# Neural Networks: Optimization & Regularization

Shan-Hung Wu
*shwu@cs.nthu.edu.tw*

Department of Computer Science,
National Tsing Hua University, Taiwan

Machine Learning

# Outline

1. **Optimization**
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. **Regularization**
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Outline

1. **Optimization**
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. **Regularization**
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Challenges

- NN a complex function:

$$
\begin{aligned}
\hat{\boldsymbol{y}} &= f(\boldsymbol{x};\Theta) \\
&= f^{(L)}(\cdots f^{(1)}(\boldsymbol{x};\boldsymbol{W}^{(1)});\boldsymbol{W}^{(L)})
\end{aligned}
$$

# Challenges

- NN a complex function:

$$\begin{aligned}\hat{\boldsymbol{y}} &= f(\boldsymbol{x};\Theta) \\ &= f^{(L)}(\cdots f^{(1)}(\boldsymbol{x};\boldsymbol{W}^{(1)});\boldsymbol{W}^{(L)})\end{aligned}$$

- Given a training set $\mathbb{X}$, our goal is to solve:

$$\begin{aligned}\arg\min_{\Theta} C(\Theta) &= \arg\min_{\Theta} -\log \mathrm{P}(\mathbb{X}\,|\,\Theta) \\ &= \arg\min_{\Theta} \textstyle\sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)},\Theta) \\ &= \arg\min_{\Theta} \textstyle\sum_i C^{(i)}(\Theta) \\ &= \arg\min_{\boldsymbol{W}^{(1)},\cdots,\boldsymbol{W}^{(L)}} \textstyle\sum_i C^{(i)}(\boldsymbol{W}^{(1)},\cdots,\boldsymbol{W}^{(L)})\end{aligned}$$

# Challenges

- NN a complex function:

$$\begin{aligned}\hat{\boldsymbol{y}} &= f(\boldsymbol{x};\Theta)\\ &= f^{(L)}(\cdots f^{(1)}(\boldsymbol{x};\boldsymbol{W}^{(1)});\boldsymbol{W}^{(L)})\end{aligned}$$

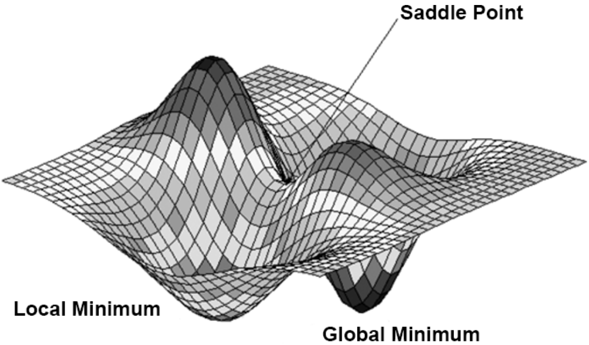
- Given a training set $\mathbb{X}$, our goal is to solve:

$$\begin{aligned}\arg\min_{\Theta} C(\Theta) &= \arg\min_{\Theta} -\log \mathrm{P}(\mathbb{X}\,|\,\Theta)\\ &= \arg\min_{\Theta} \textstyle\sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)},\Theta)\\ &= \arg\min_{\Theta} \textstyle\sum_i C^{(i)}(\Theta)\\ &= \arg\min_{\boldsymbol{W}^{(1)},\cdots,\boldsymbol{W}^{(L)}} \textstyle\sum_i C^{(i)}(\boldsymbol{W}^{(1)},\cdots,\boldsymbol{W}^{(L)})\end{aligned}$$

- What are the challenges of solving this problem with SGD?

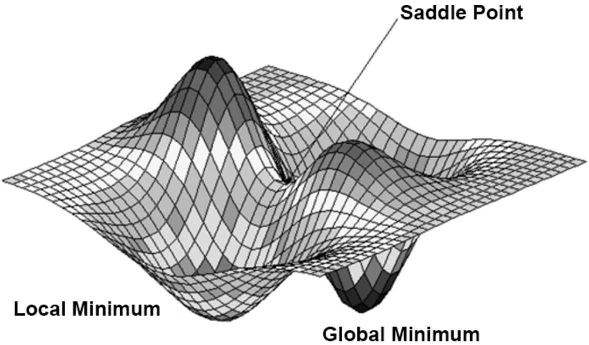
# Non-Convexity

- The loss function $C^{(i)}$ is ***non-convex***



# Non-Convexity

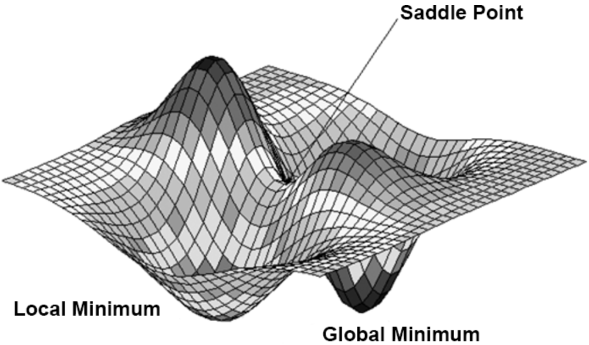
- The loss function $C^{(i)}$ is ***non-convex***




- SGD stops at local minima or saddle points

# Non-Convexity

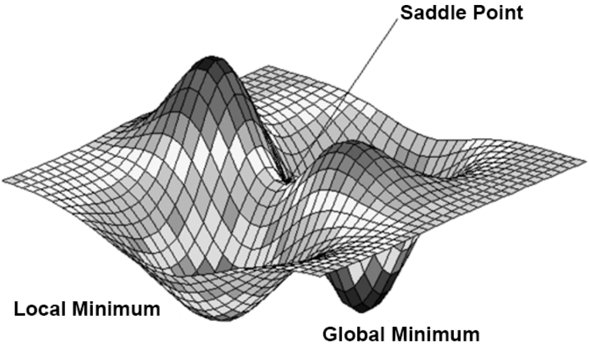
- The loss function $C^{(i)}$ is ***non-convex***




- SGD stops at local minima or saddle points
- Prior to the success of SGD (in roughly 2012), NN cost function surfaces were generally believed to have many non-convex structure

# Non-Convexity

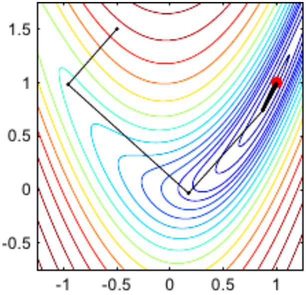
- The loss function $C^{(i)}$ is ***non-convex***

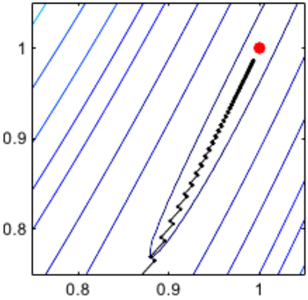



- SGD stops at local minima or saddle points
- Prior to the success of SGD (in roughly 2012), NN cost function surfaces were generally believed to have many non-convex structure
- However, studies [2, 4] show SGD seldom encounters critical points when training a large NN

# Ill-Conditioning

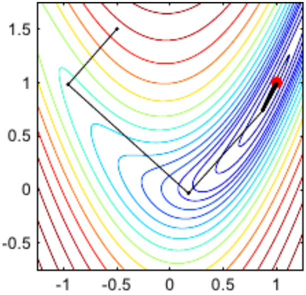
- The loss $C^{(i)}$ may be ill-conditioned (in terms of $\Theta$)
  - Due to, e.g., dependency between $W^{(k)}$'s at different layers

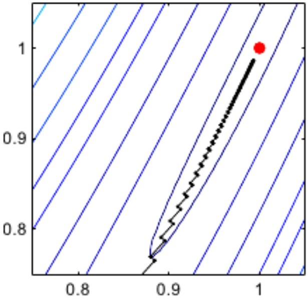


# Ill-Conditioning

- The loss $C^{(i)}$ may be ill-conditioned (in terms of $\Theta$)
  - Due to, e.g., dependency between $W^{(k)}$'s at different layers




- SGD has slow progress at valleys or plateaus

# Lacks Global Minima

- The loss $C^{(i)}$ may lack a global minimum point

# Lacks Global Minima

- The loss $C^{(i)}$ may lack a global minimum point
- E.g., for multiclass classification
  - $\mathrm{P}(\mathbf{y}\,|\,\boldsymbol{x},\Theta)$ provided by a softmax function

# Lacks Global Minima

- The loss $C^{(i)}$ may lack a global minimum point
- E.g., for multiclass classification
  - $\mathrm{P}(\mathbf{y}\,|\,\boldsymbol{x},\Theta)$ provided by a softmax function
  - $C^{(i)}(\Theta) = -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)},\Theta)$ can become arbitrarily close to zero (if classifying example $i$ correctly)

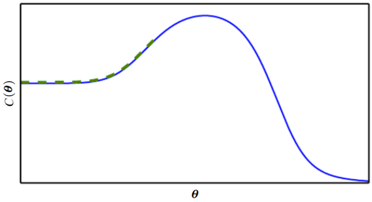
# Lacks Global Minima

- The loss $C^{(i)}$ may lack a global minimum point
- E.g., for multiclass classification
  - $\mathrm{P}(\mathbf{y}\,|\,\boldsymbol{x},\Theta)$ provided by a softmax function
  - $C^{(i)}(\Theta) = -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)},\Theta)$ can become arbitrarily close to zero (if classifying example $i$ correctly)
  - But ***not*** actually reaching zero

# Lacks Global Minima

- The loss $C^{(i)}$ may lack a global minimum point
- E.g., for multiclass classification
  - $\mathrm{P}(\mathbf{y}\,|\,\boldsymbol{x},\Theta)$ provided by a softmax function
  - $C^{(i)}(\Theta) = -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)},\Theta)$ can become arbitrarily close to zero (if classifying example $i$ correctly)
  - But ***not*** actually reaching zero

- SGD may proceed along a direction forever
- ***Initialization*** is important



# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input

# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input
    - Prevents dominating features
    - Improves conditioning

# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input
  - Prevents dominating features
  - Improves conditioning
- When training, remember to:

1. Initialize all weights to small ***random values***
   - Breaks "symmetry" between different units so they are not updated in the same way

# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input
  - Prevents dominating features
  - Improves conditioning
- When training, remember to:

1. Initialize all weights to small ***random values***
   - Breaks "symmetry" between different units so they are not updated in the same way
   - Biases $b^{(k)}$'s may be initialized to zero

# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input
  - Prevents dominating features
  - Improves conditioning
- When training, remember to:

1. Initialize all weights to small ***random values***
   - Breaks "symmetry" between different units so they are not updated in the same way
   - Biases $b^{(k)}$'s may be initialized to zero (or to small positive values for ReLUs to prevent too much saturation)

# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input
  - Prevents dominating features
  - Improves conditioning
- When training, remember to:

1. Initialize all weights to small ***random values***
   - Breaks "symmetry" between different units so they are not updated in the same way
   - Biases $b^{(k)}$'s may be initialized to zero (or to small positive values for ReLUs to prevent too much saturation)
2. ***Early stop*** if the validation error does not continue decreasing

# Training 101

- Before training a feedforward NN, remember to ***standardize*** ($z$-normalize) the input
  - Prevents dominating features
  - Improves conditioning
- When training, remember to:

1. Initialize all weights to small ***random values***
   - Breaks "symmetry" between different units so they are not updated in the same way
   - Biases $b^{(k)}$'s may be initialized to zero (or to small positive values for ReLUs to prevent too much saturation)
2. ***Early stop*** if the validation error does not continue decreasing
   - Prevents overfitting

# Outline

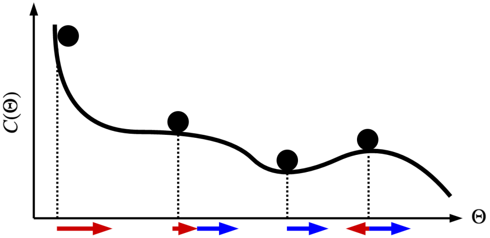
1. **Optimization**
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. **Regularization**
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Momentum

- Update rule in SGD:

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$$

where $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$



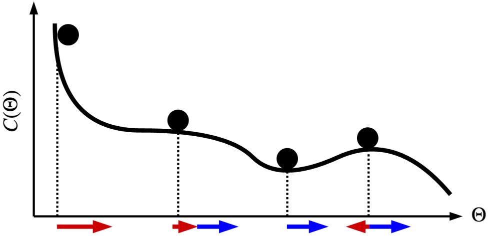
# Momentum

- Update rule in SGD:

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$$

where $\boldsymbol{g}^{(t)} = \nabla_{\Theta} C(\Theta^{(t)})$

  - Gets stuck in local minima or saddle points

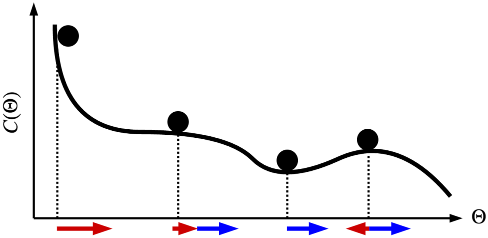


# Momentum

- Update rule in SGD:

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$$

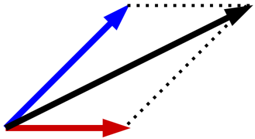
where $\boldsymbol{g}^{(t)} = \nabla_{\Theta} C(\Theta^{(t)})$
  - Gets stuck in local minima or saddle points




- Momentum: make the same movement $\boldsymbol{v}^{(t)}$ in the last iteration, corrected by negative gradient:

$$\boldsymbol{v}^{(t+1)} \leftarrow \lambda \boldsymbol{v}^{(t)} - (1-\lambda)\boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t+1)}$$




- $\boldsymbol{v}^{(t)}$ is a moving average of $-\boldsymbol{g}^{(t)}$

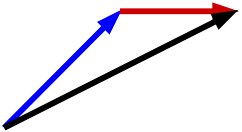
# Nesterov Momentum

- Make the same movement $\boldsymbol{v}^{(t)}$ in the last iteration, corrected by ***lookahead*** negative gradient:

$$\tilde{\Theta}^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t)}$$

$$\boldsymbol{v}^{(t+1)} \leftarrow \lambda \boldsymbol{v}^{(t)} - (1-\lambda)\nabla_{\Theta} C(\tilde{\Theta}^{(t)})$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t+1)}$$



# Nesterov Momentum

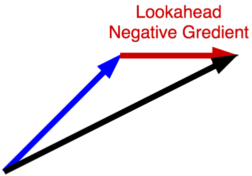
- Make the same movement $\boldsymbol{v}^{(t)}$ in the last iteration, corrected by ***lookahead*** negative gradient:

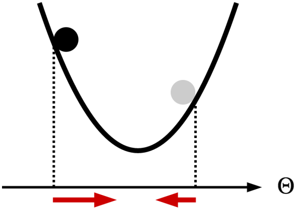
$$\tilde{\Theta}^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t)}$$

$$\boldsymbol{v}^{(t+1)} \leftarrow \lambda \boldsymbol{v}^{(t)} - (1-\lambda)\nabla_{\Theta} C(\tilde{\Theta}^{(t)})$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t+1)}$$

- Faster convergence to a minimum



# Nesterov Momentum

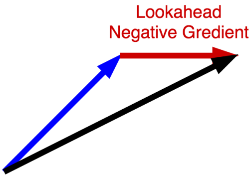
- Make the same movement $\boldsymbol{v}^{(t)}$ in the last iteration, corrected by ***lookahead*** negative gradient:

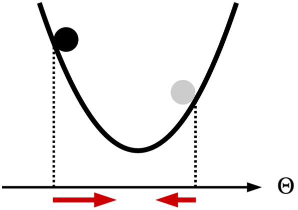
$$\tilde{\Theta}^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t)}$$

$$\boldsymbol{v}^{(t+1)} \leftarrow \lambda \boldsymbol{v}^{(t)} - (1-\lambda)\nabla_{\Theta} C(\tilde{\Theta}^{(t)})$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t+1)}$$

- Faster convergence to a minimum
- Not helpful for NNs that lack of minima

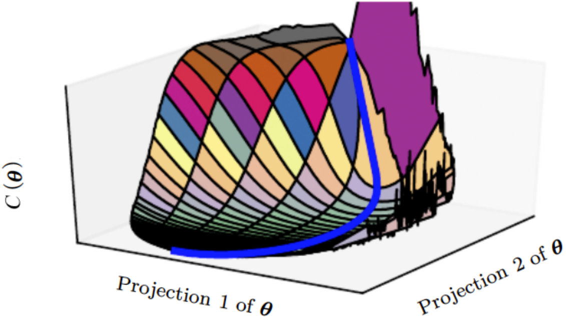


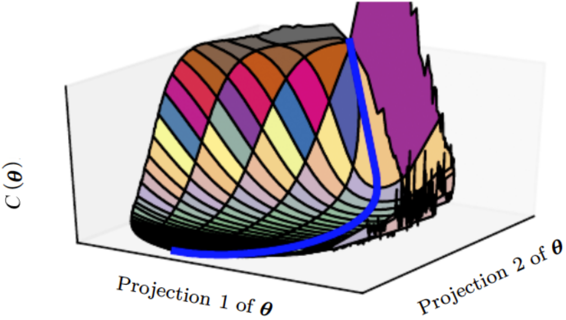
# Outline

1. **Optimization**
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. **Regularization**
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Where Does SGD Spend Its Training Time?

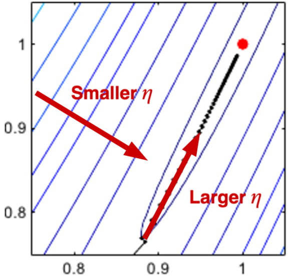


# Where Does SGD Spend Its Training Time?



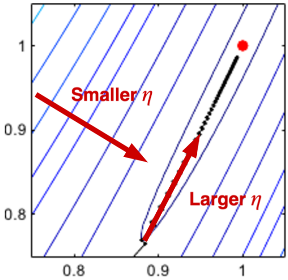

1. Detouring a saddle point of high cost
   - Better initialization
2. Traversing the relatively flat valley
   - Adaptive learning rate

# SGD with Adaptive Learning Rates



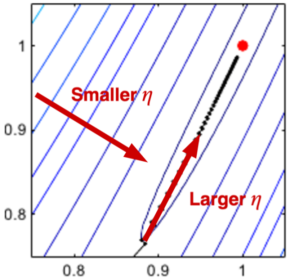

- Smaller learning rate $\eta$ along a steep direction
  - Prevents overshooting

# SGD with Adaptive Learning Rates




- Smaller learning rate $\eta$ along a steep direction
  - Prevents overshooting
- Larger learning rate $\eta$ along a flat direction
  - Speed up convergence

# SGD with Adaptive Learning Rates




- Smaller learning rate $\eta$ along a steep direction
  - Prevents overshooting
- Larger learning rate $\eta$ along a flat direction
  - Speed up convergence
- How?

# AdaGrad

- Update rule:

$$\boldsymbol{r}^{(t+1)} \leftarrow \boldsymbol{r}^{(t)} + \boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

# AdaGrad

- Update rule:

$$\boldsymbol{r}^{(t+1)} \leftarrow \boldsymbol{r}^{(t)} + \boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

  - $\boldsymbol{r}^{(t+1)}$ accumulates squared gradients along each axis

# AdaGrad

- Update rule:

$$\boldsymbol{r}^{(t+1)} \leftarrow \boldsymbol{r}^{(t)} + \boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

  - $\boldsymbol{r}^{(t+1)}$ accumulates squared gradients along each axis
  - Division and square root applied to $\boldsymbol{r}^{(t+1)}$ elementwisely

- We have

$$\frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} = \frac{\eta}{\sqrt{t+1}} \odot \frac{1}{\sqrt{\frac{1}{t+1}\boldsymbol{r}^{(t+1)}}} = \frac{\eta}{\sqrt{t+1}} \odot \frac{1}{\sqrt{\frac{1}{t+1}\sum_{i=0}^{t}\boldsymbol{g}^{(i)} \odot \boldsymbol{g}^{(i)}}}$$

# AdaGrad

- Update rule:

$$\boldsymbol{r}^{(t+1)} \leftarrow \boldsymbol{r}^{(t)} + \boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

  - $\boldsymbol{r}^{(t+1)}$ accumulates squared gradients along each axis
  - Division and square root applied to $\boldsymbol{r}^{(t+1)}$ elementwisely
- We have

$$\frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} = \frac{\eta}{\sqrt{t+1}} \odot \frac{1}{\sqrt{\frac{1}{t+1}\boldsymbol{r}^{(t+1)}}} = \frac{\eta}{\sqrt{t+1}} \odot \frac{1}{\sqrt{\frac{1}{t+1}\sum_{i=0}^{t}\boldsymbol{g}^{(i)} \odot \boldsymbol{g}^{(i)}}}$$

1. ***Smaller learning rate along all directions as $t$ grows***

# AdaGrad

- Update rule:

$$\boldsymbol{r}^{(t+1)} \leftarrow \boldsymbol{r}^{(t)} + \boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

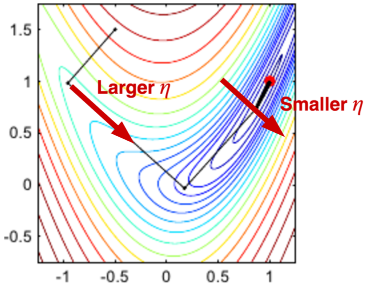
  - $\boldsymbol{r}^{(t+1)}$ accumulates squared gradients along each axis
  - Division and square root applied to $\boldsymbol{r}^{(t+1)}$ elementwisely

- We have

$$\frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} = \frac{\eta}{\sqrt{t+1}} \odot \frac{1}{\sqrt{\frac{1}{t+1}\boldsymbol{r}^{(t+1)}}} = \frac{\eta}{\sqrt{t+1}} \odot \frac{1}{\sqrt{\frac{1}{t+1}\sum_{i=0}^{t} \boldsymbol{g}^{(i)} \odot \boldsymbol{g}^{(i)}}}$$

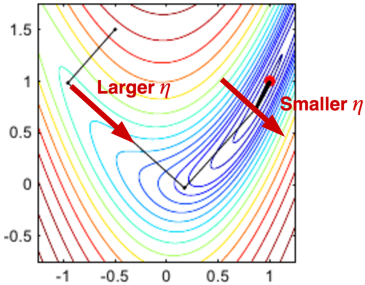
1. ***Smaller learning rate along all directions as $t$ grows***
2. ***Larger learning rate along more gently sloped directions***

# Limitations




- The optimal learning rate along a direction may ***change over time***

# Limitations




- The optimal learning rate along a direction may ***change over time***
- In AdaGrad, $\boldsymbol{r}^{(t+1)}$ accumulates squared gradients ***from the beginning of training***
  - Results in premature adaptivity

# RMSProp

- ***RMSProp*** changes the gradient accumulation in $\boldsymbol{r}^{(t+1)}$ into a moving average:

$$\boldsymbol{r}^{(t+1)} \leftarrow \lambda \boldsymbol{r}^{(t)} + (1-\lambda)\boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

# RMSProp

- ***RMSProp*** changes the gradient accumulation in $\boldsymbol{r}^{(t+1)}$ into a moving average:

$$\boldsymbol{r}^{(t+1)} \leftarrow \lambda \boldsymbol{r}^{(t)} + (1-\lambda)\boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{g}^{(t)}$$

- A popular algorithm ***Adam*** (short for ***adaptive moments***) [7] is a combination of RMSProp and Momentum:

$$\boldsymbol{v}^{(t+1)} \leftarrow \lambda_1 \boldsymbol{v}^{(t)} - (1-\lambda_1)\boldsymbol{g}^{(t)}$$

$$\boldsymbol{r}^{(t+1)} \leftarrow \lambda_2 \boldsymbol{r}^{(t)} + (1-\lambda_2)\boldsymbol{g}^{(t)} \odot \boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \frac{\eta}{\sqrt{\boldsymbol{r}^{(t+1)}}} \odot \boldsymbol{v}^{(t+1)}$$

  - With some bias corrections for $\boldsymbol{v}^{(t+1)}$ and $\boldsymbol{r}^{(t+1)}$

# Outline

# Training Deep NNs I

- So far, we modify the optimization algorithm to better train the model

# Training Deep NNs I

- So far, we modify the optimization algorithm to better train the model
- Can we modify the model to ease the optimization task?

# Training Deep NNs I

- So far, we modify the optimization algorithm to better train the model
- Can we modify the model to ease the optimization task?
- What are the difficulties in training a deep NN?

# Training Deep NNs II

- The cost $C(\Theta)$ of a deep NN is usually ill-conditioned due to the dependency between $\boldsymbol{W}^{(k)}$'s at different layers

# Training Deep NNs II

- The cost $C(\Theta)$ of a deep NN is usually ill-conditioned due to the dependency between $\boldsymbol{W}^{(k)}$'s at different layers
- As a simple example, consider a deep NN for $x, y \in \mathbb{R}$:

$$\hat{y} = f(x) = xw^{(1)}w^{(2)}\cdots w^{(L)}$$

  - Single unit at each layer
  - Linear activation function and no bias in each unit

# Training Deep NNs II

- The cost $C(\Theta)$ of a deep NN is usually ill-conditioned due to the dependency between $\boldsymbol{W}^{(k)}$'s at different layers
- As a simple example, consider a deep NN for $x, y \in \mathbb{R}$:

$$\hat{y} = f(x) = xw^{(1)}w^{(2)} \cdots w^{(L)}$$

  - Single unit at each layer
  - Linear activation function and no bias in each unit
- The output $\hat{y}$ is a linear function of $x$, but ***not*** of weights

# Training Deep NNs II

- The cost $C(\Theta)$ of a deep NN is usually ill-conditioned due to the dependency between $\boldsymbol{W}^{(k)}$'s at different layers
- As a simple example, consider a deep NN for $x, y \in \mathbb{R}$:

$$\hat{y} = f(x) = xw^{(1)}w^{(2)}\cdots w^{(L)}$$

  - Single unit at each layer
  - Linear activation function and no bias in each unit
- The output $\hat{y}$ is a linear function of $x$, but ***not*** of weights
- The curvature of $f$ with respect to any two $w^{(i)}$ and $w^{(j)}$ is

$$\frac{\partial f}{\partial w^{(i)} \partial w^{(j)}} = (w^{(i)} + w^{(j)}) \cdot x \prod_{k \neq i,j} w^{(k)}$$

  - Very small if $L$ is large and $w^{(k)} < 1$ for $k \neq i,j$
  - Very large if $L$ is large and $w^{(k)} > 1$ for $k \neq i,j$

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
    - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
  - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)
  - However, $\boldsymbol{g}^{(t)}$ updates $\Theta^{(t)}$ in all dimensions ***simultaneously*** in the same iteration

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
  - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)
  - However, $\boldsymbol{g}^{(t)}$ updates $\Theta^{(t)}$ in all dimensions ***simultaneously*** in the same iteration
  - $C(\Theta^{(t+1)})$ will be guaranteed to decrease only if $C$ is linear at $\Theta^{(t)}$

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
  - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)
  - However, $\boldsymbol{g}^{(t)}$ updates $\Theta^{(t)}$ in all dimensions ***simultaneously*** in the same iteration
  - $C(\Theta^{(t+1)})$ will be guaranteed to decrease only if $C$ is linear at $\Theta^{(t)}$
- Wrong assumption: $\Theta_i^{(t+1)}$ will decrease $C$ even if other $\Theta_j^{(t+1)}$'s are updated simultaneously

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
    - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)
    - However, $\boldsymbol{g}^{(t)}$ updates $\Theta^{(t)}$ in all dimensions ***simultaneously*** in the same iteration
    - $C(\Theta^{(t+1)})$ will be guaranteed to decrease only if $C$ is linear at $\Theta^{(t)}$
- Wrong assumption: $\Theta_i^{(t+1)}$ will decrease $C$ even if other $\Theta_j^{(t+1)}$'s are updated simultaneously
- Second-order methods?

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
  - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)
  - However, $\boldsymbol{g}^{(t)}$ updates $\Theta^{(t)}$ in all dimensions ***simultaneously*** in the same iteration
  - $C(\Theta^{(t+1)})$ will be guaranteed to decrease only if $C$ is linear at $\Theta^{(t)}$
- Wrong assumption: $\Theta_i^{(t+1)}$ will decrease $C$ even if other $\Theta_j^{(t+1)}$'s are updated simultaneously
- Second-order methods?
  - Time consuming
  - Does not take into account high-order effects

# Training Deep NNs III

- The ill-conditioned $C(\Theta)$ makes a gradient-based optimization algorithm (e.g., SGD) inefficient
- Let $\Theta = [w^{(1)}, w^{(2)}, \cdots, w^{(L)}]^\top$ and $\boldsymbol{g}^{(t)} = \nabla_\Theta C(\Theta^{(t)})$
- In gradient descent, we get $\Theta^{(t+1)}$ by $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$ based on the first-order Taylor approximation of $C$
    - The gradient $\boldsymbol{g}_i^{(t)} = \frac{\partial C}{\partial w^{(i)}}(\Theta^{(t)})$ is calculated ***individually*** by fixing $C(\Theta^{(t)})$ in other dimensions ($w^{(j)}$'s, $j \neq i$)
    - However, $\boldsymbol{g}^{(t)}$ updates $\Theta^{(t)}$ in all dimensions ***simultaneously*** in the same iteration
    - $C(\Theta^{(t+1)})$ will be guaranteed to decrease only if $C$ is linear at $\Theta^{(t)}$
- Wrong assumption: $\Theta_i^{(t+1)}$ will decrease $C$ even if other $\Theta_j^{(t+1)}$'s are updated simultaneously
- Second-order methods?
    - Time consuming
    - Does not take into account high-order effects
- Can we change the model to make this assumption not-so-wrong?

# Batch Normalization I

$$\hat{y} = f(x) = xw^{(1)}w^{(2)}\cdots w^{(L)}$$

- Why not standardize each hidden activation $a^{(k)}$, $k = 1, \cdots, L-1$ (as we standardized $x$)?

# Batch Normalization I

$$\hat{y} = f(x) = xw^{(1)}w^{(2)} \cdots w^{(L)}$$

- Why not standardize each hidden activation $a^{(k)}$, $k = 1, \cdots, L-1$ (as we standardized $x$)?
- We have
$$\hat{y} = a^{(L-1)}w^{(L)}$$
- When $a^{(L-1)}$ is standardized, $\boldsymbol{g}_L^{(t)} = \frac{\partial C}{\partial w^{(L)}}(\Theta^{(t)})$ is more likely to decrease $C$

# Batch Normalization I

$$\hat{y} = f(x) = xw^{(1)}w^{(2)}\cdots w^{(L)}$$

- Why not standardize each hidden activation $a^{(k)}$, $k = 1, \cdots, L-1$ (as we standardized $x$)?
- We have
$$\hat{y} = a^{(L-1)}w^{(L)}$$
- When $a^{(L-1)}$ is standardized, $\boldsymbol{g}_L^{(t)} = \frac{\partial C}{\partial w^{(L)}}(\Theta^{(t)})$ is more likely to decrease $C$
  - If $x \sim \mathcal{N}(0,1)$, then still $a^{(L-1)} \sim \mathcal{N}(0,1)$, no matter how $w^{(1)}, \cdots, w^{(L-1)}$ change

# Batch Normalization I

$$\hat{y} = f(x) = xw^{(1)}w^{(2)}\cdots w^{(L)}$$

- Why not standardize each hidden activation $a^{(k)}$, $k = 1, \cdots, L-1$ (as we standardized $x$)?
- We have
$$\hat{y} = a^{(L-1)}w^{(L)}$$
- When $a^{(L-1)}$ is standardized, $\boldsymbol{g}_L^{(t)} = \frac{\partial C}{\partial w^{(L)}}(\Theta^{(t)})$ is more likely to decrease $C$
  - If $x \sim \mathcal{N}(0,1)$, then still $a^{(L-1)} \sim \mathcal{N}(0,1)$, no matter how $w^{(1)}, \cdots, w^{(L-1)}$ change
  - Changes in other dimensions proposed by $\boldsymbol{g}_i^{(t)}$'s, $i \neq L$, can be zeroed out

# Batch Normalization I

$$\hat{y} = f(x) = x w^{(1)} w^{(2)} \cdots w^{(L)}$$

- Why not standardize each hidden activation $a^{(k)}$, $k = 1, \cdots, L-1$ (as we standardized $x$)?
- We have
$$\hat{y} = a^{(L-1)} w^{(L)}$$
- When $a^{(L-1)}$ is standardized, $\boldsymbol{g}_L^{(t)} = \frac{\partial C}{\partial w^{(L)}}(\Theta^{(t)})$ is more likely to decrease $C$
  - If $x \sim \mathcal{N}(0,1)$, then still $a^{(L-1)} \sim \mathcal{N}(0,1)$, no matter how $w^{(1)}, \cdots, w^{(L-1)}$ change
  - Changes in other dimensions proposed by $\boldsymbol{g}_i^{(t)}$'s, $i \neq L$, can be zeroed out
- Similarly, if $a^{(k-1)}$ is standardized, $\boldsymbol{g}_k^{(t)} = \frac{\partial C}{\partial w^{(k)}}(\Theta^{(t)})$ is more likely to decrease $C$

# Batch Normalization II

- How to standardize $a^{(k)}$ at training and test time?
  - We can standardize the input $x$ because we see multiple examples

# Batch Normalization II

- How to standardize $a^{(k)}$ at training and test time?
  - We can standardize the input $x$ because we see multiple examples
- During training time, we see a minibatch of activations $\boldsymbol{a}^{(k)} \in \mathbb{R}^M$ ($M$ the batch size)
- ***Batch normalization*** [6]:

$$\tilde{a}_i^{(k)} = \frac{a_i^{(k)} - \mu^{(k)}}{\sigma^{(k)}}, \forall i$$

  - $\mu^{(k)}$ and $\sigma^{(k)}$ are mean and std of activations across examples in the minibatch

# Batch Normalization II

- How to standardize $a^{(k)}$ at training and test time?
  - We can standardize the input $x$ because we see multiple examples
- During training time, we see a minibatch of activations $\boldsymbol{a}^{(k)} \in \mathbb{R}^M$ ($M$ the batch size)
- *Batch normalization* [6]:

$$\tilde{a}_i^{(k)} = \frac{a_i^{(k)} - \mu^{(k)}}{\sigma^{(k)}}, \forall i$$

  - $\mu^{(k)}$ and $\sigma^{(k)}$ are mean and std of activations across examples in the minibatch
- At test time, $\mu^{(k)}$ and $\sigma^{(k)}$ can be replaced by running averages that were collected during training time

# Batch Normalization II

- How to standardize $a^{(k)}$ at training and test time?
  - We can standardize the input $x$ because we see multiple examples
- During training time, we see a minibatch of activations $\boldsymbol{a}^{(k)} \in \mathbb{R}^M$ ($M$ the batch size)
- ***Batch normalization*** [6]:

$$\tilde{a}_i^{(k)} = \frac{a_i^{(k)} - \mu^{(k)}}{\sigma^{(k)}}, \forall i$$

  - $\mu^{(k)}$ and $\sigma^{(k)}$ are mean and std of activations across examples in the minibatch
- At test time, $\mu^{(k)}$ and $\sigma^{(k)}$ can be replaced by running averages that were collected during training time
- Can be readily extended to NNs having multiple neurons at each layer

# Standardizing Nonlinear Units

- How to standardize a nonlinear unit $a^{(k)} = \mathrm{act}(z^{(k)})$?

# Standardizing Nonlinear Units

- How to standardize a nonlinear unit $a^{(k)} = \mathrm{act}(z^{(k)})$?
- We can still zero out the effects from other layers by normalizing $z^{(k)}$

# Standardizing Nonlinear Units

- How to standardize a nonlinear unit $a^{(k)} = \mathrm{act}(z^{(k)})$?
- We can still zero out the effects from other layers by normalizing $z^{(k)}$
- Given a minibatch of $\boldsymbol{z}^{(k)} \in \mathbb{R}^M$:

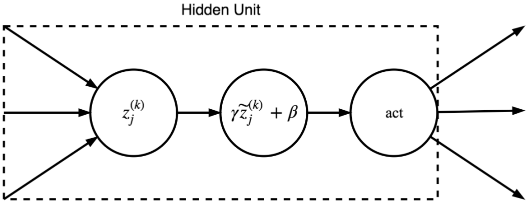
$$\tilde{z}_i^{(k)} = \frac{z_i^{(k)} - \mu^{(k)}}{\sigma^{(k)}}, \forall i$$

# Standardizing Nonlinear Units

- How to standardize a nonlinear unit $a^{(k)} = \text{act}(z^{(k)})$?
- We can still zero out the effects from other layers by normalizing $z^{(k)}$
- Given a minibatch of $\boldsymbol{z}^{(k)} \in \mathbb{R}^M$:

$$\tilde{z}_i^{(k)} = \frac{z_i^{(k)} - \mu^{(k)}}{\sigma^{(k)}}, \forall i$$

- A hidden unit now looks like:



# Expressiveness I

- The weights $\boldsymbol{W}^{(k)}$ at each layer is easier to train now
  - The "wrong assumption" of gradient-based optimization is made valid

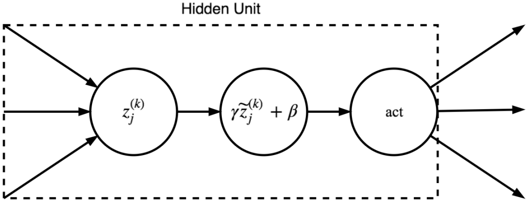
# Expressiveness I

- The weights $\boldsymbol{W}^{(k)}$ at each layer is easier to train now
    - The "wrong assumption" of gradient-based optimization is made valid
- But at the cost of expressiveness
    - Normalizing $a^{(k)}$ or $z^{(k)}$ limits the output range of a unit

# Expressiveness I

- The weights $\boldsymbol{W}^{(k)}$ at each layer is easier to train now
  - The "wrong assumption" of gradient-based optimization is made valid
- But at the cost of expressiveness
  - Normalizing $a^{(k)}$ or $z^{(k)}$ limits the output range of a unit
- Observe that there is no need to insist a $\tilde{z}^{(k)}$ to have zero mean and unit variance
  - We only care about whether it is "fixed" when calculating the gradients for other layers

# Expressiveness II

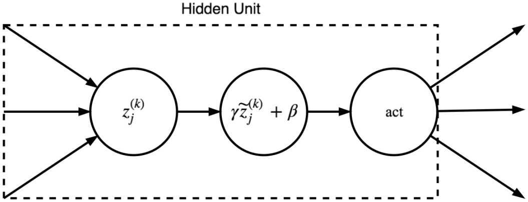



- During training time, we can introduce two parameters $\gamma$ and $\beta$ and ***back-propagate through***

$$\gamma\tilde{z}^{(k)} + \beta$$

to learn their best values

# Expressiveness II

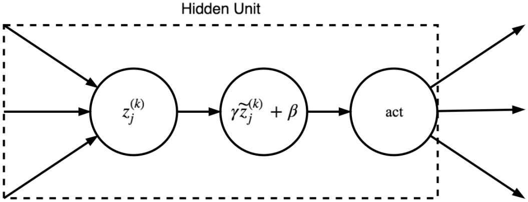



- During training time, we can introduce two parameters $\gamma$ and $\beta$ and ***back-propagate through***

$$\gamma\tilde{z}^{(k)} + \beta$$

  to learn their best values
- Question: $\gamma$ and $\beta$ can be learned to invert $\tilde{z}^{(k)}$ to get $z^{(k)}$, so what's the point?
  - $\tilde{z}^{(k)} = \frac{z^{(k)} - \mu^{(k)}}{\sigma^{(k)}}$, so $\gamma\tilde{z}^{(k)} + \beta = \sigma\tilde{z}^{(k)} + \mu = z^{(k)}$

# Expressiveness II




- During training time, we can introduce two parameters $\gamma$ and $\beta$ and ***back-propagate through***

$$\gamma\tilde{z}^{(k)} + \beta$$

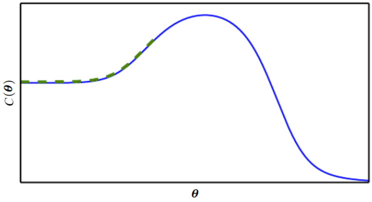
  to learn their best values
- Question: $\gamma$ and $\beta$ can be learned to invert $\tilde{z}^{(k)}$ to get $z^{(k)}$, so what's the point?
  - $\tilde{z}^{(k)} = \frac{z^{(k)} - \mu^{(k)}}{\sigma^{(k)}}$, so $\gamma\tilde{z}^{(k)} + \beta = \sigma\tilde{z}^{(k)} + \mu = z^{(k)}$
  - The weights $\boldsymbol{W}^{(k)}$, $\gamma$, and $\beta$ are now easier to learn with SGD

# Outline

1. **Optimization**
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. **Regularization**
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

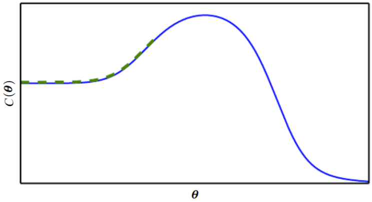
# Parameter Initialization

- *Initialization* is important



# Parameter Initialization

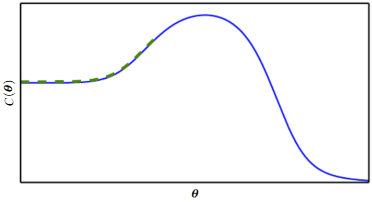
- *Initialization* is important




- How to better initialize $\Theta^{(0)}$?

# Parameter Initialization

- *Initialization* is important

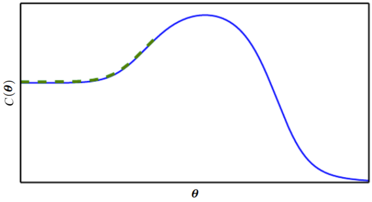



- How to better initialize $\Theta^{(0)}$?

1. Train an NN multiple times with random initial points, and then pick the best

# Parameter Initialization

- ***Initialization*** is important




- How to better initialize $\Theta^{(0)}$?

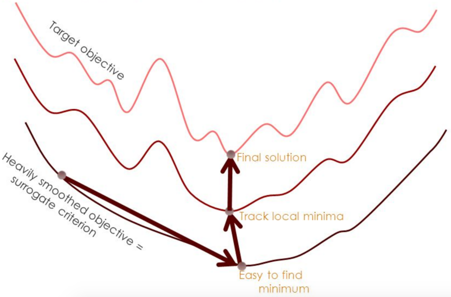
1. Train an NN multiple times with random initial points, and then pick the best
2. Design a series of cost functions such that a solution to one is a good initial point of the next
   - Solve the "easy" problem first, and then a "harder" one, and so on

# Continuation Methods I

- ***Continuation methods***: construct easier cost functions by ***smoothing*** the original cost function:

$$\tilde{C}(\Theta) = \mathrm{E}_{\tilde{\Theta} \sim \mathcal{N}(\Theta, \sigma^2)} C(\tilde{\Theta})$$

  - In practice, we sample several $\tilde{\Theta}$'s to approximate the expectation
- Assumption: some non-convex functions become approximately convex when smoothen



# Continuation Methods II

- Problems?

# Continuation Methods II

- Problems?
- Cost function might not become convex, no matter how much it is smoothen

# Continuation Methods II

- Problems?
- Cost function might not become convex, no matter how much it is smoothen
- Designed to deal with local minima; not very helpful for NNs without minima

# Curriculum Learning

- ***Curriculum learning*** (or ***shaping***) [1]: make the cost function easier by increasing the influence of ***simpler examples***
  - E.g., by assigning them larger weights in the new cost function
  - Or, by sampling them more frequently

# Curriculum Learning

- ***Curriculum learning*** (or ***shaping***) [1]: make the cost function easier by increasing the influence of ***simpler examples***
  - E.g., by assigning them larger weights in the new cost function
  - Or, by sampling them more frequently
- How to define "simple" examples?

# Curriculum Learning

- ***Curriculum learning*** (or ***shaping***) [1]: make the cost function easier by increasing the influence of ***simpler examples***
  - E.g., by assigning them larger weights in the new cost function
  - Or, by sampling them more frequently
- How to define "simple" examples?
  - Face image recognition: front view (easy) vs. side view (hard)
  - Sentiment analysis for movie reviews: 0-/5-star reviews (easy) vs. 1-/2-/3-/4-star reviews (hard)

# Curriculum Learning

- ***Curriculum learning*** (or ***shaping***) [1]: make the cost function easier by increasing the influence of ***simpler examples***
  - E.g., by assigning them larger weights in the new cost function
  - Or, by sampling them more frequently
- How to define "simple" examples?
  - Face image recognition: front view (easy) vs. side view (hard)
  - Sentiment analysis for movie reviews: 0-/5-star reviews (easy) vs. 1-/2-/3-/4-star reviews (hard)
- Learn simple concepts first, then learn more complex concepts that depend on these simpler concepts

# Curriculum Learning

- ***Curriculum learning*** (or ***shaping***) [1]: make the cost function easier by increasing the influence of ***simpler examples***
  - E.g., by assigning them larger weights in the new cost function
  - Or, by sampling them more frequently
- How to define "simple" examples?
  - Face image recognition: front view (easy) vs. side view (hard)
  - Sentiment analysis for movie reviews: 0-/5-star reviews (easy) vs. 1-/2-/3-/4-star reviews (hard)
- Learn simple concepts first, then learn more complex concepts that depend on these simpler concepts
  - Just like how humans learn
  - Knowing the principles, we are less likely to explain an observation using special (but wrong) rules

# Outline

1. **Optimization**
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. **Regularization**
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Prior Predictions of NTK-GP

- Prior (unconditioned) mean predictions for training set:

$$\hat{\boldsymbol{y}}_N = (\boldsymbol{I} - e^{-\eta \boldsymbol{T}_{N,N} t})\boldsymbol{y}$$

- Prior mean predictions for test set:

$$\hat{\boldsymbol{y}}_M = \boldsymbol{T}_{M,N} \boldsymbol{T}_{N,N}^{-1} (\boldsymbol{I} - e^{-\eta \boldsymbol{T}_{N,N} t})\boldsymbol{y}$$

- Given a training set, the $\boldsymbol{T}_{N,N}$ and $\boldsymbol{T}_{M,N}$ depends only on the network structure and hyperparameters of initial weights

# Trainability

- Prior (unconditioned) mean predictions for training set:

$$\hat{\boldsymbol{y}}_N = (\boldsymbol{I} - e^{-\eta \boldsymbol{T}_{N,N} t})\boldsymbol{y}$$

  - where $\eta \le O(1/\lambda_{\max})$
- Goal: $\hat{\boldsymbol{y}}_N \to \boldsymbol{y}$ as $t \to \infty$

# Trainability

- Prior (unconditioned) mean predictions for training set:

$$\hat{\boldsymbol{y}}_N = (\boldsymbol{I} - e^{-\eta \boldsymbol{T}_{N,N} t})\boldsymbol{y}$$

  - where $\eta \leq O(1/\lambda_{\max})$
- Goal: $\hat{\boldsymbol{y}}_N \to \boldsymbol{y}$ as $t \to \infty$
- Let $\boldsymbol{T}_{N,N} = \boldsymbol{U}^\top \begin{bmatrix} \lambda_{\max} & & \\ & \ddots & \\ & & \lambda_{\min} \end{bmatrix} \boldsymbol{U}$, we have

$$(\boldsymbol{U}\hat{\boldsymbol{y}}_N)_i \leq ((\boldsymbol{I} - e^{-\frac{\lambda_i}{\lambda_{\max}} t})\boldsymbol{U}\boldsymbol{y})_i$$

- It follows that if ***the conditioning number*** $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ ***diverges***, the NN becomes untrainable

# Generalization

- Prior mean predictions for test set: $\hat{\boldsymbol{y}}_M = \boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}(\boldsymbol{I} - e^{-\eta \boldsymbol{T}_{N,N} t})\boldsymbol{y}$
- As $t \to \infty$ (trained), we have

$$\hat{\boldsymbol{y}}_M = \boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}\boldsymbol{y}$$

- Goal: the values of $\hat{\boldsymbol{y}}_M$ depend on data $\boldsymbol{X}_M$, $\boldsymbol{X}_N$, and $\boldsymbol{y}$

# Generalization

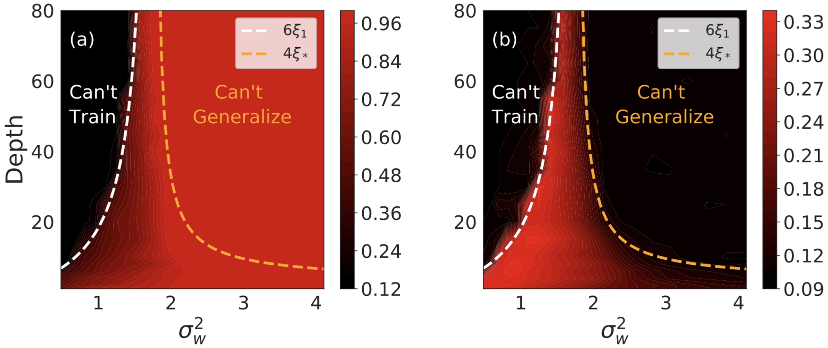
- Prior mean predictions for test set: $\hat{\boldsymbol{y}}_M = \boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}(\boldsymbol{I} - e^{-\eta \boldsymbol{T}_{N,N} t})\boldsymbol{y}$
- As $t \to \infty$ (trained), we have

$$\hat{\boldsymbol{y}}_M = \boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}\boldsymbol{y}$$

- Goal: the values of $\hat{\boldsymbol{y}}_M$ depend on data $\boldsymbol{X}_M$, $\boldsymbol{X}_N$, and $\boldsymbol{y}$
- If $\boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}$ ***is a data-independent constant matrix***, then the NN will fail to generalize
    - If $\boldsymbol{y}$ has zero mean, this implies that $\boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}\boldsymbol{y} = \boldsymbol{0}$

# Results

- The training and test accuracy (color) of a fully-connected NN trained with SGD
  - (a) The NN is untrainable because $\kappa$ is too large
  - (b) The NN is ungeneralizable because $\boldsymbol{T}_{M,N}\boldsymbol{T}_{N,N}^{-1}\boldsymbol{y}$ is too small



# Outline

1. Optimization
    - Momentum & Nesterov Momentum
    - AdaGrad & RMSProp
    - Batch Normalization
    - Continuation Methods & Curriculum Learning
    - NTK-based Initialization
2. Regularization
    - Cyclic Learning Rates
    - Weight Decay
    - Data Augmentation
    - Dropout
    - Manifold Regularization
    - Domain-Specific Model Design

# Regularization

- The goal of an ML algorithm is to perform well not just on the training data, but also on new inputs

# Regularization

- The goal of an ML algorithm is to perform well not just on the training data, but also on new inputs
- ***Regularization***: techniques that reduce the generalization error of an ML algorithm
  - But not the training error

# Regularization

- The goal of an ML algorithm is to perform well not just on the training data, but also on new inputs
- ***Regularization***: techniques that reduce the generalization error of an ML algorithm
  - But not the training error
- By expressing preference to a simpler model

# Regularization

- The goal of an ML algorithm is to perform well not just on the training data, but also on new inputs
- ***Regularization***: techniques that reduce the generalization error of an ML algorithm
  - But not the training error
- By expressing preference to a simpler model
- By providing different perspectives on how to explain the training data

# Regularization

- The goal of an ML algorithm is to perform well not just on the training data, but also on new inputs
- ***Regularization***: techniques that reduce the generalization error of an ML algorithm
  - But not the training error
- By expressing preference to a simpler model
- By providing different perspectives on how to explain the training data
- By encoding prior knowledge

# Regularization in Deep Learning I

- I have big data, do I still need to regularize my NN?
    - The excess error is dominated by optimization error (time)

# Regularization in Deep Learning I

- I have big data, do I still need to regularize my NN?
    - The excess error is dominated by optimization error (time)
- Generally, yes!

# Regularization in Deep Learning I

- I have big data, do I still need to regularize my NN?
  - The excess error is dominated by optimization error (time)
- Generally, yes!
- For "hard" problems, the true data generating process is almost certainly outside the model family
  - E.g., problems in images, audio sequences, and text domains

# Regularization in Deep Learning I

- I have big data, do I still need to regularize my NN?
  - The excess error is dominated by optimization error (time)
- Generally, yes!
- For "hard" problems, the true data generating process is almost certainly outside the model family
  - E.g., problems in images, audio sequences, and text domains
  - The true generation process essentially involves simulating the entire universe

# Regularization in Deep Learning I

- I have big data, do I still need to regularize my NN?
  - The excess error is dominated by optimization error (time)
- Generally, yes!
- For "hard" problems, the true data generating process is almost certainly outside the model family
  - E.g., problems in images, audio sequences, and text domains
  - The true generation process essentially involves simulating the entire universe
- In these domains, the best fitting model (with lowest generalization error) is usually a larger model regularized appropriately

# Regularization in Deep Learning II

- For "easy" problems, regularization may be necessary to make the problems well defined

# Regularization in Deep Learning II

- For "easy" problems, regularization may be necessary to make the problems well defined
- For example, when applying a logistic regression to a linearly separable dataset:

$$
\begin{array}{c}
\arg\max_{\boldsymbol{w}} \log \prod_i \mathrm{P}(y^{(i)} \,|\, \boldsymbol{x}^{(i)}; \boldsymbol{w}) \\
= \arg\max_{\boldsymbol{w}} \log \prod_i \sigma(\boldsymbol{w}^\top (i))^{y^{(i)}} [1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}^{(i)})]^{(1-y^{(i)})}
\end{array}
$$

# Regularization in Deep Learning II

- For "easy" problems, regularization may be necessary to make the problems well defined
- For example, when applying a logistic regression to a linearly separable dataset:

$$\arg\max_{\boldsymbol{w}} \log \prod_i \mathrm{P}(y^{(i)} \,|\, \boldsymbol{x}^{(i)}; \boldsymbol{w})$$
$$= \arg\max_{\boldsymbol{w}} \log \prod_i \sigma(\boldsymbol{w}^\top (i))^{y^{(i)}} [1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}^{(i)})]^{(1-y^{(i)})}$$

  - If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, so is $2\boldsymbol{w}$

# Regularization in Deep Learning II

- For "easy" problems, regularization may be necessary to make the problems well defined
- For example, when applying a logistic regression to a linearly separable dataset:

$$\arg\max_{\boldsymbol{w}} \log \prod_i \mathrm{P}(y^{(i)} \,|\, \boldsymbol{x}^{(i)}; \boldsymbol{w})$$
$$= \arg\max_{\boldsymbol{w}} \log \prod_i \sigma(\boldsymbol{w}^\top (i))^{y^{(i)}} [1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}^{(i)})]^{(1-y^{(i)})}$$

  - If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, so is $2\boldsymbol{w}$
  - Furthermore, $2\boldsymbol{w}$ gives higher likelihood

# Regularization in Deep Learning II

- For "easy" problems, regularization may be necessary to make the problems well defined
- For example, when applying a logistic regression to a linearly separable dataset:

$$\begin{gathered}\arg\max_{\boldsymbol{w}} \log \prod_i \mathrm{P}(y^{(i)} \,|\, \boldsymbol{x}^{(i)}; \boldsymbol{w}) \\ = \arg\max_{\boldsymbol{w}} \log \prod_i \sigma(\boldsymbol{w}^\top (i))^{y^{(i)}} [1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}^{(i)})]^{(1-y^{(i)})}\end{gathered}$$

  - If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, so is $2\boldsymbol{w}$
  - Furthermore, $2\boldsymbol{w}$ gives higher likelihood
  - Without regularization, SGD will continually increase $\boldsymbol{w}$'s magnitude

# Regularization in Deep Learning II

- For "easy" problems, regularization may be necessary to make the problems well defined
- For example, when applying a logistic regression to a linearly separable dataset:

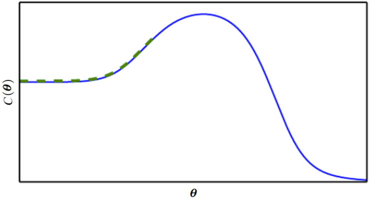
$$\arg\max_{\boldsymbol{w}} \log \prod_i \mathrm{P}(y^{(i)} \,|\, \boldsymbol{x}^{(i)}; \boldsymbol{w})$$
$$= \arg\max_{\boldsymbol{w}} \log \prod_i \sigma(\boldsymbol{w}^\top (i))^{y^{(i)}} [1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}^{(i)})]^{(1-y^{(i)})}$$

  - If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, so is $2\boldsymbol{w}$
  - Furthermore, $2\boldsymbol{w}$ gives higher likelihood
  - Without regularization, SGD will continually increase $\boldsymbol{w}$'s magnitude
- A deep NN is likely to separable a dataset and has the similar issue

# Outline

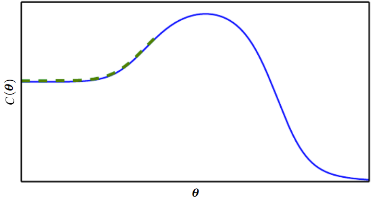
# SGD Gradients are Noisy

- ***Initialization*** is important
- SGD gradients may not be representative in the beginning (and in the end)

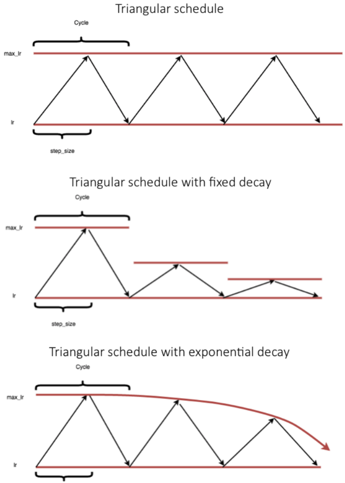


# SGD Gradients are Noisy

- *Initialization* is important
- SGD gradients may not be representative in the beginning (and in the end)




- Use a small learning rate in the very beginning [10]



# Outline

1. Optimization
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. Regularization
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Weight Decay

- To add norm penalties:

$$\arg\min_{\Theta} C(\Theta) + \alpha\Omega(\Theta)$$

  - $\Omega$ can be, e.g., $L^1$- or $L^2$-norm

# Weight Decay

- To add norm penalties:

$$\arg\min_{\boldsymbol{\Theta}} C(\boldsymbol{\Theta}) + \alpha\Omega(\boldsymbol{\Theta})$$

  - $\Omega$ can be, e.g., $L^1$- or $L^2$-norm
- $\Omega(\mathsf{W})$, $\Omega(\boldsymbol{W}^{(k)})$, $\Omega(\boldsymbol{W}^{(k)}_{i,:})$, or $\Omega(\boldsymbol{W}^{(k)}_{:,j})$?

# Weight Decay

- To add norm penalties:

$$\arg\min_{\Theta} C(\Theta) + \alpha \Omega(\Theta)$$

  - $\Omega$ can be, e.g., $L^1$- or $L^2$-norm
- $\Omega(\mathbf{W})$, $\Omega(\boldsymbol{W}^{(k)})$, $\Omega(\boldsymbol{W}^{(k)}_{i,:})$, or $\Omega(\boldsymbol{W}^{(k)}_{:,j})$?
- Limiting column norms $\Omega(\boldsymbol{W}^{(k)}_{:,j})$, $\forall j,k$, is preferred [5]
  - Prevents any one hidden unit from having very large weights and $z_j^{(k)}$

# Explicit Weight Decay I

- Explicit norm penalties:

$$\arg\min_{\Theta} C(\Theta) \text{ subject to } \Omega(\Theta) \leq R$$

# Explicit Weight Decay I

- Explicit norm penalties:

$$\arg\min_{\Theta} C(\Theta) \text{ subject to } \Omega(\Theta) \le R$$

- To solve the problem, we can use the ***projective SGD***:
  - At each step $t$, update $\Theta^{(t+1)}$ as in SGD
  - If $\Theta^{(t+1)}$ falls out of the feasible set, project $\Theta^{(t+1)}$ back to the tangent space (edge) of feasible set
- Advantage?

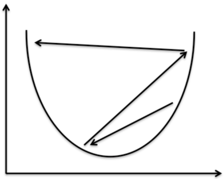
# Explicit Weight Decay I

- Explicit norm penalties:

$$\arg\min_{\Theta} C(\Theta) \text{ subject to } \Omega(\Theta) \leq R$$

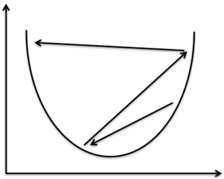
- To solve the problem, we can use the ***projective SGD***:
  - At each step $t$, update $\Theta^{(t+1)}$ as in SGD
  - If $\Theta^{(t+1)}$ falls out of the feasible set, project $\Theta^{(t+1)}$ back to the tangent space (edge) of feasible set
- Advantage?
- Prevents ***dead units*** that do not contribute much to the behavior of NN due to too small weights
  - Explicit constraints does not push weights to the origin

# Explicit Weight Decay II



- Also prevents instability due to a large learning rate
  - Reprojection clips the weights and improves numeric stability

# Explicit Weight Decay II



- Also prevents instability due to a large learning rate
  - Reprojection clips the weights and improves numeric stability
- Hinton et al. [5] recommend using:

$$\text{explicit constraints} + \text{reprojection} + \text{large learning rate}$$

  to allow rapid exploration of parameter space while maintaining numeric stability

# Outline

1. Optimization
   - Momentum & Nesterov Momentum
   - AdaGrad & RMSProp
   - Batch Normalization
   - Continuation Methods & Curriculum Learning
   - NTK-based Initialization
2. Regularization
   - Cyclic Learning Rates
   - Weight Decay
   - Data Augmentation
   - Dropout
   - Manifold Regularization
   - Domain-Specific Model Design

# Data Augmentation

- Theoretically, the best way to improve the generalizability of a model is to train it on ***more data***

# Data Augmentation

- Theoretically, the best way to improve the generalizability of a model is to train it on ***more data***
- For some ML tasks, it is not hard to create new ***fake*** data
- In classification, we can generate new $(\boldsymbol{x},\boldsymbol{y})$ pairs by transforming an example input $\boldsymbol{x}^{(i)}$ given the same $\boldsymbol{y}^{(i)}$
  - E.g, scaling, translating, rotating, or flipping images ($\boldsymbol{x}^{(i)}$'s)

# Data Augmentation

- Theoretically, the best way to improve the generalizability of a model is to train it on ***more data***
- For some ML tasks, it is not hard to create new ***fake*** data
- In classification, we can generate new $(\boldsymbol{x},\boldsymbol{y})$ pairs by transforming an example input $\boldsymbol{x}^{(i)}$ given the same $\boldsymbol{y}^{(i)}$
  - E.g, scaling, translating, rotating, or flipping images ($\boldsymbol{x}^{(i)}$'s)
- Very effective in image object recognition and speech recognition tasks

# Data Augmentation

- Theoretically, the best way to improve the generalizability of a model is to train it on ***more data***
- For some ML tasks, it is not hard to create new ***fake*** data
- In classification, we can generate new $(\boldsymbol{x},\boldsymbol{y})$ pairs by transforming an example input $\boldsymbol{x}^{(i)}$ given the same $\boldsymbol{y}^{(i)}$
  - E.g, scaling, translating, rotating, or flipping images ($\boldsymbol{x}^{(i)}$'s)
- Very effective in image object recognition and speech recognition tasks

**Caution**

Do not to apply transformations that would change the correct class!

# Data Augmentation

- Theoretically, the best way to improve the generalizability of a model is to train it on ***more data***
- For some ML tasks, it is not hard to create new ***fake*** data
- In classification, we can generate new $(\boldsymbol{x},\boldsymbol{y})$ pairs by transforming an example input $\boldsymbol{x}^{(i)}$ given the same $\boldsymbol{y}^{(i)}$
  - E.g, scaling, translating, rotating, or flipping images ($\boldsymbol{x}^{(i)}$'s)
- Very effective in image object recognition and speech recognition tasks

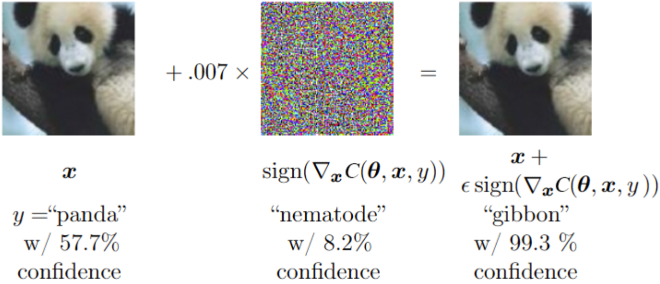
**Caution**

Do not to apply transformations that would change the correct class!

- E.g., in OCR tasks, avoid:
  - Horizontal flips for 'b' and 'd'
  - $180^\circ$ rotations for '6' and '9'

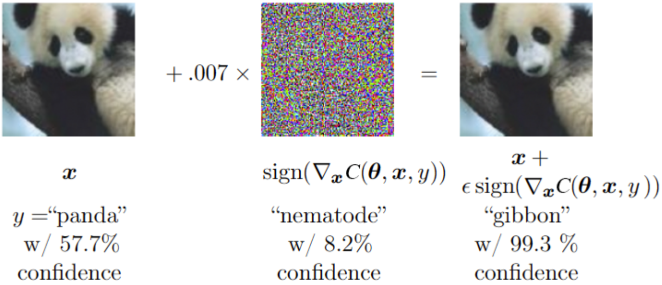
# Noise and Adversarial Data

- NNs are ***not*** very robust to the perturbation of input ($\boldsymbol{x}^{(i)}$'s)
  - Noises [12]
  - Adversarial points [3]



# Noise and Adversarial Data

- NNs are ***not*** very robust to the perturbation of input ($\boldsymbol{x}^{(i)}$'s)
  - Noises [12]
  - Adversarial points [3]




- How to improve the robustness?

# Noise Injection

- We can train an NN ***with artificial random noise*** applied to $\boldsymbol{x}^{(i)}$'s

# Noise Injection

- We can train an NN ***with artificial random noise*** applied to $\boldsymbol{x}^{(i)}$'s
- Why noise injection works?

# Noise Injection

- We can train an NN ***with artificial random noise*** applied to $\boldsymbol{x}^{(i)}$'s
- Why noise injection works?
- Recall that the analytic solution of Ridge regression is

$$\boldsymbol{w} = \left(\boldsymbol{X}^\top \boldsymbol{X} + \alpha^{(t)} \boldsymbol{I}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

  - In this case, weight decay = adding variance (noises)

# Noise Injection

- We can train an NN ***with artificial random noise*** applied to $\boldsymbol{x}^{(i)}$'s
- Why noise injection works?
- Recall that the analytic solution of Ridge regression is

$$\boldsymbol{w} = \left(\boldsymbol{X}^\top \boldsymbol{X} + \alpha^{(t)} \boldsymbol{I}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

  - In this case, weight decay = adding variance (noises)
- More generally, makes the function $f$ ***locally constant***
  - Cost function $C$ insensitive to small variations in weights
  - Finds solutions that are not merely minima, but minima surrounded by flat regions

# Noise Injection

- We can train an NN ***with artificial random noise*** applied to $\boldsymbol{x}^{(i)}$'s
- Why noise injection works?
- Recall that the analytic solution of Ridge regression is

$$\boldsymbol{w} = \left(\boldsymbol{X}^\top \boldsymbol{X} + \alpha^{(t)} \boldsymbol{I}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

  - In this case, weight decay = adding variance (noises)
- More generally, makes the function $f$ ***locally constant***
  - Cost function $C$ insensitive to small variations in weights
  - Finds solutions that are not merely minima, but minima surrounded by flat regions

# Variants

- We can also inject noise to hidden representations [8]
  - Highly effective provided that the magnitude of the noise can be carefully tuned

# Variants

- We can also inject noise to hidden representations [8]
  - Highly effective provided that the magnitude of the noise can be carefully tuned
- The batch normalization, in addition to simplifying optimization, offers similar regularization effect to noise injection
  - Injects noises from examples in a minibatch to an activation $a_j^{(k)}$

# Variants

- We can also inject noise to hidden representations [8]
  - Highly effective provided that the magnitude of the noise can be carefully tuned
- The batch normalization, in addition to simplifying optimization, offers similar regularization effect to noise injection
  - Injects noises from examples in a minibatch to an activation $a_j^{(k)}$
- How about injecting noise to outputs ($\boldsymbol{y}^{(i)}$'s)?

# Variants

- We can also inject noise to hidden representations [8]
  - Highly effective provided that the magnitude of the noise can be carefully tuned
- The batch normalization, in addition to simplifying optimization, offers similar regularization effect to noise injection
  - Injects noises from examples in a minibatch to an activation $a_j^{(k)}$
- How about injecting noise to outputs ($\boldsymbol{y}^{(i)}$'s)?
  - Already done in probabilistic models

# Outline

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent
  - Boosting: increase confidence (margin) of predictions, ***if not overfitting***

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent
  - Boosting: increase confidence (margin) of predictions, ***if not overfitting***
- Ensemble methods in deep learning?

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent
  - Boosting: increase confidence (margin) of predictions, ***if not overfitting***
- Ensemble methods in deep learning?
  - Voting: train multiple NNs

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent
  - Boosting: increase confidence (margin) of predictions, ***if not overfitting***
- Ensemble methods in deep learning?
  - Voting: train multiple NNs
  - Bagging: train multiple NNs, each with resampled $\mathbb{X}$

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent
  - Boosting: increase confidence (margin) of predictions, ***if not overfitting***
- Ensemble methods in deep learning?
  - Voting: train multiple NNs
  - Bagging: train multiple NNs, each with resampled $\mathbb{X}$
- GoogleLeNet [11], winner of ILSVRC'14, is an ensemble of 6 NNs

# Ensemble Methods

- Ensemble methods can improve generalizability by offering different explanations to $\mathbb{X}$
  - Voting: reduces variance of predictions if having independent voters
  - Bagging: resample $\mathbb{X}$ to makes voters less dependent
  - Boosting: increase confidence (margin) of predictions, ***if not overfitting***
- Ensemble methods in deep learning?
  - Voting: train multiple NNs
  - Bagging: train multiple NNs, each with resampled $\mathbb{X}$
- GoogleLeNet [11], winner of ILSVRC'14, is an ensemble of 6 NNs
- Very time consuming to ensemble a large number of NNs

# Dropout I

- ***Dropout***: a feature-based bagging

# Dropout I

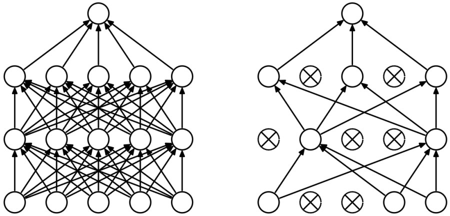
- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features

# Dropout I

- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters

# Dropout I

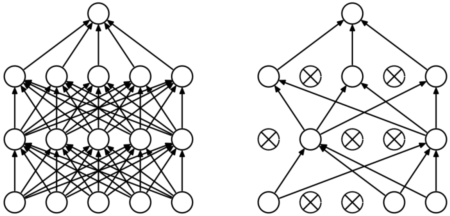
- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters



- SGD training: each time loading a minibatch, randomly sample a binary mask to apply to all input and hidden units
  - Each unit has probability $\alpha$ to be included (a hyperparameter)

# Dropout I

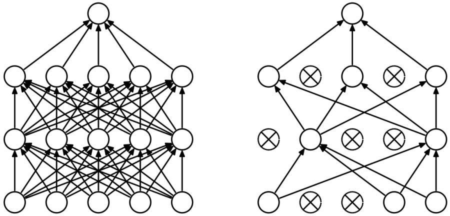
- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters



- SGD training: each time loading a minibatch, randomly sample a binary mask to apply to all input and hidden units
  - Each unit has probability $\alpha$ to be included (a hyperparameter)
  - Typically, $0.8$ for input units and $0.5$ for hidden units

# Dropout I

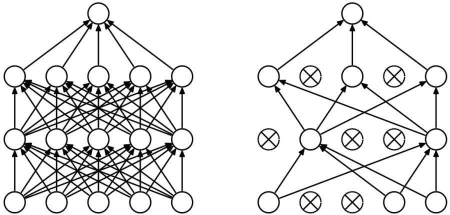
- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters



- SGD training: each time loading a minibatch, randomly sample a binary mask to apply to all input and hidden units
  - Each unit has probability $\alpha$ to be included (a hyperparameter)
  - Typically, $0.8$ for input units and $0.5$ for hidden units
- Different minibatches are used to train different parts of the NN
  - Similar to bagging, but much more efficient

# Dropout I

- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters



- SGD training: each time loading a minibatch, randomly sample a binary mask to apply to all input and hidden units
  - Each unit has probability $\alpha$ to be included (a hyperparameter)
  - Typically, $0.8$ for input units and $0.5$ for hidden units
- Different minibatches are used to train different parts of the NN
  - Similar to bagging, but much more efficient
  - No need to retrain unmasked units
  - Exponential number of voters

# Dropout II

- How to vote to make a final prediction?

# Dropout II

- How to vote to make a final prediction?
- Mask sampling:
  1. Randomly sample some (typically, $10 \sim 20$) masks
  2. For each mask, apply it to the trained NN and get a prediction
  3. Average the predictions

# Dropout II

- How to vote to make a final prediction?
- Mask sampling:
  1. Randomly sample some (typically, $10 \sim 20$) masks
  2. For each mask, apply it to the trained NN and get a prediction
  3. Average the predictions
- Weigh scaling:
  - Make a single prediction using the NN with all units
  - But weights going out from a unit is multiplied by $\alpha$

# Dropout II

- How to vote to make a final prediction?
- Mask sampling:
  1. Randomly sample some (typically, $10 \sim 20$) masks
  2. For each mask, apply it to the trained NN and get a prediction
  3. Average the predictions
- Weigh scaling:
  - Make a single prediction using the NN with all units
  - But weights going out from a unit is multiplied by $\alpha$
  - Heuristic: each unit outputs the same expected amount of weight as in training

# Dropout II

- How to vote to make a final prediction?
- Mask sampling:
  1. Randomly sample some (typically, $10 \sim 20$) masks
  2. For each mask, apply it to the trained NN and get a prediction
  3. Average the predictions
- Weigh scaling:
  - Make a single prediction using the NN with all units
  - But weights going out from a unit is multiplied by $\alpha$
  - Heuristic: each unit outputs the same expected amount of weight as in training
- The better one is problem dependent

# Dropout III

- Dropout improves generalization beyond ensembling

# Dropout III

- Dropout improves generalization beyond ensembling
- For example, in face image recognition:

# Dropout III

- Dropout improves generalization beyond ensembling
- For example, in face image recognition:
- If there is a unit that detects nose

# Dropout III

- Dropout improves generalization beyond ensembling
- For example, in face image recognition:
- If there is a unit that detects nose
- Dropping the unit encourages the model to learn mouth (or nose again) in another unit

# Outline

# Manifolds I

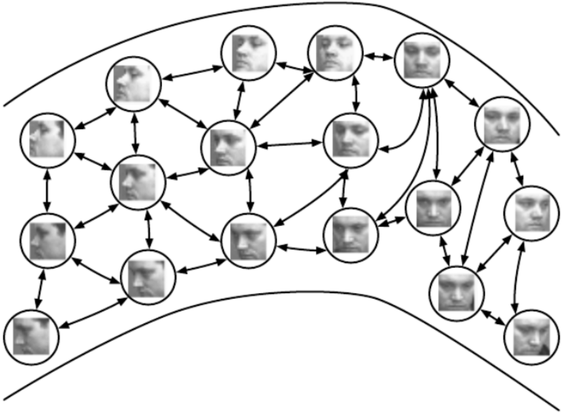
- One way to improve the generalizability of a model is to incorporate the prior knowledge

# Manifolds I

- One way to improve the generalizability of a model is to incorporate the prior knowledge
- In many applications, data of the same class concentrate around one or more low-dimensional ***manifolds***

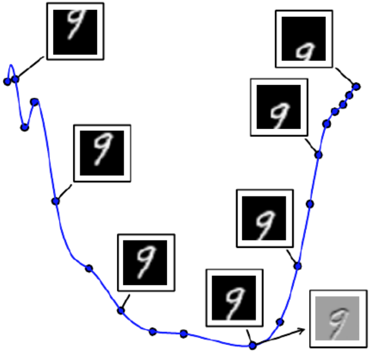
# Manifolds I

- One way to improve the generalizability of a model is to incorporate the prior knowledge
- In many applications, data of the same class concentrate around one or more low-dimensional ***manifolds***
- A manifold is a topological space that are ***linear locally***



# Manifolds II

- For each point $\boldsymbol{x}$ on a manifold, we have its ***tangent space*** spanned by ***tangent vectors***
  - Local directions specify how one can change $\boldsymbol{x}$ infinitesimally while staying on the manifold



# Tangent Prop

- How to incorporate the manifold prior into a model?

# Tangent Prop

- How to incorporate the manifold prior into a model?
- Suppose we have the tangent vectors $\{\boldsymbol{v}^{(i,j)}\}_j$ for each example $\boldsymbol{x}^{(i)}$
- Tangent Prop [9] trains an NN classifier $f$ with cost penalty:

$$\Omega[f] = \sum_{i,j} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(i)})^\top \boldsymbol{v}^{(i,j)}$$

  - To make $f$ ***local constant along tangent directions***

# Tangent Prop

- How to incorporate the manifold prior into a model?
- Suppose we have the tangent vectors $\{\boldsymbol{v}^{(i,j)}\}_j$ for each example $\boldsymbol{x}^{(i)}$
- Tangent Prop [9] trains an NN classifier $f$ with cost penalty:

$$\Omega[f] = \sum_{i,j} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(i)})^\top \boldsymbol{v}^{(i,j)}$$

  - To make $f$ ***local constant along tangent directions***
- How to obtain $\{\boldsymbol{v}^{(i,j)}\}_j$ ?

# Tangent Prop

- How to incorporate the manifold prior into a model?
- Suppose we have the tangent vectors $\{\boldsymbol{v}^{(i,j)}\}_j$ for each example $\boldsymbol{x}^{(i)}$
- Tangent Prop [9] trains an NN classifier $f$ with cost penalty:

$$\Omega[f] = \sum_{i,j} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(i)})^\top \boldsymbol{v}^{(i,j)}$$

  - To make $f$ ***local constant along tangent directions***
- How to obtain $\{\boldsymbol{v}^{(i,j)}\}_j$ ?
- Manually specified based on domain knowledge
  - Images: scaling, translating, rotating, flipping etc.

# Tangent Prop

- How to incorporate the manifold prior into a model?
- Suppose we have the tangent vectors $\{\boldsymbol{v}^{(i,j)}\}_j$ for each example $\boldsymbol{x}^{(i)}$
- Tangent Prop [9] trains an NN classifier $f$ with cost penalty:

$$\Omega[f] = \sum_{i,j} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(i)})^\top \boldsymbol{v}^{(i,j)}$$

  - To make $f$ ***local constant along tangent directions***
- How to obtain $\{\boldsymbol{v}^{(i,j)}\}_j$ ?
- Manually specified based on domain knowledge
  - Images: scaling, translating, rotating, flipping etc.
- Or learned automatically (to be discussed later)

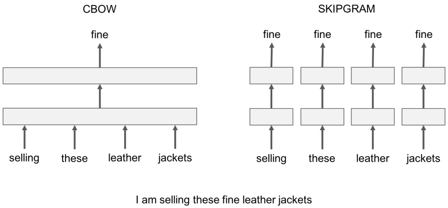
# Outline

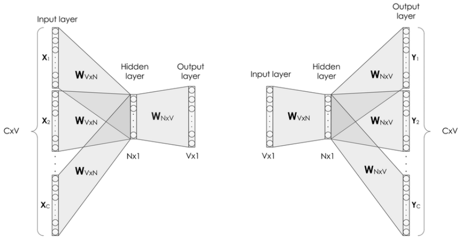
# Domain-Specific Prior Knowledge

- If done right, incorporating the domain-specific prior knowledge into a model is a highly effective way the improve generalizability
  - Better $f$ that "makes sense"
  - May also simplify optimization problem

# Word2vec

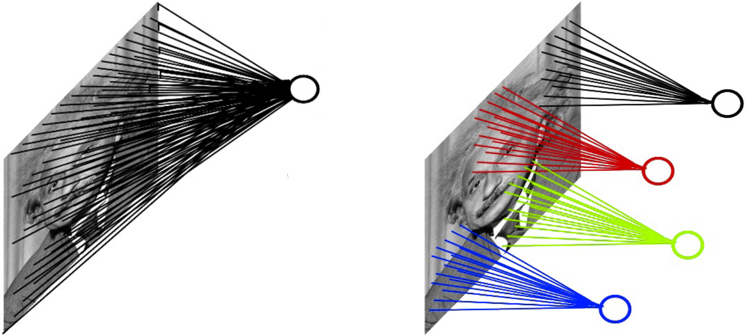
- ***Weight-tying*** leads to simpler model



# Convolution Neural Networks

- ***Locally connected*** neurons for pattern detection at different locations



# Reference I

[1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston.
Curriculum learning.
In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[2] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun.
The loss surfaces of multilayer networks.
In *AISTATS*, 2015.

[3] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy.
Explaining and harnessing adversarial examples.
*arXiv preprint arXiv:1412.6572*, 2014.

[4] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe.
Qualitatively characterizing neural network optimization problems.
*arXiv preprint arXiv:1412.6544*, 2014.

# Reference II

[5] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov.
Improving neural networks by preventing co-adaptation of feature detectors.
*arXiv preprint arXiv:1207.0580*, 2012.

[6] Sergey Ioffe and Christian Szegedy.
Batch normalization: Accelerating deep network training by reducing internal covariate shift.
*arXiv preprint arXiv:1502.03167*, 2015.

[7] Diederik Kingma and Jimmy Ba.
Adam: A method for stochastic optimization.
*arXiv preprint arXiv:1412.6980*, 2014.

[8] Ben Poole, Jascha Sohl-Dickstein, and Surya Ganguli.
Analyzing noise in autoencoders and deep networks.
*arXiv preprint arXiv:1406.1831*, 2014.

# Reference III


[9] Patrice Simard, Bernard Victorri, Yann LeCun, and John S Denker. Tangent prop-a formalism for specifying selected invariances in an adaptive network. In *NIPS*, volume 91, pages 895–903, 1991.

[10] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

# Reference IV


[12] Yichuan Tang and Chris Eliasmith.
Deep networks for robust visual recognition.
In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1055–1062, 2010.