FIT3181/5215 Deep Learning

Revision for Final Exam

Lecturer: Trung Le

Email: trunglm@monash.edu

GROUP OF EIGHT AUSTRALIA

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

The big picture …

Lectures 1,2,4,5: fundamentals

Deep Learning/ML
Deep Neural Networks
Computation graphs
Optimization, BackProp

Week 01	Machine Learning and Mathematical Background Revisit
Week 02	Prelude to Deep Learning and Feed-forward Neural Networks
Week 04	Back-Propagation and Optimization for Deep Learning
Week 05	Practical skills in deep learning

Lectures 3,6,10: Deep Computer Vision

Convolutions, CNN, architectures, robustness

Week 03	DL for Vision (I): Convolutional Neural Networks
Week 06	DL for Vision (II): Network Architectures, Visualization, Interpretability, and Robustness
Week 10	ViT and Model Fine-tuning Techniques

Lectures 7,9: Deep learning for sequential, time-series

RNNs, LSTM, GRU, Seq2Seq, attention mechanism, Transformers, Self-Attention etc.

Week 07	Deep learning for time-series and temporal data: RNNs, LSTMs
Week 09	Advanced sequential models

Lecture 8: Discrete Learning Representation

Word2Vec, Skip-Gram, CBOW

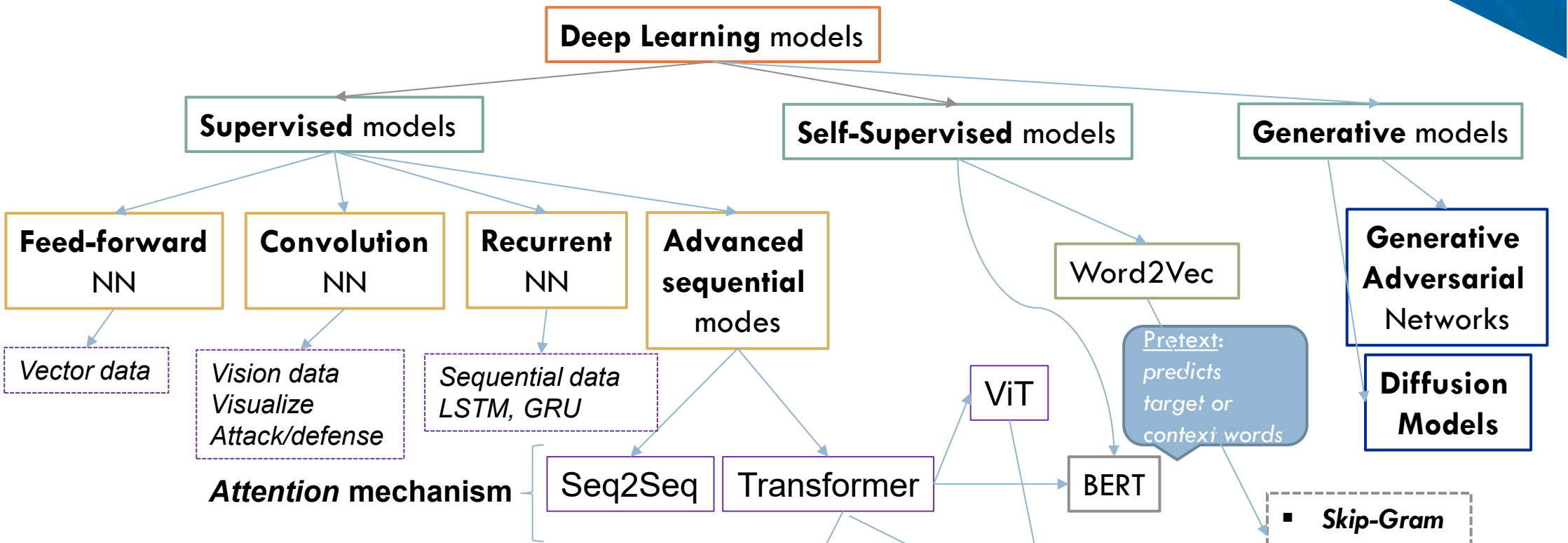
Week 08	Representation and deep learning for language: Word2Vec

Lecture 11: Deep Generative Models

Generative Adversarial Network (GAN), Diffusion Models

Week 11	Deep generative models: GANs and Diffusion Models

Knowledge graph of Deep Learning unit



Define loss function

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

Forward propagation

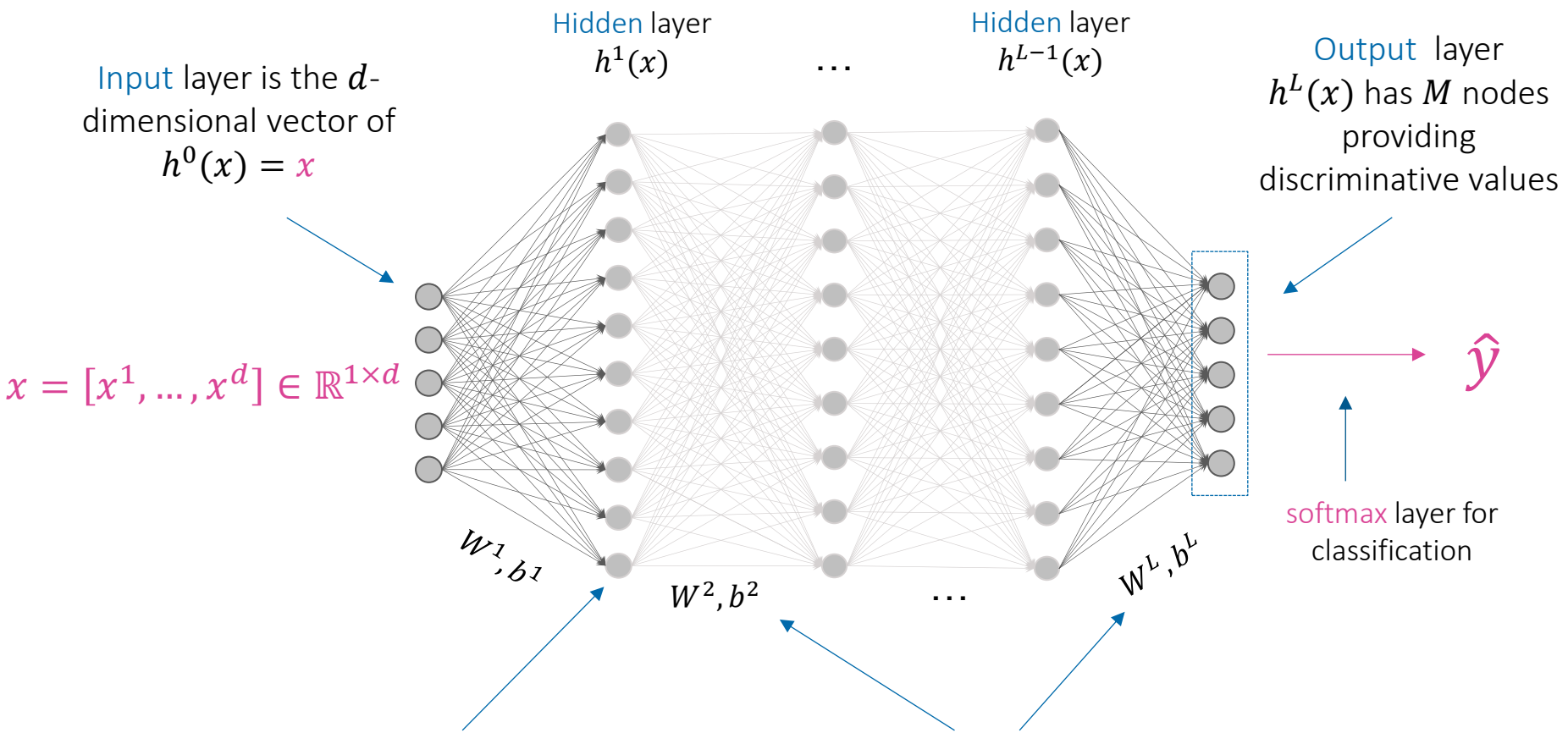
- Input → Layers → Prediction Probabilities → Loss

Backward propagation

- Loss → Layers → Input

Feed-forward NNs

Deep NNs: parameterisation



- Layer k has n_k neurons
  - $n_0 = d$ and $n_L = M$

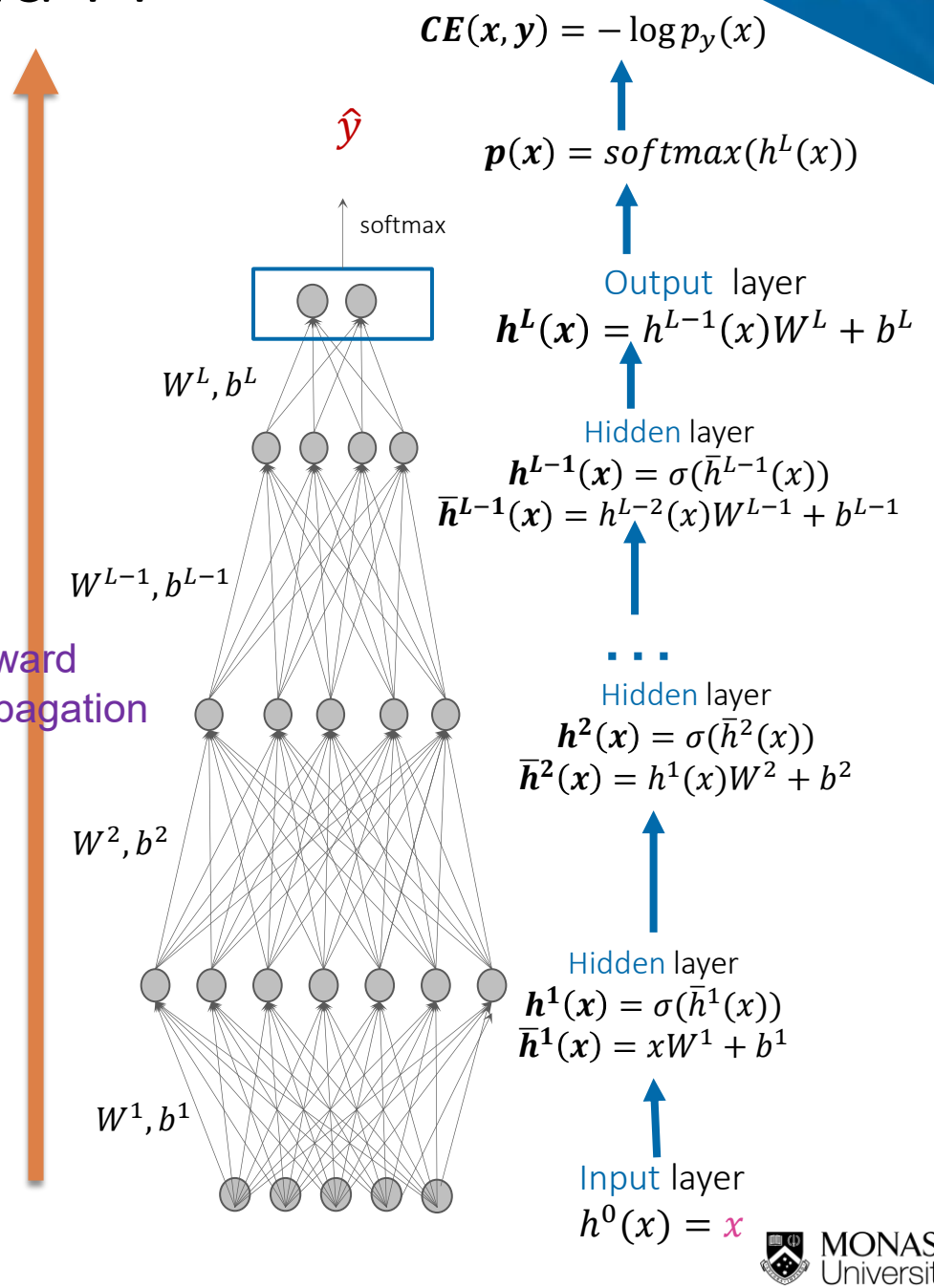
- Weight matrices and biases
  - $W^k \in \mathbb{R}^{n_{k-1}\times n_k}$ and $b^k \in \mathbb{R}^{1\times n_k}$ for $k = 1,2, \dots, L$

DNNs: making prediction via FP

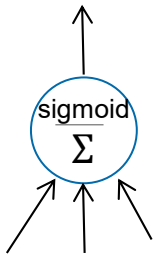
Forward propagation- Classification

$h^0(x) = x$ with the ground-truth label y
for $k = 1$ **to** $L - 1$ **do**
  $\bar{h}^k(x) = h^{k-1}(x)W^k + b^k$ //linear operation
  $h^k(x) = \sigma(\bar{h}^k(x))$ //activation to bring non-linearity
$h^L(x) = h^{L-1}(x)W^L + b^L$
$p(x) = softmax(h^L(x))$ //prediction probabilities

- Parameter: $\theta = (W^k, b^k)_{k=1}^L$ is given
- $\sigma: \mathbb{R} \to \mathbb{R}$ is the **activation function**
  - Sigmoid, tanh, ReLU, and etc.
- Output: $p(x) = \text{softmax}(h^L(x))$
  - $\Pr(y = k|x) = p_k(x) = \frac{\exp\{h_k^L(x)\}}{\sum_{j=1}^M \exp\{h_j^L(x)\}}$
  - $\hat{y} = \underset{1 \le k \le M}{\text{argmax}}\, p_k(x)$

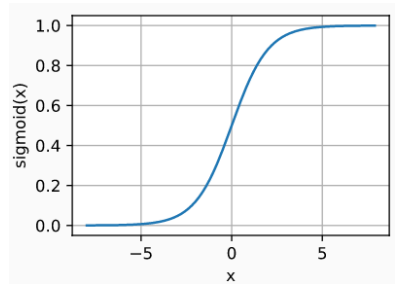


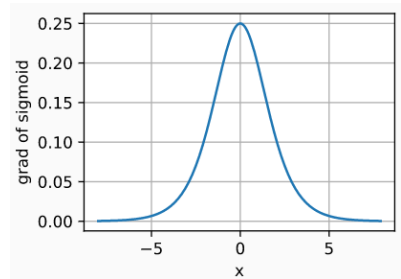
Modern Activation Functions

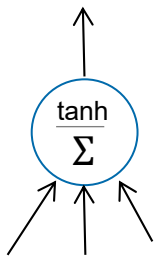


- The ***sigmoid*** function
  - $s(z) = \frac{1}{1+e^{-z}}$
  - Logistic S-shaped, continuous, and differentiable
  - Output range: 0 to 1

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

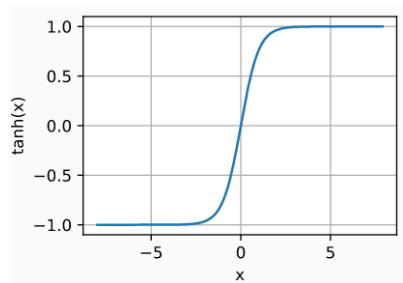


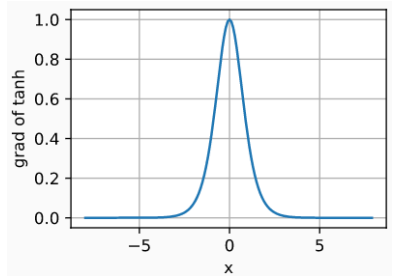


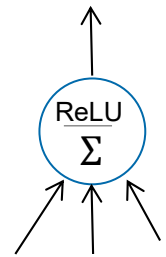


- The **hyperbolic *tangent*** function
  - $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
  - Logistic S-shaped, continuous, and differentiable
  - Output range: -1 to 1, hence help speed up convergence.

$$\sigma(z) = tanh(z) = \frac{\exp\{z\}-\exp\{-z\}}{\exp\{z\}+\exp\{-z\}} \qquad \sigma'(z) = 1-\sigma^2(z)$$

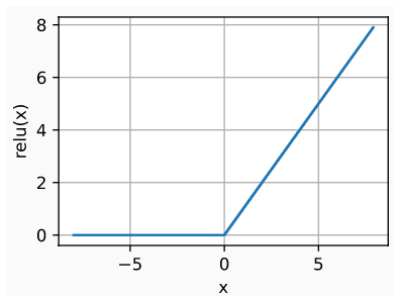


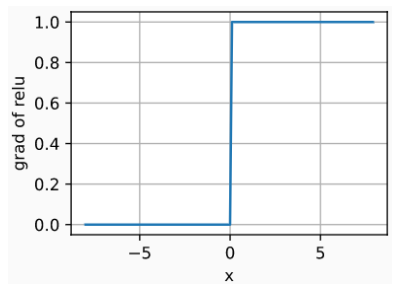




- The ***rectified linear function*** (ReLU)
  - $\mathrm{ReLU}(z) = \max(0, z)$
  - Continuous, but not differentiable at **0**
  - Hence, can make GD bounce around
  - Fast to compute, works very well in practice!
  - No cap on output value, hence help with gradient vanishing problem.

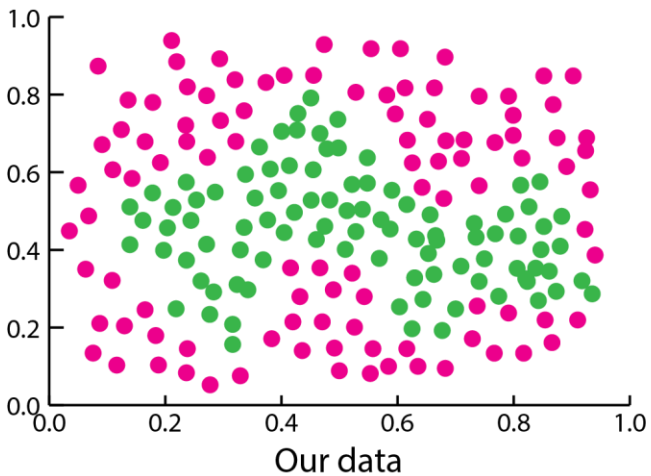
$$\sigma(z) = ReLU(z) = \max\{0, z\} \qquad \sigma'(z) = \begin{cases} 1 & if\ z \geq 0 \\ 0 & otherwise \end{cases}$$



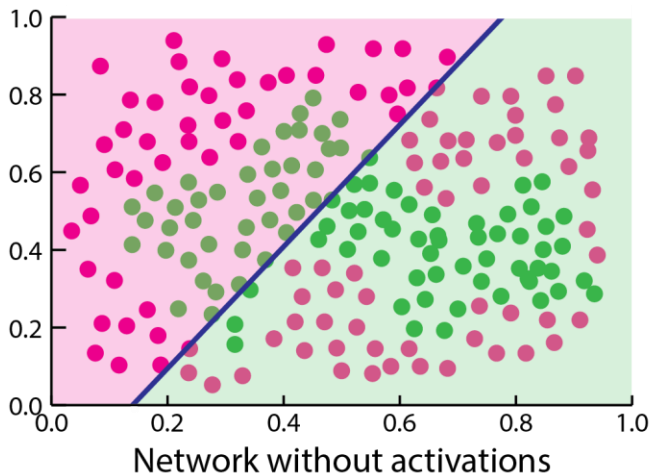


Importance of Activation Functions

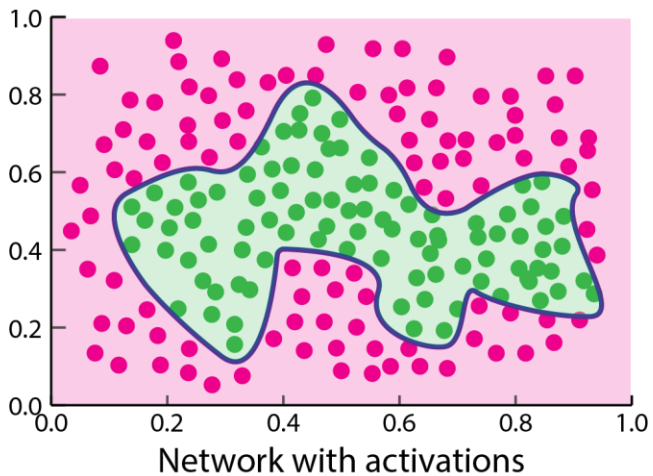
Aims to bring **non-linearities** to neural networks



Our data

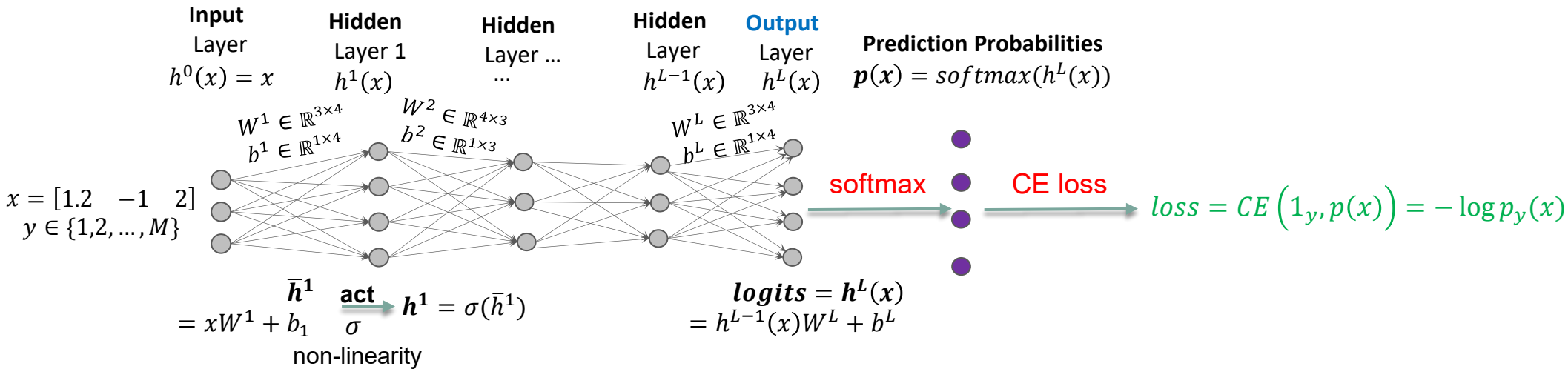


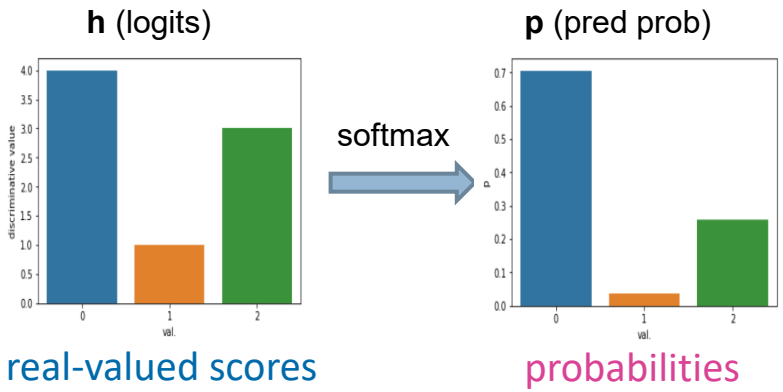
Networks without activations



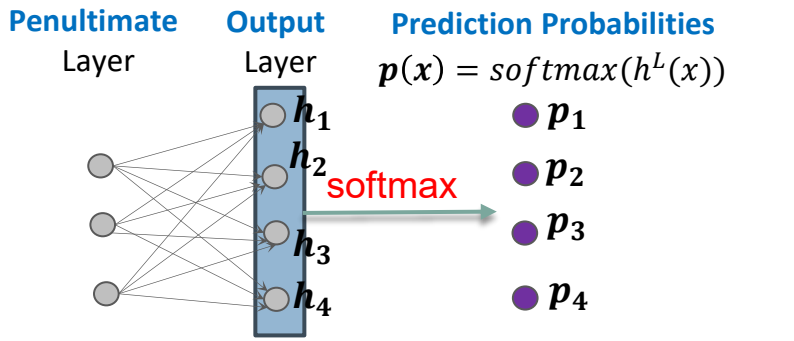
Networks with activations

Linear and Softmax Layers





transforms real-valued discriminative scores (ranged in $(-\infty, +\infty)$) to probability values (ranged in $[0, 1]$) but preserving the order.



Logit $h = [h_1\ h_2\ h_3\ h_4]$ Prob p $= [p_1\ p_2\ p_3\ p_4]$

$$p_1 = \frac{e^{h_1}}{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}$$

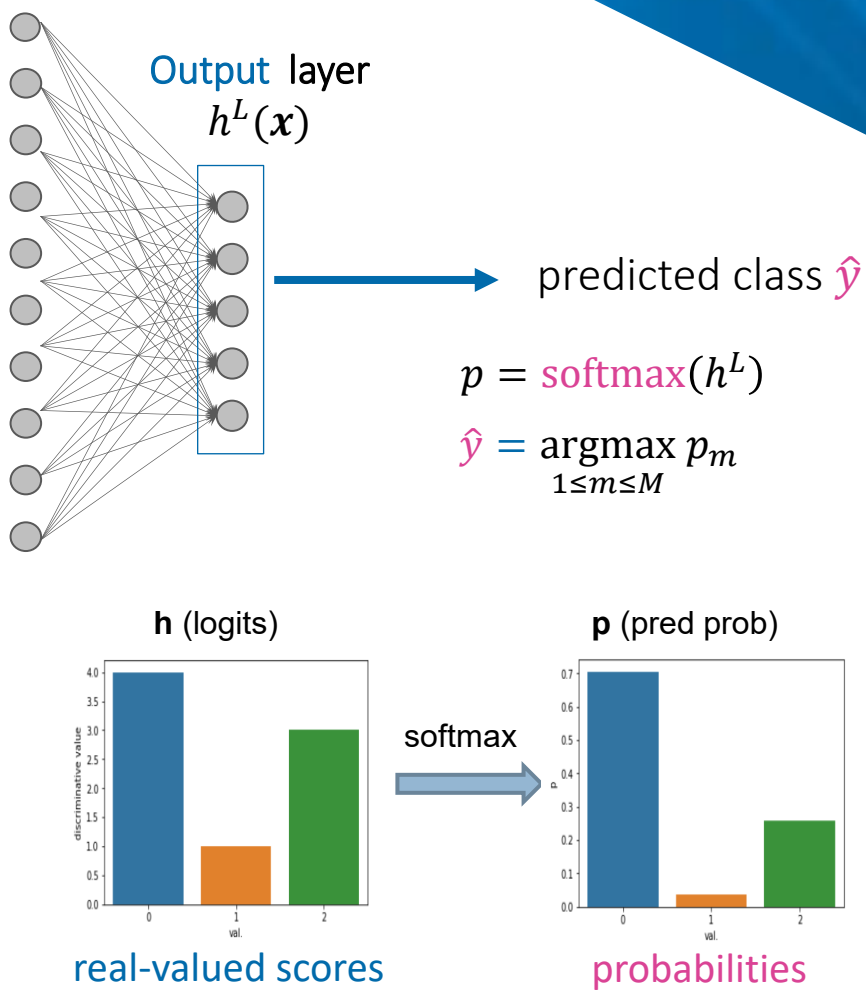
$$p_2 = \frac{e^{h_2}}{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}$$

$$p_3 = \frac{e^{h_3}}{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}$$

$$p_4 = \frac{e^{h_4}}{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}$$

What is a softmax layer?

- Softmax function transforms **real-valued discriminative scores h** to **discrete probabilities p**.
- Input: vector of dimension M
  - e.g., $h = [h_1, h_2, h_3] = [1, 2, 3]$, M $= 3$
- Output: a discrete distribution of dimension M
  - e.g., $p = [p_1, p_2, p_3] = [0.09, 0.24, 0.67]$
- How to calculate p_m?
  - $h = [h_m]_{m=1}^{M} \rightarrow p = \text{softmax}(h) := \left[\frac{\exp\{h_m\}}{\sum_{i=1}^{M}\exp\{h_i\}}\right]_{m=1}^{M}$
  - $p = [p_m]_{m=1}^{M}$ becomes a distribution over classes $\{1, \dots, M\}$
    - $p_m \geq 0$ $(1 \leq m \leq M)$ and $\sum_{m=1}^{M} p_m = 1$.



transforms real-valued discriminative scores (ranged in $(-\infty, +\infty)$) to probability values (ranged in $[0, 1]$) but preserving the order.

- Prediction step: return the prediction for the class that has the highest probability

$$\hat{y} = \underset{1\leq m\leq M}{\text{argmax}}\, p_m$$

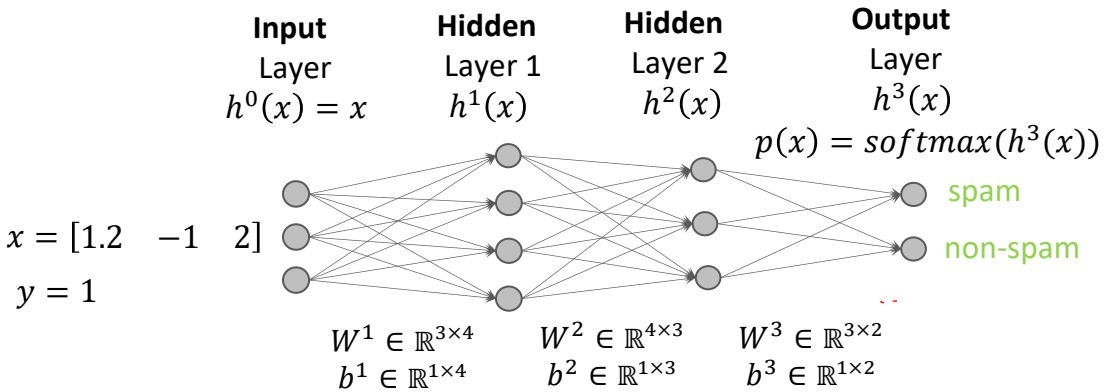
  - i.e., return 3 in the above example (why?)

DNNs: making prediction via FP

Example of spam email detection

- From emails, assume that we extract **three features**, hence data points $x \in \mathbb{R}^3$
  - $x = [x_1, x_2, x_3] \in \mathbb{R}^3$
  - There are **two** classes and labels: **spam** ($y = 1$) and **non-spam** ($y = 2$)

- Network architecture:
  - $3 \rightarrow 4(sigmoid) \rightarrow 3(sigmoid) \rightarrow 2(softmax)$
- $h^0(x) = x = [1.2, -1, 2.2] \in \mathbb{R}^{1\times 3}$
- Hidden layer 1
  - $\bar{h}^1(x) = h^0(x)W^1 + b^1 \in \mathbb{R}^{1\times 4}$ *(linear operation)*
  - $h^1(x) = sigmoid\left(\bar{h}^1(x)\right) = \left[sigmoid\left(\bar{h}_i^1(x)\right)\right]_{i=1}^{4} \in \mathbb{R}^{1\times 4}$ *(activation)*
- Hidden layer 2
  - $\bar{h}^2(x) = h^1(x)W^2 + b^2 \in \mathbb{R}^{1\times 3}$ *(linear operation)*
  - $h^2(x) = sigmoid\left(\bar{h}^2(x)\right) \in \mathbb{R}^{1\times 13}$ *(element-wise activation)*
- Output layer
  - $h^3(x) = h^2(x)W^3 + b^3 \in \mathbb{R}^{1\times 2}$ *(linear operation)*
  - $p(x) = softmax\left(h^3(x)\right) \in \mathbb{R}^{1\times 2}$ *(softmax activation)*
- Assume that $p(x) = [0.3, 0.7]$
  - Prediction $\hat{y} = 2 \neq y = 1 \rightarrow$ ***incorrect*** *prediction*



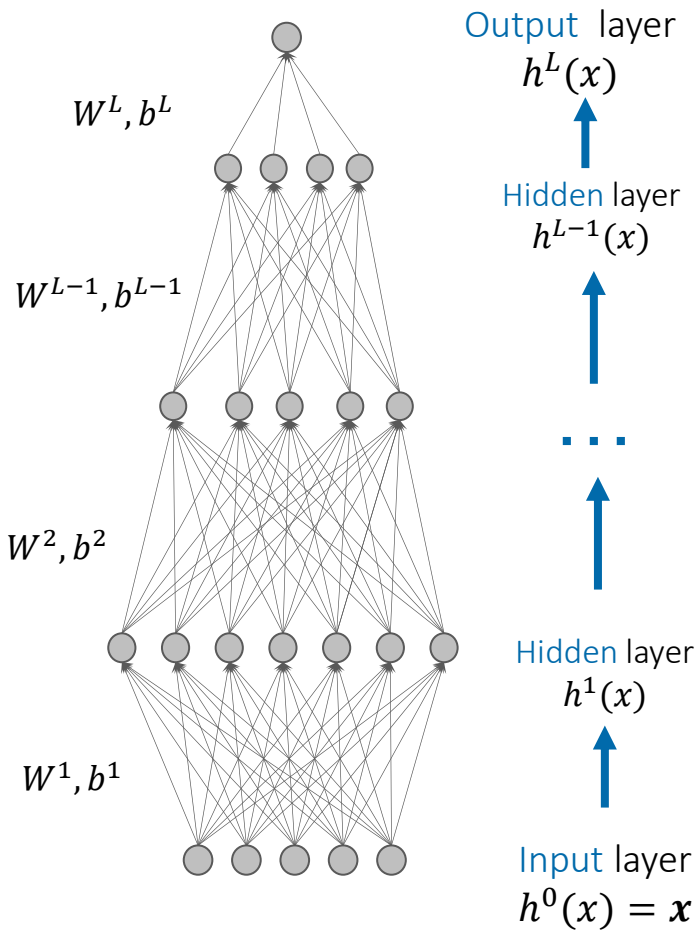
$h^0(x) = x$
for $k = 1$ **to** 2 **do**
  $\bar{h}^k = h^{k-1}(x)W^k + b^k$ //linear operation
  $h^k(x) = \sigma\left(\bar{h}^k(x)\right)$ //activation
$h^3(x) = h^2(x)W^3 + b^2$
$p(x) = softmax(h^3(x))$ //prediction probabilities

Exercise: *check all the dimensions for matrix multiplication consistency!*
What is the cross-entropy loss in this case?

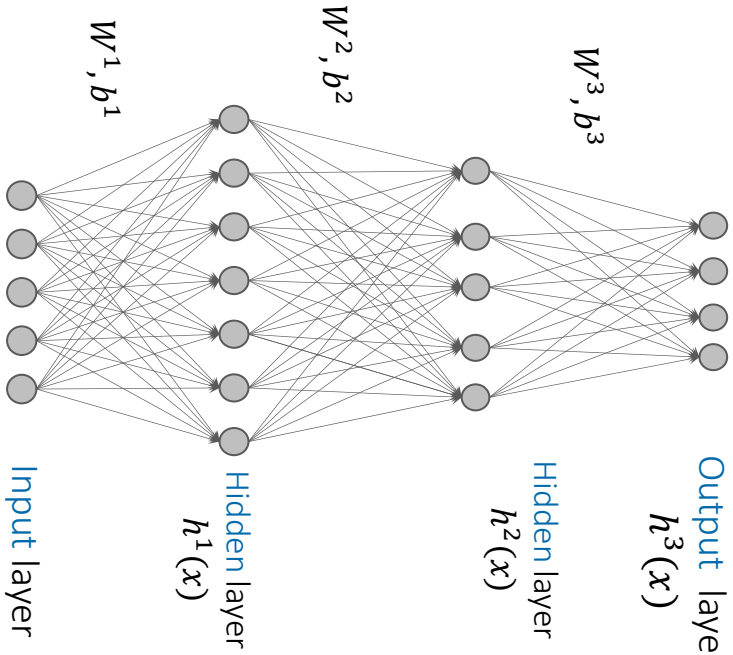
DNNs: making prediction via FP

Forward propagation- Regression

$h^0(x) = x$
for $k = 1$ **to** $L - 1$ **do**
  $\bar{h}^k(x) = h^{k-1}(x)W^k + b^k$ //linear operation
  $h^k(x) = \sigma(\bar{h}^k(x))$ //activation
$h^L(x) = h^{L-1}(x)W^L + b^L$



- Parameter: $\theta = \left(W^k, b^k\right)_{k=1}^{L}$
- $\sigma: \mathbb{R} \to \mathbb{R}$ is the activation function
  - Sigmoid, tanh, ReLU
- Output:
  - $\hat{y} = h^L(x)$



Declare FFN

```
#set the random seed for Torch
torch.manual_seed(1234)

W1 = torch.randn(5, 7)
b1 = torch.randn(1,7)
W2 = torch.randn(7, 5)
b2 = torch.randn(1,5)
W3 = torch.randn(5, 4)
b3 = torch.randn(1,4)
```

Declare one data example

```
x= torch.tensor([[1.0, 2.0, -2.0, 0.2,5.7]])
print(f'x shape: {x.shape}')
```

```
x shape: torch.Size([1, 5])
```

Forward propagation

```
hbar_1 = torch.matmul(x, W1) + b1
h1 = torch.relu(hbar_1)
print(f'hbar_1: {hbar_1}')
print(f'h1: {h1}')
```

```
hbar_1: tensor([[ 8.9391, -4.6345,  7.4453, -0.9053, -3.9509,  1.2647, -8.7399]])
h1: tensor([[8.9391, 0.0000, 7.4453, 0.0000, 0.0000, 1.2647, 0.0000]])
```

```
hbar_2 = torch.matmul(h1, W2) + b2
h2 = torch.relu(hbar_2)
print(f'hbar_2: {hbar_2}')
print(f'h2: {h2}')
```

```
hbar_2: tensor([[ -1.8582,  -3.6770,  -3.9167,   4.2753, -10.4959]])
h2: tensor([[0.0000, 0.0000, 0.0000, 4.2753, 0.0000]])
```

```
h3 = torch.matmul(h2, W3) + b3
print(f'logit h3: {h3}')
```

```
logit h3: tensor([[  3.5699, -10.3066,  -5.5052,  -2.0499]])
```

Making prediction

```
p = torch.softmax(h3, dim=1)
print(f'p: {p}')
```

```
p: tensor([[9.9627e-01, 9.3731e-07, 1.1406e-04, 3.6119e-03]])
```

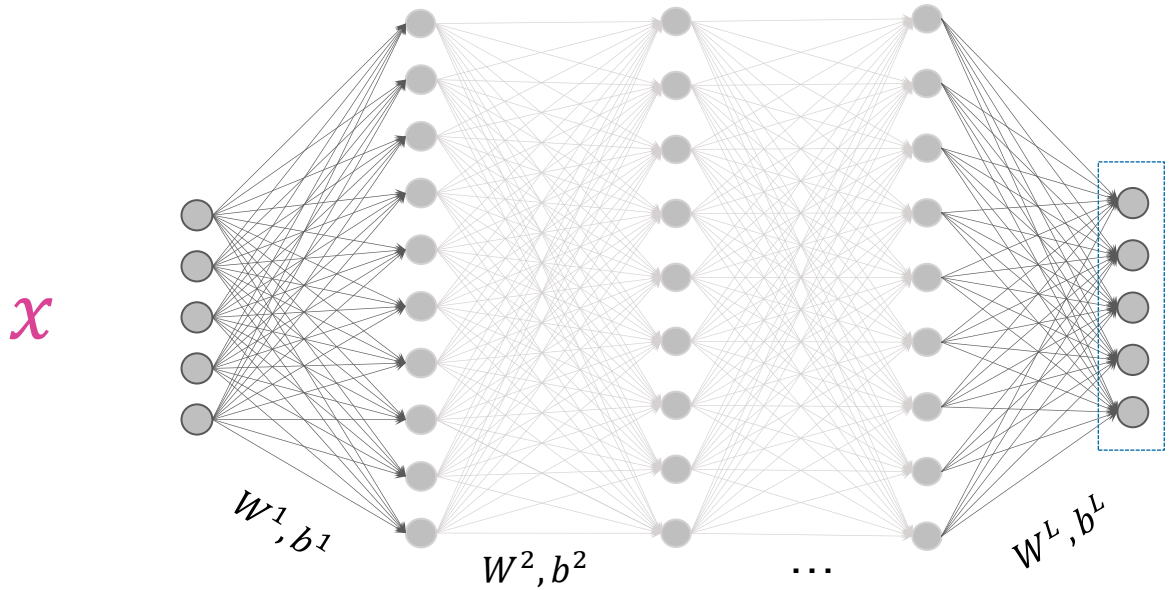
```
yhat = torch.argmax(p, dim=1)
print(f'yhat: {yhat.item()}')
```

```
yhat: 0
```

```
#Assume that ground-truth label y=1 (index starts from 0)
loss = torch.mean(-torch.log(p[0,1]))
print(f'loss: {loss}')
```

```
loss: 3.8615705966949463
```

Training deep nets



suffer loss $l(y, \hat{y}) = \mathrm{CE}\left(1_y, p(x)\right) = -\log p_y(x)$

Training set

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

Loss function

$$L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} CE\left(1_{y_i}, p(x_i)\right) = -\frac{1}{N}\sum_{i=1}^{N} \log p_{y_i}(x_i)$$

(negative log likelihood)

- DNN parameters: $\theta \coloneqq \{(W^l, b^l)\}_{l=1}^{L}$
- Find model parameters (weight matrices and biases) so that the model predictions fit the training set as much as possible:

$$\min_{\theta} L(D;\theta)$$

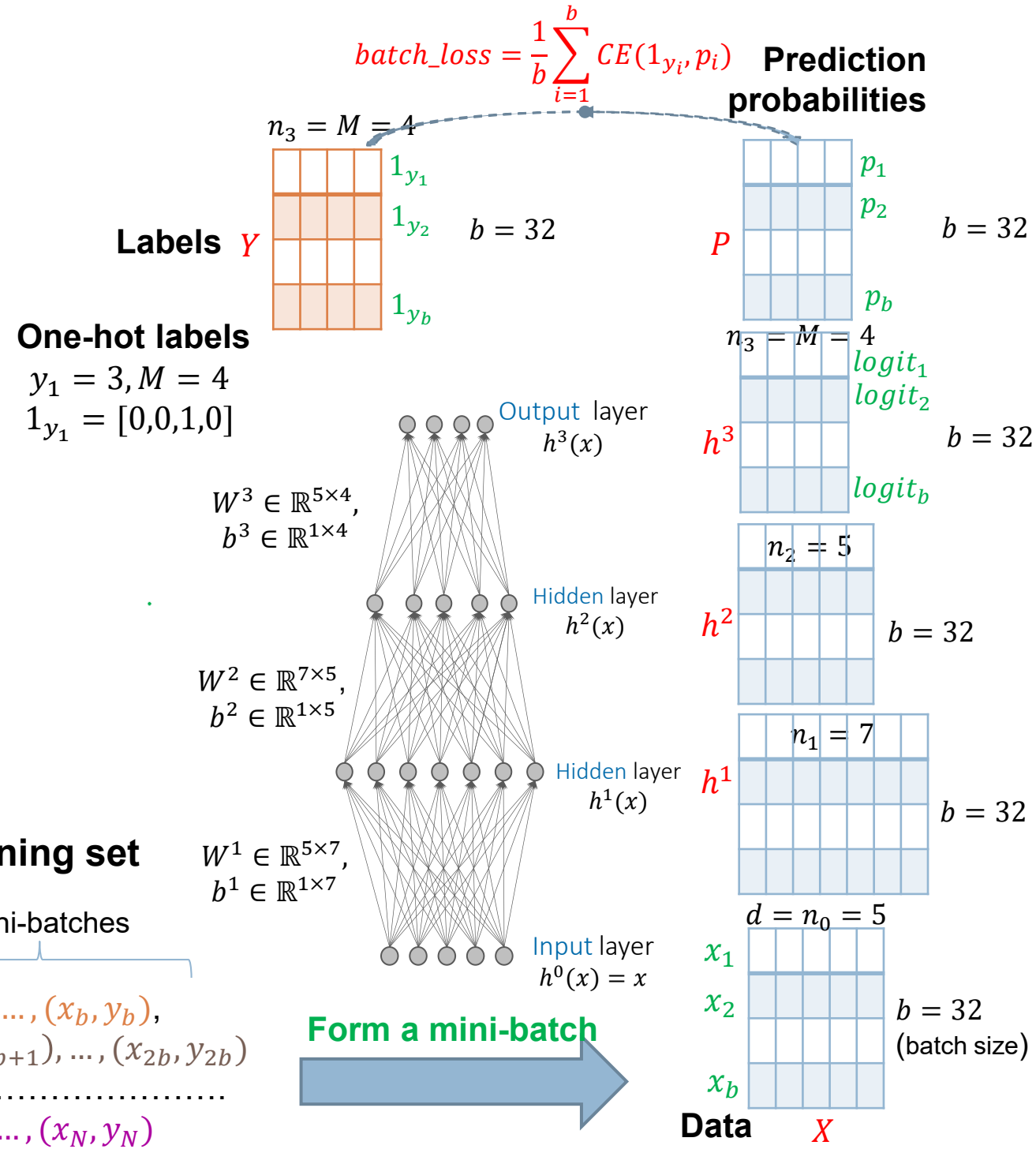
- Use optimizers SGD, Adagrad, Adam, RMSProp to update the model parameters (Lecture 4)

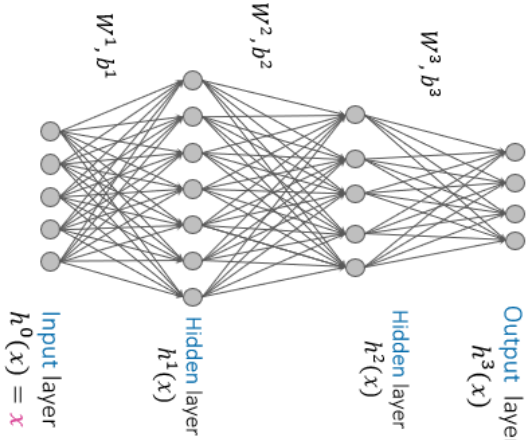
TorchOpt

```
"Adam":torch.optim.Adam
"Adadelta":torch.optim.Adadelta
"Adagrad":torch.optim.Adagrad
"Adamax":torch.optim.Adamax
"AdamW": torch.optim.AdamW
"ASGD":torhc.optim.ASGD,
"NAdam":torch.optim.NAdam,
"RMSprop":torhc.optim.RMSprop
"RAdam":torch.optim.RAdam
"Rprop": torch.optim.Rprop
"SGD":torch.optim.SGD
```

Mini-batch feed-forward

- **Input**
  - Batch X: $[b, n_0 = d = 5]$ (b is the batch size)
- **Hidden layer 1**
  - $\boldsymbol{h^1 = \sigma(XW^1 + b^1)}$
  - $Tensor\,[b, n_1 = 7]$
- **Hidden layer 2**
  - $\boldsymbol{h^2 = \sigma(h^1W^2 + b^2)}$
  - Tensor $[b, n_2 = 7]$
- **Output layer**
  - $\boldsymbol{h^3 = h^2W^3 + b^3}$
  - $P = softmax(h^3, dim = 1)$
  - $Tensor\,\mathrm{P}$: $[b, n_L = M = 4]$
- **The loss of the batch**
  - $\frac{1}{b}\sum_{i=1}^{b} CE(1_{y_i}, p_i) = -\frac{1}{b}\sum_{i=1}^{b} \log p^i_{y_i}$
  - Update weight matrices and biases to **minimize** the batch loss.





Declare FFN

```
#set the random seed for Torch
torch.manual_seed(1234)

W1 = torch.randn(5, 7)
b1 = torch.randn(1,7)
W2 = torch.randn(7, 5)
b2 = torch.randn(1,5)
W3 = torch.randn(5, 4)
b3 = torch.randn(1,4)
```

Declare a batch

```
x= torch.rand(size=(4,5))
print(f'x: {x}')

x: tensor([[0.3574, 0.1191, 0.9922, 0.8757, 0.7378],
        [0.9949, 0.2338, 0.2153, 0.2073, 0.4758],
        [0.0586, 0.8958, 0.5129, 0.7490, 0.2254],
        [0.4485, 0.5658, 0.3631, 0.9719, 0.2716]])
```

Forward propagation

```
hbar_1 = torch.matmul(x, W1) + b1
h1 = torch.relu(hbar_1)
print(f'hbar_1.shape: {hbar_1.shape}')
print(f'h1.shape: {h1.shape}')

hbar_1.shape: torch.Size([4, 7])
h1.shape: torch.Size([4, 7])
```

```
hbar_2 = torch.matmul(h1, W2) + b2
h2 = torch.relu(hbar_2)
print(f'hbar_2.shape: {hbar_2.shape}')
print(f'h2.shape: {h2.shape}')

hbar_2.shape: torch.Size([4, 5])
h2.shape: torch.Size([4, 5])
```

```
h3 = torch.matmul(h2, W3) + b3
print(f'logit h3: {h3}')

logit h3: tensor([[ 0.4458, -3.1415, -2.9431, -0.9237],
        [ 0.1560, -2.4655, -2.6027, -0.4142],
        [ 1.0816, -1.8671, -1.6175, -0.7504],
        [ 0.8463, -1.9495, -1.8475, -0.7677]])
```

```
p = torch.softmax(h3, dim=1)
print(f'p: {p}')

p: tensor([[0.7601, 0.0210, 0.0257, 0.1932],
        [0.5877, 0.0427, 0.0372, 0.3323],
        [0.7814, 0.0410, 0.0526, 0.1251],
        [0.7531, 0.0460, 0.0509, 0.1499]])
```

Making prediction

```
yhat = torch.argmax(p, dim=1)
print(f'yhat: {yhat}')

yhat: tensor([0, 0, 0, 0])
```

```
#Assume that ground-truth label y = [0,1,0,1]
y = torch.tensor([0,1,0,1])
print(f'y: {y}')
one_hot_y = torch.nn.functional.one_hot(y,num_classes=4)
print(f'one_hot_y: {one_hot_y}')
print(one_hot_y * torch.log(p))
loss = -torch.mean(one_hot_y * torch.log(p))
print(f'loss: {loss}')

y: tensor([0, 1, 0, 1])
one_hot_y: tensor([[1, 0, 0, 0],
        [0, 1, 0, 0],
        [1, 0, 0, 0],
        [0, 1, 0, 0]])
tensor([[-0.2743, -0.0000, -0.0000, -0.0000],
        [-0.0000, -3.1530, -0.0000, -0.0000],
        [-0.2467, -0.0000, -0.0000, -0.0000],
        [-0.0000, -3.0793, -0.0000, -0.0000]], grad_fn=<MulBackward0>)
loss: 0.42208024859428406
```

Training FFNs using Mini-batches

Training set

1 epoch

$y_1 = cat$ $y_2 = dog$ $y_{64} = lion$ $y_* = spider$ $y_{*+63} = cow$ $y_{60000} = cat$

x_1 x_2 ... x_{32} ... x_* ... x_{*+31} ... ... x_{60000}

A mini-batch

One vector: [1, 5]
→ Batch: [32, 5]

Another mini-batch

Our FFN

32

5

Input Layer $h^0(x) = x$

Hidden Layer 1 $h^1(x)$

Hidden Layer 2 $h^2(x)$

Output Layer $h^3(x)$

$p(x) = softmax(h^3(x))$

$W^1 \in \mathbb{R}^{3\times4}$ $b^1 \in \mathbb{R}^{4\times1}$

$W^2 \in \mathbb{R}^{4\times3}$ $b^2 \in \mathbb{R}^{3\times1}$

$W^3 \in \mathbb{R}^{3\times2}$ $b^3 \in \mathbb{R}^{2\times1}$

$$batch\_loss(\theta) = \frac{1}{32}\sum_{i=1}^{32} CE(1_{y_i}, p(x_i, \theta))$$

Optimizer
- Adam,
- RMSProp,
- SGD

$\min_\theta batch\_loss(\theta)$

Update $\theta = [W^l, b^l]_{l=1}^L$

Testing set

Model parameters $\theta = [W^l, b^l]_{l=1}^L$

x'_1 x'_2 ..... x'_{1000}

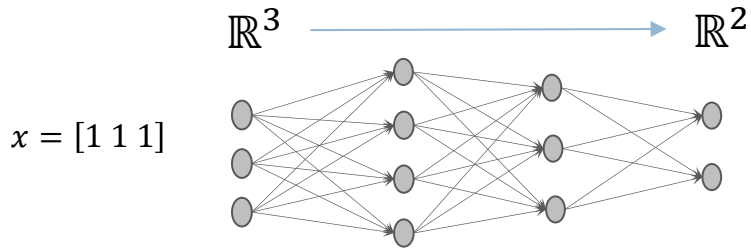
Test accuracy

Optimization in Deep Learning

A small detour to calculus

- Calculus = **mathematics of change** (very important for deep learning)
- Properties of derivative:
  - $f'(x) = \nabla f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
  - $(uv)' = u'v + uv'$
  - $\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$
  - $(e^u)' = u'e^u$
  - $(\log u)' = \frac{u'}{u}$
- Multi-variate function $f: \mathbb{R}^n \to \mathbb{R}$ with $y = f(x) = f(x_1, \ldots, x_n)$.
  - Gradient/derivative: $\frac{\partial f}{\partial x}(a) = \nabla_x f(a) = [\nabla_{x_1} f(a), \nabla_{x_2} f(a), \ldots, \nabla_{x_n} f(a)]$.
- Chain rule ⛓ :
  - $\frac{\partial u}{\partial x} = \frac{\partial u}{\partial v} \times \frac{\partial v}{\partial x}$

Derivative for multi-variate functions



- Given a function $f: \mathbb{R}^m \to \mathbb{R}^n$
  - $f(x) = (f_1(x), \dots, f_n(x))$ where $f_1, \dots, f_n: \mathbb{R}^m \to \mathbb{R}$ and $x = (x_1, \dots, x_m)$. Let denote $y = f(x)$.
  - The **derivative** of f at the **point** $\boldsymbol{a} \in \mathbb{R}^m$, denoted by $\nabla f(a)$ (**function related** notion) or $\frac{\partial y}{\partial x}(a)$ (**variable related** notion) is a matrix n by m (i.e., the Jacobian matrix).

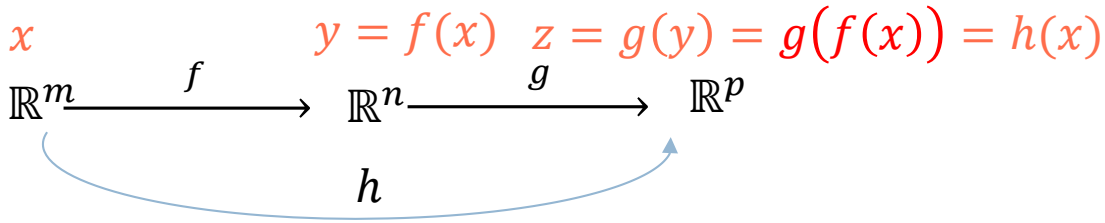
$$\frac{\partial y}{\partial x}(a) = \nabla f(a) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(a) & \dots & \dots & \frac{\partial f_1}{\partial x_j}(a) & \dots & \frac{\partial f_1}{\partial x_m}(a) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial f_i}{\partial x_1}(a) & \dots & \dots & \frac{\partial f_i}{\partial x_j}(a) & \dots & \frac{\partial f_i}{\partial x_m}(a) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial x_1}(a) & \dots & \dots & \frac{\partial f_n}{\partial x_j}(a) & \dots & \frac{\partial f_n}{\partial x_m}(a) \end{bmatrix} \begin{matrix} \\ \\ \\ n \\ \\ \\ \end{matrix}$$

m

Jacobian matrix

- **Chain rule**
  - Given a function $\boldsymbol{f}: \mathbb{R}^{\boldsymbol{m}} \to \mathbb{R}^{\boldsymbol{n}}$, $\mathbf{g}: \mathbb{R}^{\boldsymbol{n}} \to \mathbb{R}^{\boldsymbol{p}}$, denote $\boldsymbol{h} = \boldsymbol{g} \circ \boldsymbol{f}: \mathbb{R}^{\boldsymbol{m}} \to \mathbb{R}^{\boldsymbol{p}}$, meaning that $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$. We further define $y = f(x)$ and $z = g(y) = g(f(x)) = h(x)$.
  - For $x \in \mathbb{R}^m$, $\nabla h(x) = \nabla g(f(x)) \times \nabla f(x)$ or equivalently $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} . \frac{\partial y}{\partial x}$.

$p \times m \quad p \times n \quad n \times m$



Example

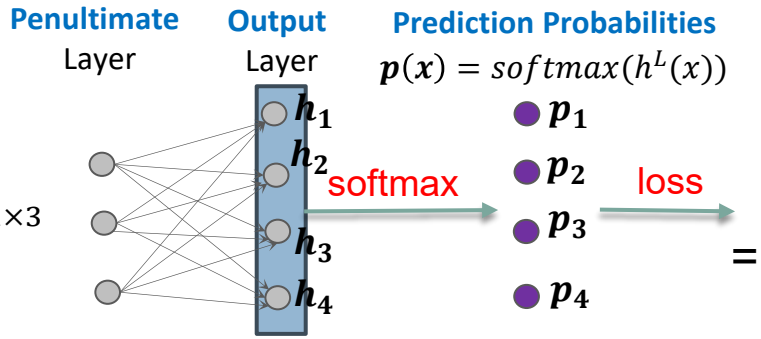
- $y = f(x) = f(x_1, x_2, x_3) = (x_1^2 + x_2^2, x_2^2 + x_3^2 x_2)$
  - $f: \mathbb{R}^3 \to \mathbb{R}^2$
  - $f_1(x) = f_1(x_1, x_2, x_3) = x_1^2 + x_2^2$
  - $f_2(x) = f_2(x_1, x_2, x_3) = x_2^2 + x_3^2 x_2$
  - $\frac{\partial y}{\partial x} = \nabla f \in \mathbb{R}^{2\times 3}$

$$\frac{\partial y}{\partial x} = \nabla_x f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 2x_1 & 2x_2 & 0 \\ 0 & 2x_2 + x_3^2 & 2x_2 x_3 \end{bmatrix}$$

Example

Output layer

$x = [x_1\ x_2\ x_2] \in \mathbb{R}^{1\times 3}$
$y = 2$



$l = loss = -\log p_2$

$= -\log \frac{e^{h_2}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}$

Logit $h = [h_1\ h_2\ h_3\ h_4]$ Prob p $= [p_1\ p_2\ p_3\ p_4]$

$$\boldsymbol{p_1} = \frac{\boldsymbol{e^{h_1}}}{\boldsymbol{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}}$$

$$\boldsymbol{p_2} = \frac{\boldsymbol{e^{h_2}}}{\boldsymbol{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}}$$

$$\boldsymbol{p_3} = \frac{\boldsymbol{e^{h_3}}}{\boldsymbol{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}}$$

$$\boldsymbol{p_4} = \frac{\boldsymbol{e^{h_4}}}{\boldsymbol{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}}$$

Compute $\frac{\partial l}{\partial \boldsymbol{h}}$?

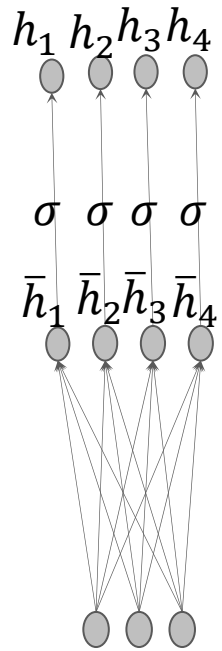
- $l = -\log \frac{e^{h_2}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = \log(\overbrace{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}^{u}) - h_2$
- $\frac{\partial l}{\partial h_1} = \frac{\nabla_{h_1} u}{u} = \frac{e^{h_1}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = p_1$
- $\frac{\partial l}{\partial h_2} = \frac{\nabla_{h_2} u}{u} - 1 = \frac{e^{h_2}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} - 1 = p_2 - 1$
- $\frac{\partial l}{\partial h_3} = \frac{\nabla_{h_3} u}{u} = \frac{e^{h_3}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = p_3$
- $\frac{\partial l}{\partial h_4} = \frac{\nabla_{h_4} u}{u} = \frac{e^{h_4}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = p_4$
- $\frac{\partial l}{\partial \boldsymbol{h}} = [p_1, p_2 - 1, p_3, p_4] = [p_1, p_2, p_3, p_4] - [0,1,0,0] = \boldsymbol{p - 1_2} = \boldsymbol{p - 1_y}$

Example
Intermediate layer

- $\bar{h} = xW + b$ and $h = \sigma(\bar{h})$
  - $h = \sigma(xW + b)$
  - σ is the **activation function**
- $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial \bar{h}} \times \frac{\partial \bar{h}}{\partial x} = diag\left(\sigma'(\bar{h})\right) W^T \in \mathbb{R}^{4\times 3}$

- $$\frac{\partial h}{\partial \bar{h}} = \begin{bmatrix} \frac{\partial h_1}{\partial \bar{h}_1} & \frac{\partial h_1}{\partial \bar{h}_2} & \frac{\partial h_1}{\partial \bar{h}_3} & \frac{\partial h_1}{\partial \bar{h}_4} \\ \frac{\partial h_2}{\partial \bar{h}_1} & \frac{\partial h_2}{\partial \bar{h}_2} & \frac{\partial h_2}{\partial \bar{h}_3} & \frac{\partial h_2}{\partial \bar{h}_4} \\ \frac{\partial h_3}{\partial \bar{h}_1} & \frac{\partial h_3}{\partial \bar{h}_2} & \frac{\partial h_3}{\partial \bar{h}_3} & \frac{\partial h_3}{\partial \bar{h}_4} \\ \frac{\partial h_4}{\partial \bar{h}_1} & \frac{\partial h_4}{\partial \bar{h}_2} & \frac{\partial h_4}{\partial \bar{h}_3} & \frac{\partial h_4}{\partial \bar{h}_4} \end{bmatrix} = \begin{bmatrix} \sigma'(\bar{h}_1) & 0 & 0 & 0 \\ 0 & \sigma'(\bar{h}_2) & 0 & 0 \\ 0 & 0 & \sigma'(\bar{h}_3) & 0 \\ 0 & 0 & 0 & \sigma'(\bar{h}_4) \end{bmatrix} = diag(\sigma'(\bar{h}))$$

- $\frac{\partial \bar{h}}{\partial x} = W^T$



$h = \sigma(\bar{h})$

$\bar{h} = xW + b$

$W \in \mathbb{R}^{3\times 4}$

$b \in \mathbb{R}^{1\times 4}$

$x \in \mathbb{R}^{1\times 3}$

How to code with PyTorch

- $\bar{h} = xW + b$ and $h = sigmoid(\bar{h})$
  - $h = sigmoid(xW + b)$
  - $\sigma = sigmoid$ is the activation function
- $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial \bar{h}} \times \frac{\partial \bar{h}}{\partial x} = diag\left(\sigma'(\bar{h})\right) W^T = diag(\sigma(\bar{h}) \otimes [1 - \sigma(\bar{h})]) W^T = diag(h \otimes (1 - h)) W^T$
  - $\otimes$ is element-wise product

```
x = torch.tensor([1,-1,1], dtype=torch.float32)
print(x)
W = torch.rand(3,4)
b = torch.rand(1,4)
print(W)
print(b)
```

```
tensor([ 1., -1.,  1.])
tensor([[0.7266, 0.7925, 0.7952, 0.2159],
        [0.3108, 0.1950, 0.6448, 0.6367],
        [0.0521, 0.2200, 0.5038, 0.9020]])
tensor([[0.3059, 0.2612, 0.9326, 0.5597]])
```

Declare W, x, b

```
hbar = x@W+b
print(hbar)
```

```
tensor([[0.7737, 1.0787, 1.5868, 1.0409]])
```

```
h = torch.nn.Sigmoid()(hbar)
print(h)
```

```
tensor([[0.6843, 0.7463, 0.8302, 0.7390]])
```

Forward propagation

```
v = h*(1-h)
v = v.squeeze()
print(v)
```

```
tensor([0.2160, 0.1894, 0.1410, 0.1929])
```

```
A = torch.diag(v)
print(A)
```

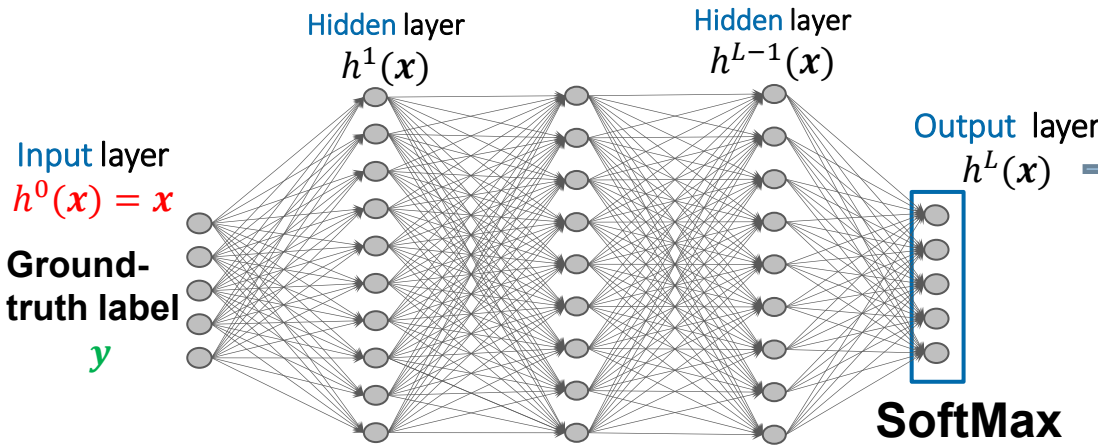
```
tensor([[0.2160, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.1894, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.1410, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.1929]])
```

```
derivative = A@W.T
print(derivative)
```

```
tensor([[0.1570, 0.0671, 0.0113],
        [0.1501, 0.0369, 0.0417],
        [0.1121, 0.0909, 0.0710],
        [0.0416, 0.1228, 0.1740]])
```

Backward propagation

Recall optimization problem in deep learning



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

$$\hat{y} = \text{argmax}_{1 \le m \le M} p_m(x)$$

$$l(y, \hat{y}) = CE\left(1_y, p(x)\right) = -\log p_y(x)$$

Training set

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

Loss function

$$L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} CE\left(1_{y_i}, p(x_i)\right) = -\frac{1}{N}\sum_{i=1}^{N} \log p_{y_i}(x_i)$$

- How to **solve** the **optimization problem** efficiently ($\theta \coloneqq \{(W^l, b^l)\}_{l=1}^{L}$)?
  - $\min_\theta L(D;\theta) \coloneqq -\frac{1}{N}\sum_{i=1}^{N} \log p_{y_i}(x_i) = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp\{h_{y_i}^L(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^L(x_i)\}}$
  - Generalize: $\min_\theta J(\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} l(f(x_i;\theta), y_i)$

Optimization problem in ML and DL

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

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

Regularization term

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

Empirical loss

- Work well on training set

- **Occam's Razor principle**: prefer simplest model that can well predict data.

How to efficiently solve this optimization problem?

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

First-order iterative methods (gradient descent, steepest descent)

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

Second-order iterative methods (Newton and quasi Newton methods)

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

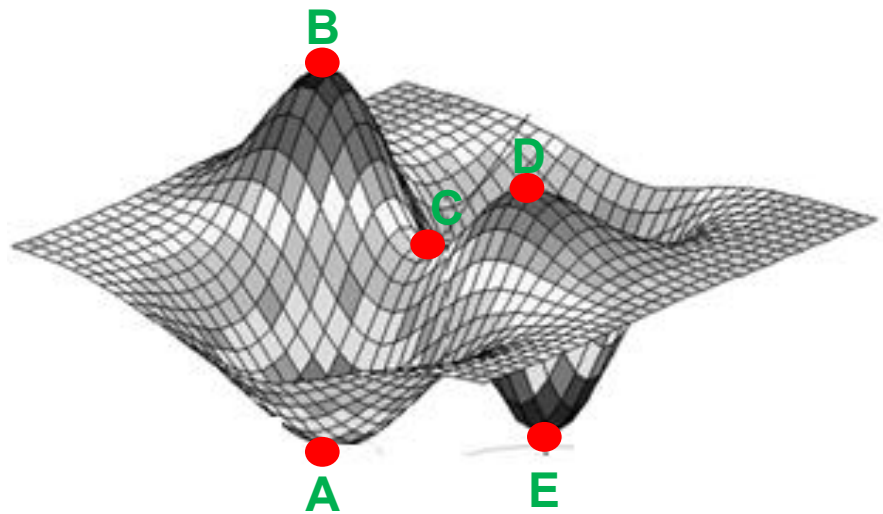
Gradient and Hessian matrix

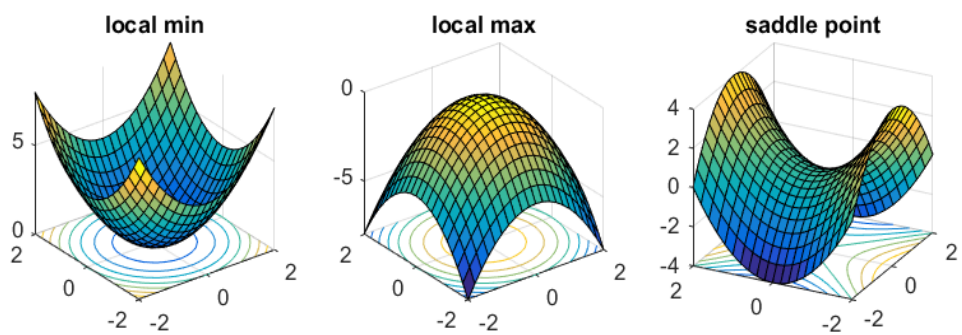
- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - For DL models
    - θ includes **weight matrices**, **filters**, and **biases** which are trainable model parameters.
    - P is the number of **trainable parameters** (P could be 20×10^6).
    - $J(\theta)$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

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

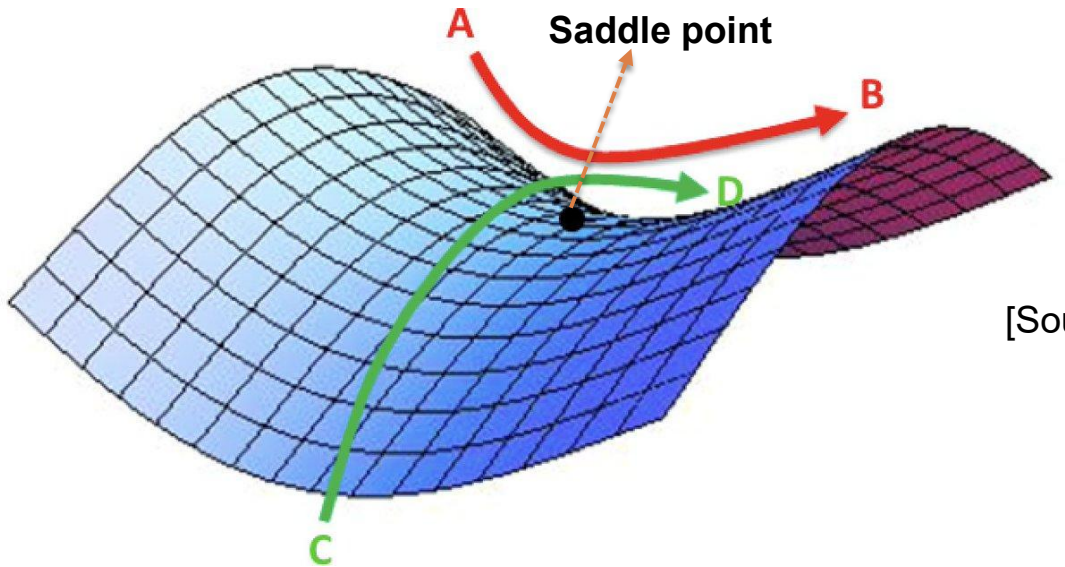
Local minima-maxima and saddle point

- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - θ 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 semi-definite matrix)
  - $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P \leq 0$
- **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]

$$f(\theta) = f(\theta_1, \theta_2) = \theta_1^2 - \theta_2^2$$

Gradient $g = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \end{bmatrix} = \begin{bmatrix} 2\theta_1 \\ -2\theta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ → a **critical point** $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

Hessian matrix is $H = \begin{bmatrix} \frac{\partial^2 f}{\partial \theta_1^2} & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} \\ \frac{\partial^2 f}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f}{\partial \theta_2^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$

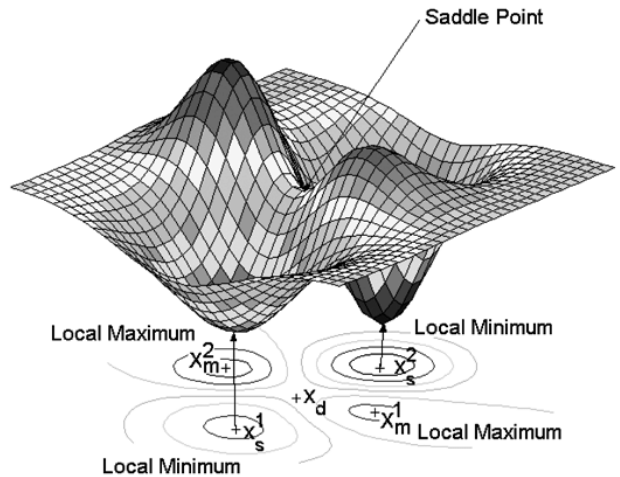
Two eigenvalues $\lambda_1 = -2 < 0 < 2 = \lambda_2$ → $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ is a **saddle point**.

Numbers of local minima vs saddle points

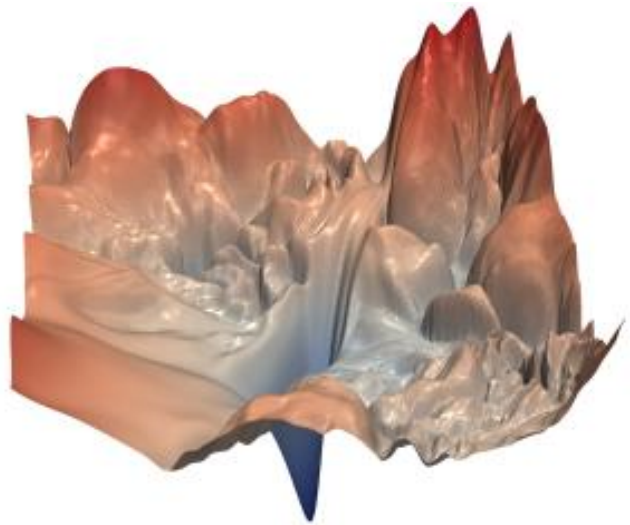
- We assume to pick randomly a training set
  - The Hessian matrix $H(\theta)$ is a random matrix with **random eigenvalues** $\lambda_1, \lambda_2, \dots, \lambda_P$
  - We assume that $\mathbb{P}(\lambda_1 \geq 0) = \mathbb{P}(\lambda_2 \geq 0) = \cdots = \mathbb{P}(\lambda_P \geq 0) = 0.5$

- Therefore, we have
  - $\mathbb{P}(minima) = \mathbb{P}(\lambda_1 \geq 0)\mathbb{P}(\lambda_2 \geq 0) \dots \mathbb{P}(\lambda_P \geq 0) = 0.5^P$
  - $\mathbb{P}(maxima) = \mathbb{P}(\lambda_1 \leq 0)\mathbb{P}(\lambda_2 \leq 0) \dots \mathbb{P}(\lambda_P \leq 0) = 0.5^P$
  - $\mathbb{P}(saddle\ point) = 1 - \mathbb{P}(minima) - \mathbb{P}(maxima) = 1 - 0.5^{P-1}$

- 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**



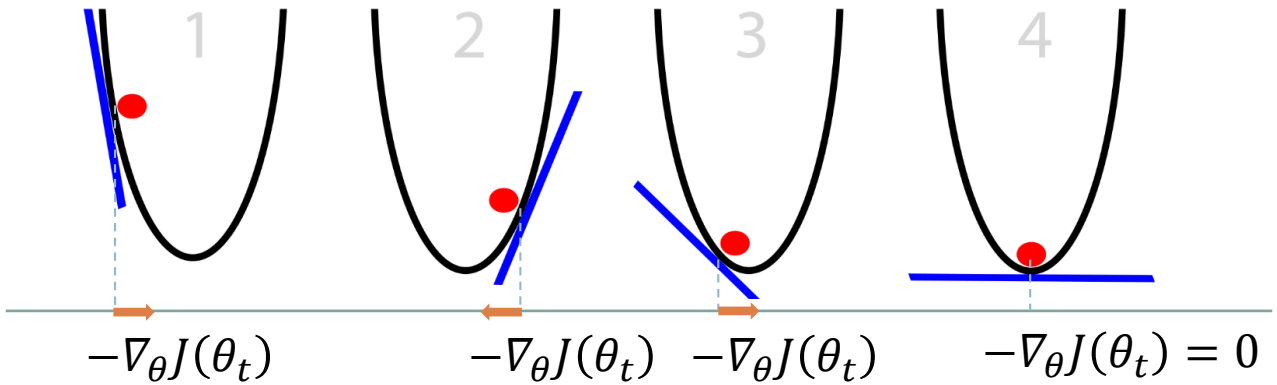
The loss surface of DL optimization problem



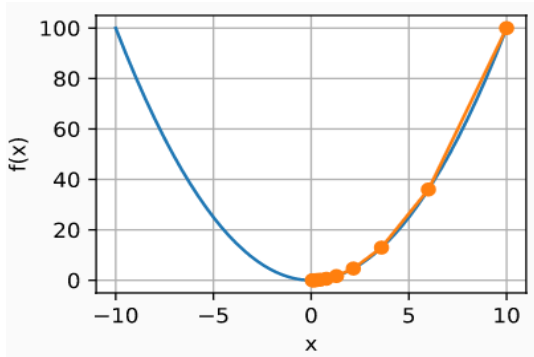
Loss surface of a ResNet without skip connection [Hao Li et al., NeurIPS 2017]

- 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**

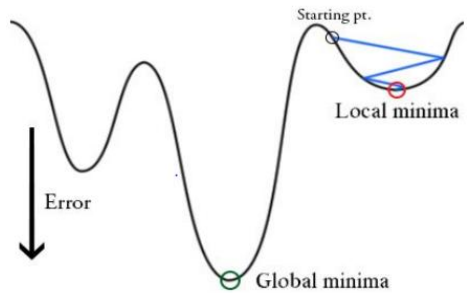
Gradient descend



- We need to solve
  - $\min_\theta J(\theta)$
- Follow to **the opposite side** of the current gradient
  - $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$ where $\eta > 0$ is the **learning rate**.
- Guarantee to converge to a **global minima** if $J(.)$ is **convex**.
- Get stuck in a **local minima** or **saddle points** if $J(.)$ is non-convex.

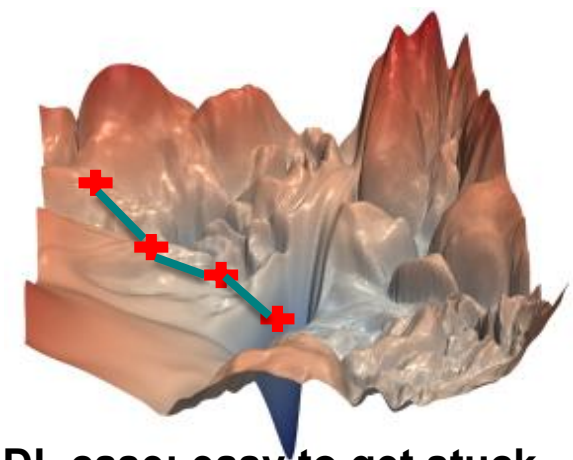


Convex case



(Source: www.cs.ubc.ca)

Non-convex case



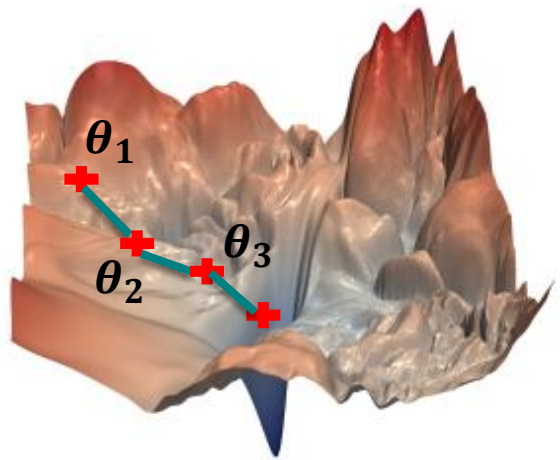
DL case: easy to get stuck in saddle points

Gradient descend

Algorithm

- **Input**: objective function $\boldsymbol{J(\theta)}$
- **Output**: optimal solution $\boldsymbol{\theta^*}$

1. Initialize parameters θ_0 randomly $\sim N(0, \sigma^2)$.
2. for t=1 to T
3. Compute gradients $\nabla_\theta J(\theta_t) = \frac{\partial J}{\partial \theta}(\theta_t)$
4. Update $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta J(\theta_t)$
5. Return $\theta^* = \theta_{T+1}$

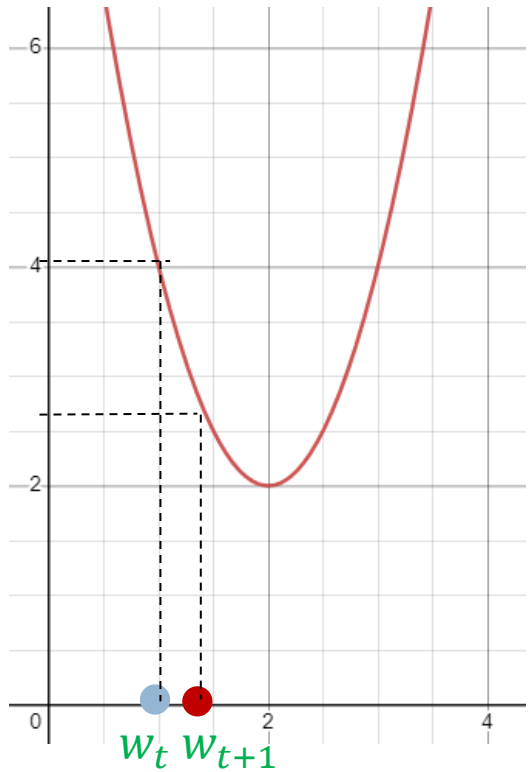


Example of Gradient Descend

- Consider the optimization problem:

$$\min_{w}[f(w) = (w-1)^2 + (w-3)^2]$$

- Currently, we are at $w_t = 1$ with $f(w_t) = f(1) = 0^2 + 2^2 = 4$. What is w_{t+1} if using learning rate $\eta = 0.1$?
  - $f'(w) = 2(w-1) + 2(w-3) = 4w - 8$
  - $f'(w_t) = f'(1) = -4$
  - $w_{t+1} = w_t - \eta f'(w_t) = 1 - 0.1(-4) = 1.4$
  - $f(w_{t+1}) = 2.72 < f(w_t) = 4$



Gradient descent for deep learning

- For **training deep nets**, we need to solve
  - $\min_{\theta} L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))$

  where $l(x_i, y_i; \theta) = -\log p(y = y_i|x_i) = -\log \frac{\exp\{h_{y_i}^L(x_i)\}}{\sum_{m=1}^{M}\exp\{h_m^L(x_i)\}}$ is the loss incurred by (x_i, y_i).
- Gradient descent update
  - $\theta_{t+1} = \theta_t - \eta\nabla_\theta L(D;\theta_t) = \theta_t - \frac{\eta}{N}\sum_{i=1}^{N}\nabla_\theta\, l(x_i, y_i; \theta_t)$ where $\boldsymbol{\eta} > \mathbf{0}$ is a learning rate.
  - To compute the gradient $\nabla_\theta L(D;\theta_t) = \frac{1}{N}\sum_{i=1}^{N}\nabla_\theta\, l(x_i, y_i; \theta_t)$, we need to go through **all data points** in D → the **computational cost** is $O(N)$.
- This is very **computationally expensive** for big datasets ($N \approx 10^6$).
- How to **estimate the gradient** $\nabla_\theta L(D;\theta_t)$ more efficiently?

Stochastic gradient descent

- The **optimization problem** in **deep learning** has the form
  - $\min_{\theta} L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta) \ = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))$
- Evaluation of the **full gradient** is **expensive**. We want to just **estimate** this gradient
  - Sample a mini-batch $\boldsymbol{i_1, i_2, \dots, i_b \sim Uni(\{1, 2, \dots, N\})}$ where b is the mini-batch (batch) size.
    - The batch size is usually $32,64,128,256$, and so on.
  - Construct $\tilde{L}(\theta) := \frac{1}{b}\sum_{k=1}^{b} l(x_{i_k}, y_{i_k}; \theta)$ as the average loss of those in the current batch.
  - $E_{i_1,\dots,i_b}[\nabla_\theta \tilde{L}(\theta_t)] = \nabla_\theta L(D;\theta_t)$
    - $\nabla_\theta \tilde{L}(\theta_t) = \frac{1}{b}\sum_{k=1}^{b} \nabla_\theta l(x_{i_k}, y_{i_k}; \theta_t)$ is **unbiased** estimation of $\nabla_\theta L(D;\theta_t)$
    - $O(b)$ compares to $O(N)$.
- The update rule of SGD
  - $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \tilde{L}(\theta_t)$ with learning rate $\boldsymbol{\eta_t \propto O(\frac{1}{t})}$
  - We use $\nabla_\theta \tilde{L}(\theta_t)$ as an **unbiased estimate** of the full gradient $\nabla_\theta L(D;\theta)$
  - How to compute $\nabla_\theta \tilde{L}(\theta_t)$ efficiently for **deep networks**?

Example of Stochastic Gradient Descent

- Given the function $f(w) = \frac{1}{1000}\sum_{i=1}^{1000}(w-i)^2$, we need to solve

$$\min_w f(w)$$

using SGD with the learning rate $\eta = 0.1$.

- Assume we sample a batch $i_1 = 1, i_2 = 2, i_3 = 3, i_4 = 4$ of indices. At the iteration t, $w_t = 10$. What is the value of w_{t+1} at the next iteration?

- $\tilde{f}(w) = \frac{1}{4}[(w-1)^2 + (w-2)^2 + (w-3)^4 + (w-4)^2]$
- $\tilde{f}'(w) = 2w - 5$
- $\tilde{f}'(w_t) = \tilde{f}'(10) = 2 \times 10 - 5 = 15$
- $w_{t+1} = w_t - \eta\tilde{f}'(w_t) = 10 - 0.1 \times 15 = 8.5.$

SGD for deep learning

b = 32 //batch size

iter_per_epoch = N/b //epoch means one round going through all data points

n_epoch = 50 //number of epochs

for epoch=1 to n_epoch do

  for i=1 to iter_per_epoch do

    Sample a minibatch $B = \left\{\left(x_{i_j}, y_{i_j}\right)\right\}_{j=1}^{b}$ from the training set

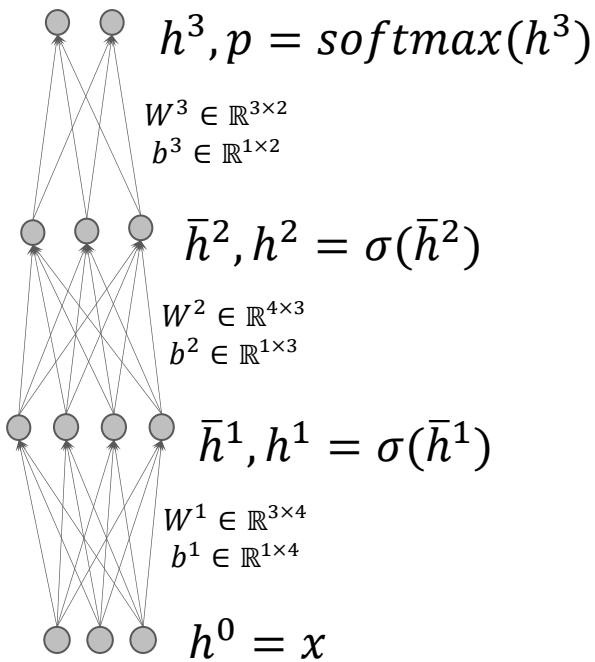
    Do forward propagation for B

    Do back propagation to compute $\left(\frac{\partial l}{\partial W^k}, \frac{\partial l}{\partial b^k}\right)_{k=1}^{L}$

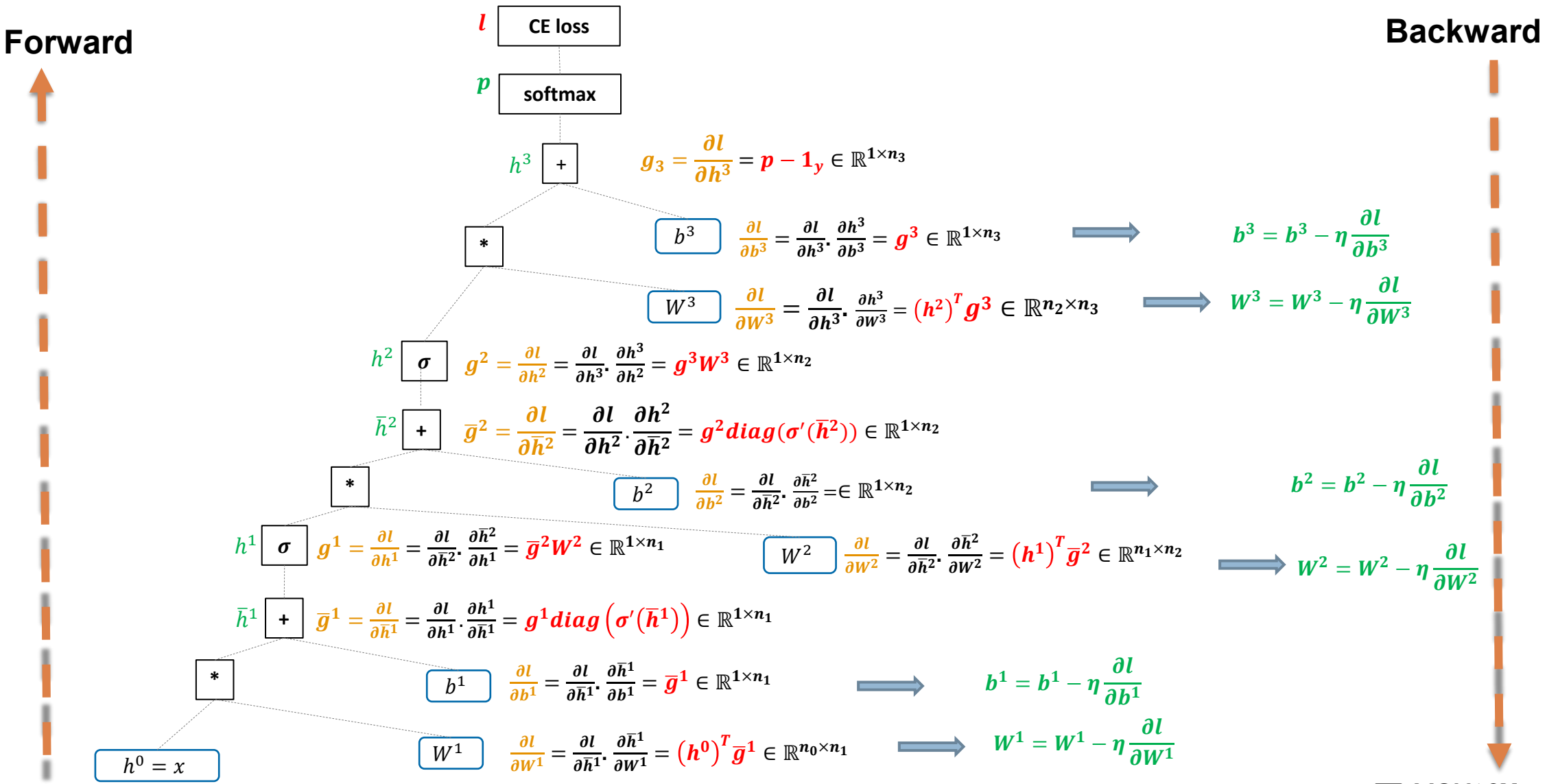
    for k=1 to L do

$$W_k = W_k - \eta \frac{\partial l}{\partial W^k}$$

$$b_k = b_k - \eta \frac{\partial l}{\partial b^k}$$

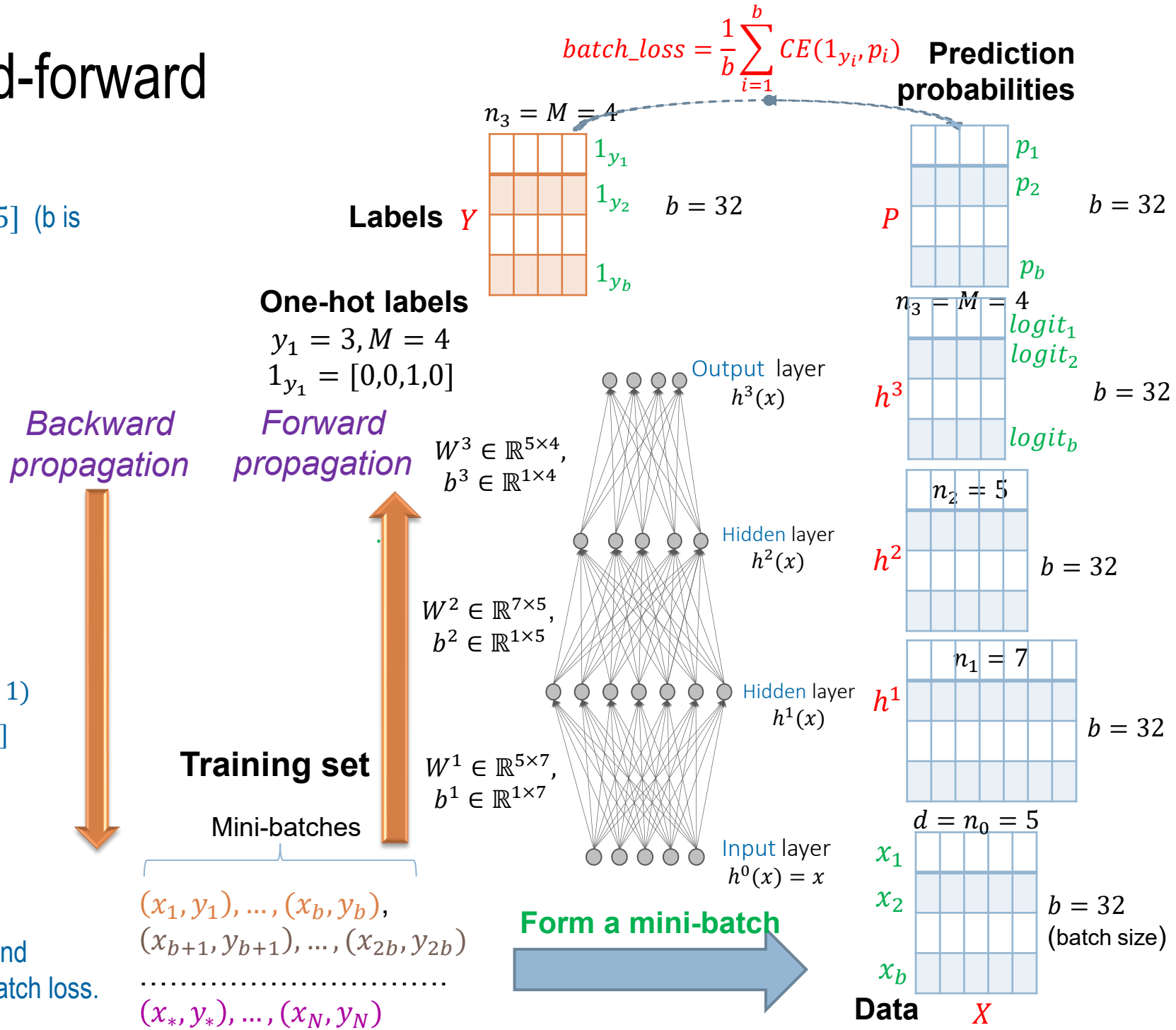


Forward – Backward Propagations



Mini-batch feed-forward

- **Input**
  - Batch X: $[b, n_0 = d = 5]$ (b is the batch size)
- **Hidden layer 1**
  - $\boldsymbol{h^1 = \sigma(XW^1 + b^1)}$
  - $Tensor\,[b, n_1 = 7]$
- **Hidden layer 2**
  - $\boldsymbol{h^2 = \sigma(h^1W^2 + b^2)}$
  - Tensor $[b, n_2 = 7]$
- **Output layer**
  - $\boldsymbol{h^3 = h^2W^3 + b^3}$
  - $P = softmax(h^3, dim = 1)$
  - $Tensor\,\mathrm{P}$: $[b, n_L = M = 4]$
- **The loss of the batch**
  - $\frac{1}{b}\sum_{i=1}^{b} CE(1_{y_i}, p_i) = -\frac{1}{b}\sum_{i=1}^{b} \log p^i_{y_i}$
  - Update weight matrices and biases to **minimize** the batch loss.



SGD and SGD with momentum

SGD

- **Input**: $\eta > 0$ and *initial model* θ

while *stopping criterion* **not met do**

Sample a mini-batch $\{(\boldsymbol{x^1}, \boldsymbol{y^1}), \ldots, (\boldsymbol{x^b}, \boldsymbol{y^b})\}$

Compute $\boldsymbol{g} = \frac{\boldsymbol{1}}{\boldsymbol{b}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{b}} \boldsymbol{\nabla_\theta l}(\boldsymbol{f}(\boldsymbol{x^i}, \boldsymbol{\theta}), \boldsymbol{y^i})$

Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\eta g}$

end while

SGD with momentum

- **Input**: $\eta > 0, \alpha \in [0,1]$ and *initial model* θ

while *stopping criterion* **not met do**

Sample a mini-batch $\{(\boldsymbol{x^1}, \boldsymbol{y^1}), \ldots, (\boldsymbol{x^b}, \boldsymbol{y^b})\}$

Compute $\boldsymbol{g} = \frac{\boldsymbol{1}}{\boldsymbol{b}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{b}} \boldsymbol{\nabla_\theta l}(\boldsymbol{f}(\boldsymbol{x^i}, \boldsymbol{\theta}), \boldsymbol{y^i})$

Compute $\boldsymbol{v} = \boldsymbol{\alpha g} + (\boldsymbol{1} - \boldsymbol{\alpha})\boldsymbol{v}$ //velocity v

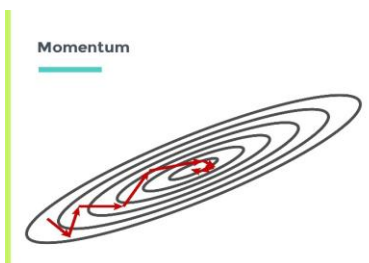
Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\eta v}$

end while

- SGD uses only the **gradient of the mini-batch** to update the model
- It is fast at first several epochs and becomes **much slower** later.

- SGD with momentum uses a **velocity vector** $\boldsymbol{v}$ which **stores the past gradients** together with the **current gradient** to speed up SGD
  - α is a hyper-parameter that indicates how quickly the contributions of previous gradients. In practice, this is usually set to 0.5, 0.9, and 0.99.
  - The momentum primarily solves 2 problems: **poor conditioning** of the Hessian matrix and **variance** in the stochastic gradient.

(Source: Sebastian Ruder)



AdaGrad

AdaGrad

- **Input**: $\eta > 0, \epsilon > 0\ (10^{-6})$, and *initial model* θ

while *stopping criterion* not met do

Sample a mini-batch $\{(\boldsymbol{x^1}, \boldsymbol{y^1}), \ldots, (\boldsymbol{x^b}, \boldsymbol{y^b})\}$

Compute $\boldsymbol{g} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\theta} \boldsymbol{l}(\boldsymbol{f}(\boldsymbol{x^i}, \boldsymbol{\theta}), \boldsymbol{y^i})$

Accumulate the square gradient: $\boldsymbol{\gamma} = \boldsymbol{\gamma} + \boldsymbol{g} \odot \boldsymbol{g}$

Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{\epsilon}+\boldsymbol{\gamma}}} \odot \boldsymbol{g}$

end while

Note: $\odot$ means *element-wise* product

$\boldsymbol{g^1}$	g_1^1	g_2^1	…	g_P^1
$\boldsymbol{g^2}$	g_1^2	g_2^2	…	g_P^2
…	…	…	…	…
$\boldsymbol{g^t}$	g_1^t	g_2^t	…	g_P^t
$\boldsymbol{\gamma}$	$\sum_{i=1}^{t}(g_1^i)^2$	$\sum_{i=1}^{t}(g_2^i)^2$	…	$\sum_{i=1}^{t}(g_P^i)^2$

- Learning rates are scaled by the square root of the cumulative sum of squared gradients
- Direction with large partial derivatives
  - Thus, rapid decrease in their learning rates
- Direction with small partial derivatives
  - Hence relatively small decrease in their learning rates
- Weakness: always decrease the learning rate!
  - Excellent for convex problem, but not so good for DL (with non-convex problems)

Convolutional Neural Network

Convolution operation

❖ W and x are **two tensors** with the **same shape**, the **convolutional operation** between W and x defined as

$$W * x = sum(W \otimes x)$$

where $W \otimes x$ specifies the **element-wise product** and sum returns the **sum of all elements** in a tensor and multi-dimensional array.

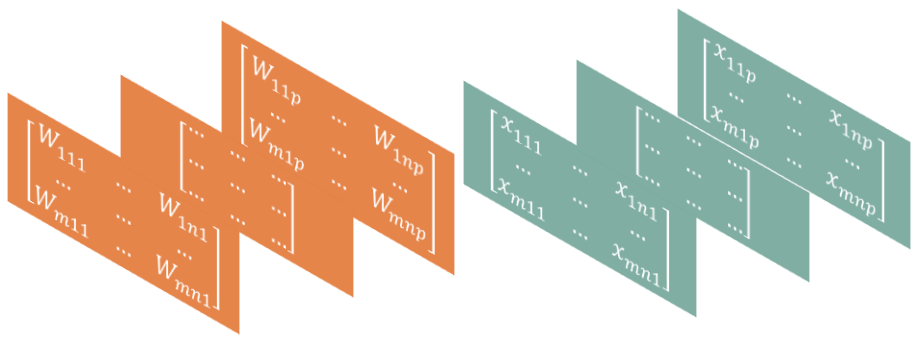
- **1D convolutional operation** $(W, x \in \mathbb{R}^{m \times 1})$
  - $W * x = \sum_{i=1}^{m} W_i x_i$

$$W = \begin{bmatrix} W_1 \\ \dots \\ W_m \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \dots \\ x_m \end{bmatrix}$$

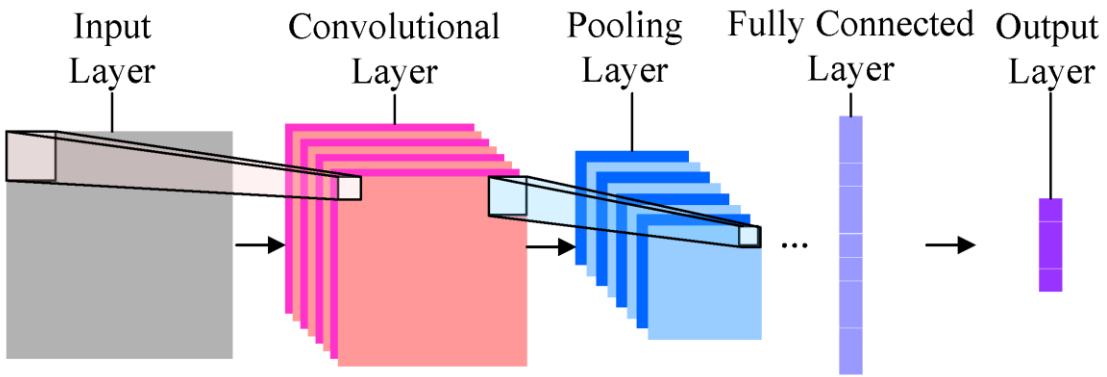
- **2D convolutional operation** $(W, x \in \mathbb{R}^{m \times n})$
  - $W * x = \sum_{i=1}^{m} \sum_{j=1}^{n} W_{ij} x_{ij}$

$$W = \begin{bmatrix} W_{11} & \dots & W_{1n} \\ \dots & \dots & \dots \\ W_{m1} & \dots & W_{mn} \end{bmatrix} \quad x = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \dots & \dots & \dots \\ x_{m1} & \dots & x_{mn} \end{bmatrix}$$

- **3D convolutional operation** $(W, x \in \mathbb{R}^{m \times n \times p})$
  - $W * x = \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{p} W_{ijk} x_{ijk}$

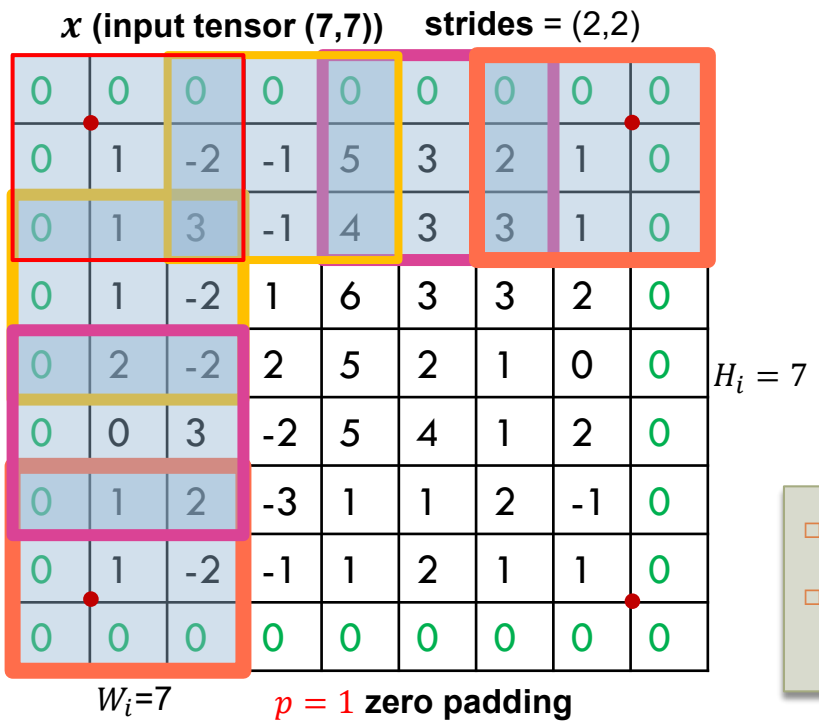


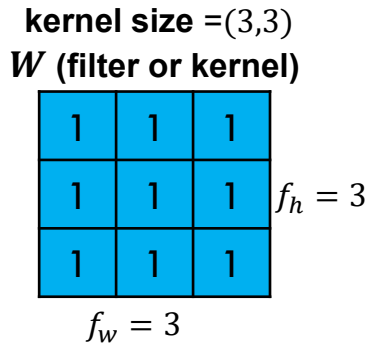
General architecture of CNNs

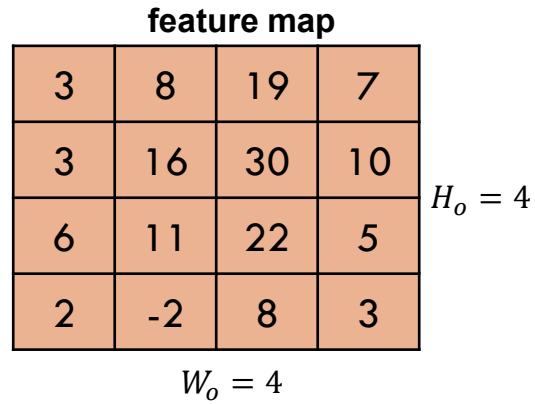


1. Convolutional layer
2. Pooling layer
3. Fully connected layer

Convolution layer with zero padding







- The **sliding window** moves from **left → right**, **top → bottom** with strides.
- We **convolve** the **filter** and the **sliding windows** to work out the **neurons** on the **feature map**.

Convolution layer with zero padding

x (input tensor (7,7)) **strides** = (2,2)

0	0	0	0	0	0	0	0	0
0	1	-2	-1	5	3	2	1	0
0	1	3	-1	4	3	3	1	0
0	1	-2	1	6	3	3	2	0
0	2	-2	2	5	2	1	0	0
0	0	3	-2	5	4	1	2	0
0	1	2	-3	1	1	2	-1	0
0	1	-2	-1	1	2	1	1	0
0	0	0	0	0	0	0	0	0

$H_i = 7$

W_i=7

$p = 1$ **zero padding**

kernel size =(3,3)

W (filter or kernel)

1	1	1
1	1	1
1	1	1

$f_h = 3$

$f_w = 3$

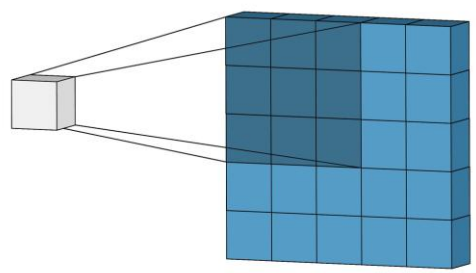
feature map

3	8	19	7
3	15	30	10
6	11	22	5
2	-2	9	3

$H_o = 4$

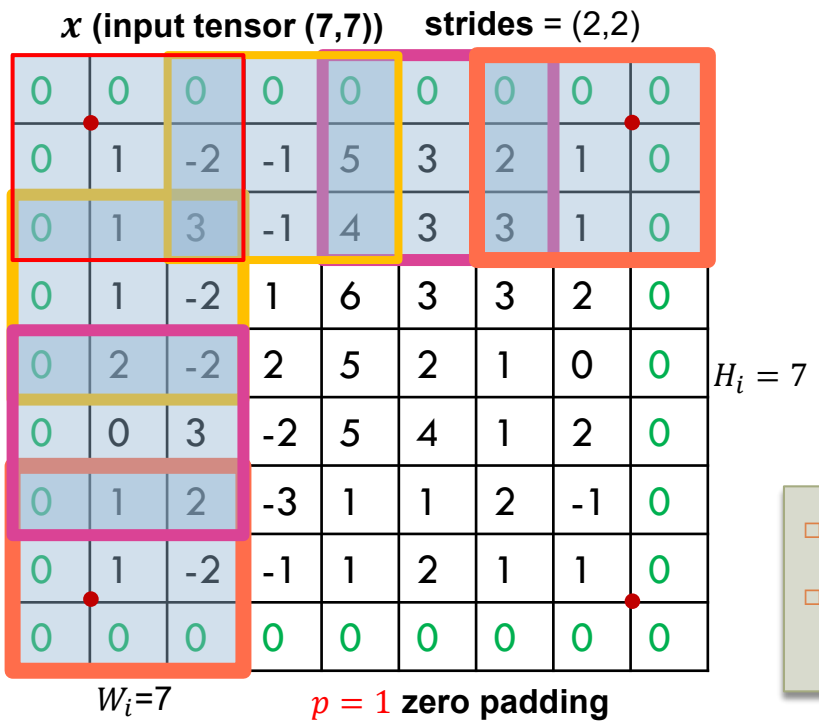
$W_o = 4$

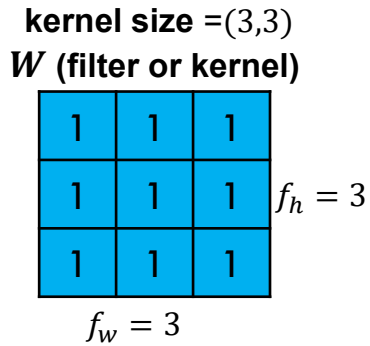
- The **sliding window** moves from **left → right**, **top → bottom** with strides.
- We **convolve** the **filter** and the **sliding windows** to work out the **neurons** on the **feature map**.

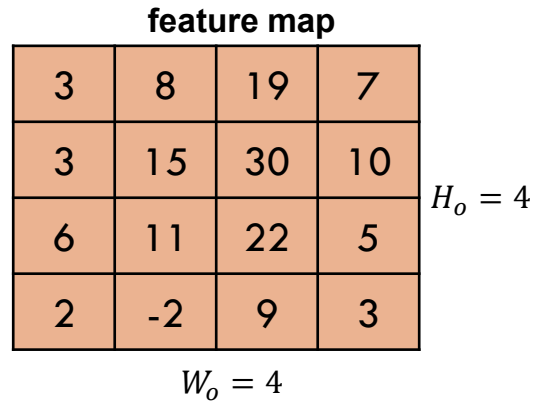


(Source: https://towardsdatascience.com/)

Convolution layer







- The **sliding window** moves from **left** → **right**, **top** → **bottom** with strides.
- We **convolve** the **filter** and the **sliding windows** to work out the **neurons** on the **feature map**.

- W_i, H_i: The width and height of the **input** image
- W_o, H_o: The width and height of the **output** image (feature map)

$$W_o = \left\lfloor \frac{W_i + 2p - f_w}{s_w} \right\rfloor + 1 \text{ and } H_o = \left\lfloor \frac{H_i + 2p - f_h}{s_h} \right\rfloor + 1$$

- Our case: $W_o = \left\lfloor \frac{7+2\times1-3}{2} \right\rfloor + 1 = 4$ and $H_o = \left\lfloor \frac{7+2\times1-3}{2} \right\rfloor + 1 = 4$

Convolution layer

x (input tensor (7,8)) **strides** = (2,2)

1	-2	-1	5	3	2	1	1
1	3	-1	4	3	3	1	-1
1	-2	1	6	3	3	2	2
2	-2	2	5	2	1	0	-1
0	3	-2	5	4	1	2	1
1	2	-3	1	1	2	-1	2
1	-2	-1	1	2	1	1	3

$H_i = 7$

$W_i = 8$ **p=0 zero padding**

kernel size =(3,3)

W **(filter or kernel)**

1	1	1
1	1	1
1	1	1

$f_h = 3$

$f_w = 3$

feature map

1	23	21
3	26	18
-1	8	13

$H_o = 3$

$W_o = 3$

Convolution layer

x **(input tensor (7,8))** **strides** = (2,2)

1	-2	-1	5	3	2	1	1
1	3	-1	4	3	3	1	-1
1	-2	1	6	3	3	2	2
2	-2	2	5	2	1	0	-1
0	3	-2	5	4	1	2	1
1	2	-3	1	1	2	-1	2
1	-2	-1	1	2	1	1	3

$H_i = 7$

$W_i = 8$ **p=0 zero padding**

kernel size =(3,3)

W **(filter or kernel)**

1	1	1
1	1	1
1	1	1

$f_h = 3$

$f_w = 3$

feature map

1	23	21
3	26	18
-1	8	13

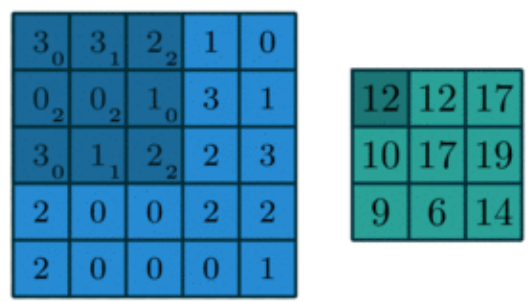
$H_o = 3$

$W_o = 3$

- W_i, H_i: The width and height of the **input** image
- W_o, H_o: The width and height of the **output** image (feature map)

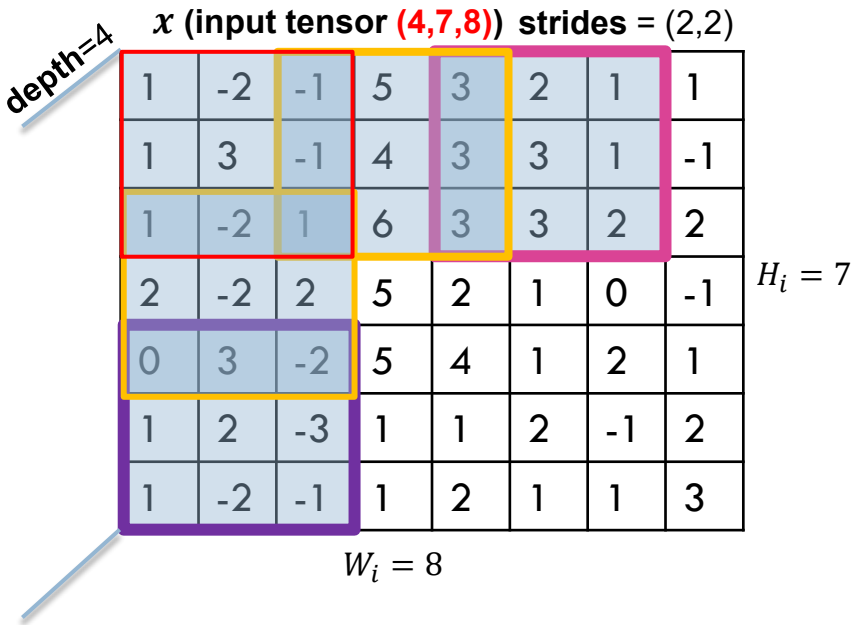
$$W_o = \left\lfloor \frac{W_i + 2p - f_w}{s_w} \right\rfloor + 1 \text{ and } H_o = \left\lfloor \frac{H_i + 2p - f_h}{s_h} \right\rfloor + 1$$

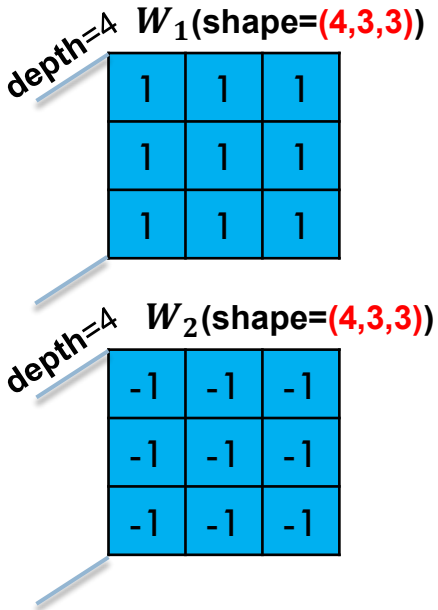
- Our case: $W_o = \left\lfloor \frac{8+0-3}{2} \right\rfloor + 1 = 3$ and $H_o = \left\lfloor \frac{7+0-3}{2} \right\rfloor + 1 = 3$

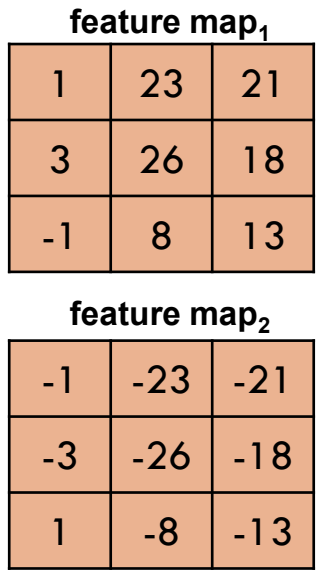


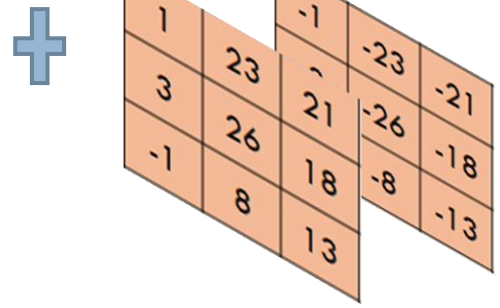
(Source: Internet)

Convolution layer with multiple filters and feature maps

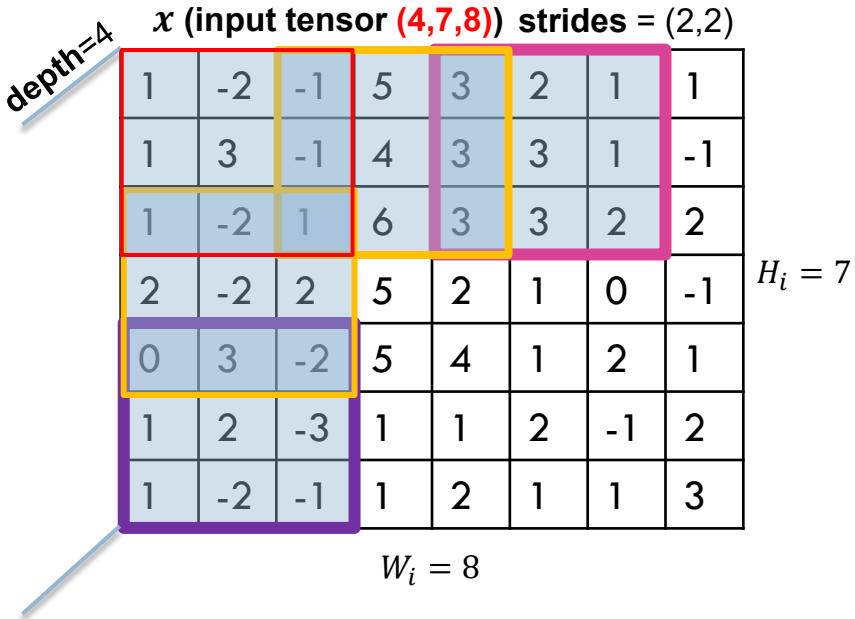


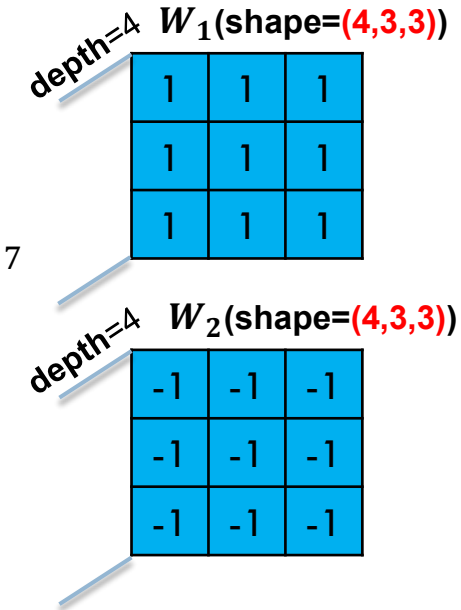


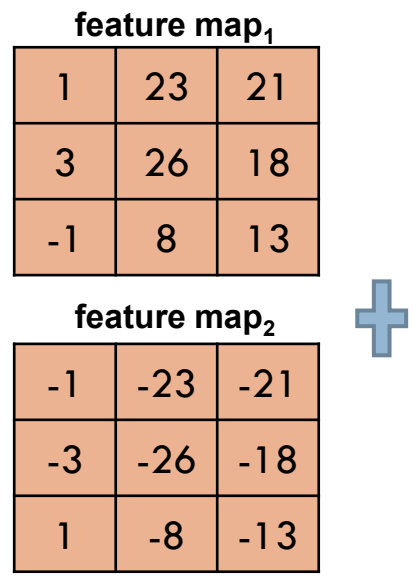


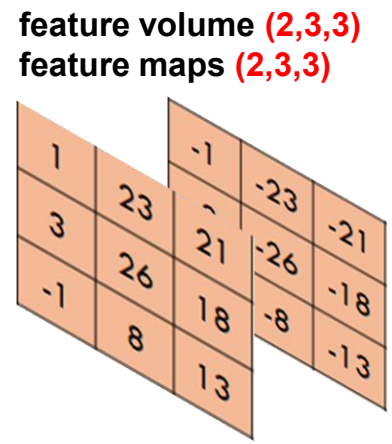


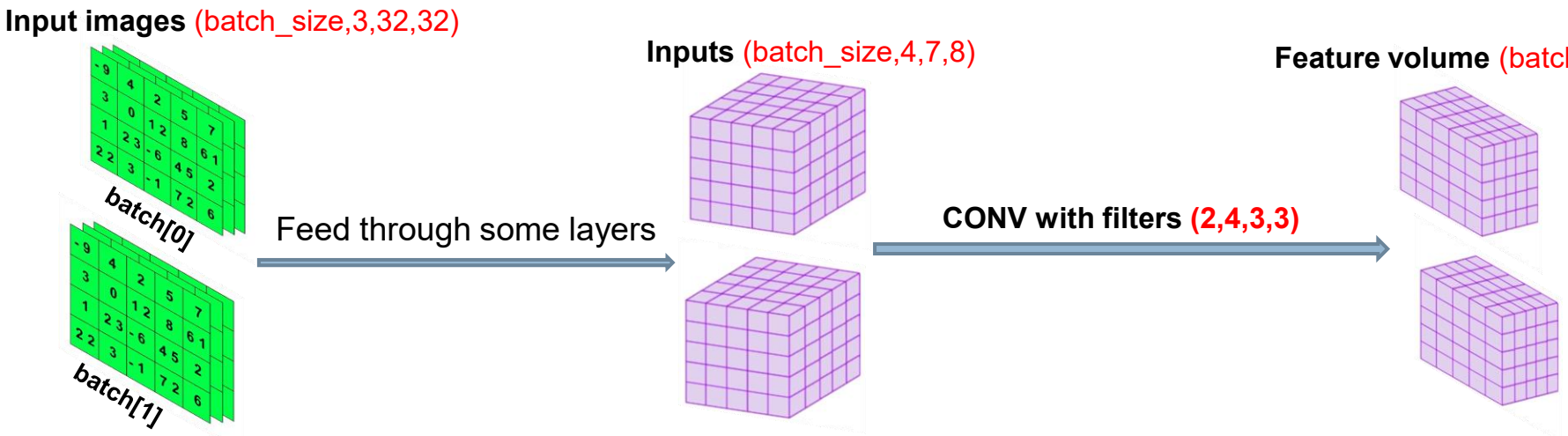
Convolution layer with multiple filters and feature maps












```
output = torch.nn.functional.conv2d(input = batch_tensor, weight= filters_tensor, stride =(2,2), padding= 3)
```

batch_size, in_channels , height, width,

out_channels, in_channels, fil_height, fil_width
,

Pooling operation

- Makes the representations smaller and more manageable
- Subsample the image
- Operates over each feature map independently

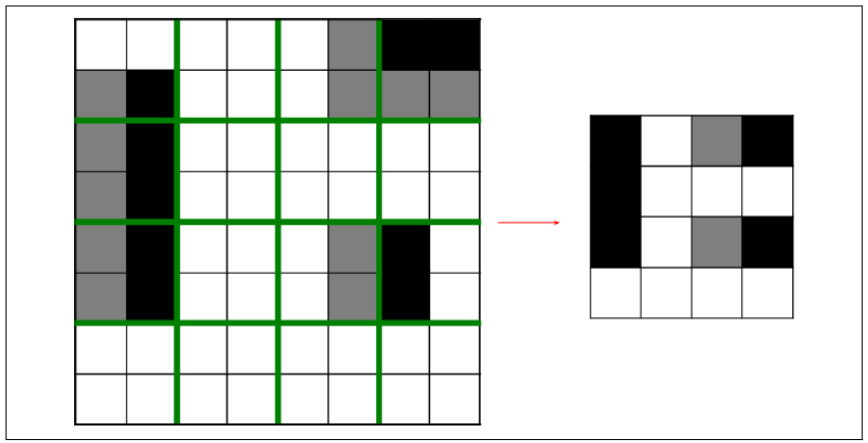


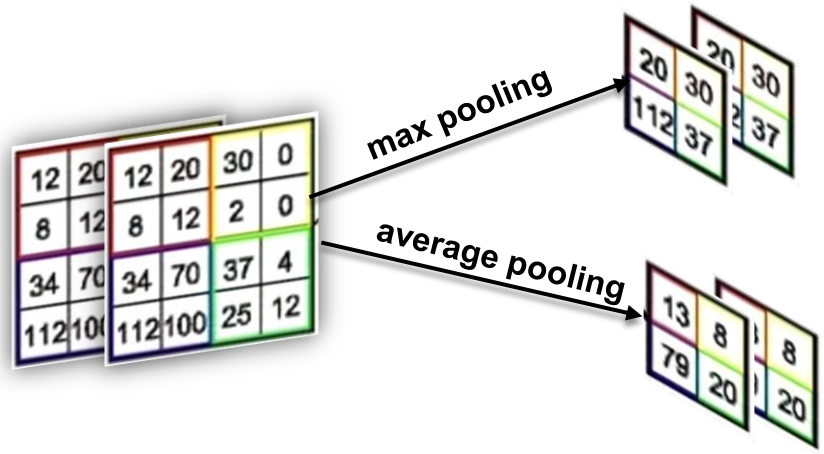
Figure 5-13. An illustration of how max pooling significantly reduces parameters as we move up the network

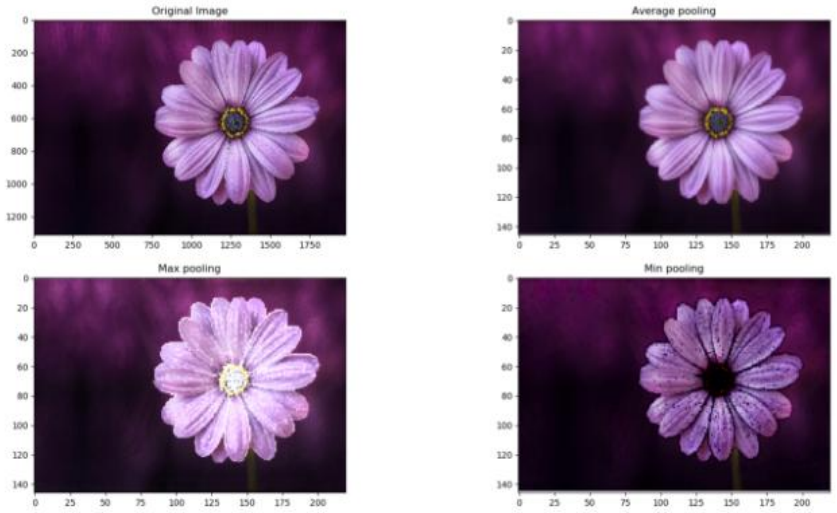
(Source: FDL Ch. 5)

Max/average pooling

- Max/average-pooling is **locally invariant** and applied **independently** to **each feature map/channel.**

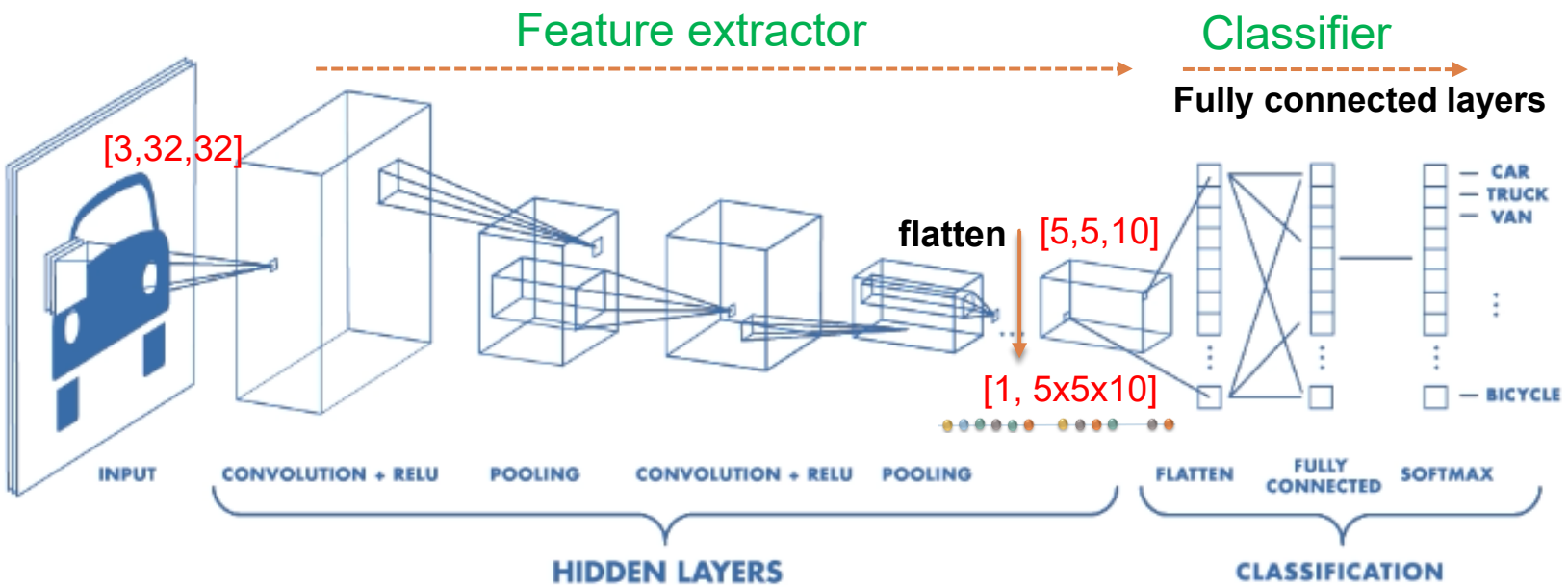
- In the SOTA CNNs (ResNet, GoogleNet, VGG), we employ the pooling layer with
  - Kernel/filter **size = (2,2), strides = (2,2), padding= 0**
  - Therefore, the output tensor has the size
    - $W_o = \left\lfloor \frac{224+0-2}{2} \right\rfloor + 1 = 112$
    - $H_o = \left\lfloor \frac{224+0-2}{2} \right\rfloor + 1 = 112$
  - With this setting, we **down-sample** the **input size** by **2**





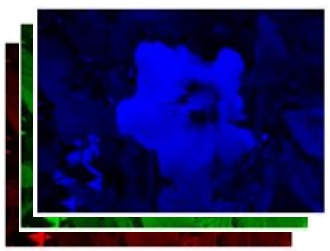
(Source: medium.com)

Fully connected layer

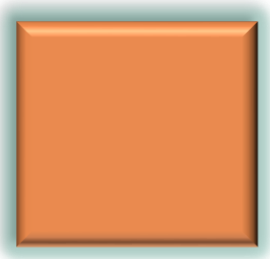


- The last tensor is **flattened,** and **some fully connected layers** are added to classify the input.
  - The last *3D tensor* **[5,5,10]** → 1 **flat layer** with **5x5x10= 250** neurons

CNN in Operation

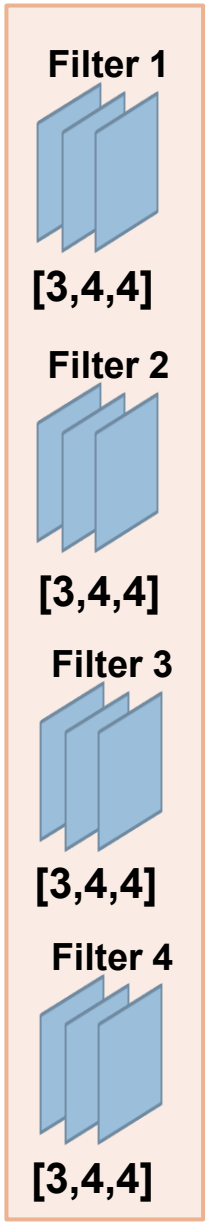


[3,32,32]



Input Layer

Filters [4,3,4,4]

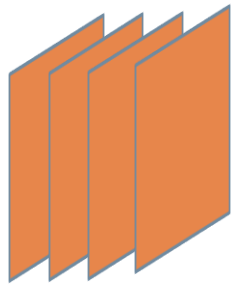


Conv2D 1

padding= 0
strides = (2,2)

$$out\_size = \left\lfloor \frac{32 + 0 - 4}{2} \right\rfloor + 1 = 15$$

Feature volume
Feature maps



[4,15,15]

Latent representation

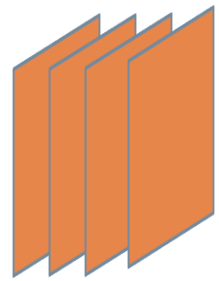
Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= 1

MaxPool2d(kernel_size=2, stride=2, padding=1)

$$out\_size = \left\lfloor \frac{15 + 2 \times 1 - 2}{2} \right\rfloor + 1 = 8$$

Feature volume
Feature maps



[4,8,8]

Latent representation

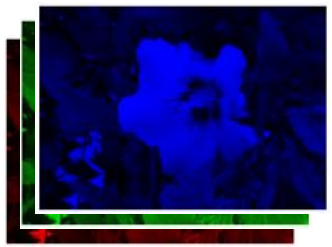
FC layer
4x8x8 neurons

softmax

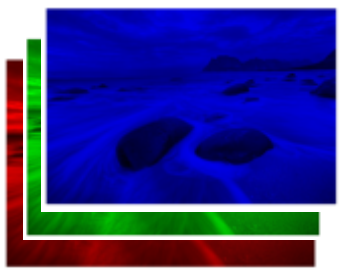
Output layer
10 neurons for 10 classes

Conv2d(3,4,kernel_size=4, stride=2, padding=0)

CNN in Operation



[3,32,32]



[3,32,32]

[2,3,32,32]

Input Layer

Filters [4,3,4,4]

Filter 1

[3,4,4]

Filter 2

[3,4,4]

Filter 3

[3,4,4]

Filter 4

[3,4,4]

Conv2D 1

padding= 0
strides = (2,2)

Feature volume
Feature maps
[2,4,15,15]

[4,15,15]

[4,15,15]

Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= 1

Feature volume
Feature maps
[2,4,8,8]

[4,8,8]

[4,8,8]

4x8x8

2

FC layer

4x8x8 neurons

10

2

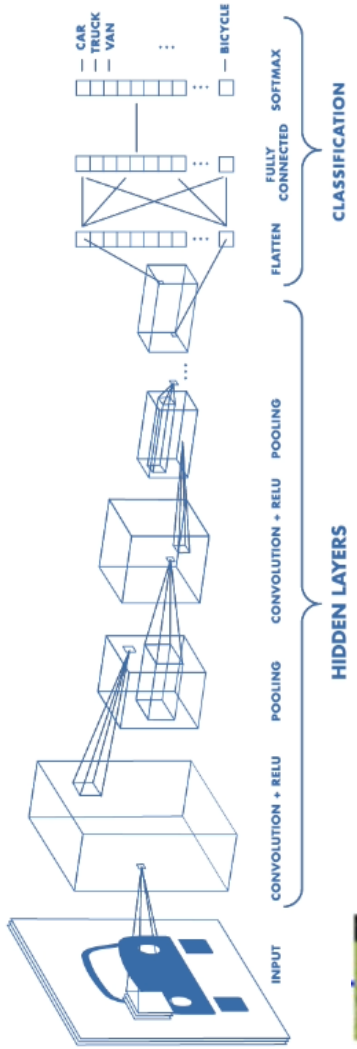
softmax

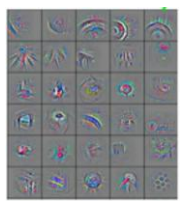
Output layer

10 neurons
for 10 classes

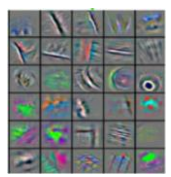
Automatic feature extractor

Deep learning for visual data





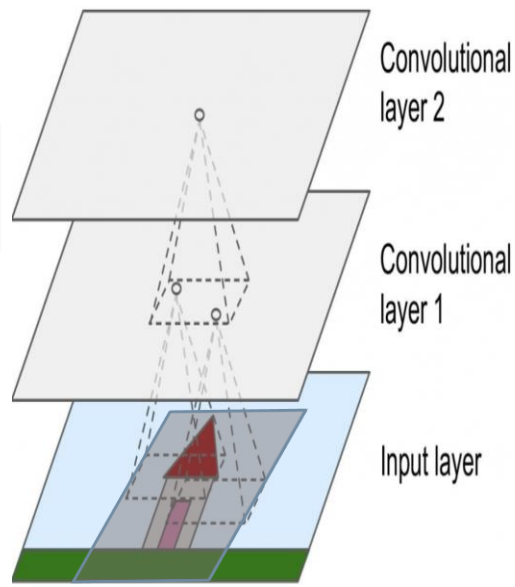
High-level feature/representation
- objects

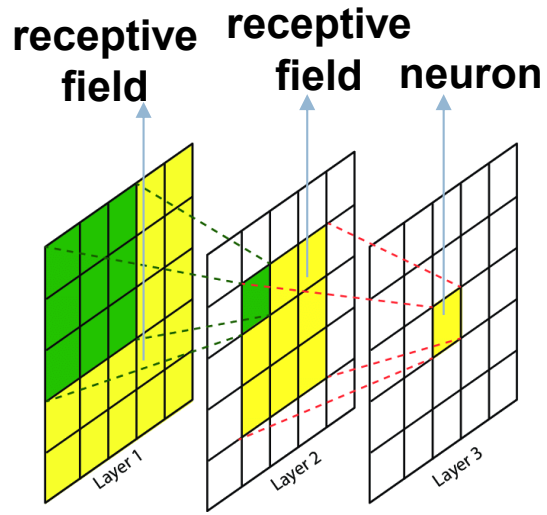


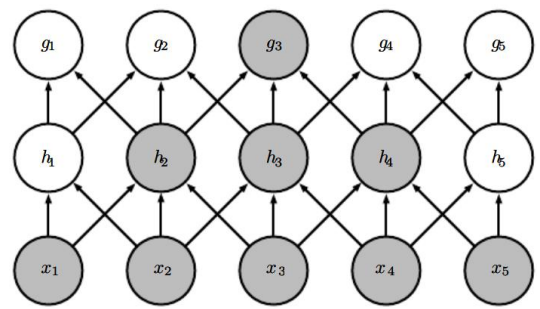
Mid-level feature/representation
- circles, triangles, boxes

Low-level feature/representation
- edges, pixels, corners

Neurons on **higher layers** have **larger receptive fields** (patches) on **input images**

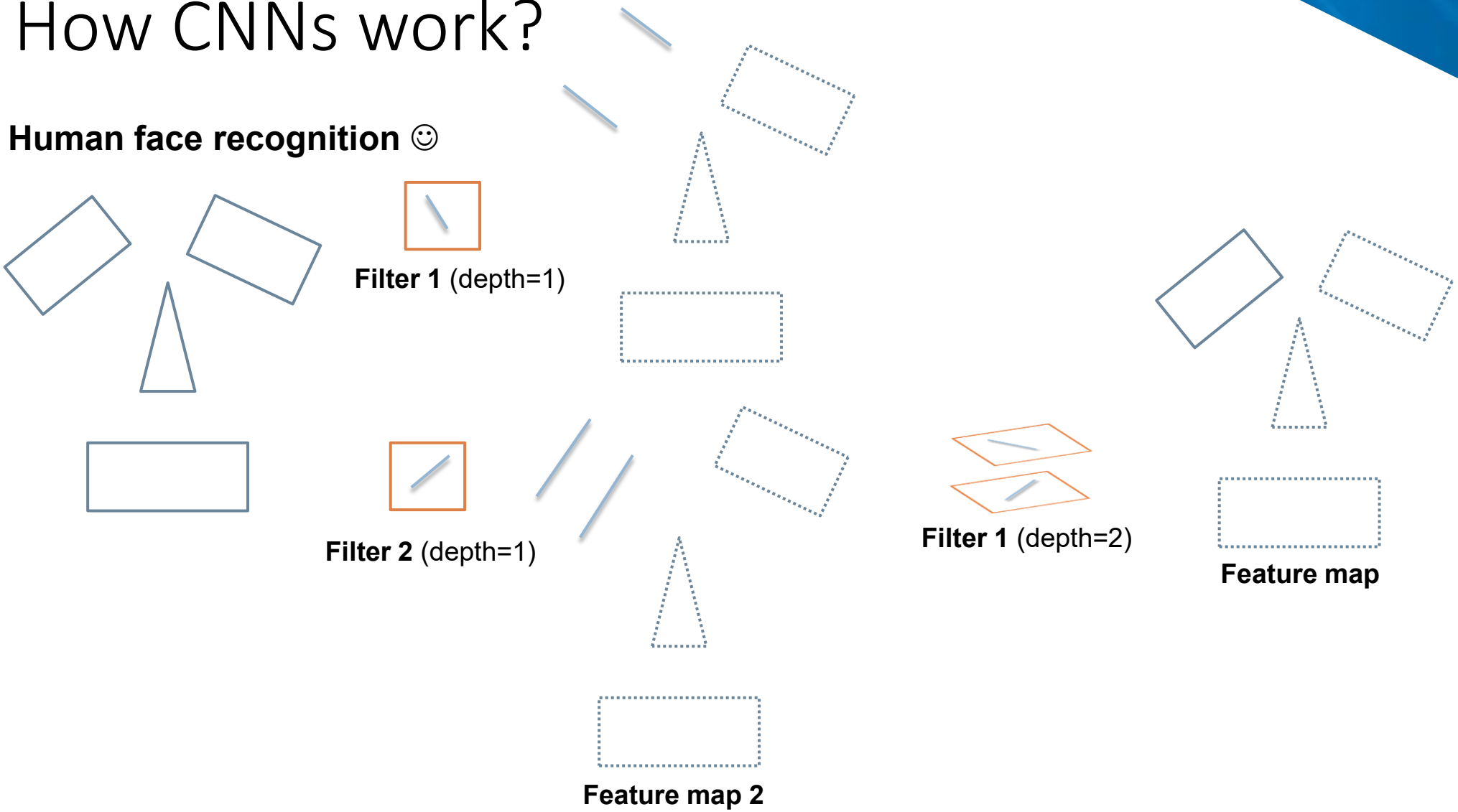






How CNNs work?

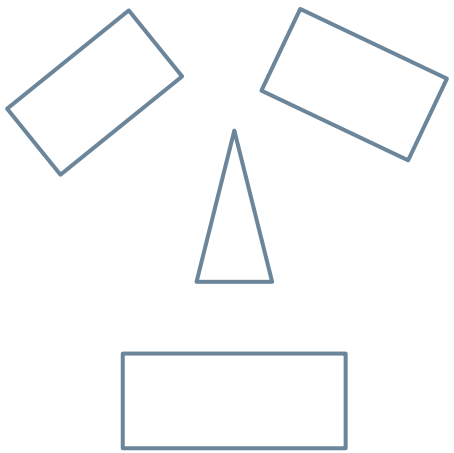
Human face recognition ☺



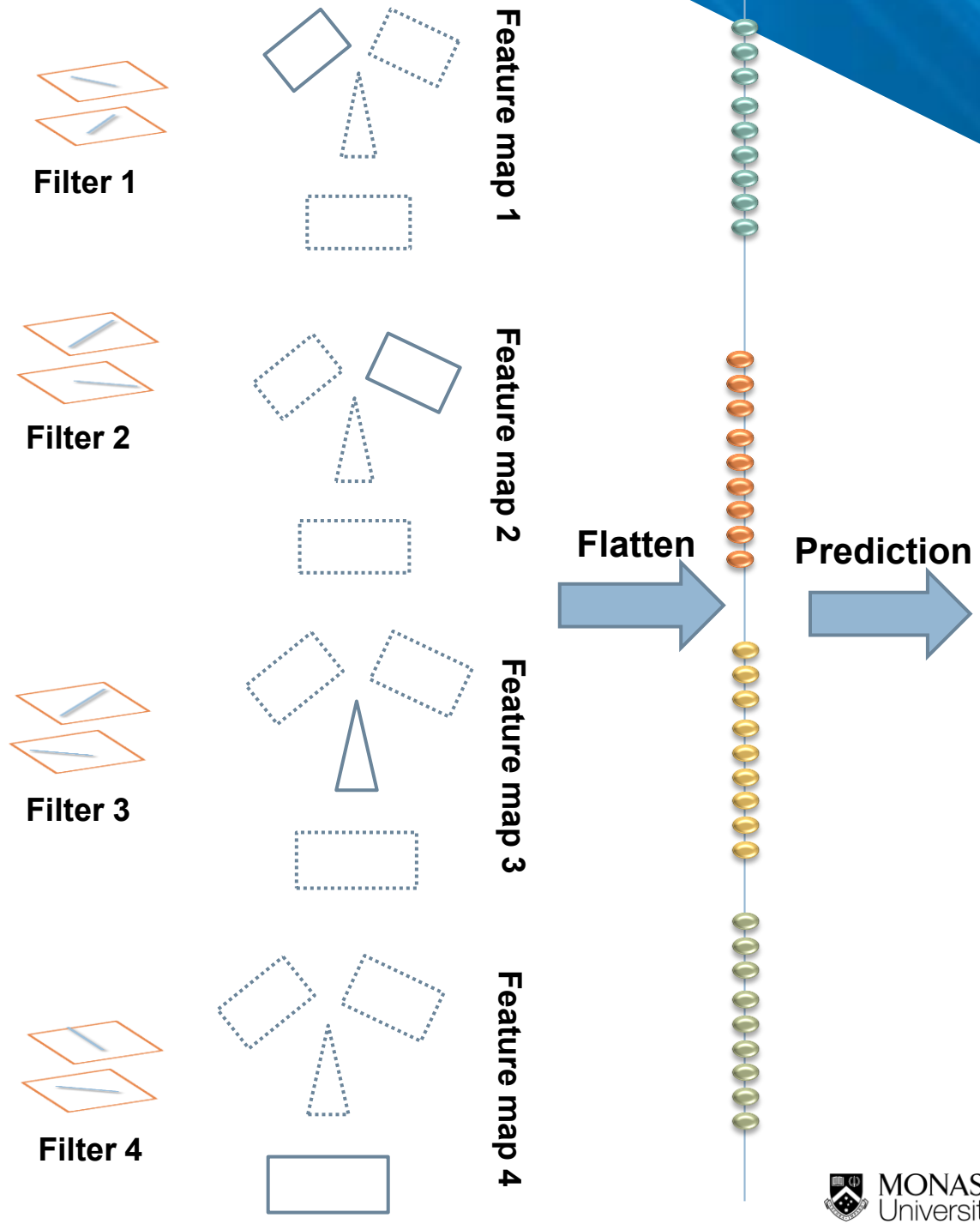
- Higher level filters **combine** lower level filters to represent **more abstract objects**
  - The context is **locally expanded**

How CNNs work?

Human face recognition ☺

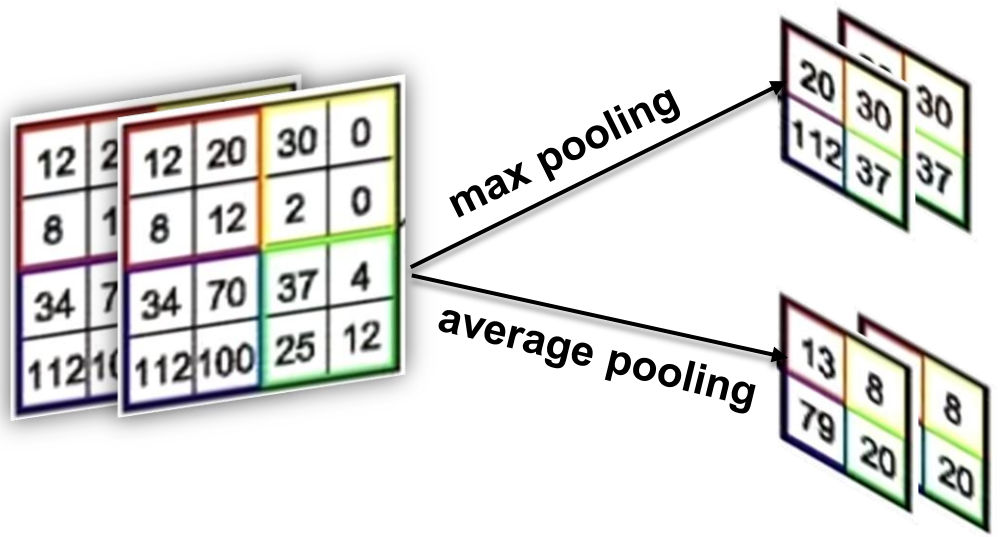


- CNN cannot capture **spatial relationships** among objects
  - How **spatially related** of two eyes, nose and eye, eye and mouth

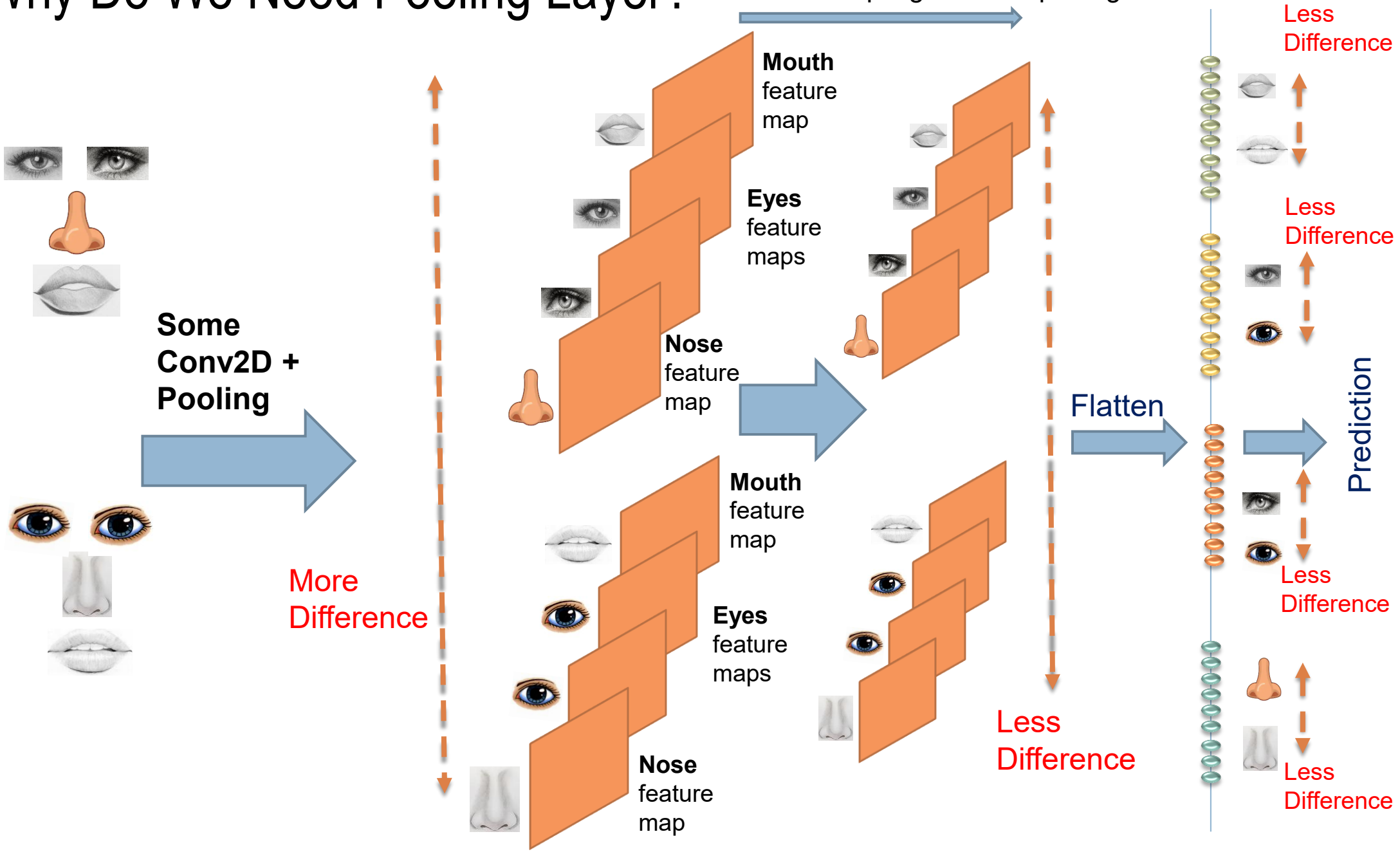


Pooling layer

- Max/average-pooling is **locally invariant** and applied **independently** to **each feature map**
- Reduce the size of feature map to be more manageable
  - Smaller size for flattening
- Reduce the difference of output tensors



Why Do We Need Pooling Layer?



Some comments from Hinton

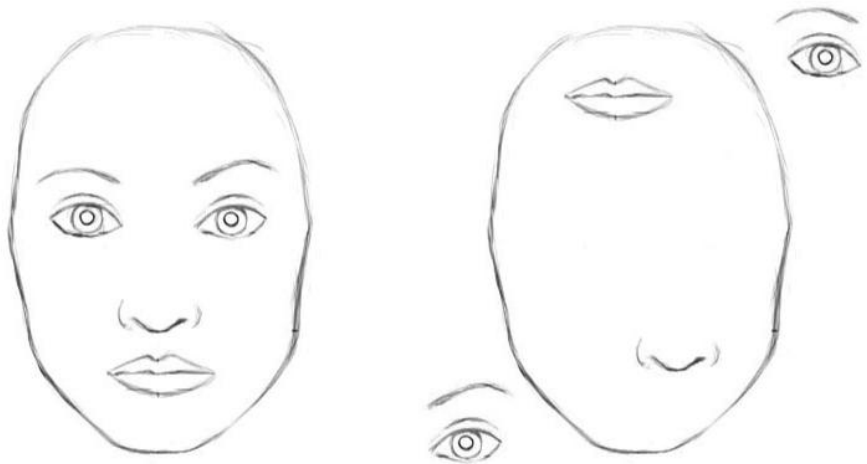
Dynamic Routing Between Capsules

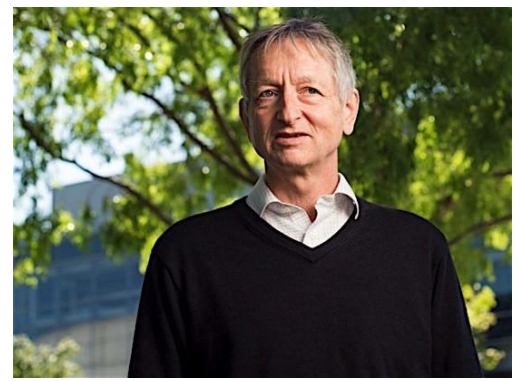
Sara Sabour **Nicholas Frosst**

Geoffrey E. Hinton
Google Brain
Toronto
{sasabour, frosst, geoffhinton}@google.com

Paper link: https://arxiv.org/pdf/1710.09829.pdf

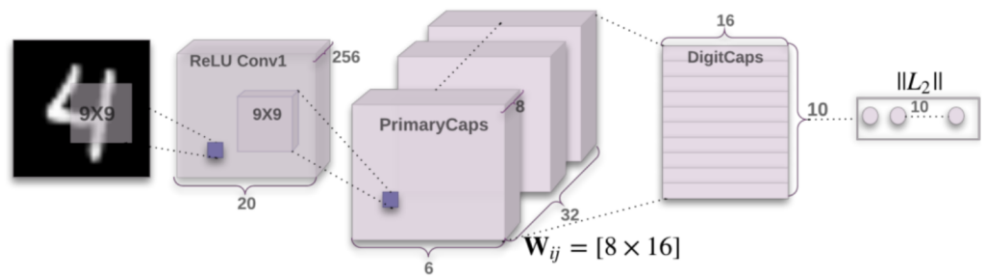
(Source: medium.com)





Hinton: "The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster."

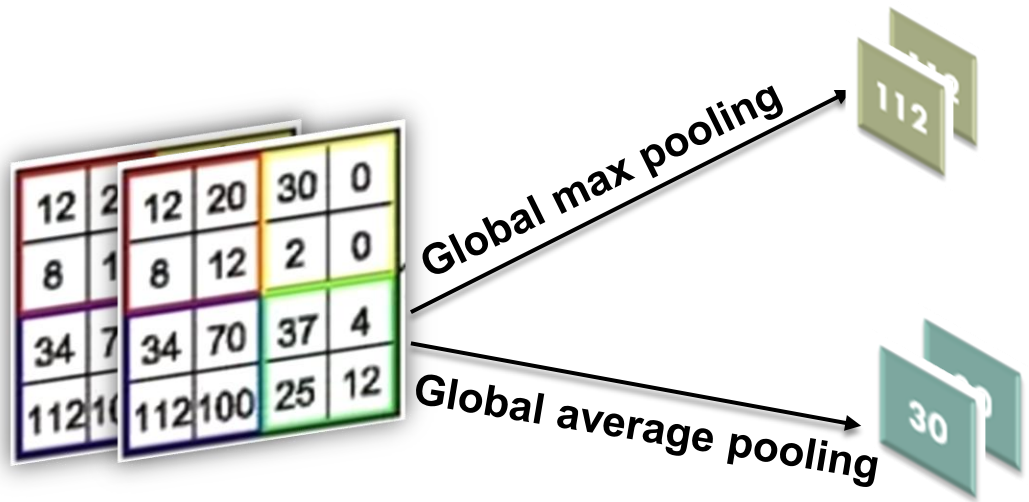
- CNNs cannot capture the **spatial relationships** among objects, hence it **wrongly predicts every face** with **two eyes**, **one nose**, and **one mouth** as a **human face**.



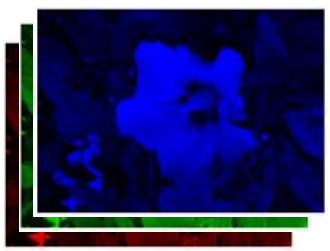
(Source: original paper of **Capsule Net**)

Global pooling layer

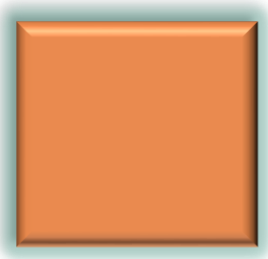
- Like **pooling layer**, but the **kernel size** is set to **input size**
  - **Each feature map** becomes a **single neuron**
  - **Global average pooling** (GAP)
  - **Global max pooling** (GMP)
  - **4D input tensor** [batch_size, in_channel, in_height, in_width]→ **2D output tensor** [batch_size, in_channel]



Global Pooling Layer

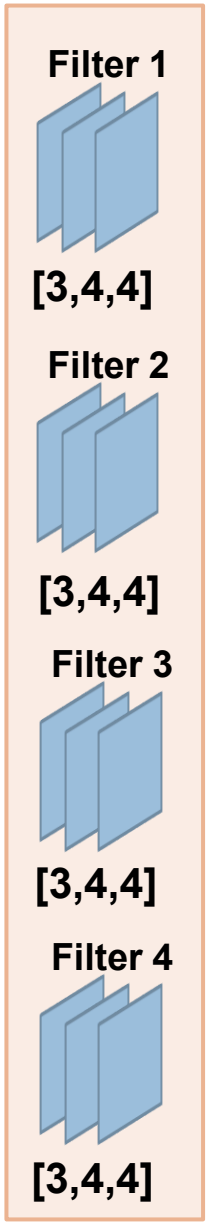


[3,32,32]



Input Layer

Filters [4,3,4,4]

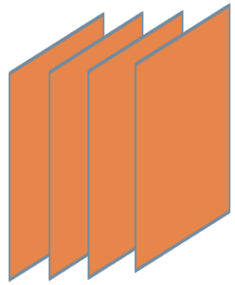


Conv2D 1

padding= 0
strides = (2,2)

Conv2d(3,4,kernel_size=4, stride=2, padding=0)

Feature volume
Feature maps



[4,15,15]

Latent representation

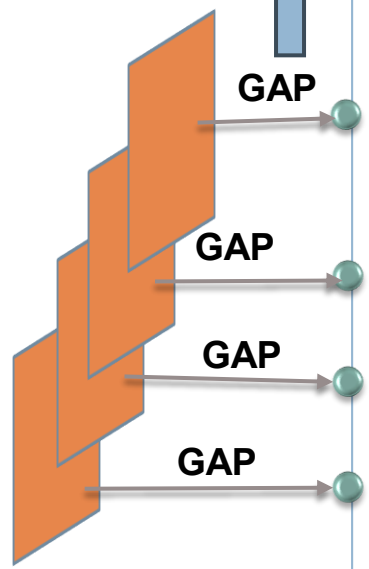
Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= 1

MaxPool2d(kernel_size=2, stride=2, padding=1)

```
AdaptiveAvgPool2d((1, 1))
Flatten(1)
```

Feature volume
Feature maps



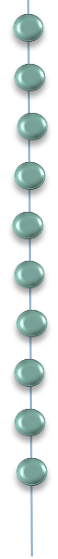
[4,8,8]

Latent representation

FC layer

4 neurons

softmax



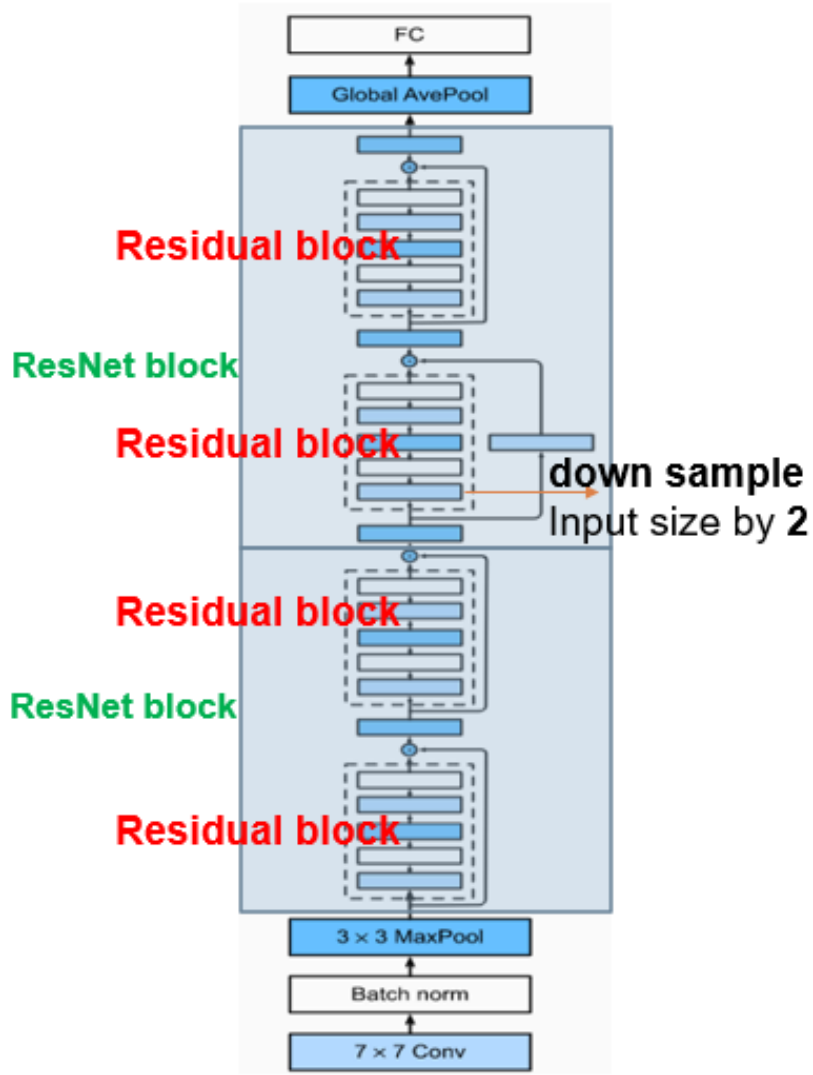
Output layer

10 neurons
for 10 classes

SOTA CNNs

ResNet

- **Won ImageNet** competition **in 2015**
- Many ResNet blocks
- Each ResNet block contains **many residual blocks**
  - Number of residual blocks
  - Number of channels for residual blocks
- Each ResNet block
  - From the **second block**, the first **residual block**
    - Use **1x1 Conv skip connection**
  - Otherwise
    - Use **standard skip connection**

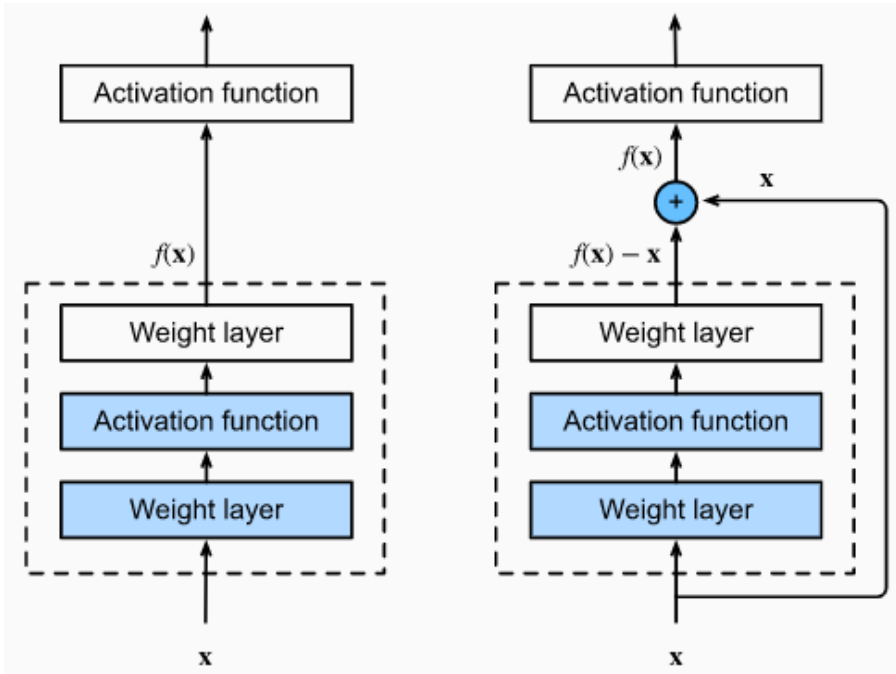


ResNet architecture

ResNet

Residual Block

(Source:Dive into DL)

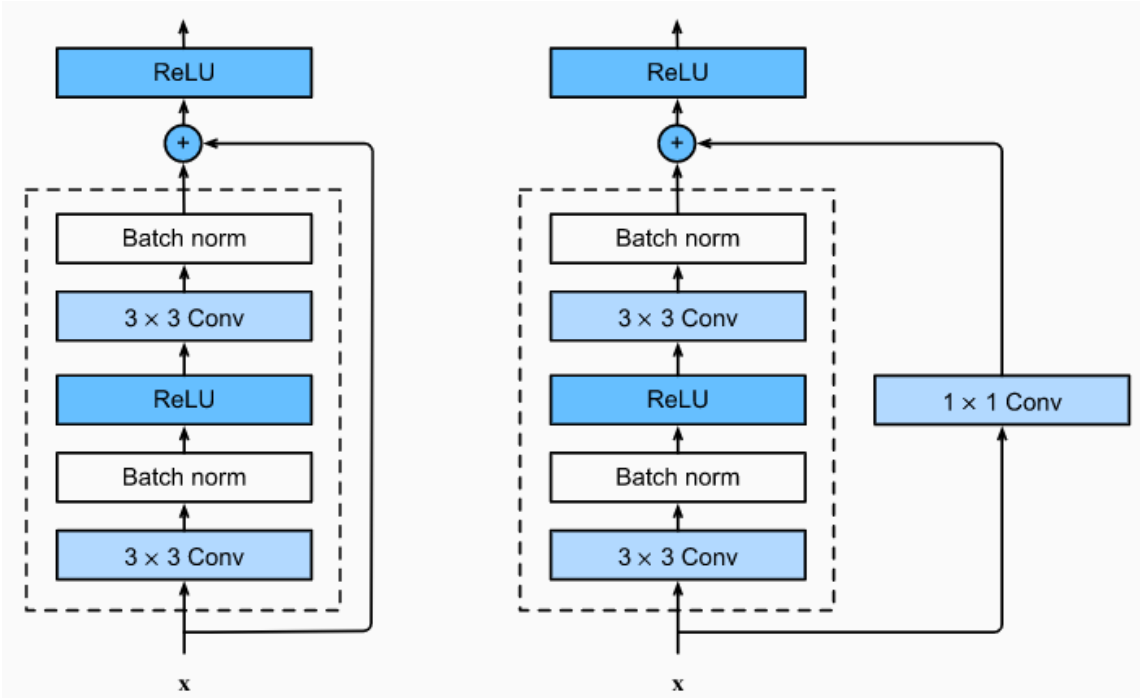


Without skip connection **With skip connection**

- Learn $f(x) = g(x) + x$ where $g(x) = f(x) - x$
  - The **model expressiveness** is the **same** as previous
  - The **gradient** $\nabla f(x) = \nabla g(x) + \mathbf{1}$ **looks better** where $\mathbf{1}$ is the vector of **all** 1 with the same shape as x

ResNet

Residual Block



Not use 1x1 Conv **Use 1x1 Conv**

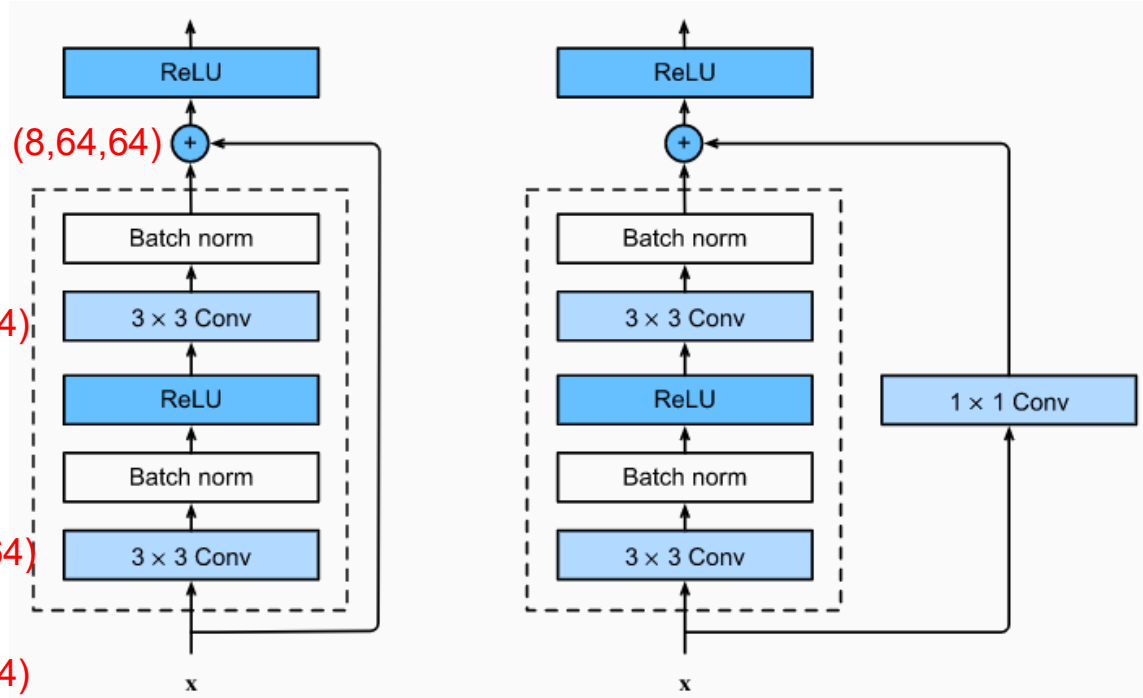
- ResNet follows VGG's full **3×3 convolutional layer** design.
  - The residual block has two **3×3 convolutional layers** with the **same number of output channels**.
  - Each **convolutional layer** is followed by a **batch normalization layer** and a **ReLU activation function**.
  - **Skip these two convolution operations** and **add the input directly** before the final ReLU activation function.
  - Requires that the output of **the two convolutional layers** must be of **the same shape** as the input, so that they can be added together.
    - If we want to change the number of channels, we need to introduce an additional **1×1 convolutional layer** to transform the input into the **desired shape** for the addition operation.

ResNet

Residual Block

padding=1
strides= (1,1)
#filters=8

padding=1
strides= (1,1)
#filters=8

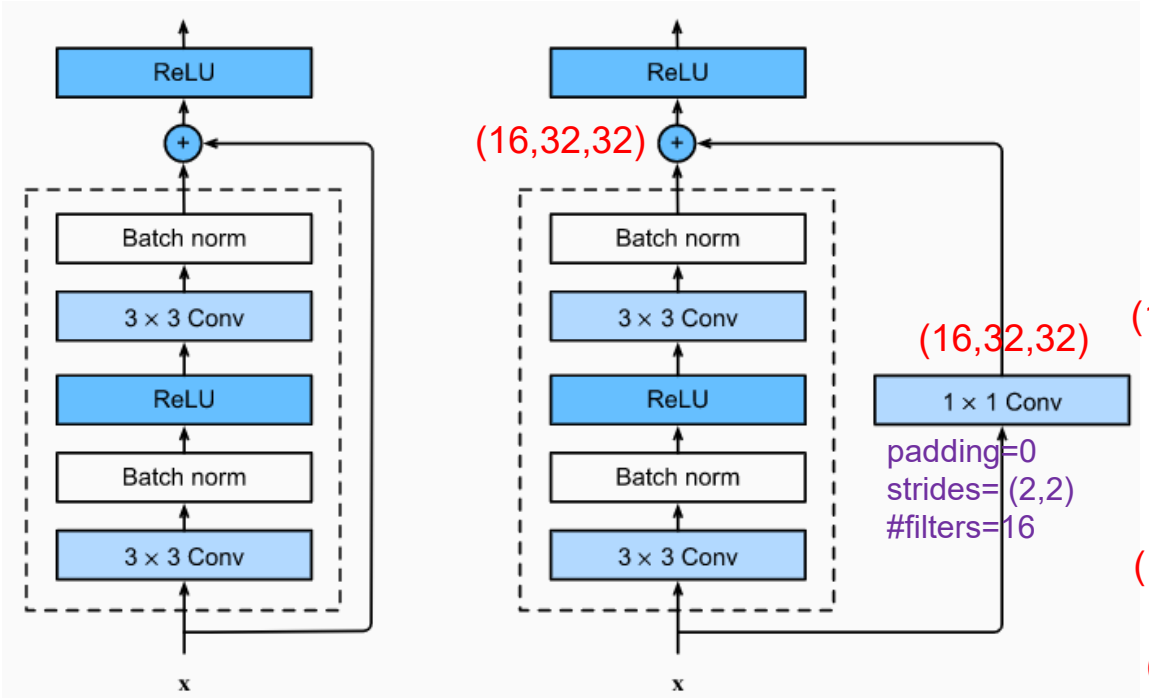


Not use 1x1 Conv **Use 1x1 Conv**

- ResNet follows VGG's full **3×3 convolutional layer** design.
  - The residual block has two **3×3 convolutional layers** with the **same number of output channels**.
  - Each **convolutional layer** is followed by a **batch normalization layer** and a **ReLU activation function**.
  - **Skip these two convolution operations** and **add the input directly** before the final ReLU activation function.
  - Requires that the output of **the two convolutional layers** must be of **the same shape** as the input, so that they can be added together.
    - If we want to change the number of channels, we need to introduce an additional **1×1 convolutional layer** to transform the input into the **desired shape** for the addition operation.

ResNet

Residual Block

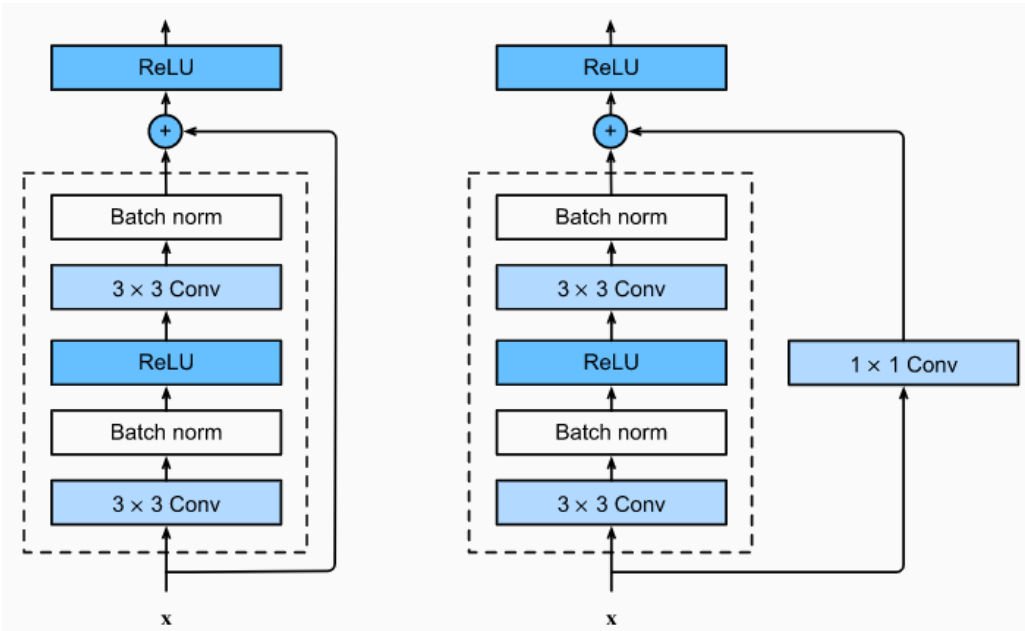


Not use 1x1 Conv **Use 1x1 Conv**

- ResNet follows VGG's full **3×3 convolutional layer** design.
  - The residual block has two **3×3 convolutional layers** with the **same number of output channels**.
  - Each **convolutional layer** is followed by a **batch normalization layer** and a **ReLU activation function**.
  - **Skip these two convolution operations** and **add the input directly** before the final ReLU activation function.
  - Requires that the output of **the two convolutional layers** must be of **the same shape** as the input, so that they can be added together.
    - If we want to change the number of channels, we need to introduce an additional **1×1 convolutional layer** to transform the input into the **desired shape** for the addition operation.

Implementation of Residual Block

```python
class Residual(nn.Module):
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, stride=strides, padding=1)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        self.conv3 = None
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=strides)
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)
        self.relu = nn.ReLU()

    def forward(self, X):
      Y = self.relu(self.bn1(self.conv1(X)))
      Y = self.bn2(self.conv2(Y))
      if self.conv3 is not None:
        X = self.conv3(X)
      Y += X
      return self.relu(Y)
```




```python
blk = Residual(3)
X = torch.rand((4, 3, 6, 6))
Y = blk(X)
print(Y.shape)
```

```
torch.Size([4, 3, 6, 6])
```

```python
blk = Residual(num_channels=3, use_1x1conv=True, strides=1)
X = torch.rand((10, 3, 32, 32))
Y = blk(X)
print(Y.shape)
```

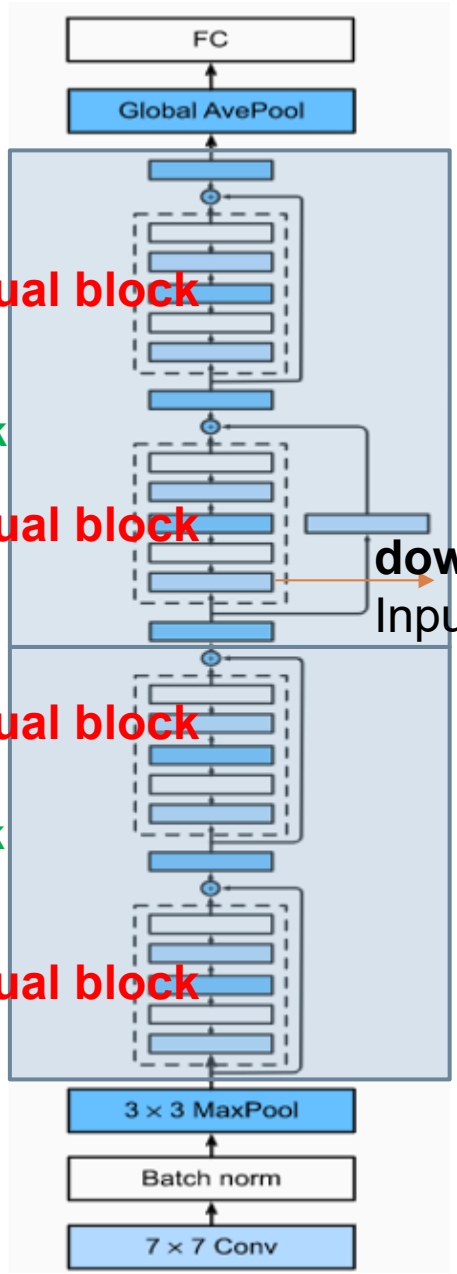
```
torch.Size([10, 3, 32, 32])
```

```python
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

(Source:Dive into DL)

Implementation of ResNet

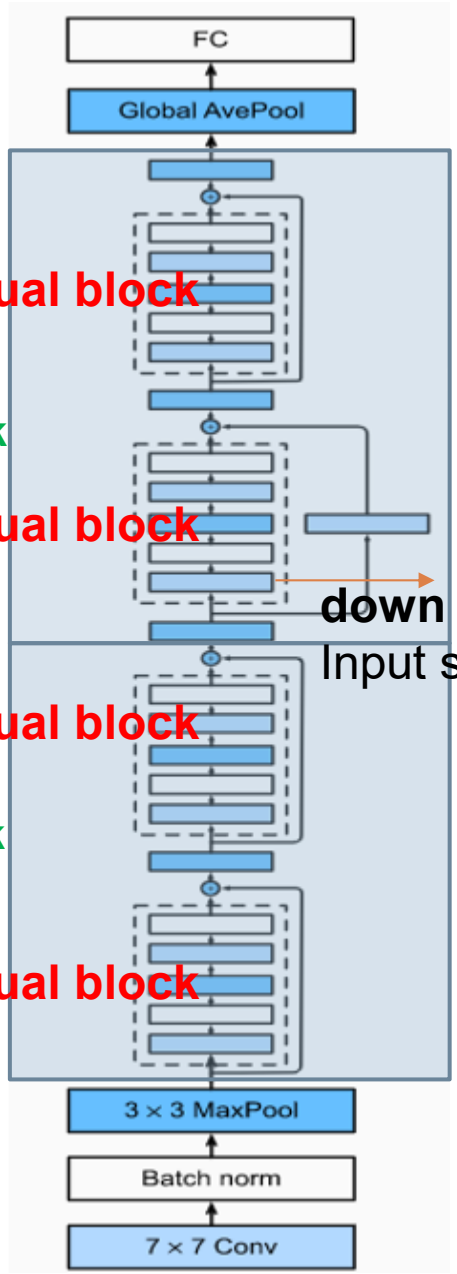


(Source:Dive into DL)

```
class ResnetBlock(nn.Module):
    def __init__(self, num_channels, num_residuals, first_block=False, **kwargs):
        super(ResnetBlock, self).__init__(**kwargs)
        self.residual_layers = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                self.residual_layers.append(
                    Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                self.residual_layers.append(Residual(num_channels))
        self.residual_blk = nn.ModuleList(self.residual_layers)

    def forward(self, X):
        for layer in self.residual_blk:
            X = layer(X)
        return X
```

```
class create_ResNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.layers = nn.ModuleList([
                  nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                  nn.LazyBatchNorm2d(),
                  nn.ReLU(),
                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
                  ResnetBlock(64, 2, first_block=True),
                  ResnetBlock(128, 2),
                  ResnetBlock(256, 2),
                  nn.AdaptiveAvgPool2d((1, 1)),
                  nn.Flatten(1),
                  nn.LazyLinear(10),
                  # nn.Softmax(dim=-1)
                  ])
  def forward(self, X):
    for _, layer in enumerate(self.layers):
      X = layer(X)
    return X
```

Implementation of ResNet



(Source:Dive into DL)

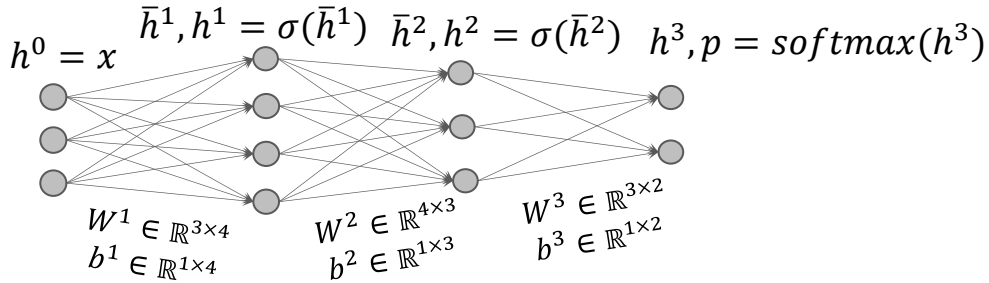
```
class ResnetBlock(nn.Module):
    def __init__(self, num_channels, num_residuals, first_block=False, **kwargs):
        super(ResnetBlock, self).__init__(**kwargs)
        self.residual_layers = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                self.residual_layers.append(
                    Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                self.residual_layers.append(Residual(num_channels))
        self.residual_blk = nn.ModuleList(self.residual_layers)

    def forward(self, X):
        for layer in self.residual_blk:
            X = layer(X)
        return X
```

```
class create_ResNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.layers = nn.ModuleList([
(1, 64,112,112) ⇐ nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
(1, 64,112,112) ⇐ nn.LazyBatchNorm2d(),
(1, 64,112,112) ⇐ nn.ReLU(),
 (1, 64,56,56)  ⇐ nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
 (1, 64,56,56)  ⇐ ResnetBlock(64, 2, first_block=True),
(1, 128,28,28)  ⇐ ResnetBlock(128, 2),
(1, 256,14,14)  ⇐ ResnetBlock(256, 2),
  (1, 256,1,1)  ⇐ nn.AdaptiveAvgPool2d((1, 1)),
     (1,256)    ⇐ nn.Flatten(1),
     (1,10)     ⇐ nn.LazyLinear(10),
                  # nn.Softmax(dim=-1)
                  ])
  def forward(self, X):
    for _, layer in enumerate(self.layers):
      X = layer(X)
    return X
```

Practical Skills in Deep Learning

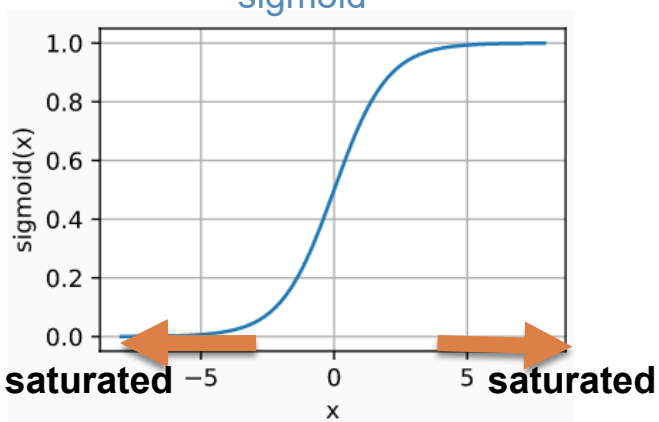
Gradient vanishing

- **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



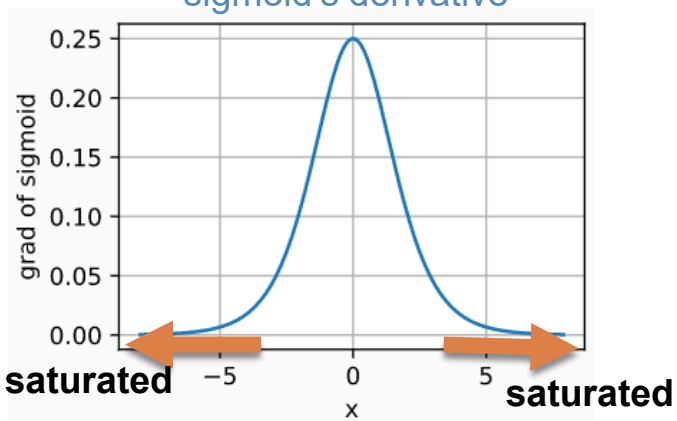
$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3}\cdot\frac{\partial h^3}{\partial h^2}\cdot\frac{\partial h^2}{\partial \bar{h}^2}\cdot\frac{\partial \bar{h}^2}{\partial h^1}\cdot\frac{\partial h^1}{\partial \bar{h}^1}\cdot\frac{\partial \bar{h}^1}{\partial W^1}$$
$$= \left[(p^T -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

sigmoid



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

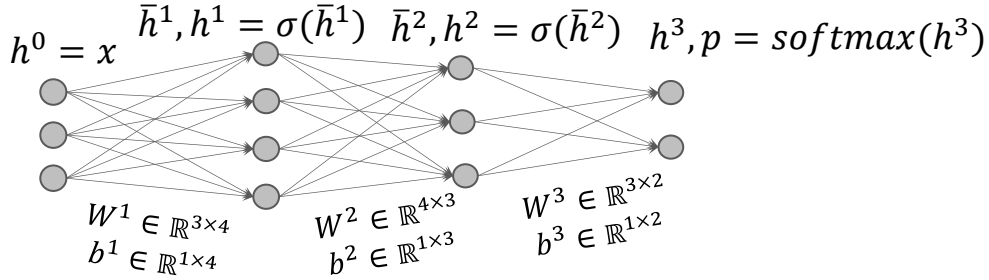
sigmoid's derivative



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

Gradient vanishing

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



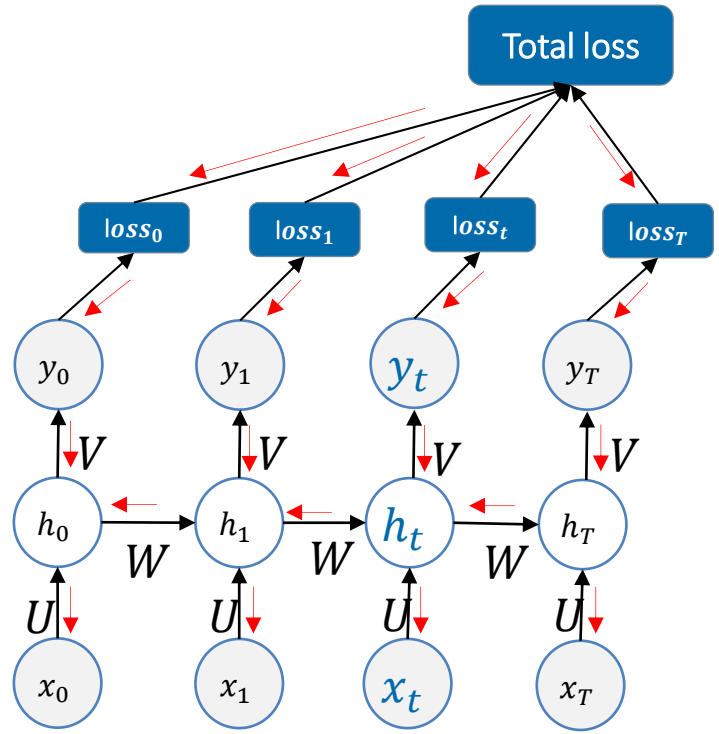
$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3}\cdot\frac{\partial h^3}{\partial h^2}\cdot\frac{\partial h^2}{\partial \bar{h}^2}\cdot\frac{\partial \bar{h}^2}{\partial h^1}\cdot\frac{\partial h^1}{\partial \bar{h}^1}\cdot\frac{\partial \bar{h}^1}{\partial W^1}$$

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

Gradient exploding

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

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



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

W W

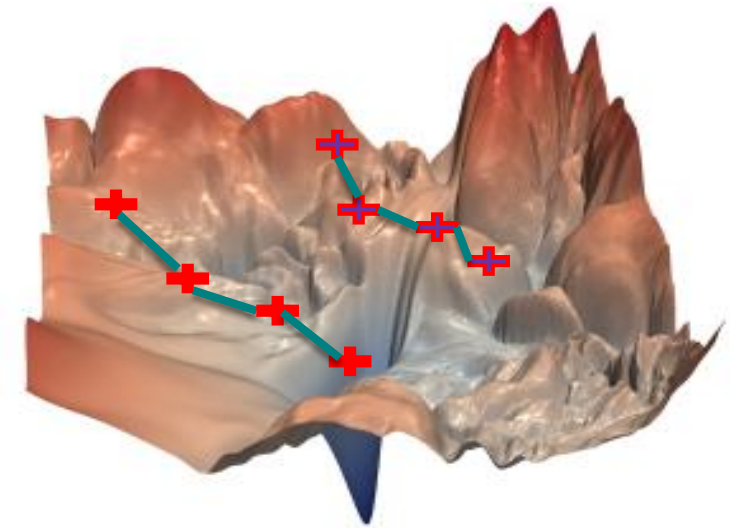
Multiplication of **many matrices** W

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

He and Xavier weight initialization

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

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



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

He and Xavier weight initialization

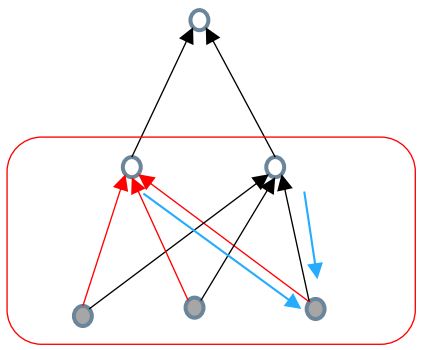
- **Xavier initialization**
  - Try to ensure the **variance of the outputs** of each layer equal to the **variance of its inputs**
  - Also need the gradients to have **equal variance** before and **after flowing through a layer** in the **reverse direction**
  - Good for **sigmoid** and **tanh** functions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

Gaussian version

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

Uniform alternative

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



$n_{in} = 3$

$n_{out} = 2$

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

Understanding the difficulty of training deep feedforward neural networks

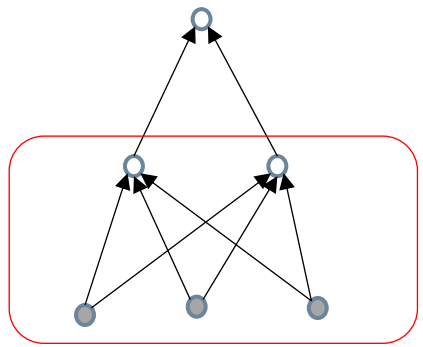
Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

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

He and Xavier weight initialization

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

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



$n_{out} = 2$
$n_{in} = 3$

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

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

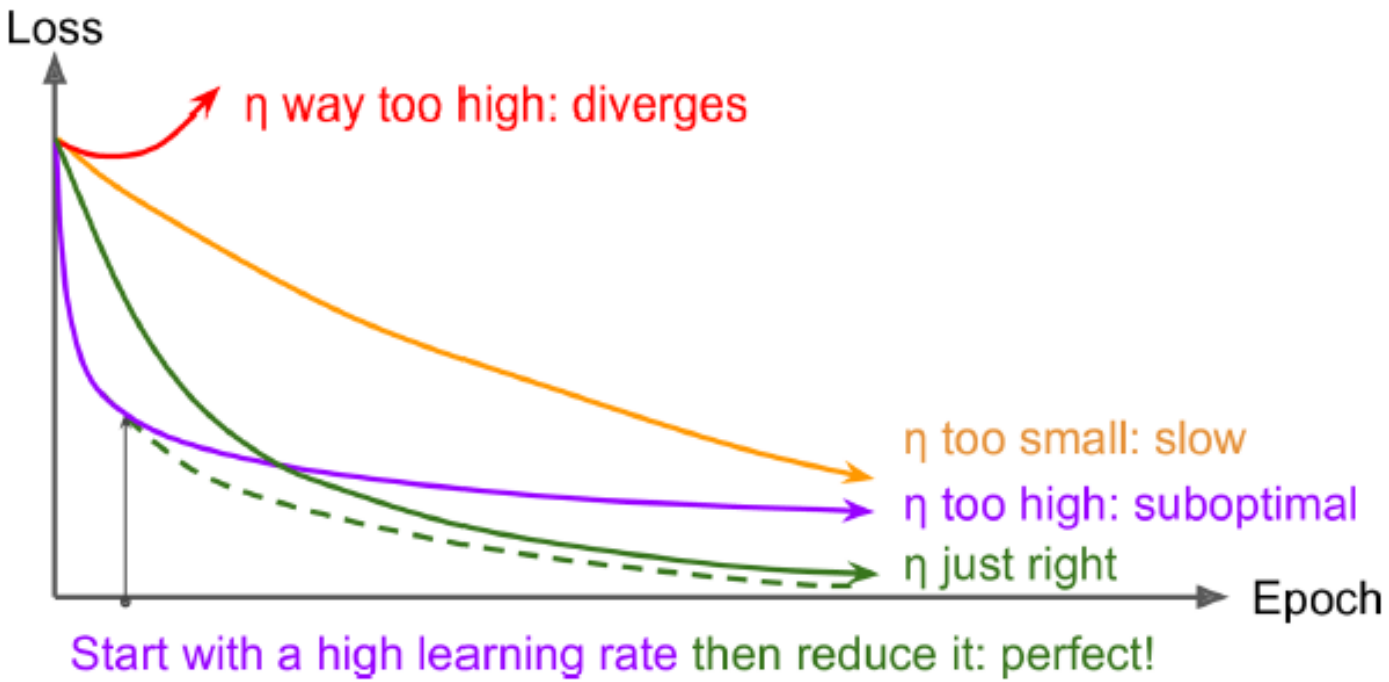
Delving Deep into Rectifiers:
Surpassing Human-Level Performance on ImageNet Classification

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research

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

Deep Learning Pipeline

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



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

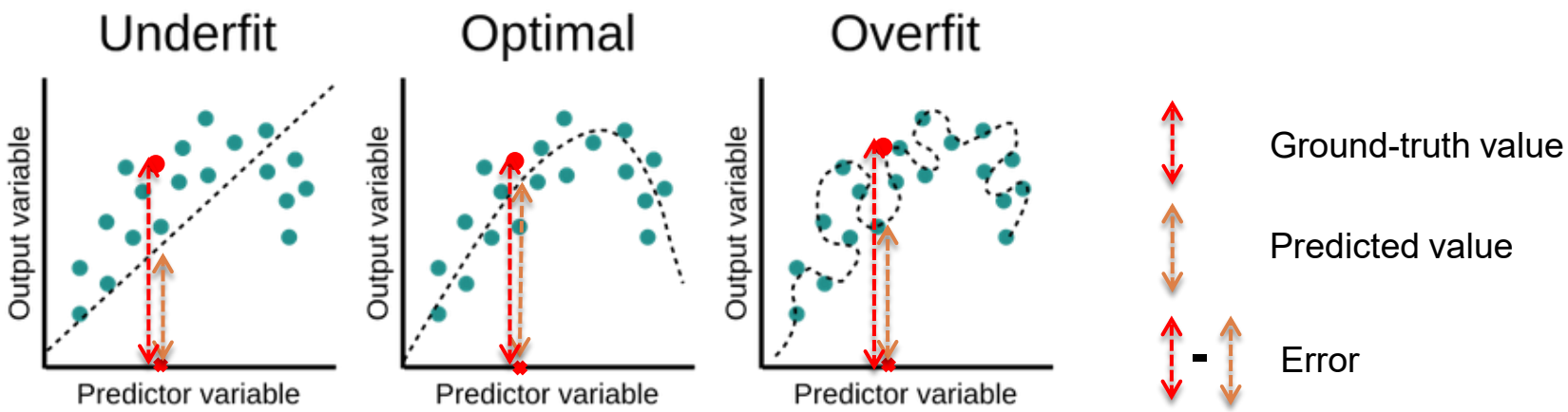
Regularization and Overfitting

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

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

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

Overfitting & Underfitting



Overfitting with regression

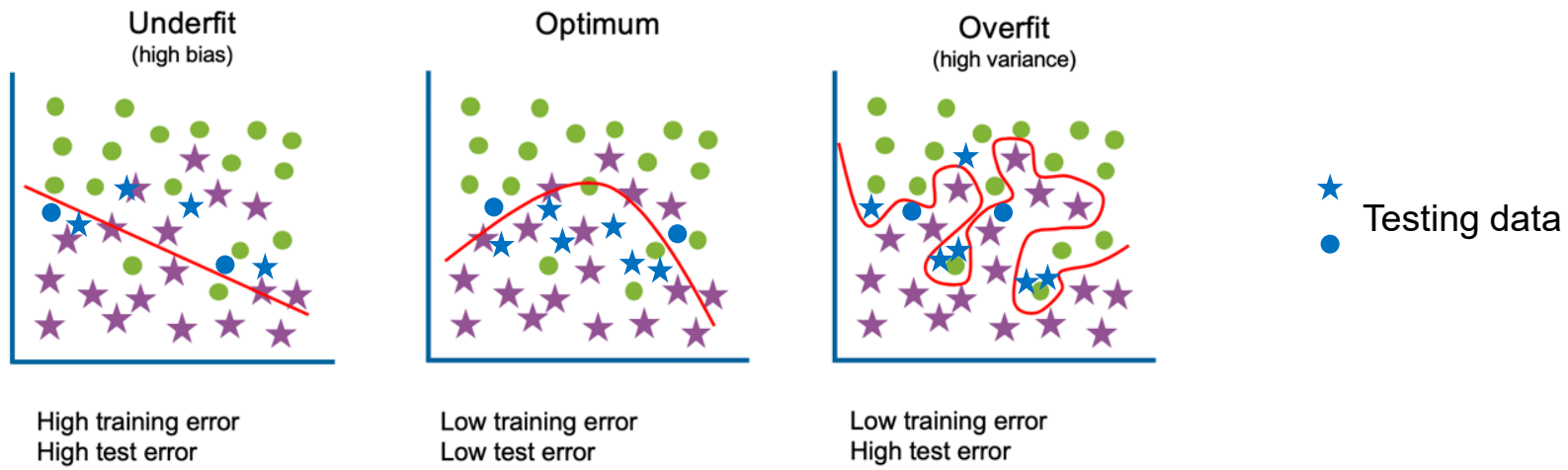
- **Underfitting**
  - The model is **too simple** to characterise a training set
    - Use linear model to learn from non-linear data

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

Overfitting in Deep NNs

Overfitting

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



Overfitting with classification

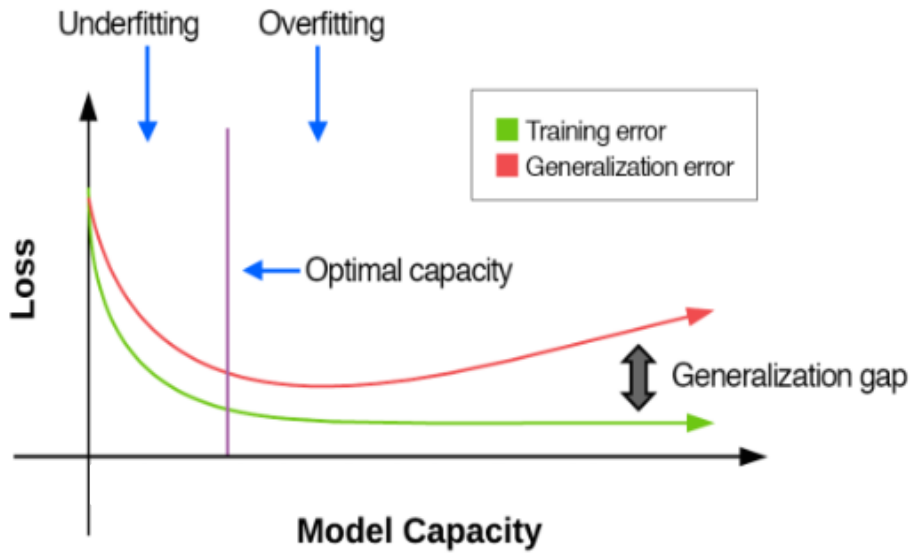
[Source: https://www.ibm.com/cloud/learn/overfitting]

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

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

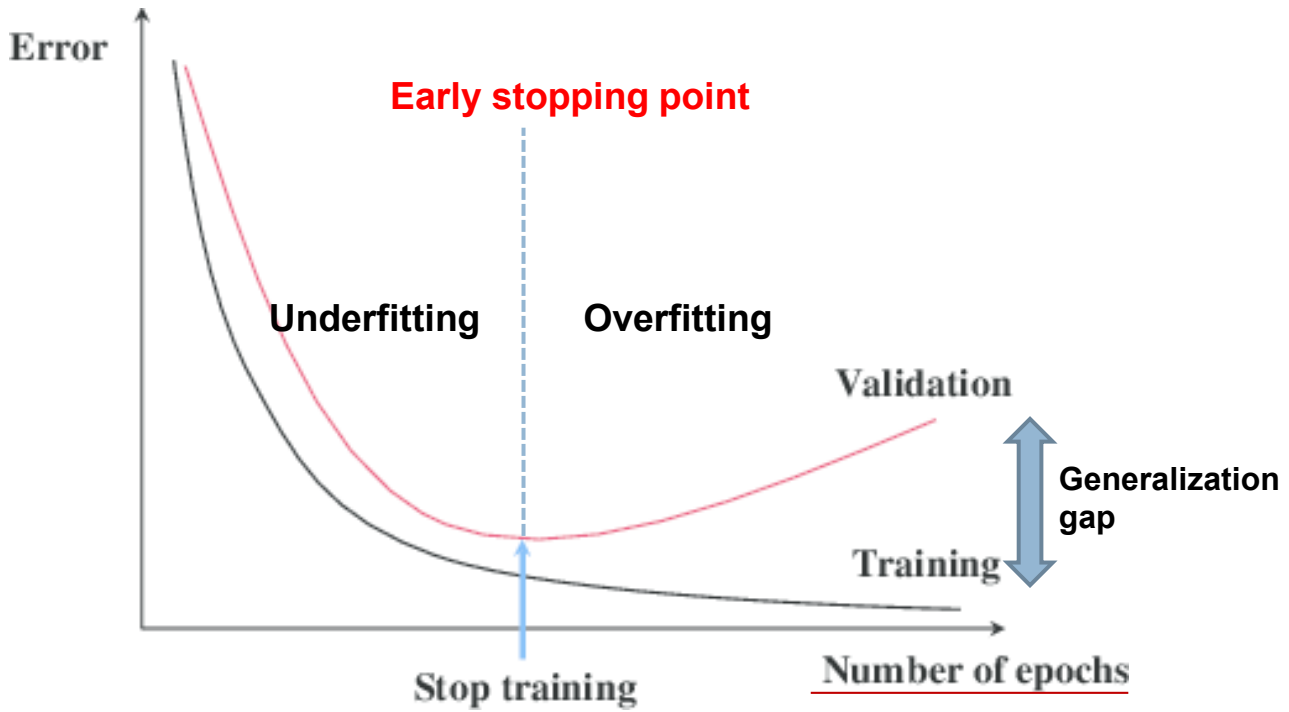
Table 11.1: The effect of various hyperparameters on model capacity.

We can also experiment with model capacity itself in parallel.



[DL, ch11]

Early stopping



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

Add Regularization

Reduce Overfitting

- **Optimization problem**

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

- **L2 regularization**

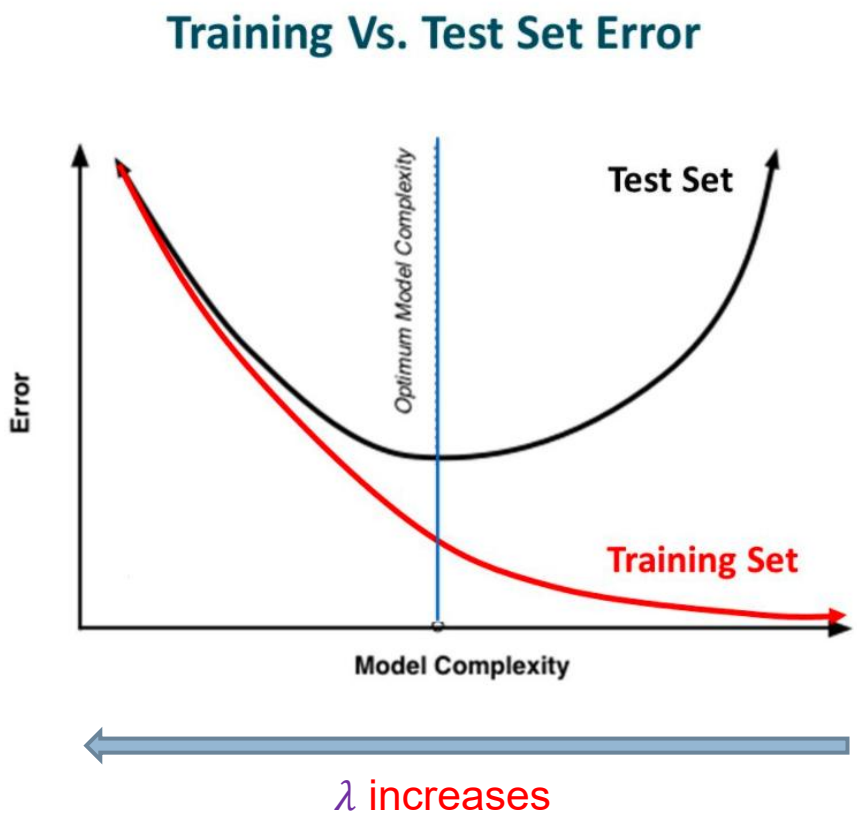
$$\Omega(\theta) = \lambda \sum_{k}\sum_{i,j}\left(W_{i,j}^{\mathbf{k}}\right)^2 = \lambda \sum_{k}\left\|\boldsymbol{W}^{\mathbf{k}}\right\|_F^2$$

  - $\lambda > 0$ 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**

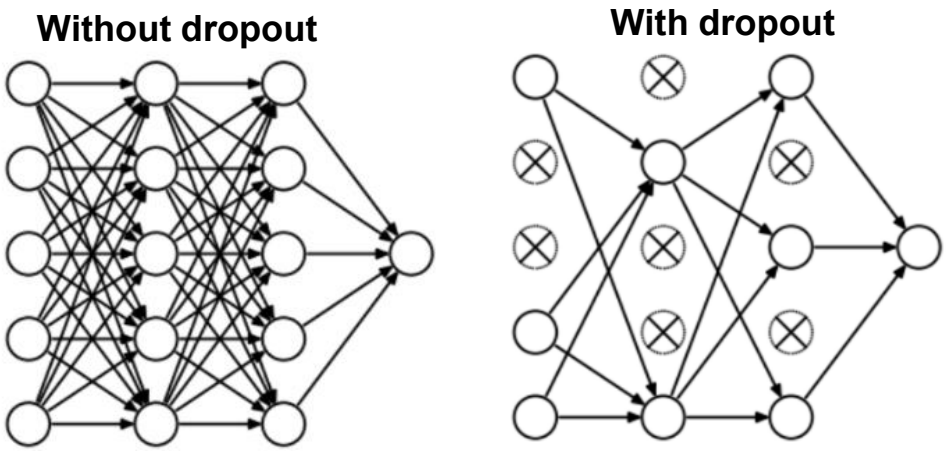
$$\Omega(\boldsymbol{\theta}) = \lambda \sum_{k}\sum_{i,j}\left|W_{i,j}^{\mathbf{k}}\right|$$

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

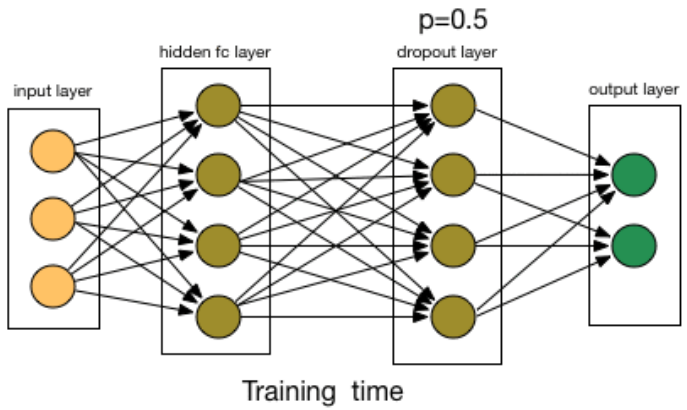


Dropout

Reduce Overfitting



(Source: Analytics Vidhya)



(Source: chatbotslife)

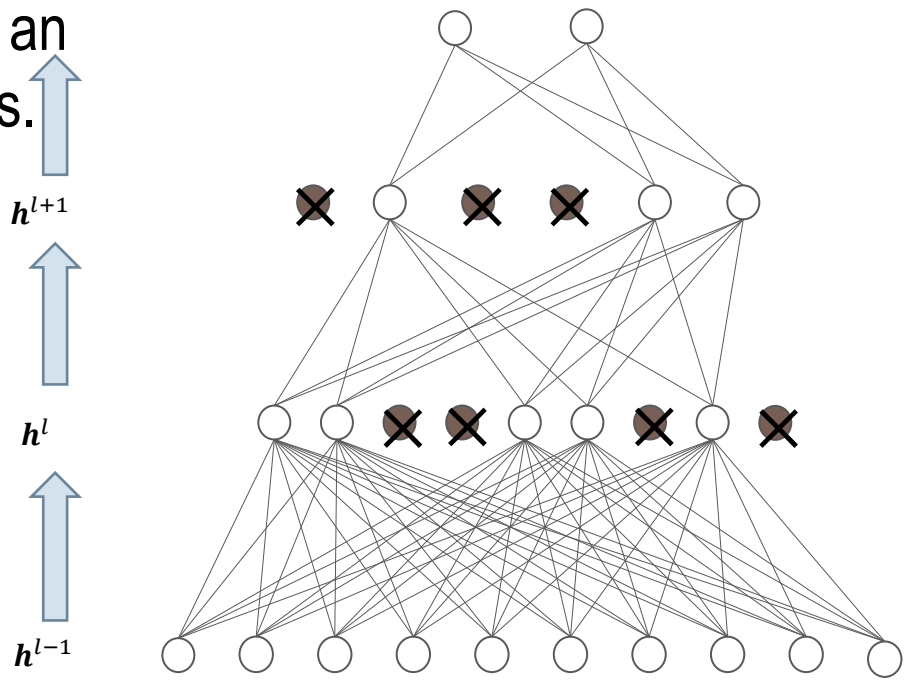
- This is a **cheap technique** to reduce model capacity
  - Reduce overfitting
- In each iteration, at each layer, **randomly choose** some neurons and **drop all connections** from these neurons
  - dropout_rate= 1 – keep_prob

Dropout

Reduce Overfitting

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

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



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

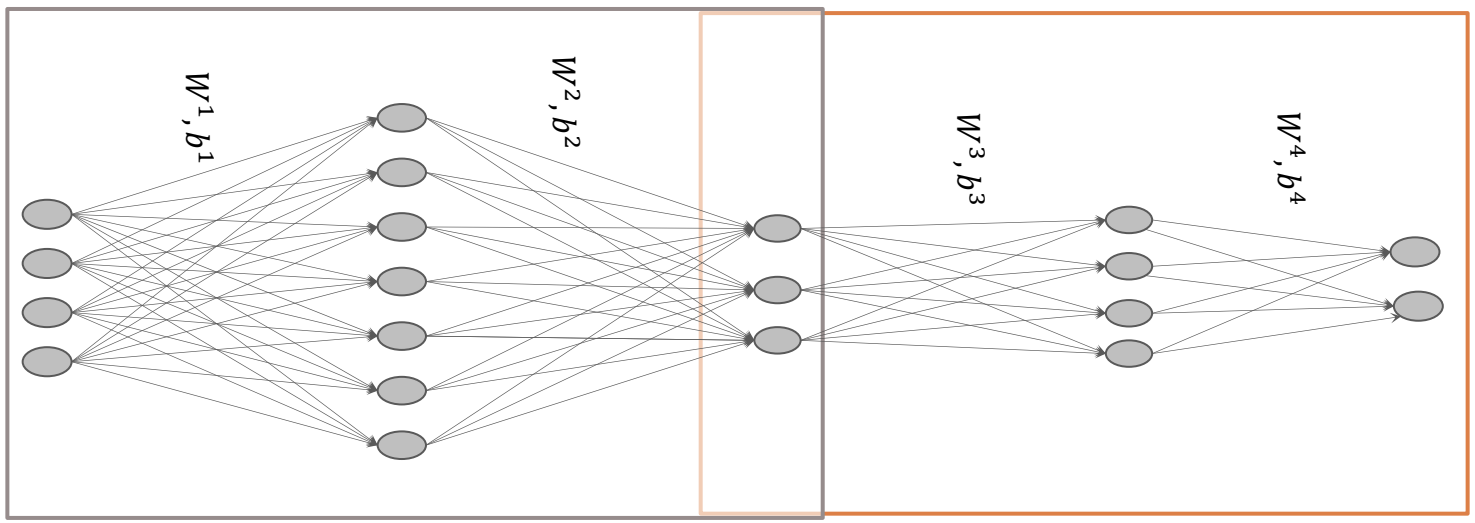
Internal covariate shift problem

Input batch

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

Representation batch

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



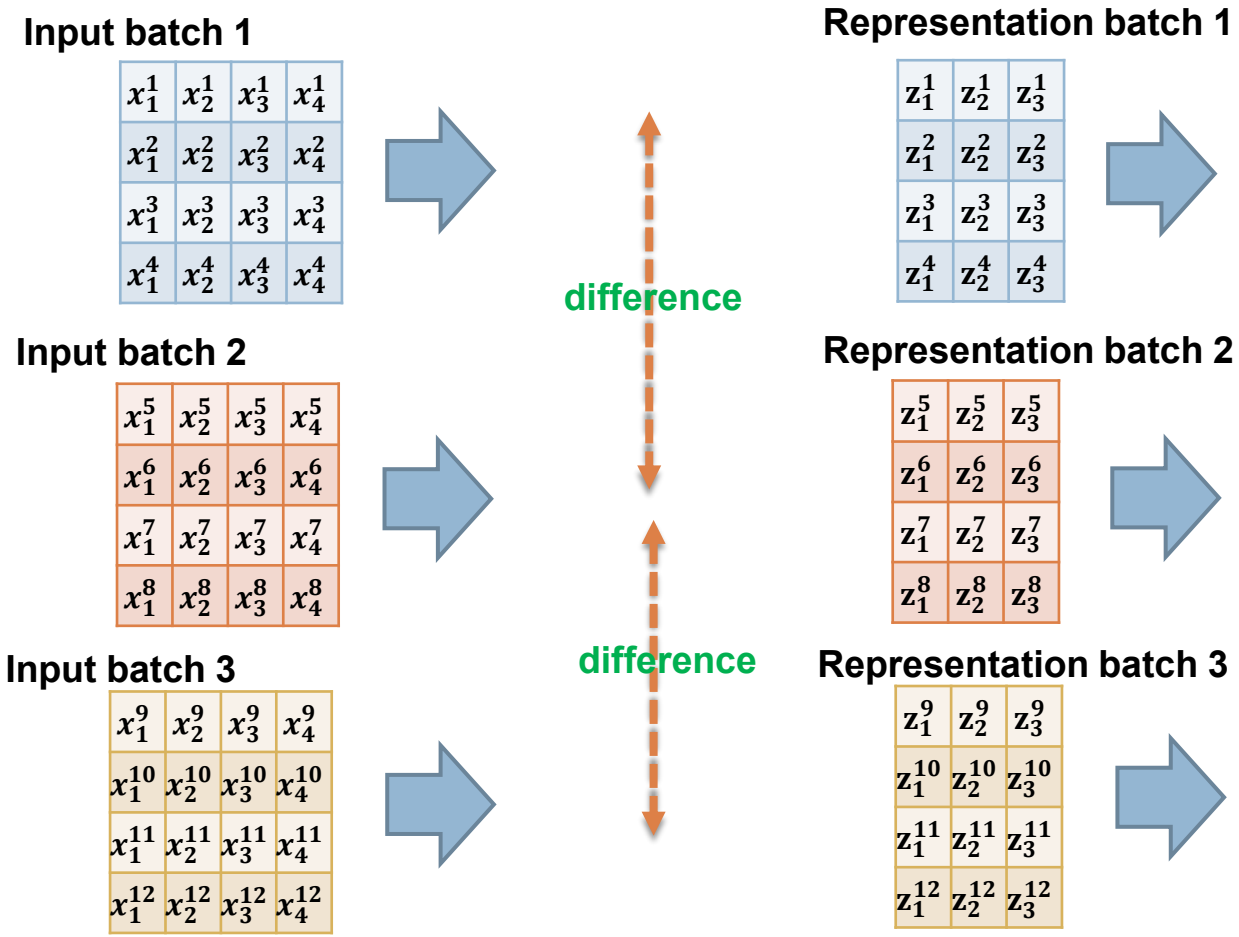
Feature extractor **Classifier**

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

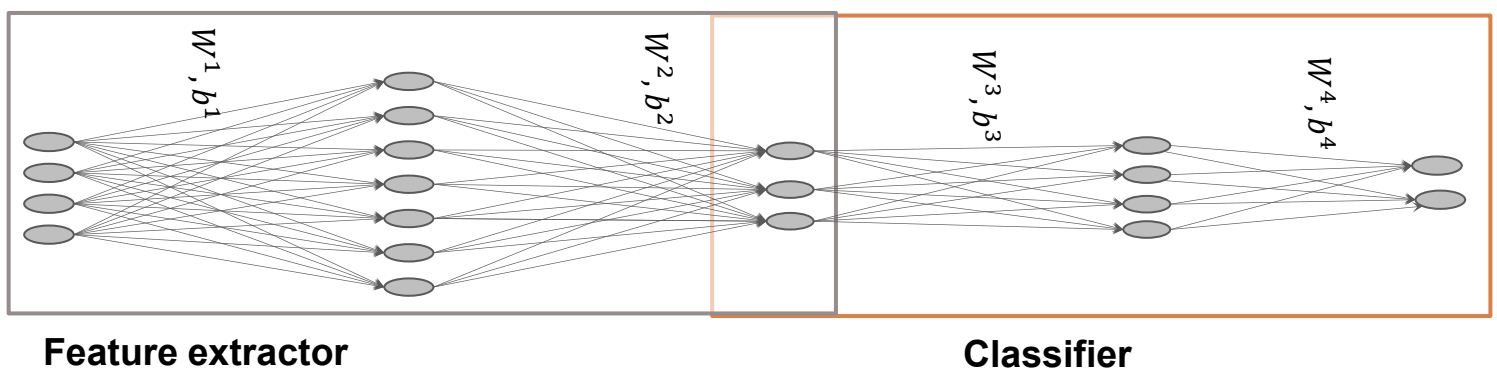
Sergey Ioffe SIOFFE@GOOGLE.COM
Christian Szegedy SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

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

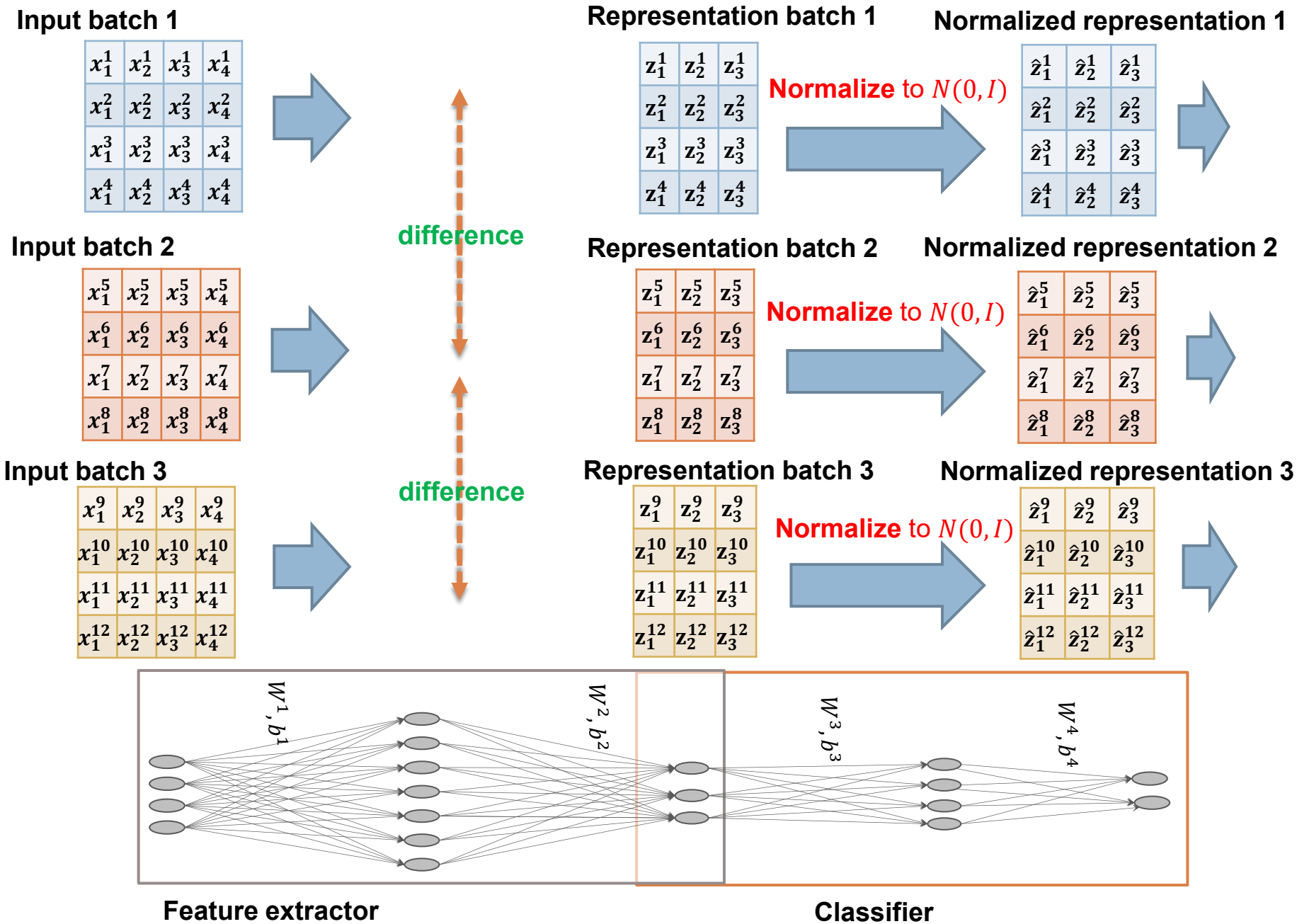
Internal covariate shift problem



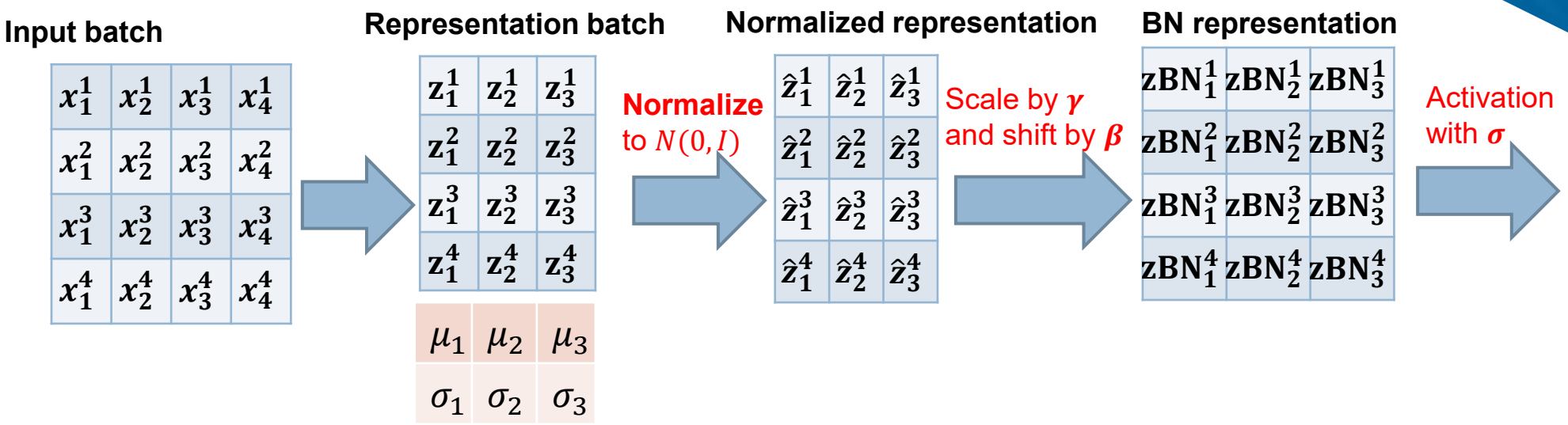
- Internal covariate shift problem:
  - Significant difference in statistics of input batches and also representation batches.
  - Statistical differences among mini-batches make the classifier harder to learn from data.

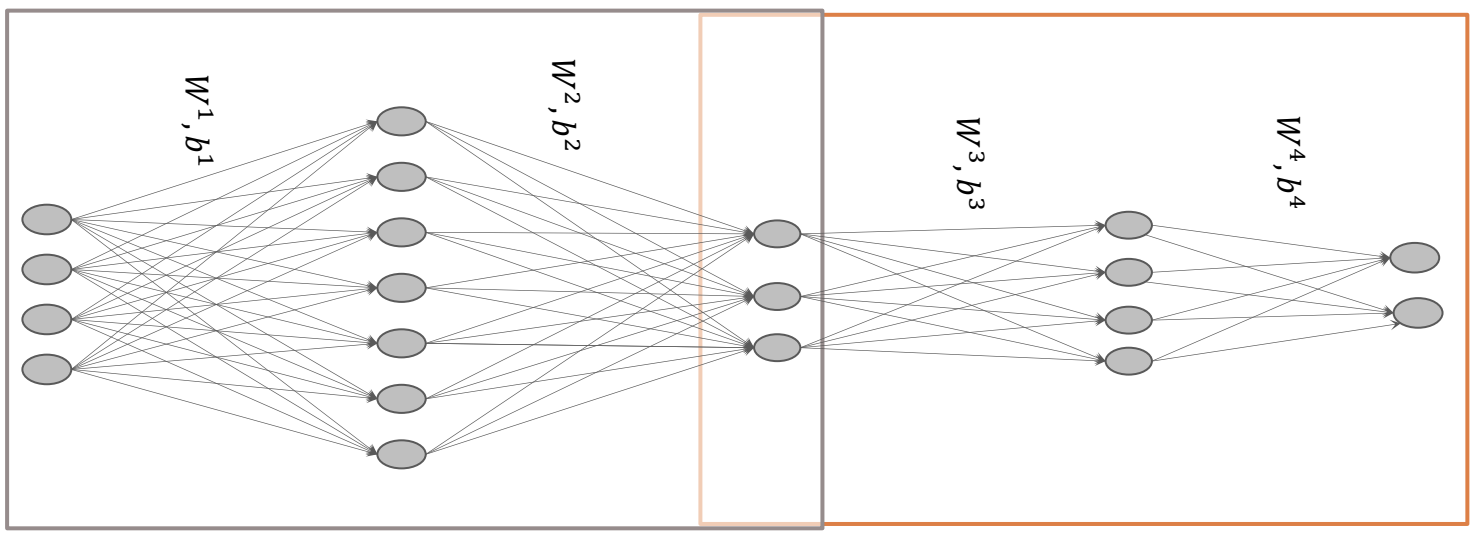


Batch Normalization



Batch Normalization

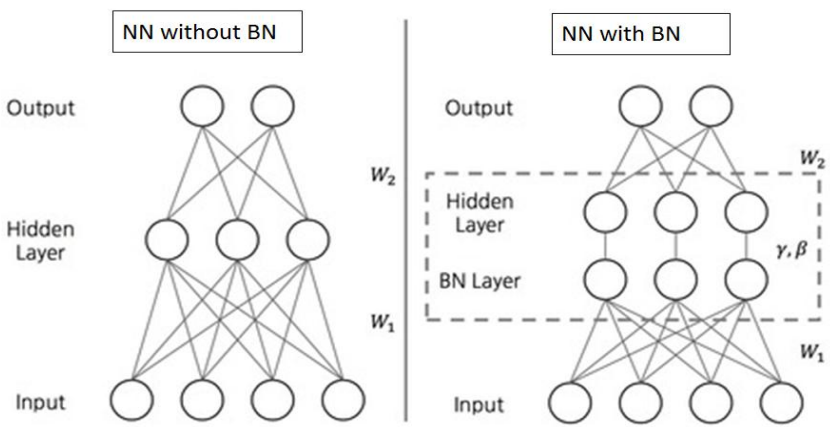


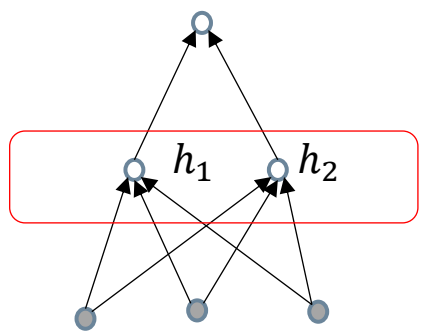


Batch Normalization

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





z_1	z_2
0.1	0.2
…	…
0.5	0.4

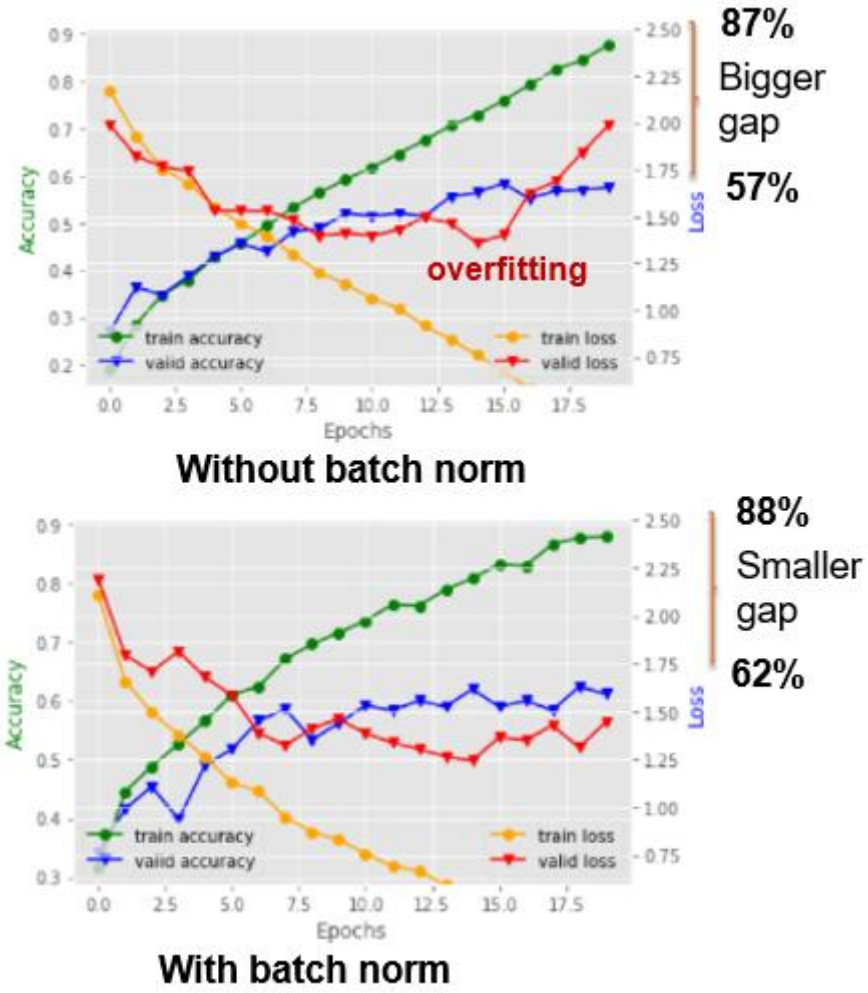
Mini-batch size

Standardize column-wise

Batch Normalization

Testing Phase

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



Data Augmentation

Reality

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

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

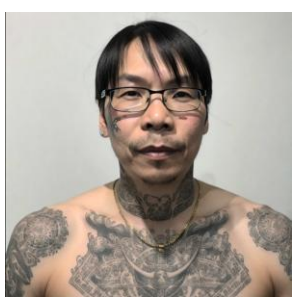
Quantity?

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

Quality?

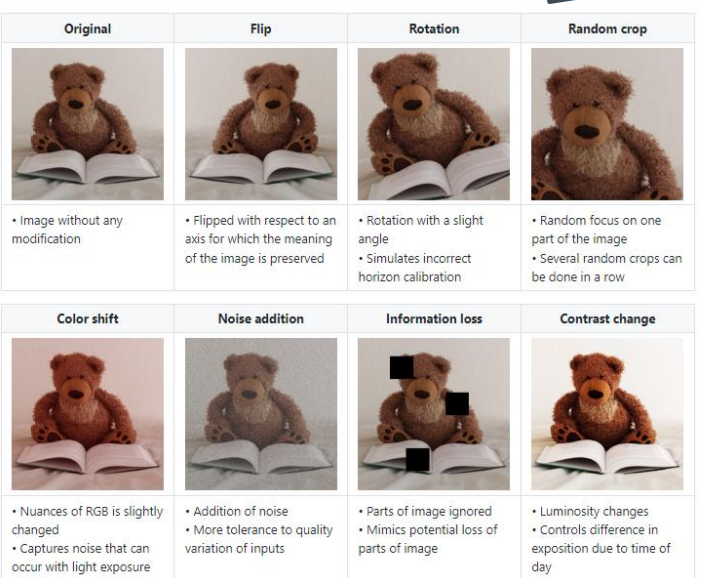
- More **diverge** data?
- More **qualified** data?

Data Augmentation



Create many variants

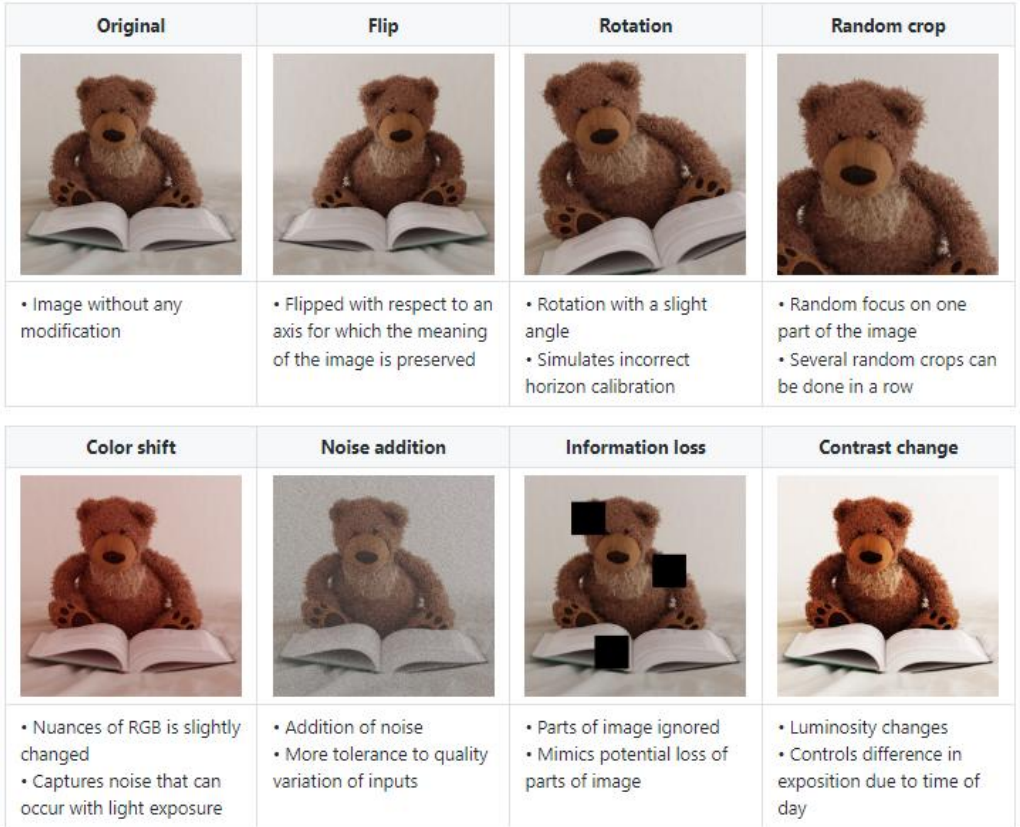
Prof Dinh Phung



Data Augmentation

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

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

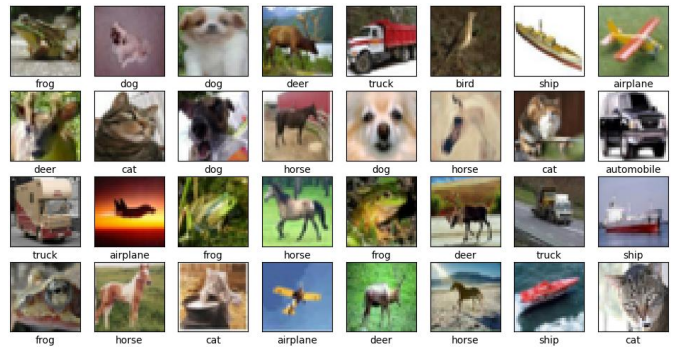


[Source: Stanford CS 230 Deep Learning]

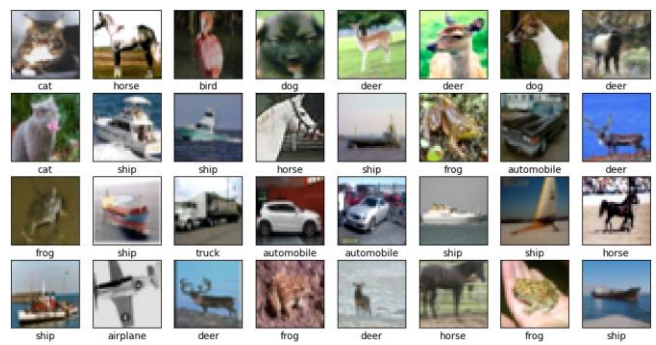
Data Augmentation in PyTorch

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

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

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

n_train, n_valid, n_test = 5000, 5000, 5000

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

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

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

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

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

Without augmentation



With augmentation

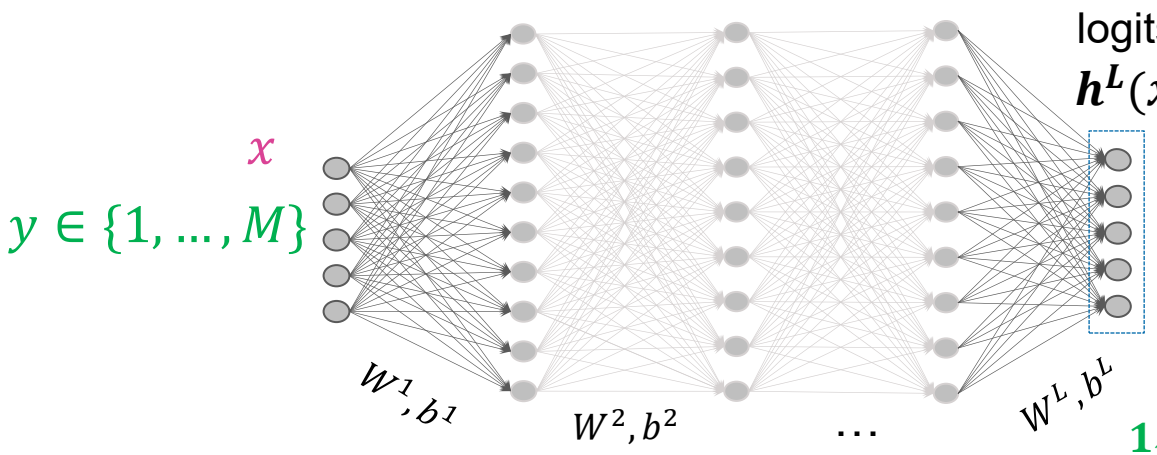


Label smoothing

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

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

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

$\boldsymbol{y}$-**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

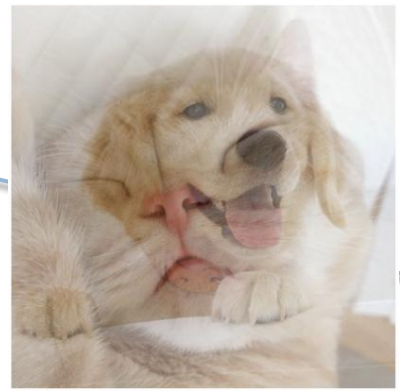
Published as a conference paper at ICLR 2018

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

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



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

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

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

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

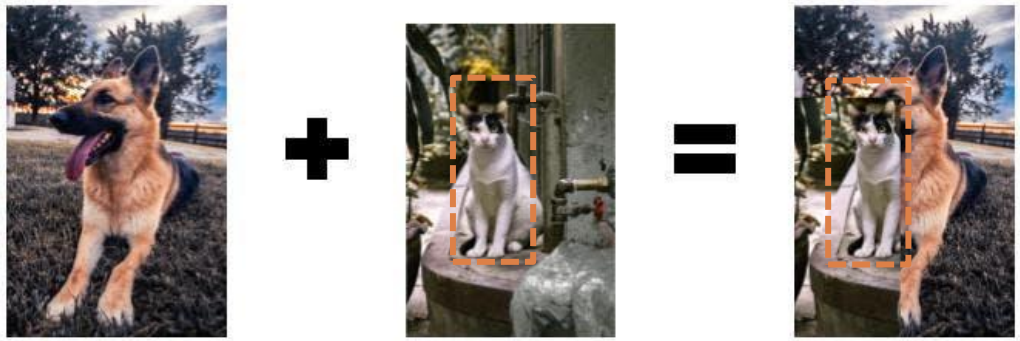
Blended label $\tilde{y} = \lambda \times \mathbf{1}_{y_1} + (1 - \lambda) \times \mathbf{1}_{y_2}$

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

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

Cut-mix

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



$(x_1, \mathbf{1}_{y_1}), x_1 \in \mathbb{R}^{C\times W\times H}$ $(x_2, \mathbf{1}_{y_2}), x_2 \in \mathbb{R}^{C\times W\times H}$ $\tilde{x}$

$$\lambda \sim \text{Beta}(\alpha, \alpha)$$
$$\tilde{x} = M \odot x_1 + (1 - M) \odot x_2, M \in \{0,1\}^{H\times W}$$
$$\tilde{y} = \lambda \times \mathbf{1}_{y_1} + (1 - \lambda) \times \mathbf{1}_{y_2}$$

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

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

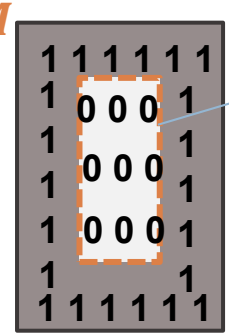
CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features

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

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

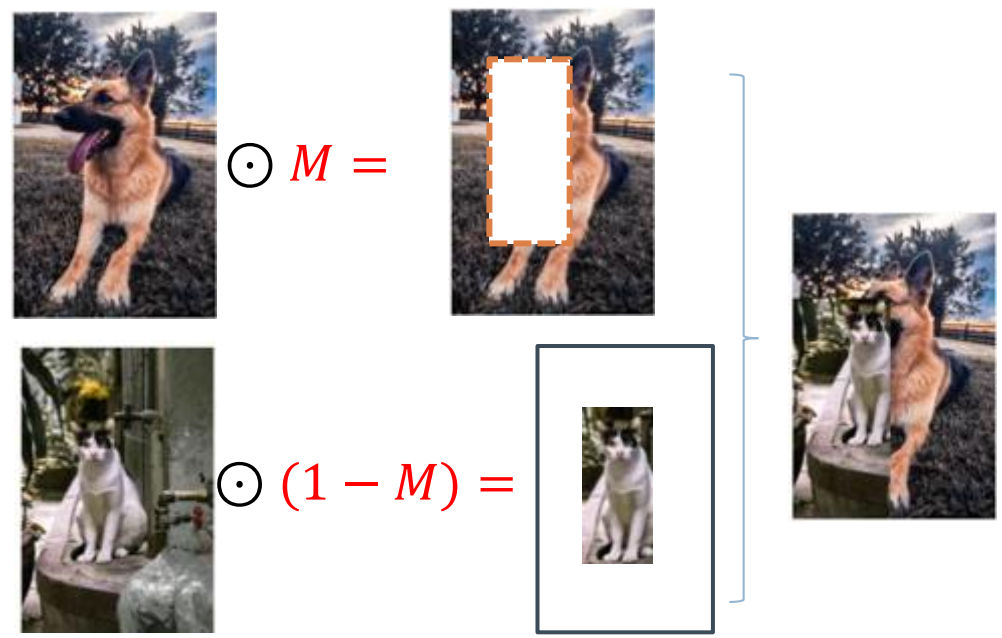
[Source: https://arxiv.org/pdf/1905.04899]

M



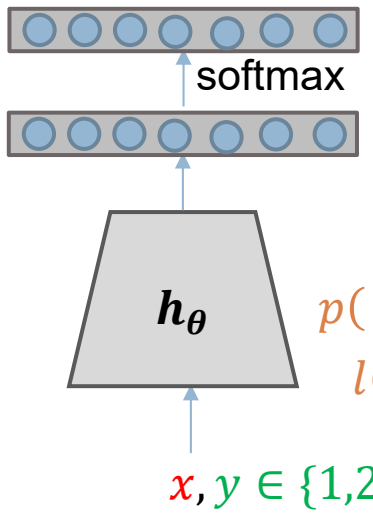
$B = [r_x, r_y, r_w, r_h]$

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



Adversarial Attack and Defense

Some notions



$f(x;\theta) = softmax(h(x;\theta))$ (prediction probabilities)

$h(x;\theta)$ (logits)

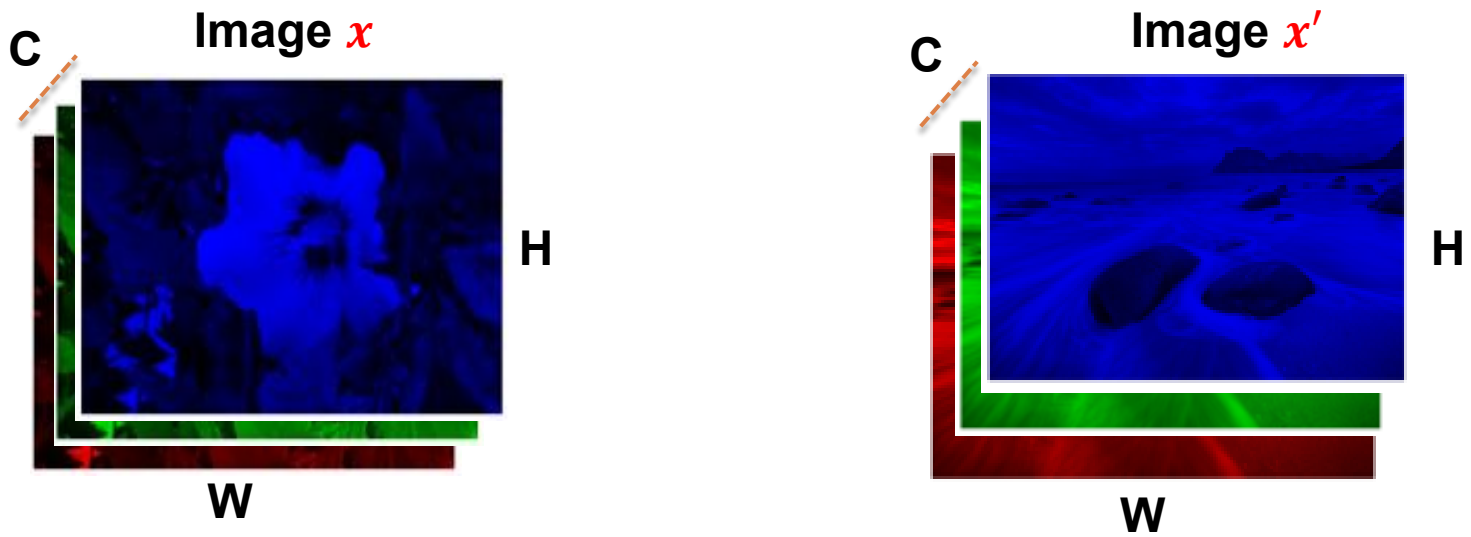
$$f(x;\theta) = [f_m(x;\theta)]_{m=1}^{M}$$

$$p(y = m \mid x) = f_m(x;\theta) \; for \; 1 \le m \le M$$

$$l(f(x;\theta), y) = CE\left(\mathbf{1}_y, f(x;\theta)\right) = -\log f_y(x;\theta)$$

$x, y \in \{1,2,\dots,M\}$

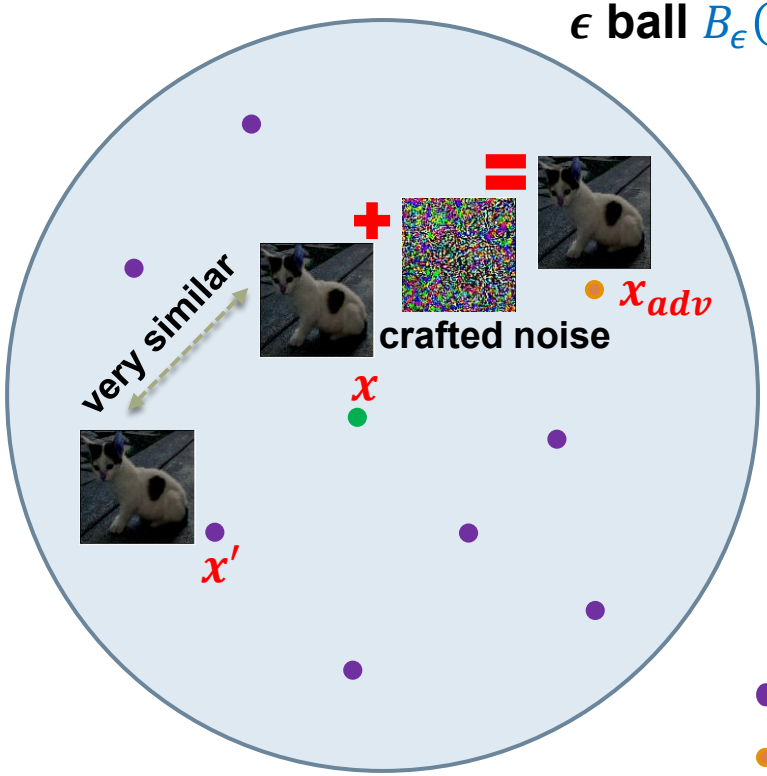
Distance between two images

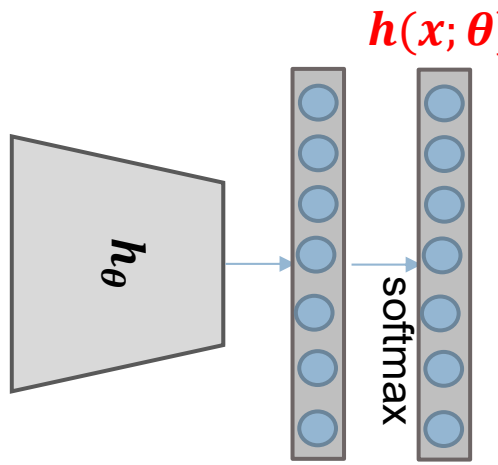


- x_{ijk} $(1 \leq i \leq C\ , 1 \leq j \leq H, 1 \leq k \leq W)$ is **the pixel** with coordinate $(height = j, width = k)$ on the channel (or slice) i of image x.
  - $0 \leq x_{ijk} \leq 255$ (original colour image) or $0 \leq x_{ijk} \leq 1$ (grey-scale image)
- L_2 distance
  - $\|x - x'\|_2 = \sqrt{\sum_{i=1}^{C} \sum_{j=1}^{H} \sum_{k=1}^{W} \left(x_{ijk} - x'_{ijk}\right)^2}$
- L_1 distance
  - $\|x - x'\|_1 = \sum_{i=1}^{C} \sum_{j=1}^{H} \sum_{k=1}^{W} |x_{ijk} - x'_{ijk}|$
- L_∞ distance
  - $\|x - x'\|_\infty = \max_{1 \leq i \leq C, 1 \leq j \leq H, 1 \leq k \leq W} |x_{ijk} - x'_{ijk}|$

Adversarial examples

ϵ ball $B_\epsilon(x) = \{x' : \|x' - x\| \leq \epsilon\}$ with a **small** $\epsilon > 0$.





$$f(x;\theta) = softmax(h(x;\theta))$$

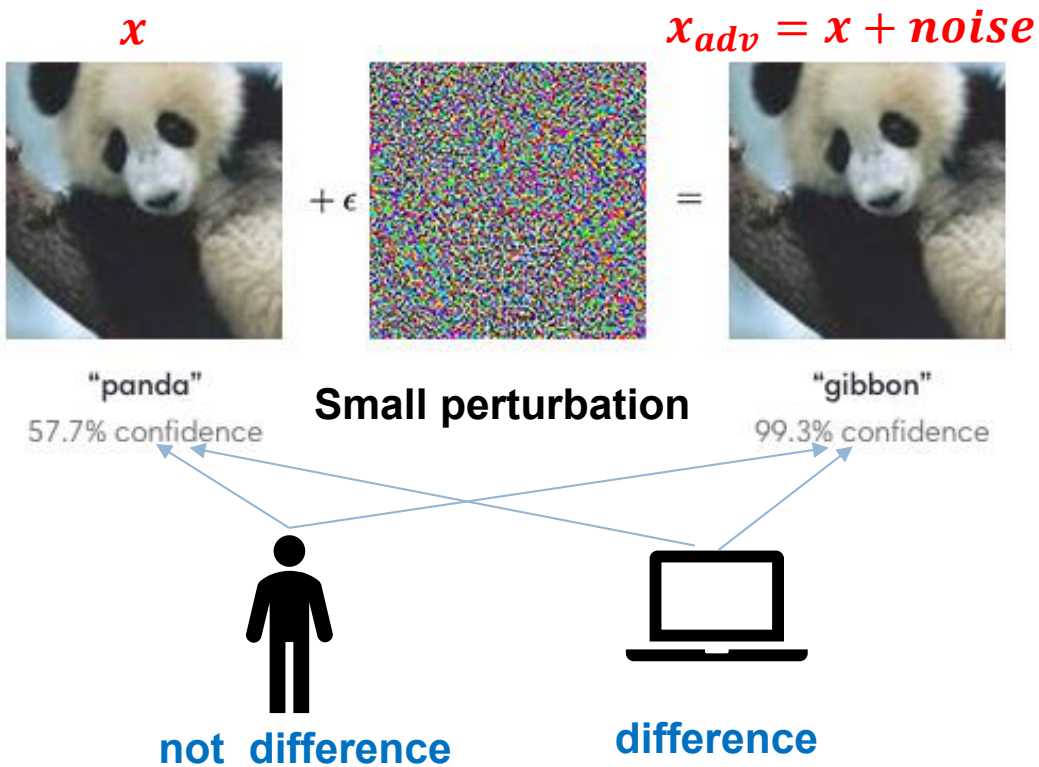
$$argmax_{1\leq i\leq M} f_i(x;\theta) = \hat{y} \neq \hat{y}_{adv} = argmax_{1\leq i\leq M} f_i(x_{adv};\theta)$$

- **Good** example which has **same predicted label** as x.
- **Adversarial** example which has **different predicted label** as x.

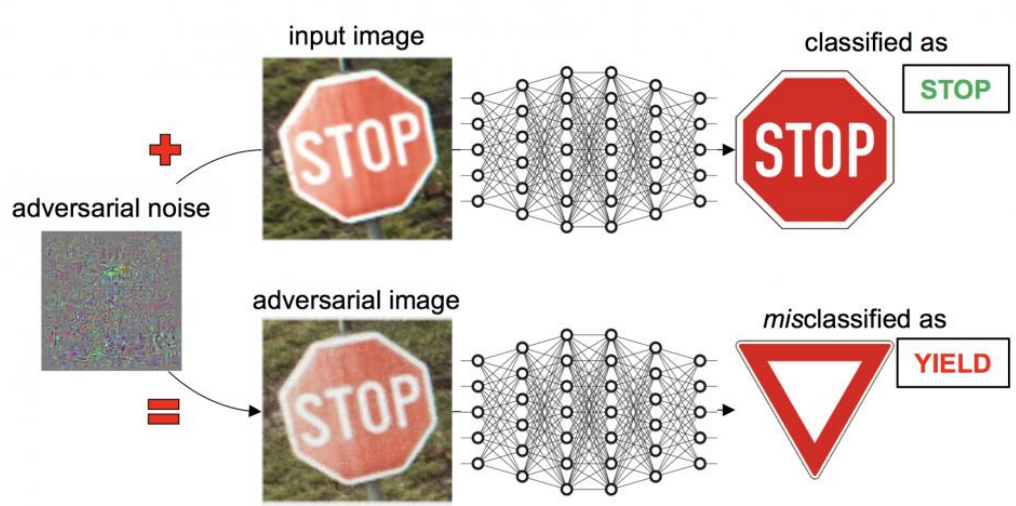
- The **true sense** of a **robust model**
  - If x' and x are **close**, the **predictions** of x' and x must be the **same**.
  - If $x' \in B_\epsilon(x)$ (i.e., $\|x' - x\| \leq \epsilon$), $argmax_{1\leq i\leq M} f_i(x;\theta) = \hat{y} = \hat{y}' = argmax_{1\leq i\leq M} f_i(x';\theta)$.
- Unfortunately, we can **easily** find **adversarial examples** $x_{adv} \in B_\epsilon(x)$ that can **fool** the classifier
  - $argmax_{1\leq i\leq M} f_i(x;\theta) = \hat{y} \neq \hat{y}_{adv} = argmax_{1\leq i\leq M} f_i(x_{adv};\theta)$

Adversarial examples

(Source: OpenAI)



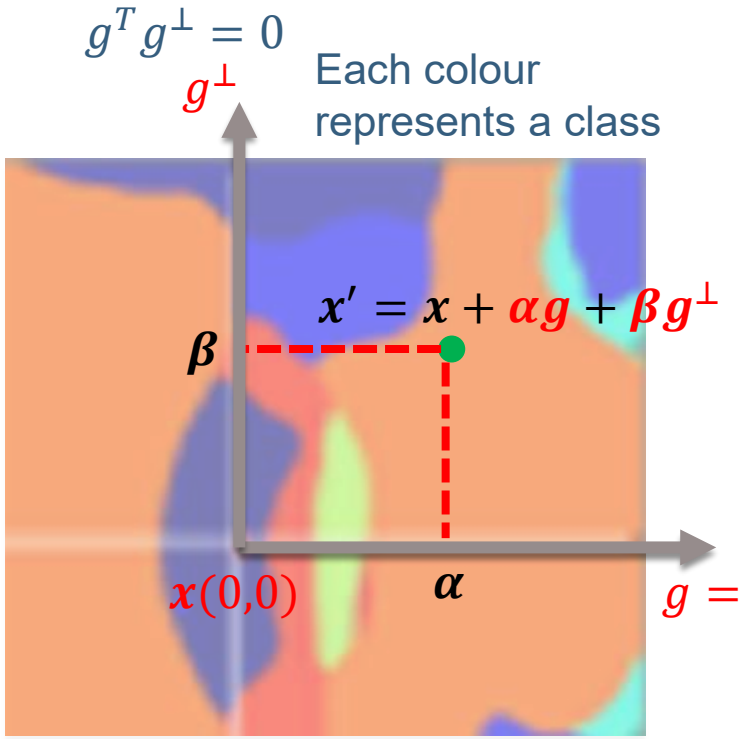
(Source: Internet)

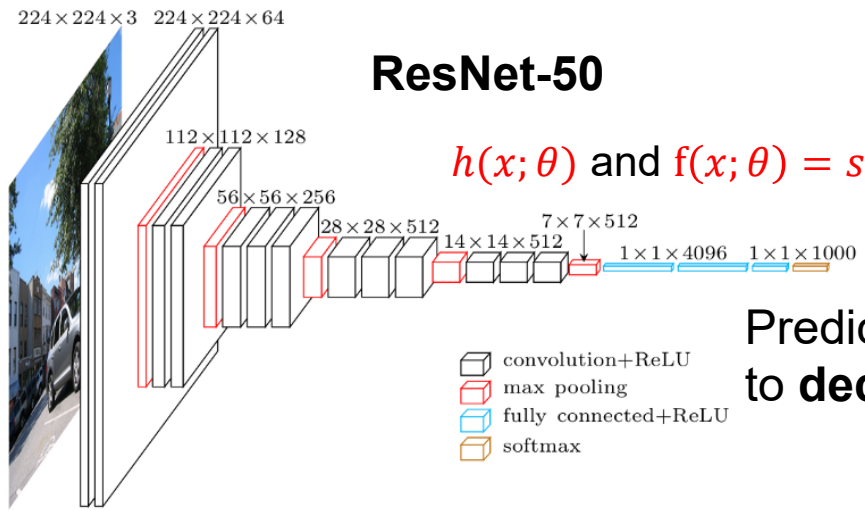


Misleading autonomous car driving system

- Adversarial example $\boldsymbol{x_{adv}}$ of a **clean data example** $\boldsymbol{x}$ w.r.t a model $f(.;\theta)$
  - $d(x_{adv}, x) = \|x_{adv} - x\| \leq \epsilon$ where $\|.\|$ is a norm (e.g., $\|.\|_1, \|.\|_2, \|.\|_\infty$)
  - f predicts x and x_{adv} with **different labels**, i.e., $argmax_i f_i(x;\theta) \neq argmax_i f_i(x_{adv};\theta)$

Visualization of decision regions





ResNet-50

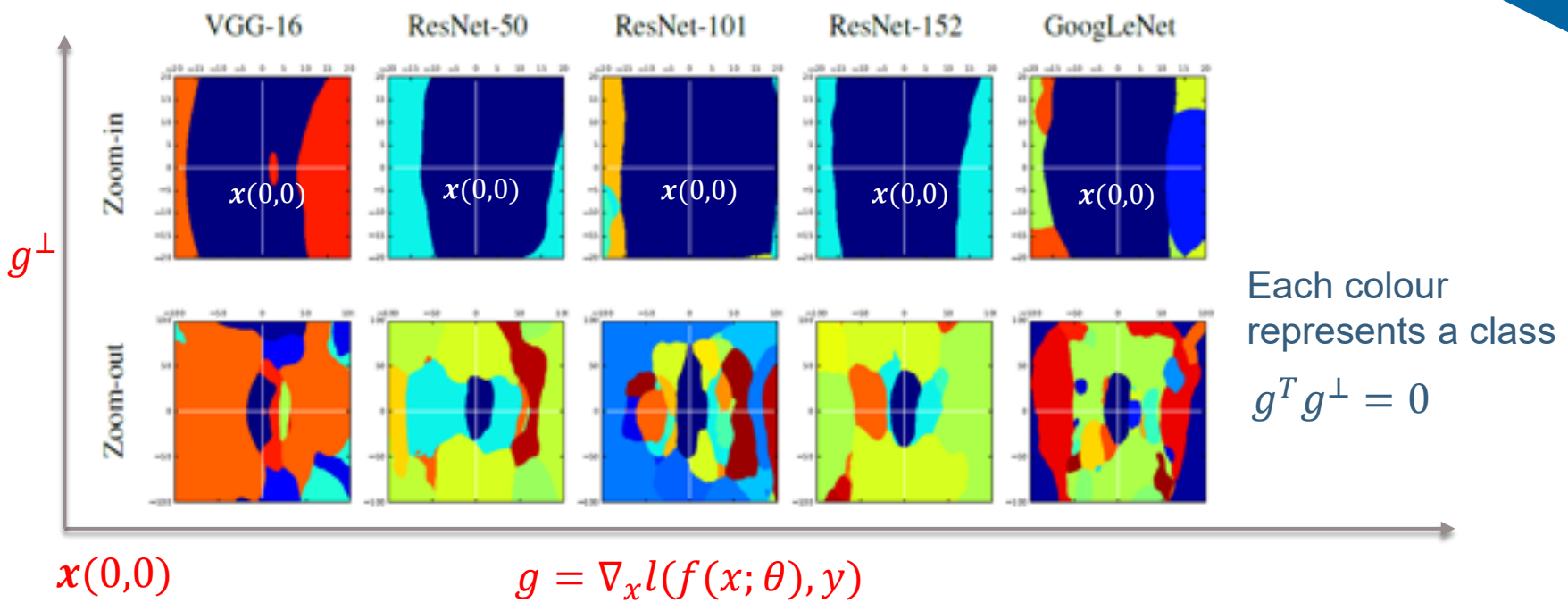
$h(x;\theta)$ and $f(x;\theta) = softmax(h(x;\theta))$

Predicts $x' = x + \alpha g + \beta g^\perp$ to **decide** the **colour**

Following the **gradient direction** is easy to **get out the blue region**.

- Start from $x = x + 0g + 0g^\perp$ (coordinate $(0,0)$), if we **go along with** the **gradient direction** $g = \nabla_x l(f(x;\theta), y)$, we can reach $x_{adv} = x + \alpha g + 0g^\perp$ such that
  - x_{adv} is very close to x → human cannot tell how they **are different** → the **same class** from human **perceptron**
  - x_{adv} is predicted to **different class** from x by the classifier
  - x_{adv} is called **adversarial example** of x

Decision regions of deep learning model



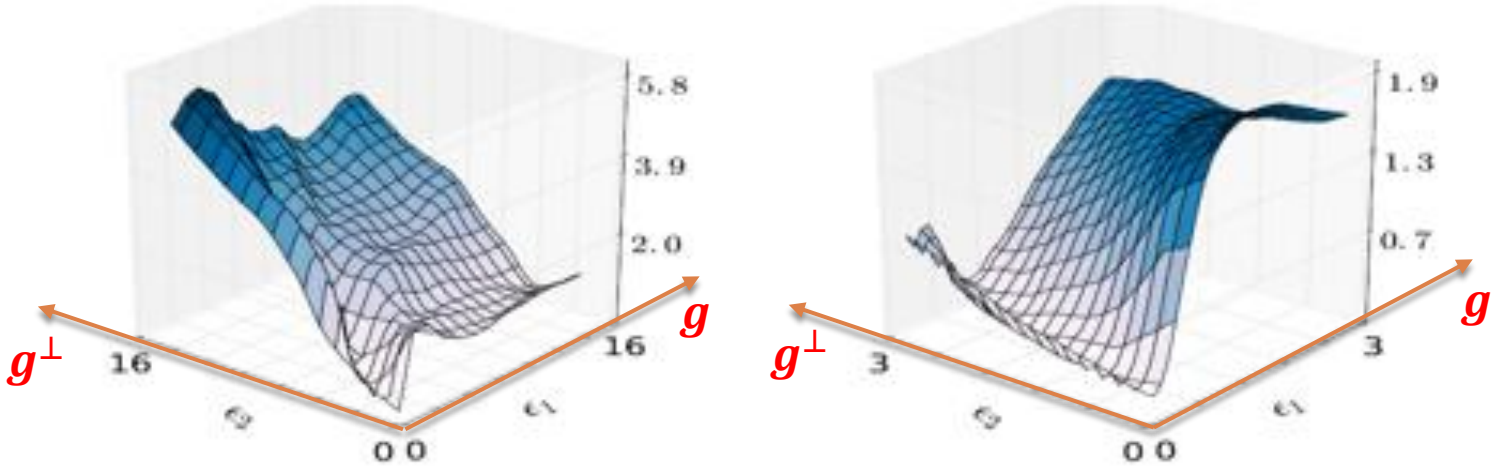
Each colour represents a class

$g^T g^{\perp} = 0$

- Deep learning models are **fragile** and **easy to be attacked**
  - Start from $x = x + 0g + 0g^{\perp}$ (coordinate $(\mathbf{0}, \mathbf{0})$), if we go along with the gradient direction $g = \nabla_x l(f(x;\theta), y)$, we can reach $x_{adv} = x + \alpha g + 0g^{\perp}$ such that
    - x_{adv} is very close to x → human cannot tell how they **are different** → the **same class** from human **perceptron**
    - x_{adv} is predicted to **different class** from x by the classifier
    - x_{adv} is called **adversarial example** of x

Delving into Transferable Adversarial Examples and Black-box Attacks
Yanpei L., et al., ICLR 2016

Loss surface of deep learning model



Zoom out **Zoom in**

The **loss surface** of DL model **around** x. This is plotted on $x' = x + \epsilon_1 g + \epsilon_2 g^{\perp}$.

- At the **data point** x, if we move along with the **gradient direction** g, the **loss function** is **maximally locally increased**.
- **Taylor expansion** around x
  - $l(f(x+h;\theta),y) \approx l(f(x;\theta),y) + g^T h$ where $g = \nabla_x l(f(x;\theta),y)$.
  - The gradient direction is the **steepest direction** to increase the loss function locally.

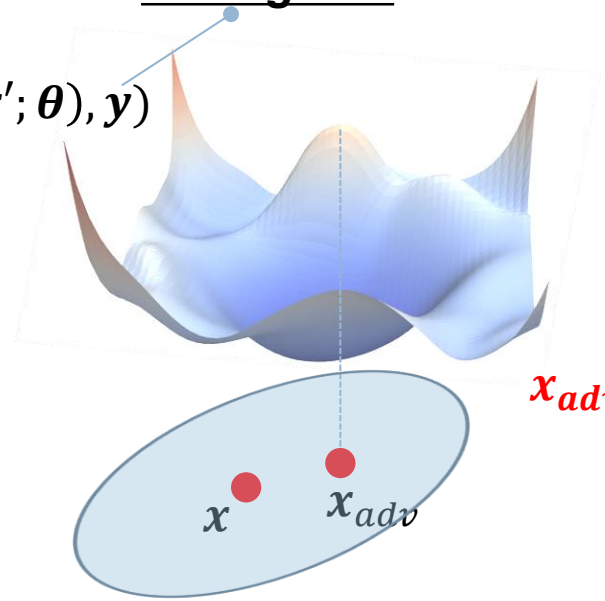
Ensemble Adversarial Training Attacks and Defences
Tramer et al., ICLR 2017

Adversarial attack

How to run untargeted attack?

***Untargeted* attacks**

Loss surface $\boldsymbol{l(f(x';\theta),y)}$



$B_\epsilon(x) = \{x' : \|x' - x\| \leq \epsilon\}$

$\boldsymbol{x}$ has true label $\boldsymbol{y} \in \{\boldsymbol{1}, \boldsymbol{2}, \ldots, \boldsymbol{M}\}$

$$\boldsymbol{x_{adv} = argmax_{x' \in B_\epsilon(x)} l(f(x';\theta),y)}$$

- Fast Gradient Sign Method (FGSM)
  - $x_{adv} = x + \epsilon sign(\nabla_x l(f(x;\theta),y))$
  - $sign(t) = \begin{cases} 1 & if\ t > 0 \\ -1 & if\ t < 0 \\ 0 & otherwise \end{cases}$
  - One-step update

Explaining and Harnessing Adversarial Examples
Goodfellow et al., ICLR 2015

- Projected Gradient Descent (Ascent) (PGD)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $\tilde{x}_{t+1} = x_t + \eta sign(\nabla_x l(f(x_t;\theta),y))$
  - $x_{t+1} = Proj_{B_\epsilon(x)}(\tilde{x}_{t+1})$
  - Run in k steps ($k = 20$), $\eta > 0$ is the learning rate
  - $x_{adv} = x_k$

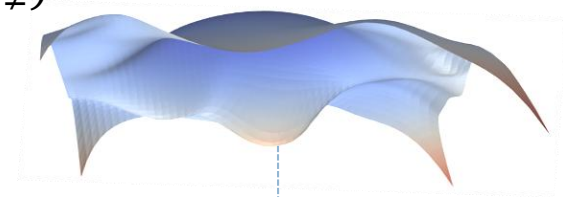
Towards Deep Learning Models Resistant to Adversarial Attacks
Madry et al., ICLR 2017

Adversarial attack

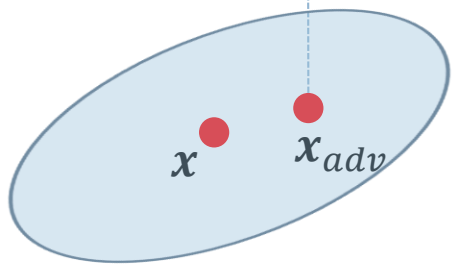
How to run targeted attack?

***Targeted* attacks**

Loss surface $\boldsymbol{l(f(x';\theta), y_{\neq})}$
$\boldsymbol{y_{\neq}}$ is different from **y**



$B_\epsilon(x) = \{x' : \|x' - x\| \leq \epsilon\}$
$\boldsymbol{x}$ has true label $\boldsymbol{y} \in \{\mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{M}\}$



$$\boldsymbol{x_{adv} = argmin_{x' \in B_\epsilon(x)} l(f(x';\theta), y_{\neq})}$$

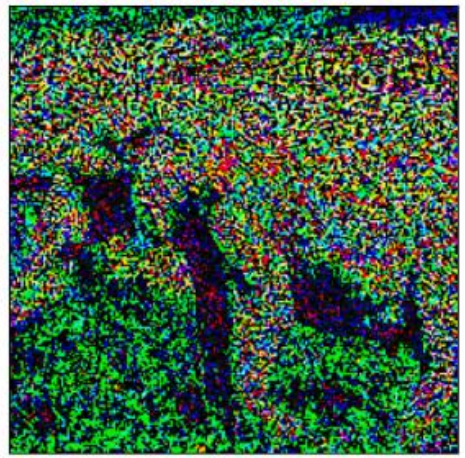
- Fast Gradient Sign Method (FGSM)
  - $x_{adv} = x - \epsilon sign(\nabla_x l(f(x;\theta), y_\#))$
  - $sign(t) = \begin{cases} 1 & if\ t > 0 \\ -1 & if\ t < 0 \\ 0 & otherwise \end{cases}$
  - One-step update

- Projected Gradient Descent (Ascent) (PGD)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $\tilde{x}_{t+1} = x_t - \eta sign(\nabla_x l(f(x_t;\theta), y_\#))$
  - $x_{t+1} = Proj_{B_\epsilon(x)}(\tilde{x}_{t+1})$
  - Run in k steps ($k = 20$), $\eta > 0$ is the learning rate
  - $x_{adv} = x_k$

Adversarial Examples and Perturbations

```
attack_types = ['fgsm', 'trades', 'pgd']
attack_type = attack_types[2]

x_pgd = attack(attack_type, vgg19, batch_t, None, epsilon=0.01, num_steps=10, step_size=0.002, clip_value_min=-255.0, clip_value_max=255.0)
```

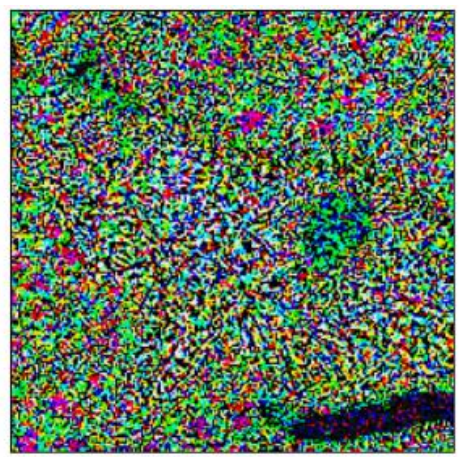
Original image: tusker



Noise

Adversarial image: African bush elephant

Original image: Siamese cat

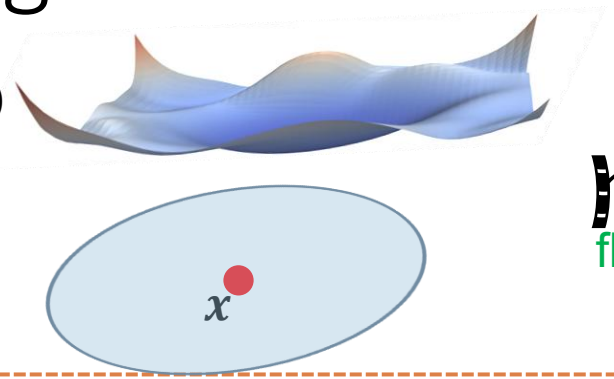


Noise

Adversarial image: borzoi

Adversarial training

Loss surface $l(f(x';\theta),y)$



We wish a **smooth** (or flat) **loss surface** around x

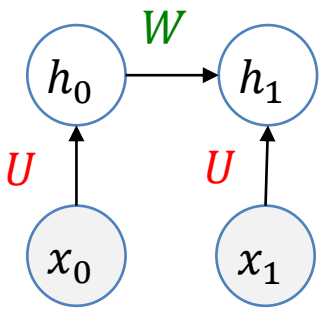
- The **ideal optimization problem** for adversarial training
  - $\min_{\theta} \mathbb{E}_{(x,y)\sim p(x,y)} \left[\max_{x' \in B_\epsilon(x)} l(f(x';\theta),y) \right]$
  - $p(x,y)$ is the **data distribution** that generates the pair of data point x and label y
  - Minimize the loss of over **most violated** adversarial examples

- **PGD adversarial training**
  - for epoch in n_epochs
    - for iter in range(n_iter_per_epoch)
      - Sample mini-batch $(x_1,y_1),\dots,(x_b,y_b)$ from the training set
      - Find PGD untargeted adversarial examples $x_1^{adv},\dots,x_b^{adv}$ for $x_1,\dots,x_b$ w.r.t labels $y_1,\dots,y_b$
      - $batch\_loss = \frac{1}{b}\sum_{i=1}^{b} l(f(x_i;\theta),y_i) + \frac{1}{b}\sum_{i=1}^{b} l(f(x_i^{adv};\theta),y_i)$
      - $\theta = \theta - \eta \frac{\partial batch\_loss}{\partial \theta}$

Recurrent Neural Networks

Recurrent Neural Networks

Simplest RNN with two time-slices (no output)



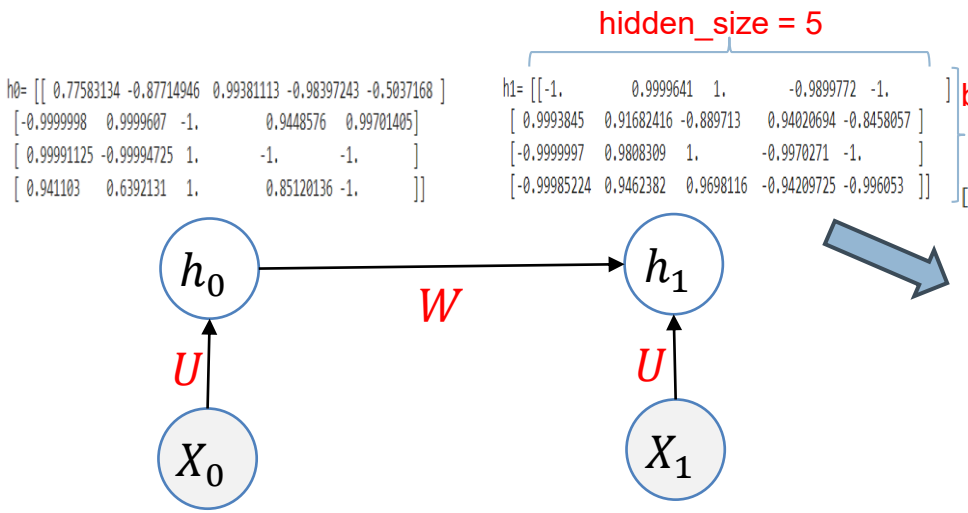
values known from data

hidden nodes, values to be inferred from our models

- Input: $x_0 \in \mathbb{R}^{1\times input\_size}$, $x_1 \in \mathbb{R}^{1\times input\_size}$
- $h_0 = \tanh(x_0 U + b) \in \mathbb{R}^{1\times hidden\_size}$
  - $U \in \mathbb{R}^{input\_size\times hidden\_size}$
- $h_1 =$ some function of h_0 and x_1

  $= \tanh(h_0 W + x_1 U + b) \in \mathbb{R}^{1\times hidden\_size}$
  - $W \in \mathbb{R}^{hidden\_size\times hidden\_size}$

Recurrent Neural Networks



[batch_size, seq_len, hidden_size]

```
[[[-0.9846, -0.9911,  0.9085, -0.5241,  0.3290],
  [ 1.0000, -1.0000, -1.0000, -1.0000,  1.0000]],

 [[ 0.9923,  1.0000, -0.9995, -0.8480, -0.9999],
  [-0.5447,  0.9360, -0.2284,  0.8301, -0.8126]],

 [[-0.9986, -1.0000,  1.0000, -1.0000,  1.0000],
  [ 1.0000, -0.9935, -0.9967, -1.0000,  1.0000]],

 [[ 1.0000, -0.9928, -0.9245, -0.9976,  1.0000],
  [ 1.0000,  0.9979, -0.9997, -1.0000, -0.9000]]]
```

- Input: $X_0, X_1 \in \mathbb{R}^{bs \times input\_size}$
- $h_0 = \tanh(X_0 U + b) \in \mathbb{R}^{bs \times hidden\_size}$
  - $U \in \mathbb{R}^{input\_size \times hidden\_size}$
- $h_1 = \tanh(h_0 W + X_1 U + b) \in \mathbb{R}^{1 \times hidden\_size}$

```
[[0.0, 1.0, -2.0],
 [-3.0, 4.0, 5.0],
 [6.0, 7.0, -8.0],
 [6.0, -1.0, 2.0]]
```

input_size = 3

```
[[9.0, 8.0, 7.0],
 [0.0, 0.0, 0.0],
 [6.0, 5.0, 4.0],
 [1.0, 2.0, 3.0]]
```

batch_size = 4

[seq_len, batch_size, input_size]

`torch.transpose(X, 0, 1)`

```
X= np.stack((X0, X1), axis=0)
X = np.transpose(X, (1, 0, 2))
```

seq_len = 2

input_size = 3

batch_size = 4

```
[ 0.  1. -2.]
[ 9.  8.  7.]
[-3.  4.  5.]
[ 0.  0.  0.]
[ 6.  7. -8.]
[ 6.  5.  4.]
[ 6. -1.  2.]
[ 1.  2.  3.]
```

[batch_size, seq_len, input_size]

```
import numpy as np

X0 = np.array([[0.0, 1.0, -2.0],
               [-3.0, 4.0, 5.0],
               [6.0, 7.0, -8.0],
               [6.0, -1.0, 2.0]], dtype= np.float32) # t = 0
X1 = np.array([[9.0, 8.0, 7.0],
               [0.0, 0.0, 0.0],
               [6.0, 5.0, 4.0],
               [1.0, 2.0, 3.0]], dtype= np.float32) # t = 1
```

```
print("h1= {}".format(h1.detach().numpy()))
```

```
h1= [[-1.          0.9999641   1.         -0.9899772  -1.        ]
 [ 0.9993845   0.91682416 -0.889713   0.94020694 -0.8458057 ]
 [-0.9999997   0.9808309   1.         -0.9970271  -1.        ]
 [-0.99985224  0.9462382   0.9698116  -0.94209725 -0.996053  ]]
```

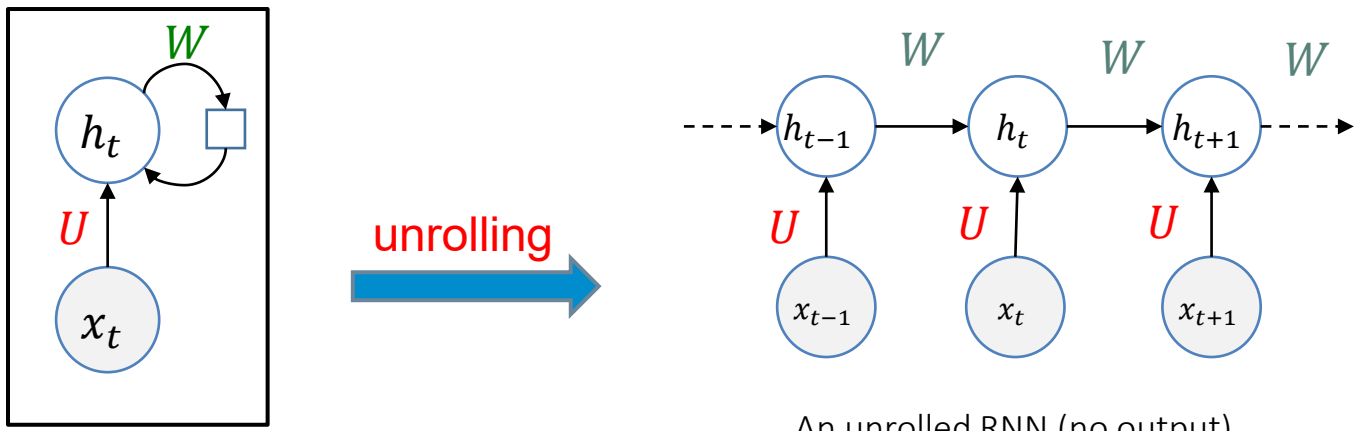
```
hidden_size = 5
input_size = 3

# Creating the parameters
U = torch.nn.Parameter(torch.randn(input_size, hidden_size, dtype=torch.float32))
W = torch.nn.Parameter(torch.randn(hidden_size, hidden_size, dtype=torch.float32))
b = torch.nn.Parameter(torch.zeros(1, hidden_size, dtype=torch.float32))

X0 = torch.tensor(X0)
X1 = torch.tensor(X1)
# Implementing the operations
h0 = torch.tanh(torch.matmul(X0, U) + b)
h1 = torch.tanh(torch.matmul(X1, U) + torch.matmul(h0, W) + b)
```

```
print("h0= {}".format(h0.detach().numpy()))
```

```
h0= [[ 0.77583134 -0.87714946  0.99381113 -0.98397243 -0.5037168 ]
 [-0.9999998   0.9999607  -1.          0.9448576   0.99701405]
 [ 0.99991125 -0.99994725  1.         -1.         -1.        ]
 [ 0.941103    0.6392131   1.          0.85120136 -1.        ]]
```

Recurrent Neural Networks

with no output



An unrolled RNN (no output)

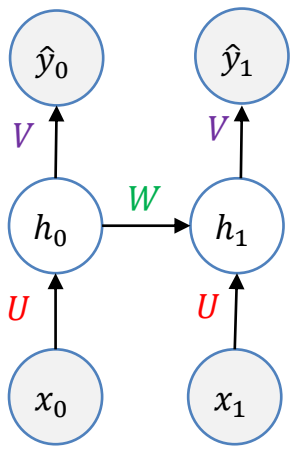
- **Idea**: *sharing parameters* for each data of the time index
- Given a data sequence $x_1, x_2, \dots, x_T$
- RNN models a dynamic system driven by an external signal x_t

$$h_t = f(h_{t-1}, x_t) = f(f(h_{t-2}, x_{t-1}), x_t) = \cdots = \text{summary}(x_{1:t}, h_0)$$

- h_t can be considered as a kind of lossy summary of the history $x_{1:t}$

Recurrent Neural Networks

Parameterization - 2 time slices

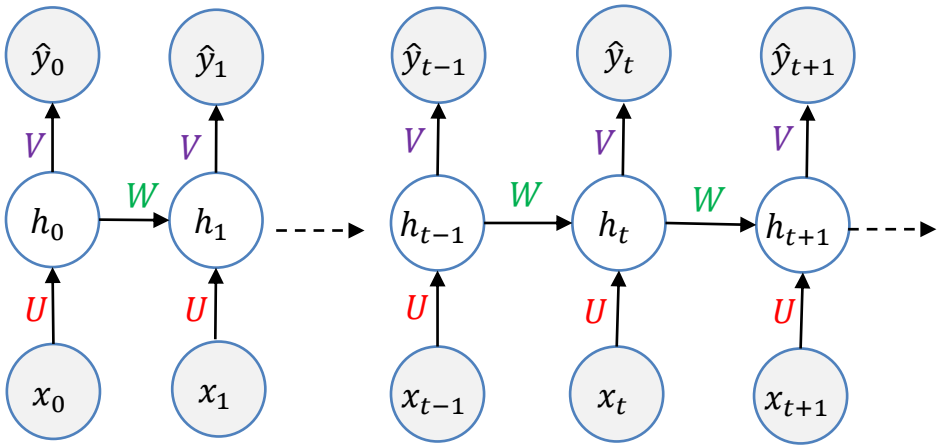


Simplest RNN with two time-slices (with output)

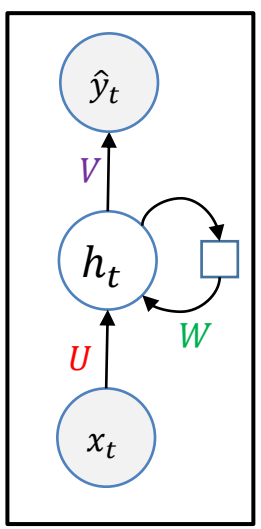
- Input: $x_0, x_1 \in \mathbb{R}^{1\times input\_size}$, $y_0, y_1 \in Y$
- $h_0 = \tanh(x_0 U + b)$
- $\hat{y}_0 = \begin{cases} h_0 V + c \quad \text{(regression)} \\ \text{softmax}\,(h_0 V + c) \quad \text{(classification)} \end{cases}$
  - Suffer loss $l(\hat{y}_0, y_0)$
- $h_1 = \text{some function of } h_0 \text{ and } x_1$
  $= \tanh(h_0 W + x_1 U + b)$
- $\hat{y}_1 = \begin{cases} h_1 V + c \quad \text{(regression)} \\ \text{softmax}\,(h_1 V + c) \quad \text{(classification)} \end{cases}$
  - Suffer loss $l(\hat{y}_1, y_1)$
- $Total\_loss = l(\hat{y}_0, y_0) + l(\hat{y}_1, y_1)$

Recurrent Neural Networks

Parametrization over multiple time slices



Simplest RNN with multiple time-slices (with output)



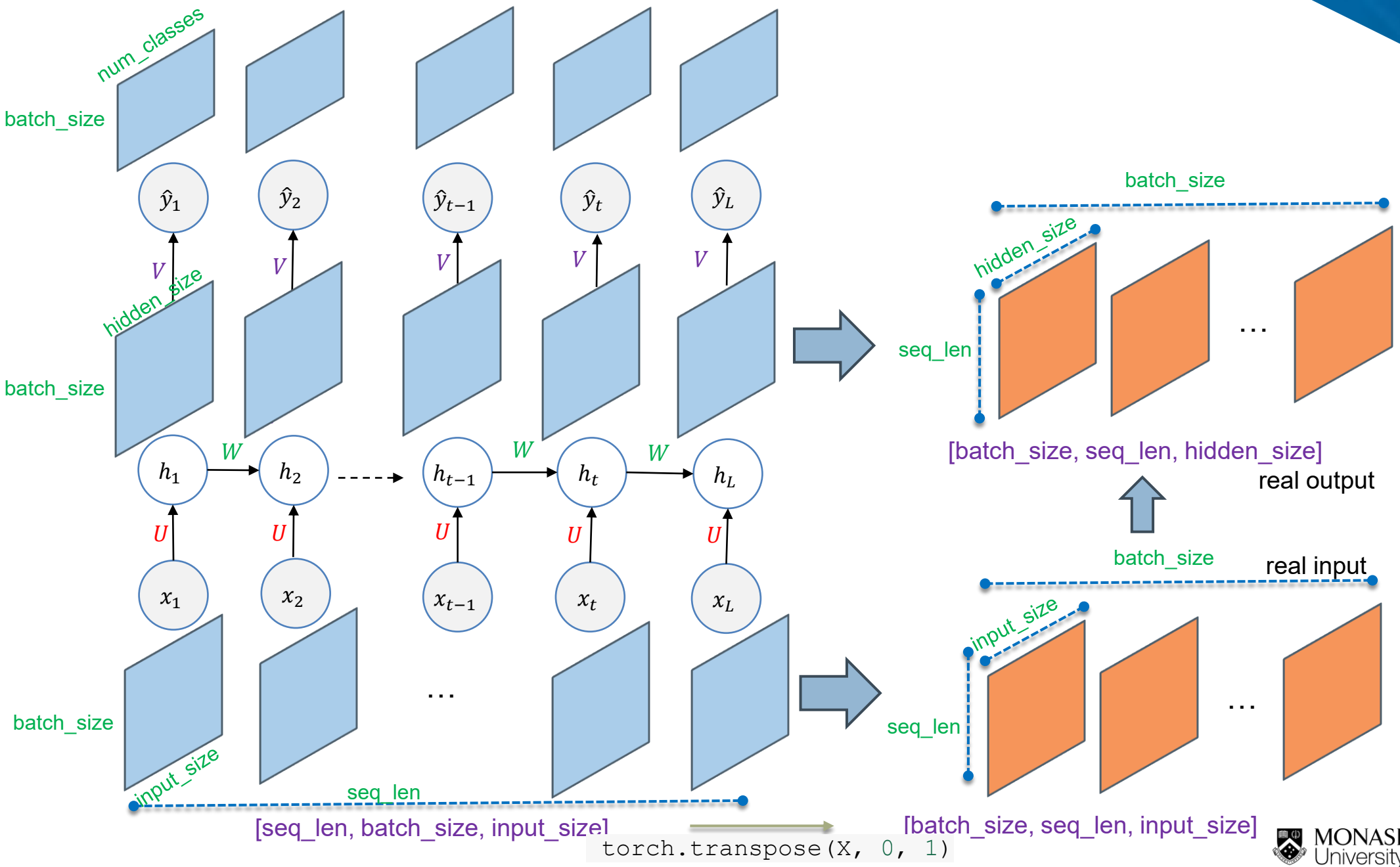
- Input: $x_0, x_1, .., x_t, \ldots \in \mathbb{R}^{1\times input\_size}$, $y_0, y_1, \ldots \in Y$
- $h_0 = \tanh(x_0 U + b)$
- $\hat{y}_0 = \begin{cases} h_0 V + c & \text{(regression)} \\ \text{softmax}\,(h_0 V + c) & \text{(classification)} \end{cases}$
  - Suffer loss $l(\hat{y}_0, y_0)$
- for t=1,2,..., T

  $h_t = \text{some function of } h_{t-1} \text{ and } x_t$

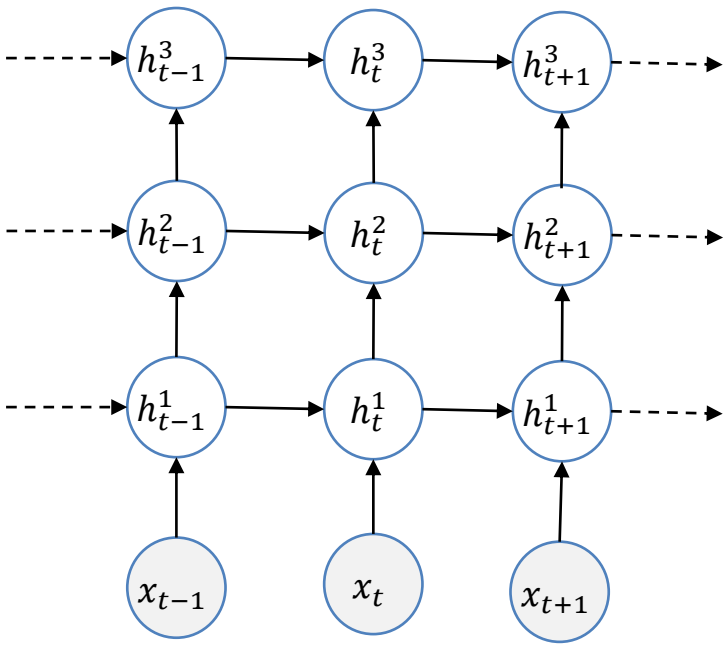
  $= \tanh(h_{t-1} W + x_t U + b)$

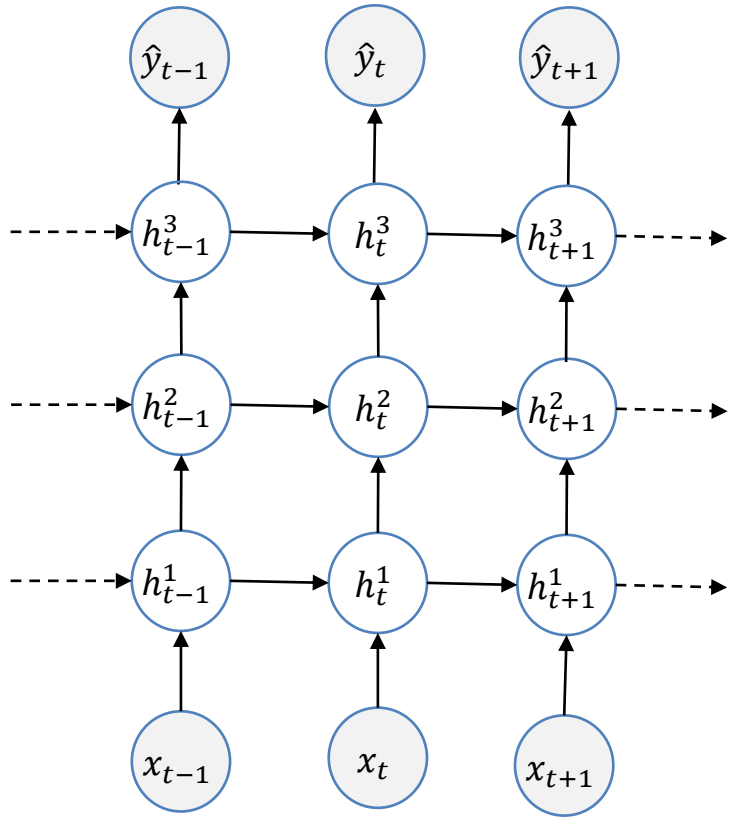
  $\hat{y}_t = \begin{cases} h_t V + c & \text{(regression)} \\ \text{softmax}\,(h_t V + c) & \text{(classification)} \end{cases}$
  - Suffer loss $l(\hat{y}_t, y_t)$
- $\boldsymbol{Total\_loss} = \sum_{t=0}^{T} l(\hat{y}_t, y_t)$

Shape Transformation in RNNs



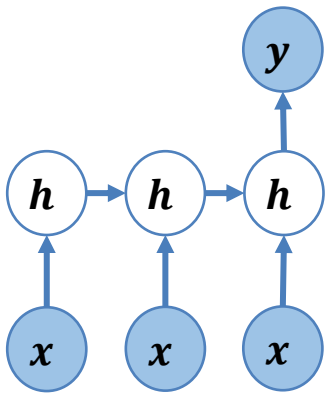
Deeper RNNs


h^3_{t-1}
h^3_t
h^3_{t+1}
h^2_{t-1}
h^2_t
h^2_{t+1}
h^1_{t-1}
h^1_t
h^1_{t+1}
x_{t-1}
x_t
x_{t+1}

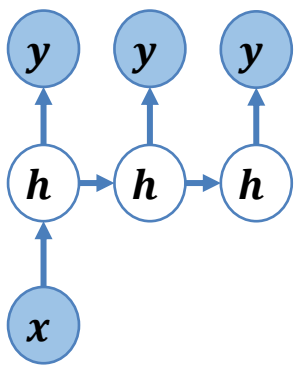

$\hat{y}_{t-1}$
$\hat{y}_t$
$\hat{y}_{t+1}$
h^3_{t-1}
h^3_t
h^3_{t+1}
h^2_{t-1}
h^2_t
h^2_{t+1}
h^1_{t-1}
h^1_t
h^1_{t+1}
x_{t-1}
x_t
x_{t+1}

RNNs Architecture Zoo

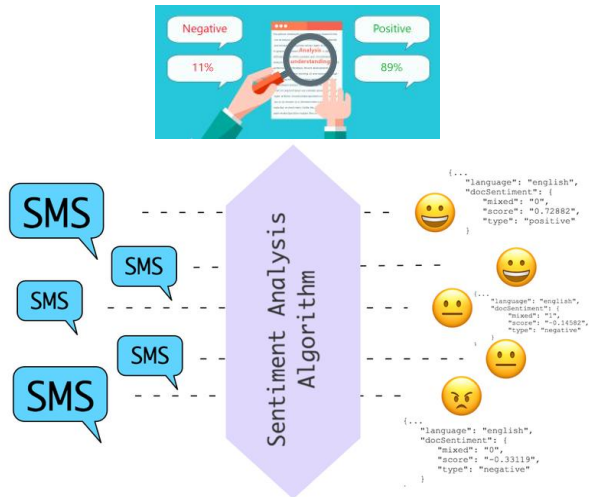
- RNN's architecture is very flexible for many real-world tasks



many to one
(sentiment analysis, image classification)



one to many
(image captioning)



Sentiment analysis
[https://www.twilio.com]

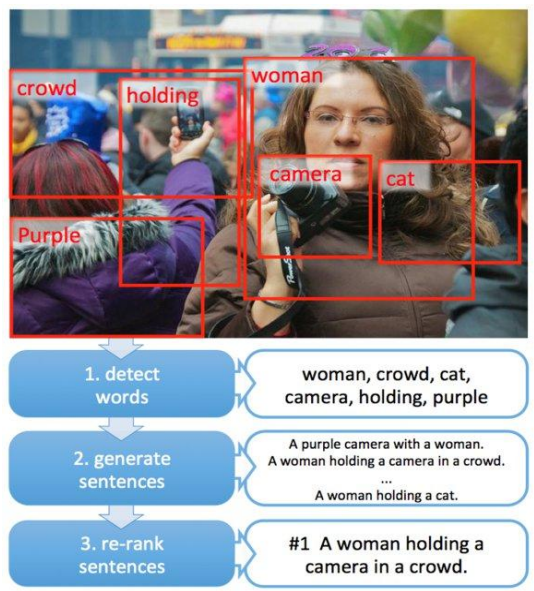
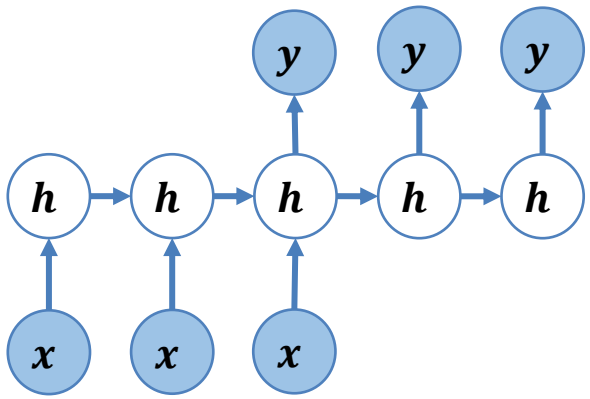


Image captioning
[https://www.pcworld.com]

MONASH University

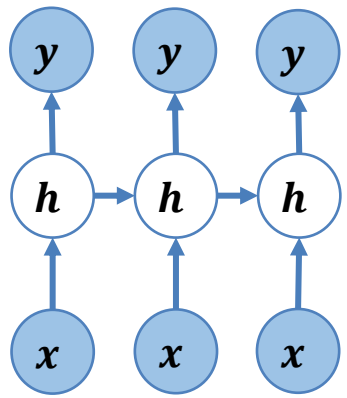
RNNs Architecture Zoo

- RNNs architecture is very flexible for many real-world tasks



many to many (1)

(machine translation)

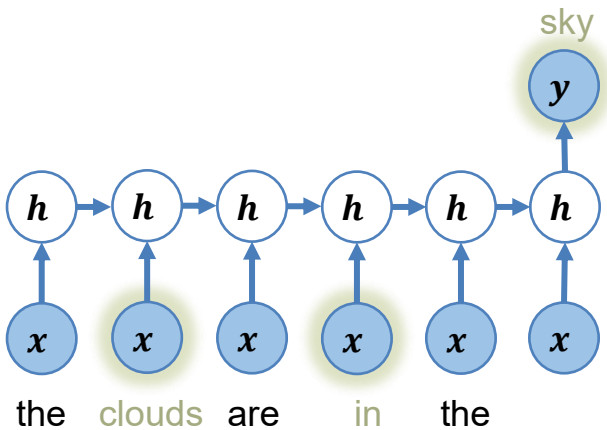


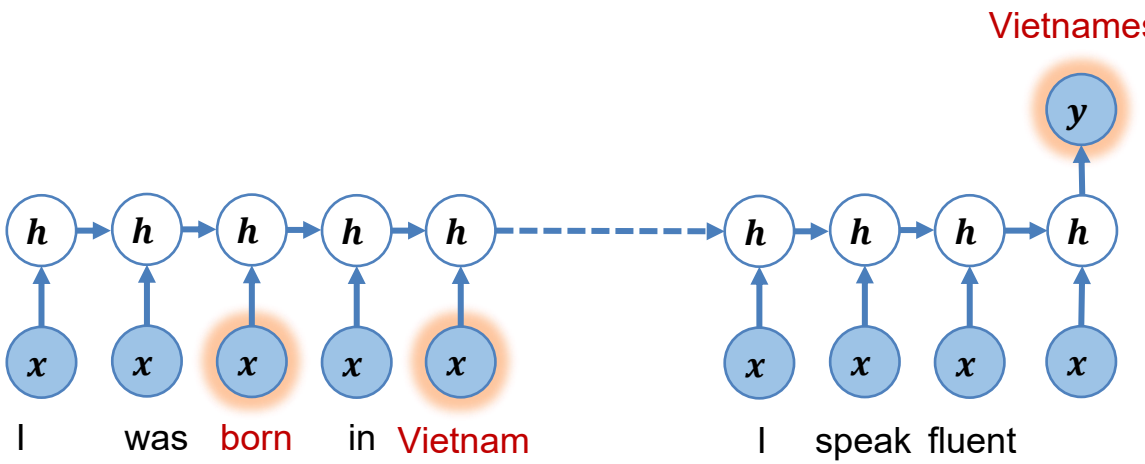
many to many (2)

(video classification)

... more to come in our next lectures

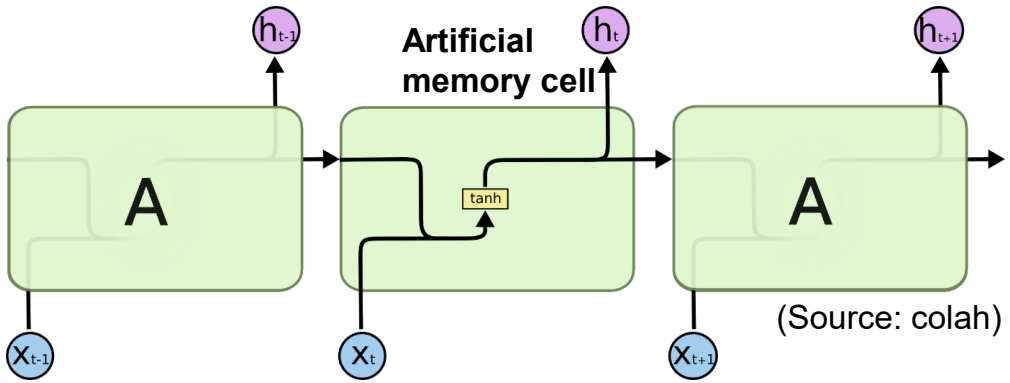
Problems of RNN

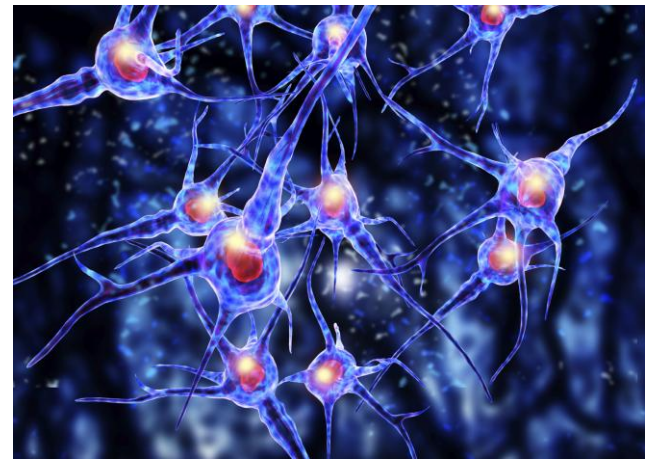




- RNNs **don't** capture **long-term dependency** adequately
  - A hidden state is computed based on only one previous state → can only capture **short-term dependency**
- Modelling drawbacks
  - Technical problem when training long sequences
    - vanishing gradient problem
  - Many layers of nonlinear transformation prevent the data signals and gradient from flowing easily through the network.
- How to address this?
  - Using gating mechanism: adding linear component from previous layer!
    - RNN → LSTM/GRU

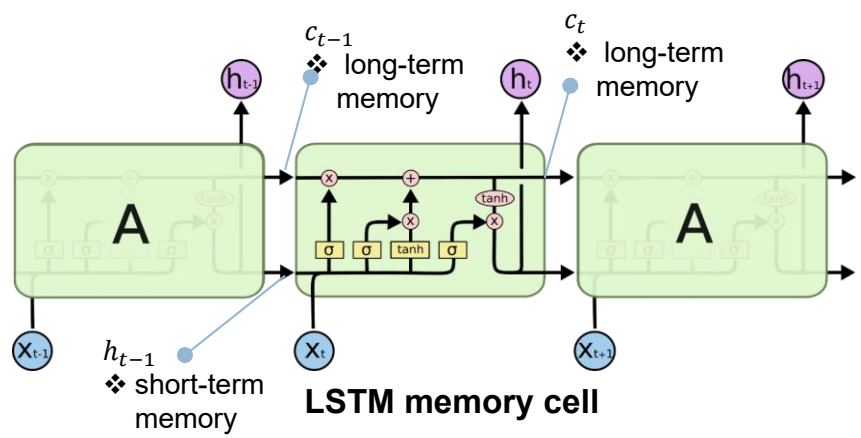
Memory cells



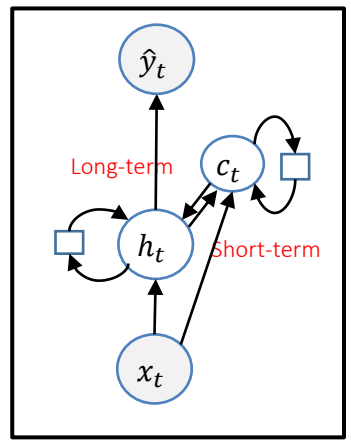


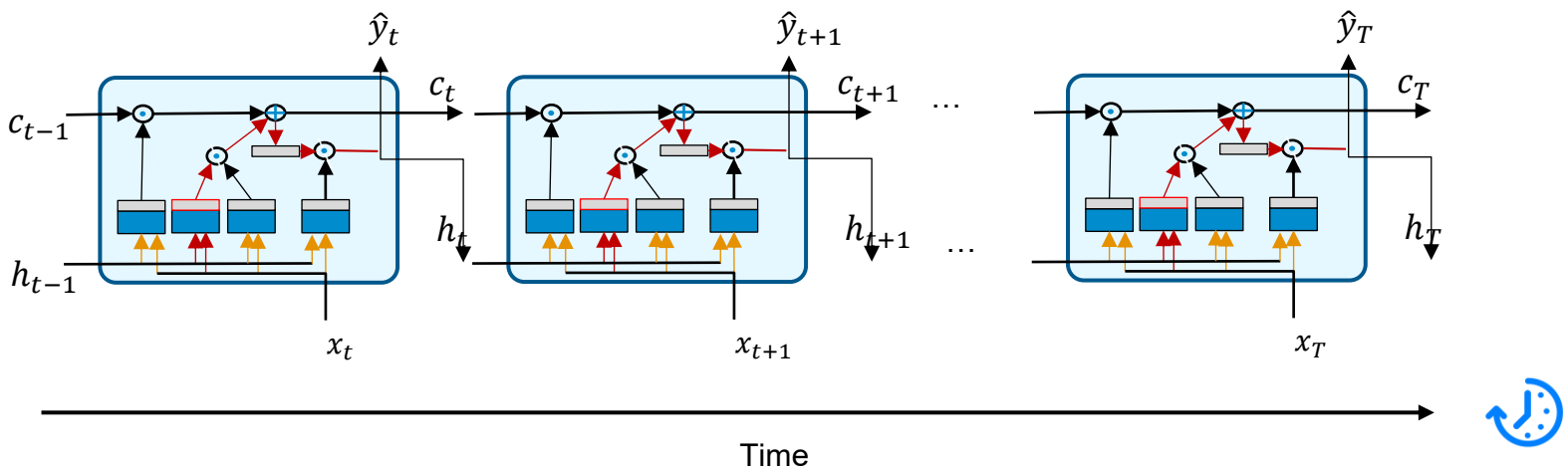
Biological **memory cells** in human brain
(Source: news.feinberg.northwestern.edu)

- Our RNN includes many **simple RNN cells**
  - **Input to a cell**: h_{t-1} (previous hidden state) and x_t (current input token)
  - **Output**: $h_t = \tanh(x_t U + h_{t-1} W + b)$
  - h_t can only capture **short-term dependency** → **short-term memory**
- How to capture **long-term memory** more efficiently?
  - **LSTM** cell and **GRU** cell

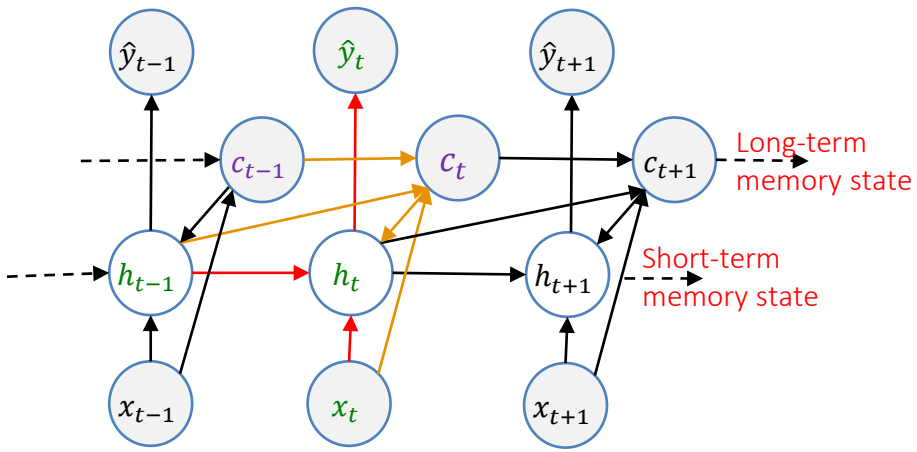


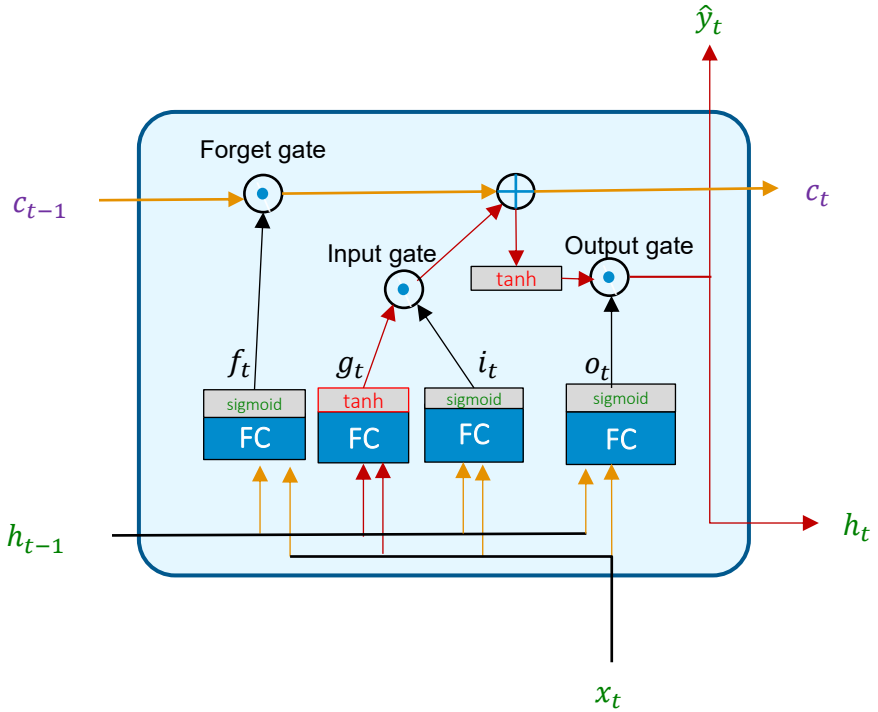
Long Short-Term Memory (LSTM)





Long Short-Term Memory (LSTM)





- Regression

$$\hat{y}_t = \boldsymbol{h_t V} + \boldsymbol{c}$$

- Classification

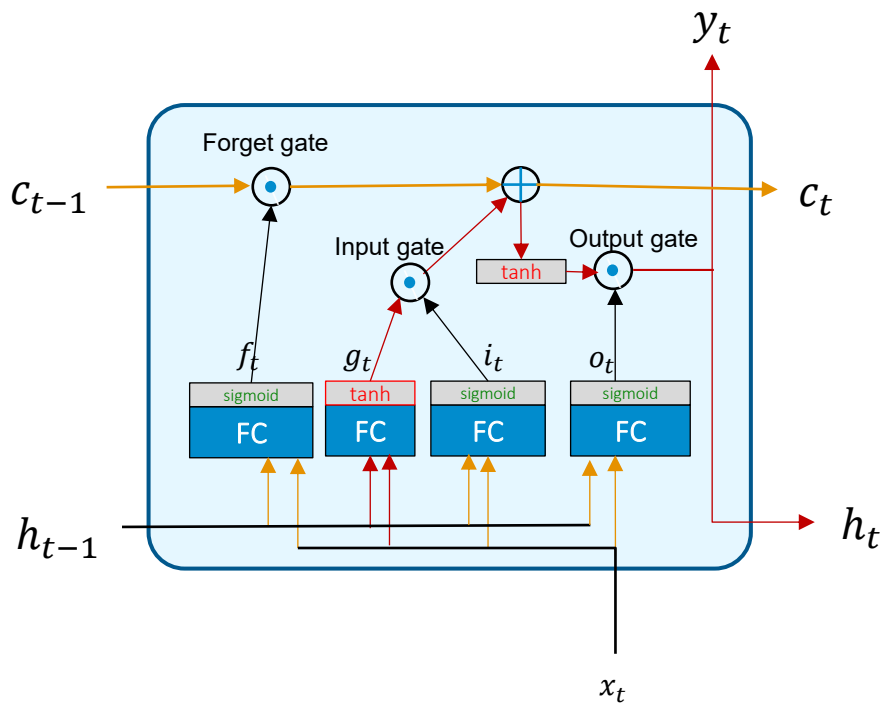
$$\hat{y}_t = \text{softmax}(\boldsymbol{h_t V} + \boldsymbol{c})$$

- $g_t = \tanh(h_{t-1}W + x_t U + b)$
- Forget gate: $f_t = \sigma\left(x_t U^f + h_{t-1} W^f + b^f\right)$
- Input gate: $i_t = \sigma(x_t U^i + h_{t-1} W^i + b^i)$
- LSTM long-term state: $c_t = f_t \odot c_{t-1} + g_t \odot i_t$
- Output gate: $o_t = \sigma(x_t U^o + h_{t-1} W^o + b^o)$
- LSTM short-term state: $h_t = o_t \odot \tanh(c_t)$
- LSTM output: $\hat{\boldsymbol{y}}_t = \boldsymbol{h_t V} + \boldsymbol{c}$ or $\hat{\boldsymbol{y}}_t = \textbf{softmax}(\boldsymbol{h_t V} + \boldsymbol{c})$

Long Short-Term Memory (LSTM)

Summary

- LSTM belongs to a class of gated RNN models
- LSTM introduces self-loops to create paths where the gradient can flow for long durations
- Improve over basic RNN cell
  - Can capture long-term dependency
  - Faster and more robust to train, often with quicker convergence
- LSTM cells manage two state vectors, and for performance reasons they are kept separate by default
  - h_t as the short-term state
  - c_t as the long-term state
- Gates can remove or add information to the cell state: forget, input, output



Gated Recurrent Unit

[Cho et. al, 2014]

- Proposed by Cho et al. 2014
  - Was introduced for Encoder-Decoder network
- Can be viewed as a **simplified version** of the LSTM:
  - Both long-term c_t and short-term h_t are **merged** to a single state h_t
  - Single **update gate** controller z_t is used to control both forget and input gates
    - = 1: open input gate, close forget gate
    - = 0: close input gate, open forget gate
    - i.e., whenever a memory must be stored, the location to be stored will be erased first!
  - Output gate is removed, but an additional reset gate r_t controls how much previous state should be carried forward

Cho et al., Learning Phrase Representation using RNN Encoder-Decoder for Statistical Machine Translation, *Proceedings of EMNLP*, 2014.

Gated Recurrent Unit

[Cho et. al, 2014]

- Update gate z_t decides how much the unit updates its state:

$$z_t = \sigma(x_t U^z + h_{t-1} W^z)$$

- Reset gate controls which parts of the state get used to compute the next target state

$$r_t = \sigma(x_t U^r + h_{t-1} W^r)$$

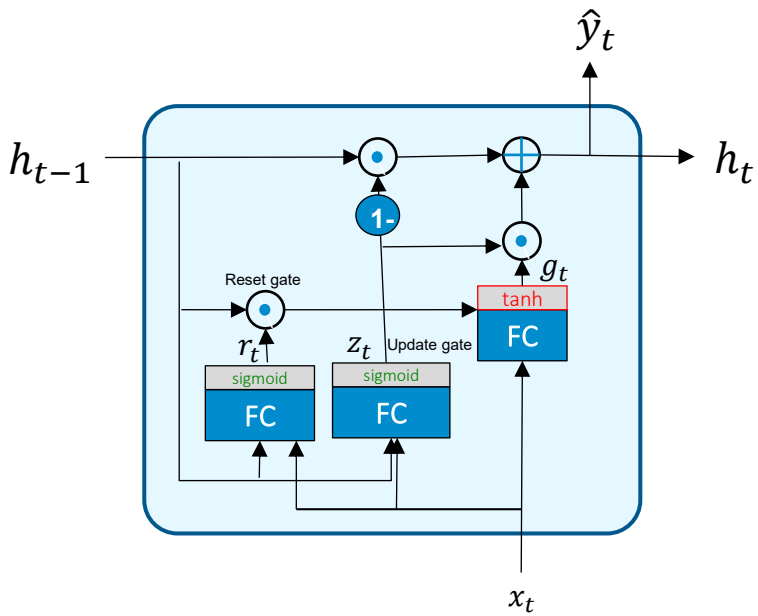
- The memory state h_t is a linear interpolation between h_{t-1} and g_t

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot g_t$$

where the candidate g_t is pre-computed

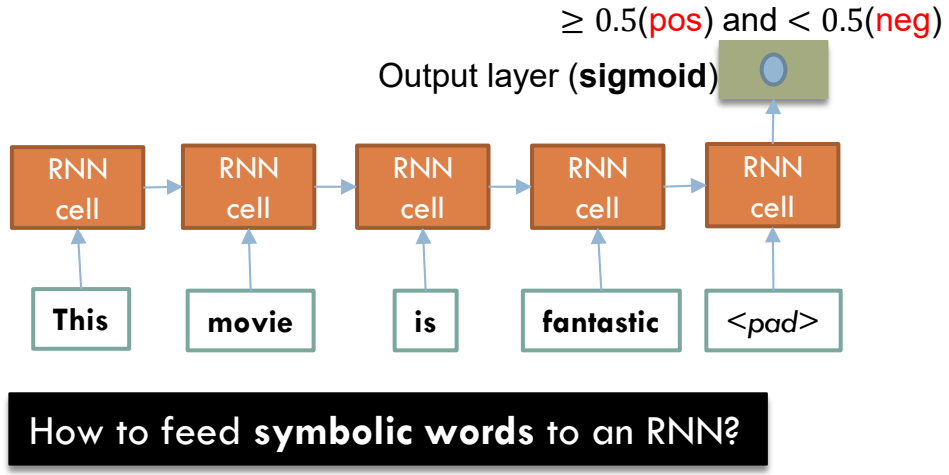
$$g_t = \tanh(x_t U^g + (r_t \odot h_{t-1}) W^g)$$

- When z_t and r_t are close to 1, GRU will be reduced to Basic RNN



Sentiment analysis

- **Movie review dataset**
  1. I like that movie (**pos**:1).
  2. This is a bad movie to watch (**neg**:0)
  3. I love the movie (**pos**: 1)
  4. I do not recommend you to watch this movie (**neg**:0)
  5. This movie is fantastic (**pos**:1)

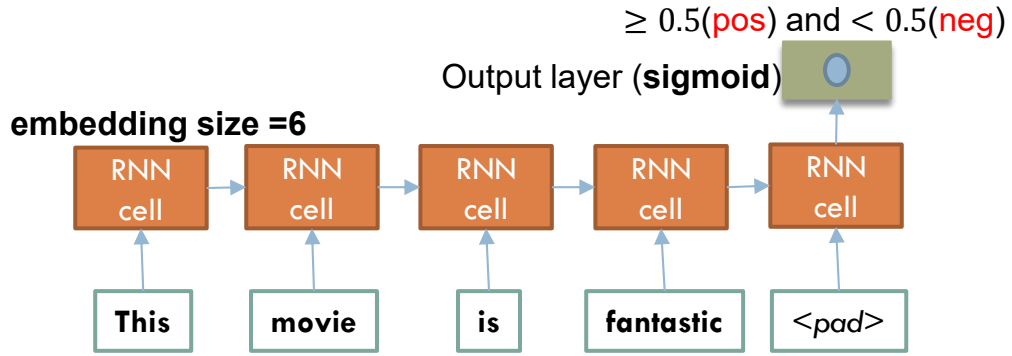


Sentiment analysis

- **Movie review dataset**
  1. I like this movie (**pos**:1).
  2. This is a bad movie to watch (**neg**:0)
  3. I love this movie (**pos**: 1)
  4. I do not recommend you to watch this movie (**neg**:0)
  5. This movie is fantastic (**pos**:1)

- **Build up vocabulary**
  1. Like (index: 1)
  2. Love (index: 2)
  3. Bad (index: 3)
  4. Fantastic (index: 4)
  5. Not (index: 5)
  6. Recommend (index: 6)

- **Not in vocabulary** (out of vocabulary bucket: 2)
  1. I, movie, to, pad (index: 7)
  2. This, is, watch (index: 8)



Embedding matrix (E [8×6])

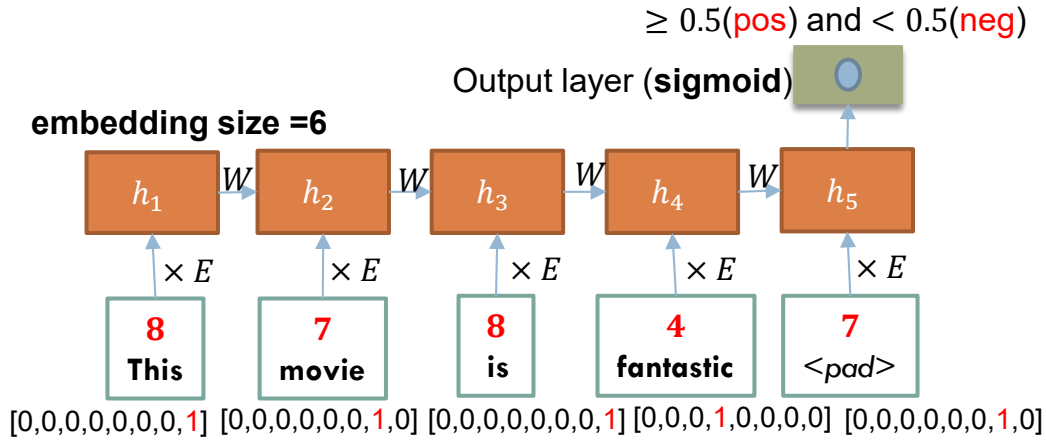
E_1	1	2	1.5	-1.2	1.3	1
E_2	-1	1.3	-2.5	-1.2	1.6	-1
E_3	1	3.3	-3.5	-1.0	2.6	1
E_4	-1	1.3	-2.5	-1.2	1.8	-1
E_5	-1.2	2.3	-2.5	-1.2	-1.6	1
E_6	-1.7	-1.3	-2.5	-1.2	3.6	1.2
E_7	-4.2	2.3	-3.5	4.3	1.8	-2
E_8	-1.7	-1.3	-4.5	-2.2	-3.6	1

Sentiment analysis

- **Movie review dataset**
  1. I like this movie (**pos**:1).
  2. This is a bad movie to watch (**neg**:0)
  3. I love this movie (**pos**: 1)
  4. I do not recommend you to watch this movie (**neg**:0)
  5. This movie is fantastic (**pos**:1)

- **Build up vocabulary**
  1. Like (index: 1)
  2. Love (index: 2)
  3. Bad (index: 3)
  4. Fantastic (index: 4)
  5. Not (index: 5)
  6. Recommend (index: 6)

- **Not in vocabulary** (out of vocabulary bucket: 2)
  1. I, movie, to, pad (index: 7)
  2. This, is, watch (index: 8)

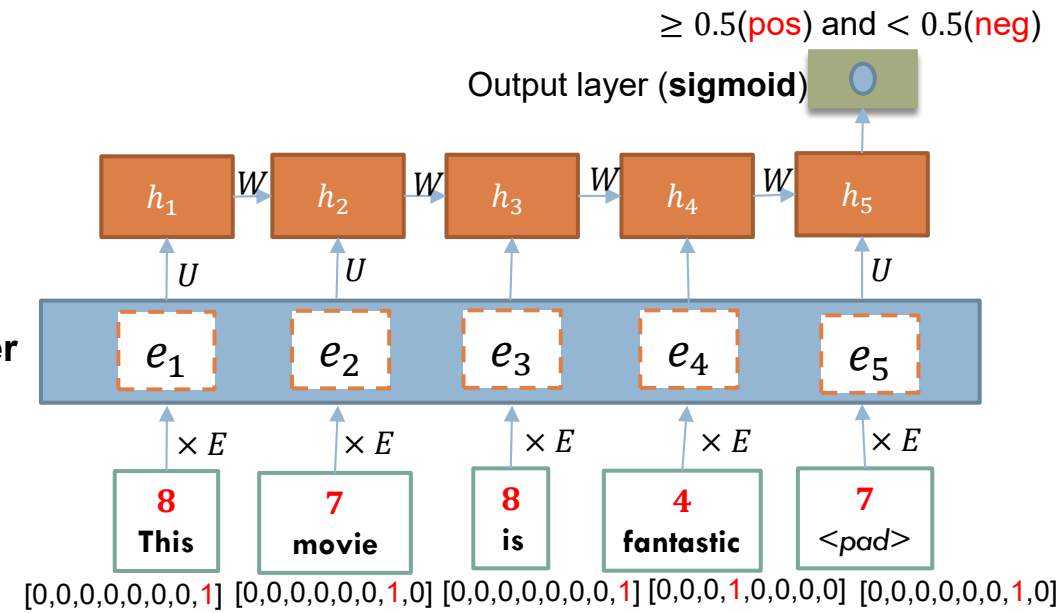


Embedding matrix (E [8 × 6])

E_1	1	2	1.5	-1.2	1.3	1
E_2	-1	1.3	-2.5	-1.2	1.6	-1
E_3	1	3.3	-3.5	-1.0	2.6	1
E_4	-1	1.3	-2.5	-1.2	1.8	-1
E_5	-1.2	2.3	-2.5	-1.2	-1.6	1
E_6	-1.7	-1.3	-2.5	-1.2	3.6	1.2
E_7	-4.2	2.3	-3.5	4.3	1.8	-2
E_8	-1.7	-1.3	-4.5	-2.2	-3.6	1

Sentiment analysis

- The word/item embedding
  - $e_1 = 1_8 E = E_8 \in \mathbb{R}^{1\times 6}$
    - $[1 \times 8] \times [8 \times 6] = [1 \times 6]$
    - 1_i is one-hot vector in which the $i-th$ element is 1 and the rest are 0.
  - $e_2 = 1_7 E = E_7 \in \mathbb{R}^{1\times 6}$
  - $e_3 = 1_8 E = E_8 \in \mathbb{R}^{1\times 6}$
  - $e_4 = 1_4 E = E_4 \in \mathbb{R}^{1\times 6}$
  - $e_5 = 1_7 E = E_7 \in \mathbb{R}^{1\times 6}$



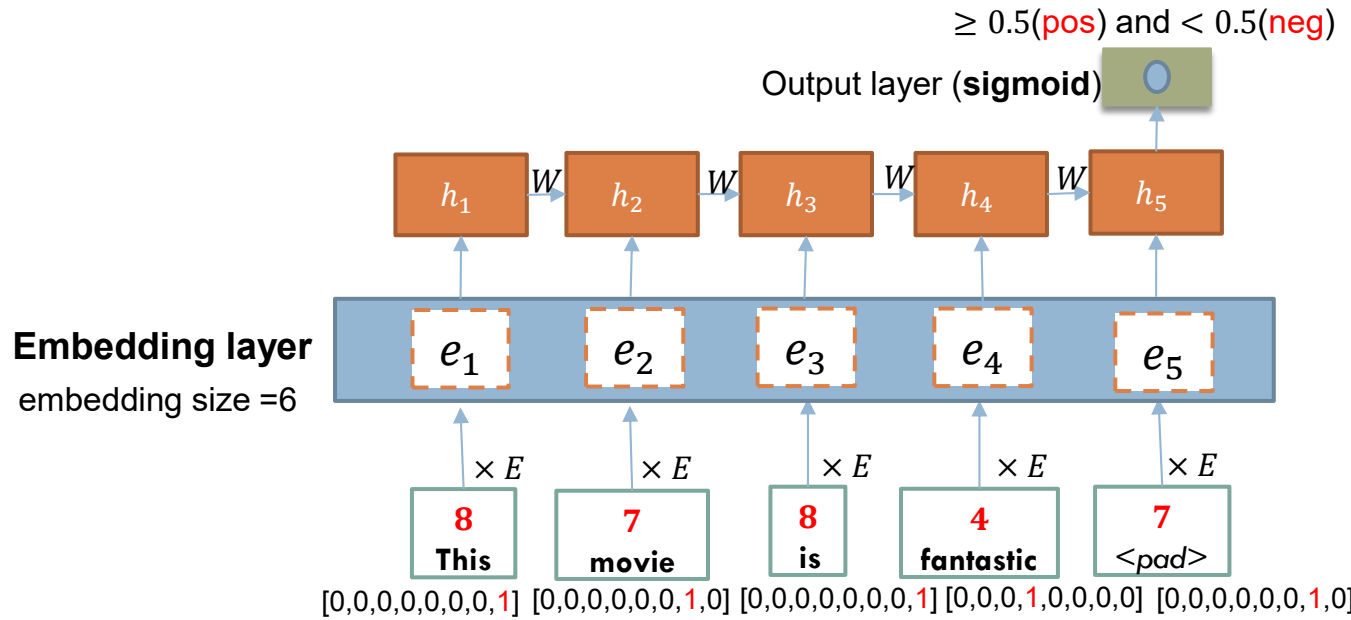
- The sequence embedding
  - $e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{bmatrix} \in \mathbb{R}^{5\times 6} = \mathbb{R}^{seq\_len \times embed\_size}$
  - **Embedding lookup** operation
    - Pick rows with indices **8, 7, 8, 4, 7**

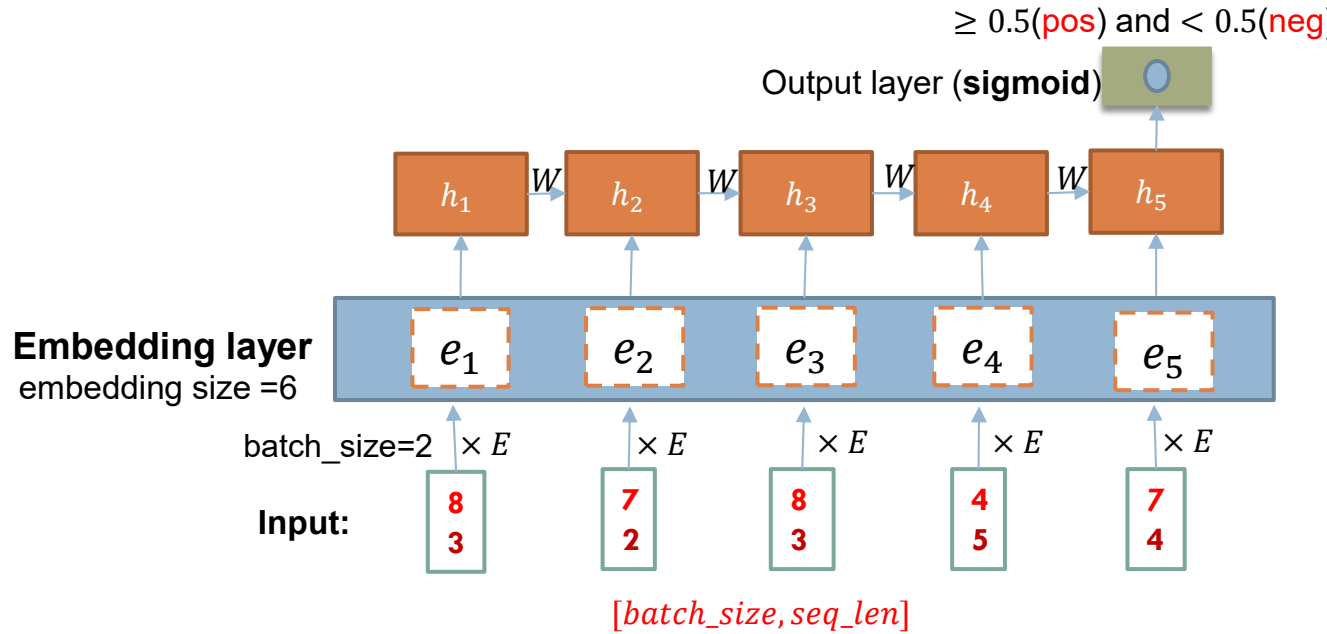
Embedding matrix (E [8×6])

E_1	1	2	1.5	-1.2	1.3	1
E_2	-1	1.3	-2.5	-1.2	1.6	-1
E_3	1	3.3	-3.5	-1.0	2.6	1
E_4	-1	1.3	-2.5	-1.2	1.8	-1
E_5	-1.2	2.3	-2.5	-1.2	-1.6	1
E_6	-1.7	-1.3	-2.5	-1.2	3.6	1.2
E_7	-4.2	2.3	-3.5	4.3	1.8	-2
E_8	-1.7	-1.3	-4.5	-2.2	-3.6	1

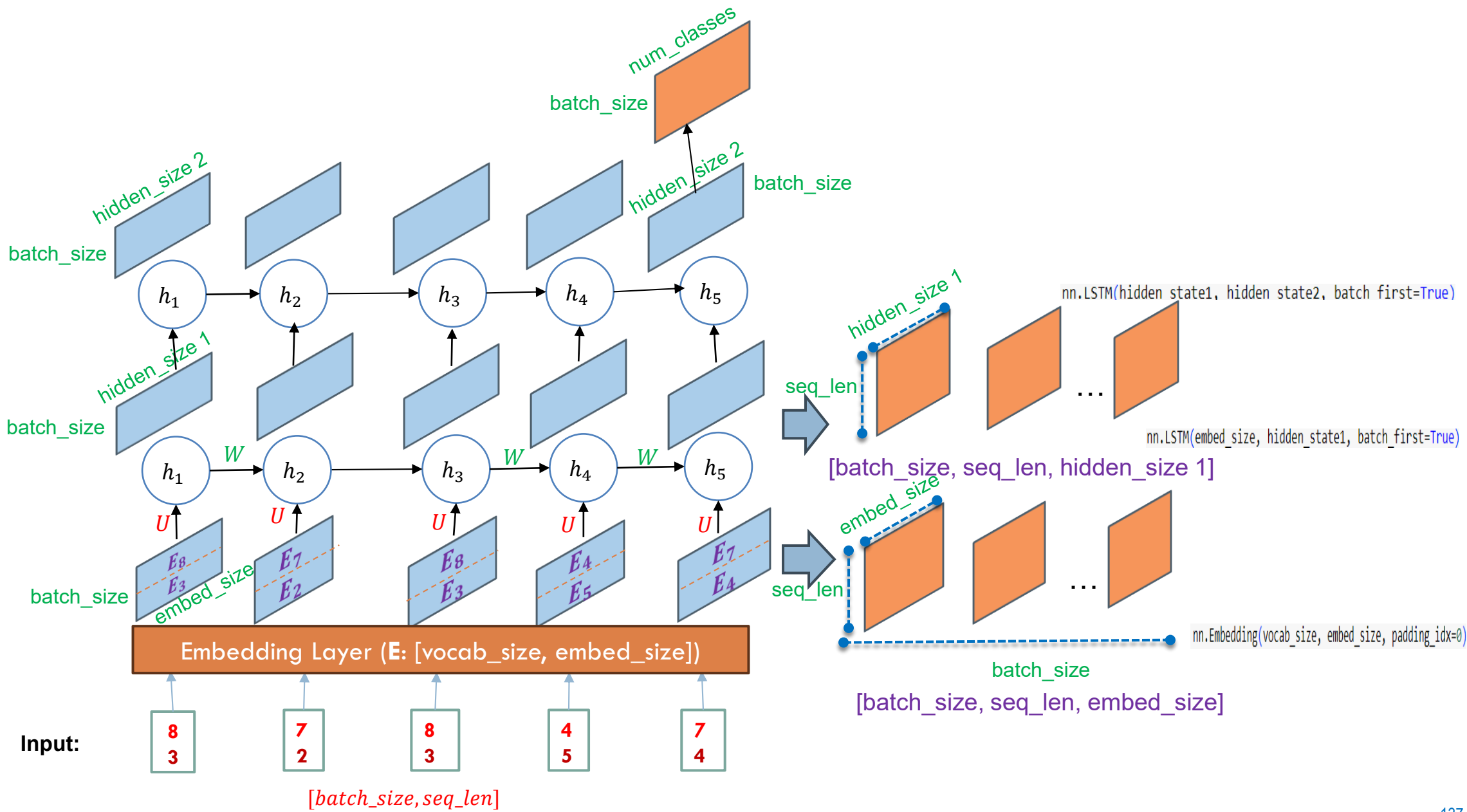
Sentiment analysis

- The word/item embedding
  - $e_1 = 1_8 E \in \mathbb{R}^{1\times 6}$
  - $e_2 = 1_7 E \in \mathbb{R}^{1\times 6}$
  - $e_3 = 1_8 E \in \mathbb{R}^{1\times 6}$
  - $e_4 = 1_4 E \in \mathbb{R}^{1\times 6}$
  - $e_5 = 1_7 E \in \mathbb{R}^{1\times 6}$
- The sequence embedding
  - $e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{bmatrix} \in \mathbb{R}^{5\times 6} = \mathbb{R}^{seq\_len \times embed\_size}$
- The sequence batch embedding
  - The embedding for one sequence: $\mathbb{R}^{seq\_len \times embed\_size}$.
  - The embedding for entire batch with $batch\_size$ sequences: $\mathbb{R}^{batch\_size \times seq\_len \times embed\_size}$





Shape Transformation in RNNs



Sentiment analysis

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init

embed_size = 128

class Model(nn.Module):
    def __init__(self, vocab_size):
        super(Model, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size, padding_idx=0)

        self.lstm1 = nn.LSTM(embed_size, 128, batch_first=True)
        self.lstm2 = nn.LSTM(128, 128, batch_first=True)

        self.fc = nn.Linear(128, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        x,(_,_) = self.lstm1(x)
        _,(x,_) = self.lstm2(x)
        x = self.fc(x.view(-1,128))
        return x.squeeze(1)
```

(batch_size, seq_len)

(batch_size, seq_len, embed_size)

(batch_size, seq_len, 128)

(batch_size, 128)

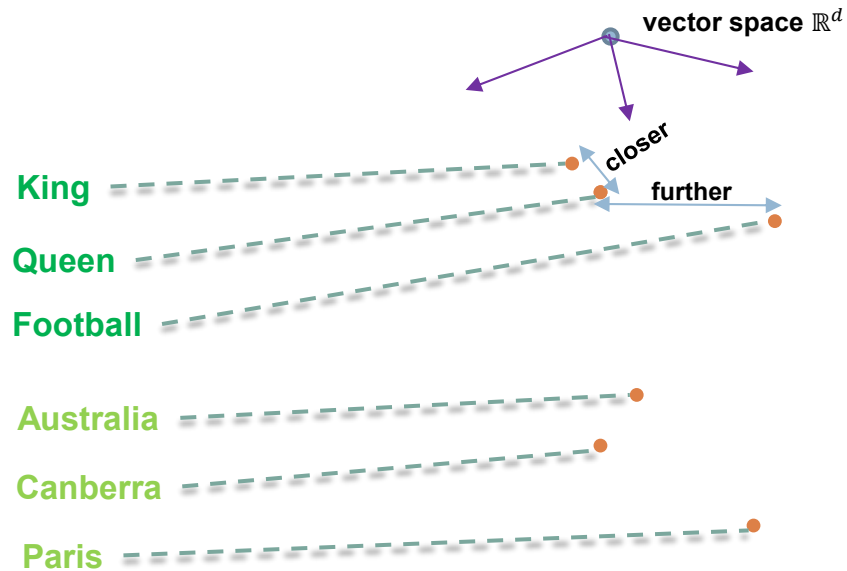
(batch_size, 1)

Word2Vec

Word embedding (Word2Vec)

Wikipedia text corpus

We desire…



Canberra : Australia = Paris : ???

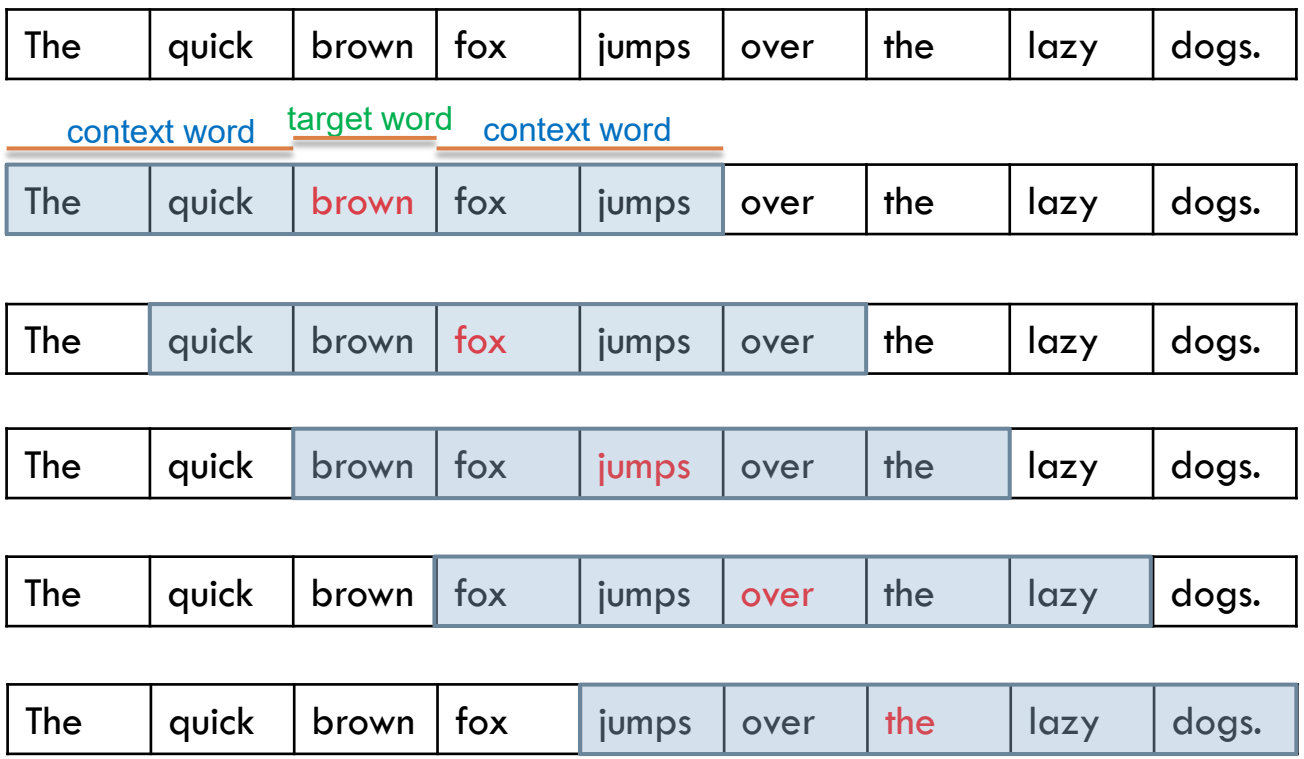
$$argmin_v \| v_{Canberra} - v_{Australia} + v_{Paris} - v \| = v_{France}$$

- **We have**: Many texts in Wikipedia
- **We want**: Learn vector representations for words that preserve semantic and linguistic relationship carried in the text corpus
- **We need**: Devise pretext task to cast the learning word representation to supervised learning.

Word2Vec: Pretext task

Pretext task

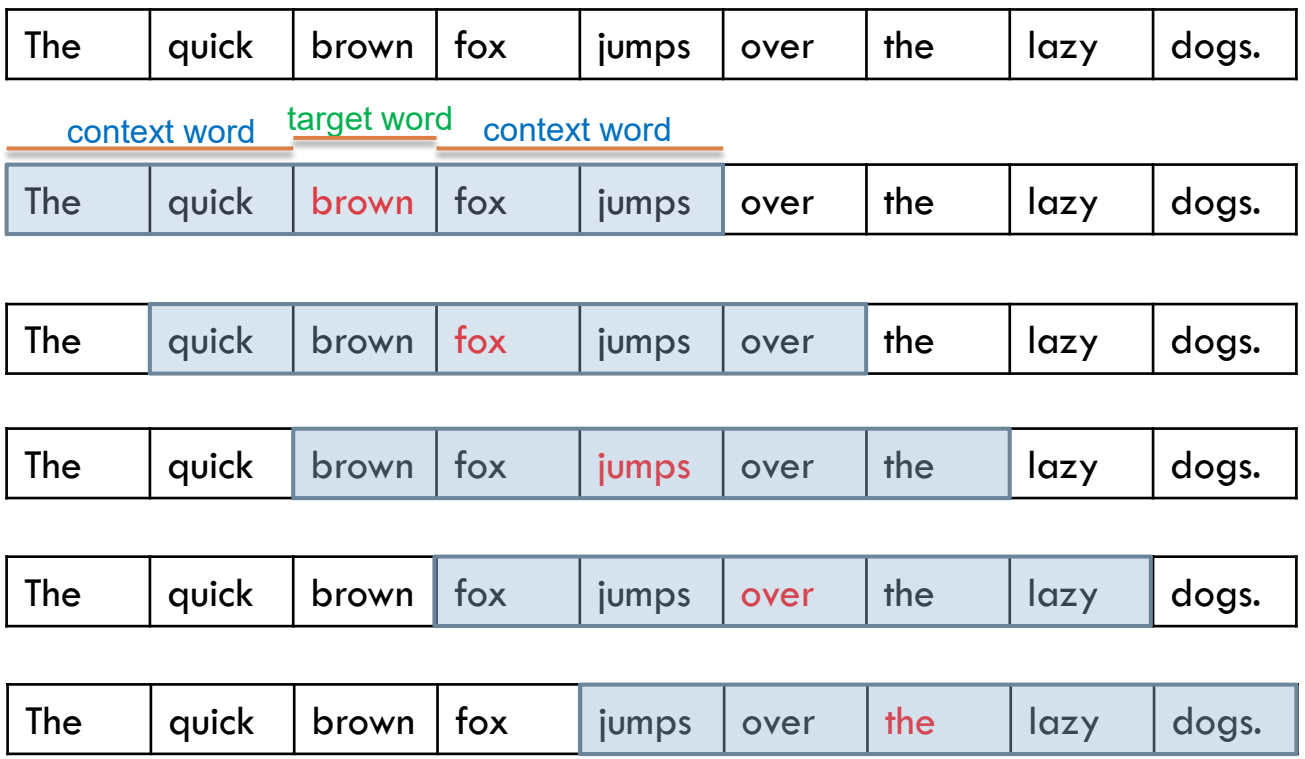
- What is the **pretext task** of Word2Vec?



Skip-gram

Pretext task

- What is the **pretext task** of Skip-gram?



Skip-gram

- **Target word** $\xrightarrow{predict}$ **context words**

(brown, the), (brown, quick), (brown, for), (brown, jumps)

(fox, quick), (fox, brown), (fox, jumps), (fox, overs)

(jumps, brown), (jumps, fox), (jumps, over), (jumps, the)

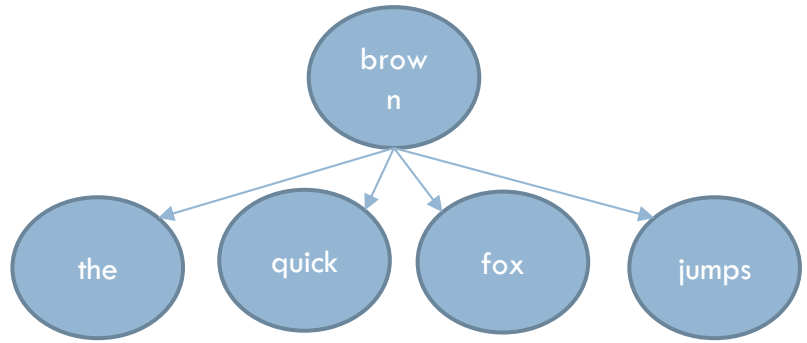
(over, fox), (over, jumps), (over, the), (over, lazy)

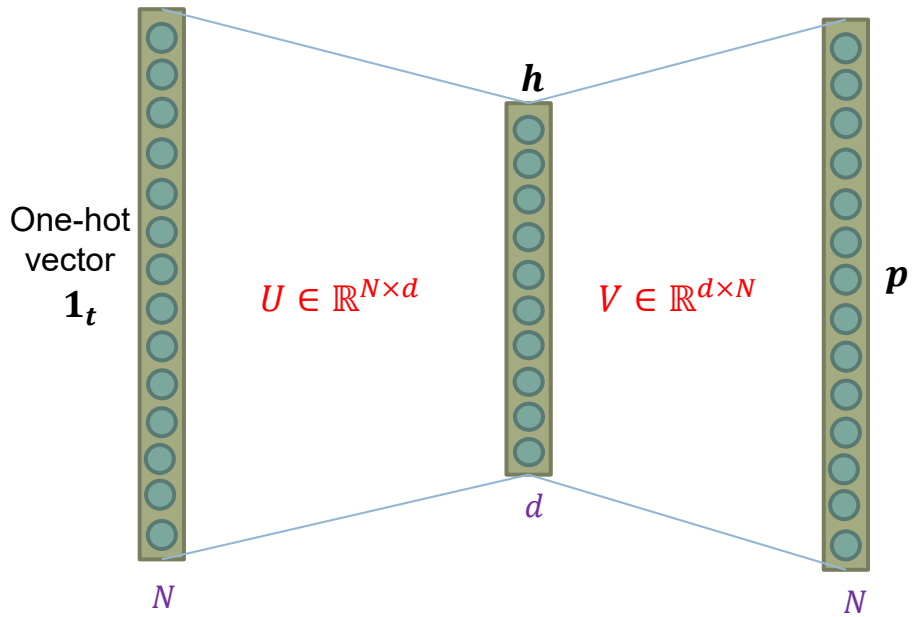
(the, jumps), (the, over), (the, lazy), (the, dogs)

Skip-gram

Modelling

- **Current window**
  - The quick brown fox jumps
- $P(the, quick, fox, jumps \mid brown) = P(the \mid brown) \times P(quick \mid brown) \times P(fox \mid brown) \times P(jumps \mid brown)$
- $\log P(the, quick, fox\ jumps \mid brown) = \log P(the \mid brown) + \cdots + \log P(jumps \mid brown)$
- (brown, the), (brown, quick), (brown, for), (brown, jumps). Let consider $tw = brown$ and $cw = the$.
- **Two matrices**
  - $\boldsymbol{U} \in \mathbb{R}^{\boldsymbol{N}\times\boldsymbol{d}}$ (N is vocabulary size, d is embedding size)
  - $\boldsymbol{V} \in \mathbb{R}^{\boldsymbol{d}\times\boldsymbol{N}}$
- Assume that indices of tw and cw are $1 \leq t, c \leq N$ respectively. The forward propagation is as follows:
  - $h = 1_t U = U_t^r \in \mathbb{R}^{1\times d}, o = hV \in \mathbb{R}^{1\times N}, p = softmax(o) \in \mathbb{R}^{1\times N}$
  - $P(cw = the \mid tw = brown) = p_c$
  - $\log P(cw = the \mid tw = brown) = \log p_c = U_t^r V_c^c - \log(\sum_{k=1}^{N} \exp(U_t^r V_k^c))$
- Train the model by **maximizing log likelihood**.





Toy example

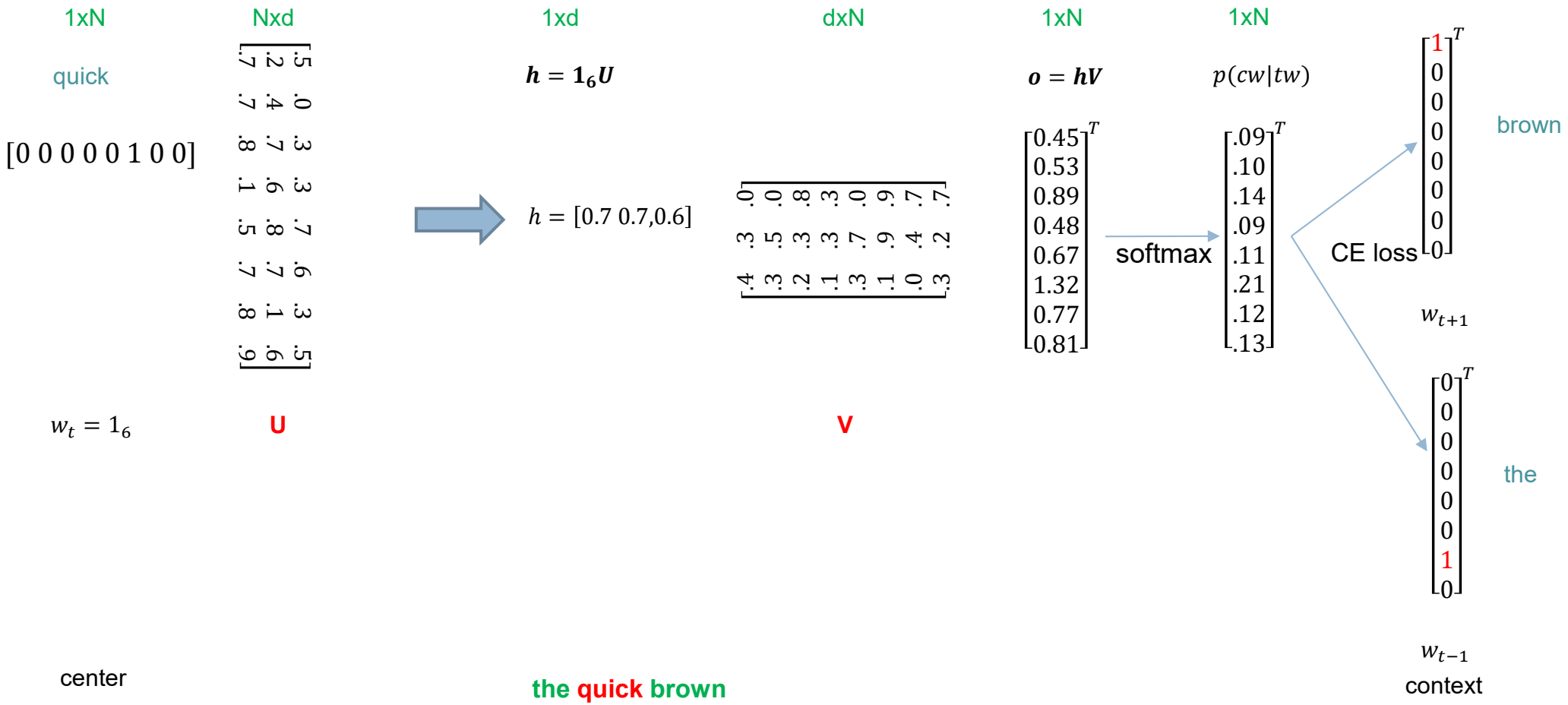
- Corpus: **the quick brown fox jumps over the lazy dog**
  - **Tokens**: {'brown': 1, 'lazy': 2, 'over': 3, 'fox': 4, 'dog': 5, 'quick': 6, 'the': 7, 'jumps': 8}
  - **Number of tokens** N = 8
  - **Context (window) size C** = 3
  - **Size of embedded vectors** d = 3
  - **U&V**: collections of input & output vectors

quick

[0 0 0 0 0 1 0 0]

one-hot encoding

Skip-gram, forward propagation

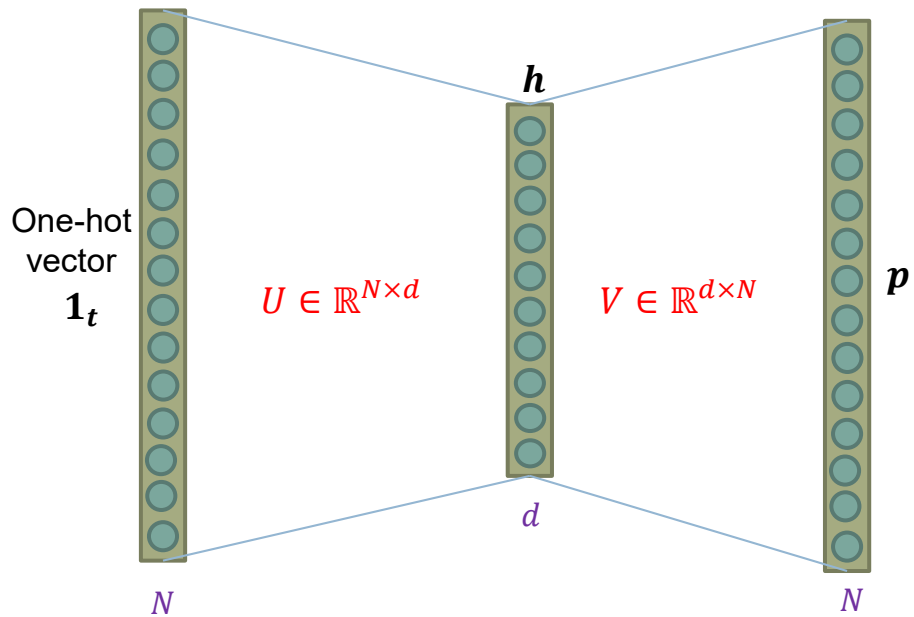


{'**brown**': 1, 'lazy': 2, 'over': 3, 'fox': 4, 'dog': 5, 'quick': 6, '**the**': 7, 'jumps': 8}

Skip-gram

Drawback

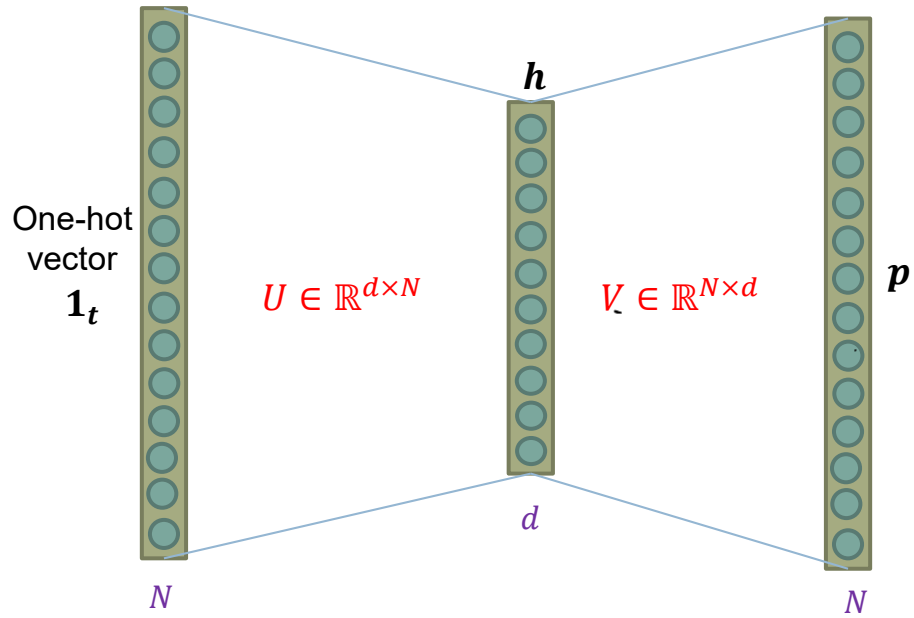
- **Two matrices**
  - $\boldsymbol{U} \in \mathbb{R}^{\boldsymbol{N}\times\boldsymbol{d}}$ (N is vocabulary size, d is embedding size)
  - $\boldsymbol{V} \in \mathbb{R}^{\boldsymbol{d}\times\boldsymbol{N}}$
- Assume that indices of tw and cw are $1 \leq t, c \leq N$ respectively. The forward propagation is as follows:
  - $h = 1_t U = U_t^r \in \mathbb{R}^{1\times d}, o = hV \in \mathbb{R}^{1\times N}, p = softmax(o) \in \mathbb{R}^{1\times N}$
  - $P(cw = the \mid tw = brown) = p_c$
  - $\log P(cw = the \mid tw = brown) = \log p_c = U_t^r V_c^c - \log(\sum_{k=1}^{N} \exp(U_t^r V_k^c))$
- **Some drawbacks**
  - $p = softmax(o)$ is **computationally expensive**
  - $p \in \mathbb{R}^{1\times N}$ is a distribution over the **vocabulary with size N** (usually very big) → the values of p_i are **very tiny → hard to train**
    - Hierarchical SoftMax
    - Negative sampling (more popular and efficient)



Skip-gram

Negative sampling

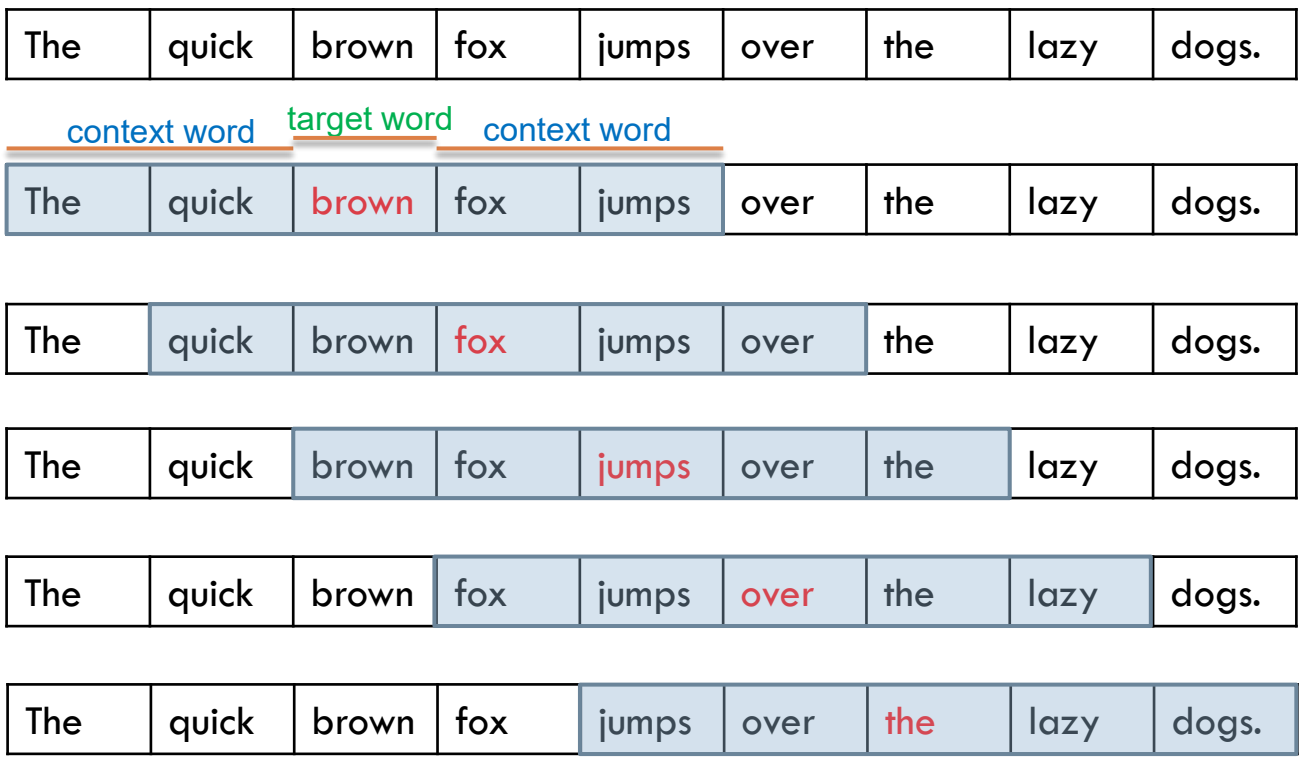
- Transform ***N*-class prediction to binary prediction** with **negative examples**
- Consider a **positive (true) pair** (tw=brown, cw = the)
  - [(brown, the), 1]
  - Sample randomly some (two) words
    - [(brown, ng_1 = hello), 0] and [(brown, ng_2 = awsome), 0]
    - Let denote the indices of ng_1 and ng_2 by $1 \le n_1, n_2 \le N$.
- The **forward propagation**
  - $h = 1_t U = U_t^r \in \mathbb{R}^{1\times d}$, $o = hV \in \mathbb{R}^{1\times N}$, $p = sigmoid(o) \in \mathbb{R}^{1\times N}$
  - $P(y = 1 \mid$ tw=brown, cw =$the) = p_c$
  - $P(y = 1 \mid$ tw=brown, ng_1=$hello) = p_{n_1}$
  - $P(y = 1 \mid$ tw=brown, ng_2=$awsome) = p_{n_2}$
- **Optimization problem**
  - $\mathbf{max}[\log p_c - \alpha \log p_{n_1} - \alpha \log p_{n_2}]$ where α>0 is a trade-off parameter.



Continuous Bag of Words (CBOW)

Pretext task

- What is the **pretext task** of CBOW?



Continuous Bag of Words (CBOW)

- **Context words** $\xrightarrow{predict}$ **target word**

(the | quick | for | jumps, brown)

(quick | brown | jumps | over, fox)

(brown | fox | over | the, jumps)

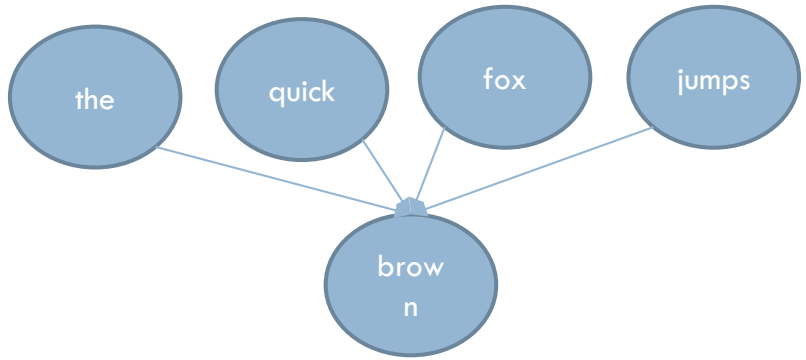
(for | jumps | the | lazy, over)

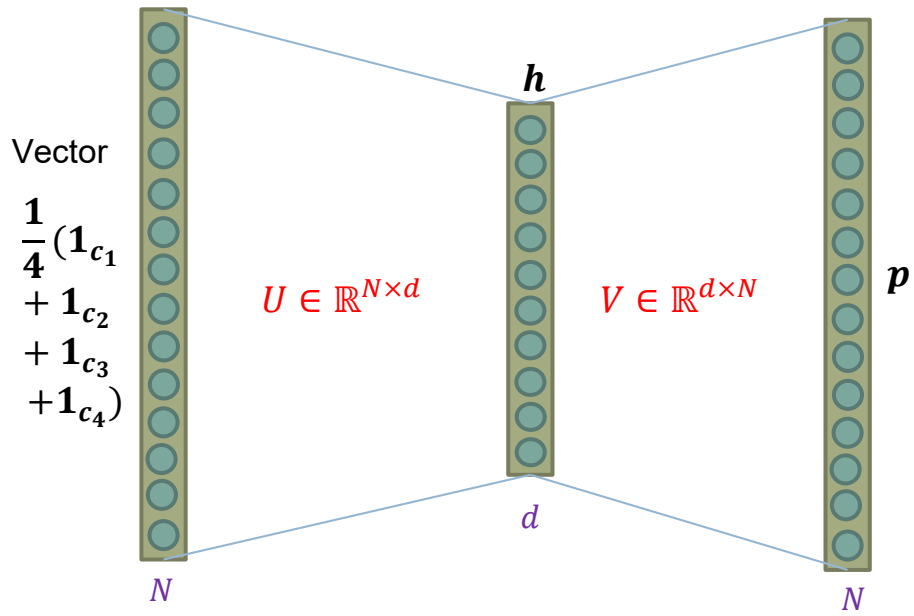
(jumps | over | lazy| dog, the)

Continuous Bag of Words (CBOW)

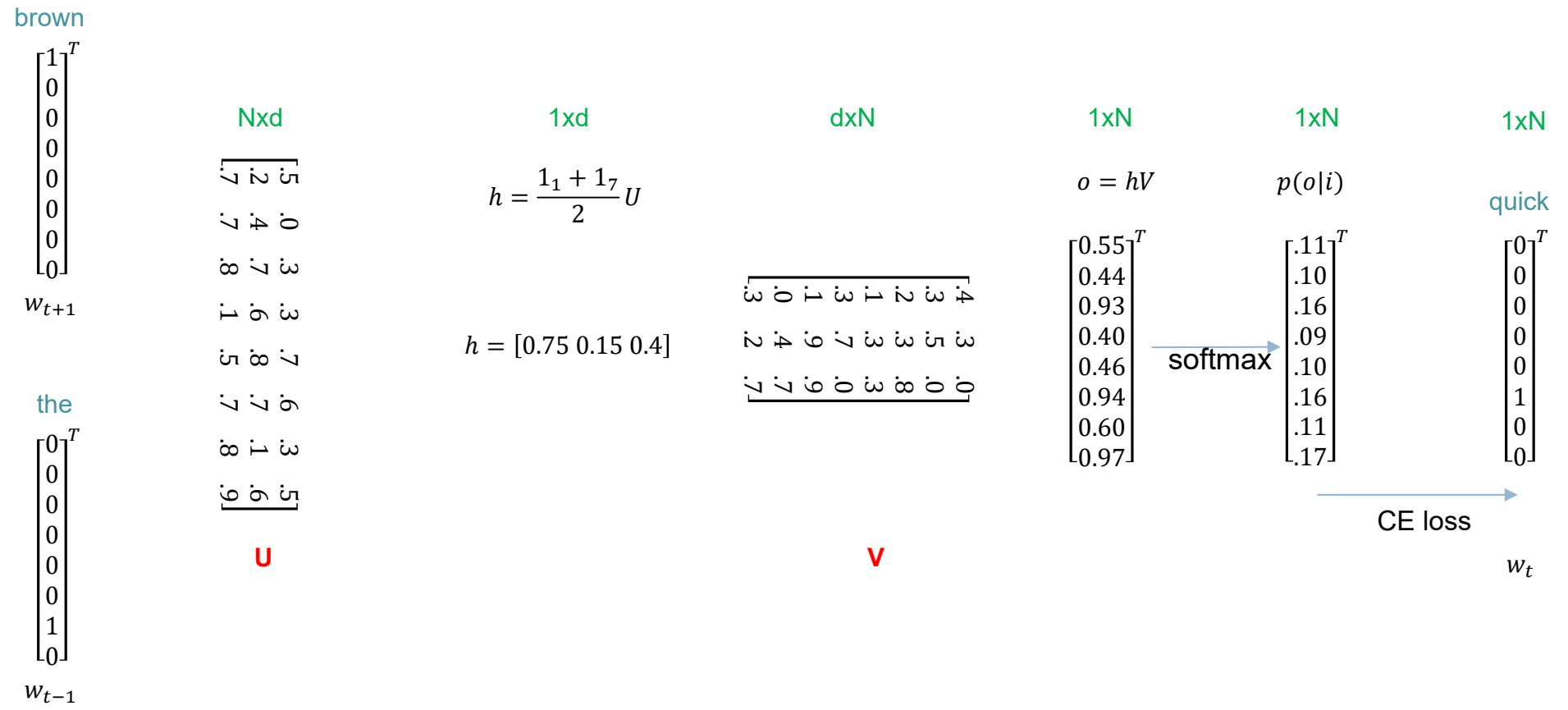
Modelling

- **Current window**
  - The quick brown fox jumps
- Need to formulate: $P(\ brown \mid the, quick, fox, jumps\ )$
- (the | quick | fox | jumps, brown). Let consider $tw = brown$ and $cw_1 = the, cw_2 = quick, cw_3 = for, cw_4 = jumps.$
- **Two matrices**
  - $\boldsymbol{U} \in \mathbb{R}^{\boldsymbol{N}\times\boldsymbol{d}}$ (N is vocabulary size, d is embedding size)
  - $\boldsymbol{V} \in \mathbb{R}^{\boldsymbol{d}\times\boldsymbol{N}}$
- Assume that indices of tw is $1 \le t \le N$ and $cw_{1:4}$ are $1 \le c_{1:4} \le N$ respectively. The forward propagation is as follows:
  - $h = \frac{1_{c_1}+\cdots+1_{c_4}}{4} U = \frac{1}{4}\left(U^r_{c_1} + \cdots + U^r_{c_4}\right) = \overline{U^r} \in \mathbb{R}^{1\times d}, o = hV \in \mathbb{R}^{1\times N}, p = softmax(o) \in \mathbb{R}^{1\times N}$
  - $P(\ brown \mid the, quick, fox\ jumps\ ) = p_t$
  - $\log P(\ brown \mid the, quick, fox\ jumps\ ) = \log p_t = \overline{U^r} V^c_t - \log(\sum_{k=1}^{N} \exp(\overline{U^r} V^c_k))$
- Train the model by **maximizing log likelihood**.





CBOW: forward propagation



Advanced Sequential Models

Encoder-Decoder Models

Context or **encode** c

Source structured data

Target structured data

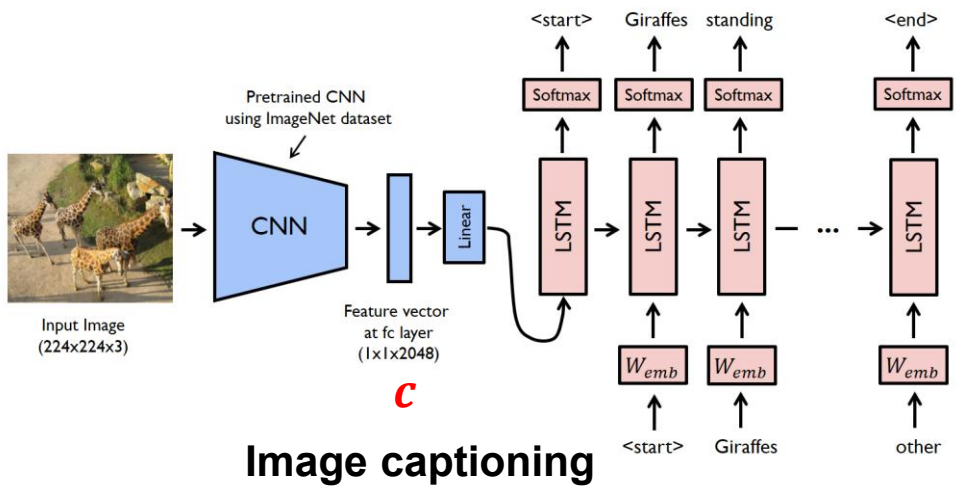
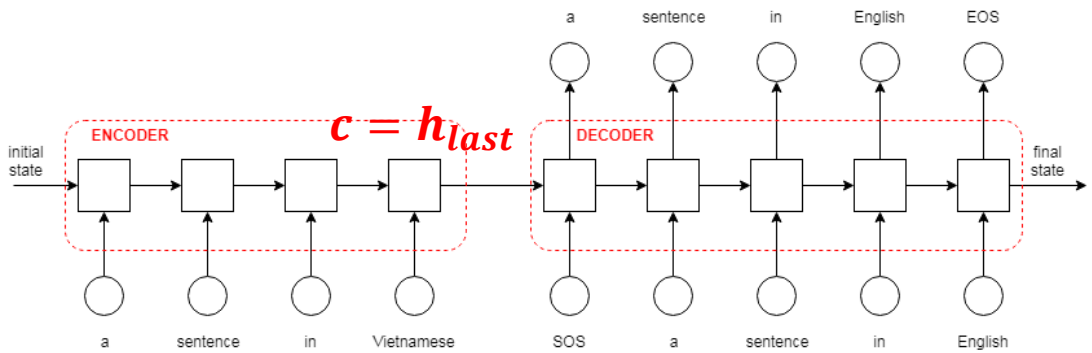


Image captioning



Machine translation

Many more applications

- C++ programs to Java programs, texts to images, question answering (questions to answers), and etc.

Introduction

- Problem statement of seq2seq
  - **Source sequence**: $x = (x_1, x_2, \dots, x_{T_x})$ where $x_i \epsilon V_x$
  - **Target sequence**: $y = (y_1, y_2, \dots, y_{T_y})$ where $y_i \epsilon V_y$
  - Training set:
    - $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})\} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$
    - S_x: set of **possible source sequences** ($\{x^{(i)}\} \subseteq S_x$)
    - S_y: set of **possible target sequences** ($\{y^{(i)}\} \subseteq S_y$)
  - **Task**
    - Learn a function $f: S_x \rightarrow S_y$
- Applications
  - Machine Translation
  - Image Captioning

Machine translation

Sequence to Sequence Learning with Neural Networks

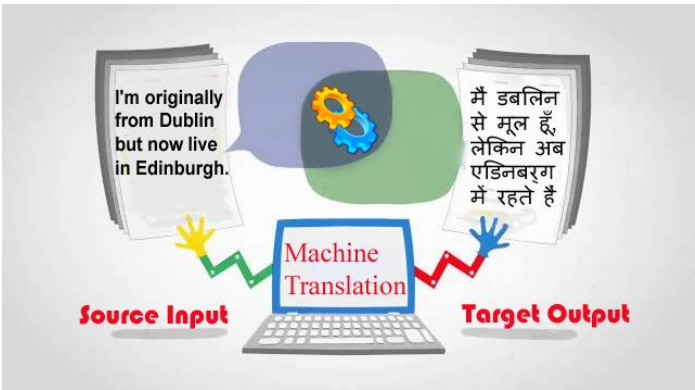
Ilya Sutskever
Google
ilyasu@google.com

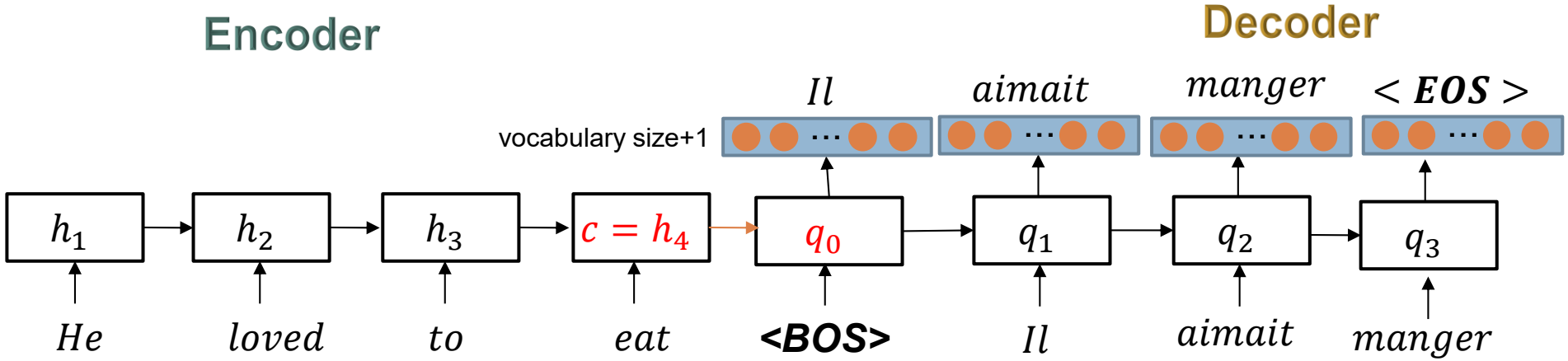
Oriol Vinyals
Google
vinyals@google.com

Quoc V. Le
Google
qvl@google.com

Paper: paper link

- Translate a sentence in a language to another language
  - **Input**: He loved to eat. (English)
  - **Output**: Il aimait manger. (French)

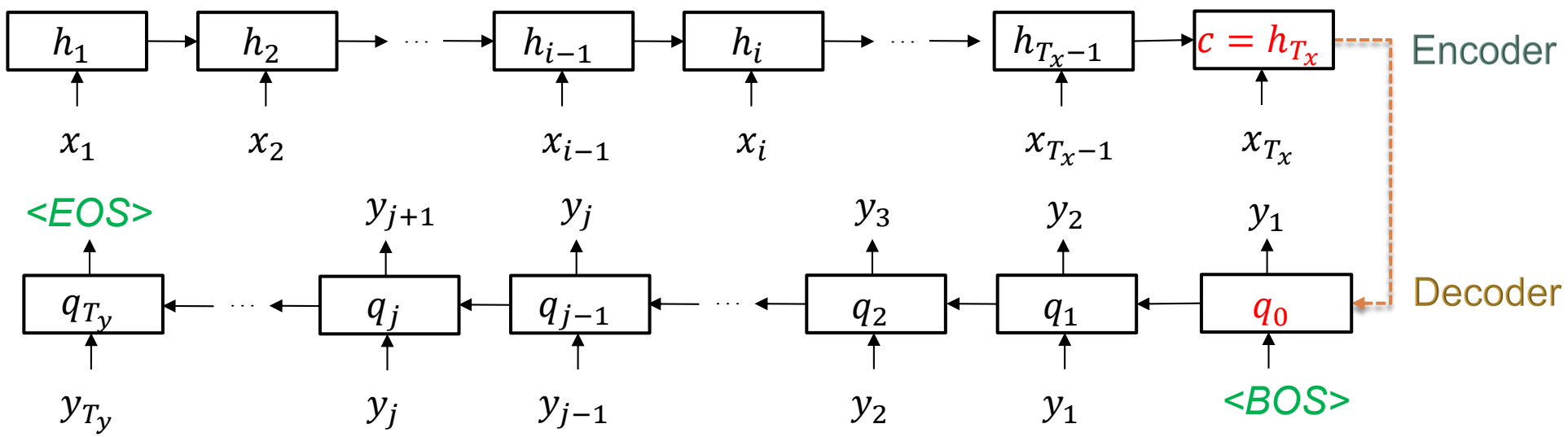




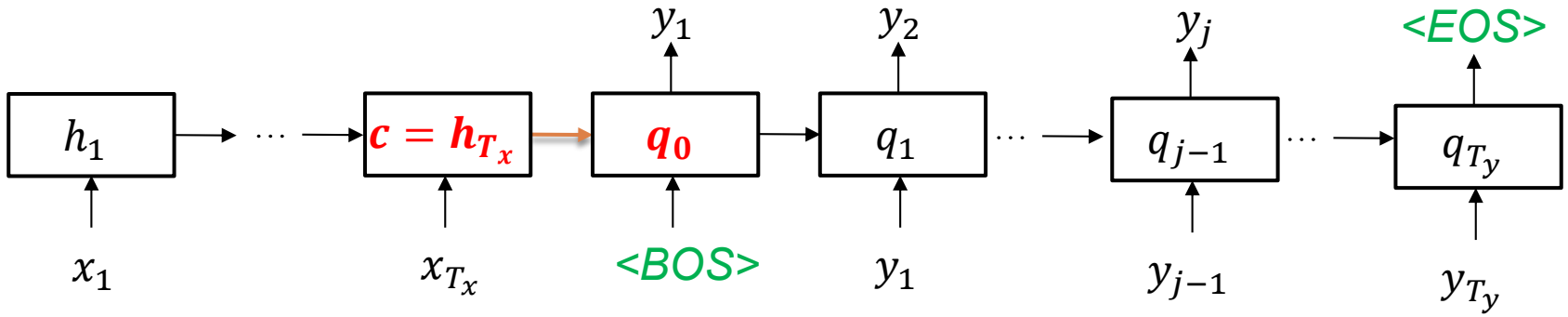
Encoder-decoder model for seq2seq

Fixed context vector

- **Encoder**
  - Produces the context vector $\boldsymbol{c = h_{T_x}}$ of the input sequence
  - **Context vector** c summarizes input sequence $[\mathbf{x_1}, \ldots, \mathbf{x_{T_x}}]$.
- **Decoder**
  - Decodes the encode c to the output sequence
- **Special symbols**
  - <EOS> signifies the end of a sequence
  - <BOS> signifies the beginning of a sequence



Training of seq2seq



- We need to maximize **the log-likelihood**:

$$\max_{\theta}\left(\mathrm{J}(\theta) = \sum_{(x,y)\epsilon\mathcal{D}} \log P\left(\boldsymbol{y}|\boldsymbol{x},\theta\right)\right)$$

where $\theta = [\theta_e, \theta_d]$ and θ_e, θ_d are encoding and decoding parameters respectively.

- Product rule:

$$P(\boldsymbol{y}|\boldsymbol{x},\theta) = P\left(\boldsymbol{y_{1:T_y}} \mid \boldsymbol{x_{1:T_x}},\theta\right) = P\left(\boldsymbol{y_{1:T_y}} \mid \boldsymbol{c},\theta\right)$$

$$= P(\boldsymbol{y_1} \mid \boldsymbol{c},\theta)P(\boldsymbol{y_2} \mid \boldsymbol{y_1},\boldsymbol{c},\theta)\ldots P\left(\boldsymbol{y_j} \mid \boldsymbol{y_{1:j-1}},\boldsymbol{c},\theta\right)\ldots P\left(\boldsymbol{y_{T_y}} \mid \boldsymbol{y_{1:T_y-1}},\boldsymbol{c},\theta\right) = \prod_{j=1}^{T_y} P\left(\boldsymbol{y_j} \mid \boldsymbol{y_{1:j-1}},\boldsymbol{c},\theta\right)$$

$$\log P\left(\boldsymbol{y}|\boldsymbol{x},\theta\right) = \log P\left(\boldsymbol{y}|\boldsymbol{c},\theta\right) = \sum_{j=1}^{T_y} \log P\left(\boldsymbol{y_j}|\boldsymbol{y_{1:j-1}},\boldsymbol{c},\theta\right) = \sum_{j=1}^{T_y} \log P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\boldsymbol{c},\theta\right)$$

  - We can compute $P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\boldsymbol{c}\right) = g\left(\boldsymbol{y_j},\boldsymbol{q_{j-1}},\boldsymbol{c}\right)$ where g is a **nonlinear**, potentially **multi-layered NN** that outputs the probability of y_j.

- Pay attention on how c is used in every step during decoding

Training of seq2seq

- We need to maximize **the log-likelihood**:

$$\max_{\theta}\left(\mathrm{J}(\theta)=\sum_{(\boldsymbol{x},\boldsymbol{y})\epsilon\mathfrak{D}}\log P\left(\boldsymbol{y}|\boldsymbol{x},\theta\right)\right)$$

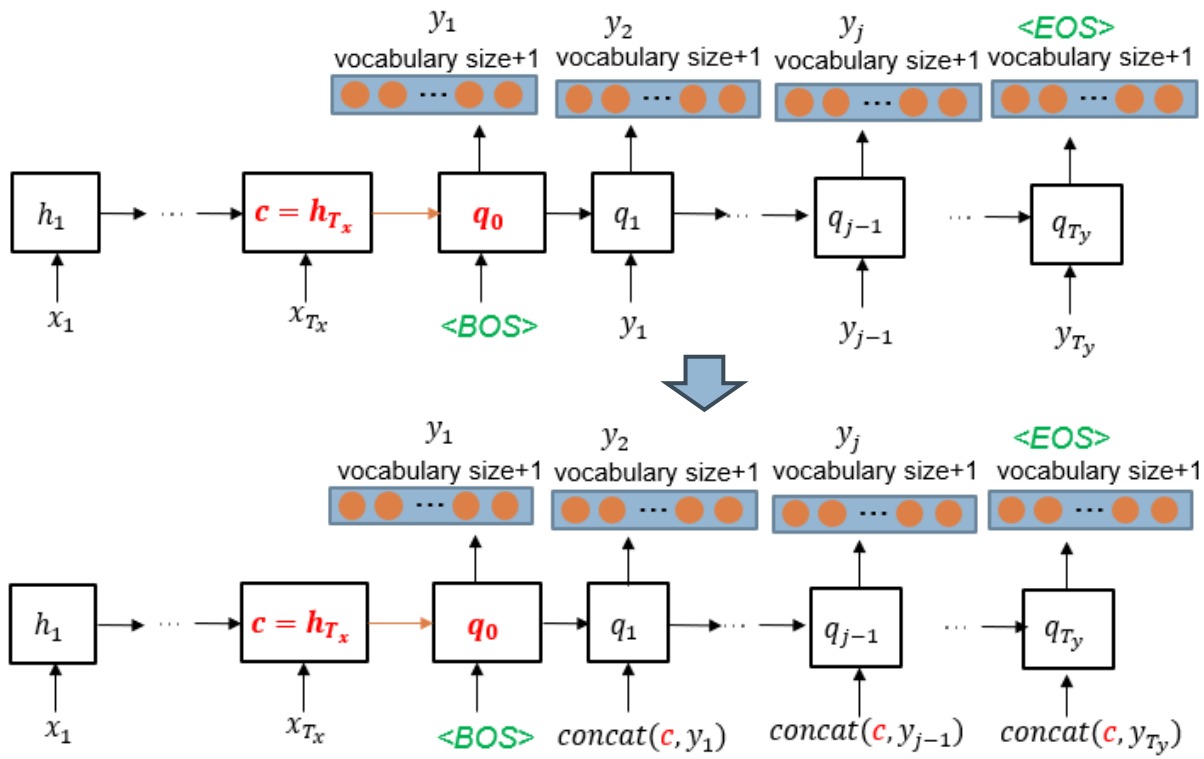
where $\theta = [\theta_e, \theta_d]$ and θ_e, θ_d are encoding and decoding parameters, respectively.

- **Product rule**:

$$\log P\left(\boldsymbol{y}|\boldsymbol{x},\theta\right)=\log P\left(\boldsymbol{y}|\boldsymbol{c},\theta\right)=\sum_{j=1}^{T_y}\log P\left(\boldsymbol{y_j}|\boldsymbol{y_{1:j-1}},\boldsymbol{c},\theta\right)=\sum_{j=1}^{T_y}\log P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\boldsymbol{c},\theta\right)$$

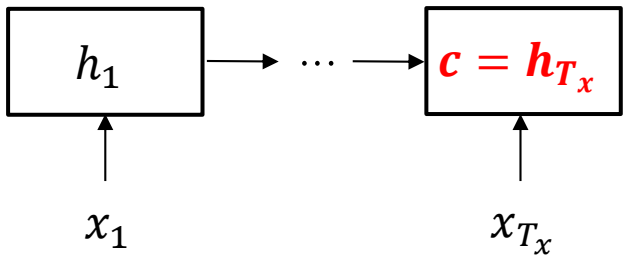
  - $\boldsymbol{q_{j-1}}$ contains the information of $\boldsymbol{c}$.
  - (1): $P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\boldsymbol{c},\theta\right)=P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\theta\right)$
    $q_{j-1}=f(q_{j-2},y_{j-1})$
  - (2): $P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\boldsymbol{c},\theta\right)=P\left(\boldsymbol{y_j}|\boldsymbol{q_{j-1}},\theta\right)$
    $q_{j-1}=f(q_{j-2},concat(c,y_{j-1}))$
  - f is a **memory cell** (e.g., LSTM or GRU)

(1) [Sutskever et al. 2014]

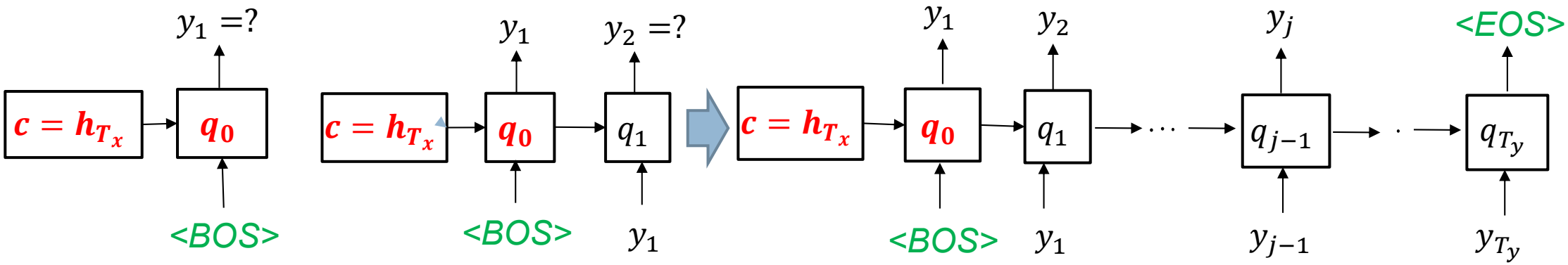


Big drawback: the **context vector** $\boldsymbol{c}$ is **fixed** across timesteps.

Inference



We know $P(y_1 = \circ | c)$ We know $P(y_2 = \circ | c)$

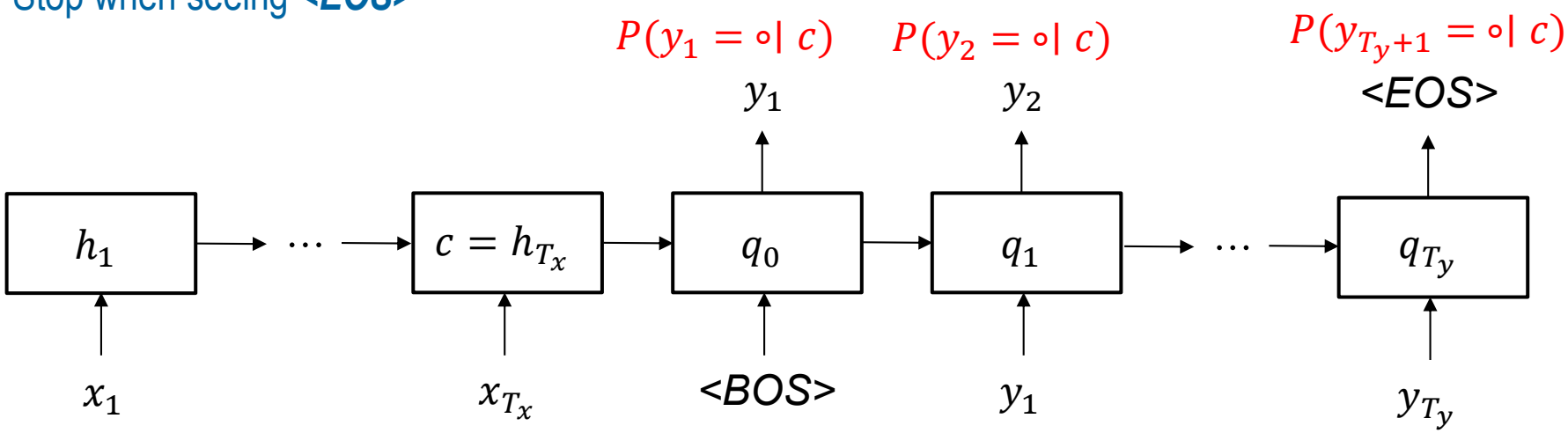


- Given a **trained model** and **input sequence x**
  - We need to infer the corresponding **output sequence y**
- Two common strategies
  - **Greedy Decoding** and **Beam Search Decoding**

Inference

Greedy decoding

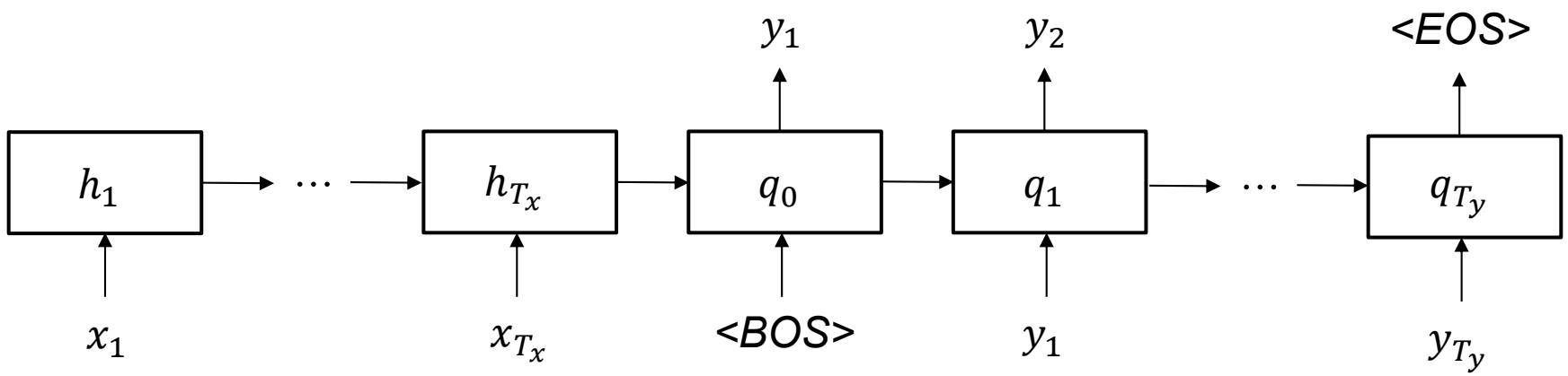
- Greedy Decoding
  - Given **x**, find word $\boldsymbol{y_1}$ with highest probability
  - Given $\boldsymbol{y_1}$ and **x**, find word $\boldsymbol{y_2}$ with highest probability
  - ...
  - Stop when seeing ***<EOS>***



Inference

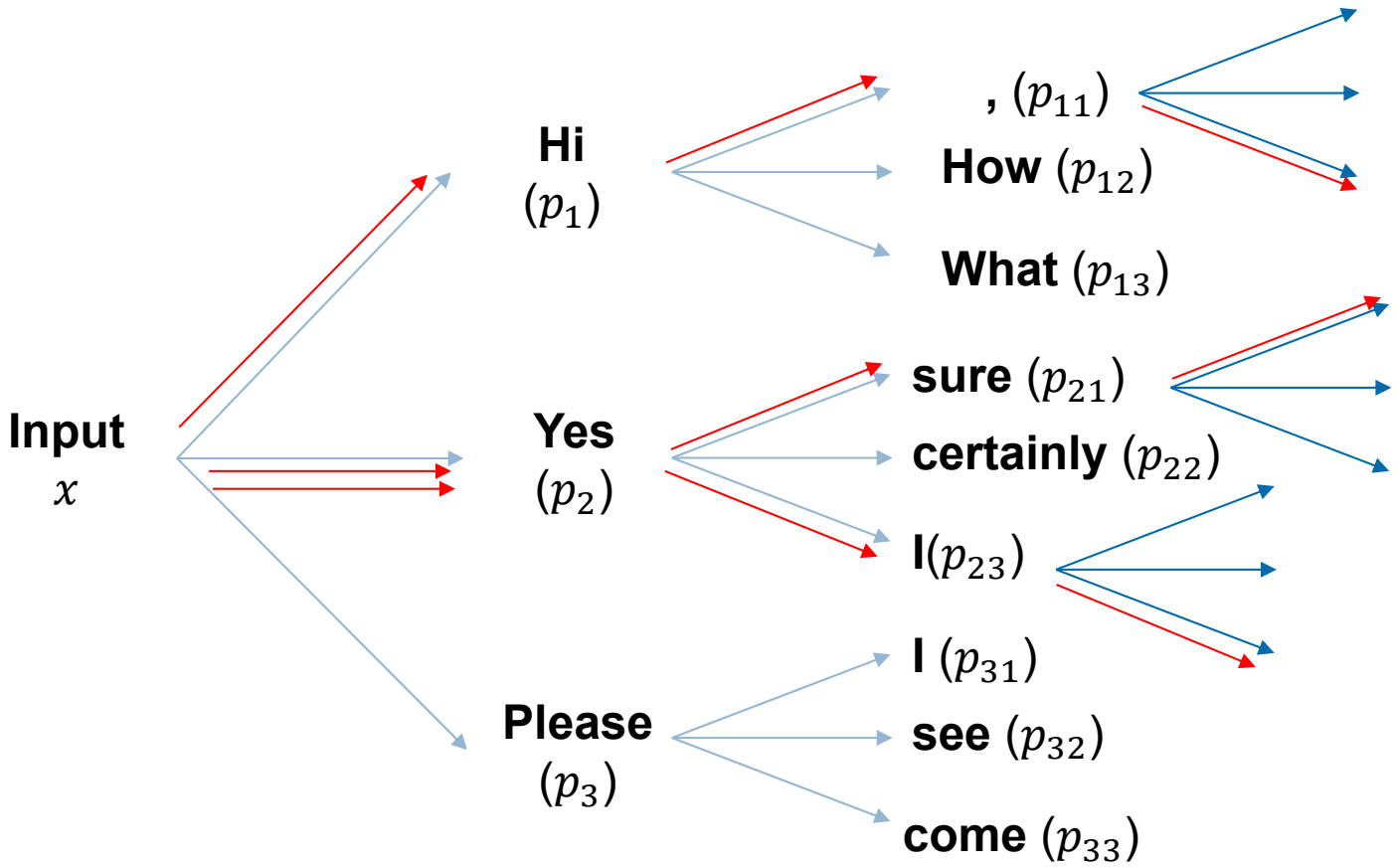
Beam search

- Beam Search Decoding with **beam width k**
  - Given **x**, find **k** candidates for y_1 with highest probability
  - Given **x**, for each candidate y_1, find **k** candidates for word y_2 with highest probability
  - Pick top-**k** sequences $y_1 y_2$ with highest **joint probability**
  - For each $y_1 y_2$, find **k** candidates for word y_3 with highest probability
  - Pick top-**k** sequences $y_1 y_2 y_3$ with highest **joint probability**
  - …..
  - Stop when see **<*EOS*>** on each beam
  - Finally, pick 1 sequence with **highest probability** from **top-k sequences**



Inference

Beam search



Joint Probability

$p_1 p_{11} \cdots$
$p_1 p_{12} \cdots$
$p_1 p_{13} \cdots$
$p_2 p_{11} \cdots$
$\cdots$

Beam width = 3

- We always choose three sentences with **highest joint probabilities**

Drawback of fixed context

- **Fixed** context vector c is **easily overwhelmed** by long inputs or long outputs.
- At a specific timestep j, some **words** or **items** in the **input sequence** might **possibly contribute more** to the **generation** of **next item** or **word** in the output sequence.
  - I want to see you every day → Je veux te voir chaque jour
  - I want to see **you** every day → Je veux te **?** (voir) ....
- How to **timely adapt** the **context vector** c_j?
  - $c_j = \alpha(h_1, .., h_{T_x}, q_{j-1})$
  - Computed using **attention mechanism**

Published as a conference paper at ICLR 2015

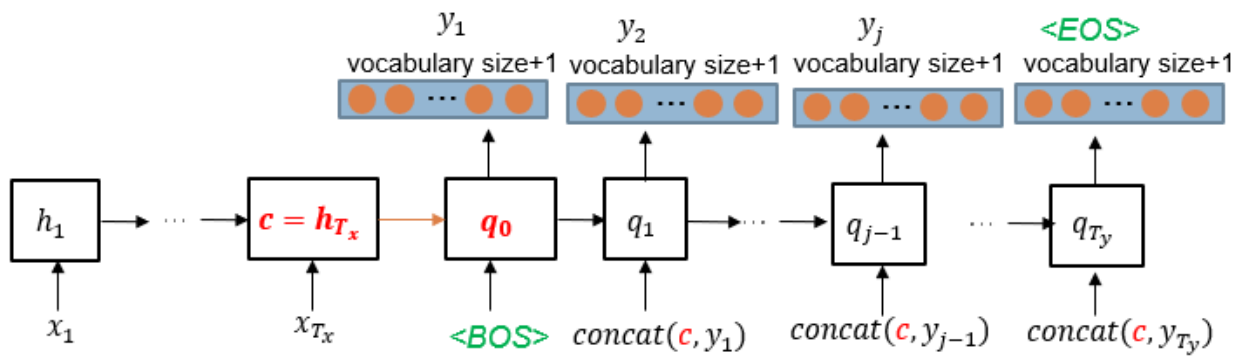
NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

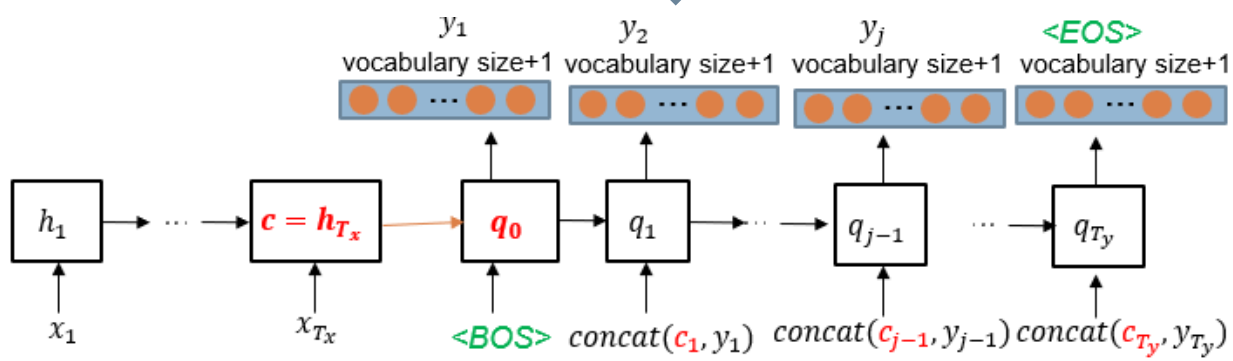
Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

Paper: paper link

Fixed context vector c





Timely varied context vectors c_j

Attention mechanism

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

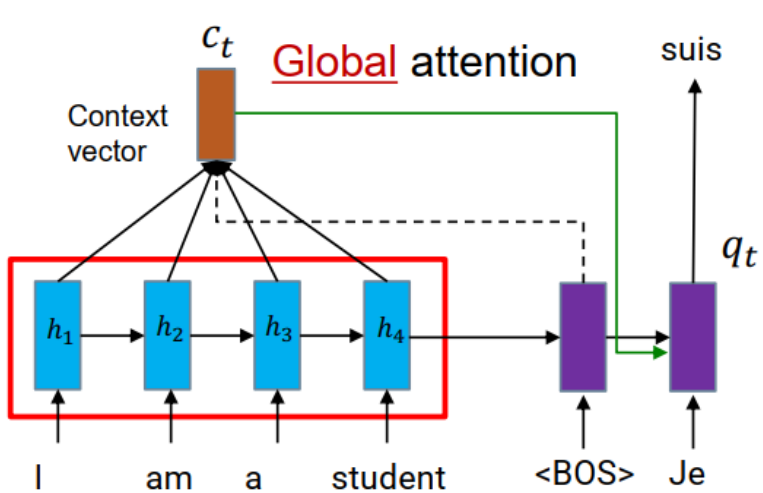
KyungHyun Cho **Yoshua Bengio***
Université de Montréal

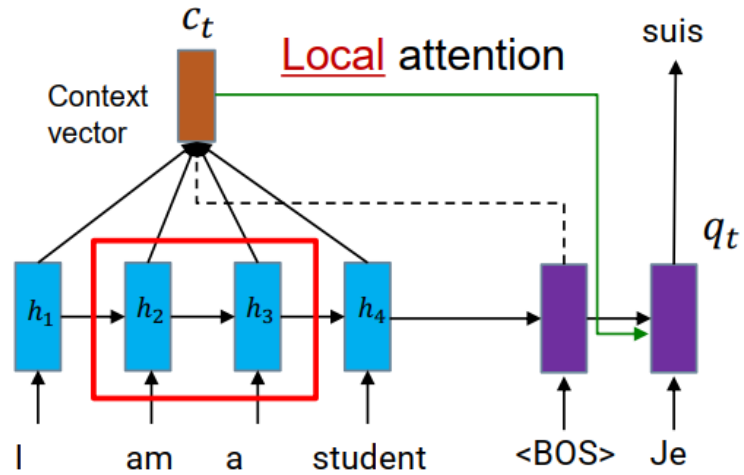
Bahdanau, Cho, Bengio, Neural Machine Translation by Jointly Learning to Align and Translate, ICLR 2015

Effective Approaches to Attention-based Neural Machine Translation

Minh-Thang Luong **Hieu Pham** **Christopher D. Manning**
Computer Science Department, Stanford University, Stanford, CA 94305
{lmthang,hyhieu,manning}@stanford.edu

Luong, Pham, Manning, Effective Approach Attention-based Neural Machine Translation, EMNLP, 2015





- **Attention mechanism allows the decoding network to refer to the input.**
  - **Global attention**
    - Use **all input hidden states** of the encoder when deriving the context c_t.
  - **Local attention**
    - Use a **selective window of input hidden states** of the encoder when deriving the context c_t.

Global attention

- Main idea
  - Consider **all input hidden states** of the encoder when deriving the context: $c_t = \sum_{s=1}^{3} a_t(s)\, h_s$

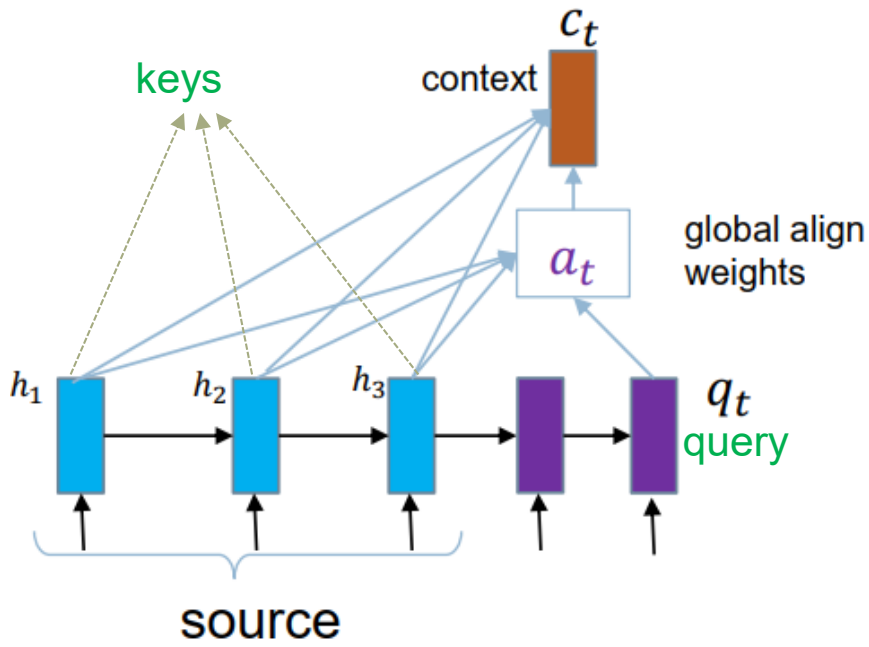
- Alignment weights:

$$a_t(s) = \text{align}(q_t, h_s) = \frac{\exp\big(score(q_t, h_s)\big)}{\sum_{s'} \exp\big(score(q_t, h_{s'})\big)}$$

where q_t is current target state, h_s is each source states.

- **Alignment score** function:

$$score(q_t, h_s) = \begin{cases} q_t^{\mathrm{T}} h_s & dot\ product \\ q_t^{\mathrm{T}} W_a h_s & general\ metric \\ v_a^{\mathrm{T}} tanh(W_a[q_t; h_s]) & concat \end{cases}$$



Global attention

- Context vector:

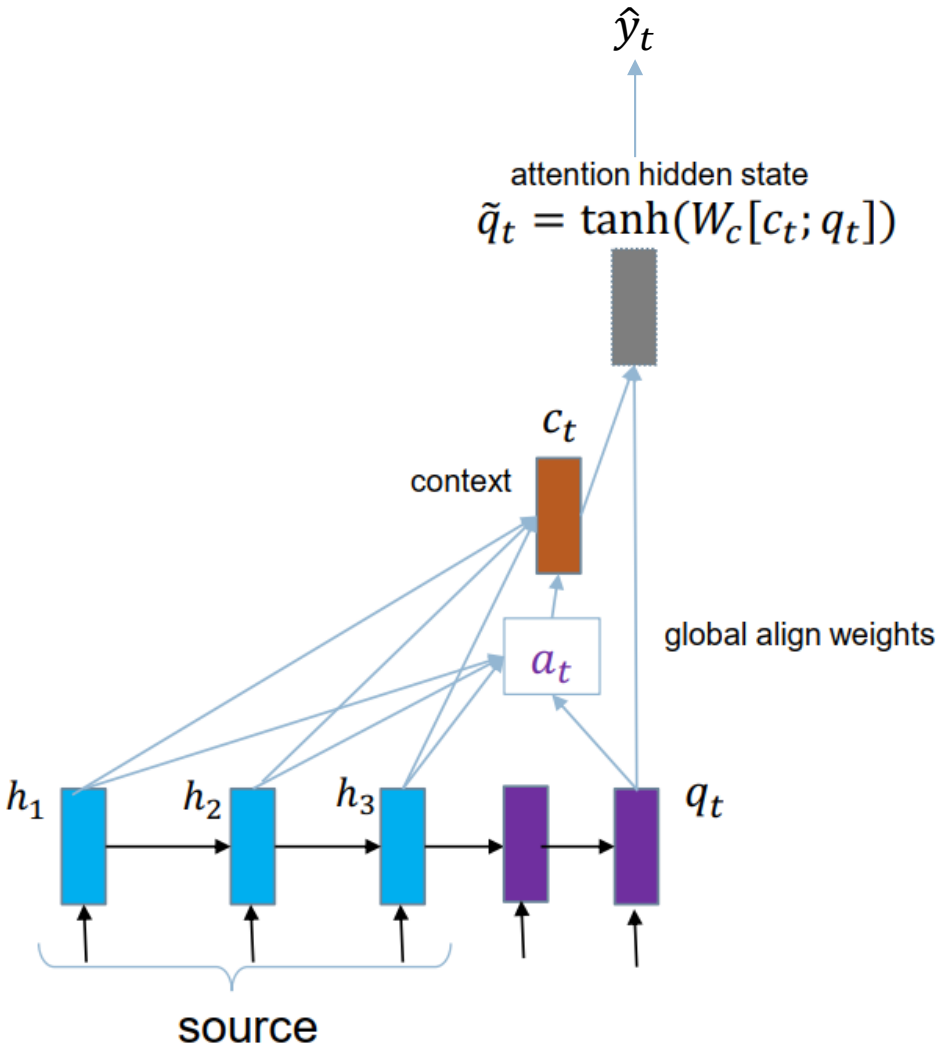
$$c_t = \sum_{s=1}^{3} a_t(s)\, h_s$$

- Attentional hidden state:

$$\tilde{q}_t = tanh(W_c[c_t; q_t])$$

- Predictive distribution:

$$p(y_t|y_{<t}, x) = softmax(W_s \tilde{q}_t)$$



Global attention

Example

1. Convert into alignment weights.

$$a_t(s) = \frac{exp\big(score(q_t, h_s)\big)}{\sum_{s'} exp\big(score(q_t, h_{s'})\big)}$$

2. Build context vector: weighted average

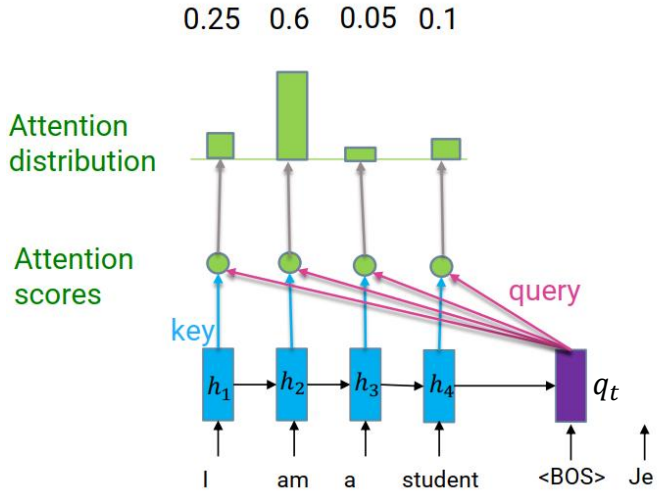
$$c_t = \sum_s a_t(s)\, h_s$$

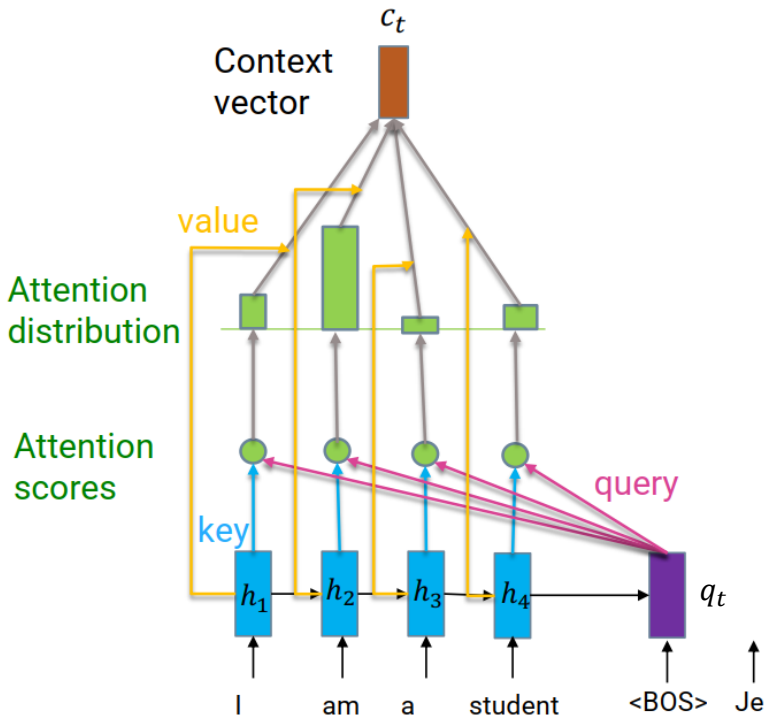
3. Compute the next hidden state.

$$\tilde{q}_t = tanh(W_c[c_t; q_t])$$

4. Predictive distribution:

$$p(y_t | y_{<t}, x) = softmax(W_s \tilde{q}_t)$$



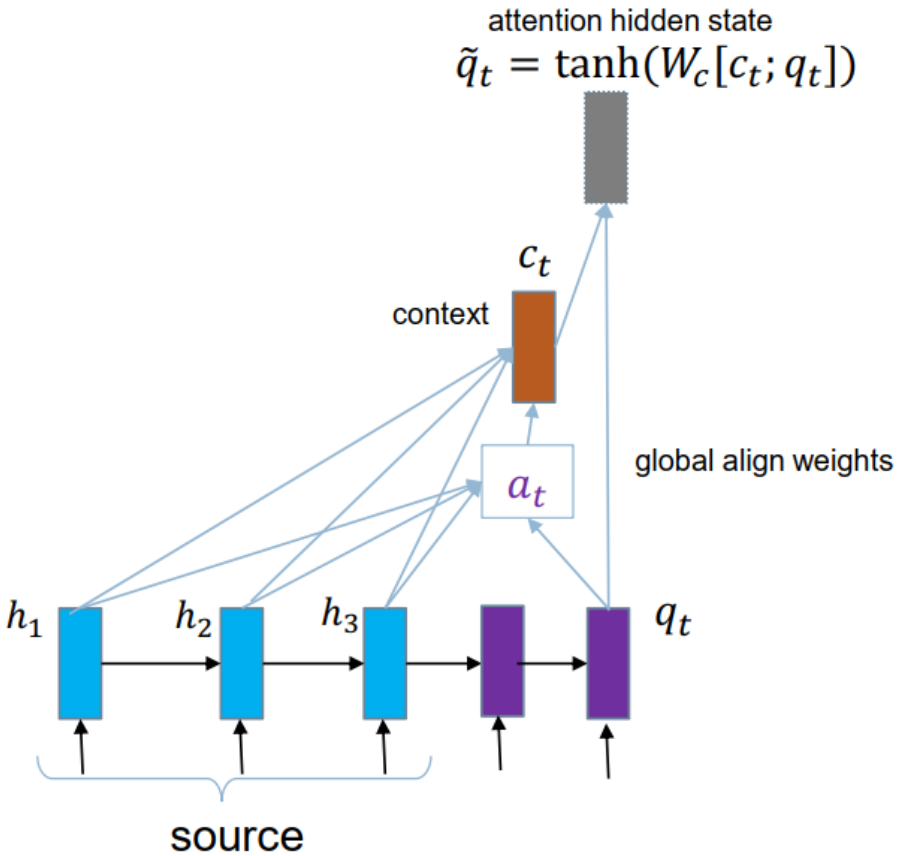


Global attention

Drawback

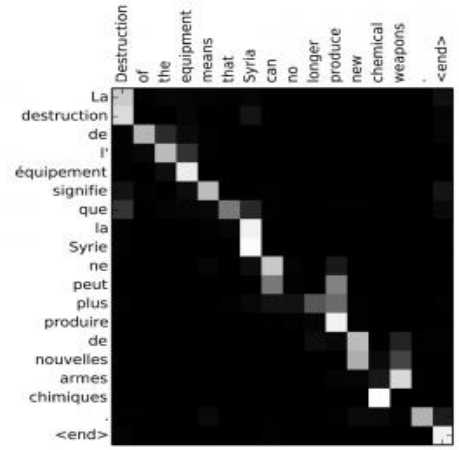
- **Drawback**: employ **all items** on the source side for deriving each target item.
  - expensive computation
  - impractical to translate longer sequences

Local attention

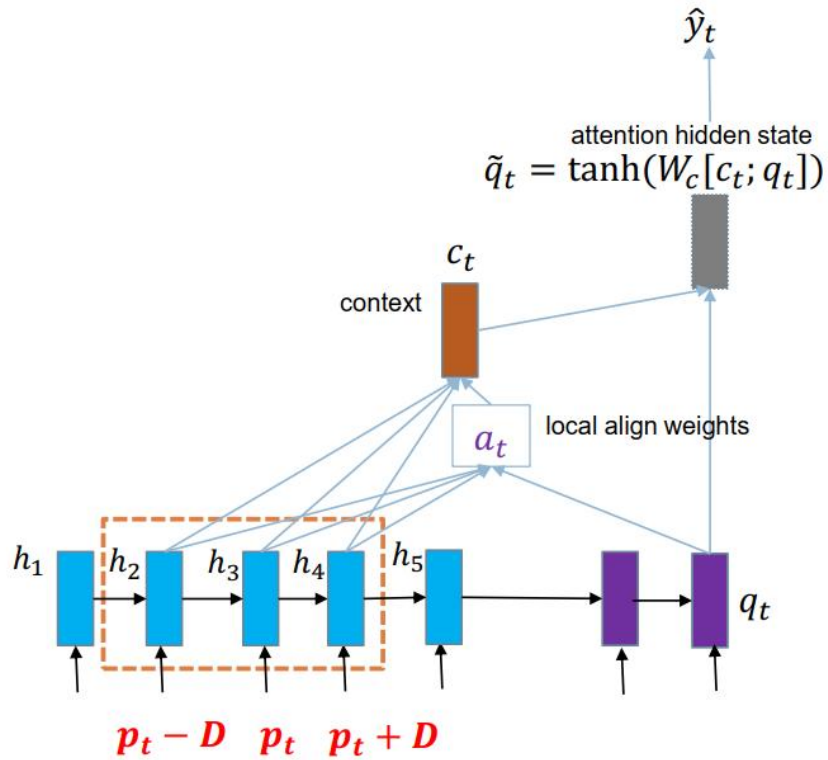


Local attention

- Main idea
  - Selectively focuses on a **small window** of context and is differentiable.



- c_t is then derived as a **weighted average** over the set of **source hidden states within the window** $[\boldsymbol{p_t - D}; \boldsymbol{p_t + D}]$, D is empirically selected.



Local attention

- Main idea
  - Selectively focuses on a **small window** of context and is differentiable
- For current target word, predict aligned position p_t
  - $p_t = S \cdot \text{sigmoid}\left(v_p^T \tanh\left(W_p q_t\right)\right)$

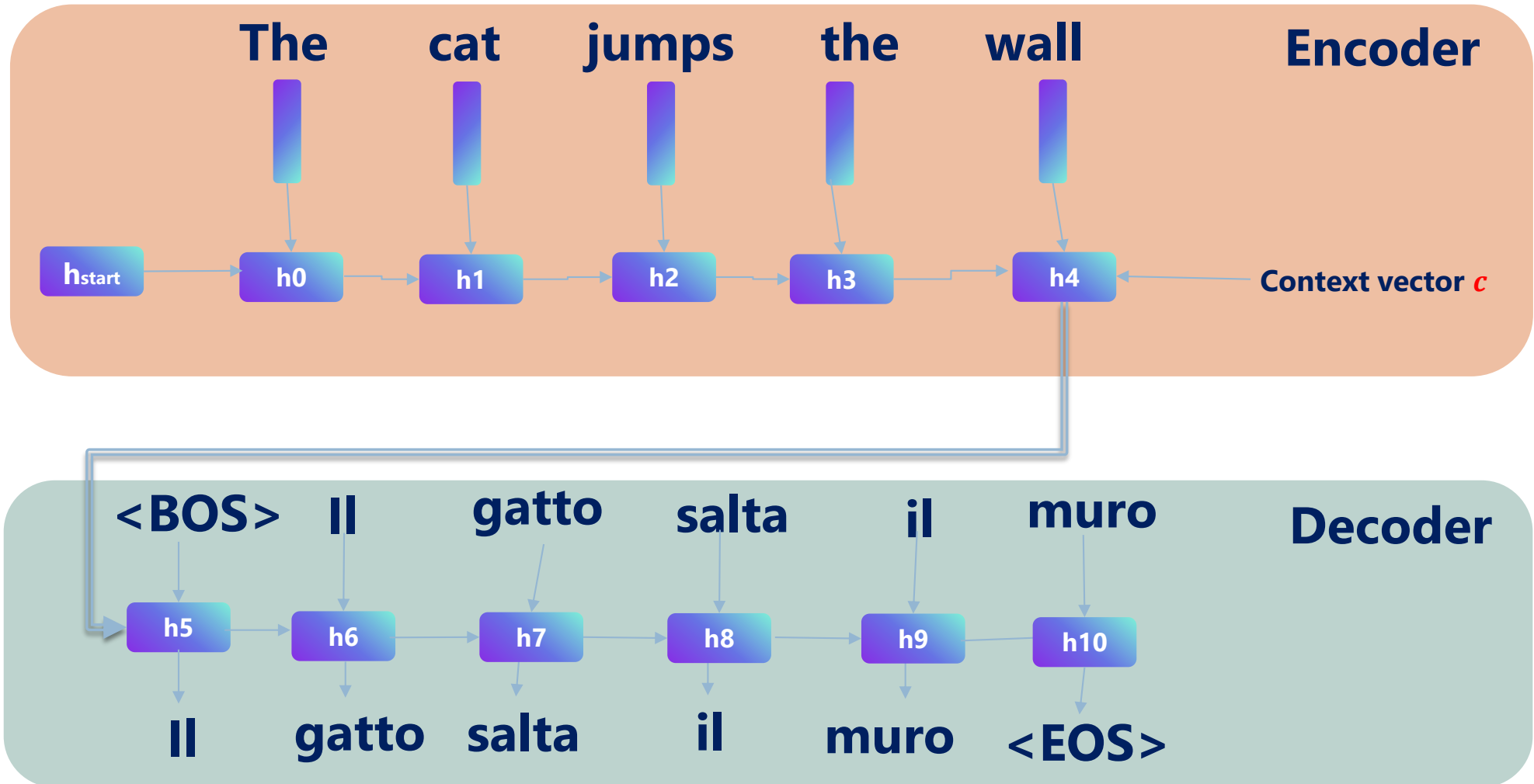
  where W_p and v_p are the model parameters, S is the source sentence length, and hence $0 \leq p_t \leq S$
- Alignment weights:
  - $a_t(s) = \text{align}(q_t, h_s)\exp\left(-\frac{(s-p_t)^2}{2\sigma^2}\right)$ with $\sigma = {}^{D}/_{2}$
  - $\text{align}(q_t, h_s) = \frac{\exp(score(q_t, h_s))}{\sum_{s'} \exp(score(q_t, h_{s'}))}$ (**temporary alignment weights**)
- Context is now **computed** as before but **within a windows of size D**, centered at p_t

- Main idea
  - Selectively focuses on a **small window** of context and is differentiable
- For current target word, predict aligned position p_t
  - $p_t = S \cdot \text{sigmoid}\left(v_p^T \tanh\left(W_p q_t\right)\right)$

  where W_p and v_p are the model parameters, S is the source sentence length, and hence $0 \leq p_t \leq S$
- Alignment weights:
  - $a_t(s) = \text{align}(q_t, h_s)\exp\left(-\frac{(s-p_t)^2}{2\sigma^2}\right)$ with $\sigma = {}^{D}/_{2}$
  - $\text{align}(q_t, h_s) = \frac{\exp(score(q_t, h_s))}{\sum_{s'} \exp(score(q_t, h_{s'}))}$ (**temporary alignment weights**)
- Context is now **computed** as before but **within a windows of size D**, centered at p_t

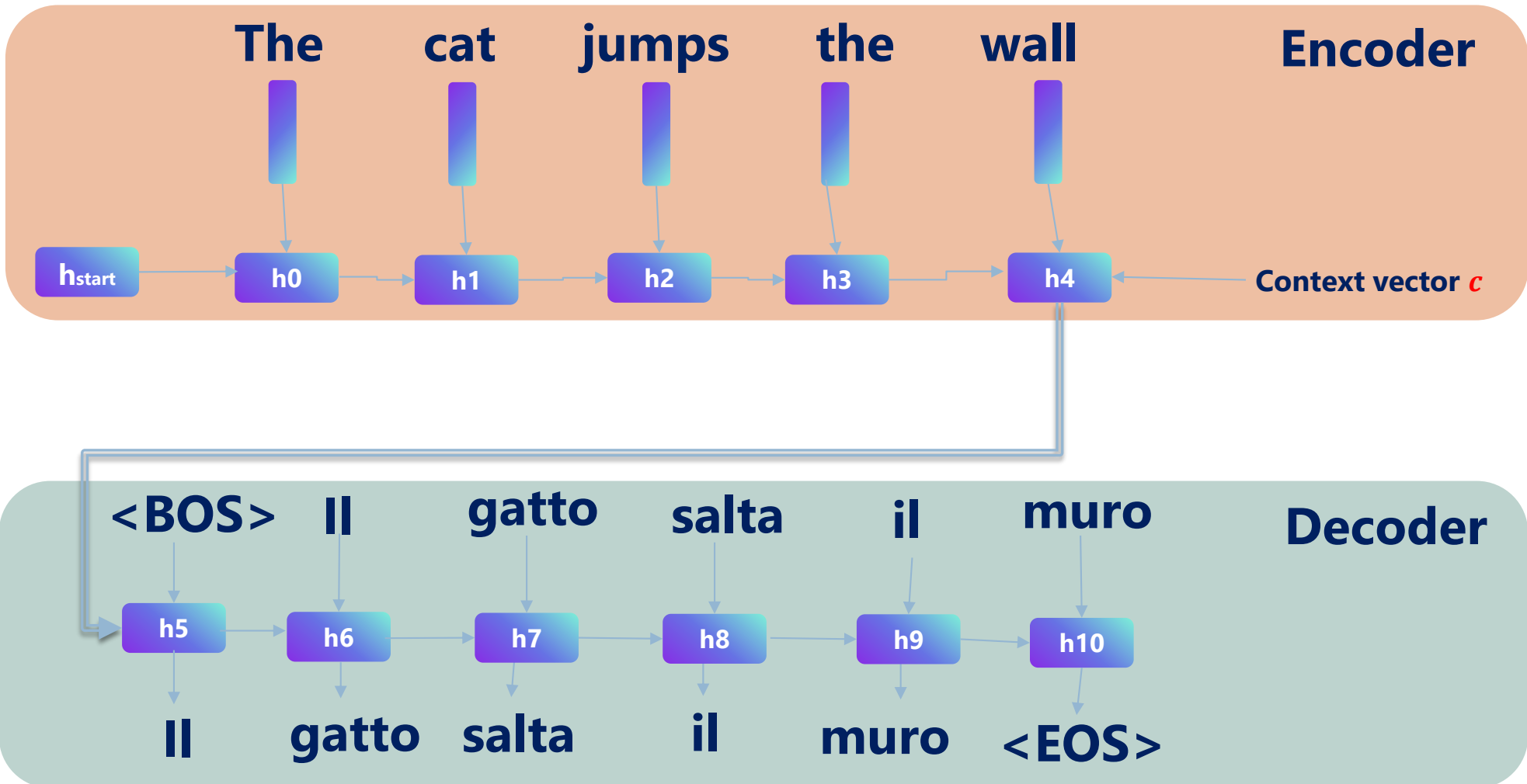
Transformers

A step back: Recurrent Networks (RNNs)



Problems

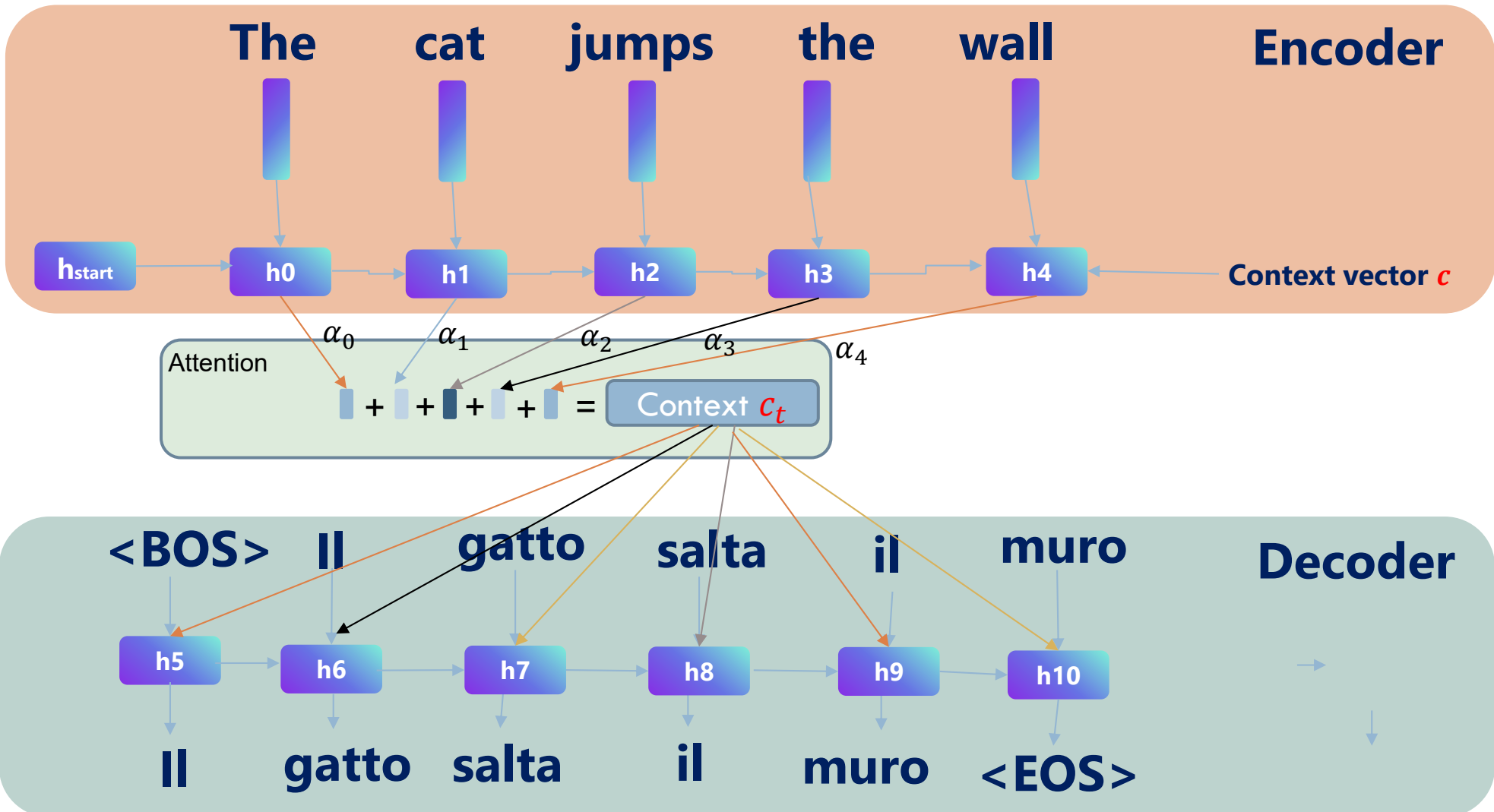
1. We **forget tokens too far** in the past in the context vector **c**
2. We need to wait the previous token to compute the next hidden-state



Solving problem 1

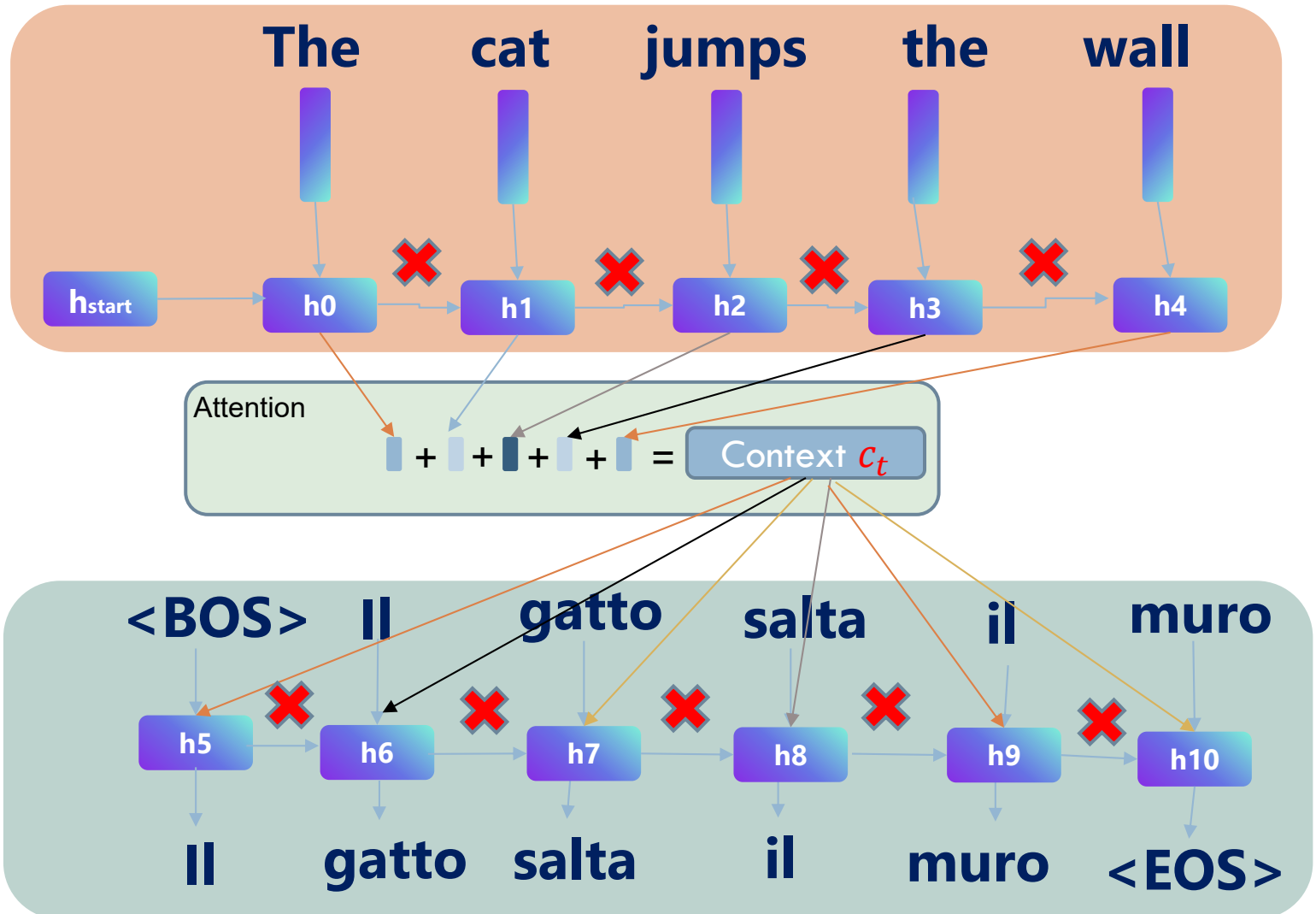
"We **forget tokens too far** in the past"

Solution: Add an **attention** mechanism



Solving problem 2

"We need to wait the previous token to compute the next hidden-state"

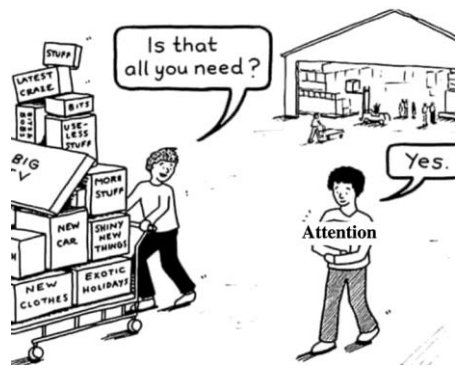


Solution

Throw away **recurrent connections**

2017 paper

"Attention Is All You Need"

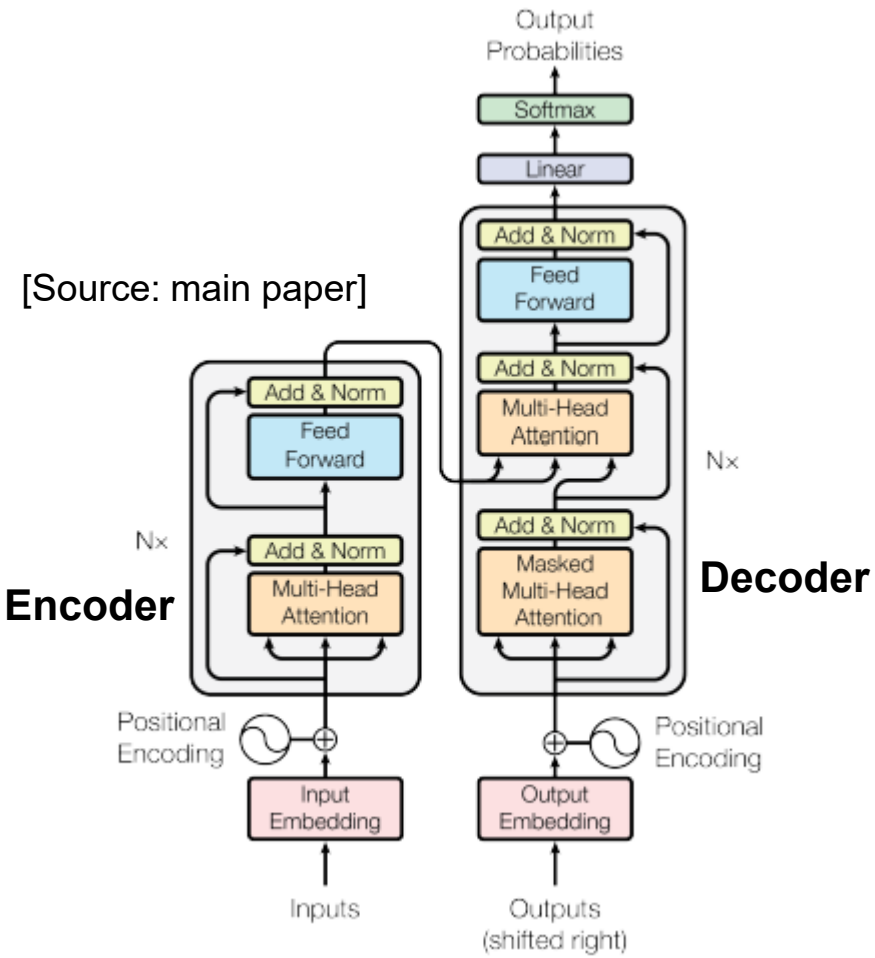


Transformer Motivations

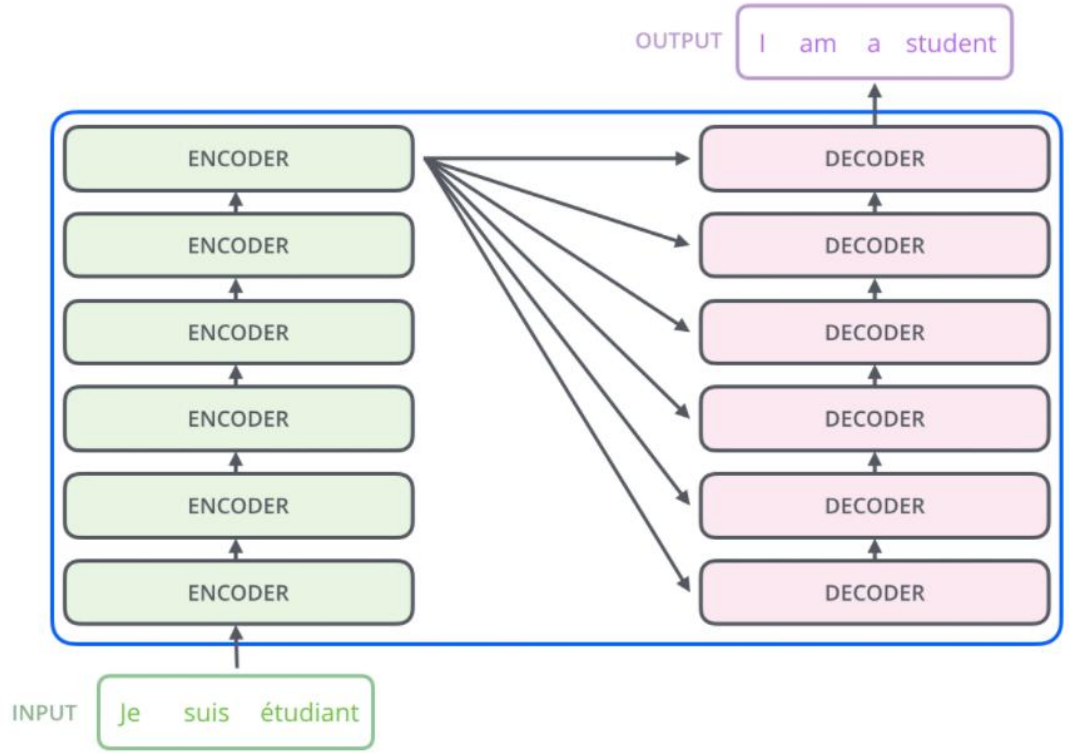
- Problem using RNNs for Seq2Seq models
  - Slow due to sequential nature
  - Poor long-range dependency modelling
- Transformer
  - Enable **parallel computation**, hence exploit the power of GPU. How?
  - Remove the **dependency** between words
  - But **position/location** matters → resolve by **positional encoding**
  - Extremely good at capturing **long-range dependency**

Transformer – General Architecture

[Source: main paper]



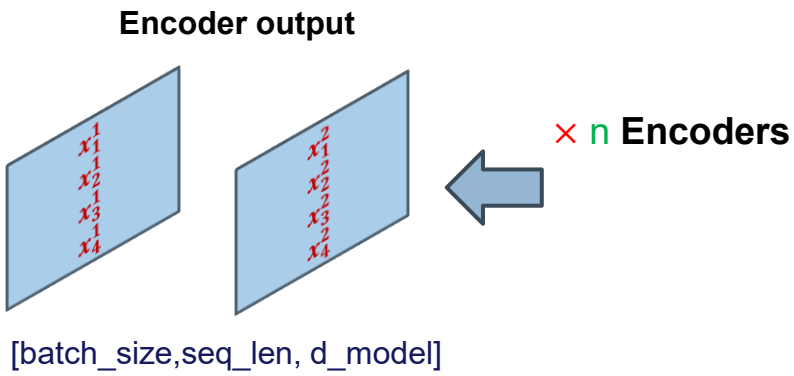
Model architecture



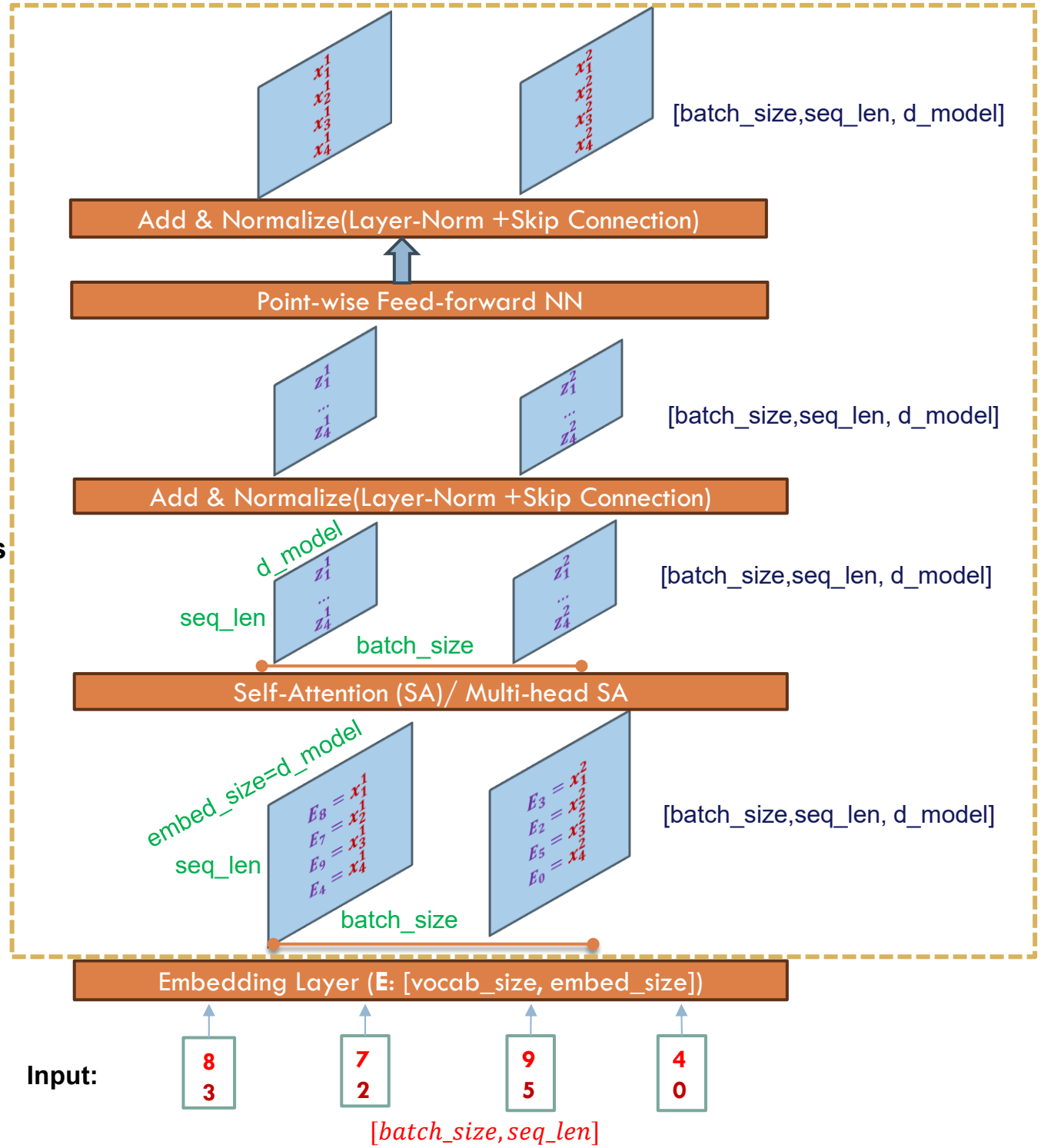
[Source: https://jalammar.github.io/illustrated-transformer/]

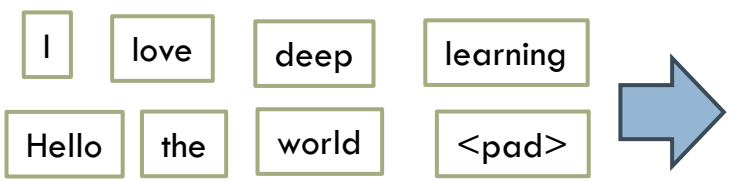
- Some **building blocks**
  - Positional encoding
  - Self-attention and Multi-Head Self-attention Attention
  - Masked Multi-Head Attention
  - Residual add and layer norm

Overview Architecture of Transformer Encoder



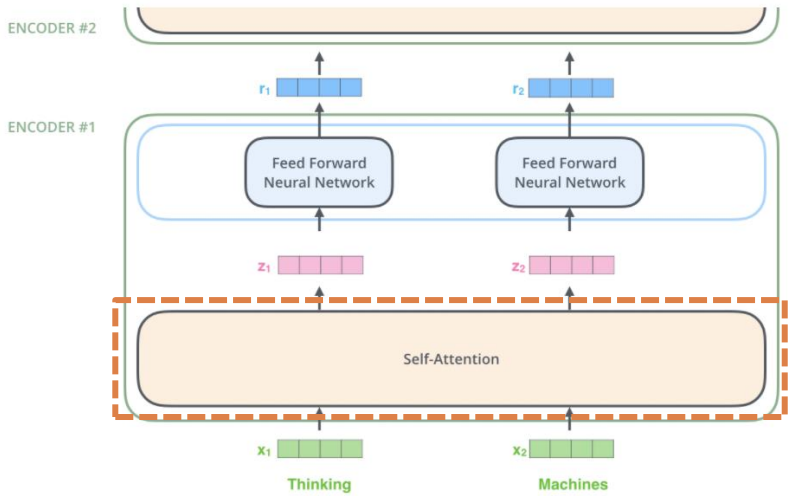
× n **Encoders**

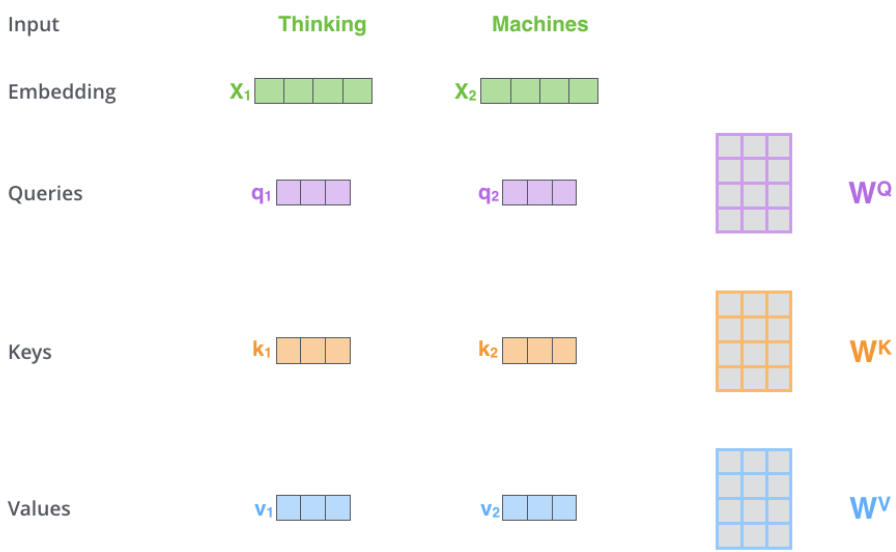




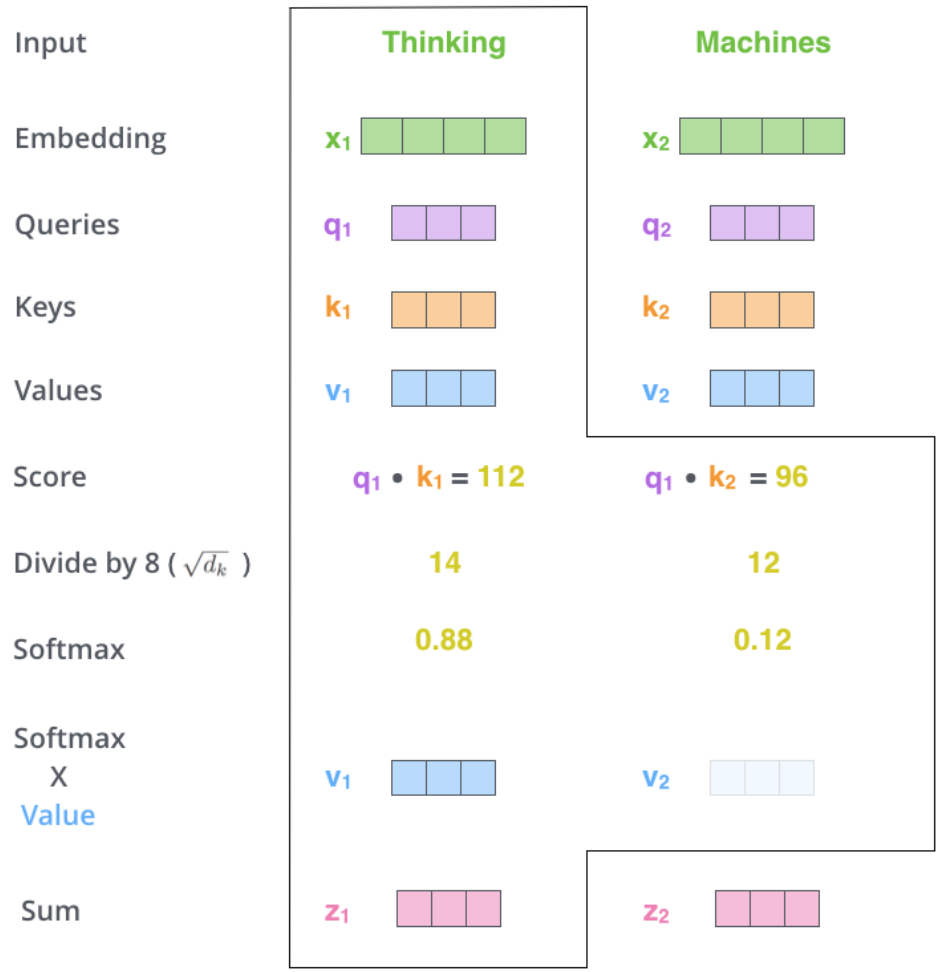
Transformer – Self Attention

[Source: https://jalammar.github.io/illustrated-transformer/]





Multiplying x_1, x_2 with W^Q, W^K, W^V to gain **queries**, **keys**, and **values**.



$$z_1 = 0.88 \times v_1 + 0.12 \times v_2$$

- **Self attention** among a sequence of **items/tokens**
  - **Keys** and **queries** for computing self attention weights
  - W^Q, W^K, W^V are **learnable matrices**.

Transformer

Self-Attention

- For each token x_i, calculate its query, key, and value

  $q_i = x_i W^Q, k_i = x_i W^K, v_i = x_i W^V$

- **Matrix/stacked** version ($X = \begin{bmatrix} x_1 \\ \cdots \\ x_L \end{bmatrix}, L = seq\_len$)

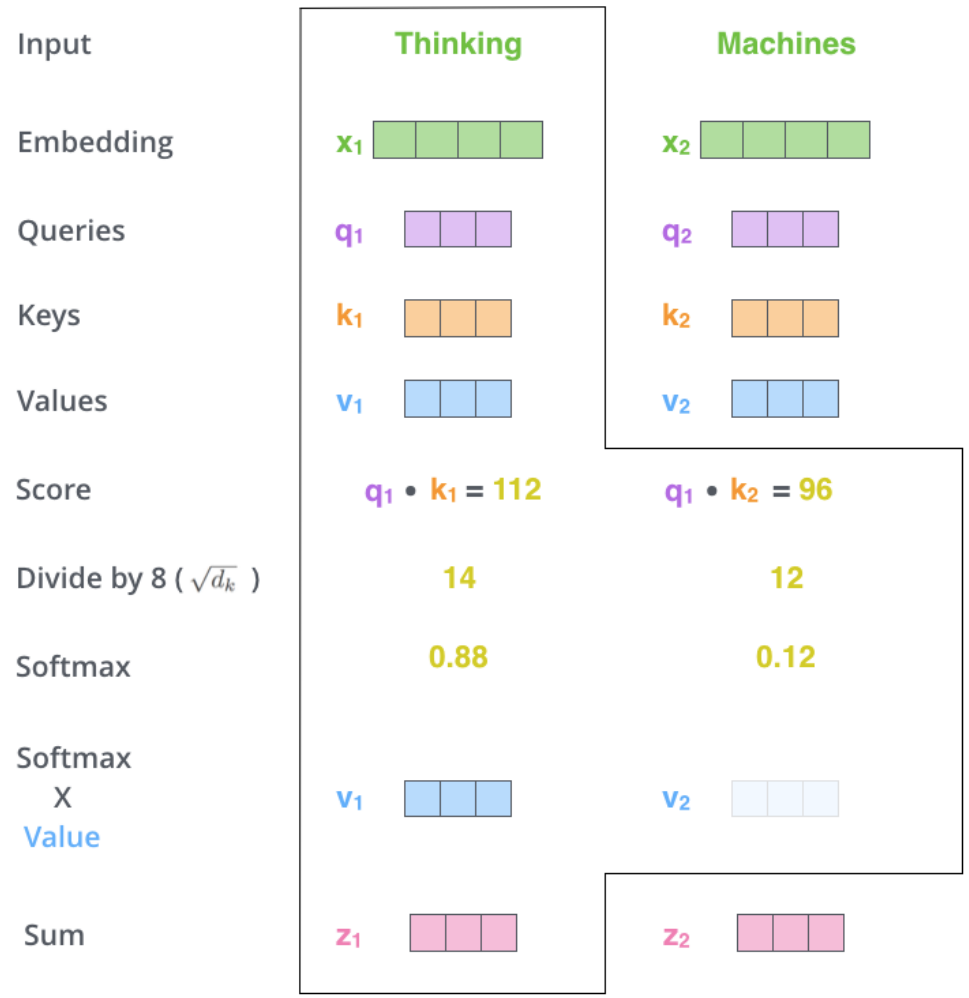
  $Q = XW^Q, K = XW^K, V = XW^V$

- Calculate the **attention probabilities** between query and key
  - Scaled dot-product attention

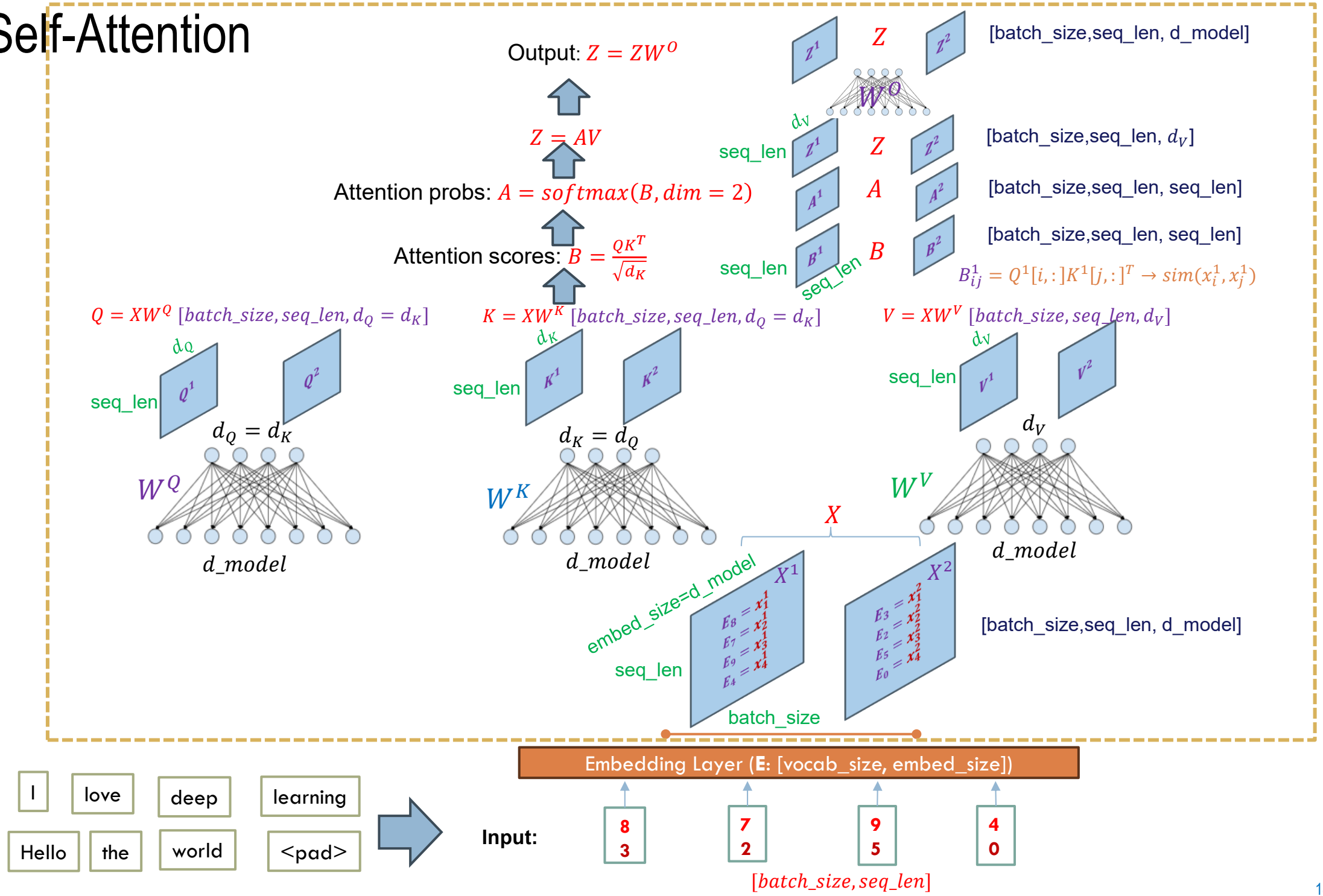
$$A = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

- Take a **weighted sum** of values

$$Z = AV$$



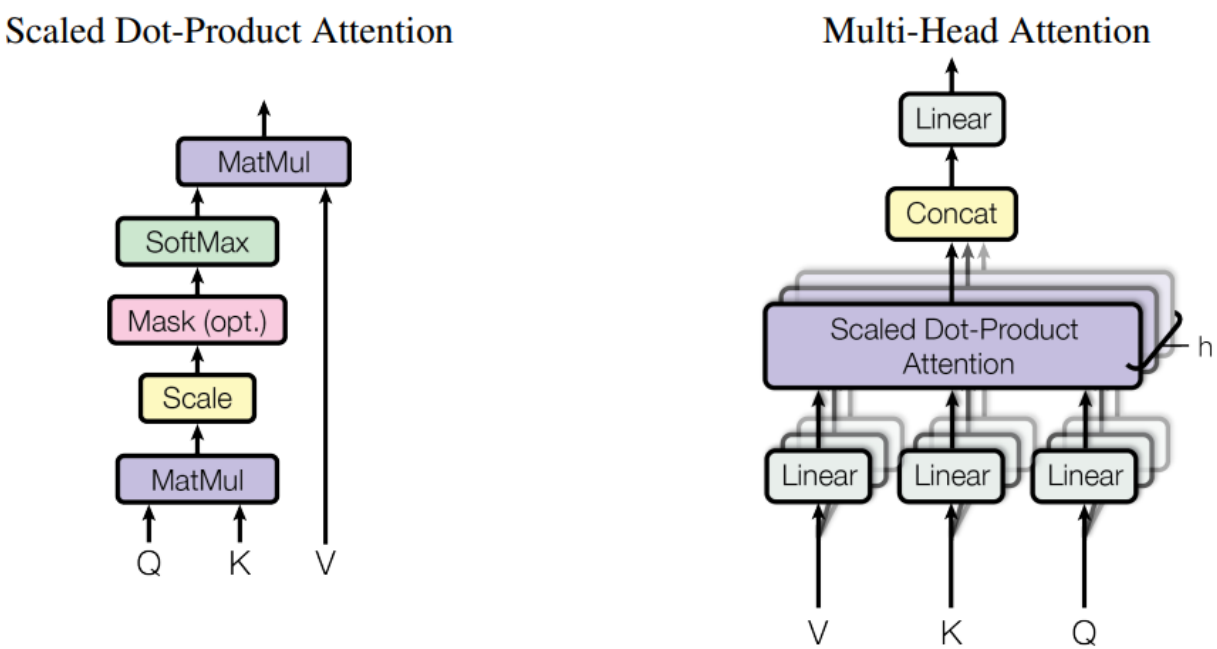
Self-Attention



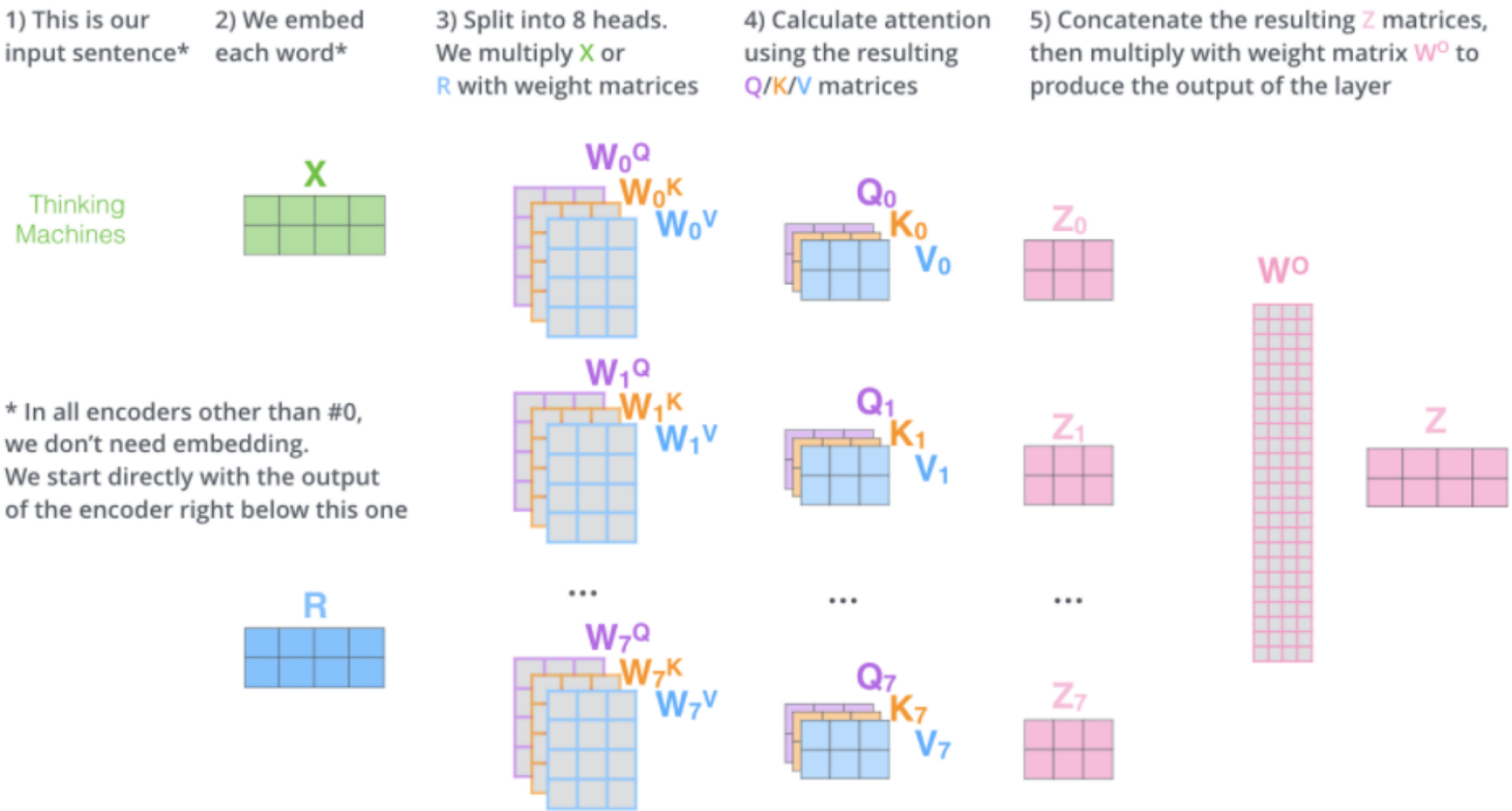
Transformer

Multi-Head Self-Attention

- Main idea:
  - Perform self-attention multiple times in parallel and combine results
- Multiple attention heads through multiple Q, K, V matrices
- Each attention head performs attention independently
  - Allow attending to parts of the sequence differently



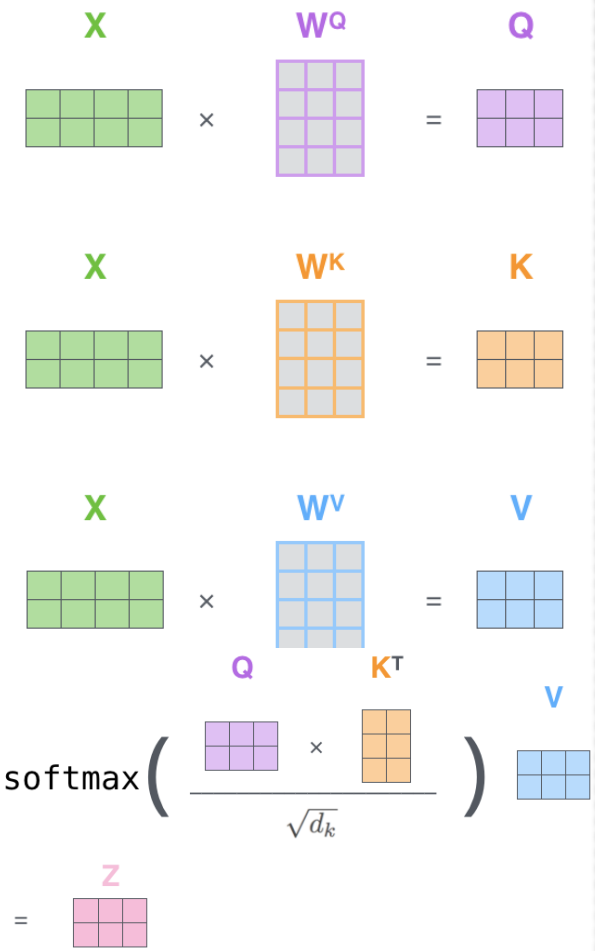
Transformer – Self Attention/Multi-Head Attention



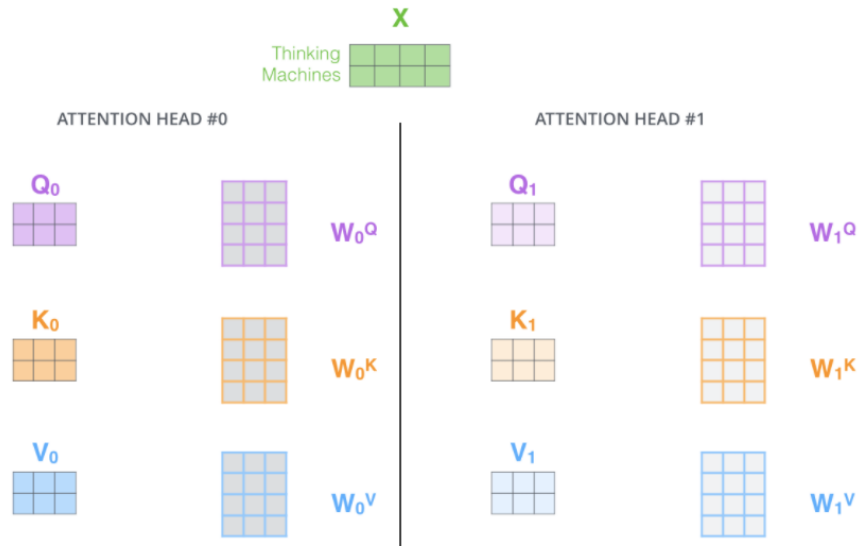
[Source: https://jalammar.github.io/illustrated-transformer/]

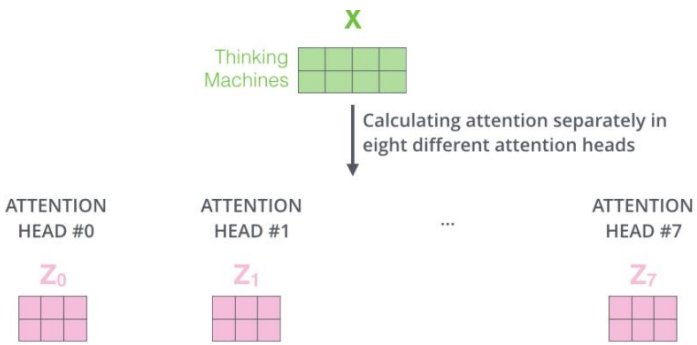
Transformer – Self Attention/Multi-Head Attention

Matrix calculation for One-Head



Matrix calculation for Multi-Head





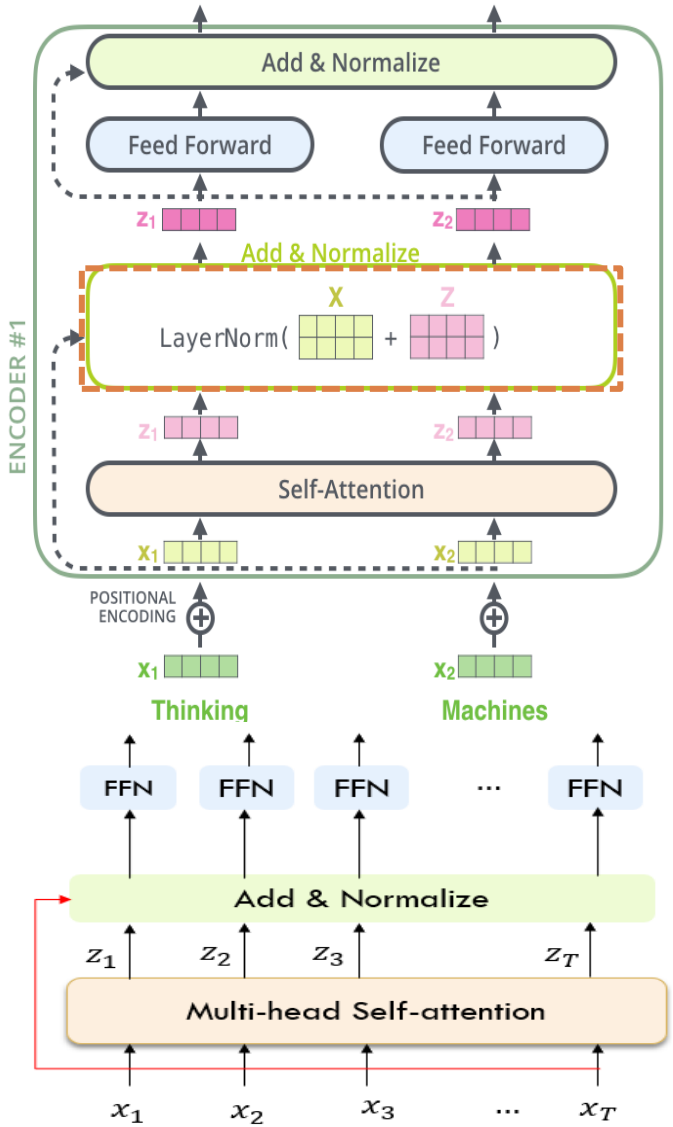
2) Multiply with a weight matrix W^O that was trained jointly with the model

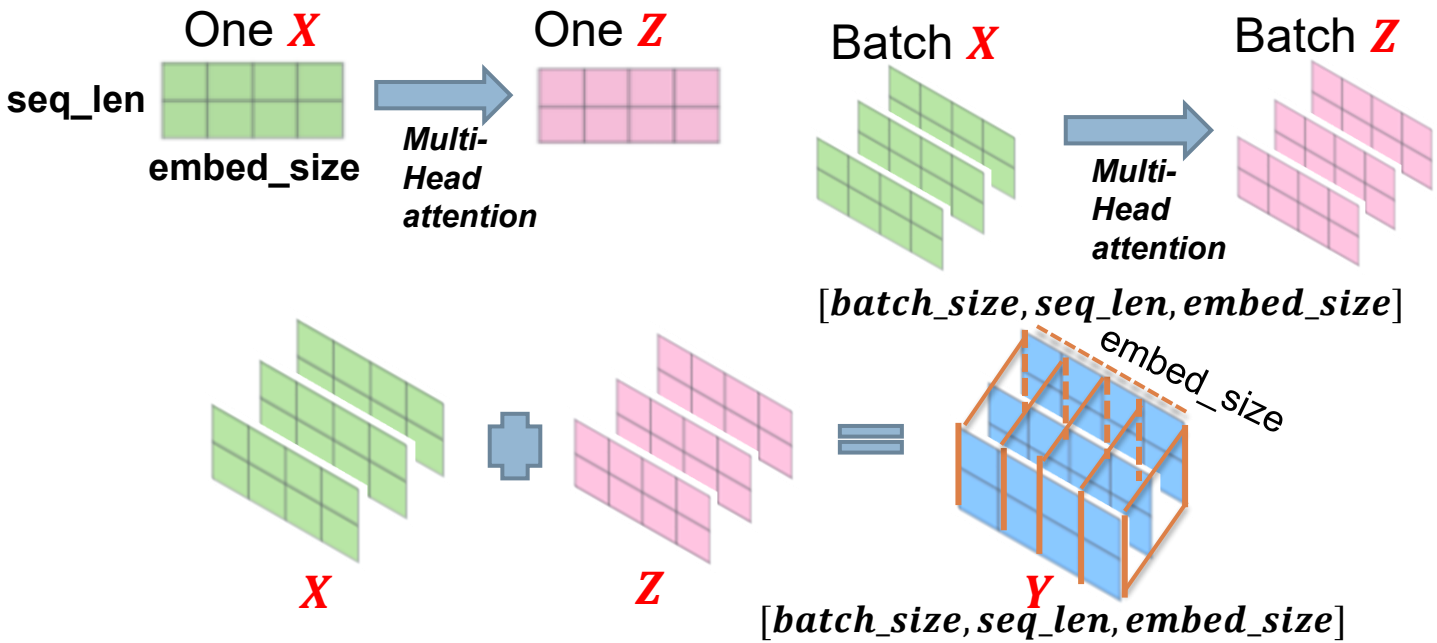
X

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

[Source: https://jalammar.github.io/illustrated-transformer/]

Transformer – Residual Connection/Layer Norm

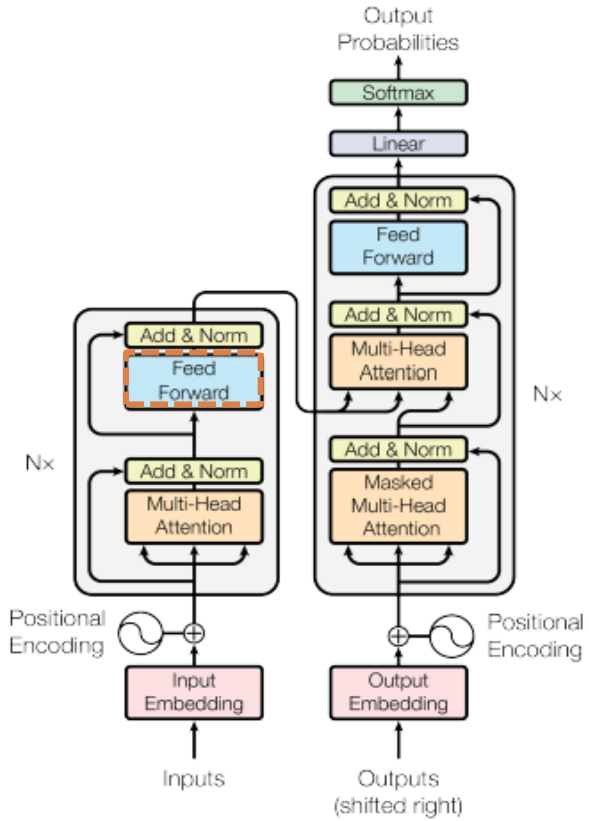


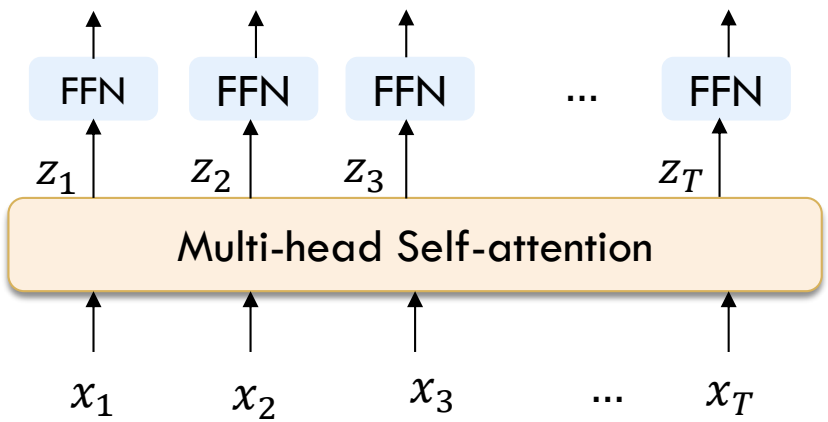


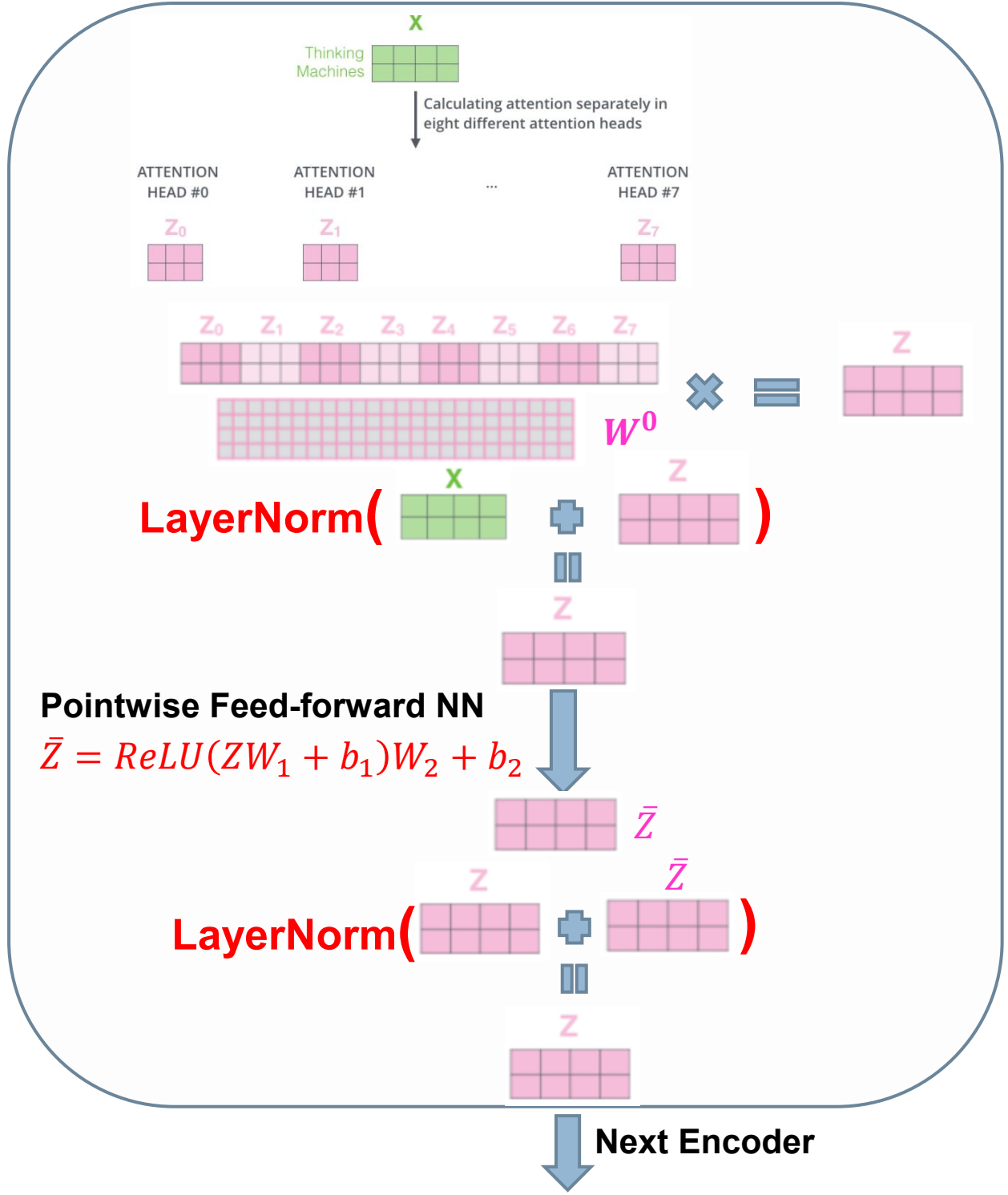
$$Y_m = mean(Y, axis = embed\_size, keep\_dims = True)$$
$$Y_\sigma = std(Y, axis = embed\_size, keep\_dims = True)$$
$$LayerNorm(Y) = \gamma \frac{Y - Y_m}{Y_\sigma} + \beta$$

- **Layer normalization** (LN) is the same as **batch normalization** except that LN **normalizes** across the **feature dimension**.
  - **Batch normalization** is **usually empirically less effective** than **layer normalization** in natural language processing tasks, whose inputs are often variable-length sequences.

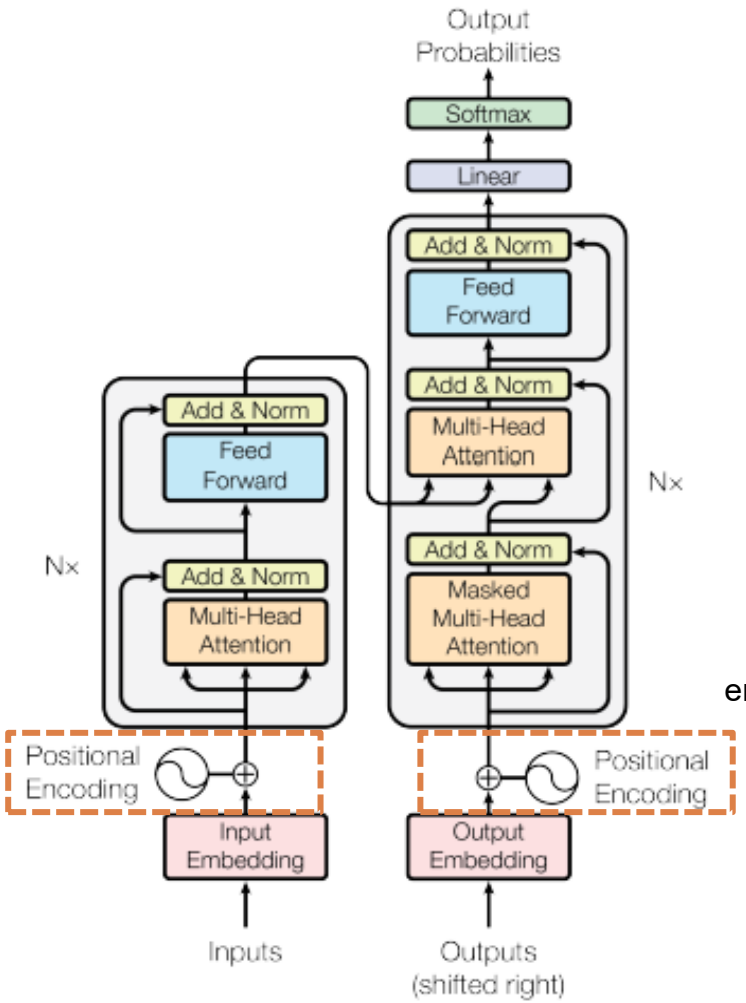
Transformer- Pointwise Feed-forward NN







Transformer – Positional Encoding



E_3 E_2 E_5 E_8 → No information of word order

I am a student

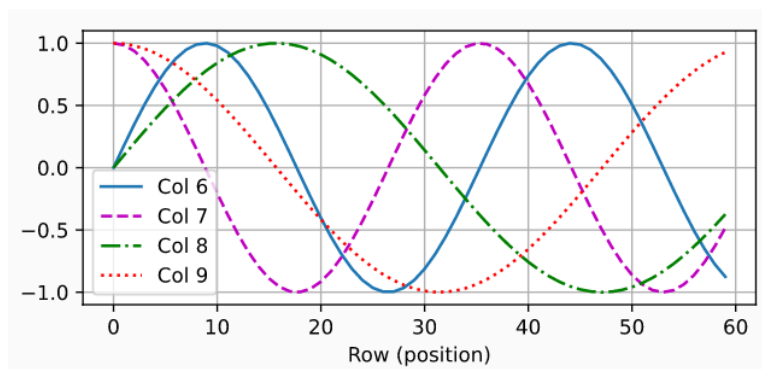
All words or **word embeddings** are **inputted simultaneously** to the Transformer

- Use to capture **position** of a word in a sentence
  - Transformer has **no recurrence** and **no convolution**.
  - For the model to make use of the **order of the sequence**, some information about the **relative or absolute position** of the tokens in the sequence **needs to be injected**

$$PE_{(pos,2i)} = \sin(pos/1000^{2i/embed\_size})$$
$$PE_{(pos,2i+1)} = \cos(pos/1000^{2i/embed\_size})$$

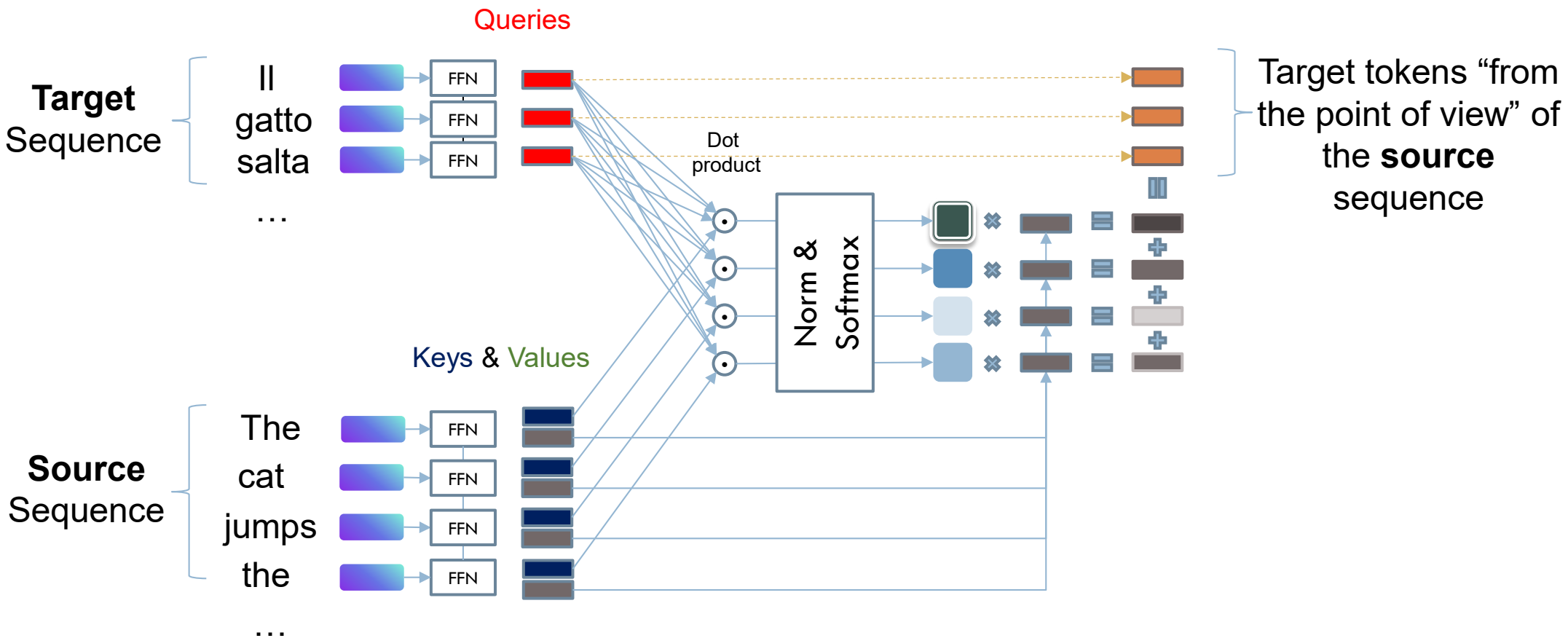
embed_size=3

$PE_{(1,1)}$	$PE_{(1,2)}$	$PE_{(1,3)}$	$PE_{(2,1)}$	$PE_{(2,2)}$	$PE_{(2,3)}$	$PE_{(3,1)}$	$PE_{(3,2)}$	$PE_{(3,3)}$	$PE_{(4,1)}$	$PE_{(4,2)}$	$PE_{(4,3)}$
E_{31}	E_{32}	E_{33}	E_{21}	E_{22}	E_{23}	E_{51}	E_{52}	E_{53}	E_{81}	E_{82}	E_{83}
pos=1	I		pos=2	am		pos=3	a		pos=4	student	
index=3			index=2			index=5			index=8		



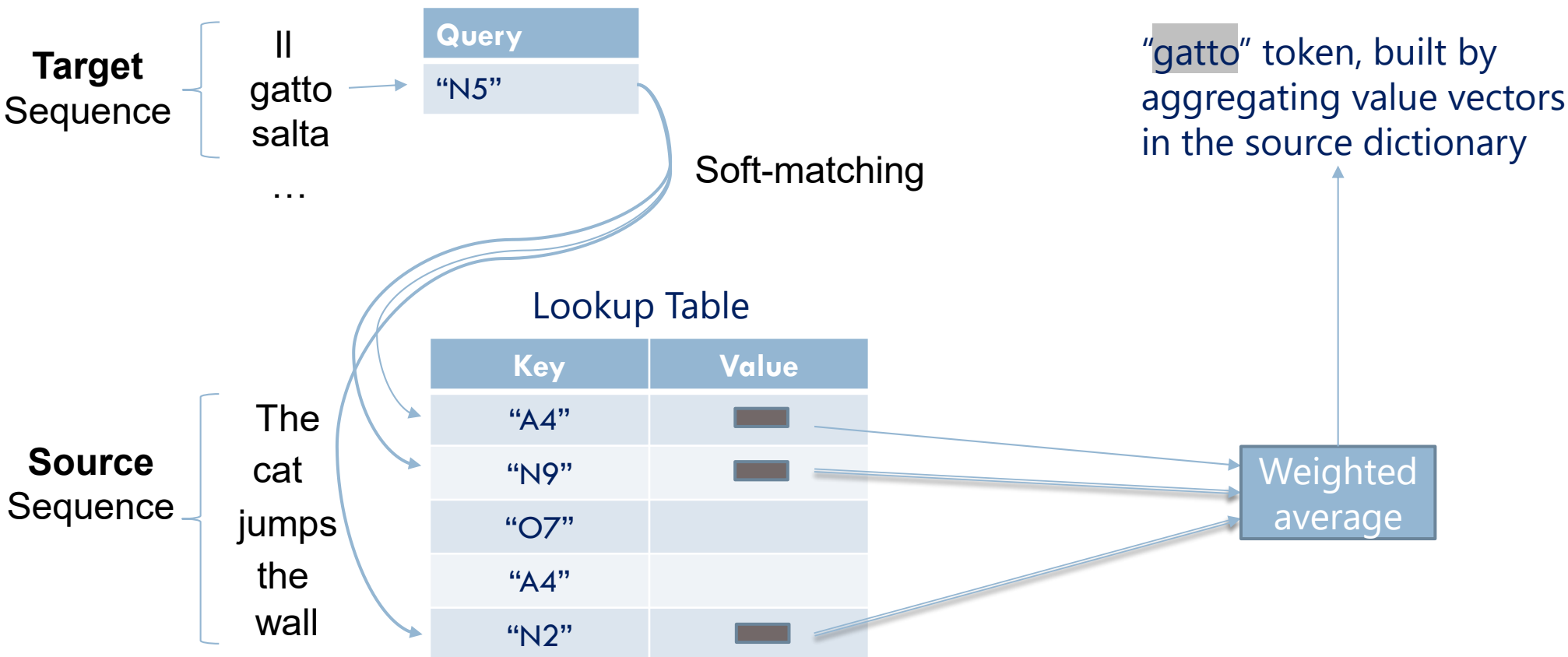
[Source: Dive into DL]

Transformer's Attention Mechanism



Transformer's Attention Mechanism

From a different perspective



Vision Transformers (ViT)

Transformers in Computer Vision

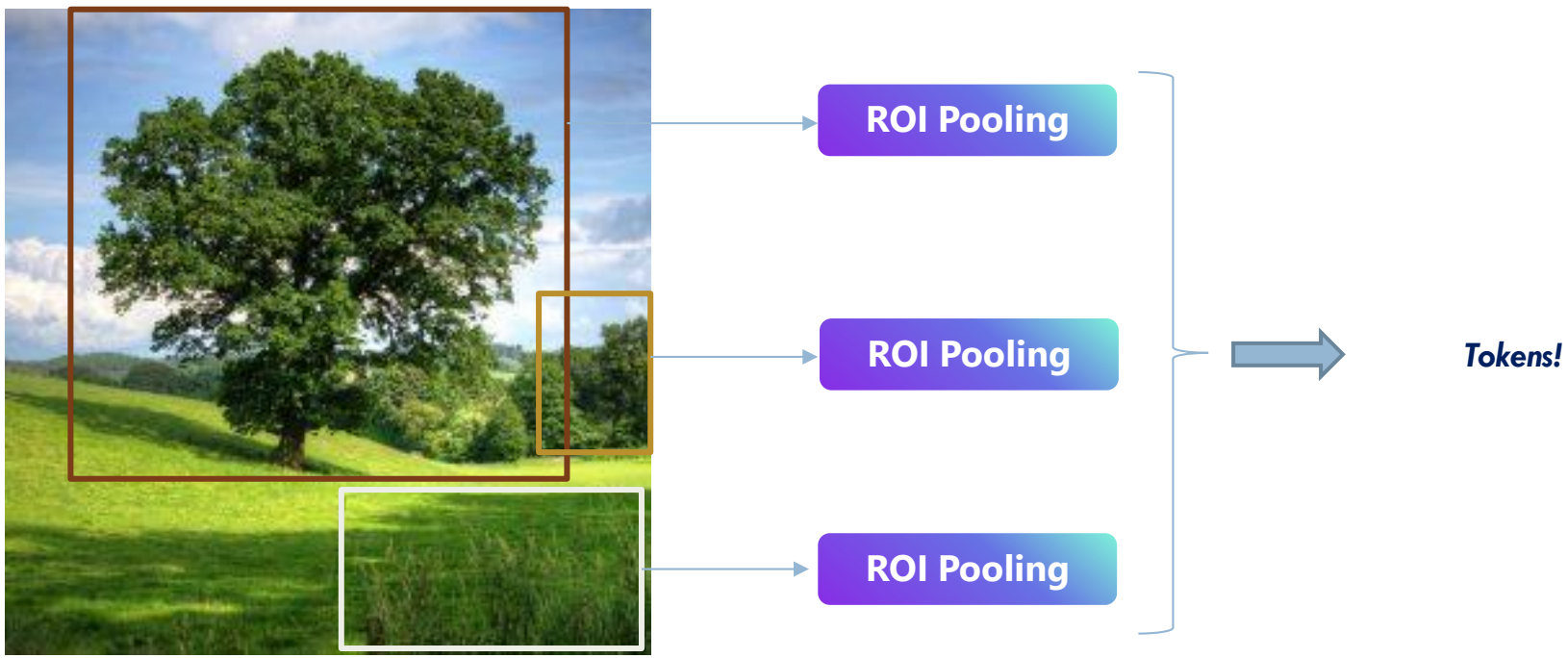
Can we use the self-attention mechanism in images?

Transformers in Computer Vision

- The transformer works with a set of tokens
- What are **tokens/visual words** in images**?**

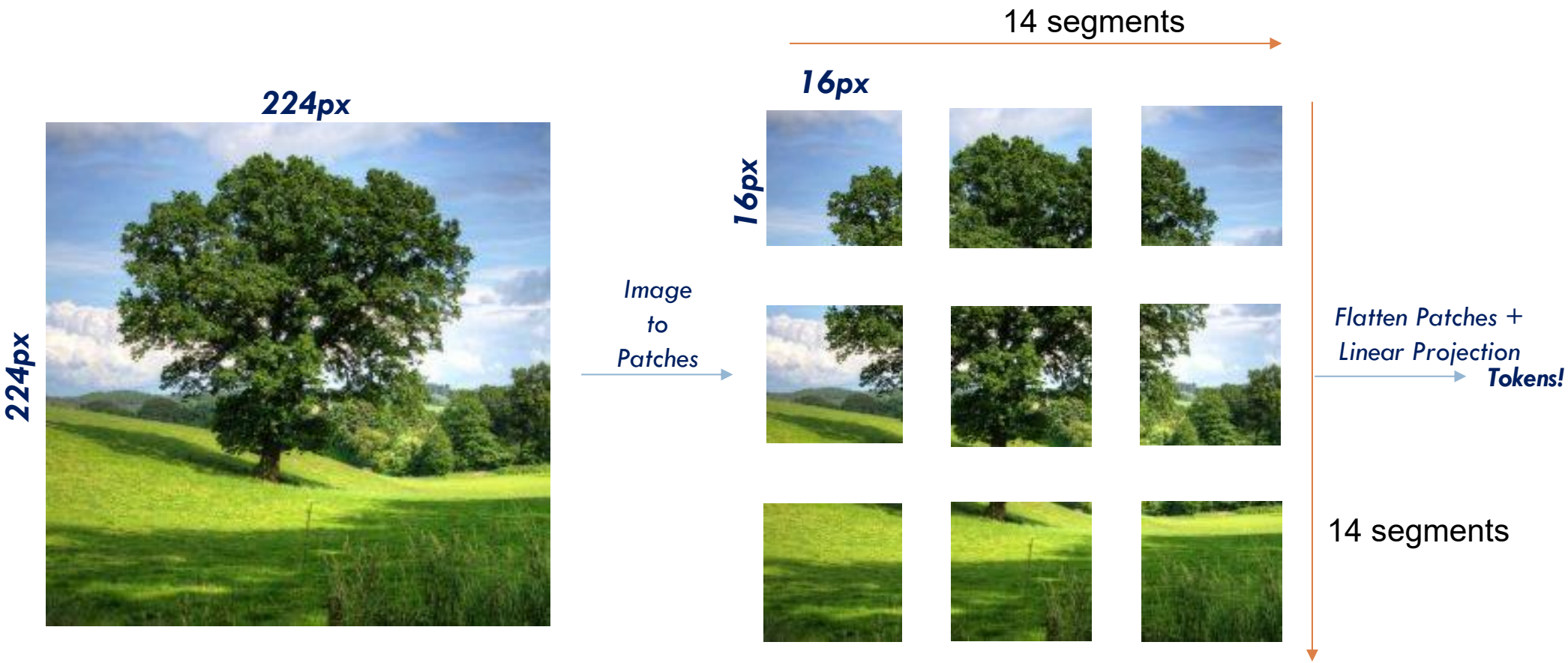
Transformers in Computer Vision

- Tokens as the features from an object detector



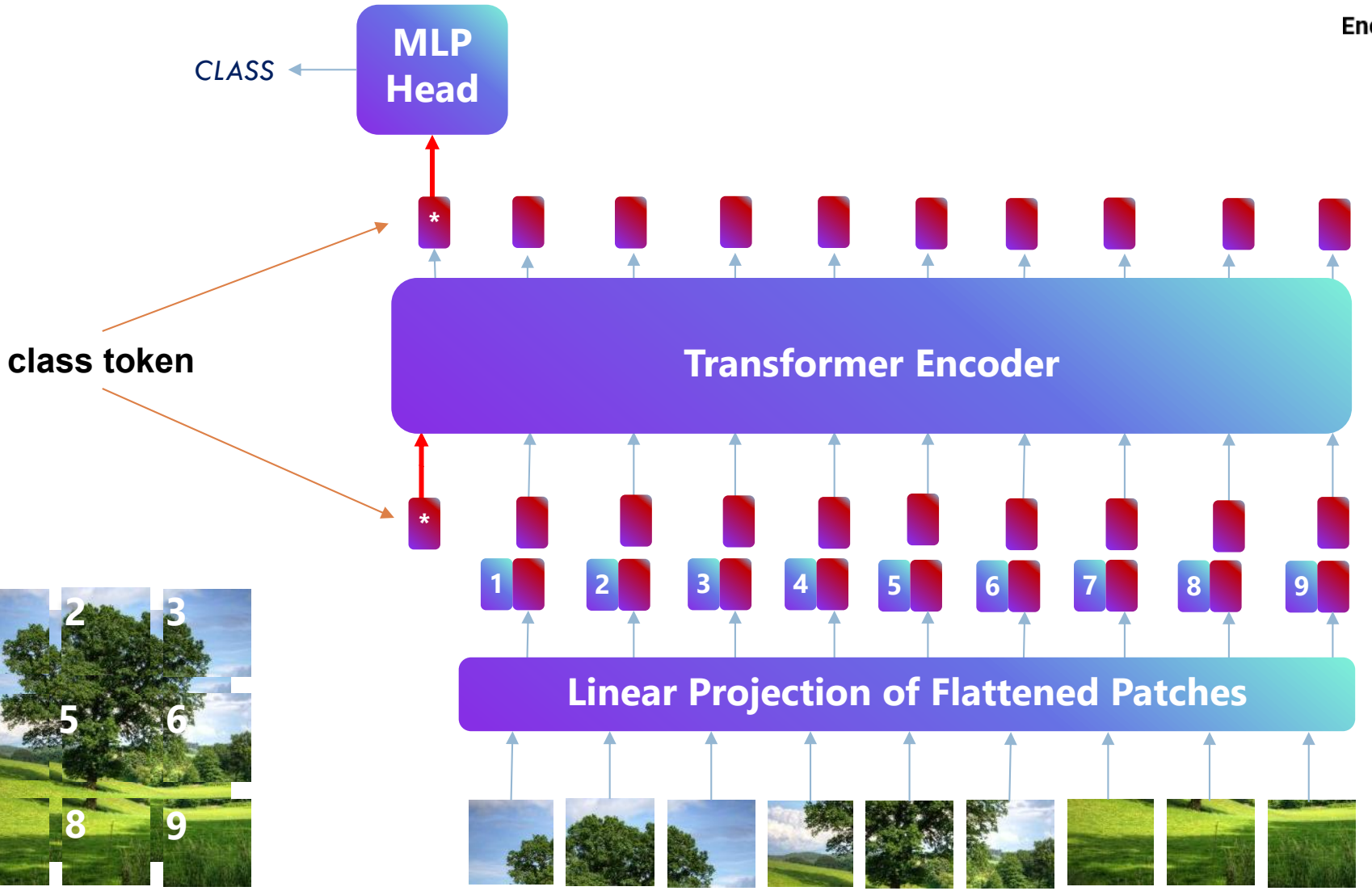
Vision Transformers (ViTs)

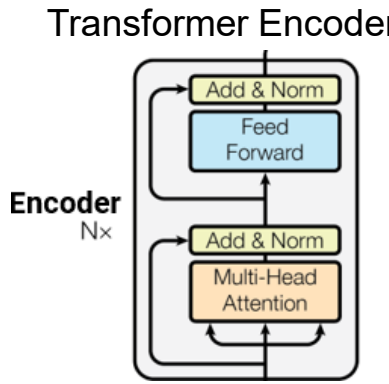
"An image is worth 16x16 words"



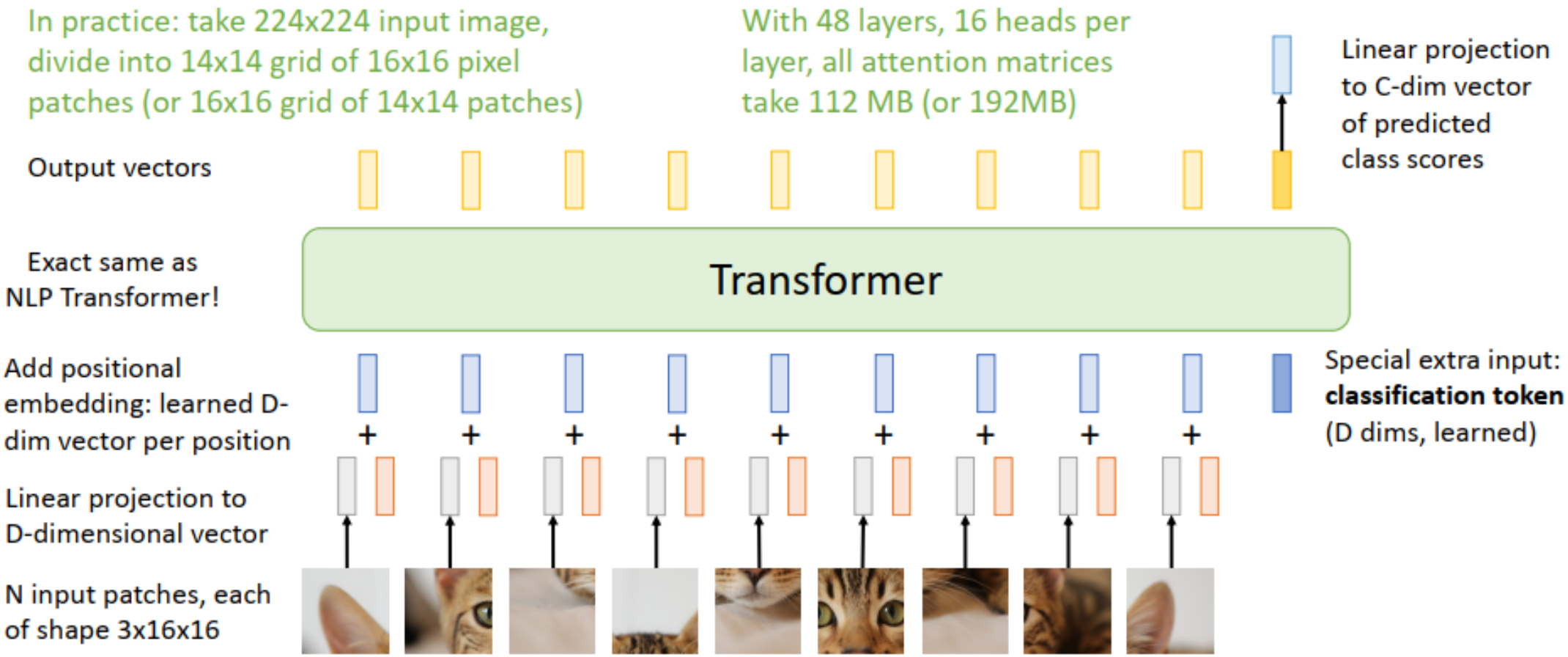
"An image is worth 16x16 words" | *Dosovitskiy et al., 2020*

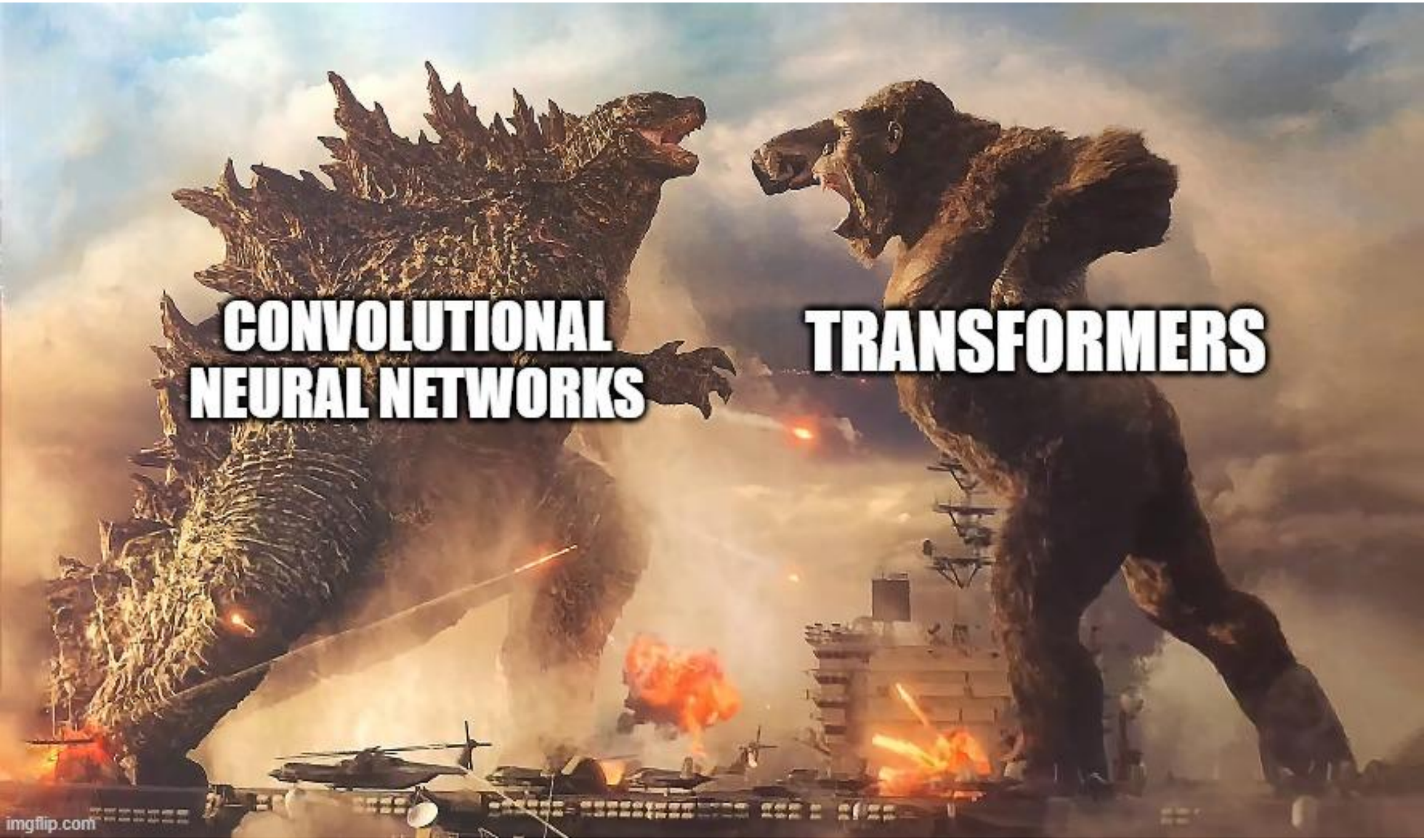
Vision Transformers (ViTs)



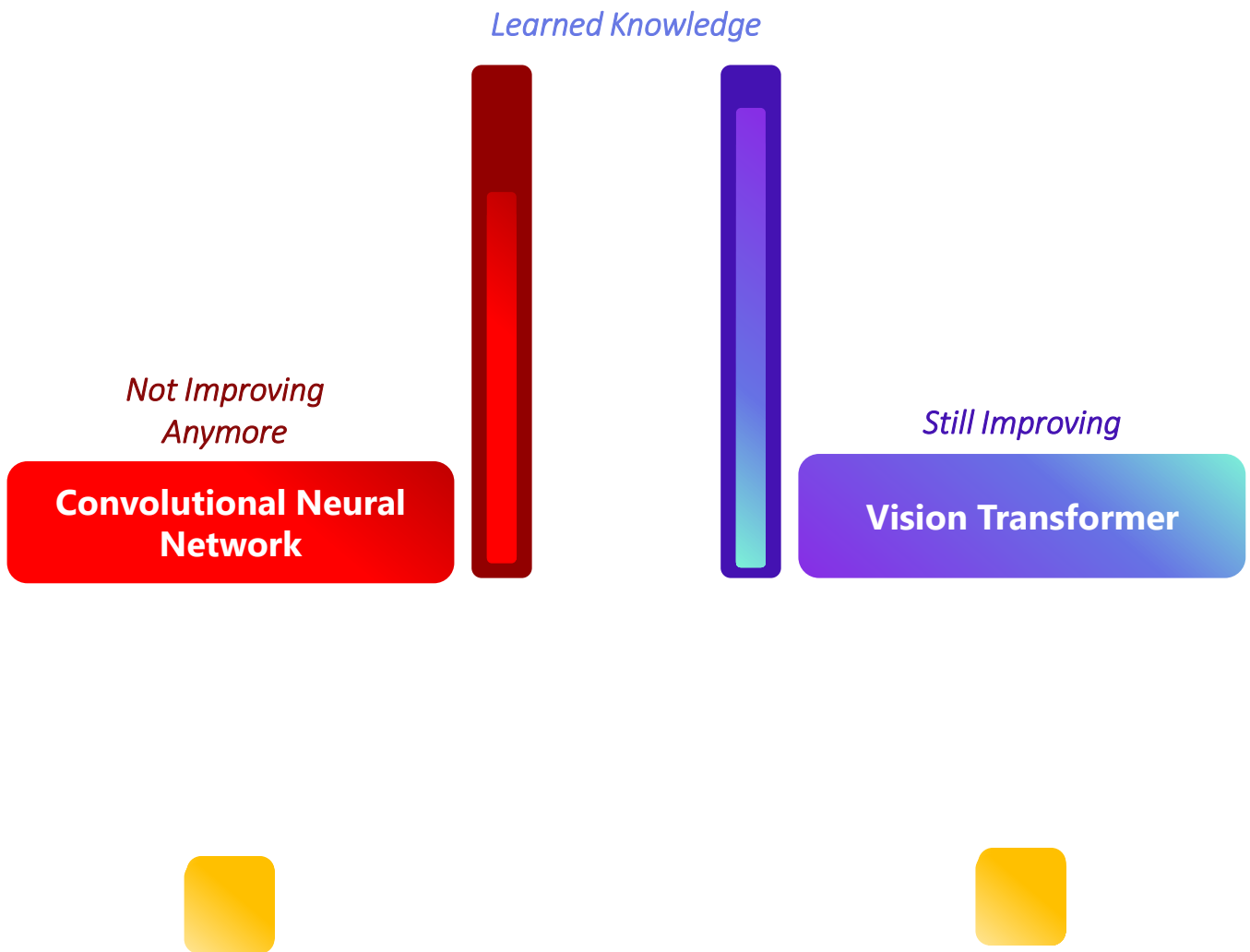


Vision Transformers (ViTs)

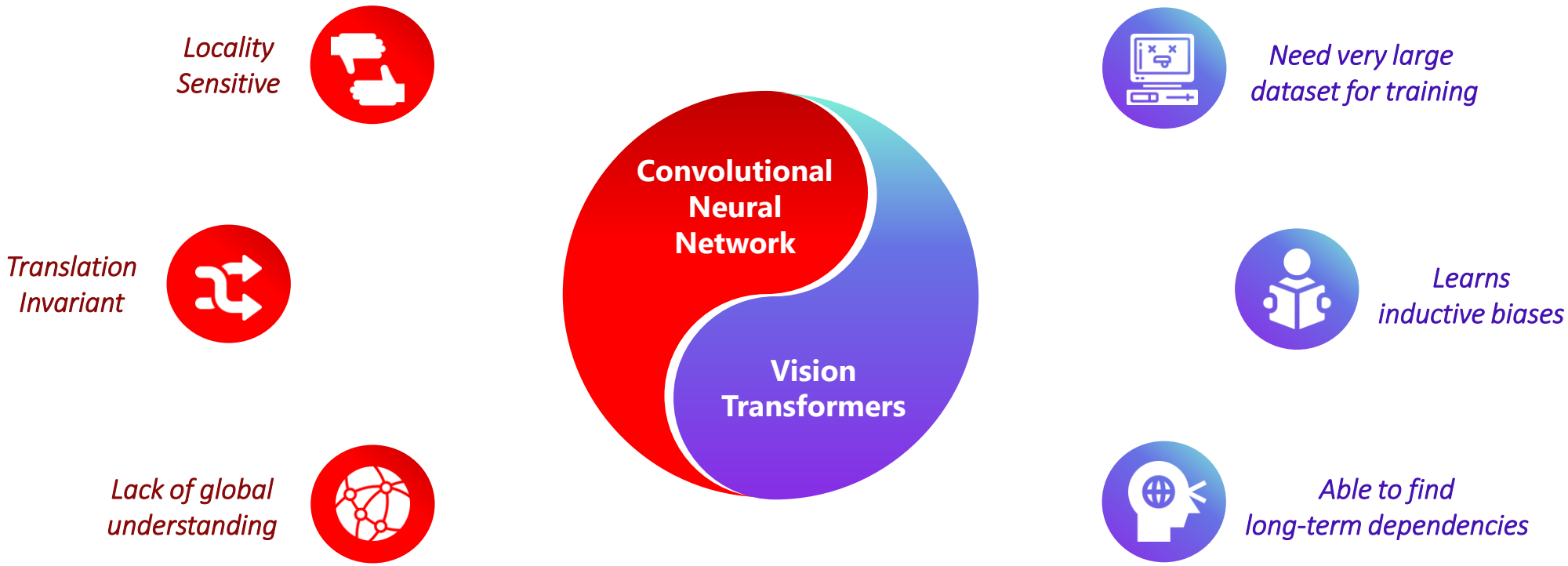



CONVOLUTIONAL
NEURAL NETWORKS
TRANSFORMERS
imgflip.com

What happens during training?

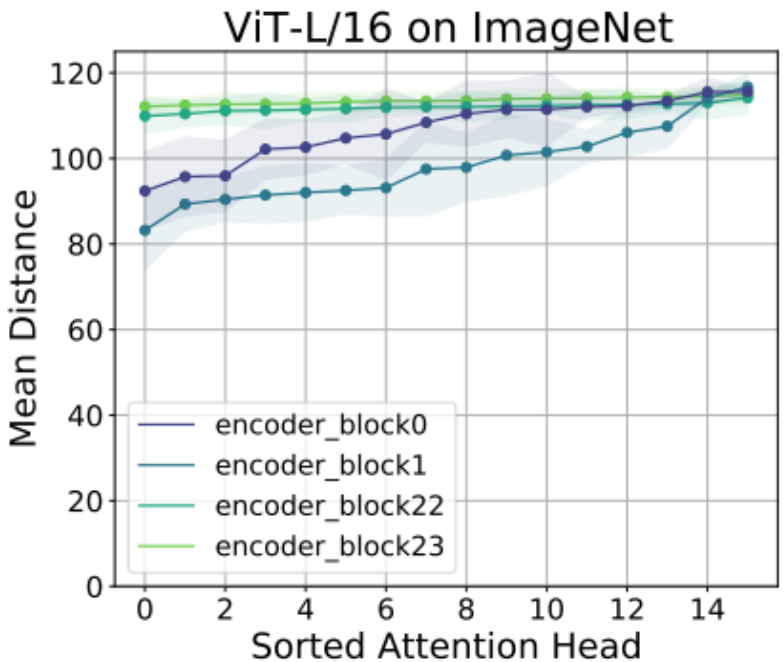


Why are they different?



A different point of view

ViTs are both local and global!

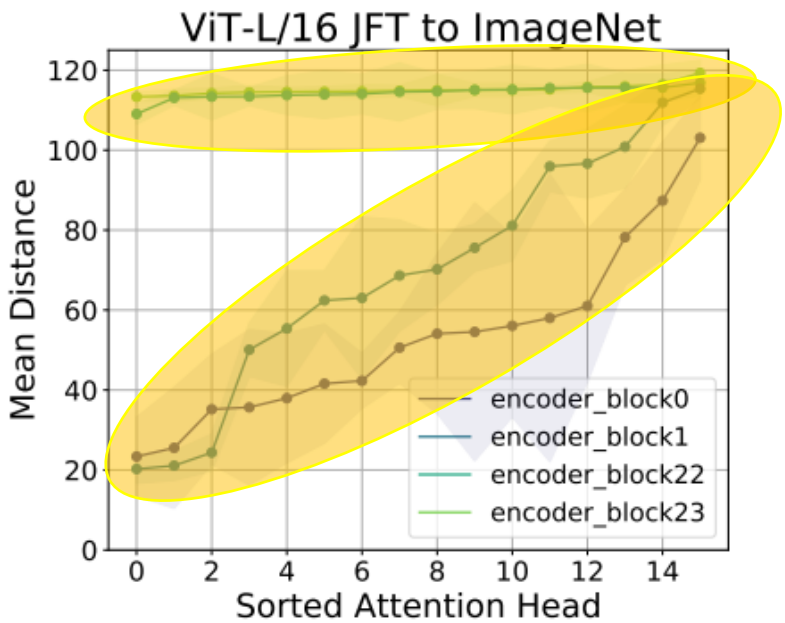


The ViT learns only global information with low amount of data

Heads focus on farther patches

A different point of view

ViTs are both local and global!



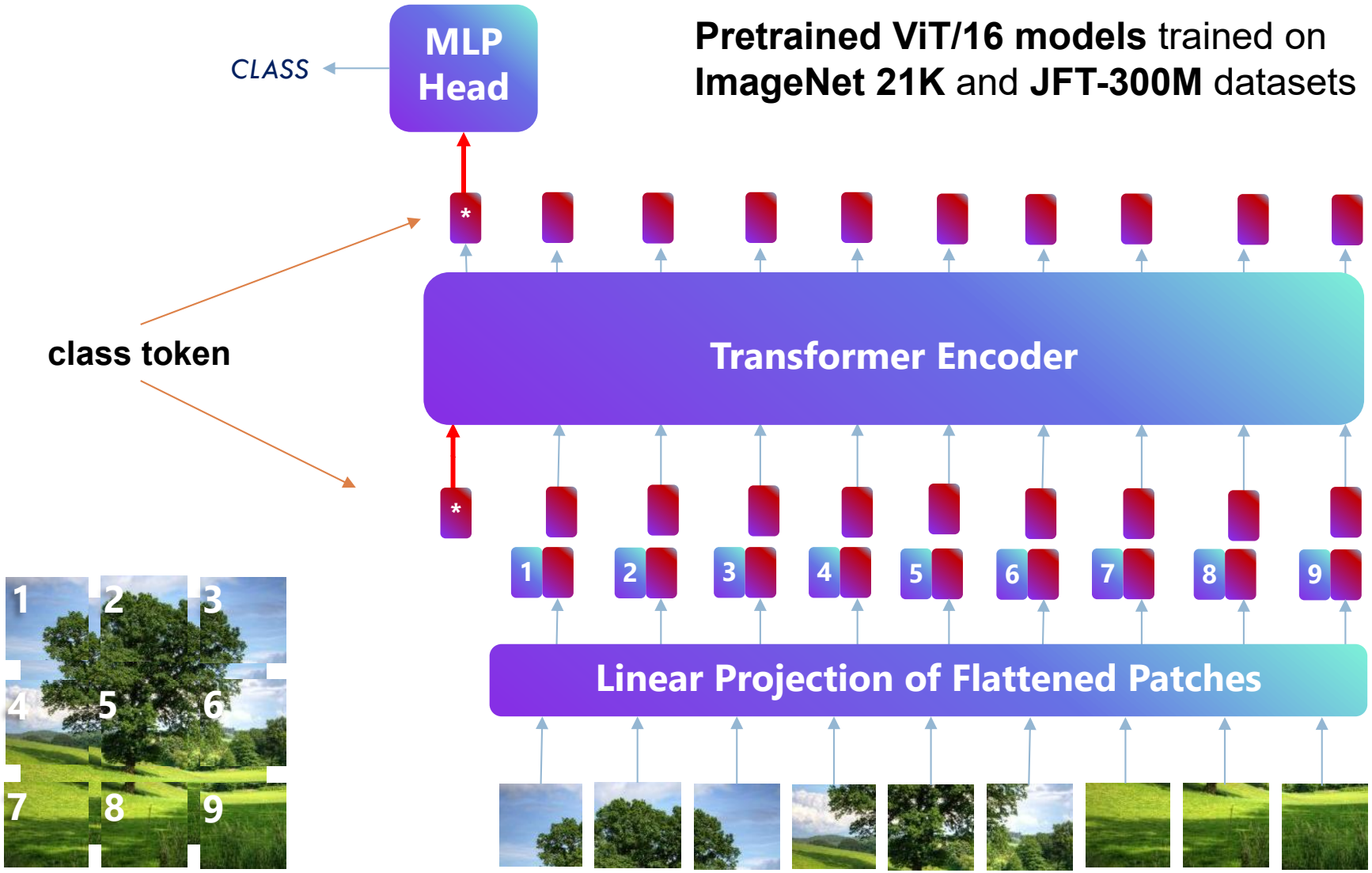
The ViT learns also local information with more data

Higher layers heads still focus on farther patches

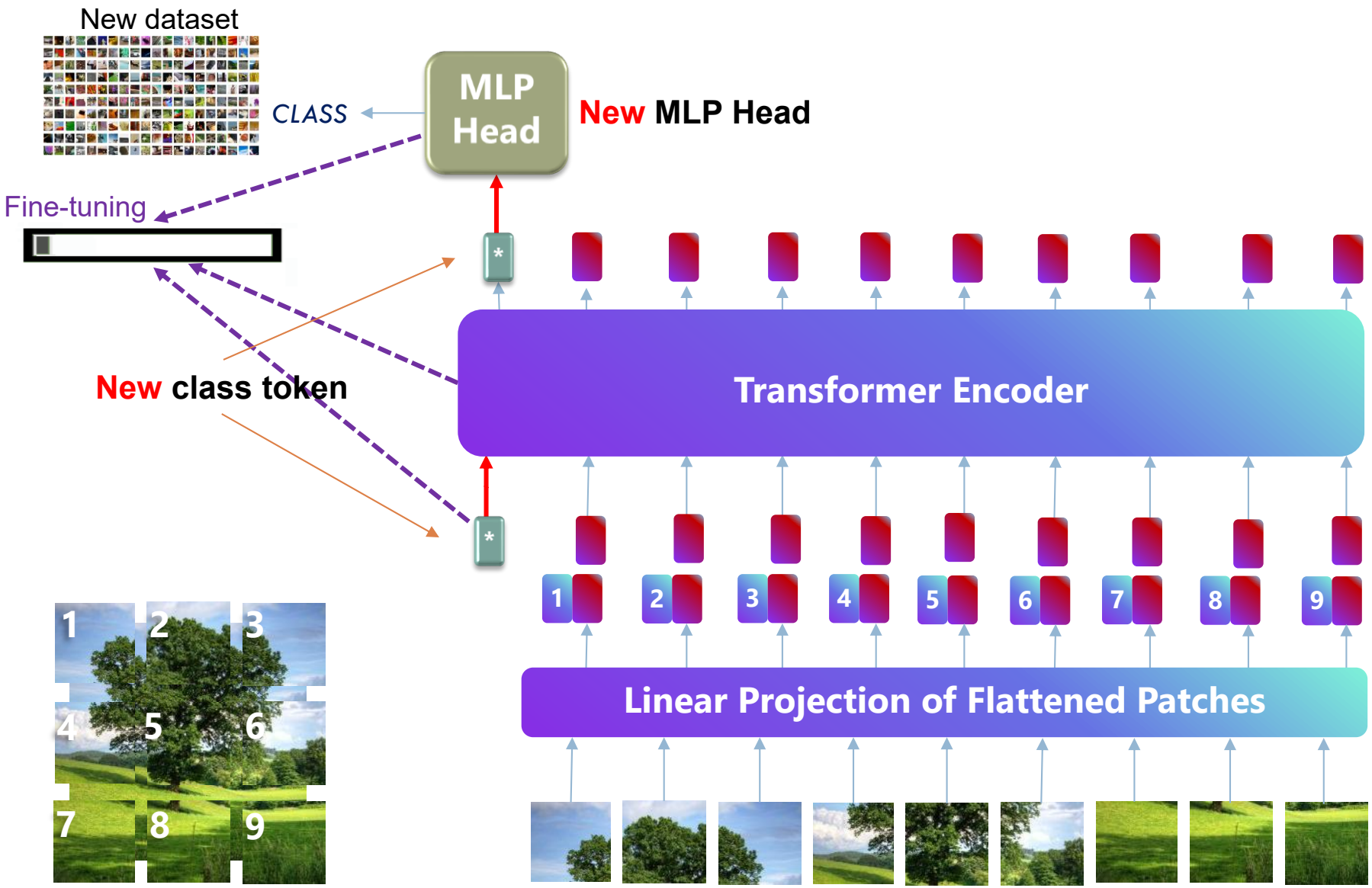
Lower layers heads focus on both farther and closer patches

Model Fine-Tuning

ViT: Model Fine-Tuning



ViT: Model Fine-Tuning



ViT: Model Fine-Tuning with Additional Components

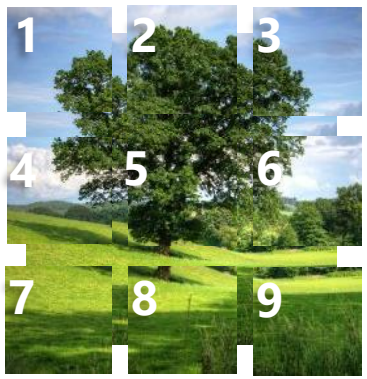
New dataset

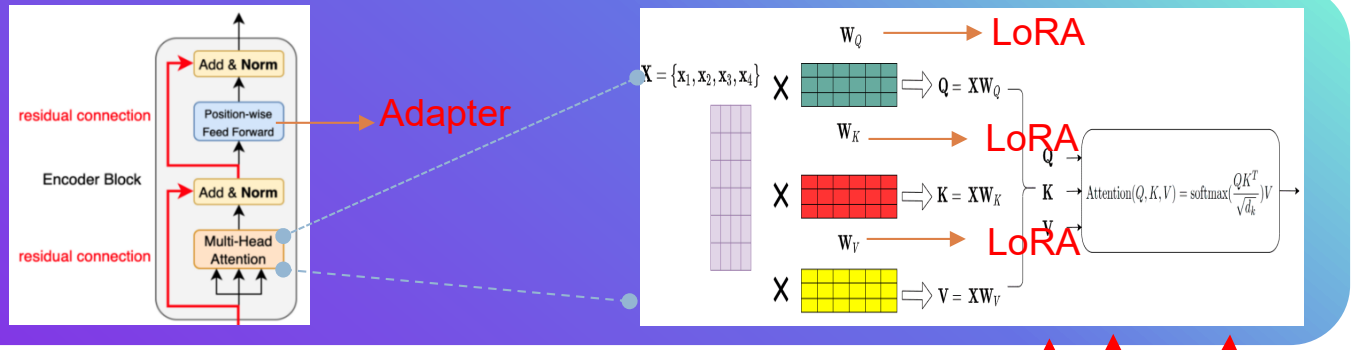
CLASS

New MLP Head

Principle: *insert to a pre-trained ViT **some additional components** that favour the computations of ViT and **fine-tune them**.*

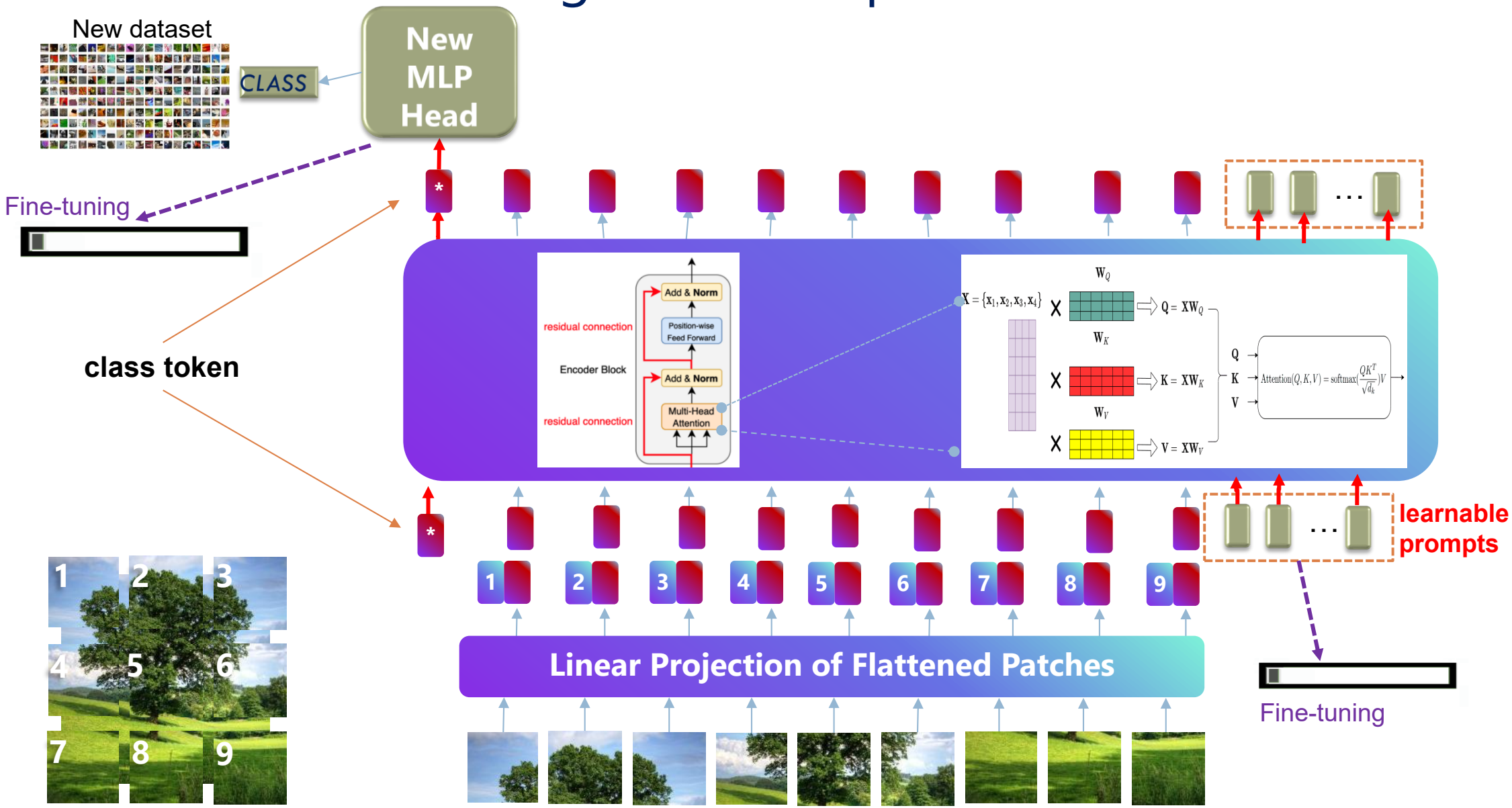
class token



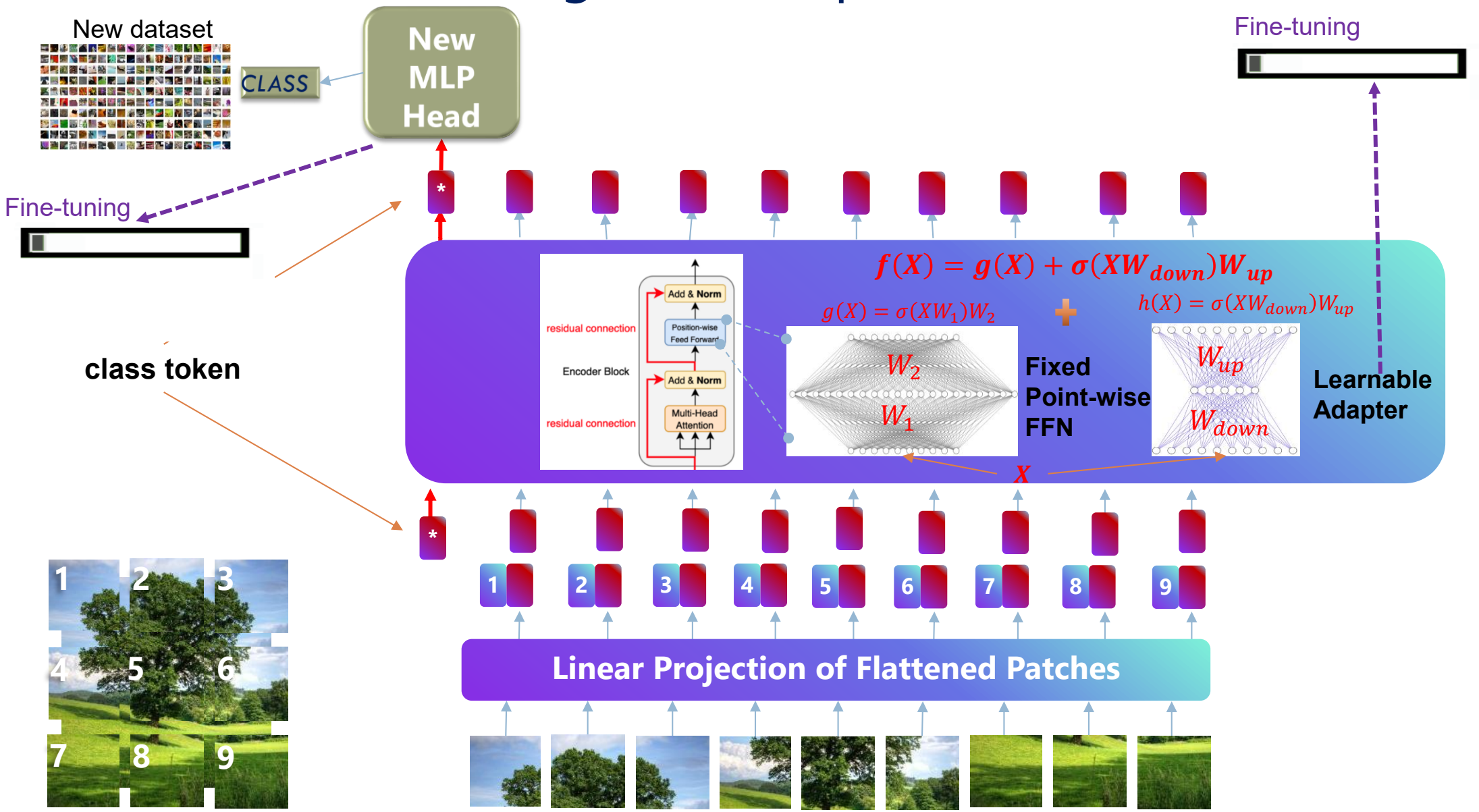


learnable prompts

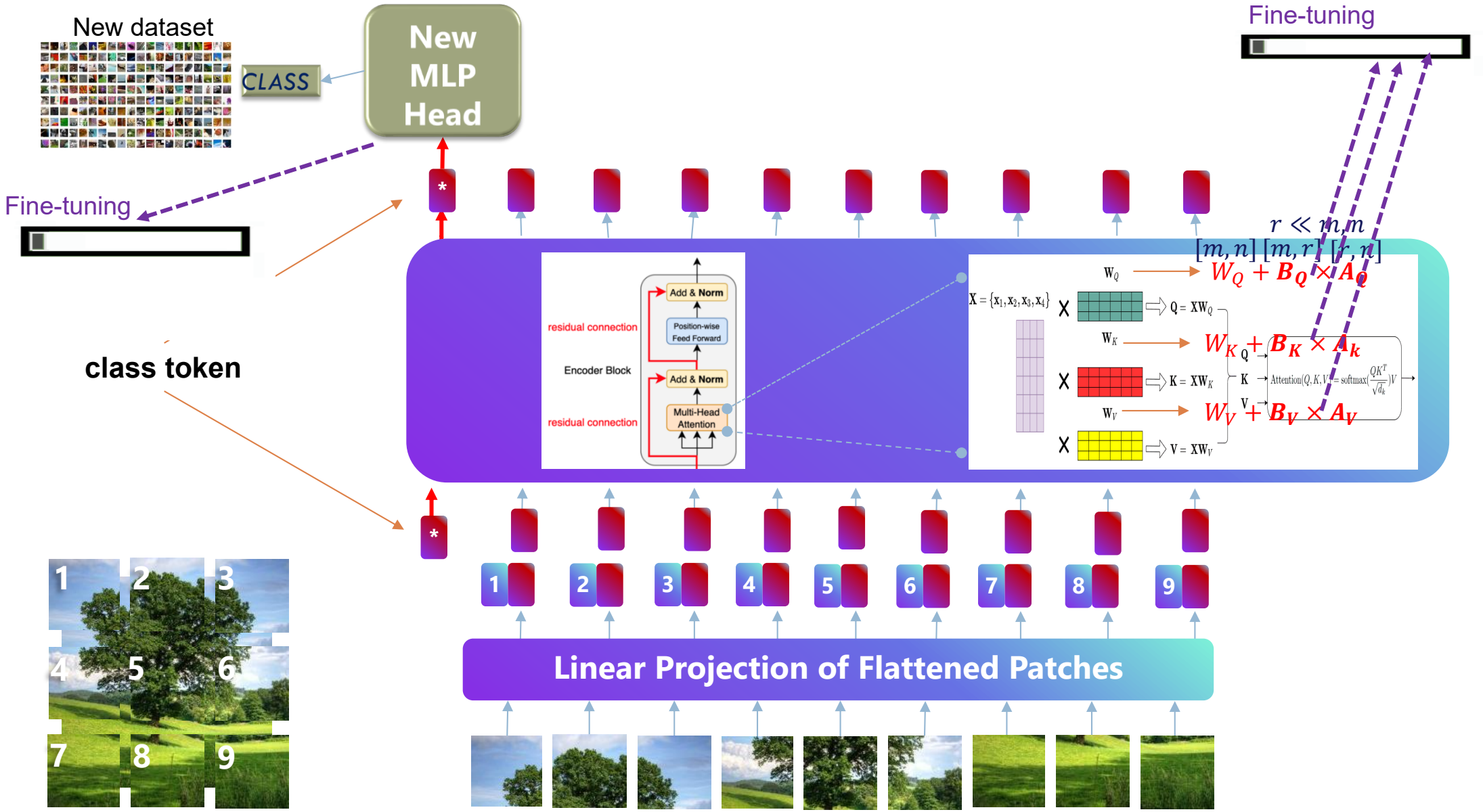
ViT: Model Fine-Tuning with Prompts



ViT: Model Fine-Tuning with Adapters



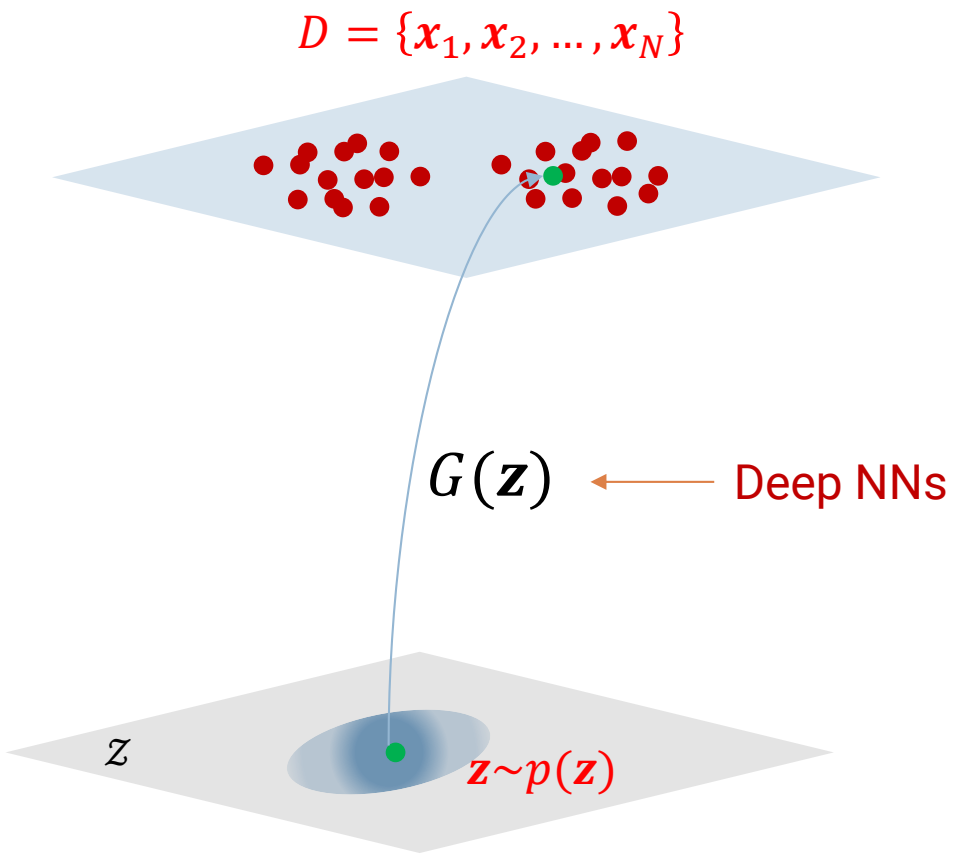
ViT: Model Fine-Tuning with Additional LoRA



Generative Adversarial Networs

General formulation for a Deep Generative Models (DGM)

- Given a training set $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ where each $\boldsymbol{x}_i \sim p_d(\boldsymbol{x})$.
  - $p_d(\boldsymbol{x})$ exists but unknown.
- Learn a **generator** $\boldsymbol{G}$ mapping from the 'noise'/latent space $\mathcal{Z}$ to data space
  - $\boldsymbol{z} \sim p(\boldsymbol{z}) \rightarrow \tilde{\boldsymbol{x}} = G(\boldsymbol{z}) \sim p_d(\boldsymbol{x})$
  - $\tilde{\boldsymbol{x}} = G(\boldsymbol{z})$ looks 'similar' to those in D.
- Use $\boldsymbol{G}$ to generate novel and new data samples.



Current DGM methods

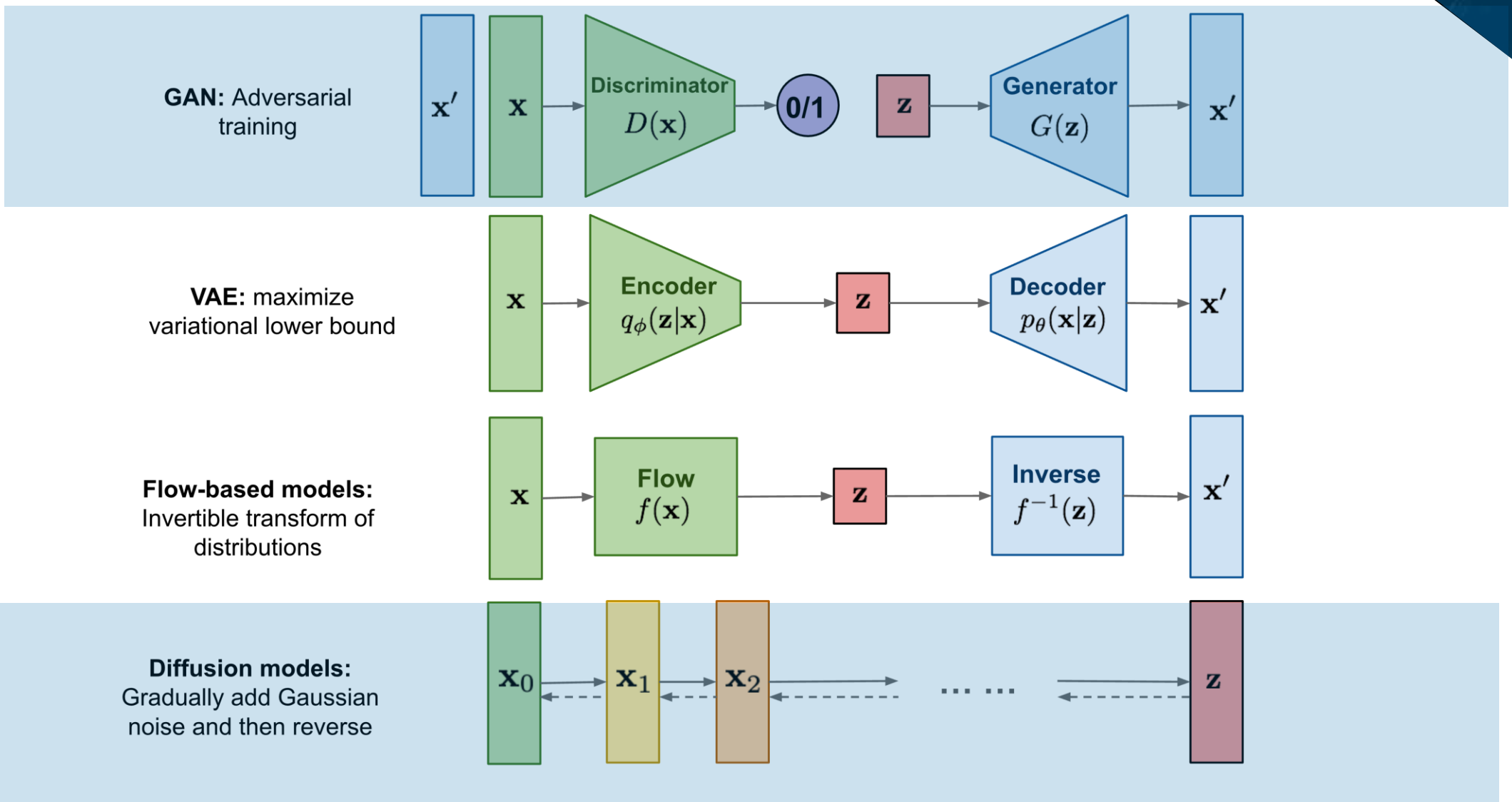


Image credit: https://lilianweng.github.io/posts/2021-07-11-diffusion-models/

Generative Adversarial Networks

Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie,* Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair,† Aaron Courville, Yoshua Bengio‡
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

NeurIPS 2014

Ian Goodfellow

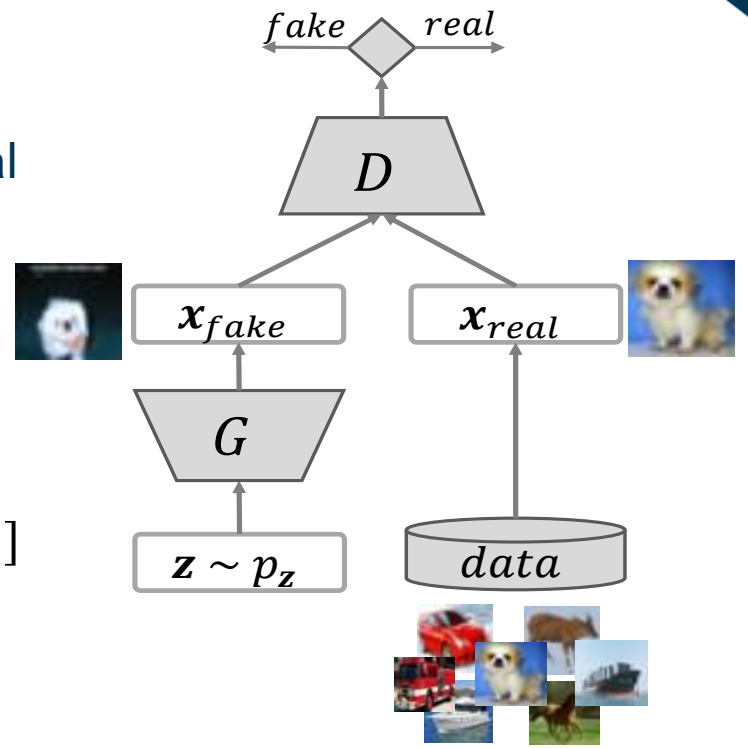
Born	1987[1]
Nationality	American
Alma mater	Stanford University Université de Montréal
Known for	Generative adversarial networks, Adversarial examples
	Scientific career
Fields	Computer science
Institutions	Apple Inc. Google Brain OpenAI DeepMind Google DeepMind
Thesis	*Deep Learning of Representations and its Application to Computer Vision* (2014)
Doctoral advisor	Yoshua Bengio Aaron Courville
Website	www.iangoodfellow.com

> The most important one, in my opinion, is adversarial training (also called GAN for Generative Adversarial Networks). This is an idea that was originally proposed by Ian Goodfellow when he was a student with Yoshua Bengio at the University of Montreal (he since moved to Google Brain and recently to OpenAI).
>
> This, and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion.

Yann LeCun

Generative Adversarial Networks

- A game of two players [Goodfellows et al., 2014]
  - **Generator G**: generate fake samples indistinguishable from real samples
  - **Discriminator D**: discriminate real and fake samples



- Min-max problem

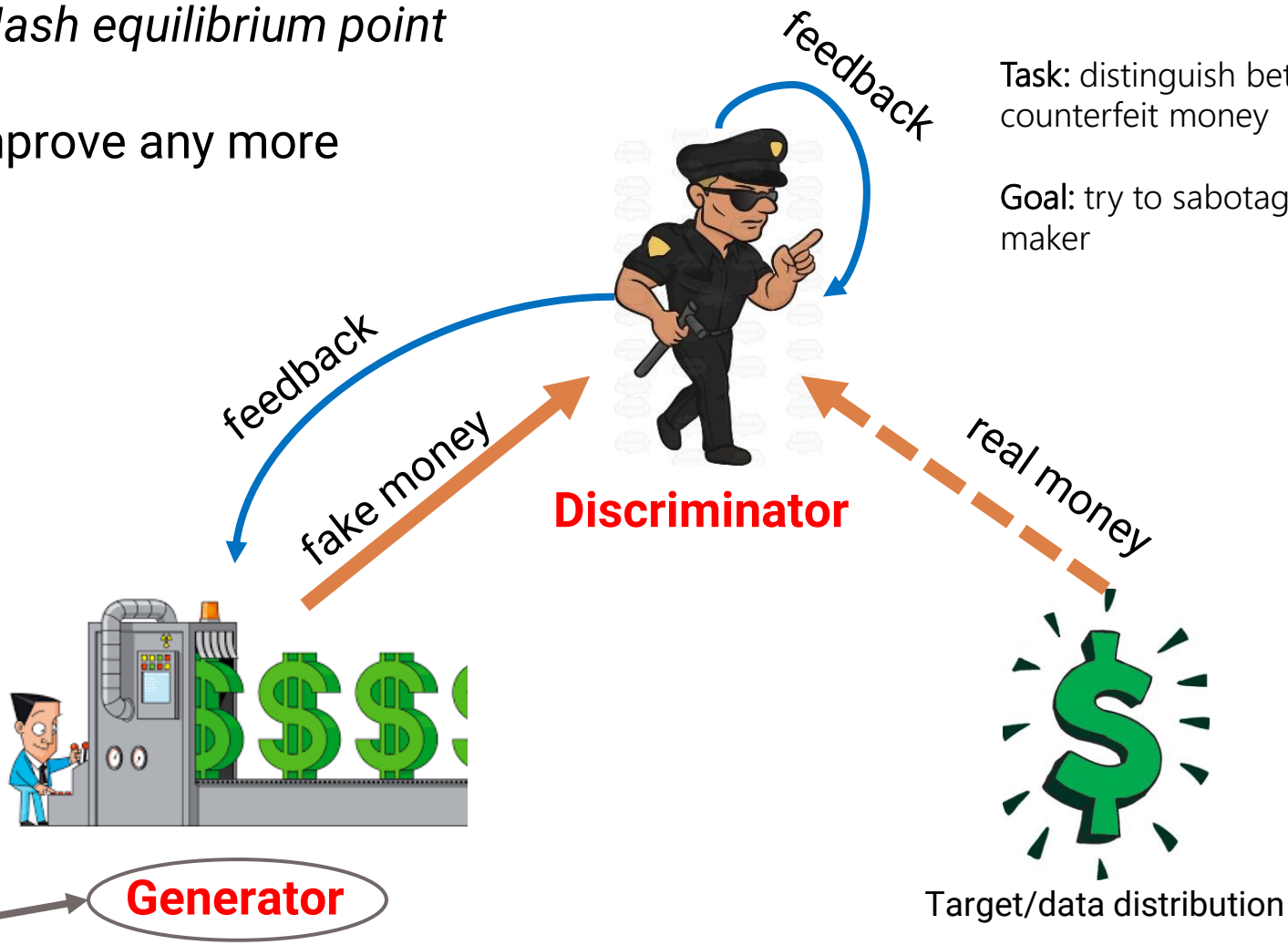
$$\min_G \max_D J(G,D) = \mathbb{E}_{x \sim p_d(x)}[\log D(x)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D(G(z))]$$

- Issue & challenge
  - Mode collapse, min-max problem

GAN: analogy to minimax game

Termination at *Nash equilibrium point*

No player can improve any more



Task: distinguish between real and counterfeit money

Goal: try to sabotage the counterfeit maker

Task: produce counterfeit money

Goal: try to fool/confuse the police officer

Adversarial or Cooperative

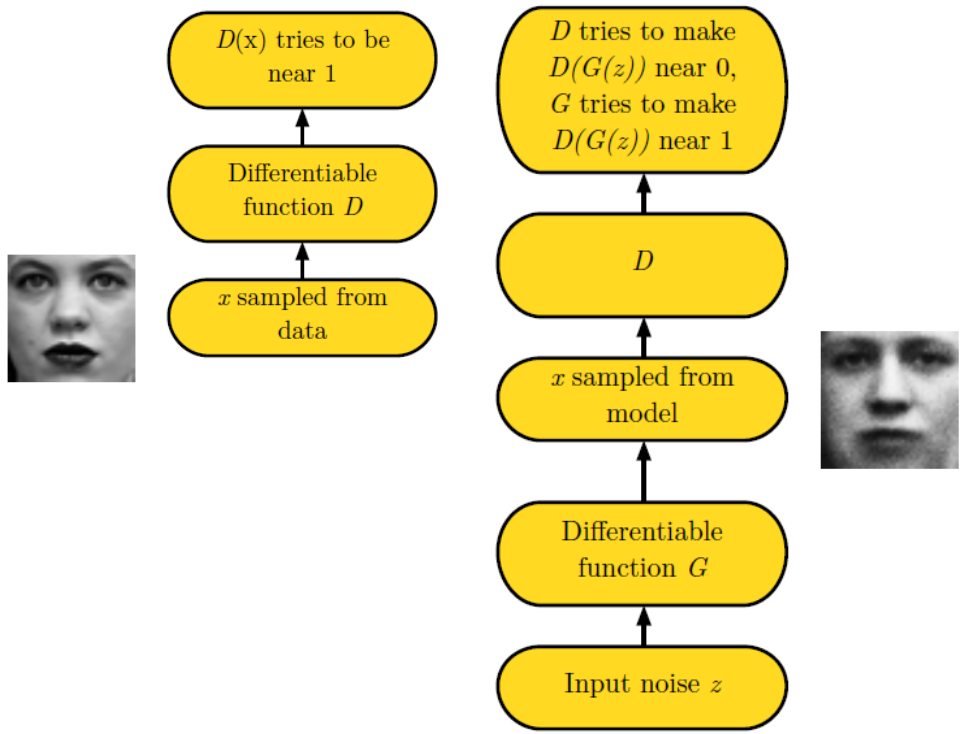
Generative Adversarial Networks (GAN)

How to formulate this?

- Introduce a noise variable $\boldsymbol{z} \sim p_{\boldsymbol{z}}$
  - This is the prior distribution; think of it as the resource used to make money.
- A generator G that maps $\boldsymbol{z}$ to $\tilde{\boldsymbol{x}}$ in data space:

$$\tilde{\boldsymbol{x}} = G(\boldsymbol{z})$$

  - Parameterize G by a deep neural network with parameter $\boldsymbol{\theta}_g$.
- A discriminator $D(\boldsymbol{x})$ maps $\boldsymbol{x}$ to a probability in $[0, 1]$ representing the probability that $\boldsymbol{x}$ comes from the real data rather than G.
  - Parameterize $D(\boldsymbol{x})$ by another deep neural network $\boldsymbol{\theta}_d$.
  - $D(x)$ close to 1 (real)
  - $D(x)$ close to 0 (fake)



$$\min_G \max_D J(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_d(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}))]$$

(Figure credit: from Goodfellow, NIPS workshop'16)

Training GAN

- Gradient ascent to train discriminator D_{θ_d}:

$$\max_{\theta_d} \mathop{\mathbb{E}}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}\left[\log D_{\theta_d}(\boldsymbol{x})\right] + \mathop{\mathbb{E}}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\left[\log(1 - D_{\theta_d}(G(\boldsymbol{z}))\right]$$

Training D is to maximize the probability of detecting correct labels

Ascent update D: $\boldsymbol{z}^{(i)}\sim p_z, \boldsymbol{x}^{(i)}\sim p_{data}$

$$\nabla_{\boldsymbol{\theta}_\mathrm{d}} \frac{1}{\mathrm{M}} \sum_{i=1}^{M} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$$

$$\min_{G}\max_{\mathrm{D}} J(G, D) = \mathop{\mathbb{E}}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}\left[\log D(\boldsymbol{x})\right] + \mathop{\mathbb{E}}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\left[\log(1 - D(G(\boldsymbol{z}))\right]$$

Training GAN

$$\min_{G} \max_{\mathrm{D}} J(G, D) = \mathop{\mathbb{E}}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})} [\log D(\boldsymbol{x})] + \mathop{\mathbb{E}}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}} [\log(1 - D(G(\boldsymbol{z}))]$$

Training G is to minimize or fool the discriminator D, i.e., minimize $(1 - D(G(\boldsymbol{z})))$ to 0

- Gradient descent to train the generator G_{θ_g}

$$\min_{\theta_g} \mathop{\mathbb{E}}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}} [\log(1 - D_{\theta_d}(G(\boldsymbol{z}))]$$

Descent update G: $\boldsymbol{z}^{(i)} \sim p_z$

$$-\nabla_{\boldsymbol{\theta}_{\mathrm{g}}} \frac{1}{\mathrm{M}} \sum_{i=1}^{M} \left[\log \left(1 - D \left(G \left(\boldsymbol{z}^{(i)} \right) \right) \right) \right]$$

Training GAN

Training D is to maximize the probability of detecting correct labels

Ascent update D: $\boldsymbol{z}^{(i)} \sim p_z, \boldsymbol{x}^{(i)} \sim p_{data}$

$$\nabla_{\boldsymbol{\theta}_\mathrm{d}} \frac{1}{\mathrm{M}} \sum_{i=1}^{M} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

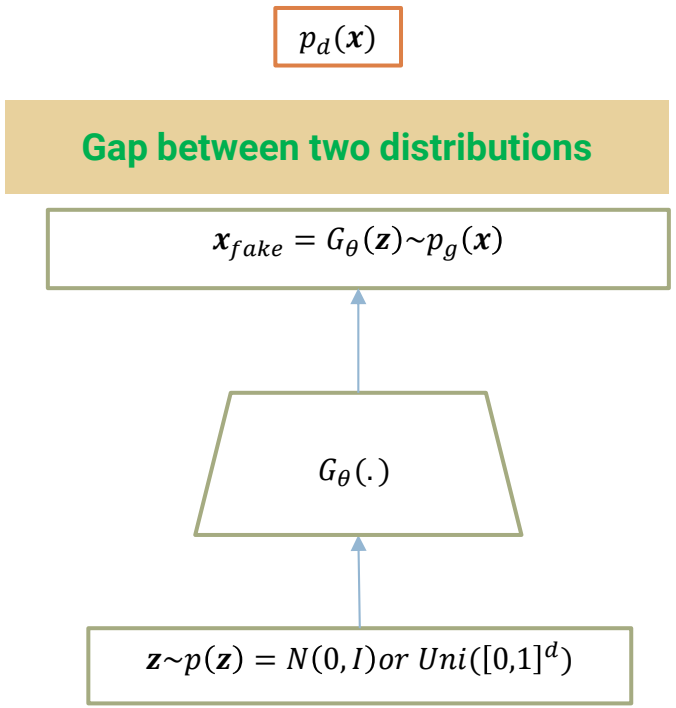
$$\min_{G} \max_{\mathrm{D}} J(G, D) = \mathop{\mathbb{E}}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})} [\log D(\boldsymbol{x})] + \mathop{\mathbb{E}}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}} [\log(1 - D(G(\boldsymbol{z}))]$$

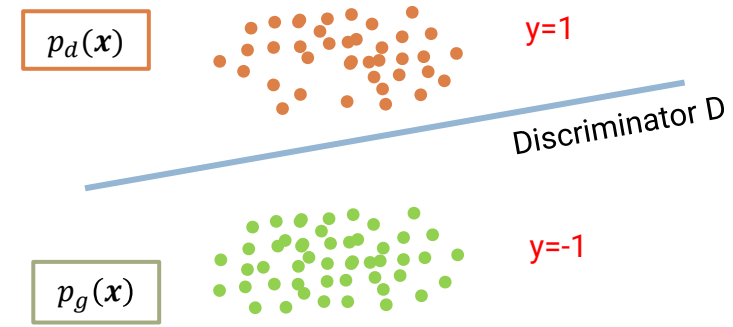
Training G is to minimize or fool the discriminator D, i.e., minimize $1 - D\left(G(\boldsymbol{z})\right)$ to 0

Descent update G: $\boldsymbol{z}^{(i)} \sim p_z$

$$-\nabla_{\boldsymbol{\theta}_\mathrm{g}} \frac{1}{\mathrm{M}} \sum_{i=1}^{M} \left[\log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

Basic theory of GAN





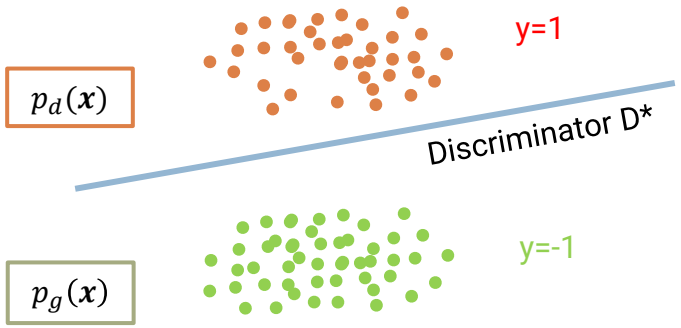
- **D(x) = P(y=1|x)**: probability **x** is true data.
- **1-D(x) = P(y=-1|x)**: probability x is fake data.

The **log likelihood function** need to be maximized

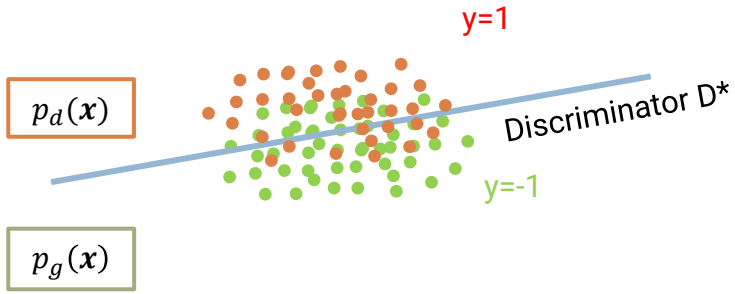
$$J(D,G) = \mathbb{E}_{x\sim P_d}[\log D(x)] + \mathbb{E}_{x\sim P_g}[\log[1 - D(x)]]$$
$$= \mathbb{E}_{x\sim P_d}[\log D(x)] + \mathbb{E}_{z\sim P_z}[\log[1 - D(G(z))]]$$

- We want to train **NN-based function** $G_\theta(.)$ such that $p_g(x) = p_d(x)$, but we do not have $p_d(x)$ at hand
  - Introduce a strong discriminator $D(x)$ to **implicitly justify** how far $p_d(x)$ and $p_g(x)$ are.

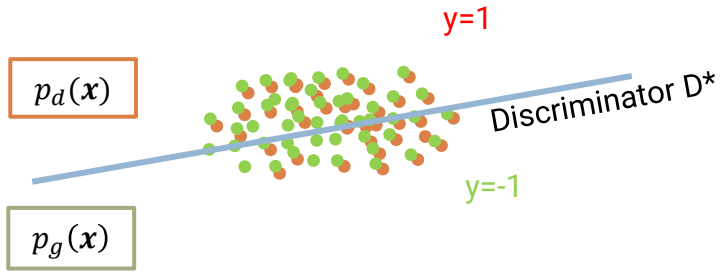
Basic theory of GAN



$J(D^*, G) = \mathbb{E}_{x \sim P_d}[\log D^*(x)] + \mathbb{E}_{x \sim P_g}[\log[1 - D^*(x)]]$ is large



$J(D^*, G) = \mathbb{E}_{x \sim P_d}[\log D^*(x)] + \mathbb{E}_{x \sim P_g}[\log[1 - D^*(x)]]$ decreases



$J(D^*, G) = \mathbb{E}_{x \sim P_d}[\log D^*(x)] + \mathbb{E}_{x \sim P_g}[\log[1 - D^*(x)]]$ is smallest

It comes with the **minimax** problem

$\boldsymbol{min_G max_D J(D, G)}$

Optimal discriminator solution

- Recall the GAN objective:

$$J(G,D) = \mathop{\mathbb{E}}_{x\sim p_{data}(x)}[\log D(x)] + \mathop{\mathbb{E}}_{z\sim p_z}[\log(1-D(G(z))]$$

- Let p_g be the distribution over $G(z)$ when $z\sim p_z$ and let p_d denotes p_{data}:

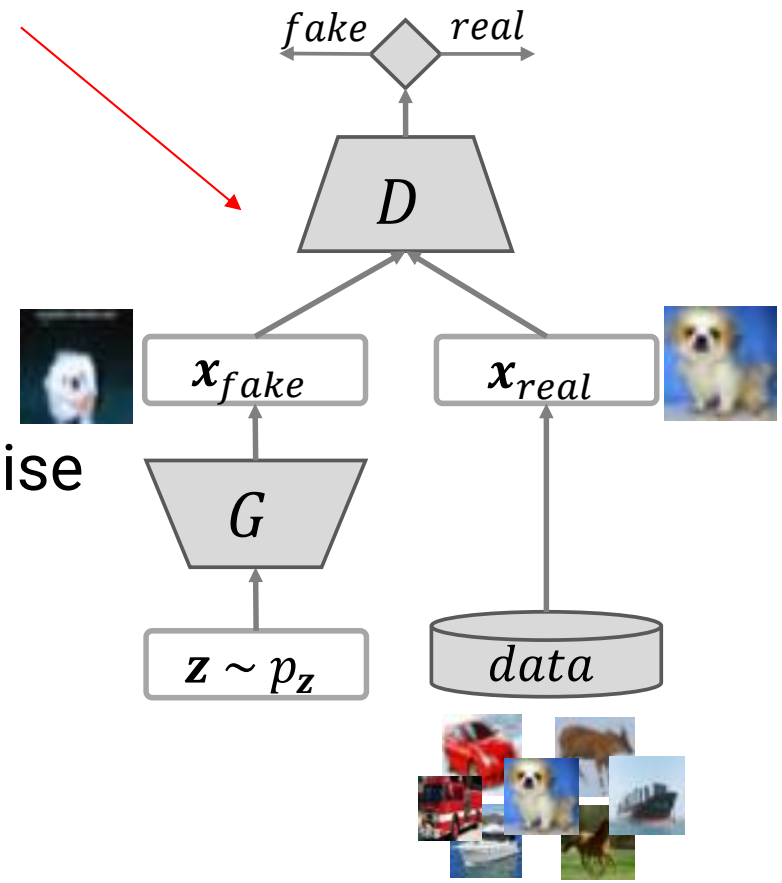
$$J(G,D) = \mathop{\mathbb{E}}_{x\sim p_d}[\log D(x)] + \mathop{\mathbb{E}}_{x\sim p_g}[\log(1-D(x)]$$
$$= \int_x p_d(x)\log D(x)dx + \int_x p_g(x)\log\big(1-D(x)\big)dx$$

- We need to maximize D, let's do this pointwise at each position of x:

$$\max_D\{\, p_d(x)\log D(x) + p_g(x)\log\big(1-D(x)\big)\}$$

  - $\nabla_D = \frac{p_d(x)}{D} - \frac{p_g(x)}{1-D} = 0$
  - Optimal discriminator: $D^*(x) = \frac{p_d(x)}{p_d(x)+p_g(x)}$
  - When G generates perfect samples, $p_g = p_d$ at which $D(x) = 0.5\ \forall x$

When training is perfect, discriminator D can NO longer tell the fake (money) from the real (money), i.e., 0.5 probability



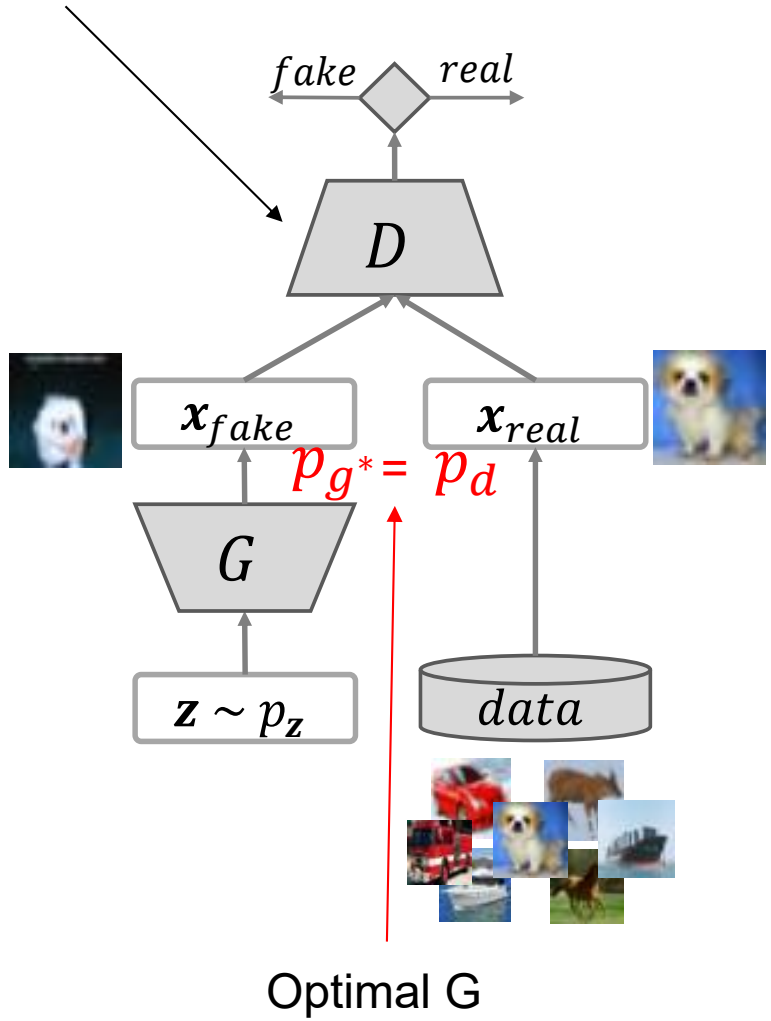
Optimal generator solution

When training is perfect, discriminator D can NO longer tell the fake (money) from the real (money), i.e., 0.5 probability

- Fix D^* in: $\boldsymbol{min_G max_D J(D,G) = min_G J(D^*,G)}$
- Recall Jensen-Shannon divergence:

$$D_{JS}(d,g) = \frac{1}{2}KL(d|m) + \frac{1}{2}KL(g|m), m = \frac{1}{2}(d+g)$$

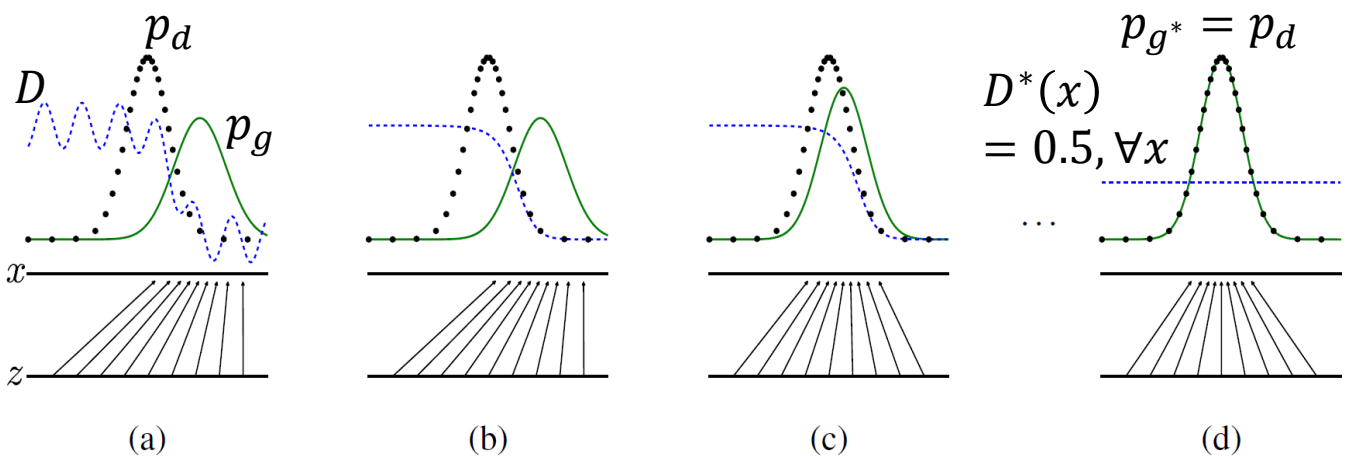
- $J(D^*,G) = \mathbb{E}_{\boldsymbol{x}\sim p_d}[\log D^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log[1 - D^*(\boldsymbol{x})]]$
- $= \mathbb{E}_{\boldsymbol{x}\sim p_d}\left[\log \frac{p_d(\boldsymbol{x})}{p_d(\boldsymbol{x})+p_g(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\log\left[1 - \frac{p_d(\boldsymbol{x})}{p_d(\boldsymbol{x})+p_g(\boldsymbol{x})}\right]\right]$
- $= \mathbb{E}_{\boldsymbol{x}\sim p_d}\left[\log \frac{p_d(\boldsymbol{x})}{p_d(\boldsymbol{x})+p_g(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\log \frac{p_g(\boldsymbol{x})}{p_d(\boldsymbol{x})+p_g(\boldsymbol{x})}\right]$
- $= D_{JS}\left(p_d(\boldsymbol{x})||p_g(\boldsymbol{x})\right) - 2\log 2$
- Therefore,
  - $min_G\left(D_{JS}\left(p_d(\boldsymbol{x})||p_g(\boldsymbol{x})\right)\right) \rightarrow p_{g^*} = p_d$



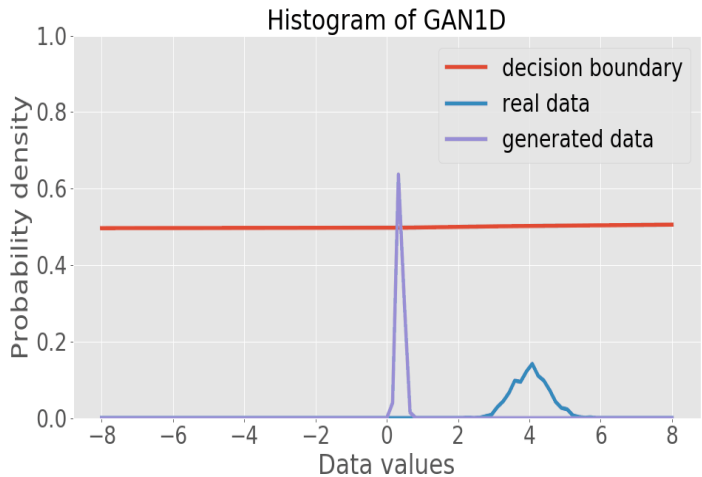
Optimal G

The Nash equilibrium point

- The solution of the minimax problem $\boldsymbol{min_G max_D J(D, G)}$
  - Nash equilibrium point (D^*, G^*) which satisfies
    - $p_{g^*} = p_d$
    - $D^*(\boldsymbol{x}) = \frac{p_d(\boldsymbol{x})}{p_d(\boldsymbol{x}) + p_{g^*}(\boldsymbol{x})} = 0.5, for\ all\ x$



[Goodfellow et al, NIPS 14]

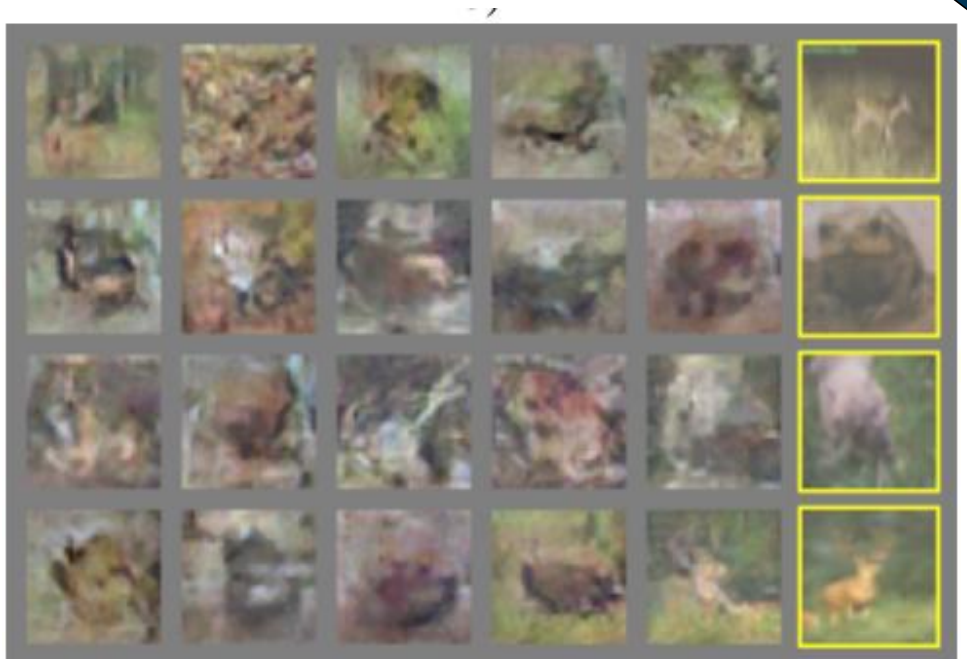


Our own simulation

GAN: example results

MNIST

nearest neighbour
samples from training

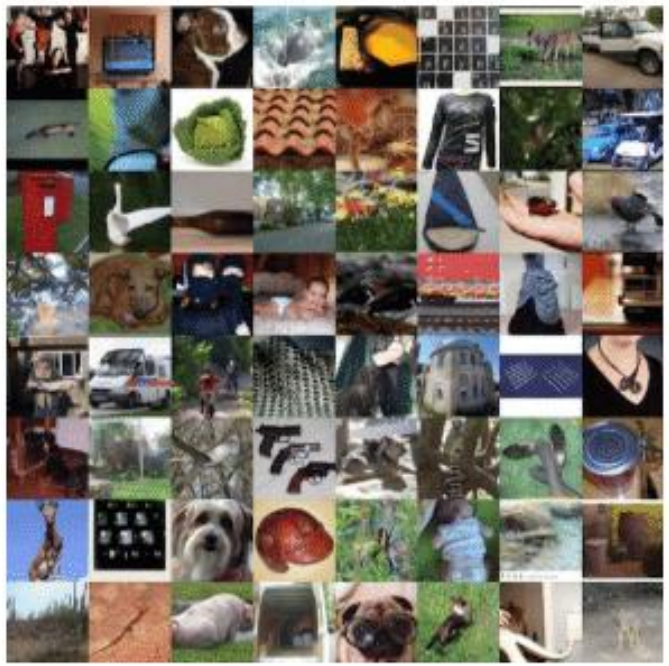


CIFAR-10

Figure 3: Digits obtained by linearly interpolating between coordinates in z space of the full model.

(from Goodfellow, et al, NIPS14's paper)

GAN: example results



Real images (ImageNet)

Generated images

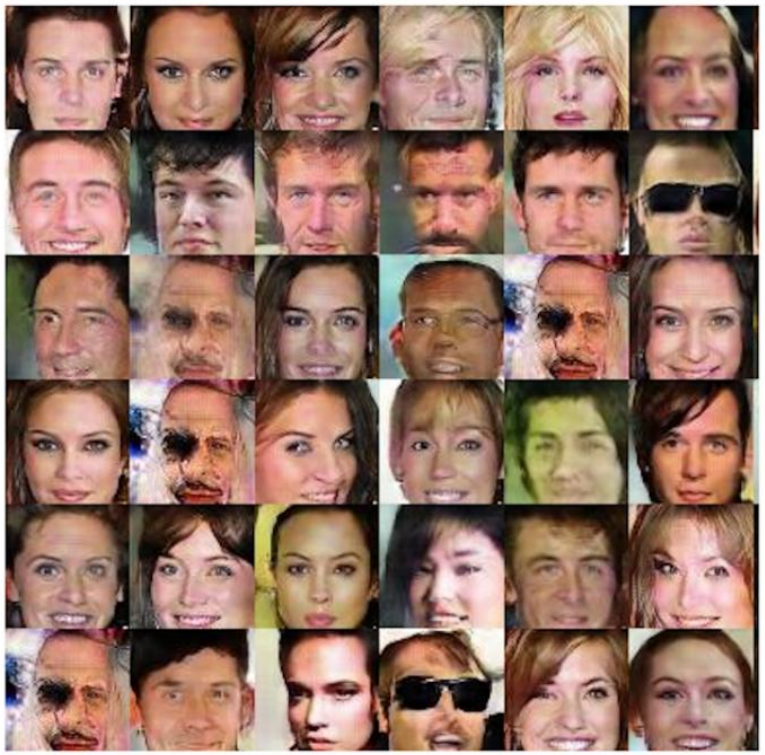
- For a complex dataset such as ImageNet
  - GANs can generate sharp images, but look unrealistic
  - Hard to know what objects they are
- Conditional GANs with labels work much better in practice!

Issues with GAN

Two serious technical problems with GAN

- #1: mode collapsing problem
  - Impeach the ability to generate diverse, realistic data/images

Mode collapse example 1: Generate only some digits {0, 1, 3, 5, 7} while learning from MNIST dataset



Mode collapse example 2: Generate only some faces while learning from CelebA dataset

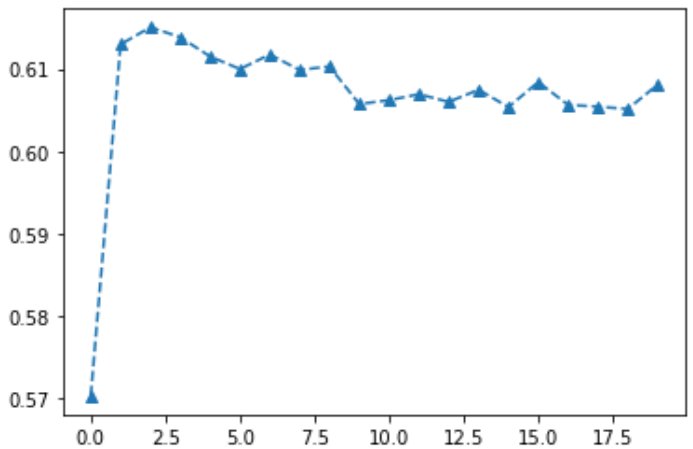
- Generated (fake) examples $G(\boldsymbol{z})$ where $\boldsymbol{z}\sim p(\boldsymbol{z})$
  - Can only cover a few modes in the data distribution $p_d(\boldsymbol{x})$
  - Miss many other modes in the data distribution.

(Image credit: deeplearning.ai)

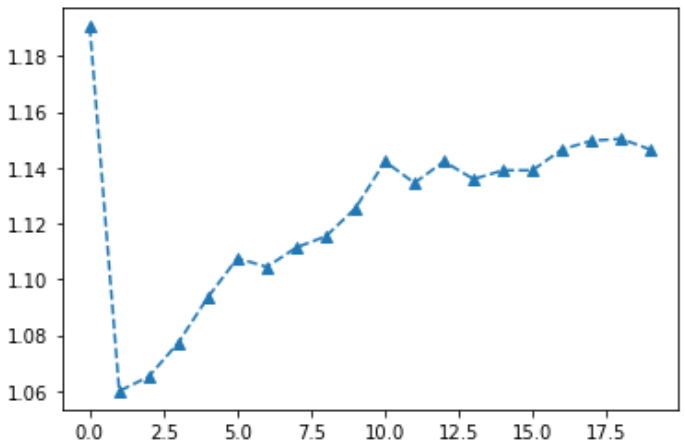
Issues with GAN

Two serious technical problems with GAN

- #2: Convergence is hard due to minimax formulation
  - Training GAN is very challenging!

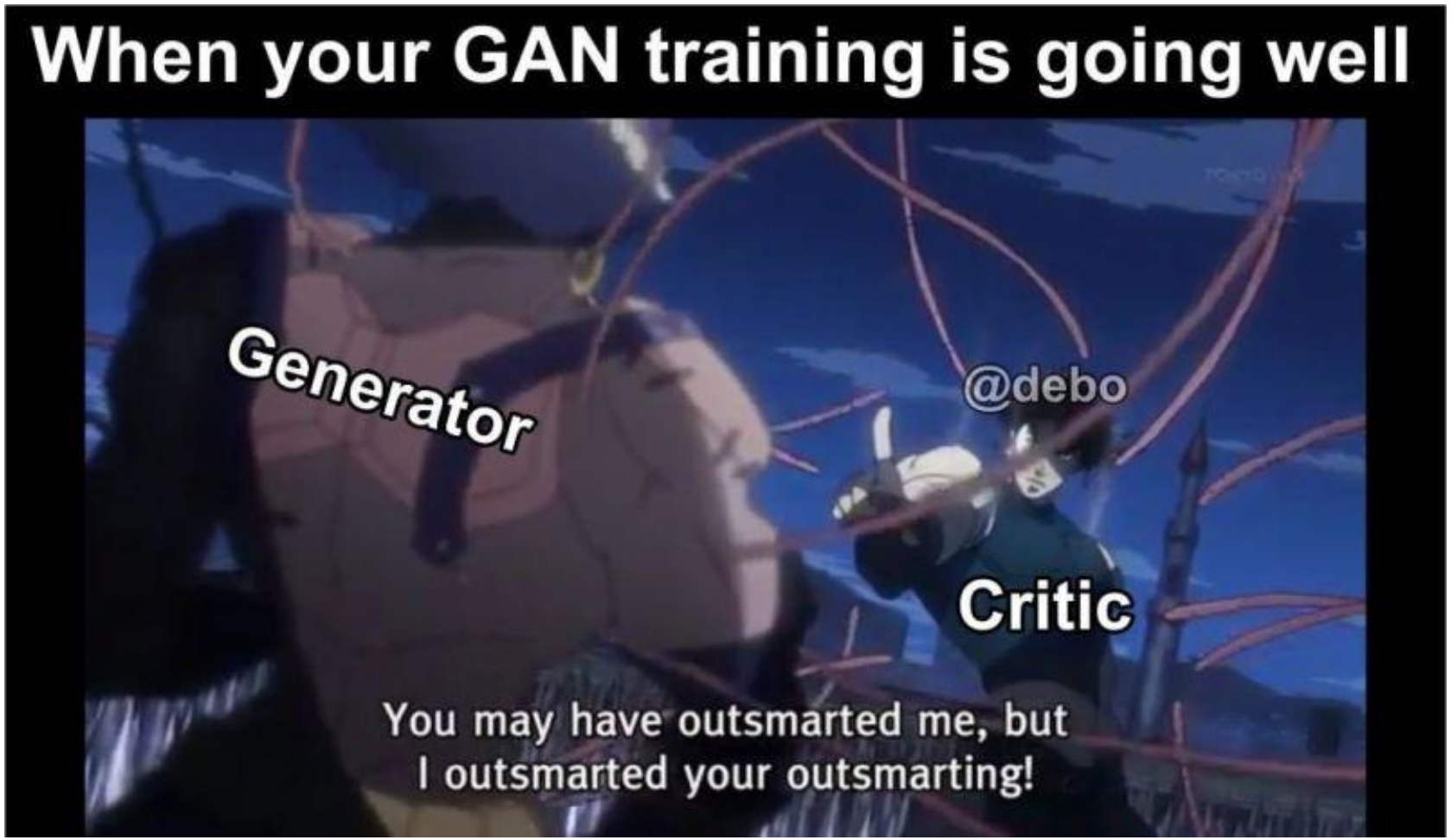


Discriminator loss



Generator loss

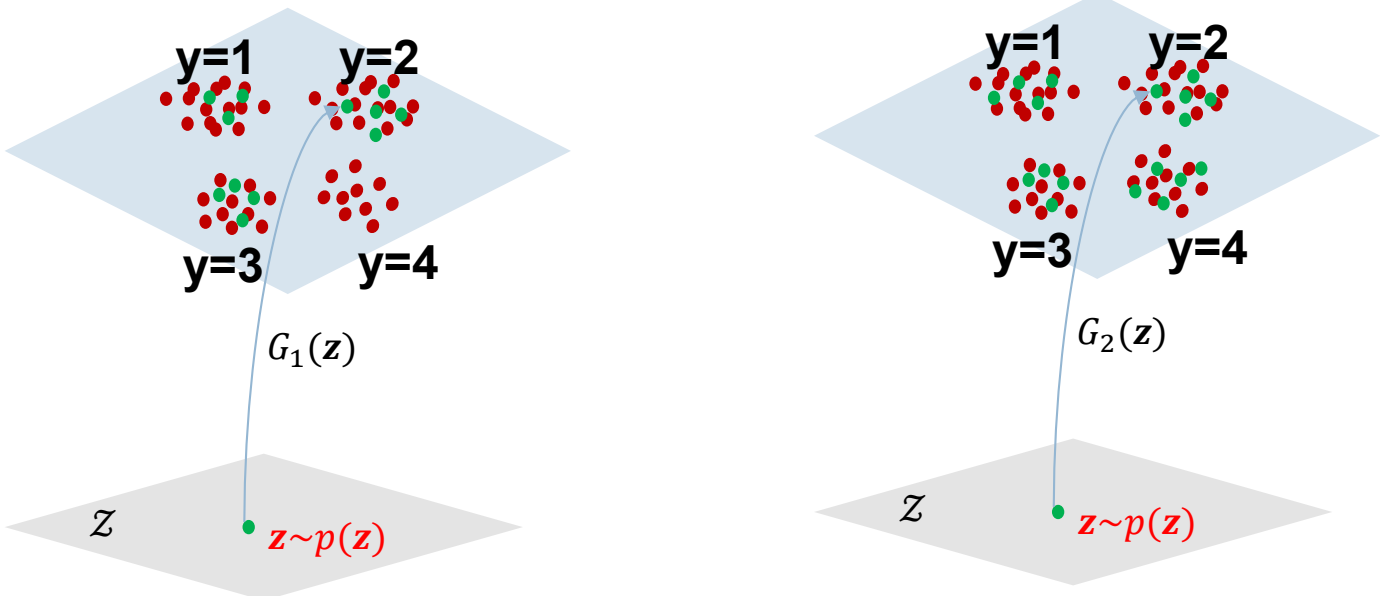
- Mini-max problem
  - $\min_G \max_D J(G, D) = \mathbb{E}_{x \sim p_d(x)} [\log D(\boldsymbol{x})] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(\boldsymbol{z}))]$
- No unique loss function



(image credit: deeplearning.ai)

How to evaluate GAN: Inception score

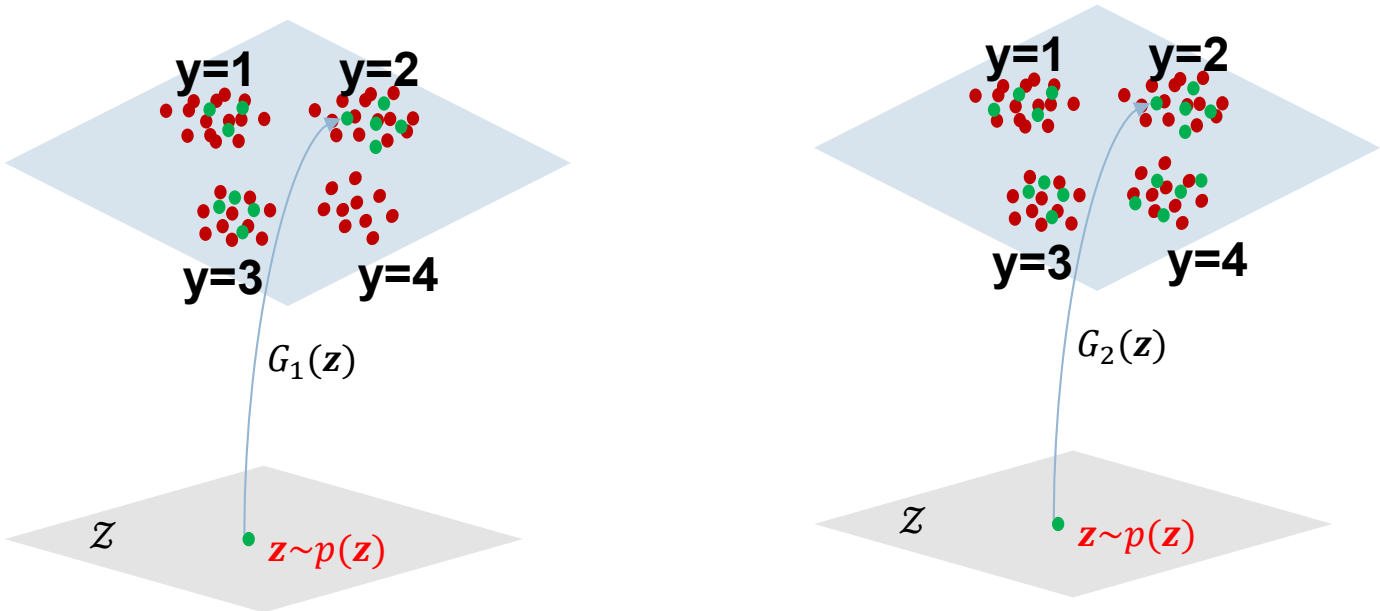
- Except for trivial cases, we don't know the true data distribution p_d so how can we say how good the generated distribution p_g is?
- How to know that a generative model can generate **diverge examples** that cover **many modes** in data?
- How to **compare two generative models** trained on the same training set?



Two generators $\boldsymbol{G_1}$ and $\boldsymbol{G_2}$, which one is better?

How to evaluate GAN: Inception score

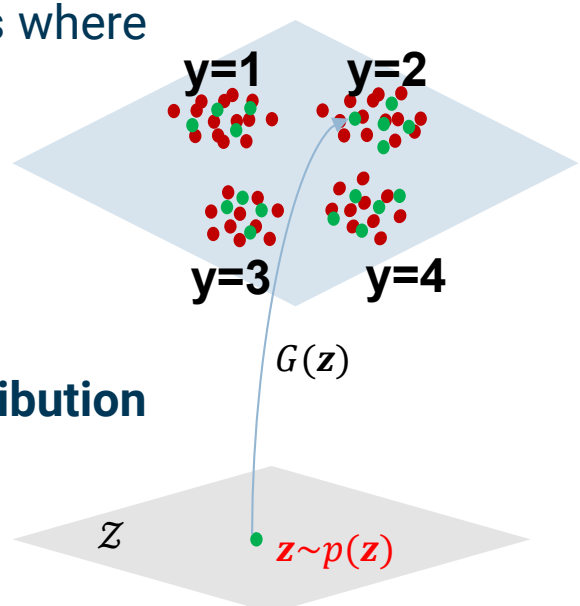
- Except for trivial cases, we don't know the true data distribution p_{data} so how can we say how good the generated distribution p_g is?
- How to know that a generative model can generate **diverge examples** that cover **many modes** in data?
- How to **compare two generative models** trained on the same training set?



$\boldsymbol{G_2}$ is better and generate more diverge examples, while $\boldsymbol{G_1}$ misses the data mode $y = 4$.

How to evaluate GAN: Inception score

- Assume that training examples have labels
  - y=1,2,3, and 4.
- We train a good classifier C on the training set with labels
  - $C(\boldsymbol{x}) = p(y \mid \boldsymbol{x}) = [p(y = k \mid \boldsymbol{x})]_{k=1}^{4}$ is the prediction probabilities where $p(y = k \mid \boldsymbol{x})$ is the probability to classify x to the class k.
- Given $\tilde{\boldsymbol{x}}_i = G(\boldsymbol{z_i}), i = 1, \dots, T$, we wish
  - $C(\tilde{\boldsymbol{x}}_i) = p(y \mid \tilde{\boldsymbol{x}}_i = G(\boldsymbol{z_i}))$ is **close** to an **one-hot vector**
    - $C(\tilde{\boldsymbol{x}}_i)$ is **confident** and has a **small entropy**.
  - $\frac{1}{T}\sum_{i=1}^{T} C(\tilde{\boldsymbol{x}}_i) = \frac{1}{T}\sum_{i=1}^{T} p(y \mid \tilde{\boldsymbol{x}}_i = G(\boldsymbol{z_i}))$ is close to the **uniform distribution** $[0.25, 0.25, 0.25, 0.25]$ (largest entropy).
- Inception score
  - $\frac{1}{T}\sum_{i=1}^{T} KL\left(C(\tilde{\boldsymbol{x}}_i), \frac{1}{T}\sum_{j=1}^{T} C(\tilde{\boldsymbol{x}}_j)\right) \approx \mathbb{E}_z[KL(p(y \mid \tilde{\boldsymbol{x}} = G(\boldsymbol{z})), p(y))]$
  - **Higher** is **better** → **more diversity** of generated images



Salimans et al, *Improved Techniques for Training GANs*, 2016

GAN: Summary

- Adversarial training: a game of two players [Goodfellows et al., 2014]
  - Generator G: generate fake samples indistinguishable from real samples
  - Discriminator D: discriminate real and fake samples
- Min-max problem

$$\min_G \max_D J(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_d(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}))]$$

- Issues and challenges
  - Mode collapse, min-max problem

> This, and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion.

Yann Lecun

Thanks for your attention!

Question time