UNIVERSITÄT BERN

Deep Feedforward Networks - Optimization

Paolo Favaro

Contents

- Optimization in Feedforward Neural Networks
  - Empirical risk optimization, batch and mini batch algorithms, ill-conditioning, plateau and saddle points, exploding gradients, long-term dependencies, momentum, AdaGrad, RMSProp, Adam, batch normalization, curriculum learning
- Based on **Chapter 8** of Deep Learning by Goodfellow, Bengio, Courville

Note

Optimization

- We use optimization for training neural networks
- Due to the problem size and the complexity of the models, understanding how training works and how it can be improved is fundamental
- We will see a number of successful techniques that have been designed and are currently in use
- The backbone to all the techniques on neural networks is gradient descent

Note

Learning and Optimization

- One key aspect of optimization in ML is that the cost function of interest (on the test set) is different from the cost function we optimize on (based on the training and validation sets)

- More explicitly we use

$$J(\theta) = \mathrm{E}_{(x,y)\sim\hat{p}_{\mathrm{data}}}[L(f(x;\theta),y)]$$ empirical risk

  instead of the true (but unavailable) goal

$$J^*(\theta) = \mathrm{E}_{(x,y)\sim p_{\mathrm{data}}}[L(f(x;\theta),y)]$$ Bayes risk

We discussed this aspect already in the Machine Learning review when introducing the generalization error. We would like to optimize the bottom error (over the full data probability distribution), but we only have access to a finite amount of data (the empirical probability distribution).

Empirical Risk Minimization

- Since we do not have the true data distribution we use the empirical risk

$$E_{x,y\sim\hat{p}_{\text{data}}}[L(f(x;\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x_i;\theta),y_i)$$

  where m is the number of training examples.

- The minimization of the empirical risk may lead to overfitting and might be unfeasible with gradient descent (due to undefined derivatives)

- Therefore, we need to regularize the risk

Overfitting might be due to: high capacity models, too little data or data that is not a good representative of the true data distribution. This may lead to the model memorizing the training set, for example.

Some other effects of interest are that sometimes the validation or test error might continue to decrease although the training error has reached zero (so it is not decreasing anymore).
The continual minimization of the training loss keeps pushing the decision boundaries away from the classes. This results in a more robust classifier (which then generalizes better and this means a lower and lower error in the test set).

Surrogate Loss Functions

- The classic 0-1 loss is not differentiable everywhere

$$\frac{1}{m}\sum_{i=1}^{m}\mathbb{1}(f(x_i;\theta)\neq y_i)$$

- As a substitute use the negative log-likelihood

$$\frac{1}{m}\sum_{i=1}^{m}\log p_{\text{model}}(x_i,y_i;\theta)$$

surrogate loss function

and then use as classifier

$$\hat{y}=\arg\max_y p_{\text{model}}(y|x;\theta)=\arg\max_y p_{\text{model}}(x,y;\theta)$$

instead of $\hat{y}=f(x;\theta)$

Training algorithms do not fully minimize the surrogate function (i.e., they do not necessarily stop when the gradient on the 0-1 loss of other surrogate losses is zero). They typically stop earlier due to other criteria, such as earlier stopping (see regularization). An important criterion to stop the minimization is to use the error on the validation set. This is used to avoid overfitting.

Batch and Minibatch Algorithms

- Loss functions in machine learning can often be written as a sum of a loss over the samples

$$J(\theta) = \mathrm{E}_{x,y \sim p_{\mathrm{data}}}[L(f(x;\theta), y)]$$
$$= \sum_{x,y} p_{\mathrm{data}}(x, y) L(f(x;\theta), y)$$

Another example of loss functions written as a sum over of the samples is shown in the next slide

Batch and Minibatch Algorithms

- Many loss functions in machine learning start from the assumption that samples are IID

$$p(x_1, x_2, \ldots, x_m) = p(x_1)p(x_2)\ldots p(x_m) = \prod_{i=1}^{m} p(x_i)$$

- Then, in the maximum likelihood formulation this can be written as a sum of a loss over the samples

$$\log \prod_{i=1}^{m} p(x_i) = \sum_{i=1}^{m} \log p(x_i)$$

IID = Independent, Identically Distributed

Batch and Minibatch Algorithms

- In ML estimation we then have

$$\theta_{\mathrm{ML}} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{\mathrm{model}}(x_i, y_i; \theta)$$

- This is equivalent to maximizing the surrogate loss function

$$J(\theta) = \mathrm{E}_{x,y\sim\hat{p}_{\mathrm{data}}}[\log p_{\mathrm{model}}(x, y; \theta)]$$

  which is an expectation over the training set

Note

Batch and Minibatch Algorithms

- Computationally challenging to use all the training set in the optimization (e.g., millions of images)
- In practice we minimize the expectation over a random subset (**a batch or minibatch**) of the training set
- In general, the computational saving grows faster than the numerical error
- For a mean estimator on m samples with standard deviation σ, the computational load grows with m, but the standard error is $\sigma/\sqrt{m}$, which grows less than linear

Recall that we do not aim to minimize the training set error, but rather the validation/test set errors. Therefore, the computation of the gradient on a batch is not necessarily a poor computational choice. Indeed, in practice one can observe that the convergence is faster when using batches than when using the full gradient (when keeping the overall execution time fixed).

Another interesting aspect is that this approximation is the same approximation of the gradient of the generalization error.

Batch and Minibatch Algorithms

- **Batch** or **deterministic** gradient methods use the whole training set at each iteration
- **Minibatch stochastic** gradient methods use a batch of samples at each iteration
- **Stochastic** gradient methods use only one sample at each iteration
- Today, it is common practice to call minibatch stochastic simply stochastic

Purely stochastic methods are used with data streams while the others are used with fixed datasets.

Choice of the Batch Size

- Larger batches give better gradients, but the estimate improvement is low
- Small batches might underutilize multicore architectures
- Examples in a batch are processed in parallel; amount of memory defines the maximum size
- GPUs may prefer sizes that are a power of 2
- Small batches may have a regularization effect

Note

Choice of the Batch Size

- The size depends also on the gradient method
  - Methods based on only the loss gradient require small batch sizes
  - Methods based on higher order derivatives (e.g., Hessian) require large batch sizes (to compensate for the larger approximation error)

Note

Shufffling

- An unbiased estimate of the expected gradient requires independent samples
- Using data where the order is fixed might lead to batches where all samples are highly correlated
- Common practice is to randomly visit the training set
- Can save a dataset where the data has been randomly permuted (**data shuffling**)

In practice one goes through the same dataset multiple times. The time it takes to visit the whole dataset is called epoch. The improvement of the model when training for multiple epochs prevails over the loss due to the bias of seeing the same sample multiple times.

Challenges

- Previous approaches to machine learning explored convex formulations to simplify optimization (e.g., support vector machines)

- In contrast, training a neural network is a highly non-convex problem and a number of challenges need to be addressed

Note

Ill-Conditioning

- Consider a second-order Taylor expansion of the cost function at the current parameter estimate

$$J(\theta - \epsilon g) \simeq J(\theta) + \nabla J(\theta)^\top (\theta - \epsilon g - \theta) + \frac{1}{2}(\theta - \epsilon g - \theta)^\top H (\theta - \epsilon g - \theta)$$

$$= J(\theta) - \epsilon g^\top g + \frac{1}{2}\epsilon^2 g^\top H g \qquad \text{with} \quad g = \nabla J(\theta)$$

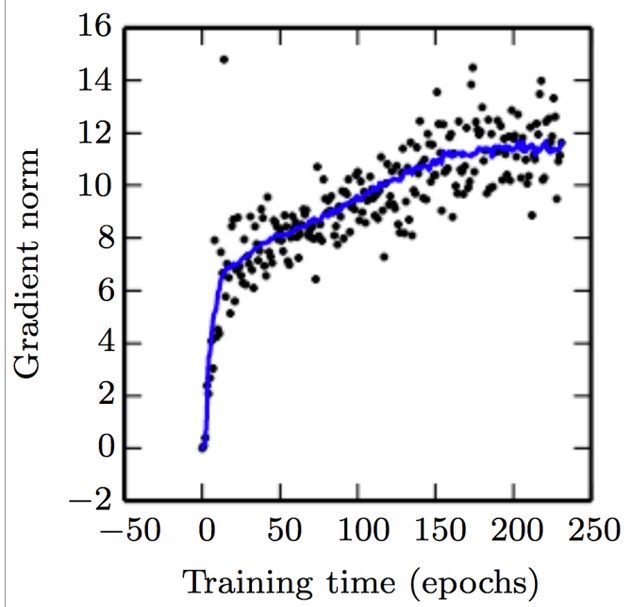
- The cost function decreases if

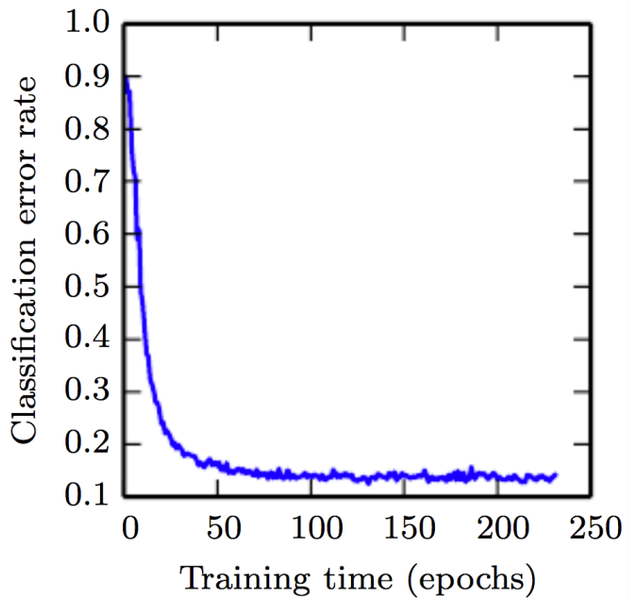
$$g^\top g > \frac{1}{2}\epsilon g^\top H g$$

which depends on the Hessian. Gradient descent may not reach a critical point due to its ill-conditioning

The ill-conditioning of the Hessian may manifest itself at a high curvature point. This high curvature combined with the learning rate e/2 may exceed 1 and cause an increase of the cost instead of a decrease.
Also, the cost function may stop to decrease (or increase) when the two terms cancel each other, although the parameter has not reached any critical point

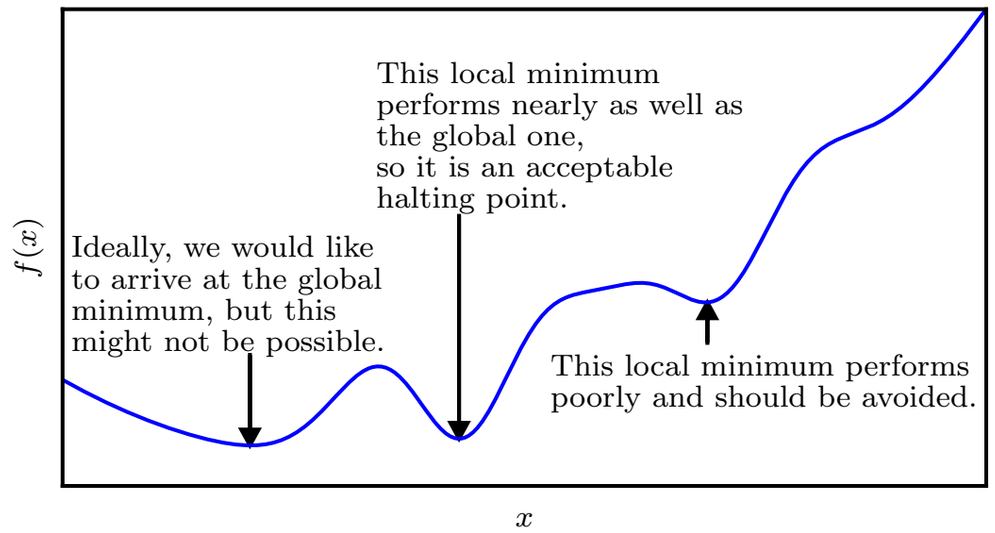
Ill-Conditioning





On the left the norm of the gradient keeps increasing although the error rate on the validation set (in classification) keeps decreasing (left).

Local Minima



Note

Local Minima

- Neural networks have infinite equivalent parameter solutions (i.e., they yield the same output)
- The number of equivalent solutions in a neural network with m layers with n units each can be $n!^m$
- This non-identifiability is known as **weight space symmetry**

In each layer we can take n! permutations of the units as long as the composition of permutations cancels out at the very end. Since we have m layers then we have n! permutations for each of the other n! permutations in each layer.

Local Minima

- Other types of ambiguities include the scaling of inputs matched by an inverse scaling of the weights
- If we ignore the possible numerical convergence issues, these ambiguities are not a problem since all of them yield an equivalent neural network

Note

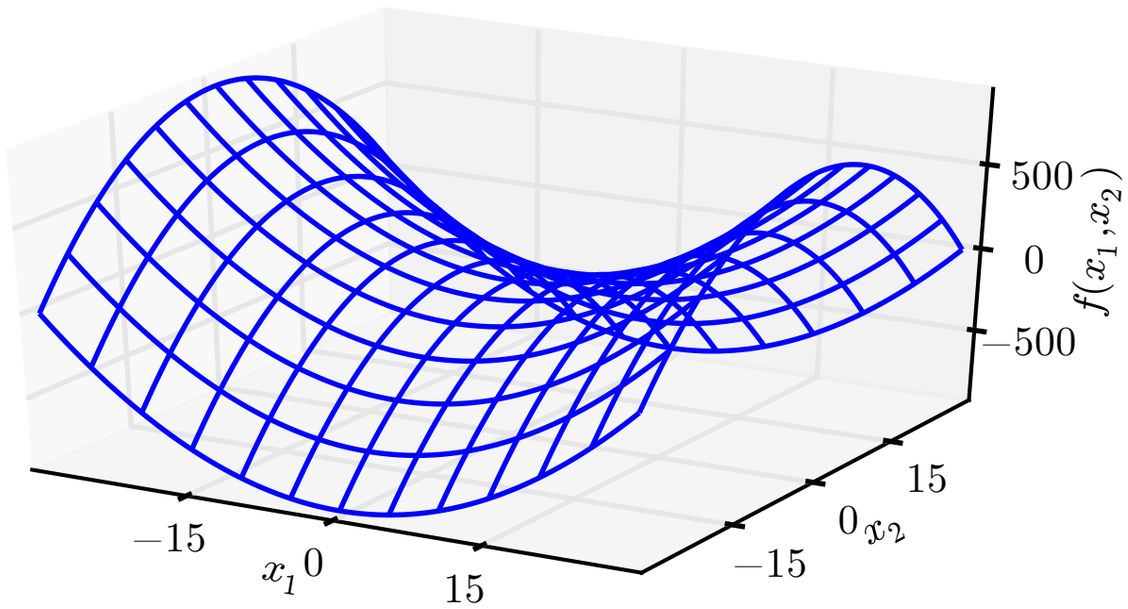
Local Minima

- Local minima can be problematic if they often have a higher cost than the global minimum
- How do we know if this is the problem in our training?

  1. Plot the norm of the gradient over time
  2. If the norm of the gradient does not shrink to zero, then the problem is not global minima (or other critical points)

Even if the gradient is small, it might not be possible to directly conclude that we reached a local minimum.

Saddle Points



At a saddle point we can see that the Hessian will have negative and positive eigenvalues.
Along one cross-section the cost decreases away from the saddle point and along another cross-section the cost increases.

Saddle Points

- In many classes of random functions
  - local minima are more common in low-dim spaces
  - saddle points are more common in high-dim spaces

Note

Saddle Points

- Given a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ the expected ratio between the number of saddle points and local minima grows exponentially with n
- Notice that at a critical point (i.e., where the gradient is zero), we look at the sign of the Hessian eigenvalues
- Local minima (maxima) require all signs to be positive (negative), but saddle points use all other combinations

Note

Saddle Points

- Other interesting properties of random functions are that
  - Saddle points are more likely to be located at critical points with an intermediate cost value
  - Minima are more likely at critical points with a low cost
  - Maxima are more likely at critical points with a high cost

Note

Saddle Points

- Neural network exhibit these behaviors
- For example, shallow autoencoders without nonlinearities (to be presented later in the course) have only global minima
- This effect seems to be observed experimentally also when nonlinearities are present

Note

Saddle Points

- Does the presence of so many saddle points affect training?
- Gradient-based methods might get trapped at saddle points
- Experimentally they seem to escape saddle points very often

Note

Saddle Points

- Notice that gradient descent just moves downhill and does not try to directly find a critical point
- In contrast, Newton methods jump at critical points and might get stuck at saddle points
- One can modify Newton method so that it avoids saddle points, but it is difficult to make it scale

Note

Theoretical Limits

- The optimization of neural networks is largely an open problem
- For example, we do not know if we are converging to the global minimum or even a local minimum
  - As mentioned before this might not be necessary
- Notice that the final solution depends on initialization as in typical non convex problems

Note

Weight Initialization Strategies

- Since the optimization problem is non convex, initialization determines the quality of the solution
- Current initialization strategies are simple and heuristic
- Some initial points may be beneficial to the optimization task, but not to generalization
- One criterion is that the initial parameters need to **break the symmetry** between different units

Same units connected to the same parts must have different initialization otherwise a deterministic algorithm will update them exactly in the same way.

Weight Initialization Strategies

- Two hidden units with the same activation function and inputs should have different initial parameters
- Otherwise a deterministic learning algorithm will update both of these units in the same way
- The goal of diversifying the computed functions motivates random initialization
- Random weights can be obtained from a Gaussian or Uniform distribution

Note

Weight Initialization Strategies

- The magnitude of the random variable matters
  - Large weights may be more effective in breaking symmetry and in preserving gradients through back-propagation
  - Large weights may also lead to exploding gradients, saturation of nonlinear units, or unstable behavior and chaos (in recurrent networks)

Note

Weight Initialization Strategies

- Initialization of weights of a fully connected layer with m inputs and n outputs
- Heuristic #1: Sample from $U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$
- Heuristic #2: Sample from $U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$

These heuristics try to equalize the variance of the activation function among the layers and the variance of the gradients. In the second heuristic the goal is to have different variances for each unit with a different set of inputs/outputs.

Recall that the variance of uniform RVs in [a,b] is 1/12(b-a)^2

Weight Initialization Strategies

- Tuning scaling factors to guarantee optimization properties: e.g., total number of iterations to convergence is independent of depth or orthogonal initialization can be avoided
- However, these methods may not achieve optimal performance, because
  - The correct criterion is unknown
  - Properties may not persist during training
  - Optimization might improve, but not generalization

Some recommendations are to use orthogonal matrices as initialization. However, careful tuning may remove this need.
The usual principle is to preserve the scale of a signal. This criterion might not be ideal in general.

Weight Initialization Strategies

- A drawback of fixing the scaling is that large layers shrink their individual weights
- An alternative is to use sparse initialization (only k elements are non zero)
- Another alternative is to search for the scale as a **hyperparameter tuning**

Sparse initialization may have issues with networks that require a coordinated training of the weights (eg maxout networks).
Gradient descent might take some time to decrease incorrect weights.
A good rule of thumb for choosing initial weights (hyperparameter tuning): compute the standard dev of the activations on a single minibatch of data.
Identify the layer with the smallest activations and increase its weights (until the variance of the output of the layer is 1). This is the layer-sequential unit variance (LSUV) method of Mishkin and Matas (2016).

Bias Initialization Strategies

- Setting the biases to zero works in most scenarios
- Output biases might be initialized to match some meaningful initial output
- Initial biases in nonlinear layers might be chosen to avoid saturation
- In RNNs we might want to set the bias of gates so that the data flows (instead of being forgotten)

For example, in ReLU we may choose a bias of 0.1 to avoid being in the saturating regime.

Learning Based Initialization Strategies

- Another strategy is to initialize weights by **transferring weights** learned via an unsupervised learning method
- This is also a technique called **fine-tuning** which aims at exploiting small annotated datasets by combining them with large unlabeled ones

This relates to Transfer Learning. We learn to solve a task and then transfer the learned knowledge to solve other tasks.

Basic Algorithms

Stochastic Gradient Descent

- Learning rate ϵ_k and initial parameter θ
- **while** (stopping criterion not met) **do**
  - Sample a minibatch of m examples from the training set with the corresponding targets
  - Compute gradient estimate $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x_i; \theta), y_i)$
  - Apply update $\theta \leftarrow \theta - \epsilon_k \hat{g}$
- **end while**

Note

Stochastic Gradient Descent

- Probably the most used algorithm in deep learning
- Main setting is the learning rate ϵ_k
  - It is necessary to gradually decrease the learning rate over iteration time k
  - Sufficient conditions (in addition to others on the cost) to guarantee convergence of SGD are that

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \qquad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

Note

Stochastic Gradient Descent

- A common learning rate update (that does not satisfy the convergence conditions) is

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \quad \text{with} \quad \alpha = \frac{k}{\tau}$$

  when $k > \tau$ then ϵ_k is left constant

- Usually τ is a few hundred epochs and ϵ_τ is 1% of ϵ_0

- In practice one monitors the first several iterations and then uses an initial learning that is as high as possible

Note

Stochastic Gradient Descent

- Computation time per update does not grow with the number of training examples
- Convergence may be achieved before visiting the whole training set

Note

Stochastic Gradient Descent

- Convergence is analyzed via the **excess error**

$$J(\theta) - \min_{\hat{\theta}} J(\hat{\theta})$$

- In a convex problem the excess error is $O\left(\frac{1}{\sqrt{k}}\right)$
- In a strongly convex problem is instead $O\left(\frac{1}{k}\right)$

Let us analyze the convergence rates of SGD. To do so we can use the excess error.
The excess error is the difference between the achieved error and the minimum possible error.

Stochastic Gradient Descent

- Batch gradient descent has better convergence rates than SGD
- The Cramer-Rao bound says that the generalization error cannot decrease faster than $O\left(\frac{1}{k}\right)$
- Thus, it is not worthwhile to look for faster convergence rates (may correspond to overfitting)
- SGD can make very rapid initial progress

Note

Momentum

- The method of **momentum** aims at accelerating learning, especially with high curvature (Hessian), small and noisy gradients
- The key idea in this method is to **accumulate** the gradients over time
- It builds up speed in directions with a gentle but consistent gradient

The convergence rates we just mentioned are in the ideal/best case. In practice, there are many scenarios where SGD can be quite slow and one can provide some useful speed ups. One such speed up comes from the method of momentum.

SGD with Momentum

- Algorithm

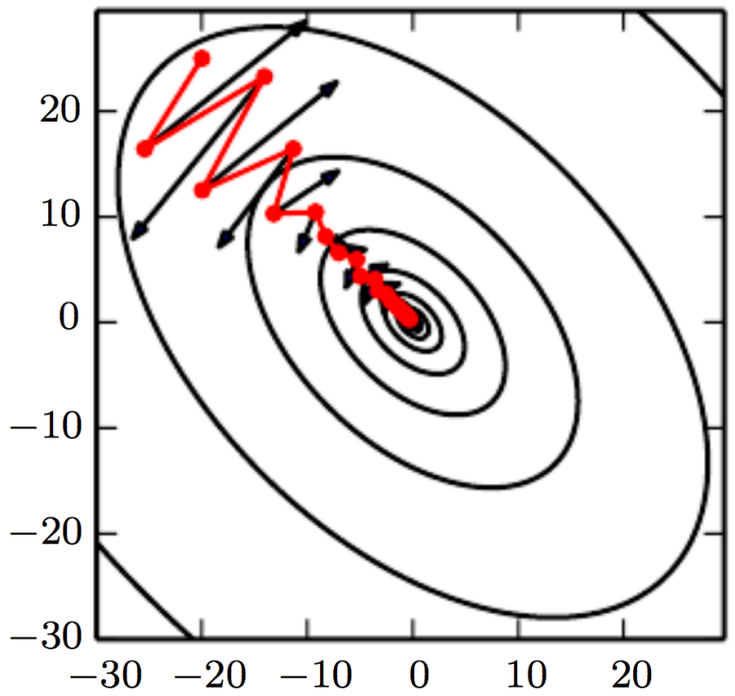
$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x_i; \theta), y_i) \right)$$

$$\theta \leftarrow \theta + v$$

with $\alpha \in [0, 1)$

When \alpha = 0 the method reduces to the usual SGD. With a non zero \alpha (but less than one) the method accumulates gradients over time, but gives more relevance to recent gradients.

SGD with Momentum



The red path is the one followed by the momentum method. The black arrows show the positions that SGD would have followed instead. As the momentum method averages gradients over iteration time, it tends to reduce the oscillations and thus results in a faster convergence.

Nesterov Momentum

- A small variant of the momentum method

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x_i; \theta + \alpha v), y_i) \right)$$

$$\theta \leftarrow \theta + v$$

with $\alpha \in [0, 1)$

Nesterov method provides an update for the conventional gradient descent method and goes from an O(1/k) to an O(1/k^2) convergence rate (for convex problems). However, this does not transfer to SGD.

Adaptive Learning Rates

- The learning rate is one of the most difficult parameters to set
- It has a significant impact on the model performance
- It is therefore treated as a hyperparameter that requires adjustment during training

Note

Delta-bar-delta Algorithm

- A heuristic to choose individual learning rates for each model parameter during training

$$\theta \leftarrow \theta - \epsilon \nabla_\theta J(\theta) \qquad \epsilon \leftarrow \epsilon + \Delta\epsilon$$

$$\Delta\epsilon = \begin{cases} \kappa & \text{if } \bar{\delta}(t-1)\delta(t) > 0 \\ -\phi\epsilon & \text{if } \bar{\delta}(t-1)\delta(t) < 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\delta(t) = \nabla_\theta J(\theta) \qquad \bar{\delta}(t) = (1-\gamma)\delta(t) + \gamma\bar{\delta}(t-1)$$

- Increases linearly but decreases exponentially fast

Note

AdaGrad

- Adapts the learning rates of all parameters by scaling them inversely proportional to the previous ones
- Parameters with large gradients have a rapid decrease of the learning rates
- Parameters with small gradients have a small decrease of the learning rates
- Designed to converge rapidly on convex problems

Note

AdaGrad

- With g the gradient of the loss function on a minibatch

$$r \leftarrow r + g \odot g$$

$$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

$$\theta \leftarrow \theta + \Delta\theta$$

Where r=0 initially, \epsilon is the global learning rate, \delta = 10^-7 (to avoid numerical problems), \odot denotes element-wise products.

One issue is that the accumulation of gradients from the very beginning of the training can lead to too fast a decrease of the learning rate. Performs well but not on all deep learning models

RMSProp

- A modification of AdaGrad for non-convex problems
- Estimated parameter trajectory may include many non-convex structures before reaching a valley
- The accumulation of gradients outside the valley is harmful to the learning rate (too small)
- RMSProp forgets the far past
- It is the commonly used optimization algorithm

Note

RMSProp

- With g the gradient of the loss function on a minibatch

$$r \leftarrow \rho r + (1 - \rho) g \odot g$$

$$\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$$

$$\theta \leftarrow \theta + \Delta\theta$$

\rho is an additional parameter that induces an exponential forgetting of the past gradients squares.

Adam

- ADAptive Moments: can be seen as a combination of RMSProp and momentum
- Includes bias corrections for the first- and second-order moments
- Fairly robust to the choice of hyperparameters, but learning rate might need adjustment

Note

Adam

- With g the gradient of the loss function on a minibatch

$$r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$$

$$s \leftarrow \rho_1 s + (1-\rho_1) g$$

$$\hat{r} \leftarrow \frac{r}{1-\rho_2^t} \qquad \hat{s} \leftarrow \frac{s}{1-\rho_1^t} \qquad t \leftarrow t+1$$

$$\Delta\theta \leftarrow -\frac{\epsilon \hat{s}}{\delta + \sqrt{\hat{r}}}$$

$$\theta \leftarrow \theta + \Delta\theta$$

The two intermediate steps with \hat r and \hat s introduce some scaling with exponentially decaying scales

Choosing the Optimization Algorithm

- Currently there is no consensus on what algorithm performs best
- Most popular choices are: SGD, SGD+Momentum, RMSProp, RMSProp+Momentum, AdaDelta, Adam
- Strategy: Pick one and get familiar with the tuning

Note

Optimization Strategies

- We now explore more specific techniques
  - Batch normalization
  - Coordinate descent
  - Polyak averaging
  - Supervised pretraining
  - Continuation methods and curriculum learning

Note

Batch Normalization

- The gradient of the cost function tells us how much the cost changes as each parameter (in isolation) changes

- When we update all layers simultaneously, due to the compositionality of the neural networks the effects may be unpredictable

"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Ioffe and Szegedy, 2015

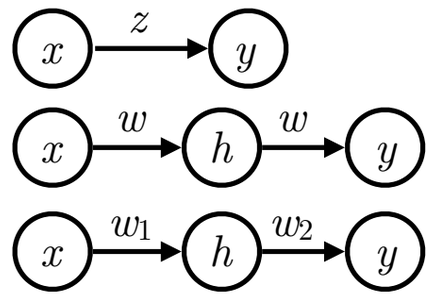
Notes

Batch Normalization

- Example

  Examine the gradient update of the weights in 3 equivalent networks (different parametrizations)

  1. 1-layer $\hat{y} = xz$
  2. 2-layer (shared) $\hat{y} = xww$
  3. 2-layer $\hat{y} = xw_1w_2$



Let us illustrate the issues of the compositionality of neural networks by examining what happens to the loss function of 3 equivalent networks (with different parametrizations).

When we pick the weights so that the I/O mapping is the same for the three networks, we expect the gradient update to do the same job in all 3 cases. However, this does not happen.

Notice that the 1-layer case gives the optimal update and we use it as a reference.

Batch Normalization

- Example

  Examine the gradient update of the weights in 3 equivalent networks (different parametrizations)

  1. 1-layer $\hat{y} \leftarrow x(w^2 - \epsilon x)$ at $z = w^2$
  2. 2-layer (shared) $\hat{y} \leftarrow x(w^2 - 2\epsilon x w)$
  3. 2-layer $\hat{y} \leftarrow x(w - \epsilon x w)^2$ at $w_1 = w_2 = w$
     $= x(w^2 - 2\epsilon x w^2 + \epsilon^2 x^2 w^2)$

Here we show the updated weights in the three cases and substitute the parameters that make the three networks identical.
We can see that the updated weights are different in the three cases.

Batch Normalization

- Example (general case)

  The loss is the output layer $\hat{y} = x w_1 w_2 \dots w_k$

  The gradient is $g_i^t = \nabla_{w_i} \hat{y} = x \prod_{j \neq i} w_j$

  Weight update $w_i^{t+1} = w_i^t - \epsilon g_i^t$

Gradient descent provides the update for each parameter to improve the loss given that all the other parameters are fixed. However we update all parameters simultaneously.

Batch Normalization

$$\hat{y}(x)^{t+1} = x\prod_{i=1}^{k} w_i^{t+1} = x\prod_{i=1}^{k}\left(w_i^t - \epsilon g_i^t\right)$$

$$= x\prod_{i=1}^{k}\left(w_i^t - \epsilon x\prod_{j\neq i} w_j^t\right) = x\prod_{i=1}^{k}\left(w_i^t - \frac{1}{w_i^t}\epsilon x\prod_{j} w_j^t\right)$$

$$= x\prod_{i=1}^{k}\frac{(w_i^t)^2 - \epsilon x\kappa}{w_i^t} = \frac{x}{\kappa}\prod_{i=1}^{k}\left((w_i^t)^2 - \epsilon x\kappa\right)$$

$$= x\left(\kappa - \epsilon x\sum_i\prod_{j\neq i}(w_j^t)^2 + \cdots + (-1)^k(\epsilon x)^k(\kappa)^{k-1}\right)$$

The output layer depends on high order terms which make the choice of \epsilon extremely difficult (because the variation of the gradient magnitude depends on the current values of the weights).

We do not know if the first order or the higher order terms will be the largest. So setting \epsilon becomes not trivial (it depends on the values of the weights) or we need to be conservative and set it to a very small value.

Using Newton's method or higher order methods is not a feasible solution.

Batch Normalization

$$\hat{y}(x)^{t+1} = x \prod_{i=1}^{k} w_i^{t+1} = x \prod_{i=1}^{k} \left(w_i^t - \epsilon g_i^t\right)$$

$$= x \prod_{i=1}^{k} \left(w_i^t - \epsilon x \prod_{j \neq i} w_j^t \right) = x \prod_{i=1}^{k} \left(w_i^t - \frac{1}{w_i^t} \epsilon x \prod_{j} w_j^t \right)$$

$$= x \prod_{i=1}^{k} \frac{(w_i^t)^2 - \epsilon x \kappa}{w_i^t} = \frac{x}{\kappa} \prod_{i=1}^{k} \left((w_i^t)^2 - \epsilon x \kappa\right)$$

$$= x \left(\kappa - \epsilon x \sum_i \prod_{j \neq i} (w_j^t)^2 + \cdots + (-1)^k (\epsilon x)^k (\kappa)^{k-1} \right)$$

Batch Normalization

$$\hat{y}(x)^{t+1} = x\prod_{i=1}^{k} w_i^{t+1} = x\prod_{i=1}^{k}\left(w_i^t - \epsilon g_i^t\right)$$

$$= x\prod_{i=1}^{k}\left(w_i^t - \epsilon x\prod_{j\neq i} w_j^t\right) = x\prod_{i=1}^{k}\left(w_i^t - \frac{1}{w_i^t}\epsilon x\prod_{j} w_j^t\right)$$

$$= x\prod_{i=1}^{k}\frac{(w_i^t)^2 - \epsilon x\kappa}{w_i^t} = \frac{x}{\kappa}\prod_{i=1}^{k}\left((w_i^t)^2 - \epsilon x\kappa\right)$$

$$= x\left(\kappa - \epsilon x\sum_i\prod_{j\neq i}(w_j^t)^2 + \cdots + (-1)^k(\epsilon x)^k(\kappa)^{k-1}\right)$$

Batch Normalization

$$\hat{y}(x)^{t+1} = x \prod_{i=1}^{k} w_i^{t+1} = x \prod_{i=1}^{k} \left(w_i^t - \epsilon g_i^t\right)$$

$$= x \prod_{i=1}^{k} \left(w_i^t - \epsilon x \prod_{j \neq i} w_j^t \right) = x \prod_{i=1}^{k} \left(w_i^t - \frac{1}{w_i^t} \epsilon x \prod_{j} w_j^t \right)$$

$$= x \prod_{i=1}^{k} \frac{(w_i^t)^2 - \epsilon x \kappa}{w_i^t} = \frac{x}{\kappa} \prod_{i=1}^{k} \left((w_i^t)^2 - \epsilon x \kappa\right)$$

$$= x \left(\kappa - \epsilon x \sum_i \prod_{j \neq i} (w_j^t)^2 + \cdots + (-1)^k (\epsilon x)^k (\kappa)^{k-1} \right)$$

Batch Normalization

- Batch normalization proposes a re-parametrization that does not compromise the original capacity but changes the learning dynamics

- Let us consider the i-th layer output on a batch

$$h_i = h_{i-1} w_i$$

with input $h_{i-1} \in \mathbf{R}^{m \times c}$ and weights $w_i \in \mathbf{R}^{c \times d}$

- Let us define $H = h_i \in \mathbf{R}^{m \times d}$

batch size

Notes

Batch Normalization

- Proposed re-parametrization

$$\hat{H} = \gamma H' + \beta = \gamma \frac{H - \mu(H)}{\sigma(H)} + \beta$$

with (batch) mean and standard deviation

$$\mu(H) = \frac{1}{m} \sum_j H_{j,\cdot}$$

$$\sigma(H) = \sqrt{\delta + \frac{1}{m} \sum_j (H_{j,\cdot} - \mu(H))^2}$$

At run-time the mean and standard deviations are fixed based on the training set (otherwise one would need to compute statistics on a single input sample — Instance Normalization).

Batch Normalization

- Recall previous 3-network example

1. 1-layer $$H' = \frac{xz - \frac{1}{m}\sum_j x_j z}{\sqrt{\frac{1}{m}\sum_j \left(x_j z - \frac{1}{m}\sum_i x_i z\right)^2}} = \frac{x - \mu_x}{\sigma_x}$$
2. 2-layer (shrd) $$H' = \frac{xw^2 - \frac{1}{m}\sum_j x_j w^2}{\sqrt{\frac{1}{m}\sum_j \left(x_j w^2 - \frac{1}{m}\sum_i x_i w^2\right)^2}} = \frac{x - \mu_x}{\sigma_x}$$
3. 2-layer $$H' = \frac{xw_1w_2 - \frac{1}{m}\sum_j x_j w_1 w_2}{\sqrt{\frac{1}{m}\sum_j \left(x_j w_1 w_2 - \frac{1}{m}\sum_i x_i w_1 w_2\right)^2}} = \frac{x - \mu_x}{\sigma_x}$$

All three cases are independent of the original (linear) parameters. Since the new mapping is invariant to the weights, their gradient will be zero. Basically, these weights will not receive any update.

Batch Normalization

- In all 3 networks we have the same mapping

$$\hat{H} = \gamma H' + \beta$$

and hence the same gradient update

$$\hat{H} \leftarrow (\gamma - \epsilon H')H' + (\beta - \epsilon)$$

The new parametrization makes the updates in all three cases identical.
The gradient step is computed wrt \gamma and \beta.

Coordinate Descent

- Rather than updating all variables (weights) one can update only a subset of them at each iteration (**block coordinate descent**)
- This is useful when updating a subset of approximately independent variables or when the update is more efficient on a separate block

Note

Coordinate Descent

- Example

$$J(H, W) = |H|_1 + |X - W^\top H|_2^2$$

- J is not convex wrt both H and W
- Optimization wrt H alone is convex (efficient)
- Optimization wrt W alone is convex (efficient)

Note

Polyak Averaging

- Average the parameters estimated during gradient descent

$$\hat{\theta}^t = \frac{1}{t}\sum_{i=1}^{t}\theta^i$$

- The average estimate dampens oscillations and may converge more quickly

- On non convex problems one uses

$$\hat{\theta}^t = \alpha\theta^{t-1} + (1-\alpha)\theta^t \qquad \alpha \in (0,1)$$

Relates to Momentum

Supervised Pretraining

- Instead of directly solving a difficult problem, break down the complexity
  - Use simpler models or
  - Work on simpler problems
- This strategy is called **pretraining**

Note

Supervised Pretraining

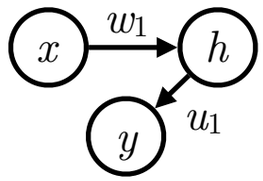
- To solve the original problem one can compose the simpler problems/models
- The composition may not be optimal; in this case one can use **fine-tuning**, which basically uses pretraining as an initialization to the full optimization

Note

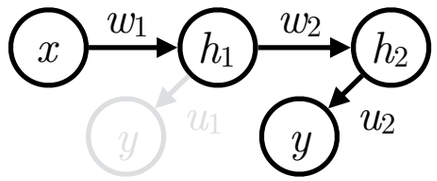
Supervised Pretraining

- Example of greedy supervised pretraining

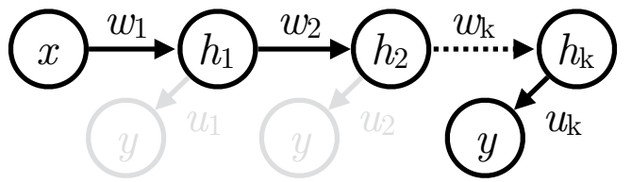
start with



then add the next layer



until the last layer



Note

Supervised Pretraining

- It relates to **transfer learning**
- Example
  - Train a network on a subset of the 1K ImageNet categories; then use the first N layers as initialization to another network trained on another subset of the 1K ImageNet categories, but with less data

Note

Supervised Pretraining

- First, train an easy to train **Teacher** network (low depth and large width)
- Then, train a **Student** network (high depth and small width) to predict the labels and the middle layer value of the Teacher network (this are so-called **hints** to the hidden layers)
- This strategy is used by **FitNets**

Note

Designing Models to Aid Optimization

- Rather than improving the optimization by working on the algorithms, design the model so that optimization is easier
- This is the strategy that has led to the modern deep learning neural networks
  - SGD is the same as in the 1980s
  - Sigmoids substituted by ReLUs and MaxOuts
  - In general: More linearity (e.g., skip connections)

Note

Continuation Methods

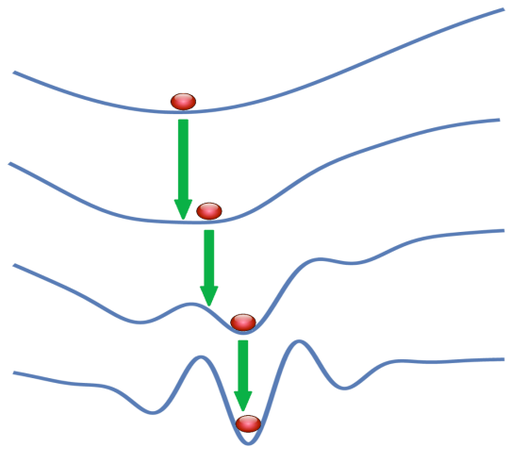
- Improving the local estimates (e.g., Adam, RMSProp, AdaGrad) has a limited effect
- A general approach is to find a good initialization (with a short path to the solution) so that gradient descent can easily find the local optimum
- **Continuation methods** provide a strategy to achieve such initialization

Note

Continuation Methods

- Build a sequence of objective functions $\{J_i\}_{i=0,..,n}$ on the same set of variables, such that one solves objectives of increasing difficulty and $J_n = J$, the original objective function

- Typically, easier objectives are smooth versions of the original one



"A Theoretical Analysis of Optimization by Gaussian Continuation," Hossein Mobahi and John W. Fisher III, 2015

Note

Continuation Methods

- A rationale behind smoothing the objective is that it becomes approximately convex
- Other related methods include the **Majorization Minimization** method, which builds a *surrogate function* (usually convex or well-behaved) that bounds the original objective

Note

Curriculum Learning

- A special type of continuation method that imitates the way humans learn
- Learn by starting from easy concepts and then move to more difficult ones built on the previous ones
- Can be achieved by weighing differently the samples in the objective function or by biasing the sampling
- In RNNs stochastic curriculum learning (a mix of easy and hard examples)

In the stochastic curriculum learning the rates of easy and hard examples are changed. Initially the mix has few hard examples and then later many more.