FIT3181/5215 Deep Learning

Week 05: Practical skills in deep learning

Lecturer: Trung Le

Email: trunglm@monash.edu

Department of Data Science and AI
Faculty of Information Technology, Monash University, Australia

Outline

- Setting of a machine learning problem
  - General loss versus empirical loss
- Gradient vanishing/exploding and network initialization.
- Overfitting and underfitting
- Recipe for overfitting
  - Use regularization term, dropout, batch norm, data augmentation, transfer learning
  - Label smoothing, data mix-up, cut-mix

- Further reading recommendation
  - [Deep Learning, Sections 4.1-4.3, 8.1 -8.5, 11.3, 11.4].
  - [Dive into Deep Learning, Chapters 5 and 11].

Machine learning setting

Not in assessment

Data-label generative process

$(x', y') \sim p_{data}(x, y) = p_{data}(x)p_{data}(y \mid x)$

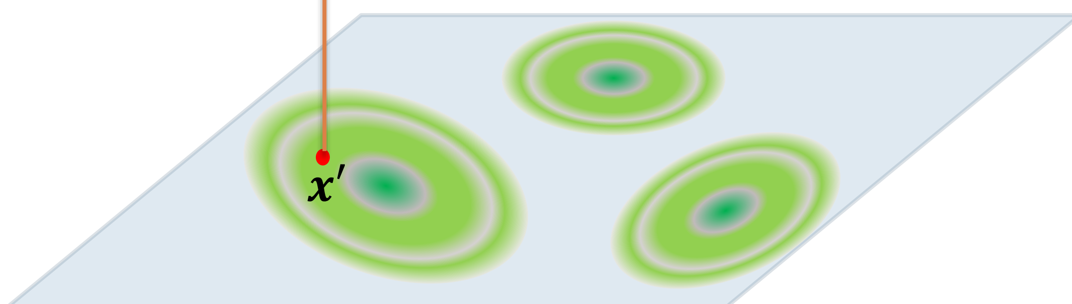
$y' = \mathbf{1}\ \textit{or cat}$

$y' \sim p_{data}(y \mid x')$

Cat (1)	Dog (2)	Lion (3)	Panda (4)
0.7	0.2	0.05	0.05

$p_{data}(y \mid x')$

Ground-truth **labelling mechanism**



Data space $\mathcal{X}$

$x' \sim p_{data}(x)$

Data distribution

Dataset collecting process

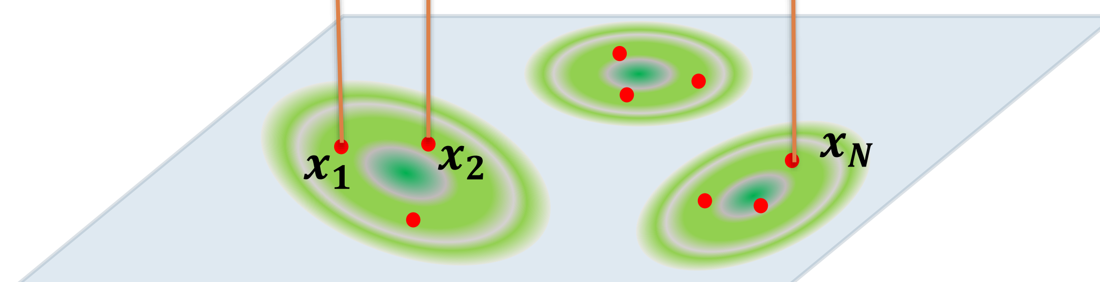
$D = \{(x_1, y_1), \dots, (x_N, y_N)\}$

$$(x_i, y_i) \sim p_{data}(x, y) \xleftarrow{N \to \infty} \tilde{p}_{data}(x, y) = \frac{1}{\mathrm{N}} \sum_{i=1}^{N} \delta_{(x_i, y_i)}(x, y)$$

$y_1 \sim p_{data}(y \mid x_1)$

$y_N \sim p_{data}(y \mid x_N)$

$y_2 \sim p_{data}(y \mid x_2)$



Data space $\mathcal{X}$

$x_1, \dots, x_N \sim p_{data}(x)$

Data distribution

Machine learning setting

Empirical data/label distribution $\tilde{p}_{data}$

(x, y)	(x_1, y_1)	(x_2, y_2)	…	(x_N, y_N)
p	$1/N$	$1/N$	…	$1/N$

Data-label generative process

$$(x', y') \sim p_{data}(x, y) = p_{data}(x) p_{data}(y \mid x)$$

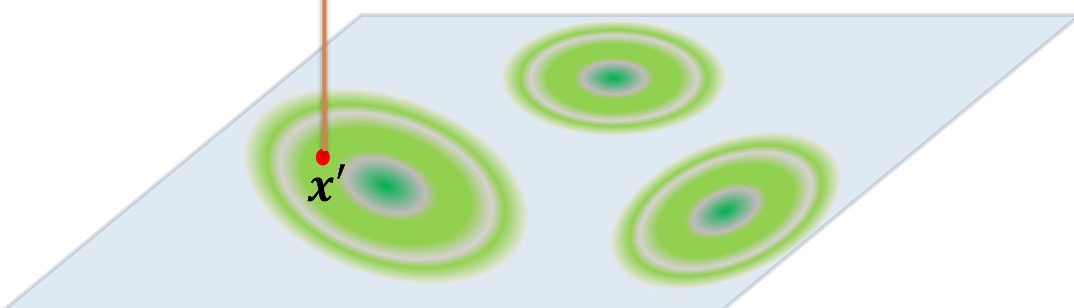
$y' = \mathbf{1\ or\ cat}$

$y' \sim p_{data}(y \mid x')$

Cat (1)	Dog (2)	Lion (3)	Panda (4)
0.7	0.2	0.05	0.05

$p_{data}(y \mid x')$

Ground-truth **labelling mechanism**



Data space $\mathcal{X}$

$x' \sim p_{data}(x)$

Data distribution

Dataset collecting process

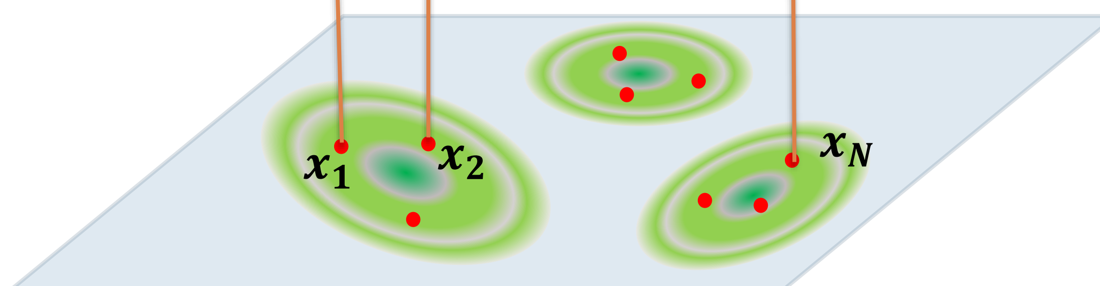
$$D = \{(x_1, y_1), \dots, (x_N, y_N)\}$$

$$(x_i, y_i) \sim p_{data}(x, y) \xleftarrow{N \to \infty} \tilde{p}_{data}(x, y) = \frac{1}{\mathrm{N}} \sum_{i=1}^{N} \delta_{(x_i, y_i)}(x, y)$$

$y_1 \sim p_{data}(y \mid x_1)$

$y_2 \sim p_{data}(y \mid x_2)$

$y_N \sim p_{data}(y \mid x_N)$



Data space $\mathcal{X}$

$x \sim p_{data}(x)$

Data distribution

Not in assessment

Machine learning setting

Not in assessment

Generalization (general) loss

(loss on data/label distribution)

$$\mathcal{L}_{gen}(\theta) = \underset{p_{data}}{\mathbb{E}}[l(f(x;\theta),y)]$$

$N \to \infty$

Law of large numbers

Empirical loss

(loss on a collected training set D)

$$\mathcal{L}_{emp}(\theta) = \underset{\tilde{p}_{data}}{\mathbb{E}}[l(f(x;\theta),y)] = \frac{1}{N}\sum_{i=1}^{N} l(f(x_i;\theta),y_i)$$

Ideal: $\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}_{gen}(\theta)$

Reality: $\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}_{emp}(\theta)$

$$D = \{(x_1,y_1),\ldots,(x_N,y_N)\}$$

$$(x',y') \sim p_{data}(x,y) = p_{data}(x)p_{data}(y \mid x)$$

$$y' \sim p_{data}(y \mid x')$$

$$(x_i,y_i) \sim p_{data}(x,y) \xleftarrow{N\to\infty} \tilde{p}_{data}(x,y) = \frac{1}{N}\sum_{i=1}^{N} \delta_{(x_i,y_i)}(x,y)$$

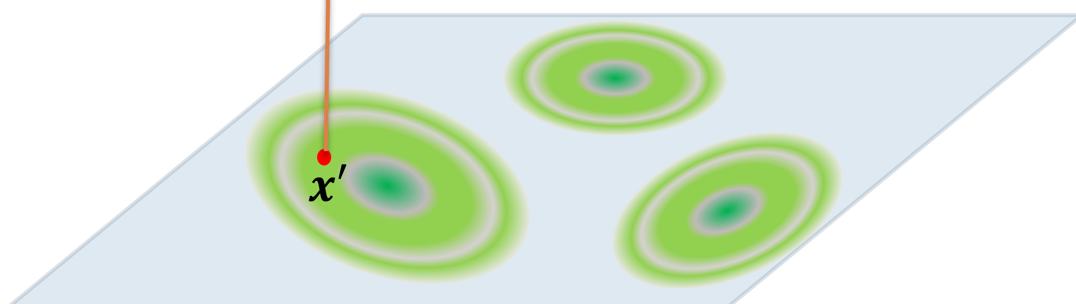
$y_1 \sim p_{data}(y \mid x_1)$

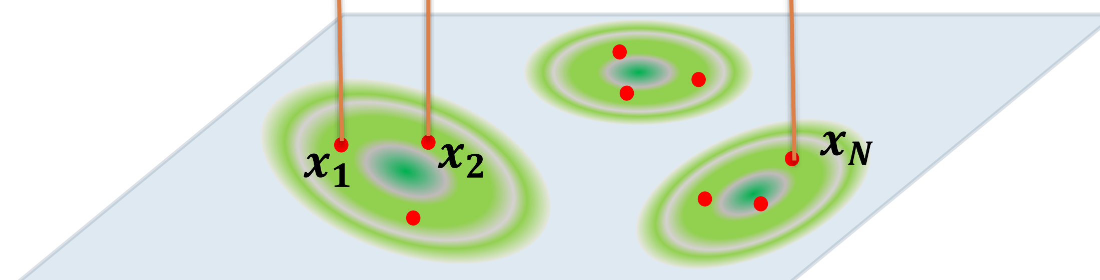
$y_N \sim p_{data}(y \mid x_N)$

$y_2 \sim p_{data}(y \mid x_2)$

$p_{data}(y \mid x')$

Labelling mechanism





Data space $\mathcal{X}$

$x' \sim p_{data}(x)$

Data distribution

Data space $\mathcal{X}$

$x \sim p_{data}(x)$

Data distribution

How Learning Differs from Pure Optimization?

- Some important notations:
  - $p_{data}(x, y)$: **existed**, but <u>**unknown**</u>, distribution of data and label
  - $\tilde{p}_{data}(x, y) = \frac{1}{N}\sum_{i=1}^{N} \delta_{(x_i, y_i)}(x, y)$: **empirical** data/label distribution - this is what we observed from <u>training</u> data $\boldsymbol{D} = \{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1}^{N}$ where $(\boldsymbol{x_i}, \boldsymbol{y_i}) \overset{\text{iid}}{\sim} p_{data}(x, y)$.
  - Per-sample loss: $l(f(x; \theta), y)$

- **Empirical loss minimisation** (pure optimisation):

$$\mathcal{L}_{emp}(\theta) = \mathop{\mathbb{E}}_{\tilde{p}_{data}}[l(f(x; \theta), y)] = \frac{1}{N}\sum_{i=1}^{N} l(f(x_i; \theta), y_i)$$

Empirical loss (<u>maths</u>)

vs.

Generalisation loss (<u>ML</u>)

- But what ML wants is **true generalisation loss**:

$$\mathcal{L}_{gen}(\theta) = \mathop{\mathbb{E}}_{p_{data}}[l(f(x; \theta), y)]$$

How to **achieve this** when we **only have access** to empirical data?

Not in assessment

Optimization Problem in ML and DL

- Most of **optimization problems** in machine learning (deep learning) has the following form:

$$\min_\theta J(\theta) = \Omega(\theta) + \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))$$

Guiding principles:

1. **Occam Razer:** prefer the simplest model that can do well.

Regularization term

- $\Omega(\theta) = \lambda \sum_k \sum_{i,j} \left(W_{i,j}^{\mathbf{k}}\right)^2 = \lambda \sum_k \left\|\boldsymbol{W}^{\mathbf{k}}\right\|_F^2$
- Encourage simple models
- Avoid overfitting

Empirical loss

- Work well on training set

How to efficiently solve this optimization problem?

N is the **training size** and might be very big (e.g., $N \approx 10^6$)

Works well for Deep Learning (non-convex)

First-order iterative methods

gradient descent, steepest descent

Use the **gradient** (first derivative) $\mathrm{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to update parameters:

$$w = w - \text{learning rate} \times \text{gradient}$$

Works well for convex problems, but not DL

Second-order iterative methods

Newton and quasi Newton methods

Use the Hessian matrix (second derivative) $\mathrm{H} = \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta})$ to update parameters

Optimization Problem in ML and DL

- Most of optimization problems in machine learning (deep learning) has the following form:

$$\min_\theta J(\theta) = \Omega(\theta) + \frac{1}{N}\sum_{i=1}^{N} l\big(y_i, f(x_i;\theta)\big)$$

What loss function should one use?

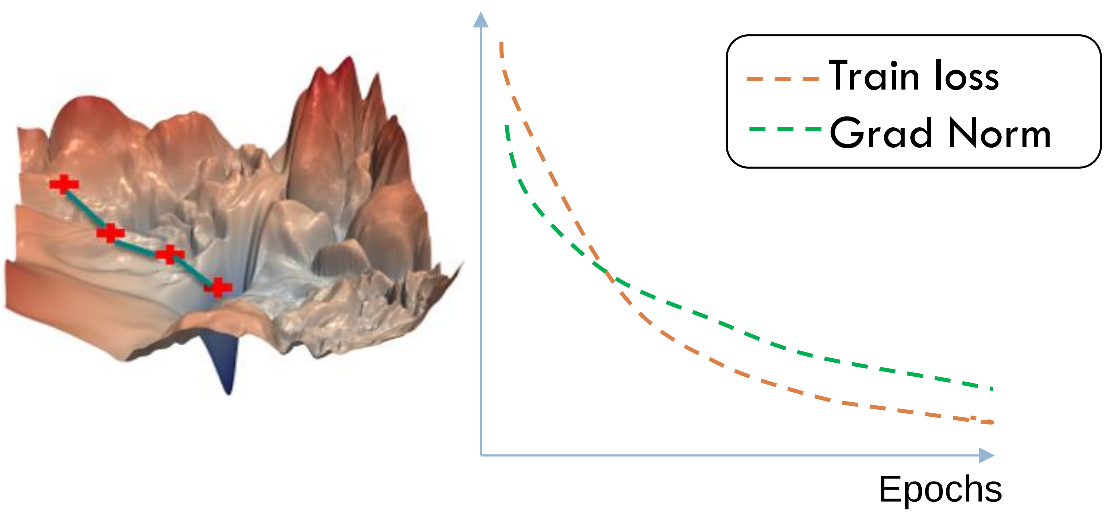
Some typical loss functions used in the classification and regression problems

- 0-1 loss, Hinge loss, Logistic loss (binary classification)
- L1 loss, L2 loss, ε −insensitive loss (regression)
- Popular surrogate loss: cross-entropy loss (multi-class classification, deep learning)

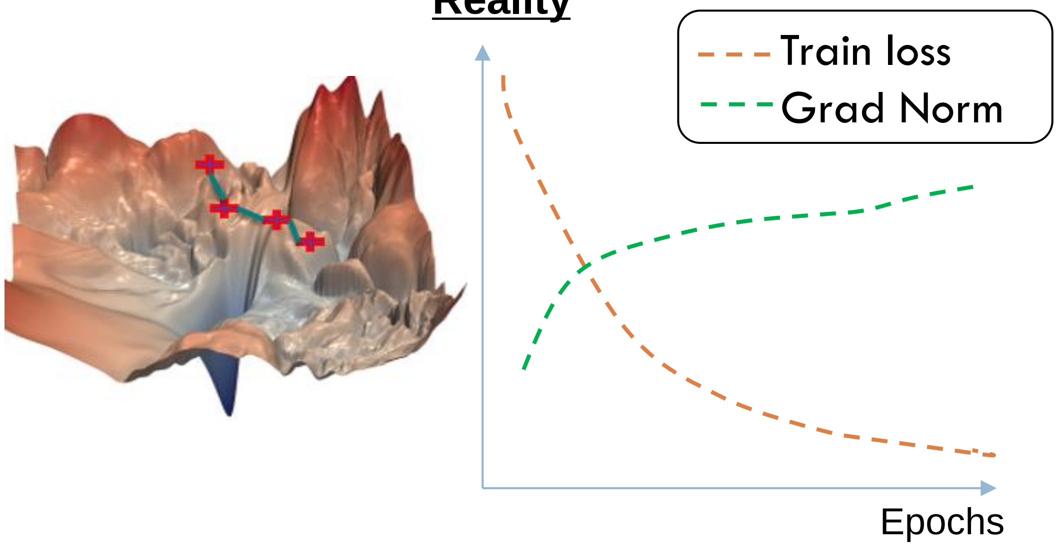
How Learning Differs from Pure Optimization?

- Achieving **generalisation capacity** is the holy-grail of machine learning.
  - Empirical risk is prone to over-fitting
  - Some times, it is not really feasible if the loss function **does not have useful derivatives** (e.g., **0-1 loss**), hence we usually resort to surrogate loss function, e.g., cross-entropy, Hinge loss, etc.
- In most cases, DL algorithm **doesn't halt** at **local minimum**
  - It halts when **certain convergence criterion** is **met** (e.g., based on **early stopping** when overfitting start to occur, reach **certain budgets**, **number of epochs**, etc).
  - For training DL models, it **might stop** when the loss function still has **large derivates**, which is **different** from a pure optimisation setting.

Ideality



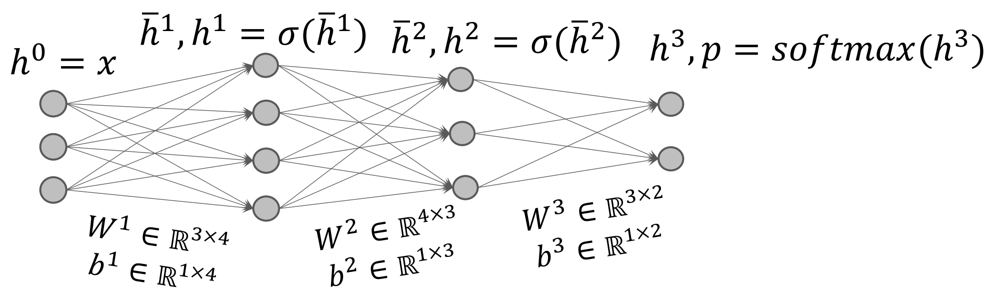
Reality



Challenges in deep learning optimization: gradient vanishing, gradient exploding

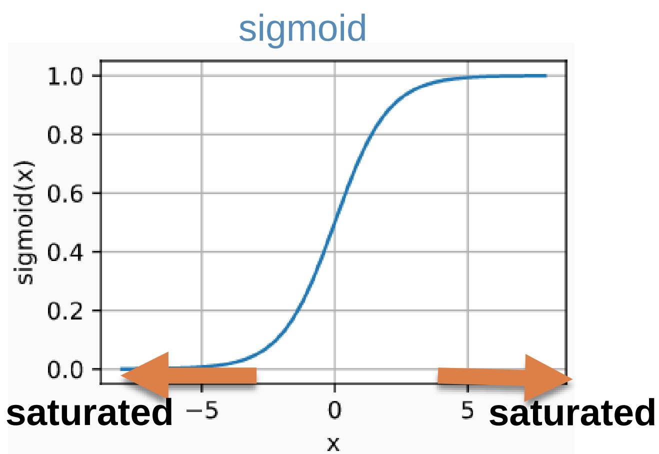
Gradient vanishing

- **Gradient Vanishing**
  - Gradients get **smaller and smaller** as the algorithm progresses down to the **lower layers**.
    - SGD update leaves the lower layer connection weights **virtually unchanged**, and training **never converges** to a **good solution**.
- Some activation functions are easy to **get saturated**
  - Sigmoid or tanh

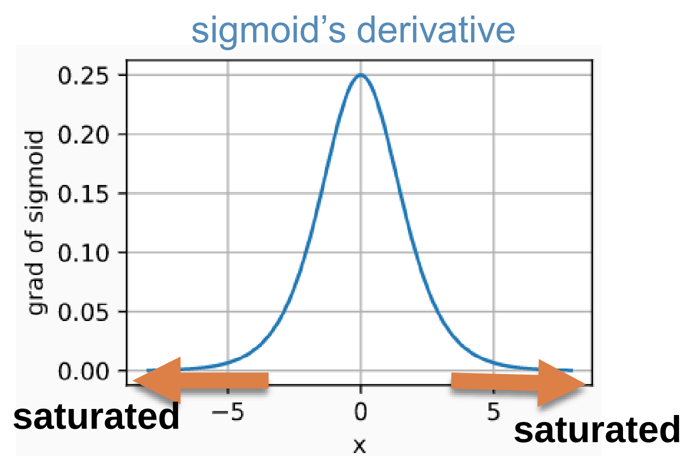


$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3}\cdot\frac{\partial h^3}{\partial h^2}\cdot\frac{\partial h^2}{\partial \bar{h}^2}\cdot\frac{\partial \bar{h}^2}{\partial h^1}\cdot\frac{\partial h^1}{\partial \bar{h}^1}\cdot\frac{\partial \bar{h}^1}{\partial W^1}$$

$$= \left[(p^T - 1_y)W^3 diag\left(\sigma'(\bar{h}^2)\right) W^2 diag\left(\sigma'(\bar{h}^1)\right)\right]^T (h^0)^T$$



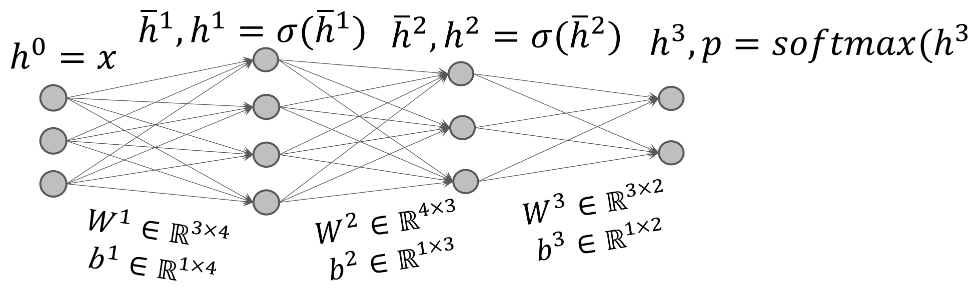
$$\sigma(z) = s(z) = \frac{1}{1 + \exp\{-z\}}$$



$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Gradient vanishing

- **Gradient Vanishing**
  - Gradients get **smaller and smaller** as the algorithm progresses down to the **lower layers**.
    - SGD update leaves the lower layer connection weights **virtually unchanged**, and training **never converges** to a good solution.
- Some activation functions are easy to **get saturated**
  - Sigmoid or tanh
- **Recipe**
  - Activation function plays an important role! Common practice:
    - Avoid sigmoid or saturated activation function
    - ReLU is a common good choice
  - Good weight initialization is critical!



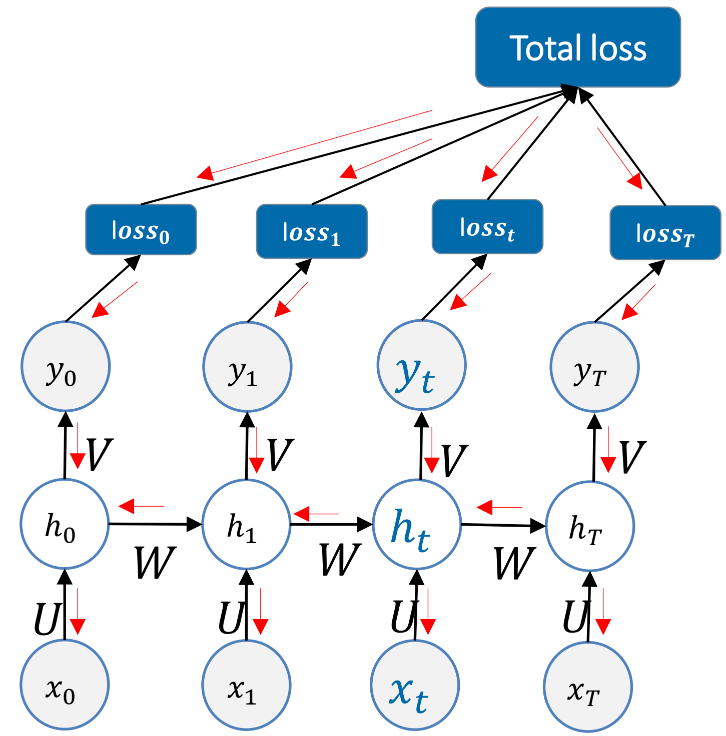
$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3} \cdot \frac{\partial h^3}{\partial h^2} \cdot \frac{\partial h^2}{\partial \bar{h}^2} \cdot \frac{\partial \bar{h}^2}{\partial h^1} \cdot \frac{\partial h^1}{\partial \bar{h}^1} \cdot \frac{\partial \bar{h}^1}{\partial W^1}$$

$$= \left[(p^T - 1_y) W^3 diag\left(\sigma'(\bar{h}^2)\right) W^2 diag\left(\sigma'(\bar{h}^1)\right) \right]^T (h^0)^T$$

Gradient exploding

- **Gradient Exploding**
  - The gradients can grow **bigger and bigger**, so many layers get **insanely large weight** updates, and the training **diverges**.
  - Mostly being encountered in **recurrent neural networks**.

- Often happen for **recursive models** for example **Recurrent Neural Network** (RNN), **Bidirectional RNN**



$$\frac{\partial l_T}{\partial h_0} = \frac{\partial l_T}{\partial h_T} \times \underbrace{\underset{W}{\frac{\partial h_T}{\partial h_{T-1}}} \times \cdots \times \underset{W}{\frac{\partial h_1}{\partial h_0}}}$$

Multiplication of **many matrices** W

Tip:
- Let simplify W as **a scalar** in the real-valued set
  - $W^m \to 0$ if $|W| < 1$.
  - $W^m \to \infty$ if $|W| > 1$.

Gradient Clipping

- One way to lessen the **exploding gradients problem** is to simply clip the gradients during backpropagation so that they never **exceed** some threshold
  - Either clip by values (direction might change)
  - or clip by norms (keep direction but rescale the magnitude)
- Widely applied to **Recurrent Neural Networks**
- Widely used for NLP tasks, not so much used for CNNs.

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Example model
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(10, 1)

    def forward(self, x):
        return self.fc(x)

# Initialize the model, loss function, and optimizer
model = SimpleModel()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Example training loop
for epoch in range(100):
    optimizer.zero_grad()
    # Example input and target tensors
    inputs = torch.randn(32, 10)  # Batch of 32, input size 10
    targets = torch.randn(32, 1)  # Batch of 32, target size 1
    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    # Backward pass
    loss.backward()
    # Gradient clipping by norm
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=2.0)
    # Update model parameters
    optimizer.step()
    # Print loss
    print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

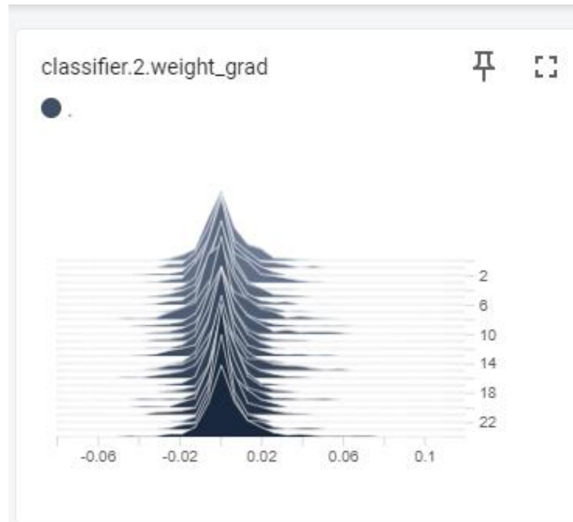
Observing Gradient Vanishing and Exploding

Log the histogram of gradients to TensorBoard

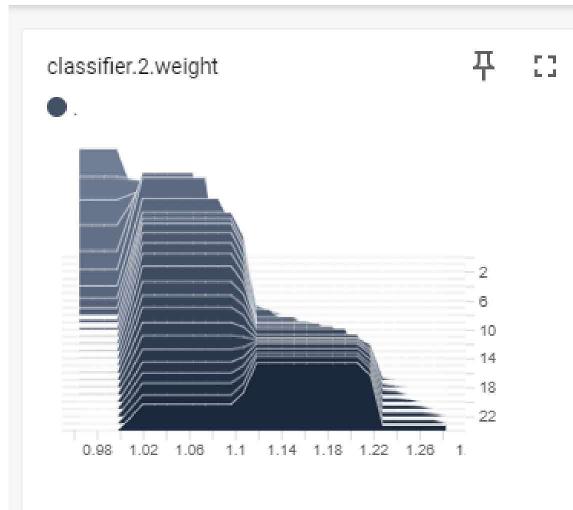
```
def on_epoch_end(self, params):
    epoch = params['epoch']

    # Log scalar values
    if self.log_scalars:
        self.writer.add_scalar('Loss/train', params['train_loss'], epoch)
        self.writer.add_scalar('Loss/validation', params['val_loss'], epoch)
        self.writer.add_scalar('Accuracy/train', params['train_accuracy'], epoch)
        self.writer.add_scalar('Accuracy/validation', params['val_accuracy'], epoch)

    # Log a batch of images (example with first batch from trainloader)
    if self.log_images:
        dataiter = iter(self.train_loader)
        images, labels = next(dataiter)
        img_grid = vutils.make_grid(images)
        self.writer.add_image('images_batch', img_grid, epoch)

    # Log histogram of model parameters
    if self.log_histograms:
        #log the model parameters
        for name, param in self.model.named_parameters():
            self.writer.add_histogram(name, param, epoch)

        #log the gradients w.r.t. the model parameters
        num_selected = 100
        subset, _ = torch.utils.data.random_split(val_subset, [num_selected, len(val_subset) - num_selected])
        subset_loader = torch.utils.data.DataLoader(subset, batch_size=num_selected) # Create a dataloader for
        inputs, labels = next(iter(subset_loader)) # Get a batch of data
        inputs, labels = inputs.to(device), labels.to(device) # Move data to device
        outputs = self.model(inputs) # Use self.model instead of models
        loss = self.criterion(outputs, labels)
        loss.backward()
        for name, param in self.model.named_parameters():
            self.writer.add_histogram(name + '_grad', param.grad, epoch)
```

Using TensorBoard

classifier.2.weight_grad



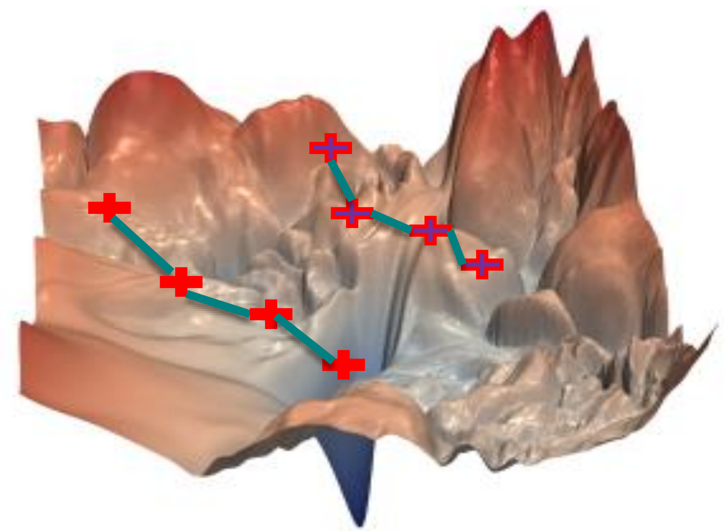
classifier.2.weight



Weight initialization

He and Xavier weight initialization

- **Initialisation** in deep learning training is **crucial**:
  - Some optimizers can be **theoretically guaranteed** to **converge** regardless of initializations
  - Deep learning algorithms do not have these luxuries:
    - It is iterative, but optimization deep neural networks is **not yet well understood**
    - **Initial point** is **extremely important**: it can determine if the algorithm converges, or with some **bad initialisation**, it becomes unstable and fails together



Initialization is **important** for training DL models.

- What is a **good weight/filter initialization**?
  - **Break the 'symmetry'** of the network: two hidden nodes with the **same input** should have **different weights**.
  - The **gradient signal to flow well** in both directions and **don't want** the signal to **die out** or to **explode** and **saturate**.
  - **Large initial weights** has **better symmetry breaking** effect, help **avoiding losing signals** and **redundant units**, but could **result in exploding values** during back-ward and forward passes, especially in Recurrent Neural Networks.

He and Xavier weight initialization

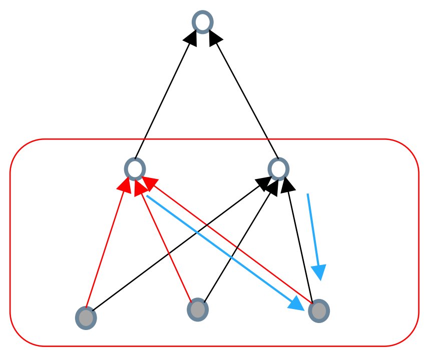
- **Xavier initialization**
  - Try to ensure the **variance of the outputs** of each layer equal to the **variance of its inputs**
  - Also need the gradients to have **equal variance** before and **after flowing through a layer** in the **reverse direction**
  - Good for **sigmoid** and **tanh** functions
  - Not good for ReLU

Gaussian version

$$w_{Xa} \sim N\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$

Uniform alternative

$$w_{Xa} \sim \text{Uniform}\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$



$n_{in} = 3$

$n_{out} = 2$

Why $\sqrt{\frac{2}{n_{in}+n_{out}}}$?

Understanding the difficulty of training deep feedforward neural networks

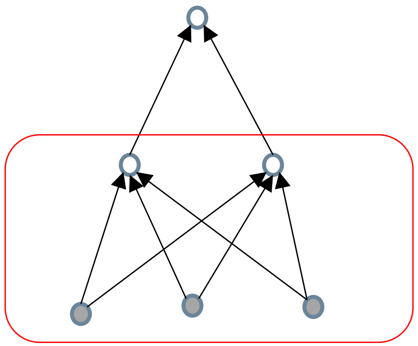
Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

Paper link: https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

He and Xavier weight initialization

- **Xavier initialization**
  - Ensure the **variance of the outputs** of each layer **equal** to the **variance of its inputs**
  - Also need the **gradients** to have **equal variance before** and **after** flowing through a layer in the reverse direction
  - Good for **sigmoid** and **tanh** activation functions
  - **Not** good for **ReLU**

$$w_{Xa} \sim N\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$



$n_{out} = 2$

$n_{in} = 3$

- **He initialization**
  - A **variant of Xavier initialization** where $\alpha = 1$
  - Works better for ReLU.

$$w_{He} \sim N\left(0, \alpha \times \sqrt{\frac{2}{n_{in} + n_{out}}}\right) \quad \alpha = \begin{cases} 1 & \text{if sigmoid} \\ 4 & \text{if tanh} \\ \sqrt{2} & \text{if ReLU} \end{cases}$$

Delving Deep into Rectifiers:
Surpassing Human-Level Performance on ImageNet Classification

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research

Paper link: https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf


Q&A
When? Why? How? What? How? Why? WHAT? Why? When?
WHAT? WHERE? WHEN? HOW? WHEN? When? WHERE?
When? WHERE? WHAT? Why? When? WHAT?
WHO? WHEN?
WHERE? What? HOW?
Why? HOW? Where? Where? WHERE? When? What? What? WHERE? WHERE? When? When?
Why? WHEN? HOW? When? When? HOW? WHAT? Why?
WHEN?

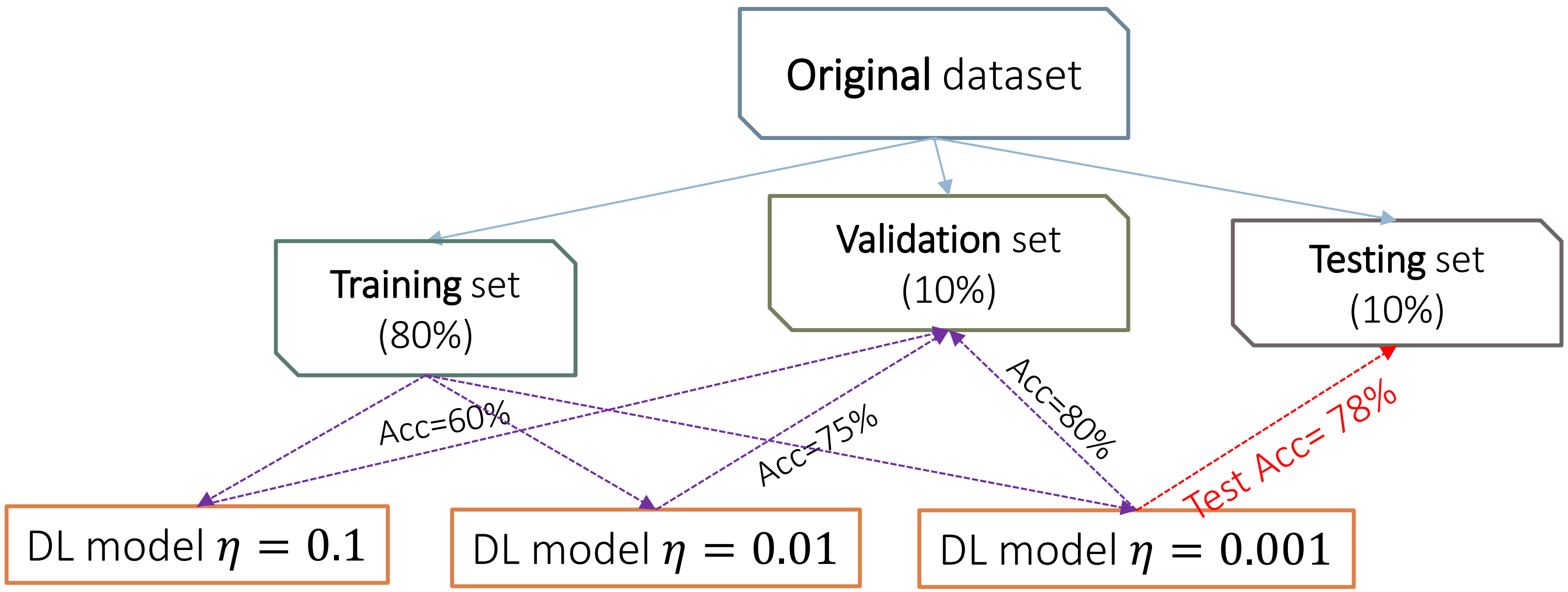
Where do we go from here?

- Challenges in deep learning optimisation and how to address them:
  - Local minima, saddle points and complex loss surfaces
  - Gradient vanishing and exploding
  - What we don't cover in this unit (see DL section 8.2):
    - Ill-conditioning problem
    - Long-term dependencies
    - Poor correspondence between local and global structures
    - Theoretical limits of optimisation (but they usually have little use in practice of deep learning)
- Initialization Strategies
- Regularization in deep learning
  - Parameter norm penalty: l1, l2 regularization
  - Early stopping
  - Dropout
  - Batch normalization
- Choice of optimizers:
  - Basic algorithms: SGD, Momentum, Nesterov Momentum
  - Algorithms with adaptive learning rate: AdaGrad, RMSProp, Adam

Overfitting and Regularization in Deep Learning

Deep Learning Pipeline

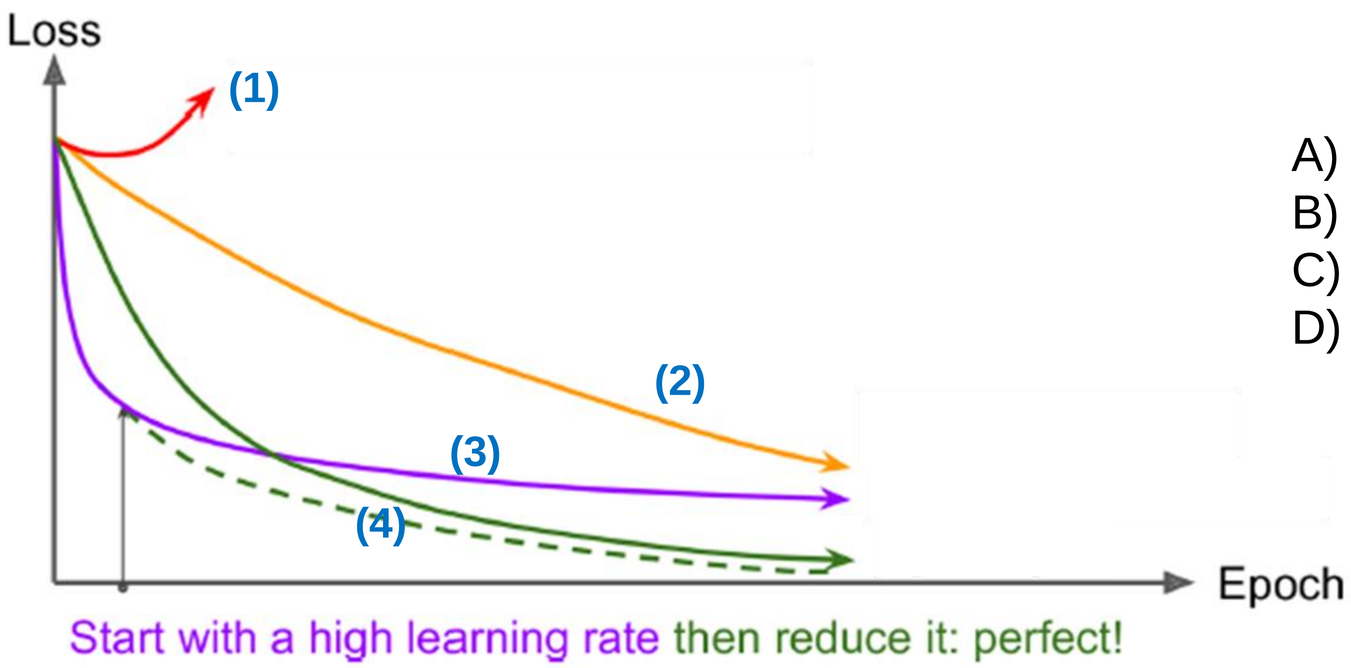
- We want to train our DL model on a training set such that the trained model can predict well unseen data in a separate testing set.



Model parameters: weight matrices, biases, filters which will be learnt
Hyper-parameters to consider: learning rate, #layers, #neurons which need to be tuned

Deep Learning Pipeline

- We want to train our DL model on a training set such that the trained model can predict well unseen data in a separate testing set.

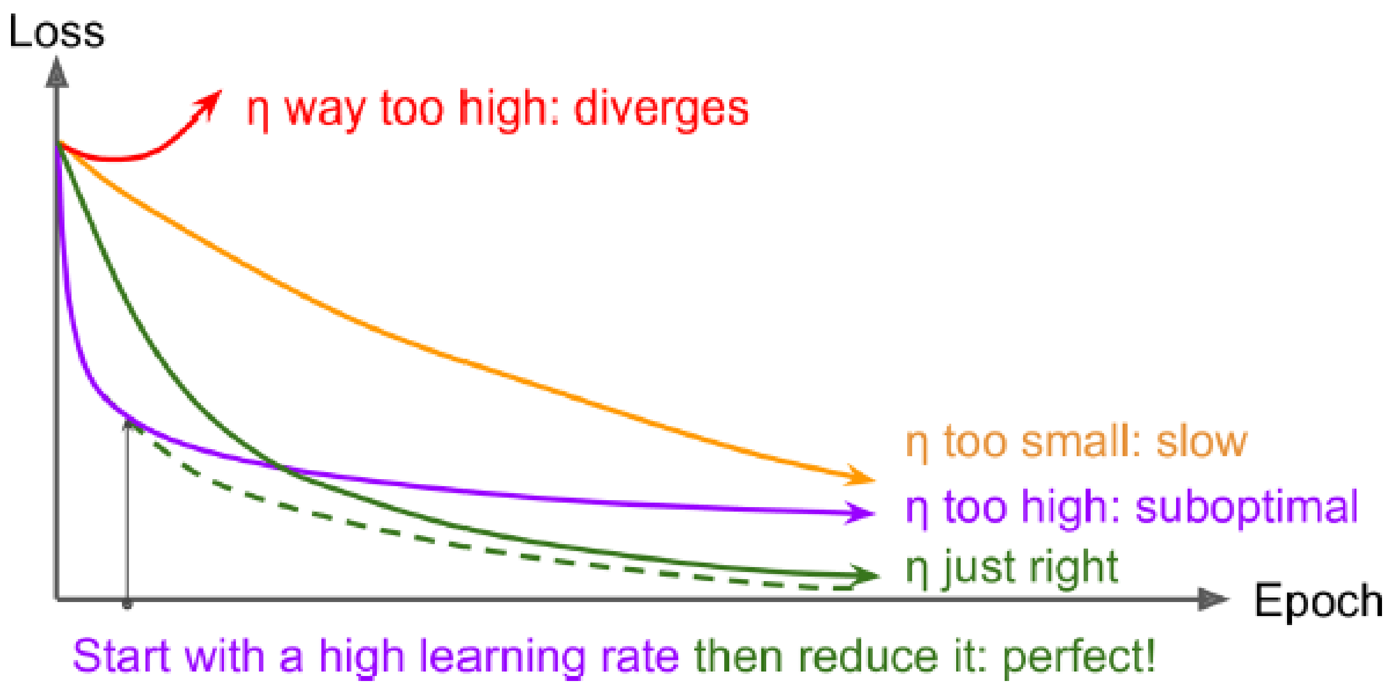


A) Extremely high
B) Too high
C) Too small
D) Just right

Hyper-parameters to consider: learning rate, #layers, #neurons

Deep Learning Pipeline

- We want to train our DL model on a training set such that the trained model can predict well unseen data in a separate testing set.



Hyper-parameters to consider: learning rate, #layers, #neurons

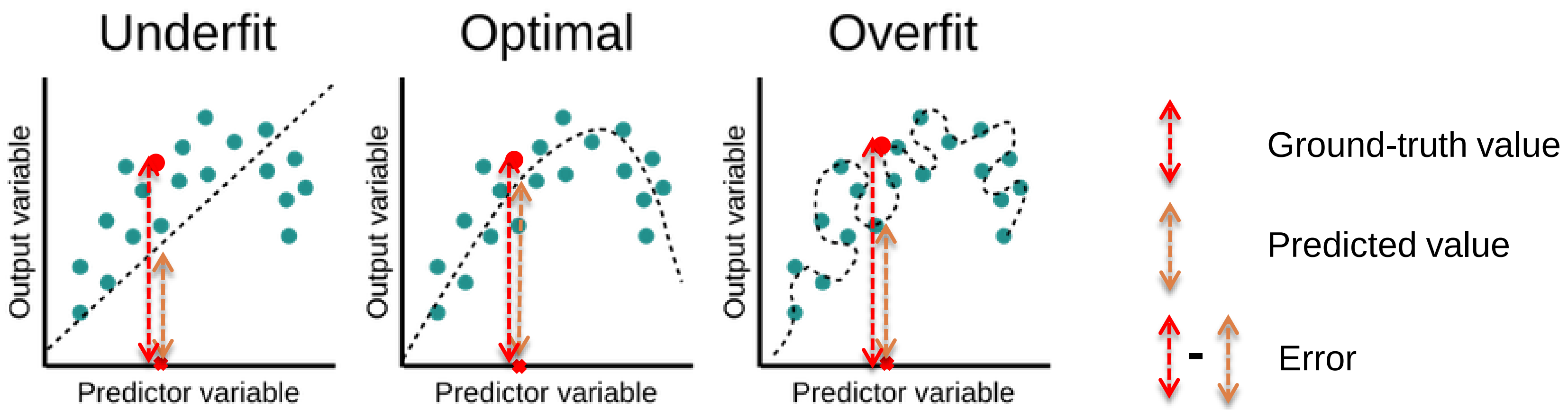
Regularization and Overfitting

- Three elements in ML and DL
  - **Data**: training data, testing data, validation data
  - **Model**: a **mathematical function** $\boldsymbol{f(x;\theta)}$ that maps an input instance x to outcome y
  - **Evaluation**: a **performance metric** to quantitively measure how well $f(x;\theta)$ is

- Machine learning as an **optimization process**. **Learning** from data by **optimizing** its loss
  - Define a measure of loss via a **loss function**: $l(f(x;\theta),y)$
  - Compute its **loss over all training data** $\mathrm{J}(\theta) = N^{-1}\sum_{i=1}^{N} l(f(x_i,\theta),y_i)$
  - Learning = **finding** $\boldsymbol{\theta}^*$ that **minimizes** the loss: $\theta^* = \arg\min_{\theta} J(\theta)$

- What might **go wrong** with this formulation?
  - The choice of learning function $f(x;\theta)$ is too hard to learn
  - The choice of loss function $l(f(x),y)$ is inadequate
  - The model **does well** on training data, but **perform poorly** on unseen test data: **overfitting** problem!

Overfitting & Underfitting



Overfitting with regression

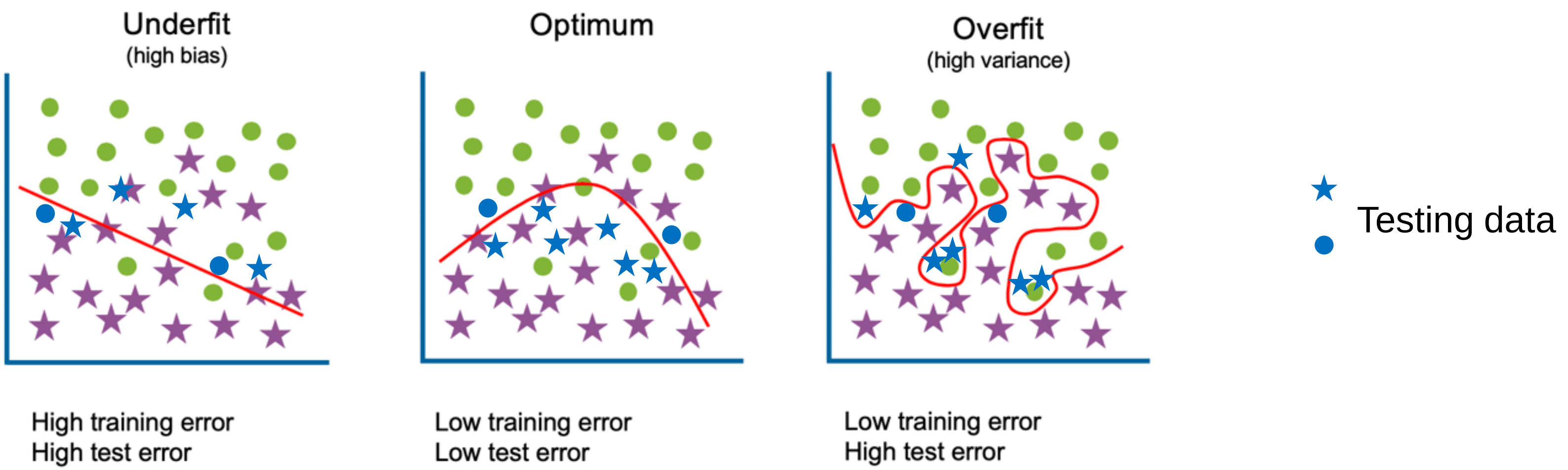
- **Underfitting**
  - The model is **too simple** to characterise a training set
    - Use linear model to learn from non-linear data

- **Overfitting**
  - The model performs **very well** on the **training set**, but **cannot generalise** to perform well on a separate **testing set**
  - This is the most common problem in DL since deep networks are very powerful!

Overfitting in Deep NNs

Overfitting

Overfit: tendency of the network to "memorize" all training samples, leading to poor generalization



Overfitting with classification

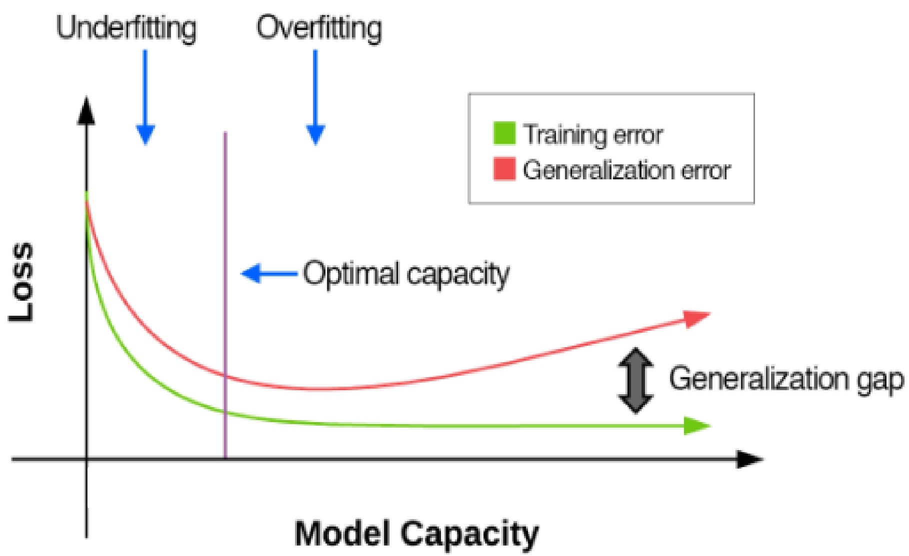
[Source: https://www.ibm.com/cloud/learn/overfitting]

- **Overfitting** occurs when your network models the training data *too well* and fails to generalize to your test (validation) data.
  - Performance measured by errors on "unseen" data.
  - **Minimize error alone** on **training data** is **not** enough
  - Causes: **too many layers, too many hidden nodes,** and **overtrained**.

Hyperparameter	Increases capacity when...	Reason	Caveats
Number of hidden units	increased	Increasing the number of hidden units increases the representational capacity of the model.	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model.
Learning rate	tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure	
Convolution kernel width	increased	Increasing the kernel width increases the number of parameters in the model	A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost.
Implicit zero padding	increased	Adding implicit zeros before convolution keeps the representation size large	Increased time and memory cost of most operations.
Weight decay coefficient	decreased	Decreasing the weight decay coefficient frees the model parameters to become larger	
Dropout rate	decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set	

Table 11.1: The effect of various hyperparameters on model capacity.

We can also experiment with model capacity itself in parallel.

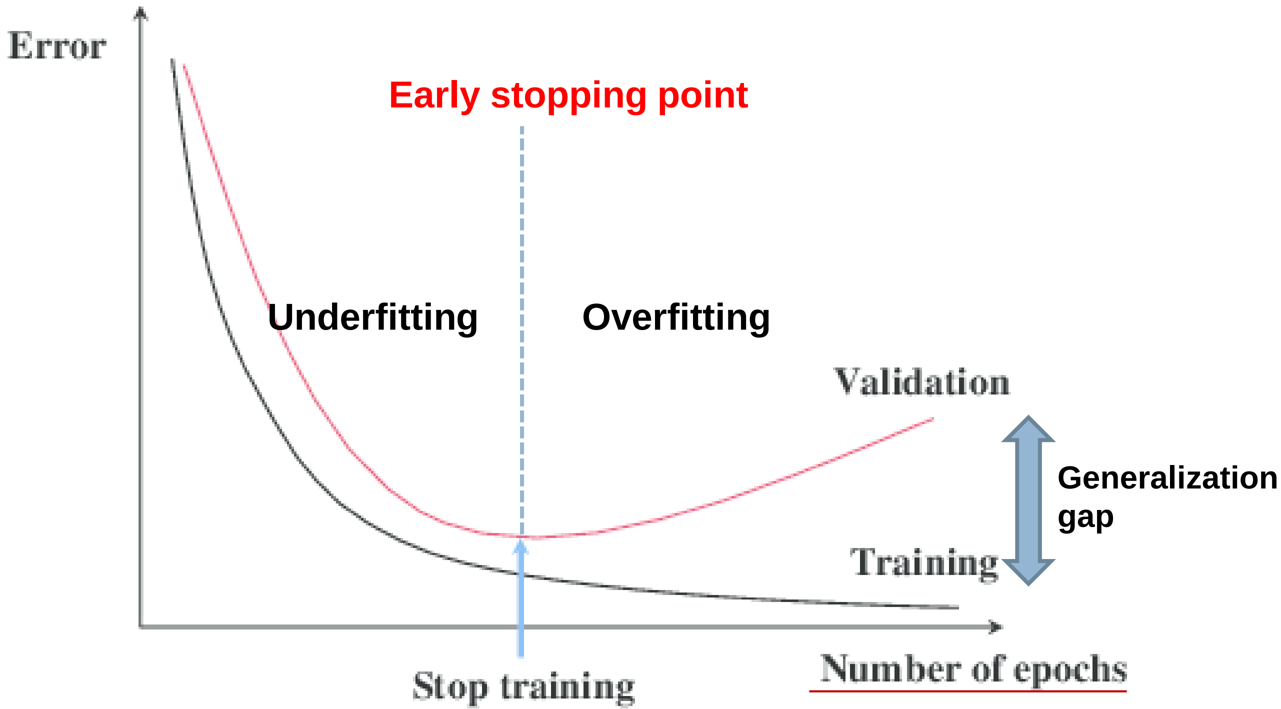


[DL, ch11]

Recipe for Overfitting

- **Early stopping**
  - Stopping the training on time before it becomes overtrained and overly complex.
- **Train with more data**
  - Expanding the training set to include more data
- **Data augmentation**
  - Creating many variations of clean data to make the model to generalize better to unseen examples.
- **Regularization**
  - Overfitting occurs when a model is too complex comparing with data.
  - Using regularization terms (L1, L2), batch norm, dropout, data mix-up, label smoothing, VAT (virtual adversarial training).
- **Ensemble methods (not cover in this lecture)**
  - Ensemble learning methods are made up of a set of classifiers and their predictions are aggregated to identify the most popular result.
  - The most well-known ensemble methods are **bagging** and **boosting**.

Early stopping



- At first, the train and valid losses **gradually drop**, but the model is **not good enough** to characterise data
  - **Underfitting** is happening
- At a certain point, the train loss **still decreases**, while the valid loss **starts increasing**
  - **Overfitting** starts happening and we need to do **early stopping** at this point

Add Regularization

Reduce Overfitting

- **Optimization problem**

$$\min_{\boldsymbol{\theta}} J(\theta) = \Omega(\theta) + \frac{1}{N}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} l\big(y_i, f(\mathrm{x_i}; \theta)\big)$$

- **L2 regularization**

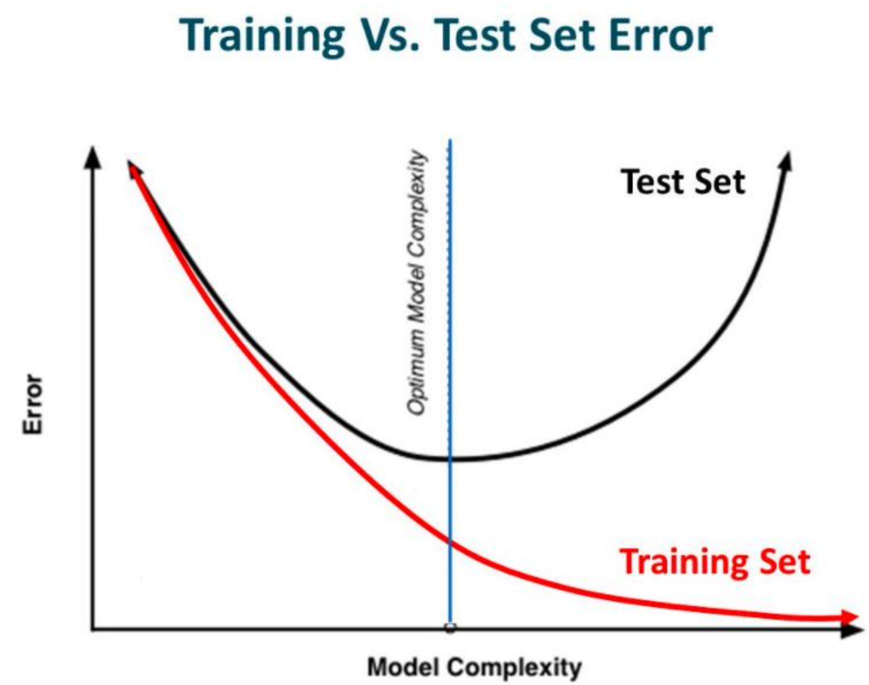
$$\Omega(\theta) = \lambda \sum_{k}\sum_{i,j}\left(W_{i,j}^{\mathbf{k}}\right)^2 = \lambda \sum_{k} \left\|\boldsymbol{W}^{\mathbf{k}}\right\|_F^2$$

  - $\lambda > 0$ is regularization parameter
  - Gradient: $\nabla_{\boldsymbol{W}^k}\Omega(\boldsymbol{\theta}) = 2\boldsymbol{W}^k$
  - Apply on weights ($\boldsymbol{W}$) only, not on biases ($\boldsymbol{b}$)

- **L1 regularization**

$$\Omega(\boldsymbol{\theta}) = \lambda \sum_{k}\sum_{i,j} \left|W_{i,j}^{\mathbf{k}}\right|$$

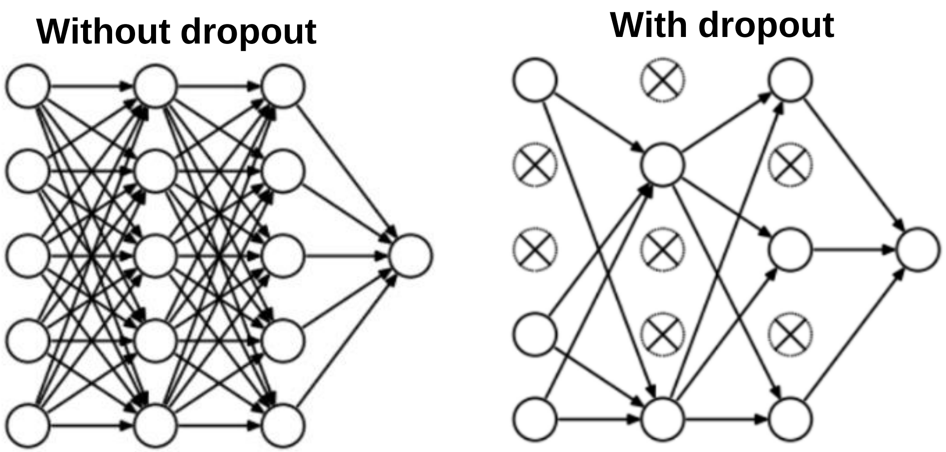
  - $\lambda > 0$ is regularization parameter
  - Optimization is now much harder – subgradient.
  - Apply on weights ($\boldsymbol{W}$) only, not on biases ($\boldsymbol{b}$)



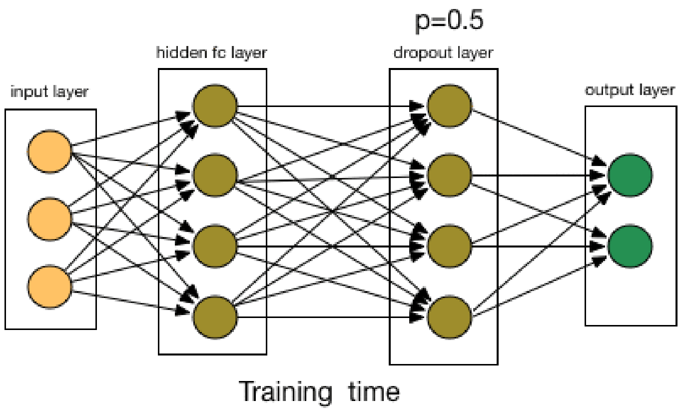
λ increases

Dropout

Reduce Overfitting



(Source: Analytics Vidhya)



(Source: chatbotslife)

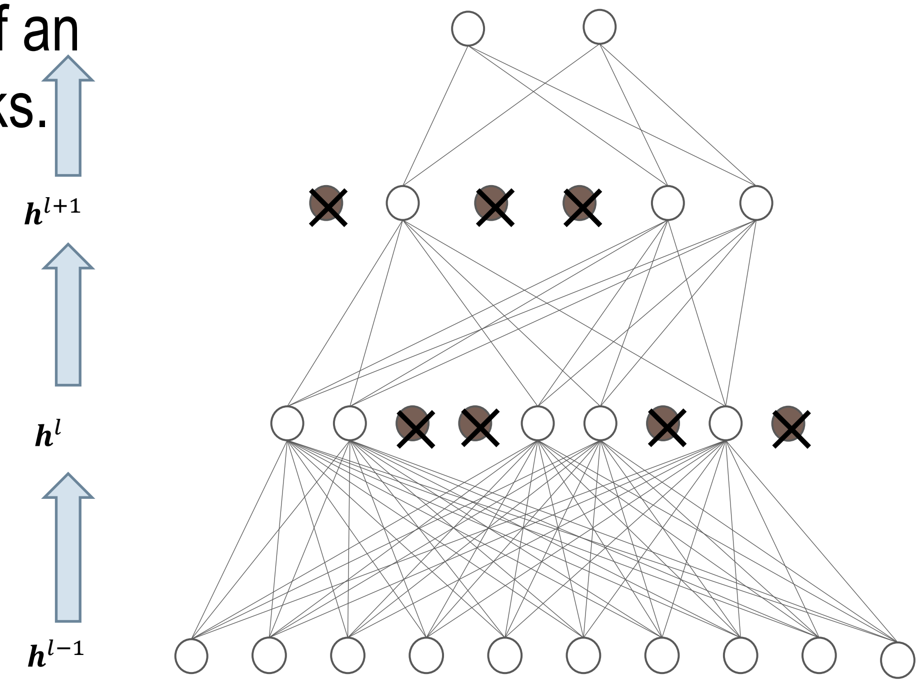
- This is a **cheap technique** to reduce model capacity
  - Reduce overfitting
- In each iteration, at each layer, **randomly choose** some neurons and **drop all connections** from these neurons
  - dropout_rate= 1 – keep_prob

Dropout

Reduce Overfitting

- Computationally efficient
- Can be considered a **bagged ensemble** of an exponential number (2^N) of neural networks.
- **Training**

$$\boldsymbol{r} \sim \text{Bernoulli}(\mu)$$
$$\widetilde{\boldsymbol{h}}^l = \boldsymbol{h}^l \odot \boldsymbol{r}$$
$$\boldsymbol{h}^{l+1} = \sigma\left(W^{(l)^\top} \widetilde{\boldsymbol{h}}^l + \boldsymbol{b}^l\right)$$



- **Testing**
  - **No dropout** (dropout_rate =0).

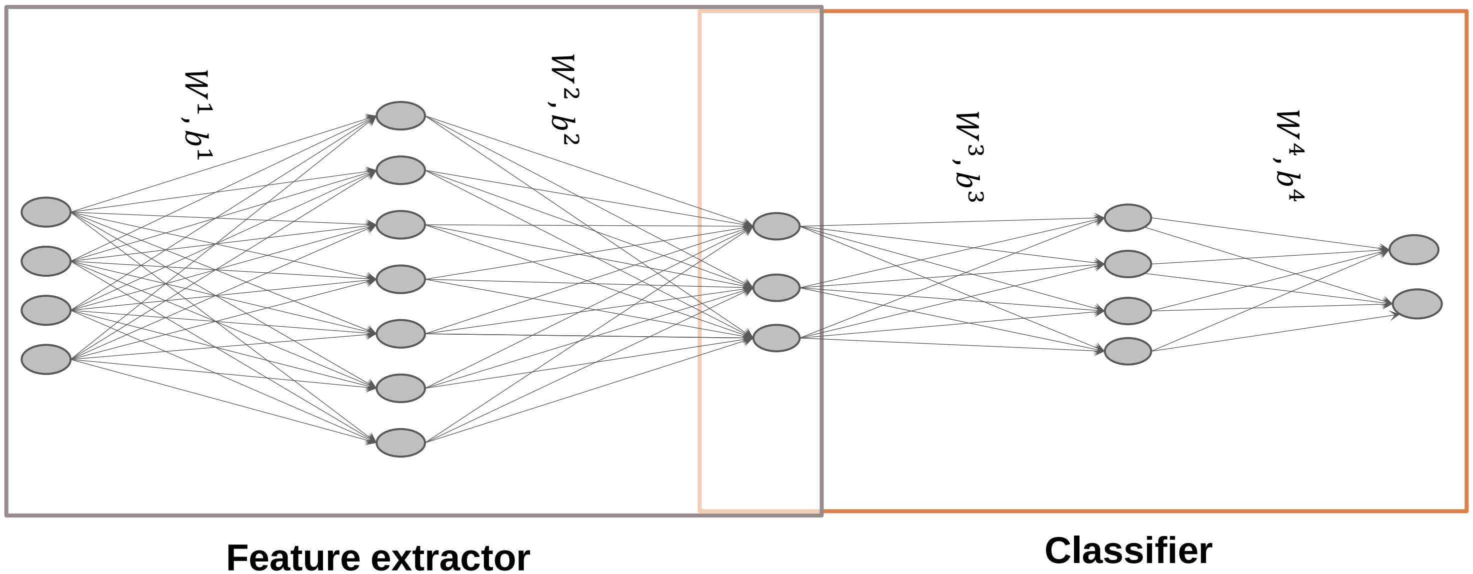
Internal covariate shift problem

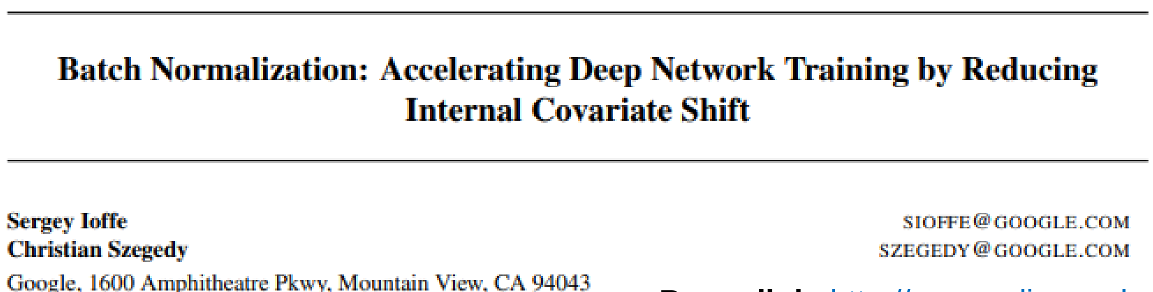
Input batch

x_1^1	x_2^1	x_3^1	x_4^1
x_1^2	x_2^2	x_3^2	x_4^2
x_1^3	x_2^3	x_3^3	x_4^3
x_1^4	x_2^4	x_3^4	x_4^4

Representation batch

z_1^1	z_2^1	z_3^1
z_1^2	z_2^2	z_3^2
z_1^3	z_2^3	z_3^3
z_1^4	z_2^4	z_3^4



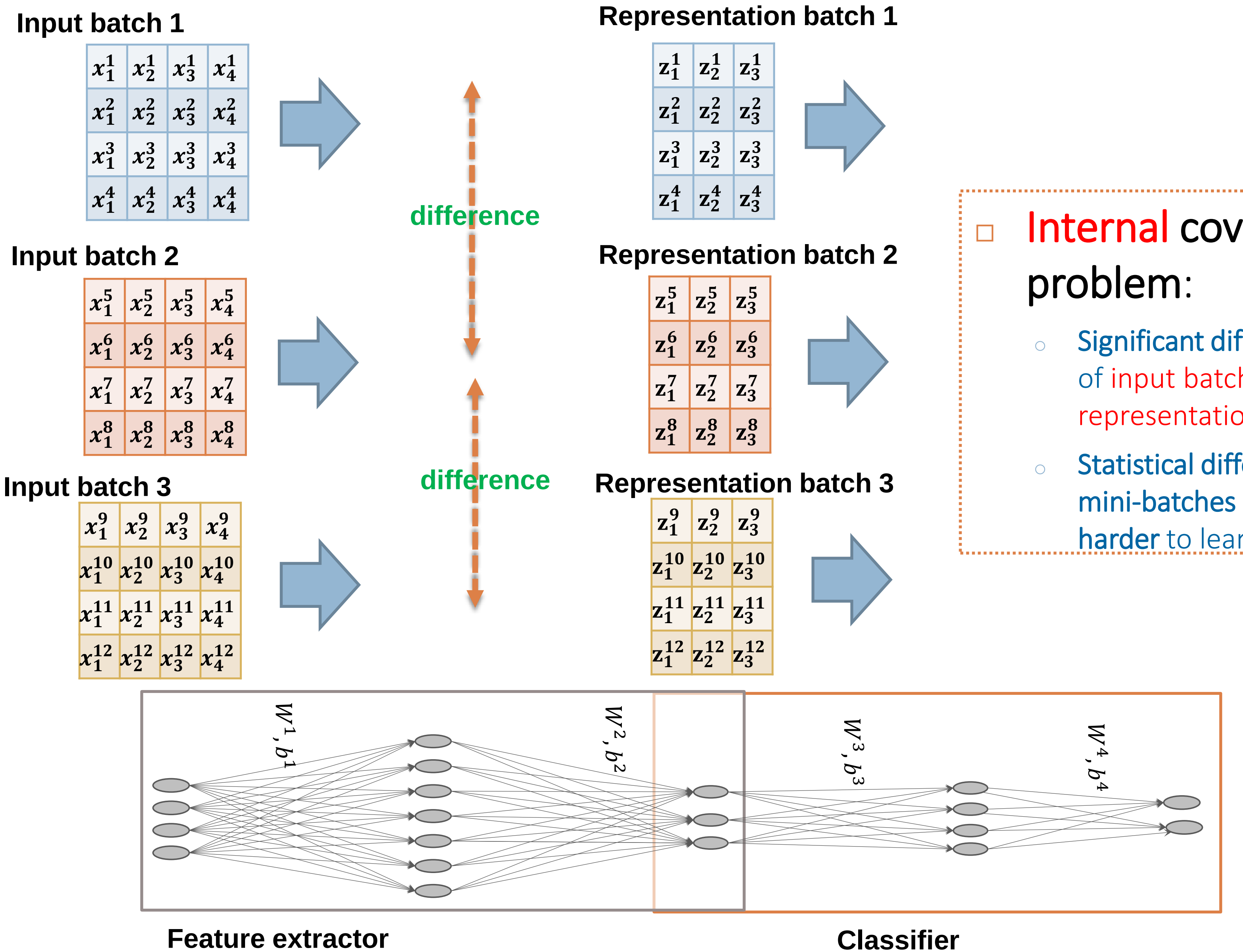


Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe SIOFFE@GOOGLE.COM
Christian Szegedy SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

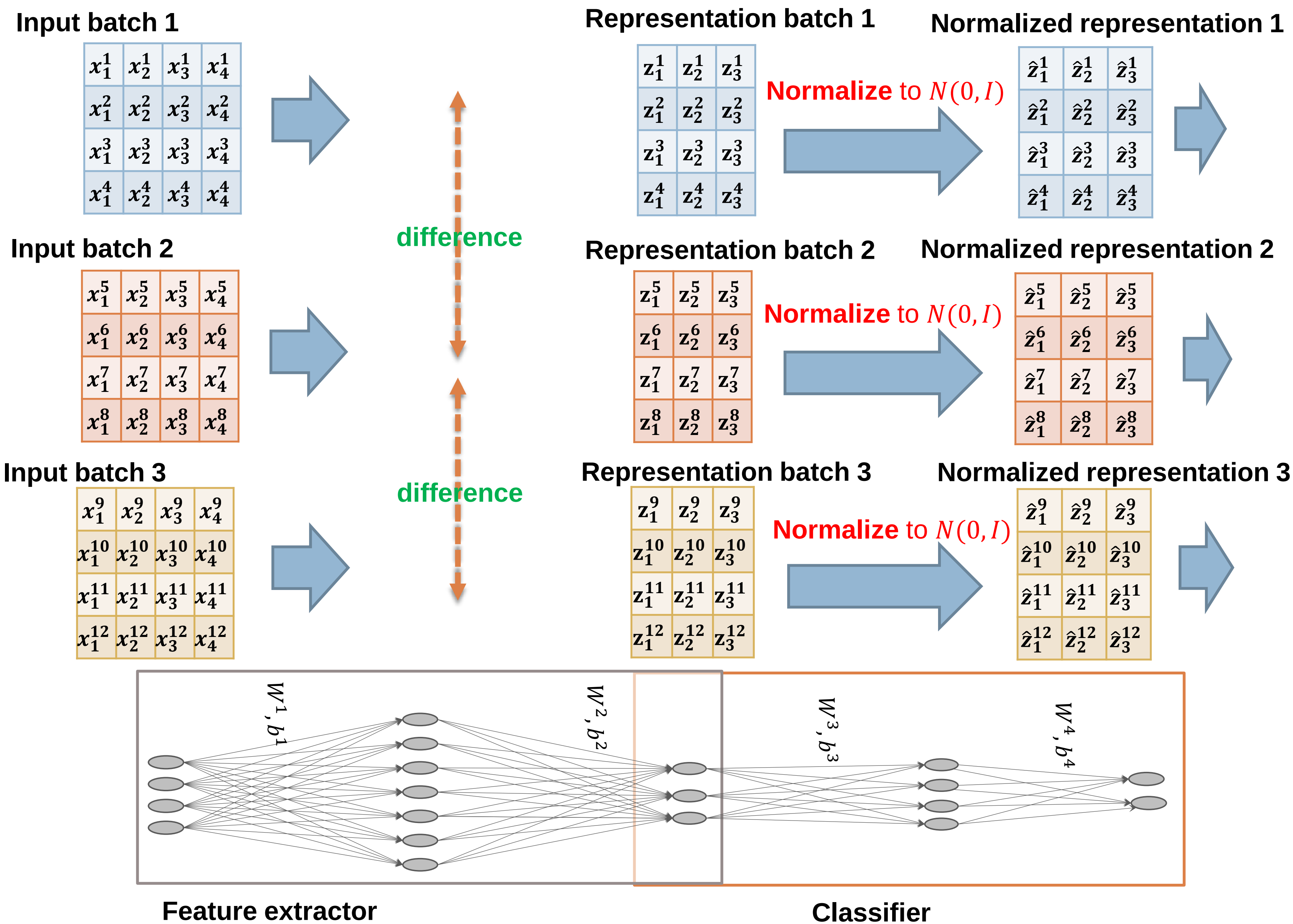
Paper link: http://proceedings.mlr.press/v37/ioffe15.pdf

Internal covariate shift problem

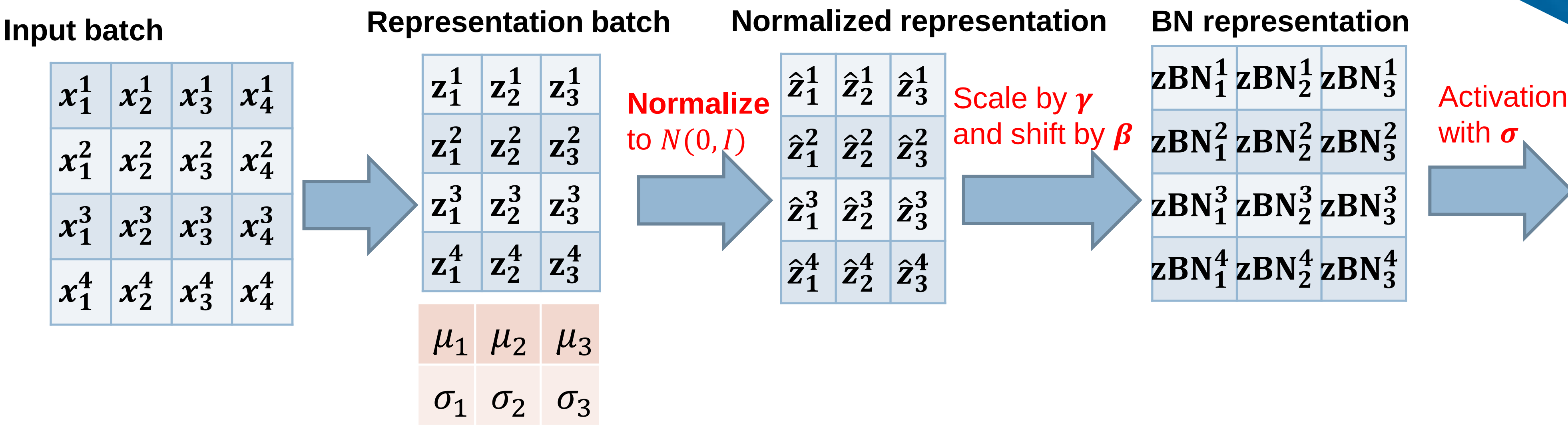


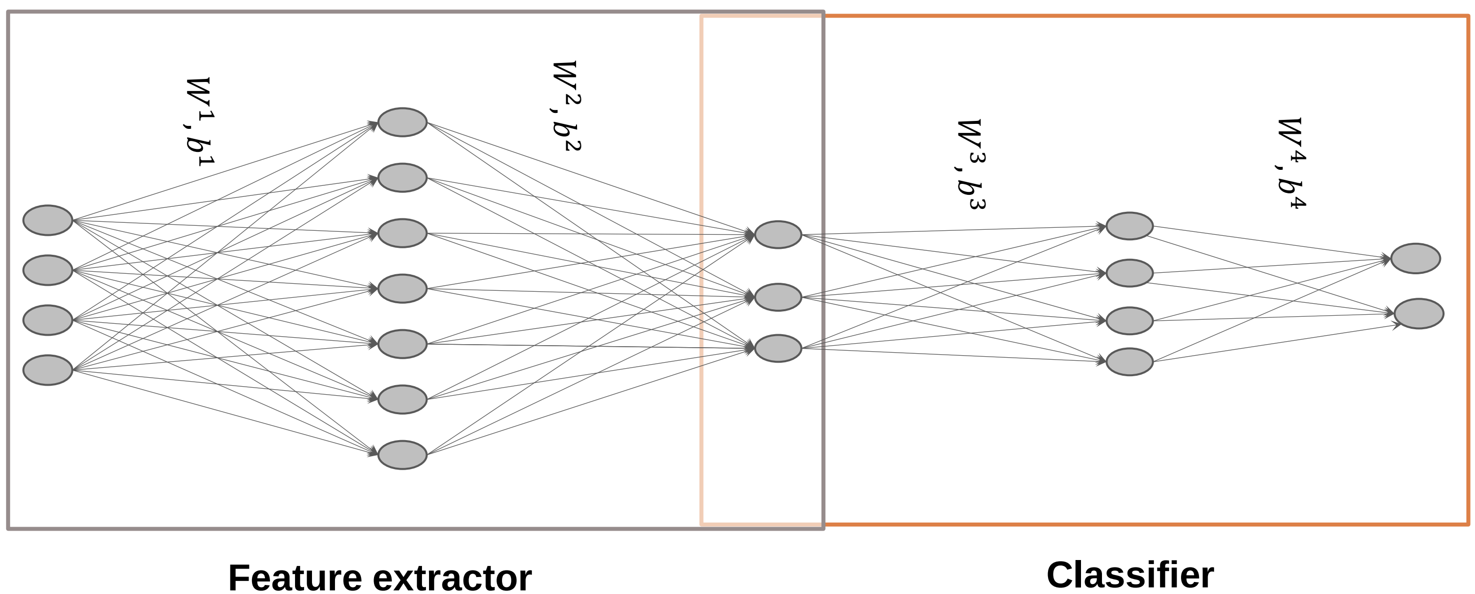
- Internal covariate shift problem:
  - Significant difference in statistics of input batches and also representation batches.
  - Statistical differences among mini-batches make the classifier harder to learn from data.

Batch Normalization



Batch Normalization

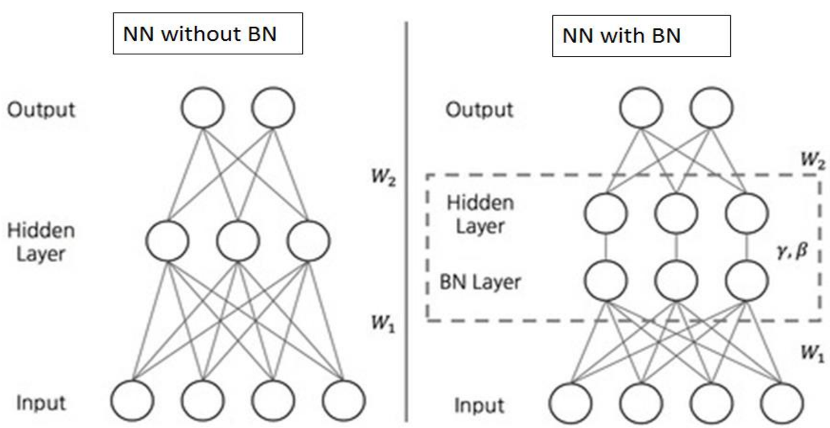


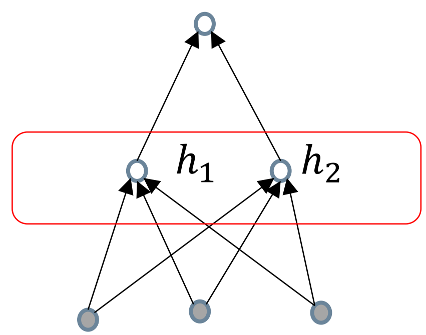


Batch Normalization

1. Cope with internal covariate shift
2. Reduce gradient vanishing/exploding
3. Reduce overfitting
4. Make training more stable
5. Converge faster
    1. Allow us to train with bigger learning rate

- Let $z = W^k h^k + b^k$ be the mini-batch before activation. We compute the normalized $\hat{z}$ as
    - $\hat{z} = \frac{z - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ where ϵ is a small value such as $1e^{-7}$
    - $\mu_B = \frac{1}{b}\sum_{i=1}^{b} z_i$ is the empirical mean
    - $\sigma_B^2 = \frac{1}{b}\sum_{i=1}^{b}(z_i - \mu_B)^2$ is the empirical variance
- We scale the normalized $\hat{z}$
    - $z_{BN} = \gamma\hat{z} + \beta$ where $\gamma, \beta > 0$ are two learnable parameters (i.e., scale and shift parameters)
- We then apply the activation to obtain the next layer value
    - $h^{k+1} = \sigma(z_{BN})$





z_1	z_2
0.1	0.2
…	…
0.5	0.4

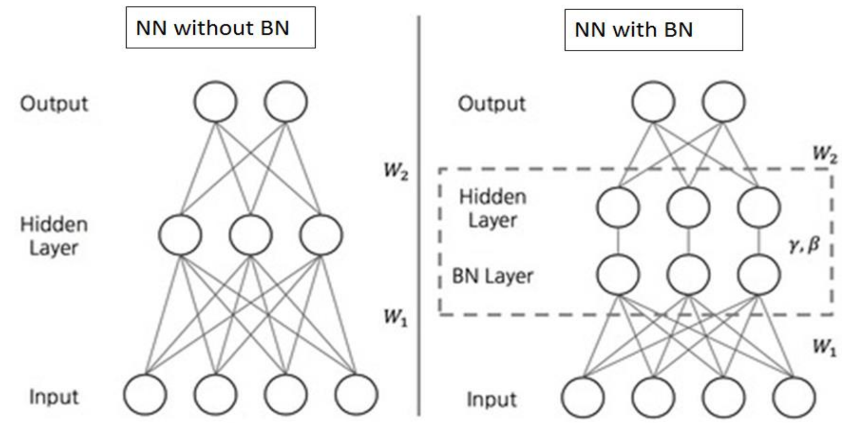
Mini-batch size

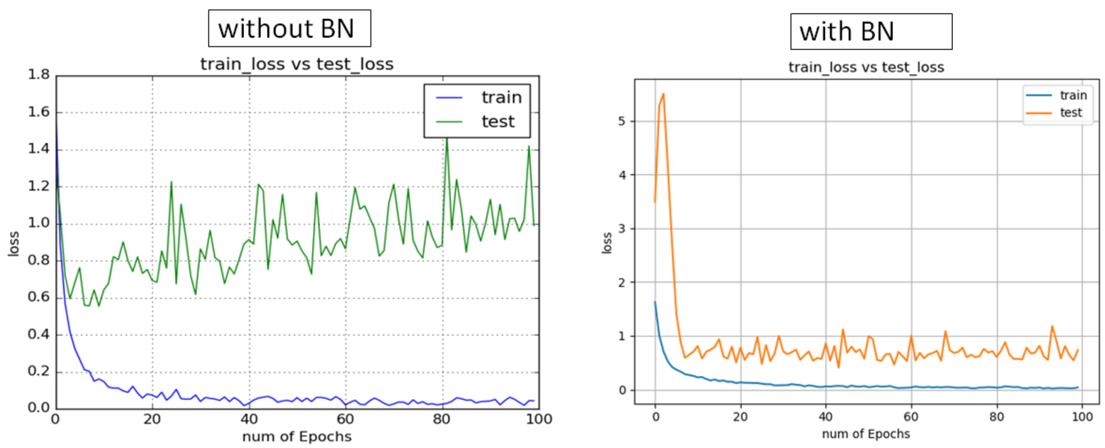
Standardize column-wise

Batch Normalization

Testing Phase

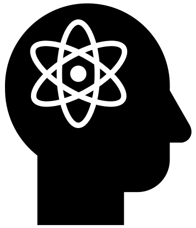
- At testing time, let's say we only want to test on a single input x
- Hence, we don't have a set of mini-batch samples to compute mean and standard deviation.
- So, we replace the mini-batch μ_B and σ_B with running averages of $\tilde{\mu}_B$ and $\tilde{\sigma}_B$ computed during the training process.
  - $\tilde{\mu}_B = \theta\tilde{\mu}_B + (1-\theta)\mu_B$
  - $\tilde{\sigma}_B = \theta\tilde{\sigma}_B + (1-\theta)\sigma_B$
  - $0 < \theta < 1$ is the momentum decay (i.e., $\theta = 0.9$).





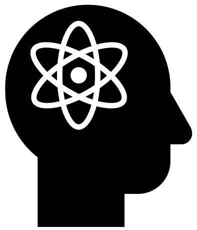
(Source: medium.com)

Data Augmentation



Reality

- You have a **tiny dataset** of **10K images**, and you need to train a **good deep net**.

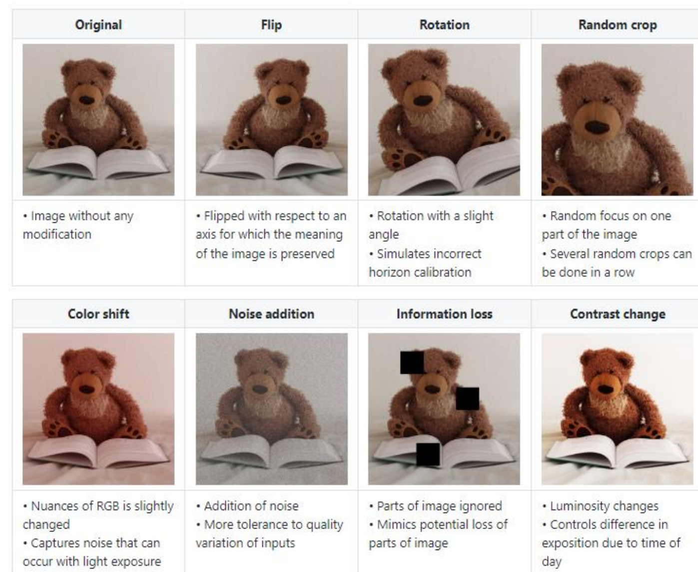


What are the **criteria** of a **qualified training set**?

Quantity?
- Collect **more** and **more data**?

Quality?
- More **diverge** data?
- More **qualified** data?

Data Augmentation





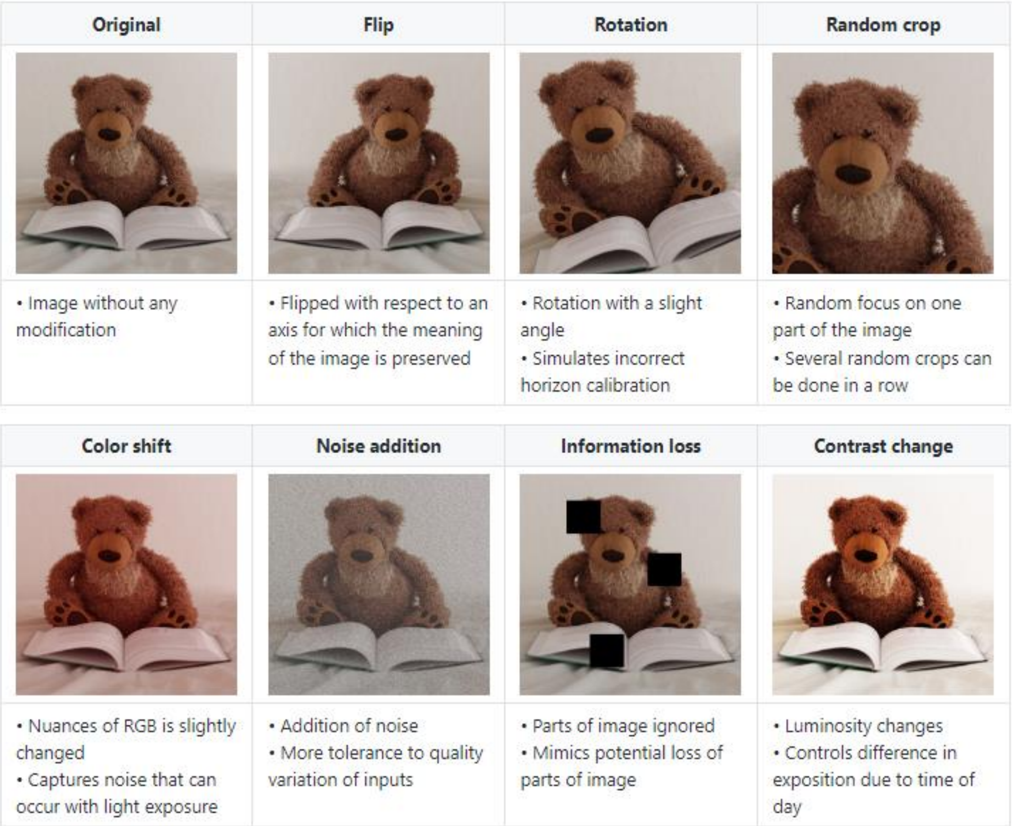
Create many variants

Prof Dinh Phung

Data Augmentation

- Use **simple transformations** to augment data examples. Models will be **challenged** with **diverge data examples** which might be **encountered** in the testing set
  - Rotation, Width Shifting, Height Shifting, Brightness, Shear Intensity, Zoom, Channel Shift, Horizontal Flip, Vertical Flip

- This will **reduce overfitting**, making this a **regularization technique**. The trick is to **generate realistic training instances**

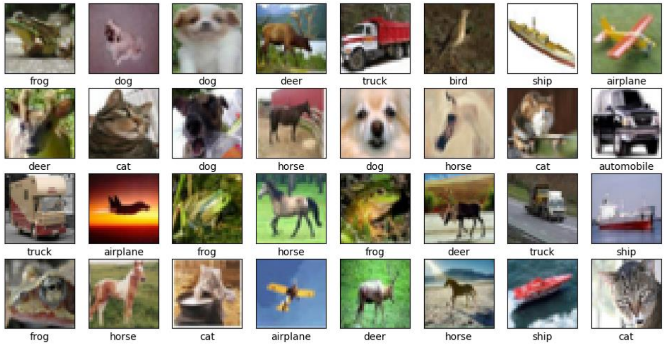


[Source: Stanford CS 230 Deep Learning]

Data Augmentation in PyTorch

```
test_transform = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),  # Normalize the images, each R,G,B value is normalized with mean=0.5 and std=0.5
                                     transforms.Resize((32,32)),  #resises the image so it can be perfect for our model.
                                     ])

train_transform = transforms.Compose([transforms.Resize((32,32)),  #resises the image so it can be perfect for our model.
                                      transforms.RandomHorizontalFlip(), # FLips the image w.r.t horizontal axis
                                      #transforms.RandomRotation(4),     #Rotates the image to a specified angel
                                      #transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)), #Performs actions like zooms, change shear angles.
                                      transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2), # Set the color params
                                      transforms.ToTensor(), # convert the image to tensor so that it can work with torch
                                      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),  # Normalize the images, each R,G,B value is normalized with mean=0.5 and std=0.5
                                      ])

full_train_set = torchvision.datasets.CIFAR10("./data", download=True, transform=train_transform)     # Apply train_transform to train set
full_valid_set = torchvision.datasets.CIFAR10("./data", download=True, transform=test_transform)      # Apply test_transform to generate valid set
full_test_set = torchvision.datasets.CIFAR10("./data", download=True, train=False, transform=test_transform)

n_train, n_valid, n_test = 5000, 5000, 5000

total_num_train = len(full_train_set)
total_num_test = len(full_test_set)
train_valid_idx = torch.randperm(total_num_train)
train_set = torch.utils.data.Subset(full_train_set, train_valid_idx[:n_train])
valid_set = torch.utils.data.Subset(full_valid_set, train_valid_idx[n_train:n_train+n_valid])

test_idx = torch.randperm(total_num_test)
test_set = torch.utils.data.Subset(full_test_set, test_idx[:n_test])

print("Traing set\n\t-Number of samples:\t{}\n\t-Shape of 1 sample:\t{}".format(len(train_set), list(train_set[0][0].shape)))
print("Valid set\n\t-Number of samples:\t{}\n\t-Shape of 1 sample:\t{}".format(len(valid_set), list(valid_set[0][0].shape)))
print("Test set\n\t-Number of samples:\t{}\n\t-Shape of 1 sample:\t{}".format(len(test_set), list(test_set[0][0].shape)))

train_loader = torch.utils.data.DataLoader(train_set, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=32)
valid_loader = torch.utils.data.DataLoader(valid_set, batch_size=32)

img_train = [train_set[idx][0].numpy().transpose((1,2,0)) for idx in range(32)]
label_train = [train_set[idx][1] for idx in range(32)]
```

Without augmentation



With augmentation

Some additional regularization techniques

Label smoothing

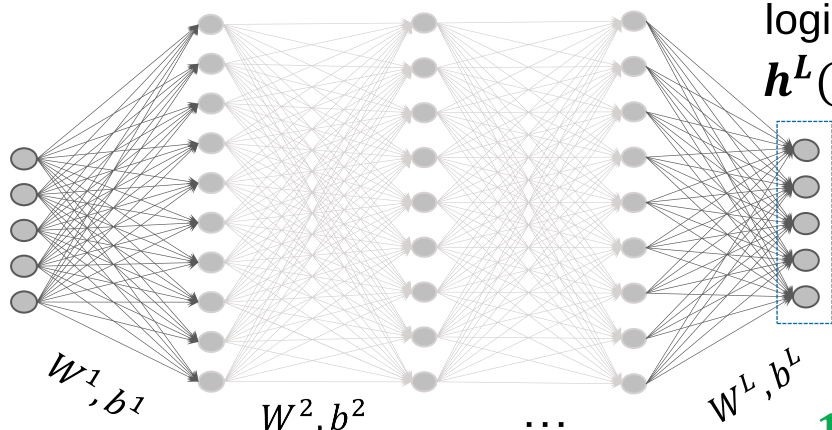
When Does Label Smoothing Help?

Rafael Müller,* Simon Kornblith, Geoffrey Hinton
Google Brain
Toronto
rafaelmuller@google.com

Paper link: https://papers.nips.cc/paper/2019/file/f1748d6b0fd9d439f71450117eba2725-Paper.pdf

x

$y \in \{1, \dots, M\}$



logits

$\boldsymbol{h^L(x)}$

$p(x) = softmax(h^L(x))$

$\hat{y} = argmax_{1 \le m \le M} p_m(x)$

No label smoothing

$l(y, \hat{y}) = CE(\mathbf{1}_y, \boldsymbol{p}(\boldsymbol{x}))$

$\mathbf{1}_y = [0, \dots, 1, \dots, 0]$

***y*-th** position

Label smoothing

$l(y, \hat{y}) = CE(\widetilde{\mathbf{1}}_y, \boldsymbol{p}(\boldsymbol{x}))$

$\widetilde{\mathbf{1}}_y = \left[\frac{\alpha}{M}, \dots, 1 - \alpha + \frac{\alpha}{M}, \dots, \frac{\alpha}{M}\right]$

***y*-th** position

- Given **data instance** $(\boldsymbol{x}, \boldsymbol{y})$ with the label $\boldsymbol{y} \in \{\mathbf{1}, \dots, \boldsymbol{M}\}$, we compute the **CE loss** between the **prediction probabilities** $\boldsymbol{p}(\boldsymbol{x})$ and the **smooth label**
  - $l(y, \hat{y}) = CE(\widetilde{\mathbf{1}}_y, \boldsymbol{p}(\boldsymbol{x}))$ with $\widetilde{\mathbf{1}}_y = (1 - \alpha) \times \mathbf{1}_y + \frac{\alpha}{M} \times \mathbf{1}$ where $\mathbf{1}$ is a vector of all 1 and $0 < \alpha < 1$.

Data mix-up

Published as a conference paper at ICLR 2018

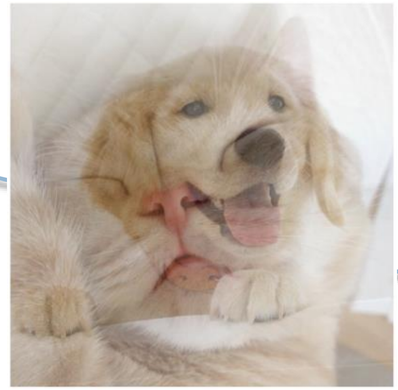
mixup: BEYOND EMPIRICAL RISK MINIMIZATION

Hongyi Zhang
MIT

Moustapha Cisse, Yann N. Dauphin, David Lopez-Paz*
FAIR

Paper link: https://openreview.net/pdf?id=r1Ddp1-Rb

$(x_1, \mathbf{1}_{y_1})$



$\lambda \sim \text{Beta}(\alpha, \alpha)$

Blended image $\tilde{x} = \lambda \times x_1 + (1 - \lambda) \times x_2$

Blended label $\tilde{y} = \lambda \times \mathbf{1}_{y_1} + (1 - \lambda) \times \mathbf{1}_{y_2}$

$\mathbf{min}\ \boldsymbol{CE}(\tilde{\boldsymbol{y}}, \boldsymbol{p}(\tilde{\boldsymbol{x}}))$

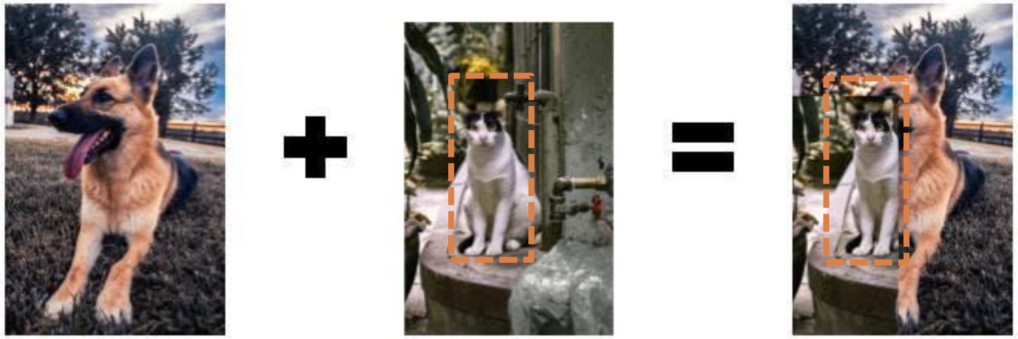
$(x_2, \mathbf{1}_{y_2})$

[Source: https://medium.com/]

- for (x_1, y_1), (x_2, y_2) in zip(batch1, batch 2)
  1. $\lambda \sim \text{Beta}(\alpha, \alpha)$
  2. $\tilde{x} = \lambda \times x_1 + (1 - \lambda) \times x_2$
  3. $\tilde{y} = \lambda \times \mathbf{1}_{y_1} + (1 - \lambda) \times \mathbf{1}_{y_2}$
  4. Update optimizer to minimize $\boldsymbol{CE}(\tilde{\boldsymbol{y}}, \boldsymbol{p}(\tilde{\boldsymbol{x}}))$

Cut-mix

[Source: https://encord.com/blog/data-augmentation-guide/]



$(x_1, \mathbf{1}_{y_1}), x_1 \in \mathbb{R}^{C\times W\times H}$ $\qquad$ $\lambda \sim \mathrm{Beta}(\alpha,\alpha)$ $\qquad$ $(x_2, \mathbf{1}_{y_2}), x_2 \in \mathbb{R}^{C\times W\times H}$

$$\tilde{x} = M \odot x_1 + (1-M) \odot x_2, M \in \{0,1\}^{H\times W}$$

$$\tilde{y} = \lambda \times \mathbf{1}_{y_1} + (1-\lambda) \times \mathbf{1}_{y_2}$$

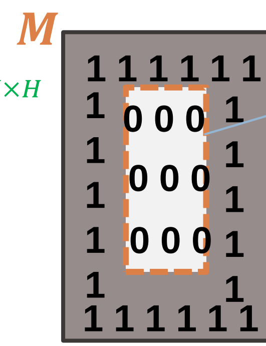
$$\Rightarrow \min CE(\tilde{y}, p(\tilde{x}))$$

CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features

Sangdoo Yun[1] Dongyoon Han[1] Junsuk Choe[1,3] Seong Joon Oh[2] Youngjoon Yoo[1] Sanghyuk Chun[1]

[1]Clova AI Research, NAVER Corp.
[2]Clova AI Research, LINE Plus Corp.
[3]Yonsei University

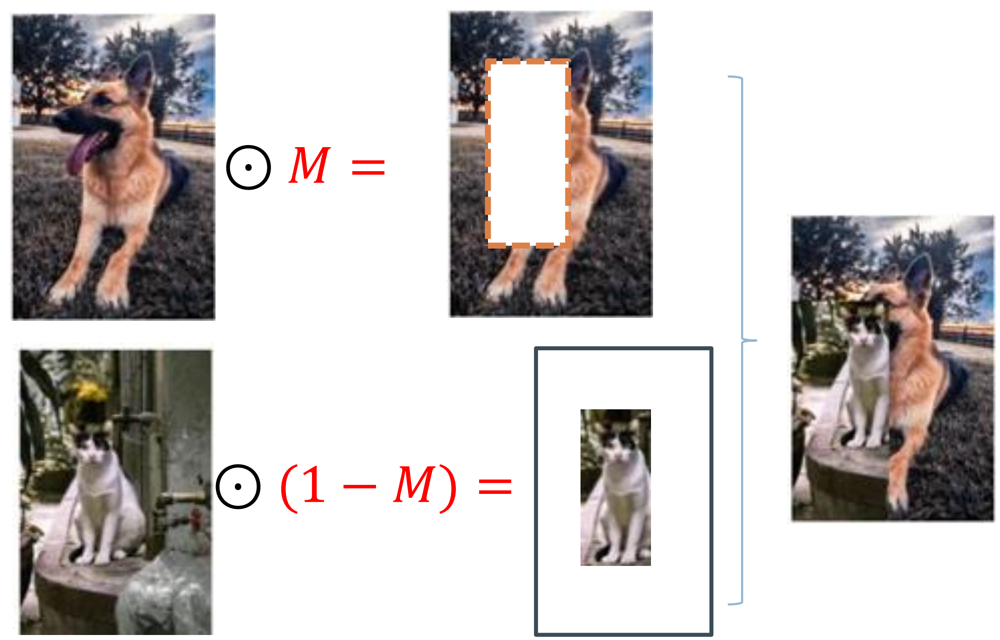
[Source: https://arxiv.org/pdf/1905.04899]



$B = [r_x, r_y, r_w, r_h]$

$$\frac{area(B)}{area(image)} = \frac{WH(1-\lambda)}{WH} = 1-\lambda$$

- for (x_1, y_1), (x_2, y_2) in zip(batch1, batch 2)
  1. $\lambda \sim \mathrm{Beta}(\alpha,\alpha)$
  2. Sample a **bounding box** $B = [r_x, r_y, r_w, r_h]$
     1. $r_x \sim Uni[0, W],\ r_w \sim W\sqrt{1-\lambda}$
     2. $r_y \sim Uni[0, H], r_h \sim H\sqrt{1-\lambda}$
  3. $\boldsymbol{M} \in \{\mathbf{0},\mathbf{1}\}^{\boldsymbol{W\times H}}$ by filling **1** within B and **0** otherwise
  4. $\tilde{x} = M \odot x_1 + (1-M) \odot x_2$
  5. Update optimizer to minimize $CE(\tilde{y}, p(\tilde{x}))$

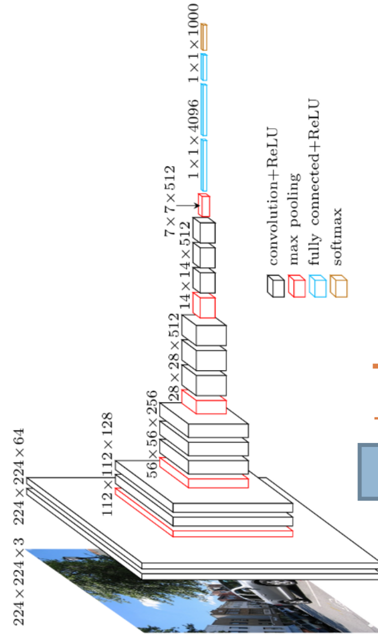


Transfer learning and fine-tuning

Transfer Learning

Motivation

Large-scale dataset (ImageNet)
- over 1.2 million images and 1,000 possible object categories



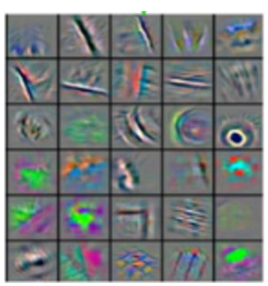
Powerful pretrained model
- VGG16, VGG19
- Inception V3
- Xception
- ResNet

Transfer learning + Fine tune

Your small-scale dataset (Flower-17)
-17 category dataset with 80 images per class

Not enough data to train a good model. HOW?

Low-level filers



Mid-level filers

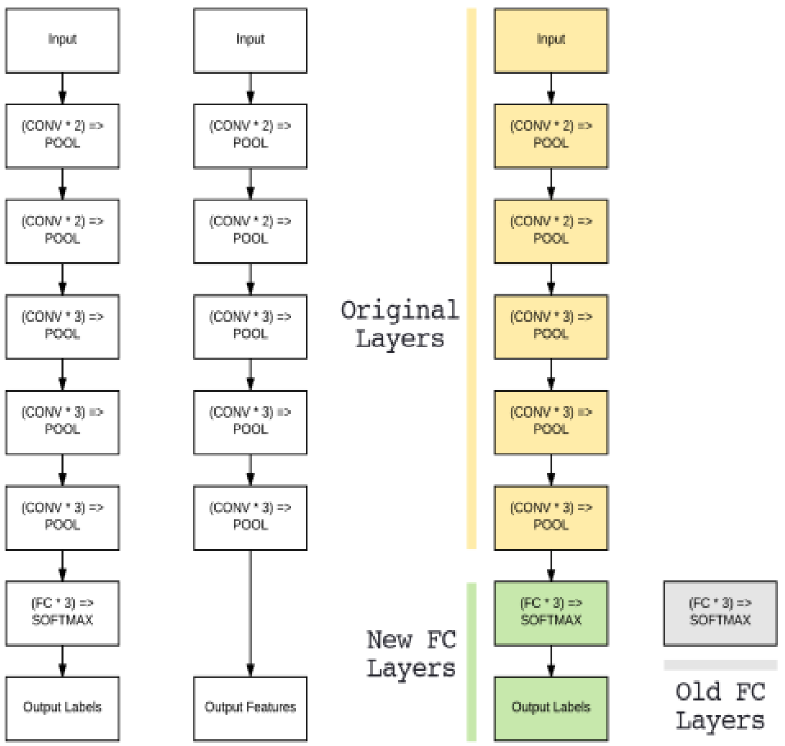
High-level filers

Fits the target dataset

Transfer Learning

How to Do That?

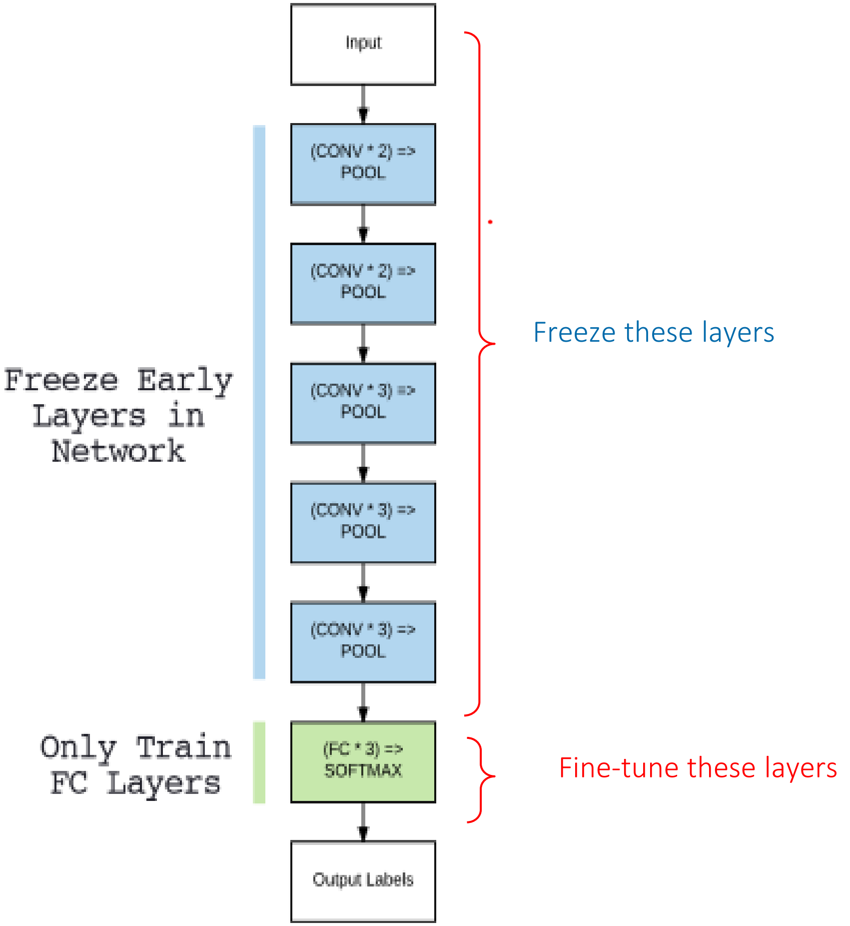
- **Remove** FC layers from the pretrained model
- **Replace** them with a brand-new FC head.
  - These new FC layers can then be fine-tuned to the specific dataset
  - The old FC layers are no longer used



Transfer Learning

How to Do That?

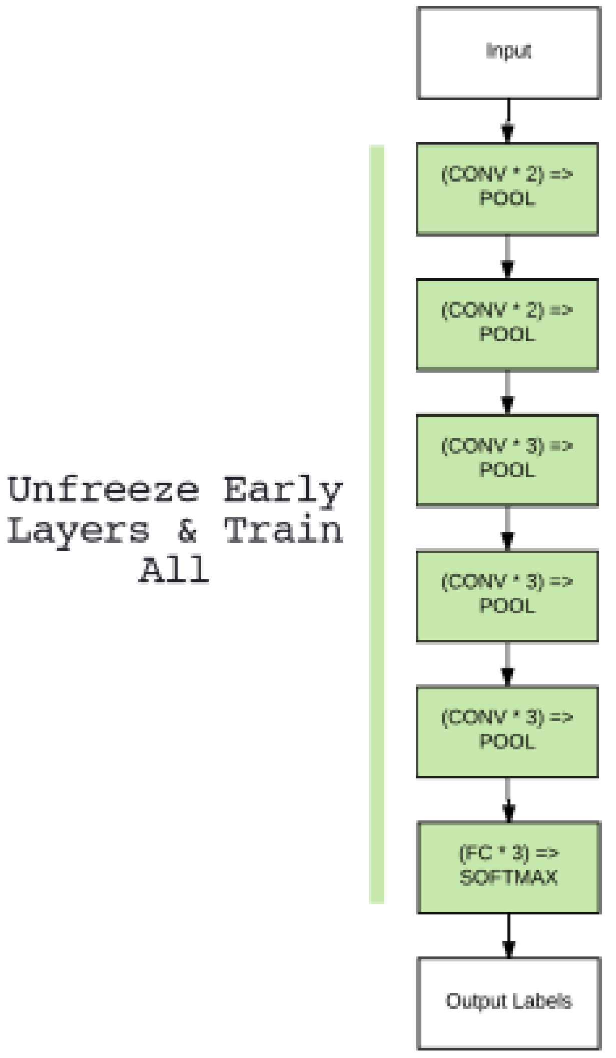
- **Freeze** all CONV layers in the network
  - Only allow the gradient to backpropagate through the FC layers
  - Doing this allows our network to **warm up**
- Training data is forward propagated through the network
  - However, the backpropagation is stopped after the FC layers
  - Allows these layers to start to learn patterns from the highly discriminative CONV layers



Transfer Learning

How to Do That?

- After the FC layers have had a chance to warm up, we may choose to **unfreeze** all layers in the network
  - Allow each of them to be **fine-tuned**.
  - Continue training the entire network, *but with a **very small learning rate***
  - We do not want to **deviate our CONV filters** dramatically. Training is then allowed to continue until sufficient accuracy is obtained.



Model zoo supported by PyTorch

Model	Acc@1	Acc@5
AlexNet	56.522	79.066
VGG-11	69.020	88.628
VGG-13	69.928	89.246
VGG-16	71.592	90.382
VGG-19	72.376	90.876
VGG-11 with batch normalization	70.370	89.810
VGG-13 with batch normalization	71.586	90.374
VGG-16 with batch normalization	73.360	91.516
VGG-19 with batch normalization	74.218	91.842
ResNet-18	69.758	89.078
ResNet-34	73.314	91.420
ResNet-50	76.130	92.862
ResNet-101	77.374	93.546
ResNet-152	78.312	94.046
SqueezeNet 1.0	58.092	80.420
SqueezeNet 1.1	58.178	80.624
Densenet-121	74.434	91.972
Densenet-169	75.600	92.806
Densenet-201	76.896	93.370
Densenet-161	77.138	93.560
Inception v3	77.294	93.450
GoogleNet	69.778	89.530
ShuffleNet V2 x1.0	69.362	88.316
ShuffleNet V2 x0.5	60.552	81.746
MobileNet V2	71.878	90.286
MobileNet V3 Large	74.042	91.340
MobileNet V3 Small	67.668	87.402
ResNeXt-50-32x4d	77.618	93.698
ResNeXt-101-32x8d	79.312	94.526
Wide ResNet-50-2	78.468	94.086
Wide ResNet-101-2	78.848	94.284
MNASNet 1.0	73.456	91.510
MNASNet 0.5	67.734	87.490

Transfer learning with PyTorch

```python
# Load pretrained VGG19 model
model = models.vgg19(pretrained=True)
model = model.to(device)
summary(model, (3, 224, 224))
```

```
/usr/local/lib/python3.10/dist-packages/torch
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/mo
100%|██████████| 548M/548M [00:07<00:00, 75.8
```

Load pretrained VGG19

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Will be **replaced** by a **fresh new linear layer**

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
           Conv2d-17          [-1, 256, 56, 56]         590,080
             ReLU-18          [-1, 256, 56, 56]               0
        MaxPool2d-19          [-1, 256, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       1,180,160
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
           Conv2d-24          [-1, 512, 28, 28]       2,359,808
             ReLU-25          [-1, 512, 28, 28]               0
           Conv2d-26          [-1, 512, 28, 28]       2,359,808
             ReLU-27          [-1, 512, 28, 28]               0
        MaxPool2d-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
           Conv2d-31          [-1, 512, 14, 14]       2,359,808
             ReLU-32          [-1, 512, 14, 14]               0
           Conv2d-33          [-1, 512, 14, 14]       2,359,808
             ReLU-34          [-1, 512, 14, 14]               0
           Conv2d-35          [-1, 512, 14, 14]       2,359,808
             ReLU-36          [-1, 512, 14, 14]               0
        MaxPool2d-37            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-38            [-1, 512, 7, 7]               0
           Linear-39                 [-1, 4096]     102,764,544
             ReLU-40                 [-1, 4096]               0
          Dropout-41                 [-1, 4096]               0
           Linear-42                 [-1, 4096]      16,781,312
             ReLU-43                 [-1, 4096]               0
          Dropout-44                 [-1, 4096]               0
           Linear-45                 [-1, 1000]       4,097,000
================================================================
Total params: 143,667,240
Trainable params: 143,667,240
```

Transfer learning with PyTorch

Warm-up the fresh new linear layer

```
# Freeze all layers
for param in model.parameters():
    param.requires_grad = False

# Modify the last fully connected layer for Flower102
num_features = model.classifier[6].in_features
model.classifier[6] = nn.Linear(num_features, 102)
model = model.to(device)
```

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3, momentum=0.9)
trainer = BaseTrainer(model, criterion, optimizer, train_loader, val_loader)
trainer.fit(num_epochs=10)
```

```
Epoch 1/10
 train_loss: 0.9840 - train_accuracy: 0.7027 - val_loss: 0.4316 - top1_acc: 0.8501 - top5_acc: 1.0000
Epoch 2/10
 train_loss: 0.4744 - train_accuracy: 0.8287 - val_loss: 0.3686 - top1_acc: 0.8706 - top5_acc: 1.0000
Epoch 3/10
 train_loss: 0.4272 - train_accuracy: 0.8505 - val_loss: 0.3627 - top1_acc: 0.8760 - top5_acc: 1.0000
Epoch 4/10
 train_loss: 0.3894 - train_accuracy: 0.8535 - val_loss: 0.3507 - top1_acc: 0.8787 - top5_acc: 1.0000
Epoch 5/10
 train_loss: 0.3738 - train_accuracy: 0.8631 - val_loss: 0.3283 - top1_acc: 0.8937 - top5_acc: 1.0000
Epoch 6/10
 train_loss: 0.3593 - train_accuracy: 0.8719 - val_loss: 0.3252 - top1_acc: 0.8815 - top5_acc: 1.0000
Epoch 7/10
 train_loss: 0.3309 - train_accuracy: 0.8886 - val_loss: 0.3134 - top1_acc: 0.8910 - top5_acc: 1.0000
Epoch 8/10
 train_loss: 0.3180 - train_accuracy: 0.8903 - val_loss: 0.3070 - top1_acc: 0.8924 - top5_acc: 1.0000
Epoch 9/10
 train_loss: 0.3062 - train_accuracy: 0.8917 - val_loss: 0.3121 - top1_acc: 0.8978 - top5_acc: 1.0000
Epoch 10/10
 train_loss: 0.3139 - train_accuracy: 0.8849 - val_loss: 0.3082 - top1_acc: 0.8924 - top5_acc: 1.0000
```

Fine-tune the entire model

```
# Unfreeze the last convolutional block
# for param in model.features[-7:].parameters():
#     param.requires_grad = True

# Unfreeze model parameter
for param in model.parameters():
    param.requires_grad = True

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
trainer = BaseTrainer(model, criterion, optimizer, train_loader, val_loader)
trainer.fit(num_epochs=10)
```

```
Epoch 1/10
 train_loss: 0.2559 - train_accuracy: 0.9114 - val_loss: 0.2631 - top1_acc: 0.9101 - top5_acc: 1.0000
Epoch 2/10
 train_loss: 0.1983 - train_accuracy: 0.9305 - val_loss: 0.2489 - top1_acc: 0.9196 - top5_acc: 1.0000
Epoch 3/10
 train_loss: 0.1561 - train_accuracy: 0.9472 - val_loss: 0.2435 - top1_acc: 0.9196 - top5_acc: 1.0000
Epoch 4/10
 train_loss: 0.1417 - train_accuracy: 0.9513 - val_loss: 0.2349 - top1_acc: 0.9210 - top5_acc: 1.0000
Epoch 5/10
 train_loss: 0.1089 - train_accuracy: 0.9632 - val_loss: 0.2379 - top1_acc: 0.9223 - top5_acc: 1.0000
Epoch 6/10
 train_loss: 0.0947 - train_accuracy: 0.9690 - val_loss: 0.2370 - top1_acc: 0.9223 - top5_acc: 1.0000
Epoch 7/10
 train_loss: 0.0865 - train_accuracy: 0.9690 - val_loss: 0.2356 - top1_acc: 0.9319 - top5_acc: 1.0000
Epoch 8/10
 train_loss: 0.0644 - train_accuracy: 0.9816 - val_loss: 0.2403 - top1_acc: 0.9319 - top5_acc: 1.0000
Epoch 9/10
 train_loss: 0.0516 - train_accuracy: 0.9816 - val_loss: 0.2477 - top1_acc: 0.9319 - top5_acc: 1.0000
Epoch 10/10
 train_loss: 0.0530 - train_accuracy: 0.9837 - val_loss: 0.2477 - top1_acc: 0.9251 - top5_acc: 1.0000
```

Summary

- Setting of a machine learning problem
  - General loss versus empirical loss
- Gradient vanishing/exploding and network initialization.
- Overfitting and underfitting
- Recipe for overfitting
  - Use regularization term
  - Dropout, batch norm
  - Data augmentation
  - Transfer learning
  - Label smoothing, data mix-up

Thanks for your attention!

Question time