UiO : Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Repetition of Annes lectures

Anne Solberg

May 26, 2017

The linear regression problem, summary

Hypothesis: $h(\theta) = \hat{y} = \theta^T x$

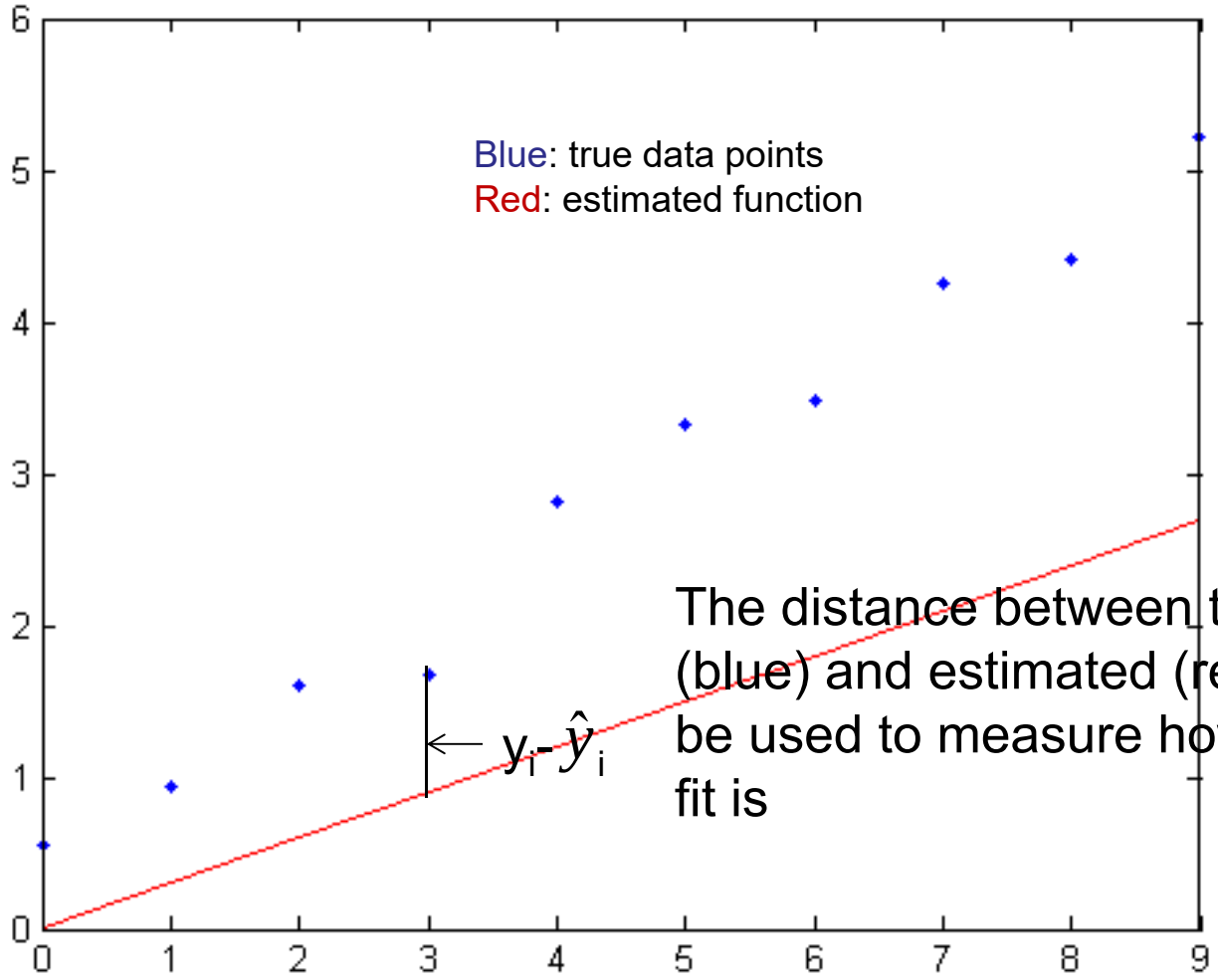
Parameters: $\theta^j, j = 0..n$

Cost function: $J(\theta^0) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$

Goal: $\underset{\theta^0}{\text{minimize}}\, J(\theta)$

Gradient descent solution:
repeat until convergence
  for j=0:n

$$\theta^j = \theta^j - \varepsilon \frac{\partial}{\partial \theta^j} J(\theta_1, \theta_2)$$



The distance between the true (blue) and estimated (red) value will be used to measure how good the fit is

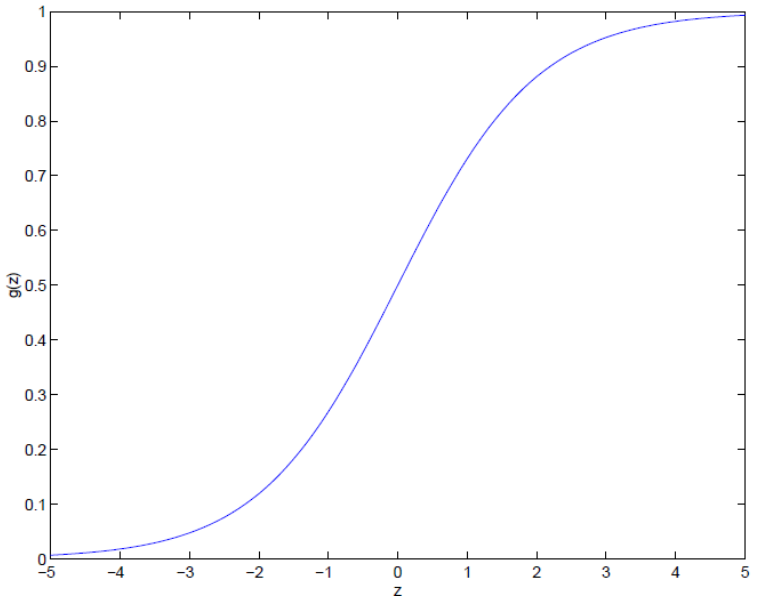
Logistic regression model

- Want $0 \leq h_\theta(x) \leq 1$
- Let

$$h_\theta(X) = g\left(\theta^T x\right)$$

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

$$h_\theta(X) = \frac{1}{1+e^{-\theta^T x}}$$



- This is called the sigmoid function

Decisions for logistic regression

- Decide y=1 if $h_\theta(x)>0.5$, and y=0 otherwise

$$h_\theta(X) = g\left(\theta^T x\right)$$

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

$$h_\theta(X) = \frac{1}{1+e^{-\theta^T x}}$$

- g(z)>0.5 if z>0
  - $\theta^T x>0$

g(z)<0.5 if z<0

$\theta^T x<0$

$\theta^T x=0$ gives the decision boundary

Logistic regression cost

Minimize $$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(Cost(h_\theta(x_i), y_i) \right)$$

Due to the sigmoid function g(z), this is a non-quadratic function, and non-convex.

Set $$Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

$\text{Cost} = 0 \text{ if } y = 1 \text{ and } h_\theta(x) = 1$

$\text{If } y = 1 \text{ and } h_\theta(x) \to 0 : \text{Cost} \to \infty$

$\text{Cost} = 0 \text{ if } y = 0 \text{ and } h_\theta(x) = 0$

$\text{If } y = 0 \text{ and } h_\theta(x) \to 1 : \text{Cost} \to \infty$

Mimick a probability

We skip deriving this cost, it is derived by maximizing the log-likelihood that θ fits the data

From 2 to multiple classes: Softmax

- The common generalization to multiple clasess is the **softmax classifier**.
- We want to predict the class label y_i={1,…C} for sample X(i,:), y can take one of C discrete values, so it follow a multinomial distribution.
- This is derived from an assumption that the probability of class y=k is

$$h_\theta(x) = p(y = k \mid x, \theta) = \frac{e^{\theta_k^T x}}{\sum_{j=1}^{C} e^{\theta_j^T x}}$$

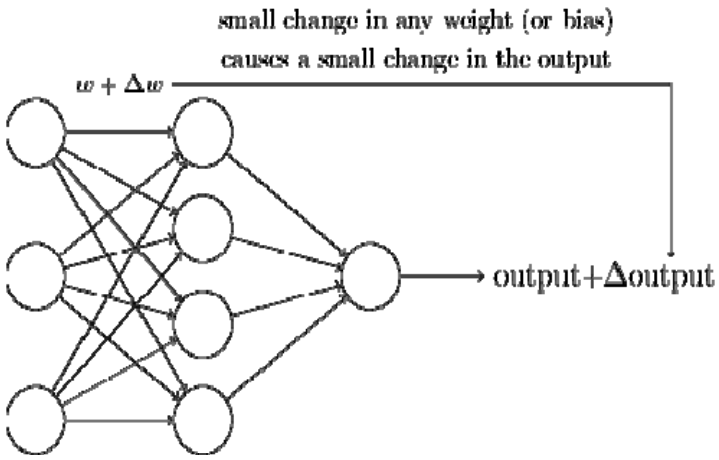
- The score or loss function for class i is

$$L_i = -\log\left(\frac{e^{\theta_i^T X(i,:)}}{\sum_{j=1}^{k} e^{\theta_j^T X(i,:)}}\right)$$

This is called the cross-entropy loss

Introduction to backpropagation and computational graphs

- We now have a network architecture and a cost function.
- A learning algorithm for the net should give us a way to change the weights in such a manner that the output is closer to the correct class labels.
- The activation function should assure that a small change in weights results in a small change in ouputs.
- Backpropagation use partial derivatives to compute the derivative of the cost function J with respect to all the weights.



Gradients and partial derivatives

$$f(x,y) = xy \rightarrow \frac{\partial f}{\partial x} = y \frac{\partial f}{\partial y} = x$$

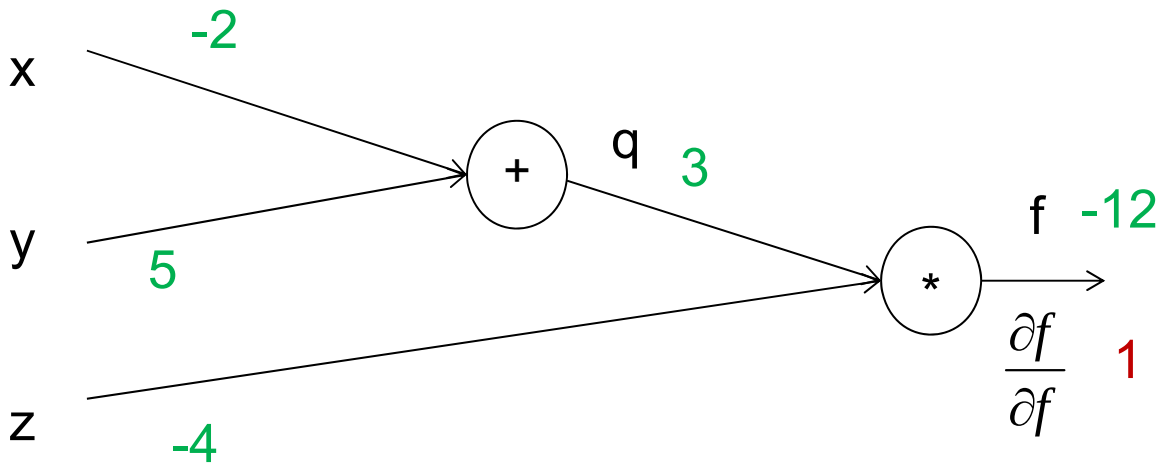
$$f(x,y) = x + y \rightarrow \frac{\partial f}{\partial x} = 1 \frac{\partial f}{\partial y} = 1$$

$$f(x,y) = \max(x,y) \rightarrow \frac{\partial f}{\partial x} = 1(x \geq y) \frac{\partial f}{\partial y} = 1(y \geq x)$$

$f(x,y,z) = (x+y)z$ Let q = x + y and $f = qz$ and use the chain rule :

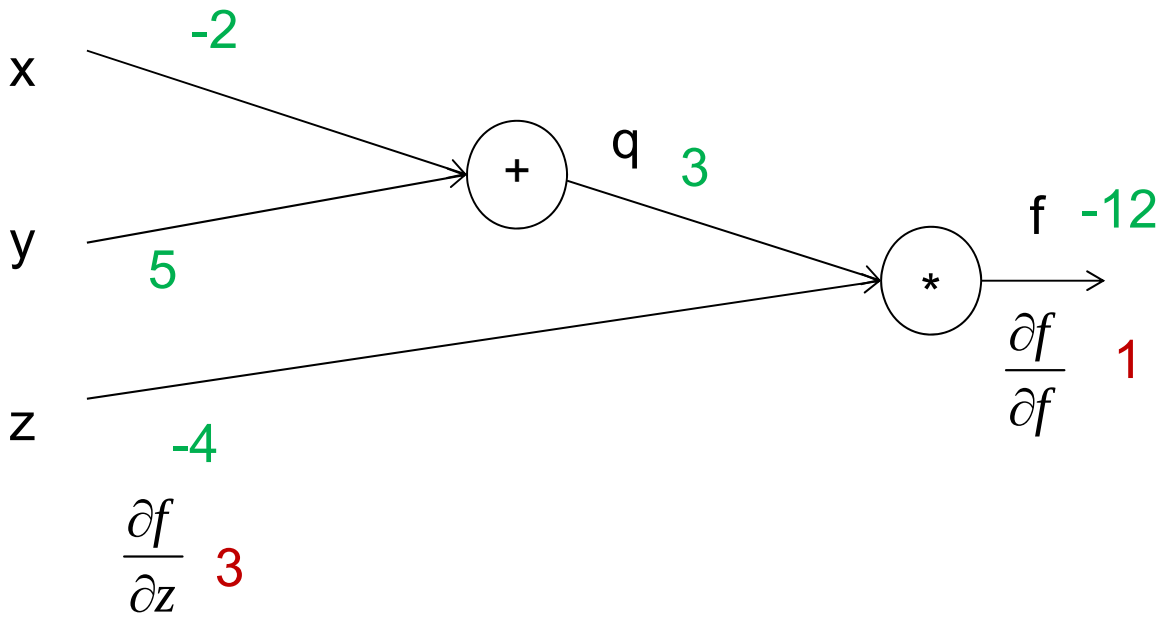
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Backwards propagation of gradients



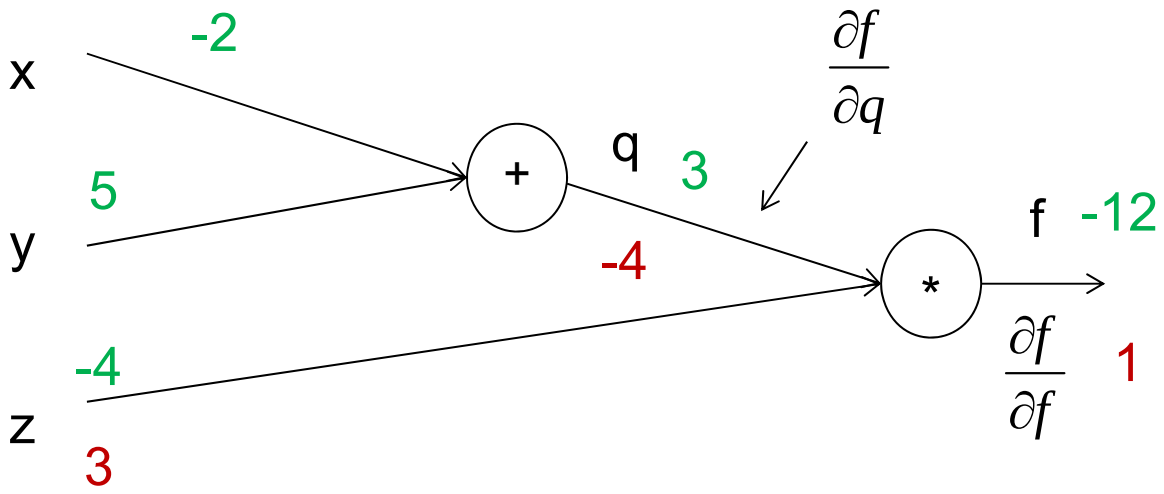
Green numbers: forward propagation
Red numbers: backwards propagation

Backwards propagation of gradients



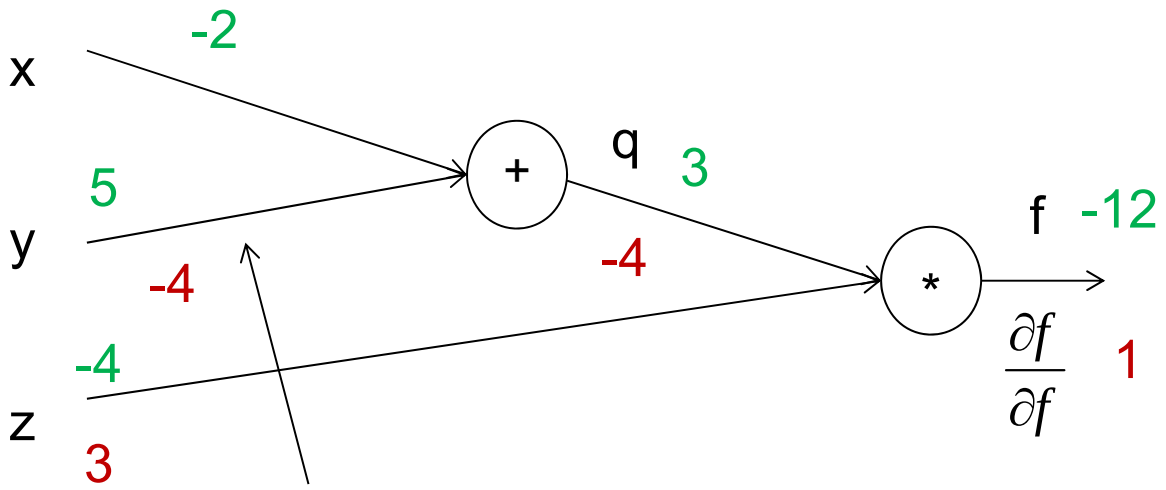
Green numbers: forward propagation
Red numbers: backwards propagation

Backwards propagation of gradients



Green numbers: forward propagation
Red numbers: backwards propagation

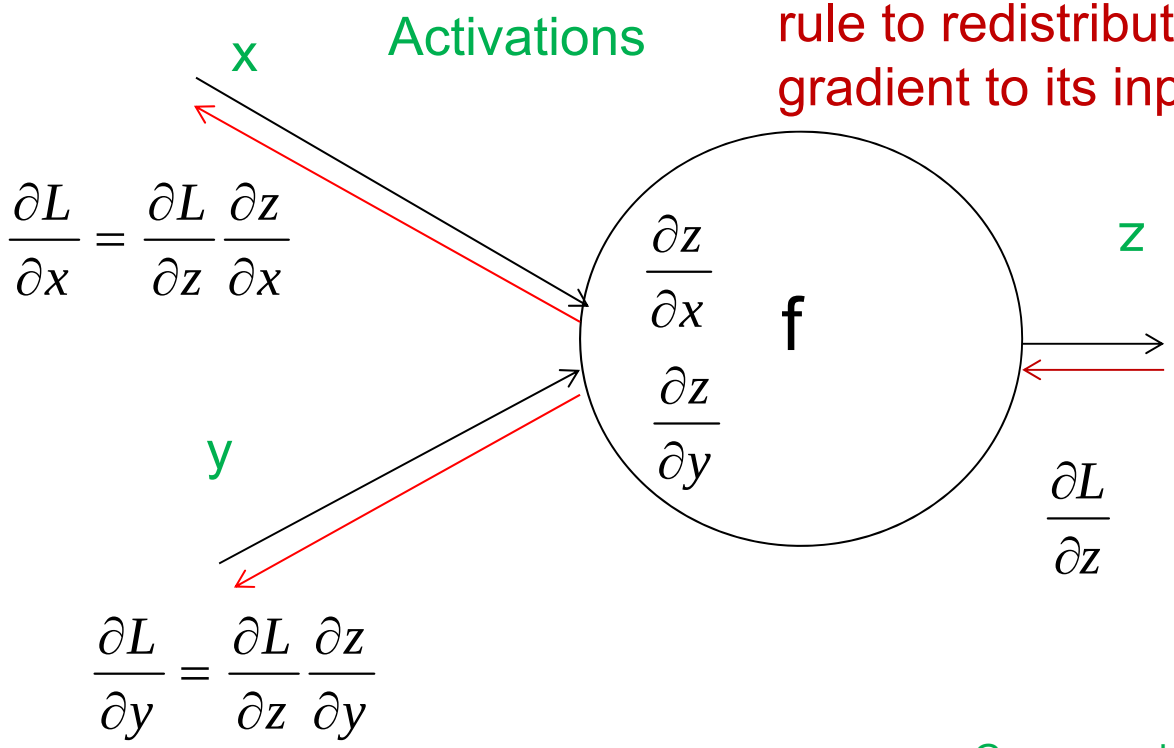
Backwards propagation of gradients



Green numbers: forward propagation
Red numbers: backwards propagation

During backpropagation, the node will learn $\frac{\partial L}{\partial z}$

The gate uses chain rule to redistribute this gradient to its inputs



Green numbers: forward propagation
Red numbers: backwards propagation

Activation functions

- Reading material:
  - cs231n.github.io/neural-networks-1
  - Deep Learning: 6.2.2 and 6.3
- Active area of research, new functions are published annually. We will consider:
  - Sigmoid activation
  - Tanh activation
  - ReLU activation
  - And mention recent alternatives like:
    - Leaky ReLU
    - Maxout
    - ELU

Data preprocessing

- Scaling of the features matters:
- If we have the samples

xi: yi

101, 101: 2

101, 99: 0

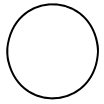
Original

xi: yi

1, 1: 2

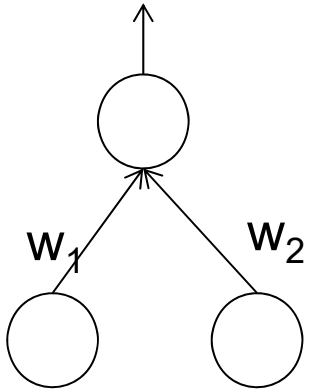
1, -1: 0

Error surface



Scale to
zero mean

Error surface



Data preprocessing

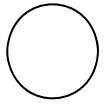
- Scaling of the features matters:
- If we have the samples

xi:	yi
0.2, 10:	2
0.2, -10:	0

Original

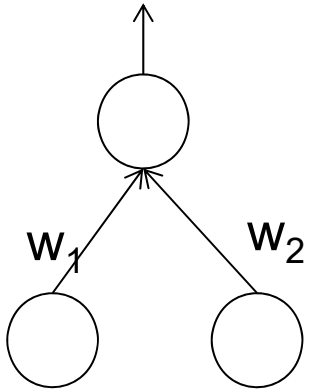
xi:	yi
1, 1:	2
1, -1:	0

Error surface



Error surface

Normalize
to unit variance



Weight initialization

- Avoid all zero initialization!
  - If all weights are equal, they will produce the same gradients and same outputs, and undergo exactly the same parameter updates.
    - They will learn the same thing.
- We break symmetry by initializing the weights to have small random numbers.

- Initialization is more complicated for deep networks

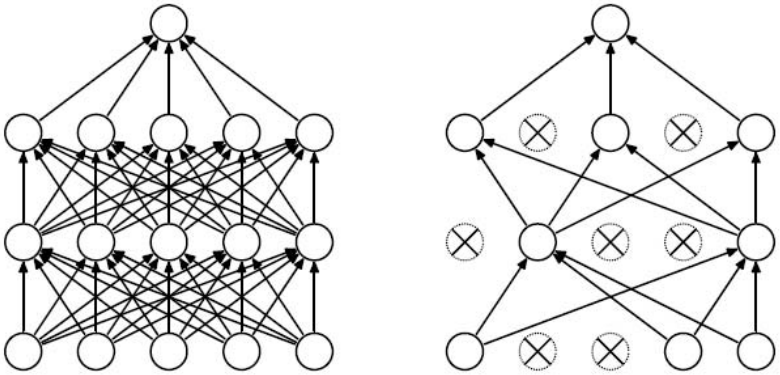
Batch normalization

- 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.
- After fully connected layers (or convolutional layers),and before the nonlinearity, a batch normalization layer is inserted.
- This layer makes the input gaussian with zero mean and unit variance by applying

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

Dropout

- Presented in http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf
- Achieves a similar effect as bagging by randomly setting the output of a node to zero (by multipying with a random vector of zeros with probability p).



(a) Standard Neural Net (b) After applying dropout.

Example: cat class with nodes detecting
- Eyes
- Ears
- ~~Tail~~
- Fur
- Legs
- ~~Mouth~~

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout - training

- Choose a dropout probability p
- We can drop both inputs and nodes in hidden layers.
- Create a binary mask for all nodes with probability of zero=p.
- Consider a 3-layer network with dropout in the hidden layers

```
# Forward pass of 3-layer net
H1 = np. maximum(0,np.dot(W1,X)+b1)
U1 = np.random.rand(*H1.shape)<p    # first dropout
H1 *= U1
H2 = np.maximum(0,np.dot(W2,H1)+b2)
U2 = np.random.rand(*H2.shape) < p # Second dropout
H2 *= U2
out = np.dot(W3,H2) +b3
```

- Backpropagate as usual, but take into account the drop.

Dropout – predict : naive implementation

- A drop rate of p will scale the outputs during training with a factor p<1.
- When we predict new data, without considering this scaling, the outputs will be larger.
- We have to scale the outputs during predict by p:

```
# predict
H1 = np.maximum(0,np.dot(W1,X)+b1)*p
H2 = np.macimum(0,np.dot(W2,H1)+b2)*p
out = np.dot(W3,H2)+b3
```

- Since test-time performance is critical, we normally apply «inverted dropout» and scale at training time.

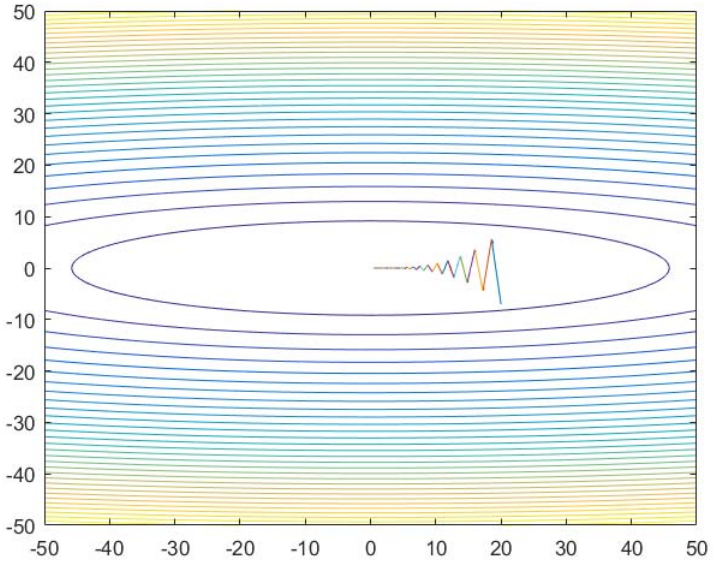
Inverted dropout

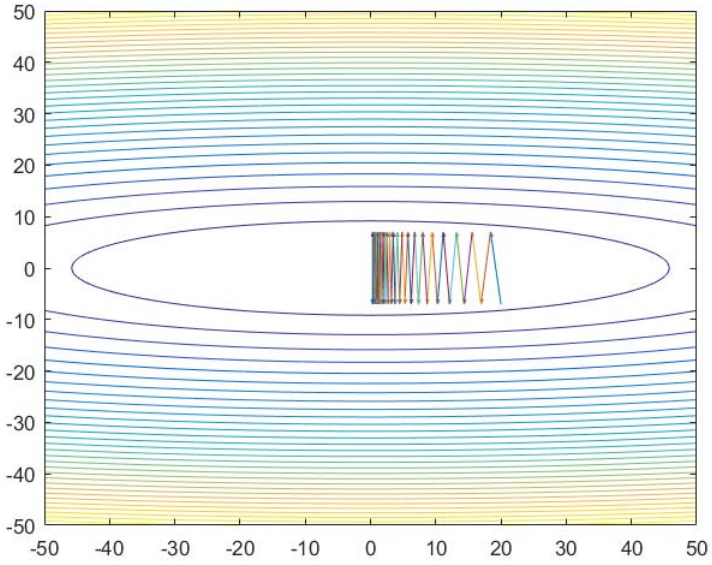
```
p=0.5
#train
H1 = np.maximum(0,np.dot(W1,X)+b1)
U1 = (np.random.rand(*H1.shape)<p)/p #Scale now
H1 *= U1
H2 = np.maximum(0,np.dot(W2,H1)+b2)
U2 = (np.random.rand(*H2.shape) < p) / p # Second scaled dropout
H2 *= U2
out = np.dot(W3,H2)+b3

# predict
H1 = np.maximum(0,np.dot(W1,X)+b1) #No scaling necessary
H2 = np.macimum(0,np.dot(W2,H1)+b2)
out = np.dot(W3,H2)+b3
```

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_τ constant:
  - $\eta_k=(1-\alpha)\,\eta_0+\alpha\,\eta_\tau$, where $\alpha=k/\tau$,

Gradient descent oscillations





η =0.19

η =0.20

This is how gradient descent moves

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.

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
```

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.

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)
```