第一章:優化算法

隨機梯度下降 (Stochastic Gradient Descent)

隨機梯度下降(SGD)是一種廣泛應用於機器學習和深度學習的優化演算法，其核心在於學習率的設定。學習率是控制模型學習進度的關鍵參數。在SGD中，由於梯度估計器引入了隨機抽樣噪聲(抽樣m個訓練實例)，即使模型接近最優解，這種噪聲也不會完全消失。因此，實踐中通常需要逐漸降低學習率，以確保模型能夠穩定地收斂到最優解。

相比之下，批次梯度下降(BGD)在接近最優解時，由於整體成本函數的真實梯度逐漸減小並趨於零，因此可以使用固定的學習率。在SGD中，學習率通常以 ϵ_k 表示，其中k是迭代次數。為了保證SGD的收斂，在實踐中，通常將學習率線性衰減至某一迭代次數τ，之後保持學習率ϵ為一個固定常數。

選擇合適的學習率通常需要通過試錯法，並通過監測學習曲線來確定。一般而言，τ 可以設置為遍歷訓練集數百次所需的迭代次數，而最終的學習率ϵ_τ通常設置為初始學習率ϵ_0的大約1%。設定合適的初始學習率ϵ_0至關重要，過高可能導致學習曲線劇烈波動，而過低則可能導致學習進展緩慢或陷入高成本值的困境。

在優化演算法的研究中，通常會測量超額誤差 $J(\theta) - min_\theta J(\theta)$，即當前成本函數超出最小可能成本的量。對於complex 問題，SGD的超額誤差在k次迭代後通常為$O(1/\sqrt{k})$，而在strongly complex情況下則為$O(1/k)$。理論上，批次梯度下降比隨機梯度下降擁有更好的收斂速度，但由於克拉美-劳界限(Cramér-Rao bound)指出，泛化誤差不能快於O(1/k)減少，因此追求超過$O(1/k)$的收斂速度可能並不划算。

對於大型數據集，SGD的一個重要優勢是能夠在評估少量樣本的梯度時迅速取得初步進展，這一點的重要性超過了其慢速的漸近收斂。此外，通過在學習過程中逐漸增加小批次大小，可以在批次和隨機梯度下降之間取得平衡。尽管本章其餘部分描述的大多數演算法在實踐中帶來了重要的益處，但這些益處往往被$O(1/k)$漸近分析的常數因子所掩蓋。

超額誤差: 當前成本函數(或損失函數)的值與理論上可達到的最小成本(或損失)之間的差異。簡單來說，它衡量了當前模型與最優模型之間的性能差距。

動量 (Momentum)

在機器學習領域，雖然隨機梯度下降(SGD)是一種廣泛採用的優化策略，但有時候它的學習速度可能會顯得遲緩。為了解決這個問題，動量法被提出，其目的是加快學習過程，特別是在面對高曲率、微小但持續的梯度或者噪聲梯度時更顯得有效。

動量法的核心思想是累積過去梯度的指數衰減移動平均值，並沿這些累積的梯度方向移動。這個過程可以用一個速度變量$\boldsymbol{v}$來描述，它代表參數在參數空間中的移動方向和速度。速度$\boldsymbol{v}$設置為負梯度的指數衰減平均值，這個概念借鑑了物理學中的動量概念——在物理學中，動量定義為質量與速度的乘積。在動量學習算法中，質量被假定為單位質量，因此速度向量v也代表了粒子的動量。

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α

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: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

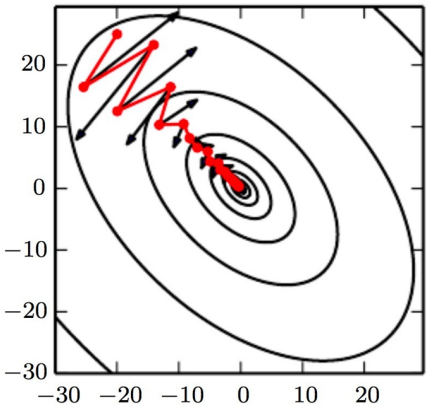
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

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

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

動量算法中的一個重要超參數是α（取值在0到1之間），它決定了先前梯度對當前更新方向的影響程度。在動量算法中，速度$\boldsymbol{v}$會累積梯度元素，而α的大小則決定了先前梯度的影響力。當α相對較大時，先前梯度對當前方向的影響就越顯著。



動量算法的效果可以通過一個紅色路徑來形象化，這條路徑穿過等高線，展示了在最小化特定函數時動量學習規則所遵循的路徑。與傳統的梯度下降方法相比，動量方法能夠更加有效地沿著目標函數的長軸移動，而不是在狹窄的軸上浪費時間。

步長的大小取決於一系列梯度的大小和它們的對齊程度。當許多連續的梯度完全同方向時，步長達到最大。如果動量算法始終觀察到相同的梯度$\boldsymbol{g}$，那麼它會朝著$-\boldsymbol{g}$的方向加速，直到達到一個終端速度，其步長大小為$\frac{\boldsymbol{g}\,||\boldsymbol{g}||}{1-\alpha}$。因此，將動量超參數視為 $\frac{1}{1-\alpha}$ 是有助於理解的。例如，α = 0.9意味著相對於傳統梯度下降算法，最大速度提高了10倍。實踐中常用的α值包括0.5、0.9和0.99，就像學習率一樣，α也可能隨時間進行調整，通常一開始設置較小，後來逐漸增加。

動量算法可以被看作是對連續時間下牛頓力學粒子的模擬。這種物理類比有助於我們建立對動量和梯度下降算法行為的直觀理解。在這個模型中，粒子在任何時刻的位置由$\boldsymbol{\theta}(t)$給出，粒子經歷的淨力為 $\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2}\theta(t)$。

這種力量使粒子加速，可以看作位置的二階微分方程。我們也可以引入一個變量 $\boldsymbol{v}(t) = \frac{\partial}{\partial t}\theta(t)$，表示時間$t$時粒子的速度，並將牛頓力學重寫為一階微分方程。動量算法包括通過數值模擬解決這些微分方程。

解決微分方程的一種簡單數值方法是歐拉方法，它包括按照每個梯度的方向進行小的、有限的步驟來模擬方程定義的動力學。這就是動量更新的基本形式。在動量算法中，作用於粒子的力量之一與成本函數的負梯度成比例：$-\nabla_{\theta} J(\theta)$。這種力量沿著成本函數曲面向下推動粒子。

此外，還需要一種力量與$-v(t)$成比例，對應於粘性阻力，就像粒子必須推動像糖漿那樣的抵抗性介質。這種力量使粒子逐漸失去能量，並最終收斂到局部最小值，從而提高了優化過程的效率和穩定性。

Nesterov動量 (Nesterov Momentum)

受到Nesterov加速梯度法的啟發，動量算法的一個變種被提出，更新規則如下圖所示:

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α

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 labels $\boldsymbol{y}^{(i)}$.

Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

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

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

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

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

Nesterov動量與標準動量的不同之處在於梯度的評估位置。使用Nesterov動量時，梯度是在當前速度應用後進行評估的。因此，可以將Nesterov動量解釋為試圖對標準動量方法添加一個修正因子。在凸批量梯度(convex batch gradient case) 情況下，Nesterov動量將過剩誤差的收斂速率從 $O(1/k)$（經過 k 步之後）提升到 $O(1/k^2)$。但不幸的是，在隨機梯度的情況下，Nesterov動量並不提高收斂速度。

具有適應性學習率的算法 (Algorithms with Adaptive Learning Rates)

在探索機器學習算法的高效設置方面，動量算法的出現被視為一種進步。它在某種程度上減輕了設置學習率的難度。然而，這種改進並非沒有代價——它引入了另一個需要調整的超參數，這也意味著更多的調試和優化工作。那麼，我們是否有其他更好的解決方案呢？

答案可能在於考慮參數的靈敏度方向。如果我們假設這些靈敏度方向在某種程度上與軸對齊，那麼對於每個參數使用獨立的學習率並在學習過程中自動調整這些學習率就顯得合理了。這樣做的好處是能夠更精確地根據每個參數的具體情況來調整學習速度，從而提高整體學習效率和模型性能。

AdaGrad

在機器學習領域，AdaGrad算法以其獨特的方式調整模型參數的學習率，受到了廣泛的關注。這個算法的核心在於個別適應所有模型參數的學習率，這是通過與梯度的歷史平方值之和的

平方根成反比的方式進行縮放實現的。具體來說，對於那些對損失函數影響最大的參數（即偏導數較大的參數），AdaGrad會更快地減少其學習率；而對於偏導數較小的參數，學習率的減少則相對較小。其核心思想是為每個參數提供一個定制化的學習率，這使得它特別適用於處理包含大量零值的數據，如文本數據或圖像數據中的像素。

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

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

Require: Small constant δ, 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

AdaGrad的工作原理如下：

1. 初始化：首先，對每個參數的學習率進行初始化。在AdaGrad中，這通常是一個小的恆定值。
2. 計算梯度：對於每次訓練迭代，算法計算損失函數（該函數衡量模型預測的準確性）相對於模型參數的梯度。梯度是對於每個參數的損失函數的偏導數，指示損失函數在該參數上的增長率。
3. 累積平方梯度：AdaGrad會為每個參數維護一個梯度累積平方和。這意味著對於每個參數，它會將每次迭代的梯度平方加到累積總和中。這個累積總和幫助調整學習率，使得經常更新的特徵（參數）的學習率降低。
4. 調整學習率：在每次迭代中，對於每個參數，AdaGrad使用累積的平方梯度來調整該參數的學習率。具體來說，它會將初始學習率除以平方根（加上一個小的平滑項以避免除以零）的累積梯度。這導致了經常更新的特徵的學習率降低，而不經常更新的特徵的學習率則相對較高。
5. 更新參數：最後，使用調整後的學習率來更新模型參數。這意味著每個參數的更新步長是根據該參數的過去梯度調整的。

這種策略的直接效果是在參數空間中較緩坡度的方向上取得更大的進展。在凸優化的背景下，AdaGrad算法具有一些理想的理論特性，並且能夠快速收斂。然而，在實際應用中，特別是在訓練深度神經網絡模型時，從訓練開始就積累的平方梯度可能導致有效學習率過早和過度地下降。這可能會使得學習率在模型到達局部凸形區域之前就變得過小。為了克服這個問題，後續發展了像RMSprop和Adam這樣的算法，它們在某種程度上可以看作是AdaGrad的改進版本。

因此，雖然AdaGrad算法在某些深度學習模型中表現良好，但它並不適用於所有的深度學習模型。對於特定的應用和模型，仍然需要仔細選擇和調整合適的優化算法，以確保模型的有效訓練和最佳性能。

RMSProp

RMSProp算法是一種在機器學習中廣泛使用的優化方法，尤其是在深度神經網絡（DNNs）的訓練上表現出色。其核心創新在於將梯度累積的方式由傳統方法改變為指數加權移動平均，這一改變使得它在非凸（non-convex）設定中表現更佳。

在傳統的優化方法中，如AdaGrad，梯度累積往往會考慮整個訓練過程的歷史記錄，這在某些情況下可能會導致學習速度變慢。與之相比，RMSProp通過使用指數衰減平均來丟棄過去的遠距離歷史記錄。這意味著RMSProp能夠更快地忘記舊的梯度信息，當它找到一個凸形（convex）碗狀結構時，就可以迅速地收斂，就像是在該碗狀結構內初始化的AdaGrad實例一樣。

RMSProp算法引入了一個新的超參數ϱ，這個超參數用於控制移動平均的長度尺度。這個超參數在實踐中非常重要，因為它決定了算法對過去梯度的記憶長度。選擇合適的ϱ值對於算法能夠高效運行至關重要。一個合適的ϱ值能夠幫助算法在快速適應新梯度的同時，保留足夠的歷史信息以確保平滑的優化過程。

在實證研究中，RMSProp已被證明是一種非常有效且實用的優化算法，尤其是在深度神經網絡的訓練中。這是因為它能夠有效地調節梯度的更新步長，從而加快學習過程，並在面對覆雜的非凸優化問題時表現出更好的穩定性和收斂性。

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

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

Require: Small constant δ, 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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

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

Adam

Adam算法是一種在機器學習領域廣泛使用的自適應學習率優化算法，其名稱“Adam”來源於“adaptive moments”（自適應時刻）。這種算法結合了RMSProp和動量方法的特點，但同時也有一些獨特的區別和創新。

首先，Adam算法中的一個關鍵特點是將動量直接作為梯度的一階時刻的估計，並且采用指數加權的方式。這意味著動量不僅是梯度的函數，而且還考慮了先前梯度的加權平均，從而使得優化過程更加平滑。不同於RMSProp只調整梯度，Adam算法將動量直接加到RMSProp調整過的梯度上，雖然這種結合方式沒有明確的理論動機，但在實踐中表現出色。

其次，Adam算法引入了對一階時刻（動量項）和二階時刻（未中心化）的估計的偏差校正。這個偏差校正是為了彌補算法在初始化階段時對這些時刻估計的影響。這種偏差校正是Adam算法的一個顯著特點，它有助於在訓練初期階段更準確地估計這些動量和規模參數。

雖然RMSProp算法也包含對未中心化二階時刻的估計，但它缺少Adam中的偏差校正機制。因此，在訓練的早期階段，RMSProp的二階時刻估計可能會有較高的偏差，這可能影響到算法的優化效率。

總體來說，Adam算法被認為對超參數的選擇相當穩健，盡管有時需要調整推薦的默認學習率。這種對超參數選擇的魯棒性使得Adam成為了深度學習和其他機器學習應用中非常受歡迎的優化算法。其結合動量和自適應學習率的特點，使得它在處理各種覆雜優化問題時表現出色。

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ 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

選擇合適的優化算法 (Choosing the Right Optimization Algorithm)

目前最受歡迎的優化算法包括: SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam 。選擇使用哪一種演算法，比較大程度來自於使用者對演算法的熟悉程度，以便於調整超參數 (hyperparameter tuning)

參數初始化策略 (Parameter Initialization Strategies)

在機器學習和特別是深度學習領域，算法的初始化選擇對其性能有著深遠的影響。初始點不僅可能決定算法是否會收斂，而且在某些情況下，不穩定的初始點甚至可能導致算法遇到數值困難並徹底失敗。例如，當學習收斂時，初始點可能影響學習的速度，以及算法是否收斂到成本高或低的點。甚至在成本相同的點，不同的初始點可能會導致泛化誤差的巨大差異。

由於神經網絡優化尚未被完全理解，設計出更好的初始化策略是一項挑戰。大多數初始化策略旨在在網絡初始化時實現一些良好性質，但我們對這些性質在學習開始後的保持程度仍有限的理解。進一步的困難在於，某些對優化有利的初始點可能從泛化的角度來看並不理想。

唯一確定的是初始參數需要打破不同單元之間的對稱性。如果具有相同激活函數的隱藏單元連接到相同的輸入並具有相同的初始參數，則任何確定性的成本和學習算法將以相同的方式更新這些單元。因此，即使是能利用隨機性的模型或訓練算法，最好也是初始化每個單元以計算不同的函數，以確保沒有輸入或梯度模式在前向或反向傳播中丟失。

通常，參數的隨機初始化基於讓每個單元計算不同函數的目標。高熵分佈的隨機初始化在計算上更便宜，且不太可能讓任何單元計算與其他單元相同的函數。雖然我們可以使用方法如格拉姆-施密特正交化來尋找互不相同的基礎函數，但這通常會帶來顯著的計算成本。

實踐中，我們通常將每個單元的偏置設置為經驗選擇的常數，並隨機初始化權重。權重通常初始化為高斯或均勻分佈隨機抽取的值，選擇高斯分佈或均勻分佈似乎影響不大。然而，初始分佈的規模對優化過程和網絡的泛化能力有很大影響。較大的初始權重有助於避免冗餘單元和信號丟失，但過大則可能導致值爆炸或激活函數飽和。

這些相互競爭的因素決定了權重的理想初始規模，而優化和正則化的觀點對初始參數的選擇提供了不同的見解。優化觀點建議權重應足夠大以成功傳遞信息，而正則化則傾向於使它們更小。使用像隨機梯度下降這樣的優化算法，通常會導致最終參數接近初始參數，這也表達了一種對初始參數的先驗偏好。

目前有幾種啟發式方法用於權重初始化，每種方法都有其特點和考量。一種常見的方法是從均勻分佈中隨機抽取權重，以初始化具有 m 個輸入和 n 個輸出的全連接層。這種方法簡單且易於實施。另一種方法是標準化初始化，其目的是在所有層的激活方差和梯度方差保持相同，從而取得平衡。這有助於避免激活和梯度在網絡中的消失或爆炸。有些專家建議使用隨機正交矩陣進行初始化，並配合一個根據每層應用的非線性精心選擇的縮放因子 g。這種方法可以訓練非常深的網絡，甚至達到1000層，而不需要正交初始化。

這些方法背後的關鍵洞察是，前饋網絡中的激活和梯度在每一層的前向或反向傳播過程中可能呈隨機漫步行為。如果這種隨機漫步能被適當調整以保持規範，則可以在很大程度上避免梯度消失和爆炸問題。然而，實踐中我們通常需要將權重的規模視為一個超參數，其最佳值與理論預測值大致接近但不完全相等，這可能是因為我們可能使用錯誤的標準，或者初始化時強加的特性在學習開始後不再持續，或者標準成功提高了優化速度但增加了泛化誤差。

使用縮放規則（例如 $1/\sqrt{m}$）設置所有初始權重的一個缺點是，當層變大時，每個單獨的權重變得極其微小。稀疏初始化是一種替代方案，其中每個單元僅初始化為有 k 個非零權重，以保持單元的總輸入量獨立於輸入數量 m。雖然稀疏初始化有助於實現單元間更多的多樣性，但它也強加了一個強烈的先驗，即選擇具有大高斯值的權重。這可能對某些單元造成問題，尤其是那些需要多個過濾器彼此仔細協調的單元，例如 maxout 單元。

當計算資源允許時，將每層權重的初始規模視為超參數並使用超參數搜索算法選擇這些規模是一個好主意。或者，可以通過觀察一批數據的激活或梯度的範圍或標準差來手動搜索最佳初始規模。這個過程可以自動化，通常比基於驗證集錯誤的超參數優化計算成本低。其他參數的初始化通常更容易。例如，將偏置設為零與大多數權重初始化方案兼容，這是一種常見的做法。总体来说，選擇合適的權重初始化策略需要考慮許多因素，包括網絡的深度、激活函數的種類以及特定任務的需求。

在一些情況下，我們可能將一些偏置設定為非零值：

1. 如果偏置是輸出單元，那麼通常初始化偏置以獲得輸出的正確邊際統計是有利的。為此，我們假設初始權重足夠小，以致單元的輸出僅由偏置決定。這種方法的基本假設是，初始權重設置得足夠小，以至於單元的輸出主要由偏置決定。這就導致了一種特定的偏置設置方法，即根據訓練集中輸出的邊際統計來調整偏置。
   例如，如果輸出是類別的分佈，且這個分佈是一個高度偏斜的分佈，類別 i 的邊際概率

由向量 $\boldsymbol{c}$ 的元素 c_i 給出，那麼我們可以通過解方程 softmax($\boldsymbol{b}$) = $\boldsymbol{c}$ 來設置偏置向量 $\boldsymbol{b}$。

這種偏置初始化方法同樣適用於像自編碼器和玻爾茲曼機這樣的模型。這些模型中，有些層的輸出應該與輸入數據 x 類似。因此，初始化這些層的偏置以匹配 x 的邊際分佈可能非常有幫助。這樣的初始化有助於模型更好地學習和重建輸入數據的特性，從而提高整體性能。

2. 有時我們可能希望選擇偏置以避免在初始化時造成過多的飽和。例如，我們可能將 ReLU 隱藏單元的偏置設為 0.1 而不是 0，以避免在初始化時飽和 ReLU。這種方法與不期望來自偏置的強輸入的權重初始化方案不兼容。例如，不建議與隨機漫步初始化一起使用。

3. 有時一個單元會控制其他單元是否能參與某個功能。在這種情況下，我們有一個輸出為 u 的單元和另一個單元 $h \in [0,1]$，它們相乘產生輸出 uh。我們可以將 h 看作是一個決定 $uh \approx u$ 或 $uh \approx 0$ 的閘門。在這些情況下，我們希望設置 h 的偏置，使得在初始化時大多數時間 $h \approx 1$。否則 u 就沒有機會學習。例如，我們可能將 LSTM 模型的遺忘閘的偏置設為 1。

另一種常見的參數類型是方差或精度參數。例如，在進行條件方差估計的線性回歸中，精度參數（如 β）扮演著重要的角色。這些方差或精度參數通常可以安全地初始化為 1。

除了這些簡單的常數或隨機方法來初始化模型參數，我們還可以使用更先進的方法，如利用機器學習本身來初始化模型參數。一個常見的策略是利用在同一輸入上訓練的無監督模型學習的參數來初始化監督模型。這種方法利用了無監督學習模型在學習數據分佈方面的能力，從而為監督學習模型提供了一個更好的起點。

此外，人們還可以通過對相關任務進行監督訓練來初始化模型。值得注意的是，即使是對不相關的任務進行監督訓練，有時也能比隨機初始化提供更快的收斂。這是因為這些初始化策略中的一些可能會在模型的初始參數中編碼了有關數據分佈的信息，從而提供更快的收斂和更好的泛化能力。

其他的策略之所以表現良好，主要是因為它們設定了參數的正確比例，或者設定了不同單元來計算彼此不同的功能。這些方法不僅有助於模型的初始化，也為整體的模型性能和學習過程帶來了積極的影響。

第二章：高階優化方法

近似二階方法 (Approximate Second-order Methods)

討論如何將二階方法應用於深度網絡的訓練。為了使講解更加清晰，我們將討論集中在一個特定的目標函數上，即經驗風險。

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

這種方法在理論上簡潔而強大，是理解深度學習優化的一個重要窗口。這裡討論的方法可以輕易地擴展到更一般的目標函數，例如那些包含參數正則化項的函數。

牛頓法 (Newton's Method)

牛頓法是一種基於二階泰勒級數展開的優化方法，用於近似目標函數 $J(\theta)$ 在某點 $\boldsymbol{\theta}_0$附近的行為，忽略更高階的導數。這個方法涉及使用 J 的海森矩陣 **H**，在 $\boldsymbol{\theta}_0$ 處對 θ 進行評估。

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

如果我們接著求解這個函數的臨界點，我們得到牛頓參數更新規則：

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

這在局部二次函數（具有正定的 **H**）的情況下，透過H^{-1}將梯度重縮放，允許直接跳至最小值。當目標函數是complex的但非二次的（即包含更高階項），這種更新可以透過迭代來進行，從而產生與牛頓法相關的訓練算法。

對於非二次曲面，只要海森矩陣保持正定，就可以迭代應用牛頓法。這種情況下，算法包括兩步迭代過程：首先，更新或計算逆海森矩陣；其次，根據海森矩陣更新參數。

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

然而，在海森矩陣的特徵值不全是正的情況下（如鞍點附近），牛頓法可能導致更新朝錯誤方向移動。這種情況可以透過正則化海森矩陣來避免，例如在海森矩陣對角線上加上一個常數 α，正則化更新變成：

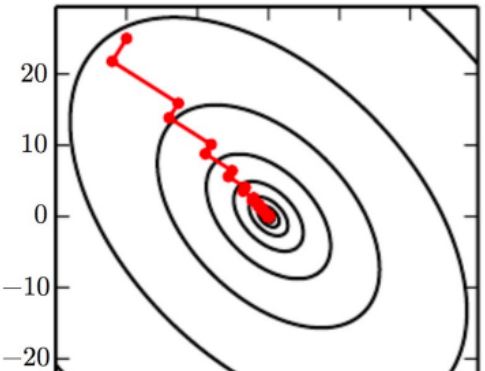
$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0).$$

這種正則化更新用於近似牛頓法的算法，例如Levenberg-Marquardt算法。當有更極端的曲率方向時，α的值必須足夠大以抵消負特徵值。隨著 α 的增大，牛頓法選擇的方向趨於標準梯度除以 α。

在存在強烈負曲率時，α 可能需要非常大，以至於牛頓法的步伐會比合適學習率的梯度下降還要小。此外，牛頓法在訓練大型神經網絡時面臨顯著計算負擔，因為海森矩陣中的元素數量是參數數量的平方。對於有 k 個參數的神經網絡，牛頓法需要求解一個 k × k 矩陣的逆運算，計算複雜度為 $O(k^3)$。由於每次訓練迭代都必須計算逆海森矩陣，因此只有參數數量非常少的網絡才能實際通過牛頓法訓練。

共軛梯度 (Conjugate Gradients)

共軛梯度法是一種高效的優化方法，旨在避免計算逆海森矩陣的需求。它，通過迭代下降共軛方向實現。這種方法的開發靈感來自於對最陡下降法弱點的研究，特別是在該方法中使用線搜索時（評估$f(\boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$的多個$\epsilon$值並選擇結果使目標函數值最小的那個）在梯度相關方向上的反覆應用所帶來的低效率。



最陡下降法中，每一步都沿著當前點的梯度方向跳到最低成本點。這個過程在

某些情況下會導致無效的來回之字形模式，因為每次線搜索的方向由梯度給出，並保證與前一次搜索的方向正交。
然而，這種正交下降的方向並不保留沿著前一次搜索方向的最小值，從而導致了進展的鋸齒形模式。

令前一次的搜索方向為 d_{t-1}，在最小值(線搜索的結束處)，其方向導數為為0 : $J(\theta) \cdot d_{t-1} = 0$。由於這一點的梯度定義了當前搜索方向，$d_t = \nabla_x J(\theta)$ 對 d_{t-1}的方向毫無貢獻，因此 dt 和 dt-1彼此正交。在共軛梯度方法中，我們尋求找到一個與前一次線搜索方向共軛的搜索方向；也就是說，它不會撤銷在那個方向上已取得的進展。

在訓練迭代t時，下一個搜索方向 d_t 的形式為：

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

其中β_t 是一個係數，它控制了我們應該將多少前一方向 d_{t-1} 加回到當前搜索方向中。

如果兩個方向 d_t 和 d_{t-1} 被定義為共軛，這意味著 $d_t^T H d_{t-1} = 0$，其中 H 是海森矩陣。而為了避免直接計算海森矩陣的特徵向量，共軛梯度法采用了如Fletcher-Reeves或Polak-Ribière等方法來計算 β_t。
對於二次函數，共軛方向確保沿著前一個方向的梯度不增加幅度，從而保持在最小值上。因此，在 k 維參數空間中，共軛梯度法最多只需要 k 次線搜索來達到最小值。這使得共軛梯度法成為一種對於大規模問題來說計算上更可行的方法，尤其是在訓練大型神經網絡時，與牛頓法相比，它顯著減少了計算負擔。

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

非線性共軛梯度 (Nonlinear Conjugate Gradients)

在應用共軛梯度算法於非二次目標函數時，我們面臨著一個關鍵的挑戰：共軛方向不再保證在先前方向的目標函數最小值處保持不變。這是因為在非二次函數的情況下，每個新的搜索方向不一定會保留之前方向上取得的進展。為了應對這個問題，非線性共軛梯度算法包括了偶爾的重置機制。在這些重置階段中，算法重新開始，沿著未改變的梯度進行新的線搜索，以避免前一次搜索的影響。

此外，通常在開始非線性共軛梯度優化之前，先使用幾次隨機梯度下降（SGD）來初始化優化過程是有益的。儘管共軛梯度算法傳統上被視為適用於批處理的方法(batch method)，但其小批量版本已被成功應用於神經網絡的訓練中。

BFGS

BFGS 算法結合了牛頓方法的優點，但避免了其計算上的負擔。這種算法與共軛梯度方法相似，但BFGS採用更直接的方式來近似牛頓更新。牛頓的更新公式

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

涉及到海森矩陣H對於$\boldsymbol{\theta}$在 $\boldsymbol{\theta}_0$ 處的評估。牛頓更新的主要計算挑戰是計算逆海森矩陣H^{-1}。

擬牛頓法，尤其是BFGS算法，通過用矩陣 M_t 來近似逆海森矩陣，並透過低秩更新來進行疊代改，以更好地近似H^{-1}。一旦逆海森矩陣近似M_t 更新之後，下降方向 $\boldsymbol{\rho}_t$ 由 $\boldsymbol{\rho}_t = M_t g_t$決定。通過線搜索沿此方向確定步長大小$\epsilon^*$，並根據此更新參數。

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

與共軛梯度方法一樣，BFGS算法在疊代線搜索時也包含了二階信息。不同之處在於，BFGS的成功不那麼依賴於線搜索沿線找到接近真實最小值的點。這意味著相比共軛梯度方法，BFGS可以在每次線搜索上花費更少的時間進行精煉。然而，BFGS算法的一個缺點是需要存儲逆海森矩陣M，這需要$O(n^2)$的記憶體空間。對於通常擁有數百萬參數的現代深度學習模型來說，這使得BFGS算法在實際應用中受限。

有限記憶體 BFGS (L-BFGS)

L-BFGS算法是BFGS算法的一種改進版，它通過避免存儲完整的逆海森矩陣近似$\boldsymbol{M}$，顯著降低了記憶體成本。這種算法假設$\boldsymbol{M}^{(t-1)}$是單位矩陣，並且不會從一步到下一步存儲近似值。這樣，L-BFGS算法可以在更大的模型和更長的時間序列上有效運作，同時保持對記憶體的需求在合理範圍內。

當與精確的線搜索結合使用時，L-BFGS定義的方向是相互共軛的，這增加了其效率。值得注意的是，即使線搜索的最小值僅僅大約達到，這種方法仍然能夠保持良好的性能。這表示L-BFGS算法對線搜索的質量不那麼敏感，這是一個重要的優勢，特別是在實際應用中，完美的線搜索可能難以實現。

L-BFGS進一步通過僅存儲用於每個時間步更新$\boldsymbol{M}$的一些向量來概括，從而包含更多關於海森的信息，而成本僅為$O(n)$。這種方法的高效性使其特別適用於大型機器學習問題，其中記憶體限制和計算資源是主要考慮因素。透過這種方式，L-BFGS為處理大規模優化問題提供了一種實用且高效的解決方案。

第三章:優化輔助技術

專注於輔助深度學習優化的技術和策略，這些技術可以與基本的優化算法結合使用，以提高其效率和有效性。

批量正規化 (Batch Normalization)

批次標準化是一種應對訓練非常深的模型所面臨困難的適應性重參數化方法。在深度學習中，梯度下降算法指出如何在其他層保持不變的情況下更新每個參數。但實際上，所有層的參數都是同時更新的。這可能導致由於多個函數同時改變而出現意外的結果，特別是因為這些更新是基於假設其他函數保持恆定的情況下計算的。

舉例來說，假設有一個每層只有一個單元的深度神經網絡，且不使用激活函數，那麼輸出 y 就是輸入 x 的線性函數，但對權重 ω^i的非線性函數。如果我們的成本函數在 y 上放置梯度 1，意味著希望降低 y，那麼反向傳播算法可以計算出一個梯度 $\boldsymbol{g} = \nabla_{\omega} \hat{y}$。在進行更新 $\omega \leftarrow \omega - \epsilon \boldsymbol{g}$ 時，一階泰勒級數近似預測 y 的值將減少$\epsilon \boldsymbol{g}^T \boldsymbol{g}$。但實際上，更新將包括更高階的效應，甚至高達第 l 階。

更新可能產生一個二/三階項:

$$\hat{y} = x(\omega_1 - \epsilon \boldsymbol{g}_1)(\omega_2 - \epsilon \boldsymbol{g}_2) \ldots (\omega_l - \epsilon \boldsymbol{g}_l)$$

以$\epsilon \boldsymbol{g}_1 \boldsymbol{g}_2 \prod_{i=3}^{l} \omega_i$ 為例。如果 $\prod_{i=3}^{l} \omega_i$ 很小，這個項可能可以忽略；但如果第 3 層到 l 層的權重大於 1，則可能會指數級增大。這使得選擇合適的學習率變得非常困難，因為對一層參數的更新效果極其依賴於所有其他層。

二階優化算法試圖解決這個問題，通過計算考慮到這些二階交互作用的更新。但在非常深的網絡中，甚至更高階的交互作用也可能顯著。這些算法通常非常昂貴，需要許多近似，並且無法真正解決所有顯著的二階交互作用。因此，為 $n > 2$ 建立第 n 階優化算法似乎是無望的。批次標準化因此成為一種有效的替代方法，以緩解在訓練非常深的模型時遇到的這些挑戰。

批次正規化重新參數化幾乎任何深度網絡，從而顯著減少在多層網絡中協調更新的問題。它可以被應用到任何輸入層或隱藏層。

這個過程開始於將要正規化的層的激活值，組成一小批（batch）的設計矩陣，其中每行代表一個實例的激活。為了正規化這個矩陣 $\boldsymbol{H}$，我們使用以下公式進行替換：$\boldsymbol{H}' = (\boldsymbol{H} - \boldsymbol{\mu}) / \boldsymbol{\sigma}$，其中 $\boldsymbol{\mu}$ 是每個單元的平均值向量，$\boldsymbol{\sigma}$ 是每個單元的標準差向量。這涉及到將向量 $\boldsymbol{\mu}$ 和 $\boldsymbol{\sigma}$ 廣播到矩陣 $\boldsymbol{H}$ 的每一行，並在每一行內逐元素進行計算。因此，例如，矩陣中的元素 $\boldsymbol{H}$i,4 會被正規化，通過減去 $\boldsymbol{\mu}$4 並除以 $\boldsymbol{\sigma}$4。

在訓練時，為了避免遇到未定義的梯度，使用了略微修改的公式 $\boldsymbol{H}' = (\boldsymbol{H} - \boldsymbol{\mu}) / (\boldsymbol{\sigma} + \delta)$，其中 δ 是一個很小的正數，比如 10^-6。關鍵的一步是將計算平均值和標準差的過程，以及將它們應用於正規化 $\boldsymbol{H}$ 的操作，納入反向傳播過程中。

這意味著梯度永遠不會提出僅僅增加單元的標準差或平均值的操作；正規化操作消除了這種行動的影響，並在梯度中將其分量歸零。這是批次正規化的一個關鍵創新。

在此之前，一些方法是在成本函數中增加懲罰項，以鼓勵單元具有正規化的激活統計，但這經常導致不完美的正規化。另一種方法是在每次梯度下降步驟後介入以重新正規化單元統計，

這通常會導致顯著的時間浪費，因為學習算法不斷嘗試改變平均值和方差，而正規化步驟又不斷撤銷這些變化。

批次正規化通過重新參數化模型，使得某些單元始終按定義進行標準化，巧妙地規避了這些問題。

Polyak平均 (Polyak Averaging)

波拉克平均法（Polyak averaging）是一種在優化過程中應用的技術，它涉及對梯度下降算法訪問的參數空間軌跡中的多個點進行平均。具體來說，如果梯度下降在其迭代過程中訪問了點$\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, ..., \boldsymbol{\theta}^{(t)}$，則波拉克平均的輸出是這些點的算數平均值，即$\widehat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t} \sum_i \boldsymbol{\theta}^{(i)}$。這種方法對於凸問題特別有效，因為在這種情況下，它具有強大的收斂保證。

當應用於神經網絡這樣的非凸問題時，波拉克平均法的思想基於一個觀察：優化算法可能在解空間的山谷兩側反复跳躍，而從未真正接近山谷底部。在這種情況下，對所有訪問過的位置進行平均，理論上應該能夠得到更接近山谷底部的點。

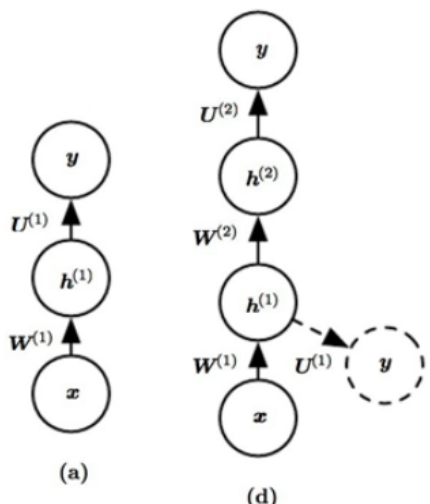
然而，在非凸問題中，優化軌跡可能非常複雜，涉及訪問許多不同的區域。在這些情況下，納入來自遠期過去的參數點（這些點可能與當前點由成本函數中的大障礙分隔開）並不總是有用的。因此，當將波拉克平均法應用於非凸問題時，一個常見的做法是使用指數衰減的滾動平均，這種方法更多地考慮近期的參數點，從而更能反映當前優化過程的狀態。

監督預訓練 (Supervised Pretraining)

在面對模型複雜且難以優化的情況，或當任務本身非常困難時，直接訓練一個模型來解決特定任務可能過於雄心勃勃。在這些情況下，採用更加逐步和漸進的方法可能更為有效。這包括先訓練一個較簡單的模型來解決任務，然後逐步增加模型的複雜性，或者先訓練模型解決一個較簡單的任務，再轉向面對最終任務。這些方法統稱為預訓練。

貪婪算法是這種思想的一個例子，它將問題分解成多個組件，然後單獨解決每個組件的最佳版本。雖然這些單獨的最佳組件組合起來可能不會產生完整解決方案的最佳結果，但貪婪算法在計算上往往更為簡便，且其結果通常是可以接受的，即使不是最佳的。

貪婪監督預訓練是這種方法的一個應用，它將監督學習問題分解為更簡單的問題。在這種方法的原始版本中，每個階段都涉及訓練深度神經網絡中的一部分層，從而逐步增加隱藏層。例如，可以先訓練一個淺層架構(a)，然後逐步添加新的隱藏層，並以前一層的輸出作為輸入(b)。



這種方法的另一個選擇是一次預訓練多層，而不是逐層進行。例如，可以預訓練一個深層卷積網絡的部分層，然後使用這些層作為更深網絡的一部分，並對新網絡進行聯合訓練。

貪婪監督預訓練有助於為深層次結構的中間層提供更好的指導。這種方法也可以擴展到轉移學習的背景下，如先

在一組任務上預訓練一個網絡，然後使用這個網絡的一部分層來初始化執行不同任務集的另一個網絡。

另一個相關的研究路徑是首先訓練一個較淺且較寬的網絡，然後讓這個網絡成為另一個更深更窄網絡的“教師”。這種“學生”網絡通過預測“教師”網絡中間層的值來進行訓練，這樣的額外任務有助於簡化優化問題。

儘管更深更窄的網絡似乎比較淺的網絡更難訓練，但它們可能具有更好的泛化能力，並且如果足夠窄，則具有更低的計算成本。這種方法在實驗中證明，如果沒有“教師”網絡的中間層提示，“學生”網絡在訓練集和測試集上的表現都非常差。

設計有利於優化的模型 (Designing Models to Aid Optimization)

在過去三十年的神經網絡發展中，模型家族的選擇顯得比優化算法更為關鍵。自1980年代以來，隨機梯度下降法配合動量一直是訓練神經網絡的首選方法，並在當今先進的神經網絡應用中依然占主導地位。現代神經網絡的設計體現了某些重要的選擇，例如在各層之間使用線性變換以及幾乎在所有地方可微分的激活函數。這些函數大多在其作用範圍內具有明顯的斜率，如長短期記憶（LSTM）、修正線性單元和最大池化單元等創新模型所展現的趨向。這些模型易於優化，主要是因為它們的線性變換能夠有效地傳遞梯度。

此外，線性函數的持續單向增長特性意味著，即使模型輸出與正確值相去甚遠，也能從梯度計算中清晰地識別出損失函數減少的方向。這就是為何現代神經網絡在設計上能夠使局部梯度信息與尋找全局解的方向高度一致。除此之外，模型設計的其他策略，如層間的線性路徑或跳過連接，也有助於簡化優化過程。這些策略減少了從底層參數到輸出的最短路徑長度，從而緩解了梯度消失的問題。GoogLeNet和深度監督網絡等模型通過在中間隱藏層添加額外的輸出副本，即所謂的“輔助頭部”，來確保底層接收到充足的梯度。這些輔助頭部在訓練完成後通常會被移除。

連續方法和課程學習 (Continuation Methods and Curriculum Learning)

優化過程中許多挑戰源自於成本函數的全局結構，這些挑戰無法僅靠估計局部更新方向來解決。為此，主要策略之一是將參數初始化在一個區域，該區域通過參數空間的短路徑與解決方案相連，使得本地下降可以發現這條路徑。

延續方法是一系列策略，目的是選擇初始點以確保局部優化大部分時間在空間中的良好行為區域進行。這方法的核心思想是在相同參數上構建一系列目標函數。為了最小化成本函數 $J^{(\boldsymbol{\theta})}$，會構建一系列新的成本函數 $\{J^{(0)}, \dots, J^{(n)}\}$，可，這些函數設計得越來越難，其中 $J^{(0)}$ 相對容易最小化，而$J^{(n)}$ 是最難的。這些函數被設計成一個的解決方案是下一個的良好初始點，從而逐步精煉解決方案以解決越來越難的問題。

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

延續方法的另一個特點是通過“模糊”原始成本函數來構造更簡單的成本函數，這種模糊操作可以通過抽樣進行近似。當模糊時，某些非凸函數可能變得近似凸，這種模糊保留了有關全局最小值位置的足夠信息，從而能夠透過解決逐漸清晰版本的問題來找到全局最小值。

然而，延續方法也有可能失敗。它可能需要如此多的增量成本函數，以至於整個過程的成本仍然很高。或者在某些情況下，函數可能不會因模糊而變得凸，或者模糊函數的最小值可能跟蹤到原始成本函數的局部最小值，而不是全局最小值。

儘管延續方法最初是為了處理局部最小值問題而設計的，但它也能幫助消除平坦區域、減少梯度估計的變異性、改善黑森矩陣的條件，或進行任何其他能使局部更新更容易計算或改善局部更新方向與全局解決方案進展之間的對應。

課程學習或塑形可以被解釋為一種延續方法。它基於先學習簡單概念，然後逐步進展到更複雜概念的想法。在DL Fall '23中，使用隨機課程獲得了更好的結果，這些課程始終向學習者呈現簡單和困難例子的隨機混合，但其中較困難例子的比例逐漸增加。与此相反，使用確定性課程時，未觀察到對基線的改進。