UiO : Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Lecture : Neural net: initialization, activations, normalizations and other practical details

Anne Solberg

March 10, 2017

Mandatory exercise 1

- Available tonight, deadline 31.3.17
- Implementing Softmax-classification
- Implementing 2-layer net with backpropagation
- Seeing the effect of adding feature extraction using histogram of gradients.
- Optimizing network parameters on validation data
  - How high accuracy can you get?

The coming weeks

- No weekly exercise set this week as you should work on Mandatory 1
- Group sessions as normal
- Next lecture: background in image convolution, filters, and filter banks/multiscale representations.
  - Gives a useful background for convolutional nets.
- In two weeks: continue with useful tricks for making learning work, mainly:
  - chapter 8 in Deep Learning
  - http://cs231n.github.io/neural_networks-3

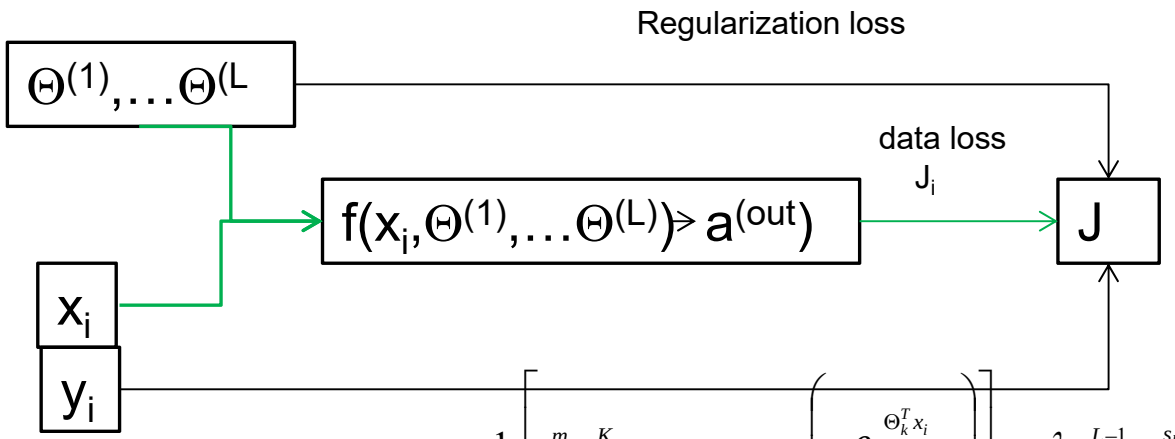
Reading material

- Reading material:
  - http://cs231n.github.io/neural_networks-2
  - Deep Learning 6.2.2 and 6.3 on activation functions
  - Deep Learning 8.7.1 on Batch normalization

Today

- Activation functions
- Mini-batch gradient descent
- Data preprocessing
- Weight initialization
- Batch normalization
- Training,validation, and test sets
- Searching for the best parameters

The feedforward net problem

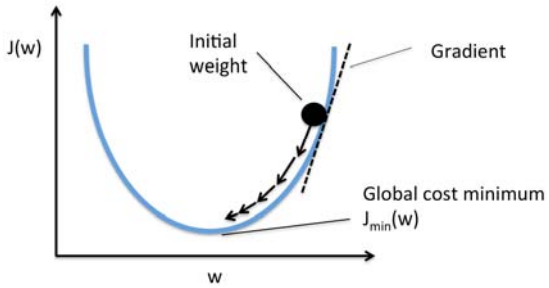


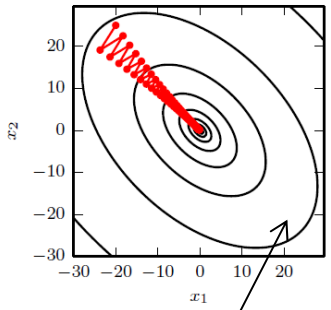
$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y_i = k\}\log\left(\frac{e^{\Theta_k^T x_i}}{\sum_{k=1}^{K} e^{\Theta_k^T x_i}}\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

Data set X,y
Loss for each sample J_i (softmax or logistic one vs. all)
Regularization loss J_r
Total loss: $J = J_i + \lambda J_r$

The error surface of a linear neuron

- For a linear neuron with squared error, the error surface is a quadratic bowl.
- For neural net loss functions it is more complex, but can be approximated by a bowl **locally**.
- One of the challenges of gradient descent is how to make it converge best possible.
  - In two weeks: other parameter update schemes like rmsprop, ADAM etc.





Which direction does gradient descent choose for an ellipse?

Convergence of batch gradient descent

- Convergence is often slow
- If the error surface locally is like an ellipse, the gradient is big in the direction we only want a small change, and small in the direction we want a big change.
- With a high learning rate the process will oscillate and convergence is slow.
- If the gradient is computed from ALL training samples, there are ways to speed up the process.
- For large networks, it is normally better to use mini-batch learning.

Batch gradient descent

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y_i = k\}\log\left(\frac{e^{\Theta_k^T x_i}}{\sum_{k=1}^{K} e^{\Theta_k^T x_i}}\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

- Batch gradient descent computes the loss summed over ALL training samples before doing gradient descent update.

$$\Theta^{(l)} = \Theta - \eta \mathrm{D}^{(l)}$$

- This is slow if the training data set is large.

Mini batch gradient descent

- Select randomly a small batch, update, then repeat:

```
for it in range(num_iterations):
        sample_ind = np.random.choice(num_train, batch_size)
        X_batch = X[sample_ind]
        y_batch  = y[sample_ind]
        Ji, dgrad_l = loss(X_batch, y_batch, lambda)
        for all l
            Theta_l  -= learning_rate*dgrad_l
```

- If batch_size=1, this is called online learning, and sometimes Stochastic Gradient Descent (SGD)
  - *But the term SGD sometimes also means mini batch gradient descent*
- Mandatory exercise 1: implement mini batch gradient descent
- We get back to other parameter update schemes in two weeks
- Common parameter value: 32, 64, 128.

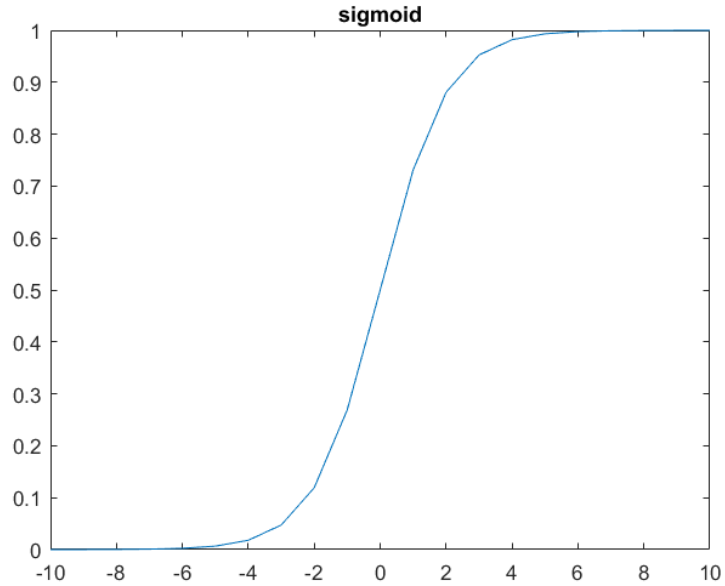
Activation functions

- 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

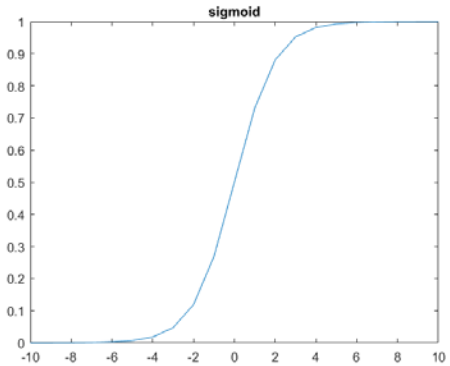
Sigmoid activation

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

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



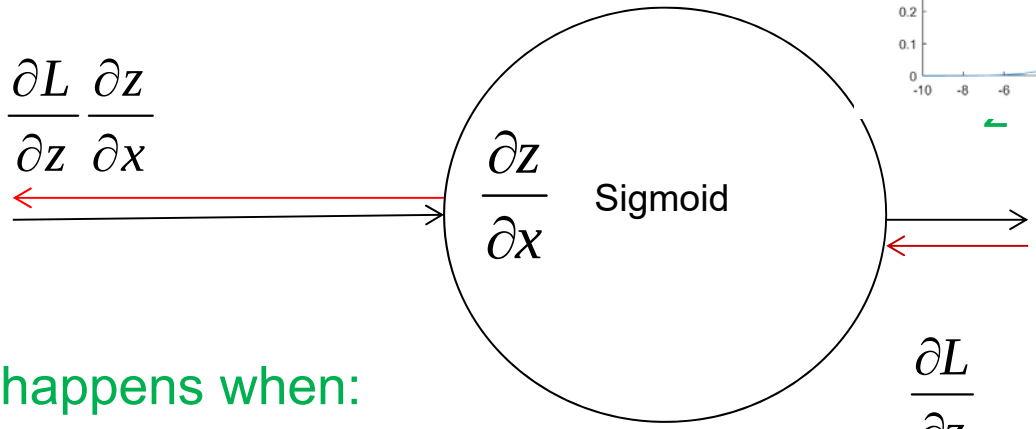
- Output between 0 and 1
- Historically popular
- Has some shortcomings



x

Activations

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



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

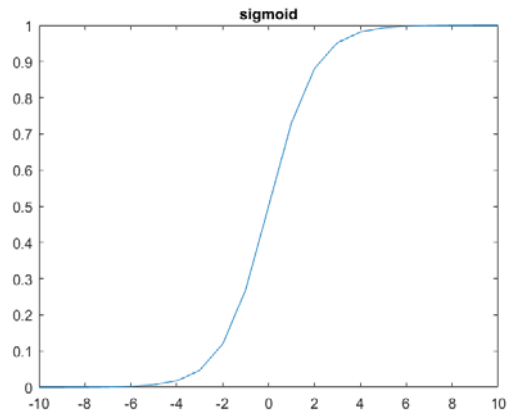
What happens when:

x=-10

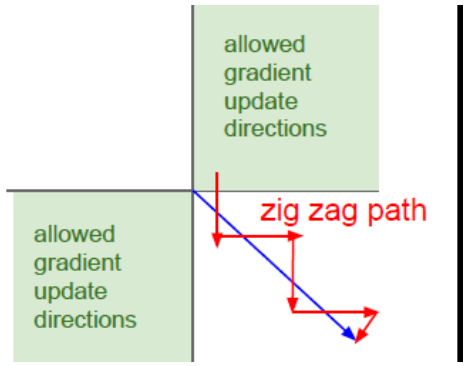
x=0

x=10

Sigmoid activation



- Sigmoids kill gradients
  - Why ? If the input is very small or large, what happens?
- Not zero-centered
  - If all inputs positive, then all gradients dJ/dΘ will be either positive of negative and gradient updates often zig-zag
- Somewhat expensive to compute

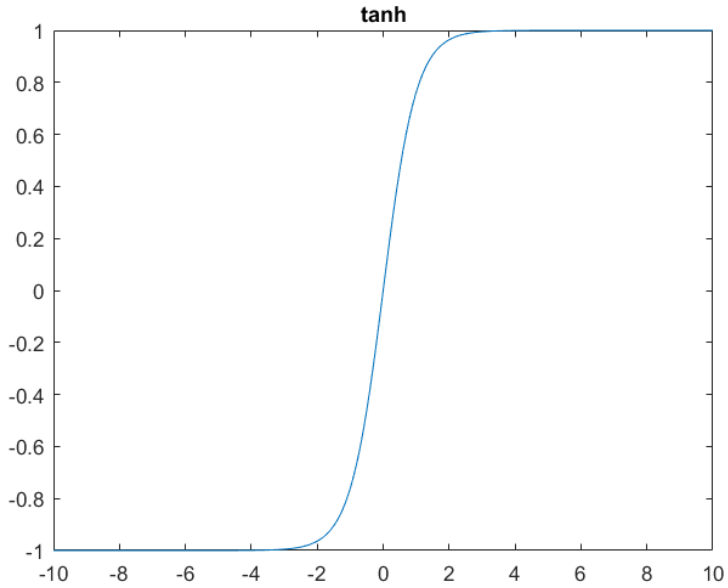


- Currently : sigmoids are rarely used!

Tanh activation

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

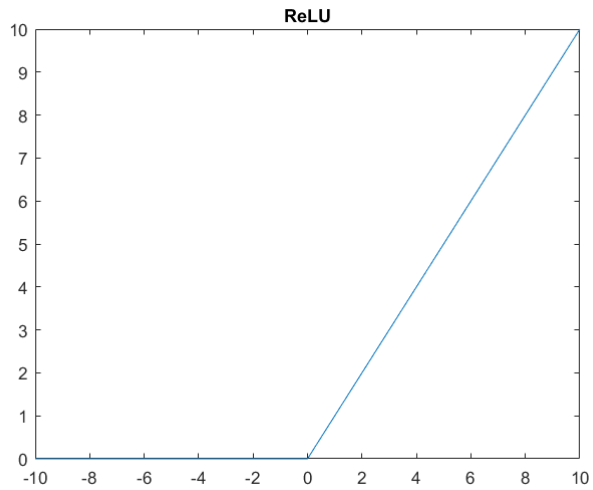
- Scaled version of sigmoid
- Output between -1 and 1
- Zero-centered
- Saturates and kill gradients
- Preferred to sigmoid due to the zero-centering



ReLU activation

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

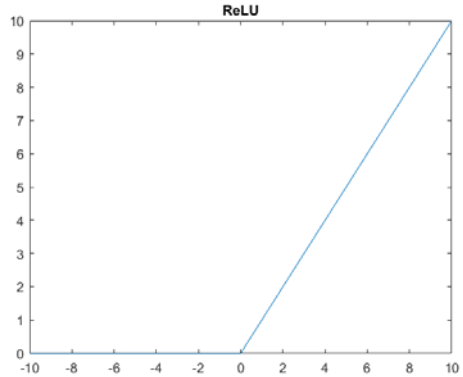
  Derivative of ReLU : $\max(z,0) = 1$ if $z > 0$ and 0 otherwise



- Rectified Linear Unit
- Does not saturate
- 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 overall recommendation

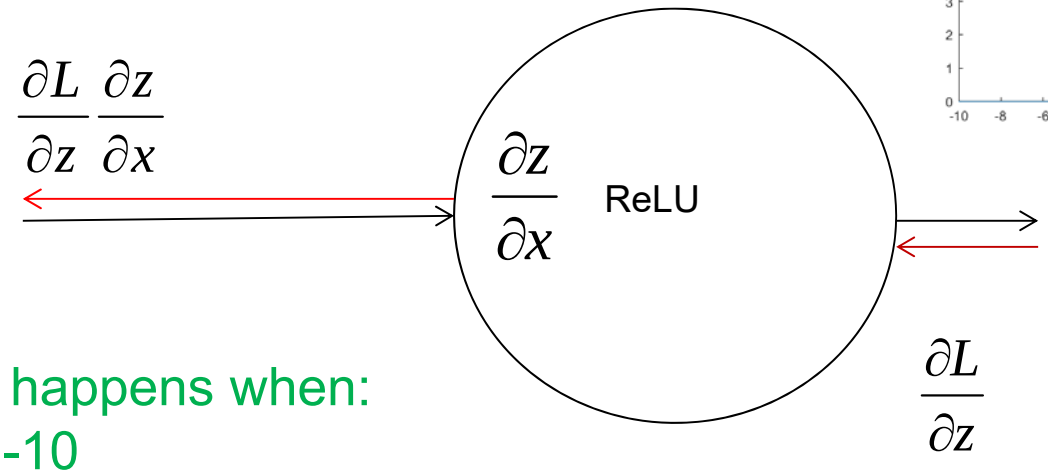
ReLU



x

Activations

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



What happens when:

x=-10

x=0

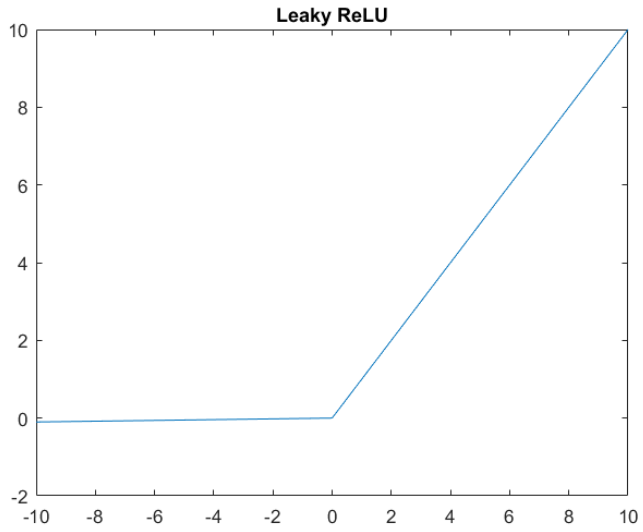
x=10

Leaky ReLU activation

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

-

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

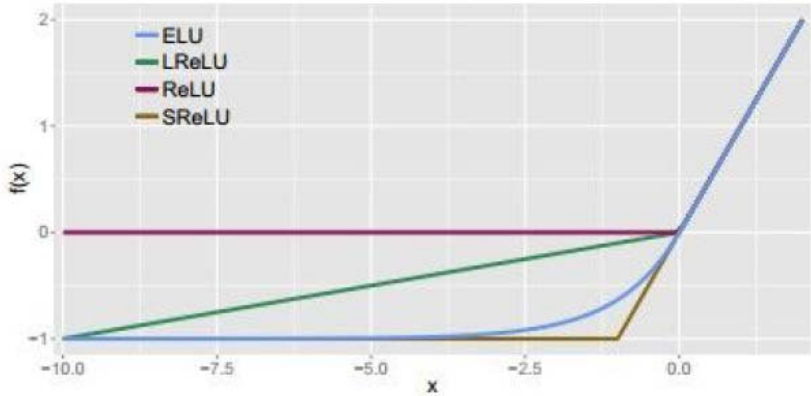


ELU activation

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

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

- Will not die
- Benefits of ReLU, but more expensive to compute



Maxout activation

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

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

Activations for output vs. hidden layers

- For classification, where the loss is either one vs. all logistic or softmax, the output layer will have:
  - Softmax activation for a softmax loss function
  - Sigmoid activation for one vs. All
- When we use ReLU or a different activation, this is normally for the hidden layers only

Remember from logistic classification that :

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

Review: Cost function for softmax neural networks

For a neural net with softmax loss function :

$$\text{Output}: a^L = h_\Theta(x) = \begin{bmatrix} P(y=1 \mid x, \Theta) \\ P(y=2 \mid x, \Theta) \\ \vdots \\ P(y=K \mid x, \Theta) \end{bmatrix} = \frac{1}{\sum_{k=1}^{K} e^{\Theta_k^T x}} \begin{bmatrix} e^{\Theta_1^T x} \\ e^{\Theta_2^T x} \\ \vdots \\ e^{\Theta_K^T x} \end{bmatrix}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y_i = k\} \log\left(\frac{e^{\Theta_k^T x_i}}{\sum_{k=1}^{K} e^{\Theta_k^T x_i}}\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

L : number of layers

s_l : Number of units (without bias) in layer l

$J(\Theta) = \text{LossTerm} + \lambda * \text{RegularizationTerm}$

Question

You can check your loss function. Set λ=0.

If you generate random data from n (say n=3) classes with equal probability, what do you expect the loss to be?

$$\text{Output}: a^L = h_\Theta(x) = \begin{bmatrix} P(y=1 \mid x, \Theta) \\ P(y=2 \mid x, \Theta) \\ \vdots \\ P(y=K \mid x, \Theta) \end{bmatrix} = \frac{1}{\sum_{k=1}^{K} e^{\Theta_k^T x}} \begin{bmatrix} e^{\Theta_1^T x} \\ e^{\Theta_2^T x} \\ \vdots \\ e^{\Theta_K^T x} \end{bmatrix}$$

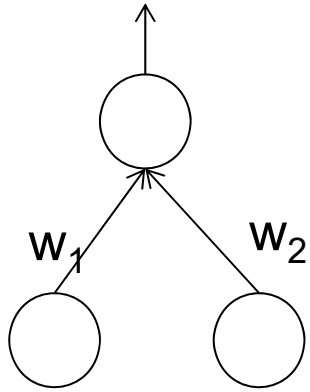
$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y_i = k\} \log\left(\frac{e^{\Theta_k^T x_i}}{\sum_{k=1}^{K} e^{\Theta_k^T x_i}}\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

Guidelines for activation function

- Active area of research, recommendations might change:
- Currently:
  - Use ReLU for hidden layers but monitor the fraction of ‘dead’ units in a network.
  - For output: most common with softmax

Data preprocessing

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



Original

xi: yi

101, 101: 2

101, 99: 0

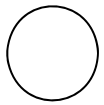
Error surface

Scale to zero mean

xi: yi

1, 1: 2

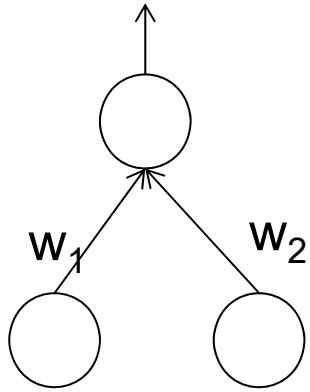
1, -1: 0



Error surface

Data preprocessing

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



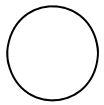
Original

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

Error surface

Normalize
to unit variance

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



Error surface

Common normalization

- Standardize data to zero mean and unit variance
- For each feature m, compute the mean μ and standdard deviation σ over the training data set and let $x_m = (x_m - \mu)/\sigma$
- Remark: STORE μ and σ because new data/test data must have the same normalization.

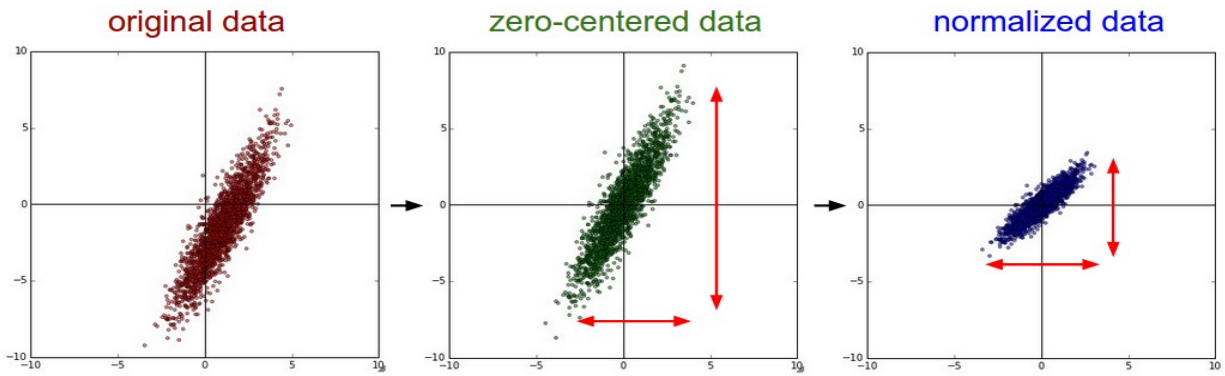


Figure from http://cs231n.github.io/neural-networks-2/

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

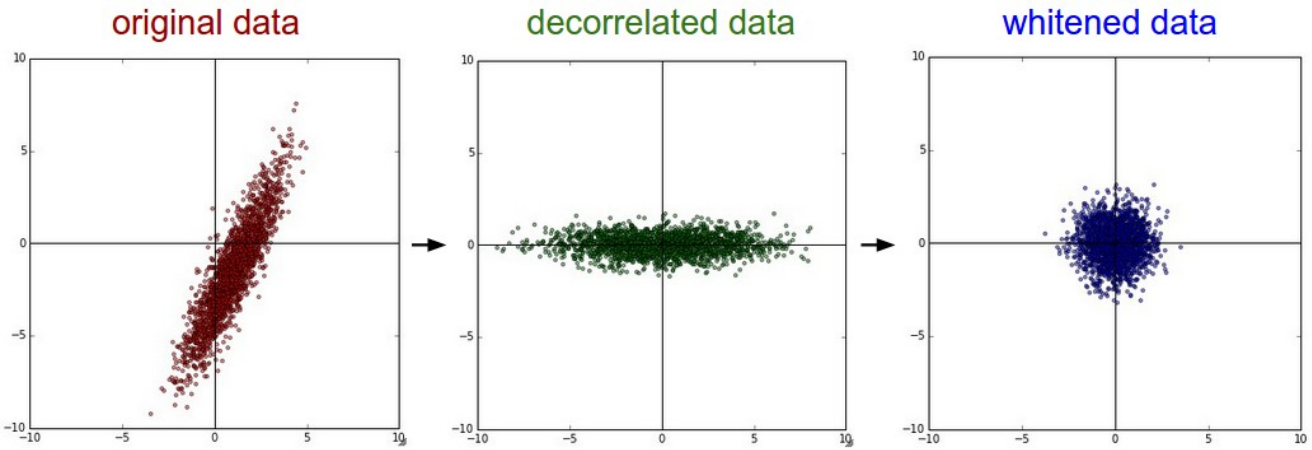


Figure from http://cs231n.github.io/neural-networks-2

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.

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

Weight initialization

- 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

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

- 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)

Use this for ReLU

Glorot 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.

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

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

- μ_k and $Var(x_k)$ is computed after each mini batch during training.
- This normalization 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 even improved the accuracy. USE IT!**

Batch normalization: training

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

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

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

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

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

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

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: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

Optimizing hyperparameters

- **Training data set**: part of data set used in backpropagation to estimate the weights.
- **Validation data set** (mean cross-validation): part of the data set used to find the best values of the hyperparameters, e.g. number of nodes and learning rate.
- **Test data**: used ONCE after fitting all parameters to estimate the final error rate.

Search strategy: coarse-to-fine

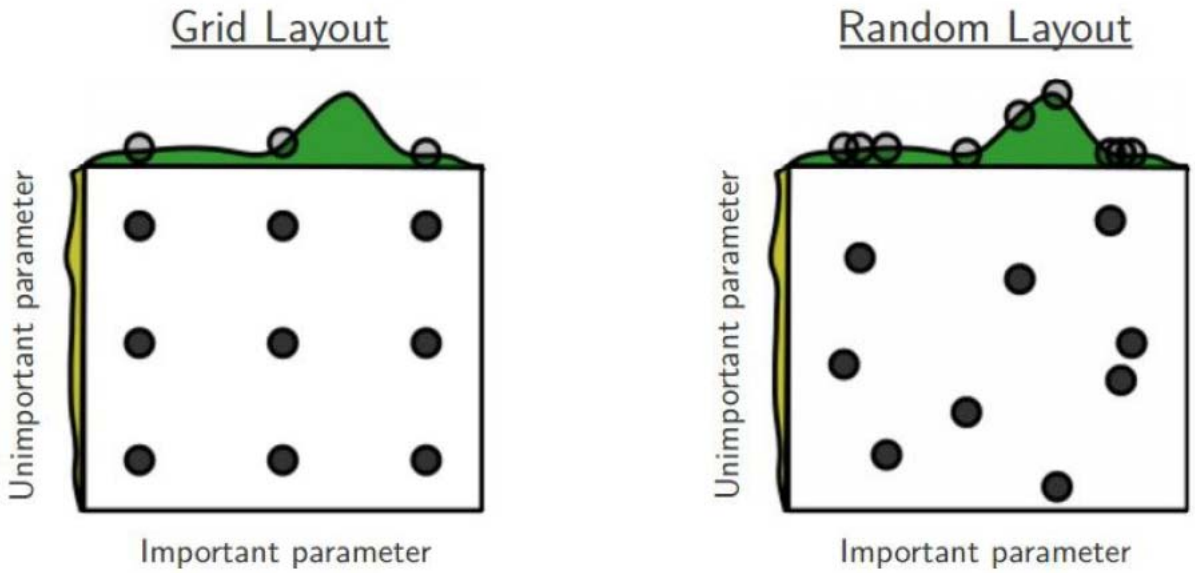
- First stage: run a few epochs (iterations through all training samples)
- Second stage: longer runs with finer search.
- Parameters like learning rate are multiplicative, search in log-space

Coarse search

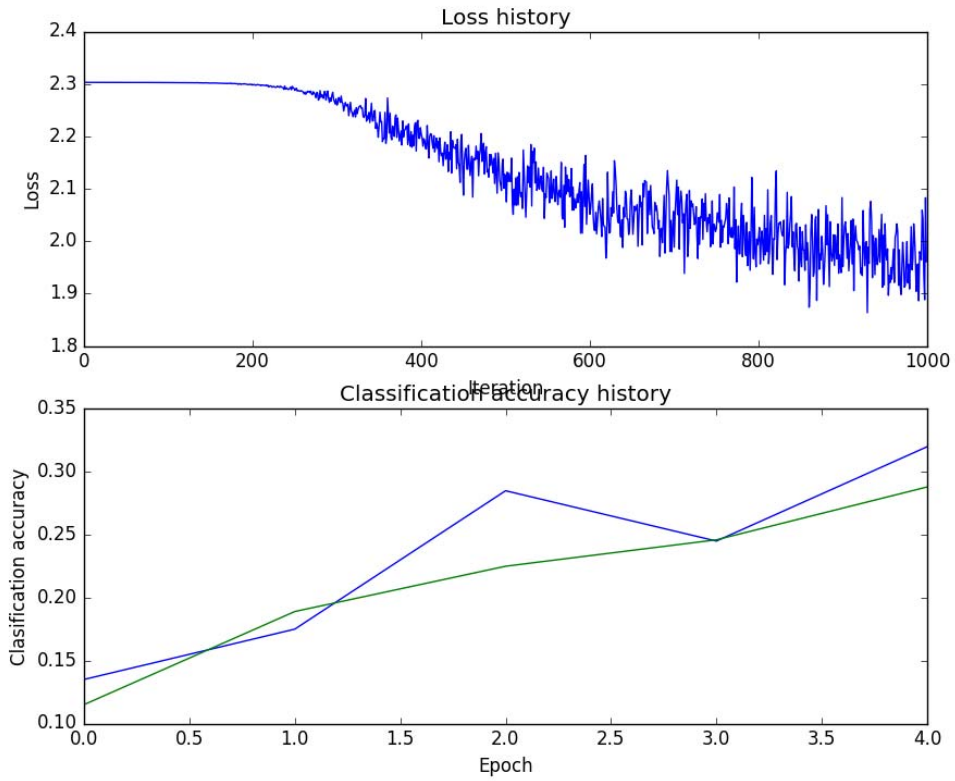
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
best_acc = -1
for hidden_size in [50, 100, 150]:
    for learning_rate in [5e-2, 1e-3, 1e-4]:
        for reg in [0.3, 0.4, 0.5, 0.6]:
            # Train the network
            net = TwoLayerNet(input_size, hidden_size, num_classes)
            stats = net.train(X_train, y_train, X_val, y_val,
                      num_iters=1000, batch_size=200,
                      learning_rate=learning_rate, learning_rate_decay=0.95,
                      reg=reg, verbose=True)
            # Predict on the validation set
            val_acc = (net.predict(X_val) == y_val).mean()
            print 'Hidden size:', hidden_size, 'Learning rate:', learning_rate, 'Reg', reg
            print 'Validation accuracy: ', val_acc
            if best_acc < val_acc:
                best_acc = val_acc
                best_net = net
```

Consider a random grid

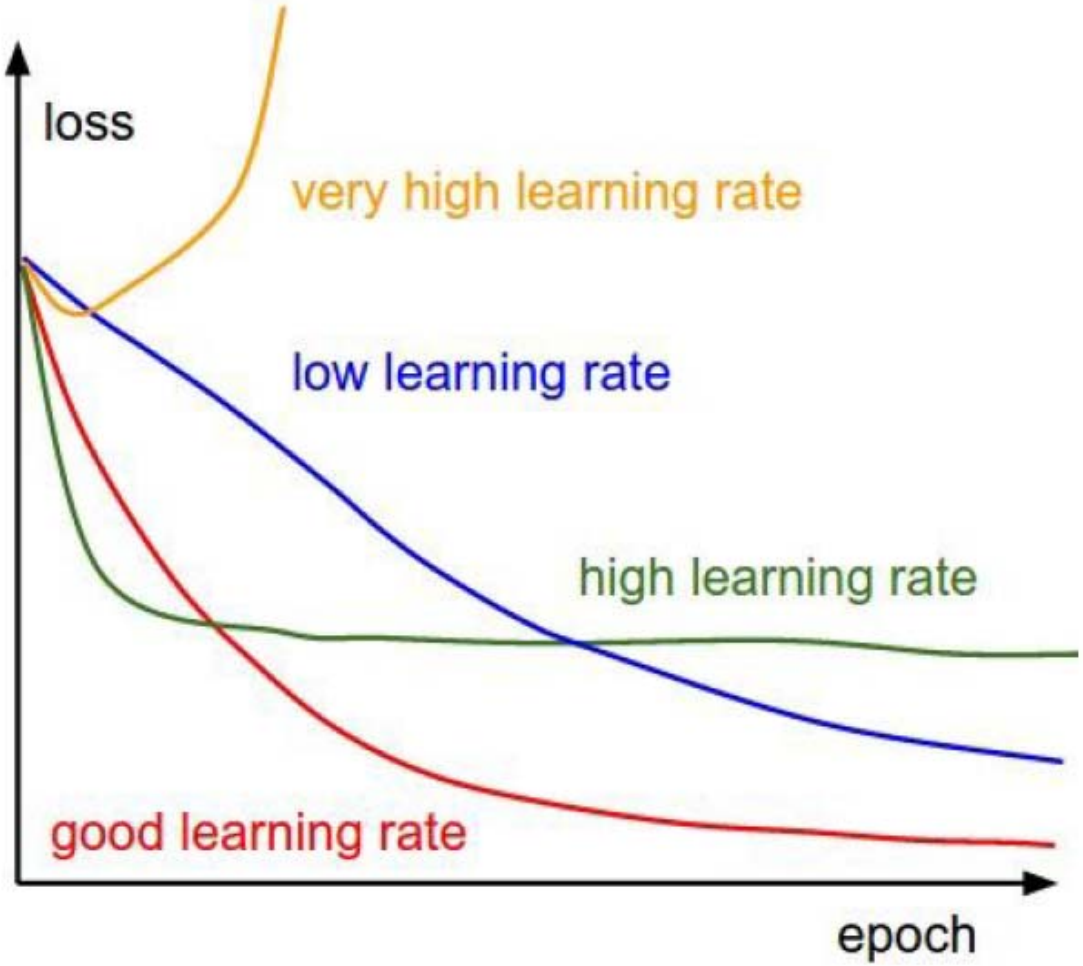
```
max_count = 100
for count in xrange(max_count):
        reg = 10**uniform(-5, 5)
        lr = 10**uniform(-3, -6)
```



Monitor the loss function



Behaviour of loss function



Debug the code

- Take a small subset of the training data
- Verify that you can overfit this subset e.g. without regularization
  - Expect very small loss and training accuracy of 1.00

Study outputs during training

- iteration 0 / 1000: loss 2.303033
- iteration 100 / 1000: loss nan
- iteration 200 / 1000: loss nan
- iteration 300 / 1000: loss nan
- iteration 400 / 1000: loss nan
- iteration 500 / 1000: loss nan
- iteration 600 / 1000: loss nan
- iteration 700 / 1000: loss nan
- iteration 800 / 1000: loss nan
- iteration 900 / 1000: loss nan
- Hidden size: 50 Learning rate: 0.05 Reg 0.6
- Validation accuracy: 0.087
- iteration 0 / 1000: loss 2.302811
- iteration 100 / 1000: loss 1.987349
- iteration 200 / 1000: loss 1.809840
- iteration 300 / 1000: loss 1.734420
- iteration 400 / 1000: loss 1.615647
- iteration 500 / 1000: loss 1.596564
- iteration 600 / 1000: loss 1.730853
- iteration 700 / 1000: loss 1.515625
- iteration 800 / 1000: loss 1.427454
- iteration 900 / 1000: loss 1.501289

Track ratio of weight updates/weight magnitudes

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

To be continued….