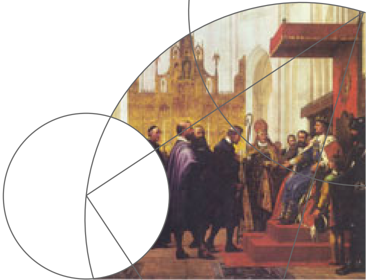
Faculty of Science

Optimization in Deep Learning

Elements of Machine Learning

Jens Petersen

with illustrations from Deep Learning, by Goodfellow et al.
Assistant Professor, Department of Computer Science
University of Copenhagen



About this lecture

Optimization in deep learning

- Batch and minibatch algorithms
- Problems encountered by gradient based algorithms
- Learning rate schedules
- Momentum
- Parameter initialization strategies
- Adaptive learning rate techniques
- Batch normalization
- Continuation methods and curriculum learning

Whats the point?

- Deep learning methods are expensive to train.
  - DL book suggests it is common to train for months on hundreds of machines (really?)
  - Training time is often the limiting factor (the primary one perhaps along with lack of data)
- We want to find the parameters θ of a neural network that reduce the expected cost function (loss) $J^*(\theta)$

$$J^*(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim p_{\mathrm{data}}} L(f(\mathbf{x};\theta), y)$$

Empirical risk minimization

Typically we would replace the expectation over the distribution with a mean over the training set

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), y^{(i)}),$$

where m is the number of training samples.

Prone to overfitting, especially with high capacity/flexibly models that see in deep learning, so we should use techniques like

- Regularization
- Early stopping

Batch and minibatch algorithms

Unlike typical optimization problems, training ML algorithms thus typically involve as sum/mean over examples.

With gradient descent

- we have to compute a gradient over this sum as well in every iteration.
- this may take a long time if the training set is large.

Batch and minibatch algorithms

Unlike typical optimization problems, training ML algorithms thus typically involve as sum/mean over examples.

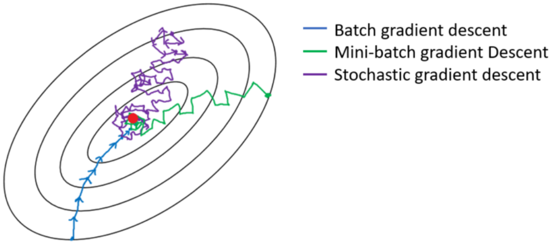
With gradient descent

- we have to compute a gradient over this sum as well in every iteration.
- this may take a long time if the training set is large.

However, we can compute the gradient over a sample of the training set in each iteration.

- Using the entire training set as a sample is called **batch** or **deterministic** gradient methods.
- Using a a single sample at a time is sometimes called **stochastic** or **online** gradient method.
- But typically one uses more than one, and fewer than all, and these methods are called **minibatch** or even also just **stochastic** methods.
  - The typical example is the stochastic gradient descent method.

Batch and minibatch algorithms



Using the complete training set (Batch), uses fewest iterations but each iteration takes longest to compute. Using a single sample (stochastic) takes the most iterations, but each iterations is fastest to compute.

Batch and minibatch algorithms

A tradeoff between two main effects

- Larger batches $\rightarrow$ more accurate estimate of the gradient, however, not a linear increase (in fact it flattens).
- Smaller batch sizes makes each gradient update faster, and this is a linear increase!.

Batch and minibatch algorithms

A tradeoff between two main effects

- Larger batches $\rightarrow$ more accurate estimate of the gradient, however, not a linear increase (in fact it flattens).
- Smaller batch sizes makes each gradient update faster, and this is a linear increase!.

Other important aspects of batch size

- Multicore architectures can better use larger batches.
- Batch size may be tied to memory usage (depending on implementation).
- Power of 2 is better for some hardware.
- Small batch sizes offer a regularizing effect, but may also require smaller learning rate.

Batch and minibatch algorithms

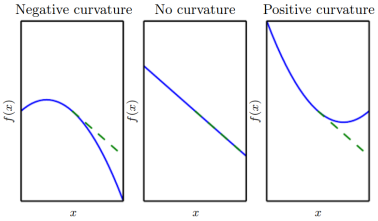
The regularization effect of minibatch algorithms

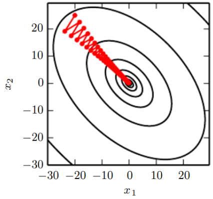
- It is harder to memorize or learn aspects of the data that does not represent the complete training set, when gradient updates rely on smaller subsets of the data.
- Stochasticity makes it easier to escape local minima.
- Batch size can be adjusted according to a schedule similar to learning rate.
  - At the start - small batch size helps to escape local minima and move in a direction that is representative for all datapoints.
  - At the end - larger batch sizes helps with convergence and stability.

Ill-conditioning

When even very small steps (learning rate) increase the loss.

- Gradient descent is based on a first order taylor series approximation
  - The error of this approximation is $O(h^2)$, where h is the step size/learning rate.
  - The error grows with the curvature of the problem.
  - We can estimate the error from the second order derivative. In practice, however, this is often not feasible.



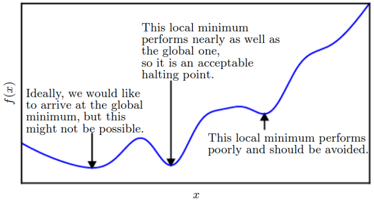


Each gradient step overshoots, because it does not consider the curvature of the problem, leading to oscillations.

Local minima

Another problem is that of getting stuck in a local minima.

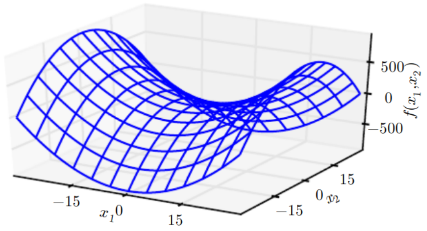
- Convex problem $\Rightarrow$ local minima = global minima
- Deep learning problems are not convex
  - but this may not be a big problem
- Solutions are not unique, e.g. we can often interchange weights and achieve the same solution.
  - **Model identifiability** problem.
  - Many local minima with same solution.
- Local minima are a problem if they have high loss compared to the global minima.



Plateaus, Saddle Points and Other Flat Regions

Saddle point

- Hessian matrix has both positive and negative eigenvalues.
- Points lying along eigenvectors with positive eigenvalues have greater loss than the saddle point, while the reverse is true with negative eigenvalues.
- Local minimum along one cross-section of the loss, and local maximum along another cross-section.



Plateaus, saddle Points and other flat regions

Typical charateristics of random functions

- in low dimensional spaces, local minima are common.
- in high dimensional spaces, local minima are rare and saddle points are more common.
- as the number of dimensions n grow, the ratio of saddle points to local minima grows exponentially with n.
  - Think of the sign of the eigenvalue as being randomly decided, to be a local minima we need all eigenvalues to have the same sign, which will be very unlikely with many random eigenvalues.

Plateaus, saddle Points and other flat regions

Typical charateristics of random functions

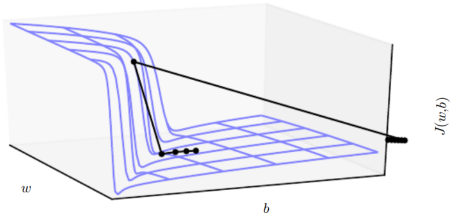
- in low dimensional spaces, local minima are common.
- in high dimensional spaces, local minima are rare and saddle points are more common.
- as the number of dimensions n grow, the ratio of saddle points to local minima grows exponentially with n.
  - Think of the sign of the eigenvalue as being randomly decided, to be a local minima we need all eigenvalues to have the same sign, which will be very unlikely with many random eigenvalues.

Large neural networks with their many parameters seem to share this characteristic.

- It is important that the optimizer does not stop at a saddle point
  - (Stochastic) Gradient descent seems to have this property

Cliffs and exploding gradients

Many layers can lead to very steep gradients regions, because of the multiplication of several large weights.



This can lead to overshooting,

- however, with non-linear functions, gradients only give us the direction, not the optimal step size.
- So we can clip them when they are too large.

Gradient cliffs are particularly a problem when dealing with long sequences with recurrent neural networks.

Long-term dependencies

Deep computational graphs and graphs with repeated application of the same parameters, say $\mathbf{W}$ can be problematic.

After t steps, this is equivalent to multiplication by $\mathbf{W}^t$.

Given an eigendecomposition $\mathbf{W} = \mathbf{V}\mathrm{diag}(\lambda)\mathbf{V}^{-1}$

$$\mathbf{W}^t = (\mathbf{V}\mathrm{diag}(\lambda)\mathbf{V}^{-1})^t = \mathbf{V}\mathrm{diag}(\lambda)^t\mathbf{V}^{-1}$$

Eigenvalues λ_i that are not near an absolute value of 1 will either explode or vanish.

The **vanishing and exploding gradient problem** refers to gradients through such graphs also being scaled by $\mathrm{diag}(\lambda)^t$

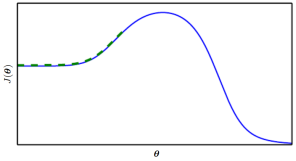
Inexact gradients

- Gradients are estimated based on samples (minibatch).
- They may be noisy or biased estimates.
- They may be approximations, if the exact gradient is intractable to compute.

It is also important to realize that the global minimum is not necessarily a solution that generalizes well. In fact we often stop optimization before reaching a minima, that is, we employ early stopping.

Poor correspondence between local and global structure

Optimization may fail if we do not move towards the global solution



Or it may take very long if the path to the global solution is very long.

There are many aspects making neural network optimization difficult, and it is an active area of research.

We may be able to initialize our way out of many of these problems.

Learning rate schedules

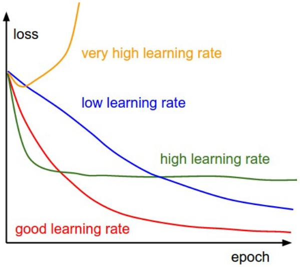
Stochastic gradient descent and variants are the most used optimization algorithms for deep learning.

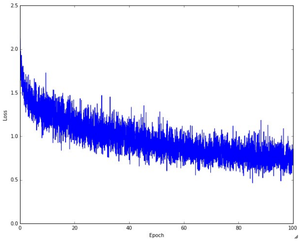
Learning rate is a crucial hyperparameter.

- The gradient of the total loss approaches zero when approaching a minimum.
- With minibatches this is not necessarily the case because of noise in the gradient estimates.
    - Stochastic/minibatch gradient descent therefore may never converge.
- Learning rates are therefore often decreased according to some schedule.
    - Larger learning rates in the beginning may also allow optimization to escape some local minima and move faster through saddlepoints.
- Choosing a good learning rate is difficult, and is usually done by monitoring learning curves.

Choosing a learning rate (rules of thumb)

- Too low - slow and linear decrease.
- Too high - exponential decrease with a high plateau or divergence and/or wild oscillations.
  - Note: mild oscillations may help to find a better solution.
- Learning rate interacts strongly with batch size.



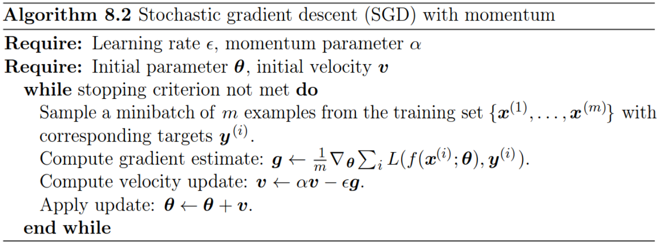


Example behaviour of different learning rates[1].

[1] Figure from http://cs231n.github.io/neural-networks-3/

Momentum

The momentum algorithm uses an exponentially decaying moving average of past gradients.



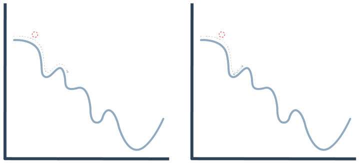
Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

- The velocity **v** accumulates the past gradients according to the parameter α. Larger α will let past gradients affect the direction more.
- Note if these past gradients point in the same direction, then the movement accellerates up to a maximum speed of $\epsilon||\mathbf{g}||/(1-\alpha)$.

Momentum

Often you will see the analogy of a ball rolling downhill building up momentum, which allows it to roll uphill for some time, perhaps escaping local minima[1].



In practice, the role it plays is perhaps more to dampen oscillations, that otherwise happen due to gradient descent errors in areas with large curvature.

Try to play with learning rate and α parameter (β in the example) at `https://distill.pub/2017/momentum/`.

[1]Figure from `https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model/optimizers-and-compiler-options`

Nesterov momentum

Update rules

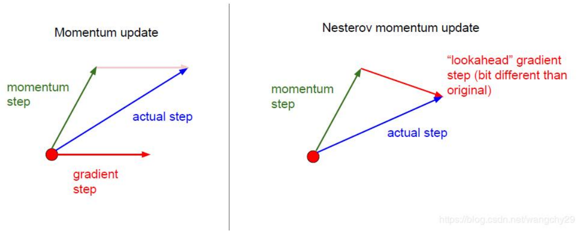
$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_\theta\left[\frac{1}{m}\sum_{i=1}^{m} L\left(f(\mathbf{x}^{(i)};\theta+\alpha\mathbf{v}),\mathbf{y}^{(i)}\right)\right]$$

$$\theta \leftarrow \theta + \mathbf{v}$$

With Nesterov momentum the new gradient is computed after an update with the current velocity.

- Think of momentum as doing two jumps, first one based on accumulation of gradients from previous steps and then one based on the current gradient.

Nesterov momentum[1]



- The jump based on the accumulated gradients is the same whether you use Nesterov or not.
- The jump based on the current gradient will differ. You can choose to use the gradient from prior to doing the first jump (momentum) or after the first jump (Nesterov momentum). The latter will in general be more accurate, as it is more accurate to use a gradient at the specific position you are at than from somewhere else.

[1] Figure from `http://cs231n.github.io/neural-networks-3/`

Parameter initialization strategies

- Neural network optimization is not well understood, so initialization strategis are often simple and based on heuristics.
- Must break symmetry, such that learning evolves parallel units in different manner.

Random initialization

- Biases set to heuristically chosen constant.
- Weights initialized randomly, from Gaussian or uniform distribution.
- Scale matters.
    - Too large weights, can lead to exploding gradients, saturation of activation functions.

Parameter initialization strategies

- Initial parameters should be close to final parameters - otherwise we might not get there.
- Many initialization strategies are based on analysing the behaviour of linear layers.

Goal: each layer should have the same gradient variance and the same activation variance. **Normalized initialization**, sampling for a layer with m inputs and n outputs using

$$W_{i,j} \sim U(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}).$$

Parameter initialization strategies

Normalized initialization and similar strategies where the size of weights depend on the width of the layer may lead to very small weights.

Sparse initialization is a different approach in which each unit has exactly *k* nonzero weights.

- Avoids very small weights.
- Enforces diversity among the units at initialization.
- May lead to too large weights that can be difficult for gradient descent to shrink.

Choice between sparse or dense initialization, and scale of weights can be regarded as a hyper parameter and optimized over a validation set.

Parameter initialization strategies

Initializing the bias

- Strategy must be compatible with weight initialization scheme
  - Setting bias to zero is compatible with most
- It may be advantageous to set bias for output units to right marginal statistics for the output.
  - Inverse of the activation function applied to the marginal statistics of the output in the training set.
  - Example - output is a skewed distribution with marginal probability of each class being some vector **c** and we use softmax as the activation function, then we can solve $\mathrm{softmax}(\mathbf{b}) = \mathbf{c}$ to give us the bias vector **b**.
- May want to avoid saturating the ReLU by setting bias to some positive value, however, often we also normalize the input to avoid this problem.
- Some units have input gates that controls if some input is passed on. We may want to initialize the bias of such gates such that all input is passed on initially to give these units a chance to learn.

Parameter initialization strategies

Initialize using a pretrained model

- Supervised learning on a related task, can yield initial weights which can then be finetuned on the task at hand.
  - A common strategy in computer vision is to finetune models initially trained on imagenet.
- Unsupervised learning can also be used. E.g. you may train a model for a task which does not require supervision and then finetune the model on the supervised task at hand.

Such strategies may lessen the amount of labelled data needed.

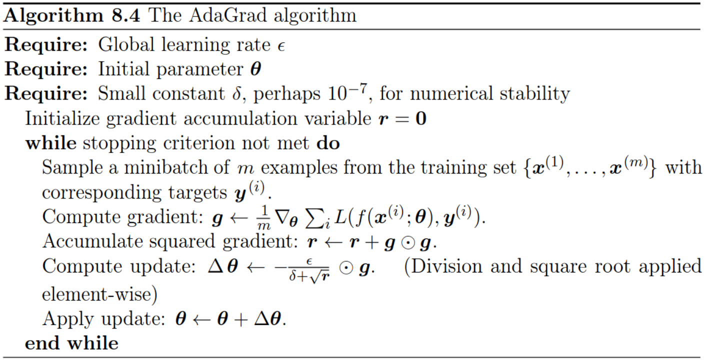
Adaptive learning rates

Learning rate is a sensitive hyperparameter, which can be difficult to set.

It is possible to adapt learning rate to each parameter

Adaptive learning rates

AdaGrad - adapts the learning rates of parameters by scaling inversely proportional to the square root of the sum of all the historical squared values of the gradient - faster progress in areas with more gentle slopes.

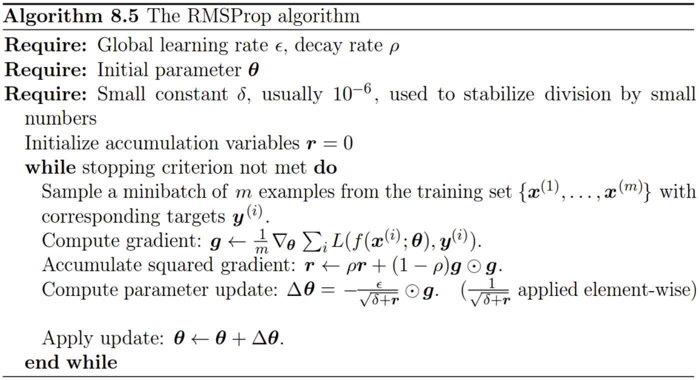


Algorithm 8.4 The AdaGrad algorithm

```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute update: $\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.  (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.
  end while
```

Adaptive learning rates

RMSProp As opposed to AdaGrad's sum, the RMSProp uses an exponentially weighted moving average.



Algorithm 8.5 The RMSProp algorithm

```
Require: Global learning rate $\epsilon$, decay rate $\rho$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small
  numbers
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
  end while
```

Adaptive learning rates

RMSProp As opposed to AdaGrad's sum, the RMSProp uses an exponentially weighted moving average.

- Adagrad was designed to converge fast with convex functions, but with nonconvex functions, one may need to pass through many structures with varying gradients.
- The shrinking of the learning rate based on sum of the history of the computed gradients squared may lead to stopping too early.
- RMSProp discards the extreme past to avoid this issue.

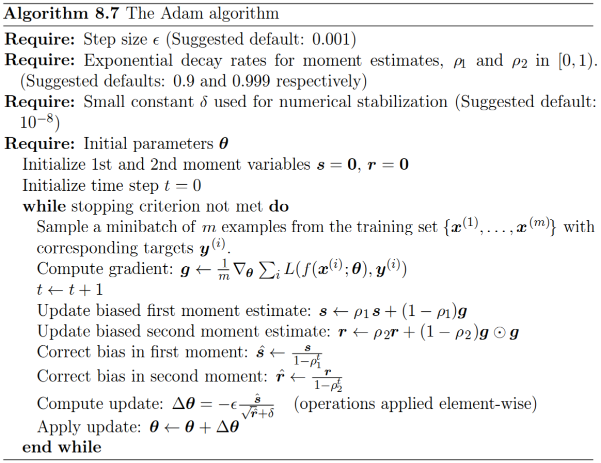
Adaptive learning rates

Adam can be seen as a combination of RMSProp and momentum. That is the update is an exponentially decaying average of the past gradients, which is then scaled by the inverse square root of an exponentially decaying average of the past gradients squared.

Adam has also been extended to use Nesterov momentum.

Adams default parameters tend to work well, so often hyperparameter optimization is not needed. The overall learning rate is sometimes adjusted.

Adaptive learning rates



Algorithm 8.7 The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while
```

Note the correction for the initialization phase.

Adaptive learning rates - which algorithm to use?

No consensus at this point.

- The book argues basically all of the covered algorithms are popular.
- Adam is perhaps the easiest to use, because it converges fast and tends to work without adjusting hyperparameters.
- SGD with Nesterov momentum has been empirically been shown to generalize better than Adam in many cases.

Batch normalization

Deep models can be difficult to train because of how multiple layers interact to change the overall outcome.

- Each gradient describes how a given parameter should be changed to change the overall output such that loss is decreased.
- However, we do not update a single parameter at a time, but all of them.
- After an update to a given layer, the following layers will now receive new changed input, and might therefore need to change to adapt to this.
  - This can lead to a situation where the output of layers shift around in response to previous layers shifting around, which may make convergence slower if not impossible.
- Ideally we want each layer to learn a representation that is more independent of previous layers, **batch normalization** attempts to do this by normalizing the output of each layer.

Batch normalization

Normalization of a layers output is done by subtracting the mean and dividing by the variance over a training batch.

To restore the ability of the network to represent relevant signals, the normalized output is multiplied by a learned 'variance' and added to a learned 'mean'.

- This makes the mean and variance of the output of a layer more independent of changes to previous layers.

At test time running averages of the batch mean and variances can be used.

Designing models to aid optimization

To improve deep learning performance

1. we may improve the optimization approach
2. or we may change the model to be easier to optimize.

Deep learning performance increases has mostly come from (2).
Examples include

- Switching to activation functions like ReLU which are close to linear
- Skip connections between layers reduce the length from input to output

Continuation methods

Suppose we start with a simple objective and gradually make the task more difficult. That is we work through a series of cost functions that gradually become more difficult

$$\{J^{(0)}, J^{(1)}, \dots, J^{(n)}\}$$

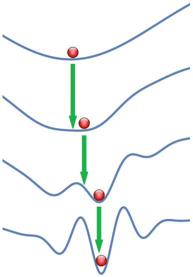
A cost function $J^{(i)}$ being easier than $J^{(i+1)}$ means it is well behaved over more of θ space.

We also want that each loss function is a good initialization for the next.

- Classic continuation methods predates neural networks, and are based on the idea of starting with smooth functions and gradually making them less smooth.
  - Non-convex functions tend to become approximately convex when smoothed.
- They are also related to simulated annealing

Continuation methods and neural networks[1]

- were designed to deal with local minima, but local minima are no longer believed to be the primary problem for NN optimization [DL book].
  - May still eliminate flatness, lessen gradient variance, and increase correspondance between the local and global structure.
- Not all functions become approximately convex when smoothed
- and it might be difficult to find an applicable series of cost functions.
- The global minima of the smoothed function may coincide with a local minima of the objective.

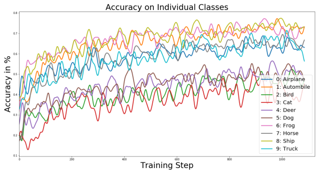


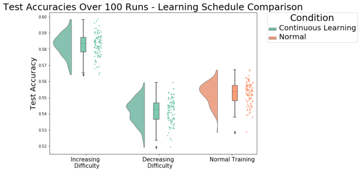
[1] Figure from Training Recurrent Neural Networks by Diffusion, Hossein Mobahi (2016)

Curriculum learning

Curriculum learning can be seen as a continuation method, which works by planning the learning process such that simple concepts is learned first and then progressing towards more and more complicated concepts.

This is similar to how humans are typically tought.





Learning CIFAR-10 classes ordered either by decreasing or increasing difficulty [1]

[1] Figure from `https://towardsdatascience.com/how-to-improve-your-network-performance-by-using-curriculum-learning-3471705efab4`

Questions?