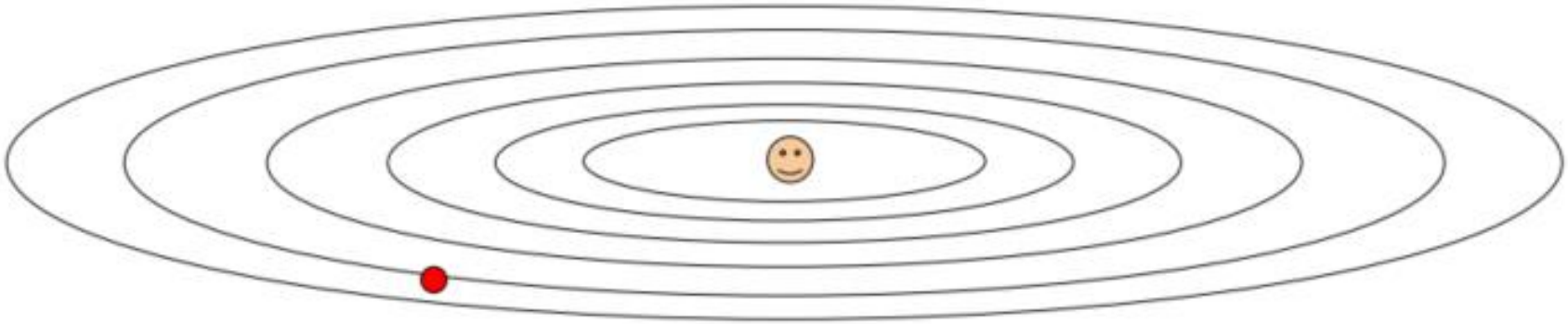
Improving Gradient Descent Learning in Neural Networks



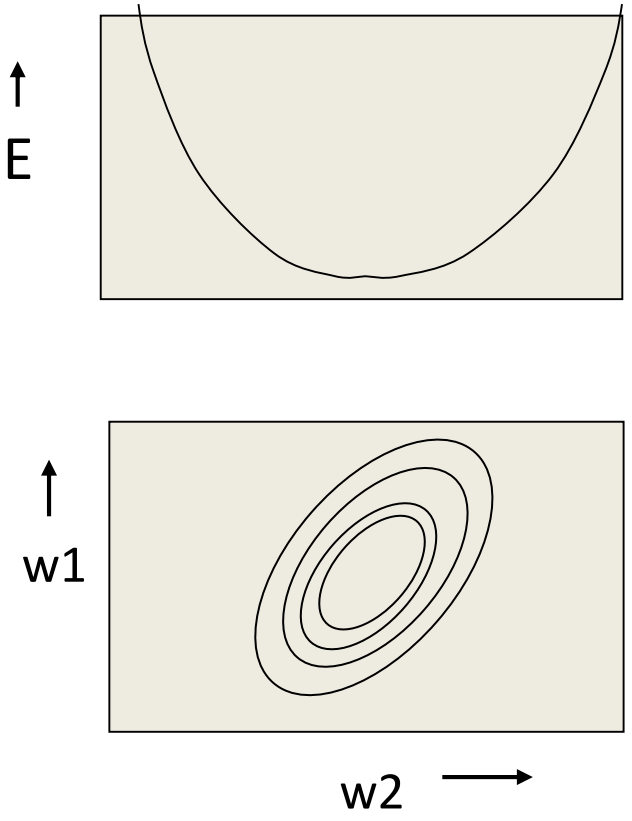
Slides from Geoffrey Hinton

CSC321: Intro to Machine Learning and Neural Networks, Winter 2016

Michael Guerzhoy

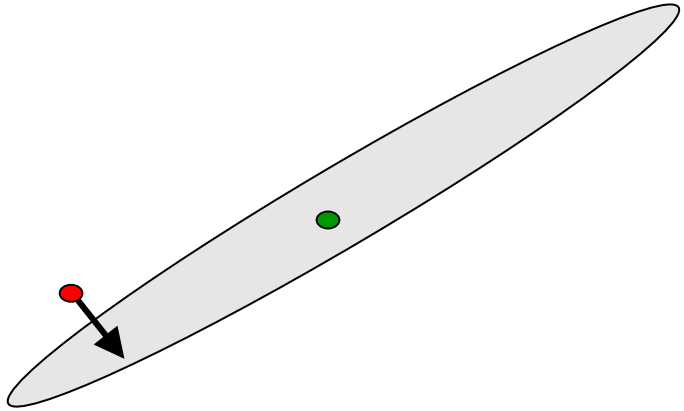
The Surface Error For Neural Networks

- 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.
- 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.



Convergence speed of full batch learning when the error surface is a quadratic bowl

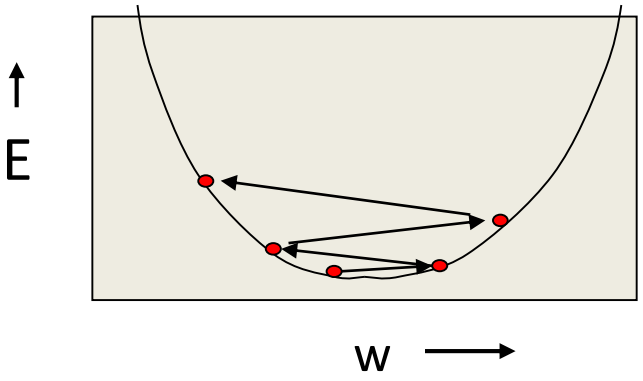
- Going downhill reduces the error, but the direction of steepest descent does not point at the minimum unless the ellipse is a circle.
  - The gradient is big in the direction in which we only want to travel a small distance.
  - The gradient is small in the direction in which we want to travel a large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



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

How Learning Goes Wrong

- 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.



Stochastic Gradient Descent

- If the dataset is highly redundant, the gradient on the first half is almost identical to the gradient on the second half.
  - So instead of computing the full gradient, update the weights using the gradient on the first half and then get a gradient for the new weights on the second half.
  - The extreme version of this approach updates weights after each case. Its called “online”.
- Mini-batches are usually better than online.
  - Less computation is used updating the weights.
  - Computing the gradient for many cases simultaneously uses matrix-matrix multiplies which are very efficient, especially on GPUs
- Mini-batches need to be balanced for classes

Two Types of Learning Algorithm

If we use the full gradient computed from all the training cases, there are many clever ways to speed up learning (e.g. non-linear conjugate gradient).

- The optimization community has studied the general problem of optimizing smooth non-linear functions for many years.
- Multilayer neural nets are not typical of the problems they study so their methods may need a lot of adaptation.

For large neural networks with very large and highly redundant training sets, it is nearly always best to use mini-batch learning.

- The mini-batches may need to be quite big when adapting fancy methods.
- Big mini-batches are more computationally efficient.

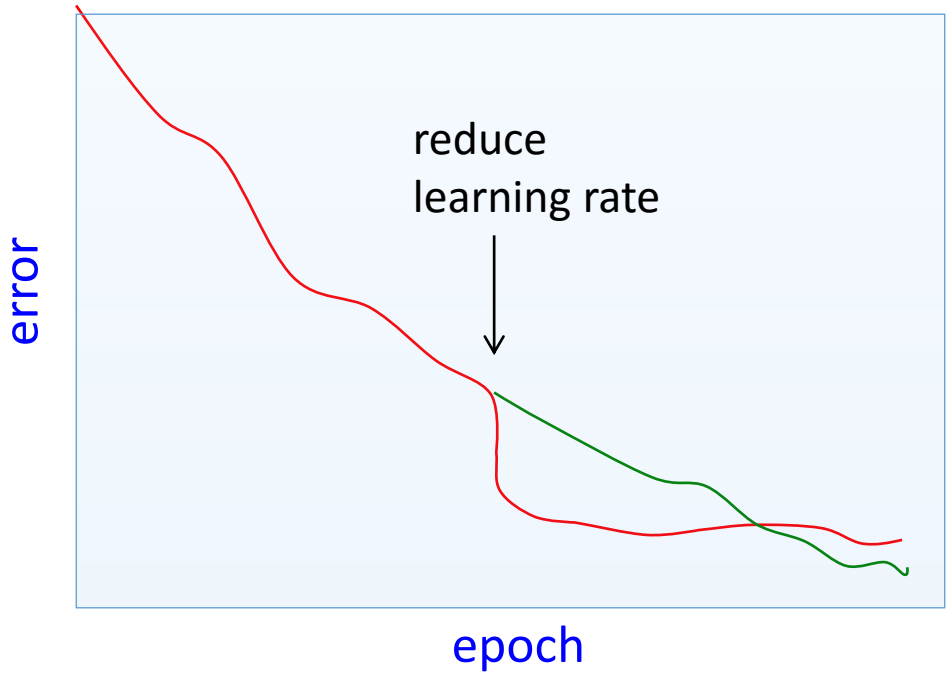
A Basic Mini-Batch Gradient Descent Algorithm

- Guess an initial learning rate.
  - If the error keeps getting worse or oscillates wildly, reduce the learning rate.
  - If the error is falling fairly consistently but slowly, increase the learning rate.
- Write a simple program to automate this way of adjusting the learning rate.

- Towards the end of mini-batch learning it nearly always helps to turn down the learning rate.
  - This removes fluctuations in the final weights caused by the variations between mini-batches.
- Turn down the learning rate when the error stops decreasing.
  - Use the error on a separate validation set

Careful about turning down the learning rate

- 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!

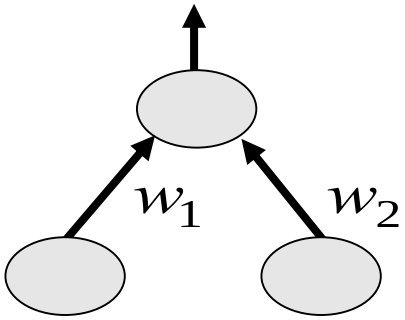


Initializing the Weights

- If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.
  - So they can never learn to be different features.
  - We break symmetry by initializing the weights to have small random values.
- 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 proportional to sqrt(fan-in).
- We can also scale the learning rate the same way.

Shifting the Inputs

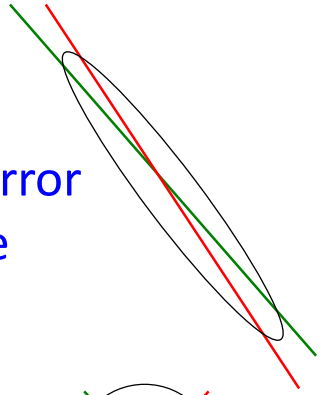
- When using steepest descent, shifting the input values makes a big difference.
  - It usually helps to transform each component of the input vector so that it has zero mean over the whole training set.
- The tanh activation (which is 2*logistic -1) produces hidden activations that are roughly zero mean.
  - In this respect its better than the logistic.



color indicates
training case

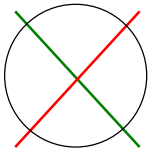
101, 101 → 2
101, 99 → 0

gives error
surface



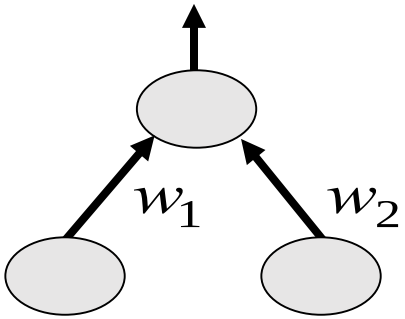
1, 1 → 2
1, -1 → 0

gives error
surface



Scaling the Inputs

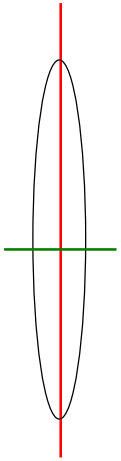
- When using steepest descent, scaling the input values makes a big difference.
  - It usually helps to transform each component of the input vector so that it has unit variance over the whole training set.



color indicates weight axis

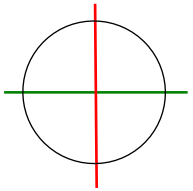
0.1, 10 → 2
0.1, -10 → 0

gives error surface



1, 1 → 2
1, -1 → 0

gives error surface



Common Problems

- 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.

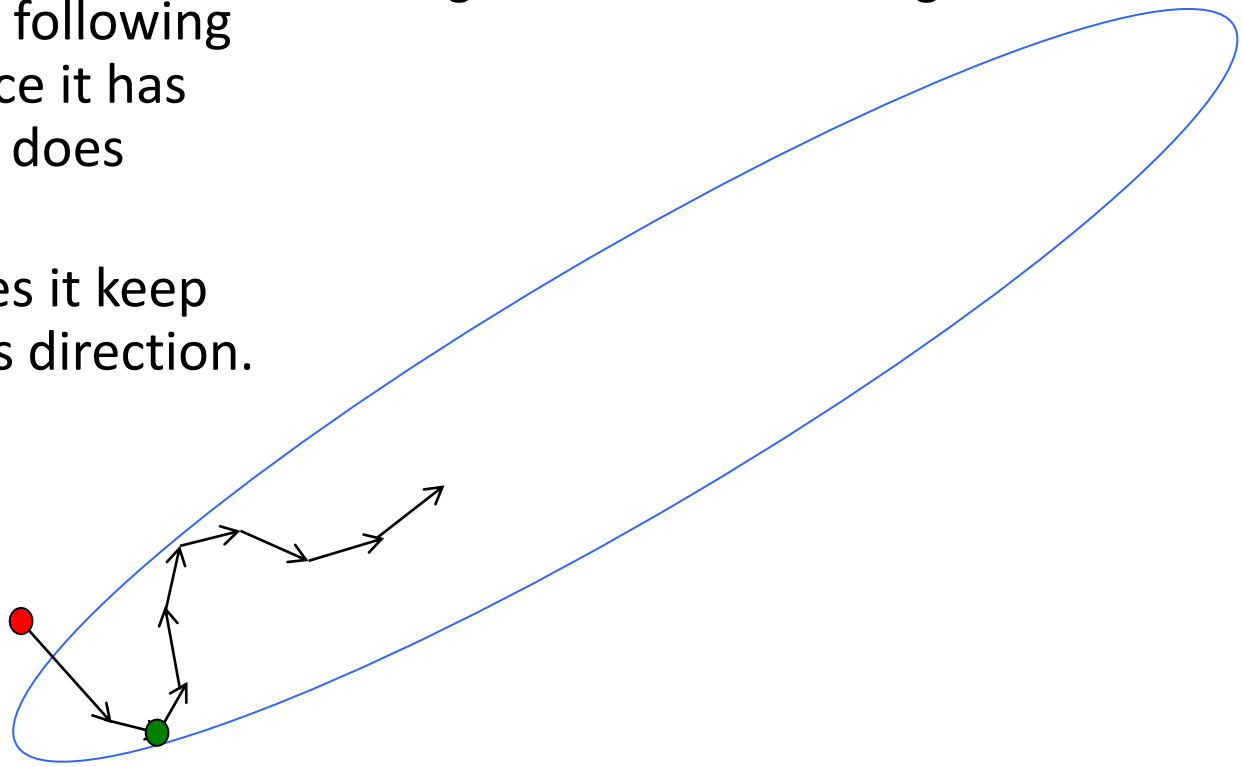
Speeding Up Learning

- 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.

Momentum Method

Imagine a ball on the error surface. The location of the ball in the horizontal plane represents the weight vector.

- The ball starts off by following the gradient, but once it has velocity, it no longer does steepest descent.
- Its momentum makes it keep going in the previous direction.
- It damps oscillations in directions of high curvature by combining gradients with opposite signs.
- It builds up speed in directions with a gentle but consistent gradient.



Momentum Method

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

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

$$\Delta w(t) = v(t)$$

The weight change is equal to the current velocity.

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

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

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

Momentum Method

- If the error surface is a tilted plane, the ball reaches a terminal velocity.
  - If the momentum is close to 1, this is much faster than simple gradient descent.

$$v(\infty) = \frac{1}{1-\alpha}(-\eta \frac{\partial E}{\partial w})$$

- At the beginning of learning there may be very large gradients.
  - So it pays to use a small momentum (e.g. 0.5).
  - 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)
- This allows us to learn at a rate that would cause divergent oscillations without the momentum.

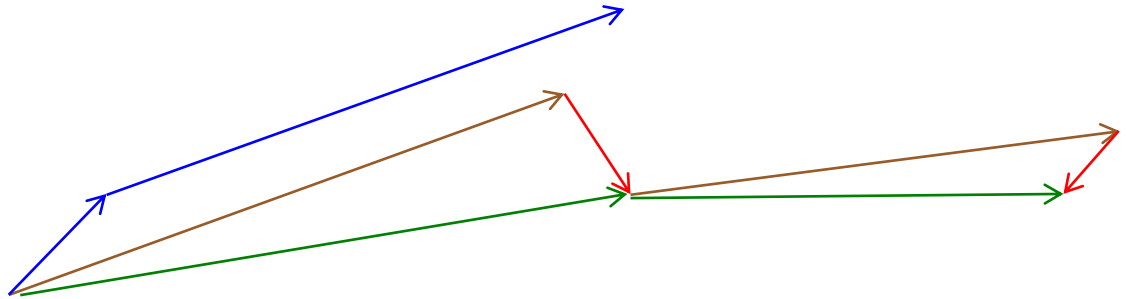
Nesterov Momentum

- 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!

Nesterov Momentum

- 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.

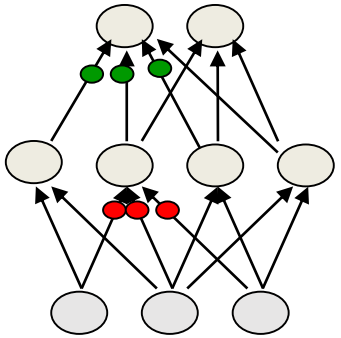


brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Adaptive Learning Rates

- 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.
- So use a global learning rate (set by hand) multiplied by an appropriate local gain that is determined empirically for each weight.



Gradients can get very small in the early layers of very deep nets.

The fan-in often varies widely between layers.

Determining Individual Rates

- Start with a local gain of 1 for every weight.
- Increase the local gain if the gradient for that weight does not change sign.
- Use small additive increases and multiplicative decreases (for mini-batch)
  - This 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 δ half the time and decrease by times $1-\delta$ half the time.

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

if $\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) > 0$

$$g_{ij}(t) = g_{ij}(t-1) + .05$$

else

$$g_{ij}(t) = g_{ij}(t-1) * .95$$

rmsprop

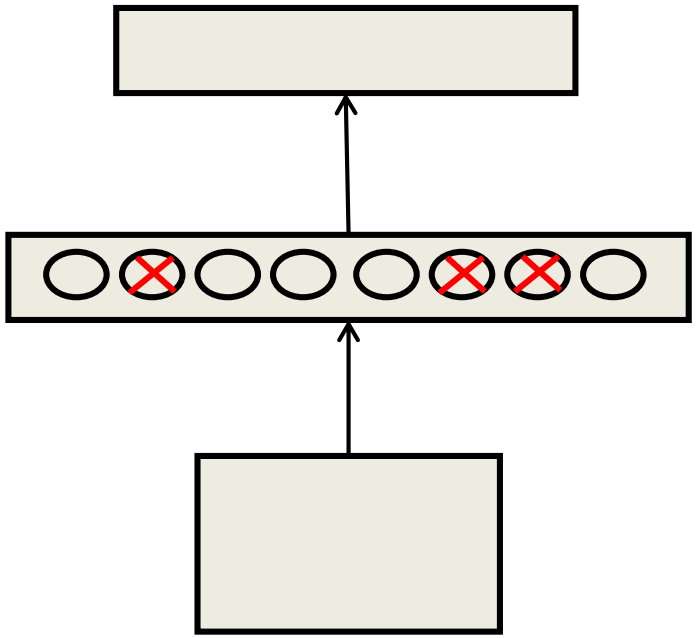
- Keep a moving average of the squared gradient for each weight:

$$MeanSquare(w,t)$$
$$= .9MeanSquare(w,t-1) + .1\left(\frac{\partial E}{\partial w}(t)\right)^2$$

- Divide the gradient by $\sqrt{MeanSquare(w,t)}$

Dropout Regularization

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
  - All architectures share weights.



Dropout as a form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
- The sharing of the weights means that every model is very strongly constrained
  - The weights must work for all the models that use them
  - This regularizes the weights and prevents overfitting

Dropout at test time

- Use all of the hidden units, but to halve their outgoing weights.
  - This exactly computes the geometric mean of the predictions of all 2^H models for a single-layer network
- Could also average some of the 2^H models