Deep Learning for Visual Computing

Training Convolutional Neural Networks

Christopher Pramerdorfer
Computer Vision Lab, TU Wien

Image from reddit, created using midjourney

This Week in AI
Video-LDM

Image from nvidia

Topics

Optimization revisited

- Gradient descent improvements
- Learning rate scheduling

Normalization layers

- Batch & layer normalization

Optimization vs. machine learning

- Underfitting & overfitting
- Data augmentation

Optimization Revisited

We already know how to train CNNs (for classification)

- Calculate (cross-entropy) loss $L(\boldsymbol{\theta})$ on training data
- Compute $\nabla L(\boldsymbol{\theta})$ using backpropagation
- Use gradient descent to update $\boldsymbol{\theta}$

Need some tweaks to make this pipeline practicable

Gradient descent update rule is $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla L(\boldsymbol{\theta})$

- Algorithm stops if $\nabla L(\boldsymbol{\theta}) = \mathbf{0}$ (at critical points)
- But should only stop at global minimum ...

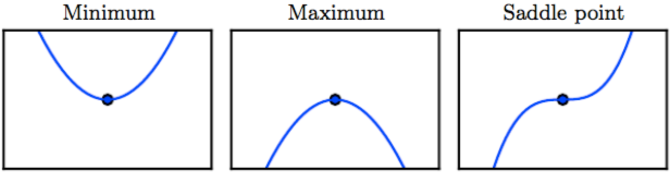


Image from [1]

... or a “good” local minimum

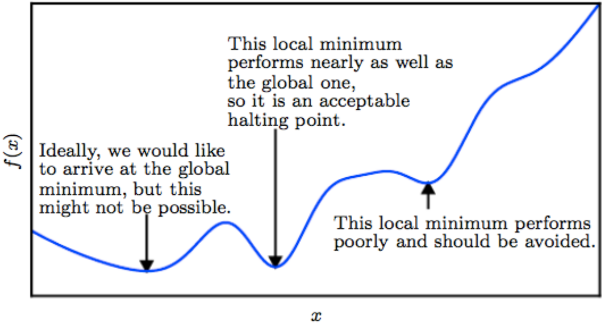


Image from [1]

Luckily studies show that in deep learning

- There are few (if any) local minima
- Local minima are “good” in above sense

Simple gradient descent thus works in theory but is

- Too expensive to compute with large datasets
- Too slow around critical points

Recall that $L(\boldsymbol{\theta})$ is average over S training samples

- Time complexity per iteration increases linearly with S
- Problem with large datasets (need many iterations)

To solve this problem we process the training set

- In minibatches of size S (one per iteration)
- Using minibatch loss and gradient as estimators

One full run through training set is called an epoch

- Training usually takes many epochs

Time complexity for single iteration independent of dataset size

Resulting algorithm called minibatch gradient descent

- Or stochastic gradient descent (SGD)

S varies between 1 and a few hundred samples

- Usually $S = 2^n$, e.g. 32, 64, 128, 256
- Powers of 2 for efficiency

Decreasing S also decreases

- Computation time per iteration
- Memory required on GPU (minibatch processed as whole)
- Accuracy of gradient estimate

Minibatch gradients are noisy estimates

- Gives gradient descent ability to escape critical points
- Can improve generalization
- Makes gradient descent non-deterministic

Important to sample minibatches randomly

- To break (possible) ordering in training set

Standard approach in practice

- Shuffle training set before every epoch
- Process sequentially in minibatches

Always use minibatch gradient descent

Optimization Revisited
Momentum

Gradient descent ignores information from previous iterations

- Slows down convergence

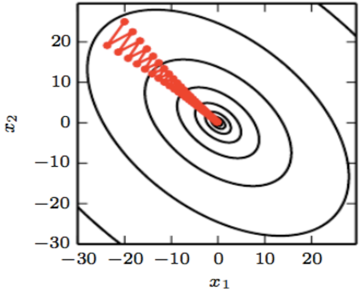


Image from [1]

Optimization Revisited
Momentum

Use exponential moving average of gradients for direction $\mathbf{v}$

- Influence of older gradients decays exponentially

Improves speed of convergence by

- Dampening oscillations (previous slide)
- Increasing step size dynamically

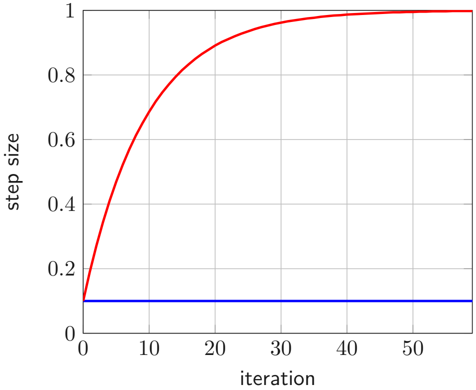
Iteration of gradient descent with momentum

- Update velocity $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla L(\boldsymbol{\theta})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}$

Hyperparameter $\beta \in [0, 1)$ called momentum

- Defines decay speed and maximum step size

Maximum speedup at constant gradient is $1/(1-\beta)$



Momentum dampens oscillations

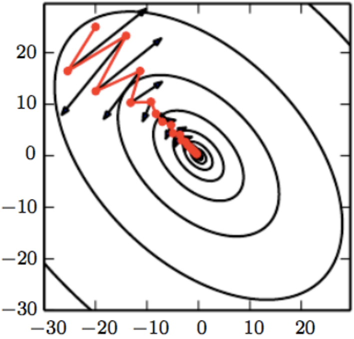


Image from [1]

Optimization Revisited
Nesterov Momentum

Evaluate gradient at $\boldsymbol{\theta} + \mathbf{v}$ instead of $\boldsymbol{\theta}$

Iteration of gradient descent with Nesterov momentum

- Update velocity $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla L(\boldsymbol{\theta} + \mathbf{v})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}$

Performance often slightly better than momentum

- Usually good idea to use this variant
- Setting $\beta = 0.9$ is usually fine

Optimization Revisited
Remaining Limitations

Gradient descent step size depends on gradient magnitude

- Small gradients, small parameter adjustments
- Problem if gradient magnitudes vary significantly

Makes it impossible to chose α such that

- We make progress in all directions
- Optimization remains stable

Optimization Revisited
Remaining Limitations

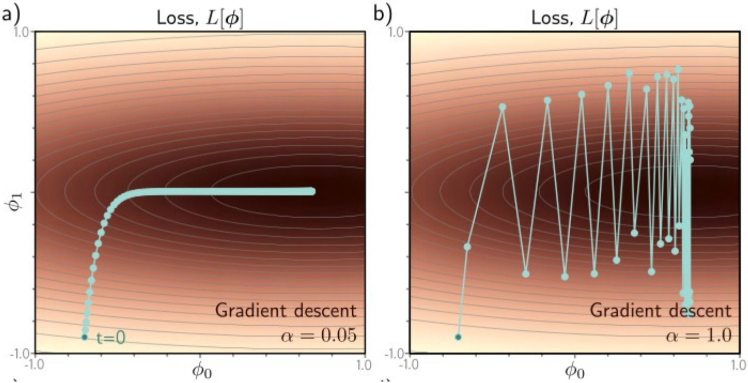


Image from [2]

Optimization Revisited
RMSProp

RMSProp aims to address this limitation

- Adapt learning rate for each parameter
- Based on its variance

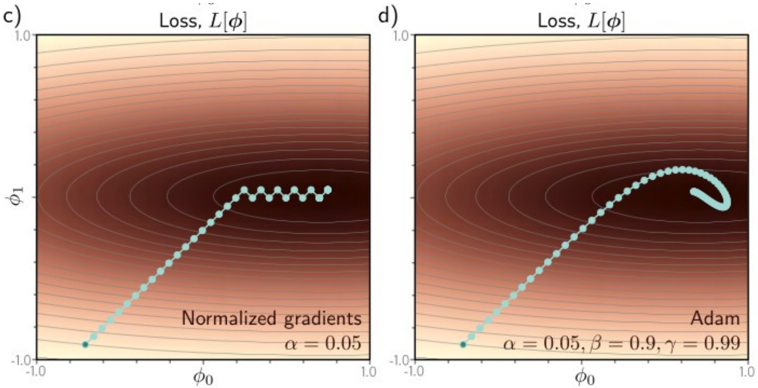
Update step (initially $\mathbf{n} = \mathbf{0}$)

- $\mathbf{n} = \beta_2 \mathbf{n} + (1 - \beta_2) \nabla^2 L(\boldsymbol{\theta})$
- $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla L(\boldsymbol{\theta}) / \sqrt{(\mathbf{n} + \epsilon)}$

Adam combines momentum and RMSProp

Update step (initially $\mathbf{m} = \mathbf{n} = \mathbf{0}$)

- $\mathbf{m} = \beta_1 \mathbf{m} + (1 - \beta_1) \nabla L(\boldsymbol{\theta})$
- $\mathbf{n} = \beta_2 \mathbf{n} + (1 - \beta_2) \nabla^2 L(\boldsymbol{\theta})$
- $\mathbf{m} = \mathbf{m} / (1 - \beta_1^t)$
- $\mathbf{n} = \mathbf{n} / (1 - \beta_2^t)$
- $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \mathbf{m} / (\sqrt{\mathbf{n}} + \epsilon)$



c)
Loss, $L[\phi]$
1.0
-1.0
ϕ_1
-1.0
ϕ_0
1.0
Normalized gradients
$\alpha = 0.05$
d)
Loss, $L[\phi]$
1.0
-1.0
-1.0
ϕ_0
1.0
Adam
$\alpha = 0.05, \beta = 0.9, \gamma = 0.99$

Image from [2]

Optimization Revisited
Adam

Adam works well in many scenarios

- Use as default optimizer
- Defaults for β_1 and β_2 are usually fine
- Tune α if feasible, or if not ... ☺

Path finding comparison on challenging f

- Different learning rates, so speed not comparable

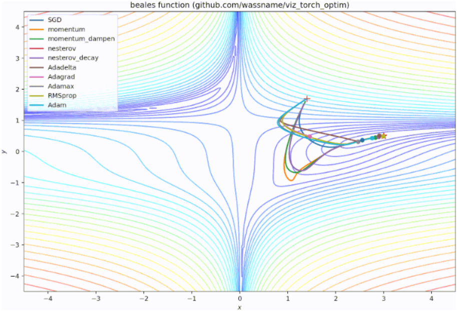


Image from github.com

Learning rate α is important with any optimizer

- Influences training time & quality of optimization

Studies show α should be varied during training

- To achieve both high speed and quality

Different strategies (schedulers) exist

- PyTorch: `torch.optim.lr_scheduler` module

A simple strategy is to

- Start with some base learning rate α
- Set $\alpha = \alpha / n$ if loss no longer decreases significantly
- `torch.optim.lr_scheduler.ReduceLROnPlateau`
- Common values for n are 2, 5, 10

Idea is to

- Make fast progress initially (large α)
- Be more careful later (smaller α)

Optimization Revisited
Learning Rate Scheduling

Most schedulers adapt α based on current epoch count

A popular variant is cosine decay

- Ensures α does not decrease too fast initially
- `torch.optim.lr_scheduler.CosineAnnealingLR`

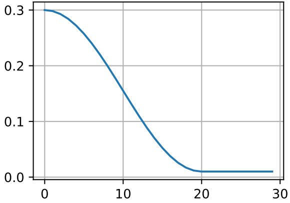


Image from d2l.ai

Warmup can be helpful in combination (e.g. Transformers)

- Start with small α
- Increase (e.g. linearly) for a few epochs
- Then decrease again (e.g. via cosine scheduling)

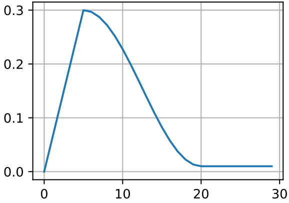


Image from d2l.ai

Cyclic variants also exist

- Repeat above pattern multiple times

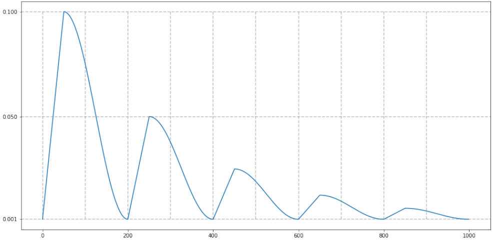


Image from github.com/katsura-jp

Always use learning rate scheduling

The `ReduceLROnPlateau` strategy is a solid default

- More intuitive than other variants
- No knowledge of sensible total epoch count needed

Other strategies may or may not work better

- Depends on network architecture, optimizer, data, etc.

Recall that deep networks are prone to vanishing/exploding signals

- Chained multiplications

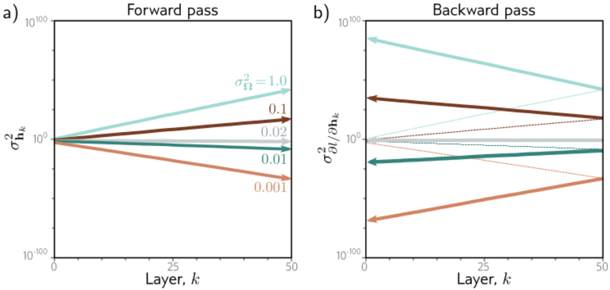


Image from [2].

Normalization Layers

Motivation

To avoid this we

- Initialize parameters carefully (previous lecture)
- Normalize input images

Standard image normalization method

- Subtract per-channel mean of training set
- Divide by per-channel standard deviation

This ensures a stable signal only initially though

- Weights change during training

Normalization Layers
Motivation

We want normalization built into our networks

- By adding normalization layers
- Batch normalization is most popular method for CNNs

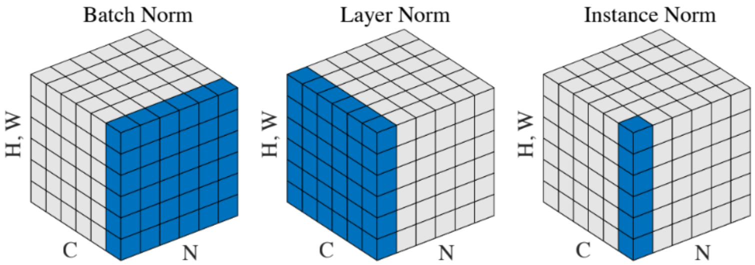


Image from [3]

Batch normalization [4] normalizes minibatch statistics

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

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

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

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

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

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

Image from [4]

Steps (variant for conv layers)

- Estimate per-channel means μ_c and variances σ_c^2
- Using current minibatch to approximate training set
- Normalize each channel as in previous slide
- Multiply channels by γ_c, add β_c

Last step ensures normalization can be skipped if sensible

- Identity function if $\gamma_c = \sigma_c$ and $\beta_c = \mu_c$

Advantages

- Well-behaved signals (enables deeper networks)
- Smoother loss functions (allows higher learning rates)
- Has regularizing effect due to noisy minibatch statistics

Add after every conv and linear hidden layer

- Usually before the activation function
- PyTorch: `torch.nn.BatchNorm2d` (conv version)
- Shuffle training set before every epoch (regularization)

What about after training?

- Inference usually done on single samples
- Not possible to compute minibatch statistics

Thus these statistics are

- Aggregated during training (moving averages)
- Used after training for normalization

Layer normalization [6] is also popular

- Normalization done per-sample instead of per-channel
- Works with single samples (no moving averages needed)
- Used in Transformers (later) and some modern CNNs

Stick with batch normalization for CNNs though

- Layer normalization often performs worse

Normalization Layers
Differences (2D Case)

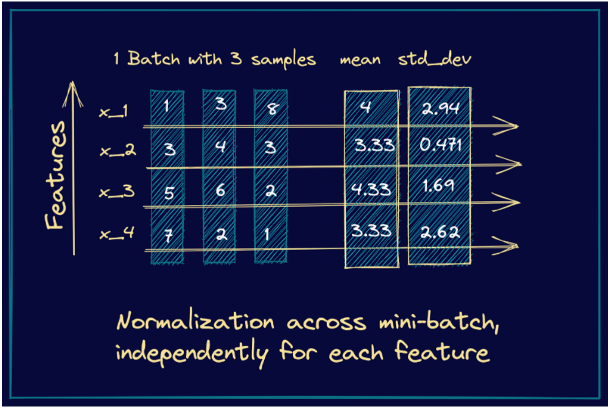


Image from pinecone.io

Normalization Layers
Differences (2D Case)

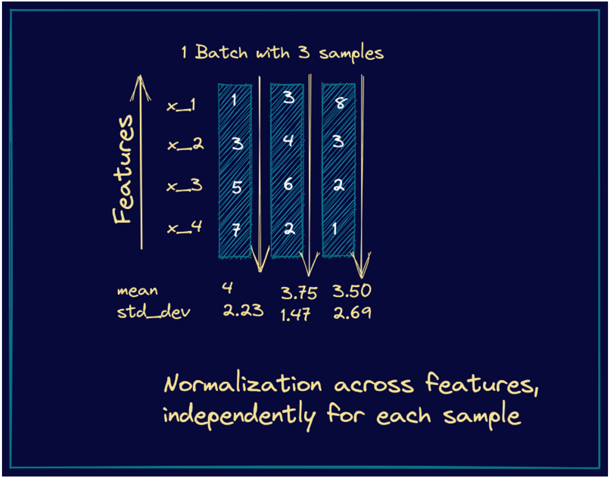


Image from pinecone.io

We covered training from an optimization perspective

- Find parameters that minimize training loss
- Known as empirical risk minimization

Prone to overfitting

- Training data must capture underlying distribution well
- Usually not the case in image analysis

Typical example of overfitting

- Training loss decreases steadily
- Validation loss remains high (gets worse)

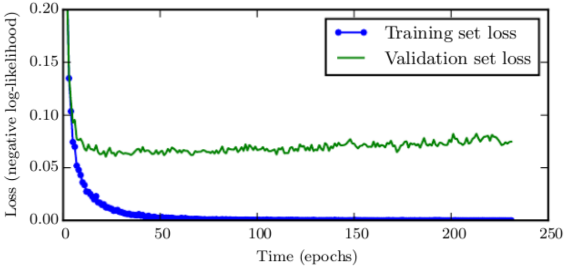


Image from [1]

So optimization was successful (loss ≈ 0)

- But disappointing validation/test performance
- Due to ability to generalize well to unseen data

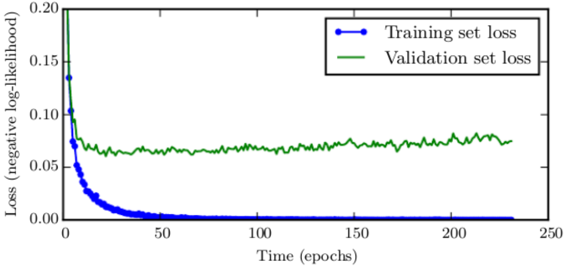


Image from [1]

In machine learning our actual goals are

- Low training loss (avoid underfitting)
- Small gap to validation loss (avoid overfitting)

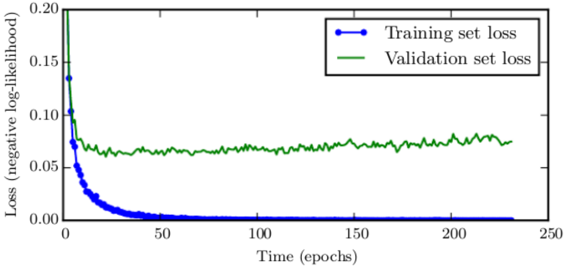


Image from [1]

To (hopefully) achieve this we

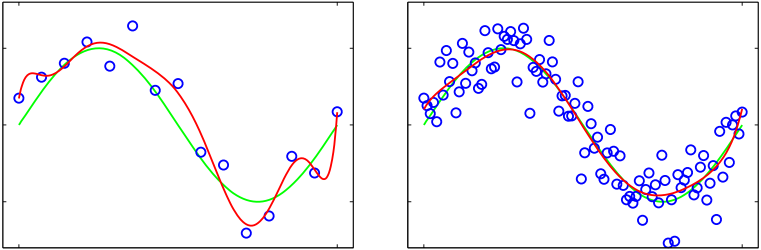
- Minimize the training loss (we have to)
- While monitoring training & validation performance
- And combat overfitting

Monitoring losses can be unintuitive

- Can be hard to interpret (e.g. cross-entropy loss)
- Training loss can be affected by regularization (next lecture)
- Use suitable performance measures instead (e.g. accuracy)

Best way to improve generalization: more training data

- But in practice data are limited



Create meaningful fake training data (training data augmentation)

- Apply transformations to training samples
- That have no effect on output (e.g. class label)

Image adapted from youtube

Can be done online, no need to store transformed samples

- Apply transformations during minibatch generation

Common transformations

- Random cropping
- Horizontal mirroring with probability 0.5
- Random similarity or affine transforms
- Random contrast, brightness, sharpness changes

PyTorch: `torchvision.transforms`

Mixup is a powerful extension

- Randomly mix two training images and their labels

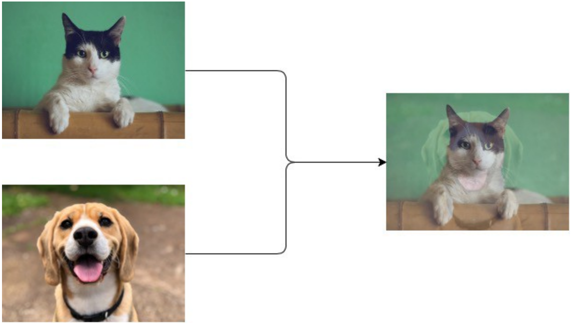


Image adapted from towardsdatascience.com

To avoid overfitting we have two options

- Use a network with just enough capacity not to overfit
- Prevent a network with more capacity from overfitting

Option 2 is always preferred in practice

- Leads to better performance
- We will see how to implement this in next lecture

Bibliography

[1] Goodfellow et al. Deep Learning. 2016

[2] Prince. Understanding Deep Learning. 2023

[3] Wu & He. Group Normalization. 2022

[4] Ioffe & Szegedy. Batch Normalization. 2015

[5] Srivastava et al. Dropout. 2014

[6] Lei Ba et al. Layer Normalization. 2016