Module 7: Training neural networks – part II

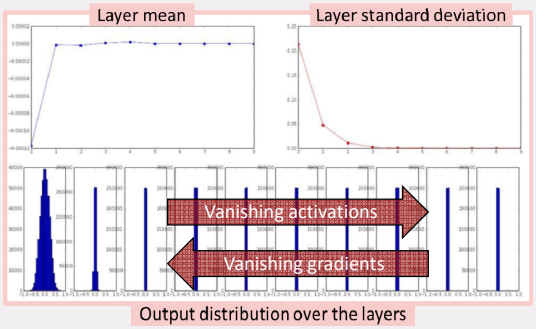
5LSM0: Convolutional neural networks for computer vision

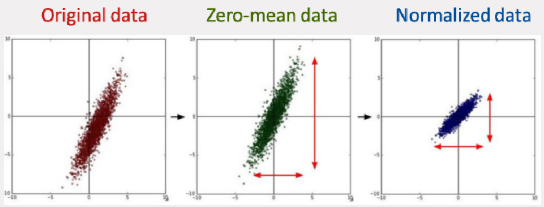
Fons van der Sommen

Electrical Engineering / VCA research group

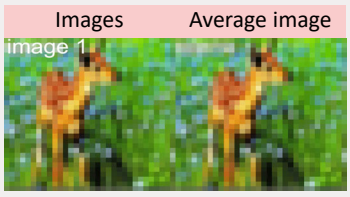
Last time

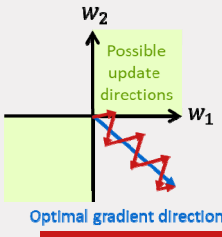
Weight initialization





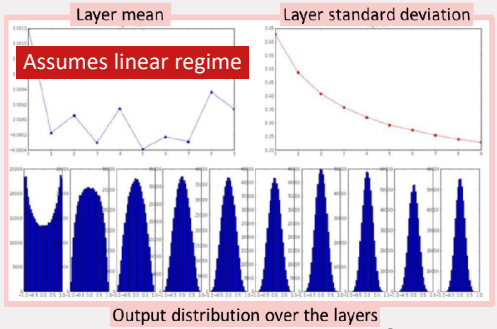
Data normalization





Not zero-centered

Last time

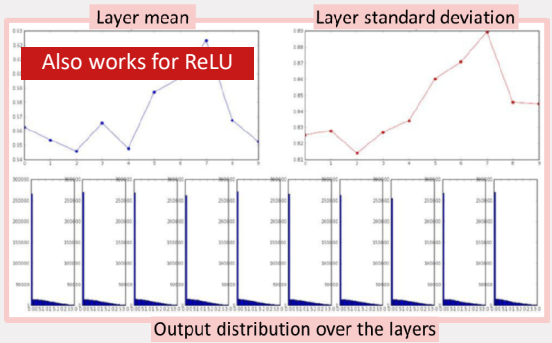


Scale by $\alpha = \sqrt{(1/n)}$

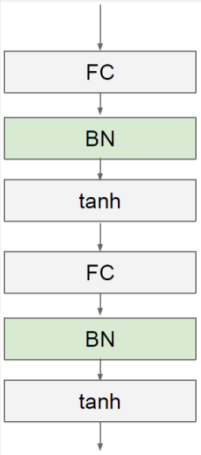
Xavier initialization

He et al.

Scale by $\alpha = \sqrt{(2/n)}$



Last time



Squashing and scaling

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Trainable parameters

The networks learns how much BatchNorm it needs for good training!

Batch normalization

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

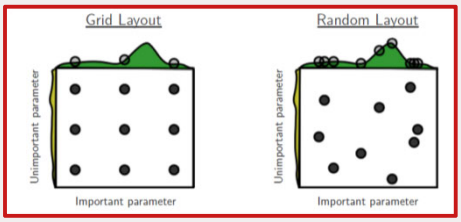
Sanity checks

- Reasonable loss?
- ↑regularization → Loss ↑
- Overfit on subset

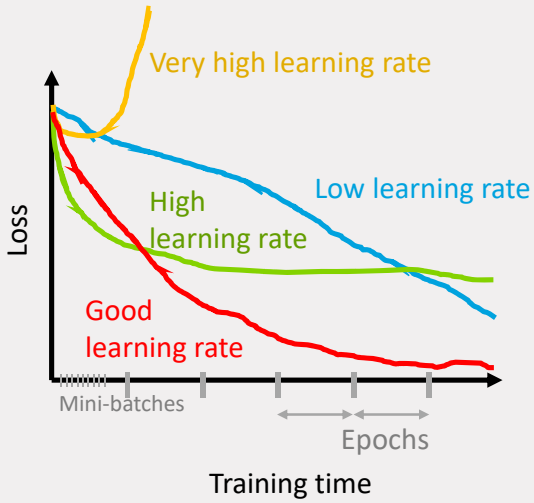
Last time

Finding the right learning rate

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



Courtesy Bergstra & Bengio, JMLR, 2012



This time

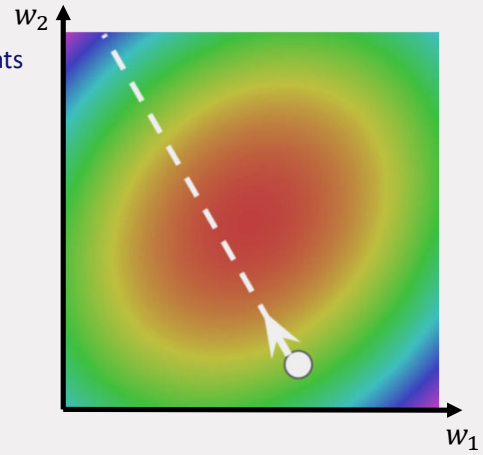
- How can we move down the loss landscape more efficiently?
- Fancy tricks with adjusting the learning rate
- Using multiple models: ensembles
- How should we regularize our model?
- Are there other methods to apply some regularization
- Can we use Convnets for (relatively) small data sets?

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!

Q: What causes of this?

*Example from cs231n (2017): lecture 7

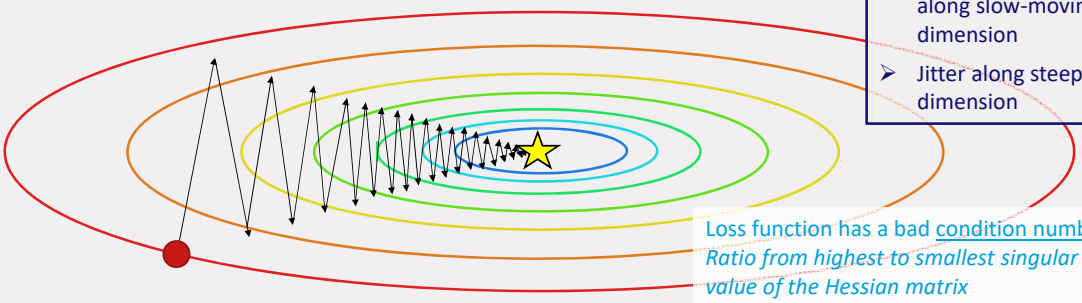
Optimization

Q: 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

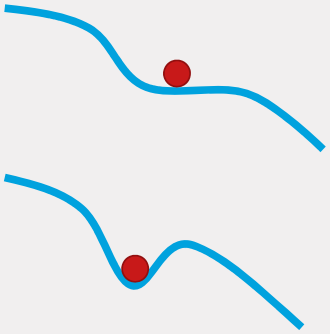
Loss function has a bad condition number: *Ratio from highest to smallest singular value of the Hessian matrix*

Optimization

Q: Another problem around saddle points?

What happens in local minima or sadle points?

- Gradient will be zero
- No movement → we get stuck!



In higher dimensions sadle 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!*

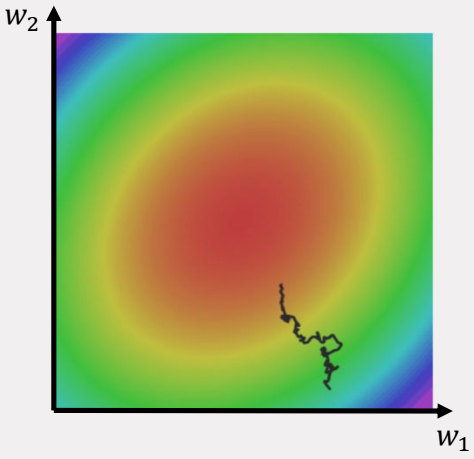
Optimization

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"

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

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



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

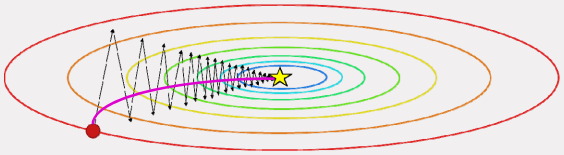
Optimization

Helps with all of the problems we've seen!

Solution: add momentum

SGD

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



Poor conditioning

SGD
+ momentum

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

velocity friction

Optimization

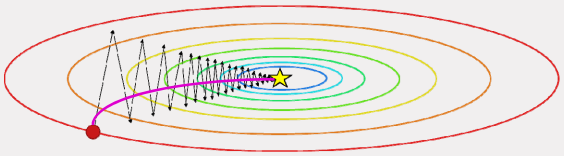
Q: To what signal processing concept is this equivalent?

Helps with all of the problems we've seen!

Solution: add momentum

SGD

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



Poor conditioning

SGD
+ momentum

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

velocity friction

Local minima Saddle points

Optimization

Solution: add momentum

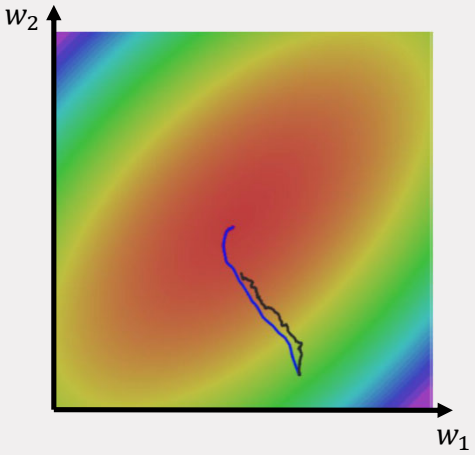
SGD

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

SGD
+ momentum

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

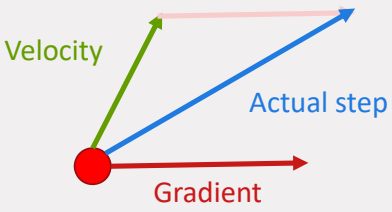
velocity friction



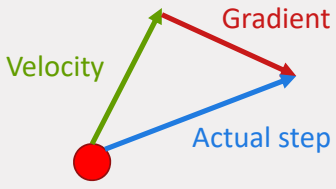
*Example from cs231n (2017): lecture 7

Optimization

Momentum update



Nesterov momentum



Optimization

Nesterov momentum

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

We want to evaluate the loss and the gradient at the same point

Nesterov breaks this ☹

Change of variables $\tilde{x}_t = x_t + \rho v_t$

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

First update similar to vanilla momentum

Second update includes an error correcting term

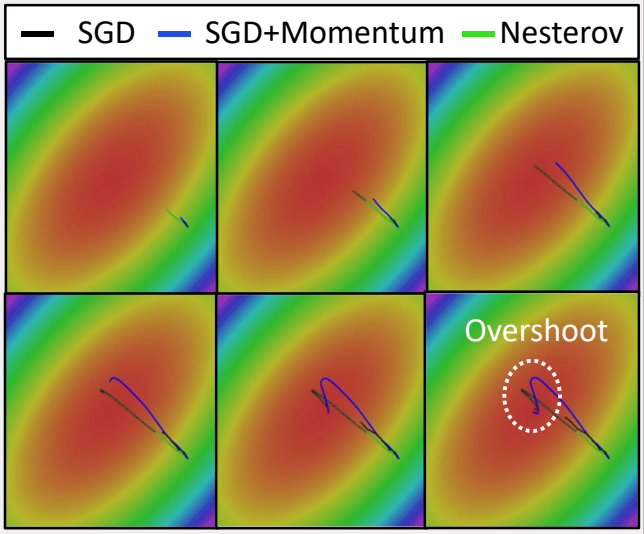
Optimization

Nesterov momentum

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

Change of variables $\tilde{x}_t = x_t + \rho v_t$

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$



*Example from cs231n (2017): lecture 7

Optimization

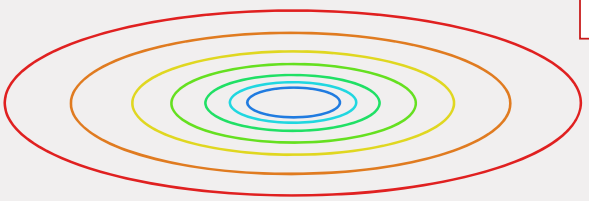
Idea: normalize the gradient updates

- **AdaGrad**: Added element-wise scaling of the gradient based on the historical sum of squares in each dimension
- Keep a running sum of all the squared gradients
  - *Instead of a velocity term, keep a grad-squared term*
- Divide by this term during update:

Update function

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

$$x_{t+1} = x_t - \frac{\alpha \nabla f(x_t)}{g_.^{-1/2}} \qquad g_i = \sum_{\tau=0}^{t} \left(\frac{\partial L}{\partial w_i}\right)^2$$



Q1: What happens in the case we have a high condition number?

Optimization

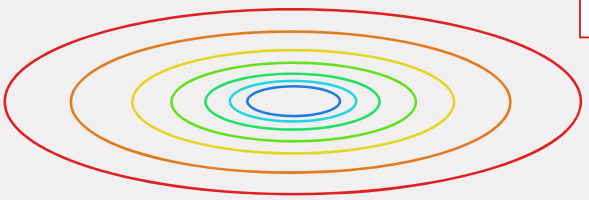
Idea: normalize the gradient updates

- **AdaGrad**: Added element-wise scaling of the gradient based on the historical sum of sqeares in each dimension
- Keep a running sum of all the squared gradients
  - *Instead of a velocity term, keep a grad-squared term*
- Divide by this term during update:

Update function

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

$$x_{t+1} = x_t - \frac{\alpha \nabla f(x_t)}{g_.^{-1/2}} \qquad g_i = \sum_{\tau=0}^{t} \left(\frac{\partial L}{\partial w_i}\right)_\tau^2$$



Q2: What happens to the step size over a long time?

Optimization

RMSProp

- Keep estimate of the squared gradients
- Introduce a decay rate of this squared-gradient term

$$x_{t+1} = x_t - \frac{\alpha \nabla f(x_t)}{g_t.^{-1/2}} \qquad g_{t,i} = \sum_{\tau=0}^{t} \gamma(1-\gamma)^{t-\tau} \left(\frac{\partial L}{\partial w_i}\right)^2_{\tau}$$

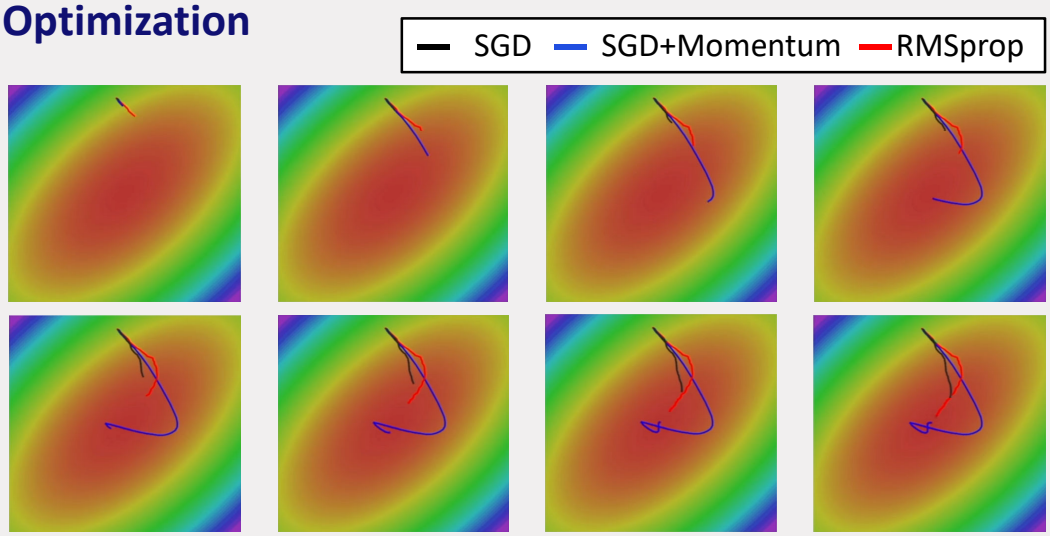
Decay rate γ

Update function

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

- Similar to momentum update!

Optimization



*Example from cs231n (2017): lecture 7

Optimization

Q: What happens at the first timestep if we initialize $m_0 = v_0 = 0$?

Momentum

- Building up velocity by adding in the gradients, step in the direction of the velocity

RMSProp

- Building up estimate of the squared gradients, dividing by the squared gradients

Both cool ideas, seem to work, why not combine them?

(Velocity)
First moment m

Momentum

AdaGrad / RMSprop

$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla f(x_t)$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla f(x_t)\right)^2$$

$$x_{t+1} = x_t - \alpha \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon}$$

Second moment v

Optimization

Kingma and Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015

Adaptive Moment Estimation (Adam)

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

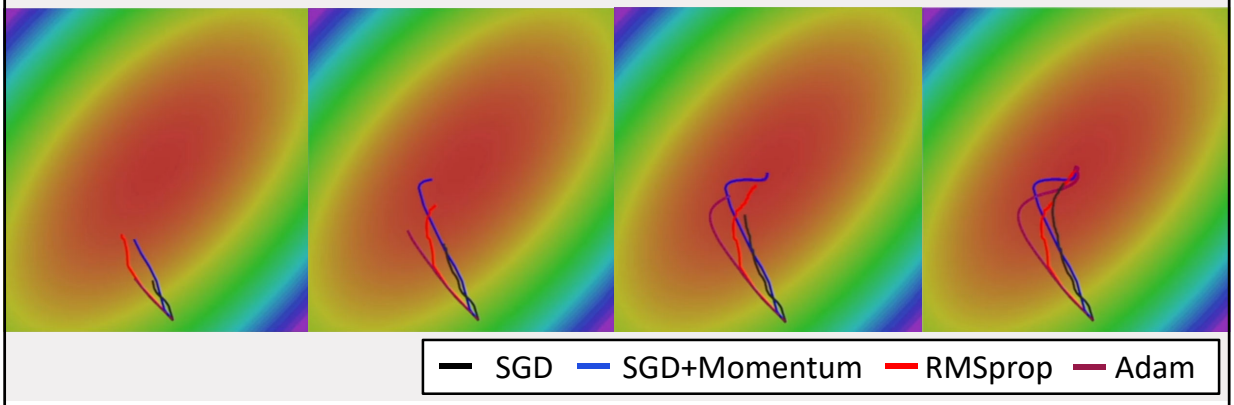
Avoid these big steps at the beginning by adding bias correction

$$\hat{m} = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v_t}{1 - \beta_2^t}$$

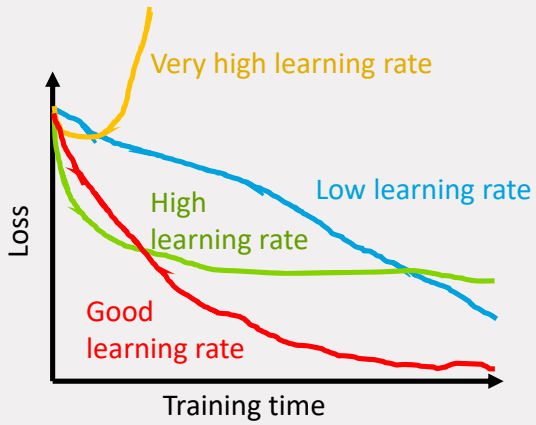
Optimization

Adaptive Moment Estimation (Adam)



*Example from cs231n (2017): lecture 7

Optimization



All discussed optimizers have learning rate as a hyper parameter

- One learning rate to rule them all?
  - *Typically: no!*

Learning rate decay over time!

- Step decay: Lower learning rate every few epochs
- Exponential decay:

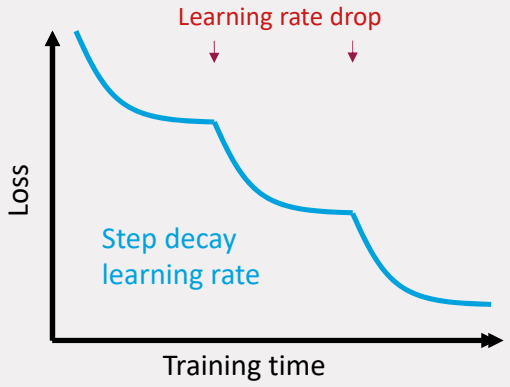
$$\alpha = \alpha_0 e^{-kt}$$

- 1/t decay

$$\alpha = \frac{\alpha_0}{1 + kt}$$

Q: Why is this a good idea?

Optimization



All discussed optimizers have learning rate as a hyper parameter

- One learning rate to rule them all?
  - *Typically: no!*

Learning rate decay over time!

- Step decay: Lower learning rate every few epochs
- Exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

- 1/t decay

$$\alpha = \frac{\alpha_0}{1 + kt}$$

Tuning the learning rate decay is more art than science

Can be a useful tool though…

Optimization

First-order approximation

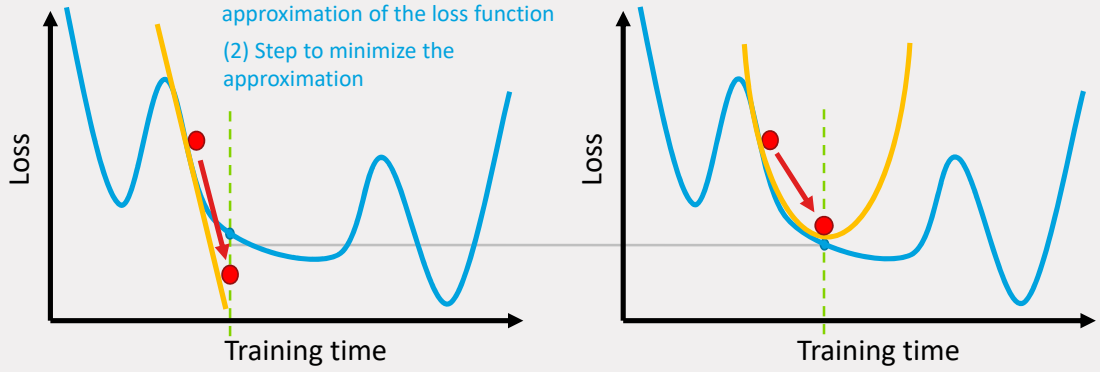
(1) Use gradient to get a linear approximation of the loss function

(2) Step to minimize the approximation

Second-order approximation

(1) Use gradient and Hessian to get a quadratic approximation of the loss function

(2) Step to minimum of approximation



Optimization

First-order approximation

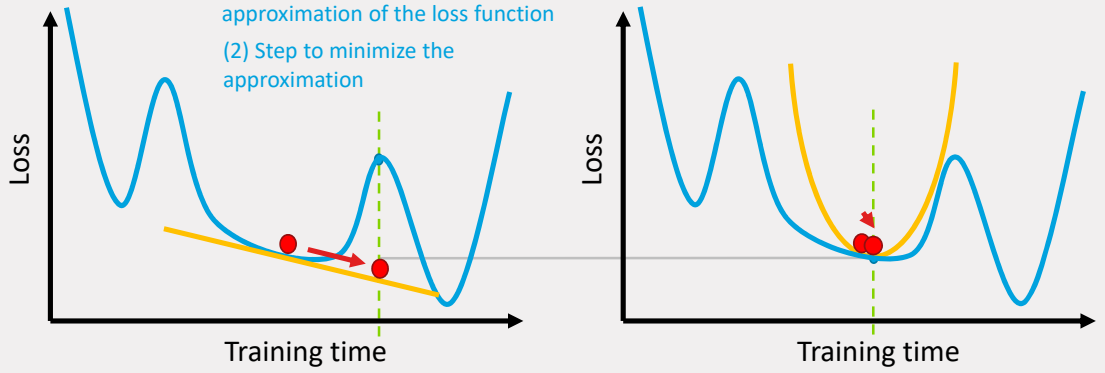
(1) Use gradient to get a linear approximation of the loss function

(2) Step to minimize the approximation

Second-order approximation

(1) Use gradient and Hessian to get a quadratic approximation of the loss function

(2) Step to minimum of approximation



Optimization

Quasi-newton methods that use a low-rank approximation of the Hessian exists and can be useful, but in most cases Adam gets the best results.

Second order Taylor expansion

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Note: Hessian has $O(N^2)$ elements…

… and inverting it takes $O(N^3)$

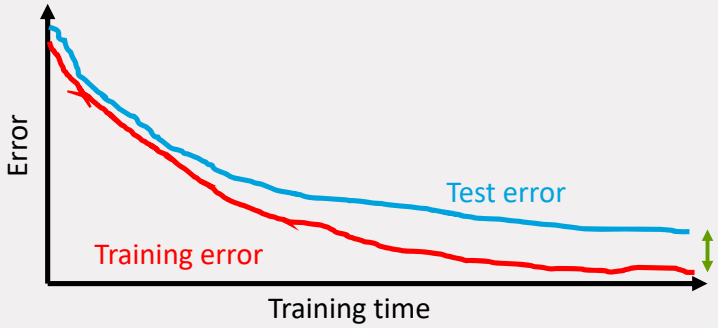
Q1: What's a crucial difference from the update rules we've seen before?

Q2: Why is this bad for deep learning?

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

Q: What's the downside of using model ensembles?

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…

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!

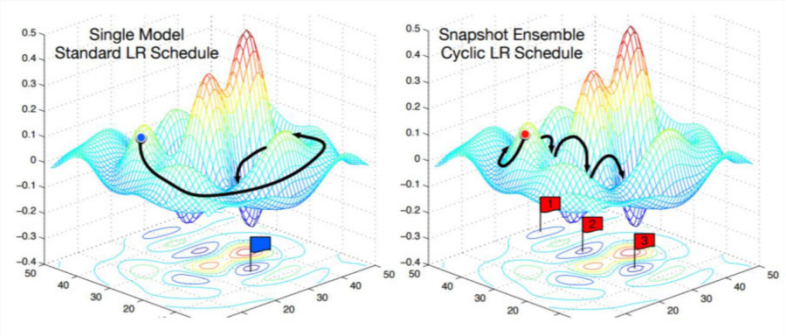
- Snapshots / Stochastic weight averaging (check [1])

[1] https://towardsdatascience.com/stochastic-weight-averaging-a-new-way-to-get-state-of-the-art-results-in-deep-learning-c639ccf36a

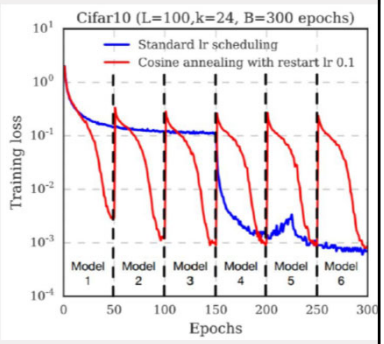
Improve test performance

Loshchilov and Hutter, "SGDR: stochastic gradient descent with warm restarts", ICLR 2017

Huang et al., "Snapshot Ensembles: Train 1, get M for free", ICLR 2017



Snapshot ensembles



Cyclical learning rates

Improve test performance

Improve single models

→ Avoid overfitting / Improve generalization

Regularization!

- L_2 regularization $R_{L2}(W) = \sum_k W_k^2$
- L_1 regularization $R_{L1}(W) = \sum_k |W_k|$
- Elastic net $R_E(W) = \beta R_{L2} + (1-\beta) R_{L1}$

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

Regularization

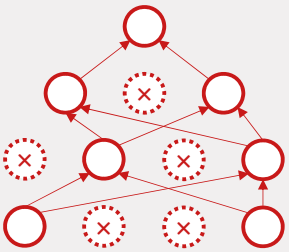
Dropout

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

Regularization

Dropout

- How can this possibly be a good idea?
  - *Seriously messing up our network!*



Interpretation 1:

- Models cannot rely to 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!!*

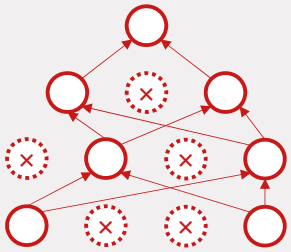
Regularization

Dropout: test time

Output prediction $y = f_W(x, z)$

Input image

Mask telling you which neurons are dropped-out



- You don't want randomness at test time!
- Marginalize over this random variable

$$y = f_W(x) = E[f(x, z)] = \int p(z) f(x, z) dz$$

- Integral not at all trivial to solve…

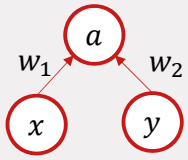
Regularization

Drop-out: test time

- Want to solve: $y = f_W(x) = E[f(x, z)] = \int p(z) f(x, z) dz$
- But very hard.. ☹

We can approximate the integral locally:

- Consider a single neuron
- At test time we have: $E[a] = w_1 x + w_2 y$
- During training time we have:



Dropout with $p = 1/2$

$$E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0 \cdot y) + \frac{1}{4}(0 \cdot x + w_2 y) + \frac{1}{4}(0 \cdot x + 0 \cdot y) = \frac{1}{2}(w_1 x + w_2 y)$$

→ At test time, multiply by dropout probability

Regularization

Dropout summary

- Drop neurons with probability p in forward pass
- Scale neuron output with p at test time

Q1: What's the main benefit of this approach?

More efficient: inverted dropout

- At test time, use entire weight matrix
- At training time divide by p

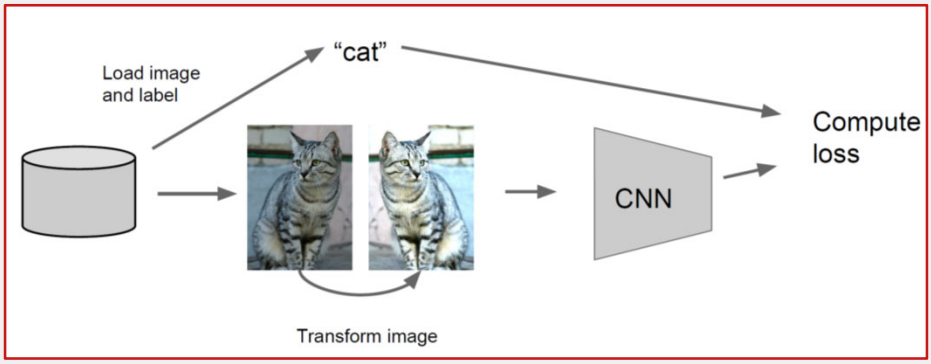
Q2: What do you expect about training time when using dropout?

Noise during training time for better regularization

- Add randomness during training to prevent it from fitting the training data too well (overfitting)
- Note: Similar to Batch-norm! Add noise during training time, average out during test time.

Regularization

Data augmentation



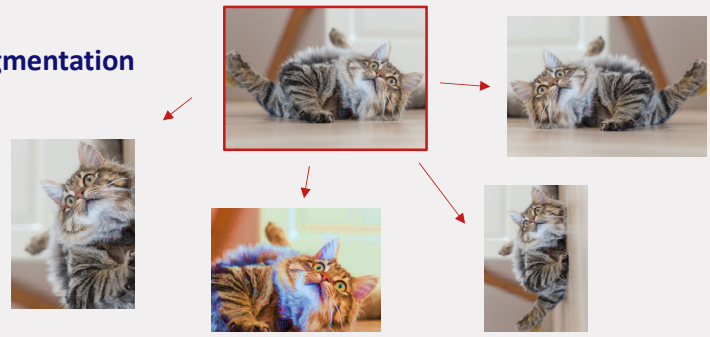
Regularization

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



Transfer learning

We want to use ConvNets, but…

- They require A LOT of labeled examples
  - *Otherwise we will likely overfit…*
- Millions of parameters to estimate!
- It may take weeks to train them

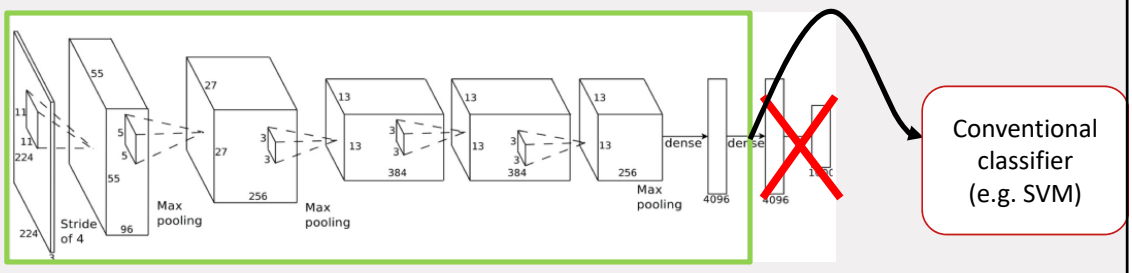
Solution: Transfer learning!

- 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

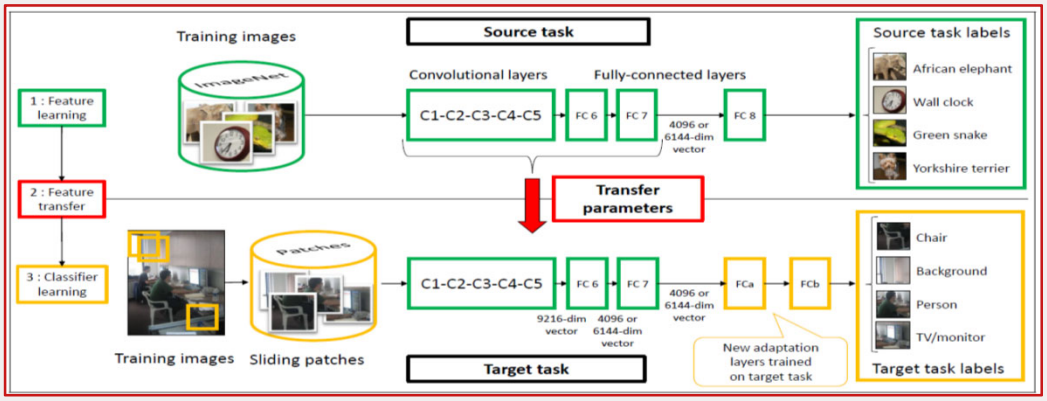
- Get the activations from one of the fully connected layers and use them as features
- Train a classifier using these CNN codes

Option 2: Fine-tuning the ConvNet

- Retrain only the last layer(s) of the network
- Use back-propagation but fix the weights from all other layers.

Transfer learning

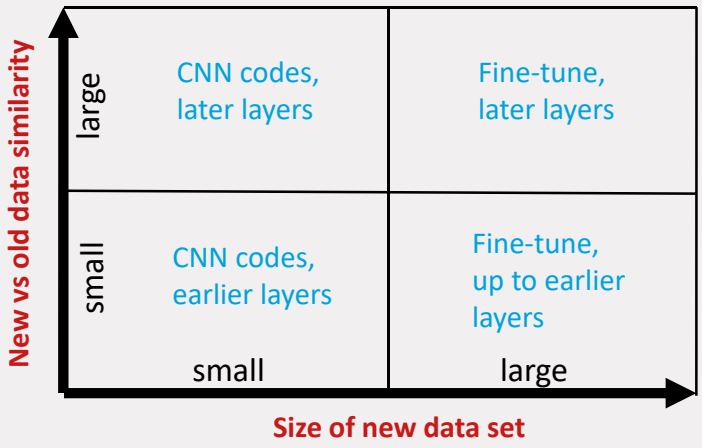
Oquab et al., CVPR 2014



Transfer learning

When and how to fine-tune?

Transfer learning works surprisingly well!!



Summary

- How can we move down the loss landscape more efficiently?
  - *Different optimizers: SGD+momentum, Nesterov, RMSprop*
  - *Typically best option: Adam → combine momentum and squared gradients*
  - *Use learning rate decay / schedulers, to better move towards minima*
- How should we regularize our model?
  - *Using an ensemble has a regularizing effect, also leads to better performance, but requires more time/memory…*
  - *Add a loss term, Dropout, data augmentation*
- Can we use convnets for (relatively) small data sets?
  - *Yes you can! Transfer learning*
  - *Either use CNN codes or fine-tune the network*