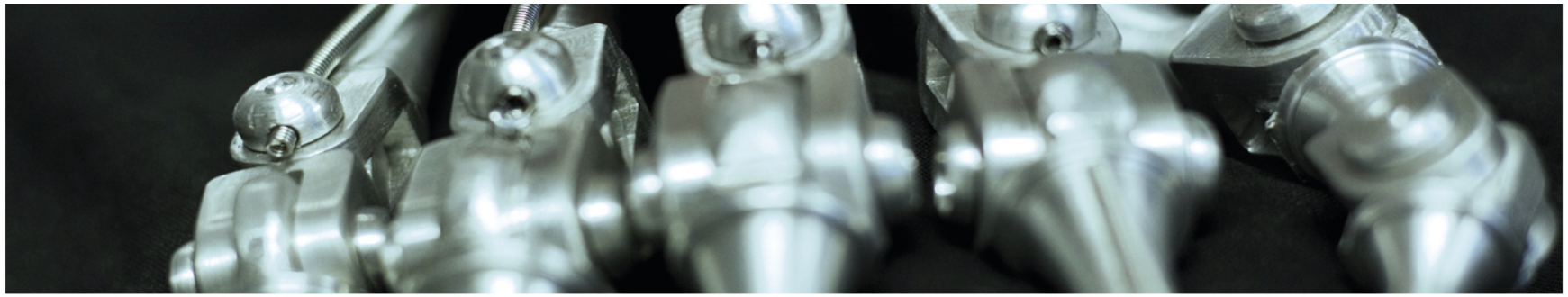


UiO : Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Lecture : Training a neural net – part I

Initialization, activations, normalizations and other practical details

Anne Solberg

February 28, 2018

Reading material

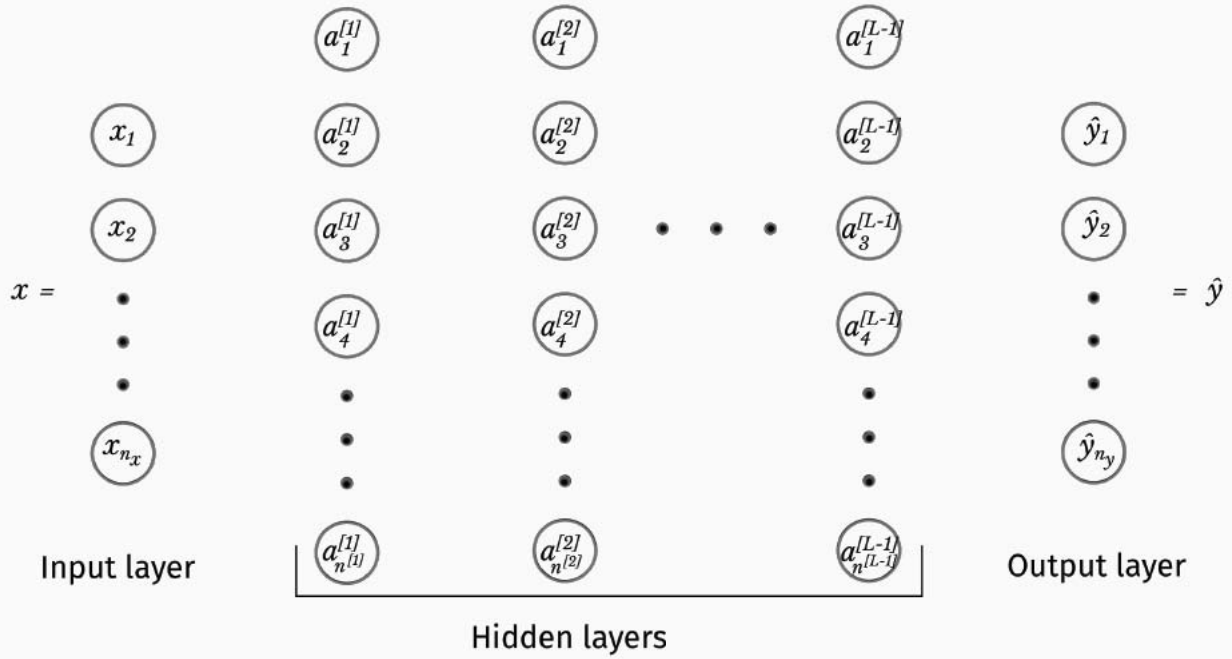
- Reading material:
  - http://cs231n.github.io/neural_networks-2
    - Data scaling, weight initialization, batch normalization
  - http://cs231n.github.io/neural-networks-3/
    - Monitoring the loss, parameter update schemes,
  - Deep Learning 6.2.2 and 6.3 on activation functions
  - Deep Learning 8.7.1 on Batch normalization

Today

- Recap of the optimization problem
- Activation functions
- Mini-batch gradient descent
- Data preprocessing
- Weight initialization
- Batch normalization
- Weight update schemes

Recap of the optimization problem

Recap: forward propagation



$$a_k^{[l]} = g\left(\sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}\right)$$

Recap:
Update weights using gradient descent

We want to find values for our weights and biases

$$w_{jk}^{[l]} \leftarrow w_{jk}^{[l]} - \lambda \frac{\partial \mathcal{C}}{\partial w_{jk}^{[l]}}$$

$$b_{k}^{[l]} \leftarrow b_{k}^{[l]} - \lambda \frac{\partial \mathcal{C}}{\partial b_{k}^{[l]}}$$

for all

$$\begin{cases} j & = 1, \ldots, n^{[l-1]} \\ k & = 1, \ldots, n^{[l]} \\ l & = 1, \ldots, L \end{cases}$$

This is done with the so-called *backpropagation algorithm*.

Recap: cross entropy cost

Cost function over a minibatch of samples

$$\mathcal{C}(\Theta, \Omega^y_{\text{train}}, \Omega^x_{\text{train}}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}.$$

Cross-entropy loss for a single sample

$$\mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = -\sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}.$$

Recap: backprop for a single sample

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_k^{[l]}} a_j^{[l-1]}, \quad l = 1, \ldots, L. \tag{21a}$$

$$\frac{\partial \mathcal{L}}{\partial b_k^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_k^{[l]}}, \quad l = 1, \ldots, L. \tag{21b}$$

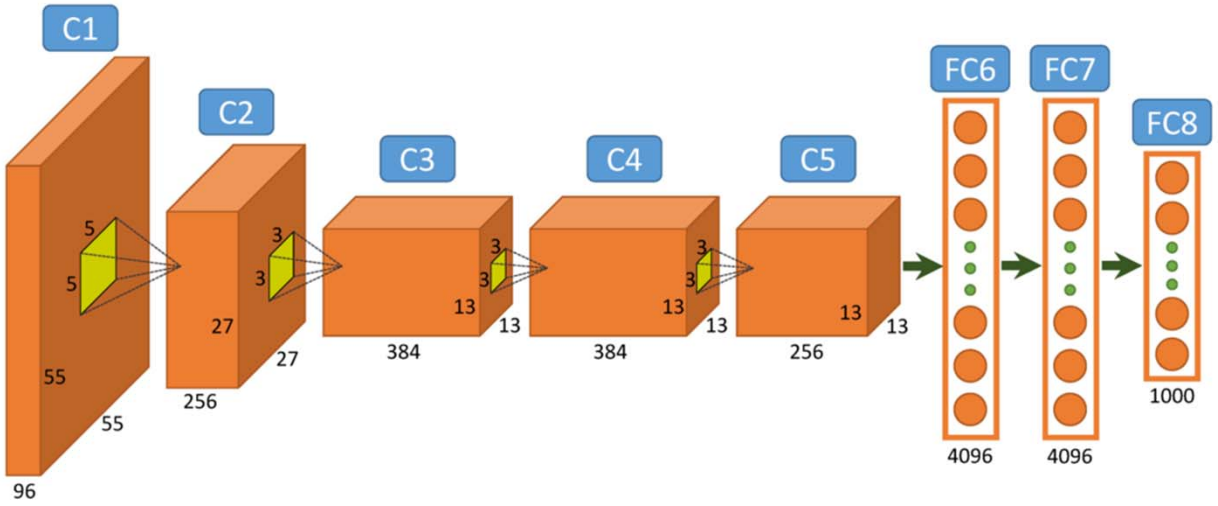
$$\frac{\partial \mathcal{L}}{\partial z_k^{[l]}} = g'(z_k^{[l]}) \sum_{j=1}^{n^{[l+1]}} \frac{\partial \mathcal{L}}{\partial z_j^{[l+1]}} w_{kj}^{[l+1]}, \quad l = 1, \ldots, L-1 \tag{21c}$$

$$\frac{\partial \mathcal{L}}{\partial z_k^{[L]}} = \hat{y}_k - \tilde{y}_k. \tag{21d}$$

Note that

- Eqs. (21a)— (21c) are generally applicable
- Eq. (21d) assumes that $\mathcal{L}$ is the cross-entropy loss, and that $a^{[L]} = s(z^{[L]})$ with s as the softmax function.

Recap: convolutional networks



Layers:
Convolutional layers
Pooling layers
Fully-connected layers

Training: we still use backpropagation

Recap: Mini-batch SGD

- Loop:

1. Sample a batch of training data
2. Forward propagate it through the net to compute the loss/cost C
3. Backprop to calculate gradients with respect to all weights
4. Update the parameters using the gradient

$$\mathcal{C}_b = \frac{1}{m_b} \sum_{i=1}^{m_b} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}$$

$$\theta \leftarrow \theta - \lambda \nabla_\theta \mathcal{C}_b$$

Next: training a neural network in practise

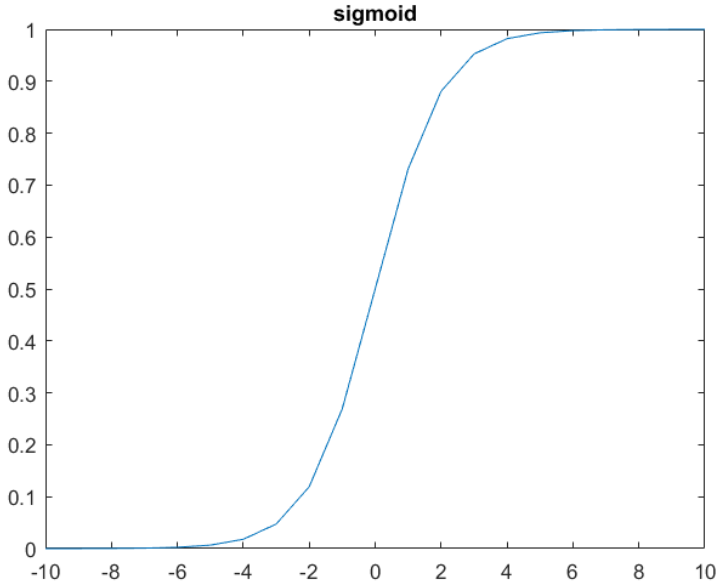
Where we are

- **Activation functions**
- Data preprocessing
- Weight initialization
- Batch normalization
- Weight update schemes
- Searching for the best parameters

Sigmoid activation

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

$$g'(z) = g(z)(1-g(z))$$



- Output between 0 and 1
- Historically popular
- Has some shortcomings

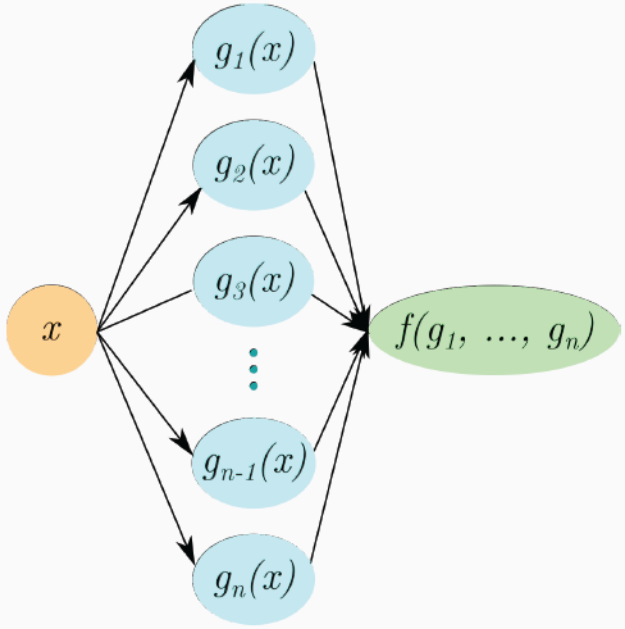
Remember: chain rule is the core of backpropagation – we need the derivative of C with respect to all W[l]

For a function f dependent on g which is dependent on x

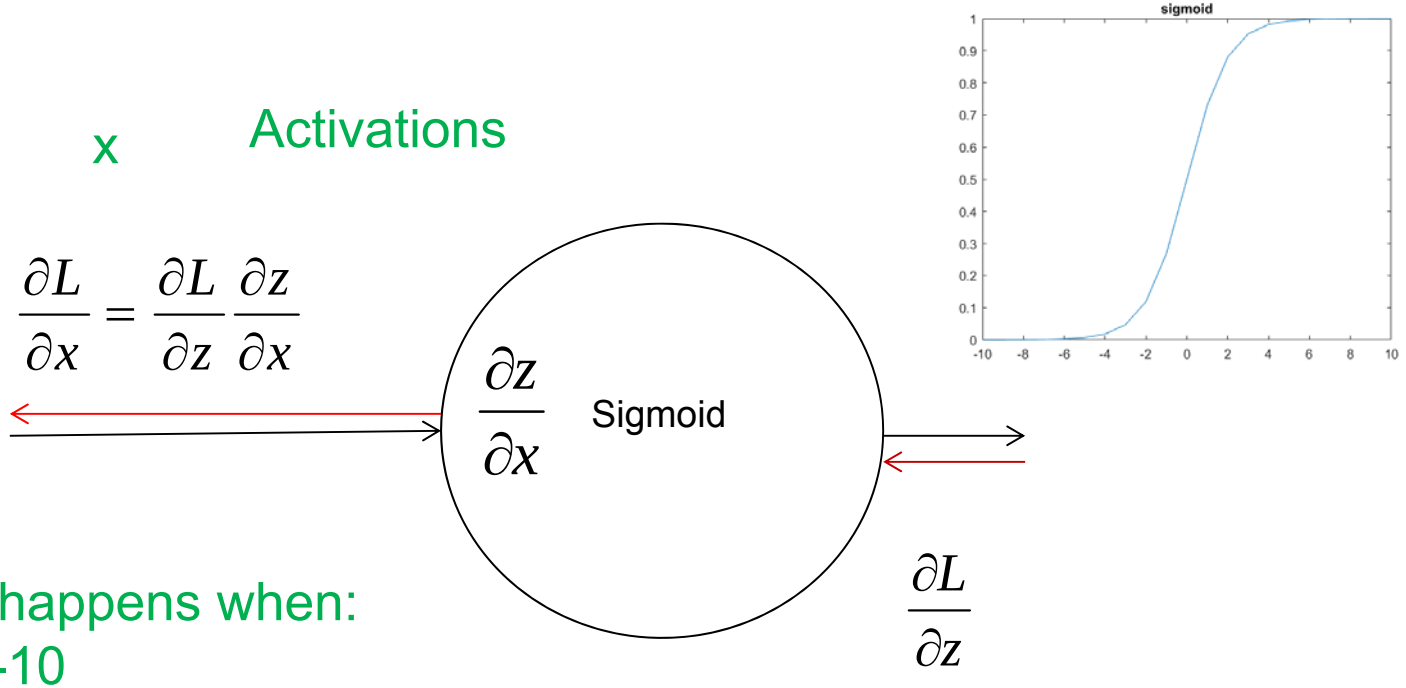
$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}$$

For a function f dependent on multiple $g_1, \ldots, g_n$, all which are dependent on x

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$$



Chain rule and gradients for a sigmoid node



What happens when:

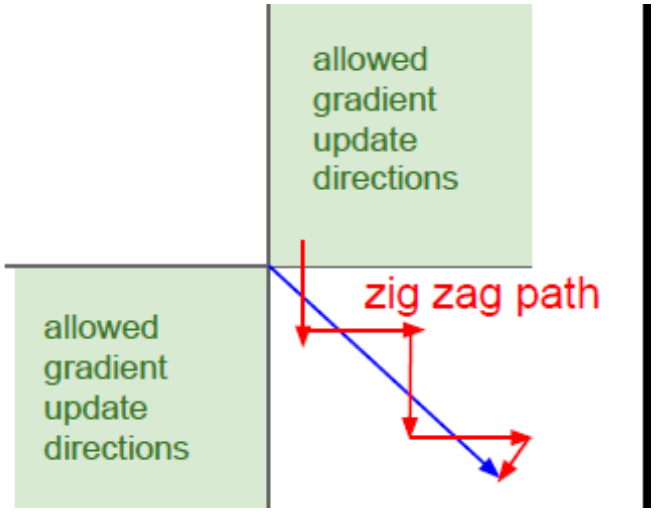
- x=-10
- x=0
- x=10

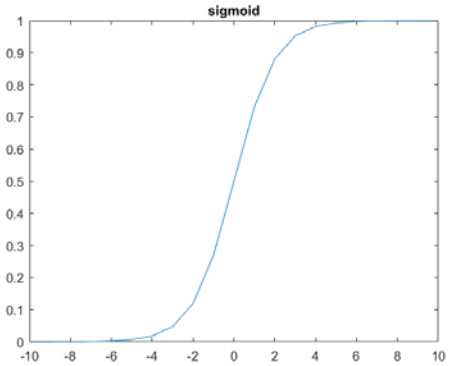
Sigmoid problems

1. Sigmoids kill gradients

What is the consequence of this?

Sigmoids are not zero-centered



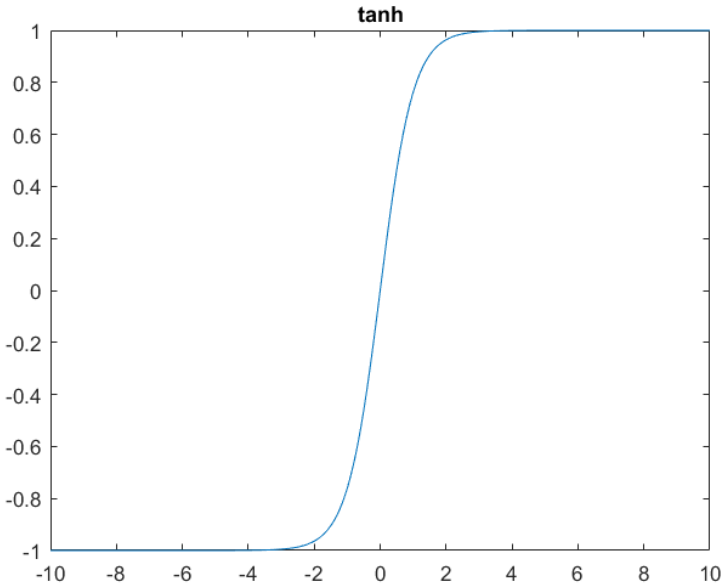


The mean is positive

Tanh activation

$$g(z) = \tanh(z)$$

- Scaled version of sigmoid
- Output between -1 and 1
- Zero-centered
- Saturates and kill gradients
- Preferred to sigmoid due to the zero-centering



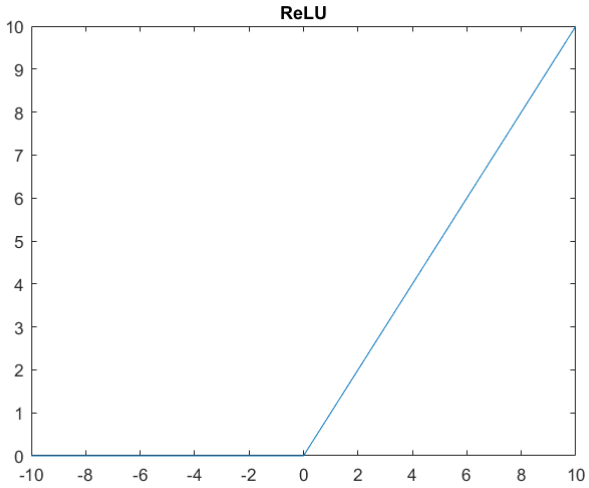
ReLU activation

$$\mathrm{ReLU}(z) = \max(z,0)$$

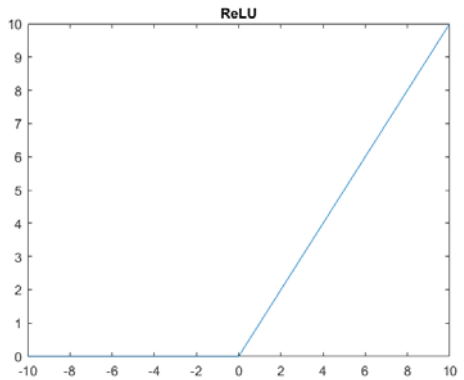
$$\text{Derivative of ReLU}: \max(z,0) = 1 \text{ if } z > 0$$

$$\text{and } 0 \text{ otherwise}$$

- Rectified Linear Unit



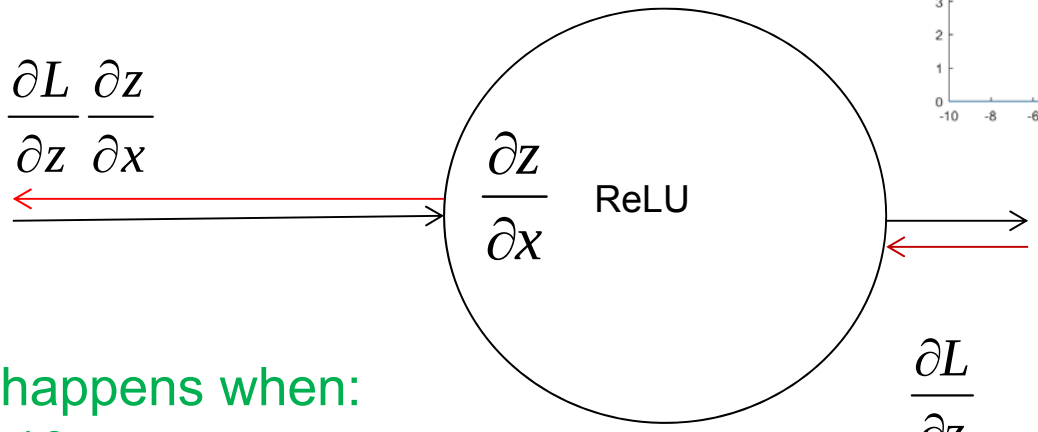
ReLU



x

Activations

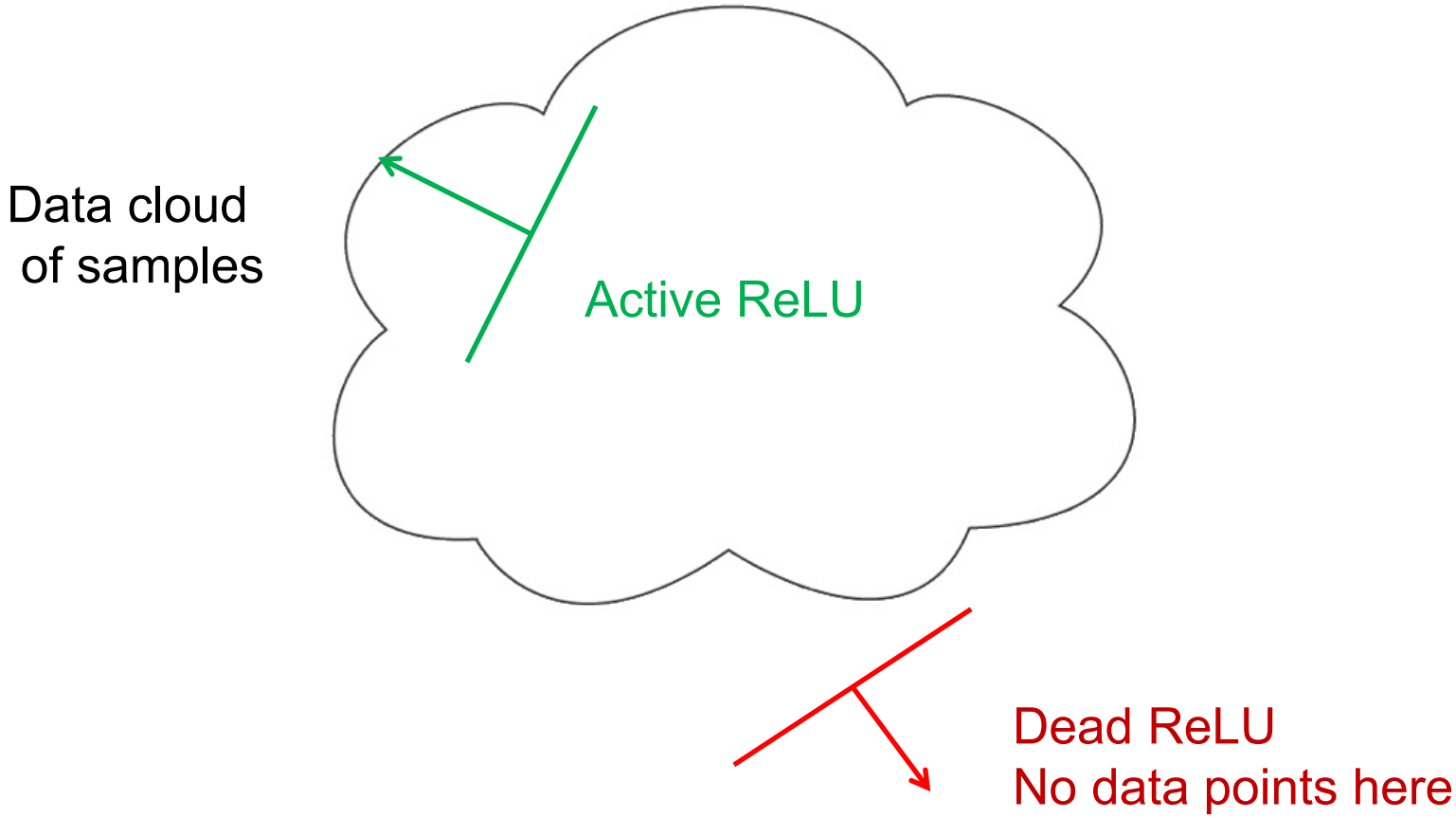
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$$



What happens when:

- x=-10
- x=0
- x=10

ReLU and dead neurons



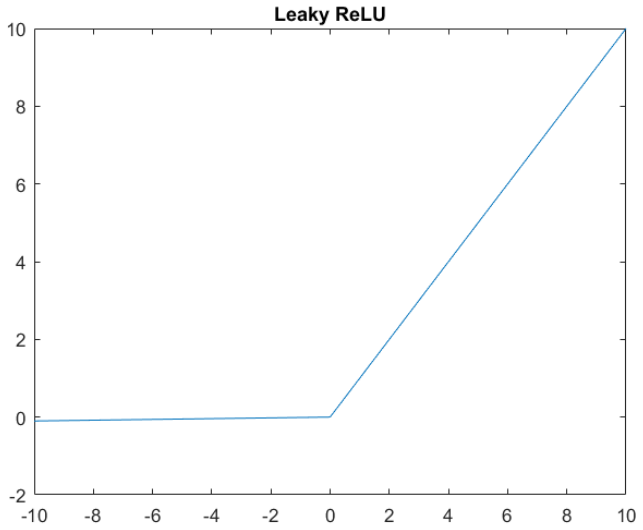
ReLU activation

- Does not saturate/kill gradients
- Fast to compute
- Converge fast
- Drawback: can sometimes ‘die’ during training and become inactive
  - If this happens, the gradients will be 0 from that point
  - Be careful with the learning rate

Currently: the best starting point recommendation

Leaky ReLU activation

$$\text{Leaky ReLU}(z) = \max(0.01z, z)$$



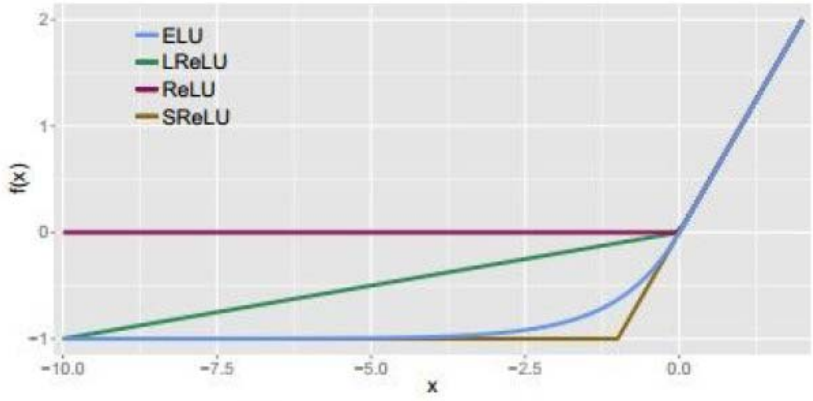
- Converge fast
- Will not die
- Results are not consistent that Leaky ReLU is better than ReLU

ELU activation

Exponential Linear Unit $(\mathrm{ELU})(z) = z, \quad z > 0$

- $\alpha(\exp(z) - 1)$

- Will not die
- Closer to zero-mean outputs
- Benefits of ReLU, but more expensive to compute
- Compared to Leaky Relu, the negative saturation adds some robustness to noise.
- Requires exp()



Maxout activation

$$\text{Maxout}(z) = \max(w_1 z + b_1, w_2 z + b_2)$$

-

- Here there are two weights for each node
- Can be seen as a generalization of ReLU/Leaky Relu
- Doubles the amount of parameters per node compared to ReLU.

Activation recommendations

- Start by using ReLU
- Monitor the training process, look or dead neurons.
  - Consider e.g. Leaky ReLU or Maxout if dead neurons seems to be an issue.
- Do not use Sigmoid

Where we are

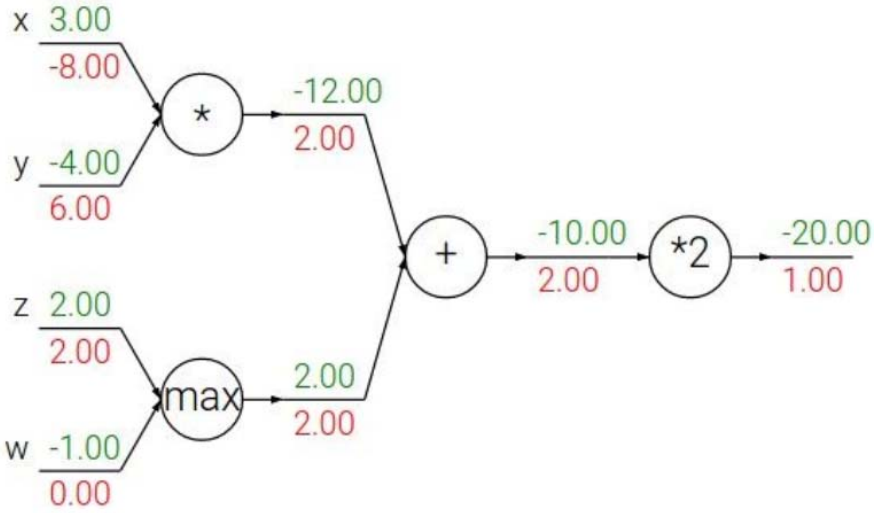
- Activation functions
- **Data preprocessing**
- Weight initialization
- Batch normalization
- Weight update schemes
- Searching for the best parameters

Patterns in backward flow

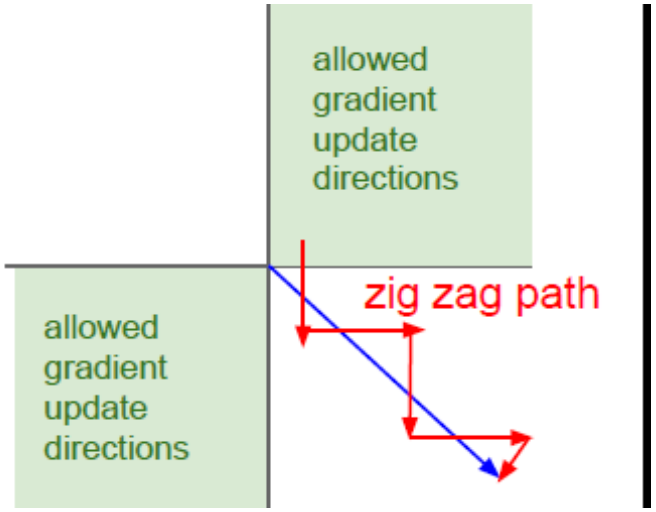
add gate: gradient distributor
max gate: gradient router
mul gate: be careful

f=x*y means that
df/dx=y and df/dy=x

Remark on multiplier gate:
If a gate get one large and one small input, backprop will use the big input to cause a large change on the small input, and vice versa. This is partly why feature scaling is important

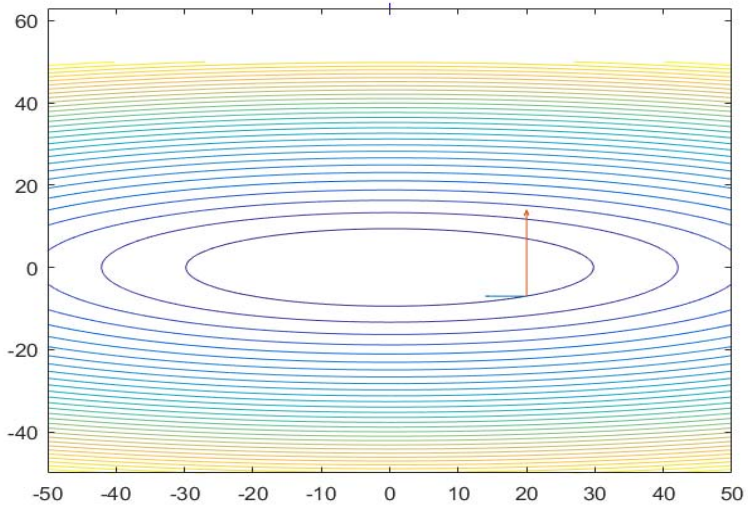


Data not zero-centered



Similar effect as sigmoid:
dynamics of the net change, slow convergence

Convergence of gradient descent



- Consider **features with different scaling.**
- The error surface is then locally like an ellipse.
- Does a gradient descent lead us fast in the direction we want?

Common normalization

- Standardize data to zero mean and unit variance
- Remark: STORE μ and σ because new data/test data must have the same normalization.

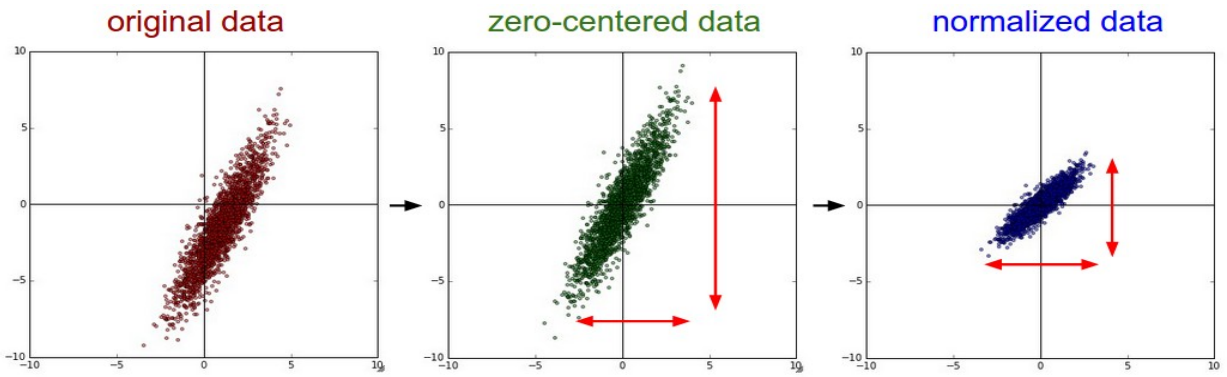
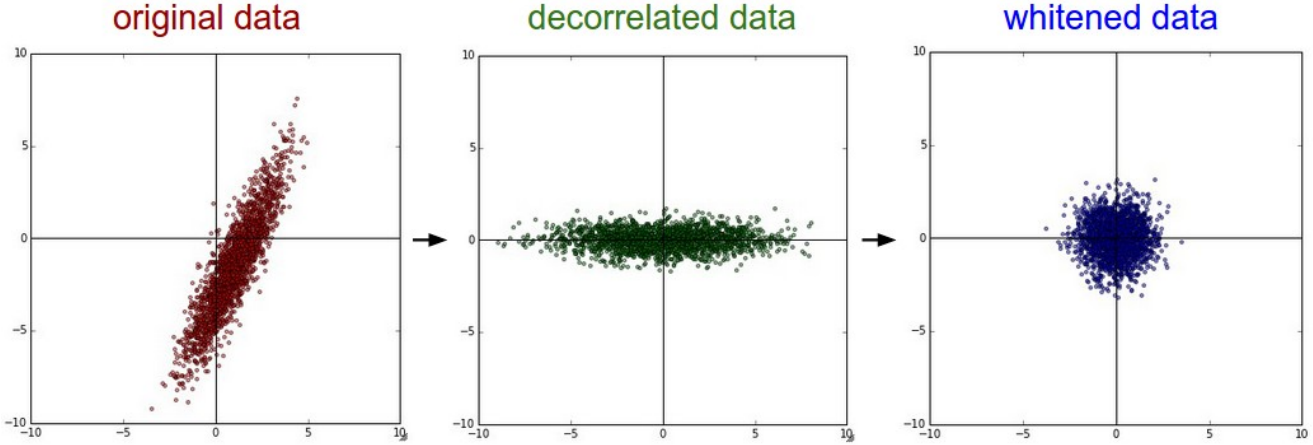


Figure from http://cs231n.github.io/neural-networks-2/

Consider whitening the data

- If features are highly correlated, principal component transform can be considered to whiten the data.
- Drawback: computationally heavy for image data,
  - Normally not used for image data
- Consider to use on other input types.



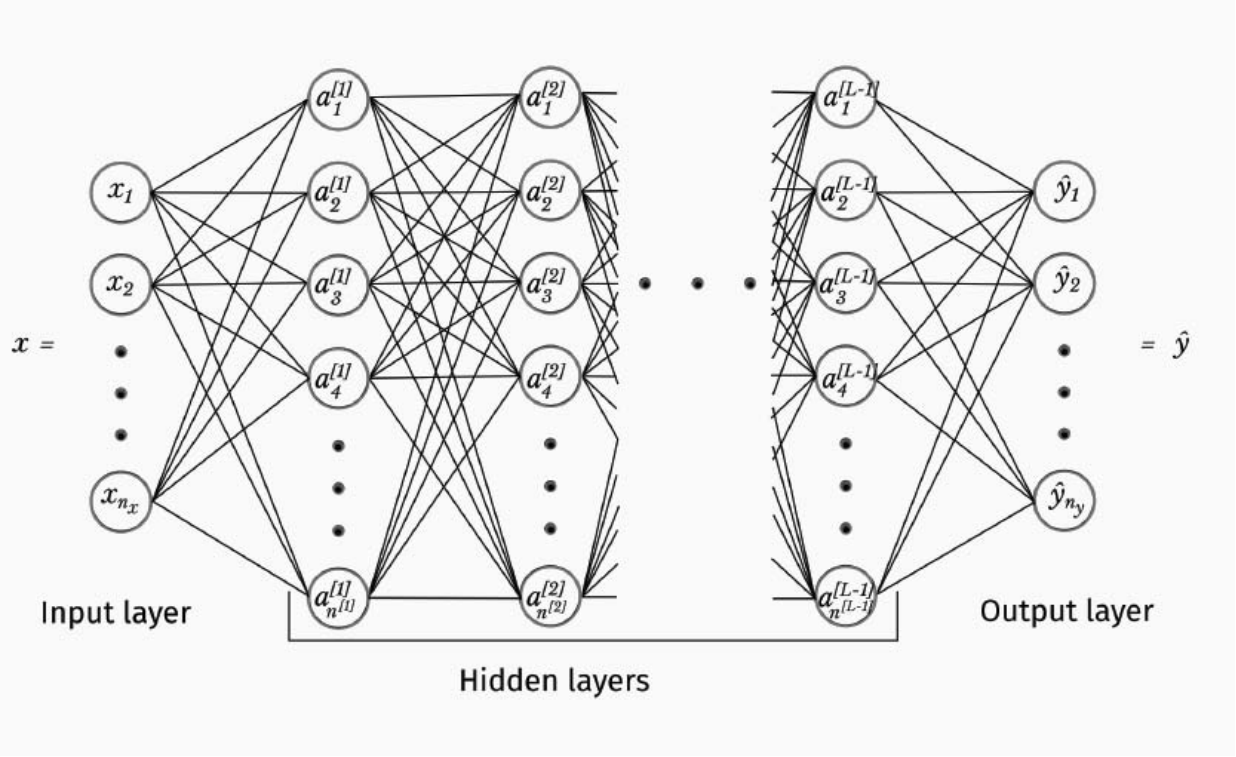
Common normalization for image data

- Consider e.g. CIFAR-10 image (32,32,3)
- Two alternatives:
  - Subtract the mean image
    - Keep track of a mean image of (32,32,3)
  - Subtract the mean of each channel (r,g,b…)
    - Keep track of the channel mean, 3 values for RGB.

Where we are

- Activation functions
- Data preprocessing
- **Weight initialization**
- Batch normalization
- Weight update schemes
- Searching for the best parameters

What if all weights are initialized with the same value?



What are the gradients during backpropagation?

Weight initialization – alternative 1

- Initialize weights to small random numbers
- W = 0.01*np.random.randn(D,H)
- Every node will have a different random value.

- Works OK for small networks, but not so good for deeper nets.
- Look at statistics for activations

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
#import time

%matplotlib inline
```

In [25]:
```
D = np.random.rand(1000,500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

In [36]:
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in range(len(hidden_layer_sizes)):
    X = D if i== 0 else Hs[i-1]
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out)*0.01
    H = np.dot(X,W)
    H = act[nonlinearities[i]](H)
    Hs[i] = H
```

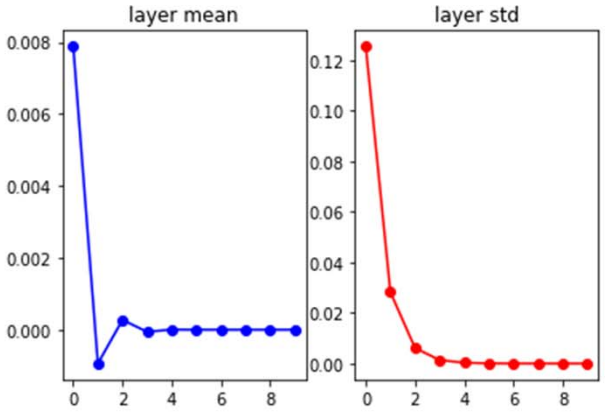
In [38]:
```
# look at distribution of each layer

print('input layer has mean %f and std %f' % (np.mean(D), np.std(D)))
layer_means = np.zeros(len(Hs))
layer_std = np.zeros(len(Hs))
for i in range(len(Hs)):
    layer_means[i] = np.mean(Hs[i])
    layer_std[i] = np.std(Hs[i])
    print ('hidden layer %d has mean %f and std %f' % (i+1, layer_means[i], layer_std[i]))

# Plot the means adn stds.
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_std, 'or-')
plt.title('layer std')

plt.figure()
plt.hist(H[0].ravel(), 30, range=[-0.5,0.5])
# Plot the raw distribution
plt.figure()
for i in range(len(Hs)):
    plt.figure()

    plt.title('Layer %d' %i)
    plt.hist(H[i].ravel(), 30, range=[-0.5,0.5])
```

Activation plots

```
input layer has mean 0.499225 and std 0.288535
hidden layer 1 has mean 0.007881 and std 0.125487
hidden layer 2 has mean -0.000940 and std 0.028207
hidden layer 3 has mean 0.000272 and std 0.006154
hidden layer 4 has mean -0.000058 and std 0.001346
hidden layer 5 has mean 0.000003 and std 0.000308
hidden layer 6 has mean -0.000000 and std 0.000066
hidden layer 7 has mean 0.000000 and std 0.000015
hidden layer 8 has mean -0.000000 and std 0.000003
hidden layer 9 has mean 0.000000 and std 0.000001
hidden layer 10 has mean 0.000000 and std 0.000000
```

click to scroll output; double click to hide

Out[39]: Text(0.5,1,'layer std')



In forward prop:
activations become 0!

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
#import time

%matplotlib inline
```

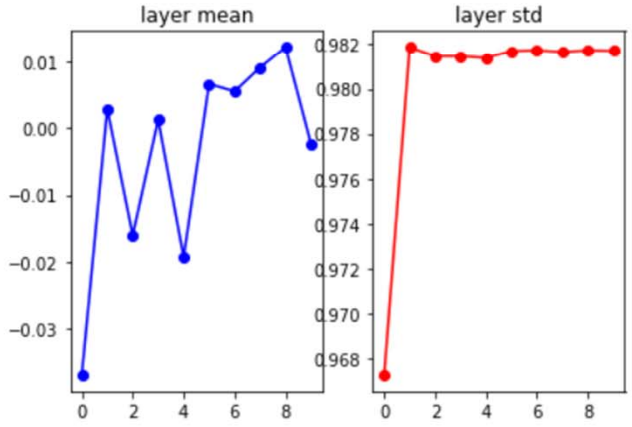
In [25]:
```
D = np.random.rand(1000,500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

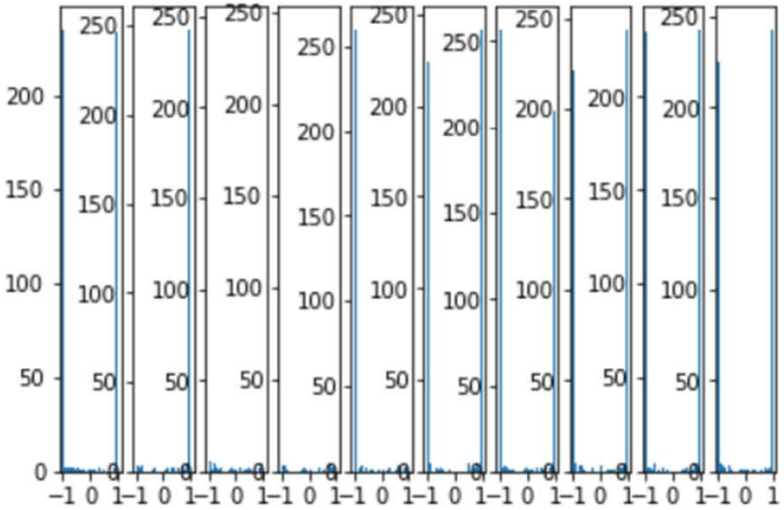
In [36]:
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in range(len(hidden_layer_sizes)):
    X = D if i== 0 else Hs[i-1]
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out)*0.01
    H = np.dot(X,W)
    H = act[nonlinearities[i]](H)
    Hs[i] = H
```

Now change the scaling to 1

With scaling 1 and tanh, the nodes are saturated to either -1 or +1
What happes to the gradient then?

```
input layer has mean 0.499941 and std 0.288825
hidden layer 1 has mean -0.036934 and std 0.967268
hidden layer 2 has mean 0.002872 and std 0.981866
hidden layer 3 has mean -0.016049 and std 0.981460
hidden layer 4 has mean 0.001268 and std 0.981459
hidden layer 5 has mean -0.019339 and std 0.981384
hidden layer 6 has mean 0.006637 and std 0.981697
hidden layer 7 has mean 0.005504 and std 0.981727
hidden layer 8 has mean 0.009093 and std 0.981660
hidden layer 9 has mean 0.012191 and std 0.981723
hidden layer 10 has mean -0.002316 and std 0.981707
```





Weight initialization – normalizing the variance.

- Consider a neuron with n inputs and $z = \sum_{i=1}^{n} w_i x_i$ (n is called fan-in)
- The variance of z is

$$Var(z) = Var(\sum_{i=1}^{n} w_i x_i)$$

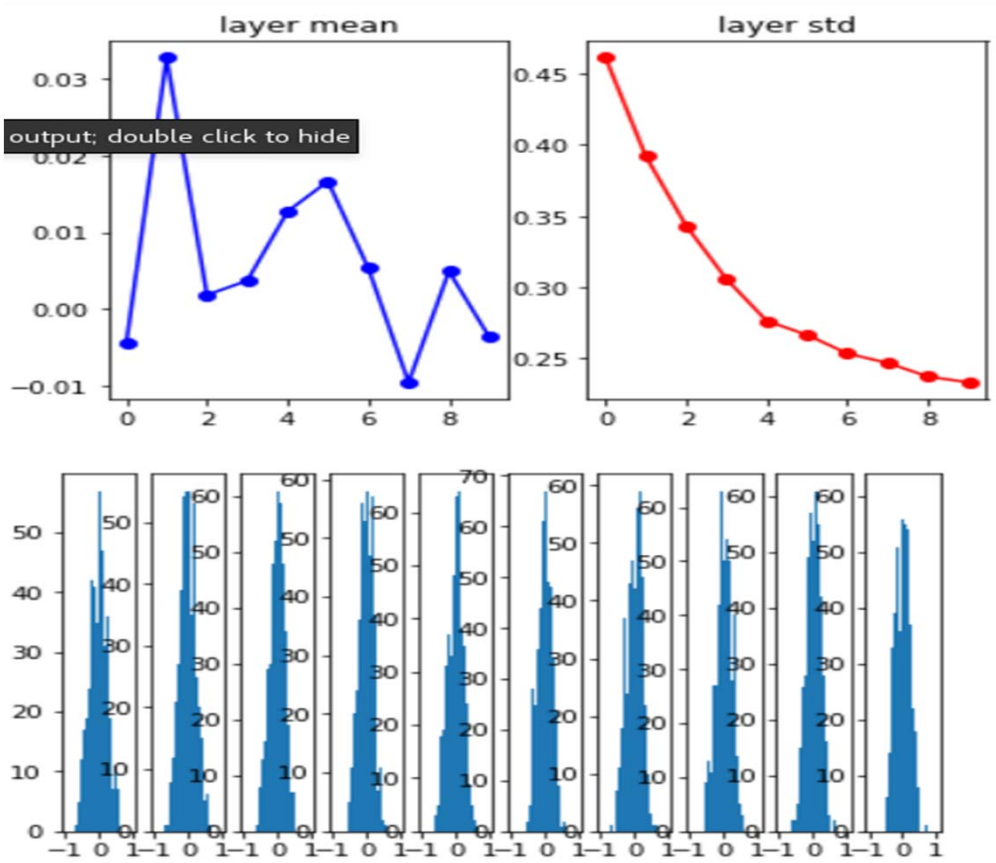
This is called Xavier-initalization

- It can be shown that

$$Var(z) = (nVar(w))(Var(x))$$

- If we make sure that $Var(w_i)=1/n$ for all i, so by scaling each weight wi by $\sqrt{1/n}$, the variance of the output will be 1. (Called Xavier initialization)

With Xavier initialization and tanh



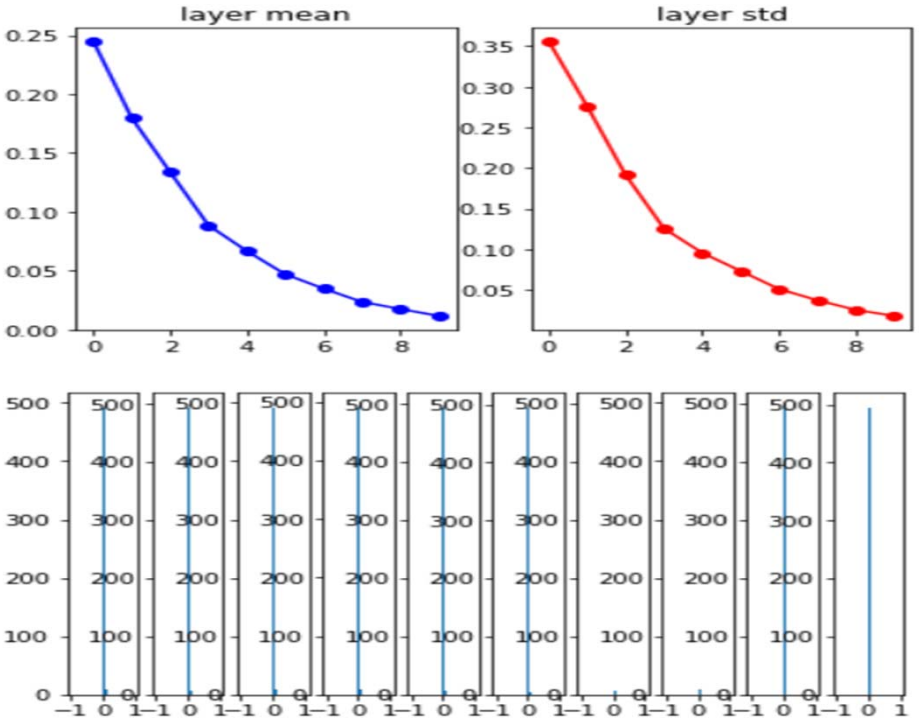
With tanh activation, Xavier works better as we want

w = np.random.rand(n)*sqrt(/n)

Xavier with ReLU – activations become zero again

```
In [25]: D = np.random.rand(1000,500)
         hidden_layer_sizes = [500]*10
         nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

Try ‘relu’



He initialization – normalizing the variance.

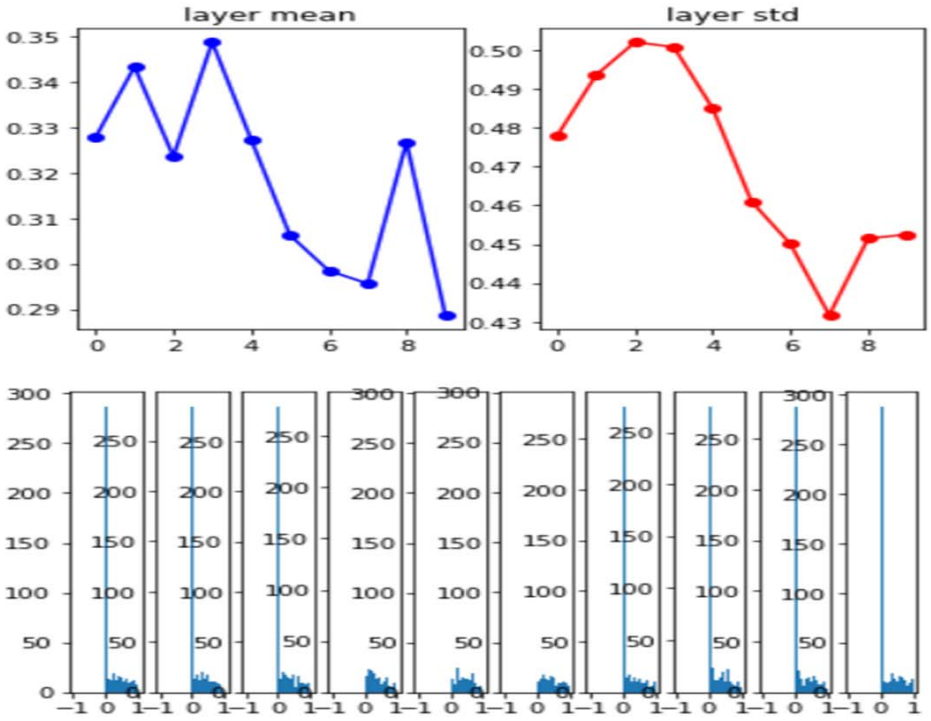
Xavier-normalization was developed for linear combinations, but we have a max-operator.

He et al. propose to use: w = np.random.rand(n)*sqrt(2/n) for ReLU because of the max-operation that will alter the distribution.

Use this or ReLU!

He initalization

w = np.random.rand(n)*sqrt(2/n)



Now the activations are not zero. Why do we have a peak at 0?

Initializing the bias terms

- When W is initalized to small random numbers, symmetry is broken and b can be initialized with 0.
- It is also common to initialize all b's to a common constant, e.g. 0.01

Initialization: Active area of research

Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

Where we are

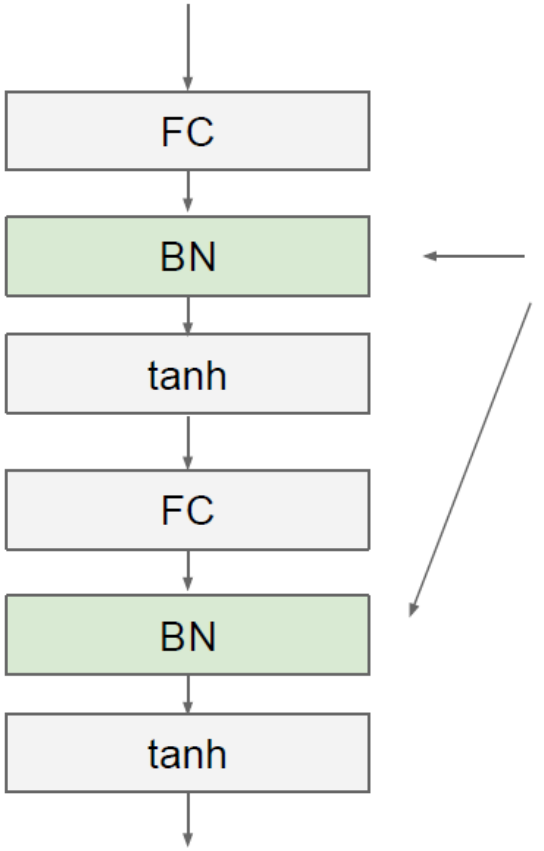
- Activation functions
- Data preprocessing
- Weight initialization
- **Batch normalization**
- Weight update schemes
- Searching for the best parameters

Batch normalization

- So far, we noticed that normalizing the inputs and the initial weights to zero mean, unit variance help convergence.
- As training progresses, the mean and variance of the weights will change, and at a certain point they make converenge slow again.
  - This is called a covariance shift.

- Batch normalization (Ioffe and Szegedy) https://arxiv.org/abs/1502.03167 countereffects this.

Batch normalization



- Idea: make your layer input to have a given mean and variance
- This layer makes the input gaussian with zero mean and unit variance by applying

$$\hat{x}_k = \frac{x_k - \mu_k}{\sqrt{Var(x_k)}}$$

μ_k and $Var(x_k)$

is computed after each mini batch during training.

- This normalization (zero mean, unit variance) can limit the expressive power of the unit. To maintain this we rescale to y_k

$$y_k = \gamma_k \hat{x}_k + \beta_k$$

- What? Does this help?
  - Yes, because the network can learn γ_k and β_k during backpropagation, and it learns faster. Learning without the new parameter scaling must be done through the input weights and is much more complicated.

- **Batch normalization significantly speeds up gradient descent, and often improves the accuracy. TRY IT!**

Batch normalization: training

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch normalization: test time

- At test time: mean/std is computed for the ENTIRE TRAINING set, not mini batches used during backprop (you should store these).
- Remark: use running average to update

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Where we are

- Activation functions
- Data preprocessing
- Weight initialization
- Batch normalization
- **Weight update schemes**
-

Learning with minibatch gradient descent

- Recently, a number of methods for improving the convergence of minibatch gradient descent have been proposed:
  - Momentum and Nesterov Momentum
    - Momentum is a well-established optimization method
  - AdaGrad
  - RMSProp
  - ADAM

Learning with minibatch gradient descent

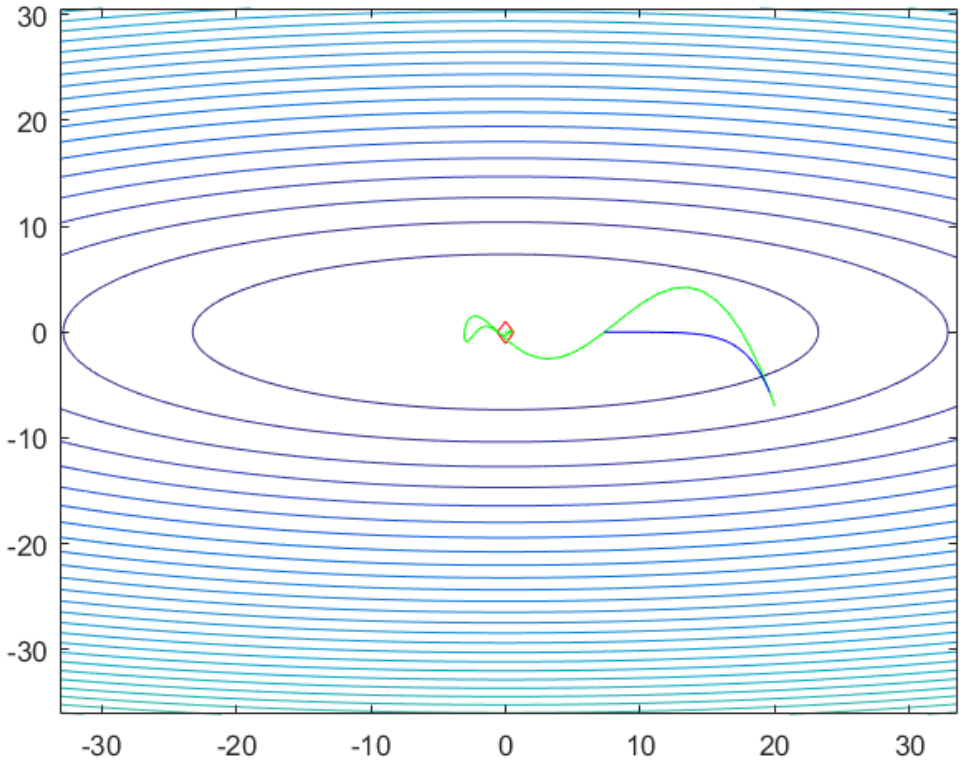
- Setting the learning η rate is difficult, and the performance is sensitive to it.
  - Too low: slow convergence
  - Too high: oscillating performance
- In practise when using minibatch gradient descent: decay the learning rate linearly until iteration τ, then leave η constant:
  - $\eta_k = (1-\alpha)\,\eta_0 + \alpha\,\eta_\tau$, where $\alpha = k/\tau$,

Gradient descent with momentum

```
v=mu*v - learning_rate*df  # Integrate velocity
f += v                     #Integrate position
```

- Physical interpretation: ball rolling downhill
- mu: friction coefficient
- mu normally between 0.5 and 0.99
  - Can gradually decrease from 0.5 to 0.99 e.g.
- Allows velocity to build up in shallow directions, but is dampened in steep directions because of the sign changes.

Gradient descent with momentum



η =0.01

Momentum with mu=0.9 (green) vs. regular gradient descent (blue), 100 it.
Notice that momentum overshoots the minimum,but then goes back.

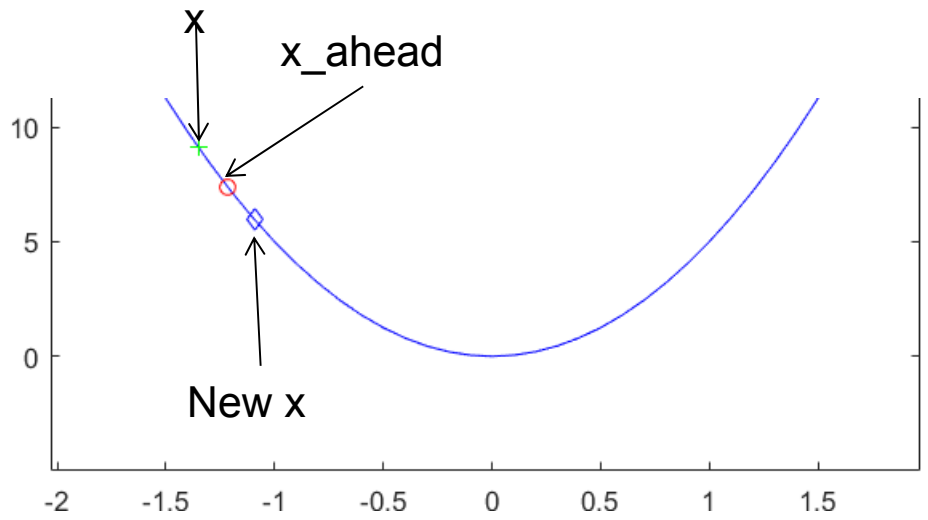
Nesterov momentum

- Idea: if we are at point x, with momentum the next estimate is x+mu*v due to velocity from previous iterations.
- Momentum update has two parts: v=mu*v - learning_rate*df
  - One due to velocity, and one due to current gradient
- Since velocity is pushing us to x+mu*v, why not compute the gradient at point x+mu*v, not point x? (Look ahead)

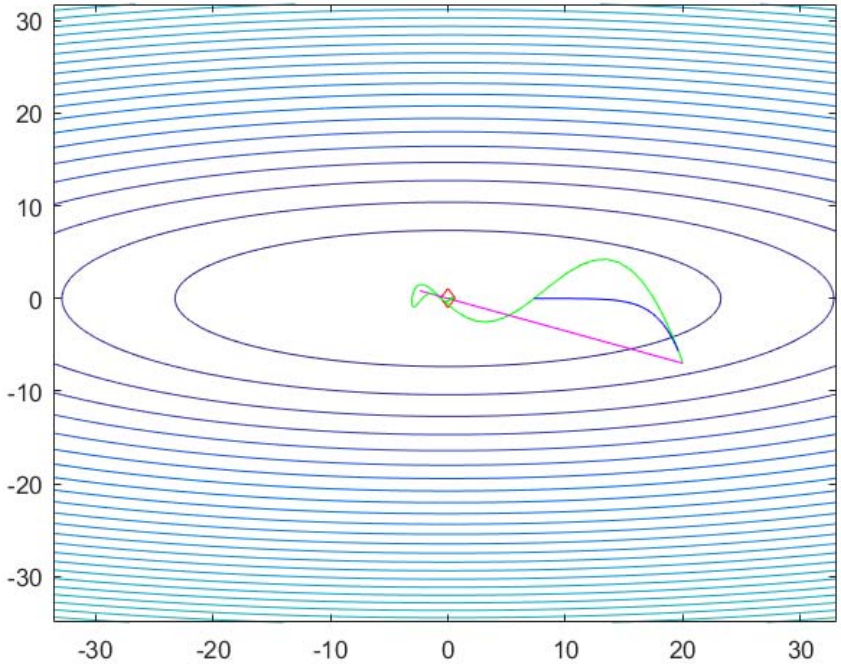
```
x_ahead = x + mu*v #Only the velocity part
# Evaluate the gradient at x_ahead
v = mu*v – learning_rate*dx(x_ahead)
x += v
```

Nesterov momentum

- x_ahead = x + mu*v #Only the velocity part
- # Evaluate the gradient at x_ahead
- v = mu*v – learning_rate*dx(x_ahead)
- x += v



Nesterov momentum



Momentum (green) vs. regular gradient descent (blue), Nesterov (magenta)
Notice that Nesterov reduces overshoot near minimum.

Implementing Nesterov

- Notice that Nesterov creates the gradient at x_ahead, while we go directly from x to x+v.
- It is more convenient to avoid computing the gradient at a different location by rewriting as:

```
v_prev = v # Back this up
v = mu * v – learning_rate * dx
x += -mu*v_prev + (1-mu)*v
```

AdaGrad updates (DL 8.5.1)

- From http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf
- Keep a cache of elementwise squared gradients g=dx

```
# Adagrad update
cache += dx**2
x += -learning_rate * dx/(np.sqrt(cache)+1e-7)
```

- Note that x, dx and cache are vectors.
- cache builds of the accumulated gradients in each direction.
  - If one direction has large gradient, we will take a smaller step in that direction.
- A problem with AdaGrad is that cache builds up larger and larger, and the step size can be smaller and smaller.
  - Use RMSprop or ADAM instead

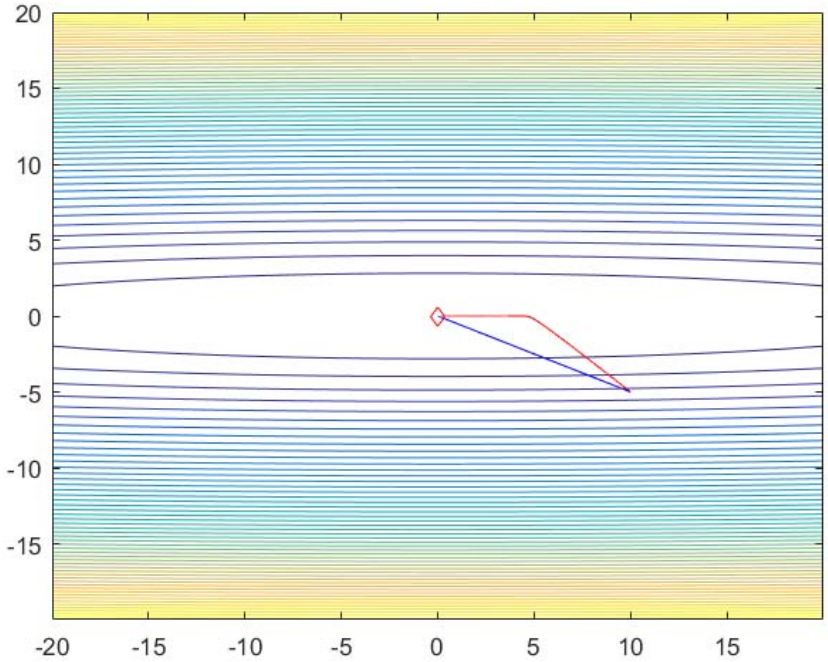
RMSprop update

- DL 8.5.2 and http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

```
# RMSprop update
decay =0.9
cache = decay*cache + (1-decay)*dx**2
x += -learning_rate * dx/(np.sqrt(cache)+1e-7)
```

- Here cache is a moving average of the gradients for each weight
- Works better than AdaGrad.

RMSprop update



Blue: Nesterov
Red: RMSprop

ADAM update

- DL 8.5.3 and https://arxiv.org/abs/1412.6980
- Like RMSprop but with momentum

```
# ADAM update, all variables are vectors
rho1 = 0.9, rho2 = 0.999, eps=0.001
# initialize first and second moment variables
s=0, r=0
tau = t+1
s = rho1*s + (1-rho1)*dx
r = rho2*r + (1-rho2)*dx.*dx #elementwise
sb=s/(1-rho1**tau)
rb =r/(1-rho2**tau)
x = x – eps*sb/(sqrt(rb) +1e-8)
```

Beyond the gradient: Hessian matrices (DL 4.3.1)

- If W has N components, we can compute the derivative **g** of the cost function J with respect to all N components
- We can compute the derivative of any of these with respect to the N components again to get the second derivative of component i with respect to component j.
- The second derivative, **H**, is then a matrix of size NxN, and is called the Hessian.
- We approximate the cost function J locally using a second-order approximation around x_0 : (**g** is the vector of derivatives and **H** the matrix of second-order derivatives):

$$J(x) \approx J(x_0) + (x - x_0)^T \mathbf{g} + \frac{1}{2}(x - x_0)^T \mathbf{H}(x - x_0)^T$$

- Remark: storing H for large nets is memory demanding!

Second-order methods and their limitations (DL 8.6)

- Newton's method would update x as:

$$x_t = x_{t-1} - \left[Hf\left(x_{t-1}\right)\right]^{-1} \nabla f\left(x_{t-1}\right)$$

- Appears convenient – no parameters!
- Challenge: if we have N parameters/weight, H has size NxN!! Impossible to invert, hard also to store H^{-1} in memory.
- One alternative that approximates H^{-1} and avoid storing it is Limited Memory BFGS (L-BFGS)
  - See https://en.wikipedia.org/wiki/Limited-memory_BFGS
  - Drawback: only works well for full batch gradient descent, so it currently not commonly used for large deep nets.

Covered today

- Activation functions
- Data preprocessing
- Weight initialization
- Batch normalization
- Weight update schemes

- To be contiued next week, with a focus on generalization and regularization