TempConv

December 10, 2015

1 Temporal Convolution

We introduce a new neural network architecture for sentence classification using temporal convolution. This is a more powerful model than the simple bag of words model introduced before.

We set up our sample data as before, but also include padding at the beginning of the sentence.

In [1]:
```
V = {["*padding*"]=1, ["I"]= 2, ["am"]= 3, ["a"]= 4, ["he"]=5,
      ["it"]=6, ["dog"]=7, ["is"]=8, ["she"]=9}
nV = 10

function make_data(sent, n, start_pad)
    out = {}
    for i = 1, start_pad do
        v = V["*padding*"]
        table.insert(out, v)
    end
    for i = 1, n - start_pad do
        if i <= #sent then
            v = V[sent[i]]
        else
            v = V["*padding*"]
        end
        table.insert(out, v)
    end
    return out
end

indata = {}
outdata = {}
table.insert(indata, make_data({"I", "am", "a", "dog"}, 10, 3))
table.insert(outdata, 1)
table.insert(indata, make_data({"he", "is", "a", "dog"}, 10, 3))
table.insert(outdata, 2)
table.insert(indata, make_data({"she", "is", "a", "dog"}, 10, 3))
table.insert(outdata, 2)
table.insert(indata, make_data({"it", "is", "a", "dog"}, 10, 3))
table.insert(outdata, 2)


X = torch.DoubleTensor(indata)
y = torch.DoubleTensor(outdata)
nY = 2 -- Two classes.
```

Now we have J data points, and

- $\mathbf{X} \in \mathcal{V}^{J \times n}$ is our input data
- $\mathbf{y} \in \mathcal{Y}^{J}$ is out output data

2 The Model

The key step in our architecture is a convolution. Our input is a sequence of vectors $V_i \in \mathbb{R}^d$ for $i = 1, \ldots, n$, i.e. the d-dimensional word embeddings of our sentence. A convolution then involves:

- a *filter* $\mathbf{w} \in \mathbb{R}^{hd}$, where h is the filter size,
- $[V_j, V_{j+1}, \ldots, V_{j+h-1}] \in \mathbb{R}^{hd}$, the concatenation of vectors j through $j + h - 1$

We then take a dot product of $\mathbf{w}$ with our concatenated word vectors j through $j + h - 1$ to get a single value. Additionally, we include a bias term $b \in \mathbb{R}$ and a nonlinear activation function f such as $\tanh(x)$ or the rectilinear unit $f(x) = \max(0, x)$, so that our value is

$$c_j = f(w \cdot [V_j, V_{j+1}, \ldots, V_{j+h-1}] + b)$$

If we do this for every j, we get a feature map $\mathbf{c} \in \mathcal{R}^{n-h+1}$ with entries c_j.

With multiple filters $\mathbf{w_1}, \ldots, \mathbf{w_{d'}}$, we can form a matrix of feature maps $[\mathbf{c_1}; \mathbf{c_2}; \ldots; \mathbf{c_{n-h+1}}] \in \mathbb{R}^{(n-h+1) \times d'}$. In our approach, we do a max-over-time pooling for each feature map to get one feature per feature map:

$$\widehat{\mathbf{c_i}} = \max_j (\mathbf{c_i})_j$$

Then we have a vector of features for our sentence (one per filter): $[\widehat{\mathbf{c_1}}, \widehat{\mathbf{c_2}}, \ldots, \widehat{\mathbf{c_{d'}}}] \in \mathbb{R}^{d'}$

Now let's build this in torch.

2.1 Building the model

```
In [2]: nn = require "nn"
        d = 10
```

We start with the word embeddings layer, as usual.

```
In [3]: model = nn.Sequential()
        matrixV = nn.LookupTable(nV, d)
        model:add(matrixV)
```

We then add the convolution. Torch has convenient built-in methods for the convolution described above, such as `nn.TemporalConvolution`. This takes the dimensionality of the previous layer (d), the number of filters we use (nd, or d' in the above), and the filter size (h). We can also do a max pooling with `nn.Max`.

```
In [4]: nd = 10
        h = 3
        conv = nn.Sequential()
        conv:add(nn.TemporalConvolution(d, nd, h))
        conv:add(nn.ReLU())
        conv:add(nn.Max(2))

        model:add(conv)
```

The output of the above gives us our feature map $[\widehat{\mathbf{c_1}}, \widehat{\mathbf{c_2}}, \ldots, \widehat{\mathbf{c_{d'}}}]$. Finally we add a logistic regression layer for predicting the sentiment from this vector of features.

In [5]:
```
logistic = nn.Sequential()

logistic:add(nn.Linear(nd, nY))
logistic:add(nn.LogSoftMax())

model:add(logistic)
```

Let's test our model on the input:

In [6]:
```
model:forward(X)
```

Out[6]:
```
-0.6319 -0.7584
-0.5740 -0.8285
-0.6088 -0.7853
-0.7404 -0.6481
[torch.DoubleTensor of size 4x2]
```

As expected, we get (log) prediction probabilities for 2 classes for each input.
Include a negative-log-likelihood criterion:

In [7]:
```
criterion = nn.ClassNLLCriterion()
```

We can also implement these modules on GPUs. Specifically, we include the `cudnn` package, which has some GPU optimized versions of some of the above modules. One thing that requires modification is the convolution step - `cudnn` has no built in `TemporalConvolution` module, so we have to adapt the `SpatialConvolution` by reshaping our feature map matrix.
Here's the full implementation on `cudnn` (using batch mode):

In [ ]:
```
require 'cutorch'
require 'cudnn'

cudnn_model = nn.Sequential()
matrixV = nn.LookupTable(nV, d)
model:add(matrixV)

nd = 10
h = 3
S = 10
conv = nn.Sequential()
conv:add(nn.Reshape(1, S, d, false))
conv:add(cudnn.SpatialConvolution(1, nd, d, h))
conv:add(nn.Reshape(nd, S-h+1, false))
conv:add(cudnn.ReLU())
conv:add(nn.Max(3))

cudnn_model:add(conv)

logistic = nn.Sequential()

logistic:add(nn.Linear(nd, nY))
logistic:add(cudnn.LogSoftMax())

cudnn_model:add(logistic)
```

```
criterion = nn.ClassNLLCriterion()

-- Move to GPU
cudnn_model:cuda()
criterion:cuda()
```

2.2 Training

We perform training with `adadelta`. In each epoch, we create a closure that returns the gradient updates.

In [8]:
```
require 'optim'
model:reset()
model:training()

params, grads = model:getParameters()

config = { rho = 0.95, eps = 1e-6 }
state = {}
for epoch = 1, 20 do
    func = function(x)
        if x ~= params then
            params:copy(x)
        end
        grads:zero()

        out = model:forward(X)
        err = criterion:forward(out, y)

        dout = criterion:backward(out, y)
        model:backward(X, dout)

        return err, grads
    end

    optim.adadelta(func, params, config, state)
    print("Epoch:", epoch, err)
end
```

Out[8]:
```
Epoch:        1        1.0637046591443
Epoch:        2        0.92574699271742
Epoch:        3        0.80757029787584
```

Out[8]:
```
Epoch:        4        0.70017095481893
Epoch:        5        0.62009300393375
Epoch:        6        0.53988474604789
Epoch:        7        0.48039989542535
```

Out[8]:
```
Epoch:        8        0.42860197930536
Epoch:        9        0.38423555419475
Epoch:        10        0.34843210201123
Epoch:        11        0.31601698906774
Epoch:        12        0.29034451988978
```

```
Out[8]: Epoch:         13        0.26565687644075
        Epoch:         14        0.24537762059148
        Epoch:         15        0.22573977925808
        Epoch:         16        0.20809219151101
        Epoch:         17        0.19274937869357
```

```
Out[8]: Epoch:         18        0.17790250866323
        Epoch:         19        0.16478886826202
        Epoch:         20        0.15308249381824
```

Note that training error goes down after every epoch, as expected.