COMP755-Lect23

November 13, 2018

1 COMP 755

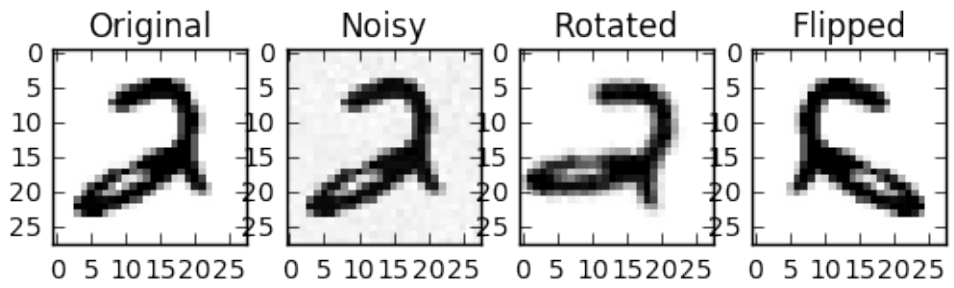
Today:
Tricks for training Neural Networks

2 Training neural networks is challenging

Key tricks 1. Data augmentation -- create more samples 2. Intelligent initialization 3. Improved optimization -- beyond gradient descent 4. Regularization -- weight decay, dropout

In [49]:
```
import numpy as np
from sklearn.datasets import fetch_mldata
import skimage.transform as trans
import matplotlib.pyplot as plt
%matplotlib inline

mnist = fetch_mldata('MNIST original')


show = lambda x: plt.imshow(x,cmap='gray_r',interpolation='None')
digit = mnist.data[15000].reshape((28,28))
plt.figure(figsize=(6,24))
plt.subplot(1,4,1)
show(digit)
plt.title('Original')
plt.subplot(1,4,2)
show(digit+5*np.random.randn(28,28))
plt.title('Noisy')
plt.subplot(1,4,3)
show(trans.rotate(digit,-20))
plt.title('Rotated')
plt.subplot(1,4,4)
show(digit[:,::-1])
plt.title('Flipped')
```

Out[49]:
```
<matplotlib.text.Text at 0x6b0356d8>
```



3 Data augmentation

Data augmentation can mean two things: 1. adding new features 2. adding new samples

In deep learning, typically, data augmentation refers to adding new samples.

For example, in MNIST data -- images of handwritten digits -- typically data augmentations are: 1. adding noise to pixels 2. rotating images 3. scaling, stretching, warping

The purpose is to make the trained network invariant to these small transformations of the data and thus robust.

Note that different types of data allow different augmentation.

For example, you would flip an image of a dog and it would still be a dog.

4 Issues with backprop

Recall back-prop recursion

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_j^{l-1}} = \sum_{i=1}^{n_l} \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) w_{ij}^l$$

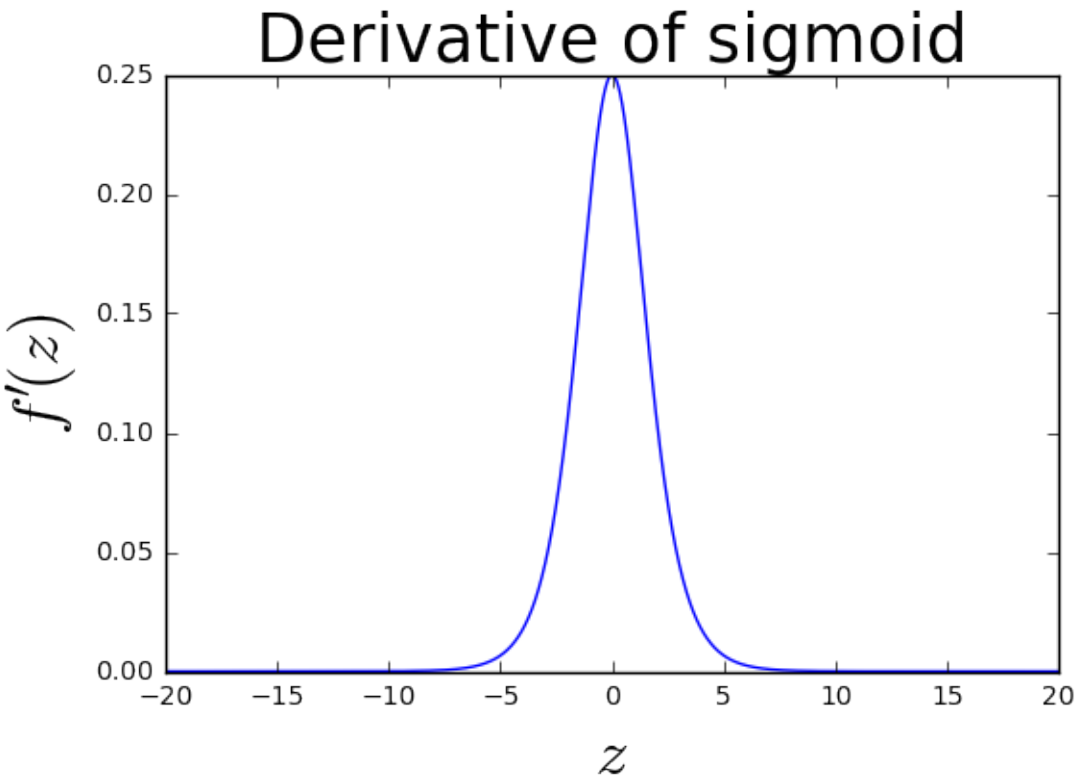
For sigmoidal units

$$f'(z_i^l) = \sigma\left(z_i^l\right)\left(1 - \sigma\left(z_i^l\right)\right)$$

where $\sigma(z) = \frac{1}{1+\exp -z}$

```
In [62]: z = np.arange(-20,20,0.01)
         sigmoid = lambda z: 1./(1. + np.exp(z))
         dsigmoid = lambda z: sigmoid(z)*(1 - sigmoid(z))

         plt.plot(z,dsigmoid(z))
         plt.title('Derivative of sigmoid',fontsize=25)
         plt.xlabel('$z$',fontsize=25)
         plt.ylabel('$f\'(z)$',fontsize=25)
```

```
Out[62]: <matplotlib.text.Text at 0x7e082978>
```



5 Issues with backprop and sigmoids

Derivatives for large inputs to sigmoid go to 0.

If we initialize networks's weights with large values, the derivatives are going to be near zero -- slow progress.

If we initalize nertworks's weight with small values, we need more iterations to get to larger values -- slow progress.

Typical intialiation for sigmoidal nets called Xavier initialization:

$$w_{i,j}^{l} \sim \text{Uniform}\left[-, \frac{\sqrt{6}}{\sqrt{n_l + n_{l-1}}}, \frac{\sqrt{6}}{\sqrt{n_l + n_{l-1}}}\right]$$

ensures that the variance of layer outputs stays nearly fixed across layers.

6 Issues with backprop and ReLU

Alternatively use rectified linear units ReLU. Their derivatives are constant.

$$\text{ReLU}(z) = \begin{cases} z, & z > 0, \\ 0, & z \leq 0 \end{cases}$$

$$\text{ReLU}'(z) = \begin{cases} 1, & z > 0, \\ 0, & z < 0 \end{cases}$$

Recall computation of derivatives for weights

$$\frac{\partial E}{w_{ij}^l} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) h_j^{l-1}$$

If a ReLU unit gets input $(z < 0)$ then $f'(z)$ becomes zero and consequently $\frac{\partial E}{w_{ij}^l}$ is zero.

If this unit is off $(z < 0)$ for all of the samples, the unit is "dead".

A "dead" unit's weights will not change because their derivatives are zero.

7 Issues with backprop and ReLU

An alternative is to use leaky ReLU:

$$\text{LReLU}(z) = \begin{cases} z, & z > 0, \\ \alpha z, & z \leq 0 \end{cases}$$

$$\text{LReLU}'(z) = \begin{cases} 1, & z > 0, \\ \alpha, & z < 0 \end{cases}$$

Leaky ReLU's do not turn off, and the derivative always is non-zero.

However, we add another parameter α that needs to be somehow tuned.

State of the art is a slight variation of the leaky ReLU called randomized leaky relu, which randomly assigns value α for each unit, throughout training.

8 Optimization issues -- learning rate

We typically use gradient descent to train neural nets

$$\mathbf{W} = \mathbf{W} - \eta \nabla_{\mathbf{W}} E(\mathbf{W})$$

Choice of a fixed η, referred to as **learning rate** or **step size**, is challenging.

Further some weights may require larger steps than others. Hence, even within a single iteration a single η might not be a good idea.

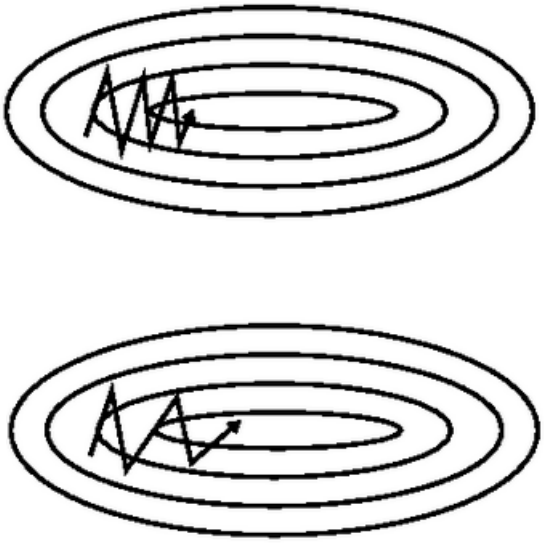
Rather than tweaking learning rate, we will talk about how the direction in which we step can be adjusted.

9 Optimization issues -- zig-zag

A typical gradient descent algorithm with fixed step size generates a zig-zag pattern

We would prefer to have our algorithm to move more along the x-axis

Key insight here is that the average of the gradients over several iterations would cancel out the bouncing along the y-axis and leave a dominant x-axis contribution.



10 Optimization issues -- using momentum to solve zig-zag

The weighted average of gradients over the past iterations is referred to as momentum.

Updates with momentum

$$\mathbf{v}^k = \gamma \mathbf{v}^{k-1} + \eta \nabla_{\mathbf{W}} E(\mathbf{W}^{k-1}) \quad (1)$$

$$\mathbf{W}^k = \mathbf{W}^{k-1} - \mathbf{v}^k \quad (2)$$

where $\gamma < 1.0$ is called momentum weight.

If you unroll this a bit

$$\mathbf{v}^k = \gamma^2 \mathbf{v}^{k-2} + \eta \left(\gamma \nabla_{\mathbf{W}} E(\mathbf{W}^{k-2}) + \nabla_{\mathbf{W}} E(\mathbf{W}^{k-1}) \right)$$

Hence, the direction along which we take a step $\mathbf{v}^k$ carries information about previous gradients.

The older gradients have smaller contribution γ^d for gradient d iterations ago.

A more sophisticated momentum method -- Nesterov's accelerated gradient

$$\mathbf{v}^k = \gamma \mathbf{v}^{k-1} + \eta \nabla_{\mathbf{W}} E(\mathbf{W}^{k-1} - \gamma \mathbf{v}^{k-1}) \quad (3)$$

$$(4)$$

11 Optimization issues -- gradient scaling techniques

A number of different optimization techniques scale and reshape the gradients

1. ADAGRAD
2. ADADelta
3. ADAM
4. RMSprop

12 Gradient scaling -- RMSprop

For each parameter, adjust the step-size based on how large its partial derivatives have been.

$$(\mathbf{m}^k)^2 = 0.9(\mathbf{m}^{k-1})^2 + 0.1\nabla_{\mathbf{W}}E(\mathbf{W}^{k-1})^2 \quad (5)$$

$$\mathbf{W}^k = \mathbf{W}^k - \frac{\eta}{\mathbf{m}^k}\nabla_{\mathbf{W}}E(\mathbf{W}^{k-1}) \quad (6)$$

The scaling is equal to a weighted average of gradient's squares across iterations.

Weights with large changes are "slowed down". Weights with small changes are "sped up".

13 Optimization issues -- dealing with large datasets

Large datasets are common. Even MNIST is fairly sizable 60k examples.

Typical neural network training does not require pass through all of the data to compute a gradient.

Instead, gradients are computed on small subsets of data (100s of samples) called mini-batches.

The training is decomposed into **epochs**.

In an epoch, training algorithm iterates over mini-batches and updates parameters after every mini-batch.

A recent paper shows that the mini-batch training actually can outperform full-batch training

14 Regularization -- weight decay

Large number of parameters in neural networks makes it regularization necessary.

Ridge penalty makes under appearance, under name **weight decay**.

Objective becomes

$$E(\mathbf{W}) + \frac{\lambda}{2}\sum_l\sum_{ij}(w_{ij}^l)^2$$

As usual this just means that gradients get a small modifiction

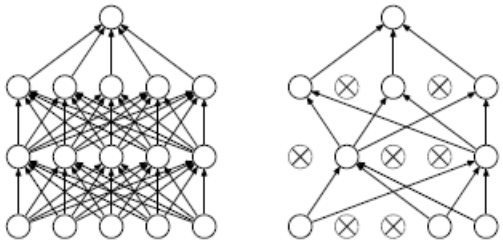
$$\frac{\partial E}{\partial w_{ij}^l} + \lambda w_{ij}^l$$

Weight decay, like ridge, shrinks weights down.

15 Regularization -- early stopping

Another effective way to ensure that weights do not get too large is to run the gradient descent for a small fixed number of iterations. This is called **early stopping**.

In effect, this is very similar to weight decay, but it also saves computational time.



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

16 Regularization -- dropout

One of the more sophisticated regularization techniques is **dropout**.

The idea is to randomly remove units from the network during the training phase. This prevents networks from replicating the same behavior in different units. Further, output of the network is more robust.

17 Demo of Theano

One particularly useful package for training neural nets is Theano

```
In [46]: import numpy
         import matplotlib.pyplot as plt
         import theano
         import theano.tensor as T

         def sample_gaussian(m,C,N):
             L = numpy.linalg.cholesky(C)
             d = len(m)
             # N samples with 0-mean and covariance C
             x = numpy.dot(L,numpy.random.randn(d,N))
             # use broadcasting to shift from 0-mean to m
             # similar to doing repmat(m,[1 N]) + x
             x = x + m
             return x

         def gen_mog(means,covs,N):
             d = means.shape[0]
             K = means.shape[1]
             assert d == covs.shape[0] and d == covs.shape[1]
             assert K == covs.shape[2]

             h = numpy.random.choice(K,size=N)
             x = numpy.empty((d,N))
```

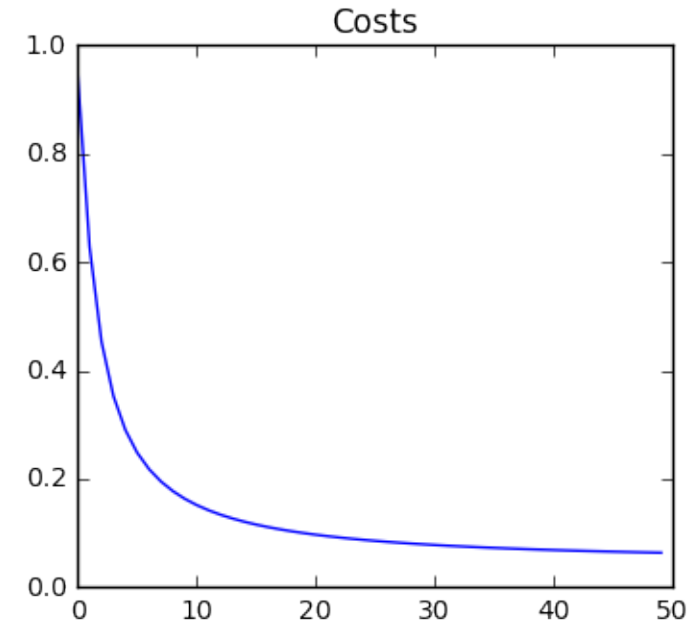
```
    for k in range(K):
        m = means[:,k]
        C = covs[:,:,k]
        ind = [i for i in range(len(h)) if h[i]==k]
        x[:,ind] = sample_gaussian(m,C,len(ind))

    x = x.astype('float32')
    return x,h

def plot_data_2d(x,h,figid = "data",marker='o',prefix=None):
    plt.title(figid)
    h= h.astype('int')
    N = x.shape[1]
    if prefix is not None:
        prefix = prefix + ' '
    else:
        prefix = ''
    for c in range(max(h)+1):
        ind = [i for i in range(N) if h[i] == c]
        if not ind == []:
            label = prefix + 'Class ' + str(c)
            plt.plot(x[0,ind],x[1,ind],marker,label = label)
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

def plot_line(w,b,x0,x1):
    y0 = -(w[0]*x0 + b)/w[1];
    y1 = -(w[0]*x1 + b)/w[1];
    plt.plot([x0,x1],[y0,y1],'-')

def plot_ws(w,b):
    axes = plt.gca()
    xlim = axes.get_xlim()
    ylim = axes.get_ylim()
    if len(w.shape)>1:
        for i in range(w.shape[1]):
            plot_line(w[:,i],b[i],xlim[0],xlim[1])
    else:
        plot_line(w,b,xlim[0],xlim[1])
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    plt.show()

def plot_costs(costs):
    plt.plot(costs)
    plt.title('Costs')


numpy.random.seed(1)
```

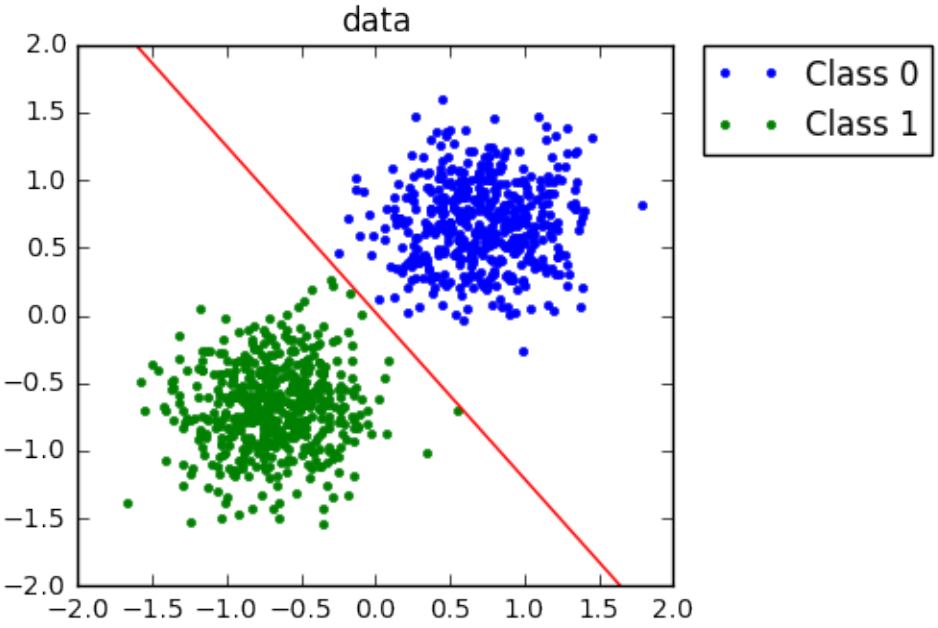
```
N = 1000
means = numpy.matrix([[ -0.7,0.7],
                      [-0.7, 0.7]])
K = means.shape[1]
d = means.shape[0]
covs = numpy.empty((d, d, K))
for i in range(K):
    covs[:, :, i] = numpy.diag([0.1, 0.1])

data,labels = gen_mog(means, covs, N)
label_to_predict = (labels == 0).astype('float32')
```

In [47]:
```
import numpy
import theano
import theano.tensor as T


theano.config.optimizer = 'None'
theano.config.exception_verbosity = 'high'

X = T.matrix('X')
y = T.vector('y')

w = theano.shared(numpy.random.randn(2), name='w')
b = theano.shared(0., name='b')
h = 1.0 / (1.0 + T.exp(-T.dot(X.T,w) - b))

ypredicted = h > 0.5

lam = 0.001
lr = 1
iterations = 50

cost = 1.0/float(N)*T.sum(-y*T.log(h) - (1.0 - y)*T.log(1.0-h))
cost = cost + lam*T.sum(w ** 2)
print cost.eval({X:data,y:label_to_predict})

gradw,gradb = T.grad(cost,[w,b])
updates = [(w, w - lr*gradw),(b, b - lr*gradb)]
train = theano.function(inputs = [X,y],
                        outputs = [cost],
                        updates = updates)
costs = []
for i in range(iterations):
    cost_value = train(data,label_to_predict)
    costs.append(cost_value[0])

plt.figure(figsize=(4,8))
```

```
        plt.subplot(2,1,1)
        plot_costs(costs)

        plt.subplot(2,1,2)
        plot_data_2d(data,
                     ypredicted.eval({X:data}),
                    marker='.')
        plot_ws(w.eval(),b.eval())
```

```
0.946480578266
```





18 A very interesting demo of convolutional nets in JavaScript

http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

19 Today

We skimmed the surface of practical issues involved in training neural nets.

Recommendations: 1. Leaky relu 2. One of the ADA-methods or rmsProp for optimization 3. Use dropout and weight-decay 4. Do not implement from scratch -- use Theano, MXNet, TensorFlow or Caffe

You do need to understand what different terms mean in order to use these techniques effectively.