pytorch_tutorial

December 16, 2018

```
In [ ]: %matplotlib inline
```

Introduction to PyTorch **********************

1 Introduction to Torch's tensor library

All of deep learning is computations on tensors, which are generalizations of a matrix that can be indexed in more than 2 dimensions. We will see exactly what this means in-depth later. First, lets look what we can do with tensors.

```
In [ ]: # Author: Robert Guthrie

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

        torch.manual_seed(1)
```

Creating Tensors ~

Tensors can be created from Python lists with the torch.Tensor() function.

```
In [ ]: # torch.tensor(data) creates a torch.Tensor object with the given data.
        V_data = [1., 2., 3.]
        V = torch.tensor(V_data)
        print(V)

        # Creates a matrix
        M_data = [[1., 2., 3.], [4., 5., 6]]
        M = torch.tensor(M_data)
        print(M)

        # Create a 3D tensor of size 2x2x2.
        T_data = [[[1., 2.], [3., 4.]],
                  [[5., 6.], [7., 8.]]]
        T = torch.tensor(T_data)
        print(T)
```

What is a 3D tensor anyway? Think about it like this. If you have a vector, indexing into the vector gives you a scalar. If you have a matrix, indexing into the matrix gives you a vector. If you have a 3D tensor, then indexing into the tensor gives you a matrix!

A note on terminology: when I say "tensor" in this tutorial, it refers to any torch.Tensor object. Matrices and vectors are special cases of torch.Tensors, where their dimension is 1 and 2 respectively. When I am talking about 3D tensors, I will explicitly use the term "3D tensor".

```
In [ ]: # Index into V and get a scalar (0 dimensional tensor)
        print(V[0])
        # Get a Python number from it
        print(V[0].item())

        # Index into M and get a vector
        print(M[0])

        # Index into T and get a matrix
        print(T[0])
```

You can also create tensors of other datatypes. The default, as you can see, is Float. To create a tensor of integer types, try torch.LongTensor(). Check the documentation for more data types, but Float and Long will be the most common.

You can create a tensor with random data and the supplied dimensionality with torch.randn()

```
In [ ]: x = torch.randn((3, 4, 5))
        print(x)
```

Operations with Tensors ~

You can operate on tensors in the ways you would expect.

```
In [ ]: x = torch.tensor([1., 2., 3.])
        y = torch.tensor([4., 5., 6.])
        z = x + y
        print(z)
```

See `the documentation <https://pytorch.org/docs/torch.html>__` for a complete list of the massive number of operations available to you. They expand beyond just mathematical operations.

One helpful operation that we will make use of later is concatenation.

```
In [ ]: # By default, it concatenates along the first axis (concatenates rows)
        x_1 = torch.randn(2, 5)
        y_1 = torch.randn(3, 5)
        z_1 = torch.cat([x_1, y_1])
        print(z_1)

        # Concatenate columns:
        x_2 = torch.randn(2, 3)
        y_2 = torch.randn(2, 5)
        # second arg specifies which axis to concat along
```

```
z_2 = torch.cat([x_2, y_2], 1)
print(z_2)

# If your tensors are not compatible, torch will complain.  Uncomment to see the error
# torch.cat([x_1, x_2])
```

Reshaping Tensors ~ ~

Use the .view() method to reshape a tensor. This method receives heavy use, because many neural network components expect their inputs to have a certain shape. Often you will need to reshape before passing your data to the component.

```
In [ ]: x = torch.randn(2, 3, 4)
        print(x)
        print(x.view(2, 12))  # Reshape to 2 rows, 12 columns
        # Same as above.  If one of the dimensions is -1, its size can be inferred
        print(x.view(2, -1))
```

2 Computation Graphs and Automatic Differentiation

The concept of a computation graph is essential to efficient deep learning programming, because it allows you to not have to write the back propagation gradients yourself. A computation graph is simply a specification of how your data is combined to give you the output. Since the graph totally specifies what parameters were involved with which operations, it contains enough information to compute derivatives. This probably sounds vague, so let's see what is going on using the fundamental flag `requires_grad`.

First, think from a programmers perspective. What is stored in the torch.Tensor objects we were creating above? Obviously the data and the shape, and maybe a few other things. But when we added two tensors together, we got an output tensor. All this output tensor knows is its data and shape. It has no idea that it was the sum of two other tensors (it could have been read in from a file, it could be the result of some other operation, etc.)

If `requires_grad=True`, the Tensor object keeps track of how it was created. Lets see it in action.

```
In [ ]: # Tensor factory methods have a ``requires_grad`` flag
        x = torch.tensor([1., 2., 3], requires_grad=True)

        # With requires_grad=True, you can still do all the operations you previously
        # could
        y = torch.tensor([4., 5., 6], requires_grad=True)
        z = x + y
        print(z)

        # BUT z knows something extra.
        print(z.grad_fn)
```

So Tensors know what created them. z knows that it wasn't read in from a file, it wasn't the result of a multiplication or exponential or whatever. And if you keep following z.grad_fn, you will find yourself at x and y.

But how does that help us compute a gradient?

```
In [ ]: # Lets sum up all the entries in z
        s = z.sum()
        print(s)
        print(s.grad_fn)
```

So now, what is the derivative of this sum with respect to the first component of x? In math, we want

$$\frac{\partial s}{\partial x_0} \tag{1}$$

Well, s knows that it was created as a sum of the tensor z. z knows that it was the sum x + y. So

$$s = \overbrace{x_0 + y_0}^{z_0} + \overbrace{x_1 + y_1}^{z_1} + \overbrace{x_2 + y_2}^{z_2} \tag{2}$$

And so s contains enough information to determine that the derivative we want is 1!

Of course this glosses over the challenge of how to actually compute that derivative. The point here is that s is carrying along enough information that it is possible to compute it. In reality, the developers of Pytorch program the sum() and + operations to know how to compute their gradients, and run the back propagation algorithm. An in-depth discussion of that algorithm is beyond the scope of this tutorial.

Lets have Pytorch compute the gradient, and see that we were right: (note if you run this block multiple times, the gradient will increment. That is because Pytorch *accumulates* the gradient into the .grad property, since for many models this is very convenient.)

```
In [ ]: # calling .backward() on any variable will run backprop, starting from it.
        s.backward()
        print(x.grad)
```

Understanding what is going on in the block below is crucial for being a successful programmer in deep learning.

```
In [ ]: x = torch.randn(2, 2)
        y = torch.randn(2, 2)
        # By default, user created Tensors have ``requires_grad=False``
        print(x.requires_grad, y.requires_grad)
        z = x + y
        # So you can't backprop through z
        print(z.grad_fn)

        # ``.requires_grad_( ... )`` changes an existing Tensor's ``requires_grad``
        # flag in-place. The input flag defaults to ``True`` if not given.
        x = x.requires_grad_()
        y = y.requires_grad_()
        # z contains enough information to compute gradients, as we saw above
        z = x + y
        print(z.grad_fn)
        # If any input to an operation has ``requires_grad=True``, so will the output
        print(z.requires_grad)

        # Now z has the computation history that relates itself to x and y
        # Can we just take its values, and **detach** it from its history?
        new_z = z.detach()

        # ... does new_z have information to backprop to x and y?
        # NO!
        print(new_z.grad_fn)
        # And how could it? ``z.detach()`` returns a tensor that shares the same storage
        # as ``z``, but with the computation history forgotten. It doesn't know anything
        # about how it was computed.
        # In essence, we have broken the Tensor away from its past history
```

You can also stop autograd from tracking history on Tensors with `.requires_grad`=True by wrapping the code block in `with torch.no_grad()`:

```
In [ ]: print(x.requires_grad)
        print((x ** 2).requires_grad)

        with torch.no_grad():
                print((x ** 2).requires_grad)
```

deep_learning_tutorial

December 16, 2018

```
In [ ]: %matplotlib inline
```

Deep Learning with PyTorch ************************

1 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

$$f(x) = Ax + b \tag{1}$$

for a matrix A and vectors x, b. The parameters to be learned here are A and b. Often, b is refered 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 i'th row of the output below is the mapping of the i'th row of the input under A, plus the bias term. Look at the example below.

```
In [ ]: # 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)

In [ ]: 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
```

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 $f(g(x))$?

$$f(g(x)) = A(Cx + d) + b = ACx + (Ad + b) \tag{2}$$

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. $\tanh(x), \sigma(x), \text{ReLU}(x)$ 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

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x)) \tag{3}$$

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 [ ]: # In pytorch, most non-linearities are in torch.functional (we have it imported as F)
        # Note that non-linearites typically don't have parameters like affine maps do.
        # That is, they don't have weights that are updated during training.
        data = torch.randn(2, 2)
        print(data)
        print(F.relu(data))
```

Softmax and Probabilities

The function $\text{Softmax}(x)$ 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 i'th component of $\text{Softmax}(x)$ is

$$\frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{4}$$

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.

```
In [ ]: # 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))  # theres also log_softmax
```

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).

2 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, $L(\theta)$ the loss function, and η a positive learning rate. Then:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta) \tag{5}$$

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 the 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.

3 Creating Network Components in PyTorch

Before we move on to our focus on NLP, lets do 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 boilerplate 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

$$[4,0] \tag{6}$$

For "hello world world hello", it is

$$[2,2] \tag{7}$$

etc. In general, it is

$$[\text{Count}(\text{hello}), \text{Count}(\text{world})] \tag{8}$$

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

$$\log \text{Softmax}(Ax+b) \tag{9}$$

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

```
In [ ]: data = [("me gusta comer en la cafeteria".split(), "SPANISH"),
                 ("Give it to me".split(), "ENGLISH"),
                 ("No creo que sea una buena idea".split(), "SPANISH"),
                 ("No it is not a good idea to get lost at sea".split(), "ENGLISH")]

        test_data = [("Yo creo que si".split(), "SPANISH"),
                      ("it is lost on me".split(), "ENGLISH")]

        # word_to_ix maps each word in the vocab to a unique integer, which will be its
        # index into the Bag of words vector
        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):  # inheriting 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 an nn.Module
        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 the input dimension is vocab_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 ne
        # to worry about that here

    def forward(self, bow_vec):
        # Pass the input through the linear layer,
        # then pass that through log_softmax.
        # 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 component to a class variable in the __init__ function
# of a module, which was done with the line
# self.linear = nn.Linear(...)
# Then through some Python magic from the PyTorch devs, 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)
```

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

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

So lets 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. nn.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.

```
In [ ]: # 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.
        # 100 is much bigger than on a real data set, but real datasets have more than
        # two instances.  Usually, somewhere between 5 and 30 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 as 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"]])
```

We got 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.

word_embeddings_tutorial

December 16, 2018

```
In [ ]: %matplotlib inline
```

1 Word Embeddings: Encoding Lexical Semantics

Word embeddings are dense vectors of real numbers, one per word in your vocabulary. In NLP, it is almost always the case that your features are words! But how should you represent a word in a computer? You could store its ascii character representation, but that only tells you what the word *is*, it doesn't say much about what it *means* (you might be able to derive its part of speech from its affixes, or properties from its capitalization, but not much). Even more, in what sense could you combine these representations? We often want dense outputs from our neural networks, where the inputs are $|V|$ dimensional, where V is our vocabulary, but often the outputs are only a few dimensional (if we are only predicting a handful of labels, for instance). How do we get from a massive dimensional space to a smaller dimensional space?

How about instead of ascii representations, we use a one-hot encoding? That is, we represent the word w by

$$\overbrace{[0, 0, \dots, 1, \dots, 0, 0]}^{\text{|V| elements}} \tag{1}$$

where the 1 is in a location unique to w. Any other word will have a 1 in some other location, and a 0 everywhere else.

There is an enormous drawback to this representation, besides just how huge it is. It basically treats all words as independent entities with no relation to each other. What we really want is some notion of *similarity* between words. Why? Let's see an example.

Suppose we are building a language model. Suppose we have seen the sentences

- The mathematician ran to the store.
- The physicist ran to the store.
- The mathematician solved the open problem.

in our training data. Now suppose we get a new sentence never before seen in our training data:

- The physicist solved the open problem.

Our language model might do OK on this sentence, but wouldn't it be much better if we could use the following two facts:

sequence_models_tutorial

December 16, 2018

```
In [ ]: %matplotlib inline
```

1 Sequence Models and Long-Short Term Memory Networks

At this point, we have seen various feed-forward networks. That is, there is no state maintained by the network at all. This might not be the behavior we want. Sequence models are central to NLP: they are models where there is some sort of dependence through time between your inputs. The classical example of a sequence model is the Hidden Markov Model for part-of-speech tagging. Another example is the conditional random field.

A recurrent neural network is a network that maintains some kind of state. For example, its output could be used as part of the next input, so that information can propogate along as the network passes over the sequence. In the case of an LSTM, for each element in the sequence, there is a corresponding *hidden state* h_t, which in principle can contain information from arbitrary points earlier in the sequence. We can use the hidden state to predict words in a language model, part-of-speech tags, and a myriad of other things.

LSTM's in Pytorch ~ ~ ~

Before getting to the example, note a few things. Pytorch's LSTM expects all of its inputs to be 3D tensors. The semantics of the axes of these tensors is important. The first axis is the sequence itself, the second indexes instances in the mini-batch, and the third indexes elements of the input. We haven't discussed mini-batching, so lets just ignore that and assume we will always have just 1 dimension on the second axis. If we want to run the sequence model over the sentence "The cow jumped", our input should look like

$$\begin{bmatrix} \overbrace{q_\text{The}}^\text{row vector} \\ q_\text{cow} \\ q_\text{jumped} \end{bmatrix} \tag{1}$$

Except remember there is an additional 2nd dimension with size 1.

In addition, you could go through the sequence one at a time, in which case the 1st axis will have size 1 also.

Let's see a quick example.

```
In [ ]: # 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)
```

```
In [ ]: lstm = nn.LSTM(3, 3)  # Input dim is 3, output dim is 3
        inputs = [torch.randn(1, 3) for _ in range(5)]  # make a sequence of length 5

        # initialize the hidden state.
        hidden = (torch.randn(1, 1, 3),
                  torch.randn(1, 1, 3))
        for i in inputs:
            # Step through the sequence one element at a time.
            # after each step, hidden contains the hidden state.
            out, hidden = lstm(i.view(1, 1, -1), hidden)

        # alternatively, we can do the entire sequence all at once.
        # the first value returned by LSTM is all of the hidden states throughout
        # the sequence. the second is just the most recent hidden state
        # (compare the last slice of "out" with "hidden" below, they are the same)
        # The reason for this is that:
        # "out" will give you access to all hidden states in the sequence
        # "hidden" will allow you to continue the sequence and backpropagate,
        # by passing it as an argument  to the lstm at a later time
        # Add the extra 2nd dimension
        inputs = torch.cat(inputs).view(len(inputs), 1, -1)
        hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # clean out hidden state
        out, hidden = lstm(inputs, hidden)
        print(out)
        print(hidden)
```

Example: An LSTM for Part-of-Speech Tagging

In this section, we will use an LSTM to get part of speech tags. We will not use Viterbi or Forward-Backward or anything like that, but as a (challenging) exercise to the reader, think about how Viterbi could be used after you have seen what is going on.

The model is as follows: let our input sentence be $w_1, \ldots, w_M$, where $w_i \in V$, our vocab. Also, let T be our tag set, and y_i the tag of word w_i. Denote our prediction of the tag of word w_i by $\hat{y}_i$.

This is a structure prediction, model, where our output is a sequence $\hat{y}_1, \ldots, \hat{y}_M$, where $\hat{y}_i \in T$.

To do the prediction, pass an LSTM over the sentence. Denote the hidden state at timestep i as h_i. Also, assign each tag a unique index (like how we had word_to_ix in the word embeddings section). Then our prediction rule for $\hat{y}_i$ is

$$\hat{y}_i = \text{argmax}_j \left(\log \text{Softmax}(Ah_i + b)\right)_j \tag{2}$$

That is, take the log softmax of the affine map of the hidden state, and the predicted tag is the tag that has the maximum value in this vector. Note this implies immediately that the dimensionality of the target space of A is $|T|$.

Prepare data:

```
In [ ]: def prepare_sequence(seq, to_ix):
            idxs = [to_ix[w] for w in seq]
            return torch.tensor(idxs, dtype=torch.long)


        training_data = [
            ("The dog ate the apple".split(), ["DET", "NN", "V", "DET", "NN"]),
            ("Everybody read that book".split(), ["NN", "V", "DET", "NN"])
        ]
        word_to_ix = {}
        for sent, tags in training_data:
            for word in sent:
                if word not in word_to_ix:
                    word_to_ix[word] = len(word_to_ix)
        print(word_to_ix)
        tag_to_ix = {"DET": 0, "NN": 1, "V": 2}

        # These will usually be more like 32 or 64 dimensional.
        # We will keep them small, so we can see how the weights change as we train.
        EMBEDDING_DIM = 6
        HIDDEN_DIM = 6
```

Create the model:

```
In [ ]: class LSTMTagger(nn.Module):

            def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
                super(LSTMTagger, self).__init__()
                self.hidden_dim = hidden_dim

                self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

                # The LSTM takes word embeddings as inputs, and outputs hidden states
                # with dimensionality hidden_dim.
                self.lstm = nn.LSTM(embedding_dim, hidden_dim)

                # The linear layer that maps from hidden state space to tag space
                self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
                self.hidden = self.init_hidden()

            def init_hidden(self):
                # Before we've done anything, we dont have any hidden state.
                # Refer to the Pytorch documentation to see exactly
                # why they have this dimensionality.
                # The axes semantics are (num_layers, minibatch_size, hidden_dim)
                return (torch.zeros(1, 1, self.hidden_dim),
                        torch.zeros(1, 1, self.hidden_dim))

            def forward(self, sentence):
                embeds = self.word_embeddings(sentence)
                lstm_out, self.hidden = self.lstm(
                    embeds.view(len(sentence), 1, -1), self.hidden)
                tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
                tag_scores = F.log_softmax(tag_space, dim=1)
                return tag_scores
```

Train the model:

```
In [ ]: model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_ix), len(tag_to_ix))
        loss_function = nn.NLLLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.1)

        # See what the scores are before training
        # Note that element i,j of the output is the score for tag j for word i.
        # Here we don't need to train, so the code is wrapped in torch.no_grad()
        with torch.no_grad():
            inputs = prepare_sequence(training_data[0][0], word_to_ix)
            tag_scores = model(inputs)
            print(tag_scores)

        for epoch in range(300):  # again, normally you would NOT do 300 epochs, it is toy dat
            for sentence, tags in training_data:
                # Step 1. Remember that Pytorch accumulates gradients.
                # We need to clear them out before each instance
                model.zero_grad()

                # Also, we need to clear out the hidden state of the LSTM,
                # detaching it from its history on the last instance.
                model.hidden = model.init_hidden()

                # Step 2. Get our inputs ready for the network, that is, turn them into
                # Tensors of word indices.
                sentence_in = prepare_sequence(sentence, word_to_ix)
                targets = prepare_sequence(tags, tag_to_ix)

                # Step 3. Run our forward pass.
                tag_scores = model(sentence_in)

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

```
# See what the scores are after training
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)

    # The sentence is "the dog ate the apple".  i,j corresponds to score for tag j
    # for word i. The predicted tag is the maximum scoring tag.
    # Here, we can see the predicted sequence below is 0 1 2 0 1
    # since 0 is index of the maximum value of row 1,
    # 1 is the index of maximum value of row 2, etc.
    # Which is DET NOUN VERB DET NOUN, the correct sequence!
    print(tag_scores)
```

Exercise: Augmenting the LSTM part-of-speech tagger with character-level features

In the example above, each word had an embedding, which served as the inputs to our sequence model. Let's augment the word embeddings with a representation derived from the characters of the word. We expect that this should help significantly, since character-level information like affixes have a large bearing on part-of-speech. For example, words with the affix *-ly* are almost always tagged as adverbs in English.

To do this, let c_w be the character-level representation of word w. Let x_w be the word embedding as before. Then the input to our sequence model is the concatenation of x_w and c_w. So if x_w has dimension 5, and c_w dimension 3, then our LSTM should accept an input of dimension 8.

To get the character level representation, do an LSTM over the characters of a word, and let c_w be the final hidden state of this LSTM. Hints:

- There are going to be two LSTM's in your new model. The original one that outputs POS tag scores, and the new one that outputs a character-level representation of each word.
- To do a sequence model over characters, you will have to embed characters. The character embeddings will be the input to the character LSTM.

BiLSTMCRF-torch

December 16, 2018

```
In [ ]: %matplotlib inline
```

1 Advanced: Making Dynamic Decisions and the Bi-LSTM CRF

1.1 Dynamic versus Static Deep Learning Toolkits

Pytorch is a *dynamic* neural network kit. Another example of a dynamic kit is `Dynet <https://github.com/clab/dynet>`__ (I mention this because working with Pytorch and Dynet is similar. If you see an example in Dynet, it will probably help you implement it in Pytorch). The opposite is the *static* tool kit, which includes Theano, Keras, TensorFlow, etc. The core difference is the following:

- In a static toolkit, you define a computation graph once, compile it, and then stream instances to it.
- In a dynamic toolkit, you define a computation graph *for each instance*. It is never compiled and is executed on-the-fly

Without a lot of experience, it is difficult to appreciate the difference. One example is to suppose we want to build a deep constituent parser. Suppose our model involves roughly the following steps:

- We build the tree bottom up
- Tag the root nodes (the words of the sentence)
- From there, use a neural network and the embeddings of the words to find combinations that form constituents. Whenever you form a new constituent, use some sort of technique to get an embedding of the constituent. In this case, our network architecture will depend completely on the input sentence. In the sentence "The green cat scratched the wall", at some point in the model, we will want to combine the span $(i, j, r) = (1, 3, \text{NP})$ (that is, an NP constituent spans word 1 to word 3, in this case "The green cat").

However, another sentence might be "Somewhere, the big fat cat scratched the wall". In this sentence, we will want to form the constituent $(2, 4, NP)$ at some point. The constituents we will want to form will depend on the instance. If we just compile the computation graph once, as in a static toolkit, it will be exceptionally difficult or impossible to program this logic. In a dynamic toolkit though, there isn't just 1 pre-defined computation graph. There can be a new computation graph for each instance, so this problem goes away.

Dynamic toolkits also have the advantage of being easier to debug and the code more closely resembling the host language (by that I mean that Pytorch and Dynet look more like actual Python code than Keras or Theano).

1.2 Bi-LSTM Conditional Random Field Discussion

For this section, we will see a full, complicated example of a Bi-LSTM Conditional Random Field for named-entity recognition. The LSTM tagger above is typically sufficient for part-of-speech tagging, but a sequence model like the CRF is really essential for strong performance on NER. Familiarity with CRF's is assumed. Although this name sounds scary, all the model is is a CRF but where an LSTM provides the features. This is an advanced model though, far more complicated than any earlier model in this tutorial. If you want to skip it, that is fine. To see if you're ready, see if you can:

- Write the recurrence for the viterbi variable at step i for tag k.
- Modify the above recurrence to compute the forward variables instead.
- Modify again the above recurrence to compute the forward variables in log-space (hint: log-sum-exp)

If you can do those three things, you should be able to understand the code below. Recall that the CRF computes a conditional probability. Let y be a tag sequence and x an input sequence of words. Then we compute

$$P(y|x) = \frac{\exp\left(\text{Score}(x, y)\right)}{\sum_{y'} \exp\left(\text{Score}(x, y')\right)} \tag{1}$$

Where the score is determined by defining some log potentials $\log \psi_i(x, y)$ such that

$$\text{Score}(x, y) = \sum_i \log \psi_i(x, y) \tag{2}$$

To make the partition function tractable, the potentials must look only at local features.

In the Bi-LSTM CRF, we define two kinds of potentials: emission and transition. The emission potential for the word at index i comes from the hidden state of the Bi-LSTM at timestep i. The transition scores are stored in a $|T|x|T|$ matrix $\mathbf{P}$, where T is the tag set. In my implementation, $\mathbf{P}_{j,k}$ is the score of transitioning to tag j from tag k. So:

$$\text{Score}(x, y) = \sum_i \log \psi_{\text{EMIT}}(y_i \rightarrow x_i) + \log \psi_{\text{TRANS}}(y_{i-1} \rightarrow y_i) \tag{3}$$

$$= \sum_i h_i[y_i] + \mathbf{P}_{y_i, y_{i-1}} \tag{4}$$

where in this second expression, we think of the tags as being assigned unique non-negative indices.

If the above discussion was too brief, you can check out `this <http://www.cs.columbia.edu/%7Emcollins/crf.pdf>__` write up from Michael Collins on CRFs.

1.3 Implementation Notes

The example below implements the forward algorithm in log space to compute the partition function, and the viterbi algorithm to decode. Backpropagation will compute the gradients automatically for us. We don't have to do anything by hand.

The implementation is not optimized. If you understand what is going on, you'll probably quickly see that iterating over the next tag in the forward algorithm could probably be done in one big operation. I wanted to code to be more readable. If you want to make the relevant change, you could probably use this tagger for real tasks.

```
In [ ]: # Author: Robert Guthrie

        import torch
        import torch.autograd as autograd
        import torch.nn as nn
        import torch.optim as optim

        torch.manual_seed(1)
```

Helper functions to make the code more readable.

```
In [ ]: def argmax(vec):
            # return the argmax as a python int
            _, idx = torch.max(vec, 1)
            return idx.item()


        def prepare_sequence(seq, to_ix):
            idxs = [to_ix[w] for w in seq]
            return torch.tensor(idxs, dtype=torch.long)


        # Compute log sum exp in a numerically stable way for the forward algorithm
        def log_sum_exp(vec):
            max_score = vec[0, argmax(vec)]
            max_score_broadcast = max_score.view(1, -1).expand(1, vec.size()[1])
            return max_score + \
                torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))
```

Create model

```
In [ ]: class BiLSTM_CRF(nn.Module):

            def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
                super(BiLSTM_CRF, self).__init__()
                self.embedding_dim = embedding_dim
                self.hidden_dim = hidden_dim
                self.vocab_size = vocab_size
                self.tag_to_ix = tag_to_ix
```

```
        self.tagset_size = len(tag_to_ix)

        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # Maps the output of the LSTM into tag space.
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # Matrix of transition parameters.  Entry i,j is the score of
        # transitioning *to* i *from* j.
        self.transitions = nn.Parameter(
            torch.randn(self.tagset_size, self.tagset_size))

        # These two statements enforce the constraint that we never transfer
        # to the start tag and we never transfer from the stop tag
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000
        self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

        self.hidden = self.init_hidden()

    def init_hidden(self):
        return (torch.randn(2, 1, self.hidden_dim // 2),
                torch.randn(2, 1, self.hidden_dim // 2))

    def _forward_alg(self, feats):
        # Do the forward algorithm to compute the partition function
        init_alphas = torch.full((1, self.tagset_size), -10000.)
        # START_TAG has all of the score.
        init_alphas[0][self.tag_to_ix[START_TAG]] = 0.

        # Wrap in a variable so that we will get automatic backprop
        forward_var = init_alphas

        # Iterate through the sentence
        for feat in feats:
            alphas_t = []  # The forward tensors at this timestep
            for next_tag in range(self.tagset_size):
                # broadcast the emission score: it is the same regardless of
                # the previous tag
                emit_score = feat[next_tag].view(
                    1, -1).expand(1, self.tagset_size)
                # the ith entry of trans_score is the score of transitioning to
                # next_tag from i
                trans_score = self.transitions[next_tag].view(1, -1)
                # The ith entry of next_tag_var is the value for the
                # edge (i -> next_tag) before we do log-sum-exp
                next_tag_var = forward_var + trans_score + emit_score
```

```python
                # The forward variable for this tag is log-sum-exp of all the
                # scores.
                alphas_t.append(log_sum_exp(next_tag_var).view(1))
            forward_var = torch.cat(alphas_t).view(1, -1)
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        alpha = log_sum_exp(terminal_var)
        return alpha

    def _get_lstm_features(self, sentence):
        self.hidden = self.init_hidden()
        embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
        lstm_out, self.hidden = self.lstm(embeds, self.hidden)
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, tags):
        # Gives the score of a provided tag sequence
        score = torch.zeros(1)
        tags = torch.cat([torch.tensor([self.tag_to_ix[START_TAG]], dtype=torch.long),
        for i, feat in enumerate(feats):
            score = score + \
                self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = []

        # Initialize the viterbi variables in log space
        init_vvars = torch.full((1, self.tagset_size), -10000.)
        init_vvars[0][self.tag_to_ix[START_TAG]] = 0

        # forward_var at step i holds the viterbi variables for step i-1
        forward_var = init_vvars
        for feat in feats:
            bptrs_t = []  # holds the backpointers for this step
            viterbivars_t = []  # holds the viterbi variables for this step

            for next_tag in range(self.tagset_size):
                # next_tag_var[i] holds the viterbi variable for tag i at the
                # previous step, plus the score of transitioning
                # from tag i to next_tag.
                # We don't include the emission scores here because the max
                # does not depend on them (we add them in below)
                next_tag_var = forward_var + self.transitions[next_tag]
                best_tag_id = argmax(next_tag_var)
                bptrs_t.append(best_tag_id)
```

```
                    viterbivars_t.append(next_tag_var[0][best_tag_id].view(1))
                # Now add in the emission scores, and assign forward_var to the set
                # of viterbi variables we just computed
                forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
                backpointers.append(bptrs_t)

            # Transition to STOP_TAG
            terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
            best_tag_id = argmax(terminal_var)
            path_score = terminal_var[0][best_tag_id]

            # Follow the back pointers to decode the best path.
            best_path = [best_tag_id]
            for bptrs_t in reversed(backpointers):
                best_tag_id = bptrs_t[best_tag_id]
                best_path.append(best_tag_id)
            # Pop off the start tag (we dont want to return that to the caller)
            start = best_path.pop()
            assert start == self.tag_to_ix[START_TAG]  # Sanity check
            best_path.reverse()
            return path_score, best_path

        def neg_log_likelihood(self, sentence, tags):
            feats = self._get_lstm_features(sentence)
            forward_score = self._forward_alg(feats)
            gold_score = self._score_sentence(feats, tags)
            return forward_score - gold_score

        def forward(self, sentence):  # dont confuse this with _forward_alg above.
            # Get the emission scores from the BiLSTM
            lstm_feats = self._get_lstm_features(sentence)

            # Find the best path, given the features.
            score, tag_seq = self._viterbi_decode(lstm_feats)
            return score, tag_seq
```

Run training

```
In [ ]: START_TAG = "<START>"
        STOP_TAG = "<STOP>"
        EMBEDDING_DIM = 5
        HIDDEN_DIM = 4

        # Make up some training data
        training_data = [(
            "the wall street journal reported today that apple corporation made money".split()
            "B I I I O O O B I O O".split()
        ), (
    "georgia tech is a university in georgia".split(),
    "B I O O O O B".split()
)]

word_to_ix = {}
for sentence, tags in training_data:
    for word in sentence:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)

tag_to_ix = {"B": 0, "I": 1, "O": 2, START_TAG: 3, STOP_TAG: 4}

model = BiLSTM_CRF(len(word_to_ix), tag_to_ix, EMBEDDING_DIM, HIDDEN_DIM)
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)

# Check predictions before training
with torch.no_grad():
    precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
    precheck_tags = torch.tensor([tag_to_ix[t] for t in training_data[0][1]], dtype=to
    print(model(precheck_sent))

# Make sure prepare_sequence from earlier in the LSTM section is loaded
for epoch in range(
        300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is,
        # turn them into Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = torch.tensor([tag_to_ix[t] for t in tags], dtype=torch.long)

        # Step 3. Run our forward pass.
        loss = model.neg_log_likelihood(sentence_in, targets)

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

# Check predictions after training
with torch.no_grad():
    precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
    print(model(precheck_sent))
# We got it!
```

1.4 Exercise: A new loss function for discriminative tagging

It wasn't really necessary for us to create a computation graph when doing decoding, since we do not backpropagate from the viterbi path score. Since we have it anyway, try training the tagger where the loss function is the difference between the Viterbi path score and the score of the gold-standard path. It should be clear that this function is non-negative and 0 when the predicted tag sequence is the correct tag sequence. This is essentially *structured perceptron*.

This modification should be short, since Viterbi and score_sentence are already implemented. This is an example of the shape of the computation graph *depending on the training instance*. Although I haven't tried implementing this in a static toolkit, I imagine that it is possible but much less straightforward.

Pick up some real data and do a comparison!

seq2seq_translation_tutorial

December 16, 2018

```
In [ ]: %matplotlib inline
```

Translation with a Sequence to Sequence Network and Attention *** **Author**: `Sean Robertson <https://github.com/spro/practical-pytorch>`_

In this project we will be teaching a neural network to translate from French to English.

::

```
[KEY: > input, = target, < output]

> il est en train de peindre un tableau .
= he is painting a picture .
< he is painting a picture .

> pourquoi ne pas essayer ce vin delicieux ?
= why not try that delicious wine ?
< why not try that delicious wine ?

> elle n est pas poete mais romanciere .
= she is not a poet but a novelist .
< she not not a poet but a novelist .

> vous etes trop maigre .
= you re too skinny .
< you re all alone .
```

... to varying degrees of success.

This is made possible by the simple but powerful idea of the `sequence to sequence network <http://arxiv.org/abs/1409.3215>`__, in which two recurrent neural networks work together to transform one sequence to another. An encoder network condenses an input sequence into a vector, and a decoder network unfolds that vector into a new sequence.

.. figure:: /_static/img/seq-seq-images/seq2seq.png :alt:

To improve upon this model we'll use an `attention mechanism <https://arxiv.org/abs/1409.0473>`__, which lets the decoder learn to focus over a specific range of the input sequence.

Recommended Reading:

I assume you have at least installed PyTorch, know Python, and understand Tensors:

- https://pytorch.org/ For installation instructions
- :doc:`/beginner/deep_learning_60min_blitz` to get started with PyTorch in general
- :doc:`/beginner/pytorch_with_examples` for a wide and deep overview
- :doc:`/beginner/former_torchies_tutorial` if you are former Lua Torch user

It would also be useful to know about Sequence to Sequence networks and how they work:

- `Learning Phrase Representations using RNN Encoder-Decoder for    Statistical Machine Translation <http://arxiv.org/abs/1406.1078>__`
- `Sequence to Sequence Learning with Neural    Networks <http://arxiv.org/abs/1409.3215>__`
- `Neural Machine Translation by Jointly Learning to Align and    Translate <https://arxiv.org/abs/1409.0473>__`
- `A Neural Conversational Model <http://arxiv.org/abs/1506.05869>__`

You will also find the previous tutorials on :doc:`/intermediate/char_rnn_classification_tutorial` and :doc:`/intermediate/char_rnn_generation_tutorial` helpful as those concepts are very similar to the Encoder and Decoder models, respectively.

And for more, read the papers that introduced these topics:

- `Learning Phrase Representations using RNN Encoder-Decoder for    Statistical Machine Translation <http://arxiv.org/abs/1406.1078>__`
- `Sequence to Sequence Learning with Neural    Networks <http://arxiv.org/abs/1409.3215>__`
- `Neural Machine Translation by Jointly Learning to Align and    Translate <https://arxiv.org/abs/1409.0473>__`
- `A Neural Conversational Model <http://arxiv.org/abs/1506.05869>__`

Requirements

```
In [ ]: from __future__ import unicode_literals, print_function, division
        from io import open
        import unicodedata
        import string
        import re
        import random

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

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

1 Loading data files

The data for this project is a set of many thousands of English to French translation pairs.

`This question on Open Data Stack Exchange <http://opendata.stackexchange.com/questions/3888/` pointed me to the open translation site http://tatoeba.org/ which has downloads available at

http://tatoeba.org/eng/downloads - and better yet, someone did the extra work of splitting language pairs into individual text files here: http://www.manythings.org/anki/

The English to French pairs are too big to include in the repo, so download to `data/eng-fra.txt` before continuing. The file is a tab separated list of translation pairs:

::

```
I am cold.    J'ai froid.
```

.. Note:: Download the data from `here <https://download.pytorch.org/tutorial/data.zip>_` and extract it to the current directory.

Similar to the character encoding used in the character-level RNN tutorials, we will be representing each word in a language as a one-hot vector, or giant vector of zeros except for a single one (at the index of the word). Compared to the dozens of characters that might exist in a language, there are many many more words, so the encoding vector is much larger. We will however cheat a bit and trim the data to only use a few thousand words per language.

.. figure:: /_static/img/seq-seq-images/word-encoding.png :alt:

We'll need a unique index per word to use as the inputs and targets of the networks later. To keep track of all this we will use a helper class called `Lang` which has word index (`word2index`) and index word (`index2word`) dictionaries, as well as a count of each word `word2count` to use to later replace rare words.

```
In [ ]: SOS_token = 0
        EOS_token = 1


        class Lang:
            def __init__(self, name):
                self.name = name
                self.word2index = {}
                self.word2count = {}
                self.index2word = {0: "SOS", 1: "EOS"}
                self.n_words = 2  # Count SOS and EOS

            def addSentence(self, sentence):
                for word in sentence.split(' '):
                    self.addWord(word)

            def addWord(self, word):
                if word not in self.word2index:
                    self.word2index[word] = self.n_words
                    self.word2count[word] = 1
                    self.index2word[self.n_words] = word
                    self.n_words += 1
                else:
                    self.word2count[word] += 1
```

The files are all in Unicode, to simplify we will turn Unicode characters to ASCII, make everything lowercase, and trim most punctuation.

```
In [ ]: # Turn a Unicode string to plain ASCII, thanks to
        # http://stackoverflow.com/a/518232/2809427
        def unicodeToAscii(s):
            return ''.join(
                c for c in unicodedata.normalize('NFD', s)
                if unicodedata.category(c) != 'Mn'
            )

        # Lowercase, trim, and remove non-letter characters


        def normalizeString(s):
            s = unicodeToAscii(s.lower().strip())
            s = re.sub(r"([.!?])", r" \1", s)
            s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
            return s
```

To read the data file we will split the file into lines, and then split lines into pairs. The files are all English Other Language, so if we want to translate from Other Language English I added the `reverse` flag to reverse the pairs.

```
In [ ]: def readLangs(lang1, lang2, reverse=False):
            print("Reading lines...")

            # Read the file and split into lines
            lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
                read().strip().split('\n')

            # Split every line into pairs and normalize
            pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

            # Reverse pairs, make Lang instances
            if reverse:
                pairs = [list(reversed(p)) for p in pairs]
                input_lang = Lang(lang2)
                output_lang = Lang(lang1)
            else:
                input_lang = Lang(lang1)
                output_lang = Lang(lang2)

            return input_lang, output_lang, pairs
```

Since there are a *lot* of example sentences and we want to train something quickly, we'll trim the data set to only relatively short and simple sentences. Here the maximum length is 10 words (that includes ending punctuation) and we're filtering to sentences that translate to the form "I am" or "He is" etc. (accounting for apostrophes replaced earlier).

```
In [ ]: MAX_LENGTH = 10
```

```
eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)


def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

The full process for preparing the data is:

- Read text file and split into lines, split lines into pairs
- Normalize text, filter by length and content
- Make word lists from sentences in pairs

```
In [ ]: def prepareData(lang1, lang2, reverse=False):
            input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
            print("Read %s sentence pairs" % len(pairs))
            pairs = filterPairs(pairs)
            print("Trimmed to %s sentence pairs" % len(pairs))
            print("Counting words...")
            for pair in pairs:
                input_lang.addSentence(pair[0])
                output_lang.addSentence(pair[1])
            print("Counted words:")
            print(input_lang.name, input_lang.n_words)
            print(output_lang.name, output_lang.n_words)
            return input_lang, output_lang, pairs


        input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
        print(random.choice(pairs))
```

2 The Seq2Seq Model

A Recurrent Neural Network, or RNN, is a network that operates on a sequence and uses its own output as input for subsequent steps.

A `Sequence to Sequence network <http://arxiv.org/abs/1409.3215>`, **or seq2seq network, or** `Encoder Decoder network <https://arxiv.org/pdf/1406.1078v3.pdf>`, is a model consisting of two RNNs called the encoder and decoder. The encoder reads an input sequence and outputs a single vector, and the decoder reads that vector to produce an output sequence.

.. figure:: /_static/img/seq-seq-images/seq2seq.png :alt:

Unlike sequence prediction with a single RNN, where every input corresponds to an output, the seq2seq model frees us from sequence length and order, which makes it ideal for translation between two languages.

Consider the sentence "Je ne suis pas le chat noir" "I am not the black cat". Most of the words in the input sentence have a direct translation in the output sentence, but are in slightly different orders, e.g. "chat noir" and "black cat". Because of the "ne/pas" construction there is also one more word in the input sentence. It would be difficult to produce a correct translation directly from the sequence of input words.

With a seq2seq model the encoder creates a single vector which, in the ideal case, encodes the "meaning" of the input sequence into a single vector — a single point in some N dimensional space of sentences.

2.1 The Encoder

The encoder of a seq2seq network is a RNN that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state, and uses the hidden state for the next input word.

.. figure:: /_static/img/seq-seq-images/encoder-network.png :alt:

```
In [ ]: class EncoderRNN(nn.Module):
            def __init__(self, input_size, hidden_size):
                super(EncoderRNN, self).__init__()
                self.hidden_size = hidden_size

                self.embedding = nn.Embedding(input_size, hidden_size)
                self.gru = nn.GRU(hidden_size, hidden_size)

            def forward(self, input, hidden):
                embedded = self.embedding(input).view(1, 1, -1)
                output = embedded
                output, hidden = self.gru(output, hidden)
                return output, hidden

            def initHidden(self):
                return torch.zeros(1, 1, self.hidden_size, device=device)
```

2.2 The Decoder

The decoder is another RNN that takes the encoder output vector(s) and outputs a sequence of words to create the translation.

Simple Decoder ˆ

In the simplest seq2seq decoder we use only last output of the encoder. This last output is sometimes called the *context vector* as it encodes context from the entire sequence. This context vector is used as the initial hidden state of the decoder.

At every step of decoding, the decoder is given an input token and hidden state. The initial input token is the start-of-string `<SOS>` token, and the first hidden state is the context vector (the encoder's last hidden state).

.. figure:: /_static/img/seq-seq-images/decoder-network.png :alt:

```
In [ ]: class DecoderRNN(nn.Module):
            def __init__(self, hidden_size, output_size):
                super(DecoderRNN, self).__init__()
                self.hidden_size = hidden_size

                self.embedding = nn.Embedding(output_size, hidden_size)
                self.gru = nn.GRU(hidden_size, hidden_size)
                self.out = nn.Linear(hidden_size, output_size)
                self.softmax = nn.LogSoftmax(dim=1)

            def forward(self, input, hidden):
                output = self.embedding(input).view(1, 1, -1)
                output = F.relu(output)
                output, hidden = self.gru(output, hidden)
                output = self.softmax(self.out(output[0]))
                return output, hidden

            def initHidden(self):
                return torch.zeros(1, 1, self.hidden_size, device=device)
```

I encourage you to train and observe the results of this model, but to save space we'll be going straight for the gold and introducing the Attention Mechanism.

Attention Decoder ˆ ˆ ˆ

If only the context vector is passed betweeen the encoder and decoder, that single vector carries the burden of encoding the entire sentence.

Attention allows the decoder network to "focus" on a different part of the encoder's outputs for every step of the decoder's own outputs. First we calculate a set of *attention weights*. These will be multiplied by the encoder output vectors to create a weighted combination. The result (called `attn_applied` in the code) should contain information about that specific part of the input sequence, and thus help the decoder choose the right output words.

.. figure:: https://i.imgur.com/1152PYf.png :alt:

Calculating the attention weights is done with another feed-forward layer `attn`, using the decoder's input and hidden state as inputs. Because there are sentences of all sizes in the training data, to actually create and train this layer we have to choose a maximum sentence length (input length, for encoder outputs) that it can apply to. Sentences of the maximum length will use all the attention weights, while shorter sentences will only use the first few.

.. figure:: /_static/img/seq-seq-images/attention-decoder-network.png :alt:

```
In [ ]: class AttnDecoderRNN(nn.Module):
            def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH)
```

```
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

Note

There are other forms of attention that work around the length limitation by using a relative position approach. Read about "local attention" in `Effective Approaches to Attention-based Neural Machine Translation <https://arxiv.org/abs/1508.04025>`__.

3 Training

3.1 Preparing Training Data

To train, for each pair we will need an input tensor (indexes of the words in the input sentence) and target tensor (indexes of the words in the target sentence). While creating these vectors we will append the EOS token to both sequences.

```
In [ ]: def indexesFromSentence(lang, sentence):
            return [lang.word2index[word] for word in sentence.split(' ')]
```

```
def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)


def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

3.2 Training the Model

To train we run the input sentence through the encoder, and keep track of every output and the latest hidden state. Then the decoder is given the `<SOS>` token as its first input, and the last hidden state of the encoder as its first hidden state.

"Teacher forcing" is the concept of using the real target outputs as each next input, instead of using the decoder's guess as the next input. Using teacher forcing causes it to converge faster but `when the trained network is exploited, it may exhibit instability <http://minds.jacobs-university.de/sites/default/files/uploads/papers/ESNTutorialRev.pdf>`__.

You can observe outputs of teacher-forced networks that read with coherent grammar but wander far from the correct translation - intuitively it has learned to represent the output grammar and can "pick up" the meaning once the teacher tells it the first few words, but it has not properly learned how to create the sentence from the translation in the first place.

Because of the freedom PyTorch's autograd gives us, we can randomly choose to use teacher forcing or not with a simple if statement. Turn `teacher_forcing_ratio` up to use more of it.

```
In [ ]: teacher_forcing_ratio = 0.5


        def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_op
            encoder_hidden = encoder.initHidden()

            encoder_optimizer.zero_grad()
            decoder_optimizer.zero_grad()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

            loss = 0

            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(
                    input_tensor[ei], encoder_hidden)
                encoder_outputs[ei] = encoder_output[0, 0]

            decoder_input = torch.tensor([[SOS_token]], device=device)

            decoder_hidden = encoder_hidden

            use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

            if use_teacher_forcing:
                # Teacher forcing: Feed the target as the next input
                for di in range(target_length):
                    decoder_output, decoder_hidden, decoder_attention = decoder(
                        decoder_input, decoder_hidden, encoder_outputs)
                    loss += criterion(decoder_output, target_tensor[di])
                    decoder_input = target_tensor[di]  # Teacher forcing

            else:
                # Without teacher forcing: use its own predictions as the next input
                for di in range(target_length):
                    decoder_output, decoder_hidden, decoder_attention = decoder(
                        decoder_input, decoder_hidden, encoder_outputs)
                    topv, topi = decoder_output.topk(1)
                    decoder_input = topi.squeeze().detach()  # detach from history as input

                    loss += criterion(decoder_output, target_tensor[di])
                    if decoder_input.item() == EOS_token:
                        break

            loss.backward()

            encoder_optimizer.step()
            decoder_optimizer.step()

            return loss.item() / target_length
```

This is a helper function to print time elapsed and estimated time remaining given the current time and progress %.

```
In [ ]: import time
        import math


        def asMinutes(s):
            m = math.floor(s / 60)
            s -= m * 60
            return '%dm %ds' % (m, s)
```

```
def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```

The whole training process looks like this:

- Start a timer
- Initialize optimizers and criterion
- Create set of training pairs
- Start empty losses array for plotting

Then we call `train` many times and occasionally print the progress (% of examples, time so far, estimated time) and average loss.

```
In [ ]: def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_ra
            start = time.time()
            plot_losses = []
            print_loss_total = 0  # Reset every print_every
            plot_loss_total = 0  # Reset every plot_every

            encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
            decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
            training_pairs = [tensorsFromPair(random.choice(pairs))
                              for i in range(n_iters)]
            criterion = nn.NLLLoss()

            for iter in range(1, n_iters + 1):
                training_pair = training_pairs[iter - 1]
                input_tensor = training_pair[0]
                target_tensor = training_pair[1]

                loss = train(input_tensor, target_tensor, encoder,
                             decoder, encoder_optimizer, decoder_optimizer, criterion)
                print_loss_total += loss
                plot_loss_total += loss

                if iter % print_every == 0:
                    print_loss_avg = print_loss_total / print_every
                    print_loss_total = 0
                    print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                                 iter, iter / n_iters * 100, print_loss_avg))

                if iter % plot_every == 0:
                    plot_loss_avg = plot_loss_total / plot_every
                    plot_losses.append(plot_loss_avg)
                    plot_loss_total = 0
```

```
        showPlot(plot_losses)
```

3.3 Plotting results

Plotting is done with matplotlib, using the array of loss values `plot_losses` saved while training.

```
In [ ]: import matplotlib.pyplot as plt
        plt.switch_backend('agg')
        import matplotlib.ticker as ticker
        import numpy as np


        def showPlot(points):
            plt.figure()
            fig, ax = plt.subplots()
            # this locator puts ticks at regular intervals
            loc = ticker.MultipleLocator(base=0.2)
            ax.yaxis.set_major_locator(loc)
            plt.plot(points)
```

4 Evaluation

Evaluation is mostly the same as training, but there are no targets so we simply feed the decoder's predictions back to itself for each step. Every time it predicts a word we add it to the output string, and if it predicts the EOS token we stop there. We also store the decoder's attention outputs for display later.

```
In [ ]: def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
            with torch.no_grad():
                input_tensor = tensorFromSentence(input_lang, sentence)
                input_length = input_tensor.size()[0]
                encoder_hidden = encoder.initHidden()

                encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

                for ei in range(input_length):
                    encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                             encoder_hidden)
                    encoder_outputs[ei] += encoder_output[0, 0]

                decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

                decoder_hidden = encoder_hidden

                decoded_words = []
                decoder_attentions = torch.zeros(max_length, max_length)
```

```
        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

We can evaluate random sentences from the training set and print out the input, target, and output to make some subjective quality judgements:

```
In [ ]: def evaluateRandomly(encoder, decoder, n=10):
            for i in range(n):
                pair = random.choice(pairs)
                print('>', pair[0])
                print('=', pair[1])
                output_words, attentions = evaluate(encoder, decoder, pair[0])
                output_sentence = ' '.join(output_words)
                print('<', output_sentence)
                print('')
```

5 Training and Evaluating

With all these helper functions in place (it looks like extra work, but it makes it easier to run multiple experiments) we can actually initialize a network and start training.

Remember that the input sentences were heavily filtered. For this small dataset we can use relatively small networks of 256 hidden nodes and a single GRU layer. After about 40 minutes on a MacBook CPU we'll get some reasonable results.

.. Note:: If you run this notebook you can train, interrupt the kernel, evaluate, and continue training later. Comment out the lines where the encoder and decoder are initialized and run `trainIters` again.

```
In [ ]: hidden_size = 256
        encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
        attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to(dev

        trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

```
In [ ]: evaluateRandomly(encoder1, attn_decoder1)
```

5.1 Visualizing Attention

A useful property of the attention mechanism is its highly interpretable outputs. Because it is used to weight specific encoder outputs of the input sequence, we can imagine looking where the network is focused most at each time step.

You could simply run `plt.matshow(attentions)` to see attention output displayed as a matrix, with the columns being input steps and rows being output steps:

```
In [ ]: output_words, attentions = evaluate(
            encoder1, attn_decoder1, "je suis trop froid .")
        plt.matshow(attentions.numpy())
```

For a better viewing experience we will do the extra work of adding axes and labels:

```
In [ ]: def showAttention(input_sentence, output_words, attentions):
            # Set up figure with colorbar
            fig = plt.figure()
            ax = fig.add_subplot(111)
            cax = ax.matshow(attentions.numpy(), cmap='bone')
            fig.colorbar(cax)

            # Set up axes
            ax.set_xticklabels([''] + input_sentence.split(' ') +
                               ['<EOS>'], rotation=90)
            ax.set_yticklabels([''] + output_words)

            # Show label at every tick
            ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
            ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

            plt.show()


        def evaluateAndShowAttention(input_sentence):
            output_words, attentions = evaluate(
                encoder1, attn_decoder1, input_sentence)
            print('input =', input_sentence)
            print('output =', ' '.join(output_words))
            showAttention(input_sentence, output_words, attentions)


        evaluateAndShowAttention("elle a cinq ans de moins que moi .")

        evaluateAndShowAttention("elle est trop petit .")

        evaluateAndShowAttention("je ne crains pas de mourir .")

        evaluateAndShowAttention("c est un jeune directeur plein de talent .")
```

6 Exercises

- Try with a different dataset
- Another language pair
- Human Machine (e.g. IOT commands)
- Chat Response
- Question Answer
- Replace the embeddings with pre-trained word embeddings such as word2vec or GloVe
- Try with more layers, more hidden units, and more sentences. Compare the training time and results.
- If you use a translation file where pairs have two of the same phrase (`I am test \t I am test`), you can use this as an autoencoder. Try this:
- Train as an autoencoder
- Save only the Encoder network
- Train a new Decoder for translation from there