at which the kernel passes over the input image. A stride of 2 moves the kernel in 2-pixel increments.
- *Padding* – we can add layers of 0s to the outside of the image in order to make sure that the kernel properly passes over the edges of the image.
- *Output Layers* – how many different kernels are applied to the image.

The output of the convolution process is called the “convolved feature” or “feature map.” Remember: it’s just a filtered version of our original image where we multiplied some pixels by some numbers.

The resulting feature map can be viewed as a more optimal representation of the input image that’s more informative to the eventual neural network that the image will be passed through. In practice, convolution combined with the next two steps has been shown to greatly increase the accuracy of neural networks on images.

Code: you’ll see the convolution step through the use of the torch.nn.Conv2d() function in PyTorch.

ReLU

Since the neural network forward pass is essentially a linear function (just multiplying inputs by weights and adding a bias), CNNs often add in a nonlinear function to help approximate such a relationship in the underlying data.

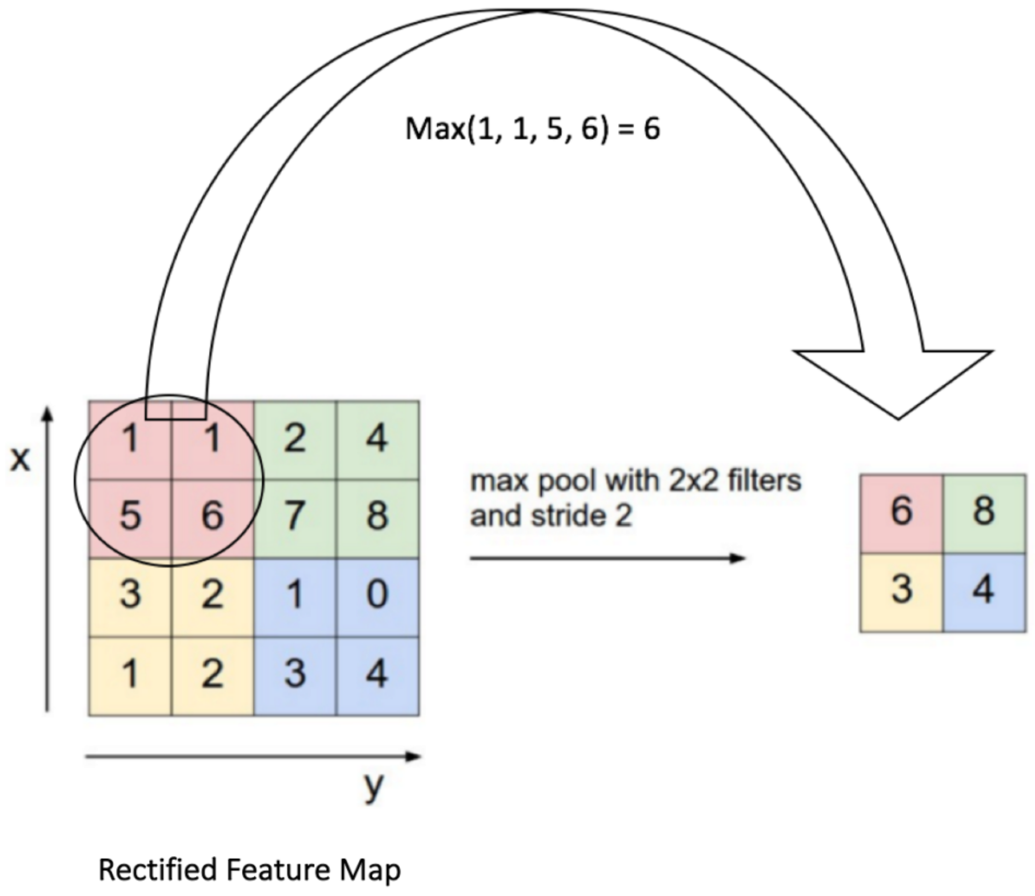
The function most popular with CNNs is called ReLU and it’s extremely simple. ReLU stands for Rectified Linear Unit, and it just converts all negative pixel values to 0. The function itself is output = Max(0, input).

There are other functions that can be used to add non-linearity, like *tanh* or *softmax*. But in CNNs, ReLU is the most commonly used.

Code: you’ll see the ReLU step through the use of the torch.nn.relu() function in PyTorch.

Max Pooling

The last part of the feature engineering step in CNNs is pooling, and the name describes it pretty well: we pass over sections of our image and pool them into the highest value in the section. Depending on the size of the pool, this can greatly reduce the size of the feature set that we pass into the neural network. This graphic from Stanford's course page visualizes it simply:



Max pooling also has a few of the same parameters as convolution that can be adjusted, like stride and padding. There are also other types of pooling that can be applied, like sum pooling or average pooling.

Code: you'll see the max pooling step through the use of the torch.nn.MaxPool2d() function in PyTorch.

Fully Connected Layers

After the above preprocessing steps are applied, the resulting image (which may end up looking nothing like the original!) is passed into the traditional neural network architecture. Designing the optimal neural network is beyond the scope of this post, and we'll be using a simple two-layer format, with one hidden layer and one output layer.

This part of the CNN is almost identical to any other standard neural network. The key to understanding CNNs is this: the driver of better accuracy is the steps we take to engineer better features, not the classifier we end up passing those values through. Convolution, ReLU, and max pooling prepare our data for the neural network in a way that extracts all the useful information they have in an efficient manner.

Code: *you'll see the forward pass step through the use of the torch.nn.Linear() function in PyTorch.*

Getting Started in PyTorch

This tutorial is in PyTorch, one of the newer Python-focused frameworks for designing deep learning workflows that can be easily productionized. PyTorch is one of many frameworks that have been designed for this purpose and work well with Python, among popular ones like TensorFlow and Keras.

There are a few key differences between these popular frameworks that should determine which is the right for you and your project, including constraints like:

- Ease of deployment
- Level of abstraction
- Visualization options
- Debugging flexibility

It's safe to say that PyTorch has a medium level of abstraction between Keras and Tensorflow. It also offers strong support for GPUs.

To install PyTorch, head to the homepage and select your machine configuration.

Gathering and Loading Data

As with most machine learning projects, a minority of the code you end up writing has to do with actual statistics–most is spent on gathering, cleaning, and readying your data for analysis. CNNs in PyTorch are no exception.

PyTorch ships with the *torchvision* package, which makes it easy to download and use datasets for CNNs. To stick with convention and benchmark accurately, we'll use the CIFAR-10 dataset. CIFAR-10 contains images of 10 different classes, and is a standard library used for building CNNs.

We'll want to start with importing the PyTorch libraries as well as the standard numpy library for numerical computation.

```python
import numpy as np
import torch
import torchvision
import torchvision.transforms as transforms
```

imports.py hosted with ❤ by GitHub view raw

We'll also want to set a standard random seed for reproducible results.

```python
seed = 42
np.random.seed(seed)
torch.manual_seed(seed)
```

random.py hosted with ❤ by GitHub view raw

The first step to get our data is to use PyTorch and download it.

```python
#The compose function allows for multiple transforms
#transforms.ToTensor() converts our PILImage to a tensor of shape (C x H x W) in the ra
#transforms.Normalize(mean,std) normalizes a tensor to a (mean, std) for (R, G, B)
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5,

train_set = torchvision.datasets.CIFAR10(root='./cifardata', train=True, download=True,

test_set = torchvision.datasets.CIFAR10(root='./cifardata', train=False, download=True,
```

download.py hosted with ❤ by GitHub view raw

We then designate the 10 possible labels for each image:

```python
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

classes.py hosted with ❤ by GitHub view raw

The final step of data preparation is to define samplers for our images. These PyTorch objects will split all of the available training examples into training, test, and cross validation sets when we train our model later on.

```python
from torch.utils.data.sampler import SubsetRandomSampler
```

```python

#Training
n_training_samples = 20000
train_sampler = SubsetRandomSampler(np.arange(n_training_samples, dtype=np.int64))

#Validation
n_val_samples = 5000
val_sampler = SubsetRandomSampler(np.arange(n_training_samples, n_training_samples + n

#Test
n_test_samples = 5000
test_sampler = SubsetRandomSampler(np.arange(n_test_samples, dtype=np.int64))
```

samplers.py hosted with ❤ by GitHub view raw

Designing a Neural Network in PyTorch

PyTorch makes it pretty easy to implement all of those feature-engineering steps that we described above. We'll be making use of four major functions in our CNN class:

1. **torch.nn.Conv2d**(in_channels, out_channels, kernel_size, stride, padding) – applies convolution
2. **torch.nn.relu**(x) – applies ReLU
3. **torch.nn.MaxPool2d**(kernel_size, stride, padding) – applies max pooling
4. **torch.nn.Linear**(in_features, out_features) – fully connected layer (multiply inputs by learned weights)

Writing CNN code in PyTorch can get a little complex, since everything is defined inside of one class. We'll create a SimpleCNN class, which inherits from the master torch.nn.Module class.

```python
from torch.autograd import Variable
import torch.nn.functional as F

class SimpleCNN(torch.nn.Module):

    #Our batch shape for input x is (3, 32, 32)

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

        #Input channels = 3, output channels = 18
        self.conv1 = torch.nn.Conv2d(3, 18, kernel_size=3, stride=1, padding=1)
        self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        #4608 input features, 64 output features (see sizing flow below)
        self.fc1 = torch.nn.Linear(18 * 16 * 16, 64)

        #64 input features, 10 output features for our 10 defined classes
        self.fc2 = torch.nn.Linear(64, 10)

    def forward(self, x):

        #Computes the activation of the first convolution
        #Size changes from (3, 32, 32) to (18, 32, 32)
        x = F.relu(self.conv1(x))

        #Size changes from (18, 32, 32) to (18, 16, 16)
        x = self.pool(x)

        #Reshape data to input to the input layer of the neural net
        #Size changes from (18, 16, 16) to (1, 4608)
        #Recall that the -1 infers this dimension from the other given dimension
        x = x.view(-1, 18 * 16 *16)

        #Computes the activation of the first fully connected layer
        #Size changes from (1, 4608) to (1, 64)
        x = F.relu(self.fc1(x))

        #Computes the second fully connected layer (activation applied later)
        #Size changes from (1, 64) to (1, 10)
        x = self.fc2(x)
        return(x)
```

simplecnn.py hosted with ❤ by **GitHub** view raw

Let's explain what's going on here. We're creating a SimpleCNN class with one class method: forward. The forward() method computes a forward pass of the CNN, which includes the preprocessing steps we outlined above. When an instance of the SimpleCNN class is created, we define internal functions to represent the layers of the net. During the forward pass, we call these internal functions.

One of the pesky parts about manually defining neural nets is that we need to specify the sizes of inputs and outputs at each part of the process. The comments should give some direction as to what's happening with size changes at each step. In general, the output size for any dimension in our input set can be defined as:

```
def outputSize(in_size, kernel_size, stride, padding):

output = int((in_size - kernel_size + 2*(padding)) / stride) + 1
```

```python

return(output)
```

outputsize.py hosted with ❤ by GitHub view raw

To use an example from our CNN, look at the max-pooling layer. The input dimension is (18, 32, 32)—using our formula applied to each of the final two dimensions (the first dimension, or number of feature maps, remains unchanged during any pooling operation), we get an output size of (18, 16, 16).

Training a Neural Net in PyTorch

Once we've defined the class for our CNN, we need to train the net itself. This is where neural network code gets interesting. If you're working with more basic types of machine learning algorithms, you can usually get meaningful output in just a few lines of code. For example, implementing a Support Vector Machine in the sklearn Python package is as easy as:

```python
#Import the support vector machine module from the sklearn framework
from sklearn import svm

#Label x and y variables from our dataset
x = ourData.features
y = ourData.labels

#Initialize our algorithm
classifier = svm.SVC()

#Fit model to our data
classifier.fit(x,y)
```

svm.py hosted with ❤ by GitHub view raw

With neural networks in PyTorch (and TensorFlow) though, it takes a lot more code than that. Our basic flow is a training loop: each time we pass through the loop (called an "epoch"), we compute a forward pass on the network and implement backpropagation to adjust the weights. We'll also record some other measurements like loss and time passed, so that we can analyze them as the net trains itself.

To start, we'll define our data loaders using the samplers we created above.

```python
#DataLoader takes in a dataset and a sampler for loading (num_workers deals with system
def get_train_loader(batch_size):
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
                                              sampler=train_sampler, num_workers=2)
    return(train_loader)
```

trainloader.py hosted with ❤ by GitHub view raw

```
#Test and validation loaders have constant batch sizes, so we can define them directly
test_loader = torch.utils.data.DataLoader(test_set, batch_size=4, sampler=test_sampler,
val_loader = torch.utils.data.DataLoader(train_set, batch_size=128, sampler=val_sampler
```

testvalloaders.py hosted with ❤ by GitHub view raw

We'll also define our loss and optimizer functions that the CNN will use to find the right weights. We'll be using Cross Entropy Loss (Log Loss) as our loss function, which strongly penalizes high confidence in the wrong answer. The optimizer is the popular Adam algorithm (not a person!).

```
import torch.optim as optim

def createLossAndOptimizer(net, learning_rate=0.001):

    #Loss function
    loss = torch.nn.CrossEntropyLoss()

    #Optimizer
    optimizer = optim.Adam(net.parameters(), lr=learning_rate)

    return(loss, optimizer)
```

lossandoptimizer.py hosted with ❤ by GitHub view raw

Finally, we'll define a function to train our CNN using a simple for loop.

```
import time

def trainNet(net, batch_size, n_epochs, learning_rate):

    #Print all of the hyperparameters of the training iteration:
    print("===== HYPERPARAMETERS =====")
    print("batch_size=", batch_size)
    print("epochs=", n_epochs)
    print("learning_rate=", learning_rate)
    print("=" * 30)

    #Get training data
    train_loader = get_train_loader(batch_size)
    n_batches = len(train_loader)

    #Create our loss and optimizer functions
    loss, optimizer = createLossAndOptimizer(net, learning_rate)

    #Time for printing
    training_start_time = time.time()

    #Loop for n_epochs
    for epoch in range(n_epochs):

        running_loss = 0.0
        print_every = n_batches // 10
        start_time = time.time()
        total_train_loss = 0

        for i, data in enumerate(train_loader, 0):

            #Get inputs
            inputs, labels = data

            #Wrap them in a Variable object
            inputs, labels = Variable(inputs), Variable(labels)

            #Set the parameter gradients to zero
            optimizer.zero_grad()

            #Forward pass, backward pass, optimize
            outputs = net(inputs)
            loss_size = loss(outputs, labels)
            loss_size.backward()
            optimizer.step()

            #Print statistics
            running_loss += loss_size.data[0]
            total_train_loss += loss_size.data[0]

            #Print every 10th batch of an epoch
            if (i + 1) % (print_every + 1) == 0:
                print("Epoch {}, {:d}% \t train_loss: {:.2f} took: {:.2f}s".format(
                        epoch+1, int(100 * (i+1) / n_batches), running_loss / print_ev
                #Reset running loss and time
                running_loss = 0.0
                start_time = time.time()

        #At the end of the epoch, do a pass on the validation set
        total_val_loss = 0
        for inputs, labels in val_loader:

            #Wrap tensors in Variables
            inputs, labels = Variable(inputs), Variable(labels)

            #Forward pass
            val_outputs = net(inputs)
            val_loss_size = loss(val_outputs, labels)
            total_val_loss += val_loss_size.data[0]

        print("Validation loss = {:.2f}".format(total_val_loss / len(val_loader)))

    print("Training finished, took {:.2f}s".format(time.time() - training_start_time))
```

training.py hosted with ❤ by GitHub view raw

During each epoch of training, we pass data to the model in batches whose size we define when we call the training loop. Data is feature-engineered using the SimpleCNN class we've defined, and then basic metrics are printed after a few passes. During each loop, we also calculate the loss on our validation set.

To actually train the net now only requires two lines of code:

```
CNN = SimpleCNN()
trainNet(CNN, batch_size=32, n_epochs=5, learning_rate=0.001)
```

call.py hosted with ❤ by GitHub view raw

And that's it! You've successful trained your CNN in PyTorch.

Next Steps and Options

Accuracy Metrics

Our training loop prints out two measures of accuracy for the CNN: training loss (after batch multiples of 10) and validation loss (after each epoch). When we defined the loss and optimization functions for our CNN, we used the *torch.nn.CrossEntropyLoss()* function. Cross Entropy Loss, also referred to as Log Loss, outputs a probability value between 0 and 1 that increases as the probability of the predicted label diverges from the actual label.

For machine learning pipelines, other measures of accuracy like precision, recall, and a confusion matrix might be used. Each project has different goals and limitations, so you should tailor your “metric of choice”—the measure of accuracy that you optimize for—towards those goals.

Network Architecture

On the CIFAR-10 dataset, the loss we’re getting translates to about 60% accuracy on the training dataset. That’s much better than the base rate–what you’d get by guessing at random–but it’s still very far from the state of the art.

One of the major differences between our model and those that achieve 80%+ accuracy is layers. Our network has one convolution layer, one pooling layer, and two layers of the neural network itself (four total layers). For example, the VGG-16 architecture utilizes more than 16 layers and won high awards at the ImageNet 2014 Challenge.

To add more layers into our CNN, we can create new methods during the initialization of our SimpleCNN class instance (although by then, we might want to change the class name to LessSimpleCNN). For example, we could try:

```
self.conv2 = torch.nn.Conv2d(3, 18, kernel_size=3, stride=1, padding=1)
self.pool2 = torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

layers.py hosted with ❤ by GitHub view raw

Hyperparameter Adjustment

In addition to varying the sizes of inputs and activation functions we use, the convolution operation and max pooling have more hyperparameters that we can adjust. Fiddling with the kernel size, stride, and padding can extract more informative features and lead to higher accuracy (if not overfitting).

Getting a CNN in PyTorch working on your laptop is very different than having one working in production. Algorithmia supports PyTorch, which makes it easy to turn this simple CNN into a model that scales in seconds and works blazingly fast. Check out our PyTorch documentation here, and consider publishing your first algorithm on Algorithmia.

Algorithmia

More Posts - Website

Follow Me:

Search

Enter your query here...