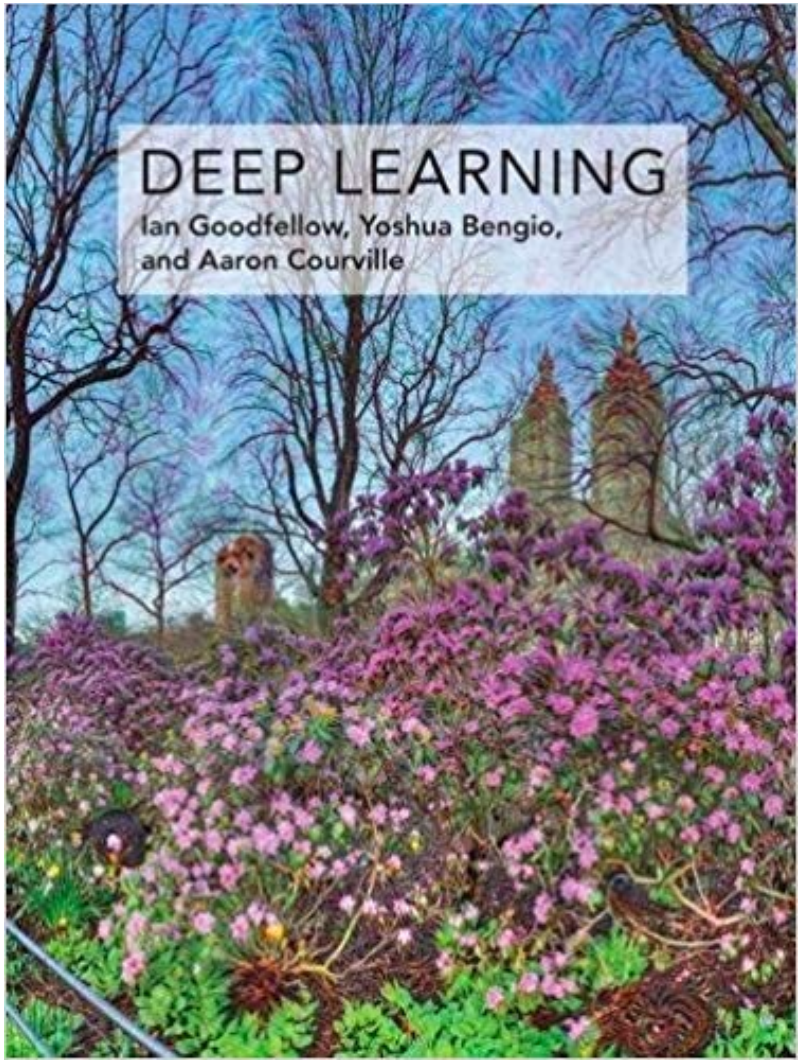
國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning
深度學習
Fall 2023

Optimization for Training
(Chapter 8.6-8.7)

Prof. Chia-Han Lee
李佳翰 教授



- Figure source: Textbook and Internet
- You are encouraged to buy the textbook.
- Please respect the copyright of the textbook. Do not distribute the materials to other people.

Approximate second-order methods

- In this lecture we discuss the application of second-order methods to training deep networks.

- For simplicity of exposition, the only objective function we examine is the empirical risk:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)}). \qquad (8.25)$$

- The methods we discuss here extend readily, however, to more general objective functions, such as those that include parameter regularization terms.

Newton's method

- Newton's method is an optimization scheme based on using a second-order Taylor series expansion to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0), \tag{8.26}$$

  where $\boldsymbol{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$.

- If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \tag{8.27}$$

Newton's method

- Thus for a locally quadratic function (with positive definite $\boldsymbol{H}$), by rescaling the gradient by $\boldsymbol{H}^{-1}$, Newton's method jumps directly to the minimum.

- If the objective function is convex but not quadratic (there are higher-order terms), this update can be iterated, yielding the training algorithm associated with Newton's method.

- For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step iterative procedure. First, update or compute the inverse Hessian. Second, update the parameters according to

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \tag{8.27}$$

Newton's method

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$

Require: Initial parameter $\boldsymbol{\theta}_0$

Require: Training set of m examples

while stopping criterion not met **do**

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian inverse: $\boldsymbol{H}^{-1}$

Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$

Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Newton's method

- If the eigenvalues of the Hessian are not all positive, for example, near a saddle point, then Newton's method can actually cause updates to move in the wrong direction.

- This situation can be avoided by regularizing the Hessian. Common regularization strategies include adding a constant, α, along the diagonal of the Hessian. The regularized update becomes

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0). \tag{8.28}$$

- This regularization strategy is used in approximations to Newton's method, such as the Levenberg-Marquardt algorithm.

Newton's method

- When there are more extreme directions of curvature, the value of α would have to be sufficiently large to offset the negative eigenvalues. As α increases in size, however, the Hessian becomes dominated by the $\alpha \boldsymbol{I}$ diagonal, and the direction chosen by Newton's method converges to the standard gradient divided by α.

- When strong negative curvature is present, α may need to be so large that Newton's method would make smaller steps than gradient descent with a properly chosen learning rate.

Newton's method

- The application of Newton's method for training large NNs is limited by the significant computational burden.

- The number of elements in the Hessian is squared in the number of parameters, so with k parameters (and for even very small neural networks, the number of parameters k can be in the millions), Newton's method would require the inversion of a $k \times k$ matrix—with computational complexity of $O(k^3)$.

- Since the parameters will change with every update, the inverse Hessian has to be computed at every training iteration. As a consequence, only networks with a very small number of parameters can be practically trained via Newton's method.

Conjugate gradients

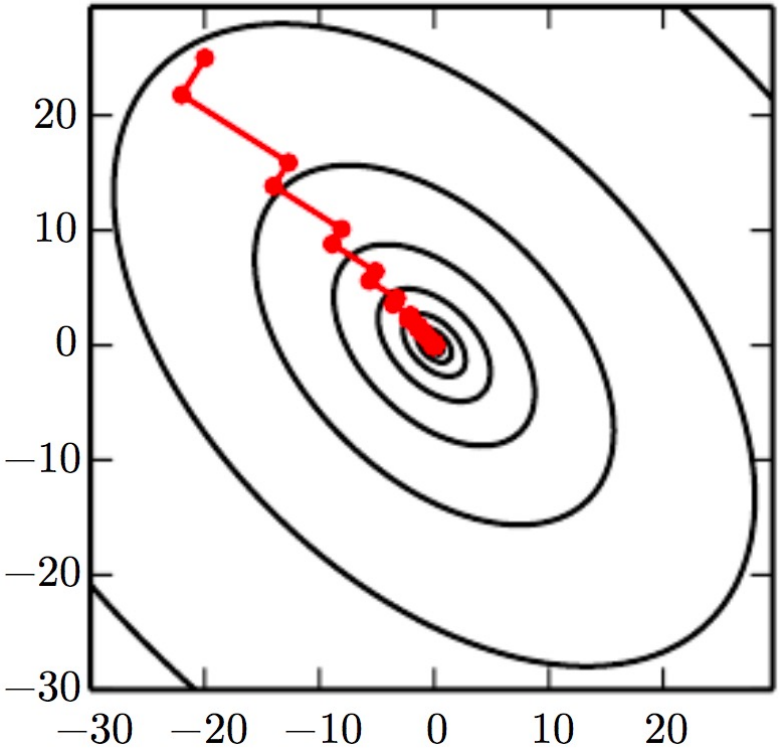
- Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions.

- The inspiration for this approach follows from the study of the weakness of the method of steepest descent, where line searches (to evaluate $f(\boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$ for several values of ϵ and choose the one that results in the smallest objective function value) are applied iteratively in the direction associated with the gradient.

- Figure 8.6 illustrates how the method of steepest descent progresses in a rather ineffective back-and-forth zig-zag pattern. This happens because each line search direction, when given by the gradient, is guaranteed to be orthogonal to the previous line search direction.

Conjugate gradients

The method of steepest descent involves jumping to the point of lowest cost along the line defined by the gradient at the initial point on each step. This resolves some of the problems seen with using a fixed learning rate in figure 4.6, but even with the optimal step size the algorithm still makes back-and-forth progress toward the optimum.



Conjugate gradients

- Let the previous search direction be $\boldsymbol{d}_{t-1}$. At the minimum, where the line search terminates, the directional derivative is zero in direction $\boldsymbol{d}_{t-1}$: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \boldsymbol{d}_{t-1} = 0$. Since the gradient at this point defines the current search direction, $\boldsymbol{d}t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ will have no contribution in the direction $\boldsymbol{d}_{t-1}$. Thus $\boldsymbol{d}_t$ is orthogonal to $\boldsymbol{d}_{t-1}$.

- The choice of orthogonal directions of descent do not preserve the minimum along the previous search directions. This gives rise to the zig-zag pattern of progress, where by following the gradient at the end of each line search we undo progress we have already made in the direction of the previous line search. Conjugate gradients seeks to address this problem.

Conjugate gradients

- In the method of conjugate gradients, we seek to find a search direction that is conjugate to the previous line search direction; that is, it will not undo progress made in that direction.

- At training iteration t, the next search direction $\boldsymbol{d}_t$ takes the form:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}, \tag{8.29}$$

where β_t is a coefficient whose magnitude controls how much of the direction, $\boldsymbol{d}_{t-1}$, we should add back to the current search direction.

Conjugate gradients

- Two directions, $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$, are defined as conjugate if $\boldsymbol{d}_t^{\mathrm{T}}\boldsymbol{H}\boldsymbol{d}_{t-1} = 0$, where $\boldsymbol{H}$ is the Hessian matrix.

- The straightforward way to impose conjugacy would involve calculation of the eigenvectors of $\boldsymbol{H}$ to choose β_t, which would not satisfy our goal of developing a method that is more computationally viable than Newton's method for large problems.

- Can we calculate the conjugate directions without resorting to the calculations of the eigenvectors of $\boldsymbol{H}$? Fortunately, the answer to that is yes.

Conjugate gradients

- Two popular methods for computing β_t are

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.30}$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.31}$$

- For a quadratic surface, the conjugate directions ensure that the gradient along the previous direction does not increase in magnitude. We therefore stay at the minimum along the previous directions. Thus, in a k-dimensional parameter space, the conjugate gradient method requires at most k line searches to achieve the minimum.

Conjugate gradients

Algorithm 8.9 The conjugate gradient method

Require: Initial parameters $\boldsymbol{\theta}_0$

Require: Training set of m examples

Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$

Initialize $g_0 = 0$

Initialize $t = 1$

while stopping criterion not met **do**

Initialize the gradient $\boldsymbol{g}_t = \mathbf{0}$

Compute gradient: $\boldsymbol{g}_t \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak-Ribière)

(Nonlinear conjugate gradient: optionally reset β_t to zero, for example if t is a multiple of some constant k, such as $k = 5$)

Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$

Perform line search to find: $\epsilon^* = \text{argmin}_\epsilon \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$

(On a truly quadratic cost function, analytically solve for ϵ^* rather than explicitly searching for it)

Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$

$t \leftarrow t + 1$

end while

Nonlinear conjugate gradients

- Without any assurance that the objective is quadratic, the conjugate directions are no longer assured to remain at the minimum of the objective for previous directions. As a result, the nonlinear conjugate gradients algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient.

- It is often beneficial to initialize the optimization with a few iterations of stochastic gradient descent before commencing nonlinear conjugate gradients.

- While the (nonlinear) conjugate gradients algorithm has traditionally been cast as a batch method, minibatch versions have been used successfully for training NNs.

BFGS

- The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm attempts to bring some of the advantages of Newton's method without the computational burden.

- BFGS is similar to the conjugate gradient method. However, BFGS takes a more direct approach to the approximation of Newton's update.

- Recall that Newton's update is given by

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \tag{8.32}$$

where $\boldsymbol{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$. The primary computational difficulty in applying Newton's update is the calculation of the inverse Hessian $\boldsymbol{H}^{-1}$.

BFGS

- The approach adopted by quasi-Newton methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix $\boldsymbol{M}_t$ that is iteratively refined by low-rank updates to become a better approximation of $\boldsymbol{H}^{-1}$.

- Once the inverse Hessian approximation $\boldsymbol{M}_t$ is updated, the direction of descent $\boldsymbol{\rho}_t$ is determined by $\boldsymbol{\rho}_t = \boldsymbol{M}_t \, \boldsymbol{g}_t$. A line search is performed in this direction to determine the size of the step, ϵ^*, taken in this direction. The final update to the parameters is given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t. \tag{8.33}$$

BFGS

- Like the method of conjugate gradients, the BFGS algorithm iterates line searches with the direction incorporating second-order information. Unlike conjugate gradients, however, the success of the approach is not heavily dependent on the line search finding a point very close to the true minimum along the line.

- Thus, relative to conjugate gradients, BFGS has the advantage that it can spend less time refining each line search.

- However, the BFGS algorithm must store the inverse Hessian matrix, $\boldsymbol{M}$, that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters.

Limited memory BFGS (L-BFGS)

- The memory costs of the BFGS algorithm can be significantly decreased by avoiding storing the complete inverse Hessian approximation $\boldsymbol{M}$.

- The L-BFGS algorithm begins with the assumption that $\boldsymbol{M}^{(t-1)}$ is the identity matrix, rather than storing the approximation from one step to the next.

- If used with exact line searches, the directions defined by L-BFGS are mutually conjugate. However, this procedure remains well behaved when the minimum of the line search is reached only approximately.

- L-BFGS can be generalized to include more information about the Hessian by storing some of the vectors used to update $\boldsymbol{M}$ at each time step, with costs $O(n)$.

Optimization strategies and meta-algorithms

- Many optimization techniques are not exactly algorithms but rather general templates that can be specialized to yield algorithms, or subroutines that can be incorporated into many different algorithms.

Batch normalization

- Batch normalization is a method of adaptive reparametrization, motivated by the difficulty of training very deep models.

- The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all the layers simultaneously.

- When we make the update, unexpected results can happen because many functions composed together are changed simultaneously, using updates that were computed under the assumption that the other functions remain constant.

Batch normalization

- As a simple example, suppose we have a deep neural network that has only one unit per layer and does not use an activation function at each hidden layer: $\hat{y} = x\omega_1\omega_2\omega_3 \ldots \omega_l$. Here, ω_i provides the weight used by layer i. The output of layer i is $h_i = h_{i-1}\omega_i$. The output $\hat{y}$ is a linear function of the input x but a nonlinear function of the weights ω_i.

- Suppose our cost function has put a gradient of 1 on $\hat{y}$, so we wish to decrease $\hat{y}$ slightly. The back-propagation algorithm can then compute a gradient $\boldsymbol{g} = \nabla_{\boldsymbol{\omega}}\hat{y}$.

Batch normalization

- Consider what happens when we make an update $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \epsilon \boldsymbol{g}$. The first-order Taylor series approximation of $\hat{y}$ predicts that the value of $\hat{y}$ will decrease by $\epsilon \boldsymbol{g}^{\mathrm{T}} \boldsymbol{g}$. If we wanted to decrease $\hat{y}$ by 0.1, this first-order information available in the gradient suggests we could set the learning rate ϵ to $\frac{0.1}{\boldsymbol{g}^{\mathrm{T}} \boldsymbol{g}}$.

- The actual update will include second/third-order effects, up to effects of order l. The new value of $\hat{y}$ is given by

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \ldots (w_l - \epsilon g_l). \tag{8.34}$$

An example of one second-order term arising from this update is $\epsilon g_1 g_2 \prod_{i=3}^{l} \omega_i$. This term might be negligible if $\prod_{i=3}^{l} \omega_i$ is small, or might be exponentially large if the weights on layers 3 through l are greater than 1.

Batch normalization

- This makes it very hard to choose an appropriate learning rate, because the effects of an update to the parameters for one layer depend so strongly on all the other layers.

- Second-order optimization algorithms address this issue by computing an update that takes these second-order interactions into account, but in very deep networks, even higher-order interactions can be significant.

- Even second-order optimization algorithms are expensive and usually require numerous approximations that prevent them from truly accounting for all significant second-order interactions. Building an n-th order optimization algorithm for $n > 2$ thus seems hopeless.

Batch normalization

- Batch normalization provides an elegant way of reparametrizing almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers. Batch normalization can be applied to any input or hidden layer.

- Let $\boldsymbol{H}$ be a minibatch of activations of the layer to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix. To normalize $\boldsymbol{H}$, we replace it with

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{8.35}$$

where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit.

Batch normalization

- The arithmetic is based on broadcasting the vector $\boldsymbol{\mu}$ and the vector $\boldsymbol{\sigma}$ to be applied to every row of the matrix $\boldsymbol{H}$. Within each row, the arithmetic is element-wise, so $\boldsymbol{H}_{i,j}$ is normalized by subtracting $\boldsymbol{\mu}_j$ and dividing by $\boldsymbol{\sigma}_j$. The rest of the network then operates on $\boldsymbol{H}'$ in exactly the same way that the original network operated on $\boldsymbol{H}$.

- At training time,

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \boldsymbol{H}_{i,:} \tag{8.36}$$

and

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2}, \tag{8.37}$$

where δ is a small positive value such as 10^{-8} to avoid encountering the undefined gradient of $\sqrt{z}$ at $z = 0$.

Batch normalization

- Crucially, we back-propagate through these operations for computing the mean and the standard deviation, and for applying them to normalize $\boldsymbol{H}$.

- This means that the gradient will never propose an operation that acts simply to increase the standard deviation or mean of h_i; the normalization operations remove the effect of such an action and zero out its component in the gradient. This was a major innovation of the batch normalization approach.

Batch normalization

- The previous approach of adding penalties to the cost function to encourage units to have normalized activation statistics usually resulted in imperfect normalization.

- The previous approach of intervening to renormalize unit statistics after each gradient descent step usually resulted in significant wasted time, as the learning algorithm repeatedly proposed changing the mean and variance, and the normalization step repeatedly undid this change.

- Batch normalization reparametrizes the model to make some units always be standardized by definition, deftly sidestepping both problems.

Batch normalization

- At test time, μ and σ may be replaced by running averages that were collected during training time. This allows the model to be evaluated on a single example, without needing to use definitions of μ and σ that depend on an entire minibatch.

Batch normalization

- Revisiting the $\hat{y} = x\omega_1\omega_2 \ldots \omega_l$ example, we can resolve the difficulties in learning this model by normalizing h_{l-1}. Suppose that x is drawn from a unit Gaussian. Then h_{l-1} will also come from a Gaussian, because the transformation from x to h_l is linear. However, h_{l-1} will no longer have zero mean and unit variance. After applying batch normalization, we obtain the normalized $\hat{h}_{l-1}$ that restores the zero mean and unit variance properties.

- For almost any update to the lower layers, $\hat{h}_{l-1}$ will remain a unit Gaussian. The output $\hat{y}$ may then be learned as a simple linear function $\hat{y} = \omega_l \hat{h}_{l-1}$. Learning in this model is now very simple because the parameters at the lower layers do not have an effect in most cases; their output is always renormalized to a unit Gaussian.

Batch normalization

- In some rare cases, the lower layers can have an effect. Changing one of the lower layer weights to 0 can make the output become degenerate, and changing the sign of one of the lower weights can flip the relationship between $\hat{h}_{l-1}$ and y.

- Without normalization, nearly every update would have an extreme effect on the statistics of h_{l-1}. Batch normalization has thus made this model significantly easier to learn.

Batch normalization

- In this example, the ease of learning came at the cost of making the lower layers useless. In our linear example, the lower layers no longer have any harmful effect, but they also no longer have any beneficial effect. This is because we have normalized out the first- and second-order statistics, which is all that a linear network can influence. In a deep neural network with nonlinear activation functions, the lower layers can perform nonlinear transformations of the data, so they remain useful.

Batch normalization

- Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network containing that unit.

- To maintain the expressive power of the network, it is common to replace the batch of hidden unit activations $\boldsymbol{H}$ with $\boldsymbol{\gamma H' + \beta}$ rather than simply the normalized $\boldsymbol{H'}$. The variables $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned parameters that allow the new variable to have any mean and standard deviation.

- At first glance, this may seem useless—why did we set the mean to 0, and then introduce a parameter that allows it to be set back to any arbitrary value $\boldsymbol{\beta}$?

Batch normalization

- The answer is that the new parametrization can represent the same family of functions of the input as the old parametrization, but the new parametrization has different learning dynamics.

- In the old parametrization, the mean of $\boldsymbol{H}$ was determined by a complicated interaction between the parameters in the layers below $\boldsymbol{H}$.

- In the new parametrization, the mean of $\boldsymbol{\gamma H' + \beta}$ is determined solely by $\boldsymbol{\beta}$. The new parametrization is much easier to learn with gradient descent.

Batch normalization

- Most neural network layers take the form of $\phi(\boldsymbol{XW} + \boldsymbol{b})$. It is natural to wonder whether we should apply batch normalization to the input $\boldsymbol{X}$, or to the transformed value $\boldsymbol{XW} + \boldsymbol{b}$.

- Batch normalization is recommended to be applied to $\boldsymbol{XW} + \boldsymbol{b}$. More specifically, $\boldsymbol{XW} + \boldsymbol{b}$ should be replaced by a normalized version of $\boldsymbol{XW}$ because the bias term becomes redundant with the β parameter.

- In convolutional networks, it is important to apply the same normalizing $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ at every spatial location within a feature map, so that the statistics of the feature map remain the same regardless of spatial location.

Polyak averaging

- Polyak averaging consists of averaging several points in the trajectory through parameter space visited by an optimization algorithm.

- If t iterations of gradient descent visit points $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(t)}$, then the output of the Polyak averaging algorithm is $\widehat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t}\sum_i \boldsymbol{\theta}^{(i)}$. On some problem classes, such as gradient descent applied to convex problems, this approach has strong convergence guarantees.

- When applied to neural networks, the idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all the locations on either side should be close to the bottom of the valley.

Polyak averaging

- In nonconvex problems, the path taken by the optimization trajectory can be very complicated and visit many different regions. Including points in parameter space from the distant past that may be separated from the current point by large barriers in the cost function does not seem like a useful behavior.

- As a result, when applying Polyak averaging to nonconvex problems, it is typical to use an exponentially decaying running average:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha \hat{\boldsymbol{\theta}}^{(t-1)} + (1-\alpha)\boldsymbol{\theta}^{(t)}. \tag{8.39}$$

Supervised pretraining

- Sometimes, directly training a model to solve a specific task can be too ambitious if the model is complex and hard to optimize or if the task is very difficult.
- It is sometimes more effective to train a simpler model to solve the task, then make the model more complex.
- It can also be more effective to train the model to solve a simpler task, then move on to confront the final task.
- These strategies that involve training simple models on simple tasks before confronting the challenge of training the desired model to perform the desired task are collectively known as pretraining.

Supervised pretraining

- Greedy algorithms break a problem into many components, then solve for the optimal version of each component in isolation. Unfortunately, combining the individually optimal components is not guaranteed to yield an optimal complete solution.

- Nonetheless, greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution, and the quality of a greedy solution is often acceptable if not optimal.

- Greedy algorithms may also be followed by a fine-tuning stage in which a joint optimization algorithm searches for an optimal solution to the full problem.

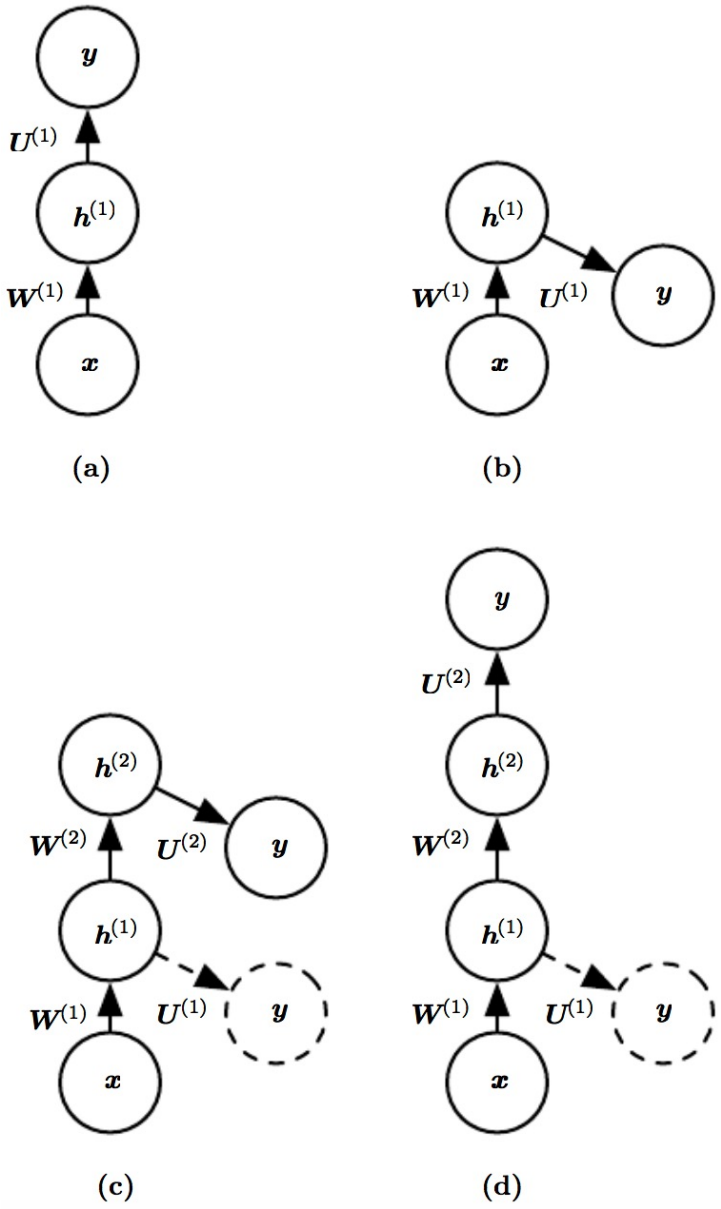
Supervised pretraining

- Initializing the joint optimization algorithm with a greedy solution can greatly speed it up and improve the quality of the solution.

- Those pretraining algorithms that break supervised learning problems into other simpler supervised learning problems are known as greedy supervised pretraining.

- In the original version of greedy supervised pretraining, each stage consists of a supervised learning training task involving only a subset of the layers in the final neural network. Each added hidden layer is pretrained as part of a shallow supervised MLP, taking as input the output of the previously trained hidden layer.

Supervised pretraining



(a) We start by training a sufficiently shallow architecture. (b) Another drawing of the same architecture. (c) We keep only the input-to-hidden layer of the original network and discard the hidden-to-output layer. We send the output of the first hidden layer as input to another supervised single hidden layer MLP that is trained with the same objective as the first network was, thus adding a second hidden layer. This can be repeated for as many layers as desired. (d) Another drawing of the result, viewed as a feedforward network.

Supervised pretraining

- Instead of pretraining one layer at a time, we may pretrain a deep convolutional network (eleven weight layers) and then use the first four and last three layers from this network to initialize even deeper networks (with up to nineteen layers of weights). The middle layers of the new, very deep network are initialized randomly. The new network is then jointly trained.

- Another option is to use the outputs of the previously trained MLPs, as well as the raw input, as inputs for each added stage.

- Why would greedy supervised pretraining help? The hypothesis is that it helps to provide better guidance to the intermediate levels of a deep hierarchy.

Supervised pretraining

An approach related to supervised pretraining extends the idea to the context of transfer learning:

- Pretrain a deep convolutional net with eight layers of weights on a set of tasks (a subset of the 1,000 ImageNet object categories).

- Then initialize a same-size network with the first k layers of the first net.

- All the layers of the second network (with the upper layers initialized randomly) are then jointly trained to perform a different set of tasks (another subset of the 1,000 ImageNet object categories), with fewer training examples than for the first set of tasks.

Supervised pretraining

- Another related line of work begins by training a network that has low enough depth and great enough width to be easy to train.

- This network then becomes a teacher for a second network, designated the student. The student network is much deeper and thinner (eleven to nineteen layers) and would be difficult to train with SGD under normal circumstances.

- The training of the student network is made easier by training the student network not only to predict the output for the original task, but also to predict the value of the middle layer of the teacher network.

Supervised pretraining

- This extra task provides a set of hints about how the hidden layers should be used and can simplify the optimization problem.

- Additional parameters are introduced to regress the middle layer of the five layer teacher network from the middle layer of the deeper student network. Instead of predicting the final classification target, the objective is to predict the middle hidden layer of the teacher network.

- The lower layers of the student networks thus have two objectives: to help the outputs of the student network accomplish their task, as well as to predict the intermediate layer of the teacher network.

Supervised pretraining

- Although a thin and deep network appears to be more difficult to train than a wide and shallow network, the thin and deep network may generalize better and certainly has lower computational cost if it is thin enough to have far fewer parameters.

- Without the hints on the hidden layer, the student network performs very poorly in the experiments, on both the training and the test set.

Designing models to aid optimization

- In practice, it is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm.

- Most of the advances in neural network learning over the past thirty years have been obtained by changing the model family rather than changing the optimization procedure.

- Stochastic gradient descent with momentum, which was used to train neural networks in the 1980s, remains in use in modern state-of-the-art neural network applications.

Designing models to aid optimization

- Specifically, modern neural networks reflect a design choice to use linear transformations between layers and activation functions that are differentiable almost everywhere, with significant slope in large portions of their domain.

- In particular, model innovations like the LSTM, rectified linear units and maxout units have all moved toward using more linear functions than previous models like deep networks based on sigmoidal units.

- These models have nice properties that make optimization easier. The gradient flows through many layers provided that the Jacobian of the linear transformation has reasonable singular values.

Designing models to aid optimization

- Moreover, linear functions consistently increase in a single direction, so even if the model's output is very far from correct, it is clear simply from computing the gradient which direction its output should move to reduce the loss function.

- In other words, modern neural nets have been designed so that their local gradient information corresponds reasonably well to moving toward a distant solution.

- Other model design strategies can help to make optimization easier. For example, linear paths or skip connections between layers reduce the length of the shortest path from the lower layer's parameters to the output, and thus mitigate the vanishing gradient problem.

Designing models to aid optimization

- A related idea to skip connections is adding extra copies of the output that are attached to the intermediate hidden layers of the network, as in GoogLeNet and deeply supervised nets.

- These “auxiliary heads” are trained to perform the same task as the primary output at the top of the network to ensure that the lower layers receive a large gradient. When training is complete, the auxiliary heads may be discarded.

Continuation methods and curriculum learning

- Many of the challenges in optimization arise from the global structure of the cost function and cannot be resolved merely by making better estimates of local update directions.

- The predominant strategy for overcoming this problem is to attempt to initialize the parameters in a region connected to the solution by a short path through parameter space that local descent can discover.

- Continuation methods are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space.

Continuation methods and curriculum learning

- The idea behind continuation methods is to construct a series of objective functions over the same parameters.

- To minimize a cost function $J(\boldsymbol{\theta})$, we construct new cost functions $\{J^{(0)}, \ldots, J^{(n)}\}$. These cost functions are designed to be increasingly difficult, with $J^{(0)}$ being fairly easy to minimize, and $J^{(n)}$, the most difficult, being $J(\boldsymbol{\theta})$, the true cost function motivating the entire process. When we say that $J^{(i)}$ is easier than $J^{(i+1)}$, we mean that it is well behaved over more of θ space.

- The series of cost functions are designed so that a solution to one is a good initial point of the next. We thus begin by solving an easy problem, then refine the solution to solve incrementally harder problems until we arrive at a solution to the true underlying problem.

Continuation methods and curriculum learning

- Continuation methods traditionally were mostly designed with the goal of overcoming the challenge of local minima. They were designed to reach a global minimum despite the presence of many local minima.

- These continuation methods would construct easier cost functions by "blurring" the original cost function. This blurring operation can be done by approximating

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\theta' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}') \tag{8.40}$$

  via sampling.

Continuation methods and curriculum learning

- The intuition is that some nonconvex functions become approximately convex when blurred. This blurring preserves enough information about the location of a global minimum that we can find the global minimum by solving progressively less-blurred versions of the problem.

This approach can break down in three different ways.

- First, it might successfully define a series of cost functions where the first is convex and the optimum tracks from one function to the next, arriving at the global minimum, but it might require so many incremental cost functions that the cost of the entire procedure remains high. NP-hard optimization problems remain NP-hard, even when continuation methods are applicable.

Continuation methods and curriculum learning

The other two ways continuation methods fail both correspond to the method not being applicable.

- First, the function might not become convex, no matter how much it is blurred. Consider, for example, the function $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\theta}$.

- Second, the function may become convex as a result of blurring, but the minimum of this blurred function may track to a local rather than a global minimum of the original cost function.

Continuation methods and curriculum learning

- Though continuation methods were mostly originally designed to deal with the problem of local minima, local minima are no longer believed to be the primary problem for neural network optimization. Fortunately, continuation methods can still help.

- The easier objective functions introduced by the continuation method can eliminate flat regions, decrease variance in gradient estimates, improve conditioning of the Hessian matrix, or do anything else that will either make local updates easier to compute or improve the correspondence between local update directions and progress toward a global solution.

Continuation methods and curriculum learning

- An approach called curriculum learning, or shaping, can be interpreted as a continuation method.
- Curriculum learning is based on the idea of planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts.
- Curriculum learning is justified as a continuation method, where earlier $J^{(i)}$ are made easier by increasing the influence of simpler examples (either by assigning their contributions to the cost function larger coefficients, or by sampling them more frequently)
- Curriculum learning was also verified as being consistent with the way in which humans teach.

Continuation methods and curriculum learning

- Much better results were obtained with a stochastic curriculum, in which a random mix of easy and difficult examples is always presented to the learner, but where the average proportion of the more difficult examples (here, those with longer-term dependencies) is gradually increased. With a deterministic curriculum, no improvement over the baseline (ordinary training from the full training set) was observed.