Neural Networks: Training Algorithms

Instructor: Lei Wu [1]

Mathematical Introduction to Machine Learning

Peking University, Fall 2025

[1]School of Mathematical Sciences and Center for Machine Learning Research

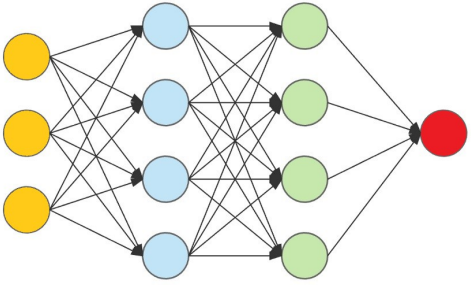
Table of Contents

The forward propagation

- Consider a general network defined by the **forward propagation**:

$$
\begin{aligned}
x^0 &= x \\
x^\ell &= h(x^{\ell-1}; \theta^\ell) \text{ for } \ell = 1, 2, \ldots, L, \\
f(x; \Theta) &= x^L,
\end{aligned}
$$

where θ^ℓ denote the ℓ-th layer's parameters and Θ denotes all parameters.



The problem for computing gradients

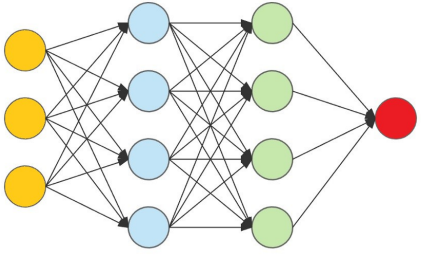
- WLOG, assuming we have only one pair of data (x, y), the loss is given by

$$E(\Theta) = l(f(x; \Theta), y).$$

- **Task:** Computing the gradients

$$\left\{ \frac{\partial E}{\partial \theta^1}, \frac{\partial E}{\partial \theta^2}, \cdots, \frac{\partial E}{\partial \theta^L} \right\}$$

- **Why is this problem not trivial?**



The back-propagation algorithm

- By the chain rule,

$$\frac{\partial E}{\partial \theta^\ell} = \frac{\partial x^\ell}{\partial \theta^\ell}\frac{\partial E}{\partial x^\ell} = \frac{\partial h(x^{\ell-1};\theta^\ell)}{\partial \theta^\ell}\frac{\partial E}{\partial x^\ell}.$$

Define ℓ-th layer's gradient signal as $\delta^\ell := \frac{\partial E}{\partial x^\ell}$. Then, it can be recursively computed via the chain rule:

$$\delta_{\ell-1} = \frac{\partial x^\ell}{\partial x^{\ell-1}}\frac{\partial E}{\partial x^\ell} = \frac{\partial h(x^{\ell-1};\theta^\ell)}{\partial x^{\ell-1}}\delta_\ell$$

$$\delta_L = \frac{\partial l(y', y)}{\partial y'}\Big|_{y'=x^L}.$$

The back-propagation algorithm

- By the chain rule,

$$\frac{\partial E}{\partial \theta^\ell} = \frac{\partial x^\ell}{\partial \theta^\ell}\frac{\partial E}{\partial x^\ell} = \frac{\partial h(x^{\ell-1};\theta^\ell)}{\partial \theta^\ell}\frac{\partial E}{\partial x^\ell}.$$

  Define ℓ-th layer's gradient signal as $\delta^\ell := \frac{\partial E}{\partial x^\ell}$. Then, it can be recursively computed via the chain rule:

$$\delta_{\ell-1} = \frac{\partial x^\ell}{\partial x^{\ell-1}}\frac{\partial E}{\partial x^\ell} = \frac{\partial h(x^{\ell-1};\theta^\ell)}{\partial x^{\ell-1}}\delta_\ell$$
$$\delta_L = \frac{\partial l(y', y)}{\partial y'}\Big|_{y'=x^L}.$$

- **Note:** Compute the red parts need to access the hidden states $\{x^\ell\}_{\ell=0}^L$, which are computed **during the forward propagation**. Keep in mind that storing these hidden states requires significant memory. In particular, the memory required scales linearly with the batch size.

A visualization of back-prop algorithm

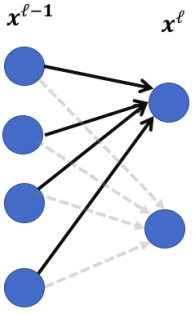
When $h(z;\theta^\ell) = A^\ell\sigma(z) + b^\ell$, we have $\frac{\partial E}{\partial b^\ell} = \frac{\partial E}{\partial x^\ell}$ and $\delta^\ell = \frac{\partial E}{\partial A^\ell}$ satisfies

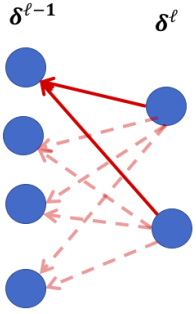
Forward Propagation

$$x^0 = x$$
$$x^\ell = A^\ell\sigma(x^{\ell-1}) + b^\ell$$

Back Propagation

$$\delta^L = l'(f, y)$$
$$\delta^{\ell-1} = \sigma'(x^{\ell-1}) \odot (A^\ell)^\top \delta^\ell$$





Computational and memory cost analysis

Backprop algorithm is a **smart** way to implement the chain rule.
Consider a network of depth L, width m, and the batch size B.

The **computational cost** is $O(Bm^2L)$.

- Reducing the dependence on B and m is not difficult via parallelization. **GPUs are great!! Nvidia Tesla A100 has 6912 cores! RTX 4090 has 16384 cores!**

Figure 1: (Left): 4090; (Right) A100.

- **Reducing the dependence on L is challenging** as the computation is essentially serial when do forward and backward propagation.

Computational and memory cost analysis

Backprop algorithm is a **smart** way to implement the chain rule.
Consider a network of depth L, width m, and the batch size B.

The **memory cost** is $O(BmL + m^2L)$. The blue part is due to we need to store the hidden state for computing gradient.

- Big memory is necessary for training large models. **A100 has 80G memory while RTX 4090 has only 24G.**

Computational and memory cost analysis

Backprop algorithm is a **smart** way to implement the chain rule.
Consider a network of depth L, width m, and the batch size B.

The **memory cost** is $O(BmL + m^2L)$. The blue part is due to we need to store the hidden state for computing gradient.

- Big memory is necessary for training large models. **A100 has 80G memory while RTX 4090 has only 24G.**

For training large models, we can

- Buy A100 and H100 if you are rich (aka "money is all you need")!
- Reduce the batch size or gradient accumulation.
- Design memory-efficient optimizers?

Gradient vanishing and how to mitigate it

Gradient vanishing and exploding

- **Gradient vanishing:**

$$\delta^{\ell} = [\sigma'(x^{\ell}) \odot (A^{\ell+1})^T][\sigma'(x^{\ell+1}) \odot (A^{\ell+2})^T] \cdots [\sigma'(x^{L-1}) \odot (A^{L})^T \delta^{L}]$$

The value is approximately the multiplication of $L - l$ term. If $\sigma'(z^{\ell}) < 1$ or $\|A^{\ell}\|_2 < 1$, then δ^{ℓ} will be exponentially small.

Gradient vanishing and exploding

- **Gradient vanishing:**

$$\delta^\ell = [\sigma'(x^\ell) \odot (A^{\ell+1})^T][\sigma'(x^{\ell+1}) \odot (A^{\ell+2})^T] \cdots [\sigma'(x^{L-1}) \odot (A^L)^T \delta^L]$$

  The value is approximately the multiplication of $L - l$ term. If $\sigma'(z^\ell) < 1$ or $\|A^\ell\|_2 < 1$, then δ^ℓ will be exponentially small.

- Roughly, $\delta^\ell \approx (\sigma'(x)\|A\|_2)^{L-\ell}$. This implies that **deep networks are harder to train than shallow networks**.

Gradient vanishing and exploding

- **Gradient vanishing:**

$$\delta^\ell = [\sigma'(x^\ell) \odot (A^{\ell+1})^T][\sigma'(x^{\ell+1}) \odot (A^{\ell+2})^T] \cdots [\sigma'(x^{L-1}) \odot (A^L)^T \delta^L]$$

  The value is approximately the multiplication of $L - l$ term. If $\sigma'(z^\ell) < 1$ or $\|A^\ell\|_2 < 1$, then δ^ℓ will be exponentially small.

- Roughly, $\delta^\ell \approx (\sigma'(x)\|A\|_2)^{L-\ell}$. This implies that **deep networks are harder to train than shallow networks**.

- More precisely, it is due to the **disparity of gradient scales across different layers** that makes it challenging to select a single learning rate that works effectively for all layers simultaneously. Recall the picture of defining **condition number**!!

Gradient vanishing and exploding

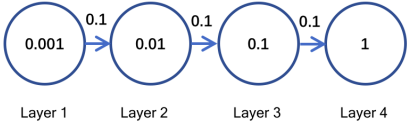
- **Gradient vanishing:**

$$\delta^\ell = [\sigma'(x^\ell) \odot (A^{\ell+1})^T][\sigma'(x^{\ell+1}) \odot (A^{\ell+2})^T] \cdots [\sigma'(x^{L-1}) \odot (A^L)^T \delta^L]$$

  The value is approximately the multiplication of $L - l$ term. If $\sigma'(z^\ell) < 1$ or $\|A^\ell\|_2 < 1$, then δ^ℓ will be exponentially small.

- Roughly, $\delta^\ell \approx (\sigma'(x)\|A\|_2)^{L-\ell}$. This implies that **deep networks are harder to train than shallow networks**.

- More precisely, it is due to the **disparity of gradient scales across different layers** that makes it challenging to select a single learning rate that works effectively for all layers simultaneously. Recall the picture of defining **condition number**!!



Observation

The **vanishing/exploding gradient** is the major obstacle in training deep nets.

Alleviate gradient vanishing: activation function

- **Saturating activation:** For saturating activation function, when $|z| > O(1)$, we have $\sigma'(z) \approx 0$. This is extremely bad for deep networks.
- **Non-saturating activation:** Use ReLU and its variants as the nonlinear activation function.

Alleviate gradient vanishing: initialization

- Denote by $\mathbf{x}^\ell$ the output of ℓ-th layer: $\mathbf{x}^{\ell+1} = \sigma(W^\ell \mathbf{x}^\ell + b^\ell)$.
- Consider the commonly random initialization $W_{i,j}^\ell \in \mathcal{N}(0, t_w^2), b_j^\ell = 0$ (we will discuss why Gaussian is preferred later) and the standard Gaussian input: $\mathbf{x} \sim \mathcal{N}(0, I_d)$.

How should the initialization scale t_w be chosen?

Alleviate gradient vanishing: initialization

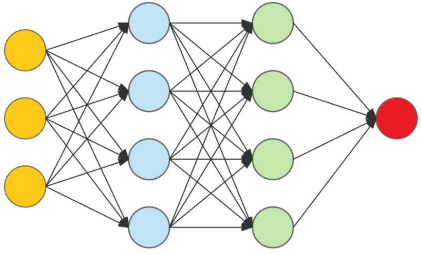
- Denote by $\mathbf{x}^\ell$ the output of ℓ-th layer: $\mathbf{x}^{\ell+1} = \sigma(W^\ell \mathbf{x}^\ell + b^\ell)$.
- Consider the commonly random initialization $W_{i,j}^\ell \in \mathcal{N}(0, t_w^2), b_j^\ell = 0$ (we will discuss why Gaussian is preferred later) and the standard Gaussian input: $\mathbf{x} \sim \mathcal{N}(0, I_d)$.

How should the initialization scale t_w be chosen?

- **Principle:** Ensure the forward process avoids vanishing. Specifically, we aim to choose an initialization such that

$$\mathbb{E}[|x_i^\ell|^2] = 1, \quad \ell \in [L], i \in [m^\ell]$$

where x_i^ℓ is the output of the i-th neuron of ℓ-th layer.



Derivation

Consider **ReLU** NNs and we have at initialization that

$$\begin{aligned}
\mathbb{E}[|x_i^{\ell+1}|^2] &= \mathbb{E}\left[\sigma^2\Big(\sum_{j=1}^{m_\ell} W_{i,j}^\ell x_j^\ell\Big)\right] \\
&= \mathbb{E}_{\mathbf{x}^\ell}\,\mathbb{E}_{\xi_{i,j}\sim\mathcal{N}(0,1)}\left[\|\mathbf{x}^\ell\|^2 t_w^2 \sigma^2\left(\sum_{j=1}^{m_\ell} \xi_{i,j}^\ell \hat{x}_j^\ell\right)\Big|\mathbf{x}^\ell\right] \\
&= \mathbb{E}_{\mathbf{x}^\ell}\,\mathbb{E}_{\zeta\sim\mathcal{N}(0,1)}\left[\|\mathbf{x}^\ell\|^2 t_w^2 \sigma^2(\zeta)\big|\mathbf{x}^\ell\right] \\
&= m^\ell t_w^2 \mathbb{E}_{\zeta\sim\mathcal{N}(0,1)}[\sigma^2(\zeta)]. \qquad \text{Assume } \mathbb{E}[|x_i^\ell|^2] = 1, \qquad (1)
\end{aligned}$$

where the second step follows from the positive homogeneity of ReLU: $\sigma(\lambda z) = \lambda\sigma(z)$ for any $\lambda \geq 0$ and $z \in \mathbb{R}$.

Derivation (cont'd)

Note that $\mathbb{E}_{\zeta\sim\mathcal{N}(0,1)}[\sigma^2(\zeta)] = \frac{1}{\sqrt{2\pi}}\int_0^\infty z^2 e^{-z^2/2}\,\mathrm{d}z = 1/2$. Hence, to ensure $\mathbb{E}[|x_i^{\ell+1}|^2] = 1$, we can take

$$t_w^2 = \frac{2}{m_\ell}.$$

- The initialization $W_{i,j}^\ell \overset{iid}{\sim} \mathcal{N}(0, 2/m_\ell), b_j = 0$ is called **Kaiming-He initialization**, which has become the default initialization for all the ReLU-like activation functions.

Derivation (cont'd)

Note that $\mathbb{E}_{\zeta\sim\mathcal{N}(0,1)}[\sigma^2(\zeta)] = \frac{1}{\sqrt{2\pi}}\int_0^\infty z^2 e^{-z^2/2}\,\mathrm{d}z = 1/2$. Hence, to ensure $\mathbb{E}[|x_i^{\ell+1}|^2] = 1$, we can take

$$t_w^2 = \frac{2}{m_\ell}.$$

- The initialization $W_{i,j}^\ell \overset{iid}{\sim} \mathcal{N}(0, 2/m_\ell), b_j = 0$ is called **Kaiming-He initialization**, which has become the default initialization for all the ReLU-like activation functions.
- Similarly, we can get $t_w^2 = 1/m^\ell$ if $\sigma(z) = z$. This corresponds to the LeCun initialization. LeCun initialization works pretty well for the $\tanh$ activation function, since $\tanh(z) \approx z$ when z is close to the origin.

Derivation (cont'd)

Note that $\mathbb{E}_{\zeta\sim\mathcal{N}(0,1)}[\sigma^2(\zeta)] = \frac{1}{\sqrt{2\pi}}\int_0^\infty z^2 e^{-z^2/2}\,\mathrm{d}z = 1/2$. Hence, to ensure $\mathbb{E}[|x_i^{\ell+1}|^2] = 1$, we can take

$$t_w^2 = \frac{2}{m_\ell}.$$

- The initialization $W_{i,j}^\ell \overset{iid}{\sim} \mathcal{N}(0, 2/m_\ell), b_j = 0$ is called **Kaiming-He initialization**, which has become the default initialization for all the ReLU-like activation functions.
- Similarly, we can get $t_w^2 = 1/m^\ell$ if $\sigma(z) = z$. This corresponds to the LeCun initialization. LeCun initialization works pretty well for the $\tanh$ activation function, since $\tanh(z) \approx z$ when z is close to the origin.
- Similar argument can be used to derive the initialization for other activation functions.

Derivation (cont'd)

Note that $\mathbb{E}_{\zeta\sim\mathcal{N}(0,1)}[\sigma^2(\zeta)] = \frac{1}{\sqrt{2\pi}}\int_0^\infty z^2 e^{-z^2/2}\,\mathrm{d}z = 1/2$. Hence, to ensure $\mathbb{E}[|x_i^{\ell+1}|^2] = 1$, we can take

$$t_w^2 = \frac{2}{m_\ell}.$$

- The initialization $W_{i,j}^\ell \overset{iid}{\sim} \mathcal{N}(0, 2/m_\ell), b_j = 0$ is called **Kaiming-He initialization**, which has become the default initialization for all the ReLU-like activation functions.
- Similarly, we can get $t_w^2 = 1/m^\ell$ if $\sigma(z) = z$. This corresponds to the LeCun initialization. LeCun initialization works pretty well for the $\tanh$ activation function, since $\tanh(z) \approx z$ when z is close to the origin.
- Similar argument can be used to derive the initialization for other activation functions.
- It is also common to use the uniform initialization: $W_{i,j}^\ell \sim \mathrm{Unif}[-t, t]$, where the specific value of t can be derived similarly.

Numerical illustration

In the following figure, we see that with the right initialization, we can avoid the vanishing/exploding for both the forward and backward propagation **at the initialization**.

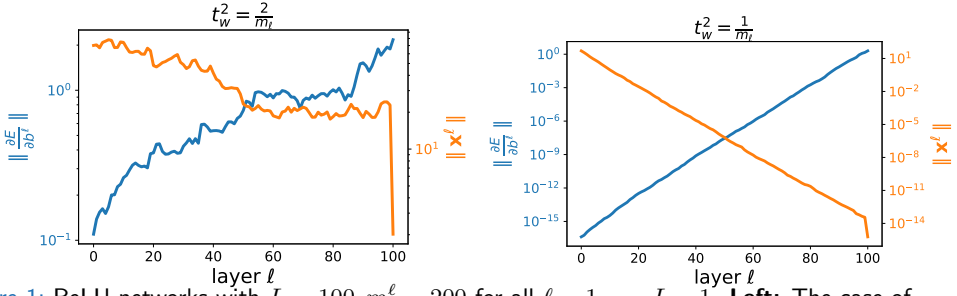


Figure 1: ReLU networks with $L = 100, m^\ell = 200$ for all $\ell = 1, \ldots, L-1$. **Left:** The case of $t_w^2 = 2/m_\ell$ (Kaiming-He initialization); **Right:** The case of $t_w^2 = 1/m_\ell$ (LeCun initialization).

Why do we choose the random initialization with a large support?

1. We have an understanding for the size of the initialization.

Why do we choose the random initialization with a large support?

1. We have an understanding for the size of the initialization.
2. We do not have an understanding for the directions we need.
    1. Consider a two-layer neural network

$$f(x) = \sum_{i=1}^{m} a_i \, \sigma\big(\mathbf{w}_i^\top \mathbf{x} + b_i\big).$$

If $(a_i, \mathbf{w}_i, b_i) = (a_j, \mathbf{w}_j, b_j)$ at initialization, then they will remain the same for all time under gradient flow optimization.

Why do we choose the random initialization with a large support?

1. We have an understanding for the size of the initialization.
2. We do not have an understanding for the directions we need.
   1. Consider a two-layer neural network

$$f(x) = \sum_{i=1}^{m} a_i \, \sigma\big(\mathbf{w}_i^\top \mathbf{x} + b_i\big).$$

   If $(a_i, \mathbf{w}_i, b_i) = (a_j, \mathbf{w}_j, b_j)$ at initialization, then they will remain the same for all time under gradient flow optimization.
   2. We want 'diverse' initialization with many different vectors in many different directions, but we do not know which directions are important.
   3. Popular: random initialization with mean zero and appropriate variance.

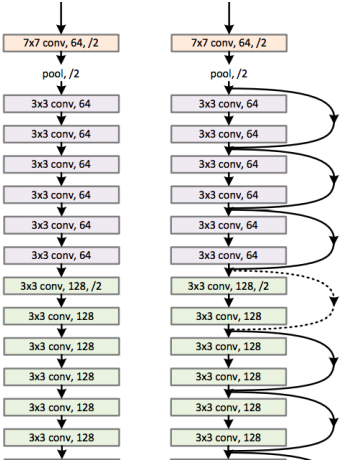
Why do we choose the random initialization with a large support?

1. We have an understanding for the size of the initialization.
2. We do not have an understanding for the directions we need.
   1. Consider a two-layer neural network

$$f(x) = \sum_{i=1}^{m} a_i \, \sigma\big(\mathbf{w}_i^\top \mathbf{x} + b_i\big).$$

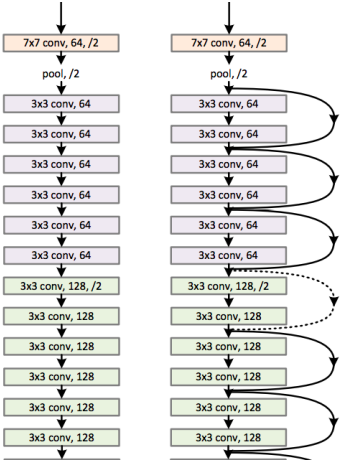
      If $(a_i, \mathbf{w}_i, b_i) = (a_j, \mathbf{w}_j, b_j)$ at initialization, then they will remain the same for all time under gradient flow optimization.
   2. We want 'diverse' initialization with many different vectors in many different directions, but we do not know which directions are important.
   3. Popular: random initialization with mean zero and appropriate variance.
3. We can explore other forms of initialization, e.g., **the orthogonal initialization**: choosing W^ℓ to be the multiple of an orthogonal matrix (if $m_{\ell+1} = m_\ell$). Whether these initialization overperform or underperform random Gaussian seems to be problem dependent and is not fully understood.

Alleviate the gradient vanishing: Skip connections



- Intuitively speaking, **skip connections** build highways for the information propagation, such that information does not need to go through the convolutional, fully-connected, and activation layers.

Alleviate the gradient vanishing: Skip connections



- Intuitively speaking, **skip connections** build highways for the information propagation, such that information does not need to go through the convolutional, fully-connected, and activation layers.
- Mathematically,
  - $x^{\ell+1} = x^{\ell} + h_{\ell}(x^{\ell})$
  - $x^{L} = x^{\ell} + \sum_{i=\ell}^{L-1} h_i(x^i)$
  - $\frac{\partial E}{\partial x^{\ell}} = \frac{\partial E}{\partial x^{L}} \left(1 + \sum_{i=\ell}^{L-1} \frac{\partial h_i(x^i)}{\partial x^{\ell}}\right)$.

  If the residual blocks $\{h_i\}$ are small, one can see that the gradients are almost independent of the depth. So the gradient is well-controlled.
- History: **LSTM** $\rightarrow$ Highway network $\rightarrow$ **ResNet**.

Numerical evidence

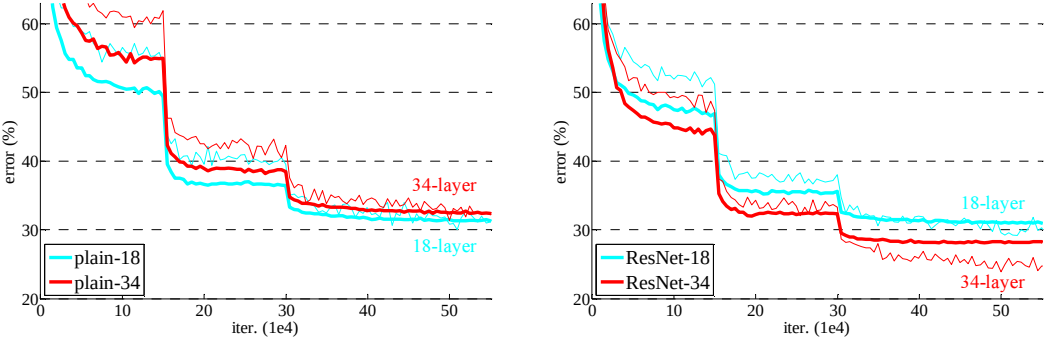


Figure 2: Training on ImageNet. Thin curves denote training error, and bold curves denote validation error. This figure is taken from (Kaiming He et al., 2015).

Alleviate the gradient vanishing: Batch normalization

- Batch normalization(BN) is one of **most effective** method to alleviate the gradient vanishing issue.

Alleviate the gradient vanishing: Batch normalization

- Batch normalization(BN) is one of **most effective** method to alleviate the gradient vanishing issue.
- A batch normalization layer define a map: $\mathrm{BN}_{\gamma,\beta} : \{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \to \{\tilde{\mathbf{x}}_1, \ldots \tilde{\mathbf{x}}_m\}$ through

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (\mathbf{x}_i - \mu_{\mathcal{B}})^2$$

$$\widehat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$\tilde{\mathbf{x}}_i \leftarrow \gamma \widehat{\mathbf{x}}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(\mathbf{x}_i)$$

where γ, β are reintroduced to preserve the network's expressivity.

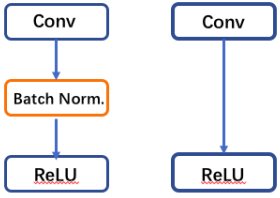


Figure 3: **Left:** Convolutional nets with BN; **Right:** Convolutional without BN.

Performance of BN

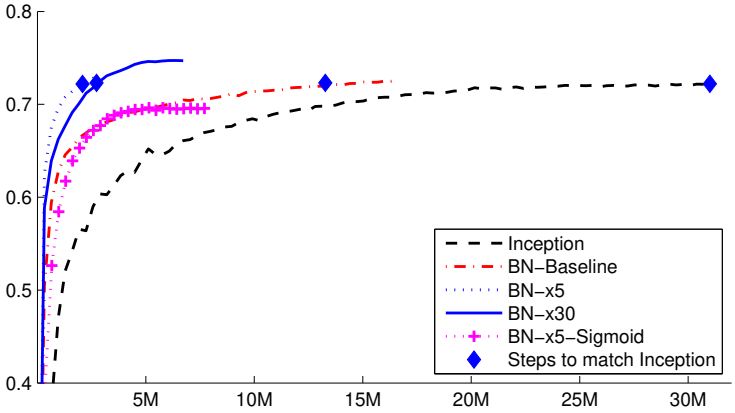


Figure 4: Validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps. BN-baseline: same as inception with BN layers added before each nonlinearity. BN-x5: inception with batch normalization and the learning rate is increased by a factor 5, compared to the baseline. BN-x30 is similar. This figure is taken from `https://arxiv.org/pdf/1502.03167.pdf`.

Black magics: Batch normalization

BN is VERY useful for training very deep nets. But it also causes several strange issues.

- For networks with BN layers, **we cannot use too small batch size**, e.g. $B = 1$, where the σ_B and μ_B are far away from the σ and μ, the ones over the whole dataset.)

Black magics: Batch normalization

BN is VERY useful for training very deep nets. But it also causes several strange issues.

- For networks with BN layers, **we cannot use too small batch size**, e.g. $B = 1$, where the σ_B and μ_B are far away from the σ and μ, the ones over the whole dataset.)
- How do we compute σ_B and μ_B during the inference, where we may only have one sample?

Black magics: Batch normalization

BN is VERY useful for training very deep nets. But it also causes several strange issues.

- For networks with BN layers, **we cannot use too small batch size**, e.g. $B = 1$, where the σ_B and μ_B are far away from the σ and μ, the ones over the whole dataset.)
- How do we compute σ_B and μ_B during the inference, where we may only have one sample?

Black magics: Batch normalization

BN is VERY useful for training very deep nets. But it also causes several strange issues.

- For networks with BN layers, **we cannot use too small batch size**, e.g. $B = 1$, where the σ_B and μ_B are far away from the σ and μ, the ones over the whole dataset.)
- How do we compute σ_B and μ_B during the inference, where we may only have one sample?
  Use the following ones obtained from the moving average during the training:

$$\sigma^{\text{inf}} \leftarrow (1-\alpha)\sigma^{\text{inf}} + \alpha\sigma_B^t \tag{2}$$

$$\mu^{\text{inf}} \leftarrow (1-\alpha)\mu^{\text{inf}} + \alpha\mu_B^t, \tag{3}$$

  where σ_B^t, μ_B^t are the statistics calculated at the t-th step of training.

Black magics: Batch normalization

Training and test disparity:

- At the training time, $\{\sigma_B, \mu_B\}$ are computed over the samples at the current batch.
- At the inference/testing time, $\{\sigma^{\text{inf}}, \mu^{\text{inf}}\}$ are fixed, which are approximations of the statistics of whole dataset obtained by the exponential moving average during the training.

Layer normalization

- Let $Z = (\mathbf{z}_1, \ldots, \mathbf{z}_B)^\top \in \mathbb{R}^{B \times H}$ be our feature map. The first and second dimensions represent the batch and feature dimensions, respectively. [2]
- A layer normalization (LN) layer define a map $\mathrm{LN}_{\boldsymbol{\gamma},\boldsymbol{\beta}} : \{\mathbf{z}_1, \ldots, \mathbf{z}_B\} \to \{\tilde{\mathbf{z}}_1, \ldots \tilde{\mathbf{z}}_B\}$ as follows

$$\mu_i = \frac{1}{H}\sum_{j=1}^{H} z_{i,j}, \quad \sigma_i = \sqrt{\frac{1}{H}\sum_{j=1}^{H}(z_{i,j} - \mu_i)^2} \quad \text{for } i = 1, \ldots, B, \tag{4}$$

$$\hat{\mathbf{z}}_i \leftarrow \boldsymbol{\gamma} \odot \frac{\mathbf{z}_i - \mu_i}{\sigma_i} + \boldsymbol{\beta}, \tag{5}$$

where the learnable rescaling factors $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^H$.

[2] For simplicity, we consider the feature map to have only one feature dimension. However, for models like CNNs, H should be interpreted as the product of width, height, and channels.

Layer normalization

- Let $Z = (\mathbf{z}_1, \ldots, \mathbf{z}_B)^\top \in \mathbb{R}^{B \times H}$ be our feature map. The first and second dimensions represent the batch and feature dimensions, respectively. [2]
- A layer normalization (LN) layer define a map $\mathrm{LN}_{\boldsymbol{\gamma},\boldsymbol{\beta}} : \{\mathbf{z}_1, \ldots, \mathbf{z}_B\} \to \{\tilde{\mathbf{z}}_1, \ldots \tilde{\mathbf{z}}_B\}$ as follows

$$\mu_i = \frac{1}{H}\sum_{j=1}^{H} z_{i,j}, \quad \sigma_i = \sqrt{\frac{1}{H}\sum_{j=1}^{H}(z_{i,j} - \mu_i)^2} \quad \text{for } i = 1, \ldots, B, \tag{4}$$

$$\hat{\mathbf{z}}_i \leftarrow \boldsymbol{\gamma} \odot \frac{\mathbf{z}_i - \mu_i}{\sigma_i} + \boldsymbol{\beta}, \tag{5}$$

  where the learnable rescaling factors $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^H$.
- Unlike from BN, LN normalizes data along the feature dimension and performs rescaling in an element-wise manner.
- **Question: Is element-wise rescaling necessary for LN?**

[2] For simplicity, we consider the feature map to have only one feature dimension. However, for models like CNNs, H should be interpreted as the product of width, height, and channels.

Layer normalization (cont'd)

- BN is often utilized in MLP and CNN, whereas LN is more frequently employed in training RNNs and Transformers.
- LN is uniquely advantageous as it can be effectively applied even when the batch size is as small as 1.

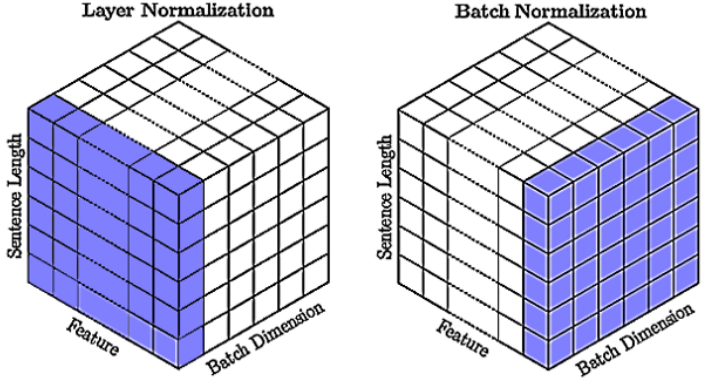


Figure 5: Taken from `https://www.kaggle.com/code/halflingwizard/how-does-layer-normalization-work`.

Avoid gradient exploding: gradient clipping

- Let g_t denote the stochastic gradient at step t.

Avoid gradient exploding: gradient clipping

- Let g_t denote the stochastic gradient at step t.
- Replace g_t with its clipped version: $g_t \to \mathrm{clip}_\gamma(g_t)$, where the clipping operator is defined as

$$\mathrm{clip}_\gamma(g) = \min\left(1, \frac{\gamma}{\|g\|}\right) g$$

Avoid gradient exploding: gradient clipping

- Let g_t denote the stochastic gradient at step t.
- Replace g_t with its clipped version: $g_t \to \mathrm{clip}_\gamma(g_t)$, where the clipping operator is defined as

$$\mathrm{clip}_\gamma(g) = \min\left(1, \frac{\gamma}{\|g\|}\right) g$$

- In certain situations, element-wise clipping is more effective:

$$\big(\mathrm{clip}_\gamma(g)\big)_i = \min\left(1, \frac{\gamma}{|g_i|}\right) g_i.$$

Avoid gradient exploding: gradient clipping

- Let g_t denote the stochastic gradient at step t.
- Replace g_t with its clipped version: $g_t \to \mathrm{clip}_\gamma(g_t)$, where the clipping operator is defined as

$$\mathrm{clip}_\gamma(g) = \min\left(1, \frac{\gamma}{\|g\|}\right) g$$

- In certain situations, element-wise clipping is more effective:

$$\big(\mathrm{clip}_\gamma(g)\big)_i = \min\left(1, \frac{\gamma}{|g_i|}\right) g_i.$$

- Gradient clipping is widely used in training recurrent neural networks (RNNs) and Transformer. One potential mechanism behind clipping is to mitigate the impact of heavy-tailed noise. Recall that in the convergence analysis of SGD, convergence requires

$$\mathbb{E}[|\xi_t|^2] < \infty.$$

What happens if the above condition is not met? Read: Why are adaptive methods good for attention models?

Adaptive learning rate optimizers

Motivation

- Gradients across different layers have varying scales, making a single global learning rate ineffective. Recall the picture behind the concept of condition number.
- Layer-wise learning rates! A great idea but hard to implement.
- Adaptive learning rates:

  Automatically tune learning rates according to the gradient size of each coordinate.

Adagrad

- Consider to minimize $\min_{x \in \mathbb{R}^p} f(x)$. Let g_t be the t-th step (stochastic) gradient.
- SGD updates as follows

$$x_{t+1} = x_t - \eta g_t.$$

Adagrad

- Consider to minimize $\min_{x\in\mathbb{R}^p} f(x)$. Let g_t be the t-th step (stochastic) gradient.
- SGD updates as follows

$$x_{t+1} = x_t - \eta g_t.$$

- The adaptive gradient (Adagrad) method updates as follows

$$\begin{aligned} G_{t+1} &= G_t + g_t^2 \\ x_{t+1} &= x_t - \eta \frac{g_t}{\sqrt{G_{t+1} + \varepsilon}}, \end{aligned}$$

where $\varepsilon \sim 10^{-7}$ prevents the division by zero. All multiplication and division should be understood in an element-wise manner.

Adagrad

- Consider to minimize $\min_{x\in\mathbb{R}^p} f(x)$. Let g_t be the t-th step (stochastic) gradient.
- SGD updates as follows

$$x_{t+1} = x_t - \eta g_t.$$

- The adaptive gradient (Adagrad) method updates as follows

$$G_{t+1} = G_t + g_t^2$$
$$x_{t+1} = x_t - \eta \frac{g_t}{\sqrt{G_{t+1} + \varepsilon}},$$

  where $\varepsilon \sim 10^{-7}$ prevents the division by zero. All multiplication and division should be understood in an element-wise manner.
- Note that where $G_t = \sum_{s=0}^{t} g_t^2$ stores the magnitude of each coordinate.

Adagrad

- Consider to minimize $\min_{x\in\mathbb{R}^p} f(x)$. Let g_t be the t-th step (stochastic) gradient.
- SGD updates as follows

$$x_{t+1} = x_t - \eta g_t.$$

- The adaptive gradient (Adagrad) method updates as follows

$$G_{t+1} = G_t + g_t^2$$
$$x_{t+1} = x_t - \eta \frac{g_t}{\sqrt{G_{t+1} + \varepsilon}},$$

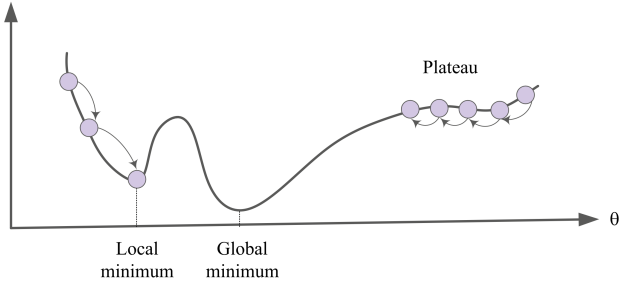
  where $\varepsilon \sim 10^{-7}$ prevents the division by zero. All multiplication and division should be understood in an element-wise manner.
- Note that where $G_t = \sum_{s=0}^{t} g_t^2$ stores the magnitude of each coordinate.
- **Issue:** G_t is increasing monotonically. Thus, the effective learning rate is decreasing in time.

rProp

- To alleviate the "the gradients of different weights/layers are very different", the **rProp** (resilient prop) [3] algorithm proposes to update weights using only the **sign** of gradient.

$$x_{t+1} = x_t - \eta \text{sign}(\nabla f(x_t)).$$

  - All weights updates all of the same magnitude!
  - It escapes from plateaus with tiny gradient quickly!



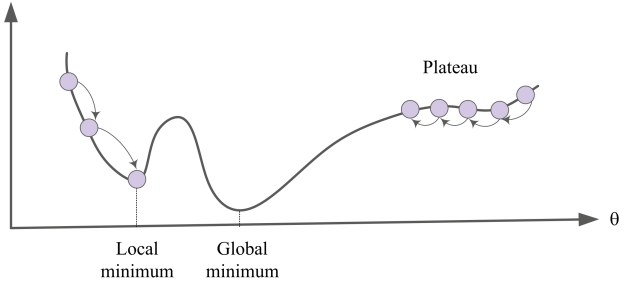
[3] M. Riedmiller, H. Braun, A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. IJCNN 1993.

rProp

- To alleviate the "the gradients of different weights/layers are very different", the **rProp** (resilient prop) [3] algorithm proposes to update weights using only the **sign** of gradient.

$$x_{t+1} = x_t - \eta \text{sign}(\nabla f(x_t)).$$

  - All weights updates all of the same magnitude!
  - It escapes from plateaus with tiny gradient quickly!



- Unfortunately, **rProp does not work with minibatch gradients!**

[3] M. Riedmiller, H. Braun, A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. IJCNN 1993.

RMSProp

- RMSProp (Tieleman & Hinton, 2012) iterates as follows

$$v_{t+1} = \beta v_t + (1-\beta) g_t^2$$
$$x_{t+1} = x_t - \eta \frac{g_t}{\sqrt{v_{t+1} + \varepsilon}}.$$

RMSProp

- RMSProp (Tieleman & Hinton, 2012) iterates as follows

$$v_{t+1} = \beta v_t + (1-\beta) g_t^2$$
$$x_{t+1} = x_t - \eta \frac{g_t}{\sqrt{v_{t+1} + \varepsilon}}.$$

- If $\beta = 0, g_t = \nabla f(x_t)$ (i.e., the full-batch case), it becomes

$$x_{t+1} = x_t - \eta \frac{\nabla f(x_t)}{\sqrt{|\nabla f(x_t)|^2 + \varepsilon}} \qquad (\textbf{rProp}\ ).$$

  It is the **sign** gradient descent (signGD) if $\varepsilon = 0$.

RMSProp

- RMSProp (Tieleman & Hinton, 2012) iterates as follows

$$v_{t+1} = \beta v_t + (1-\beta) g_t^2$$
$$x_{t+1} = x_t - \eta \frac{g_t}{\sqrt{v_{t+1} + \varepsilon}}.$$

- If $\beta = 0, g_t = \nabla f(x_t)$ (i.e., the full-batch case), it becomes

$$x_{t+1} = x_t - \eta \frac{\nabla f(x_t)}{\sqrt{|\nabla f(x_t)|^2 + \varepsilon}} \qquad (\textbf{rProp}\ ).$$

  It is the **sign** gradient descent (signGD) if $\varepsilon = 0$.
- RMSProp is originally proposed as a stochastic version of rProp in Hinton's course slide. [**Q: Why is it non-trivial?**]

ADAM [4] optimizer

1. Compute the (stochastic) gradient g_t.
2. Estimate the first-order and second-order moment:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2.$$

[4] Adaptive momentum: https://arxiv.org/pdf/1412.6980.pdf

ADAM [4] optimizer

1. Compute the (stochastic) gradient g_t.
2. Estimate the first-order and second-order moment:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2.$$

3. Bias correction:

$$m_{t+1} = \frac{m_{t+1}}{1 - \beta_1^t}, \qquad v_{t+1} = \frac{v_{t+1}}{1 - \beta_2^t}.$$

[4] Adaptive momentum: https://arxiv.org/pdf/1412.6980.pdf

ADAM [4] optimizer

1. Compute the (stochastic) gradient g_t.
2. Estimate the first-order and second-order moment:

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2) g_t^2.$$

3. Bias correction:

$$m_{t+1} = \frac{m_{t+1}}{1-\beta_1^t}, \qquad v_{t+1} = \frac{v_{t+1}}{1-\beta_2^t}.$$

4. Update parameters:

$$x_{t+1} = x_t - \eta \frac{m_{t+1}}{\sqrt{v_{t+1}+\varepsilon}}.$$

   Again, the square root and division are computed in coordinate-wise manner.

In a summary,

ADAM=RMSProp + momentum

[4] Adaptive momentum: https://arxiv.org/pdf/1412.6980.pdf

ADAM: Explanation

1. Common initialization: $m_0 = 0, v_0 = 0$. This causes m_t and g_t to be small for small t. If $\mathbb{E}[g_t^2] \approx g^2$, then

$$\mathbb{E}[v_t] = (1 - \beta_2) \sum_{s=0}^{t} \beta_2^s \, \mathbb{E}[g_{t-s}^2] \approx g^2 (1 - \beta_2^t).$$

   The bias-correction step is used to correct this **initialization bias**.

2. We can also initialize $m_0 = g_0, v_0 = g_0^2$, for which the bias-correction step is not necessary.

ADAM: Explanation

1. Common initialization: $m_0 = 0, v_0 = 0$. This causes m_t and g_t to be small for small t. If $\mathbb{E}[g_t^2] \approx g^2$, then

$$\mathbb{E}[v_t] = (1 - \beta_2) \sum_{s=0}^{t} \beta_2^s \, \mathbb{E}[g_{t-s}^2] \approx g^2 (1 - \beta_2^t).$$

   The bias-correction step is used to correct this **initialization bias**.
2. We can also initialize $m_0 = g_0, v_0 = g_0^2$, for which the bias-correction step is not necessary.
3. Default hyperparameters in the original Adam paper are $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$. The learning rate should be tuned experimentally in each problem.
4. There are (very recent) convergence results for ADAM both in the convex and non-convex case. However, all these theoretical results are not interesting since they cannot explain why ADAM converges faster than SGD.

Continuous-time limits of ADAM

Let $\beta_1 = 1 - \eta\gamma_1, \beta_2 = 1 - \eta\gamma_2$. Then, taking $\eta \to 0$, the limit becomes

$$\begin{aligned} \dot{m}_t &= \gamma_1(\nabla f(x_t) - m_t) \\ \dot{v}_t &= \gamma_2(|\nabla f(x_t)|^2 - v_t) \\ \dot{x}_t &= -\frac{m_t}{\sqrt{v_t + \varepsilon}} \end{aligned}$$

Taking $\gamma_1, \gamma_2 \to \infty$, we obtain the signGD flow:

$$\dot{x}_t = -\frac{\nabla f(x_t)}{\sqrt{|\nabla f(x_t)|^2 + \varepsilon}}.$$

A comparison of different optimizers

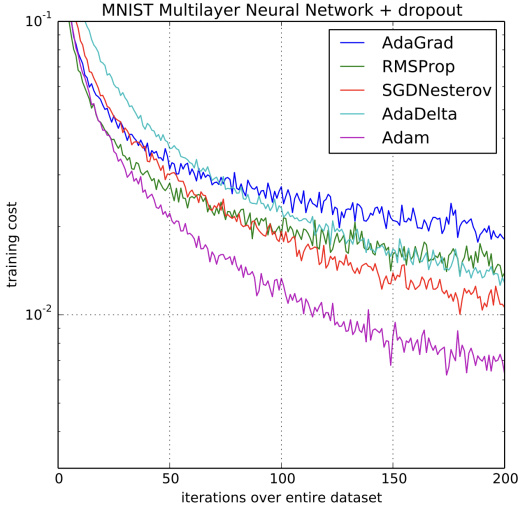
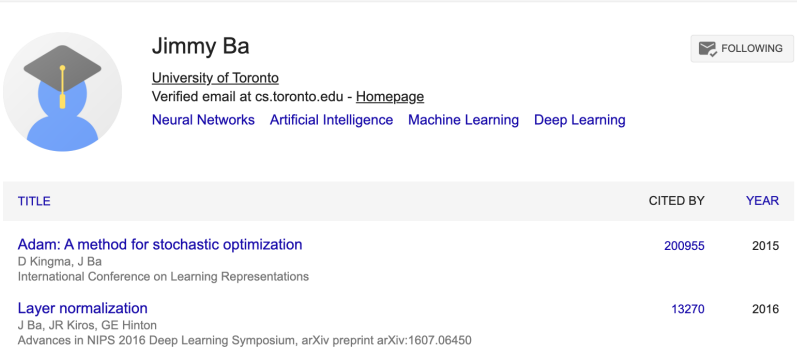


Figure 6: Taken from the original Adam paper `https://arxiv.org/pdf/1412.6980.pdf`.

Summary

- Adam has become the de facto optimizer for training large neural networks.



Jimmy Ba

FOLLOWING

University of Toronto
Verified email at cs.toronto.edu - Homepage
Neural Networks Artificial Intelligence Machine Learning Deep Learning

TITLE	CITED BY	YEAR
Adam: A method for stochastic optimization D Kingma, J Ba International Conference on Learning Representations	200955	2015
Layer normalization J Ba, JR Kiros, GE Hinton Advances in NIPS 2016 Deep Learning Symposium, arXiv preprint arXiv:1607.06450	13270	2016

- For smaller networks or tasks requiring high precision, consider using second-order optimizers such as L-BFGS, K-FAC, or the Natural Gradient method.

Regularization

Over-parameterization in deep learning

- Neural networks often work in the over-parameterized regime, i.e., the number of samples are much larger than data size.

CIFAR-10	# train: 50,000
Inception	1,649,402
Alexnet	1,387,786
MLP 1x512	1,209,866
ImageNet	**# train: ~1,200,000**
Inception V4	42,681,353
Alexnet	61,100,840
Resnet-{18;152}	11,689,512; 60,192,808
VGG-{11;19}	132,863,336; 143,667,240

Weight decay

- Let $\theta_{t+1} = \theta_t - \eta h_t$ be the update of the algorithm $\mathcal{A}$. The $\mathcal{A}$+weight decay iterates as follows

$$\theta_{t+1} = \theta_t - \eta(h_t + \lambda\theta_t) = (1 - \lambda\eta)\theta_t - \eta h_t.$$

You should always try it due to the simplicity.

Weight decay

- Let $\theta_{t+1} = \theta_t - \eta h_t$ be the update of the algorithm $\mathcal{A}$. The $\mathcal{A}$+weight decay iterates as follows

$$\theta_{t+1} = \theta_t - \eta(h_t + \lambda\theta_t) = (1 - \lambda\eta)\theta_t - \eta h_t.$$

- When $\mathcal{A}$ is SGD, it is equivalent to the squared ℓ^2 regularization as

$$\nabla\left(\hat{\mathcal{R}}(\theta) + \frac{\lambda}{2}\|\theta\|^2\right) = \nabla\hat{\mathcal{R}}(\theta) + \lambda\theta.$$

You should always try it due to the simplicity.

Weight decay

- Let $\theta_{t+1} = \theta_t - \eta h_t$ be the update of the algorithm $\mathcal{A}$. The $\mathcal{A}$+weight decay iterates as follows
$$\theta_{t+1} = \theta_t - \eta(h_t + \lambda\theta_t) = (1 - \lambda\eta)\theta_t - \eta h_t.$$
- When $\mathcal{A}$ is SGD, it is equivalent to the squared ℓ^2 regularization as
$$\nabla\left(\hat{\mathcal{R}}(\theta) + \frac{\lambda}{2}\|\theta\|^2\right) = \nabla\hat{\mathcal{R}}(\theta) + \lambda\theta.$$
- **For other optimizers like ADAM, they are not equivalent**. This issue was first pointed out in `https://openreview.net/pdf?id=rk6qdGgCZ` and currently, Adam + weight decay (**AdamW**) has become the default optimizers in training large language models (LLMs), e.g., ChatGPT.

You should always try it due to the simplicity.

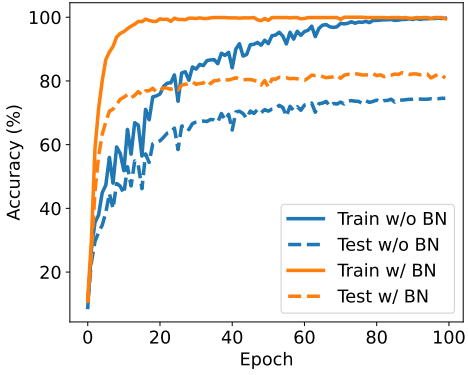
Weight decay

- Let $\theta_{t+1} = \theta_t - \eta h_t$ be the update of the algorithm $\mathcal{A}$. The $\mathcal{A}$+weight decay iterates as follows
$$\theta_{t+1} = \theta_t - \eta(h_t + \lambda\theta_t) = (1 - \lambda\eta)\theta_t - \eta h_t.$$
- When $\mathcal{A}$ is SGD, it is equivalent to the squared ℓ^2 regularization as
$$\nabla\left(\hat{\mathcal{R}}(\theta) + \frac{\lambda}{2}\|\theta\|^2\right) = \nabla\hat{\mathcal{R}}(\theta) + \lambda\theta.$$
- **For other optimizers like ADAM, they are not equivalent**. This issue was first pointed out in `https://openreview.net/pdf?id=rk6qdGgCZ` and currently, Adam + weight decay (**AdamW**) has become the default optimizers in training large language models (LLMs), e.g., ChatGPT.
- **Why weight decay is so useful in training LLMs is still unclear !!! [A research topic!!]**

You should always try it due to the simplicity.

Batch normalization

- BN is originally proposed to improve the training. In practice, it is found that **BN can also improve the generalization significantly**.
- Always try it, since it improves both convergence and generalization.
- Why BN has regularization effect is still unclear now.



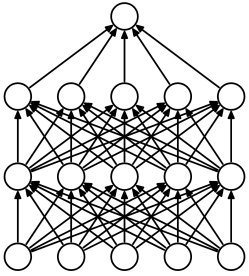
Dropout: basic definition

Vanilla forward-prop

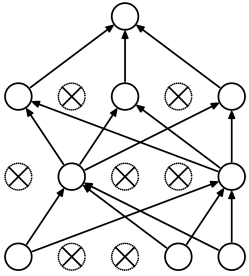
$$z_i^\ell = (\mathbf{w}_i^\ell)^\top \mathbf{x}^{\ell-1} + b_i^\ell$$
$$x_i^\ell = \sigma(z_i^\ell)$$

Dropout forward-prop

$$r_i^{\ell-1} \overset{iid}{\sim} \text{Bernoulli}(p)$$
$$z_i^\ell = (\mathbf{w}_i^\ell)^\top (\boldsymbol{r}^{\ell-1} \odot \mathbf{x}^{\ell-1}) + b_i^\ell$$
$$x_i^\ell = \sigma(z_i^\ell)$$



(a) Standard Neural Net



(b) After applying dropout.

- In each step, the dropping mask is randomly sampled. Hence, the masks can be different in different steps.
- The drop ratio is given by $1 - p$.

Dropout: A stochastic approximation explanation

- Dropout defines a stochastic network $F(x; \xi, \theta) = f(x; \xi \odot \theta)$, where ξ denotes the dropping mask.
- Denote by π the distribution of the mask ξ. Then the dropout training goes as follows,

$$\begin{aligned} \xi_t &\sim \pi \\ \theta_{t+1} &= \theta_t - \eta \nabla_\theta \widehat{\mathcal{R}}(F(\cdot; \xi_t, \theta_t)), \end{aligned} \tag{6}$$

  which is exactly SGD of batch size 1 for minimizing

$$\widehat{\mathcal{R}}_{\text{drop}}(\theta) = \mathbb{E}_{\xi \sim \pi}[\widehat{\mathcal{R}}(F(\cdot; \xi, \theta))]. \tag{7}$$

The ensemble viewpoint of dropout

- Let $\widehat{\mathcal{R}}(h) = \frac{1}{n}\sum_{i=1}^{n}(h(x_i) - y_i)^2$. Let $F_\theta(x) = \mathbb{E}_\xi[F(x;\xi,\theta)]$ be the effective model.
- Then,

$$\begin{aligned}\widehat{\mathcal{R}}_{\text{drop}}(\theta) &= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}_\xi(F(x_i;\xi,\theta) - y_i)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(F_\theta(x_i) - y_i)^2 + \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}_\xi(f(x_i;\xi,\theta) - F(x_i;\theta))^2 \\ &= \widehat{\mathcal{R}}(F_\theta) + Q_p(\theta),\end{aligned}$$

where the $Q_p(\cdot)$ term plays the role of regularization. Moreover,

$$Q_p(\theta) \to 0 \text{ as } p \to 1.$$

The ensemble viewpoint of dropout

- By the above derivation, The dropout model is given by

$$F_\theta(x) = \mathbb{E}_\xi[F(x;\xi,\theta)],$$

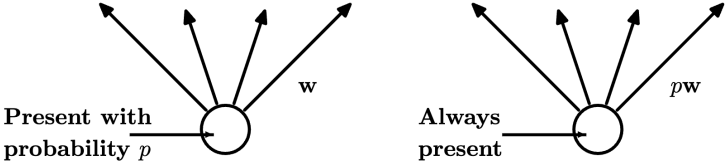
  which means that the effective model is an average/ensemble of sparse sub-networks.
- At training, stochastic approximation is applied. At testing, we can use Monte-Carlo approximation:

$$F_\theta(x) = \mathbb{E}_\xi[F(x;\xi,\theta)] \approx \frac{1}{m}\sum_{j=1}^{m} F(x;\xi_j,\theta).$$

The mean-field approximation

- MC approximation is reliable but computationally expensive. A more efficient way is the **mean-field** approximation:

$$\mathbb{E}_\xi[F(x;\xi,\theta)] \approx F(x;\mathbb{E}_\xi[\xi],\theta)] = f(x;\mathbb{E}[\xi]\odot\theta) = f(x;p\theta).$$

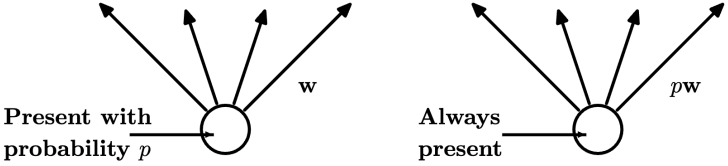


(a) At training time (b) At test time

The mean-field approximation

- MC approximation is reliable but computationally expensive. A more efficient way is the **mean-field** approximation:

$$\mathbb{E}_\xi[F(x;\xi,\theta)] \approx F(x;\mathbb{E}_\xi[\xi],\theta)] = f(x;\mathbb{E}[\xi]\odot\theta) = f(x;p\theta).$$



(a) At training time (b) At test time

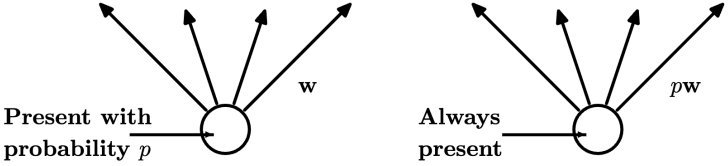
- **The error of mean-field approximation.** Let $\mu = \mathbb{E}[\xi]$. For a general $h \in C^2$,

$$\begin{aligned}\mathbb{E}[h(\xi)] &= \mathbb{E}_\xi[h(\mu) + h'(\mu)(\xi-\mu) + O(|\xi-\mu|^2)] \\ &= h(\mathbb{E}[\xi]) + O(\mathrm{Var}[\xi]).\end{aligned}$$

The mean-field approximation

- MC approximation is reliable but computationally expensive. A more efficient way is the **mean-field** approximation:

$$\mathbb{E}_\xi[F(x;\xi,\theta)] \approx F(x;\mathbb{E}_\xi[\xi],\theta)] = f(x;\mathbb{E}[\xi]\odot\theta) = f(x;p\theta).$$



(a) At training time (b) At test time

- **The error of mean-field approximation.** Let $\mu = \mathbb{E}[\xi]$. For a general $h \in C^2$,

$$\begin{aligned}\mathbb{E}[h(\xi)] &= \mathbb{E}_\xi[h(\mu) + h'(\mu)(\xi-\mu) + O(|\xi-\mu|^2)] \\ &= h(\mathbb{E}[\xi]) + O(\mathrm{Var}[\xi]).\end{aligned}$$

- Why is the above error small enough?

Dropout performance

- Usually, dropout is only applied to fully connected layer.
- Improvement depends on the problem.
- Dropout training is much slower.

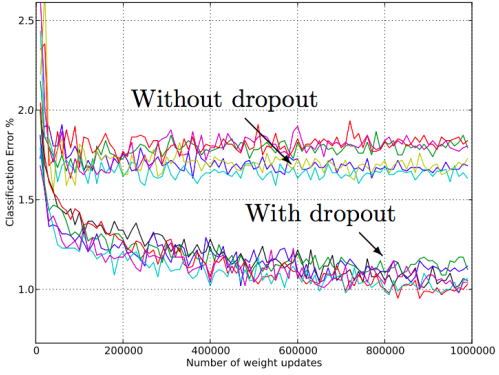
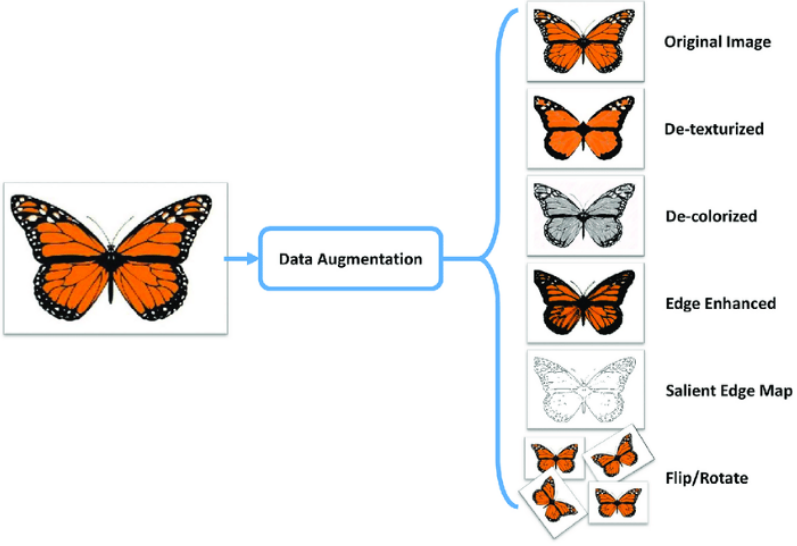


Figure 7: Taken from Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Data argumentation

- Increase the amount of data by adding slightly modified copies.
- Typically, most image transformations do not change the label, such as cropping, rotation, translation, resize, adding noise, Gaussian blurring.



Regularization in modern ML

- **Traditional viewpoint:** Add *explicit regularization*, e.g., Weight decay, batch/layer normalization, dropout, data argumentation.

Regularization in modern ML

- **Traditional viewpoint:** Add *explicit regularization*, e.g., Weight decay, batch/layer normalization, dropout, data argumentation.
- **Modern ML:** A specific algorithm (with a specific initialization) only converges to certain solution, which is called implicit regularization/bias.

Regularization in modern ML

- **Traditional viewpoint:** Add *explicit regularization*, e.g., Weight decay, batch/layer normalization, dropout, data argumentation.
- **Modern ML:** A specific algorithm (with a specific initialization) only converges to certain solution, which is called implicit regularization/bias.

- For convex problem, GD with small initialization nearly converges to minimum ℓ_2-norm solution.
- For neural networks, the mechanism of implicit regularization is still puzzling due to the non-convexity.

Implicit regularization

model	# params	random crop	weight decay	train accuracy	test accuracy
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78

Figure 8: Taken from [Chiyuan Zhang, et al, ICLR2017]

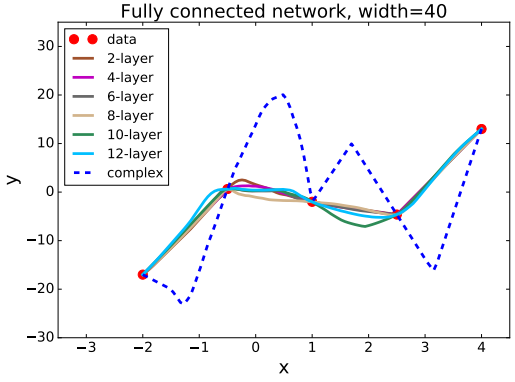


Figure 9: The algorithm is gradient descent (GD). Taken from (Wu, Zhu and E, 2017)

SGD noise

The following table is taken from `https://arxiv.org/pdf/2109.14119.pdf`.

Experiment	Mini-batching	Epochs	Steps	Modifications	Val. Accuracy %
Baseline SGD	✓	300	117,000	-	95.70($\pm$0.11)
Baseline FB	✗	300	300	-	75.42($\pm$0.13)
FB train longer	✗	3000	3000	-	87.36($\pm$1.23)
FB clipped	✗	3000	3000	clip	93.85($\pm$0.10)
FB regularized	✗	3000	3000	clip+reg	95.36($\pm$0.07)
FB strong reg.	✗	3000	3000	clip+reg+bs32	95.67($\pm$0.08)
FB in practice	✗	3000	3000	clip+reg+bs32+shuffle	95.91($\pm$0.14)

Table 2: Summary of validation accuracies in percent on the CIFAR-10 validation dataset for each of the experiments with data augmentations considered in Section 3. All validation accuracies are averaged over 5 runs.

- We can conclude that

$$\text{SGD} > \text{GD}$$

$$\text{SGD} \geq \text{GD} + (\text{sophisticated explicit regularization.})$$

- The SGD noise must impose certain implicit regularization effects. SGD with large LR and small batch size is always preferred.

Flat minima hypothesis (FMP)

The famous **flat minima hypothesis** (Hochreiter and Schmidhuber, 1995; Keskar et al., 2016):

- SGD converges to flatter minima.
- Flatter minima generalize better.

Flat minima hypothesis (FMP)

The famous **flat minima hypothesis** (Hochreiter and Schmidhuber, 1995; Keskar et al., 2016):

- SGD converges to flatter minima.
- Flatter minima generalize better.

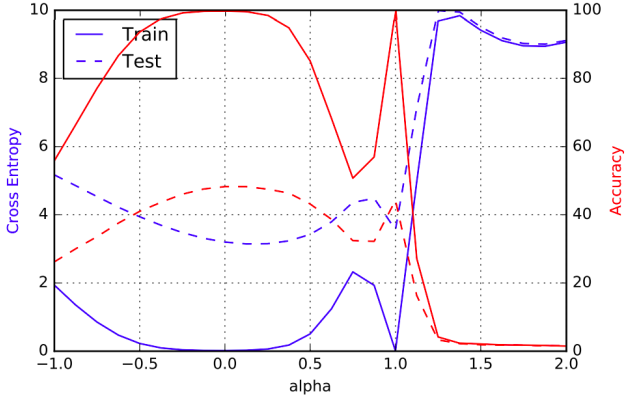


Figure 10: The landscape for for $\theta(\alpha) := (1-\alpha)\theta_{SGD} + \alpha\theta_{GD}$. Taken from (Keskar et al., 2016).

Summary

- BackProp algorithm, Gradient vanishing/exploding phenomenon.
  - Initialization, skip connections, batch normalization.
  - Layer-wise learning rates, Adaptive learning rate methods.
- Explicit regularization: Weight decay, batch normalization, dropout, data augmentation.
- Implicit regularization: SGD and SGD noise.

Reading

- https://www.deeplearningbook.org/contents/optimization.html
- https://www.deeplearningbook.org/contents/regularization.html