Introduction to Machine Learning

Practical Aspects of Training Deep Neural Networks

Prof. Subrahmanya Swamy Peruru
Department of Electrical Engineering
IIT Kanpur

Gradient Descent

$$\boldsymbol{w}_{opt} = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w})$$

- Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
- For iteration $t = 0,1,2, \ldots$ (or until convergence)
  - Calculate the gradient $\boldsymbol{g}^{(t)} = \nabla_{\boldsymbol{w}} L(\boldsymbol{w}^{(t)})$
  - Set the learning rate η_t
  - Update parameters in the opposite direction of gradient

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

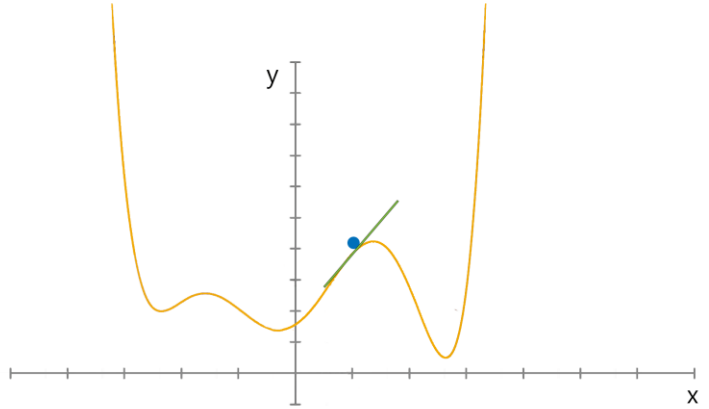


Fig Src: https://telefonicatech.uk/blog/introduction-to-artificial-neural-networks-part-two-gradient-descent-backpropagation-supervised-unsupervised-learning/

The lecture slides are adapted from Prof. Piyush Rai

Stochastic Gradient Descent (SGD)

- Consider a loss function of the form $L(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w})$

- The gradient in this case can be written as

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} L(w) = \nabla_{\boldsymbol{w}}[\frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w})] = \frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{w}}\, \ell_n(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{g}_n$$

Expensive to compute – requires doing it for all the training examples in each iteration ☹

- Stochastic Gradient Descent (SGD) approximates $\boldsymbol{g}$ using a single training example

- At iter. t, pick an index $i \in \{1,2,\ldots,N\}$ uniformly randomly and approximate $\boldsymbol{g}$ as

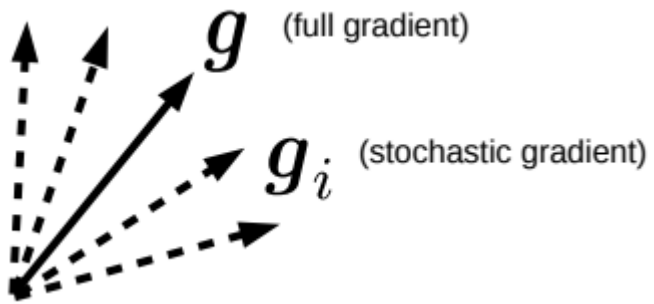
$$\boldsymbol{g} \approx \boldsymbol{g}_i = \nabla_{\boldsymbol{w}} \ell_i(\boldsymbol{w})$$

Can show that $\boldsymbol{g}_i$ is an unbiased estimate of g, i.e., $\mathbb{E}[\boldsymbol{g}_i] = \boldsymbol{g}$

- May take more iterations than GD to converge but each iteration is much faster ☺
  - SGD per iter cost is $O(D)$ whereas GD per iter cost is $O(ND)$

Minibatch SGD

- Gradient approximation using a single training example may be noisy



The approximation may have a high variance – may slow down convergence, updates may be unstable, and may even give sub-optimal solutions (e.g., local minima where GD might have given global minima)

- We can use $B > 1$ unif. rand. chosen train. ex. with indices $\{i_1, i_2, \dots, i_B\} \in \{1,2, \dots, N\}$
- Using this "minibatch" of examples, we can compute a minibatch gradient

$$\boldsymbol{g} \approx \frac{1}{B} \sum_{b=1}^{B} \boldsymbol{g}_{i_b}$$

- Averaging helps in reducing the variance in the stochastic gradient
- Time complexity is $O(BD)$ per iteration in this case

Training with Mini-batch SGD

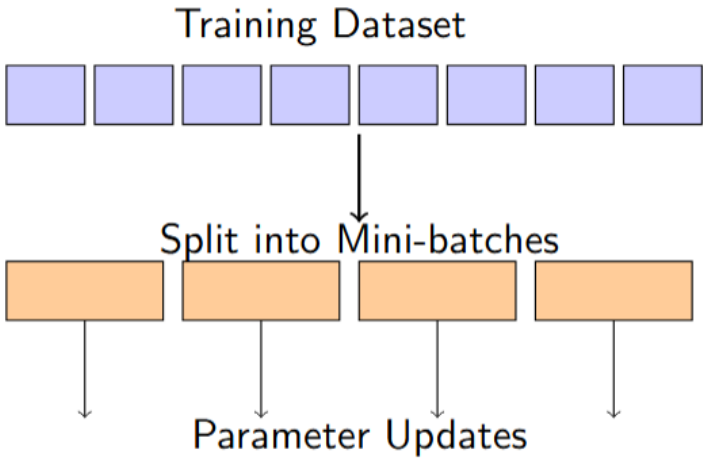
Training Procedure

1. Initialize parameters $\boldsymbol{w}$
2. For epoch:
   1. Randomly shuffle the training dataset
   2. Split the dataset into mini-batches of size B
   3. For each mini-batch $b = 1, 2 \ldots \frac{N}{B}$
      - Compute gradient on the b^{th} mini-batch $\mathcal{B}_b$

$$\boldsymbol{g} = \frac{1}{B} \sum_{i \in \mathcal{B}_b} \nabla_{\boldsymbol{w}} \ell_i(\boldsymbol{w})$$

      - Update parameters

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\, \boldsymbol{g}$$



One pass through all batches $=$ **1 Epoch**

If dataset size = N and batch size = B,

$$\text{Updates per epoch} = \frac{N}{B}$$

Example: $N = 8, B = 2$

Note: Shuffling ensures batches are different in every epoch

Adaptive Gradient Methods: AdaGrad

Instead of using a common learning rate for all dimensions $d = 1, 2, \ldots, D$

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t \, g_d^{(t)}$$

Use a separate learning rate for each parameter

$$w_d^{(t+1)} = w_d^{(t)} - e_d^{(t)} \, g_d^{(t)}$$

AdaGrad:

If some dimension had big updates (indicated by large gradient values in the past), slow down along those directions by using smaller learning rates.

$$e_d^{(t)} = \frac{1}{\sqrt{\epsilon + \sum_{\tau=1}^{t} \left(g_d^{(\tau)} \right)^2}}$$

Adaptive Gradient Methods: Momentum

Can use a momentum term to stabilize gradients by reusing info from past grads

- Move faster along directions where gradients have been consistent
- Slow down along directions where gradient has changed abruptly

The "momentum" term. Set to 0 at initialization

β usually set as 0.9

$$\boldsymbol{m}^{(t)} = \beta \boldsymbol{m}^{(t-1)} + \boldsymbol{g}^{(t)}$$

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{m}^{(t)}$$

Some advanced methods combine momentum and AdaGrad type techniques

- RMS-Prop
- Adam

Initialization of parameters

Why Initialization Matters: Training with gradient descent is sensitive to the starting weights w.

Poor initialization can lead to:

- **Vanishing gradients**: gradients become extremely small
- **Exploding gradients**: gradients grow very large
- **Symmetry problem**: neurons learn identical features

Common Initialization Schemes

Xavier Initialization (Glorot) – for sigmoid / tanh

$$w_{ij} \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$$

He Initialization – for ReLU networks

$$w_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

n_{in}: number of input connections (fan-in)

Batch Normalization

- Standardize = activation $h_{nk}^{(\ell)}$ should have zero mean and unit variance across all n
- It is achieved by inserting a "batch normalization" layer after each hidden layer
- To do so, during training, (omitting layer number ℓ) we replace each $\boldsymbol{h}_n$ by $\widetilde{\boldsymbol{h}}_n$

$$\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} \boldsymbol{h} \qquad \boldsymbol{\sigma}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} (\boldsymbol{h}-\boldsymbol{\mu}_{\mathcal{B}})^2 \qquad \widehat{\boldsymbol{h}}_n = \frac{\boldsymbol{h}_n - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}}$$

$\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are trainable batch-norm parameters

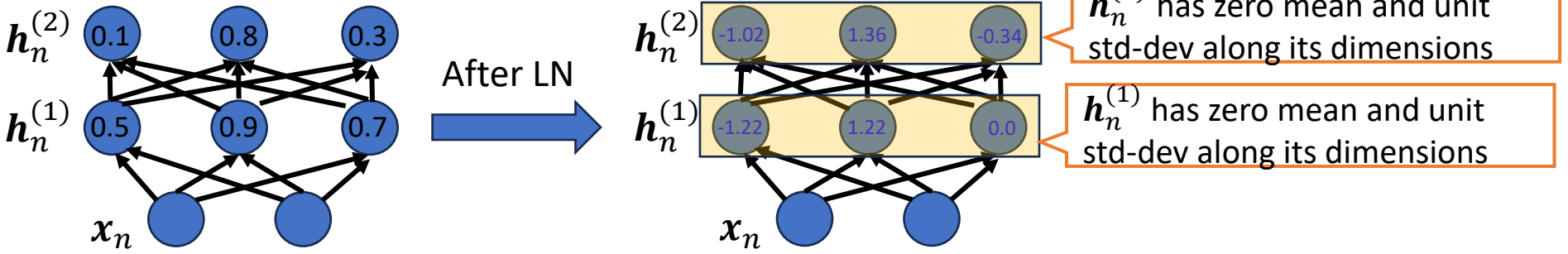
$$\widetilde{\boldsymbol{h}}_n = \boldsymbol{\gamma} \odot \widehat{\boldsymbol{h}}_n + \boldsymbol{\beta}$$

BatchNorm normalizes the signal to stabilize training, but the network is then allowed to learn the best scale and shift again to ensure flexibility

Layer Normalization

- Unlike batch normalization (BN), which we already saw, layer normalization (LN) normalizes each $\boldsymbol{h}_n$ across its dimensions (not across all minibatch examples)
  - Often used for sequence data models where BN is difficult to apply
  - Also useful when batch sizes are small where BN statistics (mean/var) aren't reliable

- For an MLP, the LN operation would look like this



Thank you