# Accelerator Architectures for Machine Learning

## Lecture 2: Deep Neural Networks

Tsung Tai Yeh

Friday: 3:30 – 6:20 pm

Classroom: ED-302

# Acknowledgements and Disclaimer


- Slides was developed in the reference with
  Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019 tutorial
  Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
  Yakun Sophia Shao,  EE290-2: Hardware for Machine Learning, UC Berkeley, 2020
  CS231n Convolutional Neural Networks for Visual Recognition, Stanford University, 2020
  CS224W: Machine Learning with Graphs, Stanford University, 2021

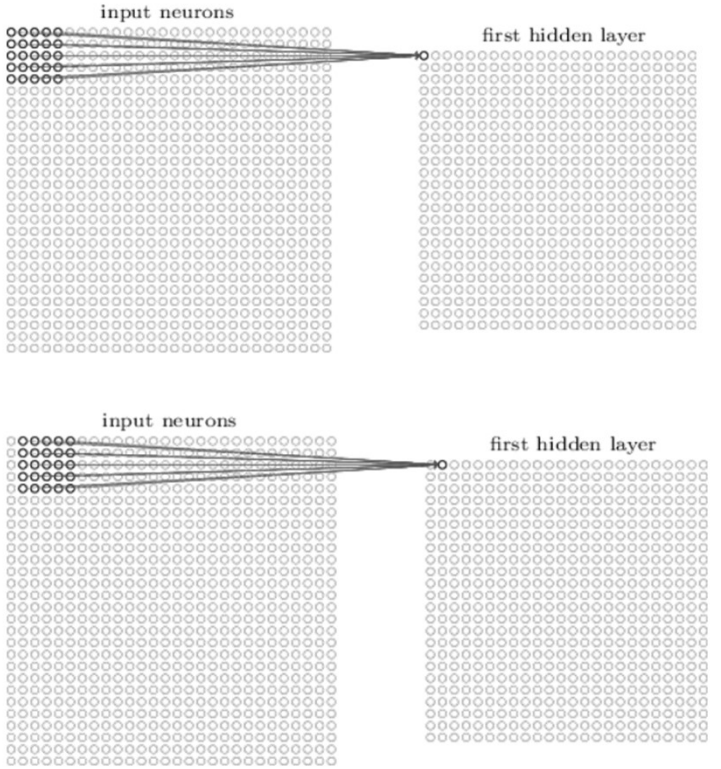
# Outline

- Convolution Neural Network
- Transformer

# Convolutional Neural Networks

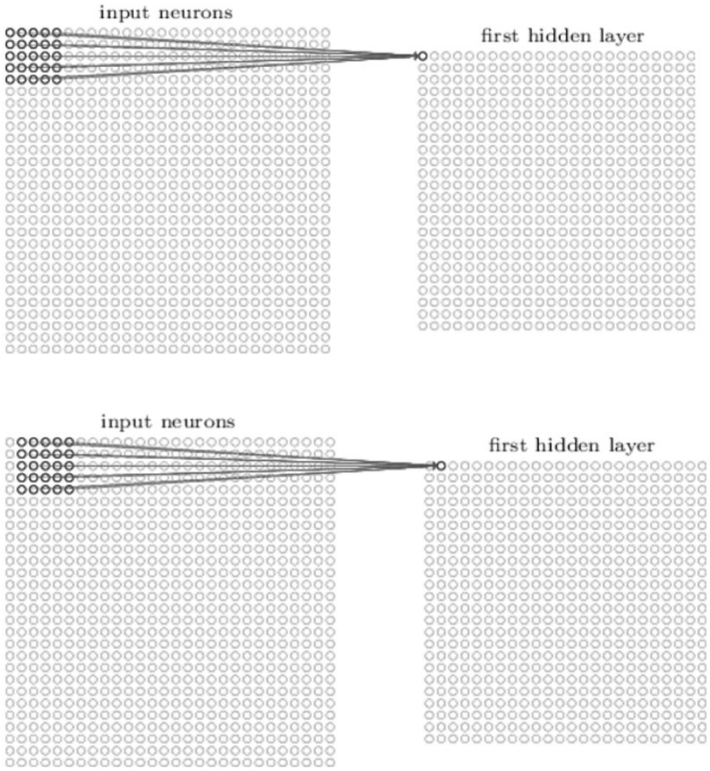
# Deep convolutional neural networks

- Each neuron only sees a "**local receptive field**"
  - 5 x 5 grid of neurons in this example
  - The first neuron is looking for feature in the top-left 5x5 corner of the image
  - Combines the 25 inputs with 25 synaptic weights to decide its output
  - The set of 5x5 weights as a "**filter**"



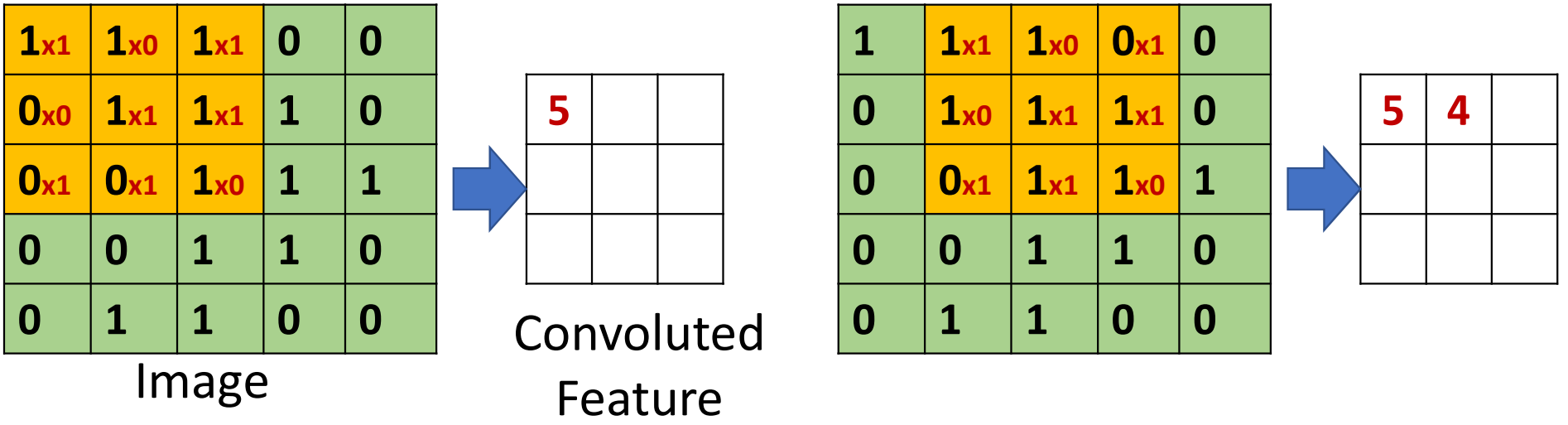
# Deep convolutional neural networks

- **Convolution**
  - Applying a 5x5 filter (kernel) to each part of the image
  - All the neurons are sharing the same set of 25 weights (plus bias)
  - Why do we create small size filter ?
    - The small local receptive field and use of shared weights can help for slow learning rate in early layers of the network



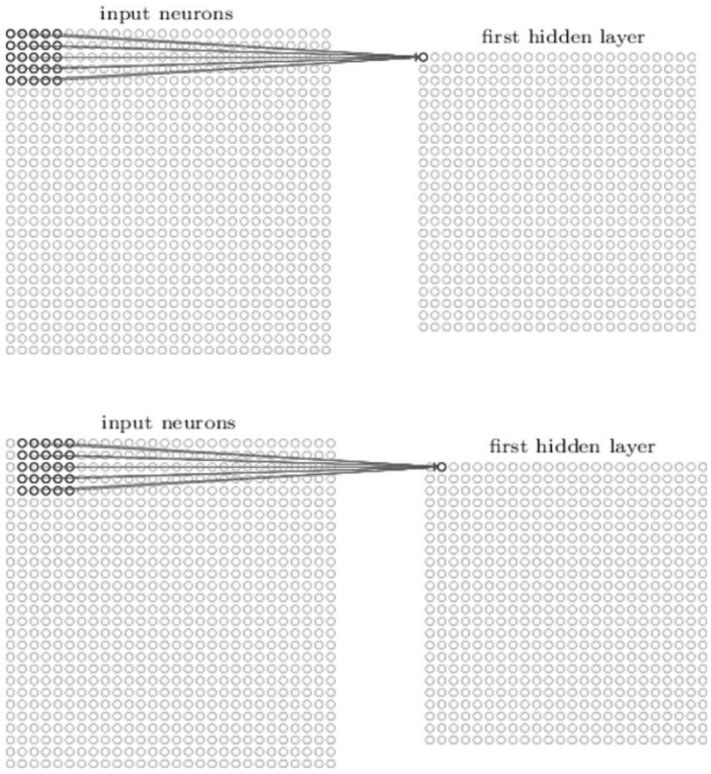
# Convolutional Computation Details

- **Convolution**
  - Expressing how the shape of one function is modified by the other (f*g)
  - Sliding dot product or cross-correlation
  - Convoluting a 5x5x1 image with a 3x3x1 filter kernel to get a 3x3x1 convoluted feature



# Deep convolutional neural networks

- **Feature map (output image)**
  - This example only shows one filter and one resulting “feature map”
  - Multiple **independent filters** -> parallelism
    - 20 filters to the input image -> 20 different feature maps
    - E.g. one filter may look for vertical lines, others may look for circles, etc.
    - This first layer has 784 (28 x 28) inputs and 20 x 24 x 24 neurons -> large neural network
    - How to reduce the amount of learning ?
    - Using **shared weights**, we would only have to learn **20 x (5 x 5 + 1) weights and biases**



# CNN Dimension Parameters

- N – Number of **input fmaps**/**output fmaps** (**batch size**)
- C – Number of 2D **input fmaps**/**filters** (**channels**)
- H – Height of **input fmap** (**activations**)
- W – Width of **input fmap** (**activations**)
- R – Height of 2D **filter** (**weights**)
- S – Width of 2D **filter** (**weights**)
- M – Number of 2D **output fmaps** (**channels**)
- F – Width of **output fmap** (**activations**)
- E – Height of **output fmap** (**activations**)

# CONV Layer Tensor Computation

**Output fmaps (Y)** **Bias (B)** **Input fmaps (X)** **Filter weights (W)**

$$Y[n][m][x][y] = Activation(B[m] + \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} \sum_{k=0}^{C-1} X[n][k][Ux+i][Uy+j] \times W[m][k][i][j])$$

$$0 \le n \le N, 0 \le m \le M, 0 \le y \le E, 0 \le x \le F$$

$$E = (H - R + U)/U, F = (W - S + U)/U$$

| Shape Parameter | Description |
|---|---|
| N | fmap batch size |
| M | # of filters or # of output fmap channels |
| C | # of input fmap or # of filter channels |
| U | Convolution stride |

# CONV Layer Implementation

```
for ( n = 0;   n < N;   n++) {
    for (m = 0;   m < M;   m++) {
        for   (x = 0;   x < F;   x++)      {
            for    ( y = 0;   y < E;   y++)    {
                    Y[n][m][x][y] = B[m];
                    for  (  i = 0;   i < R;  i++)    {
                        for  (  j = 0;  j < S;  j++)     {
                            for  (  k = 0;  k < C;  k++) {
                                Y[n][m][x][y] += X[n][k][Ux+i][Uy+j] x W[m][k][i][j];
                            }
                        }
                    }
                    Y[n][m][x][y] = Activation(Y[n][m][x][y]);
            }
        }
    }
}
```

**For each output fmap value**

**CONV & Activation**

How to run CONV in parallel ?

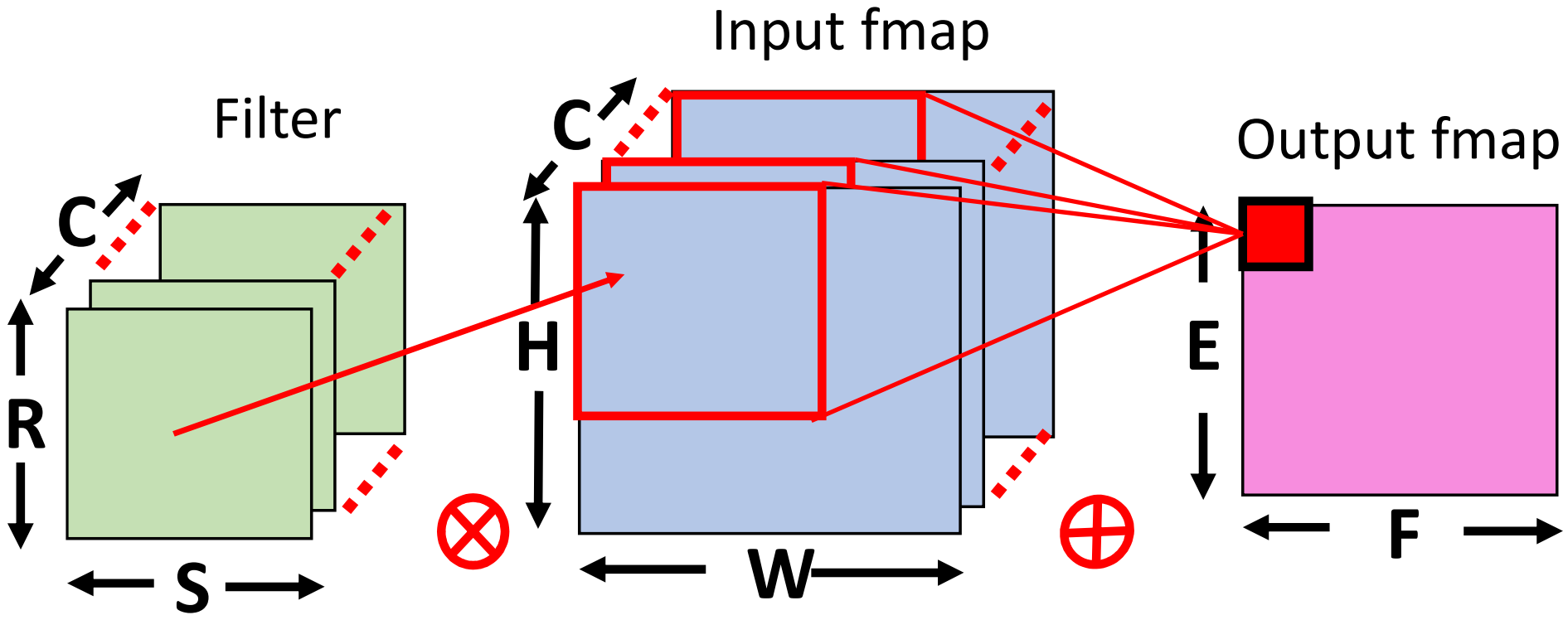
# CONV Layer Parallel Implementation

```
Parallel_for ( n = 0;   n < N;   n++) {
    Parallel_for (m = 0;   m < M;   m++) {
        Parallel_for   (x = 0;   x < F;   x++)      {
            Parallel_for    ( y = 0;   y < E;   y++)    {
                    Y[n][m][x][y] = B[m];
                    for  (  i = 0;   i < R;  i++)    {
                        for  (  j = 0;  j < S;  j++)     {
                            for  (  k = 0;  k < C;  k++) {
                                Y[n][m][x][y] += X[n][k][Ux+i][Uy+j] x W[m][k][i][j];
                            }
                        }
                    }
                    Y[n][m][x][y] = Activation(Y[n][m][x][y]);
            }
        }
    }
}
```

**For each output fmap value**

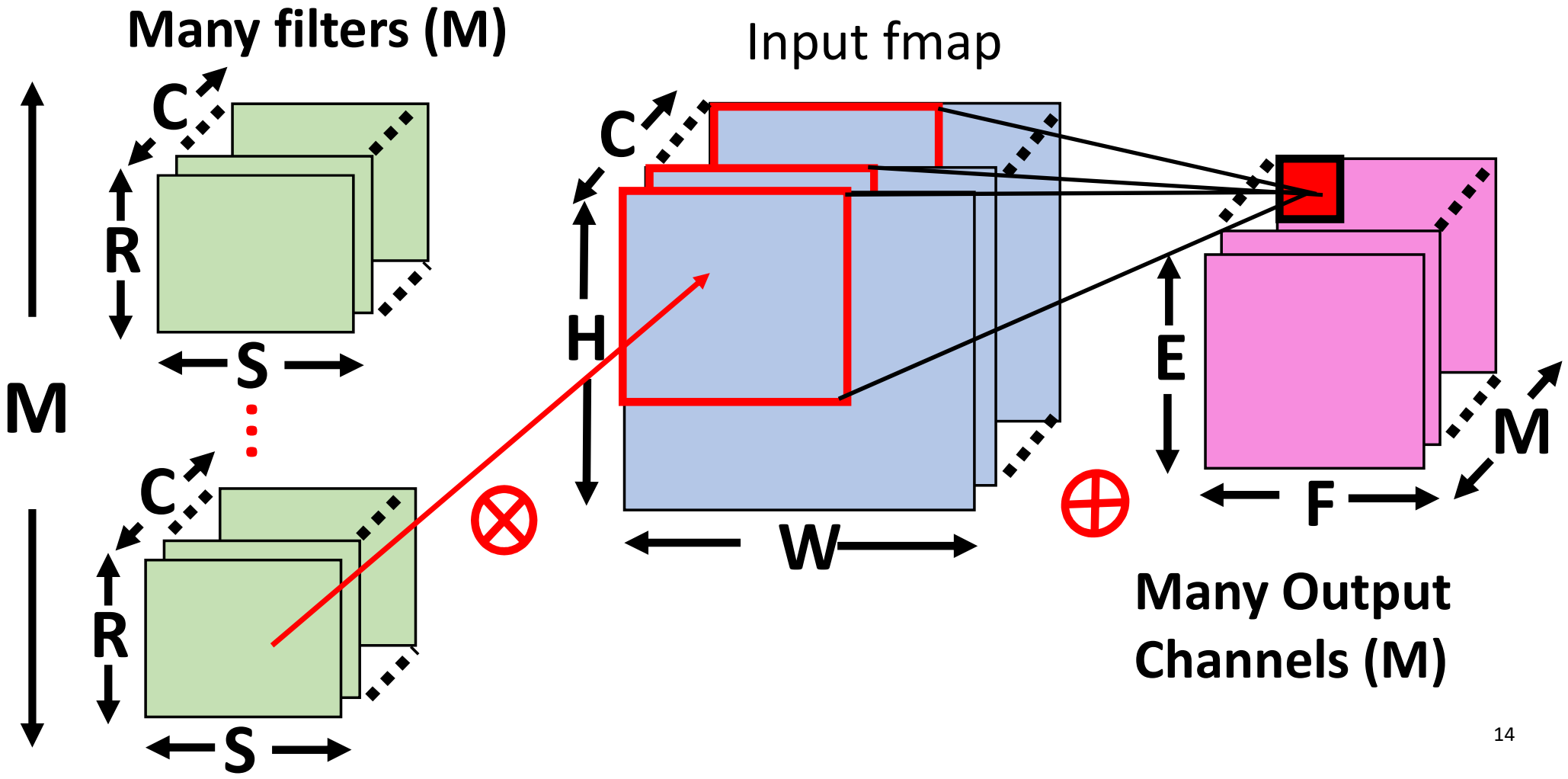
**CONV & Activation**

# Convolution (CONV) Layer

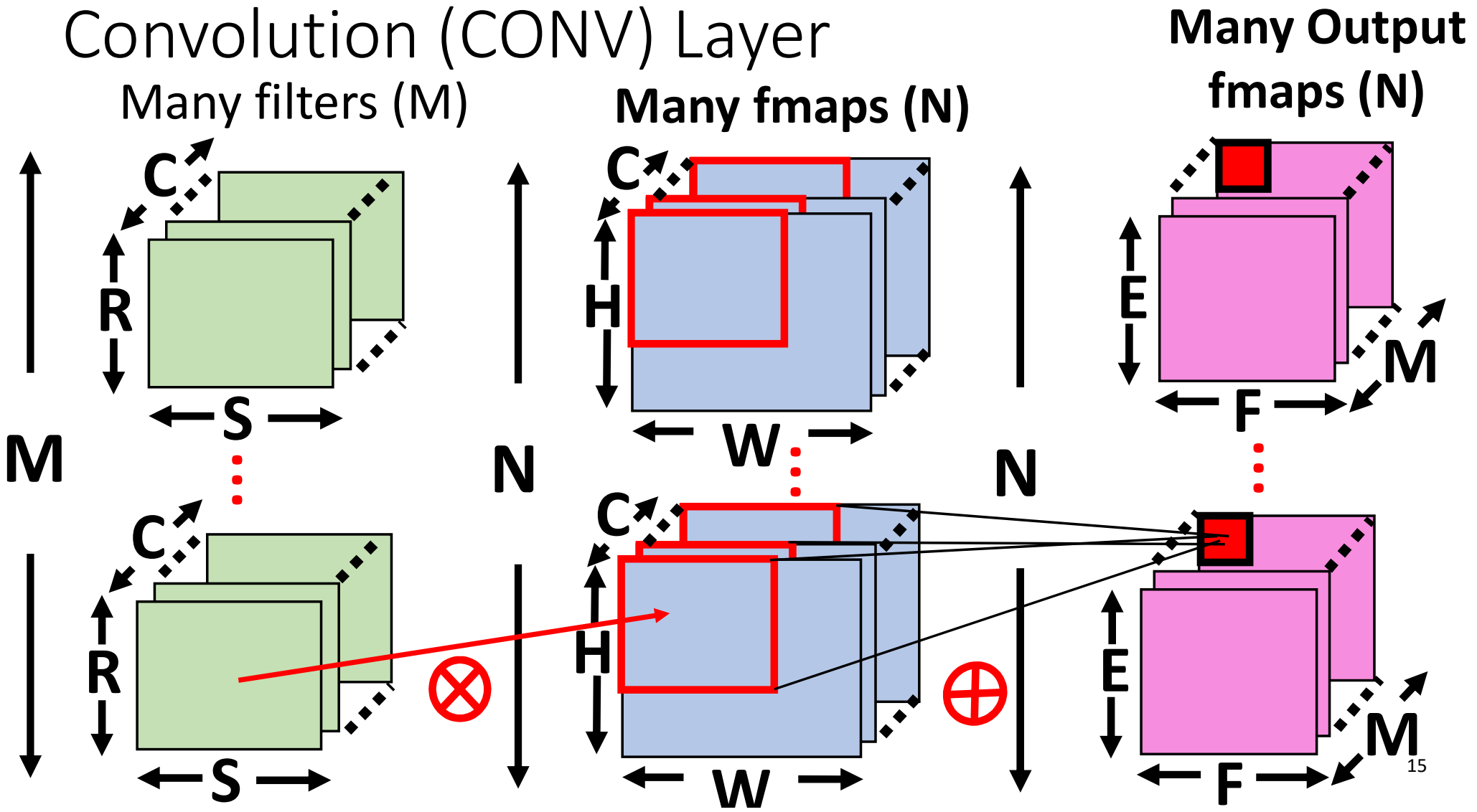



**Many Input Channels (C), e.g. RGB in an image**

# Convolution (CONV) Layer



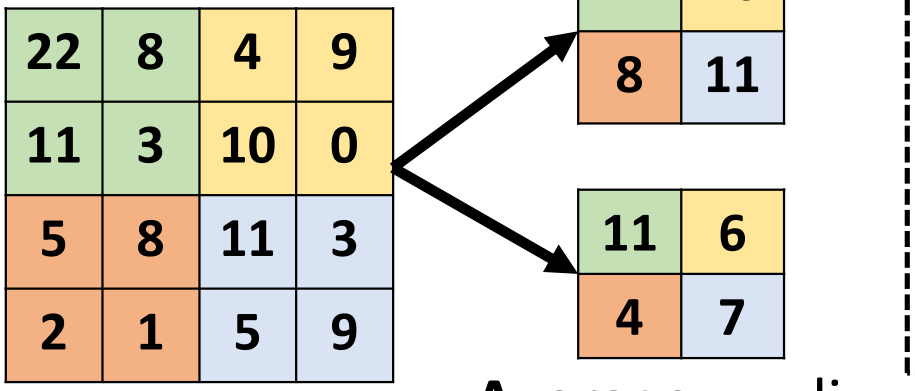
# Convolution (CONV) Layer



# Pooling

- **Pooling**
  - Once a feature has been found, its's exact location isn't as important as its relative location – help us reduce the parameters
  - Further reduce the network, say reduce 4 neurons into a single one

2x2 pooling, stride = 2




**Max** pooling

**Average** pooling

Input fmap size: $W_1 \times H_1 \times C_1$
Spatial extent: **F**
Stride **S**
Output fmap after pooling: $W_2 \times H_2 \times C_2$
$W_2 = ( W_1 - F ) / S + 1$
$H_2 = ( H_1 - F ) / S + 1$
$C_2 = C_1$

https://cs231n.github.io/convolutional-networks/

# POOL Layer Implementation

```
for ( n = 0;   n < N;   n++) {
    for (m = 0;   m < M;   m++) {
        for   (x = 0;   x < F;   x++)      {
            for    ( y = 0;   y < E;   y++)    {
                        max = -Inf
                        for  (  i = 0;   i < R;  i++)    {
                            for  (  j = 0;  j < S;  j++)     {
                                    if  (  X[n][m][Ux+i][Uy+j] > max) {
                                        max = X[n][m][Ux+i][Uy+j];
                                    }
                             }
                        }
                        Y[n][m][x][y] = max;
                }
            }
     }
}
```

for each pooled value

Find the max in each window

# GoogleNet Inception Architecture

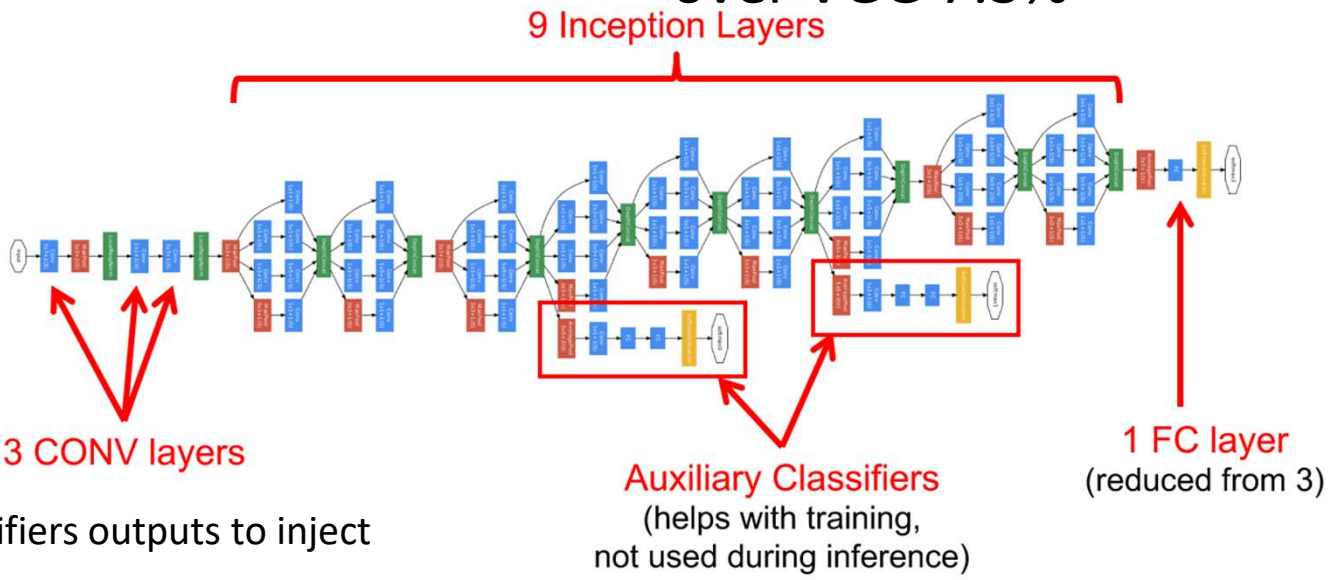
- 22 layers
- Fully-Connected Layers: 1
- **Weights: 7.0 M** (< VGG(19.7X) AlexNet(8.7X))
- MACs: 1.4G

**ILSCVR14 Winner**

**GoogleNet is used to classify images**

GoogleNet top-5 error rate is 6.67%

over VGG 7.3%



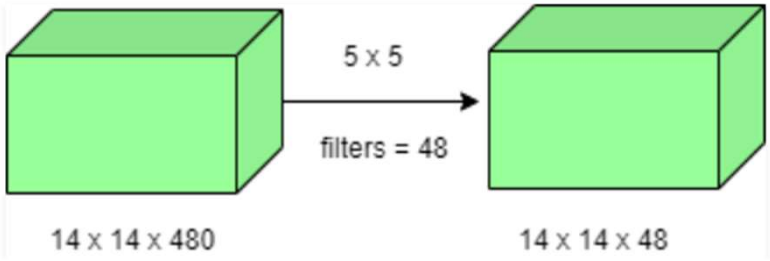

Auxiliary classifiers outputs to inject



http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

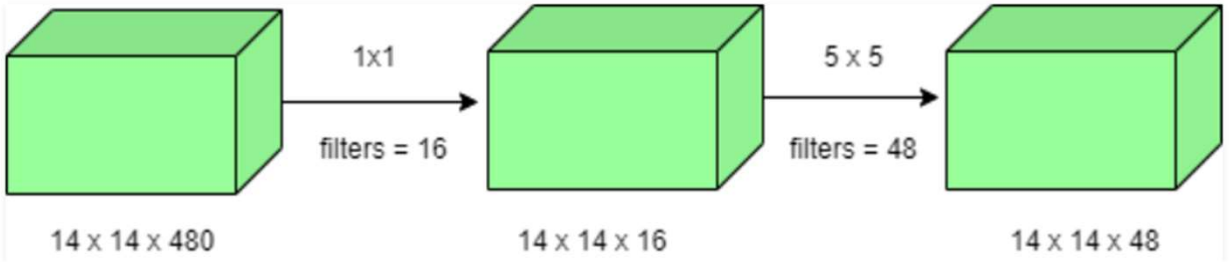
# What's New in GoogleNet ?

- **1 x 1 CONV filter (why?)**
  - Decrease the number of parameters (weights and biases)
  - Increase the depth of the network




Case 1: 5x5 filter, # of filter = 48
Total MACs: (14x14x48)x(5x5x480) = **112.9M**

Case 2: 1x1 filter, # of filter = 16 as intermediate
Total MACs: (14x14x16)x(1x1x480) +
(14x14x48)x(5x5x16)= **5.3M**




GoogleNet can be trained in a single machine such as (GPU with limit memory space) !!

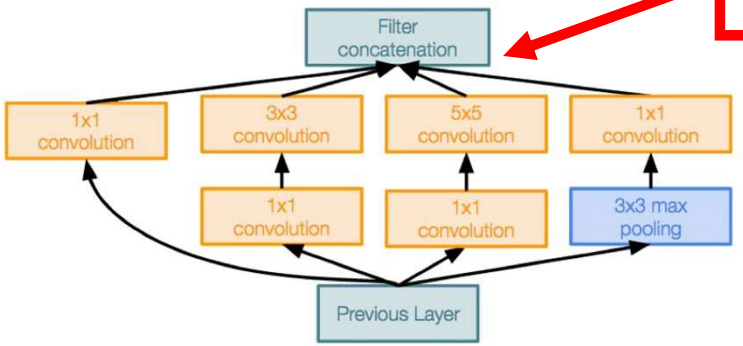


https://www.geeksforgeeks.org/understanding-googlenet-model-cnn-architecture/

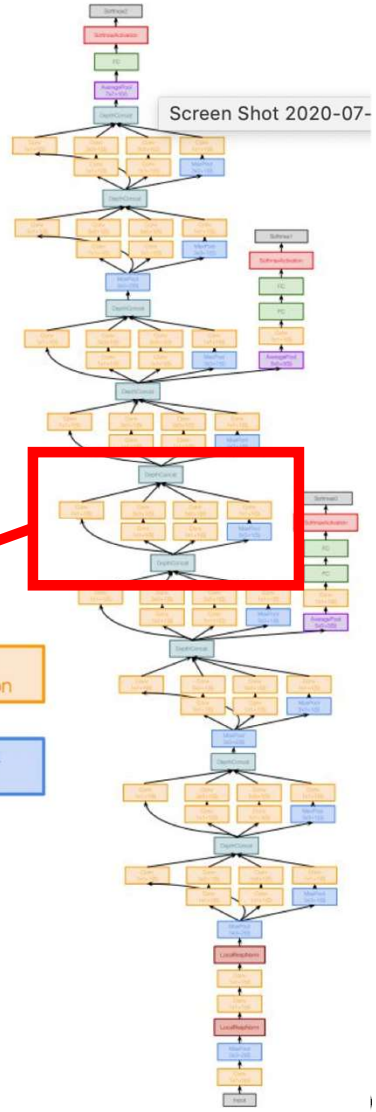
# What's New in GoogleNet ?

- **Inception module**
  - A local network topology (network within a network)
  - Stack modules on top of each other
  - Multiple receptive field sizes for CONV (1x1, 3x3, 5x5)
  - Pooling operation (3x3)
  - Depth-wise filter concatenation

What's the problem of inception module?



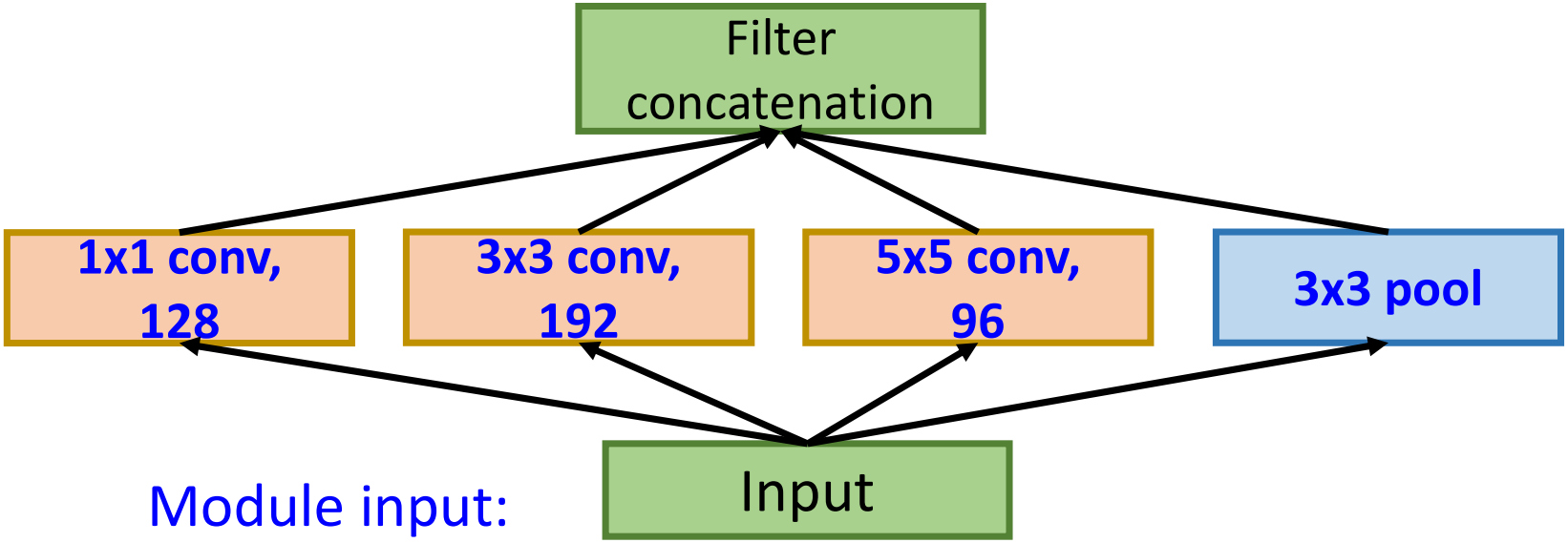

**Inception module**



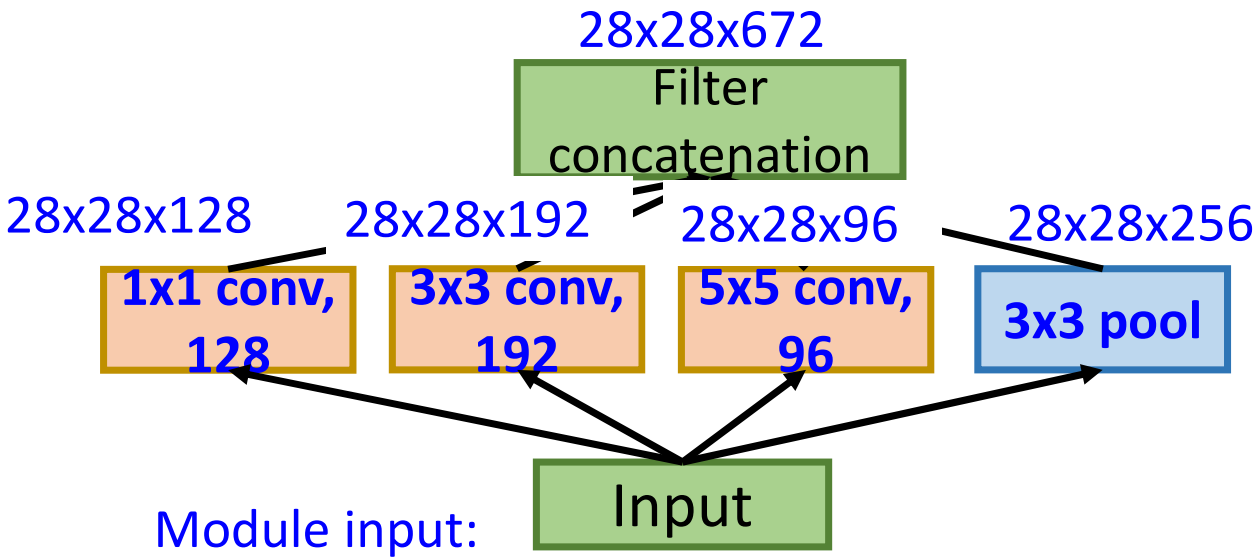

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

# GoogleNet Inception Module Problems?

- What is the output size of 1x1 conv, with 128 filters ?



# GoogleNet Inception Module Problems?




**CONV Ops:**

[1x1 conv, 128] 28x28x128x1x1x256

[3x3 conv, 192] 28x28x192x3x3x256

[5x5 conv, 96] 28x28x96x5x5x256

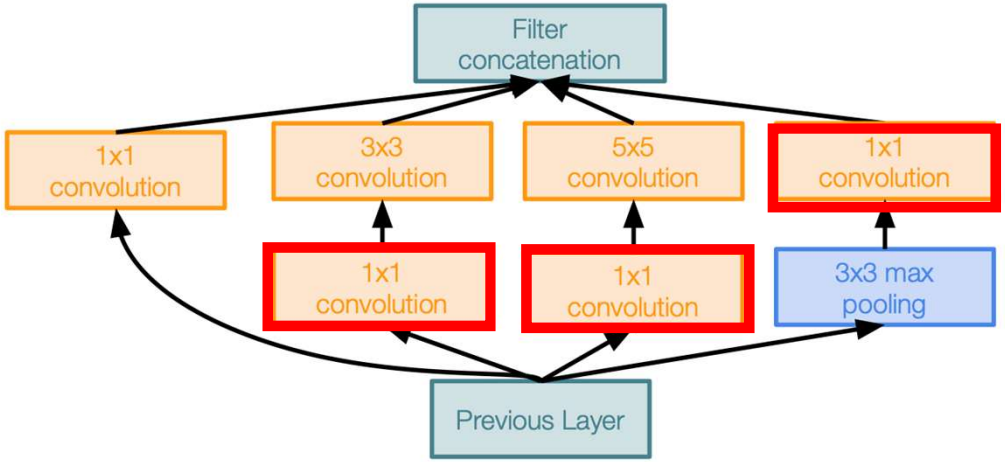
**Total: 854 M ops**

**Very expensive compute**

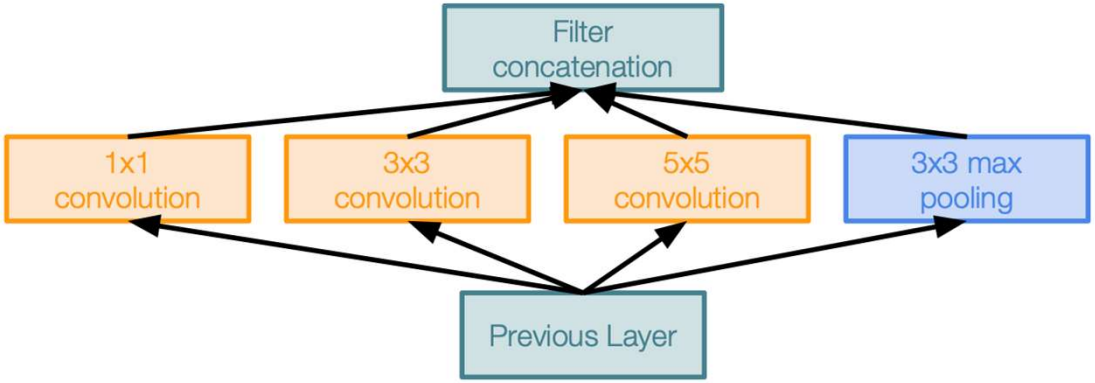
Solution: "bottleneck" layers that use 1x1 convolutions to reduce feature depth

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

# Dimension Reduction on GoogleNet



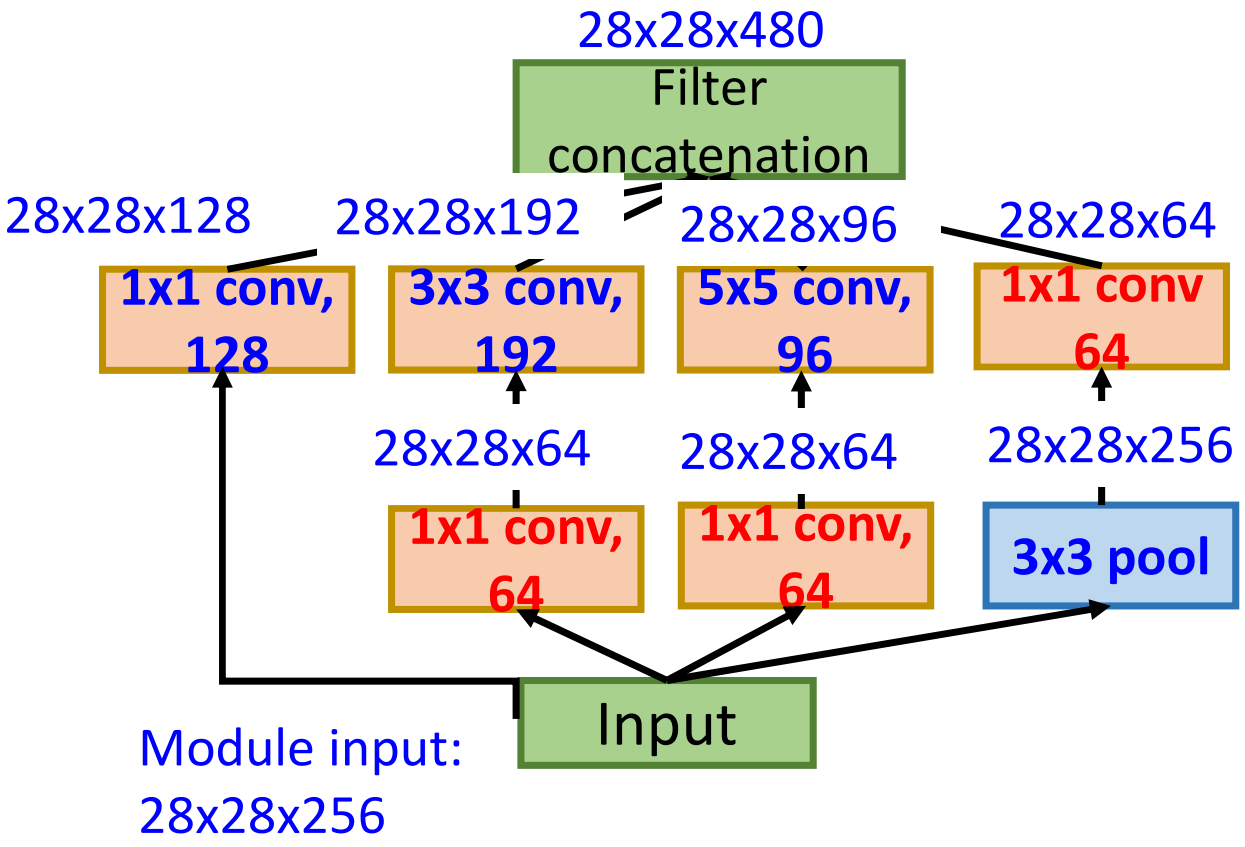

Inception module with **dimension reduction** using **1x1 conv "bottleneck" layers**




Naïve inception module

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

# GoogleNet 1x1 Bottleneck Layer Ops




**Conv Ops:**

[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256

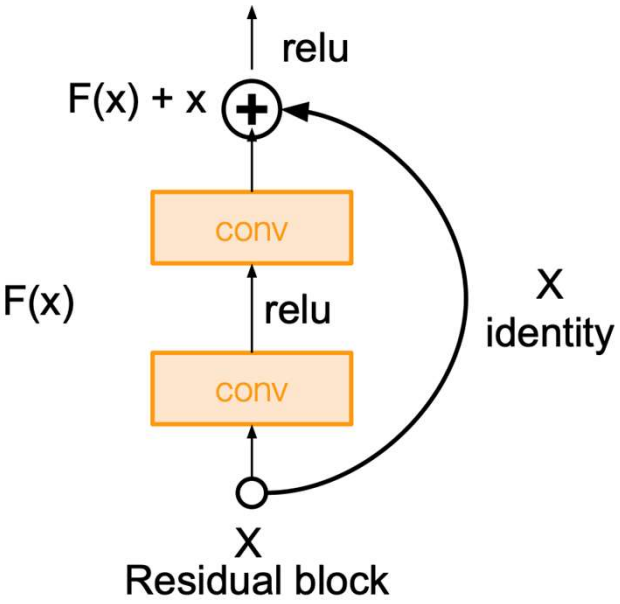
**Total: 358 M ops**

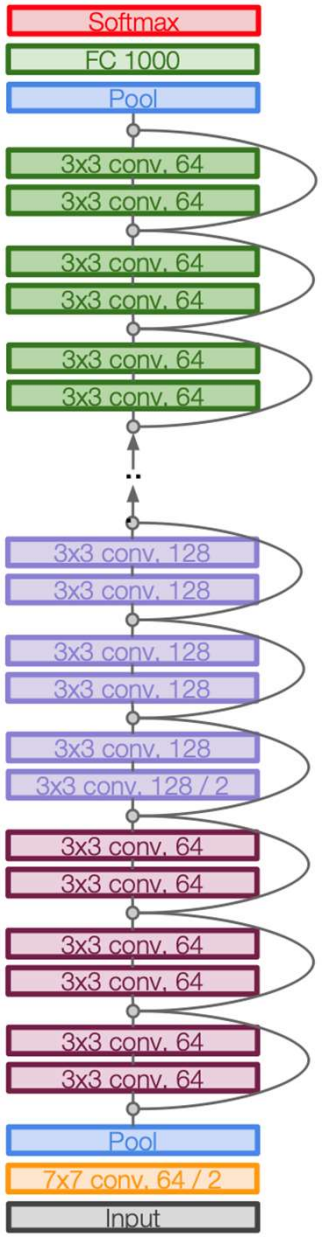
Naïve version has **854M ops**
Bottleneck layer can reduce ops using dimension reduction

# ResNet Model Overview

- 152-layer model for ImageNet Classification
- ILSVRC'15 winner (3.57% top-5 error)
- Using residual blocks and connections

**How to train the data in ULTRA-DEEP network (over 1000 layers)?**

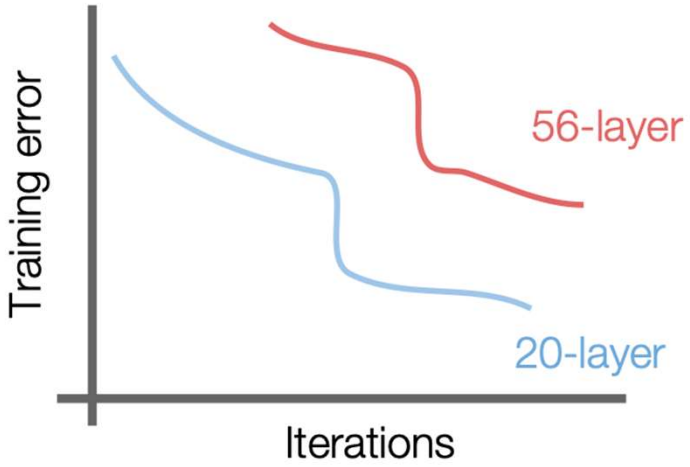


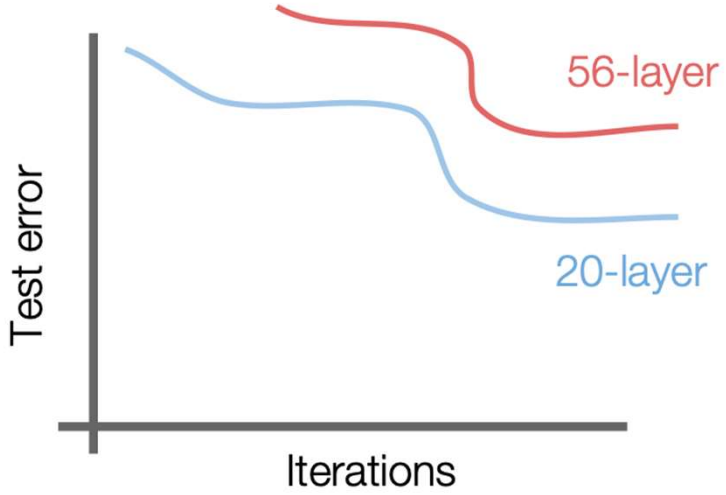




http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

# Deep Network Training Problems on ResNet

- Training and test error are increased with the length of networks



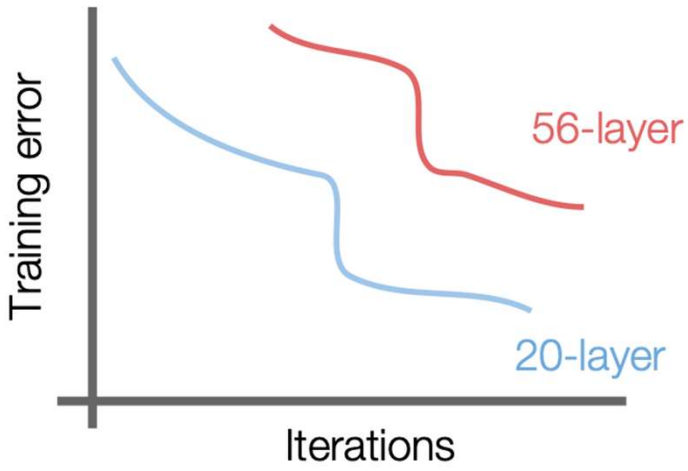



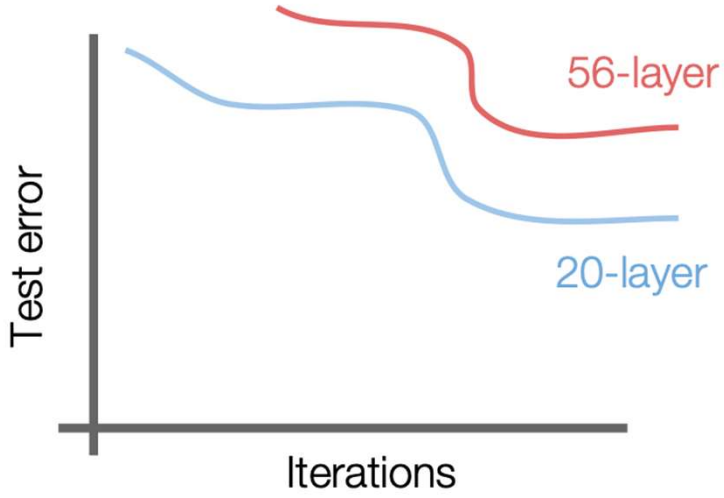

What is wrong when increasing the length of networks?
- Deeper model performs worse on both training and test error (overfitting?)

[He et al., 2015]

# Deep Network Training Problems on ResNet

- Why overfitting isn't the main reason to increase error rate of 56-layer?
- Hypothesis: vanishing gradient raises error rate of ultra-deep networks?
- Solution: Add layers to fit a residual mapping instead of fitting a desired underlying mapping directly (skipping connection)(What?)

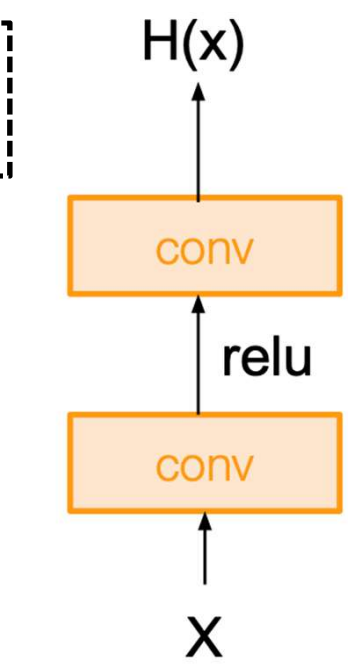


# Deep Network Training Problems on ResNet

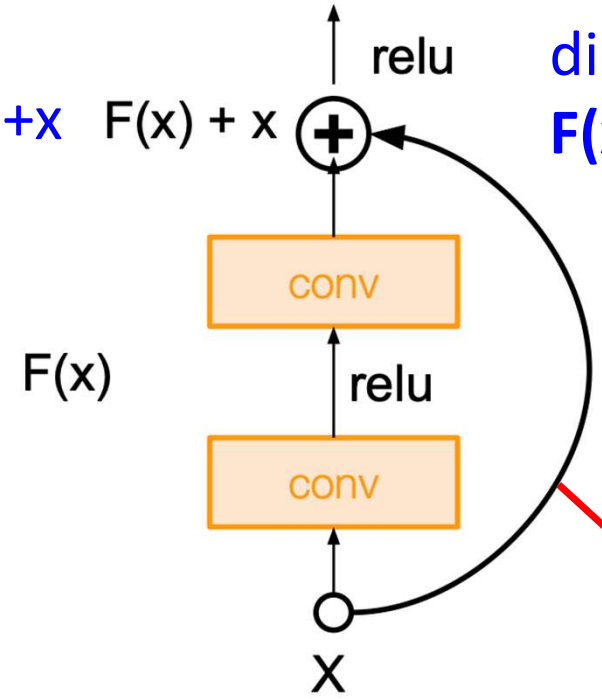
- Solution: Add layers to fit a residual mapping instead of fitting a desired underlying mapping directly

Why not use H(x) directly?




"Original" Layer

H(x) = F(X)+x




"Residual" Block

Instead of fitting H(x) directly, use **residual** **F(x) = H(x) - x**

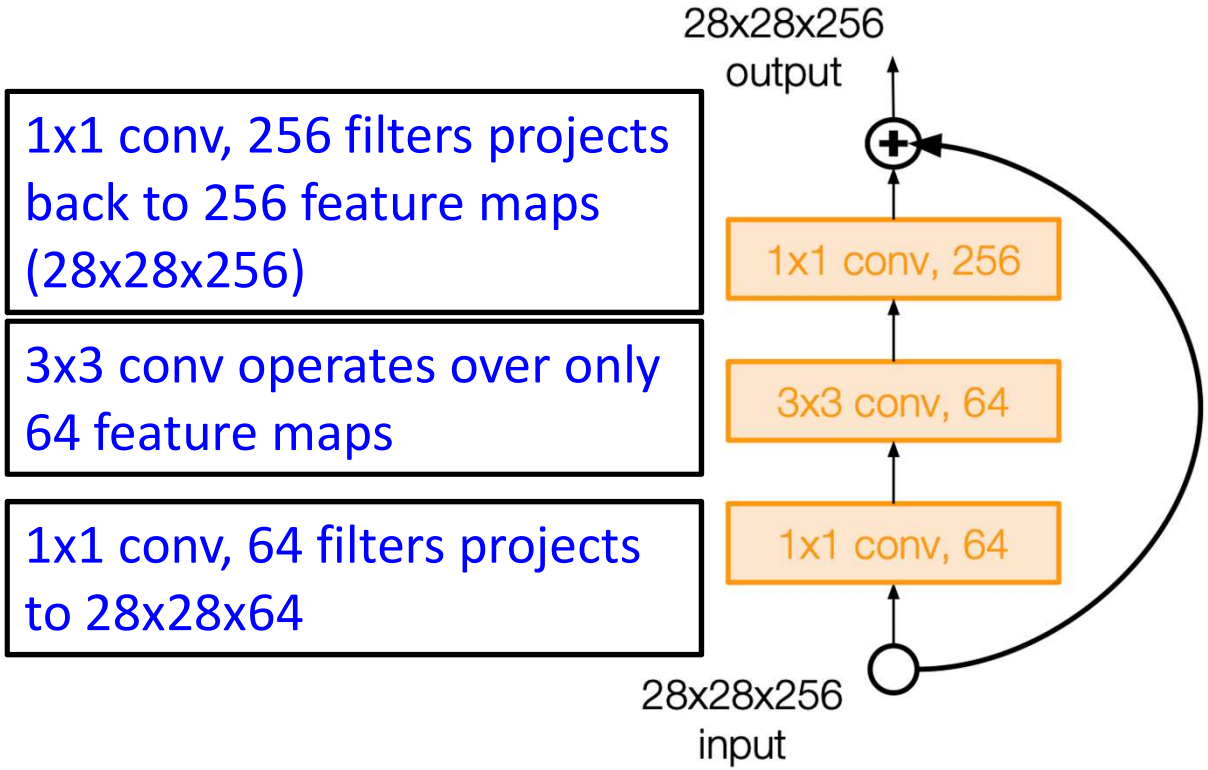
Identity mapping
F(x) = x

X
identity

Shortcut/skip connection

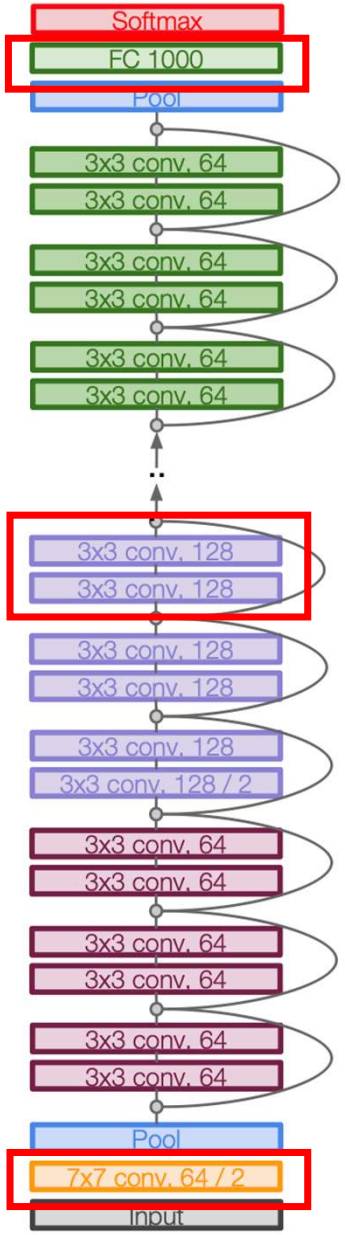
# Bottleneck Layer on ResNet

- ResNet50+ also uses "bottleneck" layer to improve efficiency for deep networks (similar to GoogleNet)



# ResNet Model Details

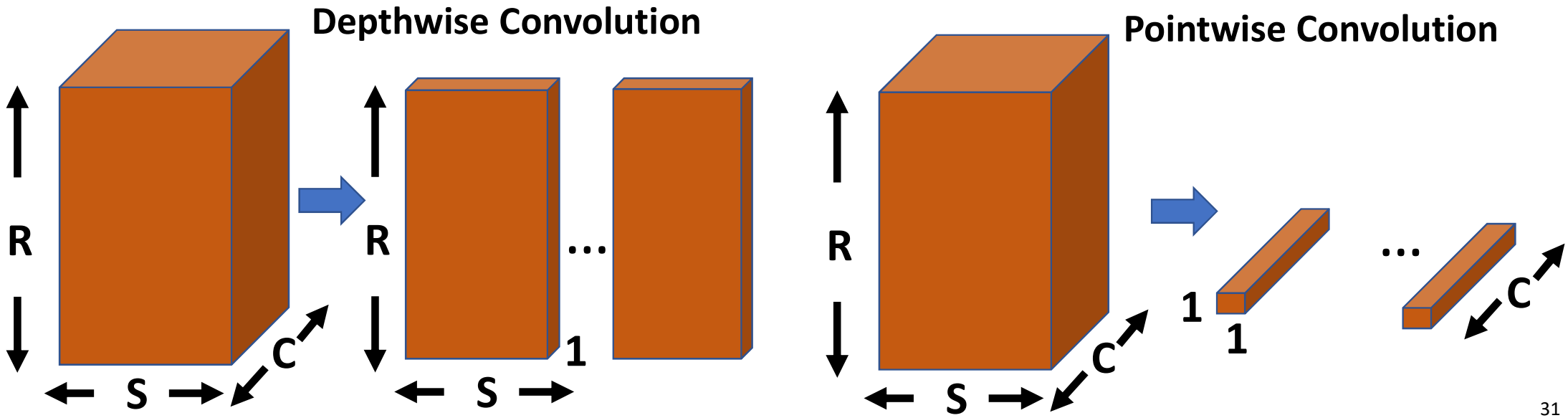
- Full ResNet architecture
  - Stack residual blocks
  - Every residual block has two 3x3 conv layers
  - Periodically, double the number of filters and down-sample using stride 2 (/2 in each dimension)
  - Additional conv layer at the beginning
  - Only FC 1000 to output class
  - Total depths of 34, 50, 101 or 152 layers for ImageNet




http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

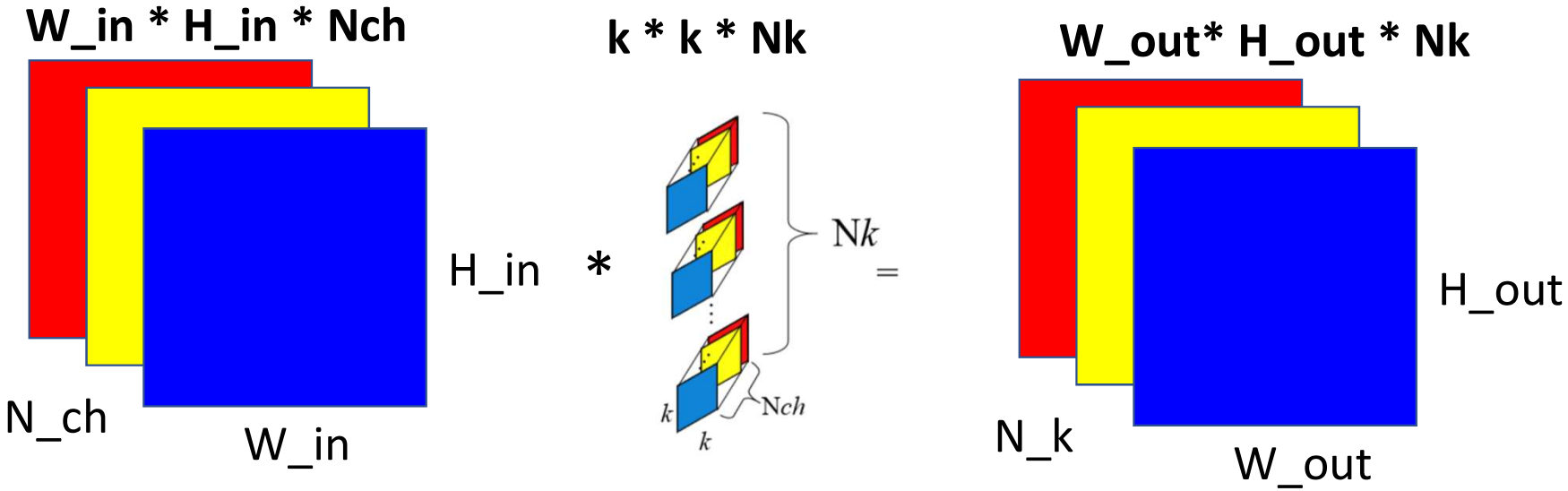
# MobileNet v1: Depthwise Separable CONV

- Decouple **cross-channel correlations** and **spatial correlations** in the feature maps
- How to reduce # of parameters? [Andrew et. al. arxiv, 2017]



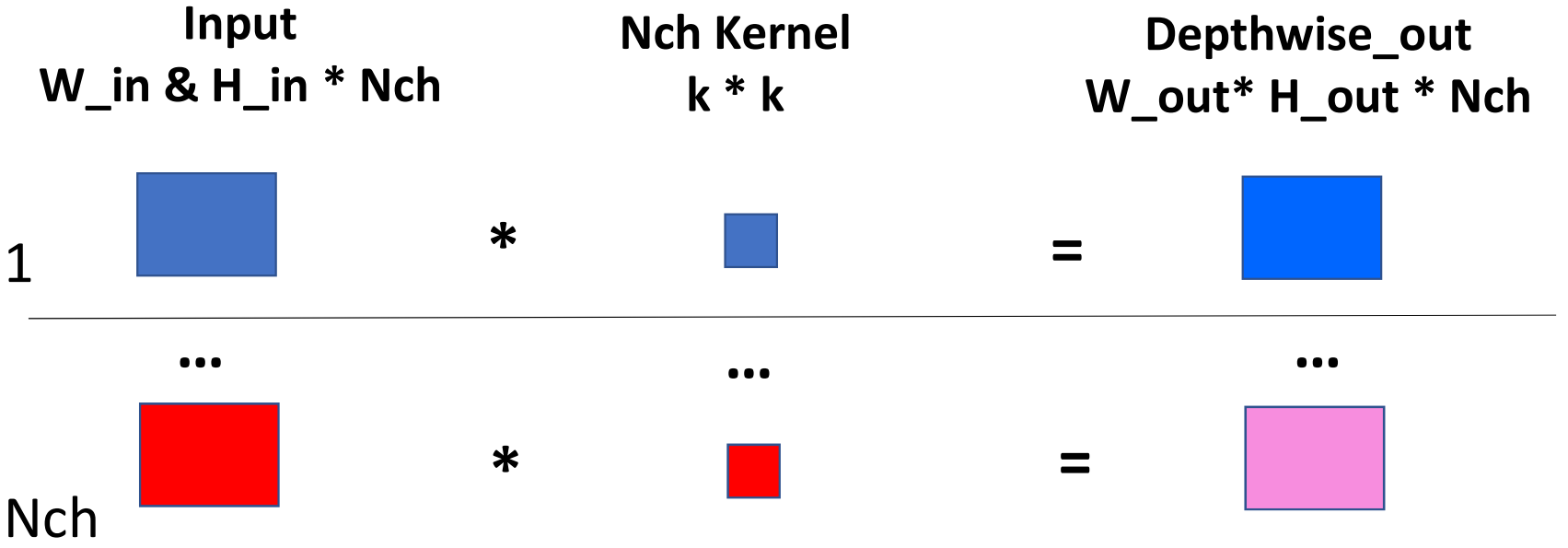
# What is Depthwise Separable Convolution ?

- **Purpose**: Reduce the amount of CONV computation
- **Input:** W_in * H_in * Nch (# of channels)
- **Kernel:** k * k * Nk (# of kernels)
- **Output:** W_out * H_out * Nk (# of kernels)



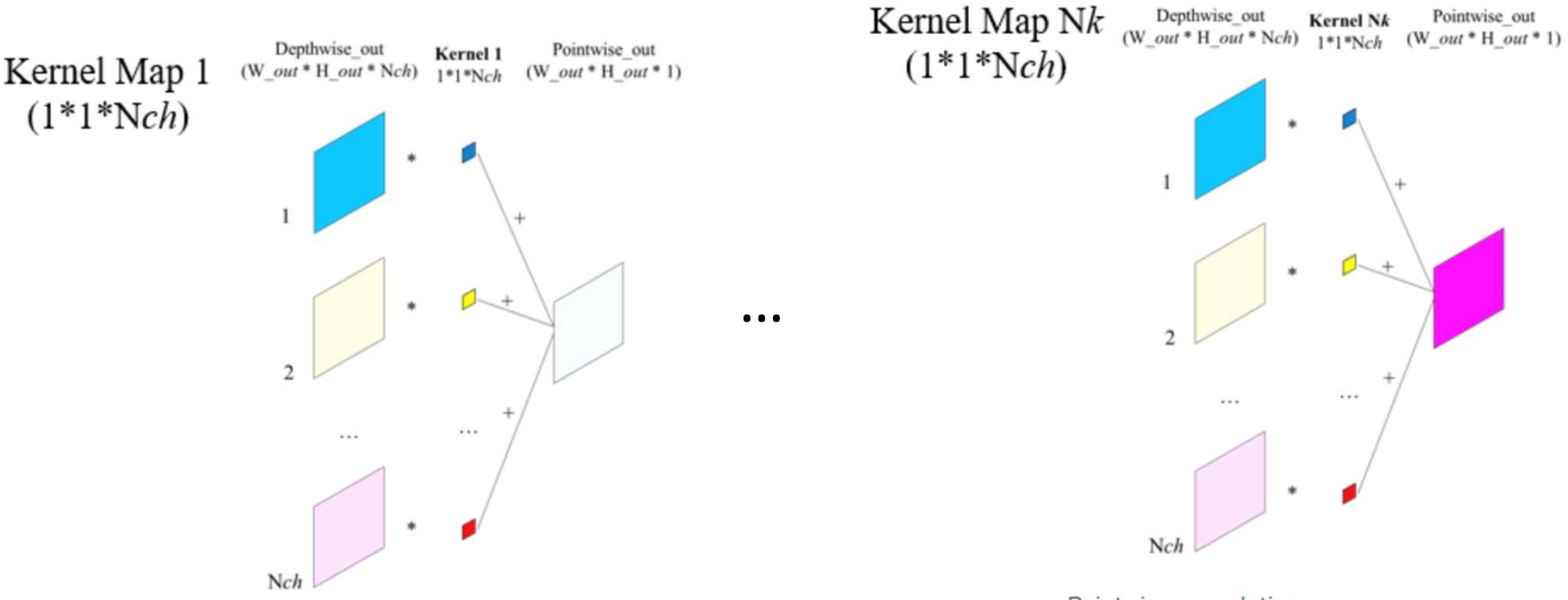
# Depthwise Convolution

- Each channel of inputs has a **k * k** kernel
- Separate the convolution of each channel
- **Difference:** Every kernel convolves with all channels in standard CONV



# Pointwise Convolution

- The number of kernel: Nk with (1 * 1 * Nch) size
- Do CONV on the outputs of depthwise convolution




Pointwise convolution

深度學習-MobileNet (Depthwise separable convolution) | by Tommy Huang | Medium

# Depthwise + Pointwise Convolution

**Depthwise convolution**

Input: W_in * H_in * Nch

Nch Kernel (k * k)

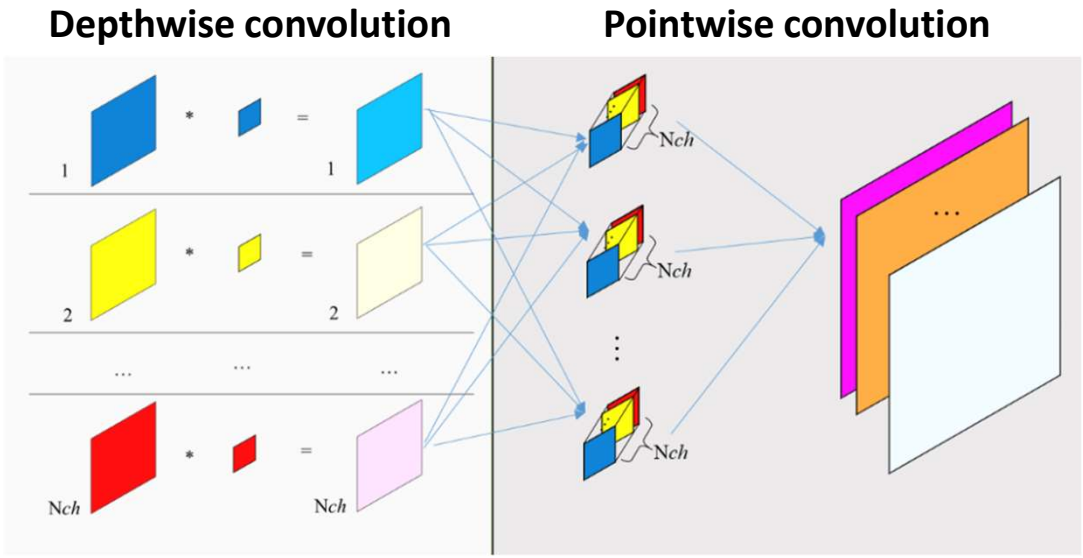
Output: W_out * H_out * Nch

**Pointwise convolution**

Input: W_out * H_out * Nch

Nk kernel = (1 * 1 * Nk)

Output = W_out * H_out * Nk




深度學習-MobileNet (Depthwise separable convolution) | by Tommy Huang | Medium

# Depthwise Separable Convolution

- **Standard CONV**
  - Input: W_in * H_in * Nch
  - Kernel: k * k * Nk
  - Output: W_out * H_out * Nk
  - Computation: W_in * H_in * Nch * k * k * Nk
- **Depthwise separable convolution**
  - Depthwise CONV computation: W_in * H_in * Nch * k * k
  - Pointwise CONV computation: Nch * Nk * W_in * H_in

$$\frac{\text{Depthwise separable CONV}}{\text{Standard CONV}} = \frac{\text{W\_in} * \text{H\_in} * \text{Nch} * k * k + \text{Nch} * \text{Nk} * \text{W\_in} * \text{H\_in}}{\text{W\_in} * \text{H\_in} * \text{Nch} * k * k * \text{Nk}}$$

$$= \frac{1}{\text{Nk}} + \frac{1}{K*k}$$

# Depthwise Separable Convolution

- Depthwise separable convolution can save more computation when
  - kernel size is large
  - The number of kernel is up
- Suppose input is 416 * 416 * 50, # of filter is 10, its size is 3 * 3.
- How much computation can be saved by depthwise separable convolution ?
  - 1/10 + 1/9 = 0.22

# Takeaway Questions

- What are problems in ultra-deep neural networks ?
  - (A) Over-fitting
  - (B) Gradient vanishing
  - (C) Low training accuracy
- Given a CNN model below, how many channels are in the second layer ?
  - (A) 4
  - (B) 8
  - (C) 16

| | Input size | # of filter | Filter size | # of channel |
|---|---|---|---|---|
| **Layer1** | 12 x 12 | 4 | 3x3 | 64 |
| **Layer2** | 12 x 12 | 16 | 3x3 | |

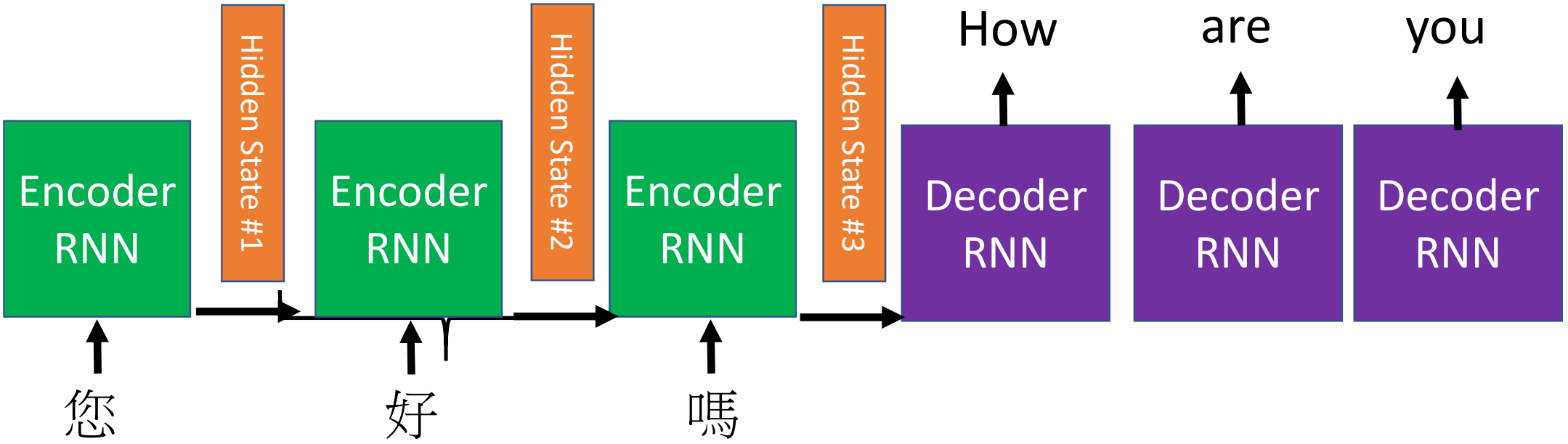
# Takeaway Questions

- A standard CONV layer
  - Input: W_in * H_in * Nch = (32 * 32 * 16)
  - Kernel: k * k * Nk = (3 * 3 * 8)
  - Computation: W_in * H_in * Nch * k * k * Nk = (32 * 32 * 16 * 3 * 3 * 8)
- What is the amount of computation that is carried out by **depthwise separable convolution ?**
  - (A) (32 * 32 * 16 * 3 * 3) + (3 * 8 * 8)
  - (B) (32 * 32 * 3 * 3 * 8) + (16 * 3 * 3)
  - (C) (32 * 32 * 16 * 3 * 3) + (16 * 8 * 32 * 32)

# Transformer

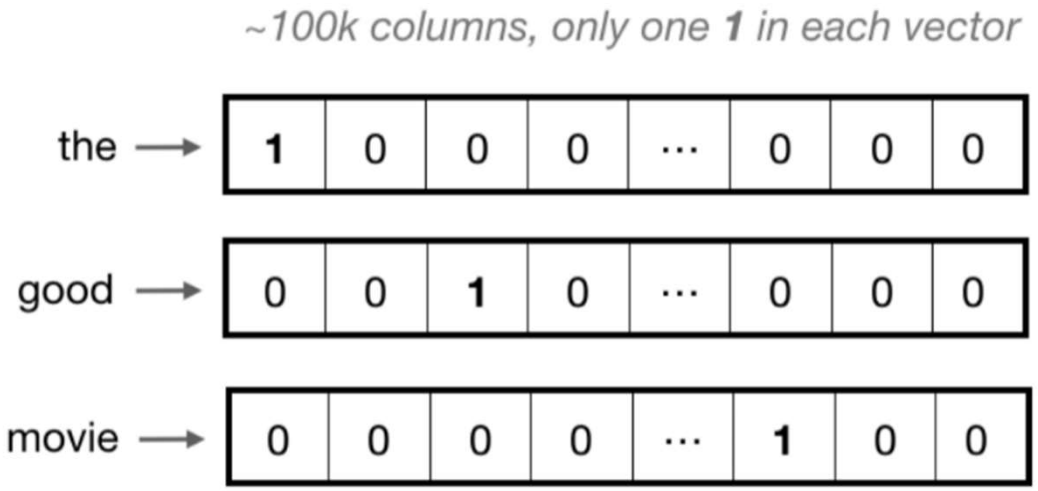
# Classical Sequence-to-Sequence Model

- Pass the last hidden state of the encoding stage
- Decoder uses this last hidden state to do the prediction



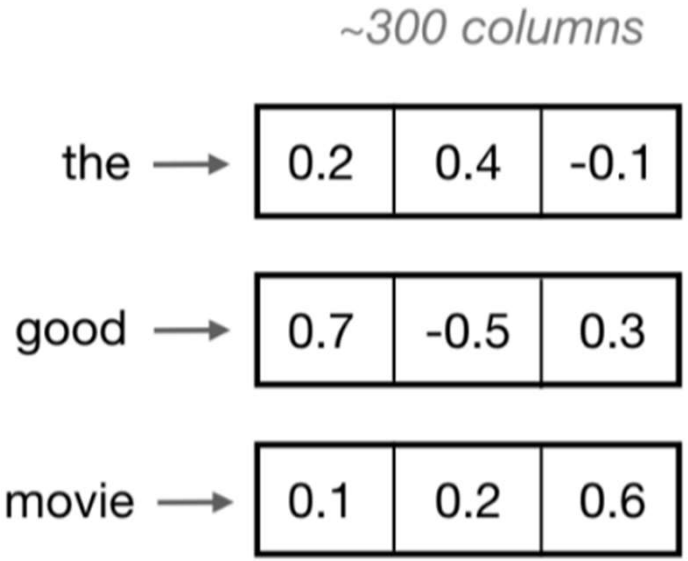
# Word Representation

- **One-Hot Encoding**
  - Representing each word as a vector that has as many values in it
  - Each column in a vector is one possible word in a vocabulary
  - Problem
    - In large vocabularies, these vectors can get very long
    - Contain all 0's except for one value
    - Sparse representation



# Word Representation

- **Word Embedding**
  - Map the word index to a continuous word embedding through a look-up table
  - Popular pre-trained word embeddings
    - Word2Vec, GloVe



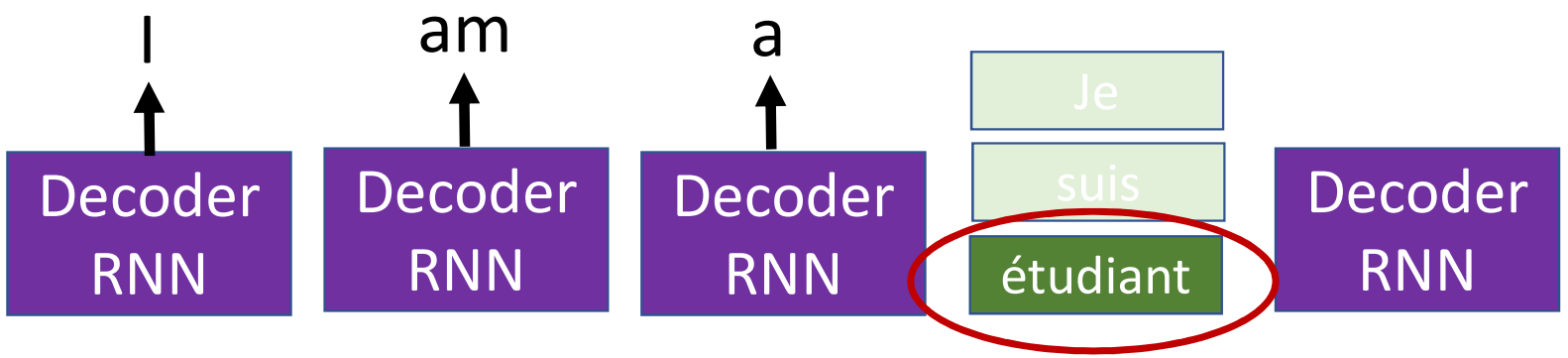
# Position Embedding (PE)

- **Position embedding (PE)**
  - Information to each word about its position in the sentence
  - **Unique** encoding for each word's position in a sentence
  - Distance between any two positions is **consistent** across sentences with different lengths
  - Encode words by using **sin(), cos()** with different frequencies
  - **Deterministic** and **generalize** to longer sentences

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases} \qquad \omega_k = \frac{1}{10000^{2k/d}}$$

# Bottleneck of Sequence-to-Sequence Model

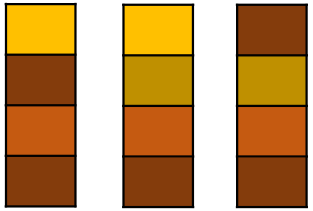
- It is challenging for the model to deal with long sentences
- **Attention**
  - The encoder passes all the hidden states to the decoder
  - The attention enables the decoder to focus on the word before it generates the English translation
  - This ability amplifies the signal from the relevant part of the input sentence

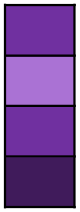


# How does the Attention Work ?

**Attention at time step 4**

1. Encoder hidden states



Decoder hidden state at time step 4 (h4)

2. Score each hidden state

| 13 | 9 | 9 |
|---|---|---|

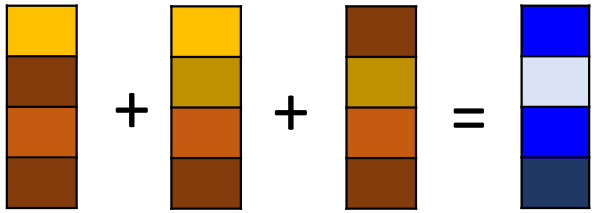
**Scores**: Attention weights for decode time step #4

3. Softmax scores

| 0.96 | 0.02 | 0.02 |
|---|---|---|

**Softmax score**

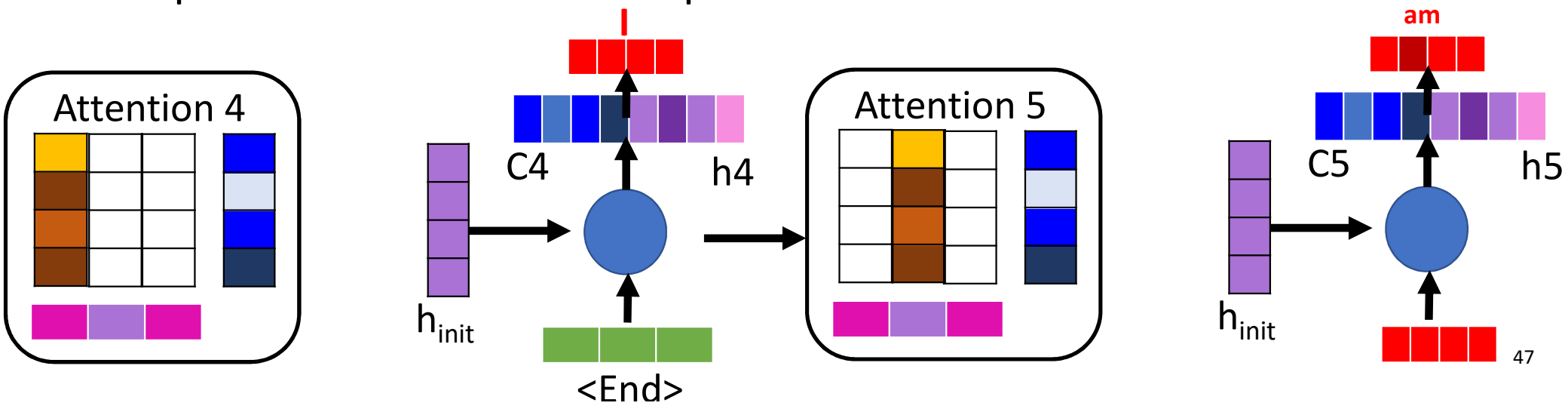
4. Multiply each vector by its softmaxed score




5. Sum up the weighted vectors to get the **context vector** (C4) for decoder time step #4

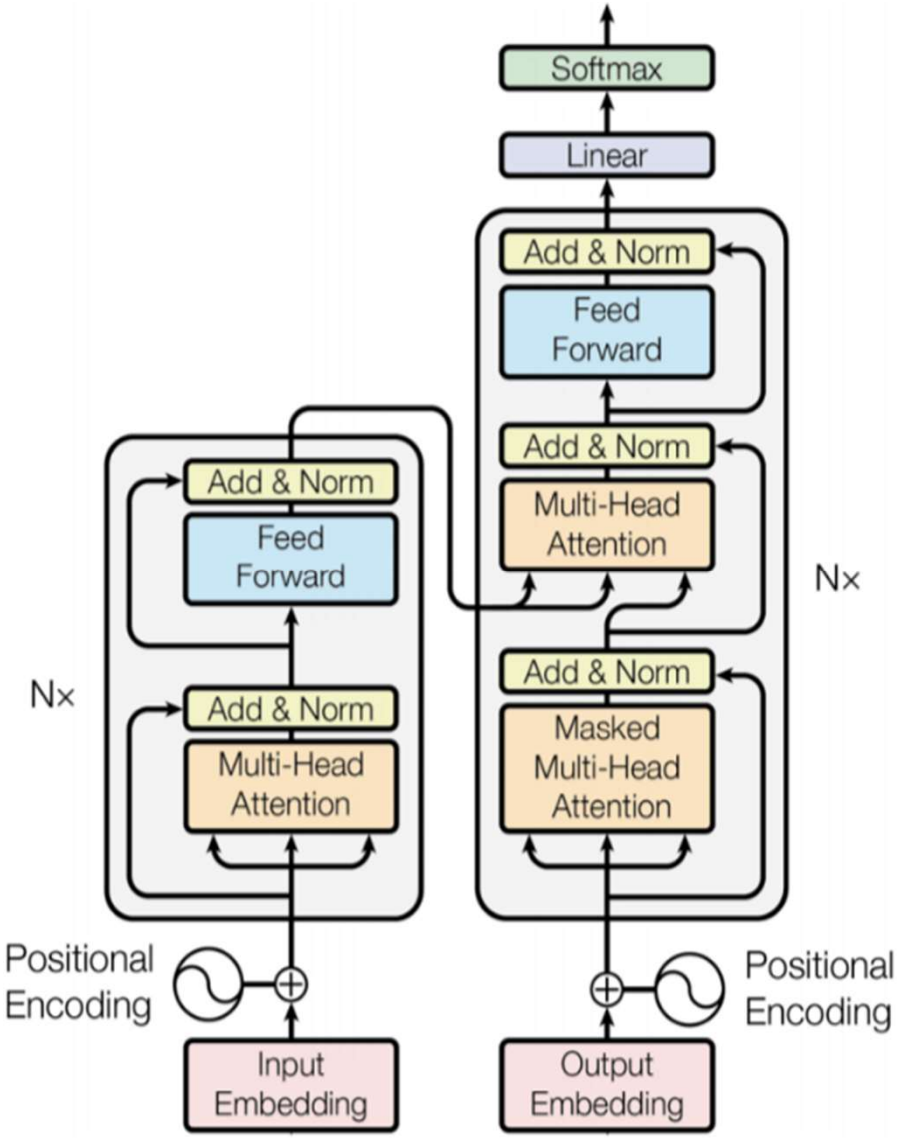
# How does the Attention Work ?

- The attention decoder takes the embedding of the <END> token, and an initial decoder hidden state
- Concatenate h4 and C4 into one vector
- The output of the feedforward neural network indicates the output word of this time step
- Repeat for the next time steps



# Transformer

- Each **encoder** block has two sub-layers
  - Multi-head self-attention
  - A position-wise fully connected feed-forward
- Each **decoder** block has an additional third sub-layer
  - The third is a masked multi-head attention over the output of the encoder stack
- A **residual connection** is added around each of the two sub-layers
- The decoder yields the output sequence of symbols one element at a time

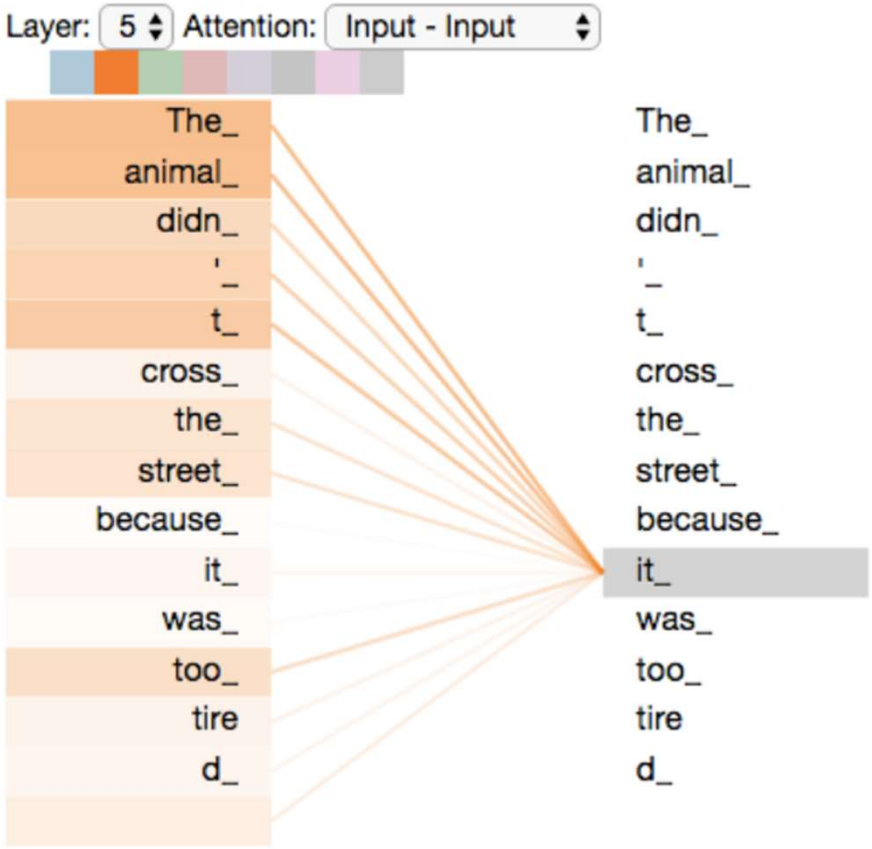



https://web.stanford.edu/class/cs224n/slides/cs224n-2020-lecture10-QA.pdf

Vaswani et al. 2017 https://arxiv.org/pdf/1706.03762.pdf

# Self-Attention

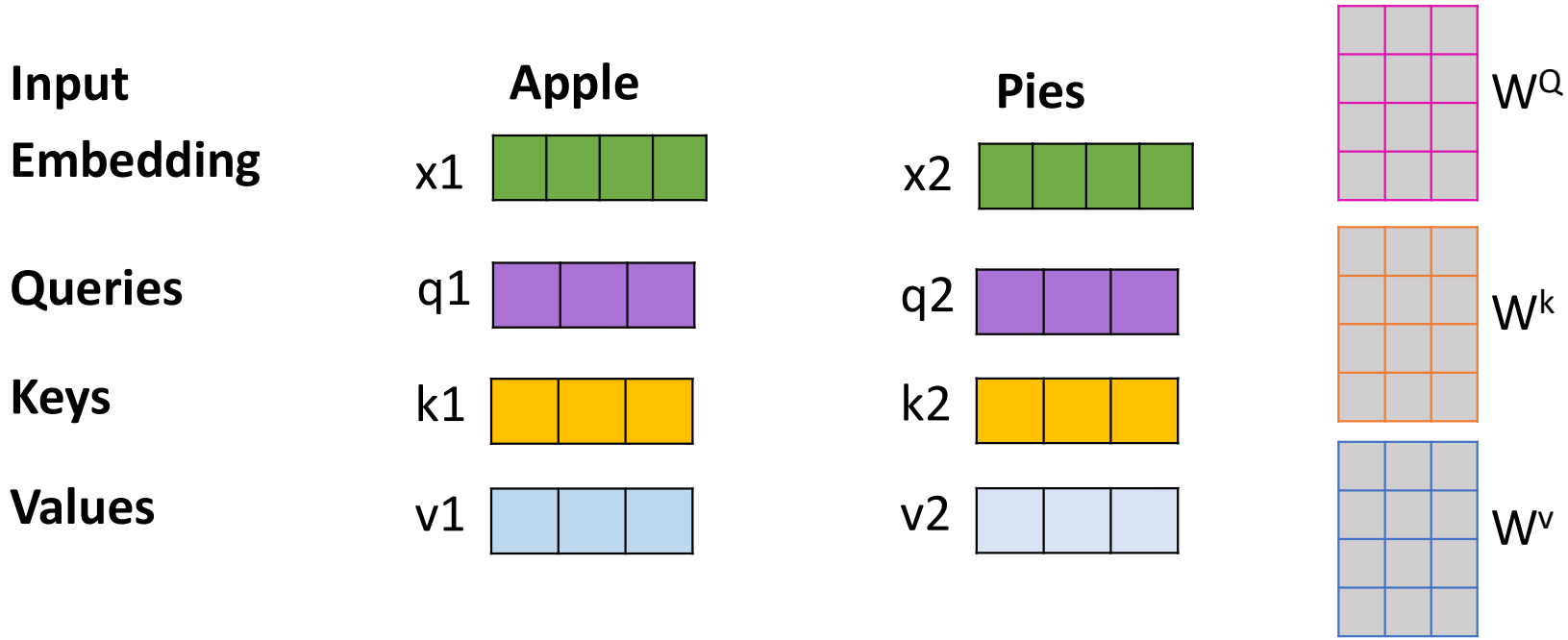
- **Self-attention**
  - The Transformer is used to understand other relevant words into the one we are currently processing
  - For example, “The animal didn’t cross the street because it was too tired”.
  - How do we know “it” that refers to the street or the animal ?




http://jalammar.github.io/illustrated-transformer/

# Self-Attention in Detail

- Multiplying input embedding (x1) by the $W^Q$ weight matrix produces **query vector** (q1) associated with that word
- A **"key" and a "value"** vector projects each word in input sentence



# Self-Attention in Detail

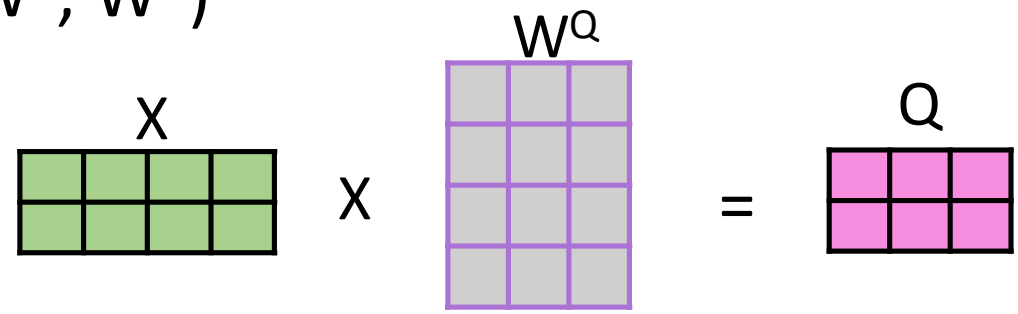
- **Calculating self-attention score**
  - Need to score each word of the input sentence against this word
  - The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position

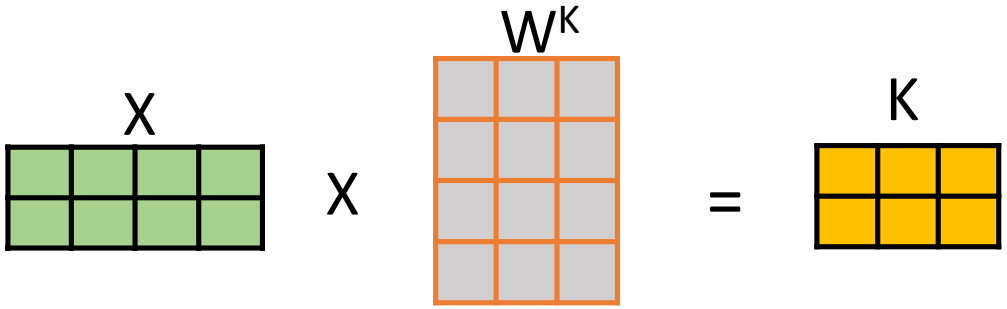

**Processing the word in position #1**

1. The first score is the dot product of q1 and k1, the second one is (q1, k2)
2. Normalized these scores by using softmax function
3. Multiplying each value vector by the softmax score – weighted value vectors
4. Summing up these weighted value vectors

# Matrix Calculation on Self-Attention

- Packing word embeddings into a matrix X
- Multiplying X by the weight matrices we have trained ($W^Q$, $W^K$, $W^V$)



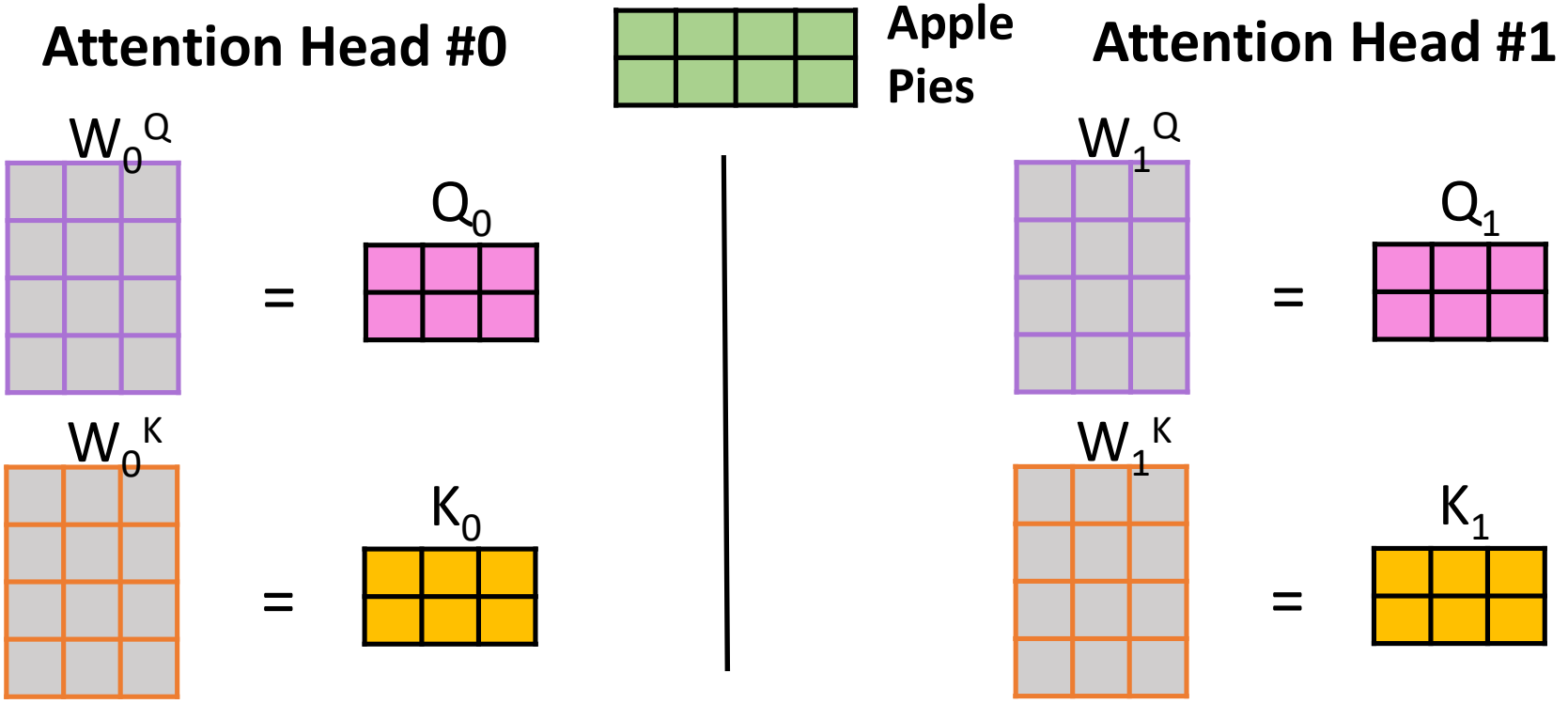




Self-attention score (Z) = softmax(Q x $K^T$/sqrt($d_k$)) V

$d_k$ is the dimension of the key vectors

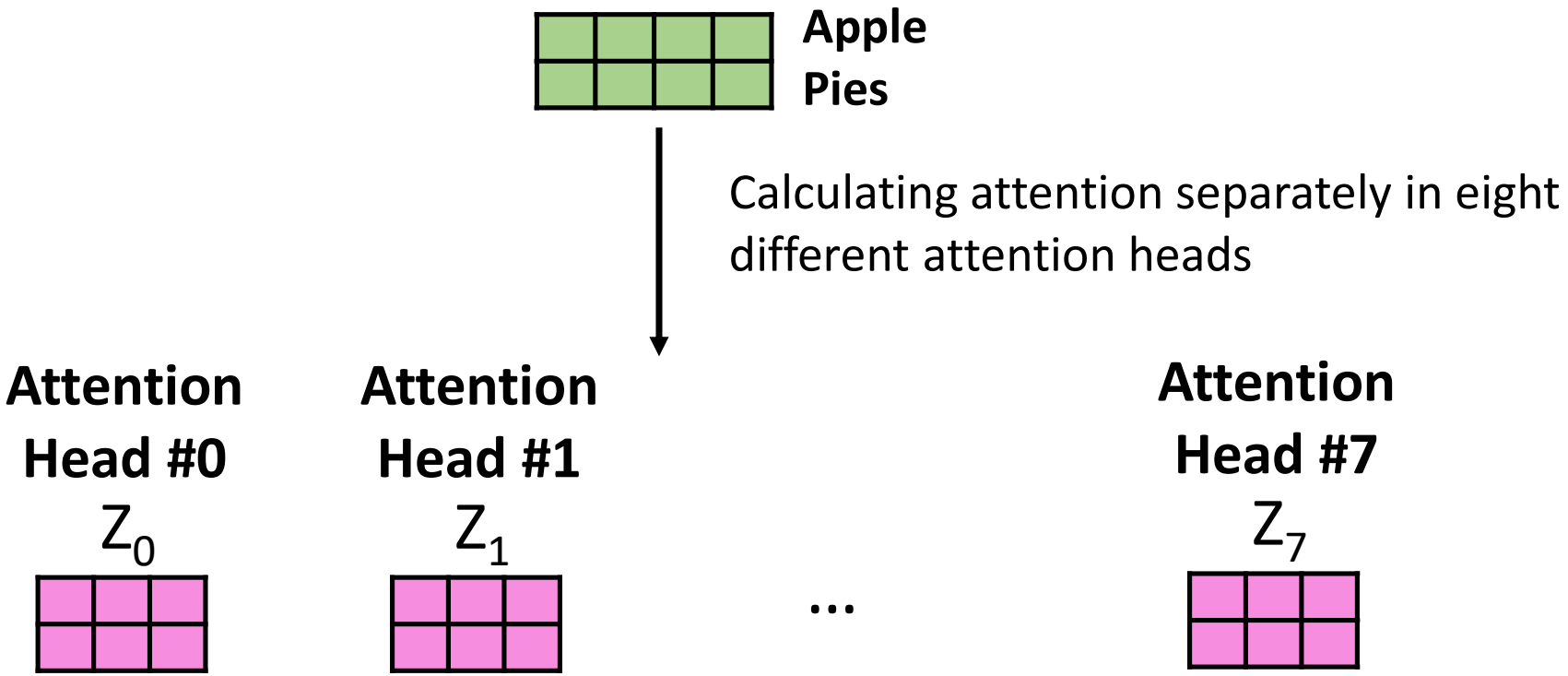
# Multi-Head Attention

- It expands the model's ability to focus on different position
- It gives the attention layer multiple "representation sets"



# Multi-Head Attention

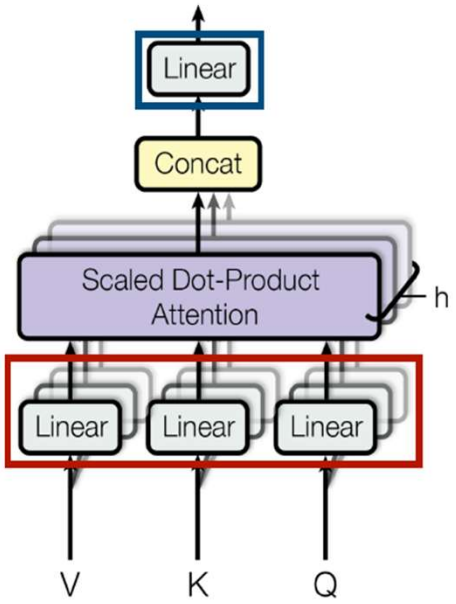
- We will produce multiple different weight matrices

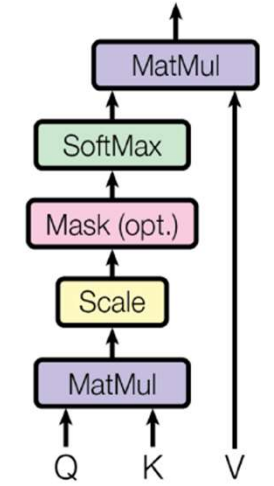


# Multi-Head Self-Attention (MHSA)

- **Project** Q, K, and V with h **different** learned linear projections
- Perform the **scaled dot-product attention** function on each of Q, K, V **in parallel**
- **Concatenate** the output values
- **Project** the output values again, resulting in the final values

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



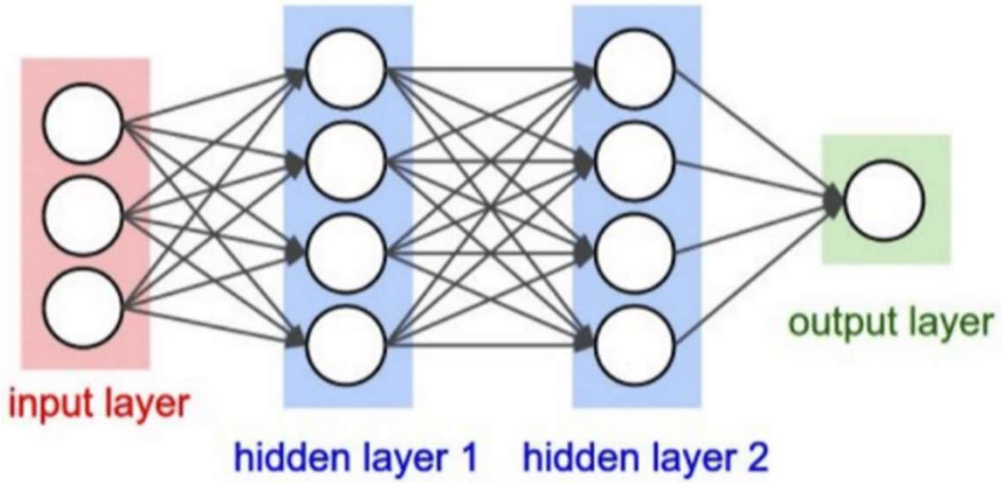




$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

# Feed-Forward Network (FFN)

- **FFN** is applied to each position separately and identically
  - Two linear transformations with a ReLU activation in between
  - The middle hidden size is usually larger than the output size

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



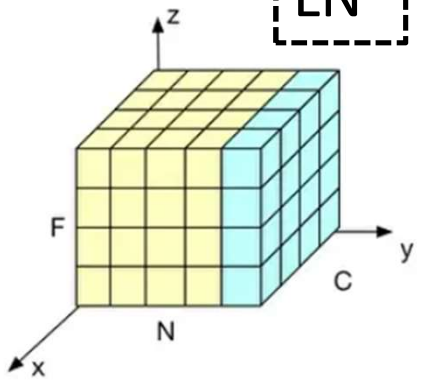
# Layer Normalization (LN)

- The small amount of sampling degrades the result of batch normalization (BN)
- LN calculation isn't related to the amount of sampling

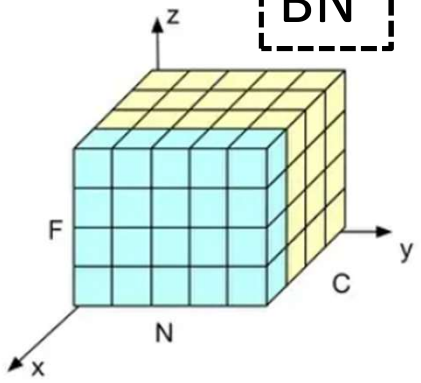
$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2}$$

H: the number of hidden layer
l: the number of MLP layer

LN



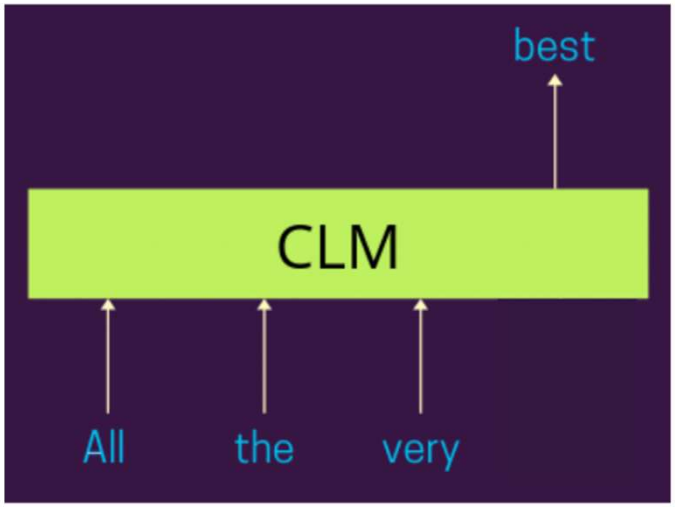

BN



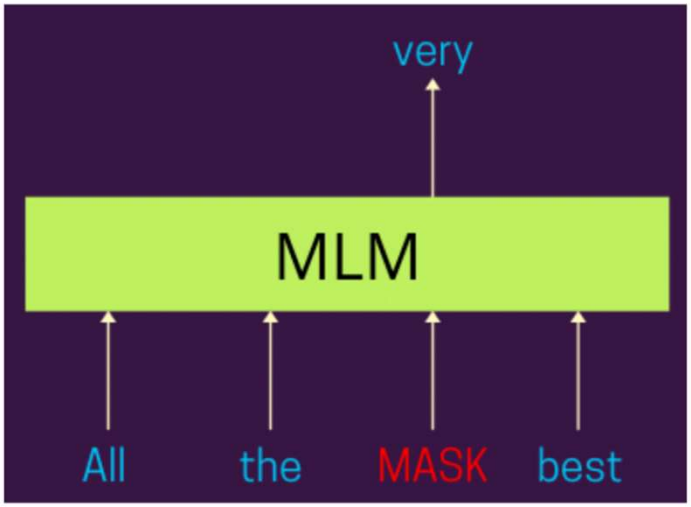

N: Sampling
C: Channel
F: The number of features in a channel

# Language Model

- Trained on large amount of unlabeled text
- Then, being fine-tuned on specific NLP tasks (such as machine translation …)




**Causal Language Models (CLM)**
(Unidirectional, from left to right)




**Masked Language Models (MLM)**
(Bidirectional, from both left and right)

Content credit: https://towardsdatascience.com/understanding-masked-language-models-mlm-and-causal-language-models-clm-in-nlp-194c15f56a5

# Casual Language Models (CLM) - GPT

- **Pre-train** generates text that is similar to the input text data in an unsupervised manner (language model)

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta)$$

- **Fine-tune** this model on much smaller supervised datasets to solve specific tasks

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y | x^1, \ldots, x^m)$$

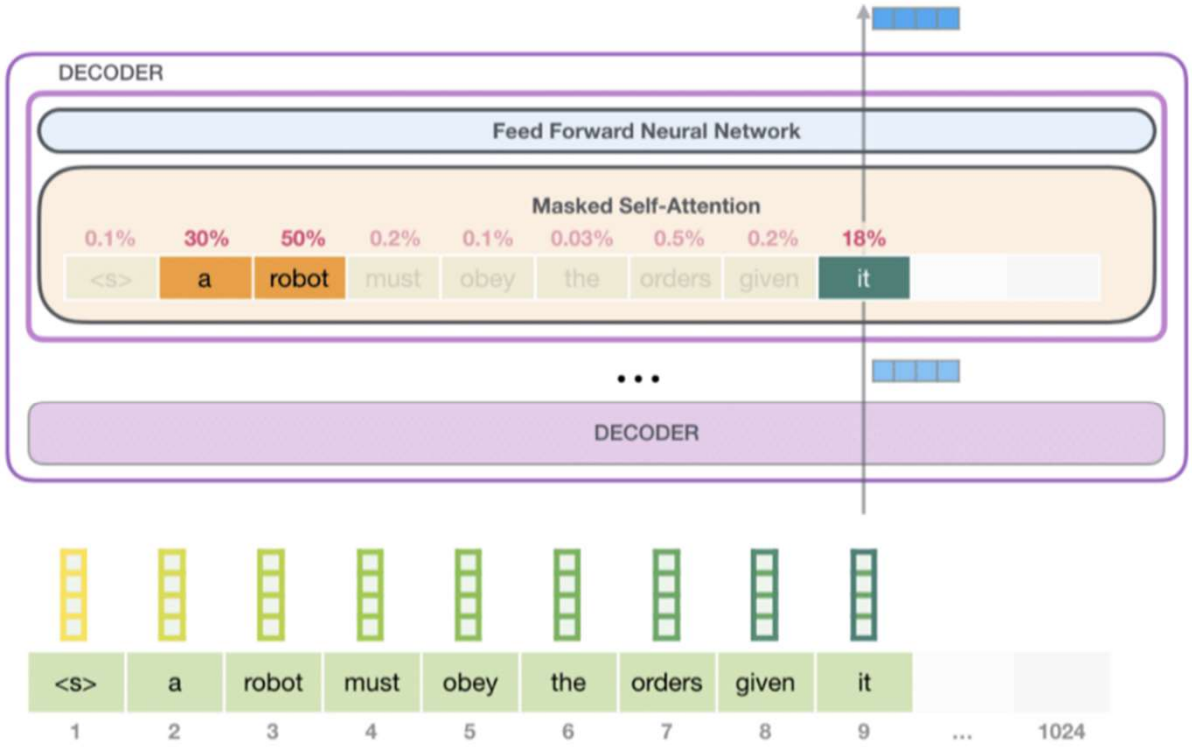



Image credit: https://jalammar.github.io/illustrated-gpt2/

# Masked Language Models (MLM) - BERT

- **#1: Masked Language Model (MLM)**
  - Mask percentage (15%) of the input tokens at random
  - Predict those masked tokens
- **#2: Next Sentence Prediction (NSP)**
  - Predict if sentence A follows sentence B make sense
  - Feeding BERT the words "sentence A" and "sentence B" twice.
  - Q: Does sentence B follow sentence A?
  - True Pair or False Pair is what BERT responds

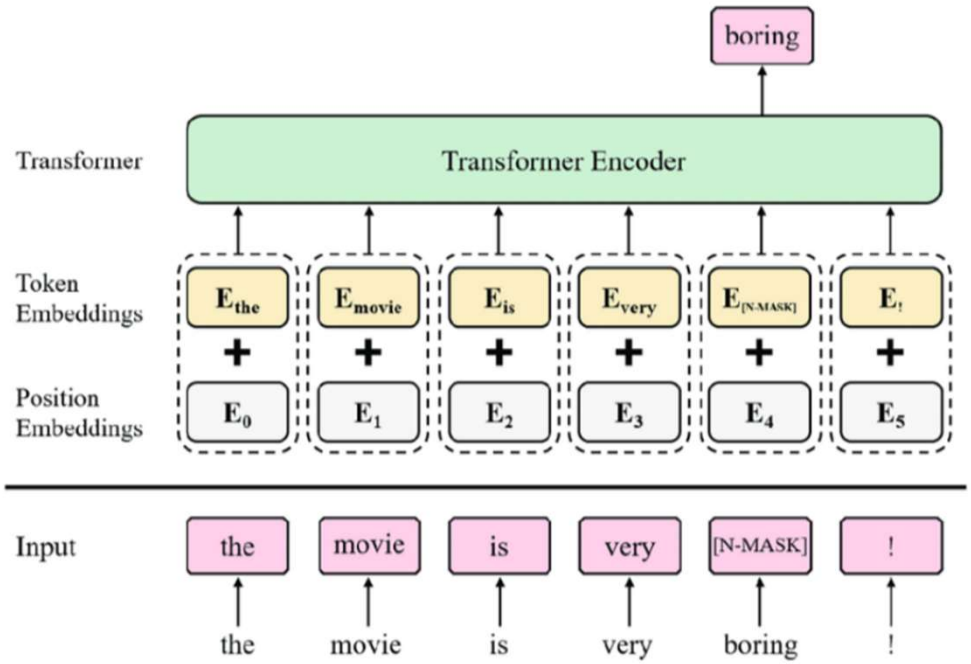
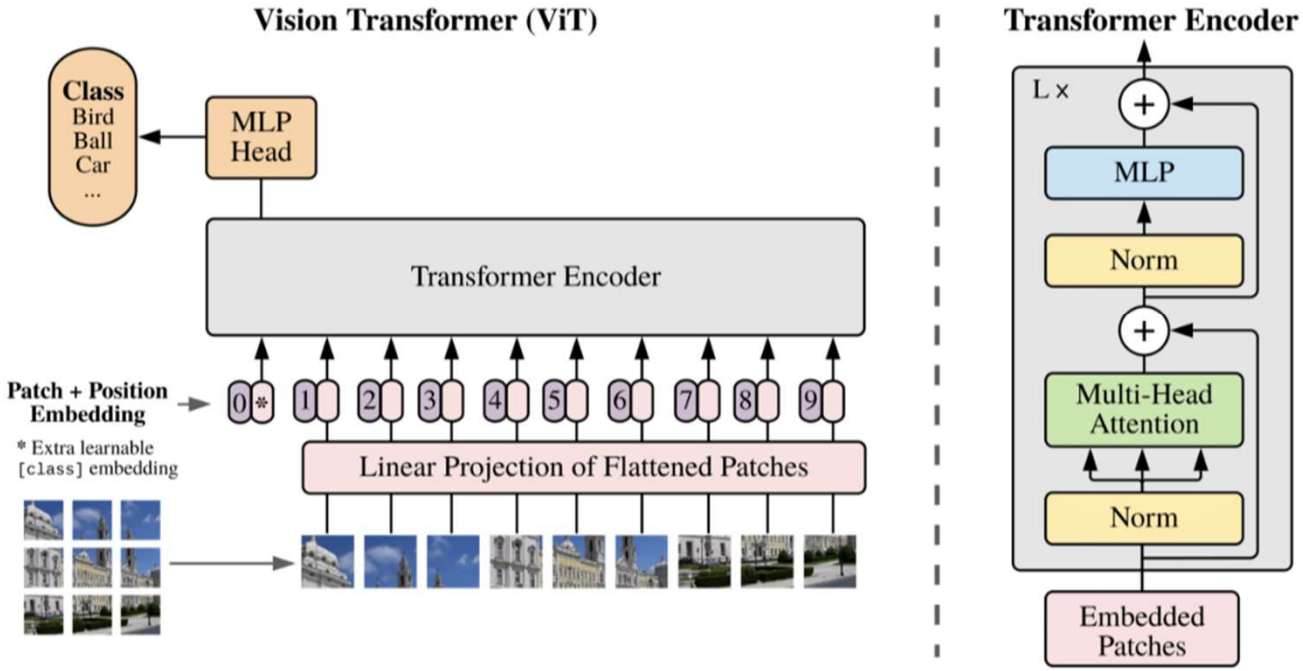



Image credit: https://www.mdpi.com/2073-8994/11/11/1393/htm

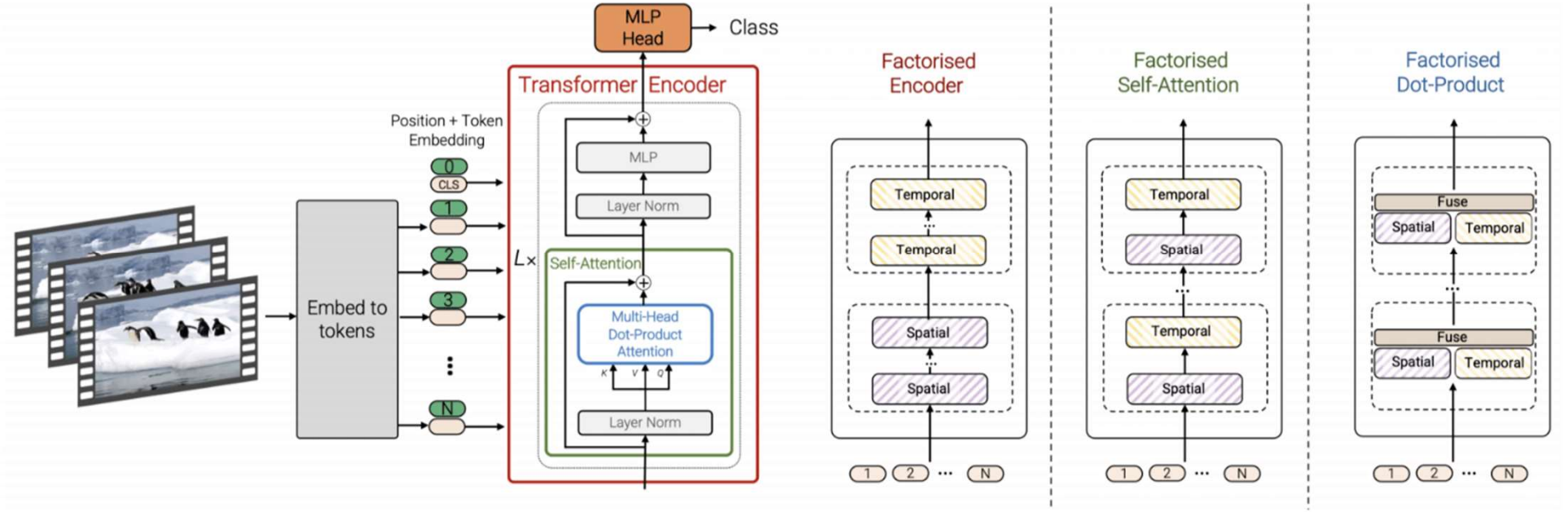
# Image Transformer -- ViT

- Split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder



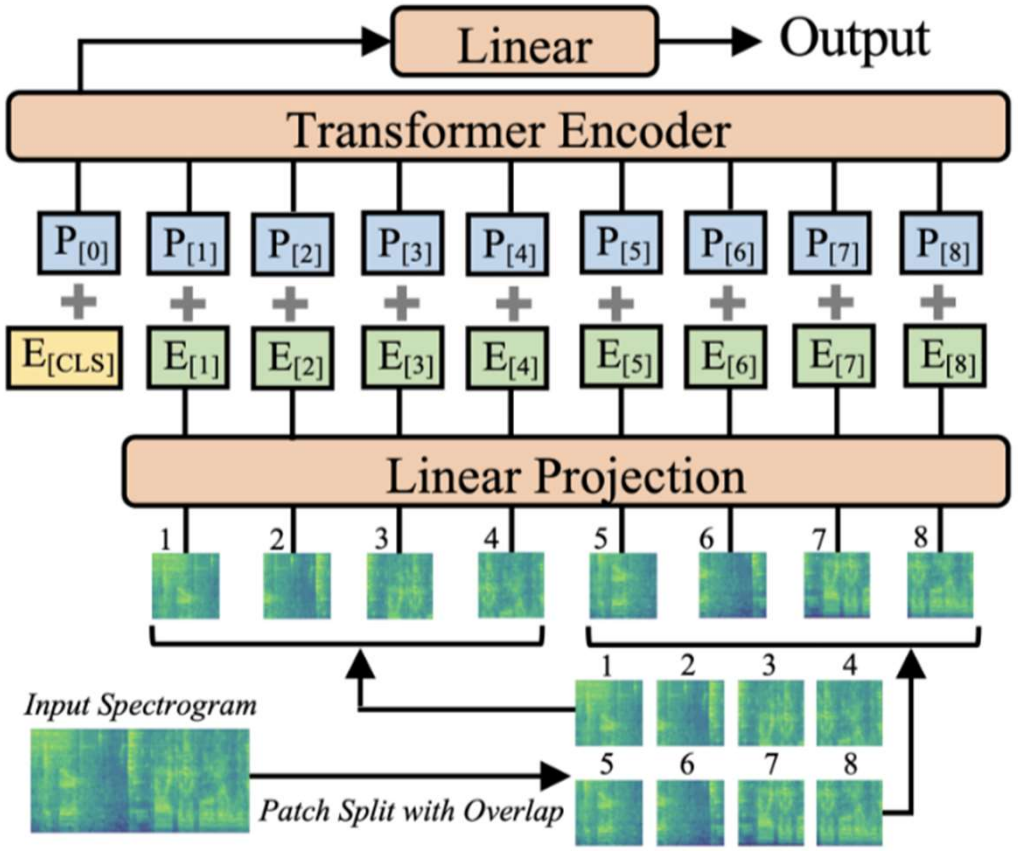
# Video Transformer -- ViViT

- Extract spatiotemporal tokens from the video, then encoded by a series of transformer layers
- Factorize the spatial- and temporal-dimensions of the input to handle the long sequences



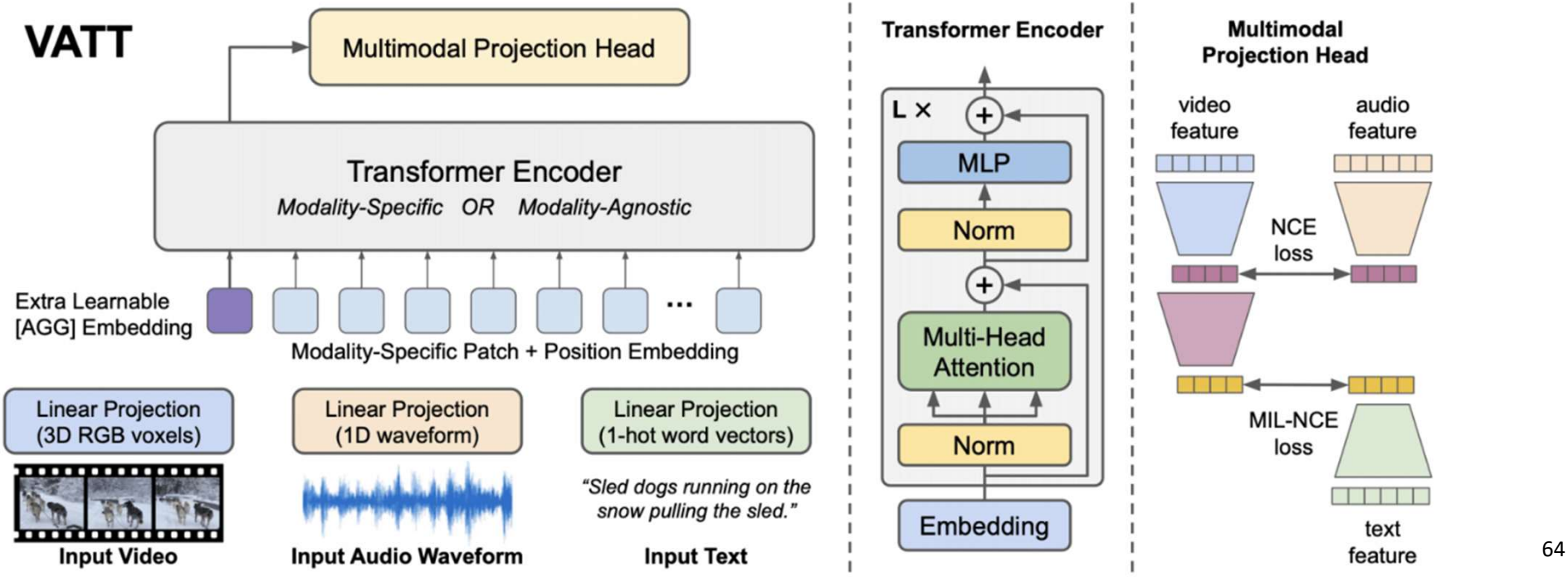
# Audio Transformer -- AST

- Split the 2D audio spectrogram a sequence of 16 x 16 patches with overlap
- Linearly projected to a sequence of 1-D patch embeddings



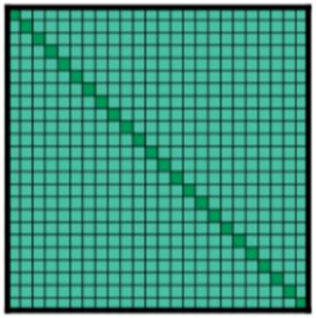
# Multi-Modal Transformer -- VATT

- Linearly project each modality into a feature vector and feed it into a Transformer encoder

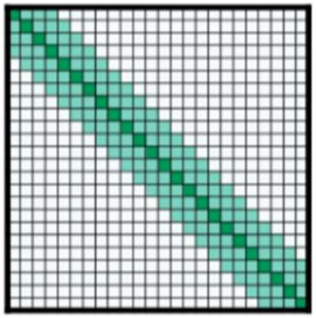


# Sparse Attention -- LongFormer

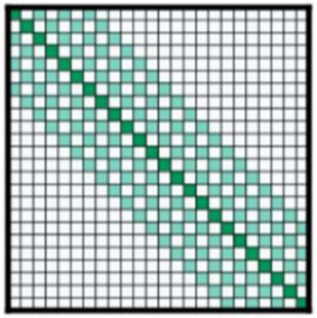
- Attention with **sliding window**
  - A fixed-size window attention surrounding each token
  - The complexity is reduced from $O(N^2)$ to **O(N x W),** W is the window size
- Attention with **dilated sliding window**
  - Dilate the sliding window with gaps of size dilation D
  - The receptive field is enlarge from W to **W x D**, with the same complexity



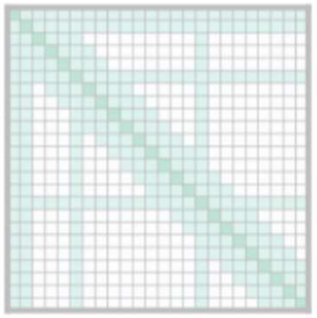
(a) Full $n^2$ attention



(b) Sliding window attention



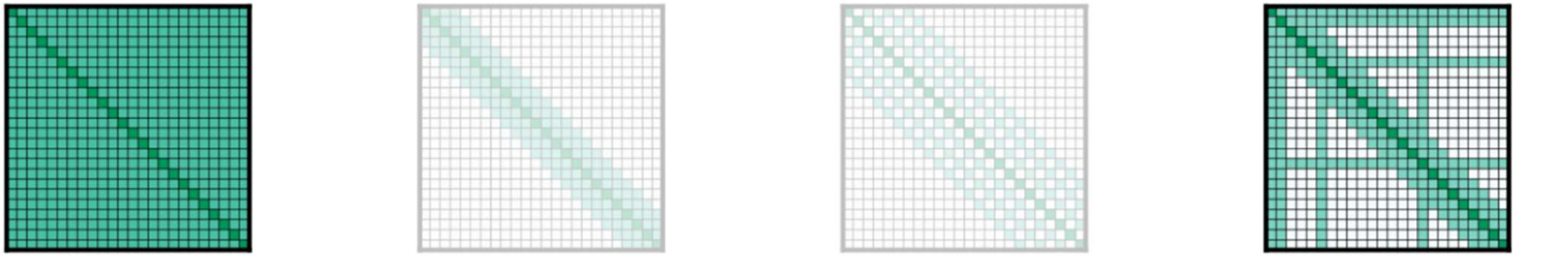
(c) Dilated sliding window



(d) Global+sliding window

# Sparse Attention -- LongFormer

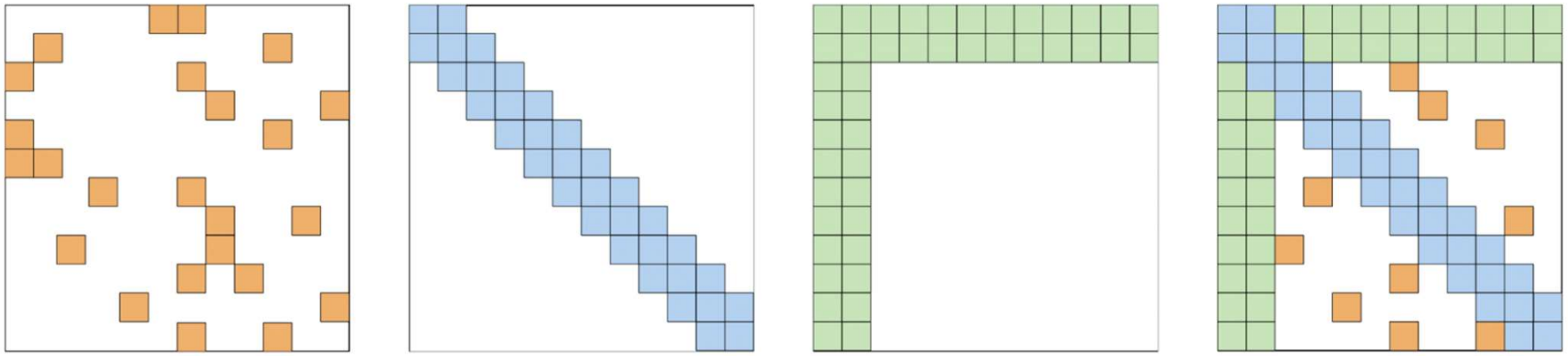
- **Global attention** added on a few pre-selected input locations
  - Classification: The special token ([CLS]), aggregating the whole sequence
  - QA: All question tokens, allowing the model to compare the question with the document



(a) Full $n^2$ attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

# Sparse Attention – Big Bird

- **Random sparse attention**
  - Each query attends over r random number of keys: i.e. A(I, .) for r randomly chosen keys
- **Big Bird**
  - Random Attention + local attention + global attention



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

# Takeaway Questions

- What's problem the "Attention" aiming to solve?
  - (A) Gradient vanishing
  - (B) Over-fitting
  - (C) Message passing in the long sequence of data
- How does the "Attention" work on sequence-to-sequence model?
  - (A) Memory gate
  - (B) Context vector
  - (C) Bi-directional network

# Takeaway Questions

- What are benefits of the “Transformer” ?
  - (A) Large hidden layer
  - (B) The amount of computation is small
  - (C) More data parallelism
- How does the “self-attention” help the encoder?
  - (A) Looking at other works in the input sentence
  - (B) Memorizing the more messages within a network
  - (C) Focus on a specific word