Notes about ESL

Jianbo Guo

0

In God we trust; all others bring data.

1 Some Useful Equations

α is a scalar, x, y, z are vectors and A is a matrix. Then we have

$$\begin{cases} y = Ax \rightarrow \frac{\partial y}{\partial x} = A \\ \alpha = y^\intercal Ax \rightarrow \frac{\partial \alpha}{\partial x} = y^\intercal A, \quad \frac{\partial \alpha}{\partial y} = x^\intercal A^\intercal \\ \alpha = y^\intercal x \rightarrow \frac{\partial \alpha}{\partial z} = y^\intercal \frac{\partial x}{\partial z} + x^\intercal \frac{\partial y}{\partial z} \\ (x \; and \; y \; are \; functions \; of \; z) \end{cases} \tag{1}$$

where the definition of $\left(\frac{\partial y}{\partial x}\right)_{ij} = \frac{y_i}{x_j}$. Please note $\frac{\partial \alpha}{\partial x}$ is a $1 \times n$ row vector and normally $\nabla_x \alpha(x)$ is a $n \times 1$ colomn vector! For more details about matrix calculus, see `http://www.atmos.washington.edu/~dennis/MatrixCalculus.pdf`.

Given a random variable X with mean μ and standard deviation σ, then we have Markov's inequality and Chebyve inequality:

$$\begin{cases} P(X \geq a) \leq \frac{\mu}{a} \\ P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \end{cases} \tag{2}$$

Also, there are 2 vital bounds, i.e. union bound:

$$p(A_1 \cup A_2 \cup ... \cup A_k) \leq p(A_1) + p(A_2) + ... + p(A_k) \tag{3}$$

where $A_1, ..., A_k$ are k different events and Hoeffding's inequaility:

$$\begin{cases} P(S_m - E[S_m] \geq \epsilon) \leq \exp(-2\epsilon^2 / \sum_{i=1}^m (b_i - a_i)^2) \\ P(S_m - E[S_m] \leq -\epsilon) \leq \exp(-2\epsilon^2 / \sum_{i=1}^m (b_i - a_i)^2) \end{cases} \tag{4}$$

where $S_m = \sum_{i=1}^m X_i$ and $X_i \in [a_i, b_i]$ for $i \in [m]$ and all X_i are independent variables. Combining this and union bound we get a specific form of Chernoff bound:

$$p(|\phi - \hat{\phi}| > \gamma) \leq 2\exp(-2\gamma^2 m) \tag{5}$$

where $\hat{\phi} = \frac{1}{m} \sum_{i=1}^m Z_i$ and $Z_i \sim$Bernoulli(ϕ), i.e. $p(Z_i = 1) = \phi$.

2 Teminology

2.1 End-to-end V.S. step-by-step

End-to-end learning tunes the parameters of the entire model based on the correctional signal from the output. No supervision added to the intermediate layers. In Step-by-step learning, we have specifically designed supervision on the intermediate representations

3 Preprocssing

3.1 Zero-mean and normalization

$$\begin{cases} x = x - \mu \\ x = x/\sigma \end{cases} \tag{6}$$

Common pitfall. An important point to make about the preprocessing is that any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation / test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake. Instead, the mean must be computed only over the training data and then subtracted equally from all splits (train/val/test).

The authors correctly asserts that using the whole 100 image sample to compute the sample mean μ is wrong. That is because in this case you would have information leakage. Information from your "out-of-sample" elements would be move to your training set. In particular for the estimation of μ , if you use 100 instead of 90 images you allow your training set to have a more informed mean than it should have too. As a result your training error would be potentially lower than it should be.

The estimated μ is common throughout the training/validation/testing procedure. The same μ is to be use to centre all your data. That is:

$$\begin{cases} \mu = np.mean(X_{tr}, axis = 0) \\ X_{tr} -= \mu \\ X_{val} -= \mu \\ X_{te} -= \mu \end{cases} \tag{7}$$

In the case of images, we only do zero-mean (zero-centered). There is no need to do normalization because all the color channels have the same scale (0-255)

3.2 Principal Component Analysis (PCA)

This method can decorrelate the data and select some major features (along this axis, the variance of data is big).

$$\begin{cases} X- = np.mean(X, axis = 0) \\ cov = X.T.dot(X)/X.shape[0] \\ U, S, V = np.linalg.svd(cov) \\ Xrot = X.dot(U) \\ Xrot_{reduced} = X.dot(U[:, : D']) \end{cases} \tag{8}$$

where D' is the new demensions that you want and the columns of U are the eigenvectors and S is a 1-D array of the singular values (which are equal to the eigenvalues squared since *cov* is symmetric and positive semi-definite).

3.3 Whitening

This method make the whitened data be a gaussian with zero mean and identity covariance matrix.

$$Xwhite = Xrot/np.sqrt(S + 1e - 5) \tag{9}$$

4 Hyperparameters tuning

More details see /Users/Jianbo/Desktop/THU/Machine_Learning_Reading/Feifei_Li/winter1516_lecture5.pdf P71-P90. There are 3 setps:

- Start with small regulraization to make sure some learning rate could let loss go down exponentially (you may visualize it) Also, you need check the loss converges to obtain num_iters
- We do a coarse hyperparameter tuning with a few iterations, as shown in Fig.1
- Run a finer search with enough iterations, as shown in Fig.2

5 Models

5.1 Multiclass SVM (do subgradient directly)

Given a sample$(x^{(i)}, y^{(i)})$, weight matrix W, we have scores vector:

$$s = f(x^{(i)}, W) = Wx^{(i)} \tag{10}$$

where the bias has been included in the above equation. And then loss function is defined by

$$L_i = \sum_{i \neq y^{(i)}} hinge(s_{y^{(i)}} - s_j) \tag{11}$$

where $hinge(t) = \max(0, 1 - t)$.

```
#Firstly we do a coarse hyperparameter tuning with a few iterations
max_out=100
d={}
for i in xrange(max_out):
    lr=10**np.random.uniform(-9, -6)
    rs=10**np.random.uniform(-5, 5)
    svm=LinearSVM()
    loss_history=svm.train(X_train, y_train, learning_rate=lr, reg=rs, num_iters=400)
    y_val_pred=svm.predict(X_val)
    acc_val=np.mean(y_val==y_val_pred)
    d[(lr, rs)]=acc_val
import operator
acc_val2hyperpara=sorted(d.items(), key=operator.itemgetter(1), reverse=True)
for item in acc_val2hyperpara:
    print "acc_val = %f, learninge rate= %.10f, regularation strength = %.6f" % (item[1], item[0][0], item[0][1])
#learning rate [5e-8, 1e-6], regularization [1e-3, 1e5]
```

```
acc_val = 0.368000, learninge rate= 0.000000, regularation strength = 22916.729569
acc_val = 0.367000, learninge rate= 0.000000, regularation strength = 32771.766373
acc_val = 0.361000, learninge rate= 0.000000, regularation strength = 68960.657860
acc_val = 0.355000, learninge rate= 0.000000, regularation strength = 60101.715440
acc_val = 0.354000, learninge rate= 0.000001, regularation strength = 0.011800
acc_val = 0.351000, learninge rate= 0.000000, regularation strength = 11163.596185
```

Fig. 1. Hyperparameter tuning (coarse)

```
max_out=50
for i in xrange(max_out):
    lr=10**np.random.uniform(-7.2, -6)
    rs=10**np.random.uniform(-3, 5)
    svm_current=LinearSVM()
    loss_history=svm_current.train(X_train, y_train,
                                   learning_rate=lr, reg=rs, num_iters=1500)
    y_val_pred=svm_current.predict(X_val)
    y_train_pred=svm_current.predict(X_train)
    acc_val=np.mean(y_val_pred==y_val)
    acc_train=np.mean(y_train_pred==y_train)
    results[(lr, rs)]=(acc_train, acc_val)
    if acc_val>best_val:
        best_val=acc_val
        best_svm=svm_current

################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print 'lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy)

print 'best validation accuracy achieved during cross-validation: %f' % best_val

#lr 7.046592e-07 reg 3.668956e+03 train accuracy: 0.384327 val accuracy: 0.397000
```

Fig. 2. Hyperparameter tuning (fine)

As for the gradient with W,

$$\begin{aligned}\frac{\partial L_i}{\partial W_{kl}} &= x_l^{(i)} \sum_{j \neq y^{(i)}} \begin{cases} (\delta_{k,j} - \delta_{k,y^{(i)}}) & if\ s_{y^{(i)}} - s_j < 1 \\ 0 & others \end{cases} \\ &= x_l^{(i)} \times \begin{cases} 1 & if\ y^{(i)} \neq k \wedge s_{y^{(i)}} - s_k < 1 \\ -C' & if\ y^{(i)} = k \\ 0 & others \end{cases} \end{aligned} \tag{12}$$

where C' is the number of j satisfying $s_{y^{(i)}} - s_j < 1$ except $j = y^{(i)}$. And then we can have the

following vertorized form of gradient

$$\begin{cases} s = X.dot(W.T) \\ correct\_score = s[range(X.shape[0]), y].reshape((X.shape[0], 1)) \\ L = s - correct\_score + 1 \\ L[L \leq 0] = 0 \\ L[L > 0] = 1 \\ L[range(X.shape[0]), y] = 0 \quad (remove\ j = y^{(i)}) \\ L[range(X.shape[0]), y] = -np.sum(L, axis = 1) \quad (i.e. - C') \\ \frac{\partial L}{\partial W} = L.T.dot(X) \end{cases} \tag{13}$$

where $X \in \mathbb{R}^{m \times D}, y \in \mathbb{R}^m, W \in \mathbb{R}^{C \times D}$($C$ is the number of classes). And then we average it and add the regularization term:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial W} / X.shape[0] + \lambda W \tag{14}$$

5.2 Softmax

Given a sample$(x^{(i)}, y^{(i)})$, weight matrix $W \in \mathbb{R}^{D \times C}$, we have scores vector:

$$f^{(i)} = f(x^{(i)}, W) = W^\intercal x^{(i)}$$

And then for softmax classifier we have the probability that sample i is in j^{th} class:

$$P_j^{(i)} = \frac{e^{f_j^{(i)}}}{\sum_j e^{f_j^{(i)}}} \tag{15}$$

And log-likelihood is

$$l = \sum_i \log \frac{e^{f_{y^{(i)}}^{(i)}}}{\sum_j e^{f_j^{(i)}}} \tag{16}$$

Based on the above function the loss function could be written as

$$L = \sum_i L_i = -\sum_i \log \frac{e^{f_{y^{(i)}}^{(i)}}}{\sum_j e^{f_j^{(i)}}} \tag{17}$$

So the gradient is

$$\frac{\partial L}{\partial W_{kl}} = \sum_i x_k^{(i)} \left(\frac{e^{f_l^{(i)}}}{\sum_j e^{f_j^{(i)}}} - \delta_{y^{(i)}, l} \right) = \sum_i x_k^{(i)} (P_l^{(i)} - \delta_{y^{(i)}, l}) \tag{18}$$

which is the same as the gradient of GLM:

$$\sum_i (h_\theta - y^{(i)}) x_j^{(i)} \tag{19}$$

Tips: For the term $\frac{e^{f_l^{(i)}}}{\sum_j e^{f_j^{(i)}}}$, in order to avoid overflow of computing, we have the following tricks

$$\frac{e^{f_l^{(i)}}}{\sum_j e^{f_j^{(i)}}} = \frac{e^{f_l^{(i)} - C}}{\sum_j e^{f_j^{(i)} - C}} \tag{20}$$

where $C = \max_l(f_l^{(i)})$.

Now we will make these equations vectorized. $X \in \mathbb{R}^{m \times D}, y \in \mathbb{R}^m, W \in \mathbb{R}^{D \times C}$($C$ is the number of classes), then

$$\begin{cases} f = X.dot(W) \\ f_{rebuild} = f - np.max(f, axis = 1).reshape(m, 1) \\ prob = np.exp(f_{rebuild}) / np.sum(np.exp(f_{rebuild}, axis = 1)).reshape(m, 1) \\ delta = np.zeros(prob.shape) \\ delta[range(m), y] = 1 \\ \frac{\partial L}{\partial W} = X.T.dot(prob - delta) \\ L = -np.sum(np.log(prob[range(m), y])) \end{cases} \tag{21}$$

5.3 Typical Neural networks

5.3.1 Tips about backpropagation

There is a gate computing $Y = WX$ where $Y \in \mathbb{R}^n$, $W \in \mathbb{R}^{(n \times m)}$ and $X \in \mathbb{R}^m$. For the forward computation, we just return Y based on the input W, X. As for the backprob, gradient of output on W, X might be complex but we can obtain them by dimension analysis. Also, this could be checked by careful computation.

$$\begin{cases} \frac{\partial o}{\partial W} = \frac{\partial o}{\partial Y} X^\intercal \\ \frac{\partial o}{\partial X} = W^\intercal \frac{\partial o}{\partial Y} \end{cases} \tag{22}$$

Analysis: If there is a little change dW_{ij} in W, then the corrsponding tiny change in Y is $dY_i = dW_{ij} X_j$ so that change in output is $do = \frac{\partial o}{\partial Y_i} dY_i = \frac{\partial o}{\partial Y_i} dW_{ij} X_j$. So we have $\frac{\partial o}{\partial W_{ij}} = \frac{\partial o}{\partial Y_i} X_j$.

5.3.2 Initialization

Small random numbers(gaussian with zero mean and 1e-2 standard deviation)

$$W = 0.01 * np.random.randn(D, H)$$

Works okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

Reasonable initialization

$$W = np.random.randn(fan\_in, fan\_out)/np.sqrt(fan\_in)$$

For ReLu

$$W = np.random.randn(fan\_in, fan\_out)/np.sqrt(fan\_in/2)$$

Batch normalization(you want unit gaussian activations? just make them so.)

$$\begin{cases} \mu_B = np.mean(X_B, axis = 0, keepdims = True) \\ \sigma_B = np.std(X_b, axis = 0, keepdims = True) \\ \hat{X_B} = \frac{X_B - \mu_B}{np.std + \epsilon} \\ Y = BN_{\gamma,\beta}(X) = \gamma * \hat{X_B} + \beta \end{cases}$$

where X_B is the mini_batch samples and γ and β are vectors of dimension D. Note: at test time BatchNorm layer functions differently: The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used. (e.g. can be estimated during training with running averages)

5.3.3 Details

We just implement a 2-layer neural network as an example, whose architecher is: input layer - fully connected (ReLu) - fully connected layer (softmax). Firstly, we have the following computation circuit (Forward Computation): $a \to h \to prob \to score \to loss$.

$$\begin{cases} a = X.dot(W1) + b1.reshape((1, H)) \\ h = np.maximum(0, a) \\ scores = h.dot(W2) + b2.reshape(1, C) \\ scores_{rebuild} = scores - np.max(scores, axis = 1).reshape(m, 1) \\ prob = np.exp(scores_{rebuild})/(np.sum(np.exp(scores_{rebuild}), axis = 1).reshape(m, 1)) \\ loss = -np.sum(np.log(prob[range(m), y])) \\ loss = loss/m \\ loss = loss + 0.5 * reg * (np.sum(W1 * W1) + np.sum(W2 * W2)) \end{cases} \tag{23}$$

where $X \in \mathbb{R}^{m \times D}, y \in \mathbb{R}^m, W1 \in \mathbb{R}^{D \times H}, W2 \in \mathbb{R}^{H \times C}, b1 \in \mathbb{R}^H, b2 \in \mathbb{R}^C$,

We know the loss function is computed as:

$$L = \sum_i L_i = -\sum_i \log \frac{e^{f^{(i)}_{y^{(i)}}}}{\sum_j e^{f^{(i)}_j}} \tag{24}$$

Then in backpropagation, we have:

$$\begin{cases} \frac{\partial L}{\partial W_{2,kl}} = \sum_i h_k^{(i)}(P_{y^{(i)}}^{(i)} - \delta_{l,y^{(i)}}) \\ \frac{\partial L}{\partial b_{2,l}} = \sum_i (P_{y^{(i)}}^{(i)} - \delta_{l,y^{(i)}}) \\ \frac{\partial L}{\partial h_{kl}} = \frac{\sum_j e^{f_j^{(k)}} W_{2,lj}}{\sum_i e^{f_i^{(k)}}} - W_{2,ly^{(k)}} = \sum_j P_j^{(k)} W_{2,lj} - W_{2,ly^{(k)}} \\ \frac{\partial L}{\partial a_{ij}} = \begin{cases} \frac{\partial L}{\partial h_{ij}} & if\ a_{ij} > 0 \\ 0 & others \end{cases} \\ \frac{\partial L}{\partial W_1} = X^\intercal \frac{\partial L}{\partial a} \\ \frac{\partial L}{\partial b_{1,k}} = \sum_i \frac{\partial L}{\partial a_{ik}} \end{cases} \tag{25}$$

Make them vectorized:

$$\begin{cases} delta = np.zeros(prob.shape) \\ delta[range(m), y] = 1 \\ \frac{\partial L}{\partial W2} = h.T.dot(prob - delta) \\ \frac{\partial L}{\partial b2} = np.sum(prob - delta, axis = 0) \\ \frac{\partial L}{\partial h} = (prob - delta).dot(W2.T) \\ \frac{\partial L}{\partial a} = \frac{\partial L}{\partial h} \\ \frac{\partial L}{\partial a}[a <= 0] = 0 \\ \frac{\partial L}{\partial W1} = X.T.dot(\frac{\partial L}{\partial a}) \\ \frac{\partial L}{\partial b1} = np.sum(\frac{\partial L}{\partial a}, axis = 0) \end{cases} \tag{26}$$

Using the above equations which includes a ReLu gate and Softmax gate, we can easily to have a multi-layer NN architecher whose backpropagation process is computed iteratedly.

5.3.4 Regularization

It is most common to use a single, global $L2$ regularization strength that is cross- validated. It is also common to combine this with dropout applied after all layers. The value of $P = 0.5$ is a reasonable default, but this can be tuned on validation data.

5.3.5 Some gates for NN

Python codes of the implement of these gates are shown in the following:

5.3.6 Suggestions

- **about size and number of hidden layers** In practice it is often the case that 3-layer neural networks will outperform 2-layer nets, but going even deeper (4,5,6-layer) rarely helps much more. This is in stark contrast to Convolutional Networks, where depth has been found to be an extremely important component for a good recognition system (e.g. on order of

10 learnable layers). One argument for this observation is that images contain hierarchical structure (e.g. faces are made up of eyes, which are made up of edges, etc.), so several layers of processing make intuitive sense for this data domain. (A kind of argument about size of hidden layers is: num_classes ĩnput_size)

- **about overfitting** In practice, it is always better to use these methods (such as L2 regularization, dropout, input noise) to control over tting instead of the number of neurons. The takeaway is that you should not be using smaller networks because you are afraid of overfitting. Instead, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control over tting. **A subtle reason** what you find is that if you train a small network the nal loss can display a good amount of variance - in some cases you get lucky and converge to a good place but in some cases you get trapped in one of the bad minima. On the other hand, if you train a large network you' ll start to nd many different solutions, but the variance in the nal achieved loss will be much smaller. In other words, all solutions are about equally as good, and rely less on the luck of random initialization

5.4 Convolutional Neural Network

Convolutional filter hyperparameter setting:

- Number of filters - $F = 2^k$, e.g. $32, 64, 128, 512$
- Filter size - $WW = HH = 1, 3, 5$(odd number)
- Zero-padding - keep the same size after Conv-operation. e.g. $P = \frac{WW-1}{2} = 0, 1, 2$ if stride $S = 1$
- Stride - $S = 1$ (maybe you can try others)

Pooling setting (max pooling is most common):

- Filter size - $WW = HH = 2, S = 2$
- Filter size - $WW = HH = 3, S = 2$

5.5 Recurrent Neural Network

5.5.1 LSTM

There are 4 gates in LSTM, input gate, forget gate, output gate as well as g gate which are shown in Fig.3 and their expressions are the following:

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix} \tag{27}$$

where h_t^l is the $l^{th} - depth$ layer at t^{th} timestep (In the above equations, you may think h_t^{l-1} is the external input x). Compared with plain RNN, there are two state vectors in LSTM. One is cell

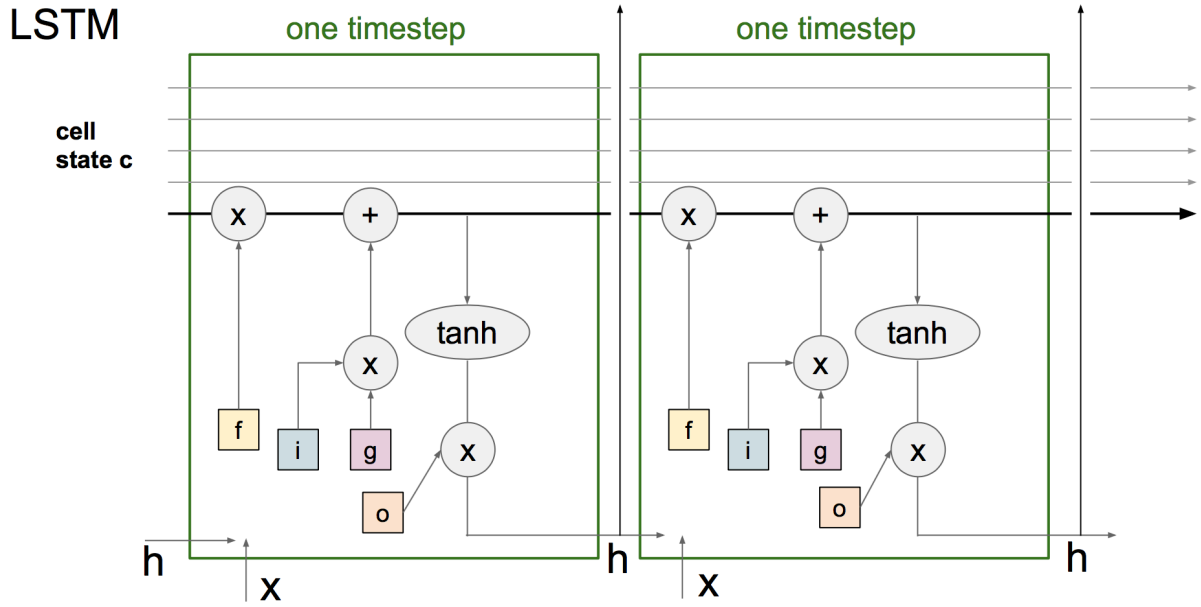

Fig. 3. LSTM architecture

state c_t^l and another hidden state h_t^l:

$$\begin{cases} c_t^l = f \odot c_{t-1}^l + i \odot g \\ h_t^l = o \odot \tanh(c_t^l) \end{cases} \tag{28}$$

More generally, we have the following equations:

$$\begin{cases} h_t^* = sigm\left(U^* x_t + W^* h_{t-1} + b^*\right) \\ h_t^g = \tanh\left(U^g x_t + W^g h_{t-1} + b^g\right) \end{cases} \tag{29}$$

where $*$ represents i, f, o i.e. input gate, forget gate and output gate. Also,

$$\begin{cases} c_t = h_t^f \odot c_{t-1} + h_t^i \odot h_t^g \\ h_t = h_t^o \odot \tanh(c_t) \end{cases} \tag{30}$$