# FIT3181 Deep Learning

## Week 05: Practical skills in deep learning

**Lecturer: Lim Chern Hong**

Email: lim.chernhong@monash.edu

GROUP OF EIGHT AUSTRALIA

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
- Challenges in solving DL optimization problem
  - Highly non-convex, complicated, overparameterized, too many saddle points.
- 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
- Visualize training process with Tensor board.

- 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

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

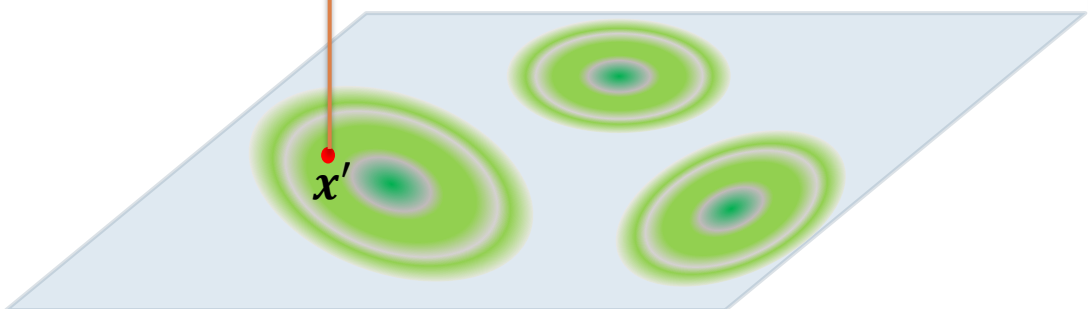
$y' = \mathbf{1}$ ***or*** $\boldsymbol{cat}$

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

| 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** $\mathcal{X}$

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

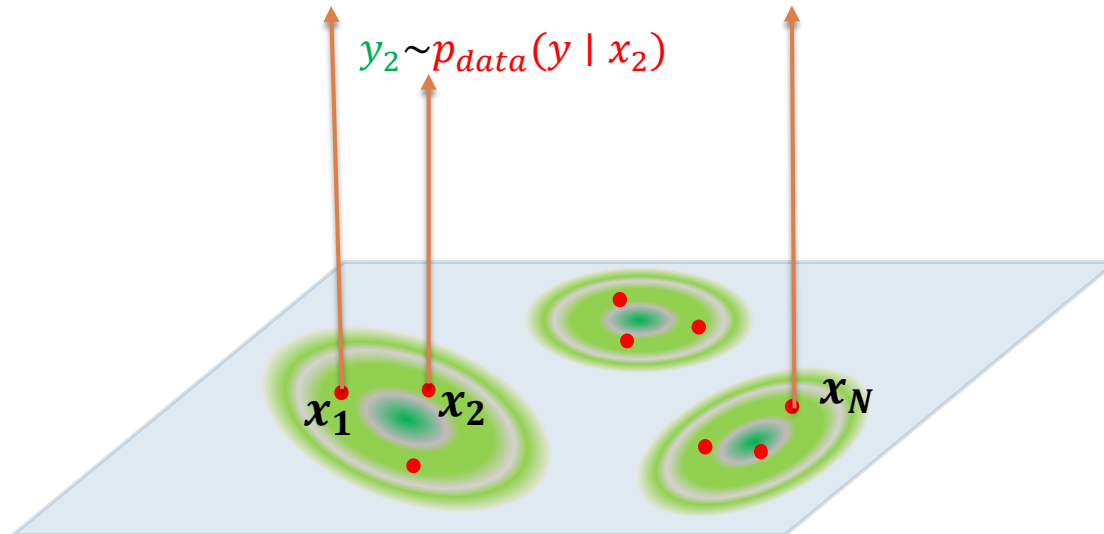
**Data distribution**

## Dataset collecting process

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

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

$y_1 \sim p_{data}(y \mid x_1)$ $y_2 \sim p_{data}(y \mid x_2)$ $y_N \sim p_{data}(y \mid x_N)$




**Data space** $\mathcal{X}$

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

**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

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

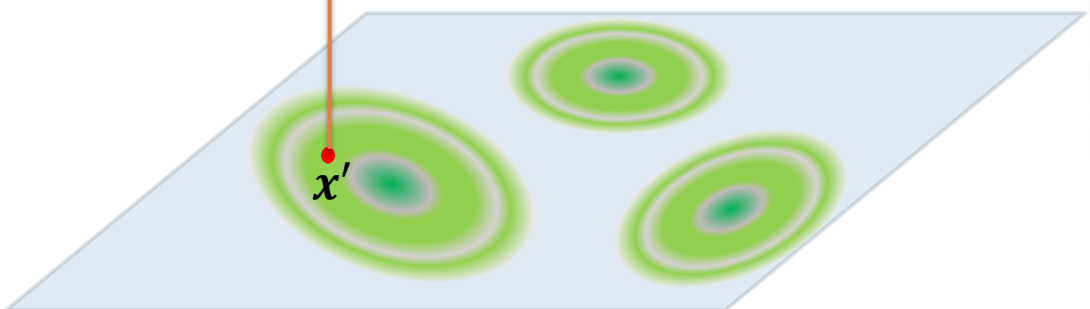
$y' = \mathbf{1}$ ***or cat***

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

| 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** $\mathcal{X}$

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

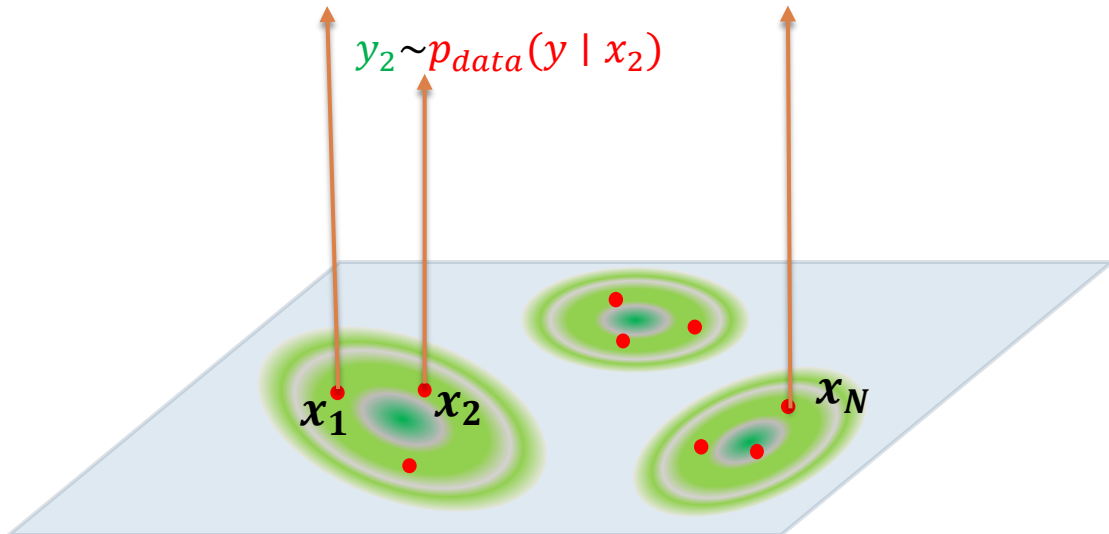
**Data distribution**

## Dataset collecting process

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

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

$y_1 \sim p_{data}(y \mid x_1)$ $y_2 \sim p_{data}(y \mid x_2)$ $y_N \sim p_{data}(y \mid x_N)$




**Data space** $\mathcal{X}$

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

**Data distribution**

Not in assessment

# Machine learning setting

Not in assessment

**testing**

**Generalization (general) loss**

***(loss on data/label distribution)***

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

$N \to \infty$

*Law of large numbers*

**training**

**Empirical loss**

***(loss on a collected training set D)***

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

**Ideal:** $\theta^* = \underset{\theta}{\operatorname{argmin}} J_{gen}(\theta)$

**Reality**: $\theta^* = \underset{\theta}{\operatorname{argmin}} J_{emp}(\theta)$

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

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

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

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

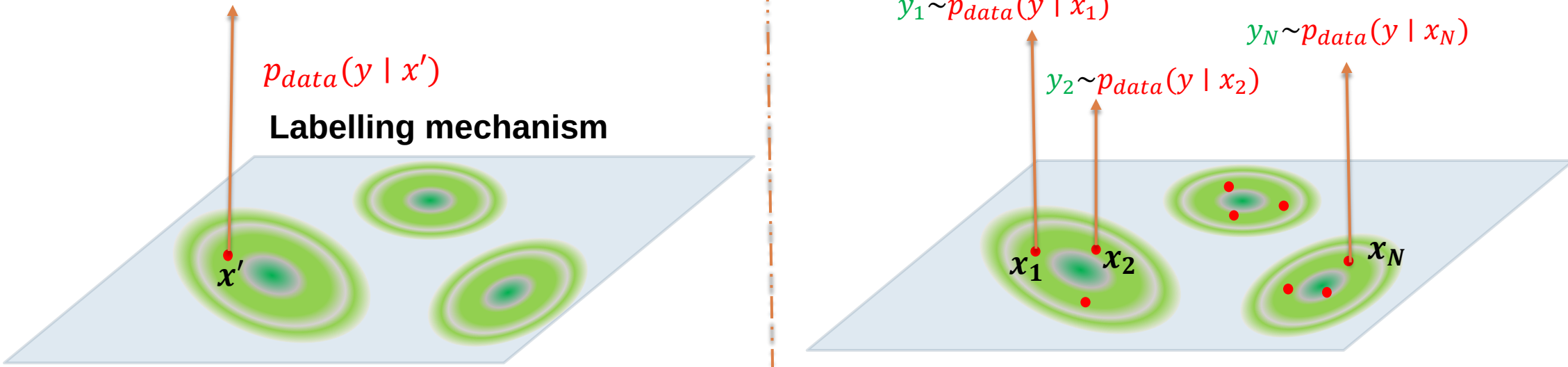
$$y_1 \sim p_{data}(y \mid x_1)$$

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

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

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

**Labelling mechanism**




**Data space** $x$

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

**Data distribution**

**Data space** $x$

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

**Data distribution**

# How Learning Differs from Pure Optimization?

- Some important notations:
  - $p_{data}(x, y)$: **existed**, but **unknown**, 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 distribution - this is what we observed from training data $\boldsymbol{D} = \{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{\boldsymbol{i=1}}^{\boldsymbol{N}}$
  - Per-sample loss: $l(f(x; \theta), y)$

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

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

Empirical loss (maths)

vs.

Generalisation loss (ML)

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

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

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:

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

Regularization term
- Encourage simple models
- Avoid overfitting

Empirical loss
- Work well on training set

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

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) $\mathbf{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ 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) $\mathbf{H} = \nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to update parameters

# Optimization Problem in ML and DL

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

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

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, $\varepsilon$ −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., negative log-likelihood, cross-entropy, Hinge loss, etc.

- In many cases, DL algorithm **doesn't halt** at **local minimum**
  - Instead, 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.

# 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: (may need to cover in another lecture)
  - Basic algorithms: SGD, Momentum, Nesterov Momentum
  - Algorithms with adaptive learning rate: AdaGrad, RMSProp, Adam

Challenges in deep learning optimization: local minima, saddle points and complex loss surfaces

# Gradient/Jacobian revisited

- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - For DL models
    - $\theta$ includes **weight matrices**, **filters**, and **biases** which are trainable model parameters.
    - $P$ is the number of **trainable parameters** ($P$ could be $20 \times 10^6$).
    - $J(\theta)$ is the loss function over a training set.

- Gradient $\mathrm{g} = \nabla J(\theta)$ is the **first order derivative** and defined as
  - $\nabla J(\theta) = g = \begin{bmatrix} \frac{\partial J}{\partial \theta_1}(\theta) \\ \dots \dots \dots \\ \frac{\partial J}{\partial \theta_P}(\theta) \end{bmatrix}$

Let $f(x, y, z) = \log(\exp(x) + \exp(y) + \exp(z))$.
What is the gradient vector $\nabla f$?

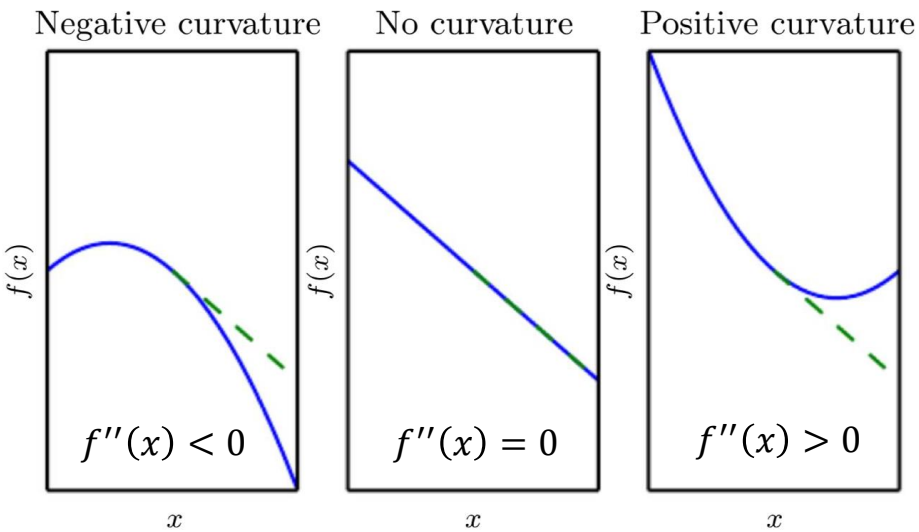
1. $f(x, y, z) = \log(u)$ where
   $u = \exp(x) + \exp(y) + \exp(z)$.
1. $f'_x = \frac{u'_x}{u} = \frac{\exp(x)}{\exp(x)+\exp(y)+\exp(z)}$
2. $f'_y = \frac{u'_y}{u} = \frac{\exp(y)}{\exp(x)+\exp(y)+\exp(z)}$.
3. $f'_z = \frac{u'_z}{u} = \frac{\exp(z)}{\exp(x)+\exp(y)+\exp(z)}$
4. $\nabla f = [f'_x, f'_y, f'_z] = softmax([x, y, z])$.

# Second order: Hessian matrix

- Gradient $\mathrm{g} = \nabla J(\theta)$ is the **first order derivative** and defined as
  - $\nabla J(\theta) = g = \begin{bmatrix} \frac{\partial J}{\partial \theta_1}(\theta) \\ \cdots \cdots \cdots \\ \frac{\partial J}{\partial \theta_P}(\theta) \end{bmatrix}$
- Hessian matrix $H(\theta)$ is the **second order derivative** $\nabla^2 J(\theta)$ and defined as
  - $\nabla^2 J(\theta) = H(\theta) =$

$$\begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1 \partial \theta_1}(\theta) & \cdots & \cdots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_j}(\theta) & \cdots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_P}(\theta) \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 J}{\partial \theta_i \partial \theta_1}(\theta) & \cdots & \cdots & \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}(\theta) & \cdots & \frac{\partial^2 J}{\partial \theta_i \partial \theta_P}(\theta) \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 J}{\partial \theta_P \partial \theta_1}(\theta) & \cdots & \cdots & \frac{\partial^2 J}{\partial \theta_P \partial \theta_j}(\theta) & \cdots & \frac{\partial^2 J}{\partial \theta_P \partial \theta_P}(\theta) \end{bmatrix}$$

- Hessian indicates the curvature!



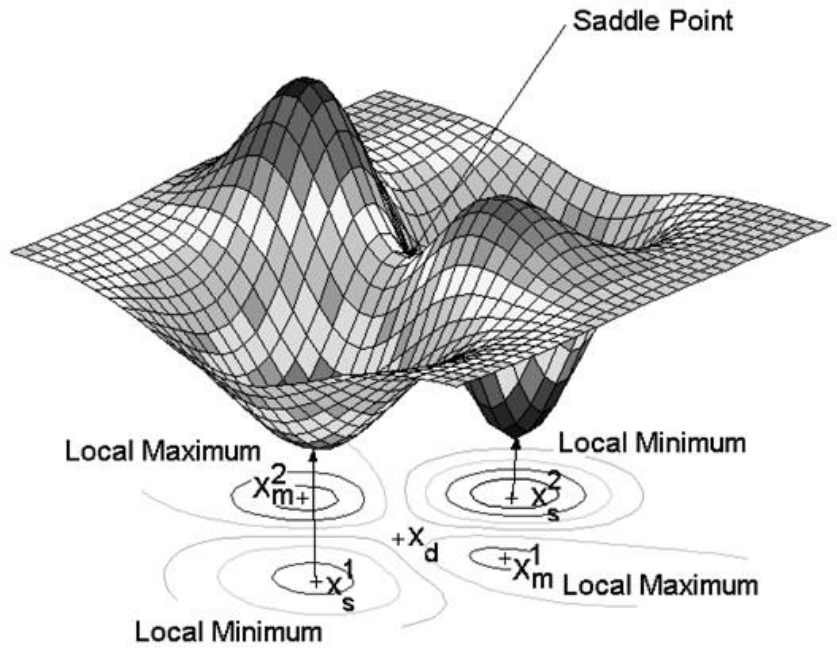

$f(x) = -x^2$
$f'(x) = -2x$
$f''(x) = -2 < 0$

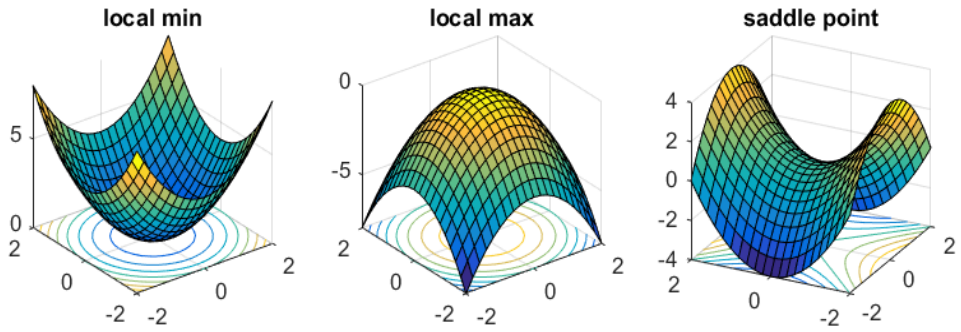
$f(x) = x$
$f'(x) = 1$
$f''(x) = 0$

$f(x) = x^2$
$f'(x) = 2x$
$f''(x) = 2 > 0$

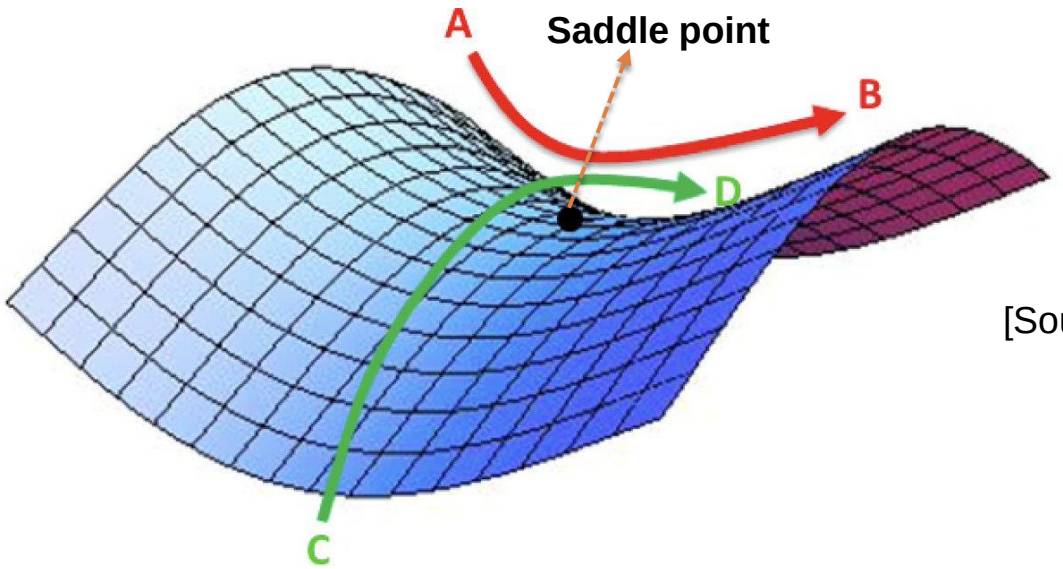
# Local minima-maxima and saddle point

- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - $\theta$ is said to be a **critical point** if $\nabla J(\theta) = \mathbf{0}$ (vector $\mathbf{0}$)
- Let us denote the **set of eigenvalues** of Hessian matrix $\nabla^2 J(\theta) = H(\theta)$ by
  - $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P$
- **Local minima**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \succ 0$ (positive semi-definite matrix)
  - $0 \leq \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P$
- **Local maxima**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \prec 0$ (negative sd)
  - $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P \leq 0$
- **Saddle point**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \prec 0$ (indefinite matrix)
  - $\lambda_1 \leq \lambda_2 \leq \cdots < 0 < \cdots \leq \lambda_P$



# More on saddle point




[Source: Internet]

$$f(x) = f(x_1, x_2) = x_1^2 - x_2^2$$

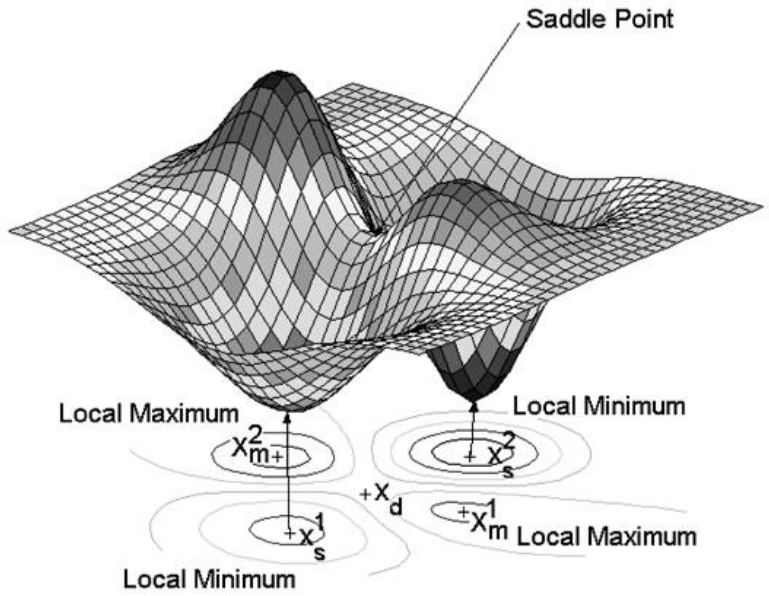
**Gradient** $g = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ -2x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ → a **critical point** $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

**Hessian matrix** is $H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$

**Two eigenvalues** $\lambda_1 = -2 < 0 < 2 = \lambda_2$ → $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ is a **saddle point**.

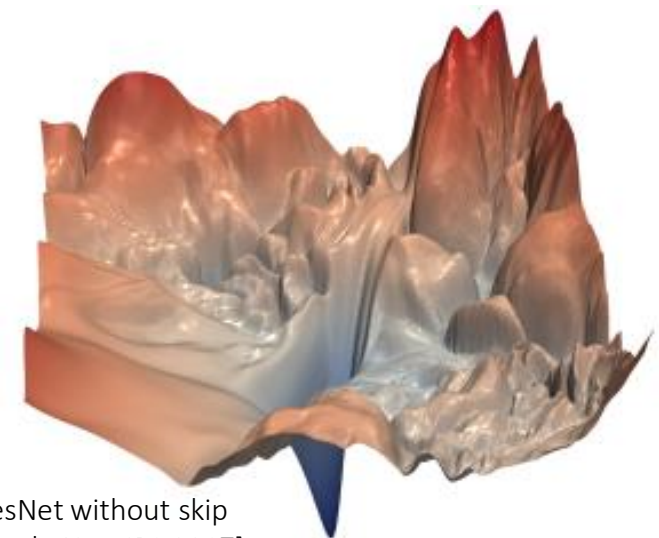
# Numbers of local minima vs saddle points

- We assume to **pick randomly** a training set $((x_1, y_1), \dots, (x_N, y_N) \sim \mathrm{p}_{data}(x, y))$
  - The Hessian matrix $H(\theta)$ is a random matrix with **random eigenvalues** $\lambda_1, \lambda_2, \dots, \lambda_P$
  - We assume that $\mathbb{P}(\lambda_1 \geq 0) = \mathbb{P}(\lambda_2 \geq 0) = \cdots = \mathbb{P}(\lambda_P \geq 0) = 0.5$
- Therefore, we have
  - $\mathbb{P}(minima) = \mathbb{P}(\lambda_1 \geq 0)\mathbb{P}(\lambda_2 \geq 0) \dots \mathbb{P}(\lambda_P \geq 0) = \frac{1}{2^P}$
  - $\mathbb{P}(maxima) = \mathbb{P}(\lambda_1 \leq 0)\mathbb{P}(\lambda_2 \leq 0) \dots \mathbb{P}(\lambda_P \leq 0) = \frac{1}{2^P}$
  - $\mathbb{P}(saddle\ point) = 1 - \mathbb{P}(minima) - \mathbb{P}(maxima) = \frac{2^P - 2}{2^P}$
- The ratio of **#local minima/maxima against #saddle points**
  - **#local-minima:#local-maxima:#saddle-point=**$1: 1: (2^P - 2)$
  - Number of **saddle points** is even **exponentially much more** than that of **local minima/maxima**

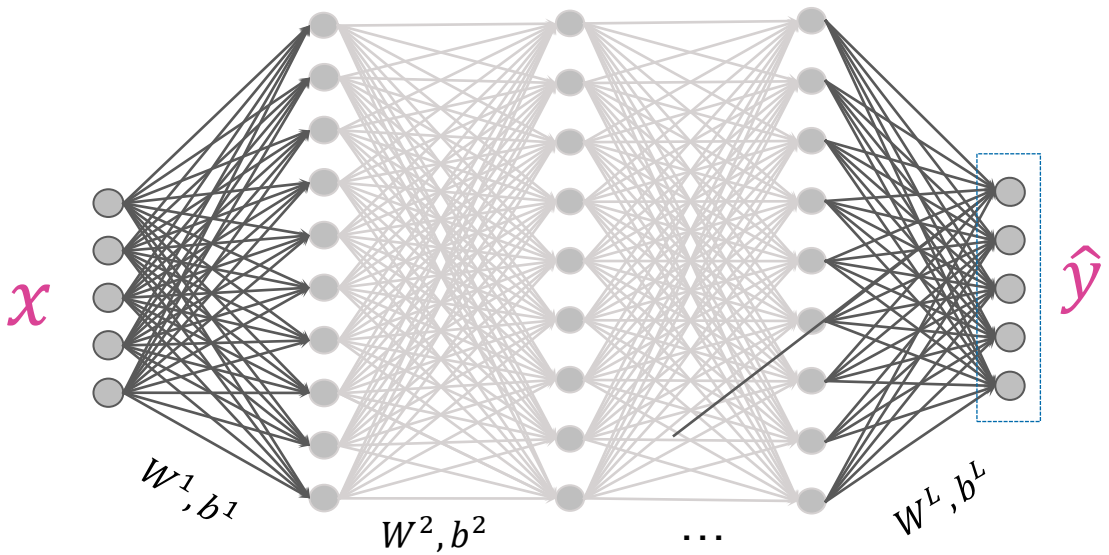


# The loss surface of DL optimization problem

- The optimization problem in deep learning:
  - $\min_{\theta} J(\theta) \coloneqq L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} l(f(x_i;\theta), y_i) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\{h_{y_i}^{L}(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^{L}(x_i)\}}$
- A very **complex** and **complicated** objective function
  - Highly **non-linear** and **non-convex** function
  - The **loss surface** is very **complex**
  - Many local minima points, but the number of saddle points is even **exponentially much more**
  - **Overparametrized problem because** the number of parameters $\boldsymbol{P = 20 \times 10^6}$ is greater than the training size $\boldsymbol{N = 1.2 \times 10^6}$.



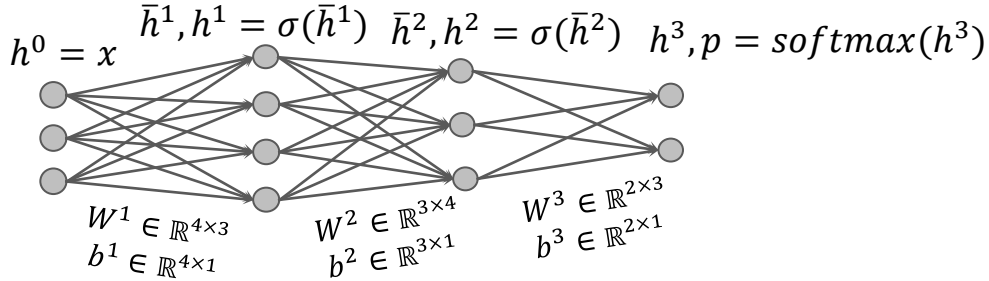
Loss surface of a ResNet without skip connection [Hao Li et al., NeurIPS 2017]



# Challenges in deep learning optimization: gradient vanishing, gradient exploding

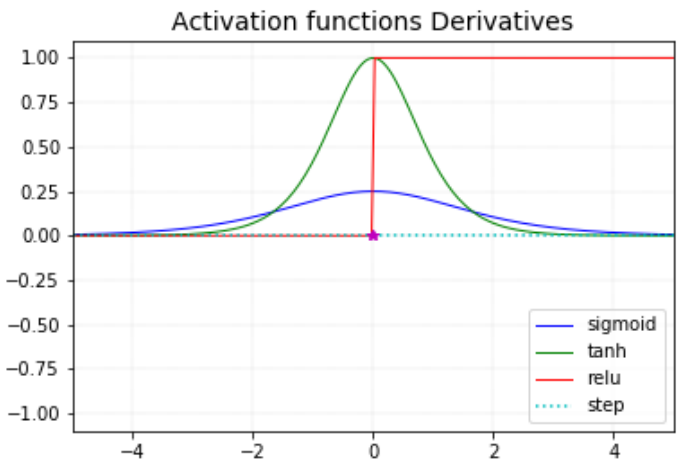
# Gradient vanishing

- **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




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

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



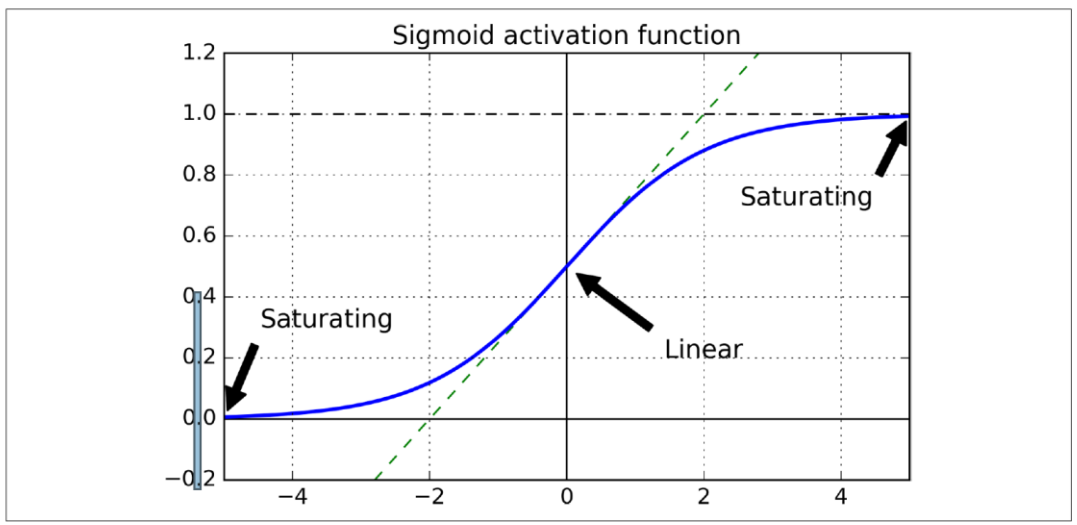
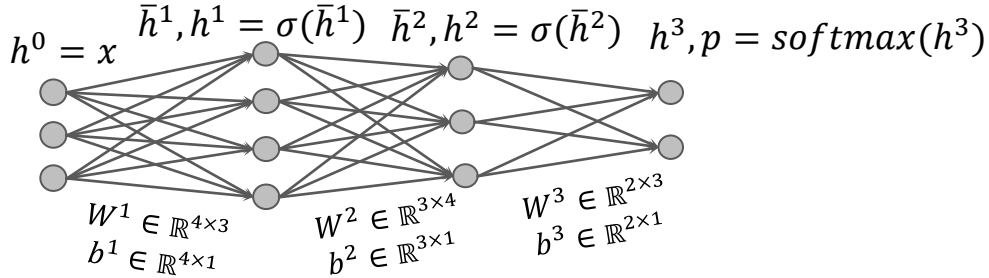




*Figure 11-1. Logistic activation function saturation*

[HandsOn, ch11]

# 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!



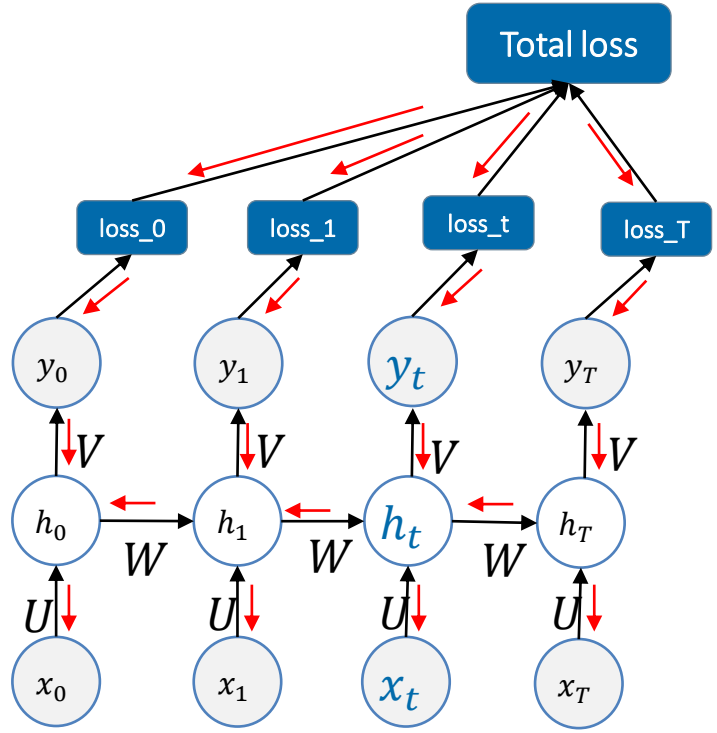

$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3}\cdot\frac{\partial h^3}{\partial h^2}\cdot\frac{\partial h^2}{\partial \bar{h}^2}\cdot\frac{\partial \bar{h}^2}{\partial h^1}\cdot\frac{\partial h^1}{\partial \bar{h}^1}\cdot\frac{\partial \bar{h}^1}{\partial W^1}$$

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

# 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**




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

$W$ $W$

**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 vanishing/exploding

## More rigorous explanation

- One source of the problem comes from **saturation of activation function**: design of effective activation function is the key
- Another source: in deep models, we've commonly seen a **matrix $W$ get multiplied several times**, proportional to its depth.
- Let $W = U\text{diag}([\lambda_1, \dots, \lambda_i, \dots, \lambda_d])U^T$ where $\lambda_i$ are **eigenvalues**
  - **SVD decomposition** with $U^T U = UU^T = I$ (identity matrix).
- Consider **multiplying $W$ repeatedly over $m$ times**, then:

$$W^m = U\text{diag}([\lambda_1, \dots, \lambda_d])U^T U\text{diag}([\lambda_1, \dots, \lambda_d])U^T \dots U\text{diag}([\lambda_1, \dots, \lambda_d])U^T U\text{diag}([\lambda_1, \dots, \lambda_d])U^T$$
$$= U\text{diag}([\lambda_1^m, \dots, \lambda_i^m, \dots, \lambda_d^m])U^T$$

- Let's examine **the effect of $\lambda_i^m$** as **$m$ gets larger**:
  - If $|\lambda_i| < 1$ then $\lambda_i^m \to 0$ as $m \to +\infty$ : **vanishing**
  - If $|\lambda_i| > 1$ then $\lambda_i^m \to \infty$ as $m \to +\infty$ : **exploding**

Not in assessment

# 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.

```
learning_rate= 0.01
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

All Keras optimizers accept `clipnorm` or `clipvalue` arguments:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
```

clip gradient to [-1,1]

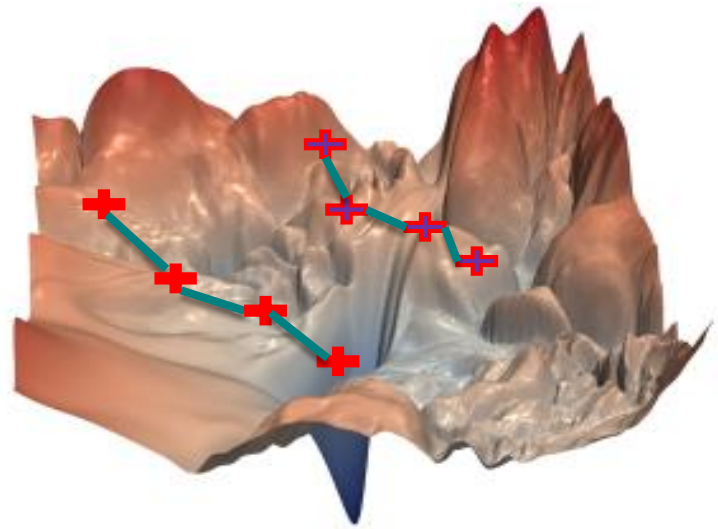
```
optimizer = keras.optimizers.SGD(clipnorm=1.0)
```

clip gradient norm to 1

# Weight initialization

# He and Xavier weight initialization

- **Initialisation** in deep learning training is **crucial**:
  - Some optimisation algorithms 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**.
  - We need the **gradient signal to flow well** in both directions; we **don't want** the signal to **die out**, nor do we want it 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

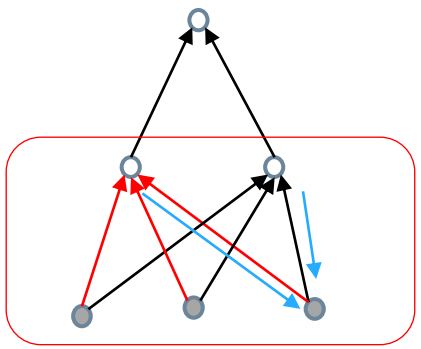
- **Xavier initialization**
  - Try to ensure the **variance of the outputs** of each layer equal to the **variance of its inputs**
  - Also need the gradients to have **equal variance** before and **after flowing through a layer** in the **reverse direction**
  - Good for **sigmoid** and **tanh** functions
  - 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**

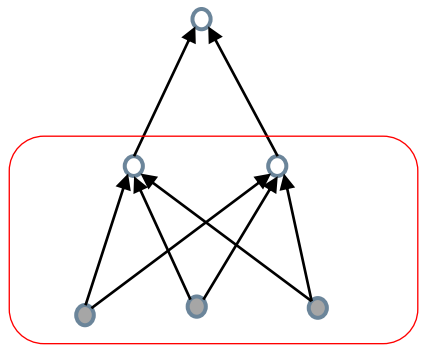
**Xavier Glorot** **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

**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 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

# Implement Xavier and He initialization

## Implement Xavier and He initialization

```
class my_Xavier_initializer(tf.keras.initializers.Initializer):
    def __init__(self, random_seed=42):
        np.random.seed(random_seed)
        tf.random.set_seed(random_seed)

    def __call__(self, shape, dtype=tf.float32):
        stddev = tf.sqrt(2. / (shape[0] + shape[1]))
        return tf.random.normal(shape, stddev=stddev, dtype=dtype)
```

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

```
import math
class my_He_initializer(tf.keras.initializers.Initializer):
    def __init__(self, random_seed=42, alpha= math.sqrt(2)):
        np.random.seed(random_seed)
        tf.random.set_seed(random_seed)
        self.alpha = alpha

    def __call__(self, shape, dtype=tf.float32):
        stddev = self.alpha*tf.sqrt(2. / (shape[0] + shape[1]))
        return tf.random.normal(shape, stddev=stddev, dtype=dtype)
```

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

## Plug them into the model training phase

```
def create_vgg_model(n_classes = 10):
    vgg_model = models.Sequential()
    vgg_model.add(layers.Conv2D(32, (3,3), padding='same', activation='elu', input_shape=(32,32,3)))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.Conv2D(32, (3,3), padding='same', activation='elu', kernel_initializer= my_He_initializer()))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.MaxPool2D(pool_size=(2,2)))  #downscale the image size by 2
    vgg_model.add(layers.Dropout(rate=0.25))  #deactivate 25% of neurons for each feed-forward
    vgg_model.add(layers.Conv2D(64, (3,3), padding='same', activation='elu', kernel_initializer= my_Xavier_initializer()))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.Conv2D(64, (3,3), padding='same', activation='elu', kernel_initializer= tf.keras.initializers.lecun_normal()))
```

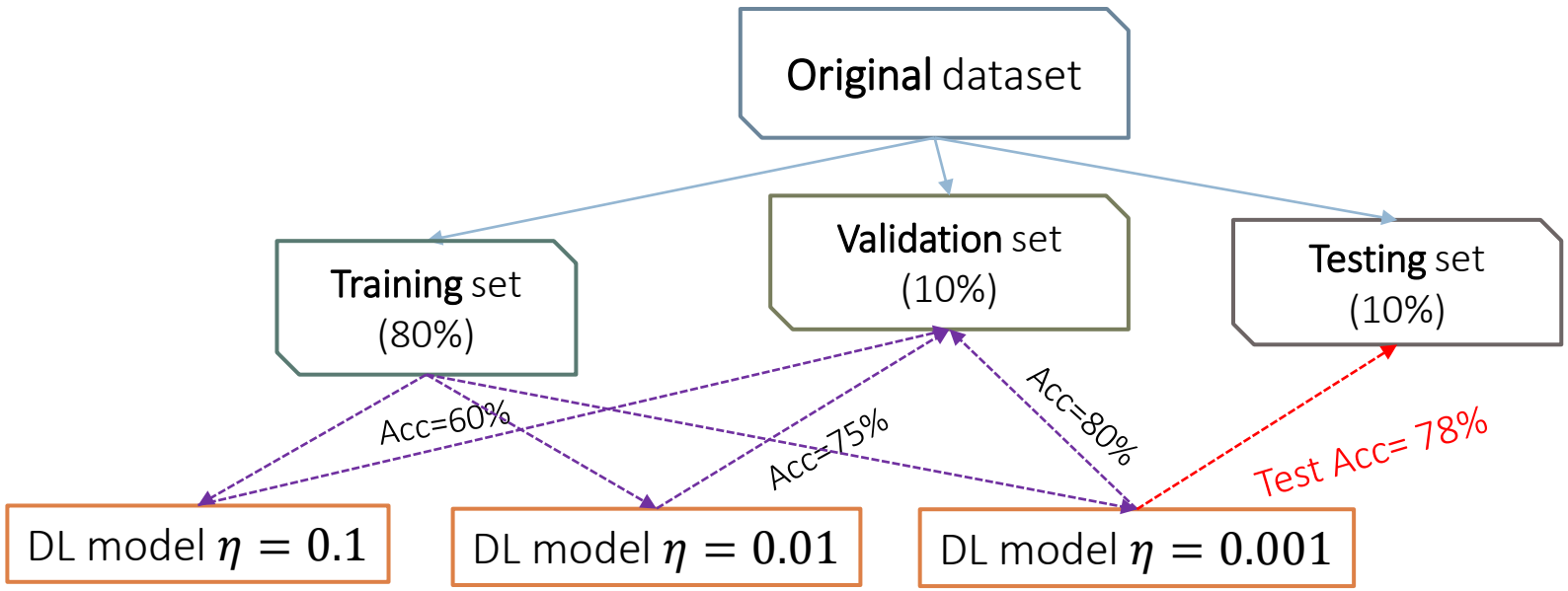
# Where do we go from here?

- 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

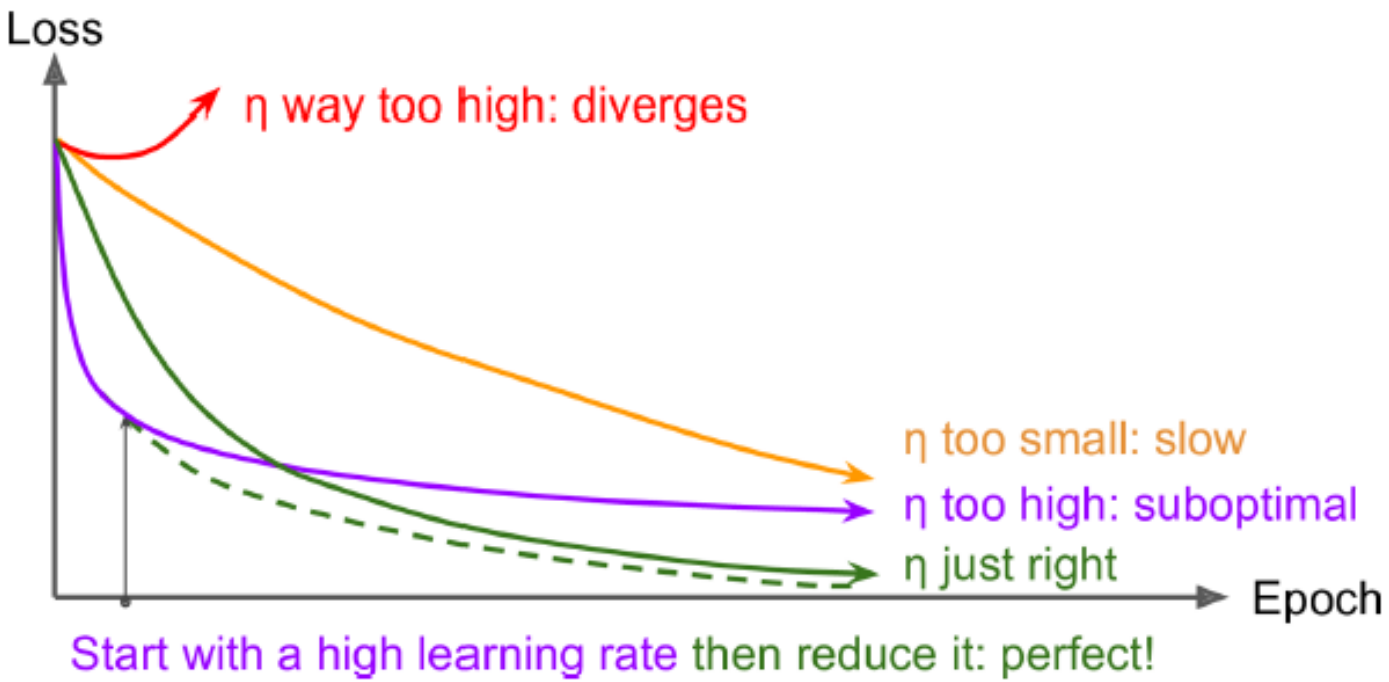
- We want to train our DL model on a training set such that the trained model can predict well unseen data in a separate testing set.




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

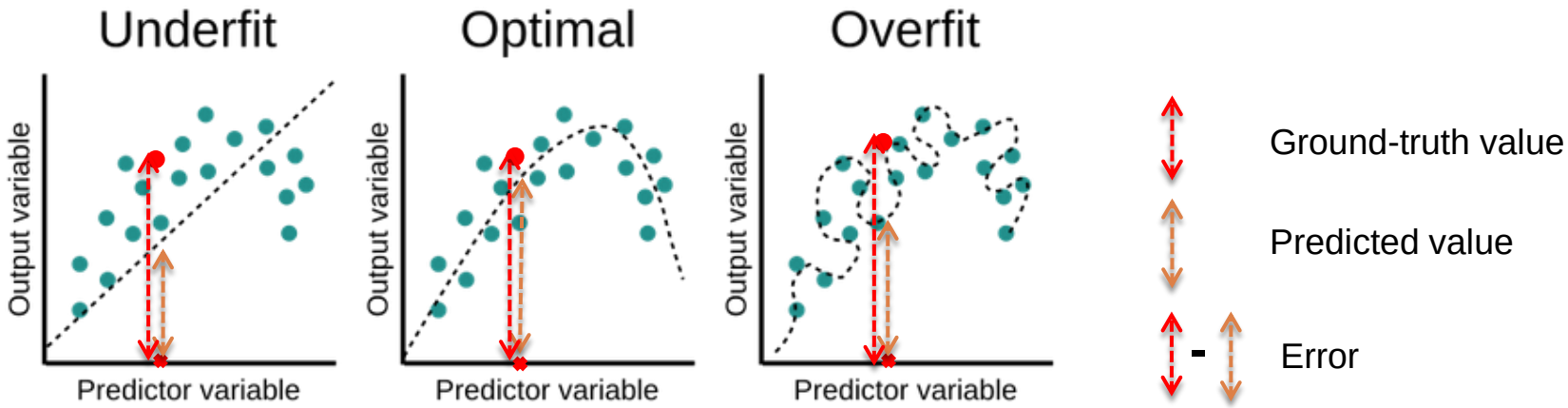
# Regularization and Overfitting

- Machine learning as an optimization process
  - **Data**: training data, testing data, validation data
  - **Model**: a **mathematical function** $f(x;\theta)$ that maps an input instance $x$ to outcome $y$
  - **Evaluation**: a **performance metric** to quantitively measure how well $f(x;\theta)$ is

- How does the machine learn from data? = optimizing its loss
  - Define a measure of loss via a **loss function**: $l(f(x;\theta),y)$
  - Compute its **loss over all training data** $J(\theta) = N^{-1}\sum_{i=1}^{N} l(f(x_i,\theta),y_i)$
  - Learning = **finding** $\theta^*$ 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**

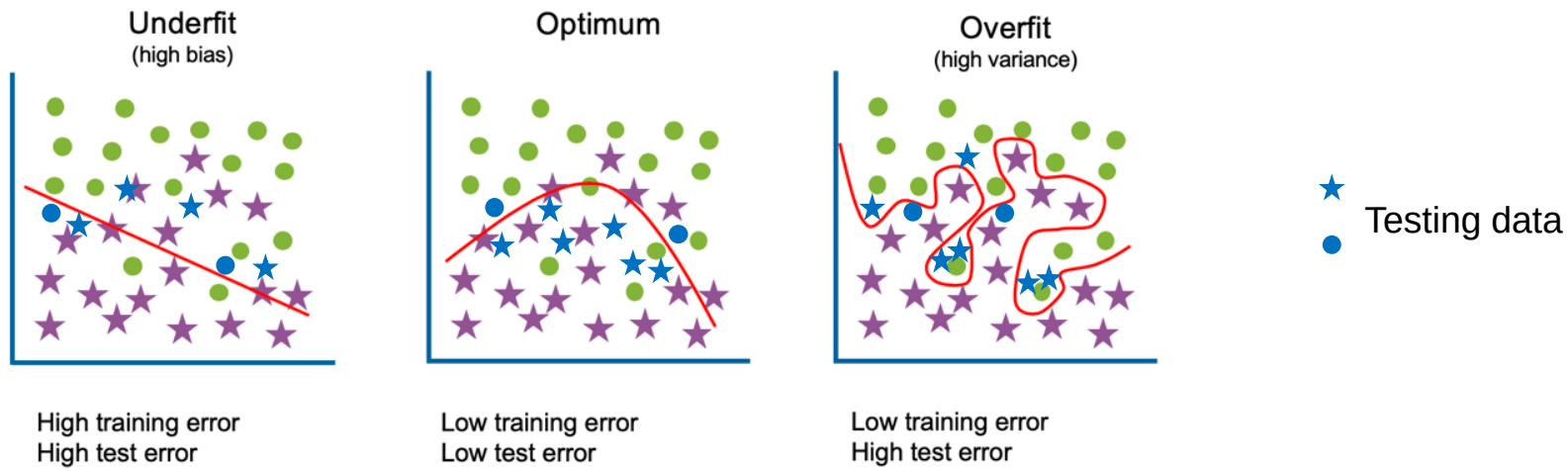
- **Underfitting**
  - Your model is **too simple** to characterise a training set
    - Use linear model to learn from non-linear data

- **Overfitting**
  - Your 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 hidden nodes** and **overtrained**.

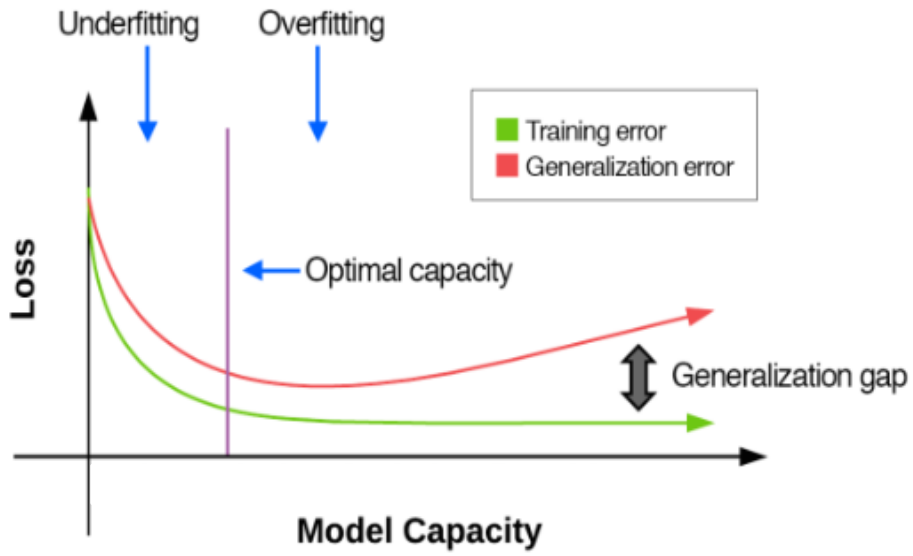
# Recipe for Overfitting

- **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**
  - If 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**.

| 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 | |

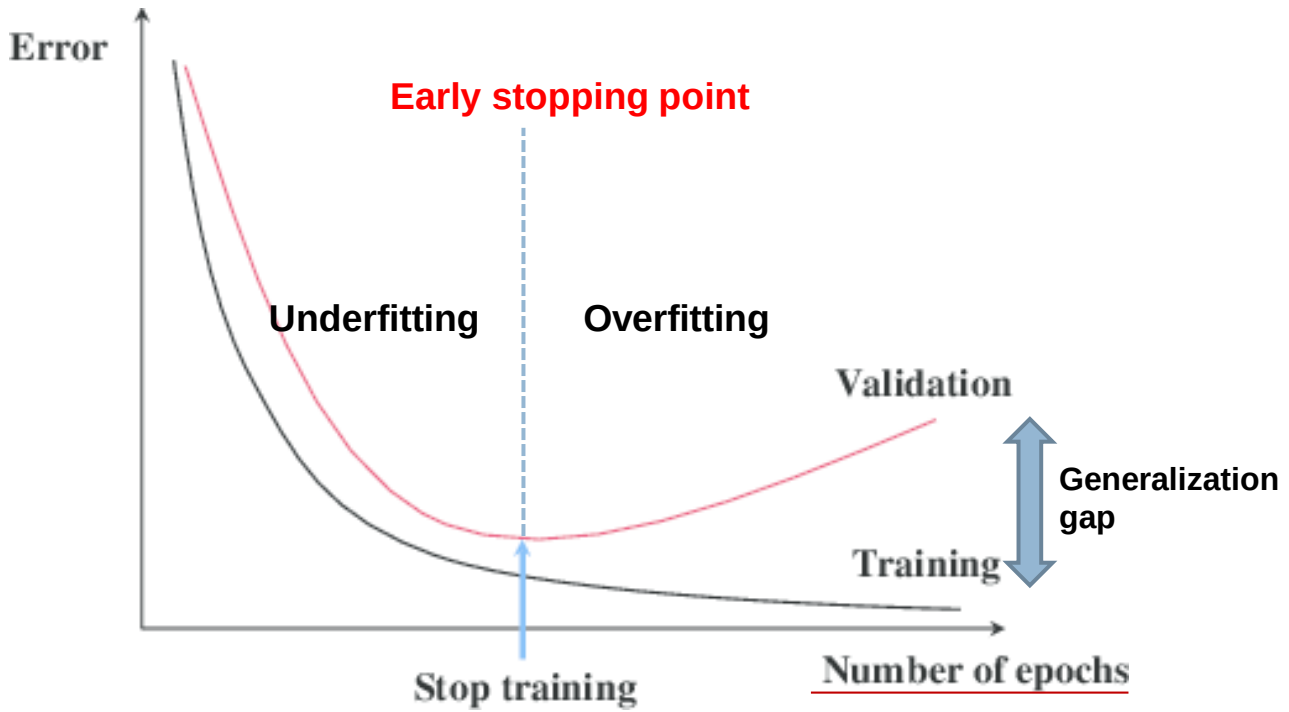
Table 11.1: The effect of various hyperparameters on model capacity.

We can also experiment with model capacity itself in parallel.




[DL, ch11]

# 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

# Early stopping with TF 2.x

```
from keras.callbacks import EarlyStopping
early_checkpoint = EarlyStopping(patience=3, monitor='val_loss', mode='min')
callbacks = [early_checkpoint]
opt = keras.optimizers.Adam(learning_rate=0.001)
vgg_model = create_vgg_model()
vgg_model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
vgg_model.fit(X_train, y_train, validation_data=(X_valid, y_valid), batch_size=32, epochs=20, callbacks=callbacks, verbose=1)
```

```
Train on 5000 samples, validate on 5000 samples
Epoch 1/20
5000/5000 [==============================] - 23s 5ms/sample - loss: 2.3459 - accuracy: 0.3404 - val_loss: 2.0277 - val_accuracy: 0.4108
Epoch 2/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 1.6240 - accuracy: 0.4560 - val_loss: 2.0812 - val_accuracy: 0.3932
Epoch 3/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 1.3864 - accuracy: 0.5238 - val_loss: 1.6854 - val_accuracy: 0.4714
Epoch 4/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 1.2247 - accuracy: 0.5692 - val_loss: 1.7588 - val_accuracy: 0.4670
Epoch 5/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 1.1217 - accuracy: 0.5948 - val_loss: 1.5521 - val_accuracy: 0.5194
Epoch 6/20
5000/5000 [==============================] - 22s 4ms/sample - loss: 1.0135 - accuracy: 0.6434 - val_loss: 2.3100 - val_accuracy: 0.4256
Epoch 7/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 0.9331 - accuracy: 0.6698 - val_loss: 1.6531 - val_accuracy: 0.4866
Epoch 8/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 0.8783 - accuracy: 0.6894 - val_loss: 1.5167 - val_accuracy: 0.5316
Epoch 9/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 0.8032 - accuracy: 0.7176 - val_loss: 1.6290 - val_accuracy: 0.5162
Epoch 10/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 0.7801 - accuracy: 0.7196 - val_loss: 1.4406 - val_accuracy: 0.5752
Epoch 11/20
5000/5000 [==============================] - 21s 4ms/sample - loss: 0.7303 - accuracy: 0.7402 - val_loss: 1.4379 - val_accuracy: 0.5750
Epoch 12/20
5000/5000 [==============================] - 23s 5ms/sample - loss: 0.6622 - accuracy: 0.7572 - val_loss: 1.4644 - val_accuracy: 0.5876
Epoch 13/20
5000/5000 [==============================] - 22s 4ms/sample - loss: 0.6012 - accuracy: 0.7904 - val_loss: 1.6171 - val_accuracy: 0.5528
Epoch 14/20
5000/5000 [==============================] - 22s 4ms/sample - loss: 0.5515 - accuracy: 0.8000 - val_loss: 1.6657 - val_accuracy: 0.5560
```

**We apply an early stopping if the validation loss is not decreasing three times in a row.**

# Checkpointing NNs model improvements

```
vgg_model = create_vgg_model()
opt = keras.optimizers.SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
vgg_model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Create a MiniVGG model

We create two checkpoints:

- val_loss_checkpoint to store the best model with **lowest validation loss**.
- val_acc_checkpoint to store the best model with **highest validation accuracy**.

```
val_loss_checkpoint = ModelCheckpoint(os.path.join(".\CheckingPoints", "best_val.hd5"), monitor="val_loss", mode="min", save_best_only=True, verbose=1)
val_acc_checkpoint = ModelCheckpoint(os.path.join(".\CheckingPoints", "best_acc.hd5"), monitor="val_accuracy", mode="max", save_best_only=True, verbose=1)
callbacks = [val_loss_checkpoint, val_acc_checkpoint]
```

```
print("[INFO] training network...")
H = vgg_model.fit(X_train, y_train, validation_data=(X_valid, y_valid), batch_size=32, epochs=20, callbacks=callbacks, verbose=1)
```

```
[INFO] training network...
Train on 5000 samples, validate on 5000 samples
Epoch 1/20
4992/5000 [============================>.] - ETA: 0s - loss: 2.4967 - accuracy: 0.3085
Epoch 00001: val_loss improved from inf to 1.96373, saving model to .\CheckingPoints\best_val.hd5
INFO:tensorflow:Assets written to: .\CheckingPoints\best_val.hd5\assets
```

Train the model with corresponding callbacks

```
from tensorflow.keras.models import load_model
best_acc_model = load_model('./CheckingPoints/best_acc.hd5')
best_acc_model.compile(optimizer="sgd", loss='sparse_categorical_crossentropy', metrics=['accuracy'])
best_acc_model.evaluate(X_test,y_test)
```

```
10000/10000 [==============================] - 7s 689us/sample - loss: 1.5039 - accuracy: 0.6004
[1.503869961166382, 0.6004]
```

Load and test the model with best validation accuracy

```
best_val_model = load_model('./CheckingPoints/best_val.hd5')
best_val_model.compile(optimizer="sgd", loss='sparse_categorical_crossentropy', metrics=['accuracy'])
best_val_model.evaluate(X_test,y_test)
```

```
10000/10000 [==============================] - 6s 649us/sample - loss: 1.4126 - accuracy: 0.5408
[1.4125793354034424, 0.5408]
```

Load and test the model with best validation loss

# Add Regularization

## Reduce Overfitting

- **Optimization problem**

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

- **L2 regularization**

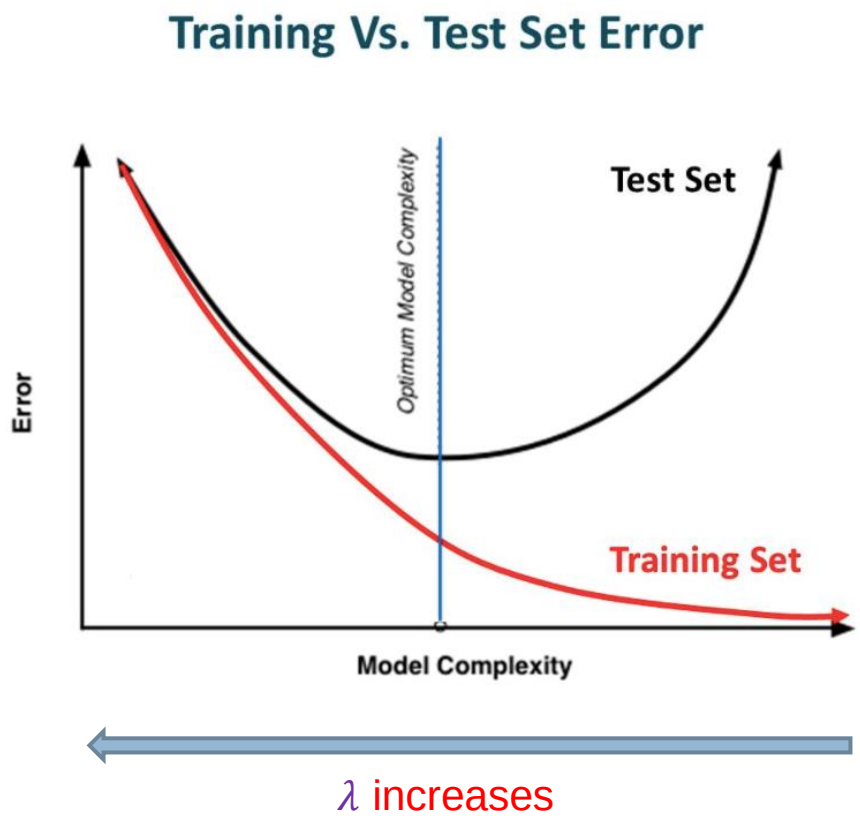
$$\Omega(\theta) = \lambda \sum_{k}\sum_{i,j}\left(W_{i,j}^{\mathbf{k}}\right)^2 = \lambda \sum_{k}\left\|\boldsymbol{W}^{\mathbf{k}}\right\|_F^2$$

  - $\lambda > 0$ 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 ($\boldsymbol{b}$)



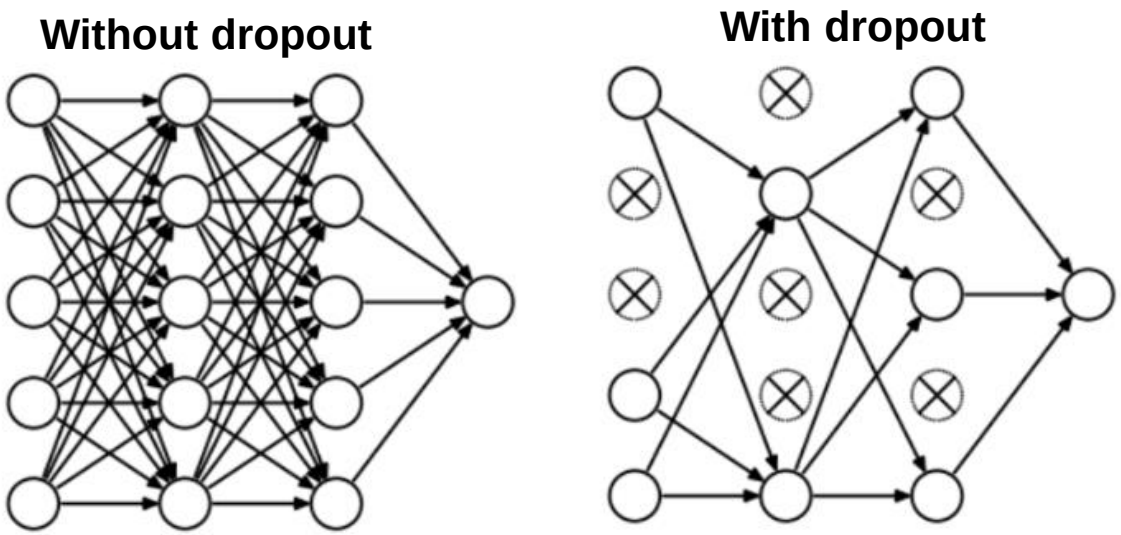
# Add regularization in TF 2.x

```
def create_vgg_model(n_classes = 10):
    vgg_model = models.Sequential()
    vgg_model.add(layers.Conv2D(32, (3,3), padding='same', activation='elu', input_shape=(32,32,3)))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.Conv2D(32, (3,3), padding='same', activation='elu', kernel_regularizer=tf.keras.regularizers.l2(0.01), kernel_initializer= my_He_initializer()))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.MaxPool2D(pool_size=(2,2)))  #downscale the image size by 2
    vgg_model.add(layers.Dropout(rate=0.25))  #deactivate 25% of neurons for each feed-forward
    vgg_model.add(layers.Conv2D(64, (3,3), padding='same', activation='elu', kernel_regularizer=tf.keras.regularizers.l2(0.01), kernel_initializer= my_Xavier_initializer()))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.Conv2D(64, (3,3), padding='same', activation='elu', kernel_regularizer=tf.keras.regularizers.l2(0.01), kernel_initializer= tf.keras.initializers.lecun_normal()))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.MaxPool2D(pool_size=(2,2)))  #downscale the image size by 2
    vgg_model.add(layers.Dropout(rate=0.25))  #deactivate 25% of neurons for each feed-forward
    vgg_model.add(layers.Flatten())
    vgg_model.add(layers.Dense(512, activation='elu', kernel_regularizer=tf.keras.regularizers.l2(0.01), kernel_initializer= 'lecun_normal'))
    vgg_model.add(layers.BatchNormalization(momentum=0.9))
    vgg_model.add(layers.Dropout(rate=0.5))
    vgg_model.add(layers.Dense(n_classes, activation='softmax')) #ten classes in Cifar10
    return vgg_model
```

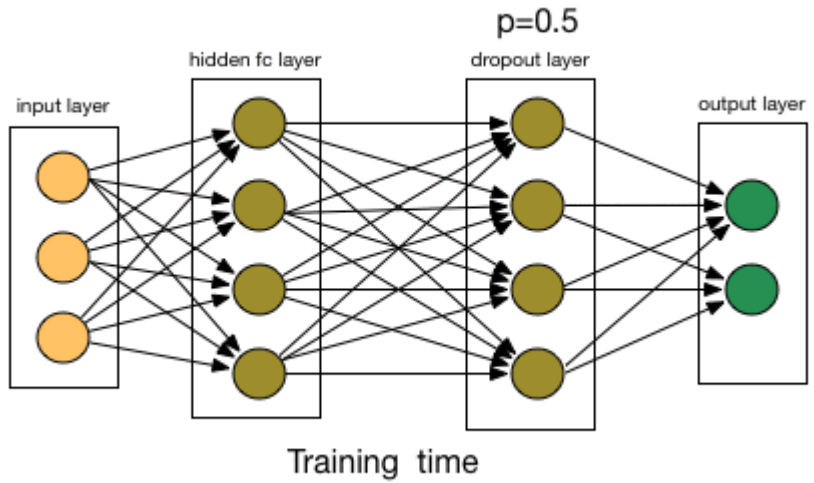
**We use L2 regularizer with the trade-off parameter $\lambda = 0.01$.**

# Dropout

## Reduce Overfitting




[Image source: Analytics Vidhya]




[Image source: chatbotslife]

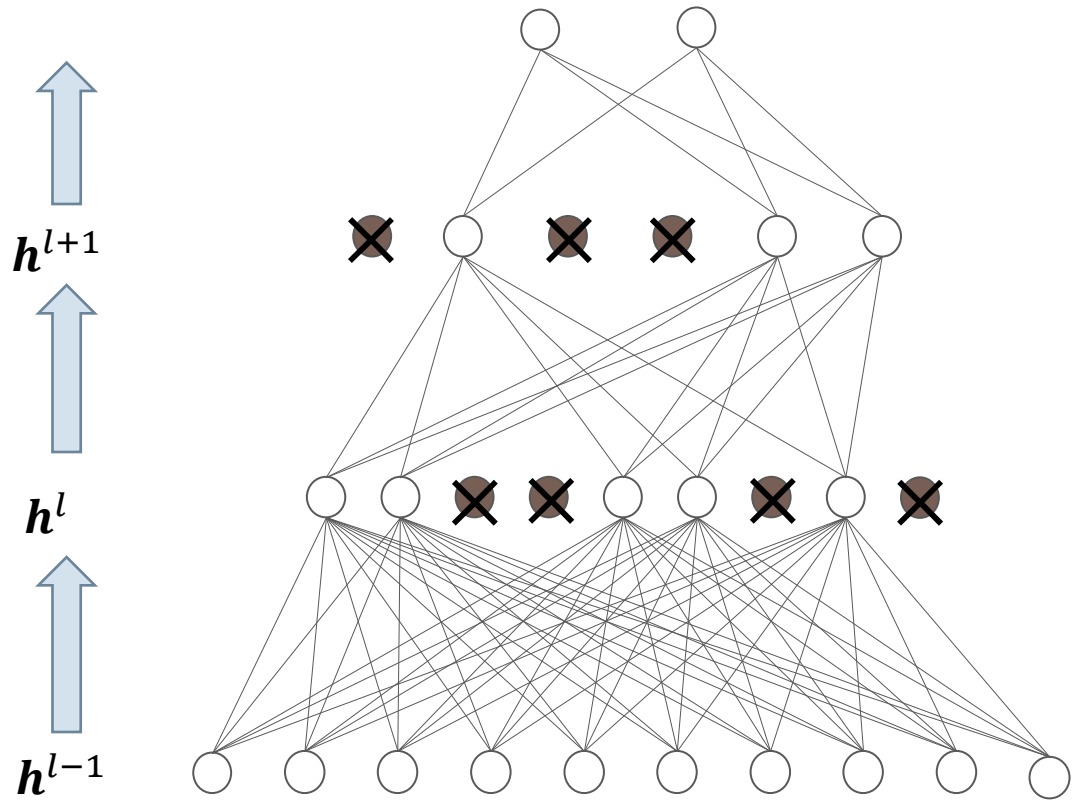
- This is a 'cheap' technique to reduce model capacity
  - Reduce overfitting
- In each iteration, at each layer, randomly choose some neurons and drop all connections from these neurons
  - dropout_rate= 1 – keep_prob

# Dropout

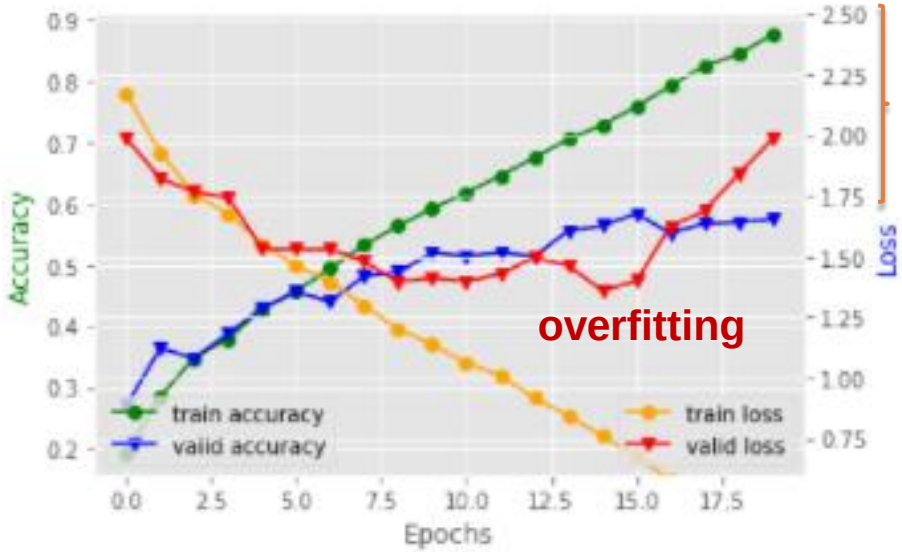
## Reduce Overfitting

- Computationally efficient
- Can be considered a bagged ensemble of an exponential number ($2^N$) of neural networks.
- An effective way to break the symmetry effect
- **Training**

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




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

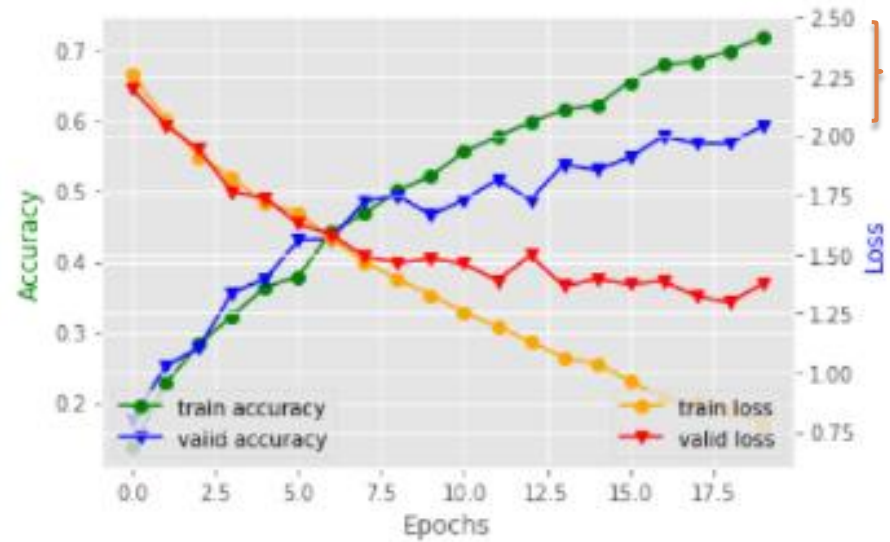



**87%**

Bigger gap

**57%**

**Without Dropout**




**71%**

Smaller gap

**59%**

**With Dropout**

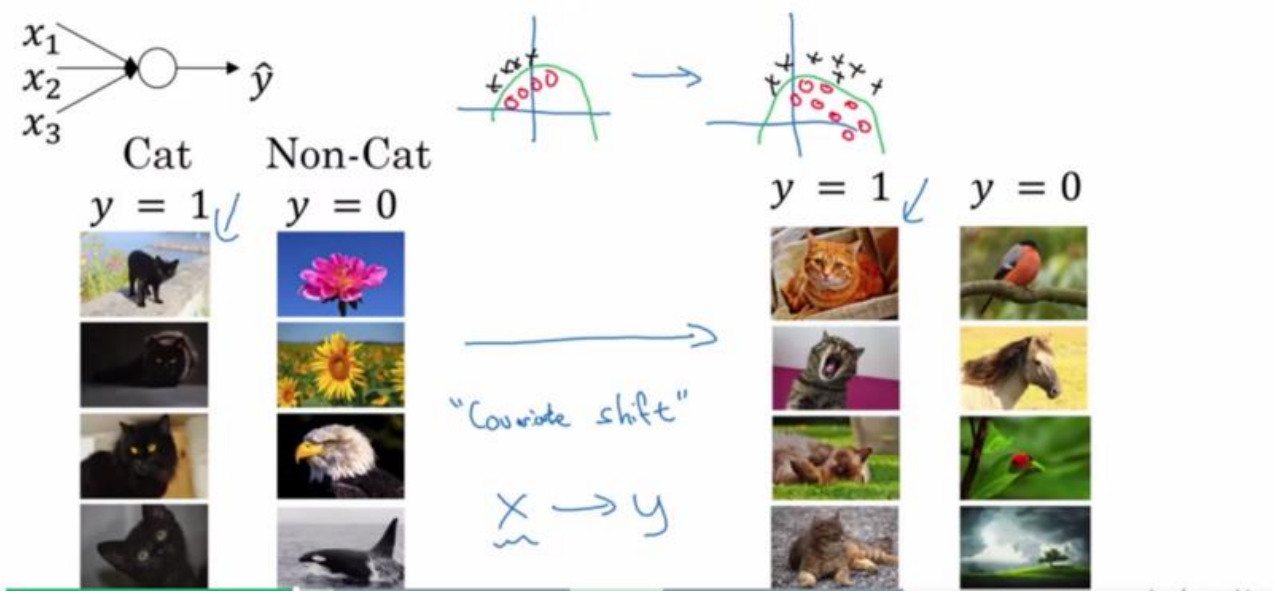
```
Epoch 1/20
63/63 [==============================] - 12s 191ms/step - loss: 2.1712 - accuracy: 0.1912 - val_loss: 1.9883 - val_accuracy: 0.2700
Epoch 2/20
63/63 [==============================] - 12s 186ms/step - loss: 1.9299 - accuracy: 0.2853 - val_loss: 1.8219 - val_accuracy: 0.3640
Epoch 3/20
63/63 [==============================] - 12s 188ms/step - loss: 1.7560 - accuracy: 0.3485 - val_loss: 1.7670 - val_accuracy: 0.3480
Epoch 4/20
63/63 [==============================] - 12s 186ms/step - loss: 1.6772 - accuracy: 0.3798 - val_loss: 1.7426 - val_accuracy: 0.3880
Epoch 5/20
63/63 [==============================] - 12s 185ms/step - loss: 1.5529 - accuracy: 0.4277 - val_loss: 1.5336 - val_accuracy: 0.4300
Epoch 6/20
63/63 [==============================] - 12s 193ms/step - loss: 1.4617 - accuracy: 0.4575 - val_loss: 1.5300 - val_accuracy: 0.4560
Epoch 7/20
63/63 [==============================] - 12s 193ms/step - loss: 1.3966 - accuracy: 0.4942 - val_loss: 1.5295 - val_accuracy: 0.4400
Epoch 8/20
63/63 [==============================] - 12s 192ms/step - loss: 1.2965 - accuracy: 0.5330 - val_loss: 1.4800 - val_accuracy: 0.4820
Epoch 9/20
63/63 [==============================] - 12s 193ms/step - loss: 1.2024 - accuracy: 0.5638 - val_loss: 1.3958 - val_accuracy: 0.4880
Epoch 10/20
63/63 [==============================] - 12s 192ms/step - loss: 1.1372 - accuracy: 0.5925 - val_loss: 1.4123 - val_accuracy: 0.5200
Epoch 11/20
63/63 [==============================] - 12s 194ms/step - loss: 1.0619 - accuracy: 0.6177 - val_loss: 1.3976 - val_accuracy: 0.5140
Epoch 12/20
63/63 [==============================] - 13s 199ms/step - loss: 1.0103 - accuracy: 0.6460 - val_loss: 1.4282 - val_accuracy: 0.5200
Epoch 13/20
63/63 [==============================] - 12s 193ms/step - loss: 0.9173 - accuracy: 0.6752 - val_loss: 1.4955 - val_accuracy: 0.5140
Epoch 14/20
63/63 [==============================] - 12s 197ms/step - loss: 0.8444 - accuracy: 0.7055 - val_loss: 1.4631 - val_accuracy: 0.5560
Epoch 15/20
63/63 [==============================] - 12s 192ms/step - loss: 0.7609 - accuracy: 0.7287 - val_loss: 1.3603 - val_accuracy: 0.5640
Epoch 16/20
63/63 [==============================] - 12s 192ms/step - loss: 0.6788 - accuracy: 0.7588 - val_loss: 1.3981 - val_accuracy: 0.5840
Epoch 17/20
63/63 [==============================] - 13s 199ms/step - loss: 0.5783 - accuracy: 0.7922 - val_loss: 1.6249 - val_accuracy: 0.5520
Epoch 18/20
63/63 [==============================] - 11s 181ms/step - loss: 0.5094 - accuracy: 0.8253 - val_loss: 1.6868 - val_accuracy: 0.5680
Epoch 19/20
63/63 [==============================] - 13s 200ms/step - loss: 0.4471 - accuracy: 0.8443 - val_loss: 1.8391 - val_accuracy: 0.5700
Epoch 20/20
63/63 [==============================] - 12s 193ms/step - loss: 0.3486 - accuracy: 0.8760 - val_loss: 1.9838 - val_accuracy: 0.5740
```

```
Epoch 1/20
63/63 [==============================] - 15s 241ms/step - loss: 2.2590 - accuracy: 0.1375 - val_loss: 2.1962 - val_accuracy: 0.1760
Epoch 2/20
63/63 [==============================] - 15s 237ms/step - loss: 2.0654 - accuracy: 0.2278 - val_loss: 2.0436 - val_accuracy: 0.2520
Epoch 3/20
63/63 [==============================] - 15s 237ms/step - loss: 1.9132 - accuracy: 0.2828 - val_loss: 1.9403 - val_accuracy: 0.2780
Epoch 4/20
63/63 [==============================] - 15s 240ms/step - loss: 1.8224 - accuracy: 0.3232 - val_loss: 1.7622 - val_accuracy: 0.3560
Epoch 5/20
63/63 [==============================] - 15s 242ms/step - loss: 1.7136 - accuracy: 0.3640 - val_loss: 1.7369 - val_accuracy: 0.3760
Epoch 6/20
63/63 [==============================] - 15s 238ms/step - loss: 1.6742 - accuracy: 0.3790 - val_loss: 1.6274 - val_accuracy: 0.4300
Epoch 7/20
63/63 [==============================] - 15s 239ms/step - loss: 1.5645 - accuracy: 0.4430 - val_loss: 1.5805 - val_accuracy: 0.4320
Epoch 8/20
63/63 [==============================] - 15s 239ms/step - loss: 1.4669 - accuracy: 0.4685 - val_loss: 1.4904 - val_accuracy: 0.4860
Epoch 9/20
63/63 [==============================] - 15s 238ms/step - loss: 1.3943 - accuracy: 0.5008 - val_loss: 1.4624 - val_accuracy: 0.4940
Epoch 10/20
63/63 [==============================] - 15s 240ms/step - loss: 1.3266 - accuracy: 0.5215 - val_loss: 1.4785 - val_accuracy: 0.4660
Epoch 11/20
63/63 [==============================] - 15s 239ms/step - loss: 1.2562 - accuracy: 0.5558 - val_loss: 1.4598 - val_accuracy: 0.4880
Epoch 12/20
63/63 [==============================] - 15s 239ms/step - loss: 1.1920 - accuracy: 0.5775 - val_loss: 1.3882 - val_accuracy: 0.5160
Epoch 13/20
63/63 [==============================] - 15s 239ms/step - loss: 1.1324 - accuracy: 0.5975 - val_loss: 1.4941 - val_accuracy: 0.4880
Epoch 14/20
63/63 [==============================] - 15s 239ms/step - loss: 1.0612 - accuracy: 0.6158 - val_loss: 1.3655 - val_accuracy: 0.5380
Epoch 15/20
63/63 [==============================] - 15s 239ms/step - loss: 1.0385 - accuracy: 0.6215 - val_loss: 1.3924 - val_accuracy: 0.5300
Epoch 16/20
63/63 [==============================] - 15s 244ms/step - loss: 0.9641 - accuracy: 0.6555 - val_loss: 1.3736 - val_accuracy: 0.5480
Epoch 17/20
63/63 [==============================] - 15s 245ms/step - loss: 0.8937 - accuracy: 0.6793 - val_loss: 1.3853 - val_accuracy: 0.5780
Epoch 18/20
63/63 [==============================] - 15s 240ms/step - loss: 0.8748 - accuracy: 0.6845 - val_loss: 1.3227 - val_accuracy: 0.5680
Epoch 19/20
63/63 [==============================] - 15s 241ms/step - loss: 0.8398 - accuracy: 0.6990 - val_loss: 1.2952 - val_accuracy: 0.5680
Epoch 20/20
63/63 [==============================] - 15s 240ms/step - loss: 0.7836 - accuracy: 0.7180 - val_loss: 1.3732 - val_accuracy: 0.5920
```

# Dropout

## Real-world Example

# Covariate shift problem




(Source: DL course Andrew Ng)

- Covariate shift problem:
  - The **distribution (nature)** of training data is **different** from that of testing data.

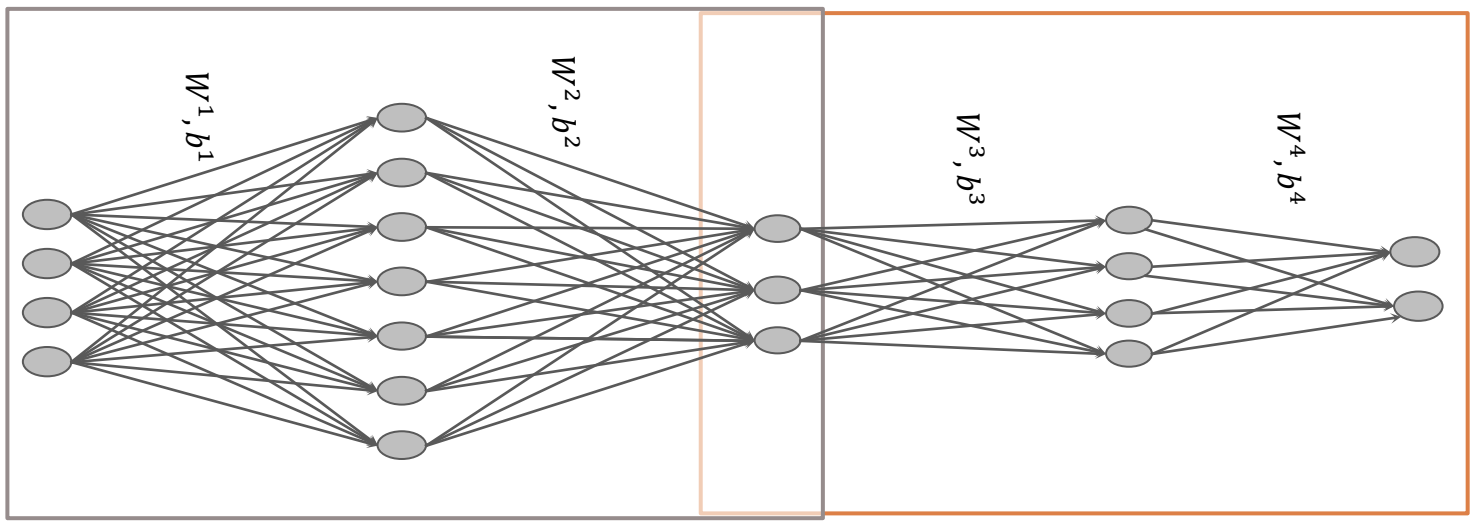
# Internal covariate shift problem

**Input batch**

$$\begin{bmatrix} x_1^1 & x_2^1 & x_3^1 & x_4^1 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 \\ x_1^3 & x_2^3 & x_3^3 & x_4^3 \\ x_1^4 & x_2^4 & x_3^4 & x_4^4 \end{bmatrix}$$

**Representation batch**

$$\begin{bmatrix} z_1^1 & z_2^1 & z_3^1 \\ z_1^2 & z_2^2 & z_3^2 \\ z_1^3 & z_2^3 & z_3^3 \\ z_1^4 & z_2^4 & z_3^4 \end{bmatrix}$$



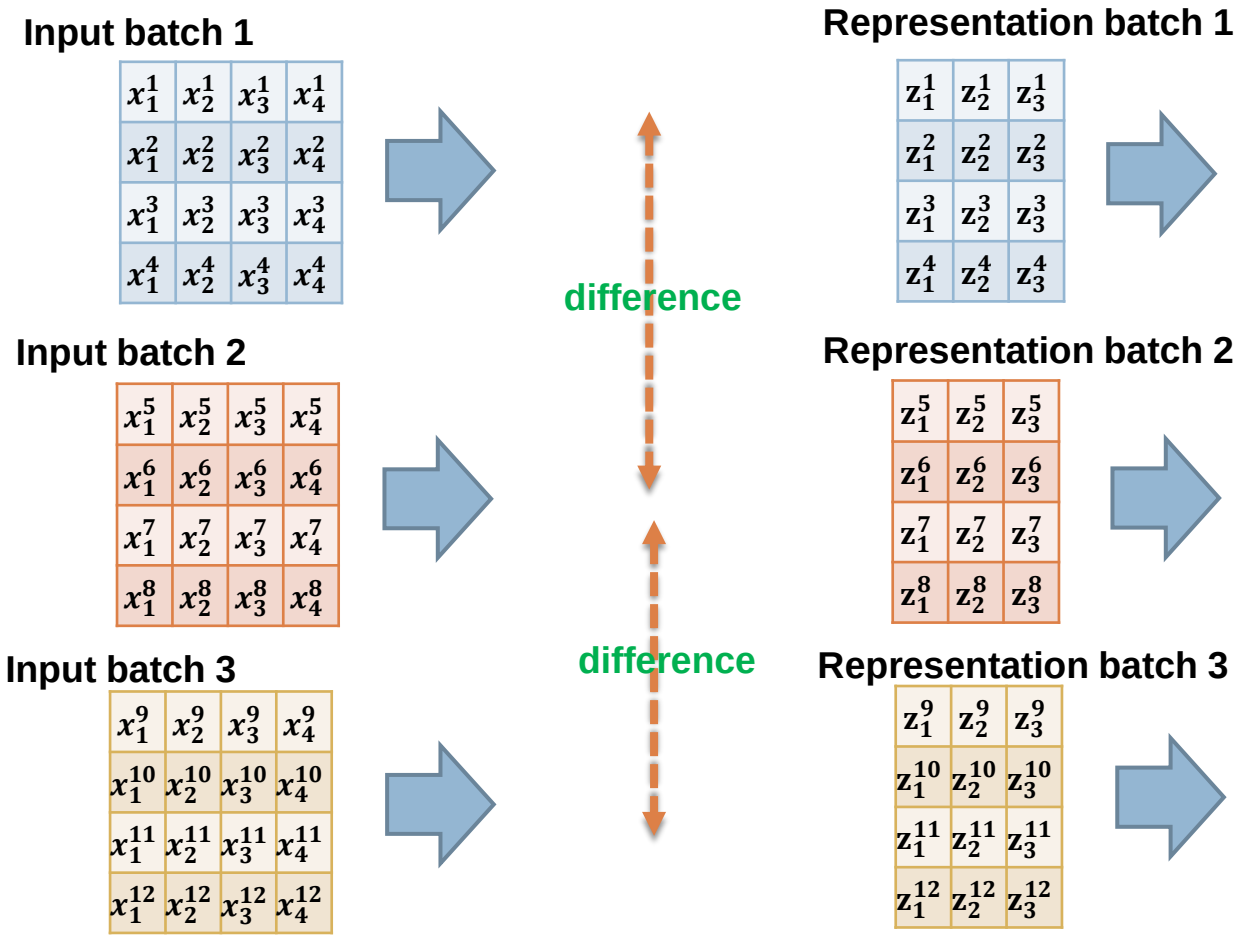

**Feature extractor** **Classifier**

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

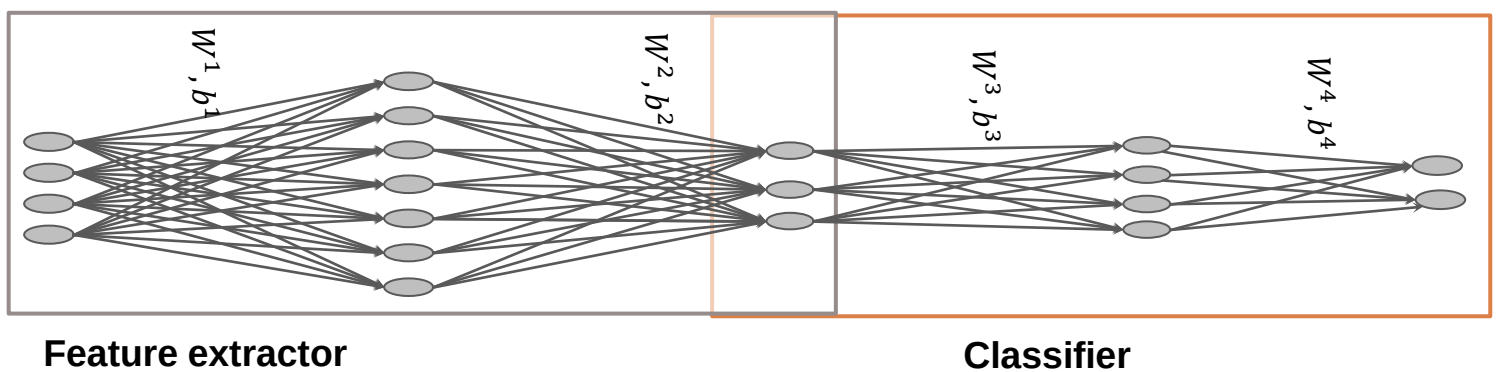
**Sergey Ioffe** SIOFFE@GOOGLE.COM
**Christian Szegedy** SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

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

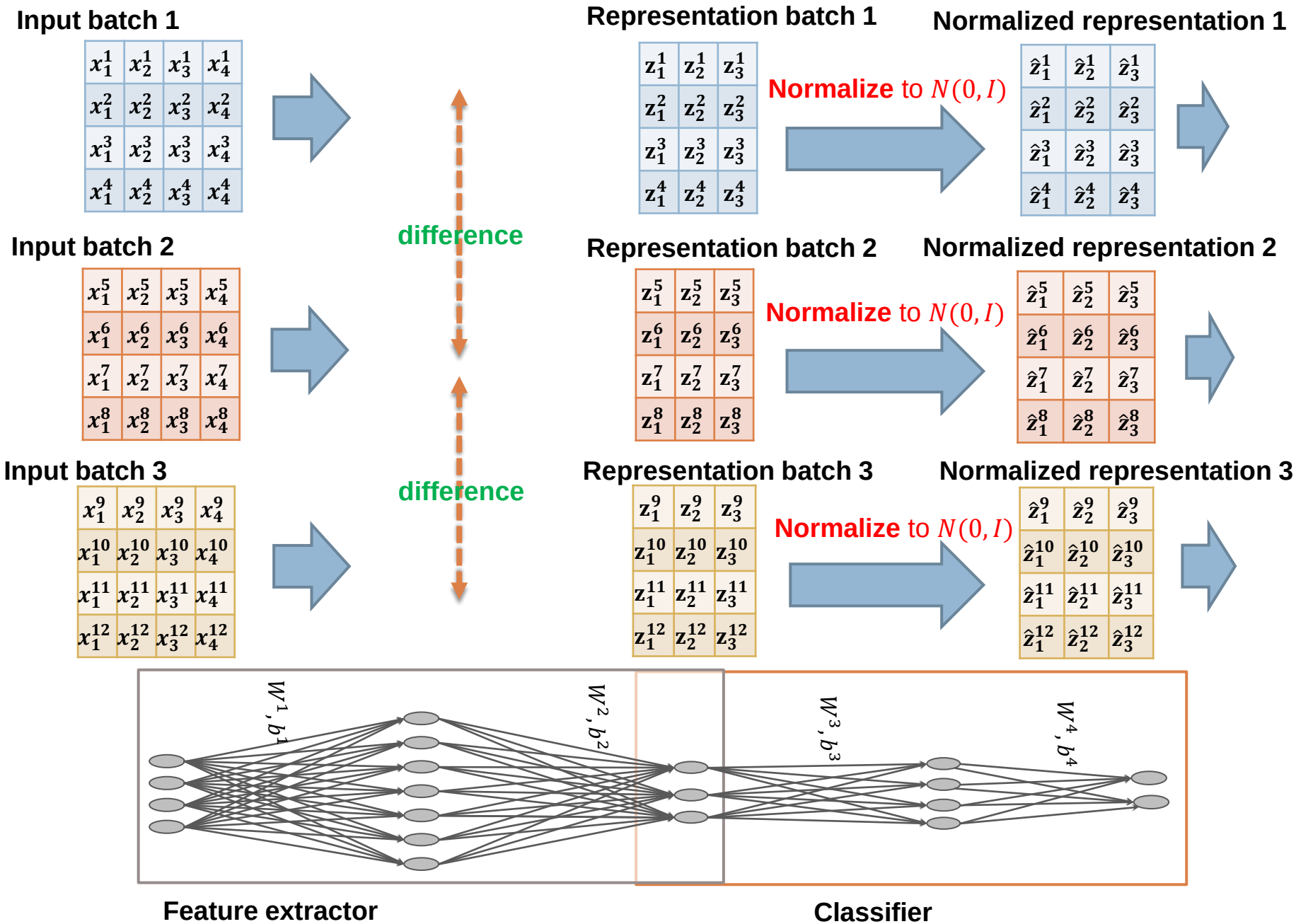
# Internal covariate shift problem



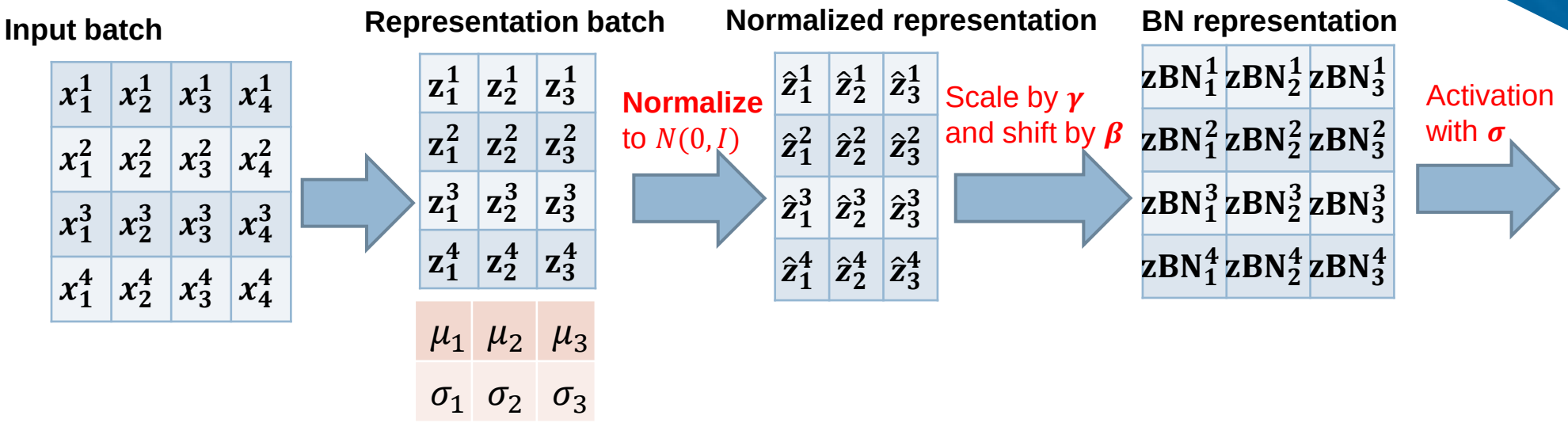

- Internal covariate shift problem:
  - Significant difference in statistics of input batches and also representation batches.
  - Statistical differences among mini-batches make the classifier harder to learn from data.

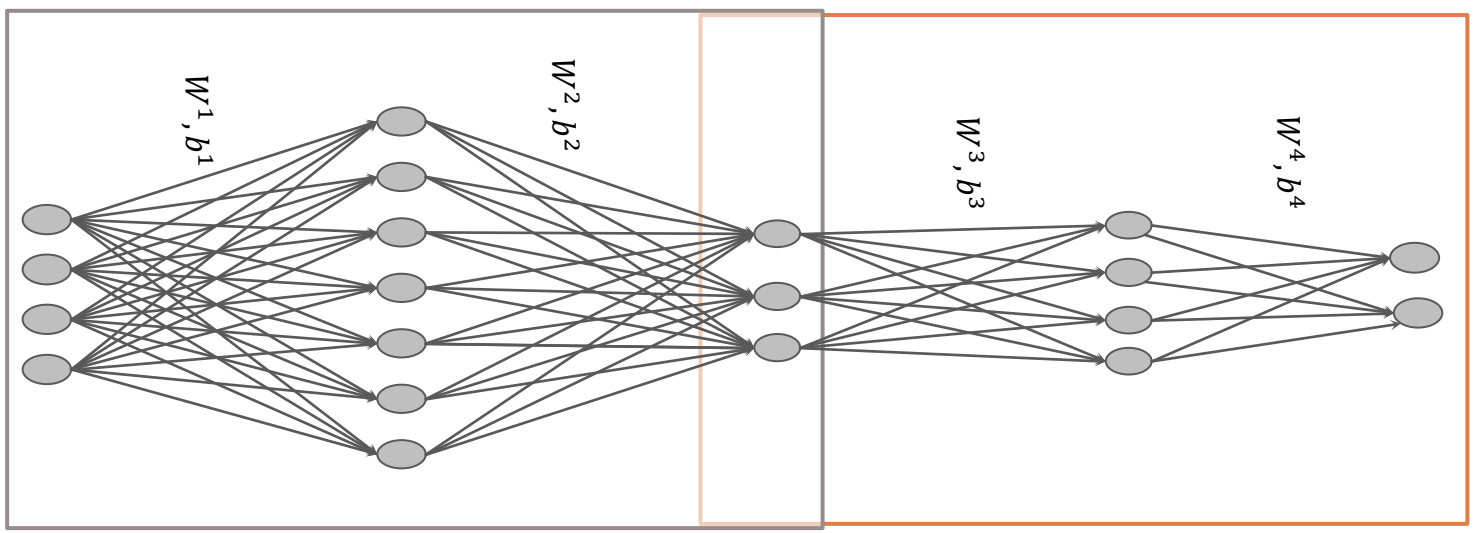


# Batch Normalization



# Batch Normalization

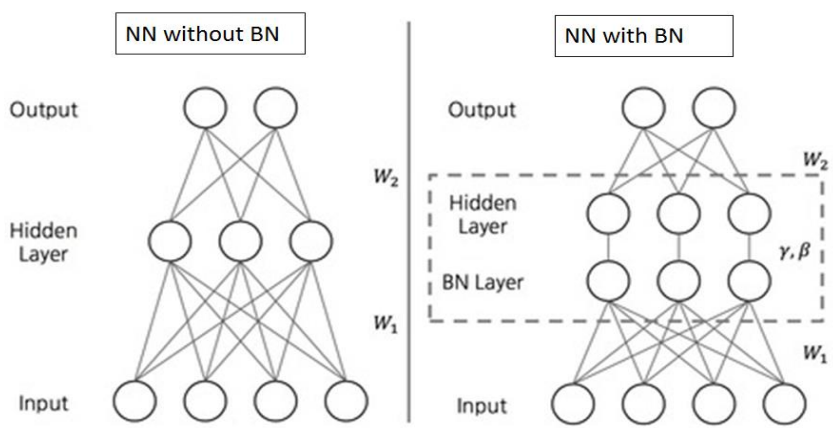


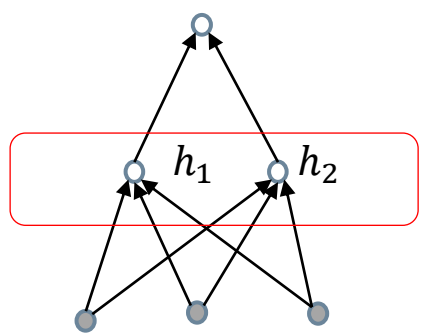



# Batch Normalization

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 $\epsilon$ 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

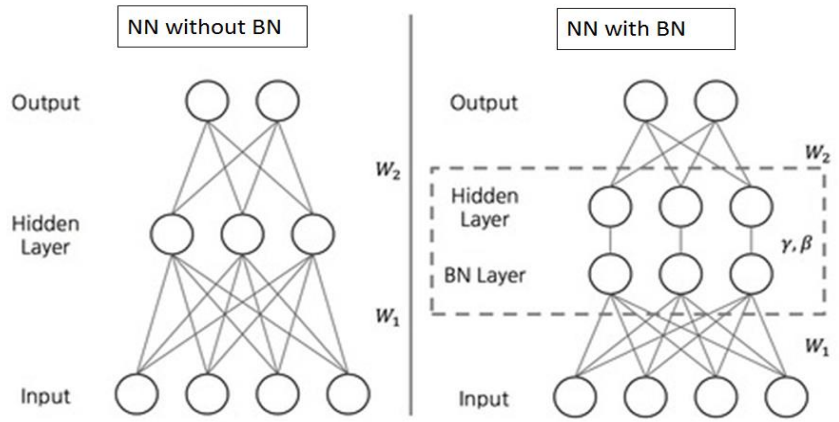
# Batch Normalization

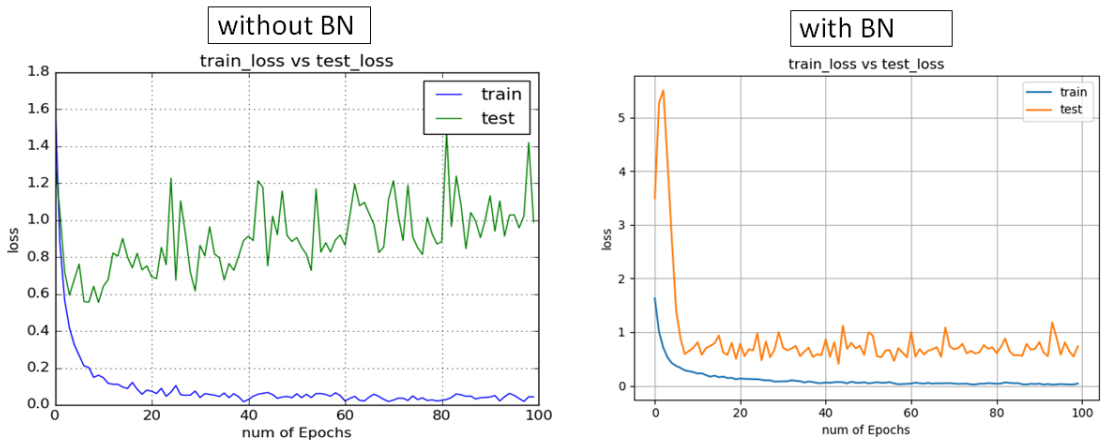
## Testing Phase

- 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 $\mu_B$ and $\sigma_B$ 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

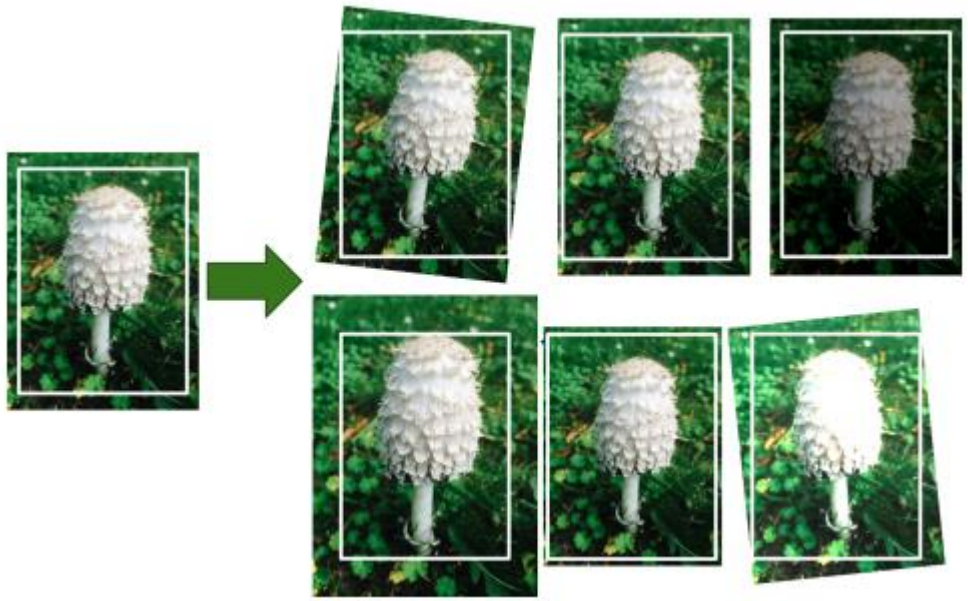



(Source: medium.com)

# Data augmentation

- Use **simple transformations** to augment data examples. Model 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**
  - Ideally, a human should **not** be able to tell which instances **were generated** and which ones **were not**.

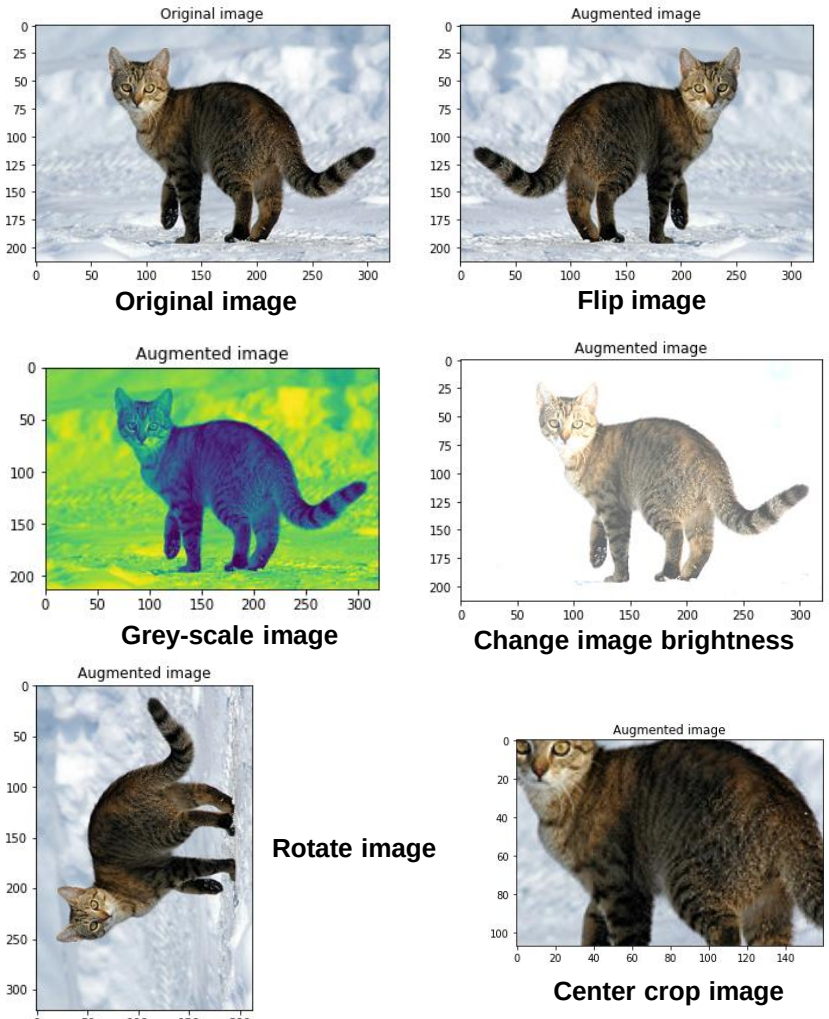


Slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set

[HandsOn, ch11]

# Data augmentation

- Use **simple transformations** to augment data examples. Model 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**
  - Ideally, a human should **not** be able to tell which instances **were generated** and which ones **were not**.

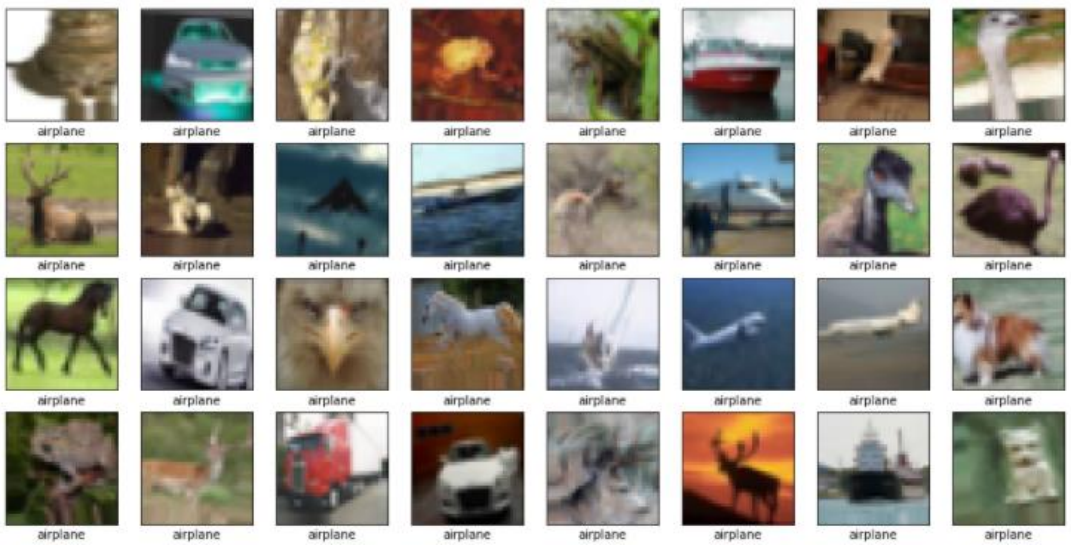


# Data augmentation in TF2.x

## Create ImageDataGenerator

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
datagen = ImageDataGenerator( horizontal_flip= True, width_shift_range=0.1, height_shift_range=0.1,
                              rotation_range=10, shear_range=0.1, zoom_range=0.1, fill_mode="nearest")
datagen.fit(X_train)

it = datagen.flow(X_train, y_train, shuffle=False, batch_size =32)
batch_images, batch_labels = next(it)
visualize_data(batch_images, batch_labels)
```




## Train with ImageDataGenerator

```
vgg = create_vgg()
opt = keras.optimizers.SGD(lr=0.001, decay=0.01 / 40, momentum=0.9, nesterov=True)
vgg.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
with_aug_history = vgg.fit(it, epochs=20, steps_per_epoch=len(X_train) / 32, validation_data= (X_valid, y_valid))
```

```
Epoch 1/20
313/312 [==============================] - 37s 119ms/step - loss: 2.4855 - accuracy: 0.2762 - val_loss: 1.6499 - val_accuracy: 0.4280
Epoch 2/20
313/312 [==============================] - 39s 126ms/step - loss: 1.9083 - accuracy: 0.3672 - val_loss: 1.4478 - val_accuracy: 0.4803
Epoch 3/20
313/312 [==============================] - 39s 125ms/step - loss: 1.7071 - accuracy: 0.4126 - val_loss: 1.4103 - val_accuracy: 0.4998
Epoch 4/20
313/312 [==============================] - 39s 125ms/step - loss: 1.5854 - accuracy: 0.4416 - val_loss: 1.3794 - val_accuracy: 0.5161
```

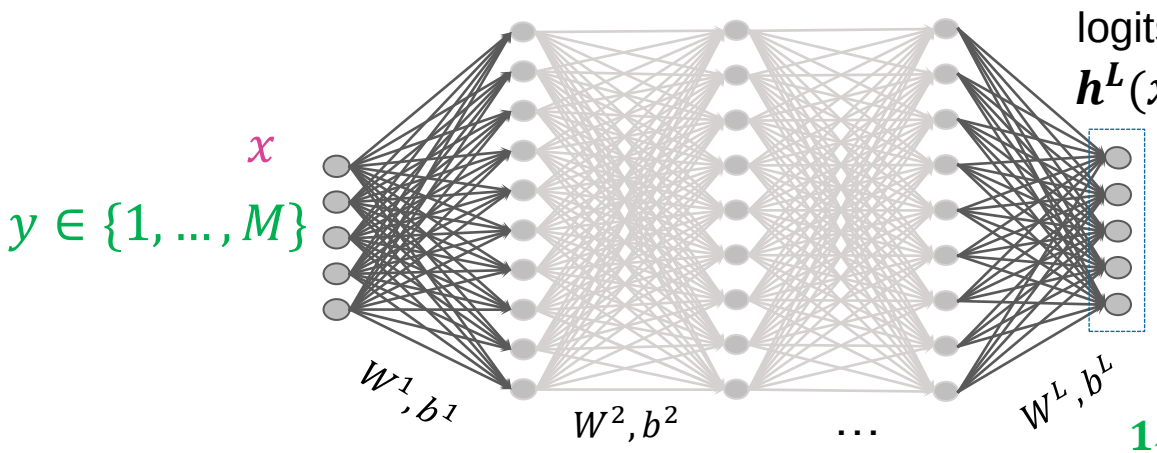
# Some additional regularization techniques

# 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




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**

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

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

$\boldsymbol{y}$-**th** position

**Label smoothing**

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

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

$\boldsymbol{y}$-**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** $\boldsymbol{p(x)}$ and the **smooth label**
  - $l(y, \hat{y}) = CE(\widetilde{\mathbf{1}}_y, \boldsymbol{p(x)})$ 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

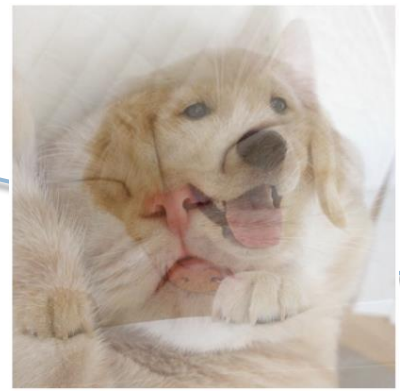
Published as a conference paper at ICLR 2018

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

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

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

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

basically adding 90% of the 2nd image onto the 1st image

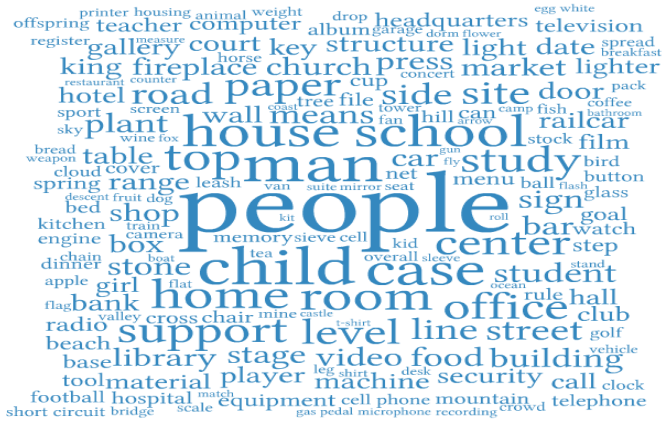
- for $(x_1, y_1)$, $(x_2, y_2)$ in zip(batch1, batch 2)
  1. $\lambda \sim \text{Beta}(\alpha, \alpha)$
  2. $\tilde{x} = \lambda \times x_1 + (1 - \lambda) \times x_2$
  3. $\tilde{y} = \lambda \times \mathbf{1}_{y_1} + (1 - \lambda) \times \mathbf{1}_{y_2}$
  4. Update optimizer to minimize $\boldsymbol{CE}(\tilde{\boldsymbol{y}},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

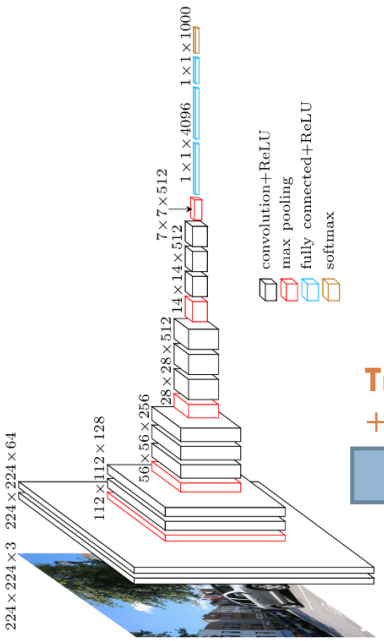
# Transfer learning and fine-tuning

# Transfer Learning

## Motivation

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



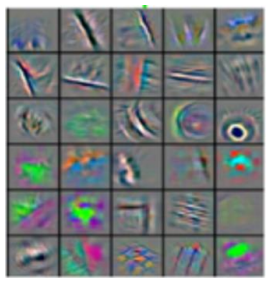

**Powerful pretrained model**
- VGG16, VGG19
- Inception V3
- Xception
- ResNet

**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

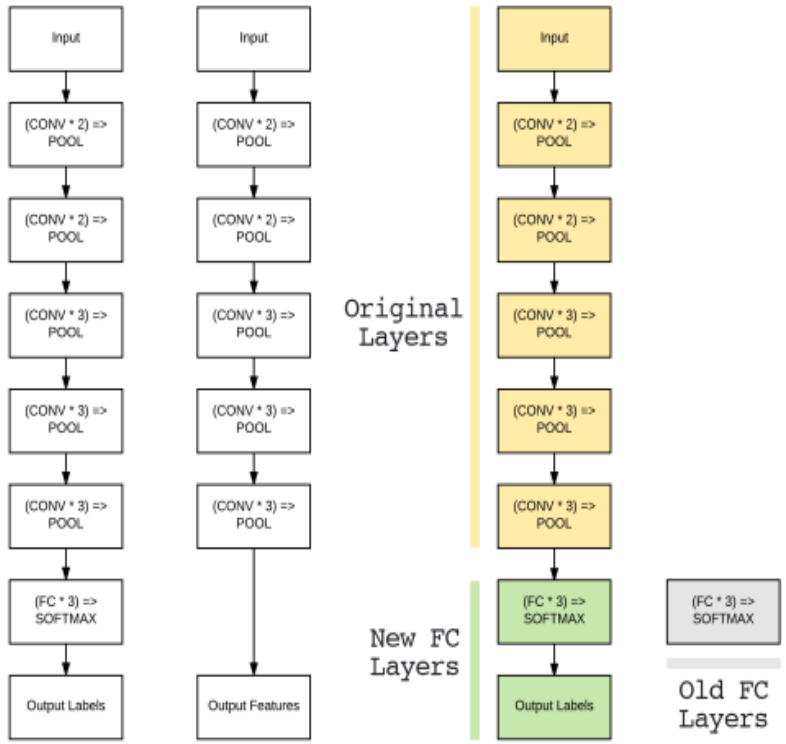
High-level filers

**Fits the target dataset**

# Transfer Learning

## How to Do That?

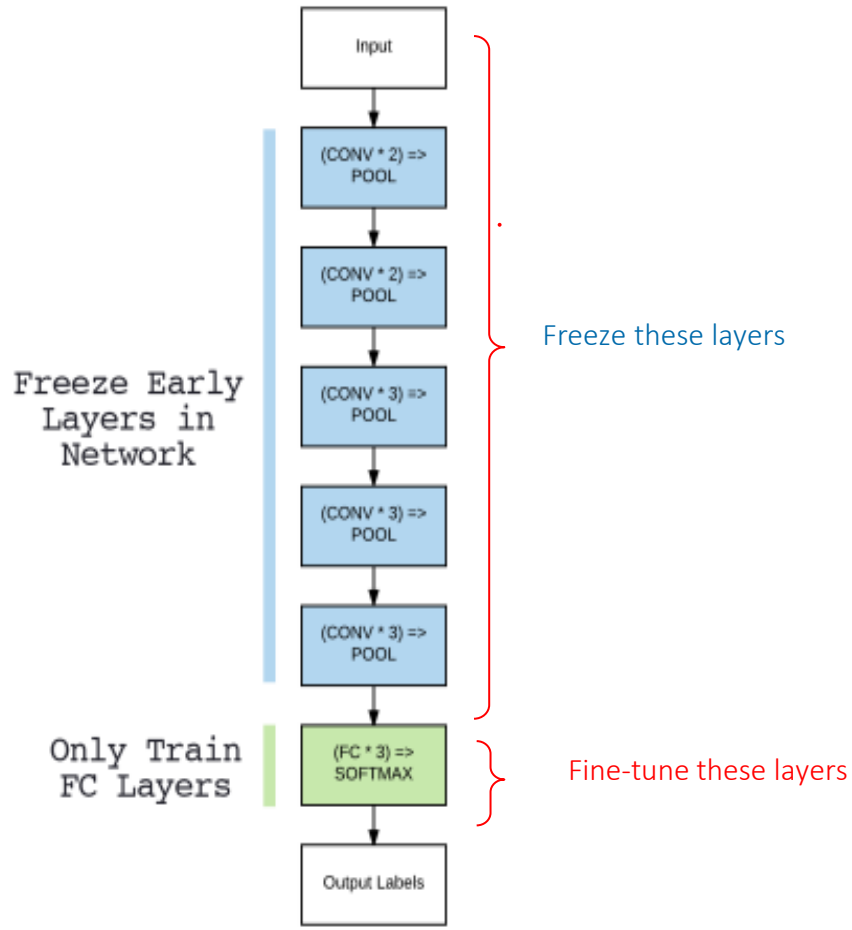
- **Remove** FC layers from the pretrained model
- **Replace** them with a brand-new FC head.
  - These new FC layers can then be fine-tuned to the specific dataset
  - The old FC layers are no longer used



# Transfer Learning

## How to Do That?

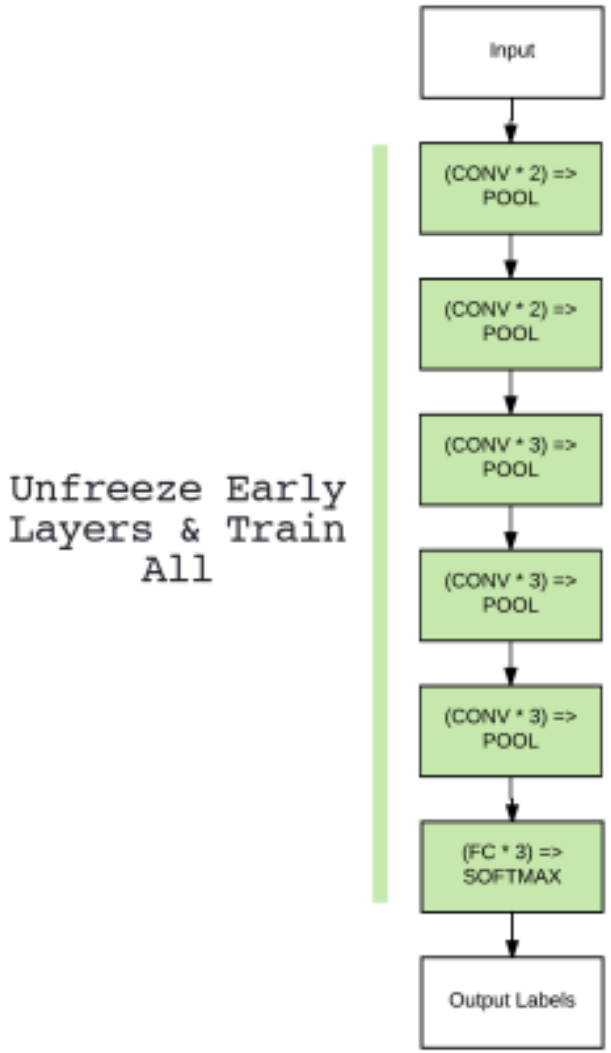
- **Freeze** all CONV layers in the network
  - Only allow the gradient to backpropagate through the FC layers
  - Doing this allows our network to **warm up**
- Training data is forward propagated through the network
  - However, the backpropagation is stopped after the FC layers
  - Allows these layers 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 TF 2.x

| Model | Size | Top-1 Accuracy | Top-5 Accuracy | Parameters | Depth |
|---|---|---|---|---|---|
| Xception | 88 MB | 0.790 | 0.945 | 22,910,480 | 126 |
| VGG16 | 528 MB | 0.713 | 0.901 | 138,357,544 | 23 |
| VGG19 | 549 MB | 0.713 | 0.900 | 143,667,240 | 26 |
| ResNet50 | 98 MB | 0.749 | 0.921 | 25,636,712 | - |
| ResNet101 | 171 MB | 0.764 | 0.928 | 44,707,176 | - |
| ResNet152 | 232 MB | 0.766 | 0.931 | 60,419,944 | - |
| ResNet50V2 | 98 MB | 0.760 | 0.930 | 25,613,800 | - |
| ResNet101V2 | 171 MB | 0.772 | 0.938 | 44,675,560 | - |
| ResNet152V2 | 232 MB | 0.780 | 0.942 | 60,380,648 | - |
| InceptionV3 | 92 MB | 0.779 | 0.937 | 23,851,784 | 159 |
| InceptionResNetV2 | 215 MB | 0.803 | 0.953 | 55,873,736 | 572 |
| MobileNet | 16 MB | 0.704 | 0.895 | 4,253,864 | 88 |
| MobileNetV2 | 14 MB | 0.713 | 0.901 | 3,538,984 | 88 |
| DenseNet121 | 33 MB | 0.750 | 0.923 | 8,062,504 | 121 |
| DenseNet169 | 57 MB | 0.762 | 0.932 | 14,307,880 | 169 |
| DenseNet201 | 80 MB | 0.773 | 0.936 | 20,242,984 | 201 |
| NASNetMobile | 23 MB | 0.744 | 0.919 | 5,326,716 | - |
| NASNetLarge | 343 MB | 0.825 | 0.960 | 88,949,818 | - |
| EfficientNetB0 | 29 MB | - | - | 5,330,571 | - |
| EfficientNetB1 | 31 MB | - | - | 7,856,239 | - |
| EfficientNetB2 | 36 MB | - | - | 9,177,569 | - |
| EfficientNetB3 | 48 MB | - | - | 12,320,535 | - |
| EfficientNetB4 | 75 MB | - | - | 19,466,823 | - |
| EfficientNetB5 | 118 MB | - | - | 30,562,527 | - |
| EfficientNetB6 | 166 MB | - | - | 43,265,143 | - |
| EfficientNetB7 | 256 MB | - | - | 66,658,687 | - |

# Transfer learning with TF 2.x

```
from tensorflow.keras.applications import ResNet50
```

```
print("Loading network...")
base_model = ResNet50(weights="imagenet", include_top=False, input_tensor = layers.Input(shape=(32,32,3)))
print("Showing layers...")
# loop over the layers in the network and display them to the console
for (i, layer) in enumerate(base_model.layers):
    print("{}\t{}".format(i, layer.__class__.__name__))
```

**Load ResNet50 to the base model.**

```
Loading network...
Downloading data from https://github.com/keras-team/keras-applications/releases/download/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94773248/94765736 [==============================] - 20s 0us/step
Showing layers...
0       InputLayer
1       ZeroPadding2D
2       Conv2D
3       BatchNormalization
4       Activation
5       ZeroPadding2D
6       MaxPooling2D
7       Conv2D
8       BatchNormalization
9       Activation
10      Conv2D
11      BatchNormalization
12      Activation
13      Conv2D
14      Conv2D
15      BatchNormalization
16      BatchNormalization
17      Add
18      Activation
19      Conv2D
20      BatchNormalization
155     Conv2D
156     BatchNormalization
157     Activation
158     Conv2D
159     BatchNormalization
160     Activation
161     Conv2D
162     BatchNormalization
163     Add
164     Activation
165     Conv2D
166     BatchNormalization
167     Activation
168     Conv2D
169     BatchNormalization
170     Activation
171     Conv2D
172     BatchNormalization
173     Add
174     Activation
```

# Transfer learning with TF 2.x

```
class FCHeadNet:
    @staticmethod
    def build(base_model, n_classes, D): # initialize the head model that will be placed on top of he base, then add a FC layer
        head_model = base_model.output
        head_model = layers.Flatten(name="flatten")(head_model)
        head_model = layers.Dense(D, activation="relu")(head_model)
        head_model = layers.Dropout(0.3)(head_model)
        # add a softmax layer
        head_model = layers.Dense(n_classes, activation="softmax")(head_model)
        return head_model
```

**Build two fully-connected layers on the top of the base model.**

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
aug = ImageDataGenerator(rotation_range=5, width_shift_range=0.1, height_shift_range=0.1, shear_range=0.1, zoom_range=0.1, horizontal_flip=True, fill_mode="nearest")
```

```
head_model = FCHeadNet.build(base_model, 10, 256)
model = models.Model(inputs=base_model.input, outputs=head_model)
```

**Create a real model for transfer learning.**

```
opt = keras.optimizers.RMSprop(lr=0.001)
model.compile(loss="sparse_categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
```

```
print("Training head...")
model.fit_generator(aug.flow(X_train, y_train, batch_size=64), validation_data=(X_valid, y_valid), epochs=5, steps_per_epoch=len(X_train) // 64, verbose=1)
```

```
Training head...
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train for 78 steps, validate on 5000 samples
Epoch 1/5
78/78 [==============================] - 185s 2s/step - loss: 2.3034 - accuracy: 0.2838 - val_loss: 155.1090 - val_accuracy: 0.0942
Epoch 2/5
78/78 [==============================] - 191s 2s/step - loss: 1.8698 - accuracy: 0.3618 - val_loss: 869.8668 - val_accuracy: 0.0982
Epoch 3/5
78/78 [==============================] - 192s 2s/step - loss: 1.6280 - accuracy: 0.4477 - val_loss: 427.1880 - val_accuracy: 0.1046
Epoch 4/5
78/78 [==============================] - 196s 3s/step - loss: 1.5509 - accuracy: 0.4858 - val_loss: 2.3219 - val_accuracy: 0.1132
Epoch 5/5
78/78 [==============================] - 189s 2s/step - loss: 1.4504 - accuracy: 0.5217 - val_loss: 2.4074 - val_accuracy: 0.1198
```

# Top-k Accuracy

# Top-k (ranked-k) accuracy

- Given a **positive integer number $k$** (i.e., $k = 1,5$ or $10$)
  - To compute **top-k (ranked-k) accuracy**, a given data example is counted as a correct prediction if the **ground-truth label** is in **the top-k predicted labels** (the labels with top-k highest predicted probability).
  - **Top-1 (ranked-1) accuracy** is identical to the **standard accuracy**.

| Class Label | Probability |
|---|---|
| Airplane | 0.0% |
| Automobile | 0.0% |
| Bird | 2.1% |
| Cat | 0.03% |
| Deer | 0.01% |
| Dog | 0.56% |
| Frog | **97.3%** |
| Horse | 0.0% |
| Ship | 0.0% |
| Truck | 0.0% |

**Ground-truth label**= Bird
**Top-3 prediction**: Correct/Incorrect?

**Correct**

| Class Label | Probability |
|---|---|
| Airplane | **1.1%** |
| Automobile | **38.7%** |
| Bird | 0.0% |
| Cat | 0.5% |
| Deer | 0.0% |
| Dog | 0.4% |
| Frog | 0.11% |
| Horse | **1.4%** |
| Ship | **2.39%** |
| Truck | **55.4%** |

**Ground-truth label**= Dog
**Top-3 prediction:** Correct/Incorrect?

**Incorrect**

# Top-k accuracy implementation in TF 2.x

```python
class TopkAcc(tf.keras.metrics.Metric):
    def __init__(self, k=5, **kwargs):
        super().__init__(**kwargs) # handles base args (e.g., dtype)
        self.k = k
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = tf.cast(y_true, tf.int32)
        y_true = tf.reshape(y_true, shape= [-1])
        b_array = tf.math.in_top_k(y_true, y_pred, self.k)
        num_corrects = tf.reduce_sum(tf.cast(b_array, tf.float32))
        self.total.assign_add(tf.reduce_sum(num_corrects))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

    def result(self):
        return self.total / self.count
```

**Update for each mini-batch**

**Return the final result**

```python
vgg_model = create_vgg_model(10)
vgg_model.summary()
opt = tf.keras.optimizers.Adam(learning_rate=0.01)
vgg_model.compile(optimizer= opt, loss= 'sparse_categorical_crossentropy', metrics=['accuracy',TopkAcc(k=5)])
```

# Observing training progress using TensorBoard

# Observing training progress using TensorBoard

```
tf.keras.backend.clear_session()
```

```
vgg_model = create_vgg_model(10)
vgg_model.summary()
opt = tf.keras.optimizers.Adam(learning_rate=0.01)
vgg_model.compile(optimizer= opt, loss= 'sparse_categorical_crossentropy', metrics=['accuracy',TopkAcc(k=5)])
```

```
import time
import os
log_dir = os.path.join('.\\logs', 'logs_' + time.strftime('%Y-%m-%d_%H.%M.%S'))
os.makedirs(log_dir)
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, write_graph=True, histogram_freq=1, update_freq='epoch', write_images=True, profile_batch=1000000)
```

```
vgg_model.fit(X_train, y_train, epochs= 5, batch_size = 64, validation_data= (X_valid, y_valid), callbacks= [tensorboard_callback])
```

# Showing TensorBoard

# Visualize the gradients

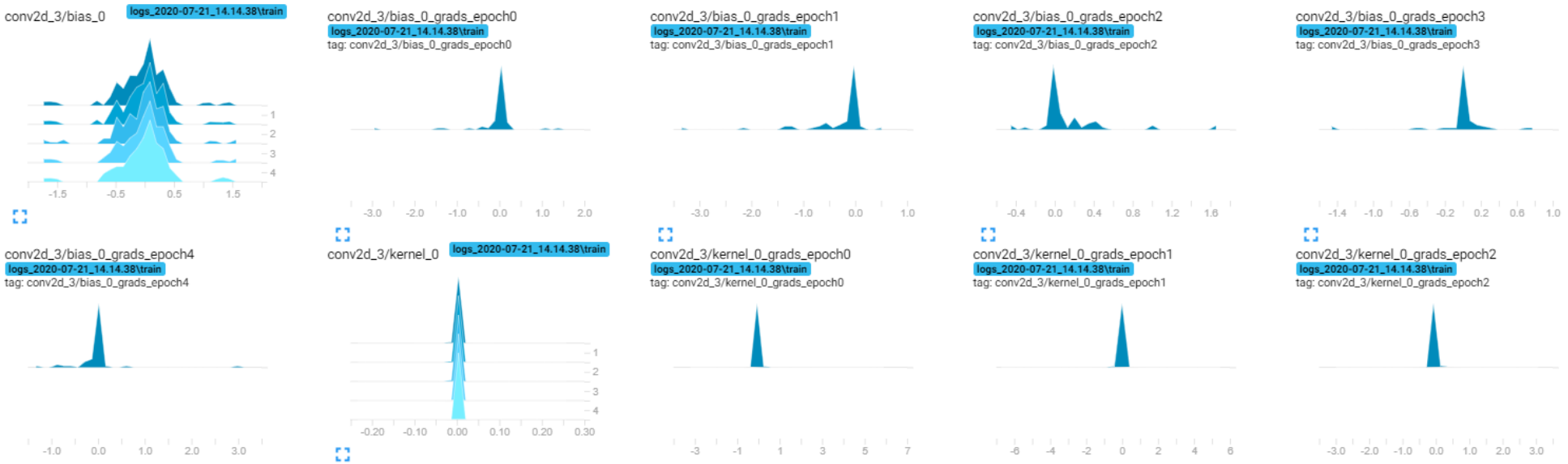
```
import numpy as np
class GradientLogTensorBoard(tf.keras.callbacks.TensorBoard):
    def __init__(self, log_dir='logs', histogram_freq=1, write_graph=False, write_images=False, update_freq='epoch', profile_batch=100000000, embeddings_freq=0, embeddings_metadata=None, num_instance= 100):
        super(GradientLogTensorBoard, self).__init__(log_dir, histogram_freq, write_graph, write_images, update_freq, profile_batch, embeddings_freq, embeddings_metadata)
        self.log_dir = log_dir
        self.histogram_freq = histogram_freq
        self.write_graph = write_graph
        self.write_images= write_images
        self.update_freq = update_freq
        self.profile_batch = profile_batch
        self.embeddings_freq = embeddings_freq
        self.embeddings_metadata = embeddings_metadata
        self.num_instance = num_instance

    def log_gradient(self, epoch):
        idxs = np.random.choice(X_train.shape[0], self.num_instance, replace= False)
        X_batch = X_train[idxs]
        y_batch = y_train[idxs]

        writer = self._get_writer(self._train_run_name)
        with writer.as_default(), tf.GradientTape() as g:
            g.watch(tf.convert_to_tensor(X_batch, dtype=tf.float64))
            _y_pred = self.model(X_batch)  # forward-propagation
            loss = tf.losses.sparse_categorical_crossentropy(y_batch, _y_pred)  # calculate loss
            gradients = g.gradient(loss, self.model.trainable_weights)  # back-propagation

            # In eager mode, grads does not have name, so we get names from model.trainable_weights
            for weights, grads in zip(self.model.trainable_weights, gradients):
                tf.summary.histogram(weights.name.replace(':', '_')+'_grads_epoch' + str(epoch), data=grads, step = epoch)
        writer.flush()


    def on_epoch_end(self, epoch, logs=None):
        super(GradientLogTensorBoard, self).on_epoch_end(epoch, logs=logs)
        if self.histogram_freq:
            self.log_gradient(epoch)
```

```
import time
import os
log_dir = os.path.join('.\\logs\\gradient_logs', 'logs_' + time.strftime('%Y-%m-%d_%H.%M.%S'))
os.makedirs(log_dir)
gradient_log = GradientLogTensorBoard(num_instance=32, histogram_freq=1, update_freq='epoch', write_images=True, profile_batch=1000000, log_dir= log_dir)
```

```
vgg_model.fit(X_train, y_train, epochs= 5, batch_size = 64, validation_data= (X_valid, y_valid), callbacks= [gradient_log])
```

```
Train on 5000 samples, validate on 5000 samples
Epoch 1/5
5000/5000 [==============================] - 20s 4ms/sample - loss: 3.5334 - accuracy: 0.4444 - topk_acc: 0.9094 - val_loss: 3.5303 - val_accuracy: 0.4098 - val_topk_acc: 0.9042
Epoch 2/5
5000/5000 [==============================] - 22s 4ms/sample - loss: 3.4641 - accuracy: 0.4286 - topk_acc: 0.9018 - val_loss: 3.4870 - val_accuracy: 0.3230 - val_topk_acc: 0.8858
Epoch 3/5
```

# Visualize the gradients



# Summary

- Setting of a machine learning problem
  - General loss versus empirical loss
- Challenges in solving DL optimization problem
  - Highly non-convex, complicated, overparameterized, too many saddle points.
- 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
- Visualize training process with Tensor board.

# Thanks for your attention!

## Question time

# Appendix
# 37 reasons why your NNs do not work

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

# 37 reasons why not work

## Dataset issues

- Check your input data
- Try random input
- Check the data loader
- Make sure input is connected to output
- Is the relationship between input and output too random>
- Is there too much noise in the dataset?
- Shuffle the dataset
- Reduce class imbalance
- Do you have enough training examples?
- Make sure your batches don’t contain a single label
- Reduce batch size: very large batch can reduce generalization

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

# 37 reasons why not work

## Data Normalization/Augmentation

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- Standardize the features: zero mean, unit variance
- Do you have too much data augmentation
- Check the preprocessing of your pretrained model: use the same format
- Check the preprocessing for train/validation/test set

# 37 reasons why not work

## Implementation issues

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- Try solving a simpler version of the problem
- Look for correct loss “at chance”
  - “Again from the excellent CS231n: *Initialize with small parameters, without regularization. For example, if we have 10 classes, at chance means we will get the correct class 10% of the time, and the Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302.* After this, try increasing the regularization strength which should increase the loss.”
- Check your loss function
- Verify loss input
- Adjust loss weights
- Monitor other metrics
- Test any custom layers

# 37 reasons why not work

## Implementation issues

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- Check for “frozen” layers or variables
- Increase network size
- Check for hidden dimension errors
- Explore Gradient checking
  - If implement GD by hand, do gradient checking to make sure BackProp

# 37 reasons why not work

## Training issues

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- Solve for a really small dataset
  - Overfit a small subset of the data and make sure it works
- Check weights initialization
  - Use Xavier or He if unsure
- Check your hyperparameters
  - E.g. grid search
- Reduce regularization
  - Too much regularization can cause underfit
  - Get rid of underfit, instead overfit the training data first, then address overfitting.
- Give it time
  - It might take time before meaningful prediction appears, watch your loss to see if it decreases.

# 37 reasons why not work

## Training issues

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- Switch from Train to Test mode
  - "Some frameworks have layers like Batch Norm, Dropout, and other layers behave differently during training and testing. Switching to the appropriate mode might help your network to predict properly."
- Visualize the training
  - Monitor activations, weights, and update for each layer
  - Watch out for layer activation with mean >> 0 (try batch norm or ELUs)
  - In general layer update should have a Gaussian distribution
- Try a different optimizer
  - SGD with momentum and Adam
- Exploding/Vanishing gradients
  - Very large values at layer updates might indicate exploding gradients, try gradient clipping.
  - "*A good standard deviation for the activations is on the order of 0.5 to 2.0. Significantly outside of this range may indicate vanishing or exploding activations.*"

# 37 reasons why not work

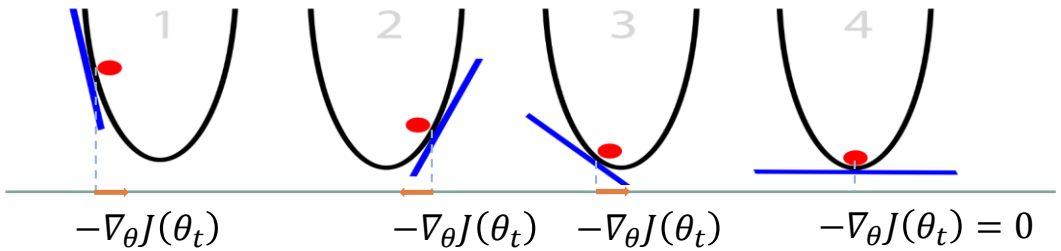
## Training issues

https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- Increase/Decrease Learning Rate
  - Low learning rate causes slow convergence
  - High learning decreases loss quickly but might not settle to good local minima.
- Overcoming NaNs
  - “Getting a NaN (Non-a-Number) is a much bigger issue when training RNNs (from what I hear). Some approaches to fix it:
    - Decrease the learning rate, especially if you are getting NaNs in the first 100 iterations.
    - NaNs can arise from division by zero or natural log of zero or negative number.
    - Try evaluating your network layer by layer and see where the NaNs appear.”

# Gradient Descent Revisited

- Objective is to solve: $\min_{\theta} J(\theta)$
- Move along the opposite direction of the current gradient
  - $\theta_{t+1} = \theta_t - \eta \nabla_w J(\theta_t)$ where $\eta > 0$ is the learning rate



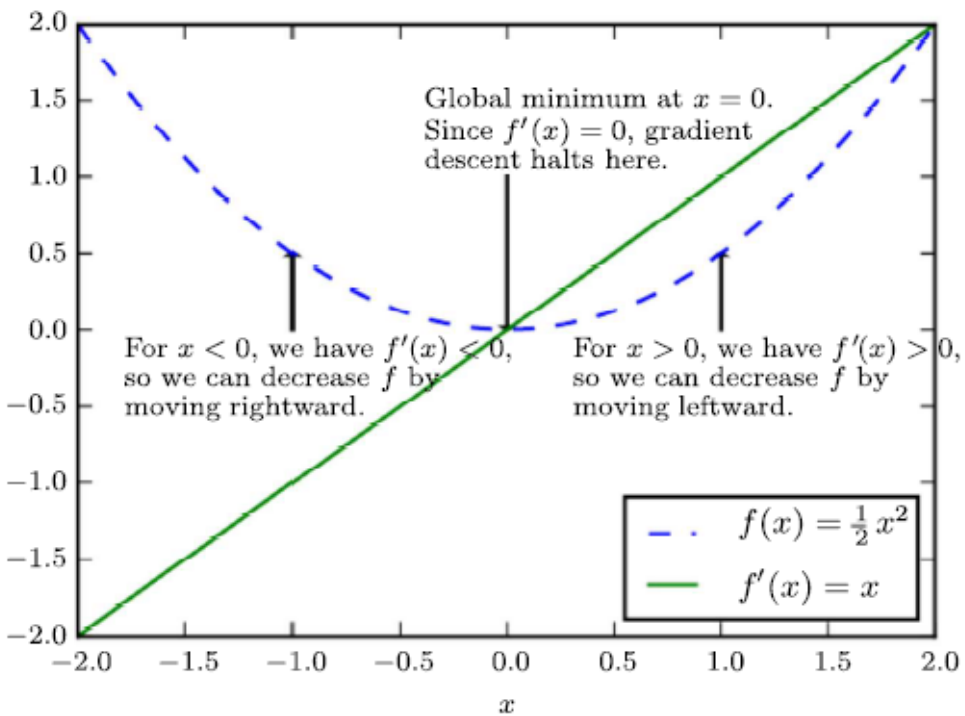



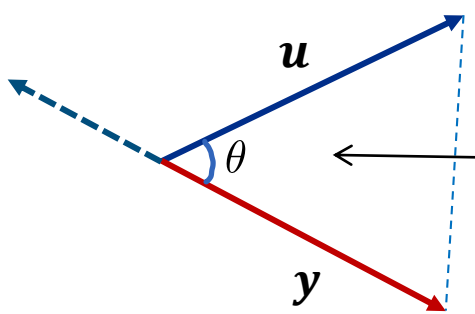

Figure 4.1: An illustration of how the gradient descent algorithm uses the derivatives of a function can be used to follow the function downhill to a minimum.

# Gradient Descent Revisited

- Objective is to solve: $\min_{\theta} J(\theta)$
- Move along the opposite direction of the current gradient
  - $\theta_{t+1} = \theta_t - \eta \nabla_w J(\theta_t)$ where $\eta > 0$ is the learning rate
- Alternative: **line search** = search for $\epsilon$ that minimize $f(x + \epsilon \nabla_x f(x))$
- GD guaranteed to converge to a global minima if $J(.)$ is convex.
- Could get stuck in a local minima if $J(.)$ is nonconvex.

- Why the opposite direction of the gradient gives the best/steepest descent?
- Answer: Let $u$ be any unit vector an consider how much $f$ descent along direction of $u$.
- This is equivalent to directional gradient = slope of function $f$ along direction $u$ = derivative of function $f(x + \alpha u)$ w.r.t $\alpha$, evaluated at $\alpha = 0$.
- Let $h = x + \alpha u$, $\frac{\partial h}{\partial \alpha} = u$, and when $\alpha = 0$, then $h = x$ and $\nabla_h f(x) = \nabla_x f(x)$

$$\left.\frac{\partial f(x + \alpha u)}{\partial \alpha}\right|_{\alpha=0} = \frac{\partial f}{\partial h} \times \frac{\partial h}{\partial \alpha} = u^\top \nabla_x f(x)$$




$$\cos(\theta) = \frac{\boldsymbol{u}^T \boldsymbol{y}}{||\boldsymbol{x}||||\boldsymbol{y}||}$$

$$\cos \pi = -1$$

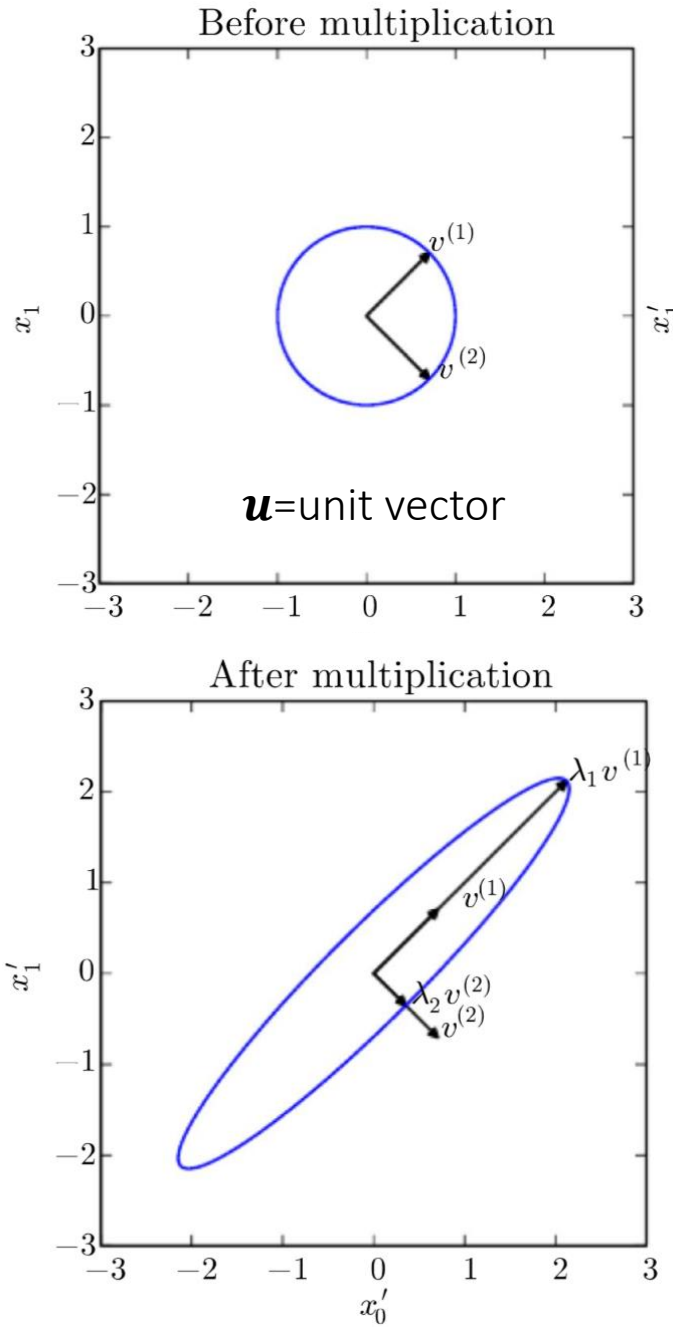
# Eigenvectors/Eigenvalues

- A vector $u$ is an eigenvector of a square matrix $A$ if its multiplication with $A$ doesn't change its direction, i.e.,

$$Av = \lambda v$$

- Respectively $\lambda$ is called the eigenvalue corresponding to $v$.
  - This eigenvalue might not be real-valued and can be complex-valued.
- In deep learning/machine learning, we mostly deal with square and symmetric matrix, (e.g., from covariance, Hessian matrices, etc). In this case, all eigenvalues are real.
- If $v$ is an eigenvector, then its scaled $\beta v$ for $\beta \neq 0$ is also an eigenvector, hence we usually restrict $v$ to a unit vector, i.e., $\|v\|_2 = 1$ or $v^\top v = 1$ .
- Let $V$ be the matrix of eigenvectors, then $A$ can further be written as:

$$A = \mathrm{V}\mathrm{diag}(\boldsymbol{\lambda})V^\top$$

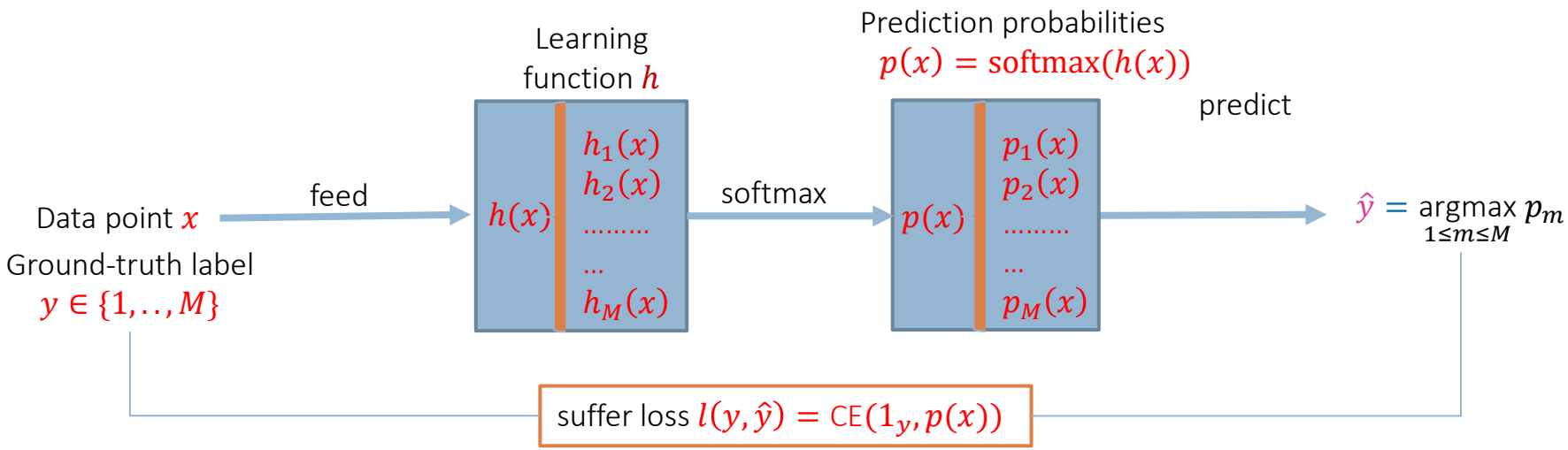



$\boldsymbol{Au}$ is stretched in direction $v_i$ by $\lambda_i$

(see DL book, chapter 2 for more comprehensive revision)

from previous lecture ….

# Training a deep NN

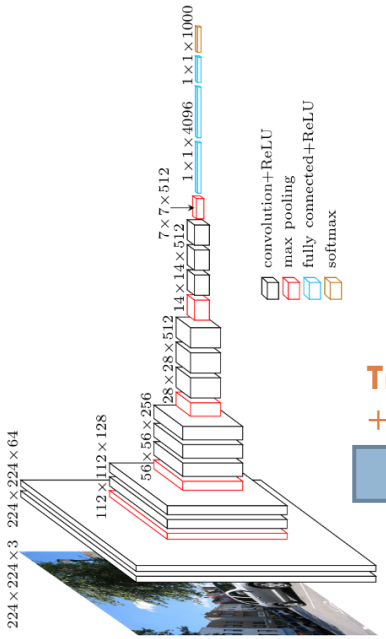



- A data point $x$ receives discriminative values $h_1(x), \dots, h_M(x)$ from the model
  - $h_m(x)$ represents the **possibility** to **classify** $x$ to **class** $m$ for $1 \le m \le M$
- We use these discriminative values to predict the label $\hat{y}$ as
  - $\hat{y} = \text{argmax}_{1 \le m \le M} h_m(x)$, meaning the class with highest discriminative value
- The prediction $x$ with the predicted label $\hat{y}$ suffers a loss
  - $l(\hat{y}, y)$ where $l$ is a loss function (if $\hat{y} = y$ then $l(\hat{y}, y) = 0$).
- Given a training set $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$, the loss incurred is
  - $\frac{1}{N} \sum_{i=1}^{N} l(y_i, \hat{y}_i)$

# 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**

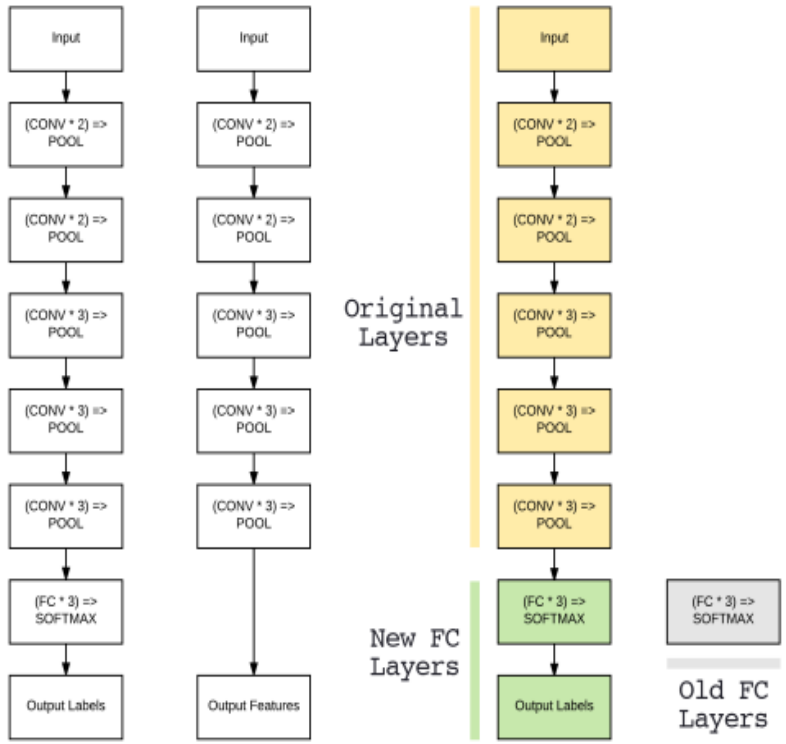
**Your small-scale dataset**
(Flower-17)
-17 category dataset with 80 images per class


**Not enough data** to train a good model.
HOW?

# Transfer Learning

## How to Do That?

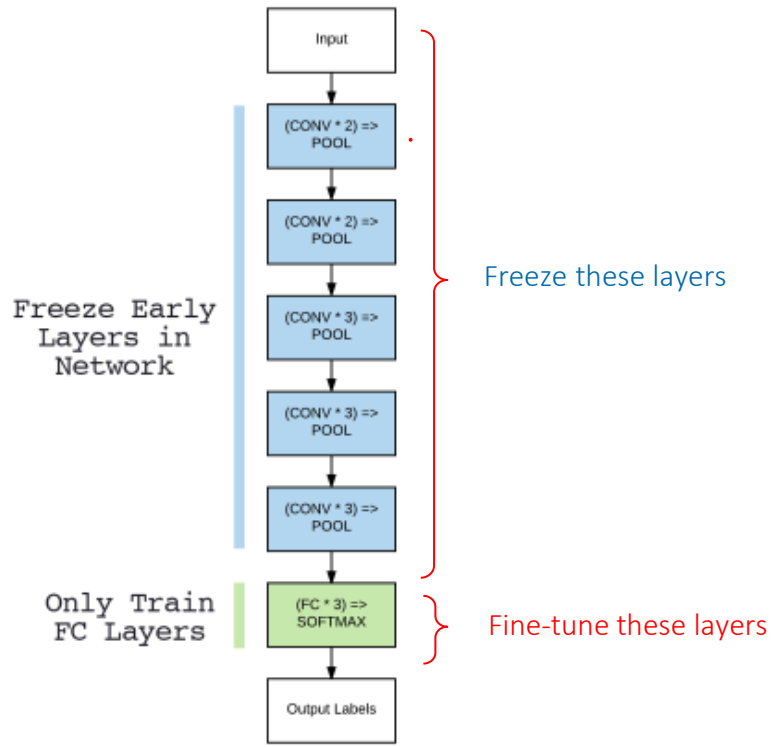
- **Remove** FC layers from the pretrained model
- **Replace** them with a brand new FC head. These new FC layers can then be fine-tuned to the specific dataset (the old FC layers are no longer used)



# Transfer Learning

## How to Do That?

- **Freeze** all CONV layers in the network and 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 as we normally would; however, the backpropagation is stopped after the FC layers
  - Allows these layers to start to learn patterns from the highly discriminative CONV layers

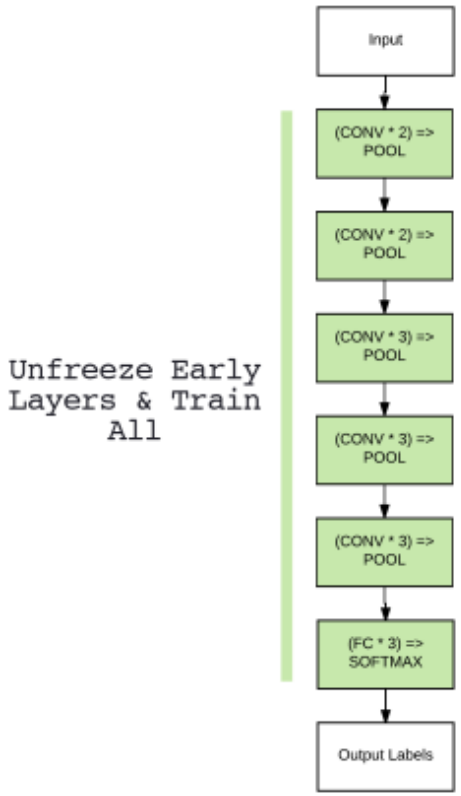


# Transfer Learning

## 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 and allow each of them to be **fine-tuned**.

- After the FC head has started to learn patterns in our dataset, pause training, unfreeze the body, and then continue the training, *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 TF 2.x

| Model | Size | Top-1 Accuracy | Top-5 Accuracy | Parameters | Depth |
|---|---|---|---|---|---|
| Xception | 88 MB | 0.790 | 0.945 | 22,910,480 | 126 |
| VGG16 | 528 MB | 0.713 | 0.901 | 138,357,544 | 23 |
| VGG19 | 549 MB | 0.713 | 0.900 | 143,667,240 | 26 |
| ResNet50 | 98 MB | 0.749 | 0.921 | 25,636,712 | - |
| ResNet101 | 171 MB | 0.764 | 0.928 | 44,707,176 | - |
| ResNet152 | 232 MB | 0.766 | 0.931 | 60,419,944 | - |
| ResNet50V2 | 98 MB | 0.760 | 0.930 | 25,613,800 | - |
| ResNet101V2 | 171 MB | 0.772 | 0.938 | 44,675,560 | - |
| ResNet152V2 | 232 MB | 0.780 | 0.942 | 60,380,648 | - |
| InceptionV3 | 92 MB | 0.779 | 0.937 | 23,851,784 | 159 |
| InceptionResNetV2 | 215 MB | 0.803 | 0.953 | 55,873,736 | 572 |
| MobileNet | 16 MB | 0.704 | 0.895 | 4,253,864 | 88 |
| MobileNetV2 | 14 MB | 0.713 | 0.901 | 3,538,984 | 88 |
| DenseNet121 | 33 MB | 0.750 | 0.923 | 8,062,504 | 121 |
| DenseNet169 | 57 MB | 0.762 | 0.932 | 14,307,880 | 169 |
| DenseNet201 | 80 MB | 0.773 | 0.936 | 20,242,984 | 201 |
| NASNetMobile | 23 MB | 0.744 | 0.919 | 5,326,716 | - |
| NASNetLarge | 343 MB | 0.825 | 0.960 | 88,949,818 | - |
| EfficientNetB0 | 29 MB | - | - | 5,330,571 | - |
| EfficientNetB1 | 31 MB | - | - | 7,856,239 | - |
| EfficientNetB2 | 36 MB | - | - | 9,177,569 | - |
| EfficientNetB3 | 48 MB | - | - | 12,320,535 | - |
| EfficientNetB4 | 75 MB | - | - | 19,466,823 | - |
| EfficientNetB5 | 118 MB | - | - | 30,562,527 | - |
| EfficientNetB6 | 166 MB | - | - | 43,265,143 | - |
| EfficientNetB7 | 256 MB | - | - | 66,658,687 | - |