

Training neural networks

# Training

- Can I feed raw data into the model?
- Where should I start the training?
- How can I monitor my training?
- How should I optimize my hyperparameters?

# Data normalization

$$\frac{\partial f}{\partial \mathbf{w}} = \frac{\partial f}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \mathbf{w}}$$

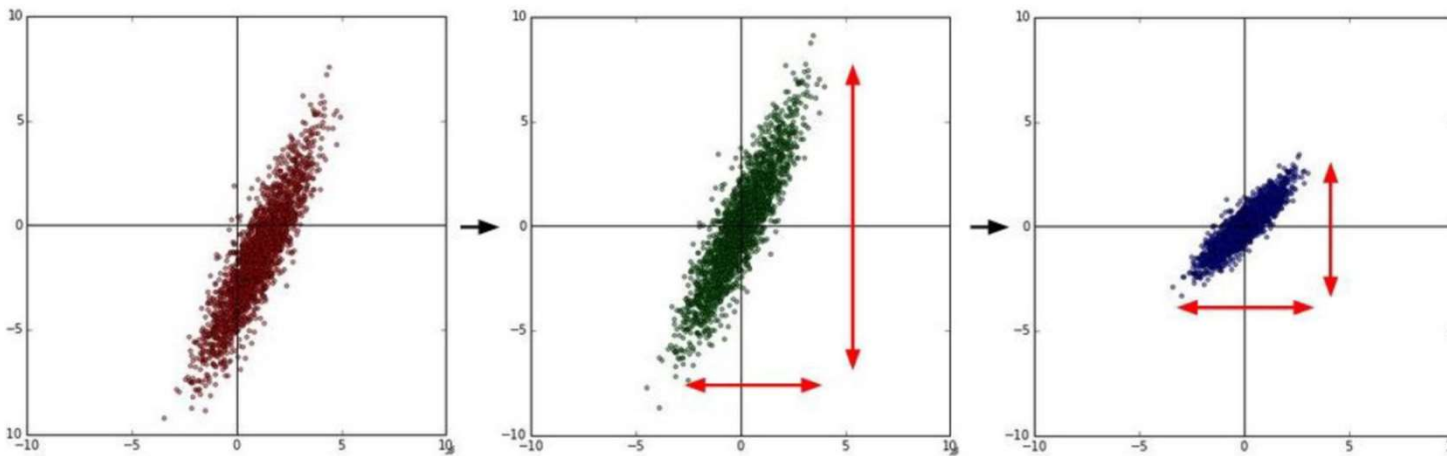
Original data

Zero-mean data

Normalized data

$$\frac{\partial \mathbf{q}}{\partial \mathbf{w}_n} = \mathbf{x}_n$$

What happens to the gradient when input data is all positive?

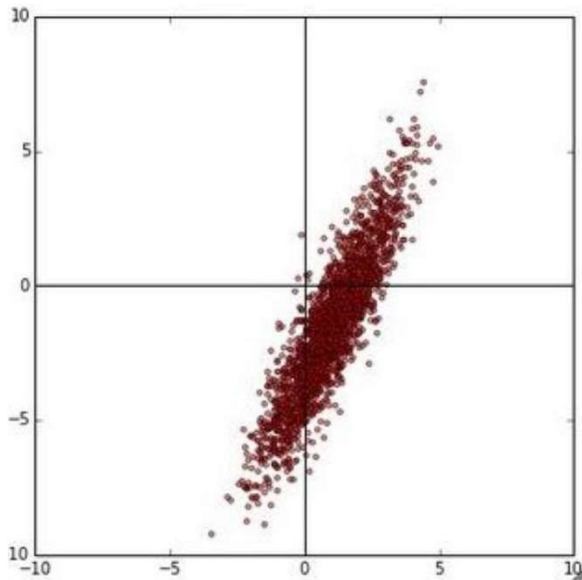


# Data normalization

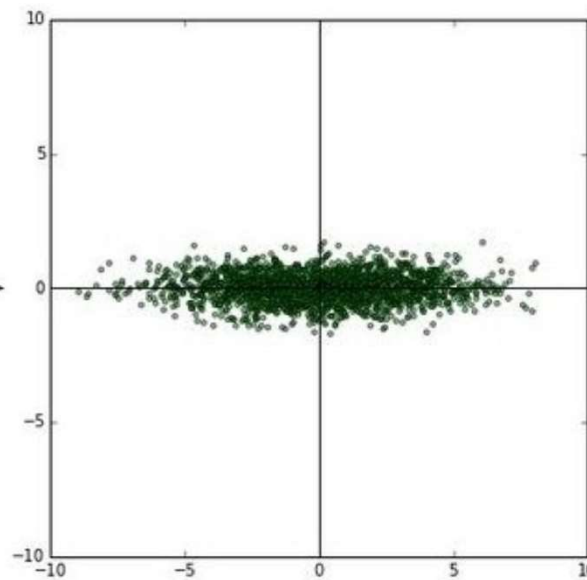
How do we deal  
with normalization  
at test time?

- Principal Component Analysis (PCA) or whitening

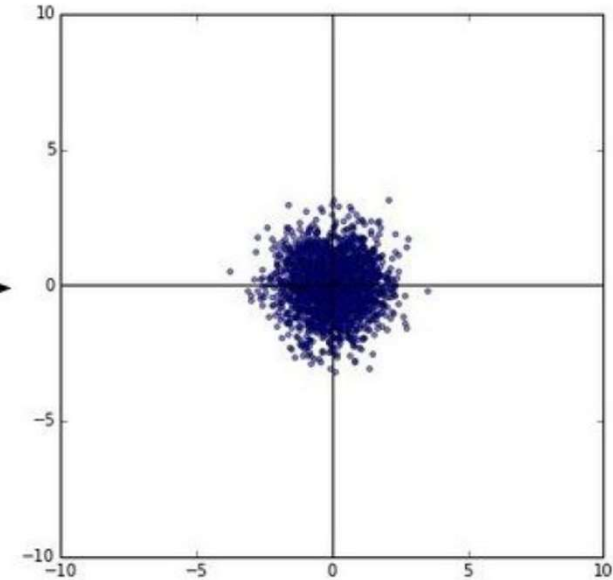
Original data



De-correlated data

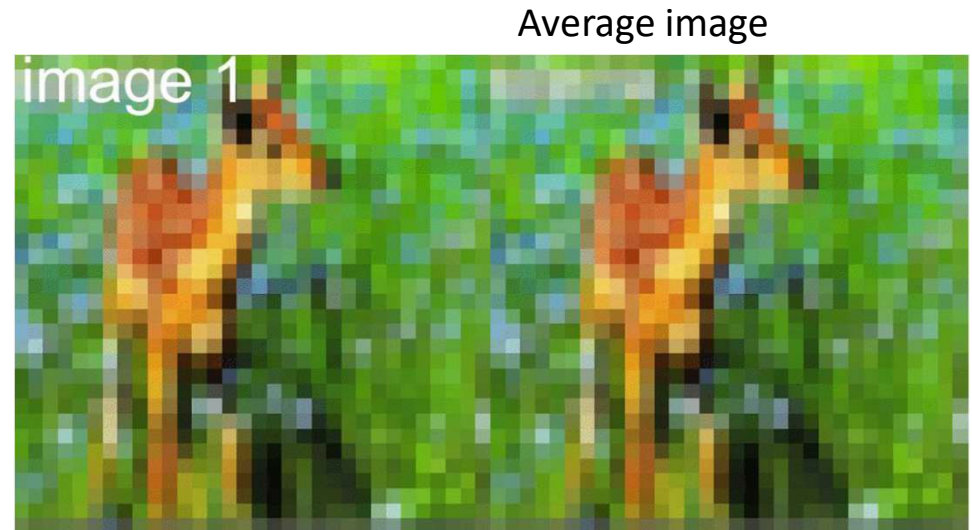


Whiten data



# Data normalization

- Subtract mean image
  - Mean over all images (in the training set!)
  - $M \times N \times 3$  for RGB images
- Subtract mean along channel
  - Values within a channel quite similar
  - $1 \times 3$  vector (with average R, G and B values)
- You don't need to actually do this over all the images
  - Law of large numbers --> for sufficiently large subset of M samples:  $u_m \approx u_{total}$



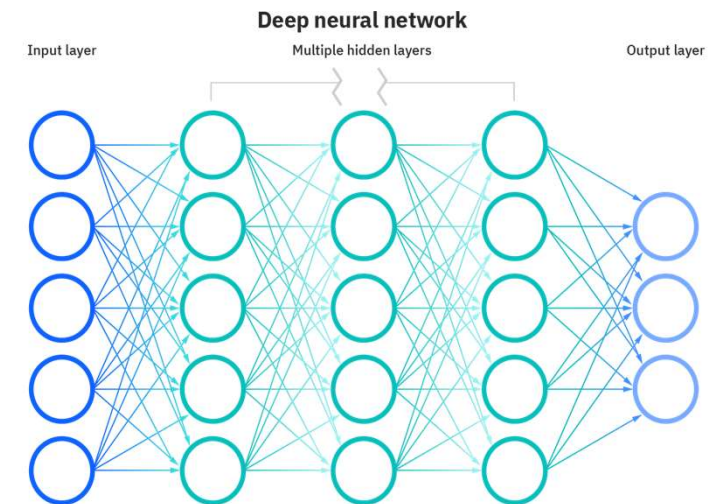
sometimes we don't need to use std to normalize, why?  
because std doesn't matter, it's just a scale. can be changed by adjusting weights

Why not mean over ALL data?

Does this solves the sigmoid problem?

# Weight initialization

- Before we can start descending in our loss landscape, we need to pick a starting point
- Option 1: start with all zero weights & biases.
  - Good idea?
  - NO: output will be zero!
- Option 2: start with all non zero weights, but same
  - Good idea?
  - NO: if all neurons compute the same output, they all have the same influence on the loss, and will follow exactly the same gradient updates.
- Option 3: Initialize with small random numbers



# Weight initialization

- Initialize with small random numbers
- Sample weights from scaled Gaussian
  - E.g. zero-mean, 1e-2 std dev.
- Works OK for small networks...
- But problems for deeper networks
- Example: 10-layer network with 500 neurons in each layer and tanh activation functions

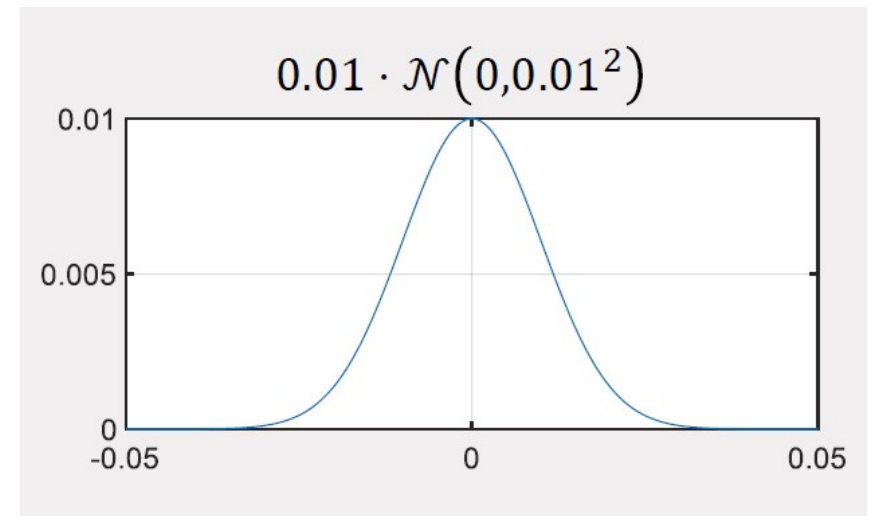
Formula

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

$f(x)$  = probability density function

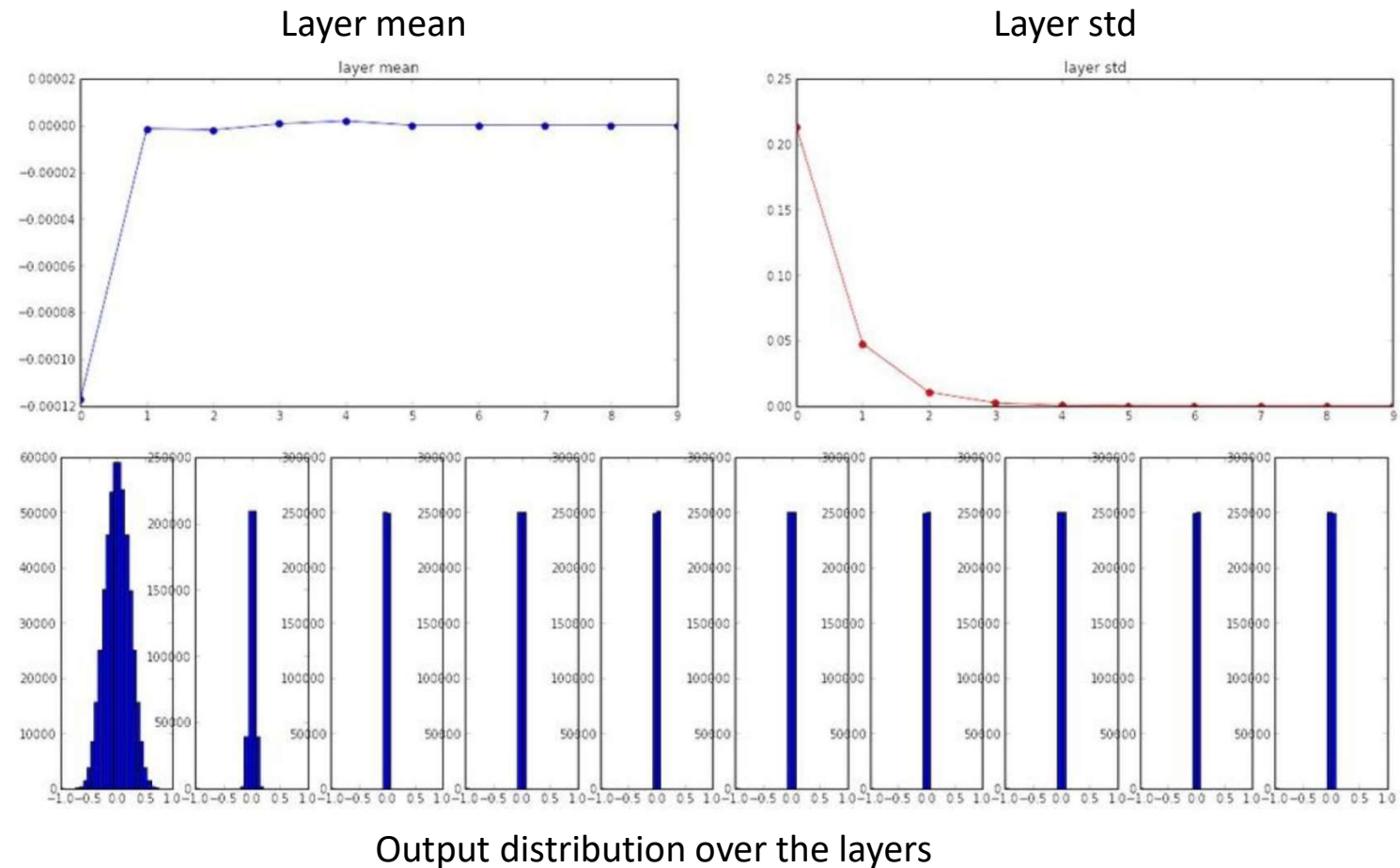
$\sigma$  = standard deviation

$\mu$  = mean



# Weight initialization

- **Observation:**
- Stddev quickly collapses to zero when we go deeper in the network
- What happens during forwardpass?
  - *Weights small*
  - *Input gets multiplied by a small number each layer*
  - *Becomes smaller and smaller with each layer*

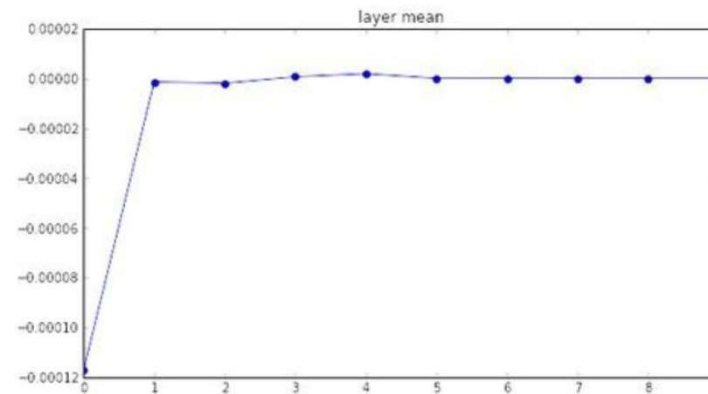




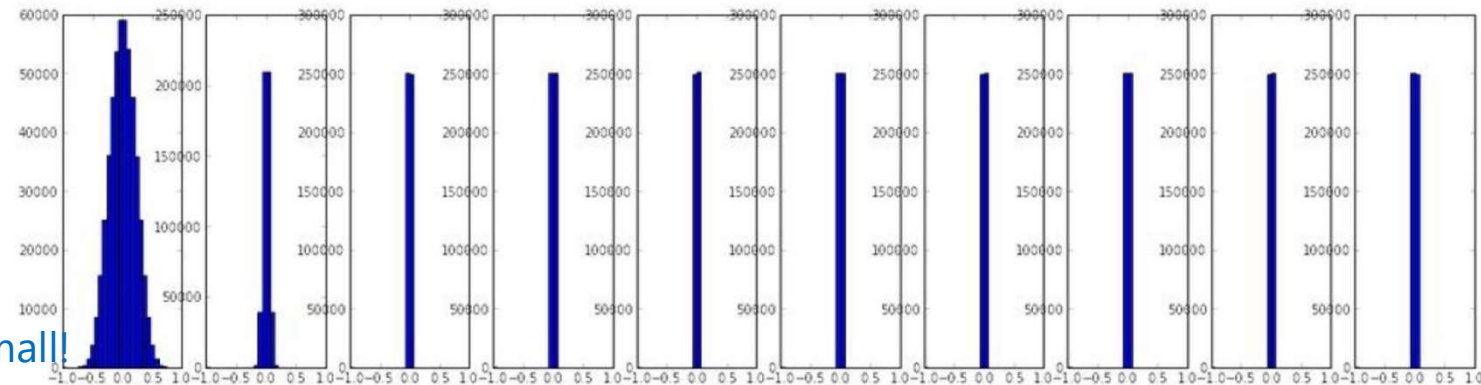
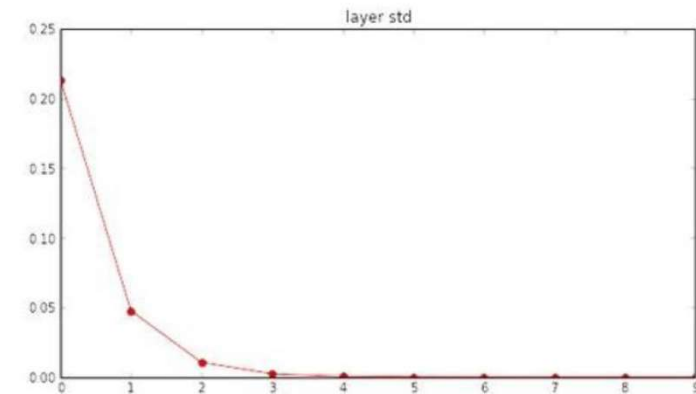
# Weight initialization

- **Observation:**
- Stddev quickly collapses to zero when we go deeper in the network
- What happens during forwardpass?

Layer mean



Layer std



Output distribution over the layers

Weight updates

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

= x

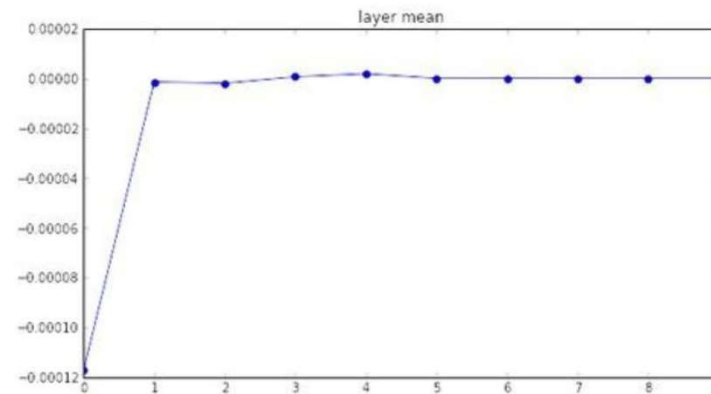
Will be very small!

Basically no updates!

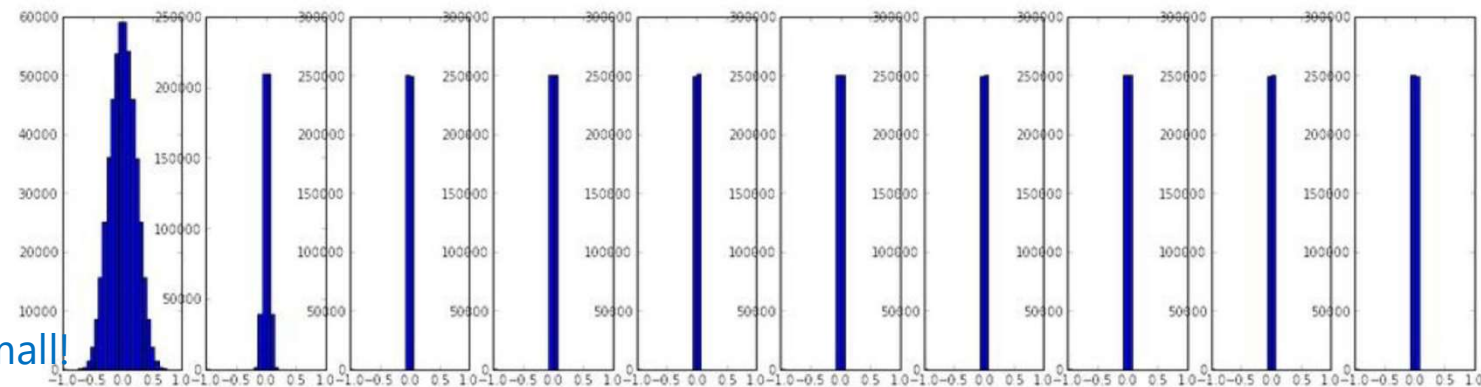
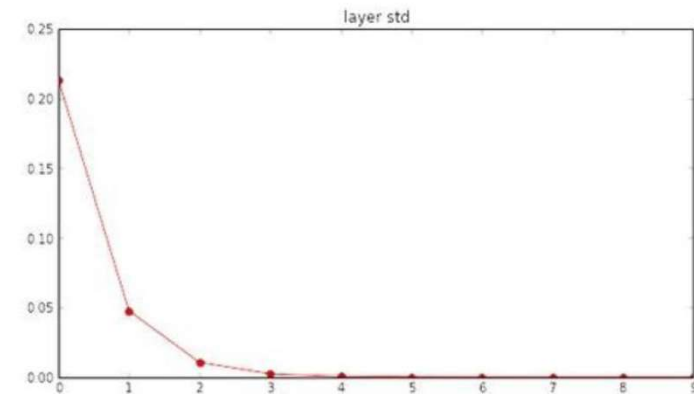
# Weight initialization

- **Observation:**
- Stddev quickly collapses to zero when we go deeper in the network
- What happens during forwardpass?

Layer mean



Layer std



Output distribution over the layers

Backflow gradient

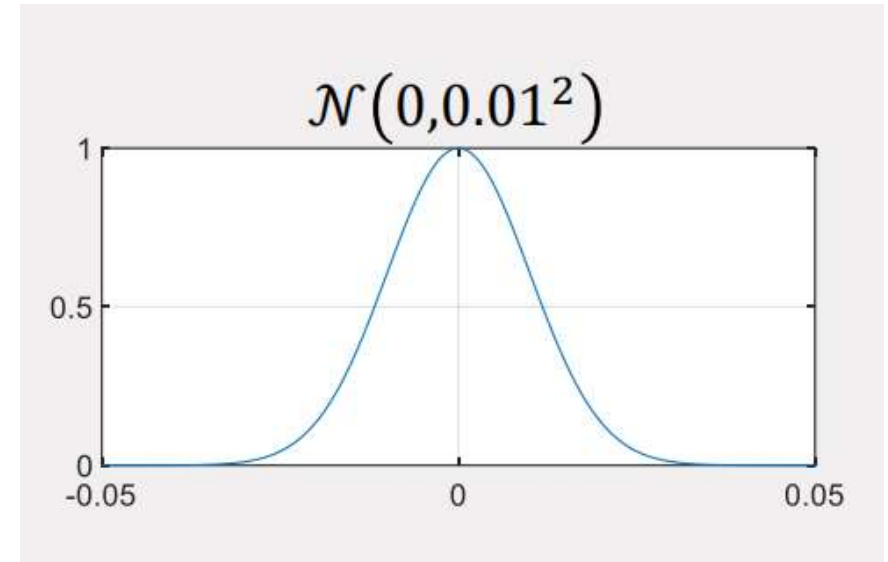
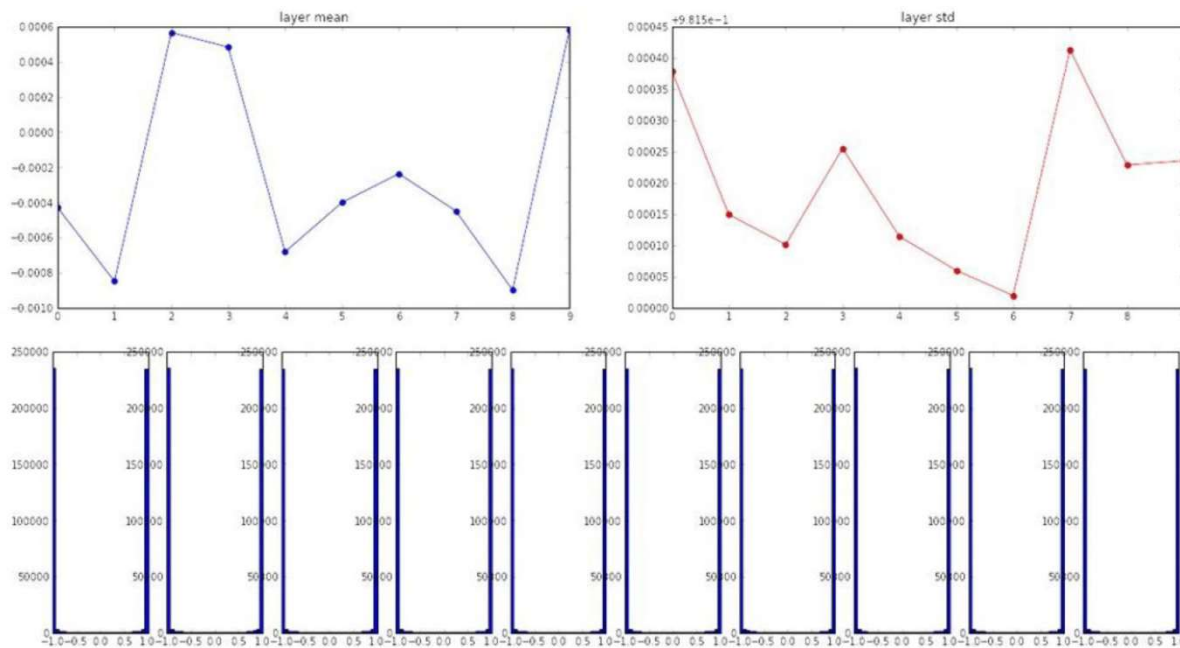
$$\frac{\partial f}{\partial p} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial p}$$

$= w$  Will be very small!

Small backflowing gradient

# Weight initialization

- **Option 3: Let's try with big weights!**



- **Observation**
  - Almost all neurons saturate
  - *Output either -1 or +1*
  - Gradients all become zero!

# Weight initialization

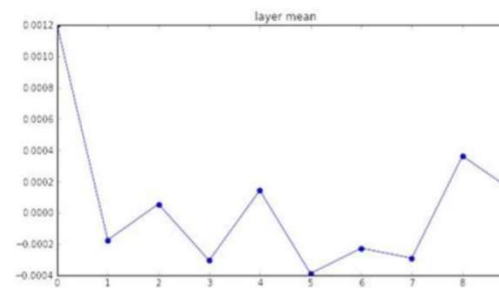
- **Weight initialization is not trivial!**
- When they're zero you get inefficient updates...
- When they're too small the layer outputs all collapse...
- When they're too large the layer outputs all saturate...

- **Reasonable solution:**

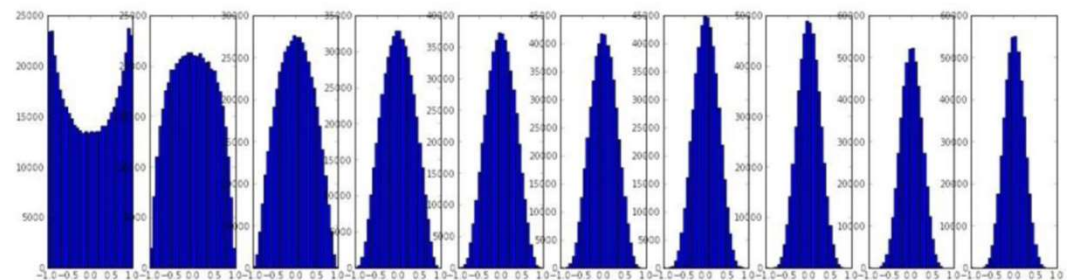
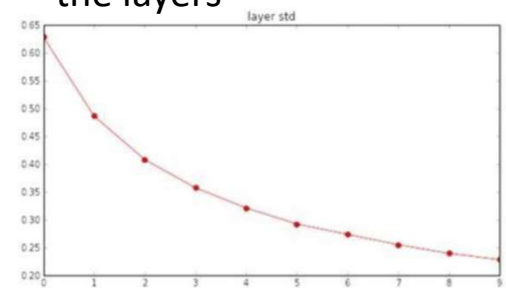
- Xavier initialization
  - Enforce  $\sigma_{in} = \sigma_{out}$

Scale  $x$  by  $\alpha = \sqrt{1/n}$

Layer mean



Output distribution over the layers



Output distribution over the layers

# Monitoring the learning process

- **Start training!**

- Use small regularization factor

- Find an appropriate learning rate

- *Start with a small learning rate... (e.g.  $1e-6$ )*
- *Try a big learning rate... (e.g.  $1e+6$ )*
- *Adjust learning rate down again... (e.g.  $1e-3$ )*

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

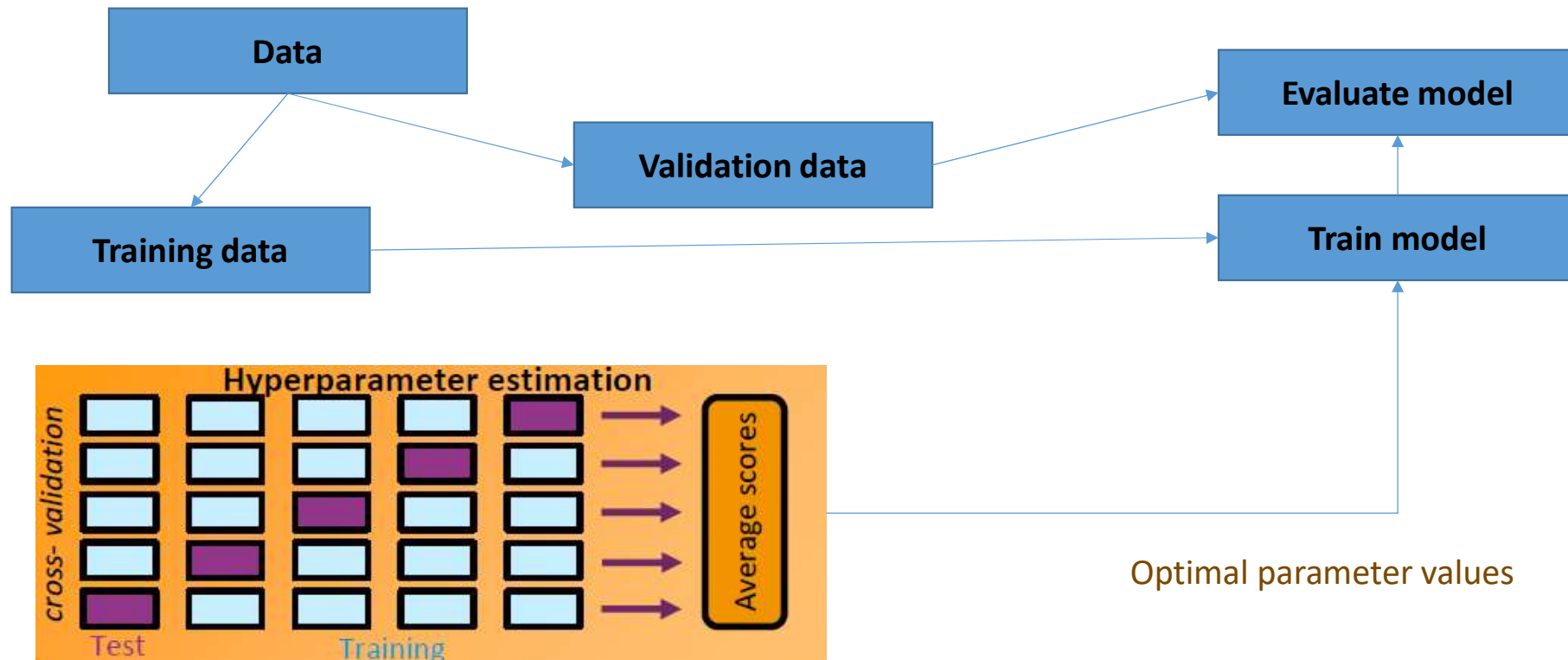
Cost infinity → Loss still exploding

- **Finding the right learning rate:**

- Loss not going down → Learning rate too low
- Loss exploding → Learning rate too high
- Hyper-parameter optimization

- Cross-validate [on the training data] for a certain range of different learning rates to find an optimal value.

# Hyper parameter optimization



Repeat for N different settings

# Hyper parameter optimization

- **Finding the parameter range**

- Start with a coarse range

- *Few epochs*
- *Rough estimate for the optimal values*
- *Example:*
  - $lr = 10^x, x \in -6, -5, \dots, -3$
  - $reg = 10^x, x \in -5, -4, \dots, +5$

*Typically sample in log-space*

val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)

Optimal cross-validation results

$$L = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \underbrace{\lambda R(W)}_{= 0}$$



# Hyper parameter optimization

## Optimal cross-validation results

- Finding the parameter range
- Finer grid search in optimal range
  - Longer running time
  - Finer search
  - Example:
    - $lr=10^x, x \in -3-4$
    - $reg=10^x, x \in -4, -3, \dots, 0$

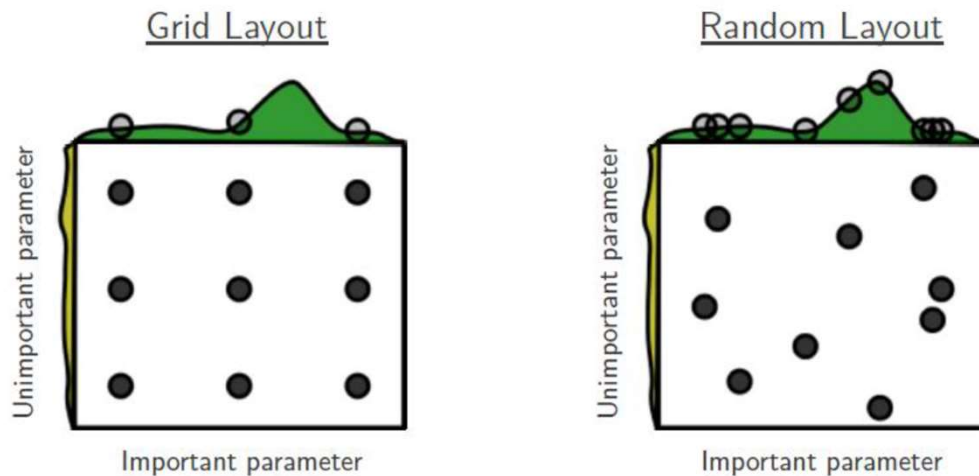
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)

*These are just two parameters, there are typically much more values to be set!*



# Hyper-parameter optimization

- **Search range for hyper-parameter optimization**



*Courtesy Bergstra & Bengio, JMLR, 2012*

- Reasoning:
- ➤ Typically, low correlation between different hyperparameters.
- ➤ Hence, random sampling offers more information.

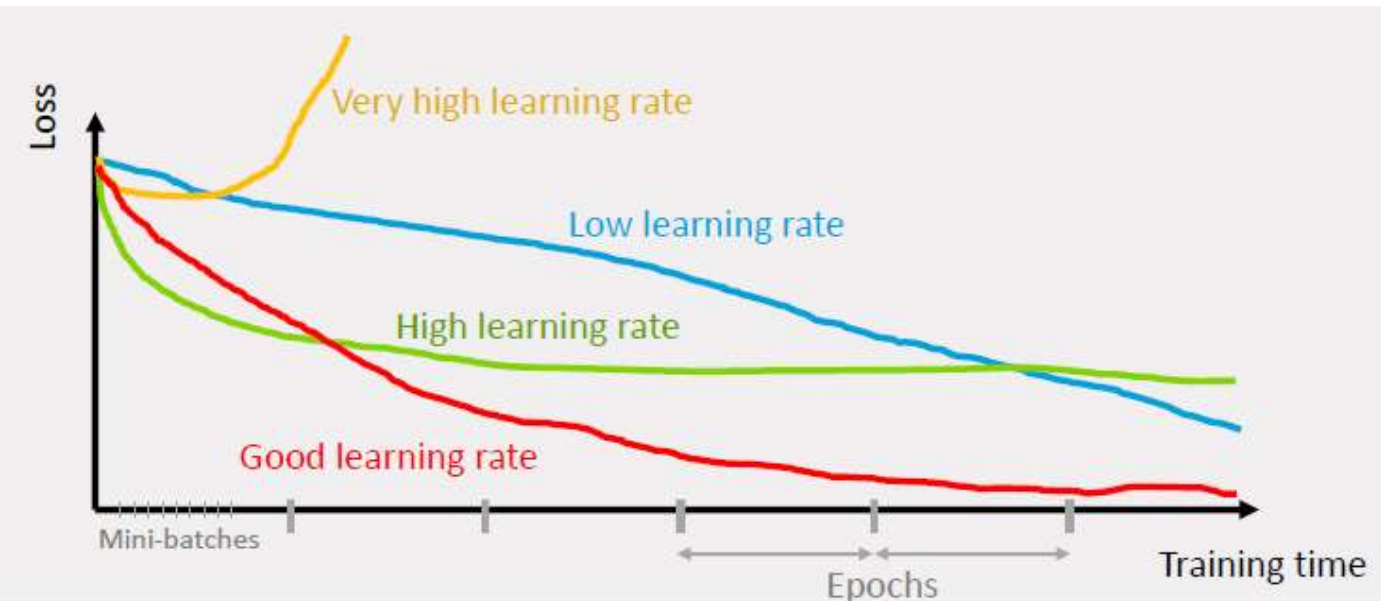
Another reason for random search?

grid search: computationally heavy and the computational complexity grows rapidly if parameters are a lot

# Loss curves

too high  $Lr$ : if it goes to  $\infty$ , then everything (gradients, ...) all goes to  $\infty$ , then it won't be able to come down again

- Why can loss curves go up?
- (>2 reasons)



- 1.  $Lr$  too high
- 2. there's some problem with the network
- 3. loss function wrongly defined
- 4. stochastic gradient descent: depends on the batch. maybe this will lead to loss goes up a bit.

# Summary

## Training convnets!

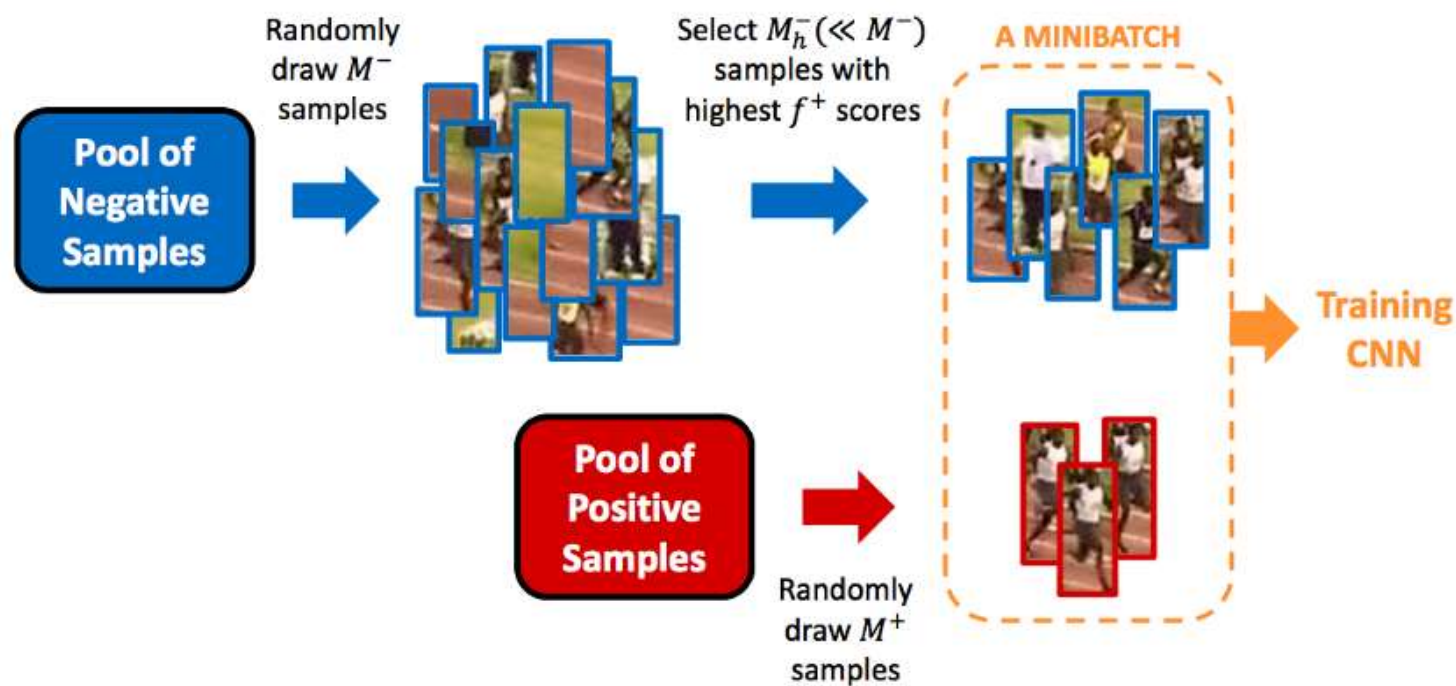
- Can I feed raw data into my network?
- *You could, but it's better to normalize data!*
- Where should I start in my loss landscape?
- *Xavier or He initialization to avoid vanishing gradients*
- How can I see if the training goes well?
- *Check the training and validation loss after each batch/epoch*
- How should I pick my hyperparameters?
- *Coarse search / fine search, use random sampling across parameter space*
- How can I achieve more robustness against poor initialization?
- *Batch norm! Normalize before feeding into activation function.*

## What's next?

- How can we move down the loss landscape more efficiently?
- Fancy tricks with adjusting the learning rate
- Using multiple models: ensembles
- How should we regularize our model?
- Are there other methods to apply some regularization
- Can we use convnets for (relatively) small data sets?

# Hardsample mining

- Hard sample mining is a tried and true method to distill a large amount of raw unlabeled data into smaller high quality labeled datasets.
- *A hard sample is one where your machine learning (ML) model finds it difficult to correctly predict the label.*



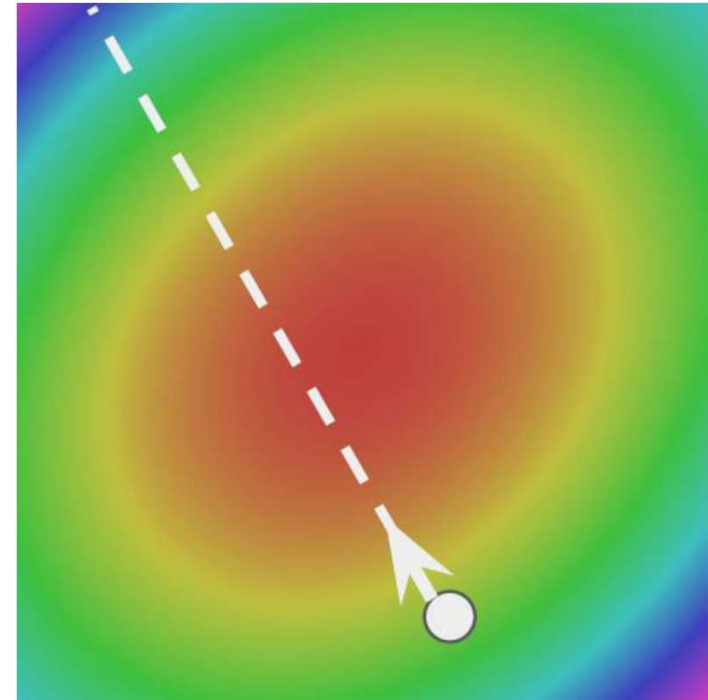
Example of hard negative mining (from Jamie Kang [blog](#))

# Hardsample mining

- Code example

# Optimization

- Loss is a function of weights
- Evaluate the loss function of a certain set of weights
- Position in the loss landscape
- Use gradient to move down in the loss landscape
- GOAL: find the lowest point
- ➤ Find the weights that minimize the loss



Note that the loss landscape slightly changes after each step!

# Optimization

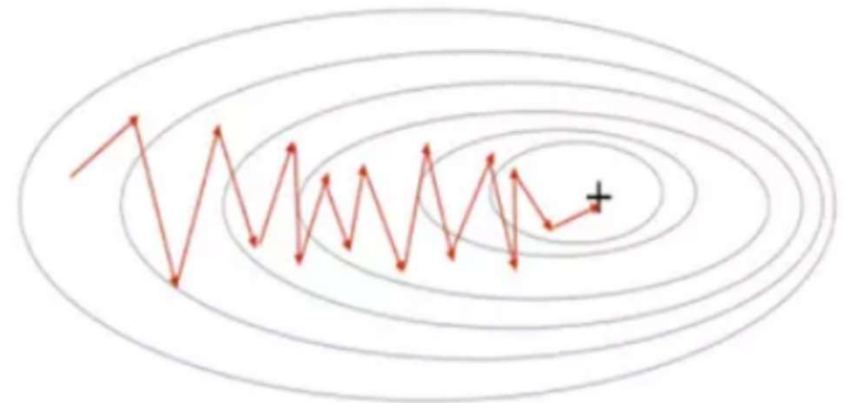
More or less of a problem in higher dimensions ?

- Problems with Stochastic Gradient Descent (SGD)
- What happens if the loss landscape changes much faster in one direction than another?
- Direction of gradient does not align with the direction of the minimum!

Very slow progress  
along slow moving  
dimension

Jitter along steep  
dimension

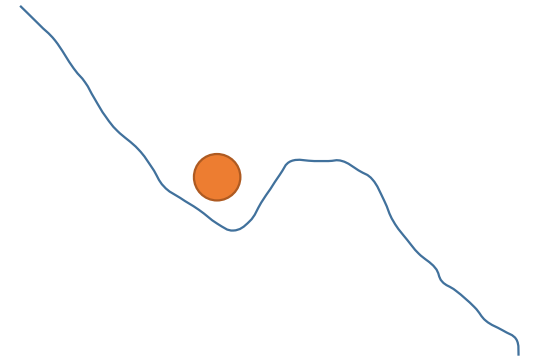
Stochastic Gradient Descent





# Optimization

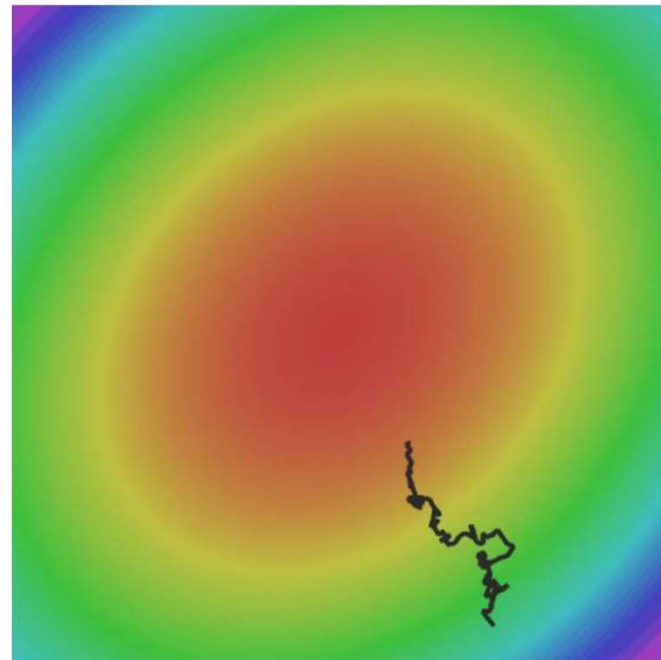
- What happens in local minima or saddle points?
- Gradient will be zero
- No movement  $\rightarrow$  we get stuck!
- In higher dimensions saddle points pose much more of a problem than local minima
- Saddle point: in some directions the loss goes up, in some directions the loss goes down
- Local minima: in all directions the loss goes up
  - Much more rare that this happens!



# Optimization

- Another problem with SGD:
- Gradients come from noisy updates
  - We don't go down directly, may take a longer time before we reach the minimum
  - Each mini-batch only yields an estimation of the “true loss landscape”

$$L_{\text{mini-batch}}(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) \approx L_{\text{total}}(W)$$
$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(f(x_i, W), y_i) \approx \nabla_W L_{\text{total}}(W)$$



# Optimization

- Solution: add momentum

- SGD

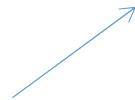
$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$



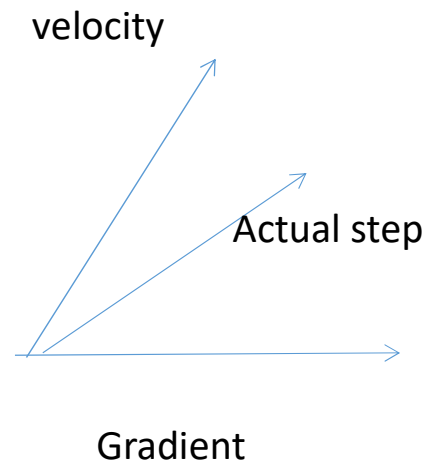
- SGD  
+ momentum

$$\begin{aligned} x_{t+1} &= x_t - \alpha v_{t+1} \\ v_{t+1} &= \rho v_t + \nabla f(x_t) \end{aligned}$$

velocity



friction



# Optimization

- RMSProp
- Keep estimate of the squared gradients
- Introduce a decay rate of this squared gradient term (what will happen when derivative is large or small?)

$$x_{t+1} = x_t - \frac{\alpha \nabla f(x_t)}{g_t^{-1/2}}$$

Update function

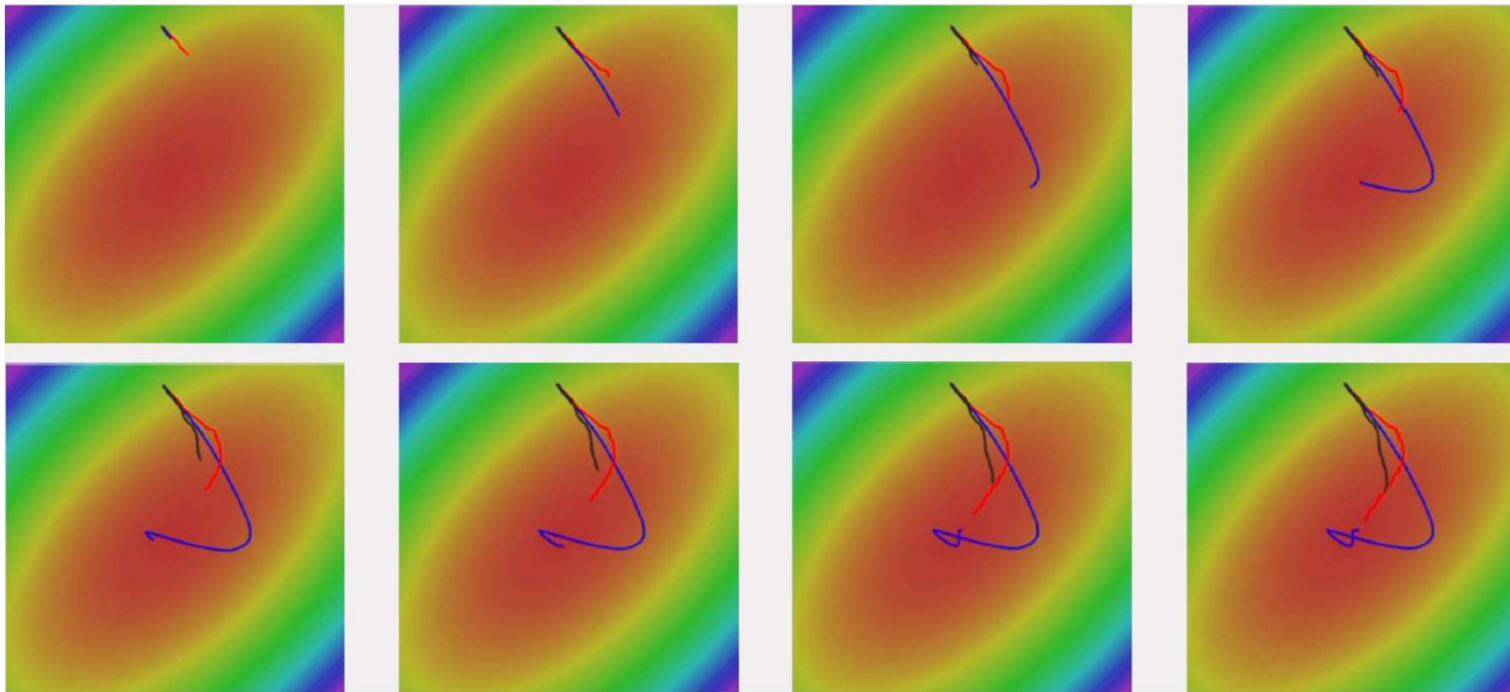
$$g_{t+1,i} = \gamma g_{t,i} + (1 - \gamma) \left( \frac{\partial L}{\partial w_i} \right)_t^2$$

Decay rate  $\gamma$

- Similar to momentum update!

# Optimization

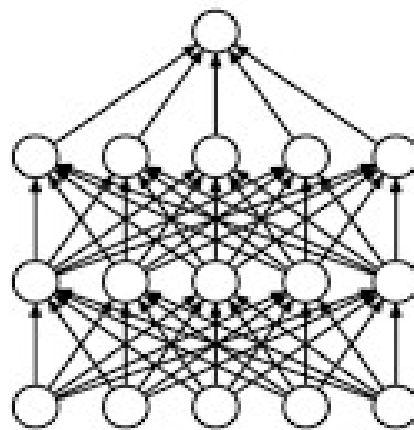
— SGD — SGD+Momentum — RMSprop



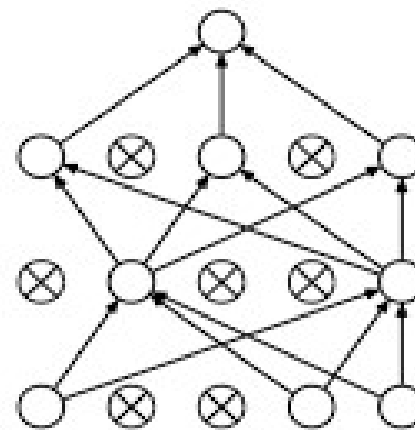
\*Example from cs231n (2017) 2017): lecture 7

# Regularization

- Dropout
- In each forward pass, randomly set some neurons to zero
- Probability of dropping is a hyperparameter (0.5 is common)



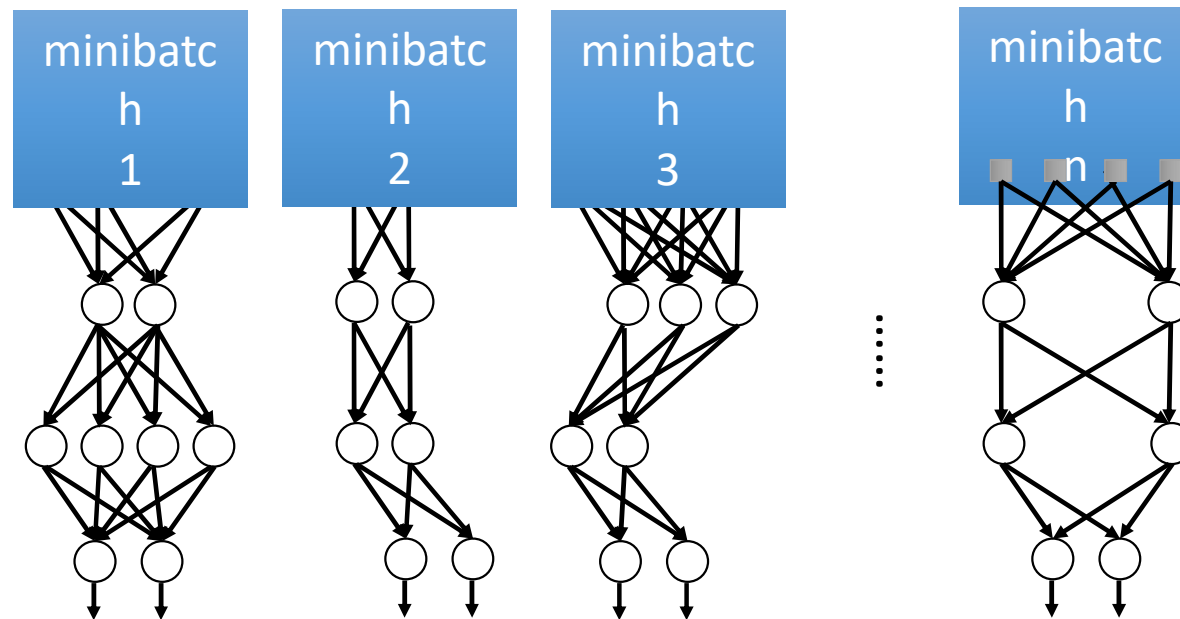
(a) Standard Neural Net



(b) After applying dropout.

# Regularization: Dropout

- Dropout is a kind of ensemble learning
  - Using one mini-batch to train one network with a slightly different architecture



# Regularization

- Dropout
- How can this possibly be a good idea?
- Seriously messing up our network!
- Interpretation 1:
- Models cannot rely too much on single visual clues as they may not always be visible...
- Distribute the prediction over different clues
- Interpretation 2:
- Drop out yields an ensemble of models within one model (sharing parameters!) Huge number of potential sub networks!!



# Regularization

At test time, multiply by dropout probability

Consider a single neuron

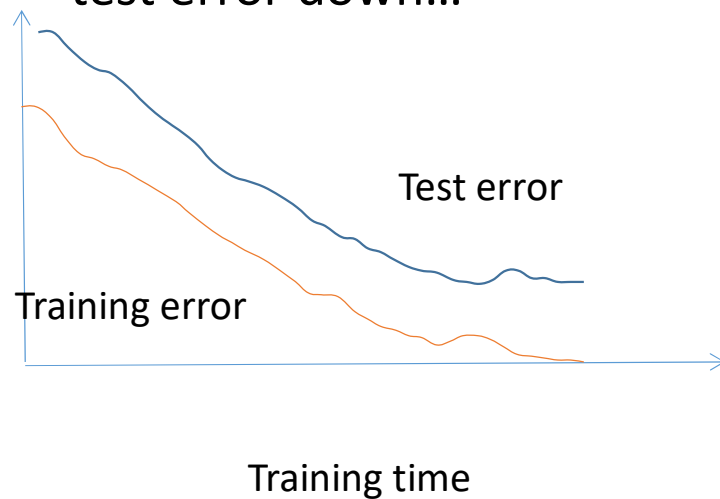
- At test time we have:  $E[a] = w_1x + w_2y$
- During training time we have:

$$E[a] = 1/4(w_1x + w_2y) + 1/4(w_1x + 0y) + 1/4(0x + w_2y) + 1/4(0x + 0y) = 1/2(w_1x + w_2y)$$

# Improve test performance

- Beyond training error

- So far we've been talking about getting training error down
- But what we really care about is getting the test error down...



How well does our model do on new, unseen data?

How can we reduce this gap?

# Improve test performance

- Model ensembles
  - Train multiple independent models
  - At test time, average their results
  - Tends to improve performance slightly
- Idea: store intermediate versions of the model and use as ensemble!

What's  
the downside of  
using model ensembles?

# Improve test performance

- Improve single models
- → Avoid overfitting / Improve generalization
- $L_2$  regularization  $R_{l2} = \sum_k w_k^2$
- $L_1$  regularization  $R_{l1} = \sum_k |w|$

# Regularization

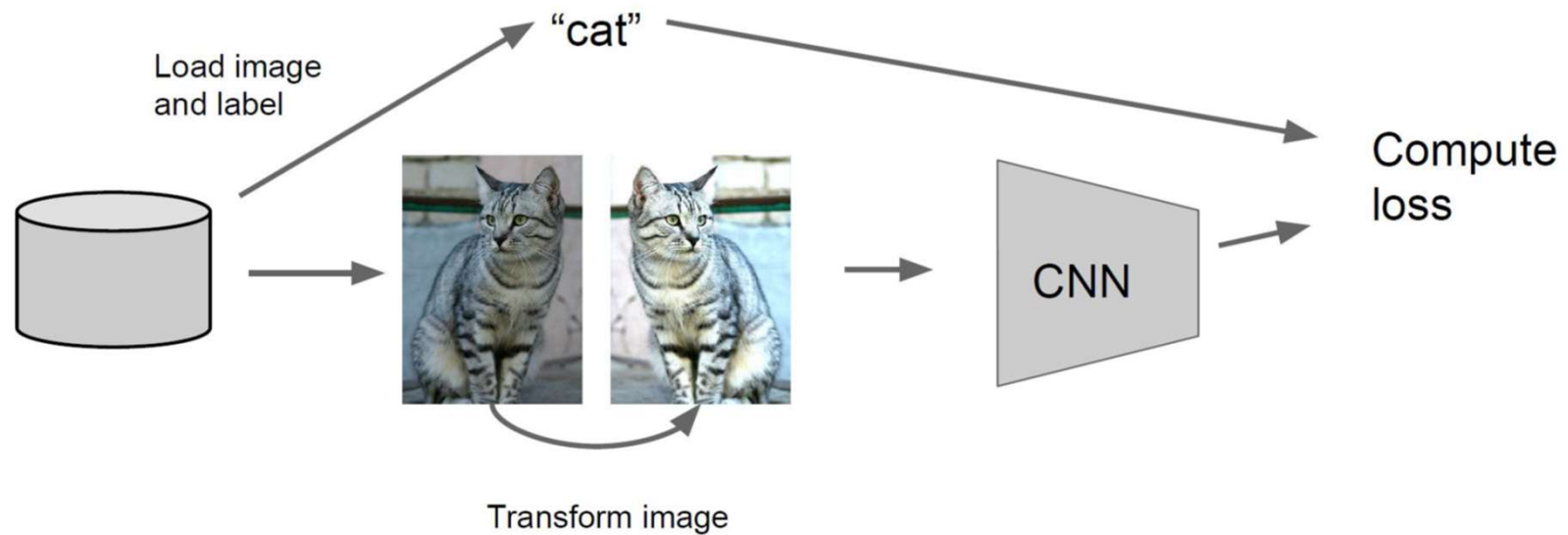
- Dropout summary
- Drop neurons with probability  $p$  in forward pass
- Scale neuron output with  $p$  at test time
- More efficient: inverted dropout
- At test time, use entire weight matrix
- At training time divide by  $p$
- Noise during training time for better regularization
- Add randomness during training to prevent it from fitting the training data too well
- Note : Similar to Batch norm! Add noise during training time, average out during test time.

What's  
the main benefit  
of this approach?

What  
do you expect  
about training time  
when using dropout?

# Regularization

- Data augmentation
















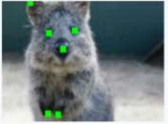











# Regularization Data

## Data augmentation

- Artificially add new samples to your data by applying random (realistic) transformations to real samples while maintaining their label

## Common types of data augmentation

- Flipping / rotating
- Stretching
- Shearing
- Cropping
- Colour jittering

	Image	Heatmaps	Seg. Maps	Keypoints	Bounding Boxes, Polygons
Original Input					
Gauss. Noise + Contrast + Sharpen					
Affine					
Crop + Pad					
Fliplr + Perspective					

<https://github.com/aleju/imgaug>

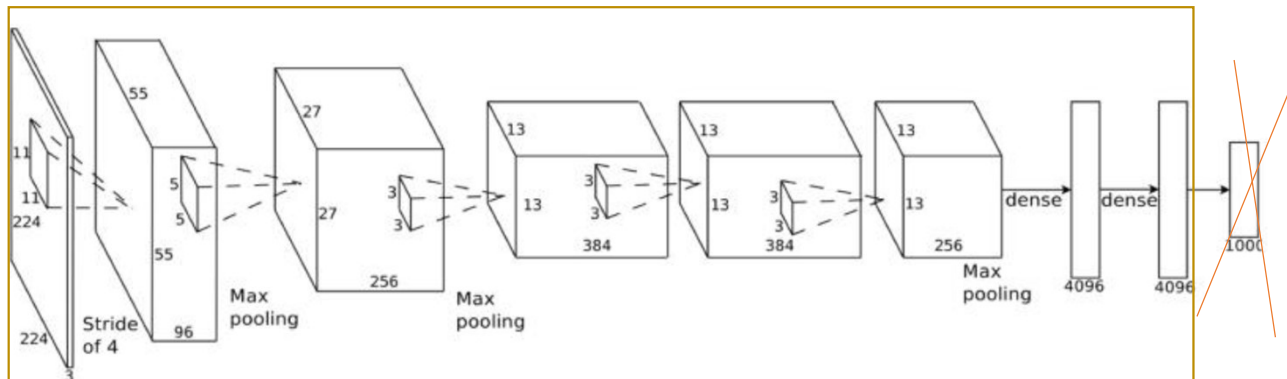
# Transfer learning

- We want to use ConvNets
  - They require A LOT of labeled examples
  - Otherwise we will likely overfit
  - Millions of parameters to estimate!
  - It may take weeks to train them
- Solution: Transfer learning!
  - Do not start from scratch
  - Use networks trained on large data sets and retrain /modify them for your purpose
  - 
  -



# Transfer learning

- Option 1: ConvNet as feature extractor
- Get the activations from one of the fully connected layers and use them as features
- Train a classifier using these CNN codes

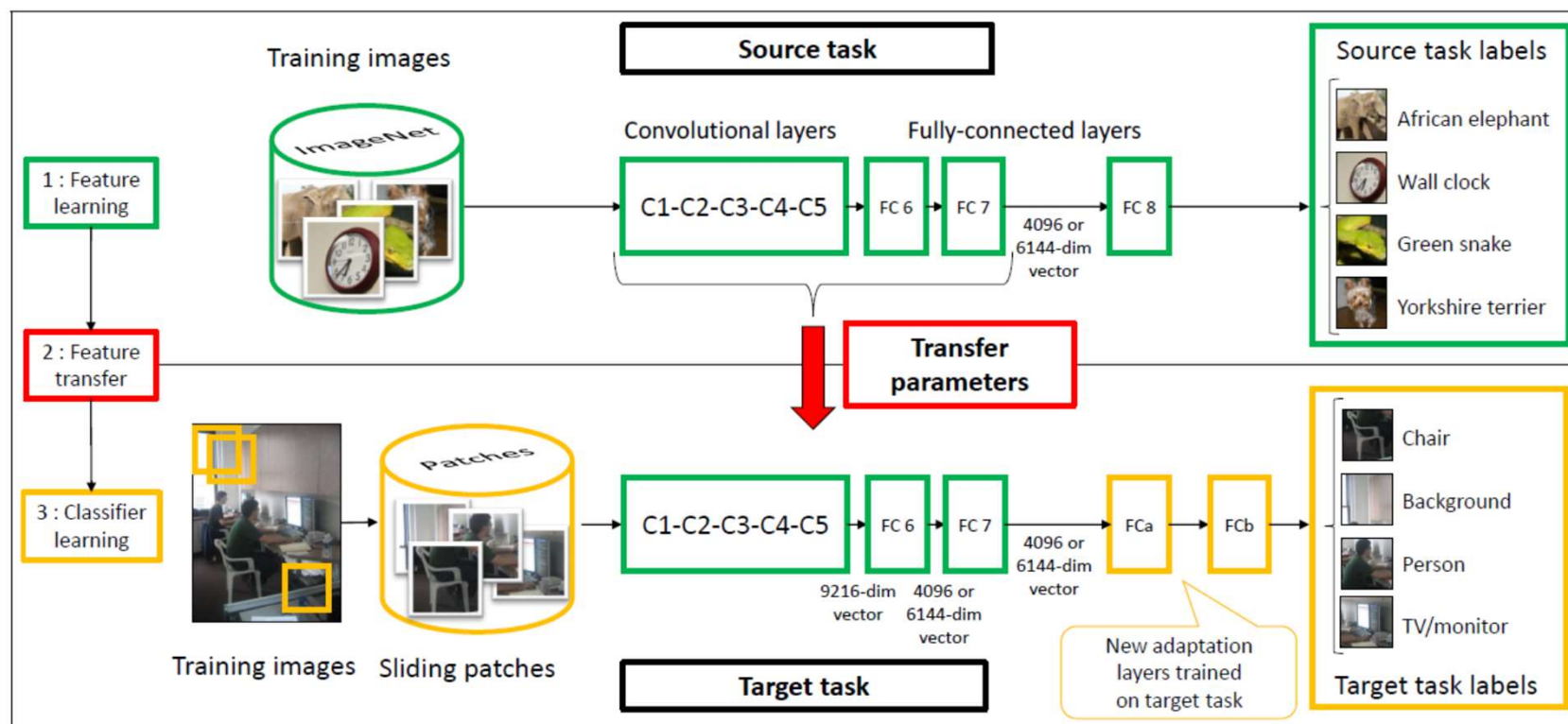


# Transfer learning

- Option 1: ConvNet as feature extractor
  - Get the activations from one of the fully connected layers and use them as features
  - Train a classifier using these CNN codes
- Option 2: Finetuning the ConvNet
  - Retrain only the last layer(s) of the network
  - Use back propagation but fix the weights from all other layers.

# Transfer learning

Oquab et al., CVPR 2014



# Regularization: Dropout

- Code practice

# Prevent over-training

- **Data enhancement/augmentation:** copy the existing data and add random noise, resampling, etc. We can apply different Angle rotation, translation transformation, random cropping, center cropping, blur, etc., so as to increase the amount of existing data.
- **Parameter regularization:** L1 regularization and L2 regularization are usually adopted to make a compromise between loss and model complexity.
- **Dropout:** During forward propagation, the activation value of a neuron stops working with a certain probability  $P$ . It does not change the network itself, but randomly removes the general hidden neurons in the network, and leaves the neurons in the input and output layers unchanged. This makes the model more generalizable because it doesn't rely so much on some local feature,
- **Select the appropriate network structure:** reduce the number of network layers, the number of neurons, the number of fully connected layers and so on to reduce the network capacity.

# Prevent over-training

- **Early Stopping:** The longer the training time, the greater the value of some network weights may be. If we stop training at the right time, we can limit the capability of the network within a certain range, thereby reducing overfitting,
- **Model combination:** Weak classifiers are fused to form a strong classifier, and the effect of fusion will be better than the final weak classifier.
- **Batch Normal:** When BN trains a layer, it will conduct standardized processing on each mini-batch data, which reduces the change of the distribution of internal neurons. It can speed up the training speed of large convolutional neural networks by many times, and the accuracy of classification can also be greatly improved after convergence.

# Next

- Case study Hand writing digits
- Make up missed lessons
- Release the second assignment