L6 - Optimization Algorithms

2017/11/13 19:35

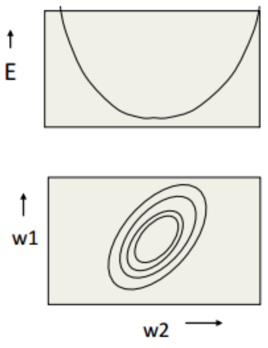
- Summary of learning methods for neural networks
  - For small datasets (e.g. 10,000 cases) or bigger datasets without much redundancy, use a full-batch method.
    - - Conjugate gradient, LBFGS ...
      - they have own package, which saves the justification of methods for publications
    - - adaptive learning rates, rprop ...
      - both are good
  - For big, redundant datasets (must) use minibatches.
    - - Try gradient descent with momentum.
      - tune the learning rate
    - - Try rmsprop (with momentum ?)
      - works same or better than GD+momentum
    - - Try LeCun's latest recipe.
      - the expert in SGD

- Overview of mini-batch gradient descent
  - SUM
    - For multi-layer, non-linear nets, locally, a piece of a quadratic bowl is usually a very good approximation.
    - for big and highly redundant data sets, use mini-batch GD
    - tune the learning rate through the learning
      - guess and then increase or decrease
      - weight decay - gradually reduce the learning rate

  - Reminder: The error surface for a linear neuron

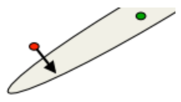
The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
- For a linear neuron with a squared error, it is a quadratic bowl.
- Vertical cross-sections are parabolas.
- Horizontal cross-sections are ellipses.

For multi-layer, non-linear nets the error surface is much more complicated.
- But locally, a piece of a quadratic bowl is usually a very good approximation.



        - ATT
            - For multi-layer, non-linear nets, locally, a piece of a quadratic bowl is usually a very good approximation. So when considering optimization, a cluster of ellipses is a good approximation
    - Problem with full batch GD

Going downhill reduces the error, but the direction of steepest descent does not point at the minimum unless the ellipse is a circle.
- The gradient is big in the direction in which we only want to travel a small distance.
- The gradient is small in the direction in which we want to travel a large distance.

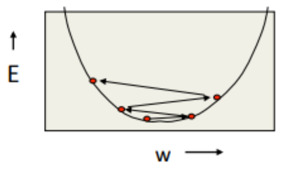


Even for non-linear multi-layer nets, the error surface is locally quadratic, so the same speed issues apply.

        - the direction is not optimal
    - About the learning rate

If the learning rate is big, the weights slosh to and fro across the ravine.
- If the learning rate is too big, this oscillation diverges.

What we would like to achieve:
- Move quickly in directions with small but consistent gradients.
- Move slowly in directions with big but inconsistent gradients.



        - too big learning rate cause overshoot;
        - the desirable direction is pointing to the minimum
    - Stochastic gradient descent
        - update the weights with each data points
        - when to apply it
            - the data set is highly redundant [almost all data sets suffers ceratin degree of redundant]
    - Mini-batch are usually better than SGD
        - AD
            - less computation compared to GD/SGD

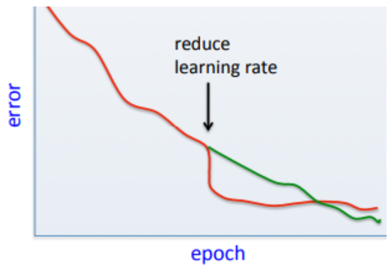
        - more efficiently computation than SGD due to parallel computing, matrix-matrix computation
    - ATT
        - Mini-batches need to be balanced for classes, i.e. containing almost equal #examples for all classes
        - For large neural networks with very large and highly redundant training sets, it is nearly always best to use mini-batch learning.
    - algorithm - how to apply MBGD
        - tune learning rate
            - guess one and then tune it
            - when error keeps getting worse or oscillates wildly, decrease it
            - If the error is falling fairly consistently but slowly, increase the learning rate.
        - Write a simple program to automate this way of adjus@ng the learning rate.
        - weight decay - gradually reduce the learning rate to smoothe away the fluctuations
        - Turn down the learning rate when the error stops decreasing.
        - the criteria to measure the error stops decrease is using validation sets

- A bag of tricks for mini-batch gradient descent
    - Be careful about turning down the learning rate, Don’t turn down the learning rate too soon!
        - trade-off between reducing fluctuation and update speed

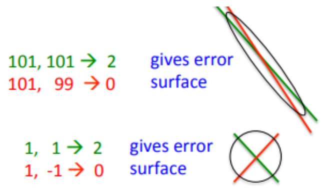
Turning down the learning rate reduces the random fluctuations in the error due to the different gradients on different mini-batches.

- So we get a quick win.
- But then we get slower learning.

Don't turn down the learning rate too soon!



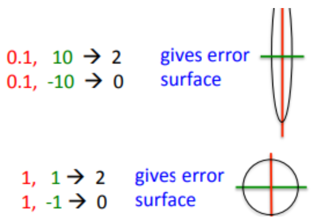
- Initializing the weights
  - symmetry breaking:
    - initializing the weights to have small random values, never with equal values.
  - fan-in = #input (Fan-in is the number of inputs a gate can handle.)
    - If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
      - - We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportioonal to sqrt(fan-in).
    - • We can also scale the learning rate the same way.
- Shifting the inputs to make them have a zero mean which is beneficial for learning
  - how
    - adding a constant to all the inputs
  - the reasons



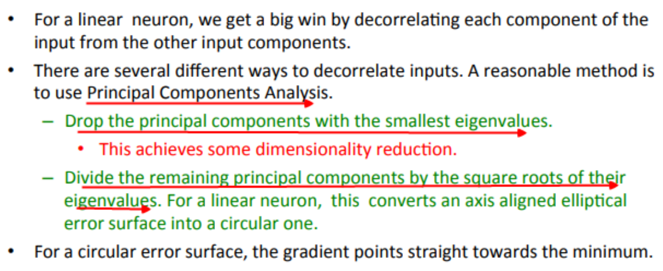
  - so tanh() is better than sigmoid
    - but sigmoid has its advantages;
      - e.g. logistic gives you a rug to sweep things under. It gives an output of zero,

and if you make the input even smaller than it was, the output is still zero. So fluctuations in big native inputs are ignored by the logistic. For the hyperbolic tangent you have to go out to the end of its plateaus before it can ignore anything.

- Scaling the inputs to make it have unit variance over the whole training set.



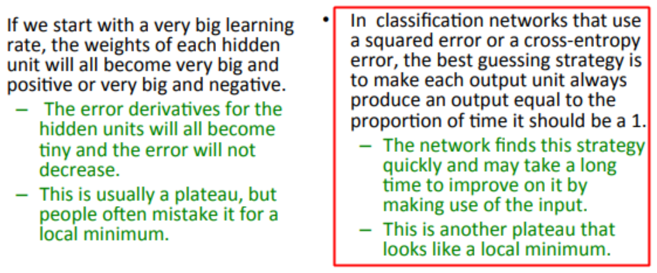
- A more thorough method: Decorrelate the input components grantees a circle



- For a linear neuron, we get a big win by decorrelating each component of the input from the other input components.
- There are several different ways to decorrelate inputs. A reasonable method is to use Principal Components Analysis.
  - Drop the principal components with the smallest eigenvalues.
    - This achieves some dimensionality reduction.
  - Divide the remaining principal components by the square roots of their eigenvalues. For a linear neuron, this converts an axis aligned elliptical error surface into a circular one.
- For a circular error surface, the gradient points straight towards the minimum.

    - PCA
        - drop the PC with smallest eigenvalues
        - divide the rest by sqrt(eigenvalues): elliptical => circle

- plateau problems that occur in multilayer networks
    - never initailize weights to be big weights

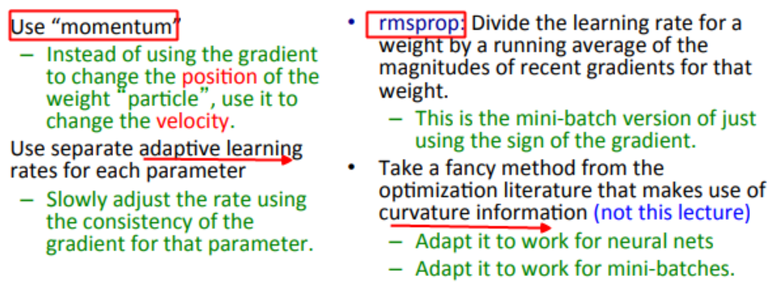


If we start with a very big learning rate, the weights of each hidden unit will all become very big and positive or very big and negative.
- The error derivatives for the hidden units will all become tiny and the error will not decrease.
- This is usually a plateau, but people often mistake it for a local minimum.

- In classification networks that use a squared error or a cross-entropy error, the best guessing strategy is to make each output unit always produce an output equal to the proportion of time it should be a 1.
  - The network finds this strategy quickly and may take a long time to improve on it by making use of the input.
  - This is another plateau that looks like a local minimum.

- Four ways to speed up mini-batch learning
    - Momentum,
    - separate adaptive learning rate for each parameter
    - rmsprop

- curvature information

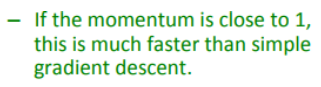


Use "momentum"
- Instead of using the gradient to change the position of the weight "particle", use it to change the velocity.

Use separate adaptive learning rates for each parameter
- Slowly adjust the rate using the consistency of the gradient for that parameter.

- rmsprop: Divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
  - This is the mini-batch version of just using the sign of the gradient.
- Take a fancy method from the optimization literature that makes use of curvature information (not this lecture)
  - Adapt it to work for neural nets
  - Adapt it to work for mini-batches.

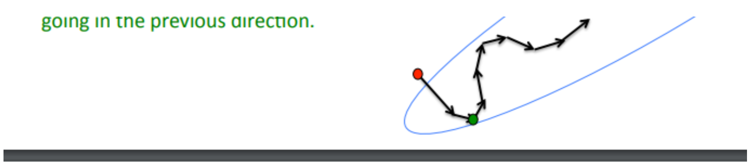
- The momentum method
  - SUM
    - use mini-batch GD + momentum learns much faster than without it
    - tune the momentum rate, 0.5 when gradients are large, then gradually increase to 0.99/0.9 the gradients get small.
  - domain
    - full-batch or mini-batch GD; the most popular is mini-batch + momentum
  - benefits with it
    - directions of consistant change get amplified
    - directions of fluctuations get damped
    - allows using much larger learning rate
    - much fast than without it



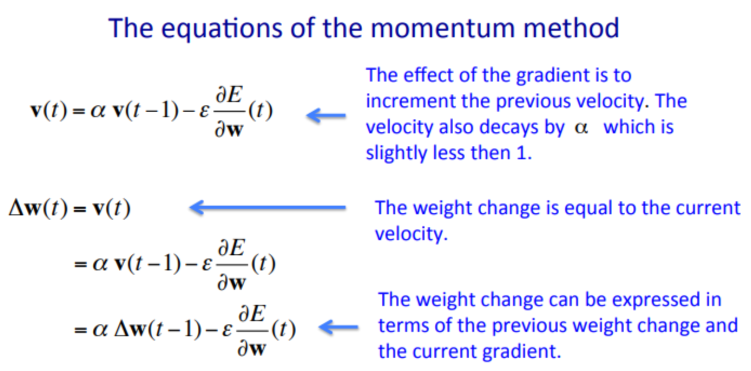
- If the momentum is close to 1, this is much faster than simple gradient descent.

$$\mathbf{v}(\infty) = \frac{1}{1-\alpha}\boxed{\left(-\varepsilon\frac{\partial E}{\partial \mathbf{w}}\right)}$$

- the term in the red box is the original update term
  - why can it be



going in the previous direction.



The equations of the momentum method

$$\mathbf{v}(t) = \alpha\ \mathbf{v}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$

The effect of the gradient is to increment the previous velocity. The velocity also decays by α which is slightly less then 1.

$$\Delta \mathbf{w}(t) = \mathbf{v}(t)$$

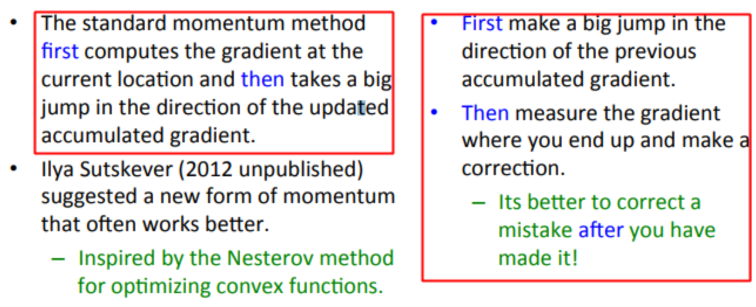
The weight change is equal to the current velocity.

$$= \alpha\ \mathbf{v}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$

$$= \alpha\ \Delta \mathbf{w}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$

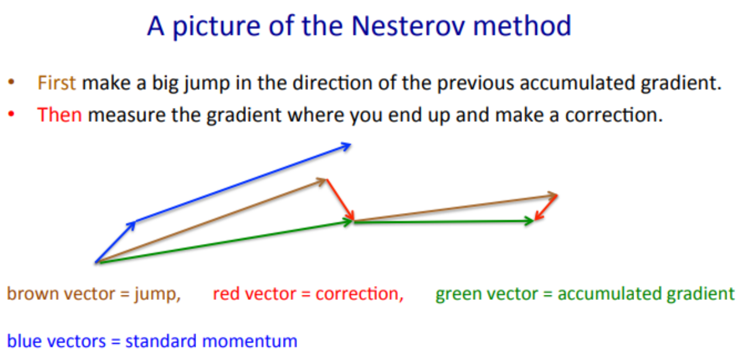
The weight change can be expressed in terms of the previous weight change and the current gradient.

- How to make use of it
  - At the beginning of learning there may be very large gradients.
    - - So it pays to use a small momentum
      - (e.g. 0.5). it will average out some sloshes and obvious ravines
    - - Once the large gradients have disappeared and the weights are stuck in a ravine, the momentum can be smoothly raised to its final value (e.g. 0.9 or even 0.99)
  - normal learning_rate + momentum faster than normal learning rate
- A better type of momentum (Nesterov 1983)
  - take a step then correct it is better than a step after correction



- The standard momentum method first computes the gradient at the current location and then takes a big jump in the direction of the updated accumulated gradient.
- Ilya Sutskever (2012 unpublished) suggested a new form of momentum that often works better.
  - Inspired by the Nesterov method for optimizing convex functions.

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.
  - Its better to correct a mistake after you have made it!



A picture of the Nesterov method

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.

brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

  -

- A separate, adaptive learning rate for each connection
  - SUM
    - adaptive learning is powerful;
      - if gd(t)*gd(t-1)>0
        - increase g by adding 0.05
      - else
        - decrese g by multiply 0.095
    - ATT
      - range
      - full batch GD or big mini-batch GD
      - + momentum
  - core idea
    - if the weight keeps reversing its gradient, we turn down the learning weight.
    - And if the gradient stays consistent, we turn up the learning weight.
  - reasons
    - In a multilayer net, the appropriate learning rates can vary widely between weights
      - The magnitudes of the gradients are often very different for different layers, especially if the initial weights are small.
      - The fan-in of a unit determines the size of the “overshoot” effects caused by simultaneously changing many of the incoming

weights of a unit to correct the same error.

- how it is implemented
    - use a global learning rate (set by hand) multiplied by an appropriate local gain that is determined empirically for each weight.
    - one way to do this
        - procedure
            - initialize the gain with 1 for each weight
            - if the gradient doesn't change the sign
                - increase the gain wieh a little value
            - else
                - multiply the gain with a large value (close to 1)
        - e.g.

$$\Delta w_{ij} = -\varepsilon \, g_{ij} \frac{\partial E}{\partial w_{ij}}$$

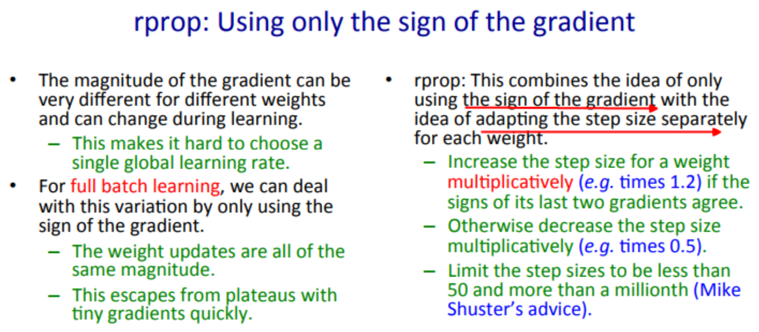
$$\begin{aligned} &\mathit{if} \quad \left(\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0 \\ &\mathit{then} \quad g_{ij}(t) = g_{ij}(t-1) + .05 \\ &\mathit{else} \quad g_{ij}(t) = g_{ij}(t-1) * .95 \end{aligned}$$

        - AD
            - ensures that big gains decay rapidly when oscillations start.
            - If the gradient is totally random the gain will hover around 1 when we increase by plus half the time and decrease by times half the time.
            - for oscillations, gain < 1; for

right direction, gain >1

- Tricks for making adaptive learning rates work better
    - • Limit the gains to lie in some reasonable range
        - - e.g. [0.1, 10] or [.01, 100] •
        - if it's too big, they won't die down fast and destroy all the weights
    - Use full batch learning or big minibatches
        - - This ensures that changes in the sign of the gradient are not mainly due to the sampling error of a minibatch.
    - • Adaptive learning rates can be combined with momentum.
        - - Use the agreement in sign between the current gradient for a weight and the velocity for that weight (Jacobs, 1989).
        - i.e. gradient(t) & Δw(t-1) / v(t-1)
    - ???• Adaptive learning rates only deal with axis-aligned effects.
        - - Momentum does not care about the alignment of the axes.
        - Momentum can deal with these diagonal ellipses and going in that diagonal direction quickly which adaptive learning rates can't do.

- rmsprop: Divide the gradient by a running average of its recent magnitude
    - Misc
        - it's Geoffrey's favorite learning methods fro large NN with a large redundant data set
    - rprop: Using only the sign of the gradient without caring about the magnitude

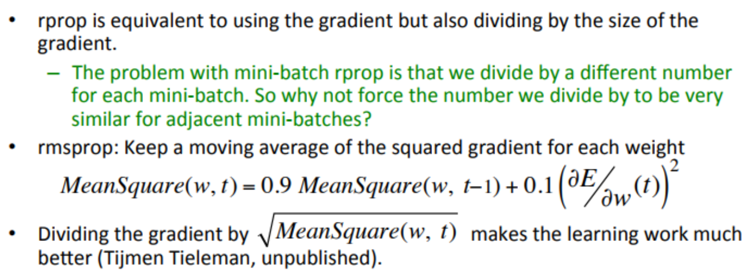


rprop: Using only the sign of the gradient

- The magnitude of the gradient can be very different for different weights and can change during learning.
  - This makes it hard to choose a single global learning rate.
- For full batch learning, we can deal with this variation by only using the sign of the gradient.
  - The weight updates are all of the same magnitude.
  - This escapes from plateaus with tiny gradients quickly.
- rprop: This combines the idea of only using the sign of the gradient with the idea of adapting the step size separately for each weight.
  - Increase the step size for a weight multiplicatively (*e.g.* times 1.2) if the signs of its last two gradients agree.
  - Otherwise decrease the step size multiplicatively (*e.g.* times 0.5).
  - Limit the step sizes to be less than 50 and more than a millionth (Mike Shuster's advice).

        - for the size range
            - If for example you have a problem with some tiny inputs, you might need very big weights on those inputs for them to have an effect. I suspect that if you're not dealing with that kind of problem, having an upper limit on the weight changes that's much less than 50 would be a good idea.
    - Why rprop does not work with mini-batches
        - because rprop violates the The idea behind stochastic gradient descent, that when the learning rate is small, it averages the gradients over successive minibatches. Since it only consider the sign of the gradient
            - e.g. Consider a weight that gets a gradient of +0.1 on nine minibatches and a gradient of -0.9 on the tenth mini-batch.
                - when learning rate is small and similar, it averages the difference, so this weight stays roughly where it is.
                - but rprop would increase the gain 9 times and decrease it once. So the weight would grow a lot.
    - rmsprop: A mini-batch version of rprop
        - AD: it combines:
            - - The robustness of rprop.
            - - The efficiency of mini-batches.

            - - The effective averaging of gradients over mini-batches.
        - how it is implemented

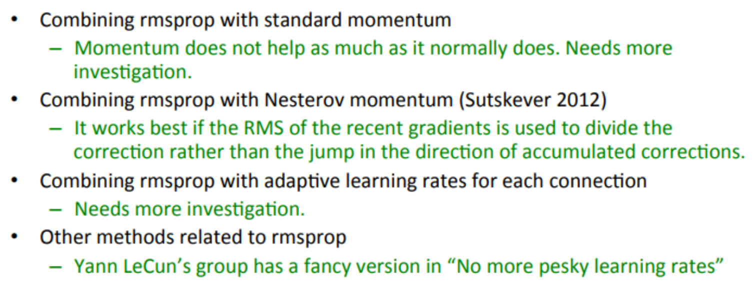


- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t{-}1) + 0.1\left(\partial E/\partial w\,(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

    - Further developments of rmsprop



- Combining rmsprop with standard momentum
  - Momentum does not help as much as it normally does. Needs more investigation.
- Combining rmsprop with Nesterov momentum (Sutskever 2012)
  - It works best if the RMS of the recent gradients is used to divide the correction rather than the jump in the direction of accumulated corrections.
- Combining rmsprop with adaptive learning rates for each connection
  - Needs more investigation.
- Other methods related to rmsprop
  - Yann LeCun's group has a fancy version in "No more pesky learning rates"

- Summary of learning methods for neural networks
    - For small datasets (e.g. 10,000 cases) or bigger datasets without much redundancy, use a full-batch method.
        - - Conjugate gradient, LBFGS ...
            - they have own package, which saves the justification of methods for publications
        - - adaptive learning rates, rprop ...
    - For big, redundant datasets (must) use minibatches.
        - - Try gradient descent with momentum.
            - tune the learning rate
        - - Try rmsprop (with momentum ?)
            - works same or better than GD+momentum
        - - Try LeCun’s latest recipe.
            - the expert in SGD
- Why there is no simple recipe:
    - Neural nets differ a lot:
        - - Very deep nets (especially ones with narrow bottlenecks).
        - - Recurrent nets.
        - - Wide shallow nets.
    - Tasks differ a lot:

- - Some require very accurate weights, some don’t.
- - Some have many very rare cases (e.g. words).