Computing Exact Graph Embeddings with PyTorch

In this tutorial we will learn how to solve the graph reconstruction problem using PyTorch. We will reconstruct our graph's full adjacency matrix, and we will frame this problem as a logistic regression.

Namely, we want to ask, given two node IDs, is there going to be an edge between them or not? This would be a classical logistic regression problem, **if only we had some features!**

The good news is that we can **learn our own features** to solve this problem. To a social scientist this notion may seem a bit counter-intuitive. How can we learn about variables ("features") we did not even have an idea existed?

One way to understand the feature learning approach is to think about the missing data problem. There are many techniques to do imputation of missing values in survey datasets for instance. In a sense, this is what we'll be doing, only that we will be imputing values over entire columns of data.

This may seem a bit like black magic. Normally, when we impute missing data, we look at existing values in our data matrix for guidance on what values we should assign to empty cells. In our case the only information we have is whether there exists a tie between two edges or not. As it turns out, this is enough.

Problem setup

For every node i we assume there exists a vector $\vec{v}_i$ of dimensionality d. This vector is the *feature vector* and will encode all the information we learn about the node's participation in the graph. We will randomly sample pairs of nodes (i, j) from the graph's adjacency matrix.

Let $Y_{i,j}$ an indicator variable denoting the existence of an edge between nodes i and j. We will then write:

$$Y_{i,j} = \text{sigmoid}(\vec{v}_i \cdot \vec{v}_j) + \epsilon_{i,j},$$

where sigmoid is the sigmoid function, $\frac{1}{1+e^{-x}}$.

We will seek to reduce the values of $\epsilon_{i,j}$ as much as possible, under certain constraints expressed by a **loss function**. In our case (logistic regression), this simply the log loss (http://wiki.fast.ai/index.php/Log_Loss) function, also known among neural network researchers as the **binary cross entropy loss**:

$$L = -\sum_i \sum_j y_{i,j} \log p_{i,j} + (1 - y_{i,j}) \log(1 - p_{i,j})$$

Here, $p_{i,j} = \text{sigmoid}(\vec{v}_i \cdot \vec{v}_j)$.

The quantity L presented above is an unweighted loss. Such a loss quantifies how well we're doing at predicting our edges. Most real-world graphs are sparse, however, meaning they have many fewer 1s than 0s in their adjacency matrices. In this case, we will want to compute a *weighted* loss:

$$L = -\sum_i \sum_j w_{i,j} \left[y_{i,j} \log p_{i,j} + (1 - y_{i,j})(\log 1 - p_{i,j}) \right]$$

The goal of introducing $w_{i,j}$ is to ensure that doing poorly at predicting 1s in our matrix is penalized just as heavily as doing poorly at predicting 0s. Thus if N_0 and N_1 quantify the number of 1s and 0s in our adjacency matrix, then we can set:

$$w_{i,j} = \begin{cases} 1 & Y_{i,j} = 1 \\ \frac{N_1}{N_0} & Y_{i,j} = 0 \end{cases}$$

Now that we have the apparatus behind our loss function, let's take a look at how we will be computing the values of this function in practice. For now we won't worry about how we will actually minimize this loss, nor will we pay much attention to computational efficiency.

In [1]:
```
import torch
import numpy as np
```

Then, we will load an example graph to play with. We will only look at a small graph, an email network of about 1000 persons from a EU institution, which we can obtain from SNAP (https://snap.stanford.edu/data/email-Eu-core.html). Make sure to download the `email-Eu-core` file and unzip it (via `gunzip` in UNIX-like systems). Then change the `DATA_PATH` directory appropriately.

We will use `np.loadtxt` to read the data, skipping the first 4 rows (which contain metadata in all SNAP datasets). Note that for the sake of clarity we will forego some smarter, vectorized data manipulation operations you could do in `numpy`.

In [2]:
```
DATA_PATH = './'
FILE_NAME = 'email-Eu-core.txt'
mat = np.loadtxt(DATA_PATH + FILE_NAME, dtype=int)
```

Now that we have loaded the edgelist, we can inspect it:

In [3]:
```
mat[:3,:]
```

Out[3]:
```
array([[0, 1],
       [2, 3],
       [2, 4]])
```

Note how the nodes are zero-indexed. We will need to keep track of the total number of nodes and edges, so we should take that factor into account when counting nodes.

In [4]:
```
num_nodes = int(np.max(mat)) + 1
num_edges = mat.shape[0]
print(num_nodes)
print(num_edges)
```

```
1005
25571
```

Now we have an edgelist, but this only represents the 1s in our matrix. We also need to find all the zeros. We will do so by traversing our adjacency matrix exhaustively and finding all the pairs (i, j) which are not 1s. Note that this is a very expensive operation for large graphs -- but do not worry! In practice, it is enough to *sample* from among the zeros, and what we're doing right now is a bit overkill. Since a graph with 1000-ish nodes has just over 1 million possible edges, we can afford to look at all of them.

In [5]:
```
ones = set([tuple(mat[i,:].tolist()) for i in range(num_edges)])
```

In [6]:
```
zeros = [(i, j)
         for i in range(num_nodes)
         for j in range(num_nodes)
         if i != j and not (i, j) in ones]
```

In [7]:
```
ones = list(ones)
```

In [8]:
```
print(ones[:10])
print(zeros[:10])
```

```
[(86, 820), (185, 101), (155, 589), (473, 418), (87, 459), (66, 66),
(513, 513), (218, 313), (329, 652), (271, 305)]
[(0, 2), (0, 3), (0, 4), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0,
12), (0, 13)]
```

In [9]:
```
print(len(ones))
print(len(zeros))
```

```
25571
984091
```

We need one more thing to generate our logistic regression dataset -- the weight $w_{i,\$}$, set at 1 for 1s and at N_1/N_0 for 0s:

In [10]:
```
zeros_weight = len(ones) * 1.0 / len(zeros)
```

Now we're ready to create our dataset, a big matrix with 4 columns: two columns for the left- and right-hand side IDs, one column for the weight, and one column for the labels.

In [11]:
```
data = [list(x) + [zeros_weight, 0] for x in zeros] + [list(x) + [1, 1
] for x in ones]
```

In [12]:
```
print(data[:10])
```

```
[[0, 2, 0.025984385590356988, 0], [0, 3, 0.025984385590356988, 0], [0,
4, 0.025984385590356988, 0], [0, 7, 0.025984385590356988, 0], [0, 8,
0.025984385590356988, 0], [0, 9, 0.025984385590356988, 0], [0, 10, 0.0
25984385590356988, 0], [0, 11, 0.025984385590356988, 0], [0, 12, 0.025
984385590356988, 0], [0, 13, 0.025984385590356988, 0]]
```

Introducing PyTorch datatypes

The essential data structure in PyTorch is called a "tensor." It's similar to an `ndarray` in `numpy`: the n-dimensional homologue of a matrix. The difference is that tensor operations are *really* fast in PyTorch, which is optimized to do these things very quickly on both the CPU and GPU.

Speaking of which, moving tensors between the computer's main RAM and the GPU RAM is extremely easy in PyTorch, a feature which is nothing short of magical. We won't use this feature just yet, however. When first developing a model it makes sense to keep things on the CPU, where the errors are generally a lot more informative.

It's really easy to create this tensor -- just call the `FloatTensor` constructor.

In [13]:
```
tensor = torch.FloatTensor(data)
```

In [14]:
```
print(tensor)
```

```
tensor([[    0.0000,     2.0000,     0.0260,     0.0000],
        [    0.0000,     3.0000,     0.0260,     0.0000],
        [    0.0000,     4.0000,     0.0260,     0.0000],
        ...,
        [  116.0000,   548.0000,     1.0000,     1.0000],
        [  815.0000,   880.0000,     1.0000,     1.0000],
        [   63.0000,   255.0000,     1.0000,     1.0000]])
```

Datasets and Dataloaders

PyTorch is built to deal with very large datasets, many of which will not really fit into memory. To elegantly isolate away the custom code required to load certain kinds of data, PyTorch introduces two classes -- a `Dataset` and a `DataLoader`.

The `Dataset` class keeps track of where the data actually lives: in a file, a folder on disk, somewhere on the Internet, etc. It also defines the notion of an *example*: a row in a survey matrix, an image on disk, a time series, an audio recording, or a text document.

In our simple case, our data lives in a tensor stored in memory, and a row in this tensor constitutes an example. This means we can use the `TensorDataset` class, which is effectively a wrapper around our `FloatTensor`.

In [15]:
```
from torch.utils.data import TensorDataset
```

In [16]:
```
dataset = TensorDataset(tensor)
```

Something to note: `dataset` is effectively a list of rows from `tensor`.

In [17]:
```
some_rows, = dataset[:10]
print(some_rows)
```

```
tensor([[  0.0000,   2.0000,   0.0260,   0.0000],
        [  0.0000,   3.0000,   0.0260,   0.0000],
        [  0.0000,   4.0000,   0.0260,   0.0000],
        [  0.0000,   7.0000,   0.0260,   0.0000],
        [  0.0000,   8.0000,   0.0260,   0.0000],
        [  0.0000,   9.0000,   0.0260,   0.0000],
        [  0.0000,  10.0000,   0.0260,   0.0000],
        [  0.0000,  11.0000,   0.0260,   0.0000],
        [  0.0000,  12.0000,   0.0260,   0.0000],
        [  0.0000,  13.0000,   0.0260,   0.0000]])
```

The rows above appear in the same order as they were originally generated. This is a problem when using **minibatch Stochastic Gradient Descent**, the standard optimization algorithm that powers neural network research.

Gradient descent as an optimization strategy works by taking small steps (slightly changing the values of the estimated parameters) in the direction which looks most promising, i.e. would most reduces the total loss. The gradient of the loss with respect to the model's parameters tells the model in which direction to head.

The *exact* gradient would require the calculation of the Jacobian matrix (https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant), the matrix of first-order partial derivatives with of the entire loss with respect to all parameters in our model. Given that we "only" have $2 * 1005 = 2010$ parameters and 1005^2 examples in our toy model, it is technically possible to compute this matrix. But this matrix becomes very expensive to compute for "real-world" sized models which may have many millions of parameters.

Instead, Stochastic Gradient Descent (https://en.wikipedia.org/wiki/Stochastic_gradient_descent) uses an approximation of the gradient, which is computed on a *mini-batch*, a subset of examples, which can comfortably fit into memory. And this mini-batch cannot have all 1s, or all 0s -- in that case it would be trivial to minimize the loss by making the model always predict all 1s or 0s. Instead, it is desirable to have a mini-batch that is a random sample from the dataset. Moreover, we will usually need to go through the same dataset multiple times ("epochs") as we iterate towards a solution, and in that case it is also desirable to re-randomize our data between epochs.

The `DataLoader` class answers these requirements, by giving us a quick way to produce batches up to a maximum size from our `Dataset`:

In [18]:
```
from torch.utils.data import DataLoader
```

In [19]:
```
loader = DataLoader(dataset, batch_size=100000, shuffle=True)
```

So what does a `DataLoader` *actually* do? Everytime you call its `__iter__` method, a `DataLoader` gives back an iterator -- but this iterator is over *batches*, not over examples. The examples in the batches cover all the data in `Dataset`, but the order in which rows are distributed into batches is randomized (via `shuffle=True`).

We can see the first batch yielded by the loader by calling `__next__` on the iterator it just created.

In [20]:
```
batch = loader.__iter__().__next__()[0]
```

In [21]:
```
print(batch)
```

```
tensor([[  436.0000,   594.0000,     0.0260,     0.0000],
        [  550.0000,   250.0000,     0.0260,     0.0000],
        [  621.0000,   102.0000,     0.0260,     0.0000],
        ...,
        [  105.0000,   110.0000,     0.0260,     0.0000],
        [  798.0000,   193.0000,     0.0260,     0.0000],
        [  252.0000,   847.0000,     0.0260,     0.0000]])
```

The Embedding module

You can think of the features we will learn as numbers in a look-up table. When nodes i and j are under consideration, we look for the i-th and j-th entry in this table, get the vectors from there, take their cross product, run it through a sigmoid and we have a prediction for whether a tie is likely to exist or not between the two nodes. We also know whether a tie actually exists. If we did poorly at this prediction, the values in our lookup table may be in need of a lot of revision. If we did pretty well, maybe we'll move the values around just a little bit.

The `Embedding` module in PyTorch expresses exactly this idea of a lookup table. We set it up by calling `torch.nn.Embedding` and indicating the number of entities in our graph, as well as the size of our feature vectors in the constructor:

In [22]:
```
emb = torch.nn.Embedding(num_nodes, 2)
```

Calls to the `Embedding` module require `LongTensor` node IDs, which must also be numbered from 0 to $n - 1$, where n is the number of vertices. In our case we obtain `LongTensors` by first converting the first two columns in our batch to their `numpy` equivalent:

In [23]:
```
node_ids = torch.LongTensor(batch[:,:2].numpy())
```

In [24]:
```
print(node_ids)
```

```
tensor([[  436,   594],
        [  550,   250],
        [  621,   102],
        ...,
        [  105,   110],
        [  798,   193],
        [  252,   847]])
```

Now that we have our `LongTensor` node IDs, we can issue lookups in the `Embedding` table for their vectors. We do this for the left- and right-hand side vectors separately. We retrieve vectors for the entire batch at once.

In [25]:
```
lhs = emb(node_ids[:, 0])
rhs = emb(node_ids[:, 1])
```

In [26]:
```
print(lhs[:10,:])
```

```
tensor([[-1.0952, -0.3440],
        [ 0.3098,  0.2339],
        [-0.6589,  0.1565],
        [-0.6285,  2.7189],
        [ 0.0848,  0.4112],
        [-0.2191,  0.1433],
        [ 1.6574,  0.8133],
        [-2.0182, -1.0061],
        [ 0.1538, -0.1730],
        [ 1.6420, -1.2262]])
```

It's trivial to get the cross-product of the two vectors for every example in the batch.

In [27]:
```
cross_prod = (lhs * rhs).sum(dim=1)
print(cross_prod)
```

```
tensor([-2.7618e+00,  6.3492e-01, -4.2345e-01,  ..., -1.0914e+00,
         8.2291e-01, -1.6657e-01])
```

The Loss Function

We will quantify how well (or poorly) we are doing at predicting the existence of edges in our graph by using the Binary Cross-Entropy loss. To do so, we will need a prediction, a label, and -- in our case -- a weight.

First we turn our cross product into a prediction, by running it through a sigmoid transform.

In [28]:
```
pred = cross_prod.sigmoid()
print(pred)
```

```
tensor([ 0.0594,  0.6536,  0.3957,  ...,  0.2513,  0.6949,  0.4585])
```

Finally, we also get labels as part of our batch:

In [29]:
```
labels = batch[:, 3]
```

In [30]:
```
print(labels[:100])
```

```
tensor([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.])
```

Now that we have computed predictions and retrieved our labels, we can then use the binary cross-entropy loss discussed in the introduction:

In [31]:
```
casewise_loss = -1 * ((labels * pred.log()) + (1-labels) * (1-pred).lo
g())
```

In [32]:
```
print(casewise_loss)
```

```
tensor([ 6.1263e-02,  1.0602e+00,  5.0367e-01,  ...,  2.8948e-01,
         1.1870e+00,  6.1333e-01])
```

But the losses must be weighted according to whether the example had a 1 (rare) or a 0 (frequent) label. The batch loss is the weighted sum of casewise losses.

In [33]:
```
weights = batch[:, 2]
```

In [34]:
```
loss = (casewise_loss * weights).sum()
print(loss)
```

```
tensor(4357.6221)
```

The Variable Class

To really understand the power of PyTorch you need to understand the function of the `Variable` class. A `Variable` is a tensor that "remembers" what operations it participated in.

What do I mean by this, and why is it useful? Let's go back to our loss function:

$$L = -\sum_i \sum_j w_{i,j} \left[y_{i,j} \log p_{i,j} + (1 - y_{i,j})(\log 1 - p_{i,j}) \right]$$

The mini-batch SGD algorithm requires us to make a small step in the direction indicated by the gradient. The step has a "width" α, also known as the **learning rate**. The gradient tells us how much the loss *increases* if we increase a parameter by α. Thus, to minimize the loss, we will move in the opposite direction of the gradient. We will update all vectors $\overrightarrow{v_i}$ using this equation:

$$\vec{v}_i := \vec{v}_i - \alpha \frac{\partial L}{\partial \overrightarrow{v_i}}$$

But how do we get this gradient, when all we have is a loss? This is where it's useful to rewrite our loss as follows:

$$L = -\sum_i \sum_j w_{i,j} \times \mathrm{BCE}(y_{i,j}, \mathrm{sig}(\overrightarrow{v_i} \cdot \overrightarrow{v_j}))$$

To take the derivative $\partial L / \partial \overrightarrow{v_i}$, we need to go back to elementary calculus and remember that the derivative of a sum is the sum of the terms' derivatives, and that $w_{i,j}$ is a constant that does not depend on $\overrightarrow{v_i}$:

$$\frac{\partial L}{\partial \overrightarrow{v_i}} = -\sum_i \sum_j w_{i,j} \partial \frac{\mathrm{BCE}(y_{i,j}, \mathrm{sig}(\overrightarrow{v_i} \cdot \overrightarrow{v_j}))}{\partial \overrightarrow{v_i}}$$

Okay, this was the easy part. Now we need to get the gradient of the binary cross-entropy function with respect to our vector. To do so, we can enlist the chain rule (https://en.wikipedia.org/wiki/Chain_rule), which stipulates that:

$$(f(g(x)))' = f'(g(x))g'(x)$$

It would be a long and rather tedious exercise to go into exactly how the chain rule is applied in our particular case, but suffice to say that for certain derivatives $f'(x)$ you need to know what the value of x is in order to compute the derivative. But note how we have already computed our loss: for every function along the chain leading to our final loss value, we had to compute this x. The `Variable` class "remembers" this value where applicable, and uses it to give back a gradient.

Of course, `Variables` can only be subject to differentiable operations, the logic for which has to be implemented and optimized. But there are a lot of differentiable operations available in PyTorch -- and because of the chain rule, `Variables` can be composed into the endlessly complex chains of operations that power modern AI models.

So how do we use Variables? Let's rewrite our loss calculation with them:

In [35]:
```
from torch.autograd import Variable

def compute_loss(batch):
    lhs_node_ids = torch.LongTensor(batch[:,0].numpy())
    rhs_node_ids = torch.LongTensor(batch[:,1].numpy())

    lhs_var = Variable(lhs_node_ids)
    rhs_var = Variable(rhs_node_ids)
    labels = Variable(batch[:, 3])

    lhs_vec = emb(lhs_var)
    rhs_vec = emb(rhs_var)
    cross_prod = (lhs_vec * rhs_vec).sum(dim=1)
    pred = cross_prod.sigmoid()
    casewise_loss = -1 * ((labels * pred.log()) + (1-labels) * (1-pred
).log())
    weights = batch[:, 2]
    loss = (casewise_loss * weights).sum()
    return loss
```

That was... not so hard. In fact the use of `Variable` is so straightforward that PyTorch 0.4 has deprecated their use, and just unified them with `Tensor` types. But in my view it's still educational to look at the enhancement this data structure brings as separate from what `Tensors` do.

Optimizers

Remember how for SGD we'd do a parameter update following this formula:

$$\vec{v}_i := \vec{v}_i - \alpha \frac{\partial L}{\partial \overrightarrow{v_i}}$$

This is pretty straightforward in theory, but it's actually quite nasty in practice, since we'd have to visit every single parameter in our model. The model presented here is extremely simple, but nonetheless it would potentially be quite tedious to update every single parameter, depending on how our gradients are calculated. Moreover, there are many "better" versions of SGD (http://ruder.io/optimizing-gradient-descent/), such as AdaGrad or Adam that get models to converge faster. These algorithms often require complex parameter updates that are best abstracted away -- which is exactly what PyTorch does. So how does an optimizer operate? We can find out by writing a few lines of code:

In [36]:
```
optim = torch.optim.SGD(emb.parameters(), lr=10e-3)

def learn_from_batch(batch):
    optim.zero_grad()
    loss = compute_loss(batch)
    loss.backward()
    optim.step()
    return loss
```

What's happening in this function?

First we initialize our optimizer, and give it jurisdiction over our Embedding table's parameters. We also set a learning rate -- 10e-3 seems like a good value to start with, though this parameter will need to be tuned. Note that different optimizers could have jurisdiction over different model parameters.

Then, in the function, the optimizer object begins by zeroing its gradients (resetting its state). Then we compute the loss, and then the gradient computation happens through `loss.backward()`. This is where the magic occurs: we do not need to write a single line of code indicating how gradients are to be computed -- the PyTorch internals take care of all of this. Then, finally, a call to `step` ensures the parameter update. (A clear explanation of how this works in a 2-layer neural network can be found here (https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_optim.html)).

To see how the parameters change, we can inspect them before and after the training:

In [37]:
```
print(emb.weight)
learn_from_batch(batch)
print(emb.weight)
```

```
Parameter containing:
tensor([[ 5.8352e-01,  2.3258e+00],
        [ 5.3681e-01,  1.1407e+00],
        [-1.0678e+00, -7.1846e-01],
        ...,
        [ 2.9300e-01,  4.2089e-01],
        [-1.5026e-01, -3.9012e-01],
        [-1.1051e+00, -5.9288e-01]])
Parameter containing:
tensor([[ 0.5498,  2.3030],
        [ 0.5199,  1.1350],
        [-1.0439, -0.6840],
        ...,
        [ 0.2899,  0.4196],
        [-0.1552, -0.3634],
        [-1.0928, -0.5847]])
```

If you look closely, you will see that parameters only changed a little, or not a at all, but the values did shift. How fast the parameters need to change depends on the specifics of our model -- but seing shifts in their values is a sign the `optim` object is doing its job.

Training for an epoch

Now that we have figured out how to deal with a single batch, it's time to run our algorithm for an entire epoch, going through every single batch in our data and iteratively updating our parameters. Our `DataLoader` makes this quite easy. In our implementation we will keep track of the average loss per case, which should go down as our model learns. Given that every batch only contains a fraction of the data, we do expect the per-batch loss to jump around a little. We return a per-epoch loss, which will be very informative as we train our model for multiple epochs.

In [38]:
```
def train_one_epoch(verbose=True):
    total_loss = 0
    for batch, in loader:
        loss = learn_from_batch(batch)
        if verbose:
            per_case_loss = loss / batch.size()[0]
            print(per_case_loss)
        total_loss += loss
    return total_loss
```

In [39]:
```
train_one_epoch()
```

```
tensor(1.00000e-02 *
       4.2976)
tensor(1.00000e-02 *
       4.2168)
tensor(1.00000e-02 *
       4.2238)
tensor(1.00000e-02 *
       4.1562)
tensor(1.00000e-02 *
       4.1015)
tensor(1.00000e-02 *
       4.1067)
tensor(1.00000e-02 *
       4.0838)
tensor(1.00000e-02 *
       4.0098)
tensor(1.00000e-02 *
       4.0070)
tensor(1.00000e-02 *
       3.9915)
tensor(1.00000e-02 *
       3.9275)
```

Out[39]:
```
tensor(41574.2266)
```

Moving to the GPU

If easy tensor multiplication and automated differentiation weren't great enough, PyTorch also offers a painless way to move data and computation between the CPU and the GPU. PyTorch supports NVIDIA GPUs that run CUDA, a C++-based programming language for GPU-based computation. An incidental outcome of the requirements of computer graphics, GPUs turn out to be really fast at tensor multiplication. GPUs (and CUDA) are also, typically a pain to work with, which is why the availability of a means to work with them while abstracting away their details is a great help. To see what we need to move to CUDA, let's get together our entire learning routine.

Note: if you don't have a GPU available, just set `GPU_ENABLED` to False!

In [40]:
```
GPU_ENABLED = True

emb = torch.nn.Embedding(num_nodes, 2)
if GPU_ENABLED:
    emb = emb.cuda()
optim = torch.optim.SGD(emb.parameters(), lr=10e-3)

def compute_loss(batch, emb=emb, optim=optim):
    lhs_node_ids = torch.LongTensor(batch[:,0].numpy())
    rhs_node_ids = torch.LongTensor(batch[:,1].numpy())

    lhs_var = Variable(lhs_node_ids)
    rhs_var = Variable(rhs_node_ids)
    labels = Variable(batch[:, 3])
    weights = batch[:, 2]

    if GPU_ENABLED:
        lhs_var = lhs_var.cuda()
        rhs_var = rhs_var.cuda()
        labels = labels.cuda()
        weights = weights.cuda()

    lhs_vec = emb(lhs_var)
    rhs_vec = emb(rhs_var)
    cross_prod = (lhs_vec * rhs_vec).sum(dim=1)
    pred = cross_prod.sigmoid()
    casewise_loss = -1 * ((labels * pred.log()) + (1-labels) * (1-pred
).log())
    loss = (casewise_loss * weights).sum()
    return loss

def learn_from_batch(batch, emb=emb, optim=optim):
    optim.zero_grad()
    loss = compute_loss(batch, emb, optim)
    loss.backward()
    optim.step()
    return loss

def train_one_epoch(verbose=True, emb=emb, optim=optim):
    total_loss = 0
    for batch, in loader:
        loss = learn_from_batch(batch, emb, optim)
        if verbose:
            per_case_loss = loss / batch.size()[0]
            print(per_case_loss)
        total_loss += loss
    return total_loss
```

There were 5 things we needed to move to the GPU. In all cases, this was achieved by calling `.cuda()` on a data structure. Specifically, we had to move:

- the `Embedding` module. This is our "model", which contains the parameters we will learn.
- the `lhs_var` and `rhs_var` indices. These contain the indices we actually look up in each batch. Note that a lookup is a differentiable operation!
- the `labels` and `weights`, which are both crucial parts of our loss.

That's it! We did not really need to explicitly do anything to intermediary data, or to the `optim` object -- these objects can infer what memory they need to use from context.

In [41]:
```
print(train_one_epoch())
```

```
tensor(1.00000e-02 *
       4.3301, device='cuda:0')
tensor(1.00000e-02 *
       4.2964, device='cuda:0')
tensor(1.00000e-02 *
       4.2556, device='cuda:0')
tensor(1.00000e-02 *
       4.1789, device='cuda:0')
tensor(1.00000e-02 *
       4.1121, device='cuda:0')
tensor(1.00000e-02 *
       4.0933, device='cuda:0')
tensor(1.00000e-02 *
       4.0227, device='cuda:0')
tensor(1.00000e-02 *
       4.1015, device='cuda:0')
tensor(1.00000e-02 *
       3.9905, device='cuda:0')
tensor(1.00000e-02 *
       3.9384, device='cuda:0')
tensor(1.00000e-02 *
       4.1248, device='cuda:0')
tensor(41717.8984, device='cuda:0')
```

Bringing it all together

Running on CUDA means we can typically spend about 90% less time waiting for our computation to finish -- a pretty significant difference. Now it's time to run our algorithm for multiple epochs, only to encounter a small surprise:

In [42]:
```
emb = torch.nn.Embedding(num_nodes, 2).cuda()
optim = torch.optim.SGD(emb.parameters(), lr=10e-3)

for i in range(100):
    loss = train_one_epoch(verbose=False, emb=emb, optim=optim)
    print("Epoch: %d, loss=%.4f" % (i, loss.cpu().detach().numpy().tol
ist()))
```

```
Epoch: 0, loss=42364.4102
Epoch: 1, loss=38916.1836
Epoch: 2, loss=37250.2344
Epoch: 3, loss=36074.9805
Epoch: 4, loss=34930.3203
Epoch: 5, loss=33692.7500
Epoch: 6, loss=32461.7031
Epoch: 7, loss=31370.2910
Epoch: 8, loss=30470.7090
Epoch: 9, loss=29747.5156
Epoch: 10, loss=29162.0645
Epoch: 11, loss=28685.4688
Epoch: 12, loss=28291.5469
Epoch: 13, loss=27963.9688
Epoch: 14, loss=27689.9902
Epoch: 15, loss=27459.1152
Epoch: 16, loss=27264.0410
Epoch: 17, loss=27097.5371
Epoch: 18, loss=26955.2676
Epoch: 19, loss=26832.0664
Epoch: 20, loss=26725.1895
Epoch: 21, loss=26630.8027
Epoch: 22, loss=26547.7988
Epoch: 23, loss=26474.3750
Epoch: 24, loss=26409.8242
Epoch: 25, loss=26350.9531
Epoch: 26, loss=26298.5156
Epoch: 27, loss=26251.5977
Epoch: 28, loss=26209.1270
Epoch: 29, loss=26170.9961
Epoch: 30, loss=26135.6484
Epoch: 31, loss=26104.6270
Epoch: 32, loss=26075.8516
Epoch: 33, loss=26049.2207
Epoch: 34, loss=26024.9805
Epoch: 35, loss=26002.6035
Epoch: 36, loss=25981.5176
Epoch: 37, loss=25962.1758
Epoch: 38, loss=25943.9414
Epoch: 39, loss=25926.8184
Epoch: 40, loss=25910.5566
Epoch: 41, loss=25895.0430
Epoch: 42, loss=25881.6387
Epoch: 43, loss=25867.4922
Epoch: 44, loss=25854.3047
Epoch: 45, loss=25842.2031
Epoch: 46, loss=25829.9863
Epoch: 47, loss=25818.9863
Epoch: 48, loss=25808.1016
Epoch: 49, loss=25798.6445
Epoch: 50, loss=nan
Epoch: 51, loss=nan
Epoch: 52, loss=nan
Epoch: 53, loss=nan
Epoch: 54, loss=nan
Epoch: 55, loss=nan
```

```
---------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
 last)
<ipython-input-42-7d5b4382051d> in <module>()
      3
      4 for i in range(100):
----> 5     loss = train_one_epoch(verbose=False, emb=emb, optim=optim
)
      6     print("Epoch: %d, loss=%.4f" % (i, loss.cpu().detach().num
py().tolist()))

<ipython-input-40-79f5c5b045e2> in train_one_epoch(verbose, emb, opti
m)
     38 def train_one_epoch(verbose=True, emb=emb, optim=optim):
     39     total_loss = 0
---> 40     for batch, in loader:
     41         loss = learn_from_batch(batch, emb, optim)
     42         if verbose:

~/anaconda3/lib/python3.6/site-packages/torch/utils/data/dataloader.py
 in __next__(self)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise Stop
Iteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in
 indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)

KeyboardInterrupt:
```

Why the loss went to nan?

It's worth going back to the definition of the cross-entropy function. There's a `log(p)` and a `log(1-p)` in there. This means that if we do *really* well at predicting that two nodes will or won't have an edge, we could have some huge effects on the loss, at least as far as the gradient is concerned. This is a problem known as the "exploding gradient problem" in deep learning, but it can apply to feature learning as well.

The solution is to prevent the model from being tempted to do "really" well at the prediction task. We can do so by changing the loss function to "clamp" predictions at a minimum and a maximum value, $\epsilon = 10^{-6}$. The model will only optimize the predictions in the interval $[\epsilon, 1-\epsilon]$, and ignore any values outside this interval.

Note that regularization would be another effective remedy in this case, but it's not strictly necessary to solve this problem.

In [43]:
```
def compute_loss(batch, emb=emb, optim=optim):
    lhs_node_ids = torch.LongTensor(batch[:,0].numpy())
    rhs_node_ids = torch.LongTensor(batch[:,1].numpy())

    lhs_var = Variable(lhs_node_ids).cuda()
    rhs_var = Variable(rhs_node_ids).cuda()
    labels = Variable(batch[:, 3]).cuda()

    lhs_vec = emb(lhs_var)
    rhs_vec = emb(rhs_var)
    cross_prod = (lhs_vec * rhs_vec).sum(dim=1)

    EPS = 1e-6
    pred = cross_prod.sigmoid().clamp(min=EPS, max=1-EPS)
    casewise_loss = -1 * ((labels * pred.log()) + (1-labels) * (1-pred
).log())
    weights = batch[:, 2].cuda()

    loss = (casewise_loss * weights).sum()
    return loss
```

Now we're ready to run our training one more time, this time using clamped predictions. We will also keep track of the predicted weights at each iteration.

In [44]:
```
emb_weights = []
emb = torch.nn.Embedding(num_nodes, 2).cuda()
optim = torch.optim.SGD(emb.parameters(), lr=10e-3)

for i in range(100):
    loss = train_one_epoch(verbose=False, emb=emb, optim=optim)
    emb_weights += [emb.weight.detach().cpu().numpy().tolist()]
    print("Epoch: %d, loss=%.4f" % (i, loss.cpu().detach().numpy().tol
ist()))
```

```
Epoch: 0, loss=42646.9492
Epoch: 1, loss=39179.1797
Epoch: 2, loss=37629.4180
Epoch: 3, loss=36702.6094
Epoch: 4, loss=35954.9375
Epoch: 5, loss=35158.6328
Epoch: 6, loss=34184.7578
Epoch: 7, loss=33054.9023
Epoch: 8, loss=31919.5879
Epoch: 9, loss=30920.0293
Epoch: 10, loss=30102.5078
Epoch: 11, loss=29451.1094
Epoch: 12, loss=28930.7188
Epoch: 13, loss=28508.1113
Epoch: 14, loss=28158.3984
Epoch: 15, loss=27863.4863
Epoch: 16, loss=27611.9727
Epoch: 17, loss=27392.4590
Epoch: 18, loss=27197.0684
Epoch: 19, loss=27022.6680
Epoch: 20, loss=26866.4473
Epoch: 21, loss=26725.8086
Epoch: 22, loss=26598.5684
Epoch: 23, loss=26484.9180
Epoch: 24, loss=26383.4668
Epoch: 25, loss=26294.8262
Epoch: 26, loss=26217.2012
Epoch: 27, loss=26149.8750
Epoch: 28, loss=26091.4062
Epoch: 29, loss=26040.8691
Epoch: 30, loss=25997.8457
Epoch: 31, loss=25960.6758
Epoch: 32, loss=25929.1680
Epoch: 33, loss=25901.5469
Epoch: 34, loss=25877.8125
Epoch: 35, loss=25857.9082
Epoch: 36, loss=25840.0664
Epoch: 37, loss=25823.9941
Epoch: 38, loss=25811.3535
Epoch: 39, loss=25799.1562
Epoch: 40, loss=25787.5332
Epoch: 41, loss=25778.9043
Epoch: 42, loss=25769.4648
Epoch: 43, loss=25761.9824
Epoch: 44, loss=25754.9102
Epoch: 45, loss=25748.3359
Epoch: 46, loss=25742.5195
Epoch: 47, loss=25737.0391
Epoch: 48, loss=25731.4883
Epoch: 49, loss=25726.9355
Epoch: 50, loss=25722.1680
Epoch: 51, loss=25718.7148
Epoch: 52, loss=25713.8730
Epoch: 53, loss=25709.6250
Epoch: 54, loss=25706.2617
Epoch: 55, loss=25702.6504
Epoch: 56, loss=25699.3770
```

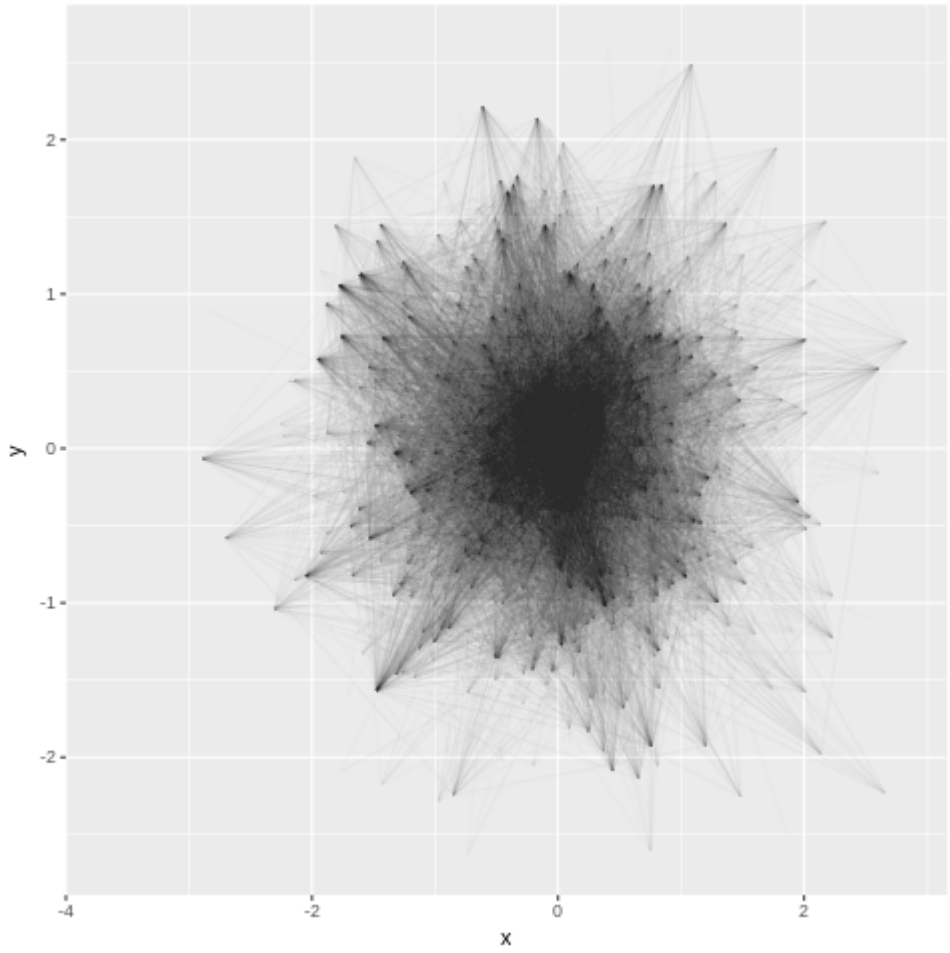
```
Epoch: 57, loss=25694.9316
Epoch: 58, loss=25691.7285
Epoch: 59, loss=25688.5586
Epoch: 60, loss=25684.7148
Epoch: 61, loss=25681.5469
Epoch: 62, loss=25678.3418
Epoch: 63, loss=25674.4629
Epoch: 64, loss=25671.1602
Epoch: 65, loss=25668.4590
Epoch: 66, loss=25665.5430
Epoch: 67, loss=25662.4746
Epoch: 68, loss=25659.4082
Epoch: 69, loss=25656.4160
Epoch: 70, loss=25652.8379
Epoch: 71, loss=25651.2773
Epoch: 72, loss=25648.4160
Epoch: 73, loss=25644.9863
Epoch: 74, loss=25643.7520
Epoch: 75, loss=25640.7383
Epoch: 76, loss=25638.7227
Epoch: 77, loss=25635.8418
Epoch: 78, loss=25634.8398
Epoch: 79, loss=25632.9648
Epoch: 80, loss=25630.5840
Epoch: 81, loss=25629.6836
Epoch: 82, loss=25627.6758
Epoch: 83, loss=25626.8164
Epoch: 84, loss=25625.5000
Epoch: 85, loss=25624.3125
Epoch: 86, loss=25623.3828
Epoch: 87, loss=25622.1582
Epoch: 88, loss=25621.0801
Epoch: 89, loss=25620.7734
Epoch: 90, loss=25619.7266
Epoch: 91, loss=25618.8535
Epoch: 92, loss=25618.6016
Epoch: 93, loss=25617.6484
Epoch: 94, loss=25617.7773
Epoch: 95, loss=25616.9453
Epoch: 96, loss=25616.8105
Epoch: 97, loss=25615.6992
Epoch: 98, loss=25615.1211
Epoch: 99, loss=25615.1992
```

Visualizing the predictions

The choice of embeddings of size-2 was a bit sneaky. In practice we would not really consider such low-dimension embeddings to be viable, but in this case they are perfect for visualization. Visualization is done best in R using the `ggnetwork` and `gganimate` packages, and we will pursue this task in a separate R notebook (visualize_emb.ipynb), but not after we write our data to a .tsv

In [45]:
```
with open('embedded_graph.tsv', 'w') as f:
    for i in range(100):
        wt = emb_weights[i]
        for j in range(len(wt)):
            f.write("%d\t%d\t%.6f\t%.6f\n" % (i, j, wt[j][0], wt[j][1
]))
```

Here's what the result looks like.



Further extensions

We won't be discussing these in this notebook, but there are quite a few things we still need to do before we can solve our feature learning problem for large graphs.

- we need more dimensions in our embedding space! 2 is not really enough.
- sampling negatives: notice how we have considered all the 0s when setting up our dataset. Ordinarily we would only consider about as many 0s as 1s. Note that sampling at random will not be a good idea, since in that case we'd just learn which nodes have high degree. There's an entire science to figuring out how to pick the right sample of negatives -- things could get really complicated if we were embedding weighted graphs, signed graphs or hypergraphs!
- regularization: we would likely also want to apply some sort of regularization to our parameters if we will increase the dimensionality of our space, to prevent overfitting.
- "shingling": for very large graphs, sampling negatives is not enough to make the problem tractable. We are bound by however many 32-bit floats can fit in our GPU's memory. There are however techniques for breaking this problem down into smaller sub-problems, which can all fit into memory.