L5-1 Making NN's work : optimization and regularization

Optimization

Following Sebastian Ruder
ruder.io/optimizing-gradient-descent

Gradient descent : batch

$\theta = \theta - \eta \nabla_\theta J(X, Y, \theta)$

Stochastic gradient descent

$\theta = \theta - \eta \nabla_\theta J(X^{(i)}, Y^{(i)}, \theta)$ $i \sim \text{Uniform}(1..n)$

- moves faster to take advantage of gradient info
- high variance ("bounces around")

Mini-batch gradient descent param $1 < K < n$

Let $\mathcal{I}$ be a set of K indices $\sim \text{Uniform}(1..n)$

$\theta = \theta - \eta \nabla_\theta J(X^{\mathcal{I}}, Y^{\mathcal{I}}, \theta)$

Tune K to decrease variance while maintaining speed

Usual strategy: randomly shuffle data, work through from beginning to end in groups of K, reshuffle, etc.

Step size

Too small → too slow

Too big → diverges, converges slowly due to oscillation

Using SGD or minibatch means we need to decrease η as a function of t — but how?

In deep networks, gradients can

exponentially

- explode: during back prop, the magnitude of ∇_{W^l} Loss increases as l goes from $L \to 1$
- vanish: " decreases "

One way to address this problem is to

- have a different η *for every W^l_{ij}*
- adapt them online

L 5-2 Momentum

Idea: try to "average" recent gradient updates

without momentum   with momentum

$$V_t = \gamma V_{t-1} + \eta \nabla_\theta J(X, y, \theta)$$

can be batch, mini, single

$$\theta = \theta - V_t$$

Running average

$$A_t = \gamma_t A_{t-1} + (1-\gamma_t) a_t \qquad A_0 = 0$$

If you set $\gamma_t = \frac{t-1}{t}$ then

$$A_1 = a_1$$

$$A_2 = \frac{1}{2} A_1 + \frac{1}{2} a_2 = \frac{1}{2}(a_1 + a_2)$$

$$A_3 = \frac{2}{3} A_2 + \frac{1}{3} a_3 = \frac{2}{3} \cdot \frac{1}{2}(a_1 + a_2) + \frac{1}{3} a_3 = \frac{1}{3}(a_1 + a_2 + a_3)$$

V_t will be bigger in dimensions that consistently have the same sign for ∇_θ and smaller for those that don't.

Need to tweak γ – 0.9 is common

Adagrad

Idea: want to take bigger steps in flat places (plateaus) and smaller steps when it's steep

too lazy to write θ $w_{j,t}$

$$g_{tj} = \nabla_\theta J(X, y, \theta_{t,j})$$

$\boxed{j\text{: index of a weight in whole network}}$

$$G_{t,j} = G_{t-1,j} + g_{tj}^2$$

how steep is it, in this dimension?

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_{t,j} + \varepsilon}} g_{tj}$$

smaller if steeper

Like running avg, but more weight on recent values.

Adadelta

Idea: adagrad is good, but G_{tj} just keeps growing — use a decaying average!

So, change

$$G_{t,j} = \gamma G_{t-1,j} + (1-\gamma)\, g_{tj}^2 \qquad \gamma \approx .9$$

$\boxed{\text{Adam}}$

Combine ideas of momentum and adadelta:

$$m_{tj} = \beta_1 m_{t-1,j} + (1-\beta_1)\, g_{tj} \qquad \text{mean}$$

$$v_{tj} = \beta_2 v_{t-1,j} + (1-\beta_2)\, g_{tj}^2 \qquad \text{variance}$$

Idea: If we initialize m and v to 0, then these estimates will always be biased. Correct with

$$\hat{m}_{tj} = \frac{m_{tj}}{1-\beta_1^t}$$

$$\hat{v}_{tj} = \frac{v_{tj}}{1-\beta_2^t}$$

$$\theta_{t+1,j} = \theta_{tj} - \frac{\eta}{\sqrt{\hat{v}_{tj}} + \varepsilon}\, \hat{m}_{tj}$$

Authors propose: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$

* Much less sensitive to these parameters than to η in original rule *

For compactness in implementation

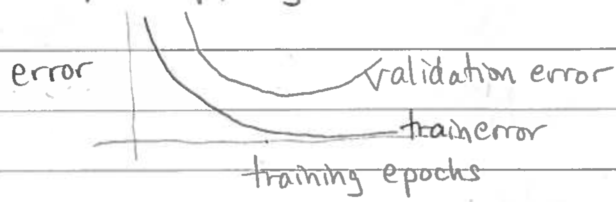
- m_t^l, v_t^l, g_t^l, $g_t^{l\,2}$ are matrices

componentwise square of g_t^l

5-4 Regularization

Many simple strategies:

- early stopping



- weight decay

$$J(X, Y, \theta) = \sum_{i=1}^{n} \text{Loss}(NN(x^{(i)}), y^{(i)}) + \lambda \|\theta\|^2$$

gives gradient rule of the form

$$\theta = \theta - \eta(\nabla_\theta \text{Loss} + \lambda\theta)$$

$$= \theta(1-\lambda\eta) - \eta \nabla_\theta \text{Loss}$$

Not so successful in deep neural nets in practice
(current research question to explain why)

- dropout : parameter p is a probability,
often set to 0.5

During training:

- for each training example, for each unit,
randomly with probability p
temporarily set $a_j^l := 0$
  - $\Rightarrow$ No contribution to output
  - $\Rightarrow$ No gradient update
- As if we are training a different subnet each time

During testing/operation:

- multiply all weights by p
to achieve same average activation levels

5-5 Dropout implementation

In forward pass during training,

$$a^l = f(z^l) * d^l$$

componentwise product

vector of 0's and 1's drawn randomly with prob p

Backward pass depends on a^l, so everything works out.