COMP 551 – Applied Machine Learning
Lecture 15: Neural Networks (cont'd)

Instructor: Ryan Lowe (ryan.lowe@cs.mcgill.ca)

Slides mostly by: Joelle Pineau

Class web page: *www.cs.mcgill.ca/~jpineau/comp551*

Announcements

- Project 3 due today!!!
  - Submit online through MyCourses, **and** submit a hard copy

- Slight update to Lecture 14 slides online, to clarify notation for backprop

- This lecture’s slides (and all future lectures) will be posted before class

Recap: Gradient-descent for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

Note: *j* here is any node in the hidden layer

$\partial J(w)/\partial \sigma = -(y-o_{N+H+1})$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y - o_{N+H+1}) o_{N+H+1} (1 - o_{N+H+1}) x_{N+H+1,j}$$

Recap: Gradient descent for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -\boxed{(y - o_{N+H+1}) o_{N+H+1} (1 - o_{N+H+1})} x_{N+H+1,j}$$

- Hence, we can write: $$\frac{\partial J}{\partial w_{N+H+1,j}} = \boxed{-\delta_{N+H+1}} x_{N+H+1,j}$$

where:

$$\delta_{N+H+1} = (y - o_{N+H+1}) o_{N+H+1} (1 - o_{N+H+1})$$

Gradient-descent update for **hidden** node

- The derivative wrt the other weights, $w_{l,j}$ where $j = 1, \ldots, N$ and $l = N+1, \ldots, N+H$ can be computed using chain rule:

$$\begin{aligned}\frac{\partial J}{\partial w_{l,j}} &= -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1}) \\ &\quad \cdot \frac{\partial}{\partial w_{l,j}}(\mathbf{w}_{N+H+1} \cdot \mathbf{x}_{N+H+1}) \\ &= -\delta_{N+H+1} w_{N+H+1,l} \frac{\partial}{\partial w_{l,j}} x_{N+H+1,l}\end{aligned}$$

Note: now j is any node in the **input** layer, and l is any node in the hidden layer

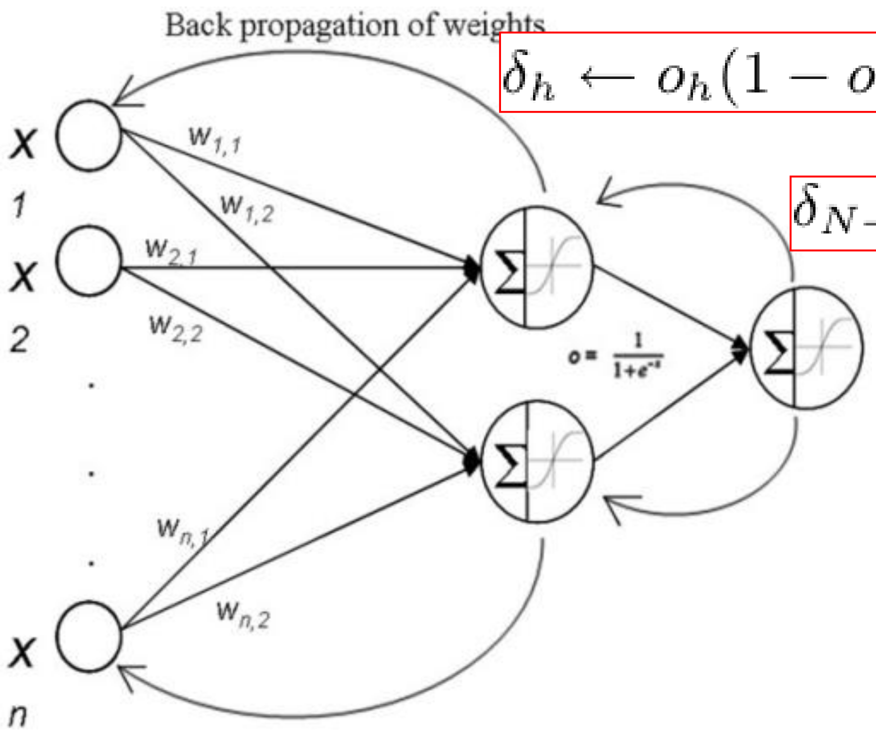
- Recall that $x_{N+H+1,l} = o_l$. Hence we have:

$$\frac{\partial}{\partial w_{l,j}} x_{N+H+1,l} = o_l(1 - o_l)x_{l,j}$$

- Putting these together and using similar notation as before:

$$\frac{\partial J}{\partial w_{l,j}} = -o_l(1 - o_l)\delta_{N+H+1} w_{N+H+1,l} x_{l,j} = -\delta_l x_{l,j}$$

Recap: Gradient descent for **hidden** node



$$\delta_h \leftarrow o_h(1 - o_h) w_{N+H+1,h} \delta_{N+H+1}$$

$$\delta_{N+H+1} \leftarrow o(1 - o)(y - o)$$

Note: now *h* is any node in the hidden layer

Image from: http://openi.nlm.nih.gov/detailedresult.php?img=2716495_bcr2257-1&req=4

Recap: SGD for LMS loss

- Initialize all weights to small random numbers.

Initialization

- Repeat until convergence:
  - Pick a training example.
  - Feed example through network to compute output $o = o_{N+H+1}$.

Forward pass

  - For the output unit, compute the correction:

$$\delta_{N+H+1} \leftarrow o(1-o)(y-o)$$

  - For each hidden unit *h*, compute its share of the correction:

$$\delta_h \leftarrow o_h(1-o_h) w_{N+H+1,h} \delta_{N+H+1}$$

Backpro-pagation

  - Update each network weight:

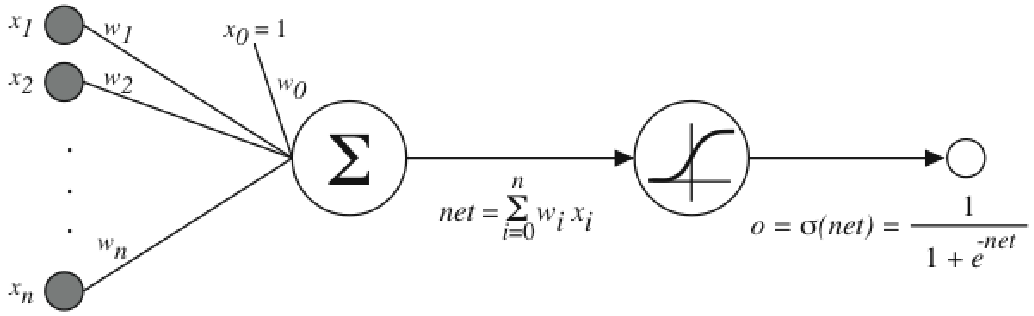
$$w_{h,i} \leftarrow w_{h,i} + \alpha_{h,i} \delta_h x_{h,i}$$

Gradient descent

Additional backprop resources

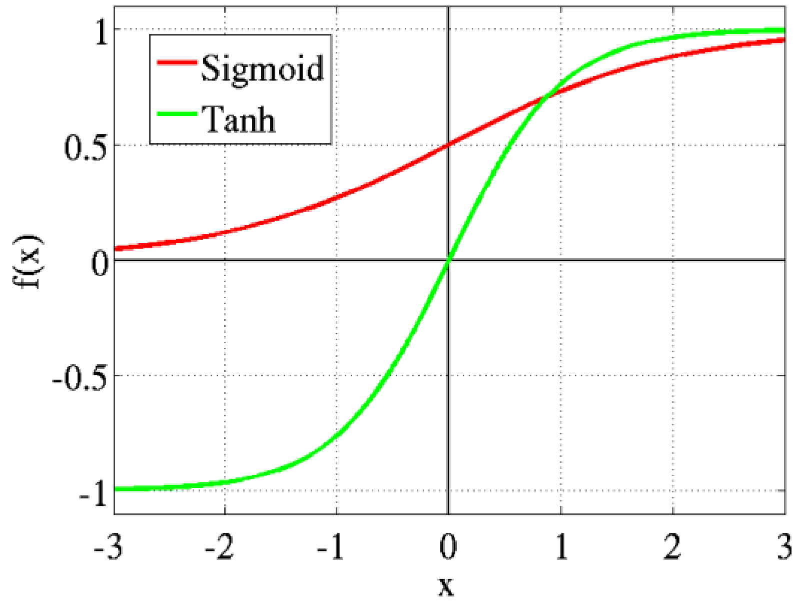
- "Neural networks and deep learning" textbook, Chapter 2, by Michael Neilson:
  - http://neuralnetworksanddeeplearning.com/chap2.html
  - Detailed, thorough look at backprop (with different notation)

Other activation functions: tanh



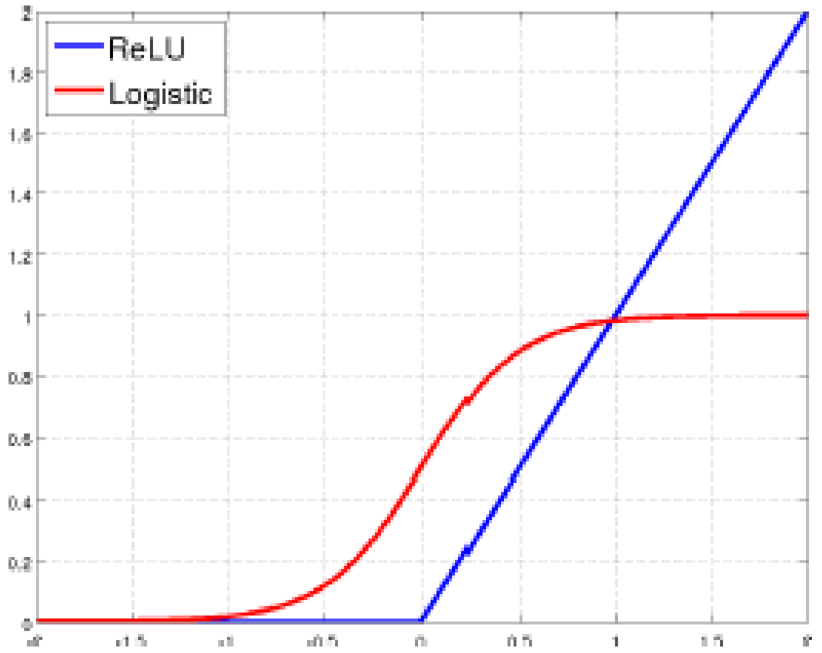
x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

tanh(z) $= (e^z - e^{-z}) / (e^z + e^{-z})$
$\partial\sigma(z)/\partial z = 1-\sigma(z)^2$

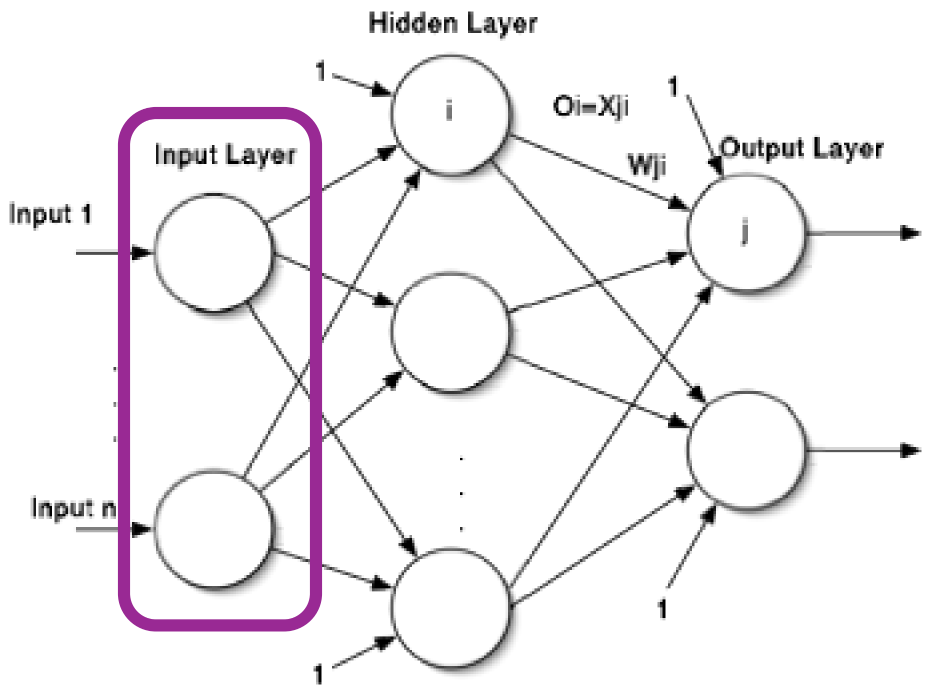


Rectified linear units (ReLU)

- Instead of using binary units, try *log(1+exp(Wx))*.
- Unit outputs linear function when input is positive, zero otherwise.
- **Most common unit** used in feed-forward and convolutional neural nets



How do we encode the input?



Encoding the input: Discrete inputs

- Discrete inputs with *k* possible values are often encoded using a *1-hot* or *1-of-k* encoding:
  - *k* input bits are associated with the variable (one for each possible value).
  - For any instance, all bits are *0* except the one corresponding to the value found in the data, which is set to *1*.
  - If the value is missing, all inputs are set to *0*.
  - **Example:** using the weather yesterday {sunny, raining, snowing} to predict the weather today

Encoding the input: Real-valued inputs

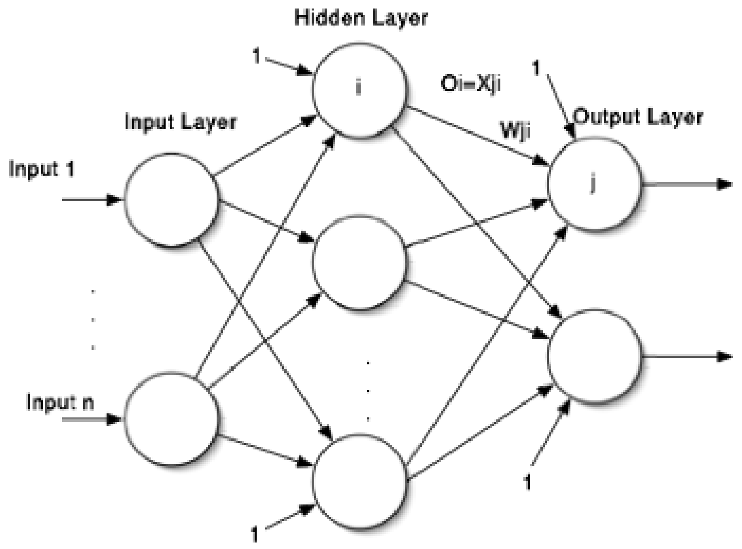
- **Example:** using the amount of rain (in mm) to predict the amount of rain today
- Important to scale the inputs, so they have a common, reasonable range
- Standard transformation: normalize the data
  - To get mean=0, variance=1, subtract the mean and divide by the standard deviation
  - Works well if the data is roughly normal, but bad if we have outliers.

Encoding the input: Real-valued inputs

- **Example:** using the amount of rain (in mm) to predict the amount of rain today
- Important to scale the inputs, so they have a common, reasonable range
- Standard transformation: **normalize the data**
  - To get mean=0, variance=1, subtract the mean and divide by the standard deviation
  - Works well if the data is roughly normal, but bad if we have outliers.
- Can also transform into discrete inputs:
  - *1-to-n encoding*: discretize the variable into a given number of intervals *n*.
  - *Thermometer encoding*: like *1-to-n* but if the variable falls in the *i*=th interval, all bits *1..i* are set to 1.

Encoding the output

- **Multi-class domains:**
  - Use a network with several output units: one per class
  - This allows *shared weights* at the hidden layers, compared to training multiple 1-vs-all classifiers.



- **Regression tasks:**
  - Use an output unit without a sigmoid function (i.e. just the weighted linear combination) to get full range of output values.

How do I choose the number of layers?

- Overfitting occurs if there are too many parameters compared to the amount of data available.
- Choosing the number of hidden units
  - Too few hidden units do not allow the concept to be learned.
  - Too many lead to slow learning and overfitting.
  - **There is no right answer.** Choose what works best on your validation set
- Choosing the number of layers
  - Always start with **one** hidden layer.
  - Add one at a time, see if solution improves on validation set.

What about local minima?

- If the learning rate is appropriate, SGD is guaranteed to converge to a **local minimum** of the cost function.
  - NOT the global minimum
- In practice, neural networks **can take a very long time to train** with SGD
- How often does SGD get stuck in local minima for deep neural networks?

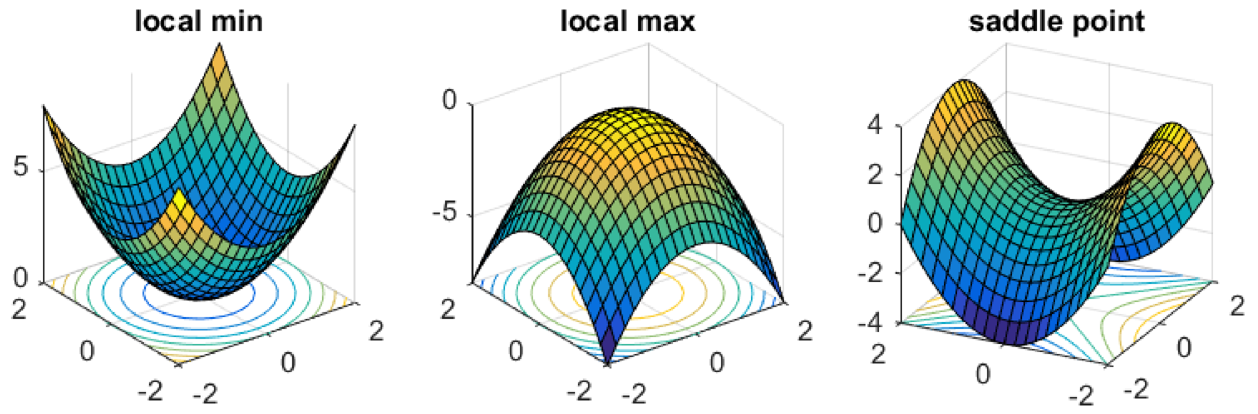
What about local minima?

*“We conjecture that […] SGD converges to the band of low critical points, and that **all critical points found there are local minima of high quality measured by the test error.** This emphasizes a major difference between large- and small-size networks where for the latter poor quality local minima have non-zero probability of being recovered. Finally, we prove that recovering the global minimum becomes harder as the network size increases and that it is in practice irrelevant as global minimum often leads to overfitting.”*

- “The Loss Surfaces of Multilayer Networks”, Choromanska et al., 2015

- **TL;DR:** for big networks, SGD almost always converges to local minima close to the global minima
- <u>Question:</u> then why can SGD seem to converge to poor solutions, and why does it take so long?

Saddle points



*"We argue that **a profound difficulty originates from the proliferation of saddle points, not local minima, especially in high dimensional problems**. Such saddle points are surrounded by high error plateaus that can dramatically slow down learning, and give the illusory impression of the existence of a local minimum."*

- "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", Dauphin et al., 2014

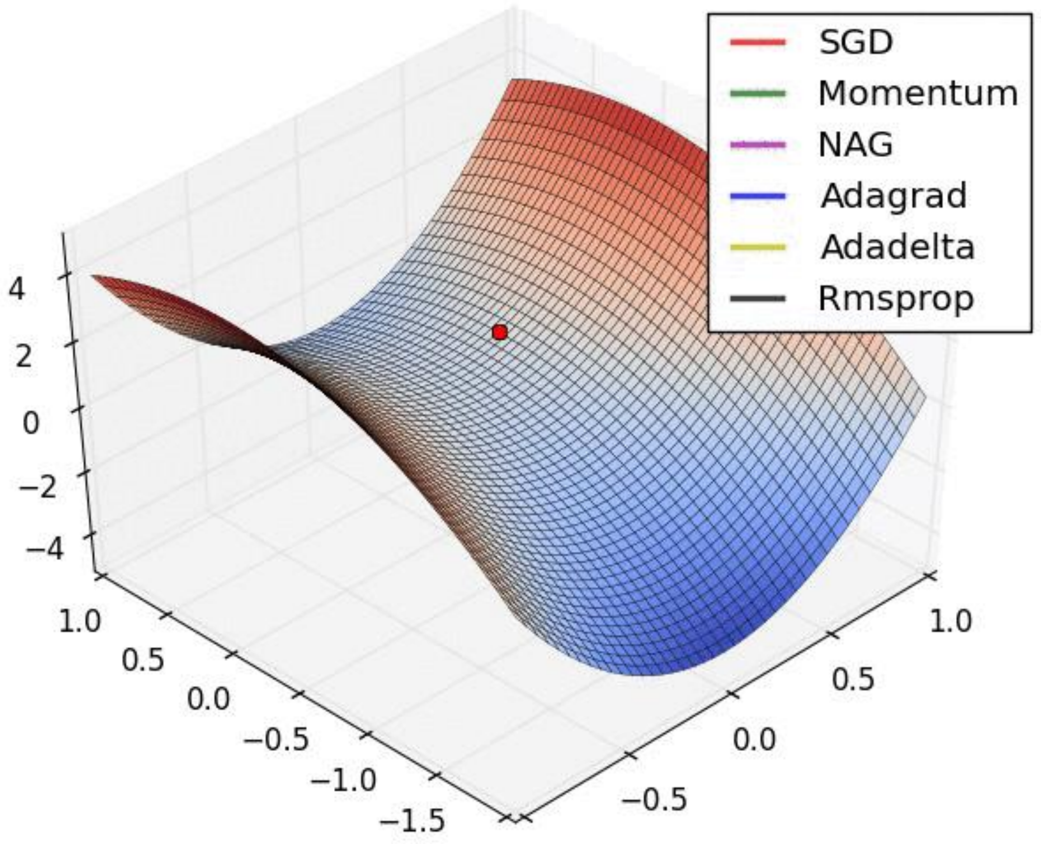
Image from offconvex.org

Saddle points

- Why are there more saddle points than local minima?
- **Intuition:** in high dimensions, a local minima means that *every direction that you go*, the loss function increases
  - If the probability of the loss increasing in a single direction from a critical point is constant (say 0.5), this becomes **exponentially less likely** as the dimension increases
- More likely that there are a small number of directions that decrease the loss (but these can be hard to find)

Saddle points

- SGD moves very slowly over saddle points



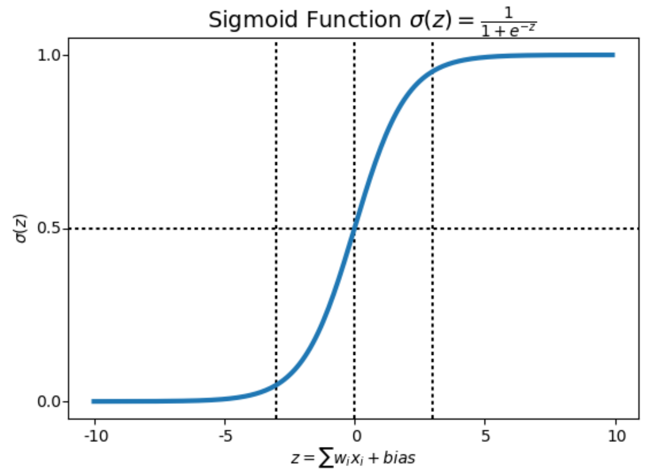
Gif from Alec Radford: imgur.com/a/Hqolp

Neural network optimization

- Saddle points are only part of the answer: optimization of neural networks still not well understood
- How do we avoid these problems?
  - Use **random restarts** = train multiple nets with different initial weights.
  - In practice, the solution found is often good (try a few parallel restarts).
  - Use modified (*accelerated*) versions of SGD (later this class)
  - Smart initialization methods

Parameter initialization

- Why not initialize all parameters to 0?
- **Symmetry breaking:** if all parameters are the same, gradients will be the same for all parameters in a layer
- So, want to initialize NN parameters **randomly**
- Don't want to initialize to large values -> unit saturation



Parameter initialization

- How do we do it?
- **Glorot initialization:** at layer k, sample each weight from a uniform distribution *U[-b, b]*, where [1]:

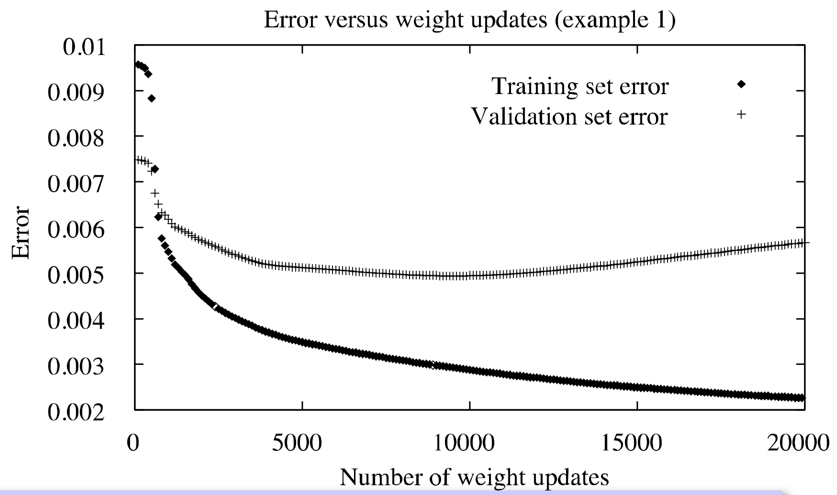
  b = sqrt(6) / sqrt(H_k + H_{k-1})

- *H_k* is the number of hidden units at layer k
- Not an exact science, many other methods possible

[1] See "*Understanding the difficulty of training deep feed-forward neural networks*", Glorot & Bengio, 2010, for more info.

Overfitting

- In neural networks, traditional overfitting occurs when the network is trained for too long
- Additional problem: units can become 'saturated' when weights take on large magnitudes, can be very slow to reverse
- Use validation set to decide when to stop training.
  - Training horizon is a hyper-parameter.
- Regularization is also effective.



Regularization in neural networks

- Incorporate an L2 penalty: $J(w) = 0.5(y-h_w(x))^2 + \boldsymbol{\lambda w^T w}$
  - Select λ with cross-validation.

- Can also use different penalties λ_1 , λ_2 for each layer.
  - Can be interpreted as a Bayesian (Gaussian) prior over weights.

- Next class: other methods (e.g. dropout, batch normalization) can act as a regularizer instead

Choosing the learning rate

- Backprop is **very sensitive** to the choice of learning rate.
  - Too large ⇒ divergence.
  - Too small ⇒ VERY slow learning.
  - The learning rate also influences the ability to escape critical points.
- Very often, different learning rates are used for units in different layers. Hard to tune by hand!
- **Heuristic**: Track performance on validation set, when it stabilizes, divide learning rate by 2.

Optimization

- SGD can be very slow. How can we speed it up?

- One way: add **momentum**

- <u>Analogy:</u> SGD is a person walking down a hill. Momentum is a ball rolling down a hill: added inertia smooths out oscillations, increases speed of convergence

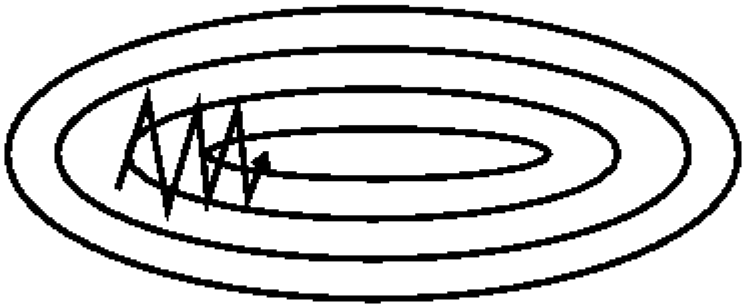
Adding momentum

- Add a fraction of the previous gradient(s). Instead of:

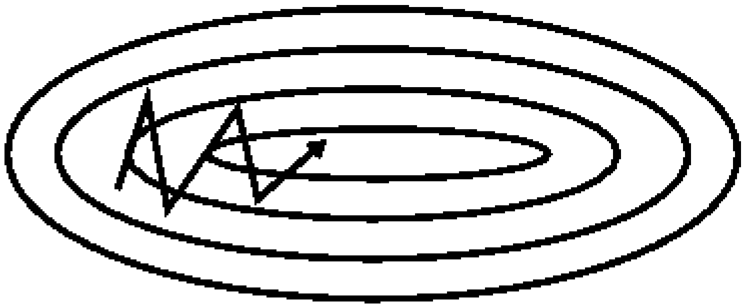
$$\Delta w_{ij} \leftarrow \alpha_{ij}\delta_j x_{ij}$$

We do: $$\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} + \beta \Delta w_{ij}(t-1)$$

  – The second term is called momentum



Without momentum



With momentum

Image source: HSE Coursera course

Adding momentum

- On the t-th training sample, instead of the update:

$$\Delta w_{ij} \leftarrow \alpha_{ij}\delta_j x_{ij}$$

We do: $\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} + \beta \Delta w_{ij}(t-1)$

Advantages:

- Easy to pass small local minima/ critical points.
- Keeps the weights moving in areas where the error is flat.
- Increases the speed where the gradient stays unchanged.

Disadvantages:

- With too much momentum, it can get out of a global minimum!
- One more parameter to tune, and more chances of divergence.

Adaptive learning rates

- Can do other things, e.g. calculating an adaptive learning rate per parameter.
- Want to learn slowly for frequent features, faster for rare but informative features
- How? Divide by the sum of squares of old gradients

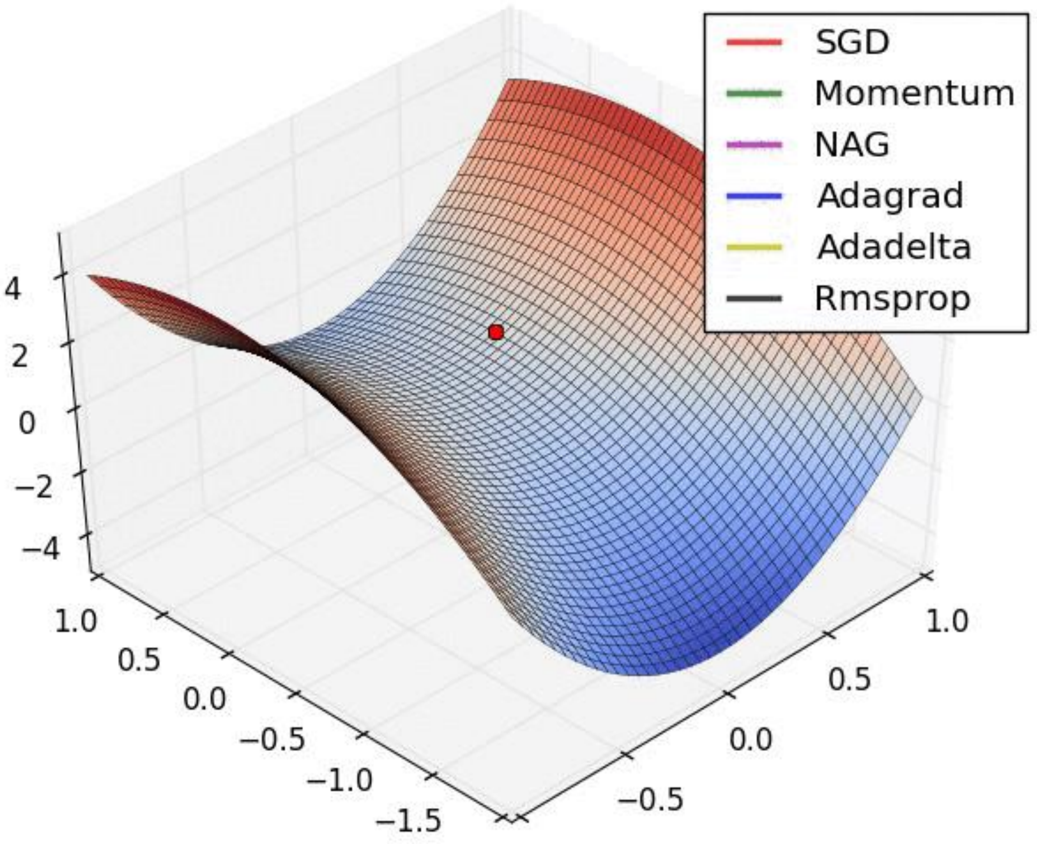
$$\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} \Big/ \gamma \sum_{i=1}^{t-1} \Delta w_{ij}(i)^2$$

- **<u>Intuition:</u>** If a parameter has been updated frequently, it will be divided by a large value, and result in a smaller update

See: Duchi, Hazan, Singer (2011) *Adaptive subgradient methods for online learning and stochastic optimization*. JMLR.

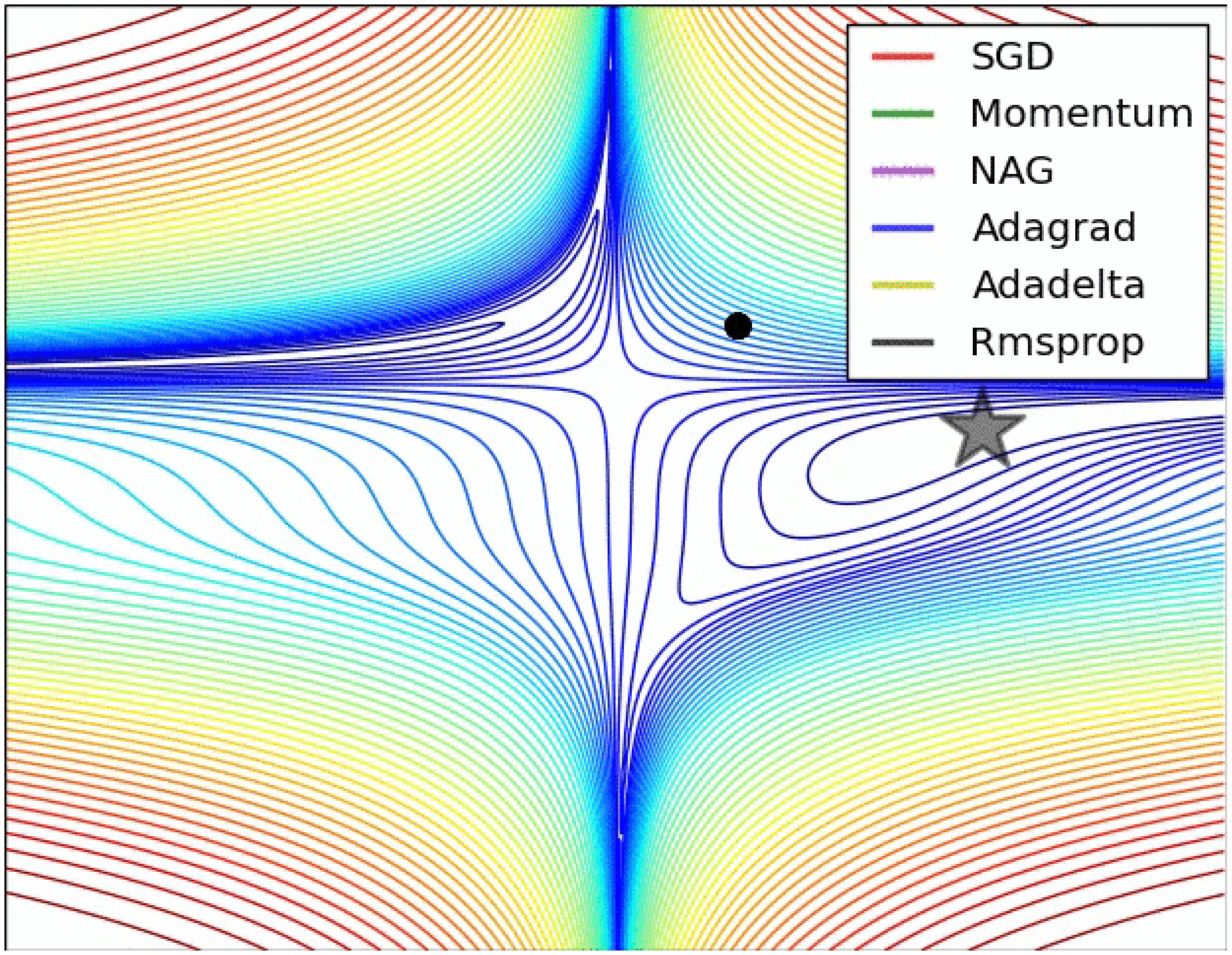
Saddle points with improved optimization

- Adding momentum/ adaptive LRs allows us to escape saddle points more easily



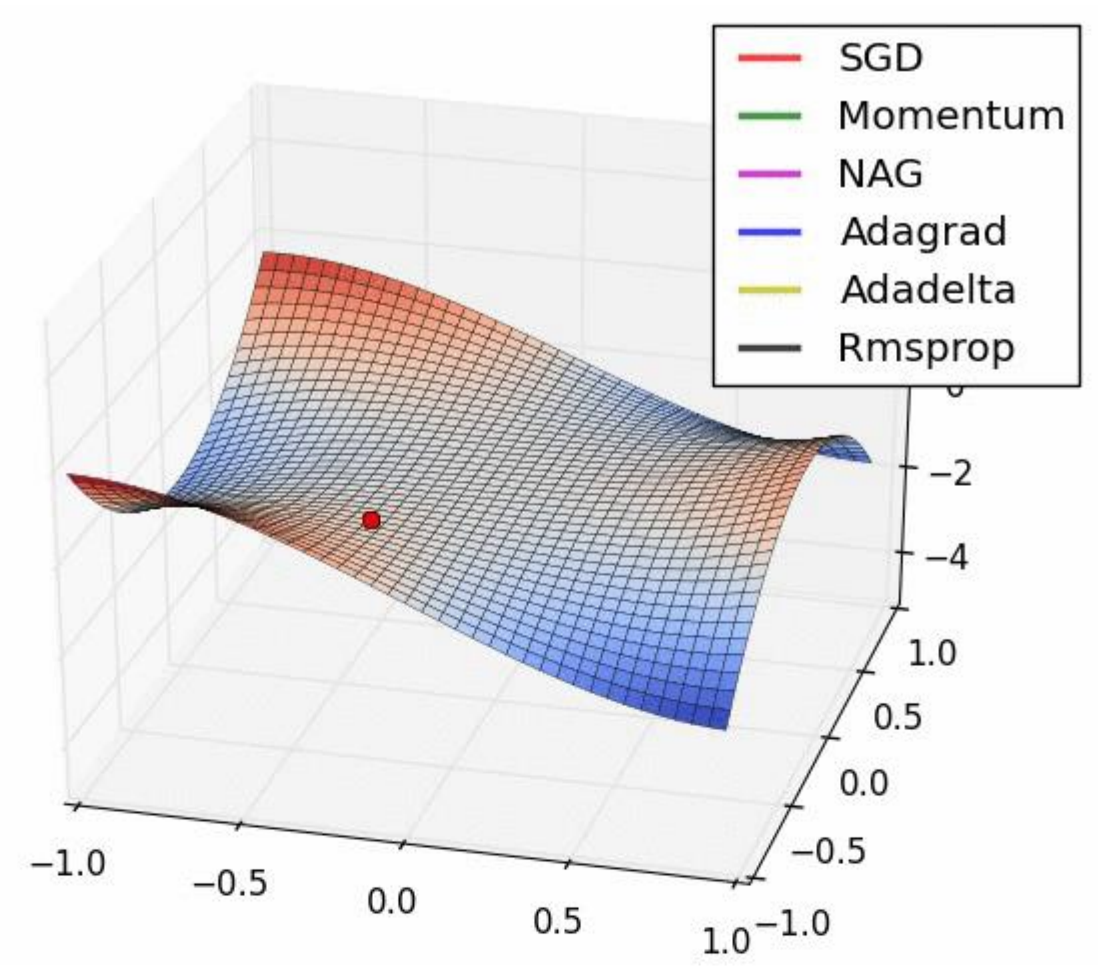
Gif from Alec Radford: imgur.com/a/Hqolp

Saddle points with improved optimization



Gif from Alec Radford: imgur.com/a/Hqolp

Saddle points with improved optimization



Gif from Alec Radford: imgur.com/a/Hqolp

Optimization

- Can do both momentum, and adaptive learning rates
  - Leads to **ADAM** (Kingma & Ba, 2015), very popular in deep learning
- Story of why momentum works is slightly more complicated
  - See "Why Momentum Really Works", https://distill.pub/2017/momentum/
- Other optimization methods: RMSProp, Adagrad, Adadelta, Nesterov Momentum

More application-specific tricks

- If there is too little data, it can be **perturbed by random noise**; this helps escape local minima and gives more robust results.
  - In image classification and pattern recognition tasks, extra data can be generated, e.g., by applying transformations that make sense.

More application-specific tricks

- If there is too little data, it can be **perturbed by random noise**; this helps escape local minima and gives more robust results.
  - In image classification and pattern recognition tasks, extra data can be generated, e.g., by applying transformations that make sense.
- **Weight sharing** can be used to indicate parameters that should have the same value based on prior knowledge.
  - Each update is computed separately using backpropagation, then the tied parameters are updated with an average.

When to consider using NNs

- **Input is high-dimensional** (e.g. raw sensor input).
- Output is discrete or real valued, or a vector of values.
- Possibly noisy data, or a large quantity of data.
- Training time is not important.
- Form of target function is unknown and complex.
- Human readability of result is not important.
- The computation of the output based on the input has to be fast.

Several applications

- Speech recognition and synthesis.
- Natural language understanding.
- Image classification, digit recognition.
- Financial prediction.
- Game playing strategies.
- Robotics.
- …..

In recent years, many state-of-the-art results obtained using **Deep Learning**.

Final notes

- What you should know:
  - Definition / components of neural networks.
  - Training by backpropagation.
  - Overfitting (and how to avoid it).
  - Saddle points and local optima for neural networks
  - When and how to use NNs.
  - Some strategies for successful application of NNs (optimization methods, initialization, activation functions, etc.).

Additional info

- Additional information about neural networks:

  Video & slides from the Montreal Deep Learning Summer School:

  http://videolectures.net/deeplearning2017_larochelle_neural_networks/

  https://drive.google.com/file/d/0ByUKRdiCDK7-c2s2RjBiSms2UzA/view?usp=drive_web

  https://drive.google.com/file/d/0ByUKRdiCDK7-UXB1R1ZpX082MEk/view?usp=drive_web

- "Neural networks and deep learning" textbook, Chapter 2, by Michael Neilson:
  - http://neuralnetworksanddeeplearning.com/chap2.html
- Overview of gradient descent optimization methods
  - http://ruder.io/optimizing-gradient-descent/