A Basic Probabilistic, Linear Classifier: Logistic Regression

Yueming Wang

2020

Logistic Regression: Overview (binary)

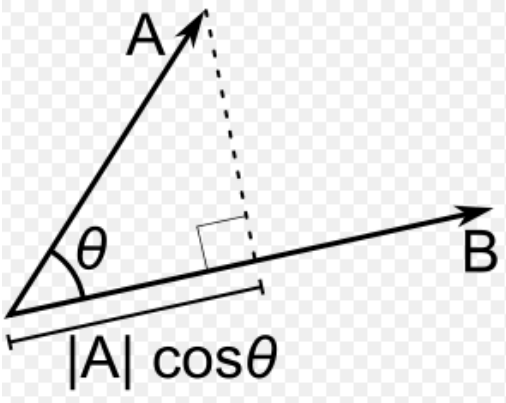
Logistic regression is a probabilistic, linear classifier. It is parametrized by a weight vector w and a bias vector b. Classification is done by projecting an input vector onto a hyperplane. The distance from the input to the hyperplane reflects the probability that the input is a member of the corresponding class.

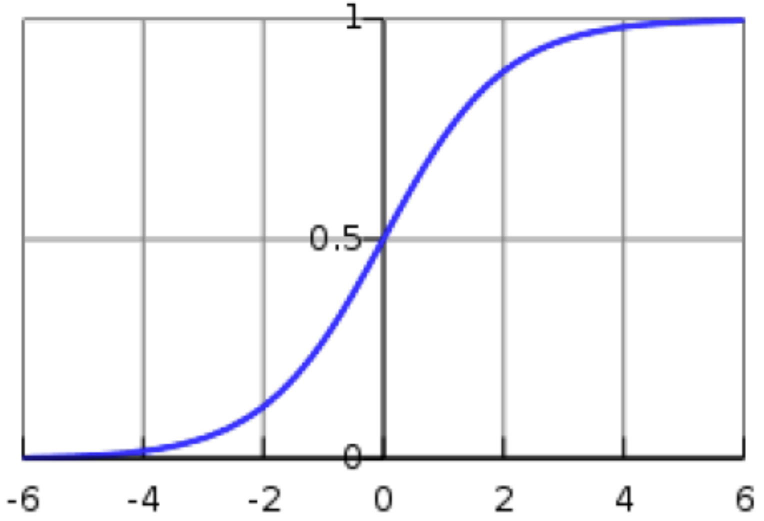
$$w^T x + b = \theta^T x',$$

$$x' = [\begin{matrix} x \\ 1 \end{matrix}], \theta = [\begin{matrix} w \\ b \end{matrix}]$$

$$s(x) = \frac{1}{1+e^{-x}}$$

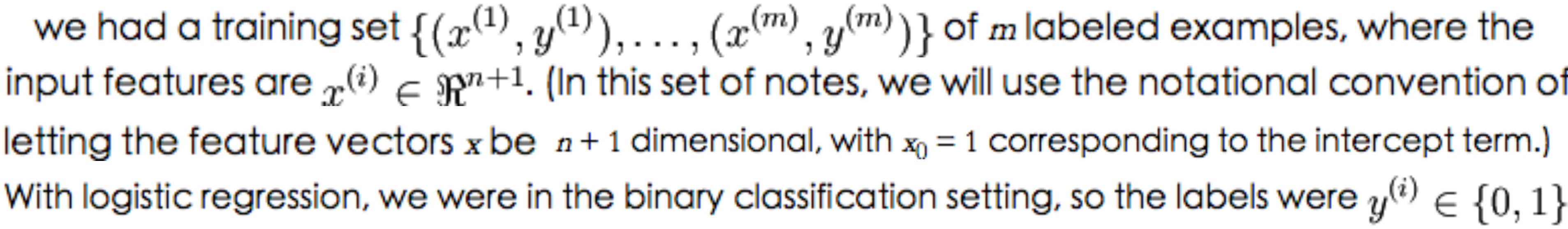
Sigmoid function



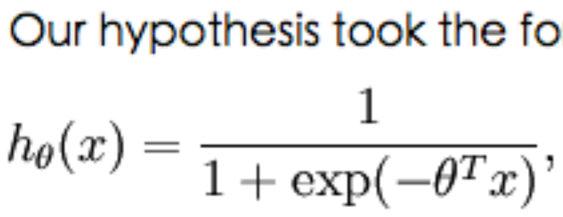


$$P(y = 1 | x'; \theta) = 1/(1 + e^{-\theta^T x'})$$

Go one step ahead

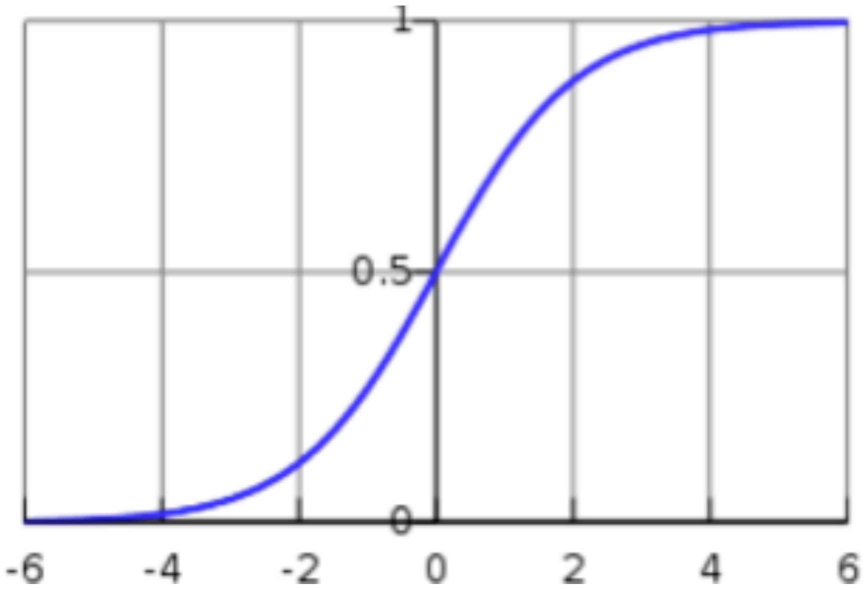


we had a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m labeled examples, where the input features are $x^{(i)} \in \Re^{n+1}$. (In this set of notes, we will use the notational convention of letting the feature vectors x be $n+1$ dimensional, with $x_0 = 1$ corresponding to the intercept term.) With logistic regression, we were in the binary classification setting, so the labels were $y^{(i)} \in \{0, 1\}$



Our hypothesis took the form:

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)},$$



Prediction:

$$y^{(i)} = 1, \text{ if } h_\theta(x^{(i)}) > 0.5$$

$$y^{(i)} = 0, \text{ otherwise}$$

Some Loss functions

Suppose we have 3 training samples with the following computed and target outputs:

Computed		Target	
0.4	0.6	0	1
0.7	0.3	1	0
0.46	0.54	1	0

The Mean Squared Error

The mean (average) squared error for this data is the sum of the squared errors divided by three.

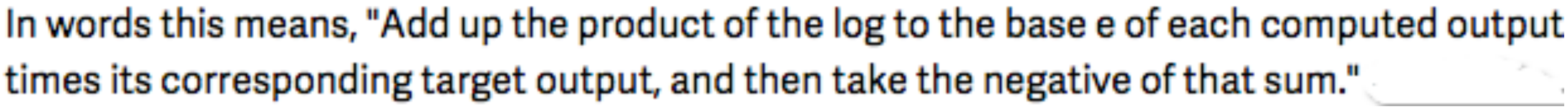
(0.4 - 0)^2 + (0.6 - 1)^2 = 0.16 + 0.16 = 0.32

(0.7 - 1)^2 + (0.3 - 0)^2 = 0.09 + 0.09 = 0.18

(0.46 - 1)^2 + (0.54 - 0)^2 = 0.29 + 0.29 = 0.58

Mean: (0.32+0.18+0.58)/3=0.36

The Mean Cross Entropy Error

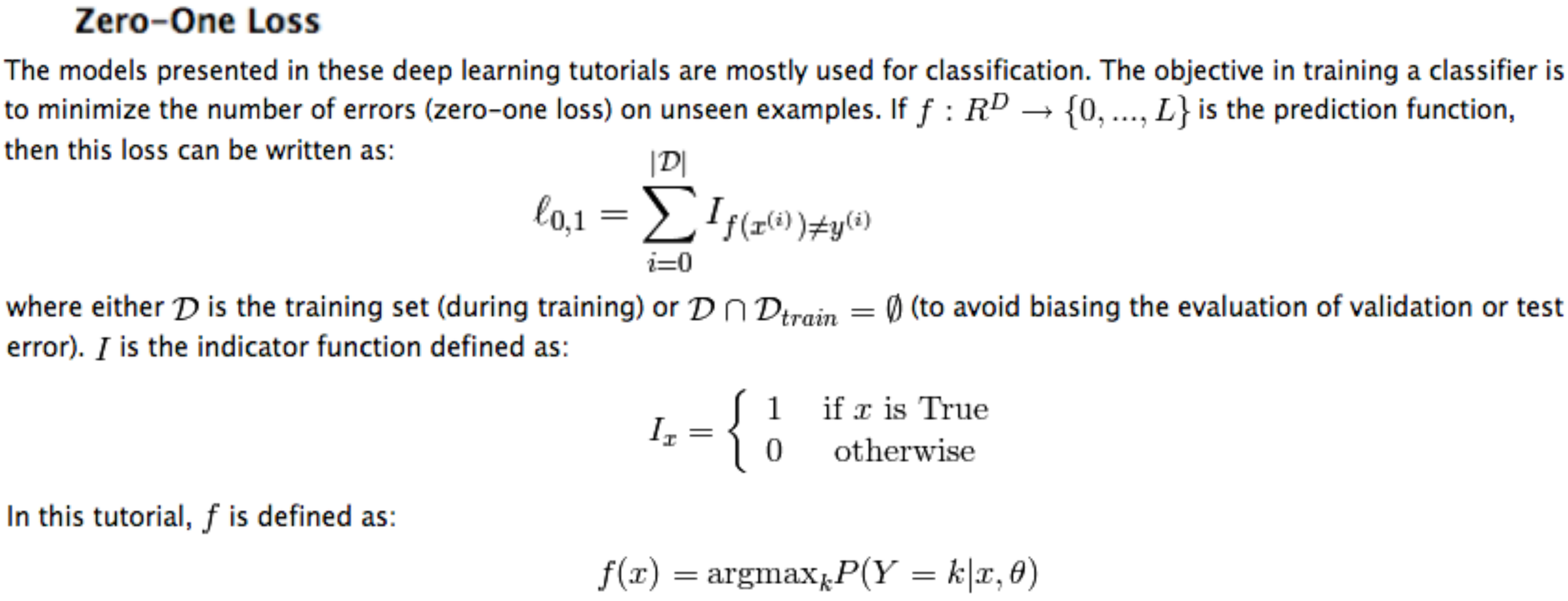


In words this means, "Add up the product of the log to the base e of each computed output times its corresponding target output, and then take the negative of that sum."

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=0}^{1} 1\left\{y^{(i)}=j\right\}\log p(y^{(i)}=j|x^{(i)};\theta)\right]$$

-(ln(0.4)*0 + ln(0.6)*1) = 0.51
-(ln(0.7)*1 + ln(0.3)*0) = 0.35
-(ln(0.46)*1 + ln(0.54)*0) = 0.77
Mean: 0.543

Some Loss functions



Zero-One Loss

The models presented in these deep learning tutorials are mostly used for classification. The objective in training a classifier is to minimize the number of errors (zero-one loss) on unseen examples. If $f : R^D \rightarrow \{0, ..., L\}$ is the prediction function, then this loss can be written as:

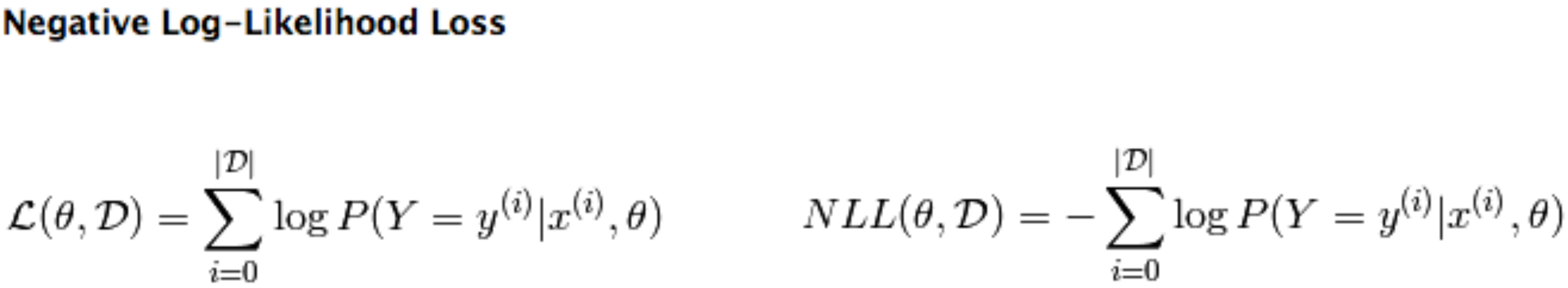
$$\ell_{0,1} = \sum_{i=0}^{|\mathcal{D}|} I_{f(x^{(i)}) \neq y^{(i)}}$$

where either $\mathcal{D}$ is the training set (during training) or $\mathcal{D} \cap \mathcal{D}_{train} = \emptyset$ (to avoid biasing the evaluation of validation or test error). I is the indicator function defined as:

$$I_x = \begin{cases} 1 & \text{if } x \text{ is True} \\ 0 & \text{otherwise} \end{cases}$$

In this tutorial, f is defined as:

$$f(x) = \text{argmax}_k P(Y = k | x, \theta)$$



Negative Log-Likelihood Loss

$$\mathcal{L}(\theta, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} \log P(Y = y^{(i)} | x^{(i)}, \theta)$$

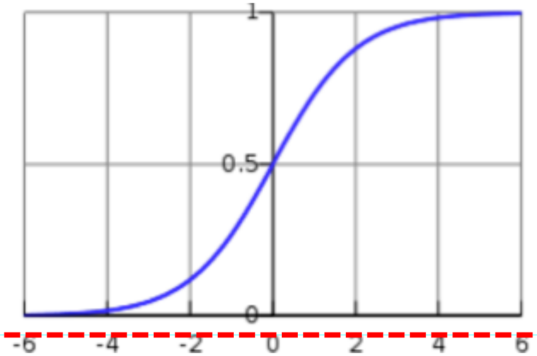
$$NLL(\theta, \mathcal{D}) = - \sum_{i=0}^{|\mathcal{D}|} \log P(Y = y^{(i)} | x^{(i)}, \theta)$$

Loss function in LR

we had a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m labeled examples, where the input features are $x^{(i)} \in \Re^{n+1}$. (In this set of notes, we will use the notational convention of letting the feature vectors x be $n+1$ dimensional, with $x_0 = 1$ corresponding to the intercept term.) With logistic regression, we were in the binary classification setting, so the labels were $y^{(i)} \in \{0, 1\}$

Our hypothesis took the form:

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)},$$

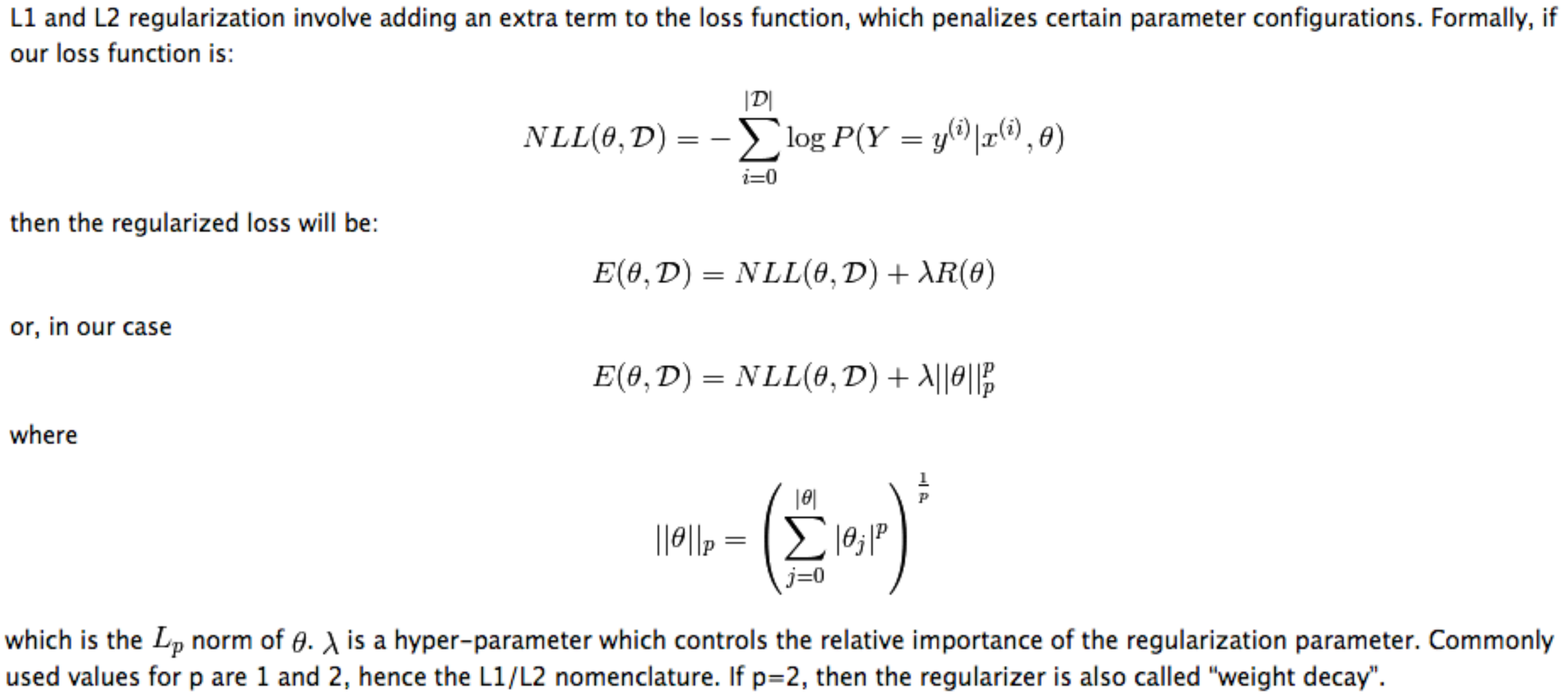


The Mean Cross Entropy Error

the model parameters θ were trained to minimize the cost function

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$$

L1 and L2 regularization



L1 and L2 regularization involve adding an extra term to the loss function, which penalizes certain parameter configurations. Formally, if our loss function is:

$$NLL(\theta, \mathcal{D}) = - \sum_{i=0}^{|\mathcal{D}|} \log P(Y = y^{(i)} | x^{(i)}, \theta)$$

then the regularized loss will be:

$$E(\theta, \mathcal{D}) = NLL(\theta, \mathcal{D}) + \lambda R(\theta)$$

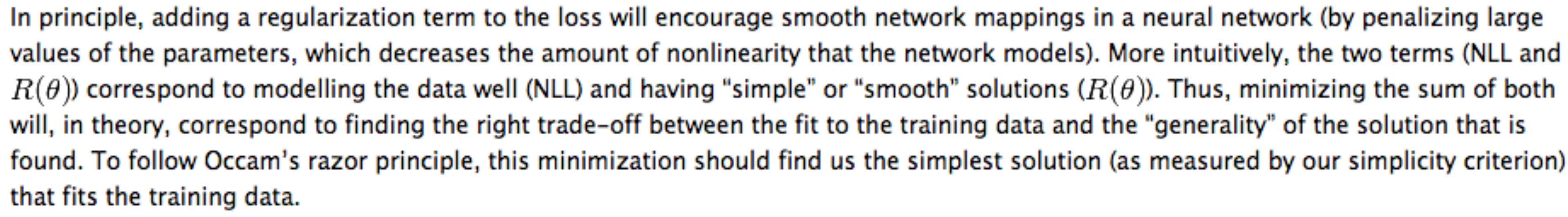
or, in our case

$$E(\theta, \mathcal{D}) = NLL(\theta, \mathcal{D}) + \lambda ||\theta||_p^p$$

where

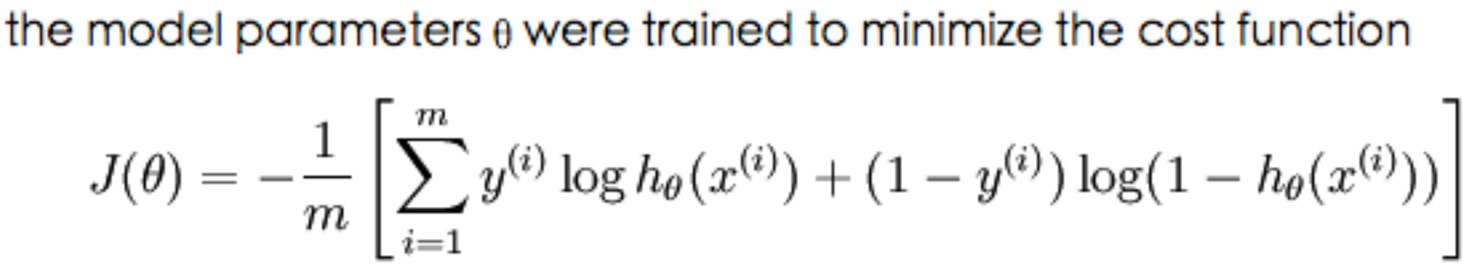
$$||\theta||_p = \left(\sum_{j=0}^{|\theta|} |\theta_j|^p \right)^{\frac{1}{p}}$$

which is the L_p norm of θ. λ is a hyper-parameter which controls the relative importance of the regularization parameter. Commonly used values for p are 1 and 2, hence the L1/L2 nomenclature. If p=2, then the regularizer is also called "weight decay".



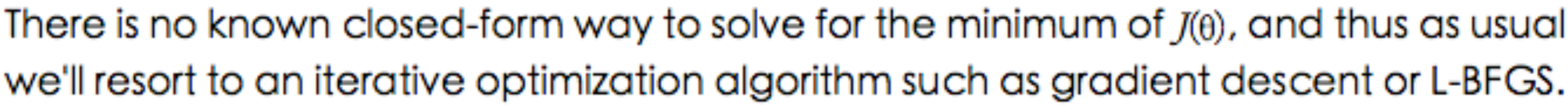
In principle, adding a regularization term to the loss will encourage smooth network mappings in a neural network (by penalizing large values of the parameters, which decreases the amount of nonlinearity that the network models). More intuitively, the two terms (NLL and $R(\theta)$) correspond to modelling the data well (NLL) and having "simple" or "smooth" solutions ($R(\theta)$). Thus, minimizing the sum of both will, in theory, correspond to finding the right trade-off between the fit to the training data and the "generality" of the solution that is found. To follow Occam's razor principle, this minimization should find us the simplest solution (as measured by our simplicity criterion) that fits the training data.

Solving it



the model parameters θ were trained to minimize the cost function

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \lambda \|\theta\|_2^2$$

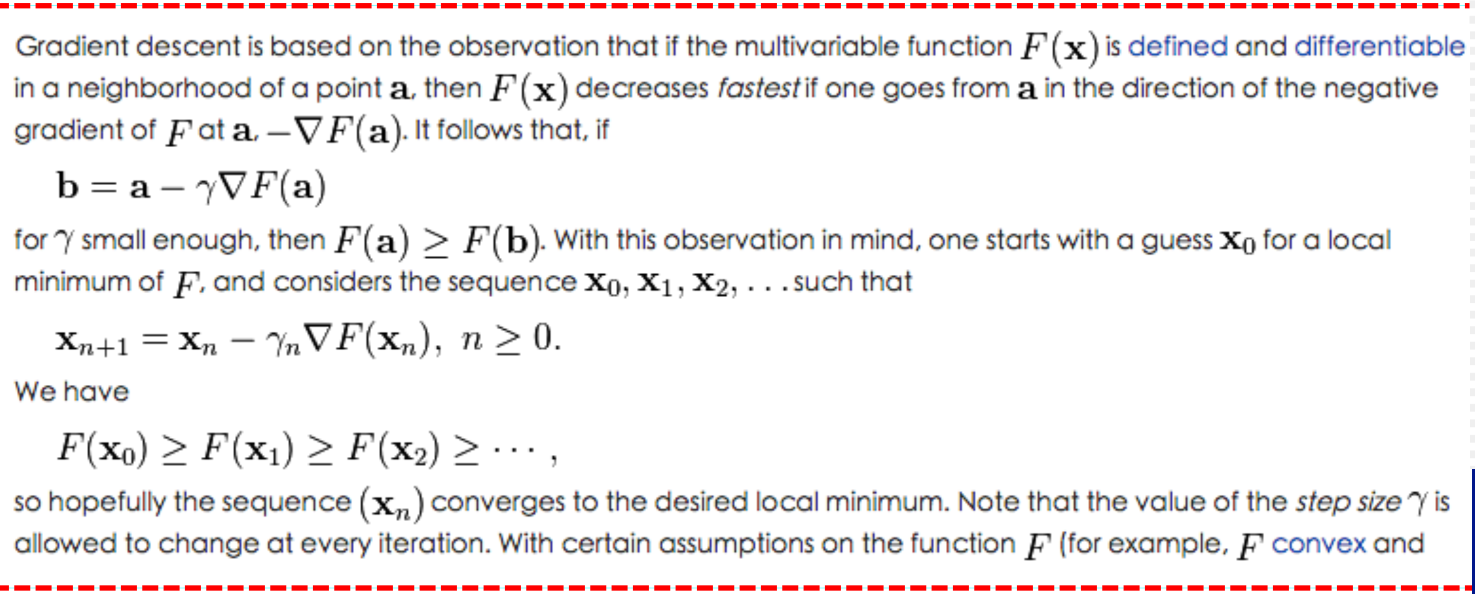


There is no known closed-form way to solve for the minimum of J(θ), and thus as usual we'll resort to an iterative optimization algorithm such as gradient descent or L-BFGS.

Gradient Descent

Prerequisite: What's Gradient Descent?

Gradient descent is a first-order optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point.



Gradient descent is based on the observation that if the multivariable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases *fastest* if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$. It follows that, if

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a})$$

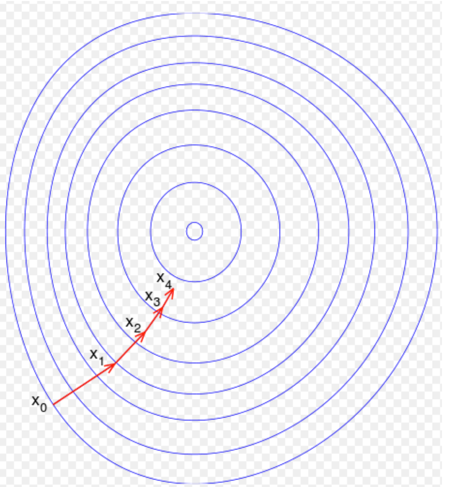
for γ small enough, then $F(\mathbf{a}) \geq F(\mathbf{b})$. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of F, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n),\ n \geq 0.$$

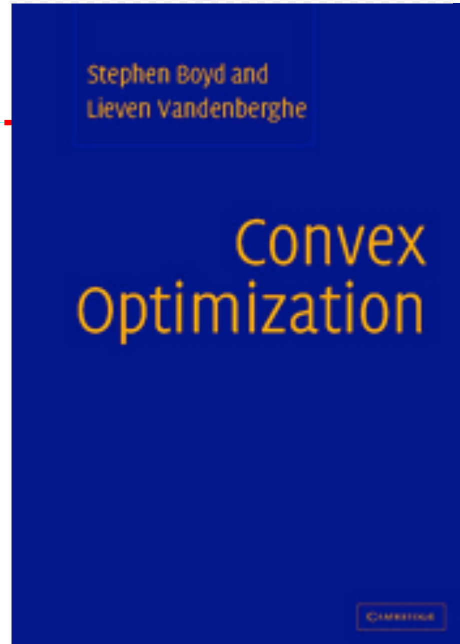
We have

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \cdots,$$

so hopefully the sequence $(\mathbf{x}_n)$ converges to the desired local minimum. Note that the value of the *step size* γ is allowed to change at every iteration. With certain assumptions on the function F (for example, F convex and



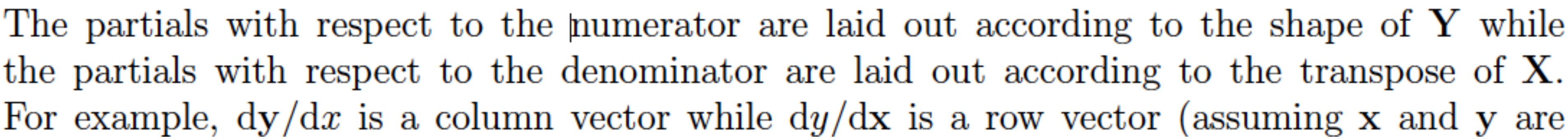
1. Set iteration counter $k = 0$, and make an initial guess, $\mathbf{x}_0$ for the minimum
2. Repeat:
3. Compute a descent direction $\mathbf{p}_k$
4. Choose α_k to 'loosely' minimize $h(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ over $\alpha \in \mathbb{R}_+$
5. Update $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$, and $k = k + 1$
6. Until $\|\nabla f(\mathbf{x}_k)\|$ < tolerance



Prerequisite: Matrix, vector Calculus

标量对向量/矩阵(或反过来)存在两种不同的标记方法，区别在于结果写成行向量形式或列向量形式，(Numerator/Denominator layout)中文网上参考五花八门，不靠谱！

Numerator layout:


The partials with respect to the numerator are laid out according to the shape of $\mathbf{Y}$ while the partials with respect to the denominator are laid out according to the transpose of $\mathbf{X}$. For example, $\mathrm{d}\mathbf{y}/\mathrm{d}x$ is a column vector while $\mathrm{d}y/\mathrm{d}\mathbf{x}$ is a row vector (assuming $\mathbf{x}$ and $\mathbf{y}$ are

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \\ \vdots \\ \frac{\partial y_m}{\partial x} \end{bmatrix}$$

$$\frac{\partial \mathbf{Y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{12}}{\partial x} & \cdots & \frac{\partial y_{1n}}{\partial x} \\ \frac{\partial y_{21}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \cdots & \frac{\partial y_{2n}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{m1}}{\partial x} & \frac{\partial y_{m2}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$$

$$\frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{u}))}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}} \frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

$$\frac{\partial \mathbf{x}^\top \mathbf{A} \mathbf{x}}{\partial \mathbf{x}} = \mathbf{x}^\top (\mathbf{A} + \mathbf{A}^\top)$$

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Jacobian

$$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{21}} & \cdots & \frac{\partial y}{\partial x_{p1}} \\ \frac{\partial y}{\partial x_{12}} & \frac{\partial y}{\partial x_{22}} & \cdots & \frac{\partial y}{\partial x_{p2}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial x_{1q}} & \frac{\partial y}{\partial x_{2q}} & \cdots & \frac{\partial y}{\partial x_{pq}} \end{bmatrix}$$

$$\frac{\partial\, \mathbf{a}^\top \mathbf{x} \mathbf{x}^\top \mathbf{b}}{\partial\, \mathbf{x}} = \mathbf{x}^\top (\mathbf{a}\mathbf{b}^\top + \mathbf{b}\mathbf{a}^\top)$$

更多？更全，见楼下参考文献

T. P. Minka, *Old and New Matrix Algebra Useful for Statistics*, 2000. (Microsoft Research).

Wiki: *Matrix calculus*

Gradient in LR

the model parameters θ were trained to minimize the cost function

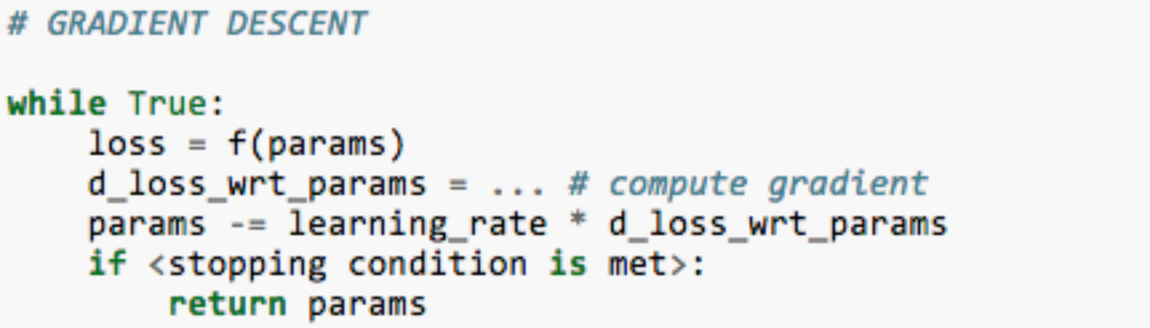
$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \tfrac{1}{2}\lambda\|\theta\|_2^2$$

where $h_\theta = \dfrac{1}{1+e^{-\theta^T x}}$ $\quad \dfrac{dh(z)}{dz} = h(z)(1-h(z))$ $\quad \dfrac{dh_\theta}{d\theta} = \dfrac{dh_\theta}{d(\theta^T x)}\dfrac{d(\theta^T x)}{d\theta} = h_\theta(1-h_\theta)x^T$

So gradient:

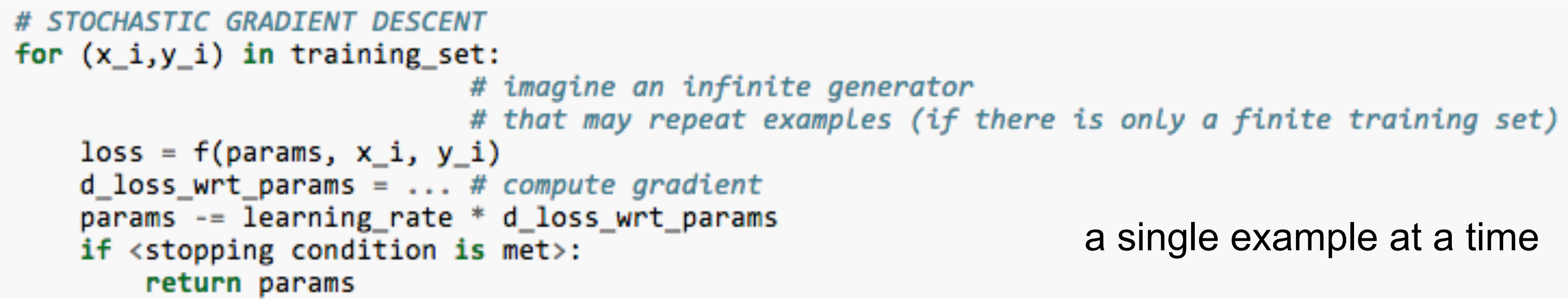
$$\begin{aligned}
\nabla_\theta J(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\frac{1}{h_\theta}\frac{dh_\theta}{d\theta} + (1-y^{(i)})\frac{1}{1-h_\theta}(-1)\frac{dh_\theta}{d\theta}] + \lambda\theta^T \\
&= -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\frac{1}{h_\theta}h_\theta(1-h_\theta)(x^{(i)})^T + (y^{(i)}-1)\frac{1}{1-h_\theta}h_\theta(1-h_\theta)(x^{(i)})^T] + \lambda\theta^T \\
&= -\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^T[y^{(i)}(1-h_\theta) + (y^{(i)}-1)h_\theta] + \lambda\theta^T \\
&= -\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^T(y^{(i)} - h_\theta) + \lambda\theta^T
\end{aligned}$$

Prerequisite: to Stochastic Gradient Descent



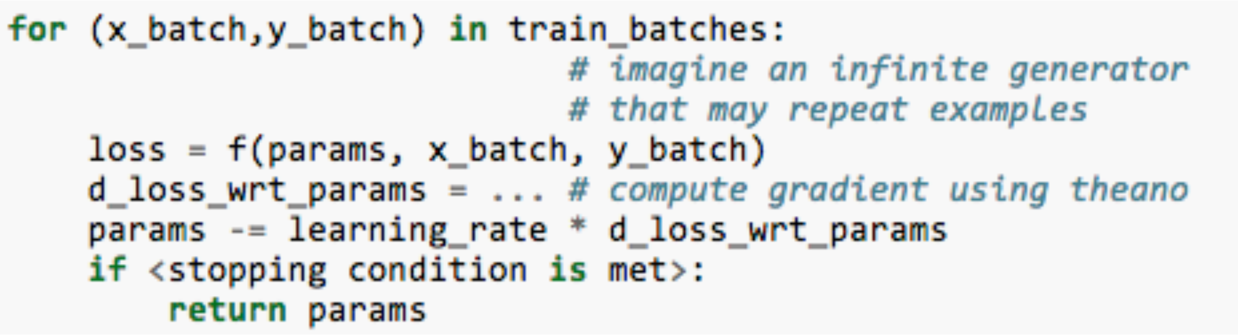
```
# GRADIENT DESCENT

while True:
    loss = f(params)
    d_loss_wrt_params = ... # compute gradient
    params -= learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```

the entire training set at a time



```
# STOCHASTIC GRADIENT DESCENT
for (x_i,y_i) in training_set:
                            # imagine an infinite generator
                            # that may repeat examples (if there is only a finite training set)
    loss = f(params, x_i, y_i)
    d_loss_wrt_params = ... # compute gradient
    params -= learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```

a single example at a time

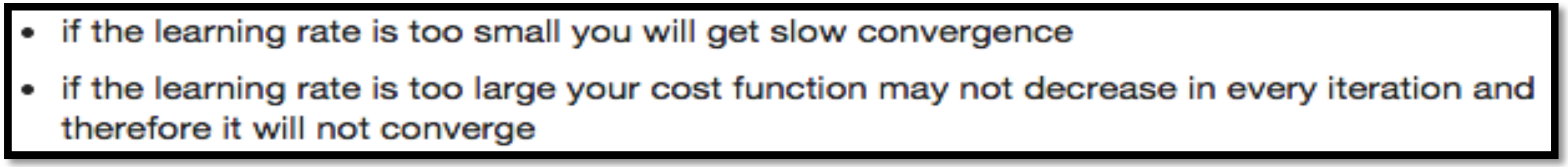


```
for (x_batch,y_batch) in train_batches:
                            # imagine an infinite generator
                            # that may repeat examples
    loss = f(params, x_batch, y_batch)
    d_loss_wrt_params = ... # compute gradient using theano
    params -= learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```

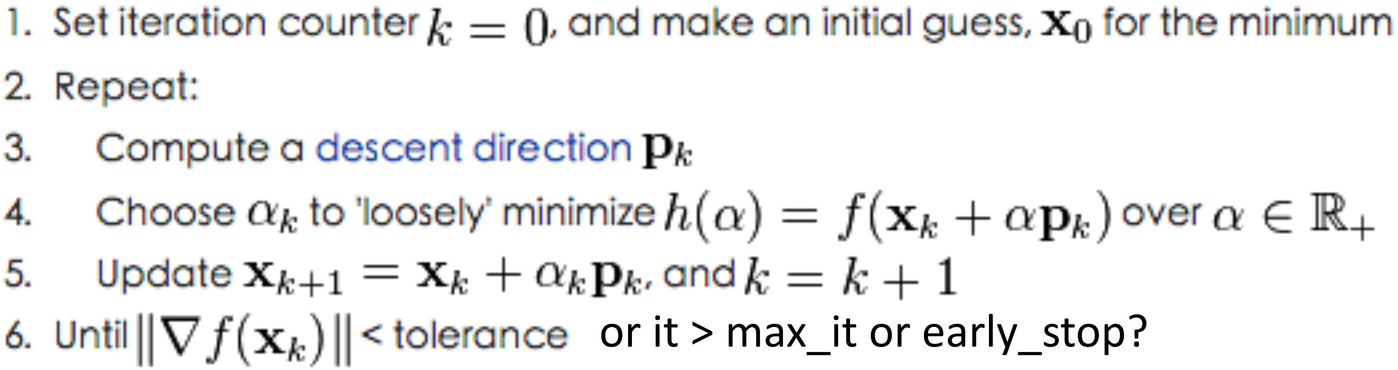
minibatches at a time
size: B

An optimal B is model-, dataset-, and hardware-dependent, and can be anywhere from 1 to maybe several hundreds, e.g. 20.

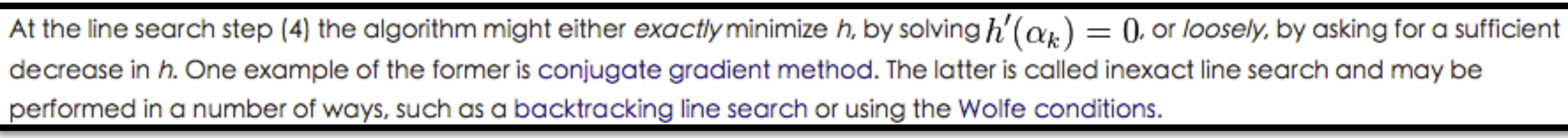
Prerequisite: How to select α



- if the learning rate is too small you will get slow convergence
- if the learning rate is too large your cost function may not decrease in every iteration and therefore it will not converge

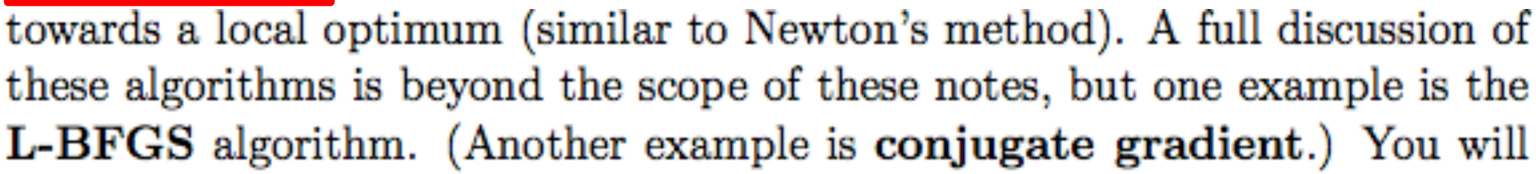


```
1. Set iteration counter $k = 0$, and make an initial guess, $\mathbf{x}_0$ for the minimum
2. Repeat:
3.     Compute a descent direction $\mathbf{p}_k$
4.     Choose $\alpha_k$ to 'loosely' minimize $h(\alpha) = f(\mathbf{x}_k + \alpha\mathbf{p}_k)$ over $\alpha \in \mathbb{R}_+$
5.     Update $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$, and $k = k + 1$
6. Until $\|\nabla f(\mathbf{x}_k)\|$ < tolerance   or it > max_it or early_stop?
```



At the line search step (4) the algorithm might either *exactly* minimize *h*, by solving $h'(\alpha_k) = 0$, or *loosely*, by asking for a sufficient decrease in *h*. One example of the former is conjugate gradient method. The latter is called inexact line search and may be performed in a number of ways, such as a backtracking line search or using the Wolfe conditions.

Andrew Ng



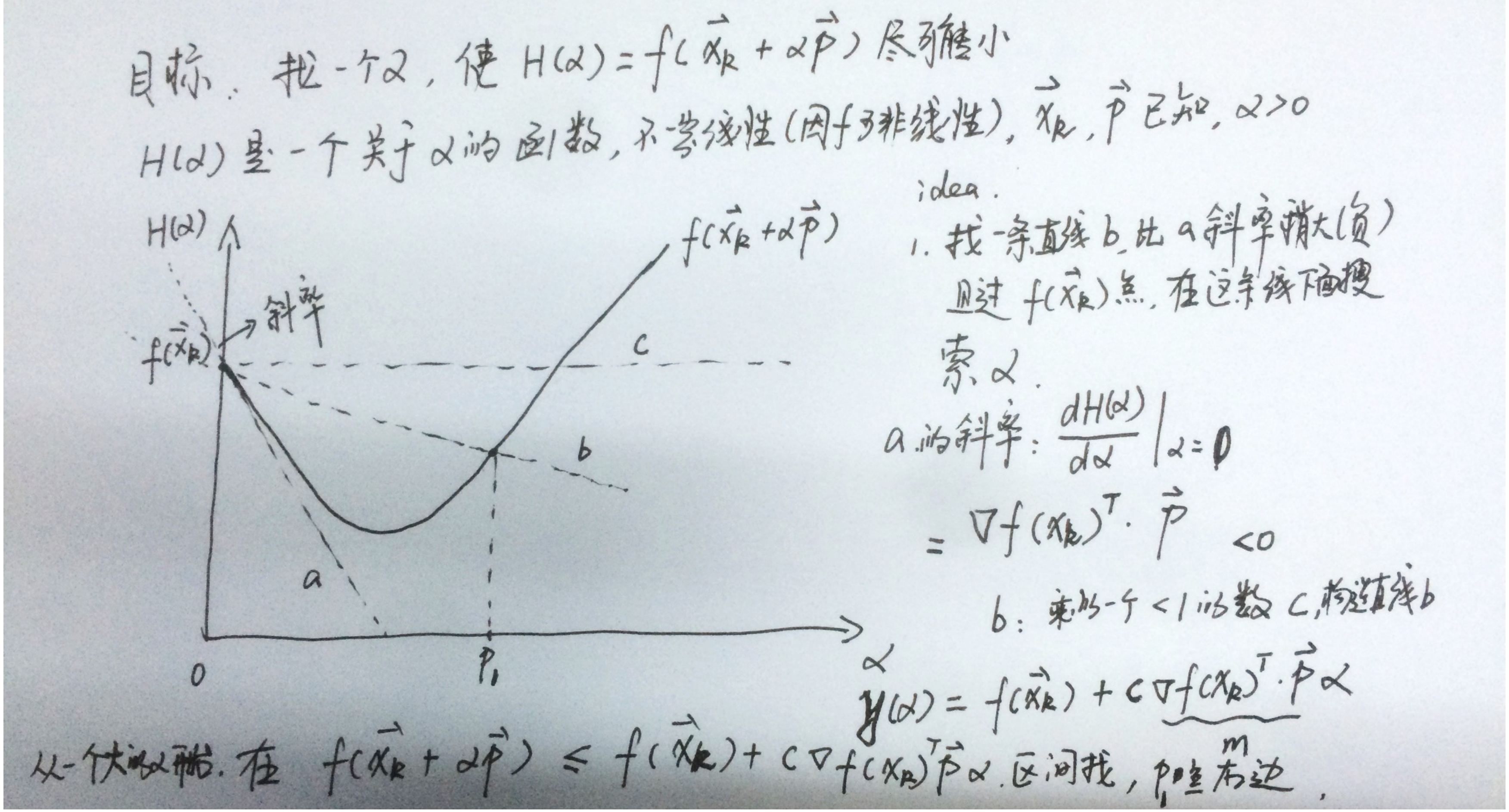
towards a local optimum (similar to Newton's method). A full discussion of these algorithms is beyond the scope of these notes, but one example is the **L-BFGS** algorithm. (Another example is **conjugate gradient**.) You will

L-BFGS

Conjugate Gradient

Prerequisite: How to select α

Understanding Backtracking line search (Armijo rule) $0 < c_1 < \frac{1}{2}$


$H(\alpha)$
$f(\vec{x}_k+\alpha\vec{p})$
斜率
$f(\vec{x}_k)$
c
b
a
0
p_1
α

目标：找一个α，使 $H(\alpha)=f(\vec{x}_k+\alpha\vec{p})$ 尽可能小

$H(\alpha)$ 是一个关于α的函数，不一定线性（因f可能非线性），$\vec{x}_k$，$\vec{p}$ 已知，$\alpha>0$

idea.

1. 找一条直线b 比a斜率稍大(负) 且过 $f(\vec{x}_k)$ 点，在这条线下面搜索α.

a的斜率：$\left.\frac{dH(\alpha)}{d\alpha}\right|_{\alpha=0}$

$= \nabla f(x_k)^T \cdot \vec{p} \quad < 0$

b：乘以一个<1的数 c，构造直线b

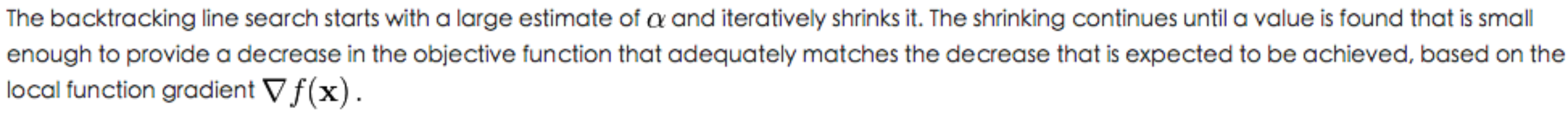
$y(\alpha) = f(\vec{x}_k) + c\,\underbrace{\nabla f(x_k)^T \cdot \vec{p}}_{m}\,\alpha$

从一个大的α开始，在 $f(\vec{x}_k+\alpha\vec{p}) \leq f(\vec{x}_k) + c\nabla f(x_k)^T\vec{p}\alpha$ 区间找，p_1左右边.

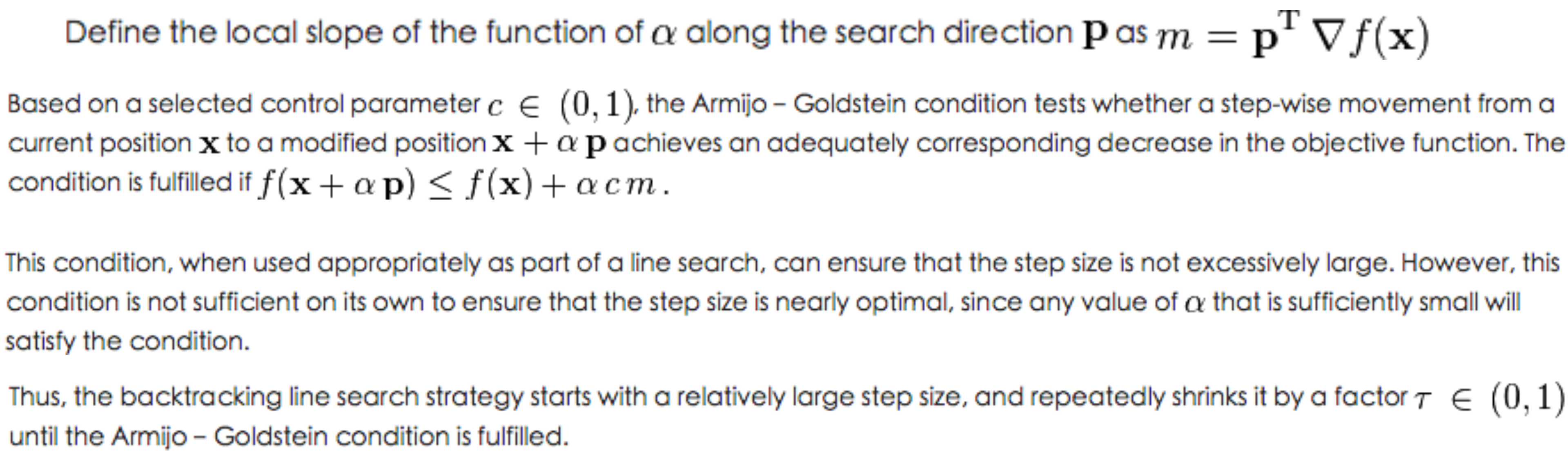
1. Set $t = -c\,m$ and iteration counter $j = 0$. $\alpha_j = 1$
2. Until the condition is satisfied that $f(\mathbf{x}) - f(\mathbf{x}+\alpha_j\,\mathbf{p}) \geq \alpha_j\,t$, repeatedly increment j and set $\alpha_j = \tau\,\alpha_{j-1}$.
3. Return α_j as the solution.

Prerequisite: How to select α

Backtracking line search



The backtracking line search starts with a large estimate of α and iteratively shrinks it. The shrinking continues until a value is found that is small enough to provide a decrease in the objective function that adequately matches the decrease that is expected to be achieved, based on the local function gradient $\nabla f(\mathbf{x})$.

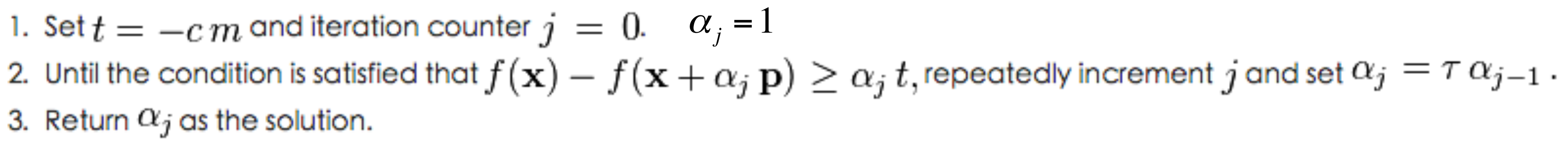


Define the local slope of the function of α along the search direction $\mathbf{p}$ as $m = \mathbf{p}^{\mathrm{T}} \nabla f(\mathbf{x})$

Based on a selected control parameter $c \in (0, 1)$, the Armijo – Goldstein condition tests whether a step-wise movement from a current position $\mathbf{x}$ to a modified position $\mathbf{x} + \alpha\,\mathbf{p}$ achieves an adequately corresponding decrease in the objective function. The condition is fulfilled if $f(\mathbf{x} + \alpha\,\mathbf{p}) \leq f(\mathbf{x}) + \alpha\, c\, m$.

This condition, when used appropriately as part of a line search, can ensure that the step size is not excessively large. However, this condition is not sufficient on its own to ensure that the step size is nearly optimal, since any value of α that is sufficiently small will satisfy the condition.

Thus, the backtracking line search strategy starts with a relatively large step size, and repeatedly shrinks it by a factor $\tau \in (0, 1)$ until the Armijo – Goldstein condition is fulfilled.



1. Set $t = -c\, m$ and iteration counter $j = 0$. $\alpha_j = 1$
2. Until the condition is satisfied that $f(\mathbf{x}) - f(\mathbf{x} + \alpha_j\,\mathbf{p}) \geq \alpha_j\, t$, repeatedly increment j and set $\alpha_j = \tau\, \alpha_{j-1}$.
3. Return α_j as the solution.

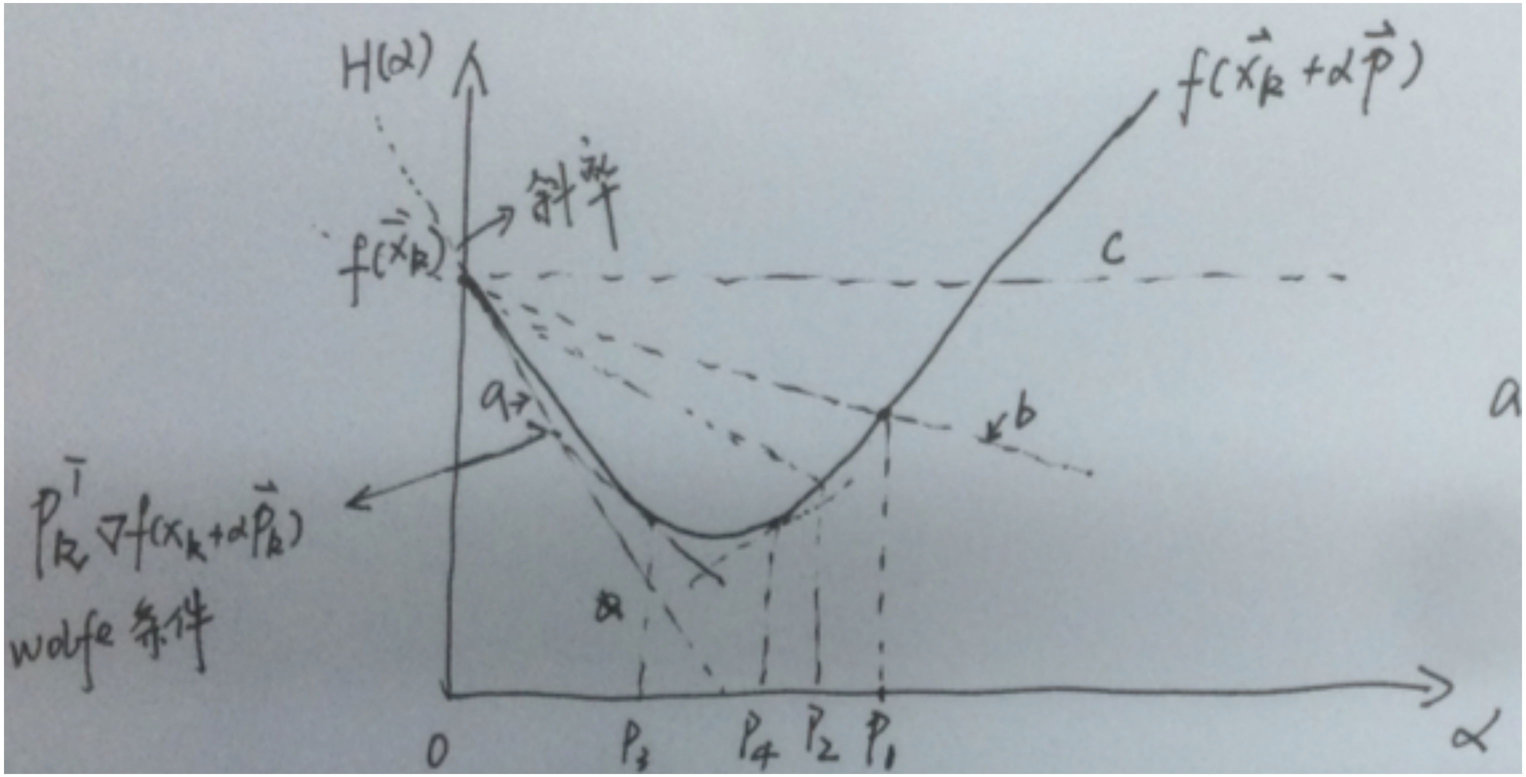
Armijo used $^1/_2$ for both c and τ

Prerequisite: How to select α

More, Wolfe condition

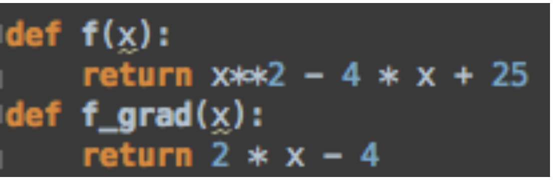
$$\mathbf{p}_k^{\mathrm{T}}\nabla f(\mathbf{x}_k+\alpha_k\mathbf{p}_k) \geq c_2\mathbf{p}_k^{\mathrm{T}}\nabla f(\mathbf{x}_k)$$

$$\left|\mathbf{p}_k^{\mathrm{T}}\nabla f(\mathbf{x}_k+\alpha_k\mathbf{p}_k)\right| \leq c_2\left|\mathbf{p}_k^{\mathrm{T}}\nabla f(\mathbf{x}_k)\right|$$

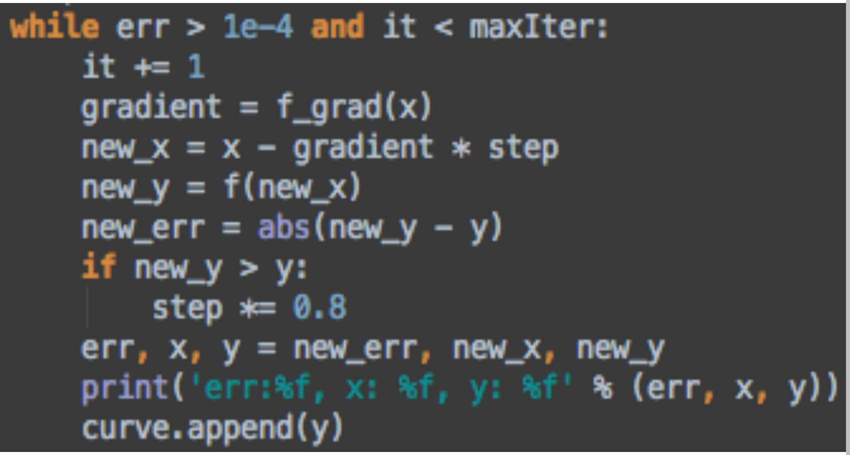


$$0 < c_1 < c_2 < 1$$

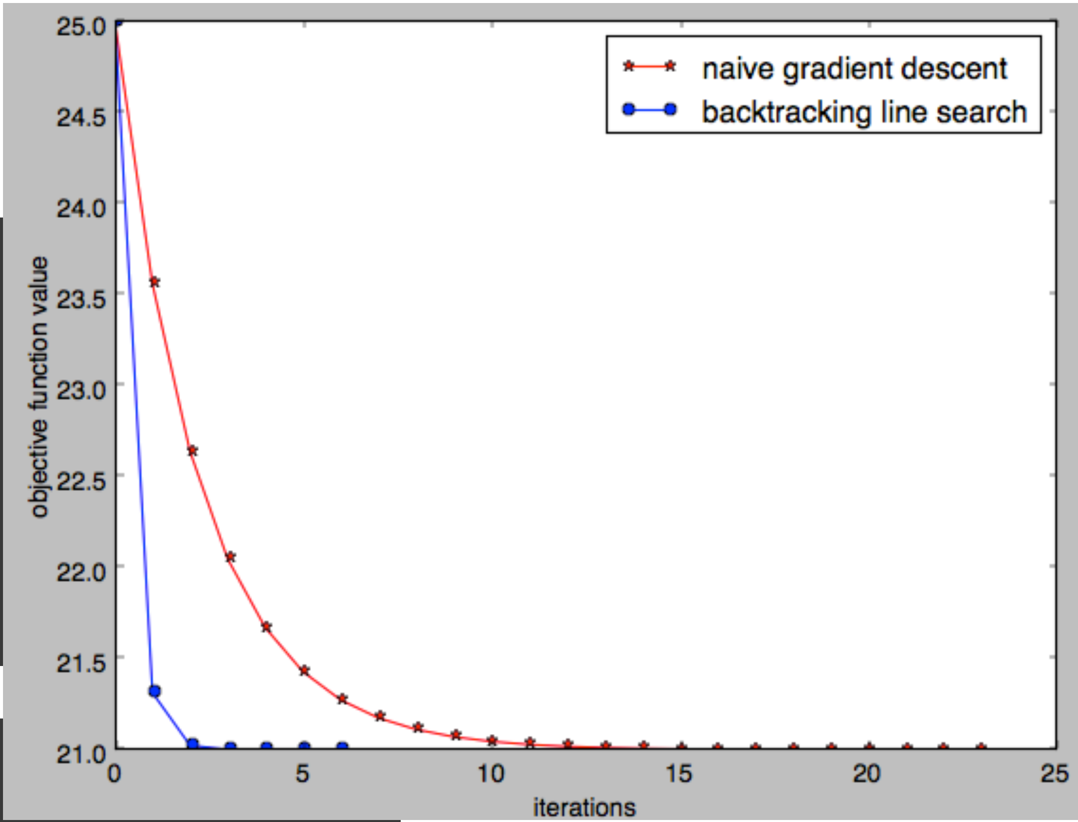
Prerequisite: Strength of line search

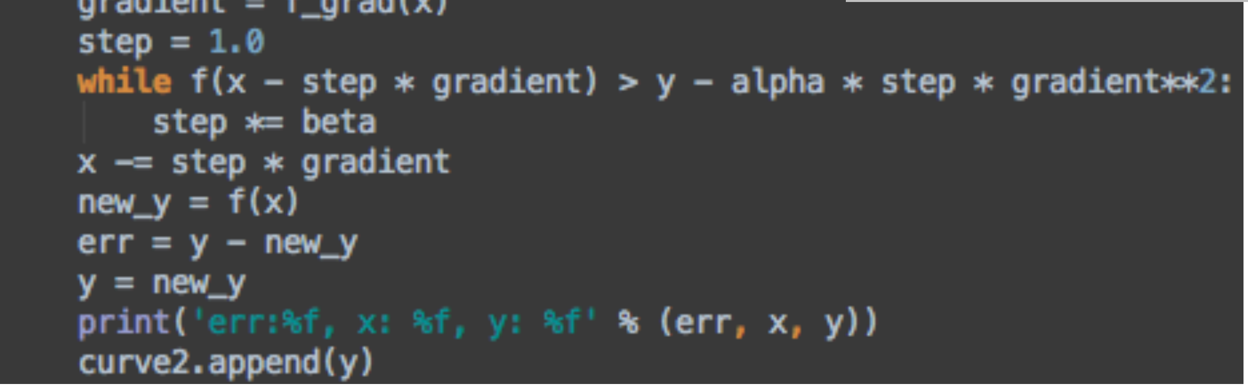


```
def f(x):
    return x**2 - 4 * x + 25
def f_grad(x):
    return 2 * x - 4
```



```
while err > 1e-4 and it < maxIter:
    it += 1
    gradient = f_grad(x)
    new_x = x - gradient * step
    new_y = f(new_x)
    new_err = abs(new_y - y)
    if new_y > y:
        step *= 0.8
    err, x, y = new_err, new_x, new_y
    print('err:%f, x: %f, y: %f' % (err, x, y))
    curve.append(y)
```



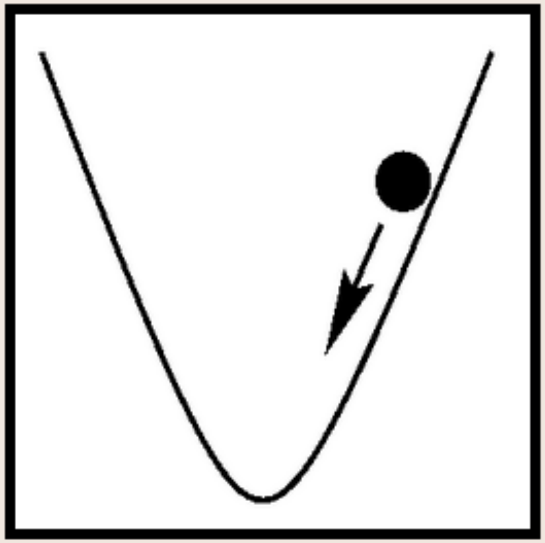


```
while err > 1e-4 and it < maxIter:
    it += 1
    gradient = f_grad(x)
    step = 1.0
    while f(x - step * gradient) > y - alpha * step * gradient**2:
        step *= beta
    x -= step * gradient
    new_y = f(x)
    err = y - new_y
    y = new_y
    print('err:%f, x: %f, y: %f' % (err, x, y))
    curve2.append(y)
```

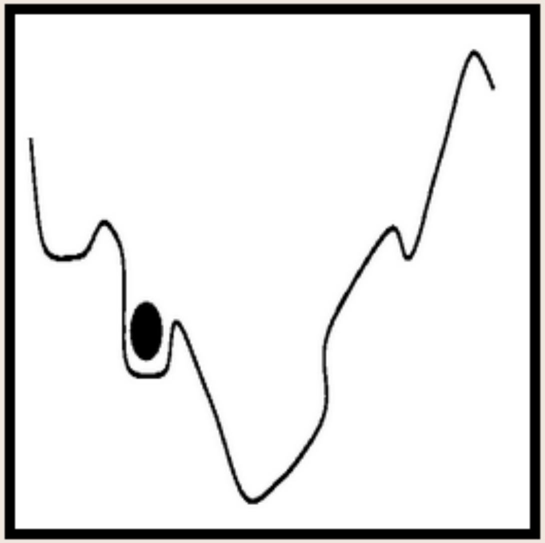
Momentum and Learning Rate Adaptation

Local Minima

In gradient descent we start at some point on the error function defined over the weights, and attempt to move to the global minimum of the function. In the simplified function of Fig 1a the situation is simple. Any step in a downward direction will take us closer to the global minimum. For real problems, however, error surfaces are typically complex, and may more resemble the situation shown in Fig 1b. Here there are numerous local minima, and the ball is shown trapped in one such minimum. Progress here is only possible by climbing higher before descending to the global minimum.



(Fig. 1a)



(Fig. 1b)

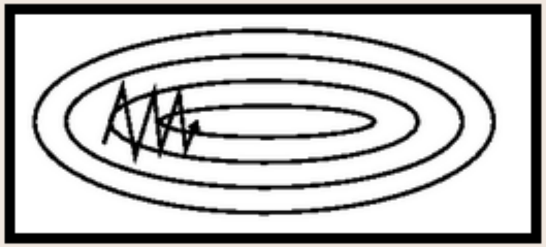
Momentum

Another technique that can help the network out of local minima is the use of a momentum term. This is probably the most popular extension of the backprop algorithm; it is hard to find cases where this is not used. With momentum m, the weight update at a given time t becomes

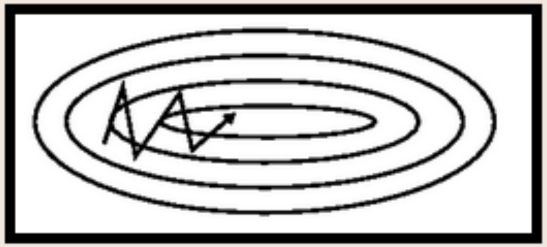
$$\Delta w_{ij}(t) \;=\; \mu_i \, \delta_i \, y_j \;+\; m \, \Delta w_{ij}(t-1) \qquad (1)$$

where 0 < m < 1 is a new global parameter which must be determined by trial and error. Momentum simply adds a fraction m of the previous weight update to the current one. When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum. It is otherefore often necessary to reduce the global learning rate μ when using a lot of momentum (m close to 1). If you combine a high learning rate with a lot of momentum, you will rush past the minimum with huge steps!

When the gradient keeps changing direction, momentum will smooth out the variations. This is particularly useful when the network is not well-conditioned. In such cases the error surface has substantially different curvature along different directions, leading to the formation of long narrow valleys. For most points on the surface, the gradient does not point towards the minimum, and successive steps of gradient descent can oscillate from one side to the other, progressing only very slowly to the minimum (Fig. 2a). Fig. 2b shows how the addition of momentum helps to speed up convergence to the minimum by damping these oscillations.



(Fig. 2a)



(Fig. 2b)

To illustrate this effect in practice, we trained 20 networks on a simple problem (4-2-4 encoding), both with and without momentum. The mean training times (in epochs) were

momentum	Training time
0	217
0.9	95

Prerequisite: An Example

Consider a nonlinear system of equations:

$$\begin{cases} 3x_1 - \cos(x_2x_3) - \frac{3}{2} = 0 \\ 4x_1^2 - 625x_2^2 + 2x_2 - 1 = 0 \\ \exp(-x_1x_2) + 20x_3 + \frac{10\pi-3}{3} = 0 \end{cases}$$

suppose we have the function

$$G(\mathbf{x}) = \begin{bmatrix} 3x_1 - \cos(x_2x_3) - \frac{3}{2} \\ 4x_1^2 - 625x_2^2 + 2x_2 - 1 \\ \exp(-x_1x_2) + 20x_3 + \frac{10\pi-3}{3} \end{bmatrix} \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

and the objective function

$$F(\mathbf{x}) = \tfrac{1}{2}G^{\mathrm{T}}(\mathbf{x})G(\mathbf{x})$$
$$= \tfrac{1}{2}((3x_1 - \cos(x_2x_3) - \tfrac{3}{2})^2 + (4x_1^2 - 625x_2^2 + 2x_2 - 1)^2 + (\exp(-x_1x_2) + 20x_3 + \tfrac{10\pi-3}{3})^2)$$

$$\nabla F(\mathbf{x}^{(0)}) = J_G(\mathbf{x}^{(0)})^{\mathrm{T}} G(\mathbf{x}^{(0)})$$

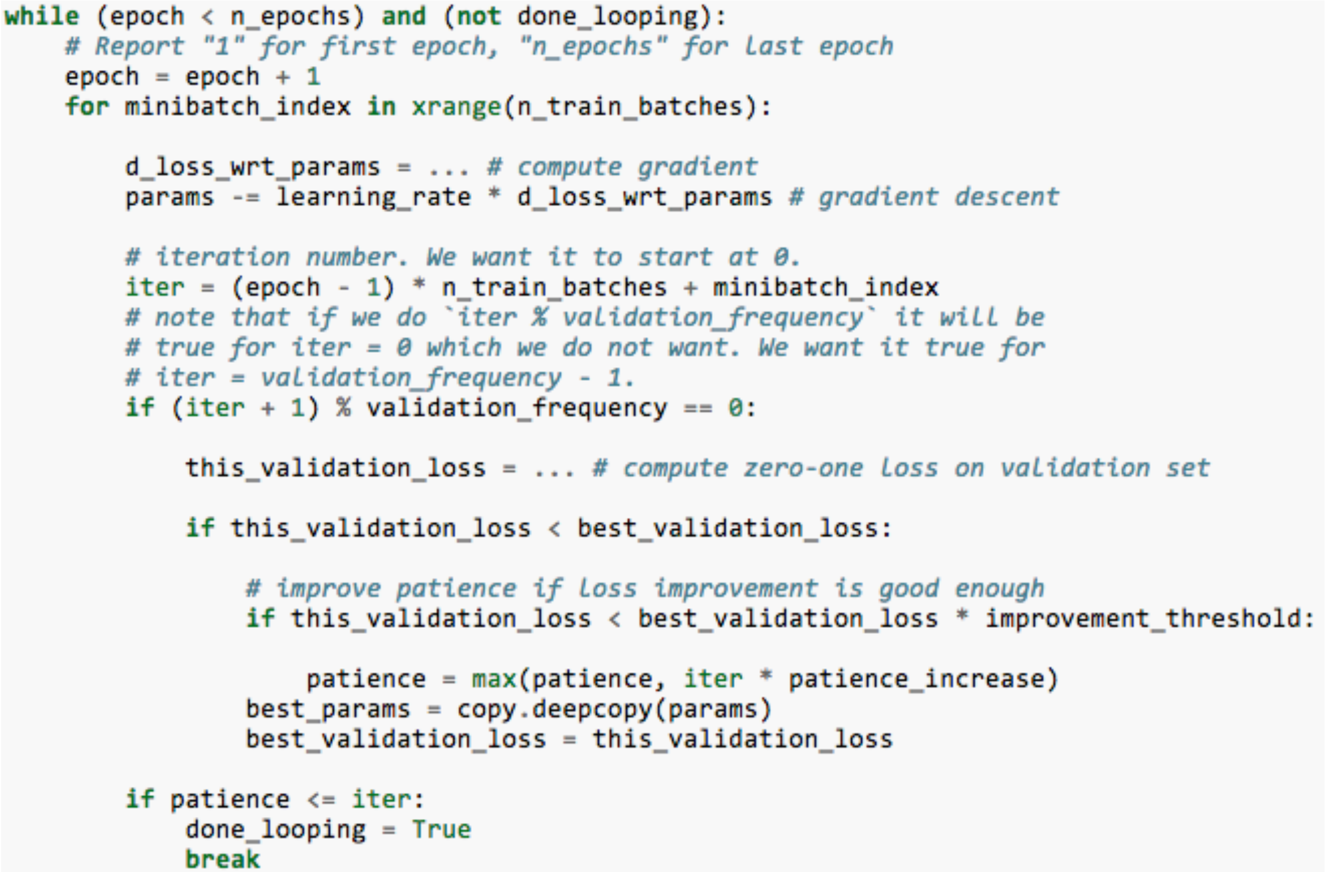
The Jacobian matrix $J_G(\mathbf{x}^{(0)})$

$$J_G = \begin{bmatrix} 3 & \sin(x_2x_3)x_3 & \sin(x_2x_3)x_2 \\ 8x_1 & -1250x_2 + 2 & 0 \\ -x_2\exp(-x_1x_2) & -x_1\exp(-x_1x_2) & 20 \end{bmatrix}$$

$$J_G\left(\mathbf{x}^{(0)}\right) = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 20 \end{bmatrix} \quad G(\mathbf{x}^{(0)}) = \begin{bmatrix} -2.5 \\ -1 \\ 10.472 \end{bmatrix} \qquad \mathbf{x}^{(1)} = 0 - \gamma_0 \begin{bmatrix} -7.5 \\ -2 \\ 209.44 \end{bmatrix}$$

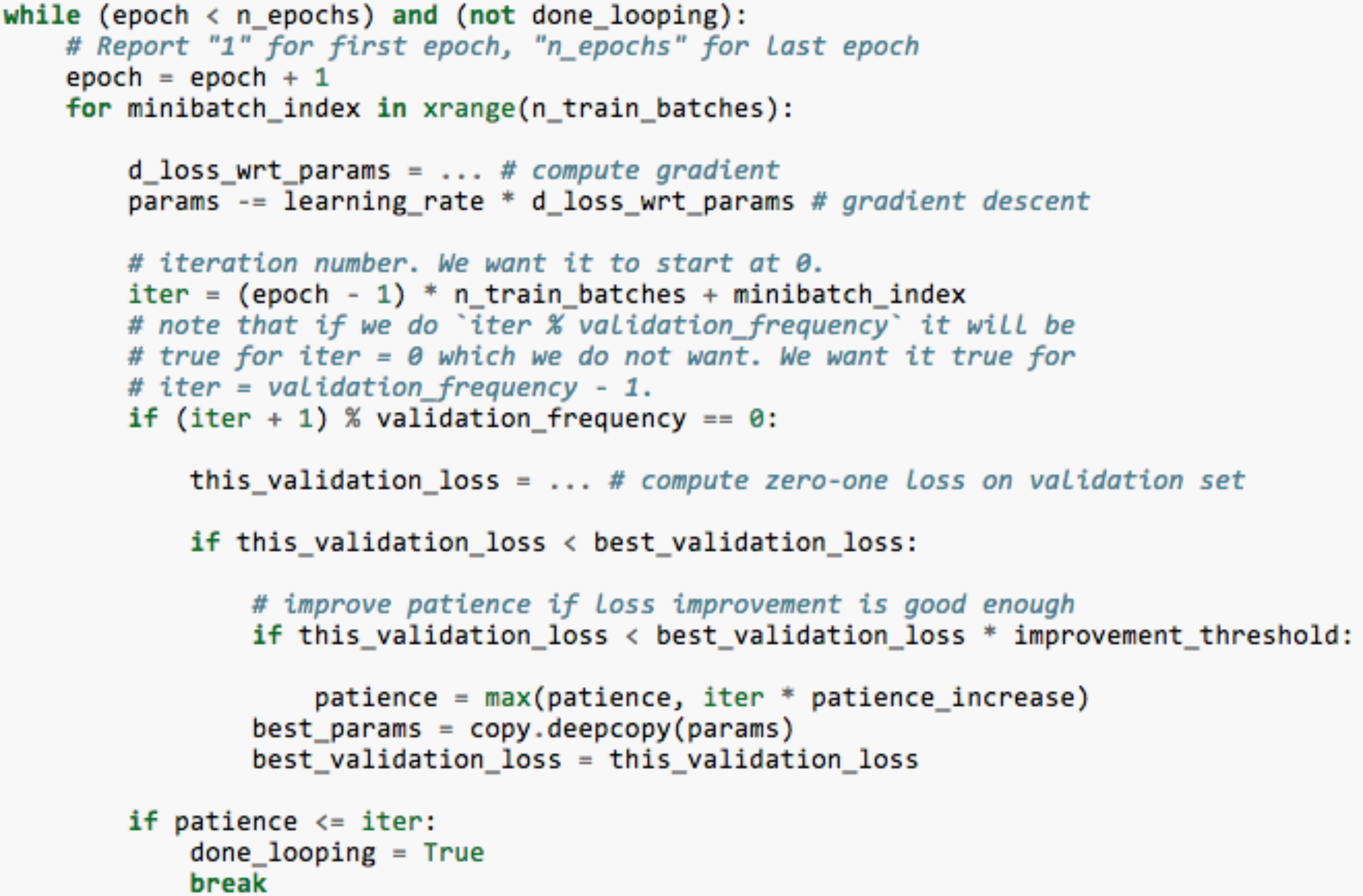
Prerequisite: Final Early Stop

Early-stopping combats overfitting by monitoring the model's performance on a validation set. A validation set is a set of examples that we never use for gradient descent, but which is also not a part of the test set. If the model's performance ceases to improve sufficiently on the validation set, or even degrades with further optimization, then the heuristic implemented here gives up on much further optimization.



```
while (epoch < n_epochs) and (not done_looping):
    # Report "1" for first epoch, "n_epochs" for last epoch
    epoch = epoch + 1
    for minibatch_index in xrange(n_train_batches):

        d_loss_wrt_params = ... # compute gradient
        params -= learning_rate * d_loss_wrt_params # gradient descent

        # iteration number. We want it to start at 0.
        iter = (epoch - 1) * n_train_batches + minibatch_index
        # note that if we do `iter % validation_frequency` it will be
        # true for iter = 0 which we do not want. We want it true for
        # iter = validation_frequency - 1.
        if (iter + 1) % validation_frequency == 0:

            this_validation_loss = ... # compute zero-one loss on validation set

            if this_validation_loss < best_validation_loss:

                # improve patience if loss improvement is good enough
                if this_validation_loss < best_validation_loss * improvement_threshold:

                    patience = max(patience, iter * patience_increase)
                best_params = copy.deepcopy(params)
                best_validation_loss = this_validation_loss

        if patience <= iter:
            done_looping = True
            break
```

Stochastic Gradient Descent: all together



```
while (epoch < n_epochs) and (not done_looping):
    # Report "1" for first epoch, "n_epochs" for last epoch
    epoch = epoch + 1
    for minibatch_index in xrange(n_train_batches):

        d_loss_wrt_params = ... # compute gradient
        params -= learning_rate * d_loss_wrt_params # gradient descent

        # iteration number. We want it to start at 0.
        iter = (epoch - 1) * n_train_batches + minibatch_index
        # note that if we do `iter % validation_frequency` it will be
        # true for iter = 0 which we do not want. We want it true for
        # iter = validation_frequency - 1.
        if (iter + 1) % validation_frequency == 0:

            this_validation_loss = ... # compute zero-one loss on validation set

            if this_validation_loss < best_validation_loss:

                # improve patience if loss improvement is good enough
                if this_validation_loss < best_validation_loss * improvement_threshold:

                    patience = max(patience, iter * patience_increase)
                best_params = copy.deepcopy(params)
                best_validation_loss = this_validation_loss

        if patience <= iter:
            done_looping = True
            break
```

http://deeplearning.net/tutorial/logreg.html

Extension: Softmax regression - (ufldl)

In the softmax regression setting, we are interested in multi-class classification (as opposed to only binary classification), and so the label y can take on k different values, rather than only two. Thus, in our training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we now have that $y^{(i)} \in \{1, 2, \ldots, k\}$. (Note that our convention will be to index the classes starting from 1, rather than from 0.) For example, in the MNIST digit recognition task, we would have $k = 10$ different classes.

Given a test input x, we want our hypothesis to estimate the probability that $p(y = j \mid x)$ for each value of $j = 1, \ldots, k$. I.e., we want to estimate the probability of the class label taking on each of the k different possible values. Thus, our hypothesis will output a k dimensional vector (whose elements sum to 1) giving us our k estimated probabilities. Concretely, our hypothesis $h_\theta(x)$ takes the form:

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1|x^{(i)}; \theta) \\ p(y^{(i)} = 2|x^{(i)}; \theta) \\ \vdots \\ p(y^{(i)} = k|x^{(i)}; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

$$\theta = \begin{bmatrix} -\theta_1^T- \\ -\theta_2^T- \\ \vdots \\ -\theta_k^T- \end{bmatrix} k \times (n+1)$$

Here $\theta_1, \theta_2, \ldots, \theta_k \in \Re^{n+1}$ are the parameters of our model. Notice that the term $\frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}}$ normalizes the distribution, so that it sums to one.

$$p(y^{(i)} = j|x^{(i)}; \theta) = \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}.$$

$$y_{pred}^{i} = \arg\max_j p(y^{(i)} = j \middle| x^{(i)}; \theta)$$

Loss Function

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} \sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}} \right]$$

http://ufldl.stanford.edu/wiki/index.php/Softmax_Regression

http://ufldl.stanford.edu/wiki/index.php/Softmax%E5%9B%9E%E5%BD%92

Extension: Softmax regression - (ufldl)

Loss Function

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\}\log\frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right] \qquad p(y^{(i)} = j|x^{(i)};\theta) = \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}.$$

$$\frac{\partial p(y^{(i)} = j)}{\partial \theta_j} = \frac{e^{\theta_j^T x^{(i)}} x^{(i)T} \sum - e^{\theta_j^T x^{(i)}} e^{\theta_j^T x^{(i)}} x^{(i)T}}{\sum^2} = x^{(i)T} p(y^{(i)} = j)(1 - p(y^{(i)} = j))$$

$$\frac{\partial p(y^{(i)} = j')_{j' \neq j}}{\partial \theta_j} = -\frac{e^{\theta_{j'}^T x^{(i)}} x^T e^{\theta_j^T x^{(i)}}}{\sum^2} = -x^{(i)T} p(y^{(i)} = j) p(y^{(i)} = j')$$

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}(\sum_{j'=1, j' \neq j}^{k} 1\{y^{(i)} = j'\}\log p(y^{(i)} = j') + 1\{y^{(i)} = j\}\log p(y^{(i)} = j))],$$

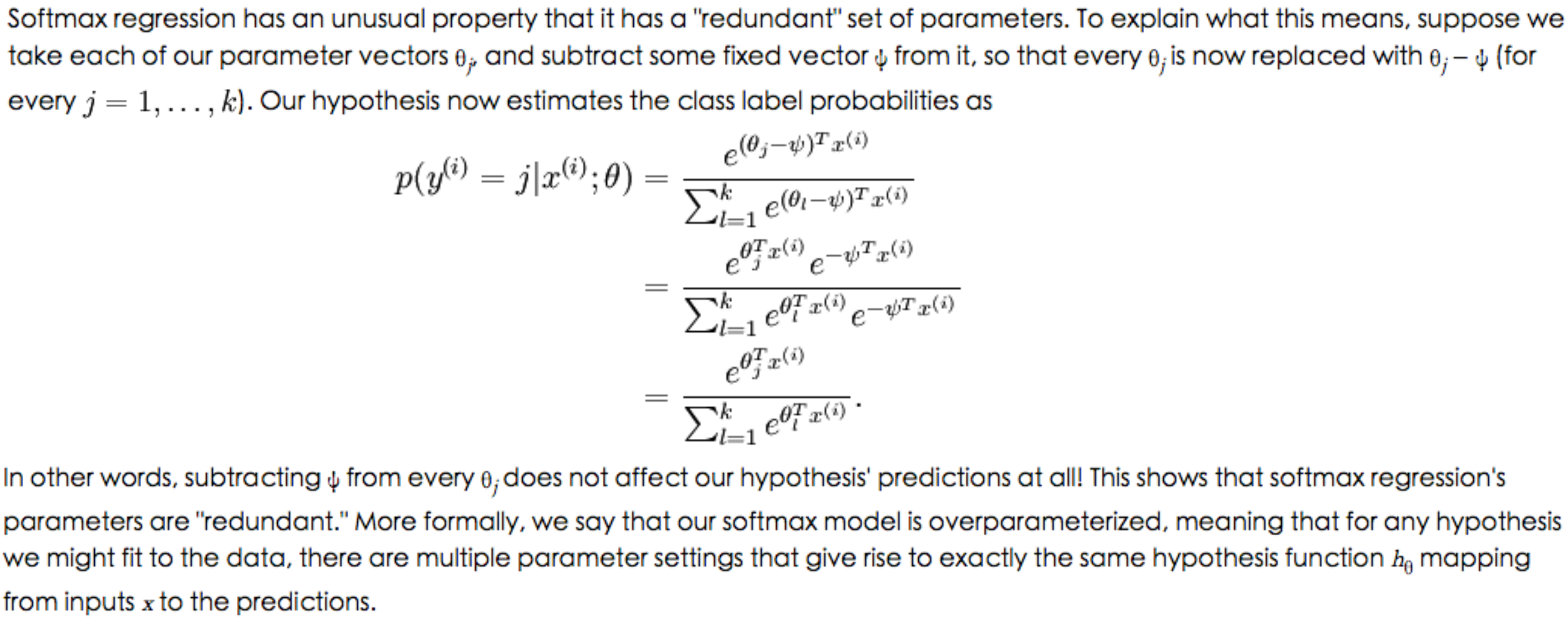
$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}(\sum_{j'=1, j' \neq j}^{k} 1\{y^{(i)} = j'\}\frac{1}{p(y^{(i)} = j')}(-1)x^{(i)T} p(y^{(i)} = j') p(y^{(i)} = j)$$

$$+ 1\{y^{(i)} = j\}\frac{1}{p(y^{(i)} = j)} x^{(i)T} p(y^{(i)} = j)(1 - p(y^{(i)} = j)))]$$

$$= -\frac{1}{m}\sum_{i=1}^{m} x^{(i)T}[1\{y^{(i)} = j\} - p(y^{(i)} = j \,|\, x^{(i)};\theta)]$$

$$\boxed{\text{update } \theta_j := \theta_j - \alpha \nabla_{\theta_j} J(\theta)}$$

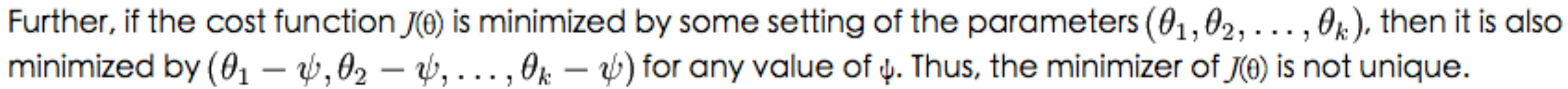
Properties of parameterization



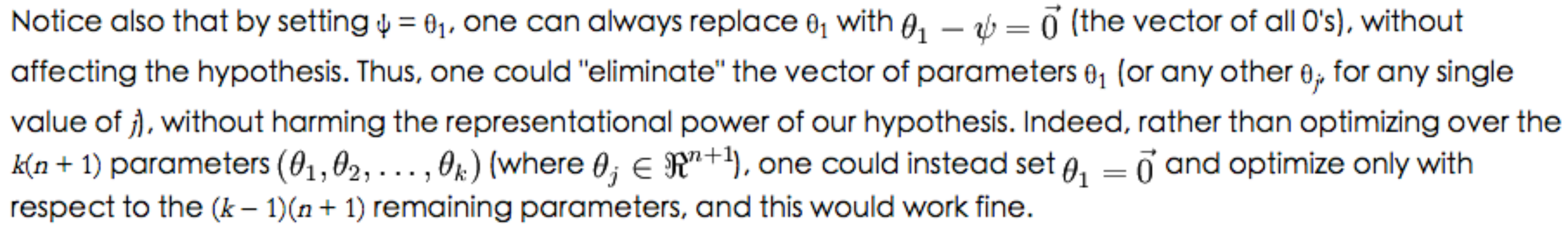
Softmax regression has an unusual property that it has a "redundant" set of parameters. To explain what this means, suppose we take each of our parameter vectors θ_j, and subtract some fixed vector ψ from it, so that every θ_j is now replaced with $\theta_j - \psi$ (for every $j = 1, \ldots, k$). Our hypothesis now estimates the class label probabilities as

$$\begin{aligned} p(y^{(i)} = j | x^{(i)}; \theta) &= \frac{e^{(\theta_j - \psi)^T x^{(i)}}}{\sum_{l=1}^{k} e^{(\theta_l - \psi)^T x^{(i)}}} \\ &= \frac{e^{\theta_j^T x^{(i)}} e^{-\psi^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}} e^{-\psi^T x^{(i)}}} \\ &= \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}. \end{aligned}$$

In other words, subtracting ψ from every θ_j does not affect our hypothesis' predictions at all! This shows that softmax regression's parameters are "redundant." More formally, we say that our softmax model is overparameterized, meaning that for any hypothesis we might fit to the data, there are multiple parameter settings that give rise to exactly the same hypothesis function h_θ mapping from inputs x to the predictions.



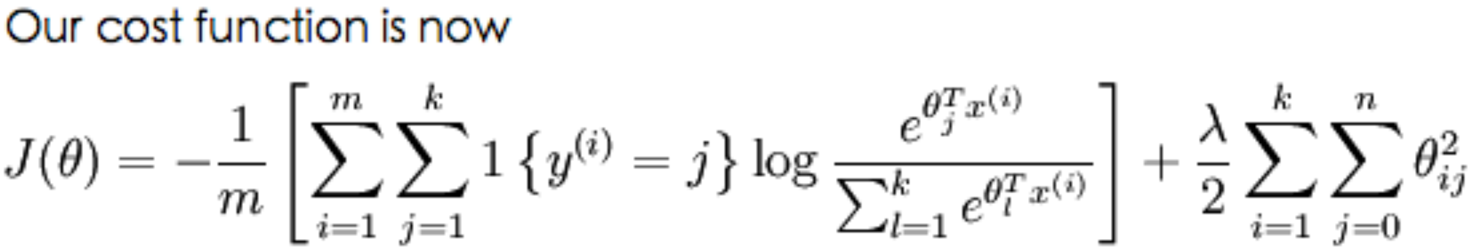
Further, if the cost function $J(\theta)$ is minimized by some setting of the parameters $(\theta_1, \theta_2, \ldots, \theta_k)$, then it is also minimized by $(\theta_1 - \psi, \theta_2 - \psi, \ldots, \theta_k - \psi)$ for any value of ψ. Thus, the minimizer of $J(\theta)$ is not unique.



Notice also that by setting $\psi = \theta_1$, one can always replace θ_1 with $\theta_1 - \psi = \vec{0}$ (the vector of all 0's), without affecting the hypothesis. Thus, one could "eliminate" the vector of parameters θ_1 (or any other θ_j, for any single value of j), without harming the representational power of our hypothesis. Indeed, rather than optimizing over the $k(n+1)$ parameters $(\theta_1, \theta_2, \ldots, \theta_k)$ (where $\theta_j \in \Re^{n+1}$), one could instead set $\theta_1 = \vec{0}$ and optimize only with respect to the $(k-1)(n+1)$ remaining parameters, and this would work fine.

Weight Decay/regularization/Norm

Make the solution unique and combat overfitting



Our cost function is now

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right] + \frac{\lambda}{2}\sum_{i=1}^{k}\sum_{j=0}^{n}\theta_{ij}^2$$

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} x^{(i)T}[1\{y^{(i)} = j\} - p(y^{(i)} = j \mid x^{(i)};\theta)] + \lambda\theta_j^T$$

Relationship to Logistic Regression

In the special case where $k = 2$, one can show that softmax regression reduces to logistic regression. This shows that softmax regression is a generalization of logistic regression. Concretely, when $k = 2$, the softmax regression hypothesis outputs

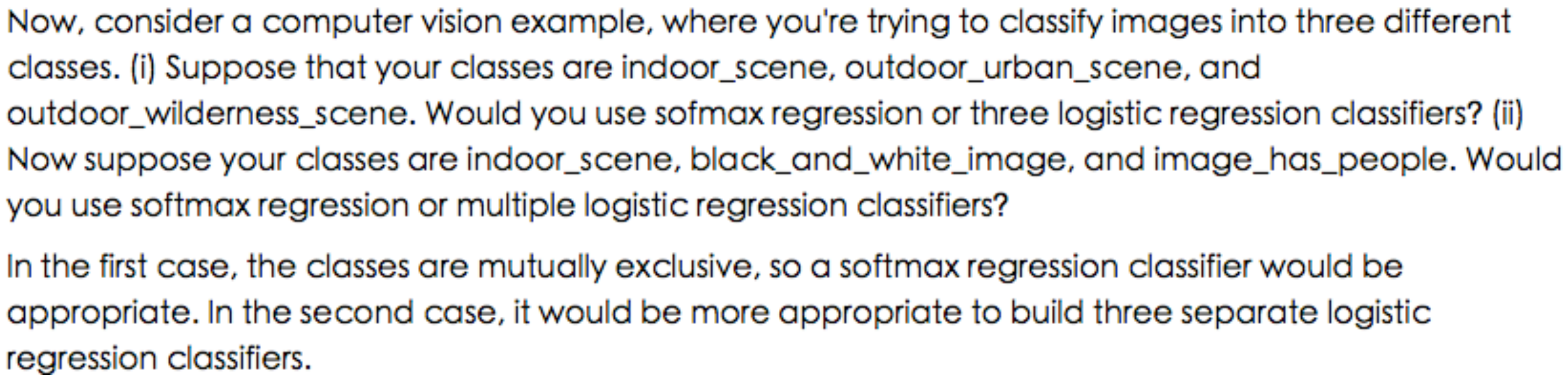
$$h_\theta(x) = \frac{1}{e^{\theta_1^T x} + e^{\theta_2^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x} \\ e^{\theta_2^T x} \end{bmatrix}$$

Taking advantage of the fact that this hypothesis is overparameterized and setting $\psi = \theta_1$, we can subtract θ_1 from each of the two parameters, giving us

$$\begin{aligned} h(x) &= \frac{1}{e^{\vec{0}^T x} + e^{(\theta_2-\theta_1)^T x^{(i)}}} \begin{bmatrix} e^{\vec{0}^T x} \\ e^{(\theta_2-\theta_1)^T x} \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \\ \frac{e^{(\theta_2-\theta_1)^T x}}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \\ 1 - \frac{1}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \end{bmatrix} \end{aligned}$$

Thus, replacing $\theta_2 - \theta_1$ with a single parameter vector θ', we find that softmax regression predicts the probability of one of the classes as $\frac{1}{1 + e^{(\theta')^T x^{(i)}}}$, and that of the other class as $1 - \frac{1}{1 + e^{(\theta')^T x^{(i)}}}$, same as logistic regression.

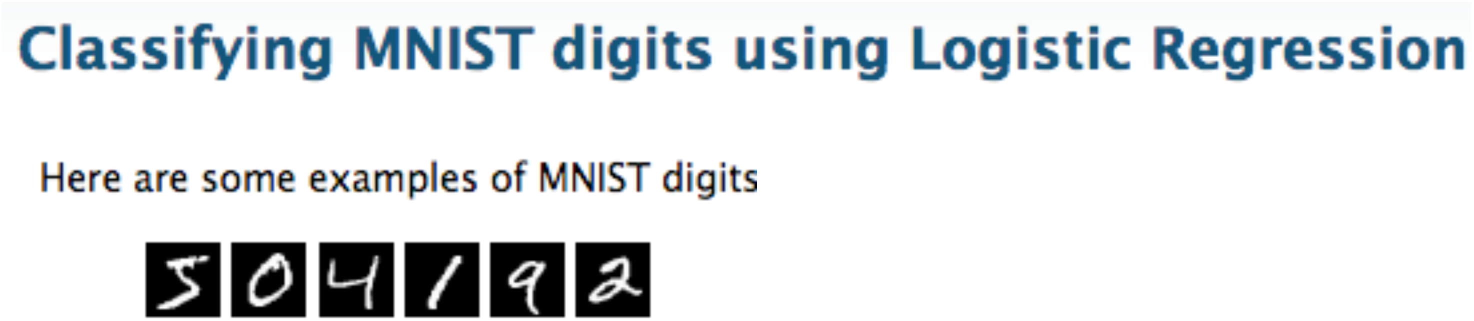
Softmax Regression vs. K Binary Classifier



Now, consider a computer vision example, where you're trying to classify images into three different classes. (i) Suppose that your classes are indoor_scene, outdoor_urban_scene, and outdoor_wilderness_scene. Would you use sofmax regression or three logistic regression classifiers? (ii) Now suppose your classes are indoor_scene, black_and_white_image, and image_has_people. Would you use softmax regression or multiple logistic regression classifiers?

In the first case, the classes are mutually exclusive, so a softmax regression classifier would be appropriate. In the second case, it would be more appropriate to build three separate logistic regression classifiers.

Code Examples



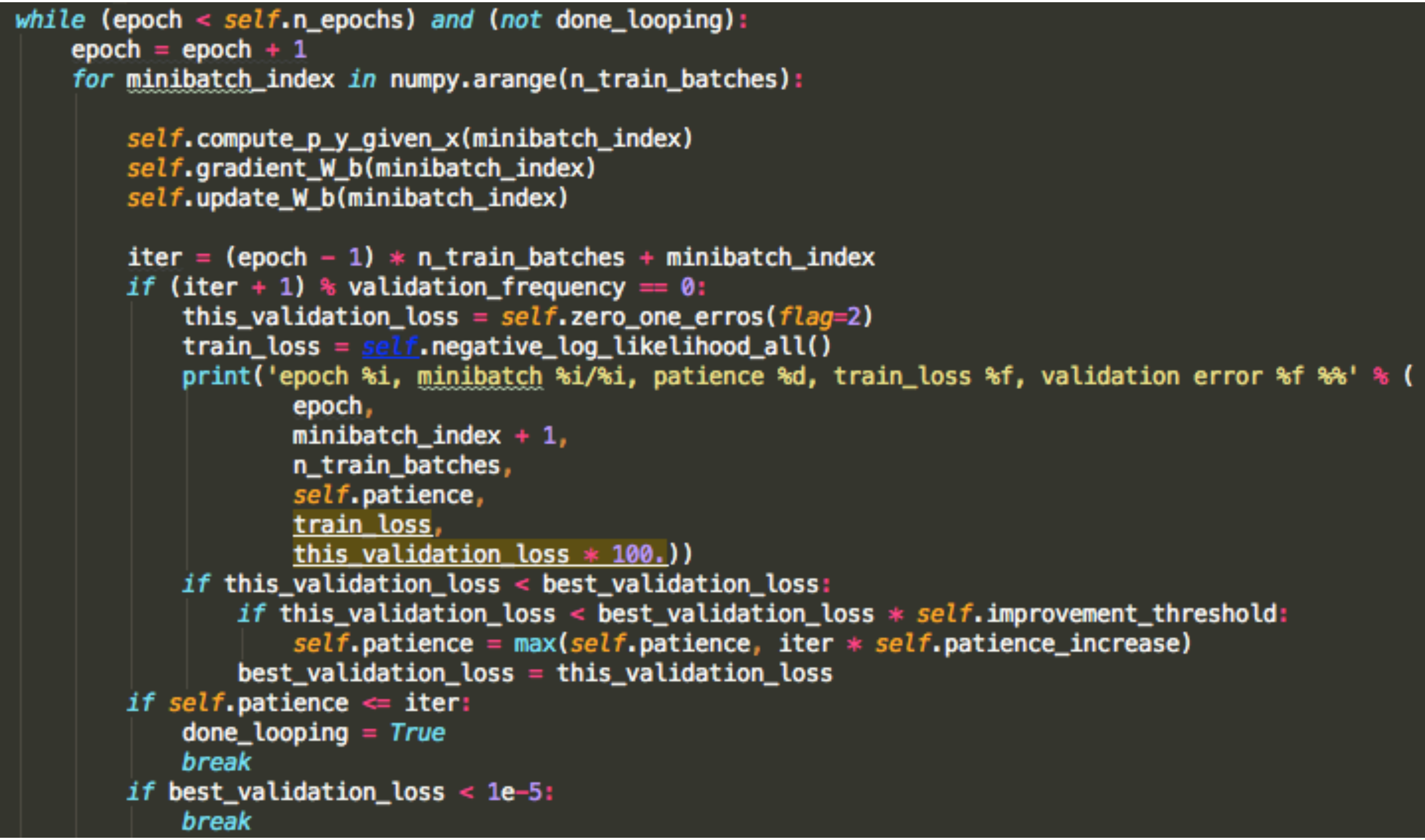
Classifying MNIST digits using Logistic Regression

Here are some examples of MNIST digits

The code is available for download (need theano, but you can change it to be theano-free. Please begin with this.)

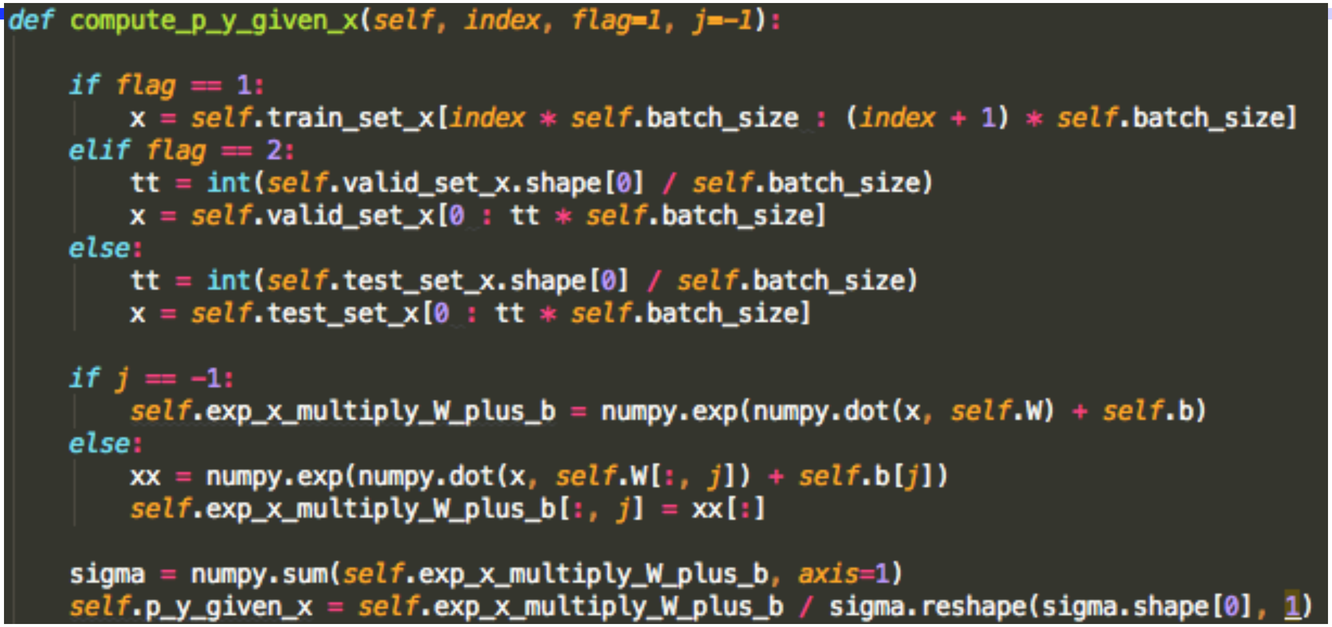
http://deeplearning.net/tutorial/logreg.html

Practice – main body

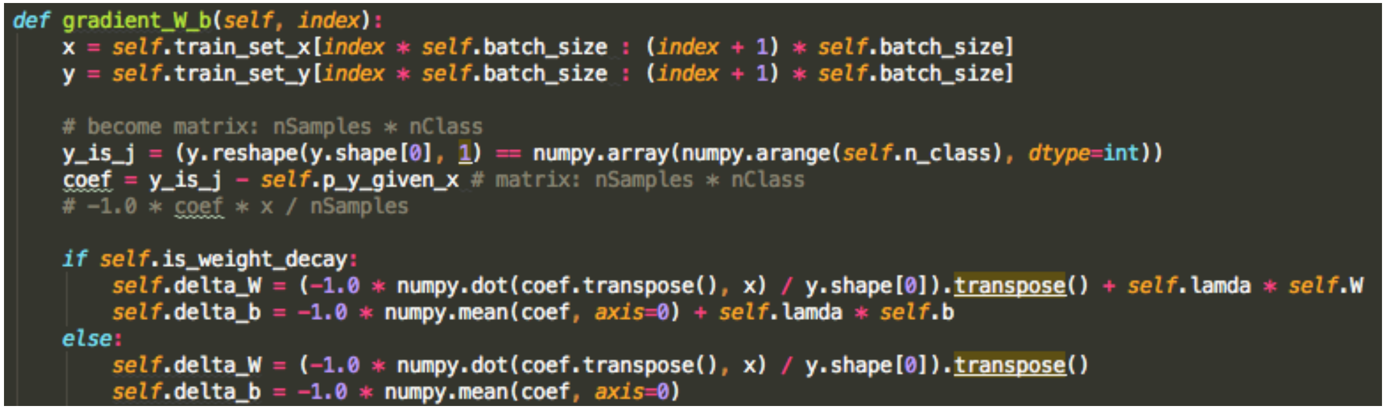


```
while (epoch < self.n_epochs) and (not done_looping):
    epoch = epoch + 1
    for minibatch_index in numpy.arange(n_train_batches):

        self.compute_p_y_given_x(minibatch_index)
        self.gradient_W_b(minibatch_index)
        self.update_W_b(minibatch_index)

        iter = (epoch - 1) * n_train_batches + minibatch_index
        if (iter + 1) % validation_frequency == 0:
            this_validation_loss = self.zero_one_erros(flag=2)
            train_loss = self.negative_log_likelihood_all()
            print('epoch %i, minibatch %i/%i, patience %d, train_loss %f, validation error %f %%' % (
                    epoch,
                    minibatch_index + 1,
                    n_train_batches,
                    self.patience,
                    train_loss,
                    this_validation_loss * 100.))
            if this_validation_loss < best_validation_loss:
                if this_validation_loss < best_validation_loss * self.improvement_threshold:
                    self.patience = max(self.patience, iter * self.patience_increase)
                best_validation_loss = this_validation_loss
        if self.patience <= iter:
            done_looping = True
            break
        if best_validation_loss < 1e-5:
            break
```

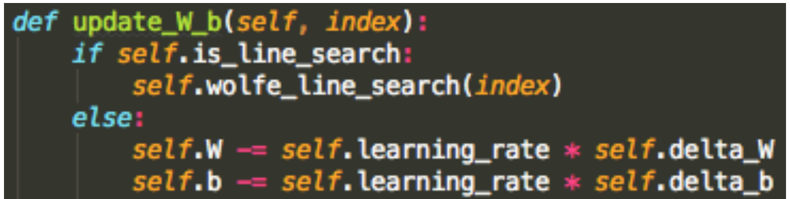
Practice – main functions



```python
def compute_p_y_given_x(self, index, flag=1, j=-1):

    if flag == 1:
        x = self.train_set_x[index * self.batch_size : (index + 1) * self.batch_size]
    elif flag == 2:
        tt = int(self.valid_set_x.shape[0] / self.batch_size)
        x = self.valid_set_x[0 : tt * self.batch_size]
    else:
        tt = int(self.test_set_x.shape[0] / self.batch_size)
        x = self.test_set_x[0 : tt * self.batch_size]

    if j == -1:
        self.exp_x_multiply_W_plus_b = numpy.exp(numpy.dot(x, self.W) + self.b)
    else:
        xx = numpy.exp(numpy.dot(x, self.W[:, j]) + self.b[j])
        self.exp_x_multiply_W_plus_b[:, j] = xx[:]

    sigma = numpy.sum(self.exp_x_multiply_W_plus_b, axis=1)
    self.p_y_given_x = self.exp_x_multiply_W_plus_b / sigma.reshape(sigma.shape[0], 1)
```



```python
def gradient_W_b(self, index):
    x = self.train_set_x[index * self.batch_size : (index + 1) * self.batch_size]
    y = self.train_set_y[index * self.batch_size : (index + 1) * self.batch_size]

    # become matrix: nSamples * nClass
    y_is_j = (y.reshape(y.shape[0], 1) == numpy.array(numpy.arange(self.n_class), dtype=int))
    coef = y_is_j - self.p_y_given_x # matrix: nSamples * nClass
    # -1.0 * coef * x / nSamples

    if self.is_weight_decay:
        self.delta_W = (-1.0 * numpy.dot(coef.transpose(), x) / y.shape[0]).transpose() + self.lamda * self.W
        self.delta_b = -1.0 * numpy.mean(coef, axis=0) + self.lamda * self.b
    else:
        self.delta_W = (-1.0 * numpy.dot(coef.transpose(), x) / y.shape[0]).transpose()
        self.delta_b = -1.0 * numpy.mean(coef, axis=0)
```



```python
def update_W_b(self, index):
    if self.is_line_search:
        self.wolfe_line_search(index)
    else:
        self.W -= self.learning_rate * self.delta_W
        self.b -= self.learning_rate * self.delta_b
```

Basketball Dataset

- □ We have,
  - ✓ a training/validation set: more than 80,000 negatives and 465 positives for 2-frame samples, and more than 80,000 negatives and 586 positives for 1-frame samples. 80 percentage is treated as the training set and 20 percentage is put to validation set.
  - ✓ a testing set: 158 positives and 28451 negatives for 2-frame samples and similar number of samples for the 1-frame case.

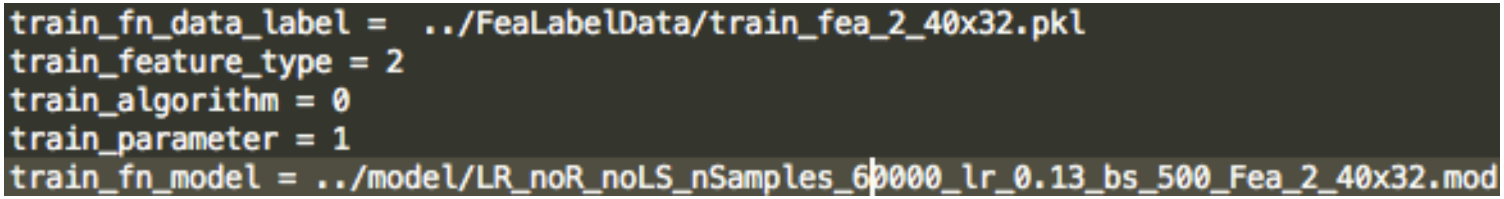
Considerations in Practice

- □ It is unclear what is the behavior of LR in different parameters
  - ✓ what's the difference between the small and large sizes of training set (1000, 2000, 4000, 8000, 16000, 32000, 48000, 60000)
  - ✓ learning rate (0.1 : 0.1 : 0.5)
  - ✓ batch size (100 : 100 : 600)
  - ✓ should we treat a few continuous frames rather than only one frame as a positive sample (1 or 2)
  - ✓ whether or not to use line search (0 / 1)
  - ✓ whether or not to use weight decay (0 / 1)

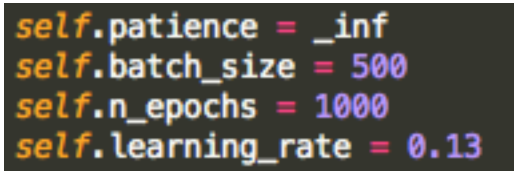
  - ✓ whether or not to use early stop (none)
  - ✓ number of epochs (1000)
  - ✓ are there better annotation methods? (fixed with 40x32 in the experiments)
  - ✓ …

Criterion: comparing ROC curves

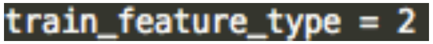
what’s the difference between the small and large sizes of training set (1000, 2000, 4000, 8000, 16000, 32000, 48000, 60000)



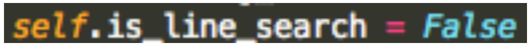
```
train_fn_data_label =  ../FeaLabelData/train_fea_2_40x32.pkl
train_feature_type = 2
train_algorithm = 0
train_parameter = 1
train_fn_model = ../model/LR_noR_noLS_nSamples_60000_lr_0.13_bs_500_Fea_2_40x32.mod
```



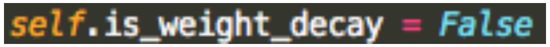
```
self.patience = _inf
self.batch_size = 500
self.n_epochs = 1000
self.learning_rate = 0.13
```



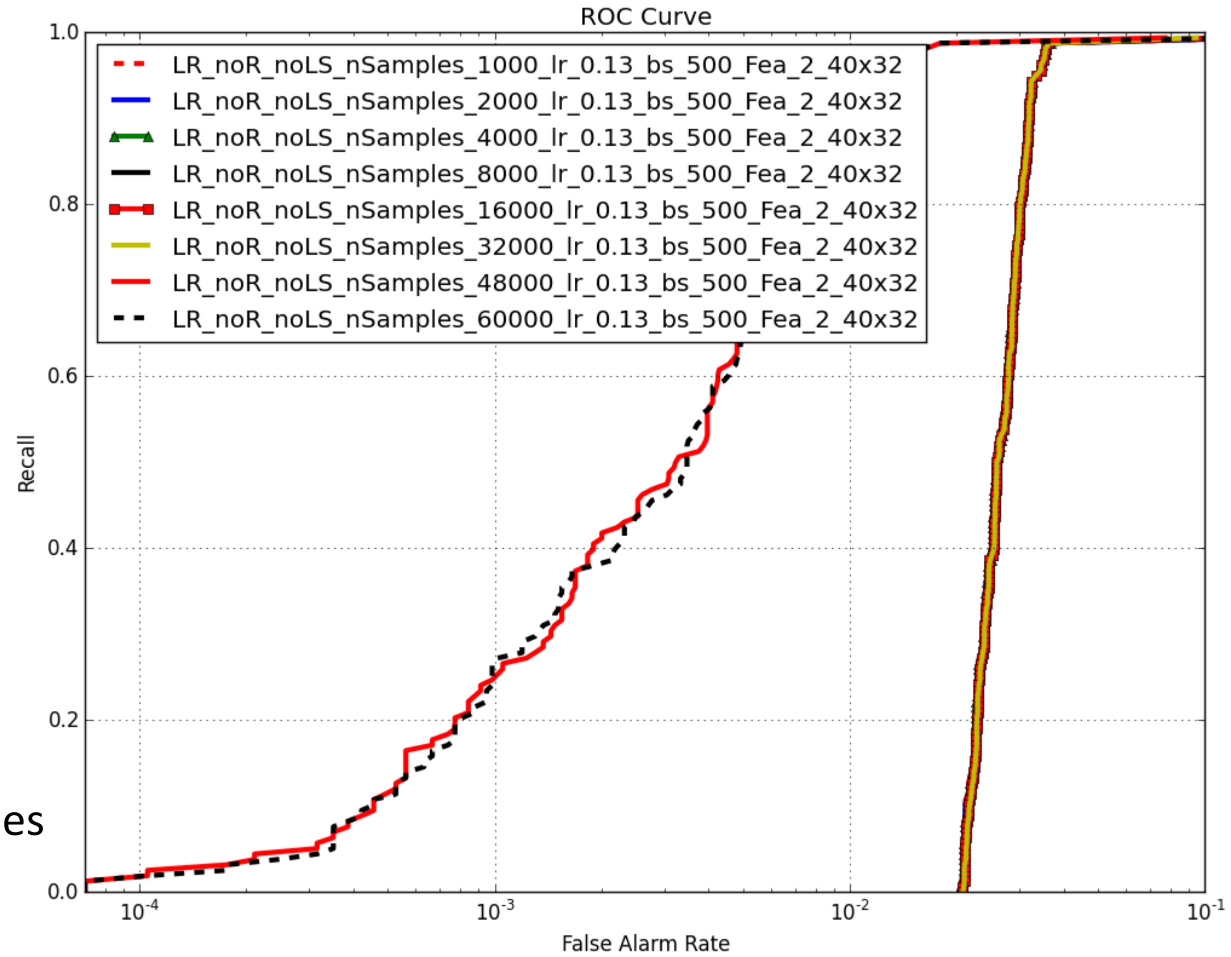
```
train_feature_type = 2
```



```
self.is_line_search = False
```

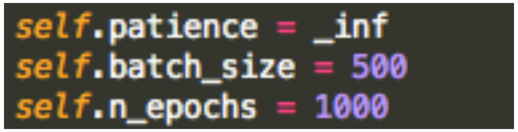


```
self.is_weight_decay = False
```

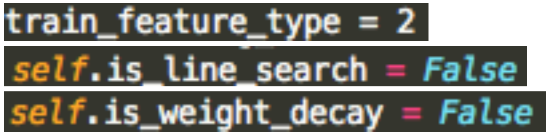


Hereafter, we use 60000 samples for training.

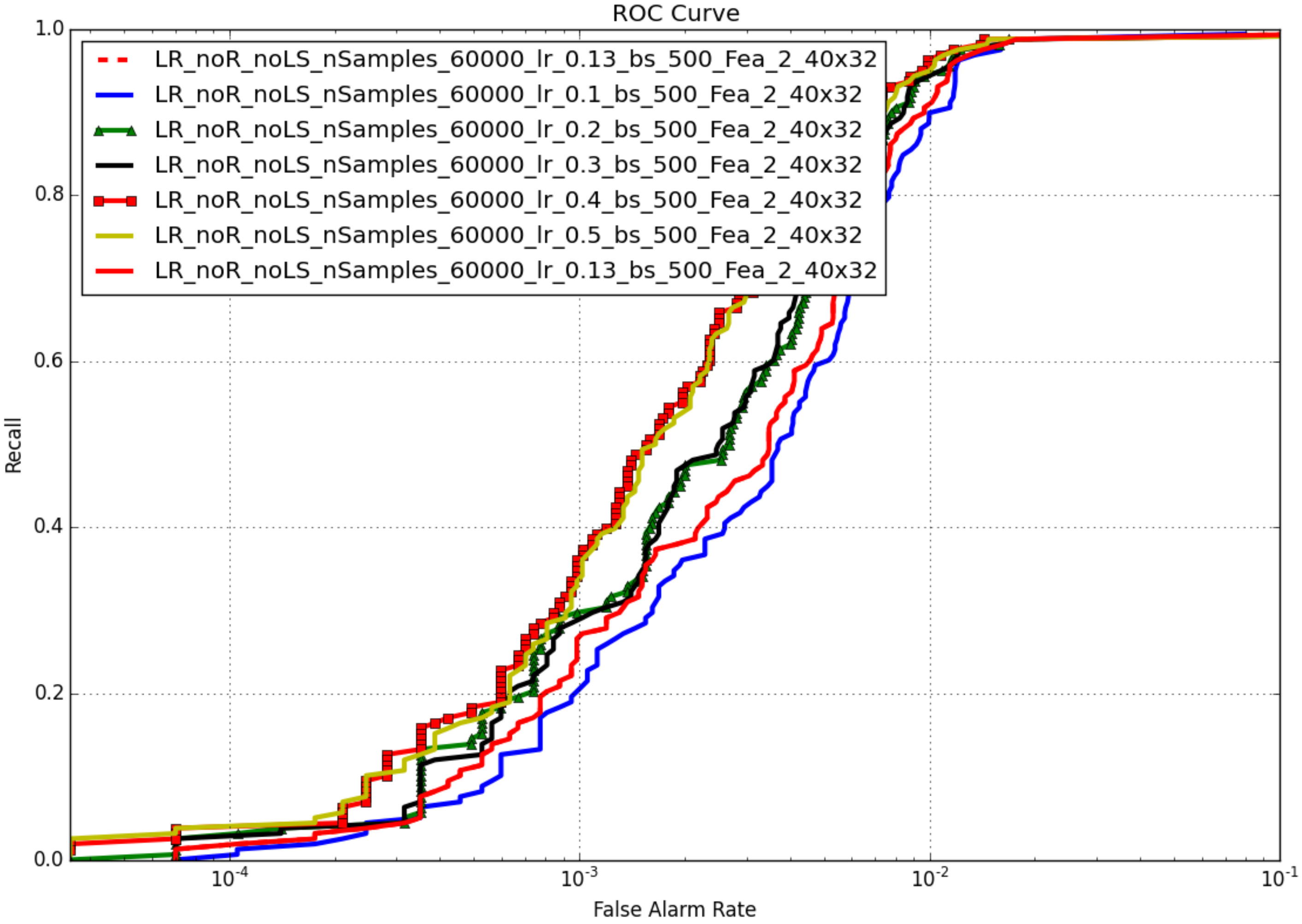
learning rate (0.1 : 0.1 : 0.5)



```
self.patience = _inf
self.batch_size = 500
self.n_epochs = 1000
```

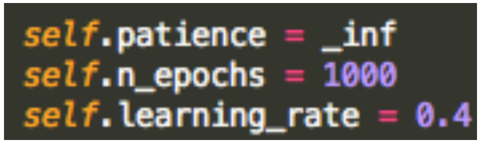


```
train_feature_type = 2
self.is_line_search = False
self.is_weight_decay = False
```

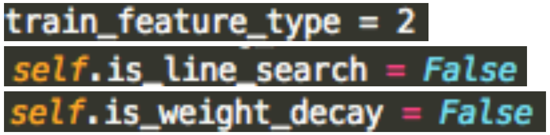


0.4 is the best, then we set learning rate to 0.4 hereafter.

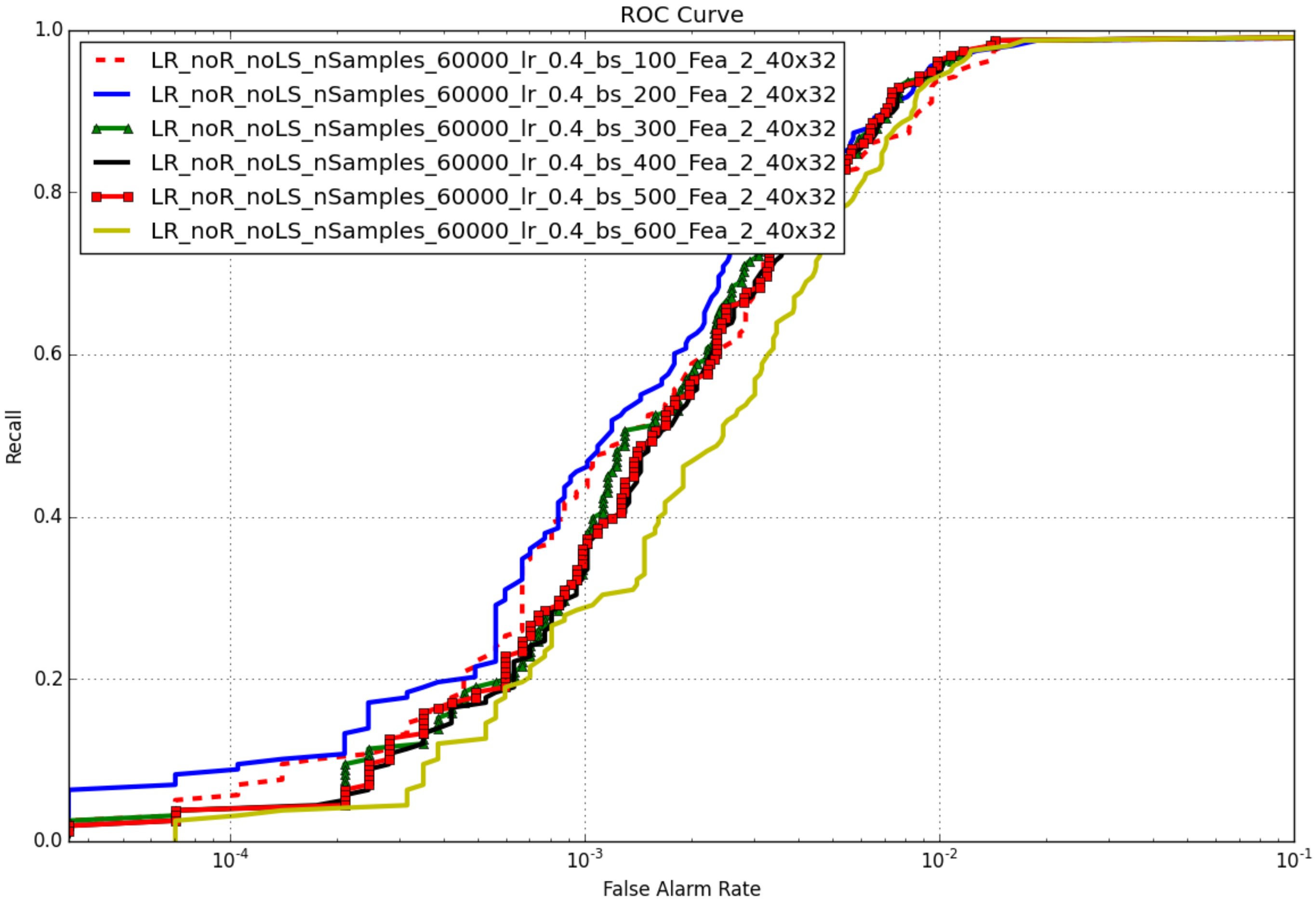
batch size (100 : 100 : 600)



```
self.patience = _inf
self.n_epochs = 1000
self.learning_rate = 0.4
```

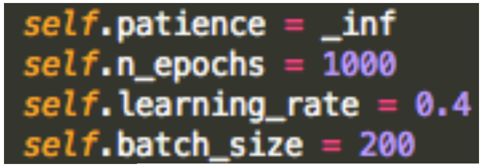


```
train_feature_type = 2
self.is_line_search = False
self.is_weight_decay = False
```

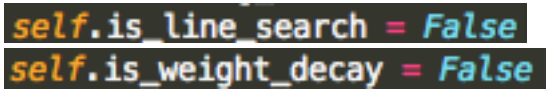


200 is the best, then we set batch size to 200 hereafter.

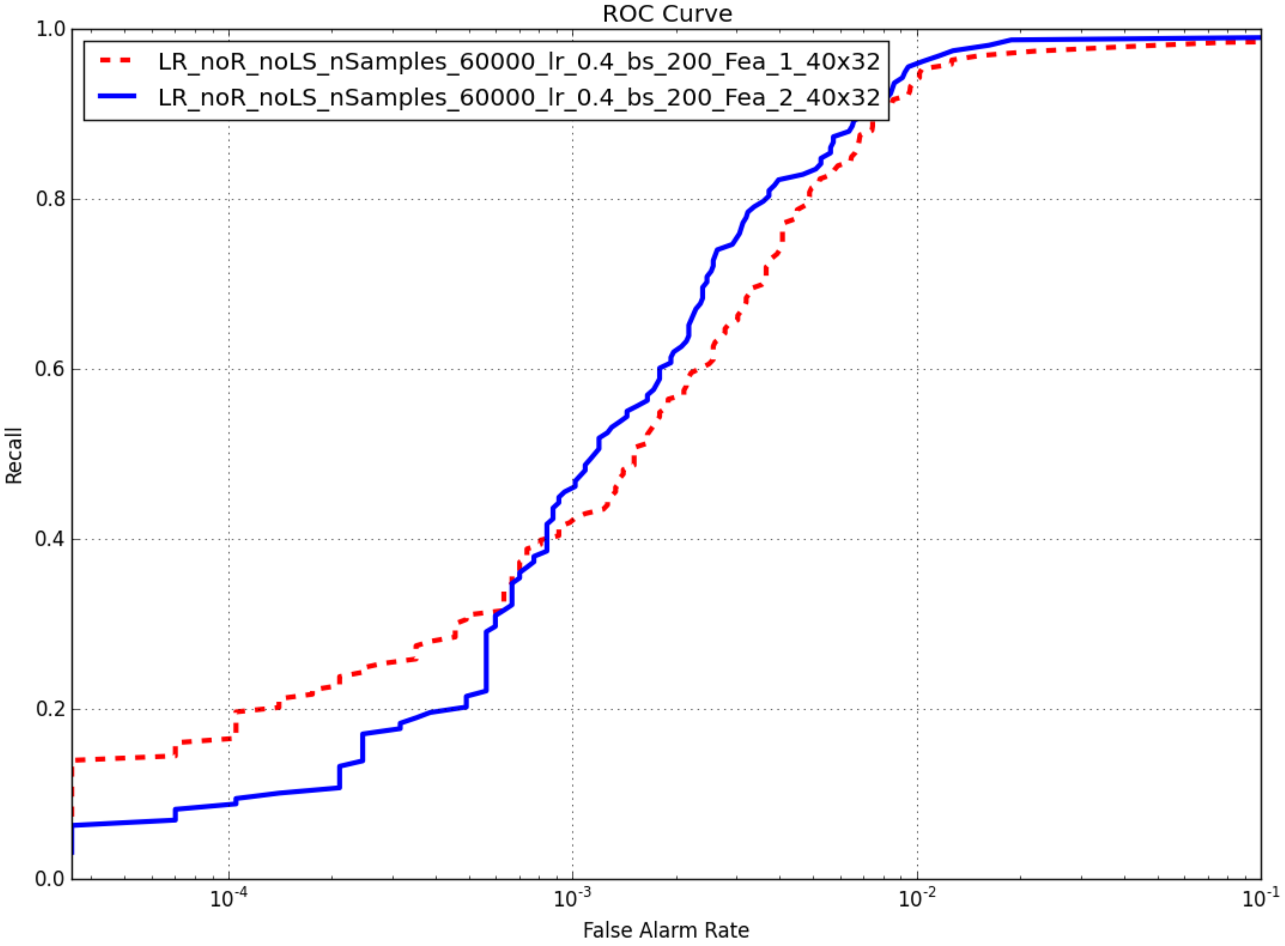
should we treat a few continuous frames rather than only one frame as a positive sample (1 or 2)



```
self.patience = _inf
self.n_epochs = 1000
self.learning_rate = 0.4
self.batch_size = 200
```

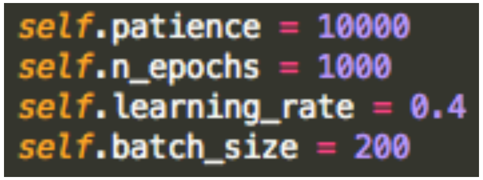


```
self.is_line_search = False
self.is_weight_decay = False
```

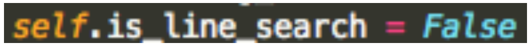


2-frame samples word better. Maybe the more the better.

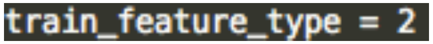
whether or not to use early stop



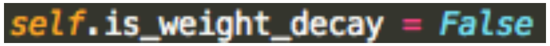
```
self.patience = 10000
self.n_epochs = 1000
self.learning_rate = 0.4
self.batch_size = 200
```



```
self.is_line_search = False
```

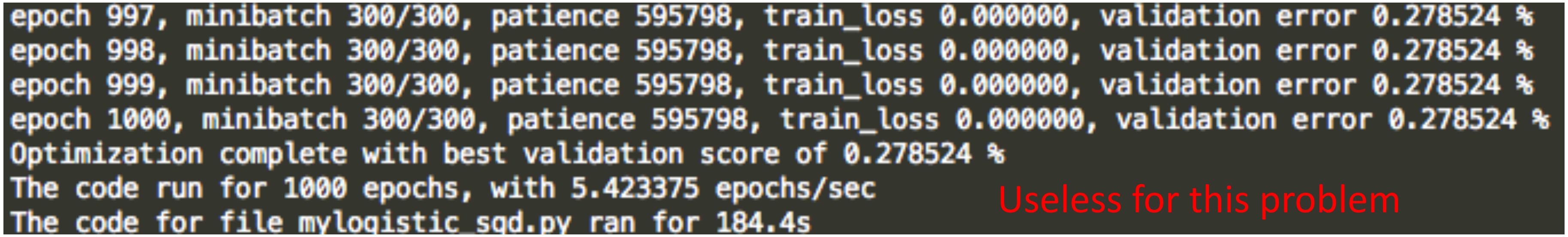


```
train_feature_type = 2
```



```
self.is_weight_decay = False
```

In total, we need 60000/200*1000 = 300,000 times iterations. Suppose our patience is about 150,000 (if smaller, the algorithm doesn’t not converge). The algorithm converges at:



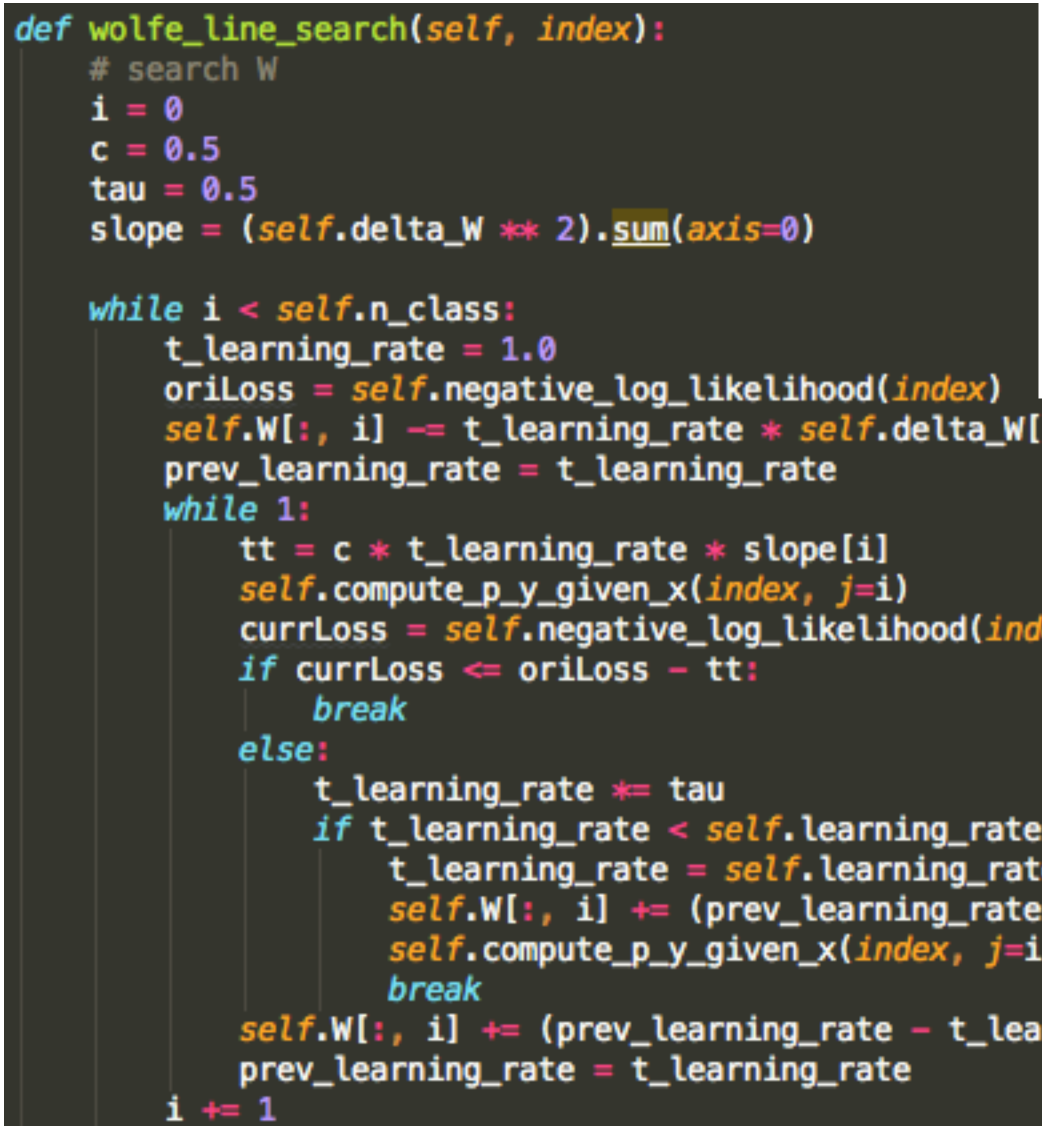
```
epoch 997, minibatch 300/300, patience 595798, train_loss 0.000000, validation error 0.278524 %
epoch 998, minibatch 300/300, patience 595798, train_loss 0.000000, validation error 0.278524 %
epoch 999, minibatch 300/300, patience 595798, train_loss 0.000000, validation error 0.278524 %
epoch 1000, minibatch 300/300, patience 595798, train_loss 0.000000, validation error 0.278524 %
Optimization complete with best validation score of 0.278524 %
The code run for 1000 epochs, with 5.423375 epochs/sec
The code for file mylogistic_sgd.py ran for 184.4s
```

Useless for this problem

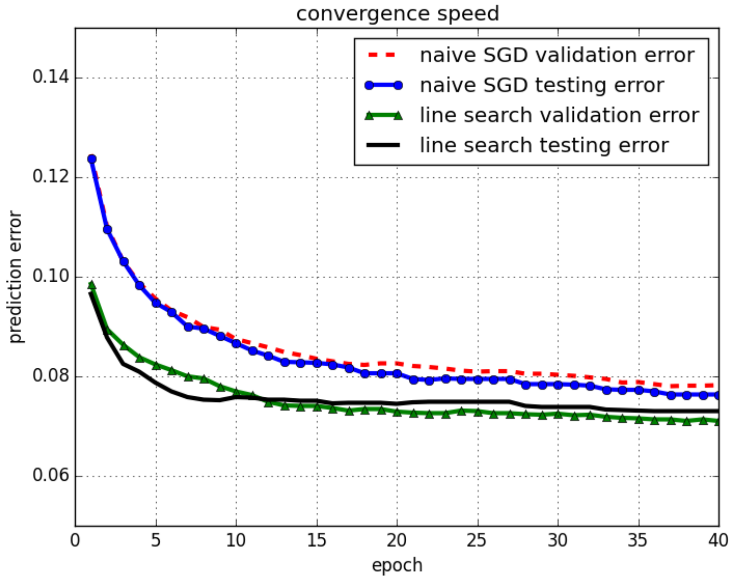
Also, line search doesn’t work well on this problem.

The purpose of regularization is same as early stop. Since we choose the best model on validation set, we almost have no overfitting problem. The problem is that we really can not stop earlier.

More: line search



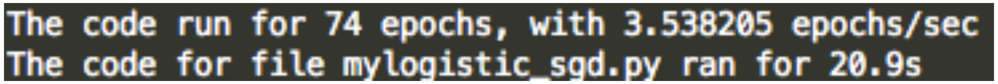
```python
def wolfe_line_search(self, index):
    # search W
    i = 0
    c = 0.5
    tau = 0.5
    slope = (self.delta_W ** 2).sum(axis=0)

    while i < self.n_class:
        t_learning_rate = 1.0
        oriLoss = self.negative_log_likelihood(index)
        self.W[:, i] -= t_learning_rate * self.delta_W[:, i]
        prev_learning_rate = t_learning_rate
        while 1:
            tt = c * t_learning_rate * slope[i]
            self.compute_p_y_given_x(index, j=i)
            currLoss = self.negative_log_likelihood(index)
            if currLoss <= oriLoss - tt:
                break
            else:
                t_learning_rate *= tau
                if t_learning_rate < self.learning_rate:
                    t_learning_rate = self.learning_rate
                    self.W[:, i] += (prev_learning_rate - t_learning_rate) * self.delta_W[:, i]
                    self.compute_p_y_given_x(index, j=i)
                    break
            self.W[:, i] += (prev_learning_rate - t_learning_rate) * self.delta_W[:, i]
            prev_learning_rate = t_learning_rate
        i += 1
```



On the MNIST dataset

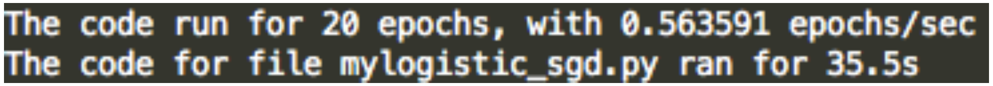
line search - Time cost

naïve update

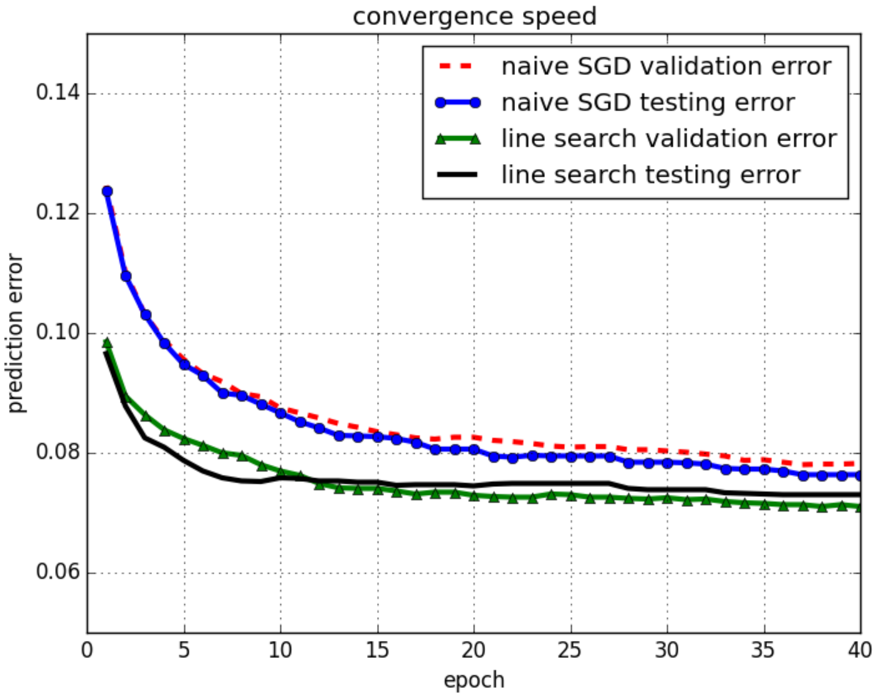


```
The code run for 74 epochs, with 3.538205 epochs/sec
The code for file mylogistic_sgd.py ran for 20.9s
```

line search



```
The code run for 20 epochs, with 0.563591 epochs/sec
The code for file mylogistic_sgd.py ran for 35.5s
```



Reference

Many slides were made based on

http://ufldl.stanford.edu/wiki/index.php/Softmax_Regression

A few examples can be found at:

http://deeplearning.net/tutorial/logreg.html#logreg

The detail of Matrix Calculus can be found at:

T. P. Minka, *Old and New Matrix Algebra Useful for Statistics*, 2000. (Microsoft Research).

For gradient descent and line search, please refer to:

Armijo, Larry (1966). "Minimization of functions having Lipschitz continuous first partial derivatives". Pacific J. Math.

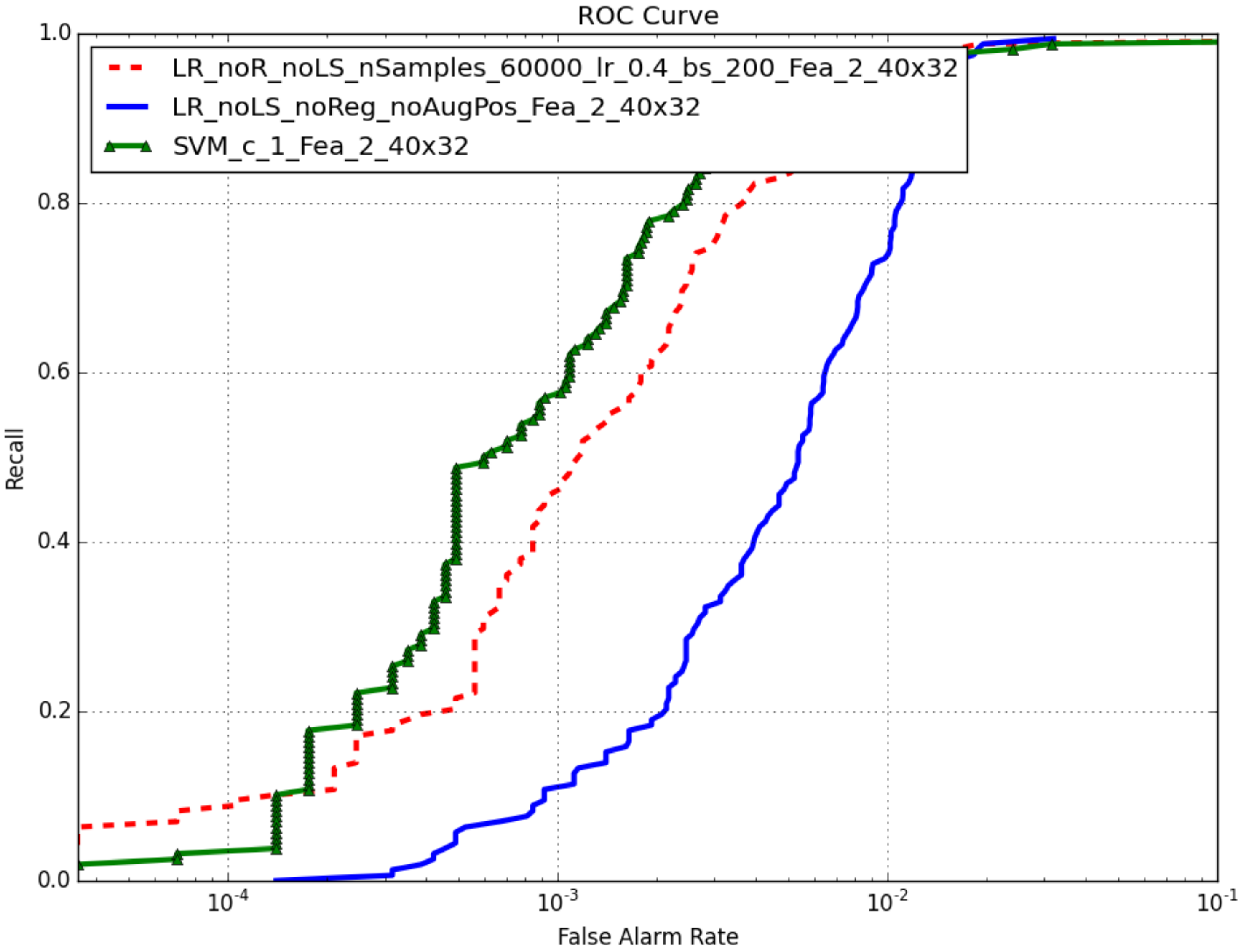
https://en.wikipedia.org/wiki/Backtracking_line_search

Assignments

- Derive every formula by yourself in the slides
- Read the logistic regression code on deep learning tutorial, run it on MNIST dataset, and rewrite / change it to be theano-free.

  http://deeplearning.net/tutorial/logreg.html
- Carry out all experiments I showed on the basketball dataset.

Next topic: support vector machine (SVM)


ROC Curve
LR_noR_noLS_nSamples_60000_lr_0.4_bs_200_Fea_2_40x32
LR_noLS_noReg_noAugPos_Fea_2_40x32
SVM_c_1_Fea_2_40x32
Recall
10^{-4}
10^{-3}
10^{-2}
10^{-1}
False Alarm Rate