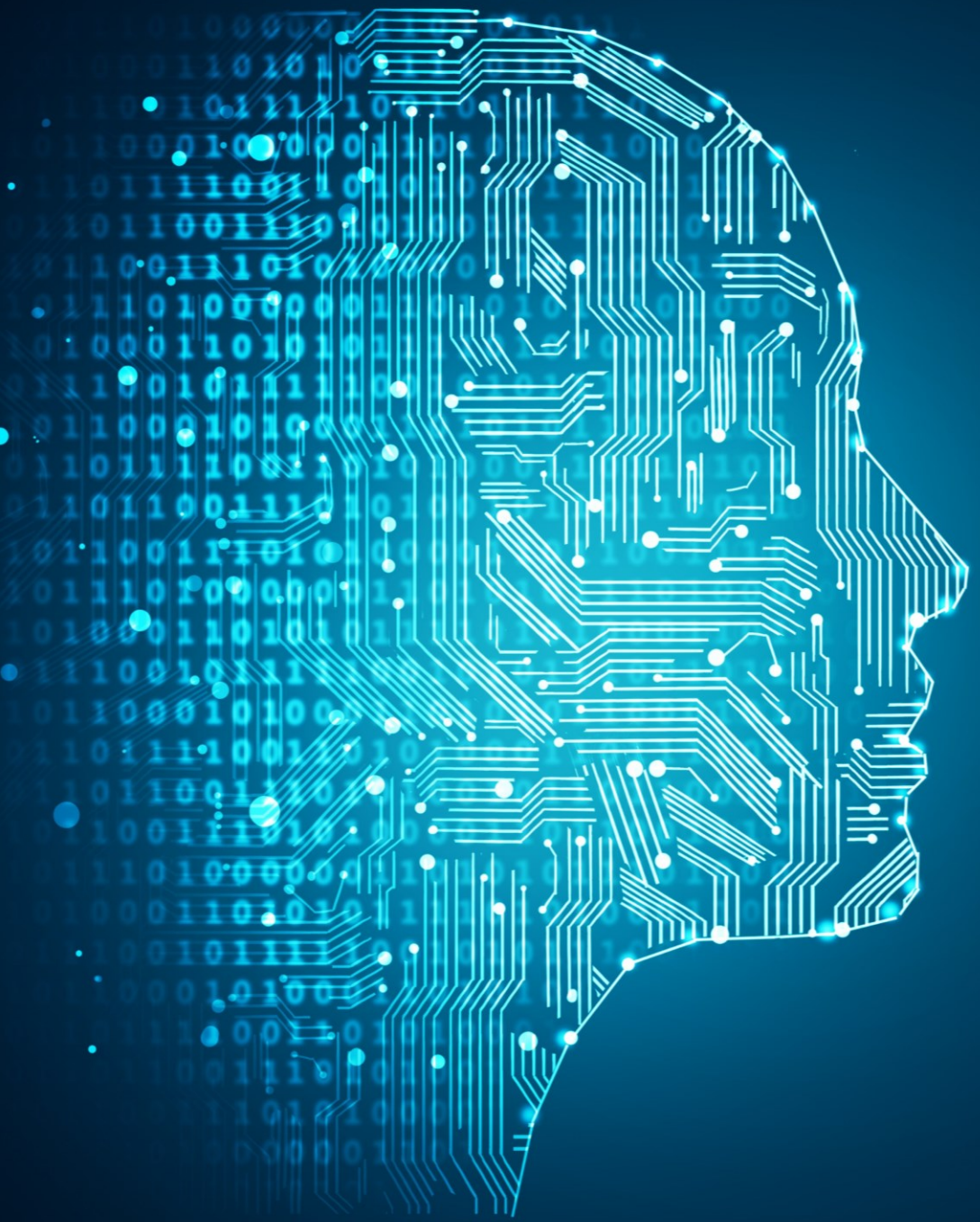


AI6103 Optimization and Regularization (Part 1)

Boyang Li, Albert

School of Computer Science and Engineering

Nanyang Technological University

Quiz

- The quiz will happen at Week 8
- Same time: 2.30pm on Saturday (Mar 12), lasting 30 minutes
- 12 MCQs
- Example Quiz released on NTULearn

Homework Assignment

- Individual assignment. Released on NTULearn.
- Report due on Mar 21, Monday
- You can discuss with other students
  - If you do, please include a sentence in your report like “I discussed this homework with XXX”.
- We take plagiarism very seriously.

Outline

- Gradient Descent
  - Backpropagation
  - Vanishing and Exploding Gradients
  - PyTorch Autograd
  - Stochastic Gradient Descent
- Regularization
  - Weight Decay
  - Representation Learning and Invariance
  - Data Augmentation
- Hyperparameter Tuning

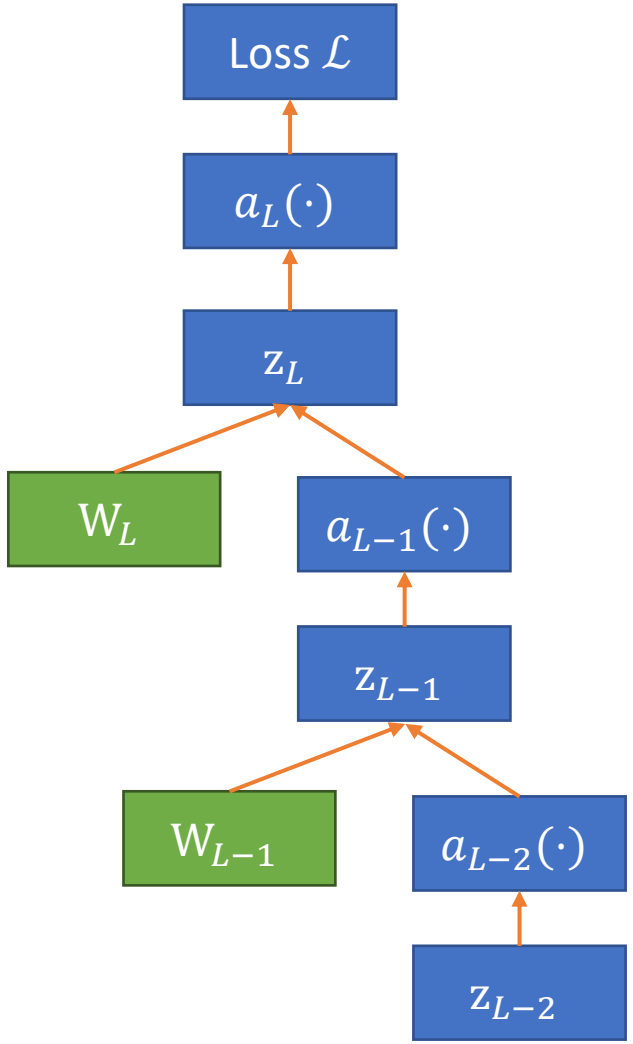
Backpropagation & Autograd

Gradient Descent

- To find the minimum of the loss function $\mathcal{L}(w)$, we update the model parameters w iteratively.

$$w_t = w_{t-1} - \eta \left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w_{t-1}}$$

- η is known as the learning rate in the deep learning literature.



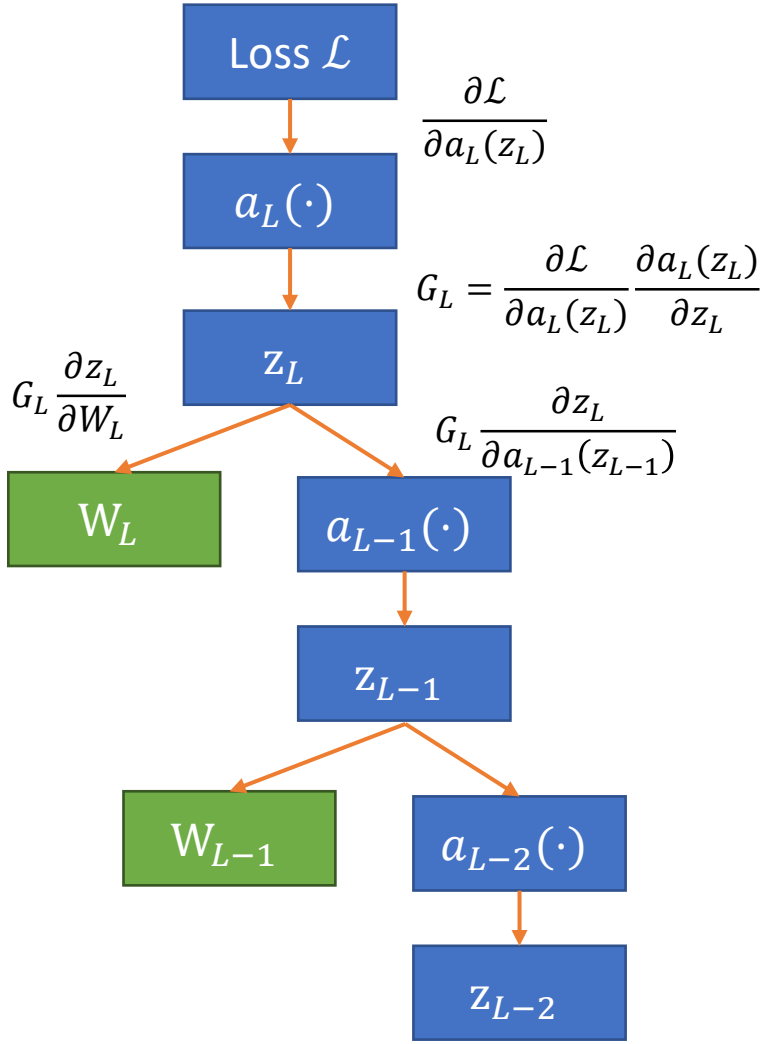
Backpropagation

- An efficient way to compute gradients for parameters W_l

$$\frac{\partial \mathcal{L}}{\partial W_L} = \frac{\partial \mathcal{L}}{\partial a_L(z_L)} \frac{\partial a_L(z_L)}{\partial z_L} \frac{\partial z_L}{\partial W_L}$$

$$\frac{\partial \mathcal{L}}{\partial W_{L-1}} = \underbrace{\frac{\partial \mathcal{L}}{\partial a_L(z_L)} \frac{\partial a_L(z_L)}{\partial z_L}}_{\text{shared}} \frac{\partial z_L}{\partial a_{L-1}(z_{L-1})} \frac{\partial a_{L-1}(z_{L-1})}{\partial z_{L-1}} \frac{\partial z_{L-1}}{\partial W_{L-1}}$$

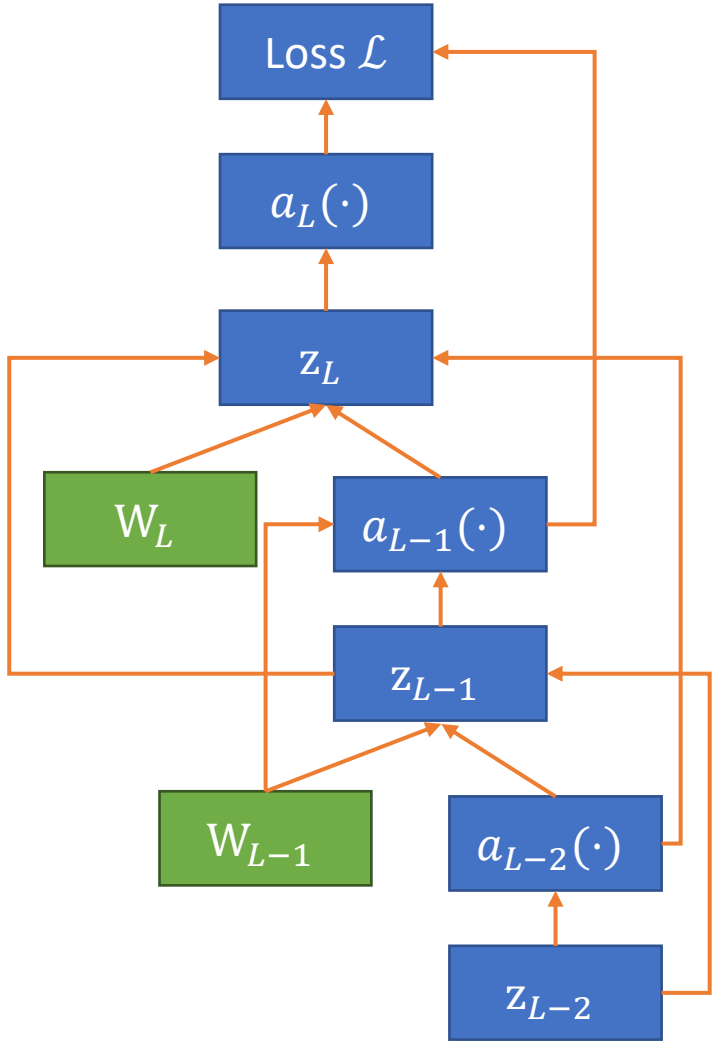
Backpropagation



- An efficient way to compute gradients for parameters W_l

$$\frac{\partial \mathcal{L}}{\partial W_L} = \frac{\partial \mathcal{L}}{\partial a_L(z_L)} \frac{\partial a_L(z_L)}{\partial z_L} \frac{\partial z_L}{\partial W_L}$$

$$\frac{\partial \mathcal{L}}{\partial W_{L-1}} = \underbrace{\frac{\partial \mathcal{L}}{\partial a_L(z_L)} \frac{\partial a_L(z_L)}{\partial z_L}}_{\text{shared}} \frac{\partial z_L}{\partial a_{L-1}(z_{L-1})} \frac{\partial a_{L-1}(z_{L-1})}{\partial z_{L-1}} \frac{\partial z_{L-1}}{\partial W_{L-1}}$$



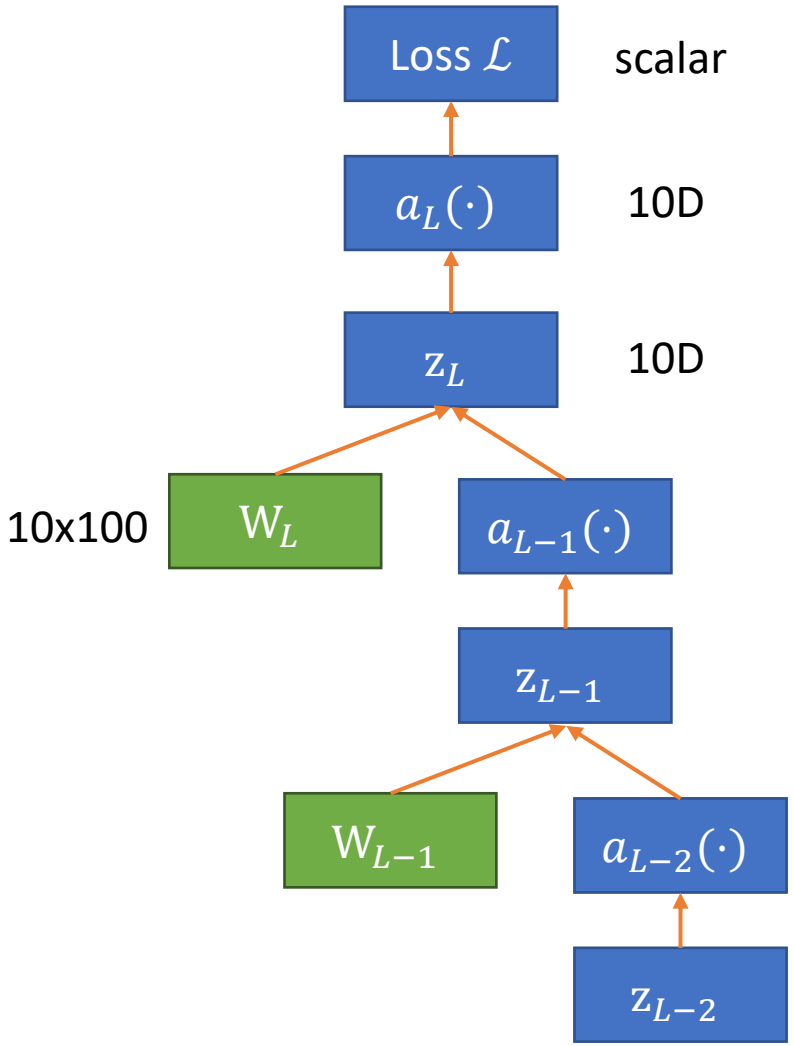
Backpropagation

In modern neural networks, the actual computational graph can be very complex.

But as long as

- the chain rule applies (when does it apply?)
- the graph is a directed acyclic graph,

we can use backpropagation.



Auto-differentiation

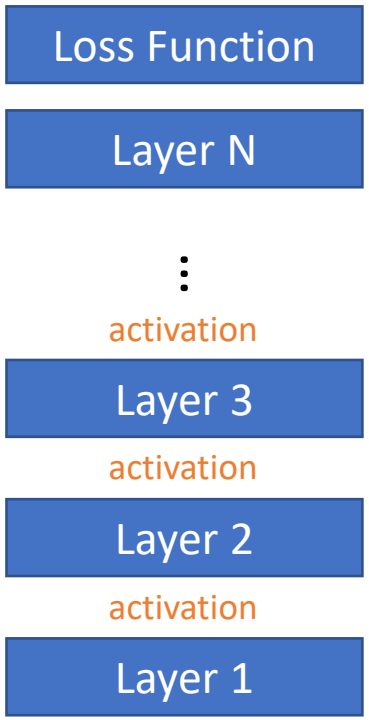
- Autograd performs exact matrix multiplication.

$$\frac{\partial \mathcal{L}}{\partial W_L} = \frac{\partial \mathcal{L}}{\partial a_L(z_L)} \frac{\partial a_L(z_L)}{\partial z_L} \frac{\partial z_L}{\partial W_L}$$

1x1000 1x10 10x10 10x1000

- For each of the operation (activation, multiplication, etc.), autograd knows about how to compute its derivative.
- Autograd is not finite-difference (imprecise), nor symbolic differentiation (no use of symbolic representations).

The Vanishing Gradient Problem



$\alpha(z_2)$

$z_2 = f_2\left(\alpha\left(f_1(x)\right)\right)$

$\alpha\left(f_1(x)\right)$

$f_1(x)$

- The gradient w.r.t the parameter at Layer 1 is computed as
- $$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial [\alpha(z_2)]} \boxed{\frac{\partial \alpha(z_2)}{\partial z_2}} \frac{\partial f_2\left(\alpha\left(f_1(x)\right)\right)}{\partial \alpha\left(f_1(x)\right)} \boxed{\frac{\partial \alpha\left(f_1(x)\right)}{\partial f_1(x)}} \frac{\partial f_1(x)}{\partial w_1}$$

The red boxes denote the derivative of the activation function $\frac{\partial \alpha(x)}{\partial x}$

Tanh and sigmoid both have derivatives less than 1, leading to difficulties in training deep networks.

Vanishing and Exploding Gradient: Math Derivation

We can understand vanishing gradient using singular value decomposition. Slides 12 to 20 are optional for AI6103.

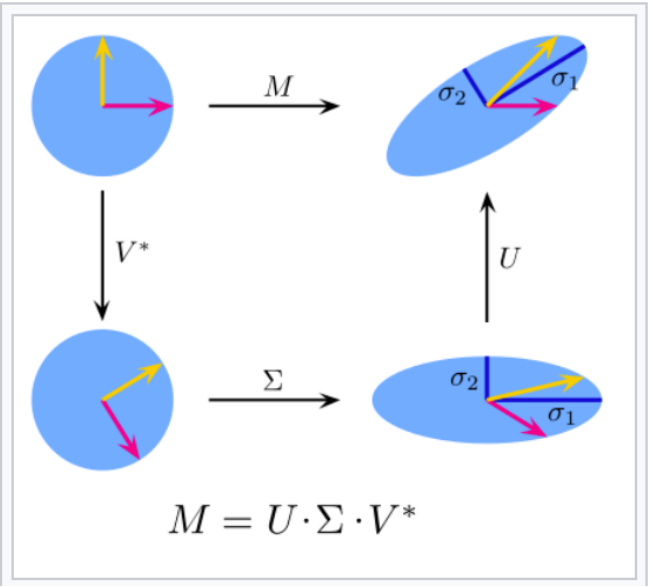
Singular Value Decomposition

- It turns out that every matrix can be written as the combination of such operations.
- Any $m \times n$ real matrix M can be written as

$$M = U\Sigma V^{\top}$$

- where U is an $m \times m$ orthonormal matrix, V is an $n \times n$ orthonormal matrices, and Σ is a $m \times n$ rectangle diagonal matrix with nonnegative values.
- The values on the diagonal of Σ are singular values
- $$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & & \sigma_n \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & & 0 \end{bmatrix}$$

Illustration of the singular value decomposition **UΣV*** of a real 2×2 matrix **M**.
Top: The action of **M**, indicated by its effect on the unit disc *D* and the two canonical unit vectors e_1 and e_2.
Left: The action of **V***, a rotation, on *D*, e_1, and e_2.
Bottom: The action of **Σ**, a scaling by the singular values σ_1 horizontally and σ_2 vertically.
Right: The action of **U**, another rotation.



The order of singular values

- Any $m \times n$ real matrix M can be written as

$$M = U\Sigma V^\top = [u_1\ u_2\ u_3\ \cdots u_n] \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & & \sigma_n \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & & 0 \end{bmatrix} \begin{bmatrix} v_1^\top \\ v_2^\top \\ v_3^\top \\ \vdots \\ v_m^\top \end{bmatrix}$$

$$= \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^\top$$

This sum does not care about the ordering of σ

- where u_i are the columns of U and v_i are the columns of V.
- We can freely switch the positions of singular values, as long as we keep (σ_i, u_i, v_i) together
- Typically, we sort them in either descending or ascending order

SVD for Low-rank Approximation

- Any $m \times n$ real matrix M can be written as

$$M = U\Sigma V^{\top} = [u_1\ u_2\ u_3\ \cdots u_n]\begin{bmatrix}\sigma_1 & 0 & \cdots & 0\\ 0 & \sigma_2 & \cdots & 0\\ \vdots & \vdots & \cdots & \vdots\\ 0 & 0 & & \sigma_n\\ \vdots & \vdots & \cdots & \vdots\\ 0 & 0 & & 0\end{bmatrix}\begin{bmatrix}v_1^{\top}\\ v_2^{\top}\\ v_3^{\top}\\ \vdots\\ v_m^{\top}\end{bmatrix}$$

$$= \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^{\top}$$

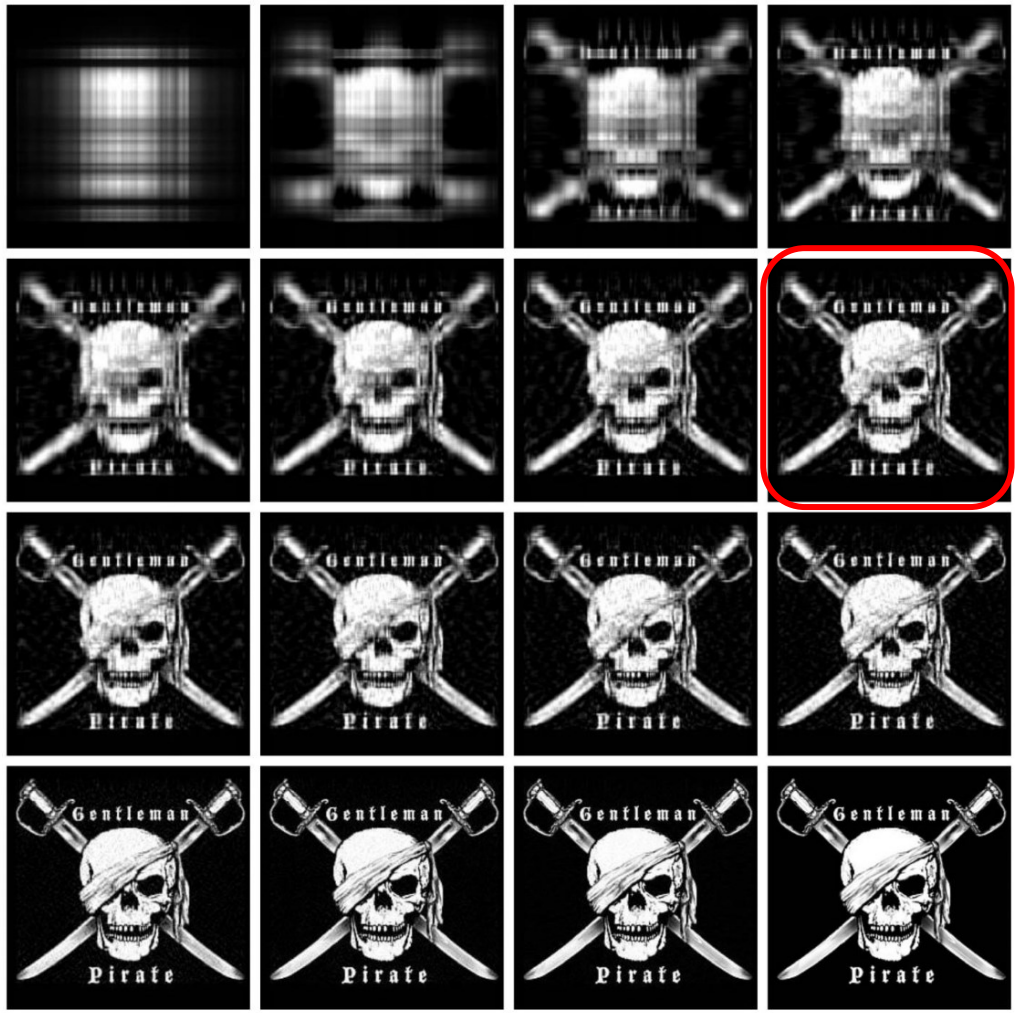
- where u_i are the columns of U and v_i are the columns of V.
- Since u_i and v_i are of unit length. If σ_i is really small, then $\sigma_i u_i v_i^{\top}$ does not matter much.
- Thus, we can approximate M using K largest σ_i and the corresponding sum.
- Reducing space complexity from O(MN) to O(K(M+N))

SVD for Low-rank Approximation

- Any $m \times n$ real matrix M can be written as

$$M = U\Sigma V^{\top} = \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^{\top}$$

- Thus, we can approximate M using K largest σ_i and the corresponding sum.
- Reducing space complexity from O(MN) to O(K(M+N))
- For the image on the right, # pixels = 59,000
- Using 14 singular values, # values stored = 6,804
- More examples: http://timbaumann.info/svd-image-compression-demo/



K=14

Figure 3.1: Image size 250x236 – modes used {{1,2,4,6},{8,10,12,14},{16,18,20,25},{50,75,100,original image}}

Brady Mathews. Image Compression using Singular Value Decomposition (SVD) 2014

Vanishing Gradient

- The gradient is computed as matrix multiplications. How does that affect gradients in deep networks?
- Compare $\|A\boldsymbol{x}\|$ and $\|\boldsymbol{x}\|$
- By SVD, $A = U\Sigma V^\top$ where U and V are orthogonal matrices. The singular values $\sigma_1 > \cdots > \sigma_n > 0$.
- $\|V^\top \boldsymbol{x}\| = \|\boldsymbol{x}\|$ because rotation, permutation, and reflection do not change vector length.

- Let $\|\boldsymbol{x}\| = c$. Assuming A is full rank, we can write

$$\boldsymbol{x} = c(a_1\boldsymbol{v}_1 + a_2\boldsymbol{v}_2 + \cdots + a_n\boldsymbol{v}_n)$$

$$\boldsymbol{x}^\top\boldsymbol{x} = c^2(a_1^2 + a_2^2 + \cdots + a_n^2)$$

- We know that $\boldsymbol{x}^\top\boldsymbol{x} = c^2$
- So $a_1^2 + a_2^2 + \cdots + a_n^2 = 1$

$$\Sigma V^\top \boldsymbol{x} = \Sigma \begin{bmatrix} \boldsymbol{v}_1^\top\boldsymbol{x} \\ \boldsymbol{v}_2^\top\boldsymbol{x} \\ \vdots \\ \boldsymbol{v}_n^\top\boldsymbol{x} \end{bmatrix} = c\Sigma \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

Vanishing Gradient

- We want to compare $\|A\boldsymbol{x}\|$ and $\|\boldsymbol{x}\|$
- By SVD, $A = U\Sigma V^\top$ where U and V are orthogonal matrices. The singular values $\sigma_1 > \cdots > \sigma_n > 0$.
- Let $\|\boldsymbol{x}\| = c$. Assuming A is full rank, we can write

$$\boldsymbol{x} = c(a_1\boldsymbol{v}_1 + a_2\boldsymbol{v}_2 + \cdots + a_n\boldsymbol{v}_n)$$
$$\boldsymbol{x}^\top\boldsymbol{x} = c^2(a_1^2 + a_2^2 + \cdots + a_n^2)$$

- We know that $\boldsymbol{x}^\top\boldsymbol{x} = c^2$
- So $a_1^2 + a_2^2 + \cdots + a_n^2 = 1$

$$\Sigma V^\top\boldsymbol{x} = \Sigma\begin{bmatrix}\boldsymbol{v}_1^\top\boldsymbol{x}\\ \boldsymbol{v}_2^\top\boldsymbol{x}\\ \vdots\\ \boldsymbol{v}_n^\top\boldsymbol{x}\end{bmatrix} = c\Sigma\begin{bmatrix}a_1\\ a_2\\ \vdots\\ a_n\end{bmatrix}$$

$$\Sigma V^\top\boldsymbol{x} = c\begin{bmatrix}a_1\sigma_1\\ a_2\sigma_2\\ \vdots\\ a_n\sigma_n\end{bmatrix}$$

$$\|\Sigma V^\top\boldsymbol{x}\|^2 = c^2(a_1^2\sigma_1^2 + a_2^2\sigma_2^2 + \cdots a_n^2\sigma_n^2)$$

- Therefore, $\|\Sigma V^\top\boldsymbol{x}\| \leq c\sigma_1$
- Equality holds when $a_1 = 1, a_2 = \cdots = a_n = 0$

Vanishing Gradient

- We want to compare $\|A\boldsymbol{x}\|$ and $\|\boldsymbol{x}\|$
- By SVD, $A = U\Sigma V^\top$ where U and V are orthogonal matrices. The singular values $\sigma_1 > \cdots > \sigma_n > 0$.
- $\|A\boldsymbol{x}\| \leq \sigma_1 \|\boldsymbol{x}\|$
- Equal sign is taken when $\boldsymbol{x} = \boldsymbol{v}_1 \|\boldsymbol{x}\|$

- What happens when multiple matrices all have maximum singular values less than 1?

$$A_L \ldots A_3 A_2 A_1 \boldsymbol{x}$$

- The magnitude of the product will diminish as L increases.

Vanishing Gradient

- What happens when multiple matrices all have maximum singular values less than 1?

$$A_L \dots A_3 A_2 A_1 \boldsymbol{x}$$

- The magnitude of the product will diminish as L increases.

$$\frac{\partial \mathcal{L}}{\partial W_{L-1}} = \frac{\partial \mathcal{L}}{\partial a_L(z_L)} \frac{\partial a_L(z_L)}{\partial z_L} \frac{\partial z_L}{\partial a_{L-1}(z_{L-1})} \frac{\partial a_{L-1}(z_{L-1})}{\partial z_{L-1}} \frac{\partial z_{L-1}}{\partial W_{L-1}}$$

- $\frac{\partial a_L(z_L)}{\partial z_L}$ is a diagonal matrix with component-wise derivative on the diagonal. For both sigmoid and tanh functions, the maximum derivative values < 1.
- Thus, the gradient diminishes
- Impossible to train very deep networks.

Exploding Gradient

- For non-singular A, $\sigma_{\min}\|\boldsymbol{x}\| \leq \|A\boldsymbol{x}\| \leq \sigma_{\max}\|\boldsymbol{x}\|$
- If many matrices have $\sigma_{\min} > 1$, their product will grow very large.
- This also makes optimization difficult, because we will take steps that are too large.

Practical Solutions

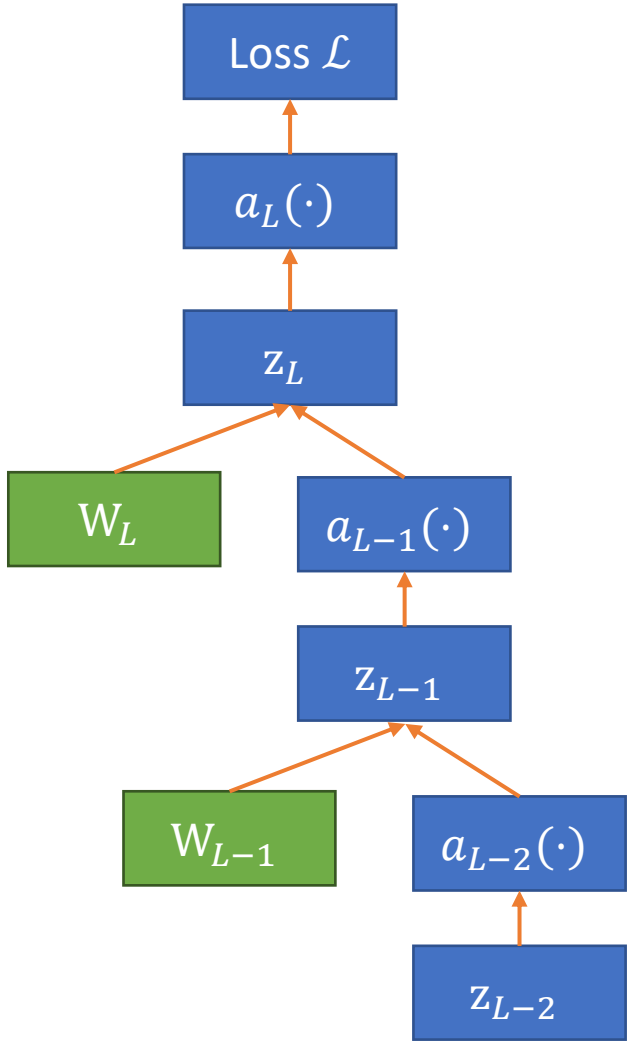
Vanishing Gradient

- In modern deep learning
  - Different activation functions, which have gradients $= 1$.
  - Optimization methods like Adam, which adaptively scales gradients.
  - Skip connections, which create different paths for gradients to flow.
  - In RNN (covered later), long short-term memory

Exploding Gradient

- **Gradient Clipping**: set a maximum value for the gradient norm. If the gradient norm exceeds that value, scale it back.
- Vanishing gradient is usually a more difficult problem.
- If the gradient is too large, we can always take small steps.
- If there is no gradient information, there is not much we can do.

Programming in PyTorch



PyTorch AutoGrad

- PyTorch records the operations performed on Tensors as the program executes.
- This is known as Dynamic Graph, which is a directed acyclic graph.
- At every training iteration, the graph is built from scratch.
- This allows arbitrary Python control flow and extreme flexibility.
- Contrast this with the static graph approach, which has to build the graph before any input can be fed to it.

PyTorch AutoGrad

- PyTorch must keep every tensor with a reference in memory, even if you never use it afterwards.
- A drawback of the dynamic graph approach
- In static graphs, it is easy to see what tensors are no longer needed.

```
A = func_a() # computes A
B = func_b() # computes B
C = torch.mm(A, B)
# A and B are kept in memory even
if you never use them again.
```

PyTorch AutoGrad

- In order to save precious GPU memory, minimize named references in your code.
- The `del` statement tells Python that a variable or memory reference is no longer needed.
- When there are no references pointing to a memory chunk, it can be recycled.

```
A = func_a() # computes A
B = func_b() # computes B
C = torch.dot(A, B)
# delete A and B
del A, B
# another example
A = func_a()
B = A
del A
# the memory chuck is still being referenced by B
```

PyTorch AutoGrad

- In order to save precious GPU memory, minimize named references in your code.
- PyTorch will keeps track of variable updates so that gradients can be computed properly.
- A lot of things happening behind the curtain, which may be controlled by the user.

```
A = func_a() # computes A
A = func_b() + A # updates A
A = torch.mm(A, B)
# updates A again
# this pattern avoids creating
multiple references altogether
```

Freeze Parameters

- Set the requires_grad attribute to False will prevent a tensor from being updated with SGD.
- Local control of autograd.

```
A = Tensor.empty([2, 2])
A.requires_grad_(False)
```

no-grad Mode

- In the *torch.no_grad()* context, no autograd will be performed.
- The result of every computation will have *requires_grad=False*, even when the inputs have *requires_grad=True*.
- Accelerates execution

```
x = torch.tensor([1], requires_grad=True)
with torch.no_grad():
    y = x * 2
print(y.requires_grad) # False
```

`inference` Mode

- The *`torch.inference_mode()`* context provides even more aggressive removal of autograd
- The result of every computation will have *`requires_grad=False`*, even when the inputs have *`requires_grad=True`*.
- These tensors cannot be used in the autograd mode even if it is enabled later on.

```
x = torch.tensor([1], requires_grad=True)
with torch.inference_mode():
    y = x * 2
print(y.requires_grad) # False
```

Gradient Descent

- GD computes the gradient on the loss function

$$\mathcal{L} = \frac{1}{N}\sum_i \ell(x^{(i)}, y^{(i)}, w)$$

Summation over all training data

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \frac{1}{N}\sum_i \frac{\partial \ell\left(x^{(i)}, y^{(i)}, \boldsymbol{w}\right)}{\partial \boldsymbol{w}}$$

- When there are too many training data, the above average is too expensive to compute.
- We use only a small subset randomly sampled from the training dataset to compute the gradient.

Stochastic Gradient Descent

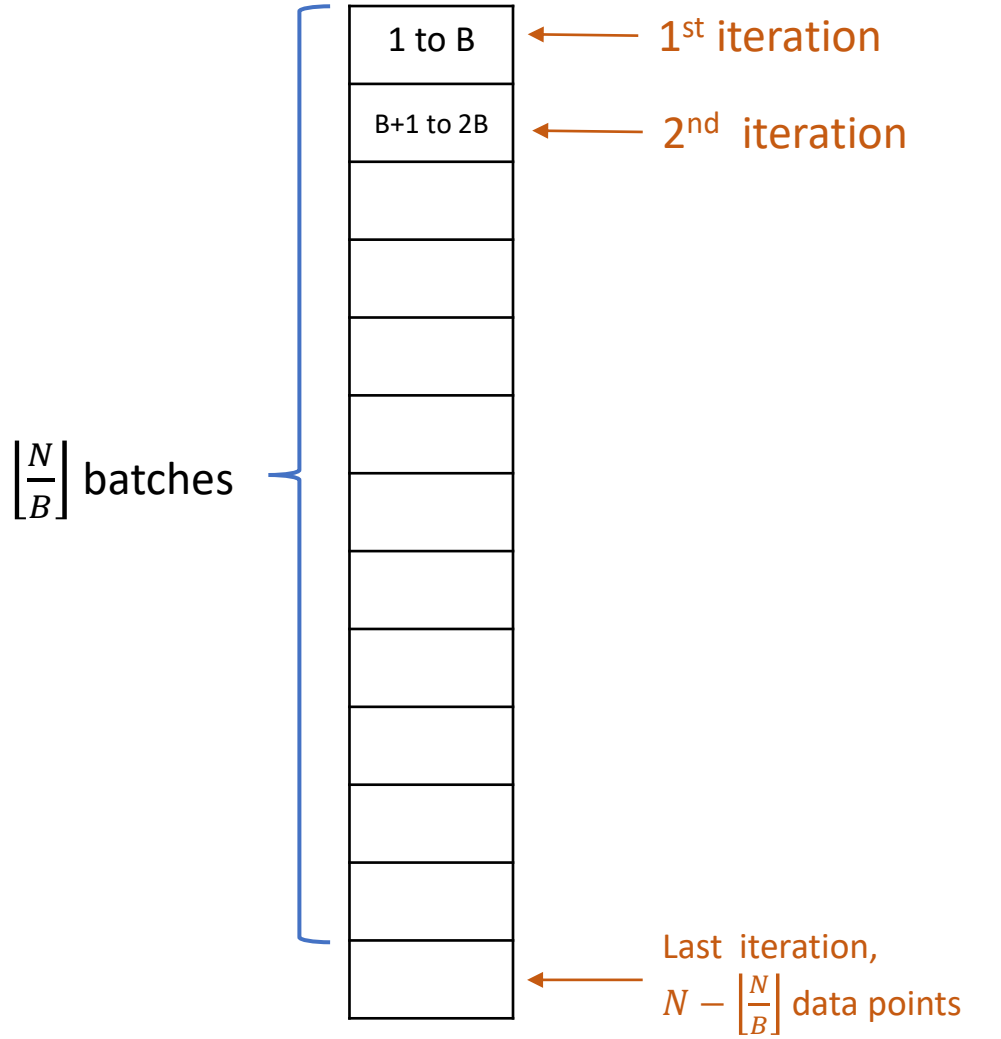
$$\hat{g}_{\text{mini-batch}} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\frac{\partial \ell\left(x^{(i)}, y^{(i)}, \boldsymbol{w}\right)}{\partial \boldsymbol{w}} = \bar{g} + \epsilon$$

- The mini-batch estimate of the gradient contains some random noise. Is that bad?
  - Bias is bad. We need to shuffle datasets to get unbiased estimates.
  - Variance may not be too bad. As long as the angle between the estimated gradient $\hat{g}$ and true gradient g is less than 90 degrees, we are still roughly in the right direction.
  - Getting more accurate gradient means a bigger batch, which can be more expensive

Implementation

1. Shuffle all training data into a random permutation.
2. Start from the beginning of the shuffled sequence.
3. In every iteration, take the next B data points, which form a mini-batch, and perform training on the mini-batch.
4. When the sequence is exhausted, go back to Step 1. This is one epoch of training.

Random shuffling is necessary for unbiased gradient estimates.



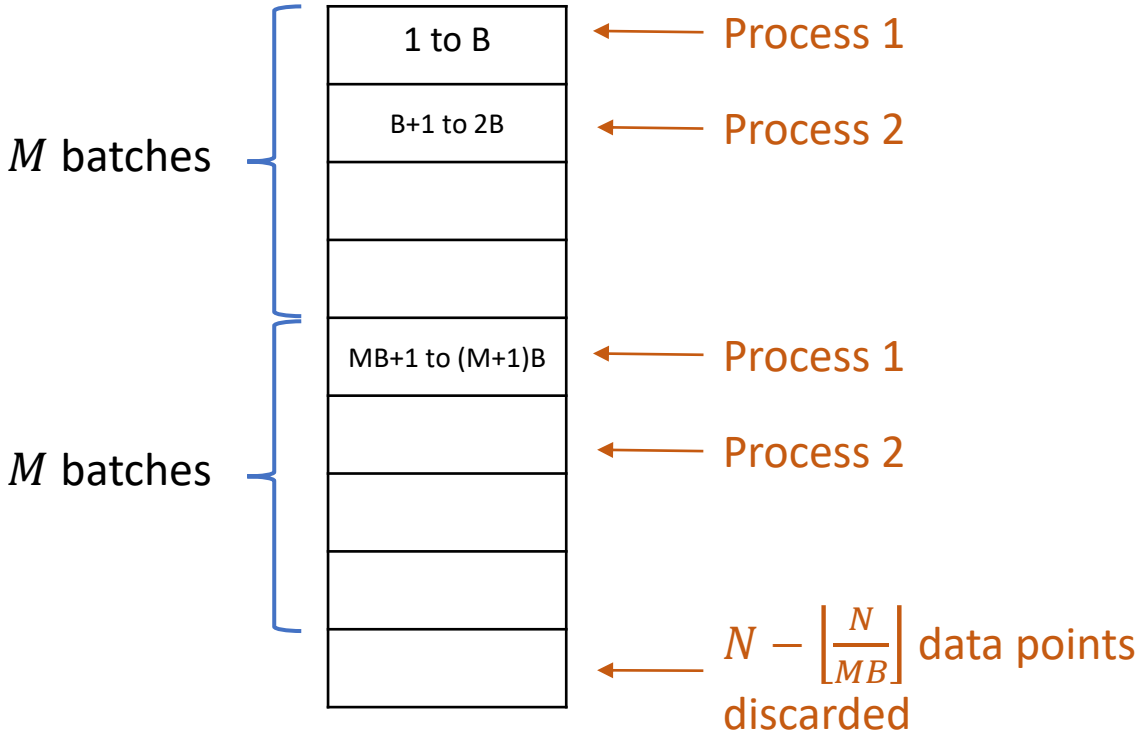
The last batch with less than B data points may be discarded if small batches cause problems (e.g., for BatchNorm)

Pseudo-random Number Generator (PNG)

- Pseudo-random number generators are not really “random”.
  - What is really random? A philosophical discussion that we will not get into.
- It is a completely deterministic function of the random seed.
  - Always generates the same sequence if the seed is the same.
  - A.k.a deterministic random bit generator
- However, the sequence of numbers satisfy certain statistical properties that they can be considered random if the seed is not known.
  - Do not change seeds if you want good randomness. Use the sequence.
- If we give the same seed to two generators w/ the same algorithm, they will create the same sequence.
  - We can align multiple processes or multiple training sessions.

Synchronous Training on Multiple GPUs / Nodes

- Multiple processes (total number = M) use PNGs sharing the same seed when shuffling the dataset.
- Based on its process id, a process uses one batch of size B every M batches.
- The gradients from M processes are averaged and the model is updated once (synchronization).
- Equivalent to a global batch size of MB



Batch Size and Learning Rate

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell\left(x^{(i)}, y^{(i)}, \boldsymbol{w}\right)}{\partial \boldsymbol{w}}$$

- We increase the batch size together with the learning rate
- The two things usually are proportional (i.e., linear relationship)
- Parallel hardware (e.g., multiple GPUs) can be effectively utilized using large batch sizes
- However, at one point, the scaling fails and large batches lead to performance deterioration.
- We will analyze this in detail when we discuss optimization.

Learning Rate Schedule

- Typically, we gradually decrease the learning rate during the optimization.
- Commonly used learning rate schedules
  - Exponential
  - Step
  - Cosine
  - Plateau (dynamic schedule)
  - Many more…

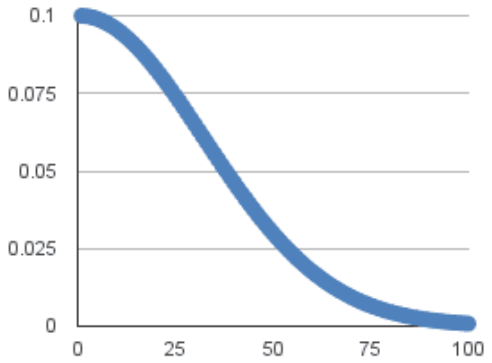
Learning Rate Schedule

- Exponential
  - $\eta_t = \alpha^t \eta_0, 0 < \alpha < 1$
- Step
  - Decrease by the factor α at predefined iteration / epoch

- This step function is popular in the training of convolutional networks for image recognition.
  - Total number of epochs = T
  - Multiply η by 1/10 at $\frac{T}{2}$, and another 1/10 at $\frac{3T}{4}$

Learning Rate Schedule

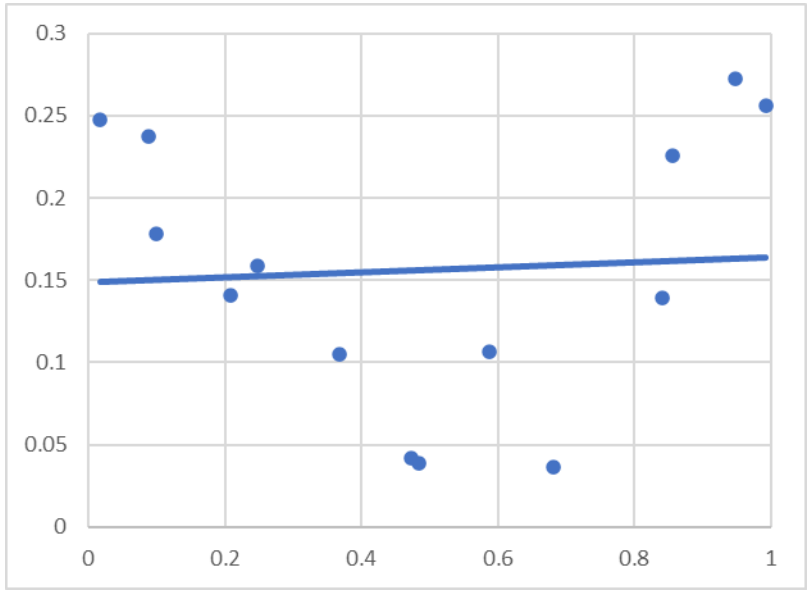
- Cosine
  - $\eta_t = \eta_{min} + \frac{1}{2}(\eta_{\max} - \eta_{min})\left(1 + \cos\frac{t}{T}\pi\right)$
  - A half cycle of the cosine function
  - Slow decrease at the beginning and at the end, but fast drop in the middle.



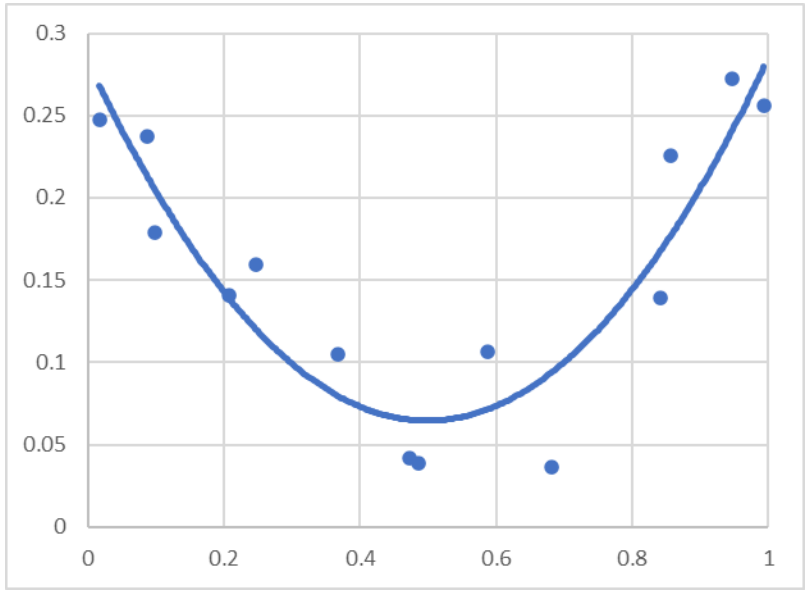
- Plateau
  - Decrease the learning rate by α when the training loss does not decrease by at least β in S epochs.
  - Useful as an exploration strategy when no prior knowledge is available.
  - May create difficulties for reproduction.

Regularization

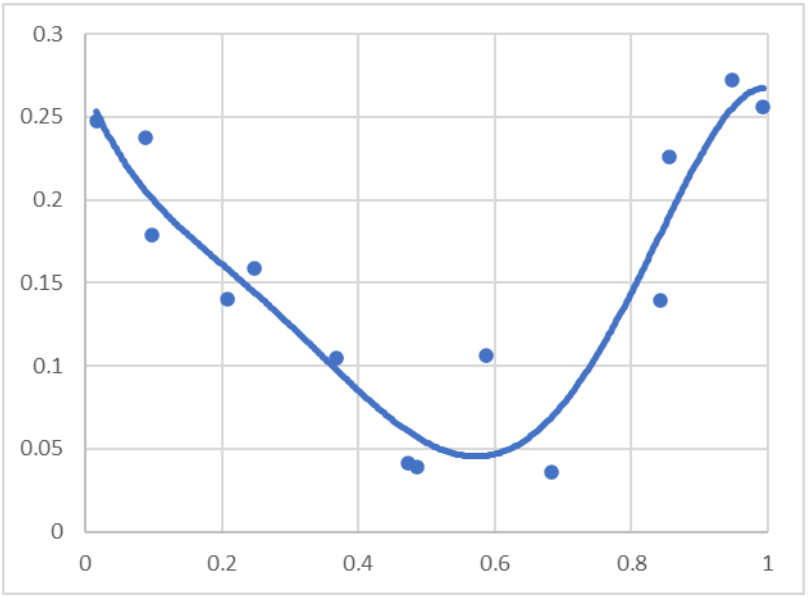
Underfitting vs. Overfitting



Not describing the data



Describing the data well



Taking the data too literally

Optimization vs. Regularization

- Optimization: fitting the data
  - Minimize the empirical loss on the training set
  - Maximize performance on the training set
- Regularization: avoid overfitting the data
  - Improve performance on future data (test set), often at the cost of training loss.
- Traditionally, these two are considered to be opposite to each other.
- In deep learning, not always.

Conventional Intuition

- Underfitting: The model is too limited and cannot represent the function that generates the data.
- Overfitting: The model is too expressive. It can represent the data-generating function as well as the noise in the data.

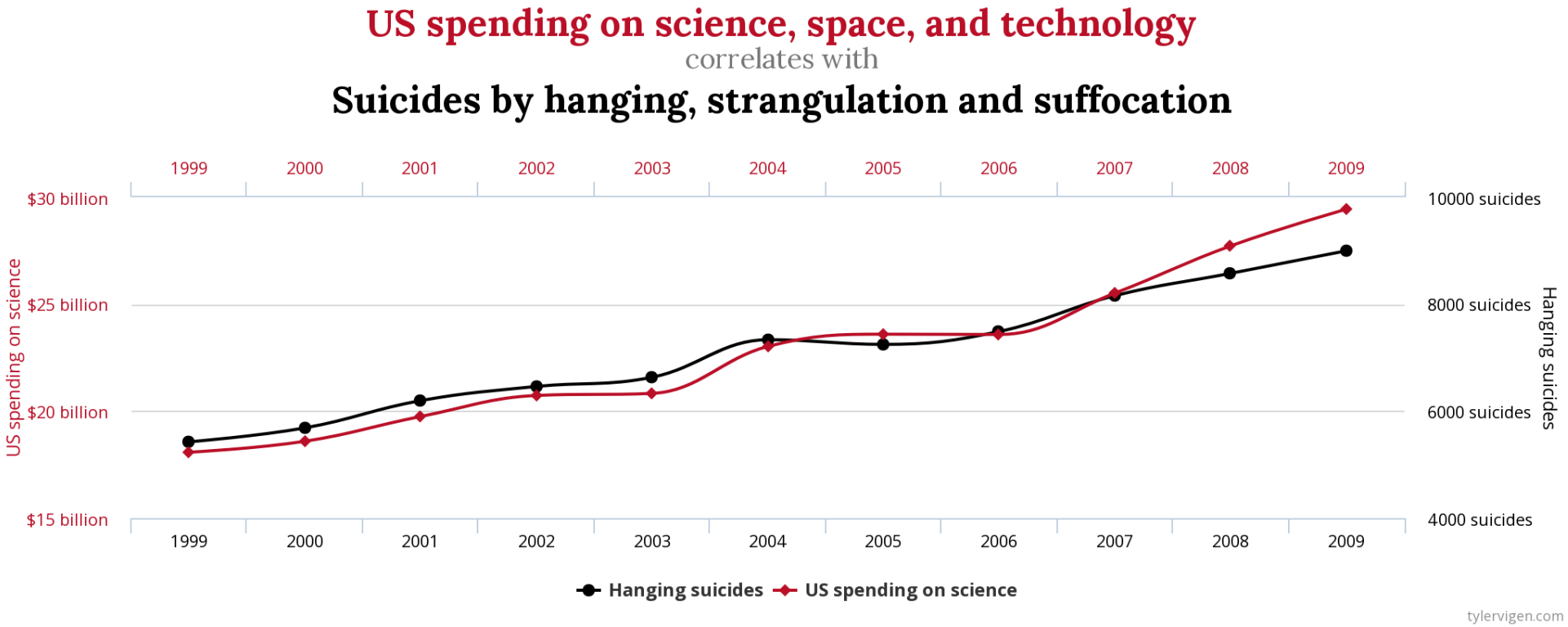
Deep Learning Intuition

- Underfitting: Not learning enough from the observed data.
  - Even if the model is large, the optimization could be poor.
- Overfitting: The observed data contain some random errors, some idiosyncrasies that do not apply to the general population. Overfitting is to learn from such noise.
  - Example: Learning from a weird horse image and thinking that it describes many real-world horses.
- DL deals with more complex semantics than traditional ML.
  - What is noise using one set of features is signal under another set of features.
  - Quality of feature representation matters a lot.
  - Sometimes increasing capacity could reduce generalization error!

Deep Learning is Representation Learning

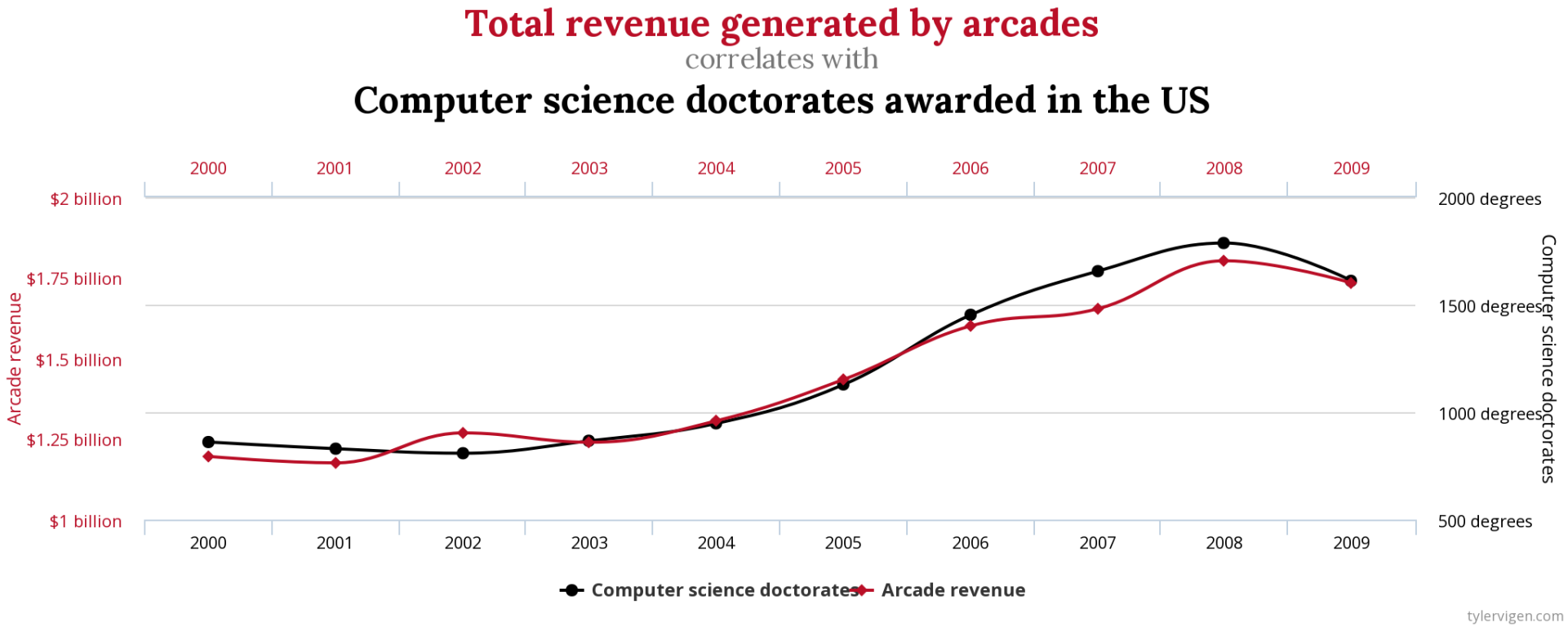
- Shallow models work with existing features
  - E.g., logistic regression: $\hat{y} = \sigma(\boldsymbol{\beta}^\top \boldsymbol{x})$
- Deep models can be seen as learning features
  - MLP: $\hat{y} = \sigma\left(w_L \cdots \sigma\left(W_2 \sigma(W_1 \boldsymbol{x})\right)\right)$
  - Let $\boldsymbol{z} = \sigma\left(W_{L-1} \cdots \sigma\left(W_2 \sigma(W_1 \boldsymbol{x})\right)\right)$, we have $\hat{y} = \sigma(\boldsymbol{w}_L^\top \boldsymbol{z})$
- Learning the right features is critical

Bad Feature (correlation 99.79%)



Guaranteed: low training error, high test error on future data

Bad Feature (correlation 98.51%)



Guaranteed: low training error, high test error on future data

Feature Extraction: Which is more robust?

- What is the make of the car?

If the dataset comes from a single year, these features can be very effective, but they will stop working if Toyota ever changes its design.

Deep Learning is Representation Learning

- Many regularization techniques in DL aim to improve the quality of features extracted by the deep network.
  - MixUp
  - Data Augmentation
  - Pretraining is effective against overfitting
- Covered in this and later lectures

Optimization vs. Regularization in DL

- The most important issue in deep learning.
- Optimization is hard in DL
  - First-order optimization is hard.
  - Second-order methods are great but we can't use them.
- Regularization is hard in DL
  - Overparameterized deep nets are capable of memorizing every training instance.

Symptoms and Diagnosis

- Effective trial-and-error requires understanding accurate diagnosis and the correct remedy
- Underfitting
  - Large training loss, large validation / test loss
  - The loss stagnates early in training
  - Solution: Improve optimization; increase model capacity
- Overfitting
  - Large gap between training set performance and val/test performance
  - Usually gap in loss values too
  - Solution: Apply stronger regularization (first priority); decrease model capacity;
- In deep learning, zero overfitting does not always give best performance.

Weight Decay

- For a loss function $\mathcal{L}(\boldsymbol{w})$, the GD update is
- $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_{\boldsymbol{w}_{t-1}} \mathcal{L}(\boldsymbol{w})$ another way to write the derivative
- We can add L2 regularization, just like we did in Ridge Regression

Linear vs. Ridge Regression

- Data points $\left(\boldsymbol{x}^{(1)}, y^{(1)}\right), \left(\boldsymbol{x}^{(2)}, y^{(2)}\right), \ldots, \left(\boldsymbol{x}^{(n)}, y^{(n)}\right)$
- Loss function for linear regression (Ordinary Least Squares).

$$\mathcal{L}_{\mathrm{OLS}}(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

- When we don't have enough data, we use Ridge Regression, which adds a scaled version of $\boldsymbol{\beta}^\top\boldsymbol{\beta} = \|\boldsymbol{\beta}\|^2$ to the loss

$$\mathcal{L}_{RR}(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \frac{1}{n}\lambda\boldsymbol{\beta}^\top\boldsymbol{\beta}$$

Weight Decay

- Old loss function $\mathcal{L}(\boldsymbol{w})$
- Old update rule:

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_{\boldsymbol{w}_{t-1}} \mathcal{L}(\boldsymbol{w}_{t-1})$$

- Adding L2 regularization leads to a new loss

$$\mathcal{L}'(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}) + \frac{1}{2}\lambda \|\boldsymbol{w}\|^2$$

- New update rule

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_{\boldsymbol{w}_{t-1}} \mathcal{L}(\boldsymbol{w}_{t-1}) - \eta\lambda \boldsymbol{w}_{t-1}$$

Weight Decay

- Old loss function $\mathcal{L}(\boldsymbol{w})$
- Old update rule:

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_{\boldsymbol{w}_{t-1}} \mathcal{L}(\boldsymbol{w}_{t-1})$$

- Adding L2 regularization leads to a new loss

$$\mathcal{L}'(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}) + \frac{1}{2}\lambda \|\boldsymbol{w}\|^2$$

- New update rule

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_{\boldsymbol{w}_{t-1}} \mathcal{L}(\boldsymbol{w}_{t-1}) - \eta\lambda \boldsymbol{w}_{t-1}$$

We can keep the old update and subtract a small portion ($\eta\lambda$) from $\boldsymbol{w}$ at every iteration

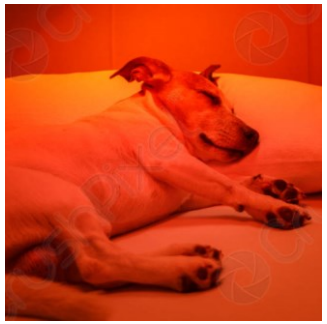
Weight Decay

- Intuitively, L2 regularization and weight decay favors model parameters $\boldsymbol{w}$ that have small norm $\|\boldsymbol{w}\|$.
- Larger norm could represent a larger range of functions.
- Thus, this regularization puts a soft constraint on the complexity of the learned model.
- Limited model complexity may lead to better generalization (performance on data unseen during training).

Representation Learning and Invariance

Same object, different pixel values

Black and white,
Low light



Red light

Viewpoint
change

Uncommon
Background


White and Gold vs. Blue and Black

The human vision system is highly invariant

- It's built-in deep in our brain
- It offers evolutionary advantages
- It's hard to consciously override it.
- We are wired to not see certain things!

Invariance in Speech Perception

- Clearly, using visual information to guide speech perception is an advantage, especially when you don't hear too clearly due to background noise or other reasons.
- This can be used against us, too, creating the illusion.
- You are not even consciously aware of your brain's processing!

Information processing in our mind often happens at the subconscious level.

That is also why we can't write them down as rules using self-inspection.

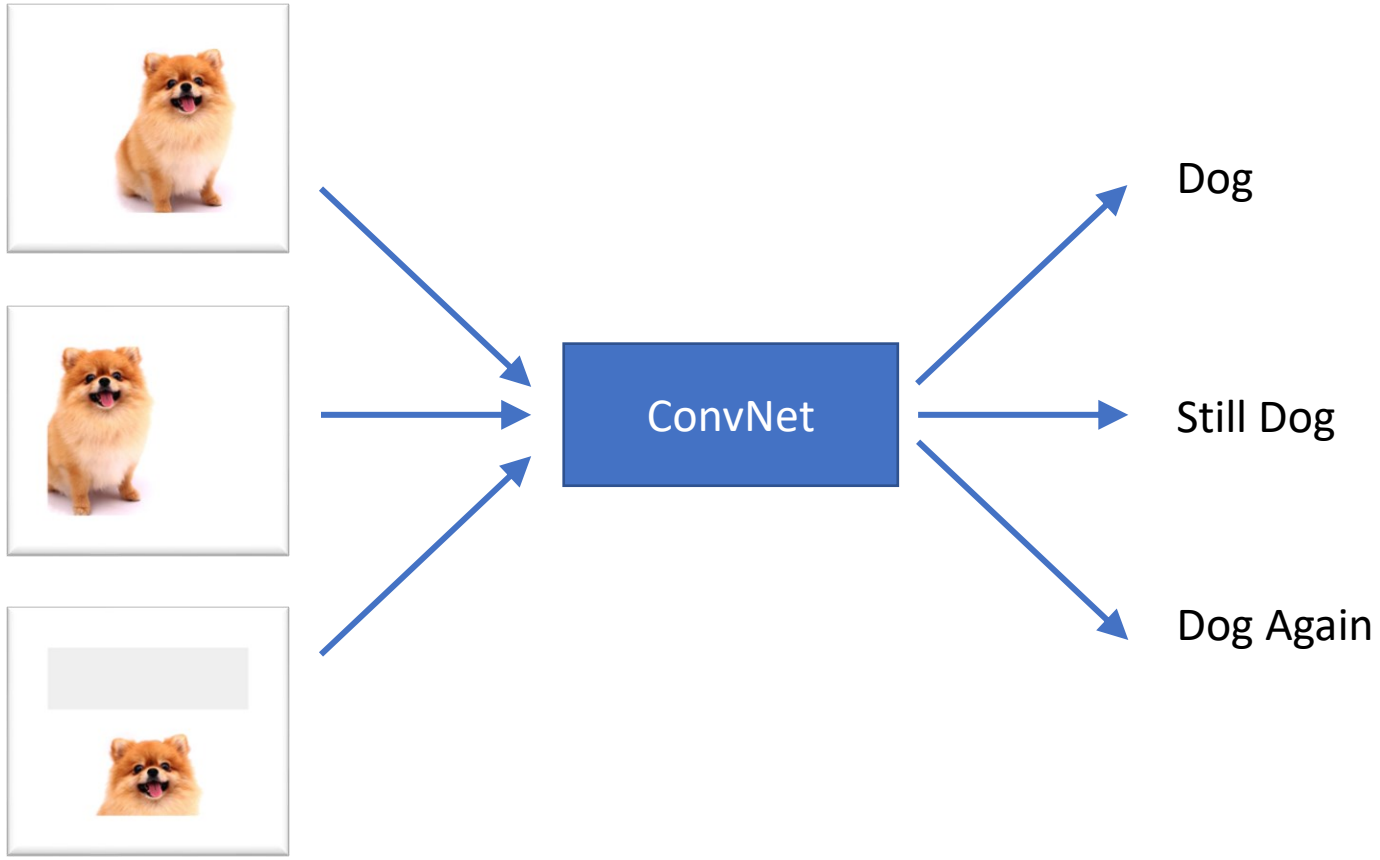
Invariance vs. Sensitivity

- We look for good vector representations of your input (image, text, audio, etc.)
- Good representations should be sensitive to the factors we care about for the task at hand
  - For animal classification, the animal species
  - For audio transcription, the phonemes (smallest unit of speech distinguishing words)
- Good representations should be invariant to factors irrelevant to the task.
  - For animal classification, it should be invariant to location of the animal, lighting, background color, hats, sunglasses, etc.
  - For location of lighting source, it should be sensitive to lighting

Invariance

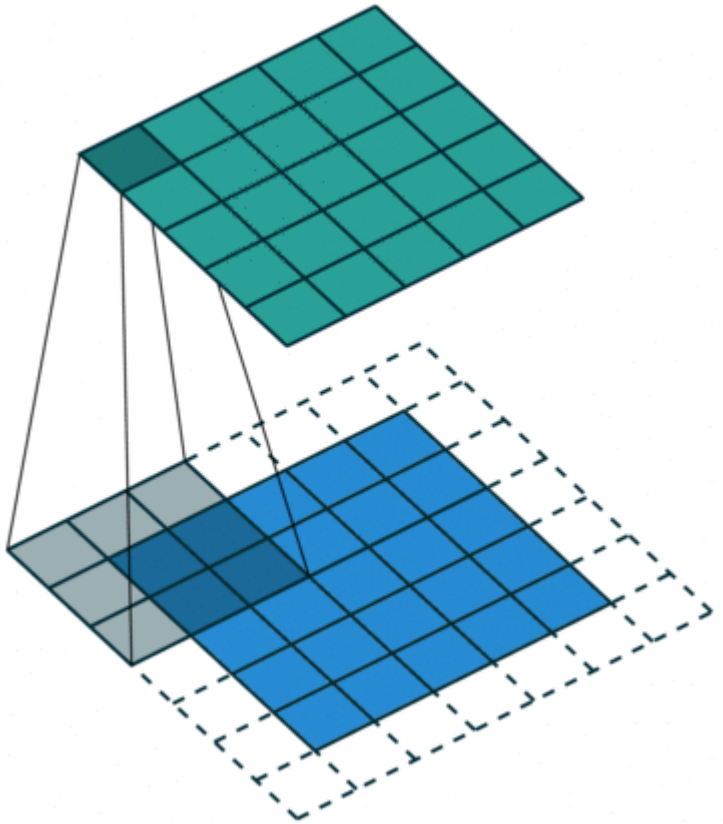
- How to create feature representations that are invariant to confounding factors is a crucial consideration in network design.
- Confounding factors = things that affect the input but should not affect the output.
- By definition, good invariance is application dependent.

Translational Invariance



Translational Invariance

- The convolution operation is inherently invariant to translation.
- Every image pixel, no matter its location, go through the same transformation.



Inductive Bias

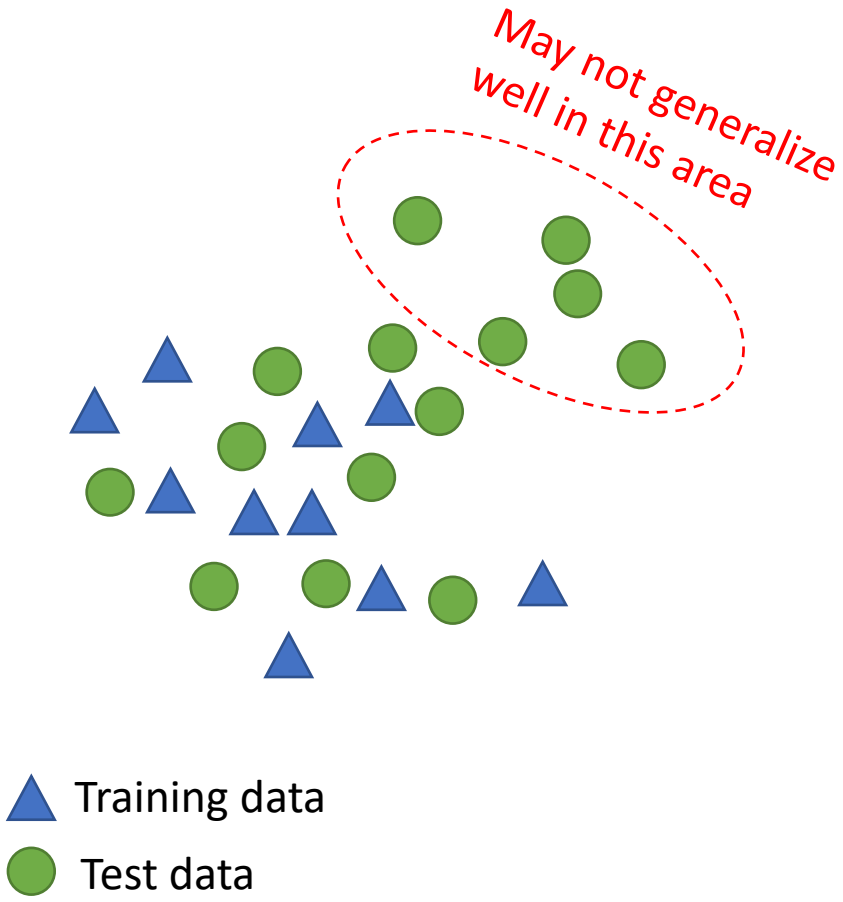
- Induction ≈ learning from data
- Inductive bias refers to the phenomenon that models and learning algorithms prefer to learn some types of functions over others.
- Could be achieved by
  - Model architecture, such as CNN
  - Optimization algorithm, such SGD
  - Regularization techniques

Data Augmentation

- Generation of variations of the same input while keep the label constant.
- Benefits
  - It helps the network learn robust and invariant features.
  - It creates more data. More data are good if they are like the data we expect to see in the future.

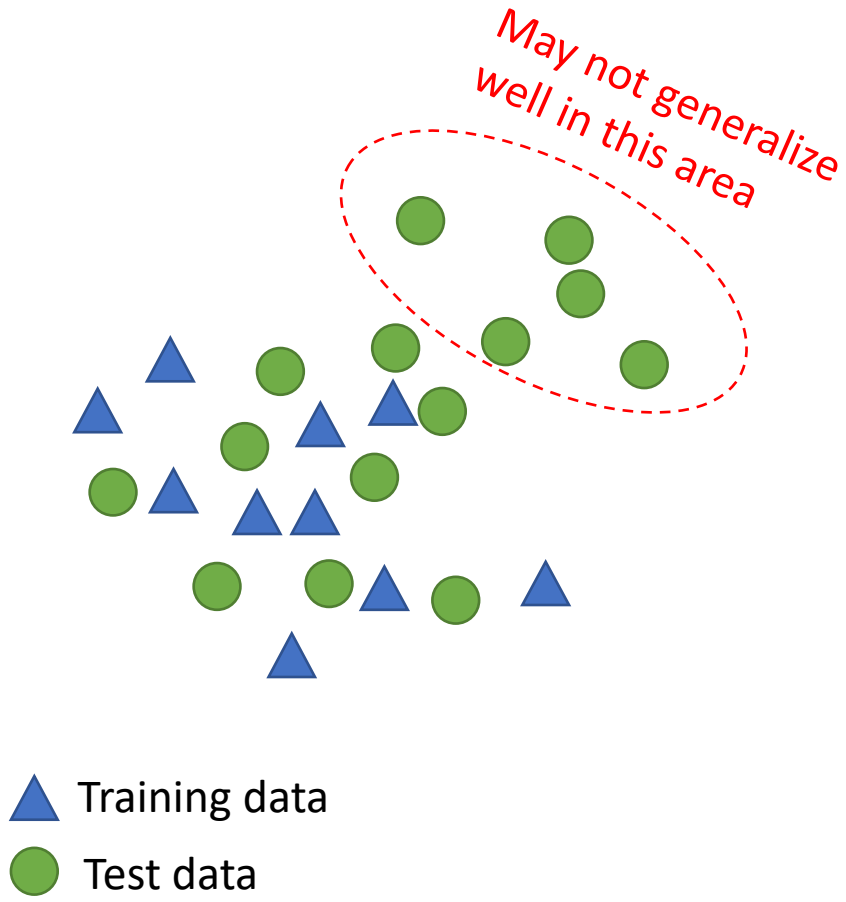
Another Perspective

- Models generalize well when the training data and test data are all drawn from the same distribution $P(X, Y)$
- This is perfectly satisfied when you have synthetic data
- In real-world data, this is hard to achieve.
  - Data-generating processes are very complex, and the observed data are only small samples in comparison



Another Perspective

- What if we can slightly modify training data so that they cover more volume of $P(X, Y)$?
- Answer: Data augmentation



Data Augmentation

- Horizontal Flipping
  - With probability p, the image will be flipped horizontally.
  - Invariance to the direction the object is facing.
- Why not vertical flipping?

Data Augmentation

- Horizontal Flipping
  - With probability p, the image will be flipped horizontally.
  - Invariance to the direction the object is facing.
- Why not vertical flipping?
  - Natural images usually have a clear “up” direction.
  - We don’t have to generalize to vertically flipped images
  - Vertical flipping may change the data distribution $P(X)$

Data Augmentation

- Cropping
  - With random crop and resize ratio, it could help the network recognize objects of slightly different scales.
  - Invariance to translation and occlusion
- Rotation
  - Invariance to changes in rotation and orientation
- Color Jittering
  - Simulates lighting changes and white balance issues

Data Augmentation: Cutout

- Randomly removes a rectangle of pixels.
- In order to maintain the mean value, we can set the removed pixels to the channel means.
- Simulates occlusion
- Forces the network to use features from multiple regions of the image.
- In the homework, you should tweak the hyperparameters and find one set that work well.
  - Ideal probability < 1. i.e., don't use it all the time.

T. DeVries and G. W. Taylor, Improved regularization of convolutional neural networks with cutout. arXiv 1708.04552, 2017.

Model Selection

Three-way Dataset Split

- Training, validation, and test
- Train the network using the training set
- Performance model selection using the validation-set performance.
- Estimate future performance using the test set.

- Model selection refers to the selection of the best hyperparameters
  - Model architectures (# layers, ReLU vs. Sigmoid, etc.)
  - Learning rate
  - Regularization coefficients
  - Many more.
- A.k.a. Hyperparameter tuning

Three-way Dataset Split

- Training, validation, and test
- Train the network using the training set
- Performance model selection using the validation-set performance.
- Estimate future performance using the test set.

- Tuning hyperparameters on the test set leads to overfitting of the test set!
- The test set performance is no longer representative of future performance.

Winner's Curse

- The government sells drilling rights by auction. The highest bidder wins.
- Every drilling company performs a series of tests, and bids on their own estimates of how much oil sits under the surface.
- Historical records show that many winners end up losing money
- **Why?**

Winner's Curse

- Drilling companies bid using their own estimates of land value.
- The estimate is
  - b (industrial average) +
  - a (production advantage) +
  - $\epsilon \sim N(0, \sigma)$ (measurement error)
- The competition on technology is intense.
- In the end, most companies have little technological advantage. That is, their as are close to 0 and their σs are comparable.
- If there are a lot of bidders, one will get really unlucky and have a large positive error in their estimate.
- The company will bid way higher than $b + a$, thinking there is a lot of oil in that land.
- Curse of randomness.

Winner's Curse

- If there are a lot of bidders, one will get really unlucky and have a large positive error in their estimate.
- The company will bid way higher than $b + a$, thinking there is a lot of oil in that land.
- Thus, curse of randomness.

- In Gaussian (very thin tail), 2.5% chance for getting $+2\sigma$ noise
- Chance that at least 1 in N company will get that

$$P = 1 - (1 - 2.5\%)^N$$

- When $N = 20, P = 0.40$
- When $N = 100, P = 0.92$

Hyperparameter Tuning

- The performance is
  - b (average of all possible hyperparameters) +
  - a (advantage over average) +
  - $\epsilon \sim N(0, \sigma)$ (measurement error)
- Measurement error depends on the data you use to compute performance
- By pure chance, the model learns something that works only for this validation / test dataset (overfitting to validation / test).

- If you try enough sets of hyperparameters, some will always be better than others.
- 92% chance of getting $+2\sigma$ out of 100 trials.
- Switching to a different dataset is equivalent to drawing another ϵ
  - Unlikely to repeat extreme values

Practical Tips: Get the Priorities Right

- It is important to focus on the most important hyperparameters
- Too many hyperparameters to perform exhaustive tuning
- Many do not have strong influence on the result.

- Optimization first, then regularization
- First, achieve close-to-zero training loss (learning rate, momentum, optimizer)
- After that, increase regularization to reduce overfitting.

Practical Tips: Grid Search

- Divide the possible range to a number of buckets and try each one.
- For example: 0.01, 0.04, 0.07, 0.1

- For hyperparameters that can have very large range (e.g., learning rate), first determine the magnitude
  - 0.1, 0.01, 0.001, or 10^{-4} ?
- May do a coarse grid followed by a fine grid.