

Deep Learning and Applications

DSA 5204 • Lecture 7
Dr Low Yi Rui (Aaron)
Department of Mathematics



NUS
National University
of Singapore

Homework 2



Homework 2 grades + comments will be released soon.

Do not wait for this – proceed with your project

Work actively together and contribute! Group member contributions will be peer reviewed at the end of the semester. Individual marks will be subject to moderation

Presentation: Last 2 classes, dual track, slides due Week 12.

Last Time

We introduced some methods to improve performance

- Parameter norm penalties
- Early stopping
- Injecting Noise

These methods mostly leave the model intact.

Today, we will look at some other popular techniques, some involving changing the model



Model Ensembling (Training, Model)

Bootstrap Aggregating (Bagging)

Bagging is the simplest method for combining models:

- We train m models $\{f_1, \dots, f_m\}$ on **random subsamples** of the training data.
- We then combine them in an obvious way to make predictions:

1. Regression

$$\bar{f}(x) = \frac{1}{m} \sum_{j=1}^m f_j(x)$$

2. Classification

$\bar{f}(x) = \text{Mode}\{f_j(x): j = 1, \dots, m\}$
alternatively, can use other voting schemes

Example

Dataset: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$

Subsample and train:

$$1. \quad \mathcal{D}_1 = \{(x_1, y_1), (x_2, y_2)\} \xrightarrow{\text{train}} f_1$$

$$2. \quad \mathcal{D}_2 = \{(x_2, y_2), (x_3, y_3)\} \xrightarrow{\text{train}} f_2$$

$$3. \quad \mathcal{D}_3 = \{(x_1, y_1), (x_1, y_1)\} \xrightarrow{\text{train}} f_3$$

Aggregate:

$$\bar{f}(x) = \frac{1}{3} (f_1(x) + f_2(x) + f_3(x))$$

Why does this help?

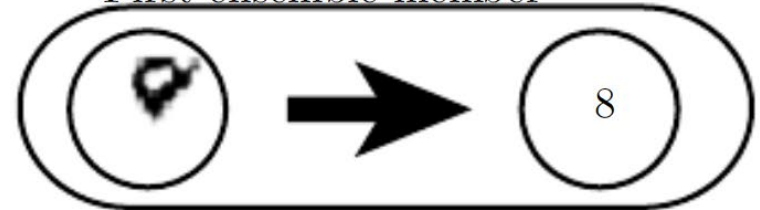
Original dataset



First resampled dataset



First ensemble member



Second resampled dataset



Second ensemble member



What does bagging do?

<Lecture Notebook>



Consider a simple scalar model where

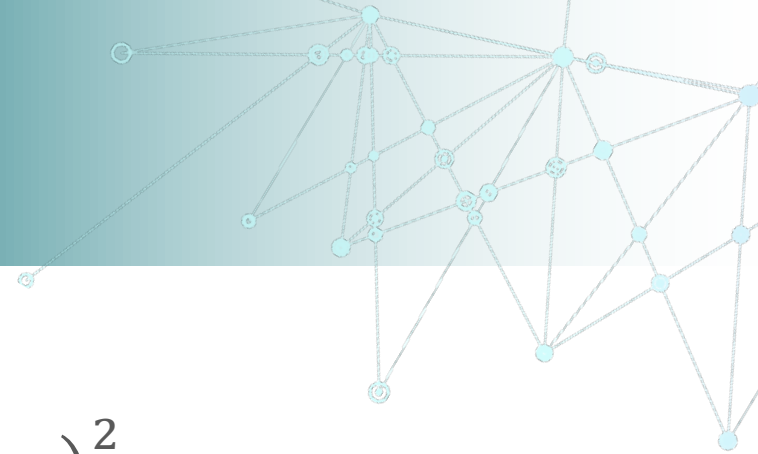
$$f_i(x) = f^*(x) + \underbrace{\epsilon_i(x)}_{\text{noise}}$$

Assume the noise satisfies

- $\mathbb{E}\epsilon_i(x) = 0$ and $\mathbb{E}\epsilon_i(x)^2 = \sigma(x)^2$
- Uncorrelated: $\mathbb{E}\epsilon_i(x)\epsilon_j(x) = 0$ for $i \neq j$

Form aggregate model

$$\bar{f}(x) = \frac{1}{m} \sum_{j=1}^m f_j(x)$$



Define the errors

$$E(x) = \frac{1}{m} \sum_{j=1}^m \mathbb{E} \left(f_j(x) - f^*(x) \right)^2$$

$$\bar{E}(x) = \mathbb{E} \left(\bar{f}(x) - f^*(x) \right)^2$$

We can show that

$$\bar{E}(x) = \frac{1}{m} E(x)$$

A significant reduction!

What is the most unrealistic assumption?

Dropout



A disadvantage of model ensembling is the cost associated with storing many versions of the model $\{f_1, \dots, f_m\}$

Dropout is an efficient, approximate way to construct model ensembles.

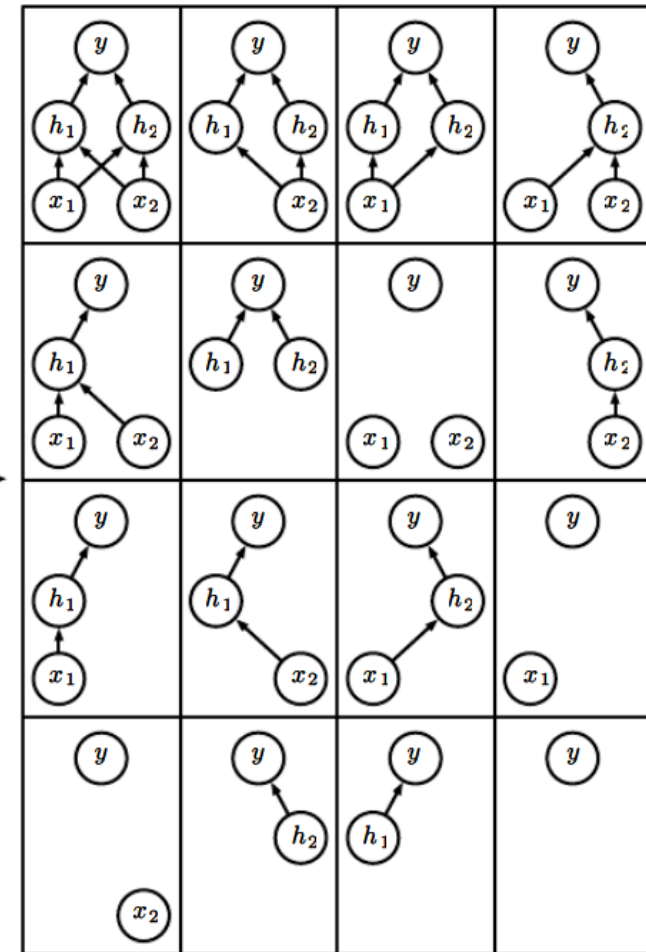
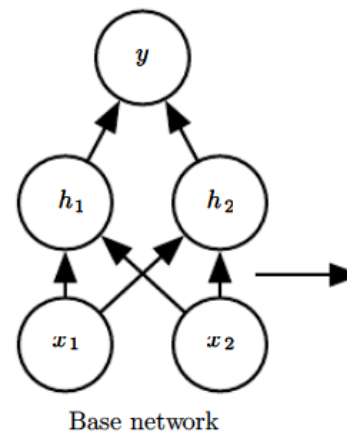
Main idea is similar to **GD**→**SGD**, by stochastic approximation

$$\frac{1}{m} \sum_i f_i = \mathbb{E}_{\gamma} f_{\gamma} \quad \gamma \sim \text{Uniform}\{1, 2, \dots, m\}$$

So, instead of computing the sum, we can sample from γ !

Dropout

Dropout implements averaging by applying a random mask over nodes/weights in a neural network





Mathematically, we can write a NN layer with dropout as

$$f(x; W, b, \mu) = \sigma(W(\mu \circ x) + b)$$

where μ is a vector mask (taking values 0 or 1) multiplied element-wise to the features.

For each iteration of SGD during training, a new sample μ is obtained by drawing each of its coordinate iid with probability p being 1 and $1 - p$ being 0

- p is called the keep probability ($1 - p$ is the drop rate)
- Note, $\mathbb{E}\mu = p \mathbf{1} = p (1, 1, \dots, 1)$, so this shrinks the weights/features by a factor of p
- Hence, during prediction, we will set $\mu = (1, 1, \dots, 1)$ (no mask) and replace the weights $W \mapsto p \times W$. This is called **weight scaling inference rule**.

Remarks on Dropout



- The entire model ensemble is
$$\{f(x, \theta, \mu): \mu_i = 0, 1, i = 1, \dots, m\}$$
- There are 2^m distinct members
- Members of the ensemble are not independent, since they share θ
- Inference and training complexity does not increase due to ensembling. The ensemble average
$$\mathbb{E}_{\mu} f(x, \theta, \mu)$$
is approximated by the weight scaling inference rule.

See deep learning book Chapter 7.12 for more on dropout

Example: Dropout for Linear Regression <Lecture Notebook>

Linear model with dropout:

$$f(x, w, \mu) = \frac{1}{p} w^T (\mu \circ x)$$

Empirical risk averaged over μ :

$$\tilde{R}(w) = \mathbb{E}_{\mu} \frac{1}{2} \|X[D_{\mu}/p]w - y\|^2$$

where D_{μ} is the diagonal matrix with μ on the diagonal.
Solving,

$$\tilde{R}(w) = \frac{1}{2} \|Xw - y\|^2 + \frac{1}{2} w^T Q(X, p)w$$

where $Q(X, p)$ depends on the dataset!

That is, dropout for linear regression is equivalent to a weighted L^2 regularization



Batch Normalization (Model)

Motivation



Consider a simple, 1D, deep linear NN

$$\hat{y}(x; \mathbf{w}) = \hat{y}(x; w_1, \dots, w_l) = x w_1 w_2 \cdots w_l$$

Loss: $L(\mathbf{w}) = \frac{1}{2} \hat{y}(x; \mathbf{w})^2$ (Square loss with label = 0).

Suppose we make an update

$$\mathbf{w}' = \mathbf{w} - \epsilon \mathbf{g} \quad (\mathbf{g} \text{ is the gradient})$$

The new loss is

$$L(\mathbf{w}') = L(\mathbf{w} - \epsilon \mathbf{g}) = L(\mathbf{w}) - \epsilon \|\mathbf{g}\|^2 + \mathcal{O}(\epsilon^2)$$

We have a decrease in the loss function only if ϵ is sufficiently small such that $\epsilon \|\mathbf{g}\|^2$ dominates the $\mathcal{O}(\epsilon^2)$ term.



In the deep network, we have

$$\begin{aligned}\hat{y}(x; \mathbf{w}') &= x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \cdots (w_l - \epsilon g_l) \\ &= \hat{y}(x; \mathbf{w}') - \epsilon x \sum_j g_j \prod_{k \neq j} w_k + \epsilon^2 x \sum_{j_1, j_2} g_{j_1} g_{j_2} \prod_{k \neq j_1, j_2} w_k + \mathcal{O}(\epsilon^3)\end{aligned}$$

And so

$$\begin{aligned}L(\mathbf{w}') - L(\mathbf{w}) \\ &= -\epsilon \hat{y}(x; \mathbf{w}) x \sum_j g_j \prod_{k \neq j} w_k + 2\epsilon^2 \hat{y}(x; \mathbf{w}) x \sum_{j_1, j_2} g_{j_1} g_{j_2} \prod_{k \neq j_1, j_2} w_k + \mathcal{O}(\epsilon^3)\end{aligned}$$

Take the first $\mathcal{O}(\epsilon^2)$ term:

$$2\epsilon^2 x \hat{y}(x, \mathbf{w}) g_1 g_2 (w_3 w_4 \dots w_l)$$

In order to make sure descent happens, this should be made small. However, this is difficult if the product $w_3 w_4 \cdots w_l$ is large!

Lessons from the Example



In deep neural networks:

- The learning rate choice depends on scales/magnitudes of weights and activations at different layers
- These change as training proceeds, making it very difficult to choose a constant learning rate

Idea: can we normalize the values at each layer so their scales/statistics are similar?

This is the primary motivation behind batch normalization!

The Batch Normalization Layer

Let $H = \{h^{(1)}, h^{(2)}, \dots, h^{(B)}\}$ be a batch of hidden state values in a deep network.

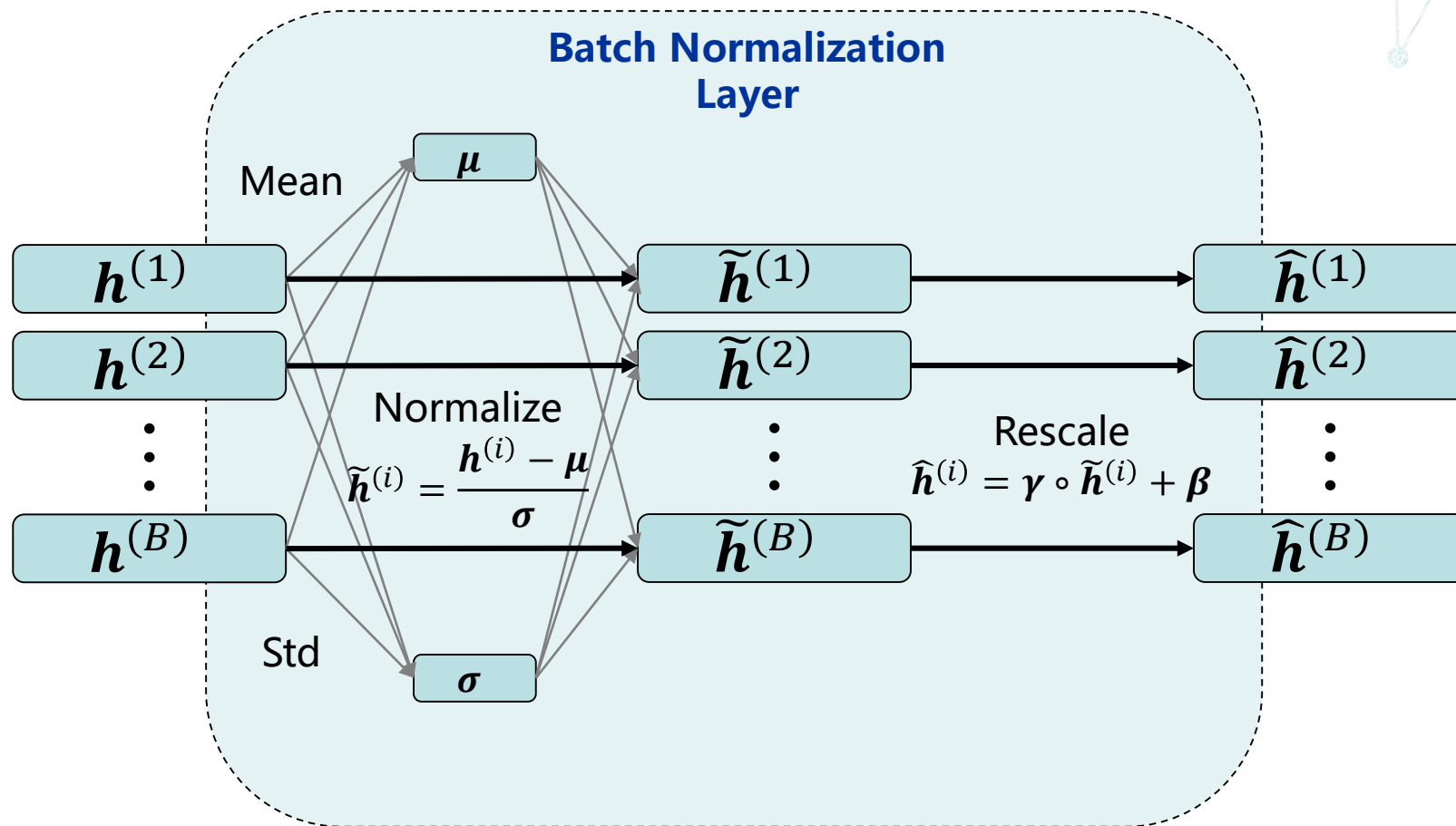
The batch normalization layer is defined by

$$\text{BN}(H; \gamma, \beta)^{(i)} = \gamma \circ \left(\frac{h^{(i)} - \mu}{\sigma} \right) + \beta$$

where

$$\underbrace{\mu = \frac{1}{B} \sum_j h^{(j)}}_{\text{Mean}} \quad \sigma = \underbrace{\sqrt{\frac{1}{B} \sum_j (h^{(j)} - \mu)^2}}_{\text{Standard Deviation}}$$

and $\gamma, \beta \in \mathbb{R}^d$ are vectors (\circ denotes element-wise product)



Note: This is the first architecture that we have seen that couples different samples!

Breaking Down BN Layer I (Normalization)



The first step in BN normalizes the hidden activations

$$\tilde{h}^{(i)} = \frac{h^{(i)} - \mu}{\sigma}$$

This ensures that the outputs $\{\tilde{h}^{(i)}\}$ has mean 0 and (element-wise) std 1:

$$\frac{1}{B} \sum_i \tilde{h}^{(i)} = \frac{1}{\sigma} (\mu - \mu) = 0$$
$$\frac{1}{B} \sum_i \left(\tilde{h}_j^{(i)} \right)^2 = \frac{\sigma_j^2}{\sigma_j^2} = 1$$

This standardizes the hidden states in the layer.

Breaking Down BN Layer I (Rescaling and Recentering)



However, since outputs are all standardized, its expressive power is reduced!

- Scale of outputs is order 1
- Always centered around 0

To reintroduce the expressive power, a rescaling step is used to introduce a scale and a bias

$$\hat{h}^{(i)} = \gamma \circ \tilde{h}^{(i)} + \beta$$

Using Batch Normalization in Deep Neural Networks



Usual deep fully connected network:

$$H_{k+1} = \text{ReLU}(H_k W + \mathbf{b})$$

With BN:

$$\begin{aligned} H'_k &= H_k W + \mathbf{b} \\ \hat{H}_k &= \text{BN}(H'_k; \boldsymbol{\gamma}, \boldsymbol{\beta}) \\ H_{k+1} &= \text{ReLU}(\hat{H}_k) \end{aligned}$$

BN during Training and Inference

A decorative network diagram in the top right corner, consisting of a series of interconnected nodes and edges, resembling a neural network or a complex graph structure.

During training, we apply the normalization

$$\tilde{h}^{(i)} = \frac{h^{(i)} - \mu}{\sigma}$$

using batch statistics.

During inference, if we only have one sample then this is impossible. Hence, we will

- Keep moving averages of μ, σ during training
- Use the moving averages of μ, σ for inference, independent of the current test sample

What is the Effect of BN?

- The normalization removes scaling and centers the data
- Different from a simple rescaling operation, the gradient flows through the normalization procedure!
- By normalizing over batches and then rescaling using scalars for each features, we “break” the inter-dependence of statistics of different layers: the scaling is controlled individually by the γ, β at every layer.

Further reading on the effect of BN

- Ioffe, Sergey, and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” *ArXiv Preprint ArXiv:1502.03167*, 2015.
- Cai, Yongqiang, Qianxiao Li, and Zuowei Shen. “A Quantitative Analysis of the Effect of Batch Normalization on Gradient Descent.” In *International Conference on Machine Learning*, 882–890, 2019. (References therein)

Some Additional Remarks on BN

- For convolutional neural networks, a common β and γ should be used for each channel of feature maps. That is, if a feature map is of size

$[\text{\#BatchSize}, \text{\#Height}, \text{\#Width}, \text{\#Channel}]$

then β, γ should be vectors of length \#Channel and applied to all pixels of a feature map with size $[\text{\#Height}, \text{\#Width}]$. **Why?**

- We usually apply batch norm between the affine transformation and the element-wise nonlinearity

$$\text{ReLU}(XW + b) \rightarrow \text{ReLU}(\text{BN}(XW + b))$$

In this case, it is no longer needed to have a bias b since

$$\text{BN}(XW + b) = \text{BN}(XW)$$

with a redefinition of $\beta \mapsto \beta + b$



Data Augmentation (Data)

Data vs Generalization

Recall the usual set up of supervised learning

- Input-output distribution: $(x, y) \sim \mu$
- Draw N samples: $(x^{(i)}, y^{(i)}) \sim \mu, i = 1, 2, \dots, N$
- Empirical Risk

$$R_{\text{emp}}^{(N)}(\theta) = \frac{1}{N} \sum_i L(\hat{y}(x^{(i)}, \theta), y^{(i)})$$

- Expected/Population Risk

$$R_{\text{pop}}(\theta) = \mathbb{E}_{x, y \sim \mu} [L(\hat{y}(x, \theta), y)]$$

Given input-output distribution $(x, y) \sim \mu$, draw N -samples

Data vs Generalization



Our primary goal is to reduce the generalization gap

$$\left| R_{\text{emp}}^{(N)}(\boldsymbol{\theta}) - R_{\text{pop}}(\boldsymbol{\theta}) \right|$$

Statistical learning theory usually gives bounds like

$$\left| R_{\text{emp}}^{(N)}(\boldsymbol{\theta}) - R_{\text{pop}}(\boldsymbol{\theta}) \right| \leq \frac{\text{Complexity of Model}}{N}$$

How to reduce this?

- Reduce complexity (regularization)
- Increase N (data augmentation)

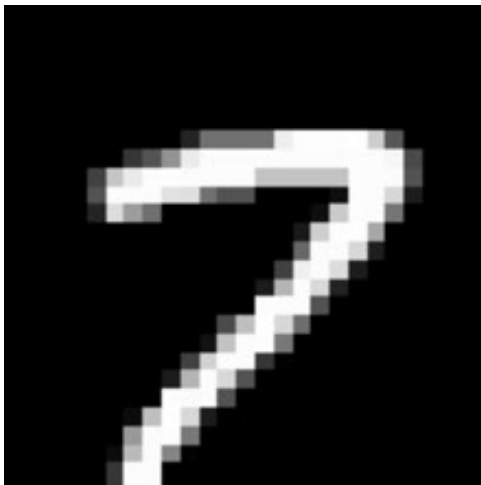
Data Augmentation



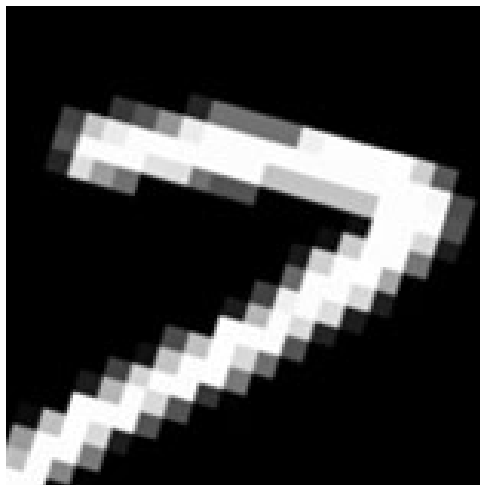
Data augmentation “increases” our dataset to reduce the generalization gap.

The main idea is to perturb a given datapoint to produce a new one with the same output

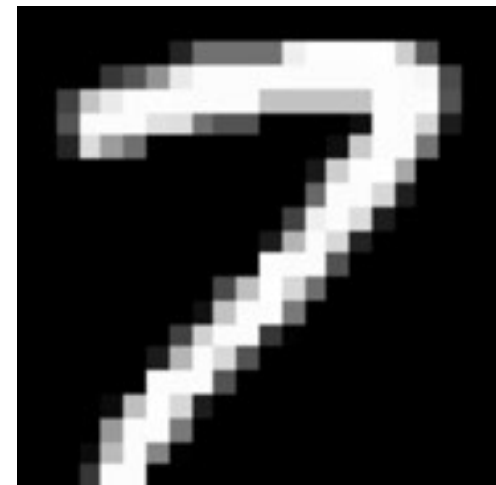
Original



Augmented 1



Augmented 2



Augmentation and Invariance



Let us consider the oracle function

$$y = f^*(x)$$

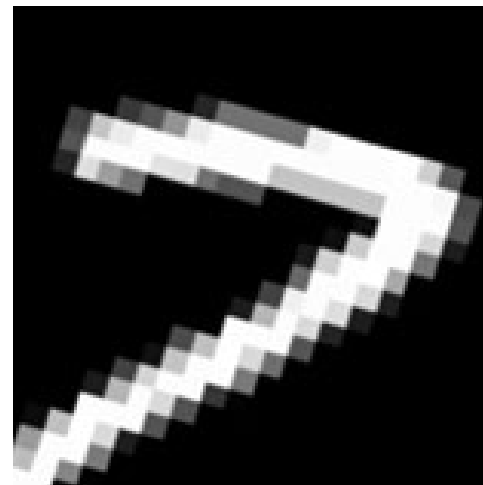
Consider a random augmentation transformation

$x \mapsto g(x; \xi)$ (ξ is a random variable)

x



$g(x; \xi)$



$\xi = \text{Random Rotation} + \text{Scaling}$



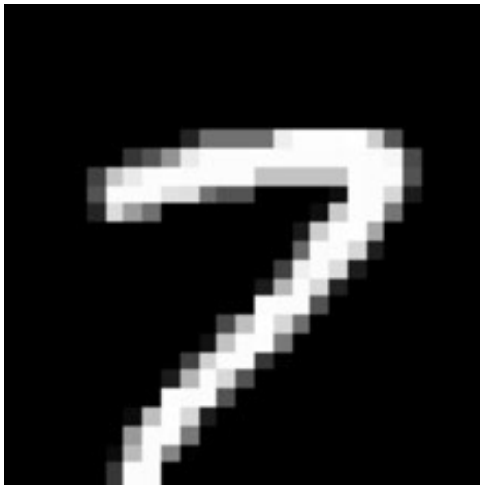
$g(\cdot; \xi)$

Fundamental principle behind data augmentation:

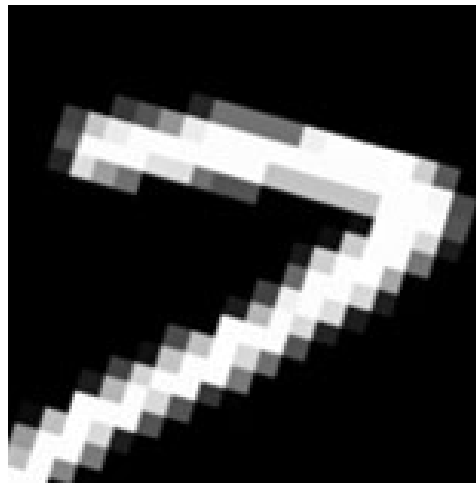
$$f^*(g(x; \xi)) \approx y = f^*(x)$$

In other words, we assumed that f^* is invariant under any random transformation $g(\cdot; \xi)$ on the inputs!

$$f^*(x) = "7"$$



$$f^*(g(x; \xi_1)) = "7"$$



$$f^*(g(x; \xi_2)) = "7"$$



Data Augmentation as a form of Prior Knowledge



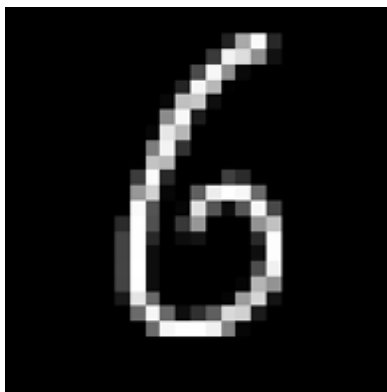
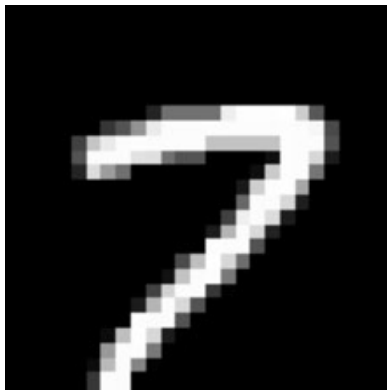
The reason data augmentation works is that we introduced prior knowledge into our model

- Using $g(\cdot; \xi)$ to augment introduces the knowledge that our oracle is invariant with respect to $x \mapsto g(x; \xi)$
- Examples
 - Augment with random translations = f^* is invariant to translations
 - Augment with rotations = f^* is invariant to rotations
- Hence, we must be careful about how we augment data

Example



Can we do arbitrary rotations to augment data for MNIST?



Oracle not invariant!





Learning Rate Decay (Training)

Convergence vs Learning Rate



Recall

- SGD is computationally efficient, but introduces noise
- For a simple quadratic problem, we have the estimate

$$\mathbb{E}R(\theta_k) \sim \underbrace{R(\theta_0)(1 - \epsilon)^{2k}}_{\text{Exponential convergence}} + \underbrace{\epsilon(1 - (1 - \epsilon)^{2k})}_{\mathcal{O}(\epsilon) \text{ fluctuations}}$$

Hence, to obtain good performance we should decrease the learning rate as training proceeds.

$$\epsilon = \epsilon_k$$

How to choose ϵ_k ?

From a theoretical point of view, the following are sufficient to ensure convergence of the empirical risk:

$$\sum_{k=0}^{\infty} \epsilon_k = +\infty \quad \text{and} \quad \sum_{k=0}^{\infty} \epsilon_k^2 < +\infty$$

Example: $\epsilon_k = \frac{a}{1+bk}$ for any $a, b \in \mathbb{R}^+$ suffices.

This is called an $\mathcal{O}(k^{-1})$ decay schedule. This is often used for pure optimization problems, but for learning problems they are not used for two reasons:

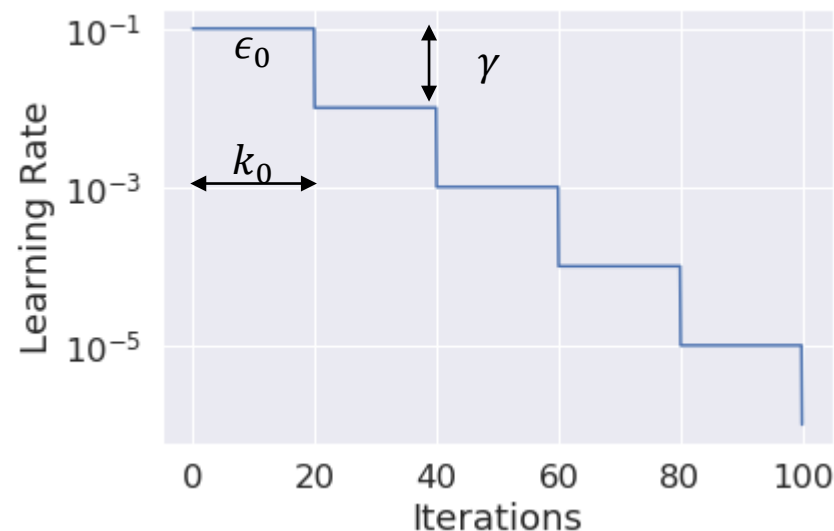
- Convergence is slowed down
- We do not really care about minimizing the empirical risk

The Exponential Schedule



Instead, we often use the exponential schedule. Set

- Initial learning rate: ϵ_0
- #Iterations (or epochs) to wait before decay: k_0
- Decay ratio: γ




Then, the exponential schedule is

$$\epsilon_k = \epsilon_0 \gamma^{\left\lfloor \frac{k}{k_0} \right\rfloor}$$

where $\lfloor \cdot \rfloor$ is the floor function (integer part)



Demo: Training a CIFAR10 Classifier



Adversarial Examples and Adversarial Training

Robustness of Classifiers



Suppose we have trained some classifier that correctly classifies a given input-label pair (x, y)

$$\hat{f}(x) = y$$

We loosely call \hat{f} a robust classifier if

$$\hat{f}(x') = \hat{f}(x) = y$$

for all slightly perturbed inputs x' for which $\|x' - x\| \ll 1$

Are deep neural networks robust?

Finding Adversarial Examples

One way to test robustness is to add noise z . However, there is a more drastic way to find such examples.

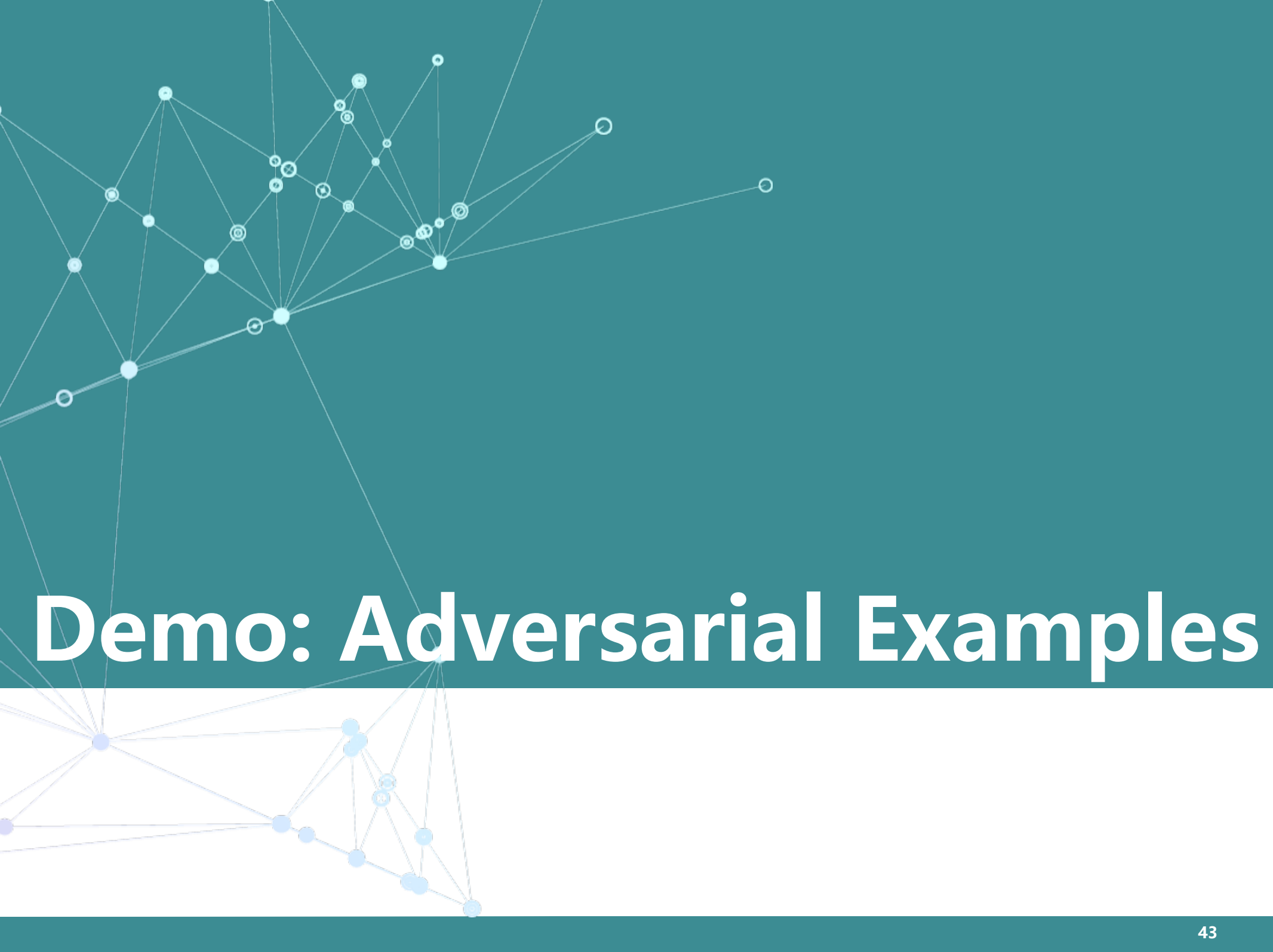
Recall that given a sample (x, y) , we train the model by

$$\hat{\theta} = \operatorname{argmin}_{\theta} L(\hat{y}(x; \theta), y)$$

With trained parameter $\hat{\theta}$, we can find an adversarial example by maximizing

$$x' = \operatorname{argmax}_{z, \|z-x\| \leq \delta} L(\hat{y}(z; \hat{\theta}), y)$$

In other words, we optimize for the worst small perturbation



Demo: Adversarial Examples

Adversarial Training

How can we reduce the effect of adversarial examples?

An effective way is adversarial training, where we solve the mini-max problem

$$\min_{\theta} \max_{z: \|x-z\| \leq \delta} L(\hat{y}(z; \theta), y)$$

The multiple-sample case is similar:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N \max_{z_i: \|x_i - z_i\| \leq \delta} L(\hat{y}(z_i; \theta), y)$$

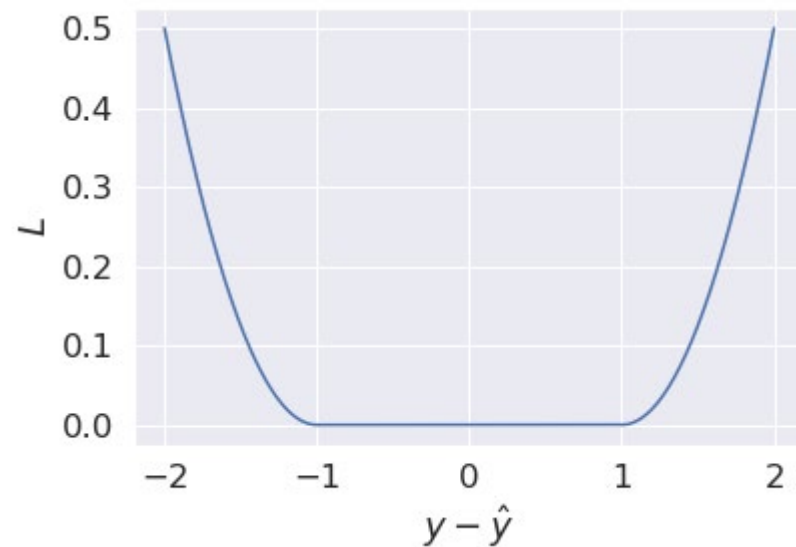
Example: Effect of Adversarial Training

<Lecture Notebook>



Consider 1D example:

- Model: $\hat{y}(x; \theta) = \theta x$
- Data: $x = 1, y = 0$



Loss:

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2} (\hat{y} - y - \text{Sign}(\hat{y} - y))^2 & |\hat{y} - y| > 1 \\ 0 & |\hat{y} - y| \leq 1 \end{cases}$$



Let us apply usual **GD** training starting from $\theta_0 > 1$. It is easy to see that

$$\theta_{\infty}^{(\text{GD})} = 1 \text{ and } L\left(\hat{y}\left(x; \theta_{\infty}^{(\text{GD})}\right), y\right) = 0$$

In this case, we can see that if we do a slight perturbation to the left the loss does not increase, but the same doesn't hold if we perturb to the right!

To see this, consider the solution of the inner maximization

$$\begin{aligned} L_{\text{adv}}(\theta) &:= \max_{z: \|x-z\| \leq \delta} L(\hat{y}(z; \theta), y) = \max_{z: \|1-z\| \leq \delta} \frac{1}{2} (\theta z - \text{Sign}(\theta z))^2 \mathbb{I}_{|\theta z| \geq 1} \\ &= \frac{1}{2} (\theta(1 + \delta) - 1)^2 \mathbb{I}_{|\theta(1+\delta)| \geq 1} \end{aligned}$$

In particular, observe that $L_{\text{adv}}\left(\theta_{\infty}^{(\text{GD})}\right) = \frac{\delta^2}{2} > 0!$

In fact, we see that in this case, we get 0 adversarial loss L_{adv} only if $|\theta| \leq \frac{1}{1+\delta}$.

An Adversarial Training Algorithm



Adversarial Training via FGSM (Fast Gradient Sign Method)

Hyper-parameters: δ (adv size), J (#adv train) ϵ_1, ϵ_2 (learning rates)

For $k = 0, 1, \dots$ **do**

$$\mathbf{z}_0 = \mathbf{x}$$

For $j = 0, 1, \dots, J - 1$ **do**

$$\mathbf{z}_{j+1} = \mathbf{z}_j + \epsilon_2 \text{Sign} \left(\nabla_{\mathbf{z}} L(\hat{y}(\mathbf{z}_j; \boldsymbol{\theta}_k), y) \right)$$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon_1 \nabla_{\boldsymbol{\theta}} L(\hat{y}(\mathbf{z}_J; \boldsymbol{\theta}_k), y)$$



Demo: Adversarial Training and Adversarial Examples

Summary



Today we introduced more ways to improve performance

- Model ensembling (e.g. dropout)
 - Batch normalization
 - Data Augmentation
 - Learning rate decay
 - Adversarial examples and adversarial training
- Model
- Data
- Training