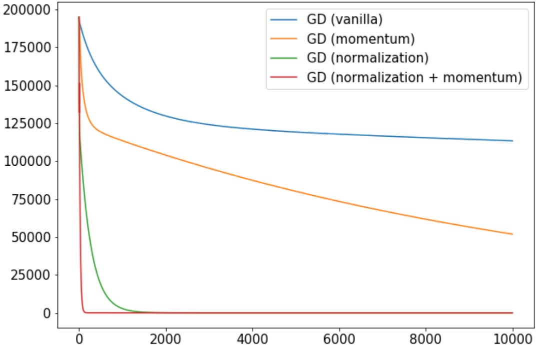
Normalization Tricks

Nan Ye

School of Mathematics and Physics
The University of Queensland

Recall: Normalized Input is Good

OLS on the diabetes dataset



For deep neural nets, can we keep inputs to non-input layers normalized?

Internal Covariate Shift

- Consider a network computing $F_2(F_1(\mathbf{x}, \mathbf{w}_1), \mathbf{w}_2)$.
- Let $\mathbf{v}_i = F_1(\mathbf{x}_i, \mathbf{w}_1)$. Then a gradient update step has the form

$$\mathbf{w}_1 \leftarrow \mathbf{w}_1 - \frac{\eta}{n} \sum_{i=1}^{n} \frac{\partial F_2(F_1(\mathbf{x}_i, \mathbf{w}_1), \mathbf{w}_2)}{\partial \mathbf{w}_1},$$

$$\mathbf{w}_2 \leftarrow \mathbf{w}_2 - \frac{\eta}{n} \sum_{i=1}^{n} \frac{\partial F_2(\mathbf{v}_i, \mathbf{w}_2)}{\partial \mathbf{w}_2}.$$

- After we normalize $\mathbf{x}$, it remain normalized during training.
- The distribution of $\mathbf{v} = F_1(\mathbf{x}, \mathbf{w}_1)$ changes during training as $\mathbf{w}_1$ will be updated.

- In general, the distribution of each layer's inputs changes during training, and they are not normalized — this is known as internal covariate shift.

  covariate is just a different name for an input variable

- Normalized input data $!=$ normalized input for hidden neurons.

Weight-dependent Normalization

- For hidden neurons, we calculate normalization parameters that depend on current weights.
- Specifically, if a neuron takes x as an input, we normalize x to

  $$\hat{x} = \frac{x - \mu}{\sigma},$$

  where μ and σ are calculated based on both the dataset and *current weights*.
- There are various ways to implement this idea.

Naive Idea

- At the beginning of each epoch in SGD, for each input x of a neuron, we first compute its mean μ and variance σ^2 over the entire training set.
- During the epoch, instead of using the input x, the neuron receives the normalized input

$$\hat{x} = \frac{x - \mu}{\sigma}.$$

- Both μ and σ depend on the original network parameters and are involved in gradient computation.

Two problems

- (Loss of representation power) The function represented by the network is different from that represented by the original network, and it may make a layer less capable of representing complex functions.
- (High computational cost) It is computationally impractical, because μ and σ are used in gradient computation, and they need to be represented by an extremely large computational graph.

Batch Normalization (BN)

- It is an improved version of the naive idea.
- It improves gradient flow by reducing dependence of gradients on the scale and initial values of parameters, thus allowing larger learning rate and faster convergence.
- It makes training with saturating units like sigmoid/tanh easier.

Ioffe and Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015

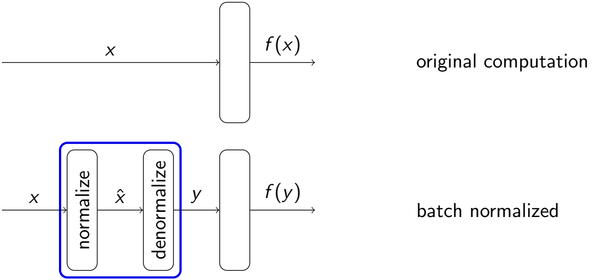
Two improvements

- (Dealing with computational cost) Compute mean and variance using the mini-batch
  - This makes it possible to efficiently compute gradients.
- (Maintaining representation power) Instead of using the normalized input $\hat{x}$ as the input, use a linearly transformed $\hat{x}$

$$y = \gamma \hat{x} + \beta.$$

  - γ and β are parameters to be learned.
  - With $\gamma = \sigma$ and $\beta = \mu$, we recover the identity mapping.

- These two operations can be represented as an additional layer in the network (the batch normalization layer).



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch normalization transform

$$
\begin{aligned}
\frac{\partial \ell}{\partial \widehat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\
\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \\
\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_{\mathcal{B}})}{m} \\
\frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\
\frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i \\
\frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}
\end{aligned}
$$

Batch normalization gradients

Implicit regularization

- During training, the μ and σ values for a BN layer are computed using examples for the current mini-batch.
- The network's predicted output on an example thus depends on the entire mini-batch, not just the example alone — this is different from the usual neural nets that we have seen so far.
- BN thus acts as a form of regularization as a consequence of producing nondeterministic outputs for a given example during training.

Using BN during testing

- At test time the BN layer behaves differently.
- μ and σ^2 are not computed based on current test mini-batch, but are computed using the estimates from multiple training mini-batches.
- Overall, during test time, the batch normalization layer takes in x and outputs

$$y = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} x + \left(\beta - \frac{\gamma \mu}{\sqrt{\sigma^2 + \epsilon}} \right).$$

Where to apply BN

- Usually inserted after fully connected or convolutional layers, and before nonlinearity.
- Specifically, if a layer computes $z = g(Wu + b)$ with g being the activation function,

$$\begin{aligned} \text{recommended:} \qquad & z = g(BN(Wu)), \\ \text{not recommended:} \qquad & z = g(W\,BN(u) + b). \end{aligned}$$

  Note that $BN(Wu + b) = BN(Wu)$.

- $Wu + b$ is more likely to be a symmetric, non-sparse distribution, thus $BN(Wu)$ is more likely to be normally distributed.

```python
# original two-layer network
net = nn.Sequential(nn.Linear(100, 50), nn.ReLU(),
   nn.Linear(50, 10), nn.Sigmoid())

# recommended: batch normalization applied after linearity
net = nn.Sequential(nn.Linear(100, 50), nn.ReLU(),
   nn.Linear(50, 10), nn.BatchNorm1d(10), nn.Sigmoid())

# not recommended: batch normalization applied before linearity
net = nn.Sequential(nn.Linear(100, 50), nn.ReLU(),
   nn.BatchNorm1d(50), nn.Linear(50, 10), nn.Sigmoid())
```

Your Turn

Which of the following statement is correct? (Multiple choice)

(a) When we normalize the input data, the inputs to all hidden neurons will be normalized too.

(b) A neural net with BN layers can produce different predictions on the same training example during testing.

(c) A BN layer first normalizes the input, and then applies a linear transformation to the normalized input.

Layer Normalization

- In batch normalization, the normalization parameters are computed for individual neurons over different examples.
- In layer normalization, the normalization parameters are computed for all neurons in the same layer.
- Specifically, consider the l-th layer of a network, which has m neurons, with $x_{l,1:m}$ as their their weighted input sums.

$$
\begin{aligned}
&\text{(compute parameters)} \quad && \mu_l = \frac{1}{m}\sum_{i=1}^{m} x_{l,i}, \qquad && \sigma_l = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_{li} - \mu_l)^2}, \\
&\text{(normalization)} \quad && \hat{x}_{li} = \frac{x_{li} - \mu_l}{\sigma_l}, \qquad && \text{for each } i = 1, \ldots, m, \\
&\text{(denormalization)} \quad && y_{li} = \gamma_{li}\hat{x}_{li} + \beta_{li}, \qquad && \text{for each } i = 1, \ldots, m.
\end{aligned}
$$

Ba, Kiros, and Hinton, Layer normalization, 2016

Weight Normalization

- Weight normalization reparametrize a weight vector as the product of a magnitude and a directional vector.
- Specifically, $x = \mathbf{w}^\top \mathbf{u} + b$ is reparametrized as

$$x = \gamma \frac{\mathbf{w}^\top}{\|\mathbf{w}\|} \mathbf{u} + b,$$

  where γ is an additional learnable parameter representing the magnitude of the weight vector.
- Weight normalization is also a special case of the generic weight-dependent normalization scheme.
- Specifically, if $x = \mathbf{w}^\top \mathbf{u} + b$, then the following normalization procedure computes $\gamma \frac{\mathbf{w}^\top}{\|\mathbf{w}\|} \mathbf{u} + b$,

$$\begin{aligned}
\text{(compute parameters)} \qquad & \mu = b \qquad \sigma = ||\mathbf{w}||_2, \\
\text{(normalization)} \qquad & \hat{x} = \frac{x - \mu}{\sigma}, \\
\text{(denormalization)} \qquad & y = \gamma \hat{x} + b.
\end{aligned}$$

Salimans and Kingma, Weight normalization: A simple reparameterization to accelerate training of deep neural networks, 2016

What You Need to Know

- Weight-dependent normalization tricks for non-input layers
  - normalization with weight-dependent parameters, followed by denormalization
  - often accelerate training process
- Batch normalization
  - neuron-specific and mini-batch-specific normalization parameters
  - acts as an implicit regularizer during training
- Layer normalization
- Weight normalization