

# Neural Networks: Training Algorithms

Instructor: Lei Wu <sup>1</sup>

Mathematical Introduction to Machine Learning

Peking University, Fall 2025

---

<sup>1</sup>School of Mathematical Sciences and Center for Machine Learning Research

# Outline

- **Part I: Computing Gradients Efficiently**

Backpropagation as a scalable implementation of the chain rule.

- **Part II: Why Deep Networks Are Hard to Train**

Gradient vanishing/exploding and architectures/initializations that stabilize training.

- **Part III: Adaptive Optimizers**

Coordinate-wise learning rates: Adagrad, SignGD, RMSProp, Adam.

- **Part IV: Generalization: Explicit and Implicit Regularization**

Explicit regularization (weight decay, dropout, BN, data augmentation) and implicit regularization (SGD noise, flat minima).

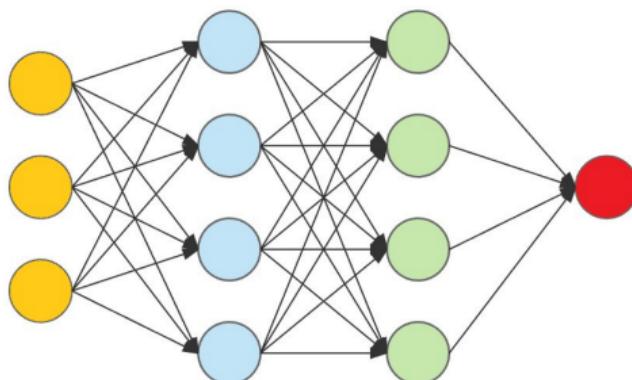
# Forward Propagation

- A deep network defines a sequence of hidden states:

$$x^0 = x, \quad x^\ell = h(x^{\ell-1}; \theta^\ell), \quad \ell = 1, \dots, L,$$

where each  $\theta^\ell$  contains the parameters of layer  $\ell$ , and

$$f(x; \Theta) = x^L, \quad \Theta = (\theta^1, \dots, \theta^L).$$



# Forward Propagation

- A deep network defines a sequence of hidden states:

$$x^0 = x, \quad x^\ell = h(x^{\ell-1}; \theta^\ell), \quad \ell = 1, \dots, L,$$

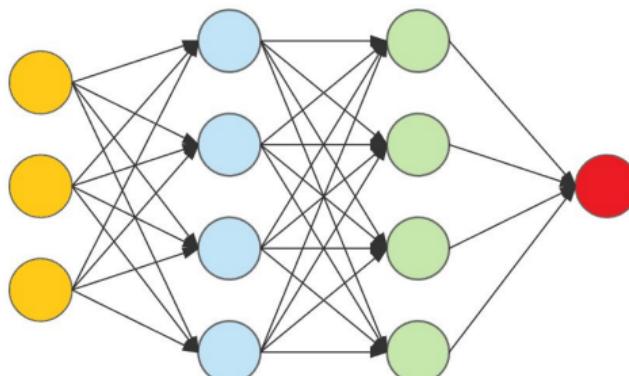
where each  $\theta^\ell$  contains the parameters of layer  $\ell$ , and

$$f(x; \Theta) = x^L, \quad \Theta = (\theta^1, \dots, \theta^L).$$

- For example, in an MLP block

$$h(x^{\ell-1}; \theta^\ell) = \sigma(A^\ell x^{\ell-1} + b^\ell),$$

where  $A^\ell, b^\ell$  are learnable parameters.



# Forward Propagation

- A deep network defines a sequence of hidden states:

$$x^0 = x, \quad x^\ell = h(x^{\ell-1}; \theta^\ell), \quad \ell = 1, \dots, L,$$

where each  $\theta^\ell$  contains the parameters of layer  $\ell$ , and

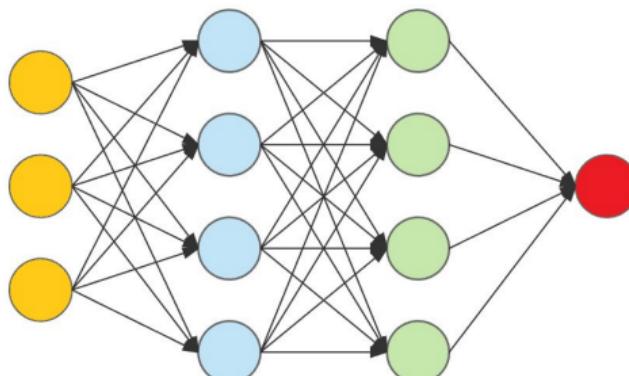
$$f(x; \Theta) = x^L, \quad \Theta = (\theta^1, \dots, \theta^L).$$

- For example, in an MLP block

$$h(x^{\ell-1}; \theta^\ell) = \sigma(A^\ell x^{\ell-1} + b^\ell),$$

where  $A^\ell, b^\ell$  are learnable parameters.

- The forward pass produces all hidden states  $\{x^\ell\}_{\ell=0}^L$ .



# The Problem of Computing Gradients

- For one sample  $(x, y)$ , the loss is

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

# The Problem of Computing Gradients

- For one sample  $(x, y)$ , the loss is

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

- Our goal is to compute all layer-wise gradients

$$\frac{\partial E}{\partial \theta^1}, \frac{\partial E}{\partial \theta^2}, \dots, \frac{\partial E}{\partial \theta^L}.$$

# The Problem of Computing Gradients

- For one sample  $(x, y)$ , the loss is

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

- Our goal is to compute all layer-wise gradients

$$\frac{\partial E}{\partial \theta^1}, \frac{\partial E}{\partial \theta^2}, \dots, \frac{\partial E}{\partial \theta^L}.$$

- Why is this non-trivial?** Because the network is a *deep composition*:

$$f = h^L \circ h^{L-1} \circ \dots \circ h^1,$$

and naïvely applying the chain rule would require recomputing the same intermediate derivatives many times (exponential blow-up).

# The Problem of Computing Gradients

- For one sample  $(x, y)$ , the loss is

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

- Our goal is to compute all layer-wise gradients

$$\frac{\partial E}{\partial \theta^1}, \frac{\partial E}{\partial \theta^2}, \dots, \frac{\partial E}{\partial \theta^L}.$$

- Why is this non-trivial?** Because the network is a *deep composition*:

$$f = h^L \circ h^{L-1} \circ \dots \circ h^1,$$

and naïvely applying the chain rule would require recomputing the same intermediate derivatives many times (exponential blow-up).

- We need a method that reuses intermediate results and computes all gradients in time comparable to one forward pass.

# The Back-propagation Algorithm

- By the chain rule, the gradient of each layer depends on two parts:

$$\frac{\partial E}{\partial \theta^\ell} = \underbrace{\frac{\partial h(x^{\ell-1}; \theta^\ell)}{\partial \theta^\ell}}_{\text{local derivative}} \underbrace{\frac{\partial E}{\partial x^\ell}}_{\text{gradient signal}}.$$

# The Back-propagation Algorithm

- By the chain rule, the gradient of each layer depends on two parts:

$$\frac{\partial E}{\partial \theta^\ell} = \underbrace{\frac{\partial h(x^{\ell-1}; \theta^\ell)}{\partial \theta^\ell}}_{\text{local derivative}} \underbrace{\frac{\partial E}{\partial x^\ell}}_{\text{gradient signal}}.$$

- Define the **gradient signal**

$$\delta^\ell := \frac{\partial E}{\partial x^\ell}.$$

# The Back-propagation Algorithm

- By the chain rule, the gradient of each layer depends on two parts:

$$\frac{\partial E}{\partial \theta^\ell} = \underbrace{\frac{\partial h(x^{\ell-1}; \theta^\ell)}{\partial \theta^\ell}}_{\text{local derivative}} \cdot \underbrace{\frac{\partial E}{\partial x^\ell}}_{\text{gradient signal}}.$$

- Define the **gradient signal**

$$\delta^\ell := \frac{\partial E}{\partial x^\ell}.$$

- Using the chain rule again, the signals satisfy the recursion

$$\delta^{\ell-1} = \underbrace{\frac{\partial h(x^{\ell-1}; \theta^\ell)}{\partial x^{\ell-1}}}_{\text{local Jacobian}} \cdot \delta^\ell, \quad \delta^L = \frac{\partial \ell(y', y)}{\partial y'} \Big|_{y' = x^L}.$$

# The Back-propagation Algorithm

- By the chain rule, the gradient of each layer depends on two parts:

$$\frac{\partial E}{\partial \theta^\ell} = \underbrace{\frac{\partial h(x^{\ell-1}; \theta^\ell)}{\partial \theta^\ell}}_{\text{local derivative}} \cdot \underbrace{\frac{\partial E}{\partial x^\ell}}_{\text{gradient signal}}.$$

- Define the **gradient signal**

$$\delta^\ell := \frac{\partial E}{\partial x^\ell}.$$

- Using the chain rule again, the signals satisfy the recursion

$$\delta^{\ell-1} = \underbrace{\frac{\partial h(x^{\ell-1}; \theta^\ell)}{\partial x^{\ell-1}}}_{\text{local Jacobian}} \cdot \delta^\ell, \quad \delta^L = \frac{\partial \ell(y', y)}{\partial y'} \Big|_{y' = x^L}.$$

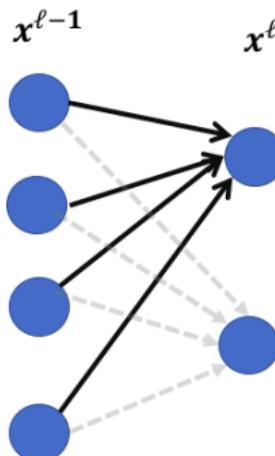
- **Key idea:** All local derivatives only depend on the stored activations  $\{x^\ell\}$ . These were computed during the **forward pass**. This reuse is what makes backprop efficient.

# A Visualization of Backpropagation

## Forward pass

$$z^\ell = A^\ell x^{\ell-1} + b^\ell,$$

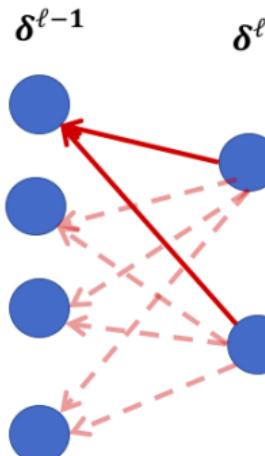
$$x^\ell = \sigma(z^\ell),$$



## Backward pass

$$\delta^\ell = \frac{\partial E}{\partial x^\ell},$$

$$\delta^{\ell-1} = (A^\ell)^\top \delta^\ell \odot \sigma'(z^{\ell-1})$$



Forward computes activations; backward propagates gradient signals.

# 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 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 batch size  $B$ .

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

- 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 batch size  $B$ .

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

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

# Gradient Vanishing and How to Mitigate It

- Activation Function
- Initialization
- Skip Connection
- Normalization
- Gradient Clipping

# 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  $\delta^\ell$  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  $\delta^\ell$  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  $\delta^\ell$  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!!**

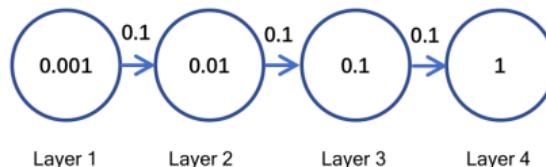
# Gradient Vanishing and Exploding

- 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  $\delta^\ell$  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.

# The Saturation of 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.

# Designing Initialization

- Denote by  $\mathbf{x}^\ell$  the output of  $\ell$ -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?**

# Designing Initialization

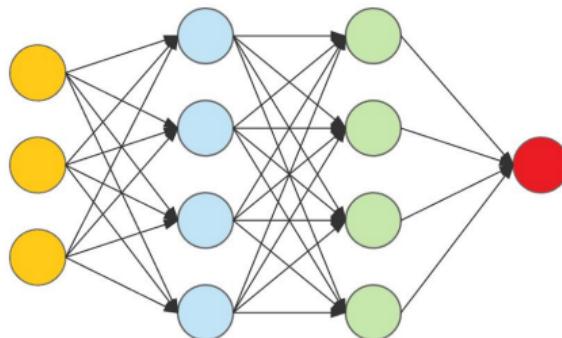
- Denote by  $\mathbf{x}^\ell$  the output of  $\ell$ -th layer:  $\mathbf{x}^{\ell+1} = \sigma(W^\ell \mathbf{x}^\ell + b^\ell)$ .
- Consider the commonly random initialization  $W_{i,j}^\ell \in \mathcal{N}(0, t_w^2)$ ,  $b_j^\ell = 0$  (we will discuss why Gaussian is preferred later) and the standard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



## Derivation

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

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

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

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

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

## Derivation (cont'd)

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

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

- The initialization  $W_{i,j}^\ell \stackrel{iid}{\sim} \mathcal{N}(0, 2/m_\ell)$ ,  $b_j = 0$  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  $\mathbb{E}_{\zeta \sim \mathcal{N}(0,1)}[\sigma^2(\zeta)] = \frac{1}{\sqrt{2\pi}} \int_0^\infty z^2 e^{-z^2/2} dz = 1/2$ . Hence, to ensure  $\mathbb{E}[|x_i^{\ell+1}|^2] = 1$ , we can take

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

- The initialization  $W_{i,j}^\ell \stackrel{iid}{\sim} \mathcal{N}(0, 2/m_\ell)$ ,  $b_j = 0$  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  $\mathbb{E}_{\zeta \sim \mathcal{N}(0,1)}[\sigma^2(\zeta)] = \frac{1}{\sqrt{2\pi}} \int_0^\infty z^2 e^{-z^2/2} dz = 1/2$ . Hence, to ensure  $\mathbb{E}[|x_i^{\ell+1}|^2] = 1$ , we can take

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

- The initialization  $W_{i,j}^\ell \stackrel{iid}{\sim} \mathcal{N}(0, 2/m_\ell)$ ,  $b_j = 0$  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:  $W_{i,j}^\ell \sim \text{Unif}[-t, t]$ , 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**.

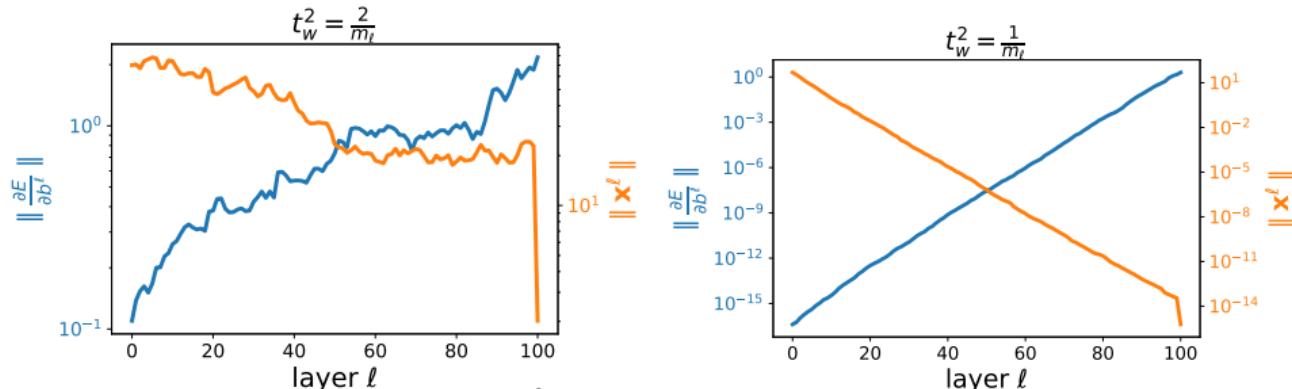


Figure 1: ReLU networks with  $L = 100$ ,  $m^\ell = 200$  for all  $\ell = 1, \dots, L - 1$ . **Left:** The case of  $t_w^2 = 2/m_\ell$  (Kaiming-He initialization); **Right:** The case of  $t_w^2 = 1/m_\ell$  (LeCun initialization).

# Why do We Choose Random Initialization with a Large Support?

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

# Why do We Choose 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

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

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 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

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

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 Random Initialization with a Large Support?

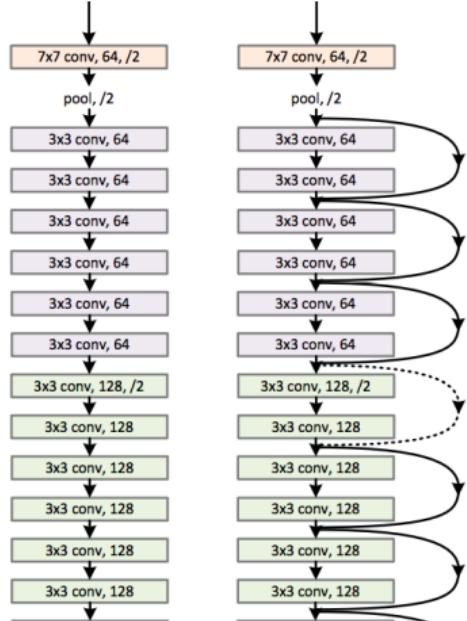
- ① We have an understanding for the size of the initialization.
- ② We do not have an understanding for the directions we need.
  - ① Consider a two-layer neural network

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

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.

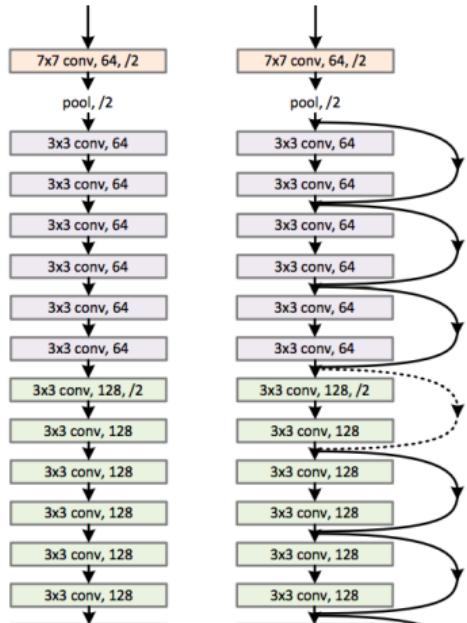
- ② We want 'diverse' initialization with many different vectors in many different directions, but we do not know which directions are important.
- ③ Popular: random initialization with mean zero and appropriate variance.
- ④ We can explore other forms of initialization, e.g., **the orthogonal initialization**: choosing  $W^\ell$  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.

# 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.

# 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** → Highway network → **ResNet**.

# Numerical Evidence

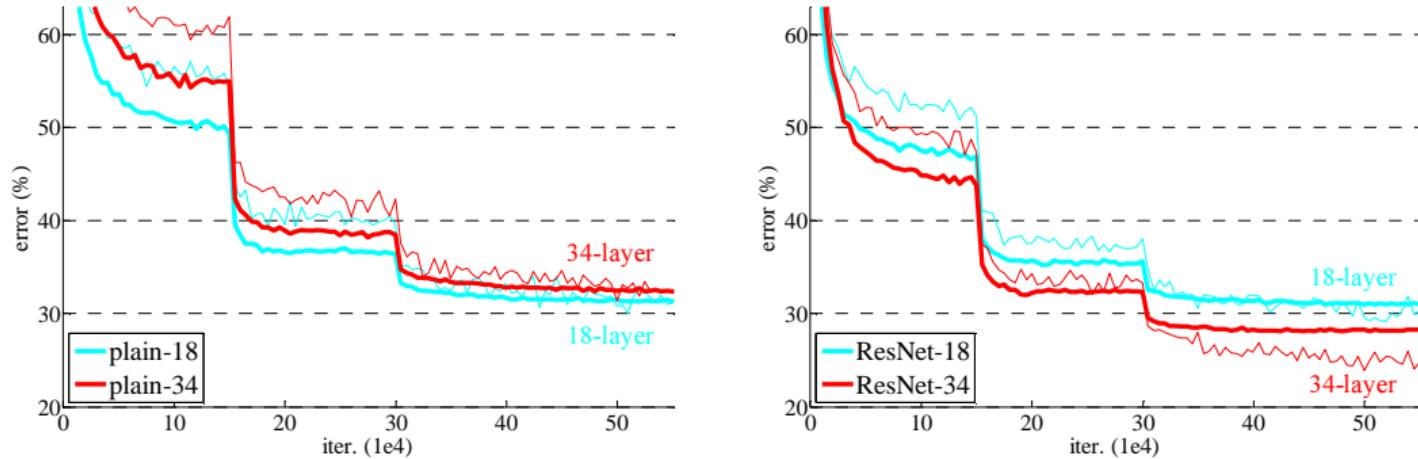


Figure 2: Training on ImageNet. Thin curves denote training error, and bold curves denote validation error. This figure is taken from (Kaiming He et al., 2015).

## Batch Normalization

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

# Batch Normalization

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

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

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

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

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

where  $\gamma, \beta$  are reintroduced to preserve the network's expressivity.

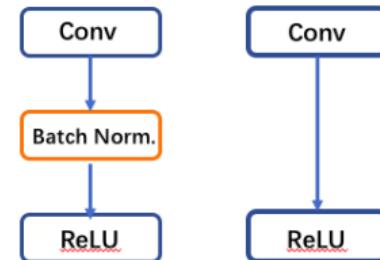


Figure 3: Left: Convolutional nets with BN; Right: Convolutional without BN.

## Performance of BN

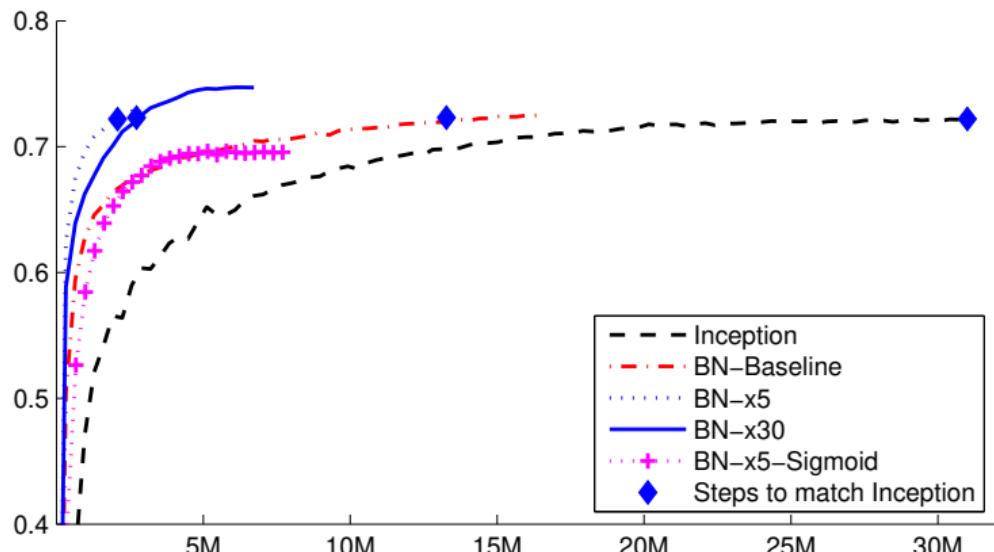


Figure 4: Validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps. BN-baseline: same as inception with BN layers added before each nonlinearity. BN-x5: inception with batch normalization and the learning rate is increased by a factor 5, compared to the baseline.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

# The Issues of 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  $\sigma_B$  and  $\mu_B$  are far away from the  $\sigma$  and  $\mu$ , the ones over the whole dataset.)

# The Issues of 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  $\sigma_B$  and  $\mu_B$  are far away from the  $\sigma$  and  $\mu$ , the ones over the whole dataset.)
- How do we compute  $\sigma_B$  and  $\mu_B$  during the inference, where we may only have one sample?

# The Issues of 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  $\sigma_B$  and  $\mu_B$  are far away from the  $\sigma$  and  $\mu$ , the ones over the whole dataset.)
- How do we compute  $\sigma_B$  and  $\mu_B$  during the inference, where we may only have one sample?

# The Issues of 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  $\sigma_B$  and  $\mu_B$  are far away from the  $\sigma$  and  $\mu$ , the ones over the whole dataset.)
- How do we compute  $\sigma_B$  and  $\mu_B$  during the inference, where we may only have one sample?

Use the following ones obtained from the moving average during the training:

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

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

where  $\sigma_B^t, \mu_B^t$  are the statistics calculated at the  $t$ -th step of training.

# The Issues of 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  $Z = (\mathbf{z}_1, \dots, \mathbf{z}_B)^\top \in \mathbb{R}^{B \times H}$  be our feature map. The first and second dimensions represent the batch and feature dimensions, respectively.<sup>2</sup>
- A layer normalization (LN) layer define a map  $\text{LN}_{\gamma, \beta} : \{\mathbf{z}_1, \dots, \mathbf{z}_B\} \rightarrow \{\tilde{\mathbf{z}}_1, \dots, \tilde{\mathbf{z}}_B\}$  as follows

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

$$\hat{\mathbf{z}}_i \leftarrow \gamma \odot \frac{\mathbf{z}_i - \mu_i}{\sigma_i} + \beta, \quad (5)$$

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

---

<sup>2</sup>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  $Z = (\mathbf{z}_1, \dots, \mathbf{z}_B)^\top \in \mathbb{R}^{B \times H}$  be our feature map. The first and second dimensions represent the batch and feature dimensions, respectively.<sup>2</sup>
- A layer normalization (LN) layer define a map  $\text{LN}_{\gamma, \beta} : \{\mathbf{z}_1, \dots, \mathbf{z}_B\} \rightarrow \{\tilde{\mathbf{z}}_1, \dots, \tilde{\mathbf{z}}_B\}$  as follows

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

$$\hat{\mathbf{z}}_i \leftarrow \gamma \odot \frac{\mathbf{z}_i - \mu_i}{\sigma_i} + \beta, \quad (5)$$

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

- 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?**

---

<sup>2</sup>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.

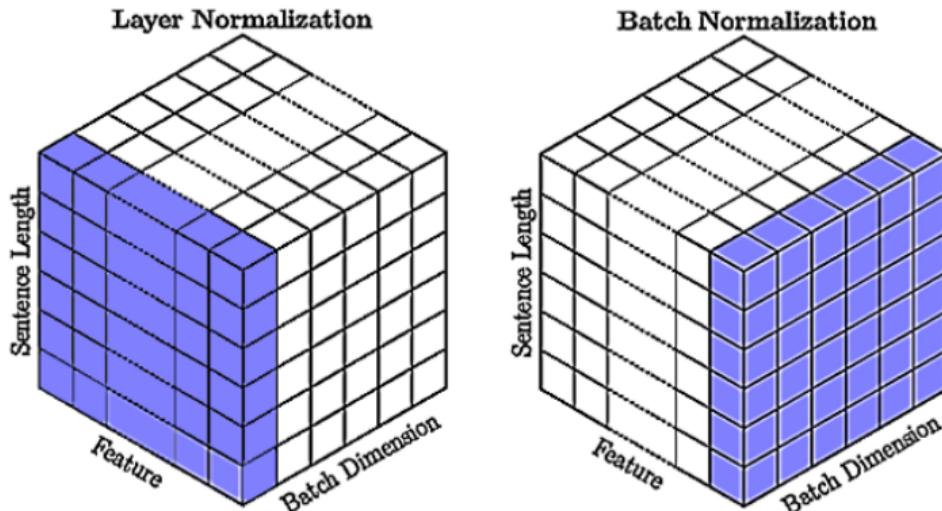


Figure 5: Taken from <https://www.kaggle.com/code/halflingwizard/how-does-layer-normalization-work>.

# Gradient Clipping

- Let  $g_t$  denote the stochastic gradient at step  $t$ .

# Gradient Clipping

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

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

# Gradient Clipping

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

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

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

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

# Gradient Clipping

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

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

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

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

- 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

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

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

- Note that where  $G_t = \sum_{s=0}^t g_s^2$  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

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

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

- Note that where  $G_t = \sum_{s=0}^t g_s^2$  stores the magnitude of each coordinate.
- **Issue:**  $G_t$  is increasing monotonically. Thus, the effective learning rate is decreasing in time.

## Sign Gradient Descent (rProp)

- Another way to handle scale differences is to ignore gradient magnitudes and update using only the **sign**:

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

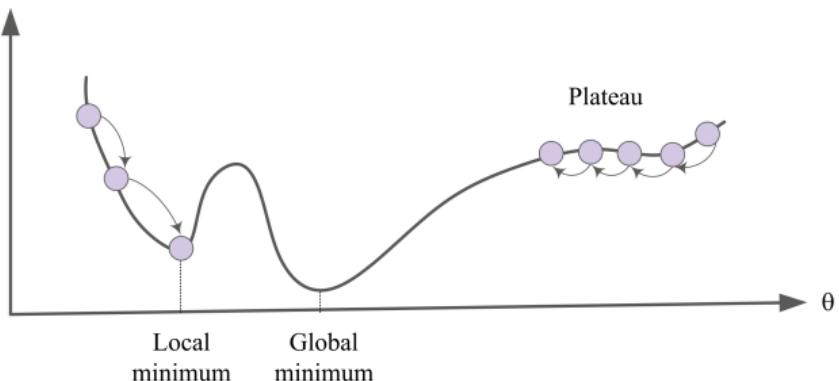
# Sign Gradient Descent (rProp)

- Another way to handle scale differences is to ignore gradient magnitudes and update using only the **sign**:

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

- Advantages:**

- All coordinates take steps of equal size.
- Very effective at escaping flat plateaus with tiny gradients.



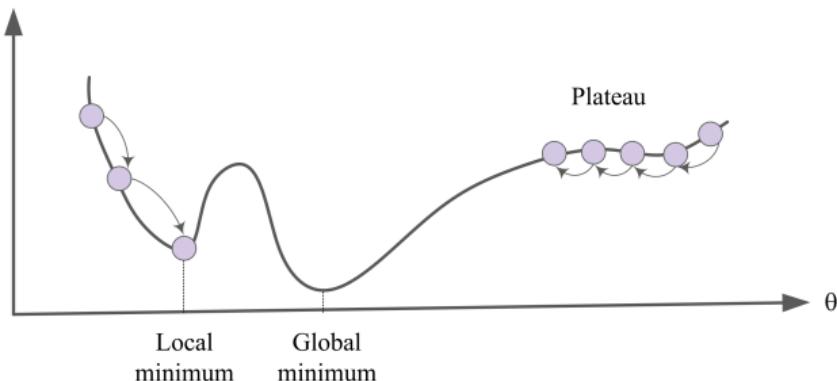
# Sign Gradient Descent (rProp)

- Another way to handle scale differences is to ignore gradient magnitudes and update using only the **sign**:

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

- Advantages:**

- All coordinates take steps of equal size.
- Very effective at escaping flat plateaus with tiny gradients.



- Limitation:** rProp is unstable for stochastic gradients: noise frequently flips signs.
- Question:** Why does gradient noise not severely affect SGD, but greatly harms sign-based methods?

## Sign Momentum

- To make sign-based updates robust to noise, we apply momentum to **reduce the variance** of the stochastic gradients.

## Sign Momentum

- To make sign-based updates robust to noise, we apply momentum to **reduce the variance** of the stochastic gradients.
- The sign-momentum update is

$$m_{t+1} = \beta m_t + (1 - \beta)g_t, \quad x_{t+1} = x_t - \text{sign}(m_{t+1}).$$

## Sign Momentum

- To make sign-based updates robust to noise, we apply momentum to **reduce the variance** of the stochastic gradients.
- The sign-momentum update is

$$m_{t+1} = \beta m_t + (1 - \beta)g_t, \quad x_{t+1} = x_t - \text{sign}(m_{t+1}).$$

- Here  $m_t$  is an exponentially weighted average of past gradients:

$$m_t = \sum_{s=0}^t \beta^{t-s} g_s,$$

which **reduces noise at the cost of a small bias**.

## Sign Momentum

- To make sign-based updates robust to noise, we apply momentum to **reduce the variance** of the stochastic gradients.
- The sign-momentum update is

$$m_{t+1} = \beta m_t + (1 - \beta)g_t, \quad x_{t+1} = x_t - \text{sign}(m_{t+1}).$$

- Here  $m_t$  is an exponentially weighted average of past gradients:

$$m_t = \sum_{s=0}^t \beta^{t-s} g_s,$$

which **reduces noise at the cost of a small bias**.

- This idea—using moments of the gradient—will reappear in RMSProp and Adam.

## RMSProp

- RMSprop (Tieleman & Hinton, 2012) can be viewed as a **stochastic variant of rProp**: it replaces the hard sign update by a *smoothed sign*.

## RMSProp

- RMSprop (Tieleman & Hinton, 2012) can be viewed as a **stochastic variant of rProp**: it replaces the hard sign update by a *smoothed sign*.
- Concretely, RMSProp updates as follows

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

## RMSProp

- RMSprop (Tieleman & Hinton, 2012) can be viewed as a **stochastic variant of rProp**: it replaces the hard sign update by a *smoothed sign*.
- Concretely, RMSProp updates as follows

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

- Heuristically, the factor

$$\frac{g_t}{\sqrt{v_{t+1} + \varepsilon}}$$

behaves like a **smoothed sign**: the numerator gives the direction, while the denominator provides a less noisy estimate of the magnitude ( $\sqrt{v_{t+1}} \approx |g_t|$  in the deterministic case).

## RMSProp

- RMSprop (Tieleman & Hinton, 2012) can be viewed as a **stochastic variant of rProp**: it replaces the hard sign update by a *smoothed sign*.
- Concretely, RMSProp updates as follows

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

- Heuristically, the factor

$$\frac{g_t}{\sqrt{v_{t+1} + \varepsilon}}$$

behaves like a **smoothed sign**: the numerator gives the direction, while the denominator provides a less noisy estimate of the magnitude ( $\sqrt{v_{t+1}} \approx |g_t|$  in the deterministic case).

- When  $\beta = 0$  and  $g_t$  is deterministic, RMSProp reduces to a smooth approximation of signGD / rProp:

$$x_{t+1} = x_t - \eta \frac{\nabla f(x_t)}{\sqrt{|\nabla f(x_t)|^2 + \varepsilon}} \approx x_t - \eta \text{sign}(\nabla f(x_t)),$$

## RMSProp

- RMSprop (Tieleman & Hinton, 2012) can be viewed as a **stochastic variant of rProp**: it replaces the hard sign update by a *smoothed sign*.
- Concretely, RMSProp updates as follows

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

- Heuristically, the factor

$$\frac{g_t}{\sqrt{v_{t+1} + \varepsilon}}$$

behaves like a **smoothed sign**: the numerator gives the direction, while the denominator provides a less noisy estimate of the magnitude ( $\sqrt{v_{t+1}} \approx |g_t|$  in the deterministic case).

- When  $\beta = 0$  and  $g_t$  is deterministic, RMSProp reduces to a smooth approximation of signGD / rProp:

$$x_{t+1} = x_t - \eta \frac{\nabla f(x_t)}{\sqrt{|\nabla f(x_t)|^2 + \varepsilon}} \approx x_t - \eta \text{sign}(\nabla f(x_t)),$$

- In the stochastic setting,  $v_t$  performs a form of **variance reduction**: the normalized update is much less sensitive to instantaneous sign flips caused by gradient noise.

# ADAM<sup>3</sup>

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

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

---

<sup>3</sup>Adaptive momentum: <https://arxiv.org/pdf/1412.6980.pdf>

# ADAM<sup>3</sup>

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

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

- ③ Bias correction (compensates for initialization at zero):

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

---

<sup>3</sup>Adaptive momentum: <https://arxiv.org/pdf/1412.6980.pdf>

# ADAM<sup>3</sup>

- 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 (compensates for initialization at zero):

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

- ④ Update parameters:

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

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

In a summary,

**ADAM=RMSProp + momentum**

---

<sup>3</sup>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 parameters from the original paper:  $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ .
- ④ Adam performs extremely well in deep learning practice, though existing convergence theory does not fully explain its empirical advantages over SGD.

# A Comparison of Different Optimizers

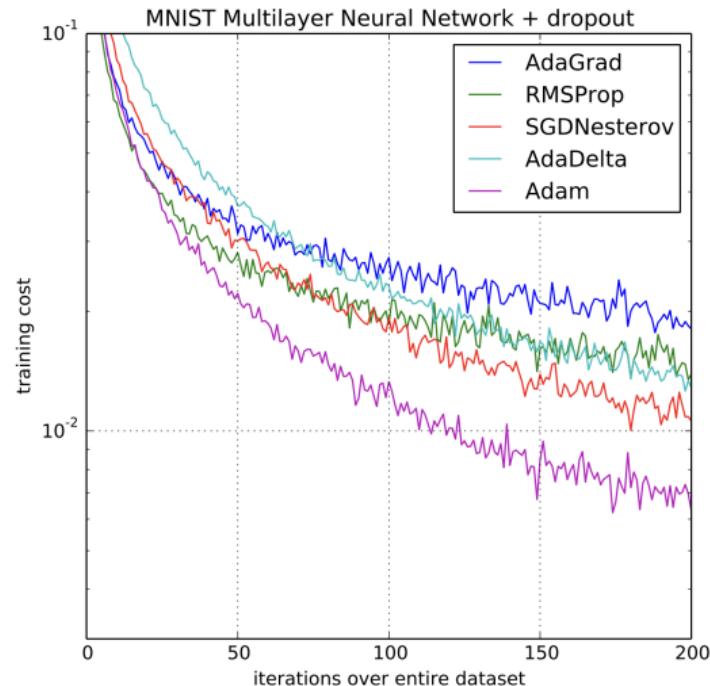


Figure 6: Taken from the original Adam paper <https://arxiv.org/pdf/1412.6980.pdf>.

# 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
<a href="#">Adam: A method for stochastic optimization</a> D Kingma, J Ba International Conference on Learning Representations	200955	2015
<a href="#">Layer normalization</a> 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 vs. L2 Regularization

- Consider an optimizer  $\mathcal{A}$  that updates

$$\theta_{t+1} = \theta_t - \eta h_t,$$

where  $h_t$  is the computed gradient-like update.

## Weight Decay vs. L2 Regularization

- Consider an optimizer  $\mathcal{A}$  that updates

$$\theta_{t+1} = \theta_t - \eta h_t,$$

where  $h_t$  is the computed gradient-like update.

- Weight decay** modifies the update as

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

This explicitly shrinks parameters toward zero.

# Weight Decay vs. L2 Regularization

- Consider an optimizer  $\mathcal{A}$  that updates

$$\theta_{t+1} = \theta_t - \eta h_t,$$

where  $h_t$  is the computed gradient-like update.

- Weight decay** modifies the update as

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

This explicitly shrinks parameters toward zero.

- When  $\mathcal{A} = \text{SGD}$  (i.e.  $h_t = \nabla \hat{\mathcal{R}}(\theta_t)$ ), weight decay is **mathematically equivalent** to adding an  $\ell_2$  regularizer:

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

# Weight Decay vs. L2 Regularization

- Consider an optimizer  $\mathcal{A}$  that updates

$$\theta_{t+1} = \theta_t - \eta h_t,$$

where  $h_t$  is the computed gradient-like update.

- Weight decay** modifies the update as

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

This explicitly shrinks parameters toward zero.

- When  $\mathcal{A} = \text{SGD}$  (i.e.  $h_t = \nabla \hat{\mathcal{R}}(\theta_t)$ ), weight decay is **mathematically equivalent** to adding an  $\ell_2$  regularizer:

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

- Important:** This equivalence holds only for vanilla SGD. For most adaptive optimizers (Adam, RMSProp, Adagrad, ...) weight decay and  $\ell_2$  regularization behave differently.

## Decoupled Weight Decay (AdamW)

- For adaptive methods such as Adam, the standard update is

$$\theta_{t+1} = \theta_t - \eta \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}.$$

## Decoupled Weight Decay (AdamW)

- For adaptive methods such as Adam, the standard update is

$$\theta_{t+1} = \theta_t - \eta \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}.$$

- If we add an  $\ell_2$  penalty, the regularization term  $\lambda\theta_t$  is also scaled by the adaptive denominator, leading to an undesirable interaction.

## Decoupled Weight Decay (AdamW)

- For adaptive methods such as Adam, the standard update is

$$\theta_{t+1} = \theta_t - \eta \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}.$$

- If we add an  $\ell_2$  penalty, the regularization term  $\lambda\theta_t$  is also scaled by the adaptive denominator, leading to an undesirable interaction.
- As shown in Loshchilov & Hutter (2017), **Adam + L2 regularization is not equivalent to weight decay.**

## Decoupled Weight Decay (AdamW)

- For adaptive methods such as Adam, the standard update is

$$\theta_{t+1} = \theta_t - \eta \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}.$$

- If we add an  $\ell_2$  penalty, the regularization term  $\lambda\theta_t$  is also scaled by the adaptive denominator, leading to an undesirable interaction.
- As shown in Loshchilov & Hutter (2017), **Adam + L2 regularization is not equivalent to weight decay.**
- Decoupled weight decay** (AdamW) fixes this by applying shrinkage *outside* the adaptive update:

$$\theta_{t+1} = (1 - \eta\lambda) \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}.$$

Thus the decay term is independent of the adaptive rescaling.

## Decoupled Weight Decay (AdamW)

- For adaptive methods such as Adam, the standard update is

$$\theta_{t+1} = \theta_t - \eta \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}.$$

- If we add an  $\ell_2$  penalty, the regularization term  $\lambda\theta_t$  is also scaled by the adaptive denominator, leading to an undesirable interaction.
- As shown in Loshchilov & Hutter (2017), **Adam + L2 regularization is not equivalent to weight decay**.
- Decoupled weight decay** (AdamW) fixes this by applying shrinkage *outside* the adaptive update:

$$\theta_{t+1} = (1 - \eta\lambda) \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}.$$

Thus the decay term is independent of the adaptive rescaling.

- AdamW is now the **default optimizer for LLMs** (e.g., GPT, PaLM, DeepSeek, LLaMA).

## Decoupled Weight Decay (AdamW)

- For adaptive methods such as Adam, the standard update is

$$\theta_{t+1} = \theta_t - \eta \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}.$$

- If we add an  $\ell_2$  penalty, the regularization term  $\lambda\theta_t$  is also scaled by the adaptive denominator, leading to an undesirable interaction.
- As shown in Loshchilov & Hutter (2017), **Adam + L2 regularization is not equivalent to weight decay**.
- Decoupled weight decay** (AdamW) fixes this by applying shrinkage *outside* the adaptive update:

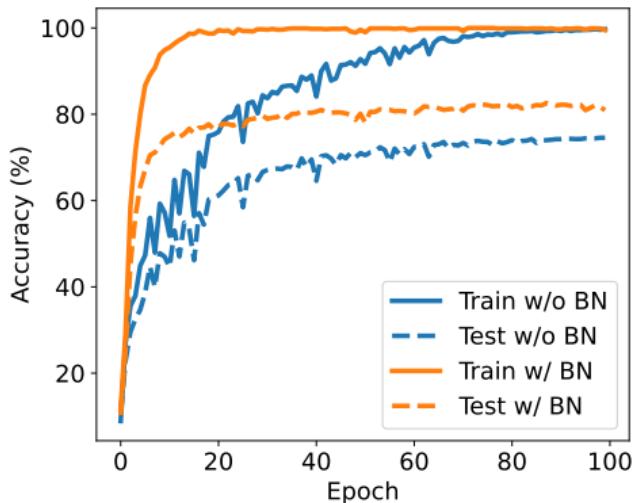
$$\theta_{t+1} = (1 - \eta\lambda) \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}.$$

Thus the decay term is independent of the adaptive rescaling.

- AdamW is now the **default optimizer for LLMs** (e.g., GPT, PaLM, DeepSeek, LLaMA).
- Why weight decay improves generalization/optimization in large models remains poorly understood.** (Open research topic)

# 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.

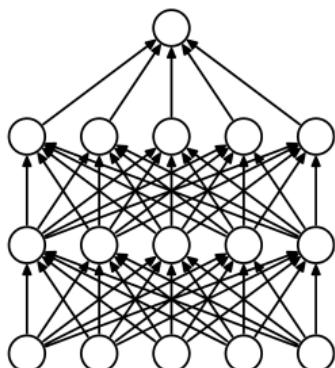


# Dropout: Basic Definition

## Vanilla forward-prop

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

$$x_i^\ell = \sigma(z_i^\ell)$$



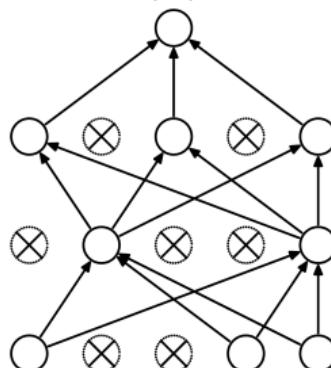
(a) Standard Neural Net

## Dropout forward-prop

$$r_i^{\ell-1} \stackrel{iid}{\sim} \text{Bernoulli}(p)$$

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

$$x_i^\ell = \sigma(z_i^\ell)$$



(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  $\xi$  denotes the dropping mask.
- Denote by  $\pi$  the distribution of the mask  $\xi$ . Then the dropout training goes as follows,

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

which is exactly SGD of batch size 1 for minimizing

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

## Dropout: The Ensemble Viewpoint

- Let the effective dropout model be

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

# Dropout: The Ensemble Viewpoint

- Let the effective dropout model be

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

- Let  $\hat{\mathcal{R}}(h) = \frac{1}{n} \sum_{i=1}^n (h(x_i) - y_i)^2$  be the training risk. Then,

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

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

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

# Dropout: The Ensemble Viewpoint

- Let the effective dropout model be

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

- Let  $\hat{\mathcal{R}}(h) = \frac{1}{n} \sum_{i=1}^n (h(x_i) - y_i)^2$  be the training risk. Then,

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

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

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

# Dropout: The Ensemble Viewpoint

- Let the effective dropout model be

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

- Let  $\hat{\mathcal{R}}(h) = \frac{1}{n} \sum_{i=1}^n (h(x_i) - y_i)^2$  be the training risk. Then,

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

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

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

# Dropout: The Ensemble Viewpoint

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

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

which means that the effective model is an average/ensemble of sparse sub-networks.

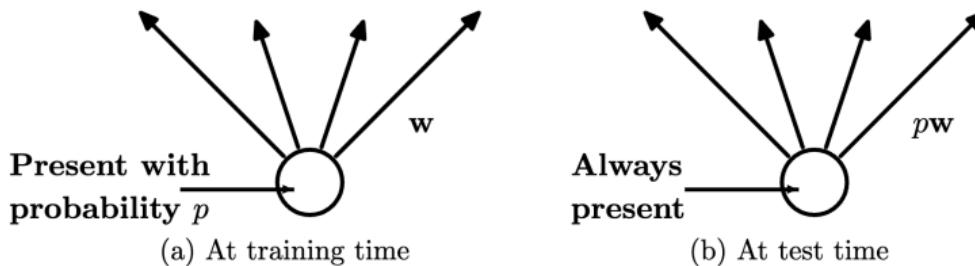
- At training, stochastic approximation is applied. At testing, we can use Monte-Carlo approximation:

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

# The Mean-Field Approximation

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

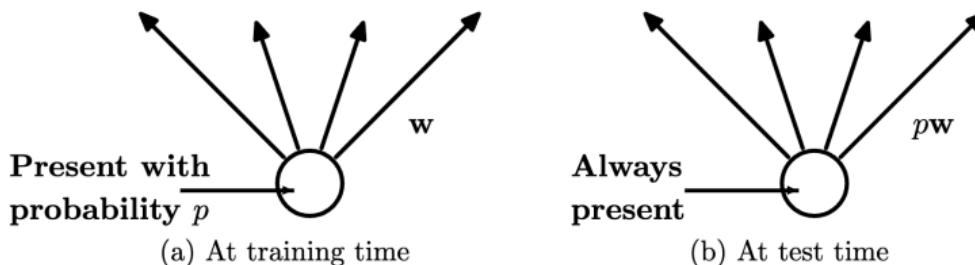
$$\mathbb{E}_\xi[F(x; \xi, \theta)] \approx F(x; \mathbb{E}_\xi[\xi], \theta) = f(x; \mathbb{E}[\xi] \odot \theta) = f(x; p\theta).$$



# The Mean-Field Approximation

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

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



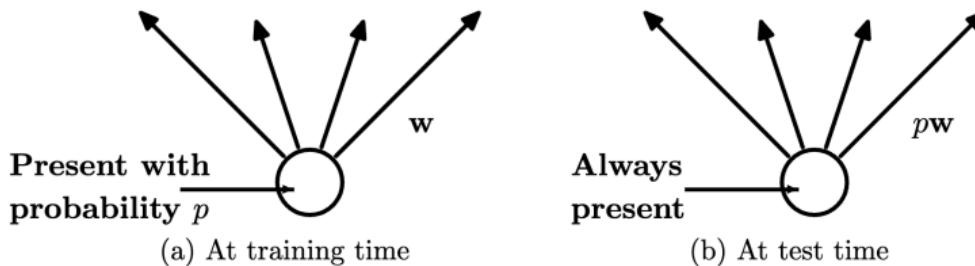
- The error of mean-field approximation.** Let  $\mu = \mathbb{E}[\xi]$ . For a general  $h \in C^2$ ,

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

# The Mean-Field Approximation

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

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



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

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

- 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.

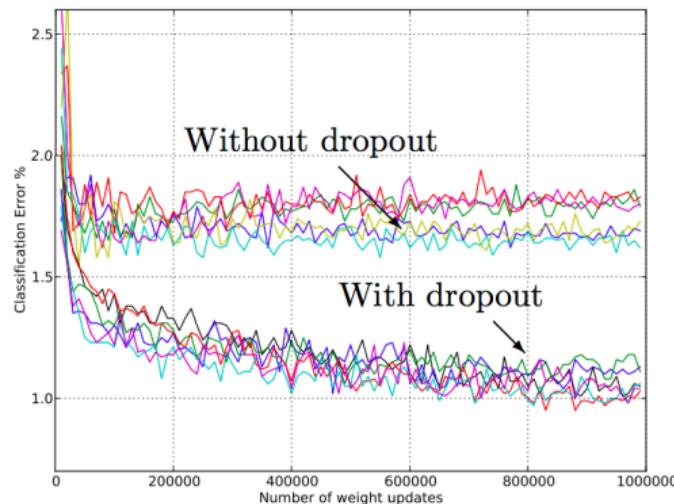
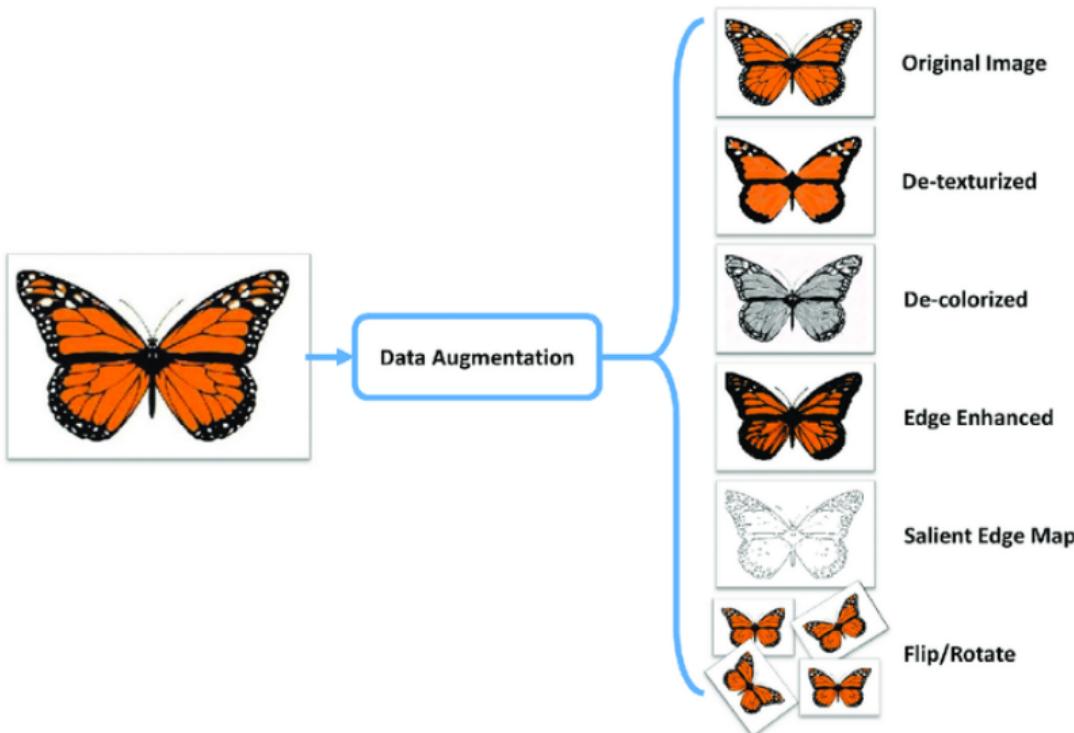


Figure 7: Taken from Dropout: A Simple Way to Prevent Neural Networks from Overfitting

# 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  $\ell_2$ -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 (fitting random labels)	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
		no	no	100.0	9.78

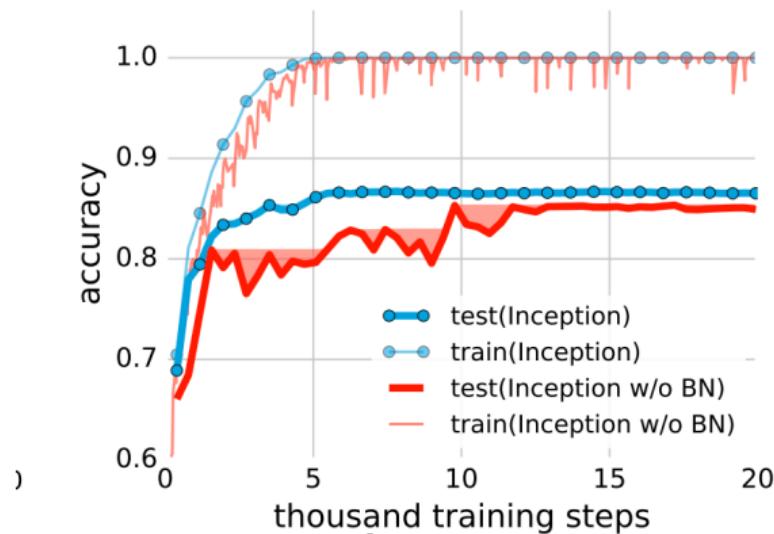


Figure 8: Taken from [Chiyuan Zhang, et al, ICLR2017]

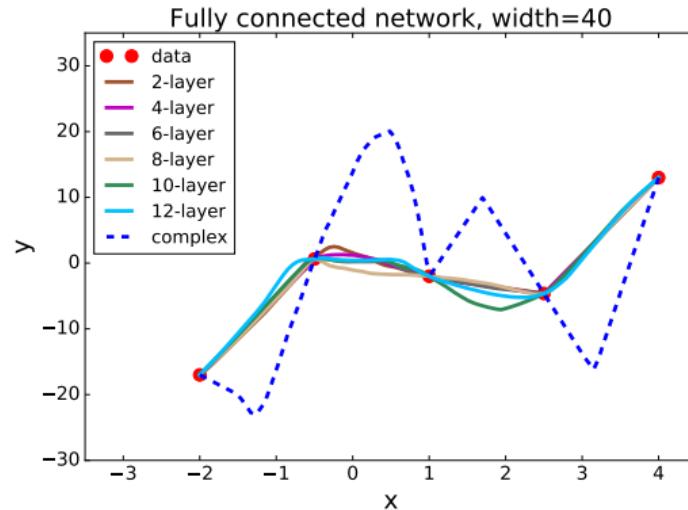


Figure 9: The algorithm is gradient descent (GD). Taken from (Wu, Zhu and E, 2017)

# 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.

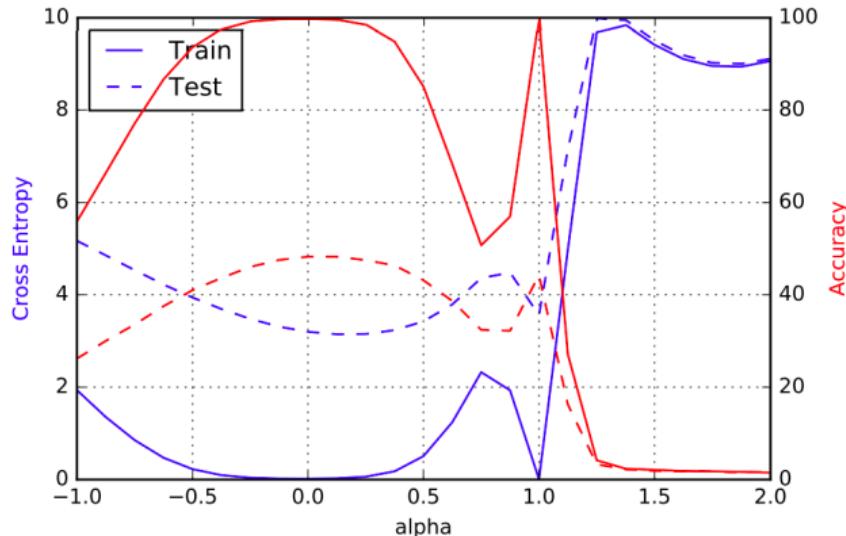


Figure 10: The landscape for  $\theta(\alpha) := (1 - \alpha)\theta_{\text{SGD}} + \alpha\theta_{\text{GD}}$ . Taken from (Keskar et al., 2016).

# 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>