UiO : Department of Informatics
University of Oslo

**INF 5860 Machine learning for image classification**

Lecture 9

Training neural nets part II

Anne Solberg

March 24, 2017

# Today

- Regularization strategies
- Variations on stochastic gradient descent learning
- Dropout
- Summarizing the training procedure

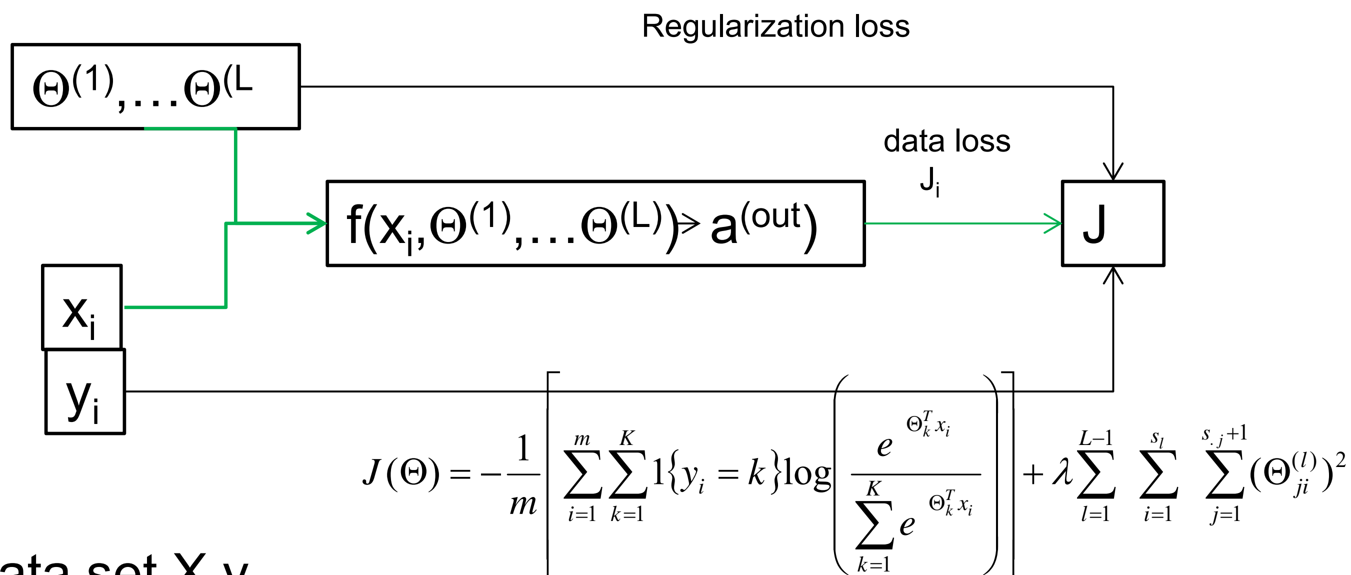
# Practical issues

- Mandatory 1 deadline in one week.
- No weekly exercises this week, practical experience in mandatory exercise.
- A set of theory exercises available after Mandatory 1.

- Midterm course evaluation: your constructive feedback requested.

- Next week: convolutional nets (finallly ☺ )

# The feedforward net problem




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

# Literature

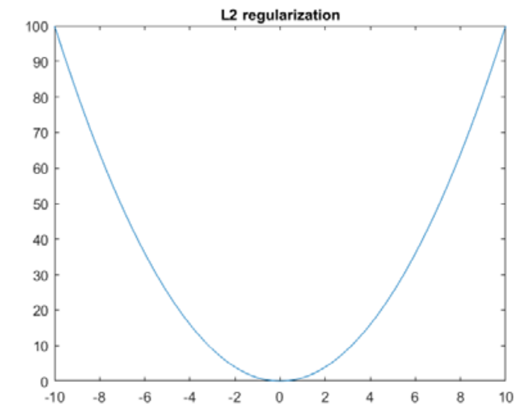
- On regularization:
  - cs231n.github.io/neural-networks-2
  - Deep Learning: 7.1

- Dropout:
  - cs231n.github.io/neural-networks-2
  - http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf
- Learning, parameter updates: DL Chapter 8.3, 8.5
- Local minima and second order methods DL 8.2 and DL 4.3.1)

# L2 regularization

- Cost function $J(\Theta) = J_i + \lambda \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_j+1} (\Theta_{ji}^{(l)})^2$

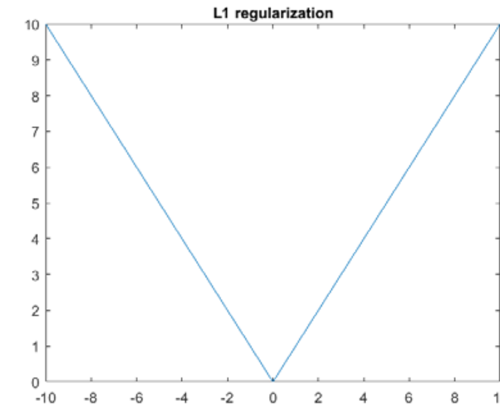
- Derivative $$\frac{\partial J}{\partial \Theta_{ji}^{(l)}} = \frac{\partial J_i}{\partial \Theta_{ji}^{(l)}} + \lambda \Theta_{ji}^{(l)}$$
$$\text{When } \frac{\partial J}{\partial \Theta_{ji}^{(l)}} = 0, \quad \Theta_{ji}^{(l)} = -\frac{1}{\lambda} \frac{\partial J_i}{\partial \Theta_{ji}^{(l)}}$$




- Keep the weights small unless they have big derivatives
- Tends to prefer many small weights

# L1 regularization

- Cost function $J(\Theta) = J_i + \lambda \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{,j}+1} |\Theta_{ji}^{(l)}|$




- L1 regularization has the effect that many weights are set to zero (or close to zero).
- The effect of setting many weights to zero and keeping a few large weights is feature extraction – select only some of the input connections.
- For deep learning, this often does not work as well as L2-regularization.

# Maxnorm regularization

- L1 and L2 regularization penalize each weight separately.
- An alternative is to constrain the maximum squared lenght of the incoming weight vector of each unit.
- If an update violates this constraint, we scale down the vector of incoming weights to the allowed length.
- When a unit hits it’s limit, the effective weight penalty of all of it’s weights is determined by the big gradients.
  - This is more effective than a fixed penalty at pushing irrelevant weights towards zero.
- Some claim that this method is better than L2-regularization.

# Regularization by early stopping

- Another kind of regularization is early stopping: stopping before the model can overfit the training data
- Remember that we initialize the weights to small random numbers.
- As training progresses (without batch normalization), the weights can grow. Too large weights often leads to overfitting.
- We can monitor the training and validation accuracy, and stop when the validation accuracy increases systematically over several steps.

# Regularization by data augmentation

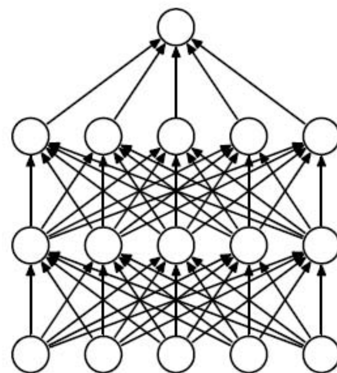
- Given a finite data set, we can make the net generalize better by adding noise to the data.
- For image data it is common to simulate larger data sets by affine transforms to
  - Shift
  - Rotate
  - Scale
  - Flip
- See e.g. https://keras.io/preprocessing/image/

# From pattern recognition: bagging

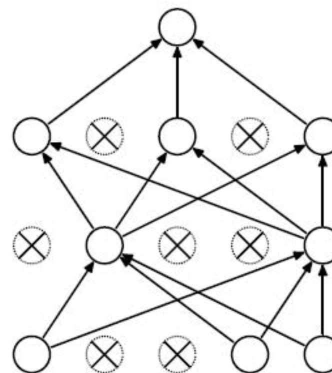
- Bagging (bootstrap aggregating) is a technique for reducing generalization error by combing several models (e.g. classifiers) training on different data subsets.
- Different subsets (minibatches) of data will normally **not** result in the SAME errors on the test set.
- The idea is to train D models and average the predictions/class estimates by taking the most frequent class among the predictions.
- This is not practical for large nets because we have to train D times.

# 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.

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.

Example: cat class with nodes detecting

- Eyes
- Ears
- ~~Tail~~
- Fur
- Legs
- ~~Mouth~~

# 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
```

# Bias regularization

- For linear classification it is important NOT to regularize the bias parameters.
- For large nets, the effect of regularizing the bias terms if often neglible, given proper preprocessing.

## Bias initialization

- It is common to initialize the biases to zero, symmetry-breaking will be done by the small random weight initialization.

# Recommendations for regularization

- Use L2 regularization
- Use Dropout with p=0.5 (p can be tuned on validation data).

# Repetition: Batch gradient descent

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

- 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.

# Repetition: 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*
- Common parameter value: 32, 64, 128.

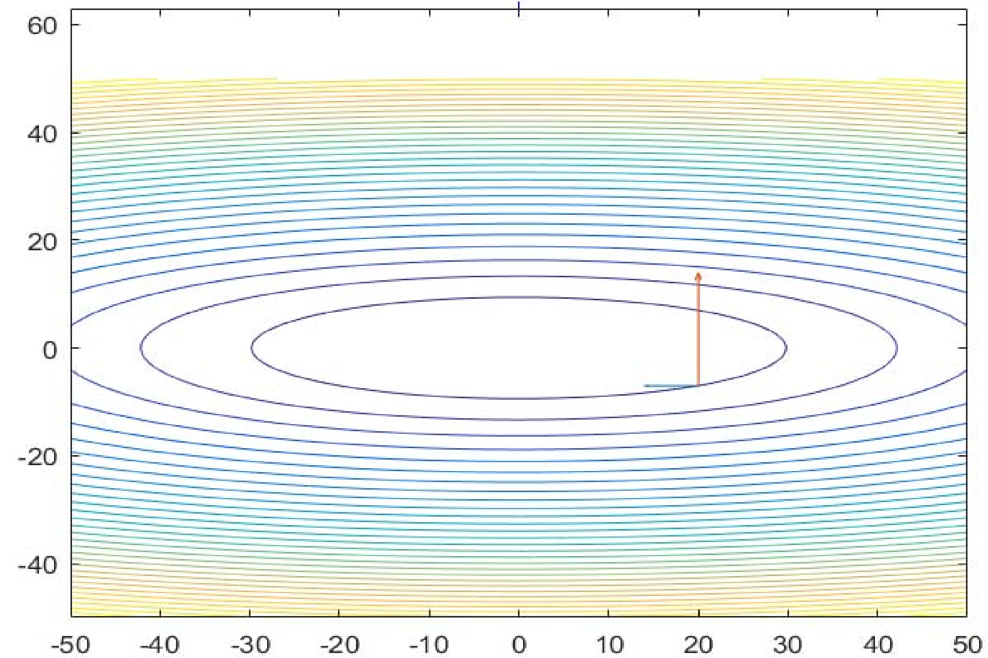
# Learning with minibatch gradient descent

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

# Learning with minibatch gradient descent

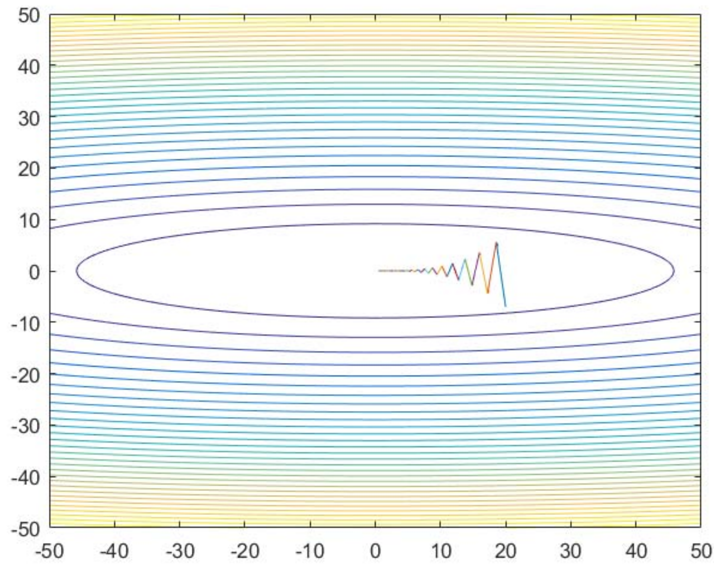
- Setting the learning $\eta$ rate is difficult, and the performance is sensitive to it.
  - Too low: slow convergence
  - Too high: oscillating performance
- In practise when using minibatch gradient descent: decay the learning rate linearly until iteration $\tau$, then leave $\eta_\tau$ constan$_t$: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

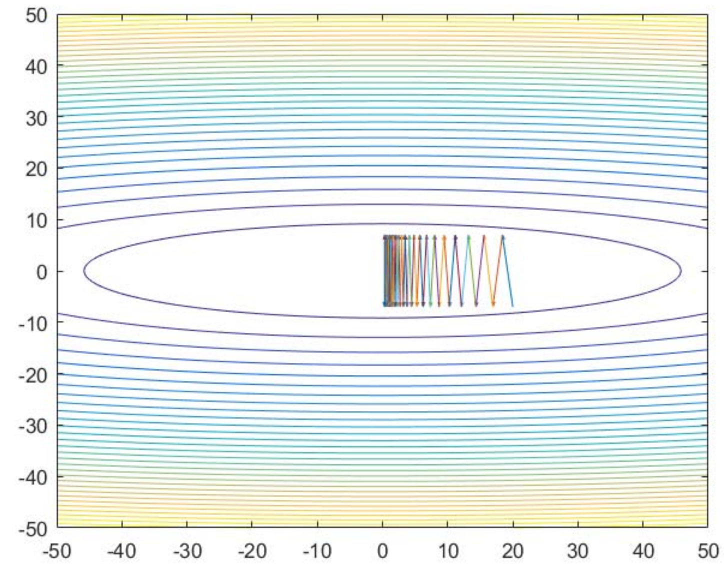
# Gradient descent oscillations




Horisontal gradient small, vertical gradient big.
In which direction do we want to move?

# Gradient descent oscillations




$\eta$ =0.19

$\eta$ =0.20

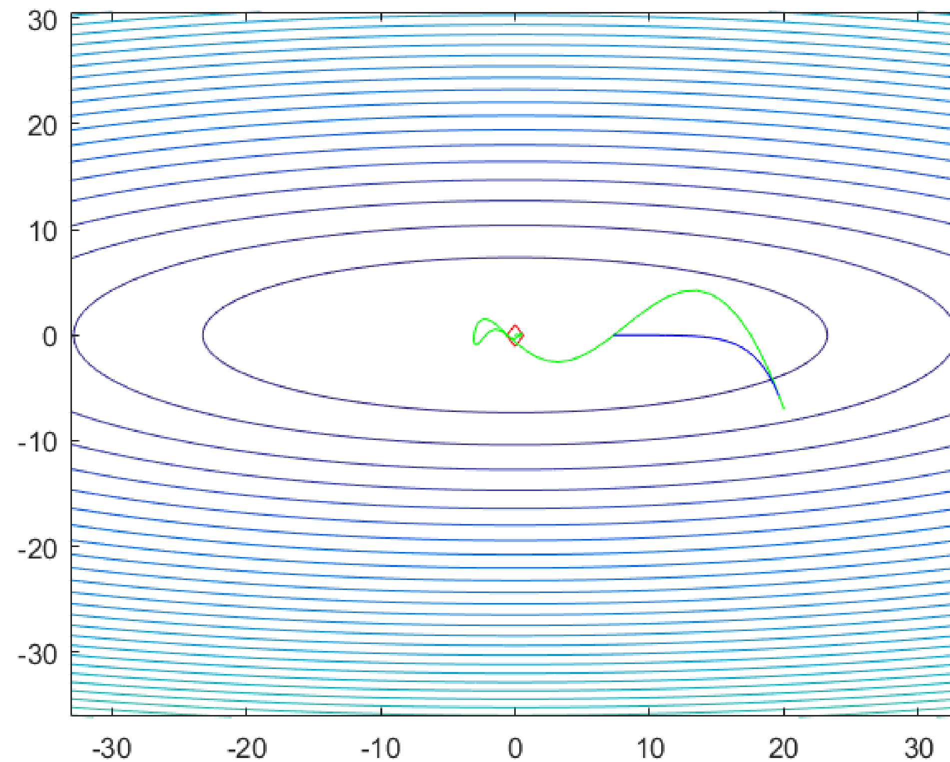
This is how gradient descent moves

# 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.

# 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.

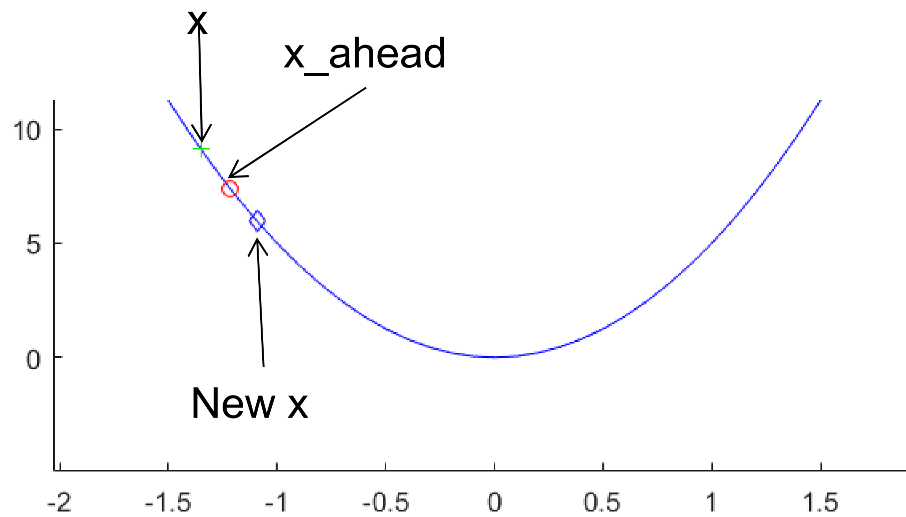
# Nesterov momentum

- 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)

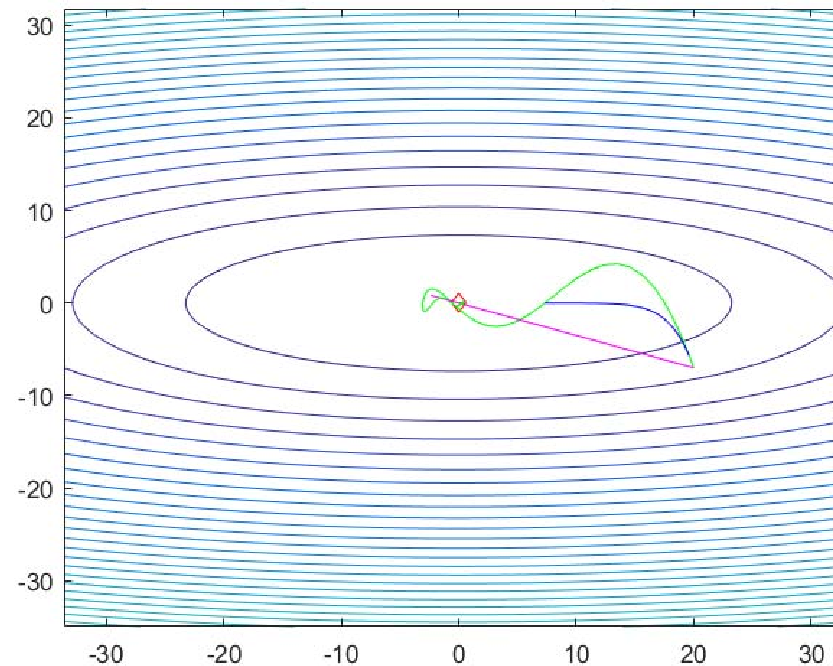
```
x_ahead = x + mu*v #Only the velocity part
# Evaluate the gradient at x_ahead
v = mu*v – learning_rate*dx(x_ahead)
x += v
```

# 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

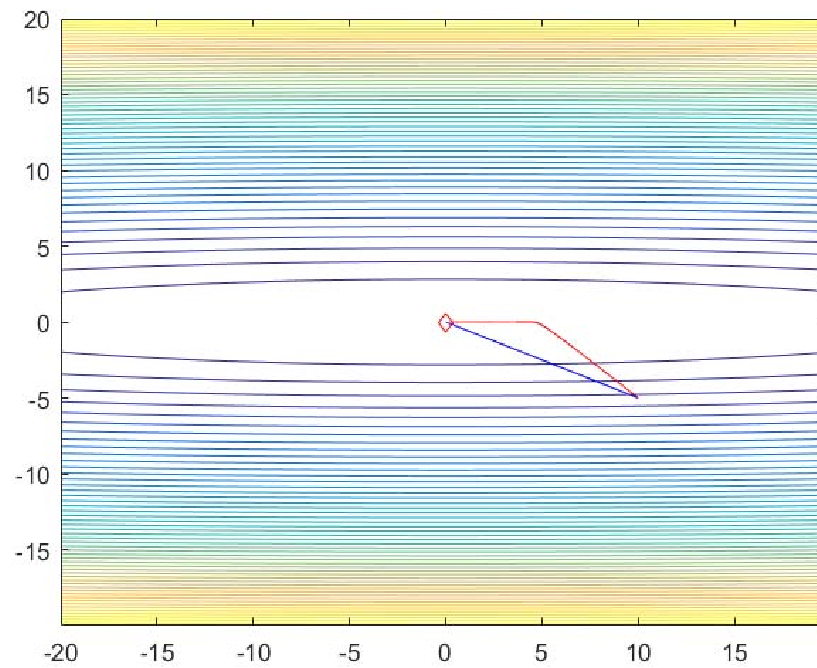
# RMSprop update

- 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!

- If we use gradient descent with learning rate $\varepsilon$,the new point will be ($\mathbf{x}_0$- $\varepsilon$ $\mathbf{g}$).
- Substitute this: $J(\mathbf{x}_0 - \varepsilon\,\mathbf{g}) \approx J(x_0) - \varepsilon\,\mathbf{g}^T\mathbf{g} + \frac{1}{2}\varepsilon^2\mathbf{g}^T\mathbf{H}\mathbf{g}$
- This will add $-\varepsilon\,\mathbf{g}^T\mathbf{g} + \frac{1}{2}\varepsilon^2\mathbf{g}^T\mathbf{H}\mathbf{g}$ to the cost
- This term will be ill-conditioned when $\frac{1}{2}\varepsilon^2\mathbf{g}^T\mathbf{H}\mathbf{g} > -\varepsilon\,\mathbf{g}^T\mathbf{g} +$

- To check how the learning rate performs, we can monitor the gradient norm $\mathbf{g}^T\mathbf{g}$ and the term $\mathbf{g}^T\mathbf{H}\mathbf{g}$
- Often, when learning is slow, the gradient norm $\mathbf{g}^T\mathbf{g}$ does not shrink, but $\mathbf{g}^T\mathbf{H}\mathbf{g}$ grows
- If this is the case, the learning rate must be shrunk.

# Second-order methods and their limitations (DL 8.6)

- Newton's method would update x as:

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

- 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.

# Local minima for deep nets (DL 8.2)

- The most common challenge in general optimization is that we end up in a local minima.
- This is not a common problem for deep nets – why?
  - The weight space is symmetric, we can get an equivalent model by exchanging e.g. incoming weight for unit i with incoming weight for unit j, and swap the output weights correspondingly. This is called model identifiability.
  - Other kinds of identifiability occur when we scale a ReLU input and output weights correspondingly.
  - There are local minima, but we often end up with approximately the samle value of J.
- Be careful to assume that a local minima is your problem with a deep net.
  - Monitor the gradient norm. If it is not small, you are not in a local minima.
  - In addition, other structures can have local minima, as plateaus or saddle points.

# Training neural nets - summary

Elements to consider:

- Network architecture/layers/nodes
- Activation functions
- Loss function
- Data preprocessing
- Weight initialization
- Batch normalization
- Dropout and other types of regularization
- Mini-batch gradient descent update schemes
- Training, validation, and test sets
- Searching for the best parameters
- Monitoring the learning process

# Activation function and loss function

- Use ReLU for hidden layers.
- If assigning an image to ONE class: SOFTMAX loss
- If multiple labels possible (e.g. this image contains a cat and a car): Logistic loss one-vs-all.

# Preprocessing: 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

- 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)

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: training

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

**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: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$

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

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

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

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

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

# Optimizing hyperparameters

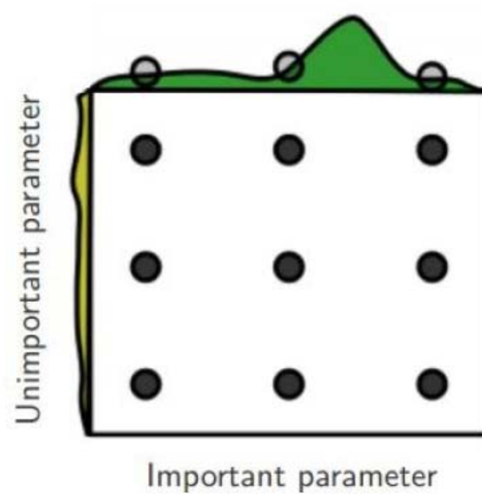
- **Training data set**: part of data set used in backpropagation to estimate the weights.
- **Validation data set** (mean cross-validation): part of the data set used to find the best values of the hyperparameters, e.g. number of nodes and learning rate.
- **Test data**: used ONCE after fitting all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

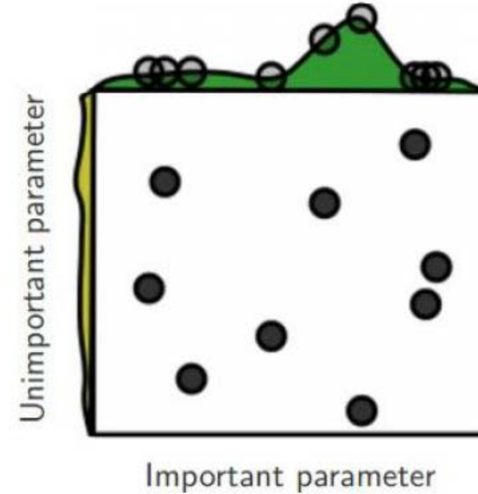
# Search strategy: coarse-to-fine

- 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
- Random sample the grids

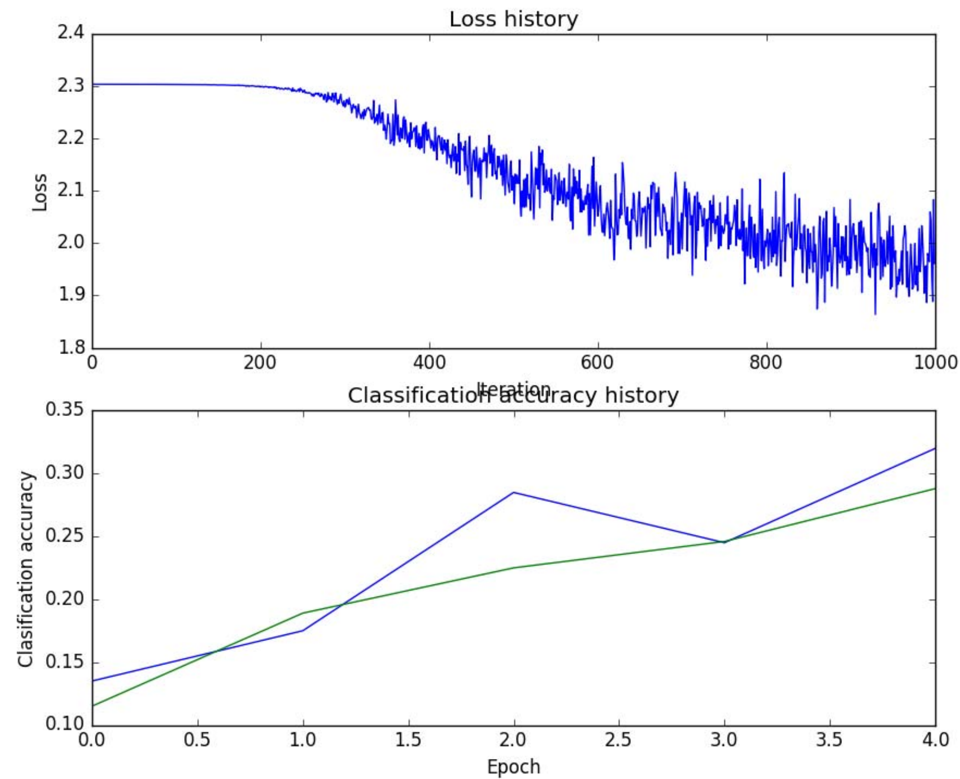
# Consider a random grid

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



# Monitor the loss function



# Regularization

- Use L2 regularization
  - Consider trying maxnorm
- If training from scratch on a deep net: use data augmentation
- Use Dropout

# Minibatch gradient descent update schemes

- Recommendations:
  - Gradient descent with Nesterov momentum
  - RMSprop
  - ADAM

- Careful monitor the loss function, take care in choosing the learning rate.