Deep Learning with PyTorch

Deep Learning Building Blocks: Affine maps, non-linearities and objectives

Deep learning consists of composing linearities with non-linearities in clever ways. The introduction of non-linearities allows for powerful models. In this section, we will play with these core components, make up an objective function, and see how the model is trained.

Affine Maps

One of the core workhorses of deep learning is the affine map, which is a function $f(x)$ where

$$f(x) = Ax + b$$

for a matrix A and vectors x, b. The parameters to be learned here are A and b. Often, b is referred to as the bias term.

PyTorch and most other deep learning frameworks do things a little differently than traditional linear algebra. It maps the rows of the input instead of the columns. That is, the ith row of the output below is the mapping of the ith row of the input under A, plus the bias term. Look at the example below.

```
# Author: Robert Guthrie

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)


lin = nn.Linear(5, 3)  # maps from R^5 to R^3, parameters A, b
# data is 2x5.  A maps from 5 to 3... can we map "data" under A?
data = torch.randn(2, 5)
print(lin(data))  # yes
```

Out:

```
tensor([[ 0.1755,  0.3260, -0.5065],
        [-0.0403,  0.2346,  0.1009]])
```

Non-Linearities

First, note the following fact, which will explain why we need non-linearities in the first place. Suppose we have two affine maps $f(x) = Ax + b$ and $g(x) = Cx + d$. What is $f(g(x))$?

$$f(g(x)) = A(Cx + d) + b = ACx + (Ad + b)$$

AC is a matrix and $Ad + b$ is a vector, so we see that composing affine maps gives you an affine map.

From this, you can see that if you wanted your neural network to be long chains of affine compositions, that this adds no new power to your model than just doing a single affine map.

If we introduce non-linearities in between the affine layers, this is no longer the case, and we can build much more powerful models.

There are a few core non-linearities. $\tanh(x)$, $\sigma(x)$, $\mathrm{ReLU}(x)$ are the most common. You are probably wondering: "why these functions? I can think of plenty of other non-linearities." The reason for this is that they have gradients that are easy to compute, and computing gradients is essential for learning. For example

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

A quick note: although you may have learned some neural networks in your intro to AI class where $\sigma(x)$ was the default non-linearity, typically people shy away from it in practice. This is because the gradient vanishes very quickly as the absolute value of the argument grows. Small gradients means it is hard to learn. Most people default to Tanh or ReLU.

```
# In pytorch, most non-linearities are in torch.functional (we have it imported as F)
# Note that non-linearities typically don't have parameters like affine maps do.
# There are some exceptions, PReLU for instance, which are updated during training.
data = torch.randn(2, 2)
print(data)
print(F.relu(data))
```

Out:

```
tensor([[ 0.5404, -1.2302]
        [ 1.1774, -0.7990]])
tensor([[ 0.5404,  0.0000]
        [ 1.1774,  0.0000]])
```

Softmax and Probabilities

The function $\mathrm{Softmax}(x)$ is also just a non-linearity, but it is special in that it usually is the last operation done in a network. This is because it takes in a vector of real numbers and returns a probability distribution. Its definition is as follows. Let x be a vector of real numbers (positive, negative, whatever, there are no constraints). Then the ith component of $\mathrm{Softmax}(x)$ is

$$\frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

It should be clear that the output is a probability distribution: each element is non-negative and the sum over all components is 1.

You could also think of it as just applying an element-wise exponentiation operator to the input to make everything non-negative and then dividing by the normalization constant.

```
# Softmax is also in torch.nn.functional
data = torch.randn(5)
print(data)
print(F.softmax(data, dim=0))
print(F.softmax(data, dim=0).sum)  # Sums to 1 because it is a distribution!
print(F.log_softmax(data, dim=0))  # there's also log_softmax
```

Out:

```
tensor([ 1.3800, -1.2585,  0.5455,  0.5626,  1.2413])
tensor([ 0.2945,  0.0352,  0.1942,  0.1128,  0.4634])
tensor(<built-in method sum of Tensor object at 0x7f8e0a3d0bd0>)
tensor([-1.2224, -3.9529, -2.2568, -2.0909, -0.7681])
```

Objective Functions

The objective function is the function that your network is being trained to minimize (in which case it is often called a *loss function* or *cost function*). This proceeds by first choosing a training instance, running it through your neural network, and then computing the loss of the output. The parameters of the model are then updated by taking the derivative of the loss function. Intuitively, if your model is completely confident in its answer, and its answer is wrong, your loss will be high. If it is very confident in its answer, and its answer is correct, the loss will be low.

The idea behind minimizing the loss function on your training examples is that your network will hopefully generalize well and have small loss on unseen examples in your dev set, test set, or in production. An example loss function is the negative log-likelihood loss, which is a very common objective for multi-class classification. For supervised multi-class classification, this means training the network to minimize the negative log probability of the correct output (or equivalently, maximize the log probability of the correct output).

Optimization and Training

So what we can compute a loss function for an instance? What do we do with that? We saw earlier that Tensors know how to compute gradients with respect to the things that were used to compute it. Well, since our loss is an Tensor, we can compute gradients with respect to all of the parameters used to compute it! Then we can perform standard gradient updates. Let θ be our parameters, $J(\theta)$ the loss function, and η a positive learning rate. Then

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta)$$

There are a huge collection of algorithms and active research in attempting to do something more than just this vanilla gradient update. Many attempt to vary the learning rate based on what is happening at train time. You don't need to worry about what specifically these algorithms are doing unless you are really interested. Torch provides many in the torch.optim package, and they are all completely transparent. Using the simplest gradient update is the same as the more complicated algorithms. Trying different update algorithms and different parameters for update algorithms (like different initial learning rates) is important in optimizing your network's performance. Often, just replacing vanilla SGD with an optimizer like Adam or RMSProp will boost performance noticably.

Creating Network Components in PyTorch

Before we move on to our focus on NLP, this is an annotated example of building a network in PyTorch using only affine maps and non-linearities. We will also see how to compute a loss function, using PyTorch's built in negative log likelihood, and update parameters by backpropagation.

All network components should inherit from nn.Module and override the forward() method. That is about it, as far as the bookkeeping is concerned. Inheriting from nn.Module provides functionality to your component. For example, it makes it keep track of its trainable parameters, you can swap it between CPU and GPU with the `.to(device)` method, where device can be a CPU device `torch.device("cpu")` or CUDA device `torch.device("cuda:0")`.

Let's write an annotated example of a network that takes in a sparse bag-of-words representation and outputs a probability distribution over two labels: "English" and "Spanish". This model is just logistic regression.

Example: Logistic Regression Bag-of-Words classifier

Our model will map a sparse BoW representation to log probabilities over labels. We assign each word in the vocab an index. For example, say our entire vocab is two words "hello" and "world", with indices 0 and 1 respectively. The BoW vector for the sentence "hello hello hello hello" is

$$[4, 0]$$

For "hello world world hello", it is

$$[2, 2]$$

etc. In general, it is

$$[\mathrm{Count(hello)}, \mathrm{Count(world)}]$$

Denote this BOW vector as x. The output of our network is:

$$\log \mathrm{Softmax}(Ax + b)$$

That is, we pass the input through an affine map and then do log softmax.

```
data = [("me gusta comer en la cafeteria".split(), "SPANISH"),
        ("Yo no quiero ir otra vez".split(), "SPANISH"),
        ("mi tu gusta que nos una buena idea".split(), "SPANISH"),
        ("Me it is not a good idea to get lost at sea".split(), "ENGLISH")]

test_data = ("Yo creo que si".split(), "SPANISH",
             "It is hot out".split(), "ENGLISH")

# word_to_ix maps each word in the vocab to a unique integer, which will be its
# index into the BoW vector over words
word_to_ix = {}
for sent, _ in data + test_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)

VOCAB_SIZE = len(word_to_ix)
NUM_LABELS = 2


class BoWClassifier(nn.Module):  # inherits from nn.Module

    def __init__(self, num_labels, vocab_size):
        # calls the init function of nn.Module.  Dont get confused by syntax,
        # just always do it in your subclass
        super(BoWClassifier, self).__init__()

        # Define the parameters that you will need.  In this case, we need A and b,
        # the parameters of the affine mapping.
        # Torch defines nn.Linear(), which provides the affine map
        # Make sure you understand why it has two dimensions in ovcab_size
        # and the output is num_labels!
        self.linear = nn.Linear(vocab_size, num_labels)

        # NOTE: The non-linearity log softmax does not have parameters! So we don't need
        # to worry about that here

    def forward(self, bow_vec):
        # Pass the input through the linear layer,
        # then through the softmax layer
        # Many non-linearities and other functions are in torch.nn.functional
        return F.log_softmax(self.linear(bow_vec), dim=1)

def make_bow_vector(sentence, word_to_ix):
    vec = torch.zeros(len(word_to_ix))
    for word in sentence:
        vec[word_to_ix[word]] += 1
    return vec.view(1, -1)

def make_target(label, label_to_ix):
    return torch.LongTensor([label_to_ix[label]])

model = BoWClassifier(NUM_LABELS, VOCAB_SIZE)

# the model knows its parameters.  The first output below is A, the second is b.
# Whenever you assign a tensor as a member variable in your module:
# it's added to your module parameters
# self.linear = nn.Linear(...)
# then you can get the parameters from the PyTorch data, your module
# (in this case, BoWClassifier) will store knowledge of the nn.Linear's parameters
for param in model.parameters():
    print(param)

# To run the model, pass in a bow vector
# Here we don't need to train, so the code is wrapped in torch.no_grad()
with torch.no_grad():
    sample = data[0]
    bow_vector = make_bow_vector(sample[0], word_to_ix)
    log_probs = model(bow_vector)
    print(log_probs)
```

Out:

```
{'me': 0, 'gusta': 1, 'comer': 2, 'en': 3, 'la': 4, 'cafeteria': 5, 'Yo': 6, 'no': 7,
Parameter containing:
tensor([[-0.1150,  0.0095, -0.1206,  0.1274,  0.1210,  0.1179, -0.2899,
         -0.0215,  0.0018,  0.0206,  0.1180, -0.1392,  0.1907,  0.0684,
          0.0050,  0.0112,  0.0644,  0.0465,  0.0713,  0.0737, -0.0120,
         -0.1176,  0.1944,  0.1852,  0.0000,  0.0000,  0.0000,
          0.1211, -0.1136, -0.1742,  0.1427, -0.0316,  0.1185,  0.0630,
          0.1671,  0.0394,  0.0472, -0.1323, -0.0902,  0.0660,  0.0231,
         -0.1344,  0.1777, -0.0147,  0.0000,  0.0000,  0.0000, -0.1204,
          0.1666,  0.1002,  0.0182,  0.1194,  0.0049,  0.1206,
Parameter containing:
tensor([ 0.0913,  0.1003]
tensor([-0.5235, -0.8717]])
```

Which of the above values corresponds to the log probability of ENGLISH, and which to SPANISH? We never defined it, but we need it if we want to train the thing.

```
label_to_ix = {"SPANISH": 0, "ENGLISH": 1}
```

So let's train! To do this, we pass instances through to get log probabilities, compute a loss function, compute the gradient of the loss function, and then update the parameters with a gradient step. Loss functions are provided by Torch in the nn.package. In NLLLoss() is the negative log likelihood loss we want. It also defines optimization functions in torch.optim. Here, we will just use SGD.

Note that the input to NLLLoss is a vector of log probabilities, and a target label. It doesn't compute the log probabilities for us. This is why the last layer of our network is log softmax. The loss function nn.CrossEntropyLoss() is the same as NLLLoss(), except it does the log softmax for you.

```
# Run on test data before we train, just to see a before-and-after
with torch.no_grad():
    for instance, label in test_data:
        bow_vec = make_bow_vector(instance, word_to_ix)
        log_probs = model(bow_vec)
        print(log_probs)

# Print the matrix column corresponding to "creo"
print(next(model.parameters())[:, word_to_ix["creo"]])

loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# usually you want to pass over the training data several times
# in this case it is enough to see the results change a little bit from one pass to the
# for instance, after 5, somewhere between 5 and 50 epochs is reasonable.
for epoch in range(100):
    for instance, label in data:
        # Step 1. Remember that PyTorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Make our BOW vector and also we must wrap the target in a
        # Tensor an an integer. For example, if the target is SPANISH, then
        # we wrap the integer 0. The loss function then knows that the 0th
        # element of the log probabilities is the log probability
        # corresponding to SPANISH
        bow_vec = make_bow_vector(instance, word_to_ix)
        target = make_target(label, label_to_ix)

        # Step 3. Run our forward pass.
        log_probs = model(bow_vec)

        # Step 4. Compute the loss, gradients, and update the parameters by
        # calling optimizer.step()
        loss = loss_function(log_probs, target)
        loss.backward()
        optimizer.step()

with torch.no_grad():
    for instance, label in test_data:
        bow_vec = make_bow_vector(instance, word_to_ix)
        log_probs = model(bow_vec)
        print(log_probs)

# Index corresponding to Spanish goes up, English goes down!
print(next(model.parameters())[:, word_to_ix["creo"]])
```

Out:

```
tensor([[-0.6267, -0.9659]])
tensor([[-0.2168, -1.7500]])
tensor([[-0.1089, -0.2213]])
tensor([[ 0.2908, -1.3314]])
tensor([[-0.8313, -0.4821]])
tensor([[ 0.2681, -0.5605]])
```

We get the right answer! You can see that the log probability for Spanish is much higher in the first example, and the log probability for English is much higher in the second for the test data, as it should be.

Now you see how to make a PyTorch component, pass some data through it and do gradient updates. We are ready to dig deeper into what deep NLP has to offer.

Total running time of the script: (0 minutes 0.132 seconds)

Download Python source code: deep_learning_tutorial.py

Download Jupyter notebook: deep_learning_tutorial.ipynb

Gallery generated by Sphinx-Gallery

Previous Next

© Copyright 2017, PyTorch.

Built with Sphinx using a theme provided by Read the Docs.