AI6126 Advanced Computer Vision

Last update: 28 January 2022 12:20pm

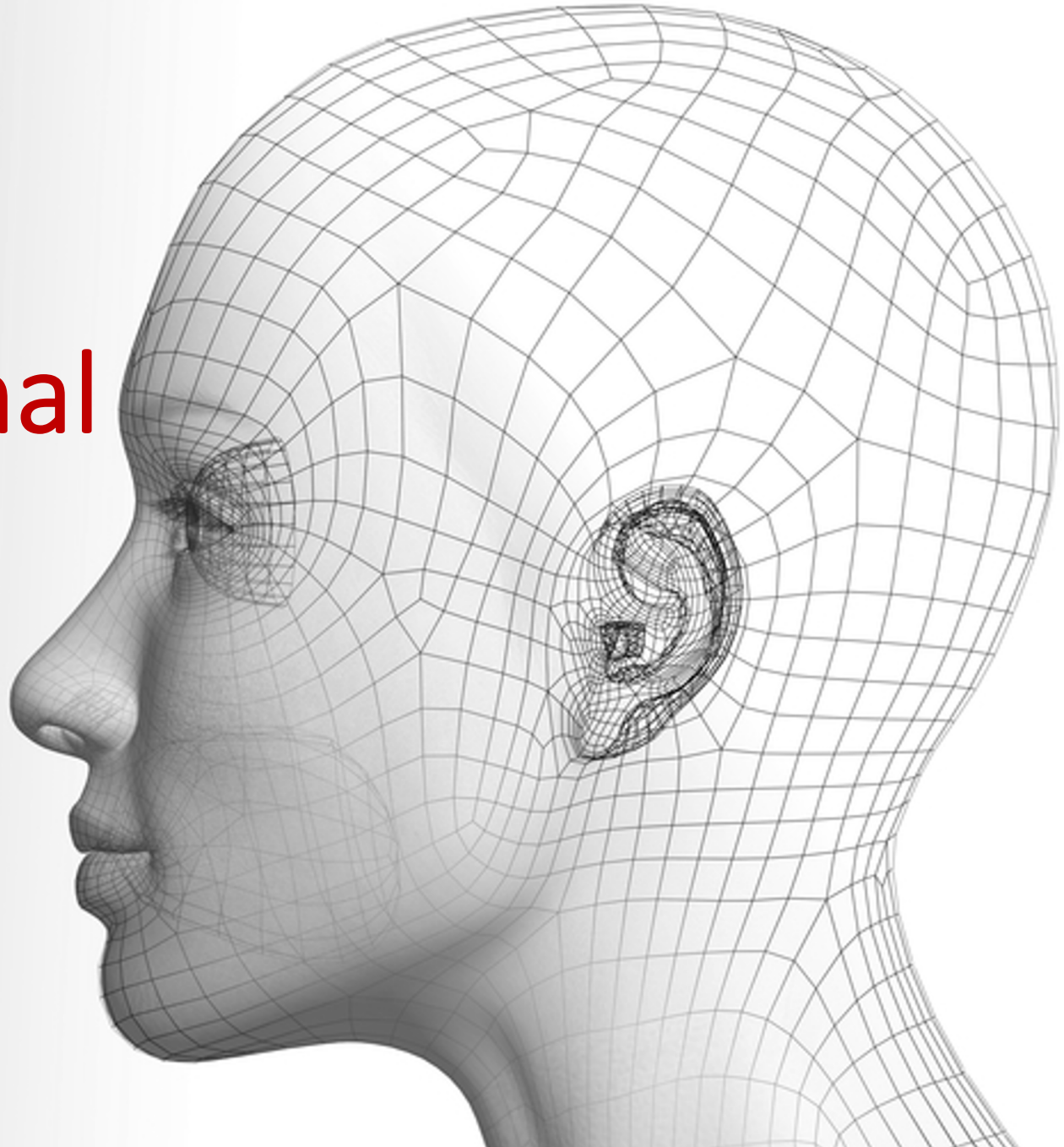
Training Convolutional Neural Networks

Chen-Change Loy

吕健勤

http://www.ntu.edu.sg/home/ccloy/

https://twitter.com/ccloy



Course outline

Week	Date	Topic	Lecturer
1	14 Jan	Introduction	Loy Chen Change
2	21 Jan	Convolutional Neural Networks	Loy Chen Change
3	28 Jan	Training Convolutional Neural Networks	Loy Chen Change
4	4 Feb	Object Detection	Loy Chen Change
5	11 Feb	Image Segmentation	Liu Ziwei
6	18 Feb	Transformers	Loy Chen Change
7	25 Feb	Autoencoder	Liu Ziwei
Recess Week			
8	11 Mar	Generative Adversarial Networks	Liu Ziwei
9	18 Mar	Image Super-Resolution	Loy Chen Change
10	25 Mar	Image Editing	Loy Chen Change
11	1 Apr	Open-World Visual Recognition	Guest Speaker
12	8 Apr	Unsupervised Representation Learning	Loy Chen Change
13	14 Apr	Creating Small and Efficient Networks	Loy Chen Change

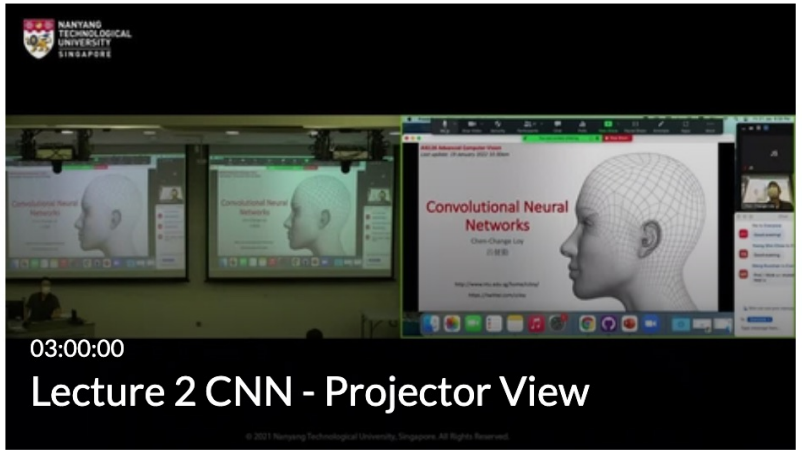
Recorded Lectures/Tutorials

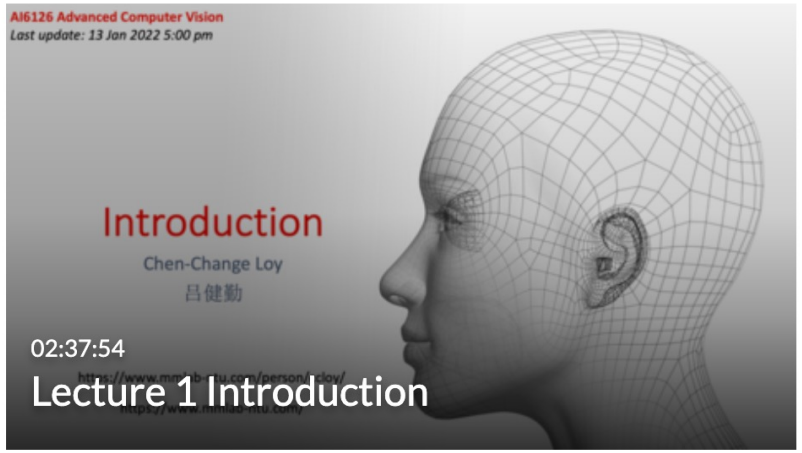
Media Gallery

3 Media

Search this gallery

Filters > | Search In: All Fields | Sort By: Creation Date - Descending | + Add Media

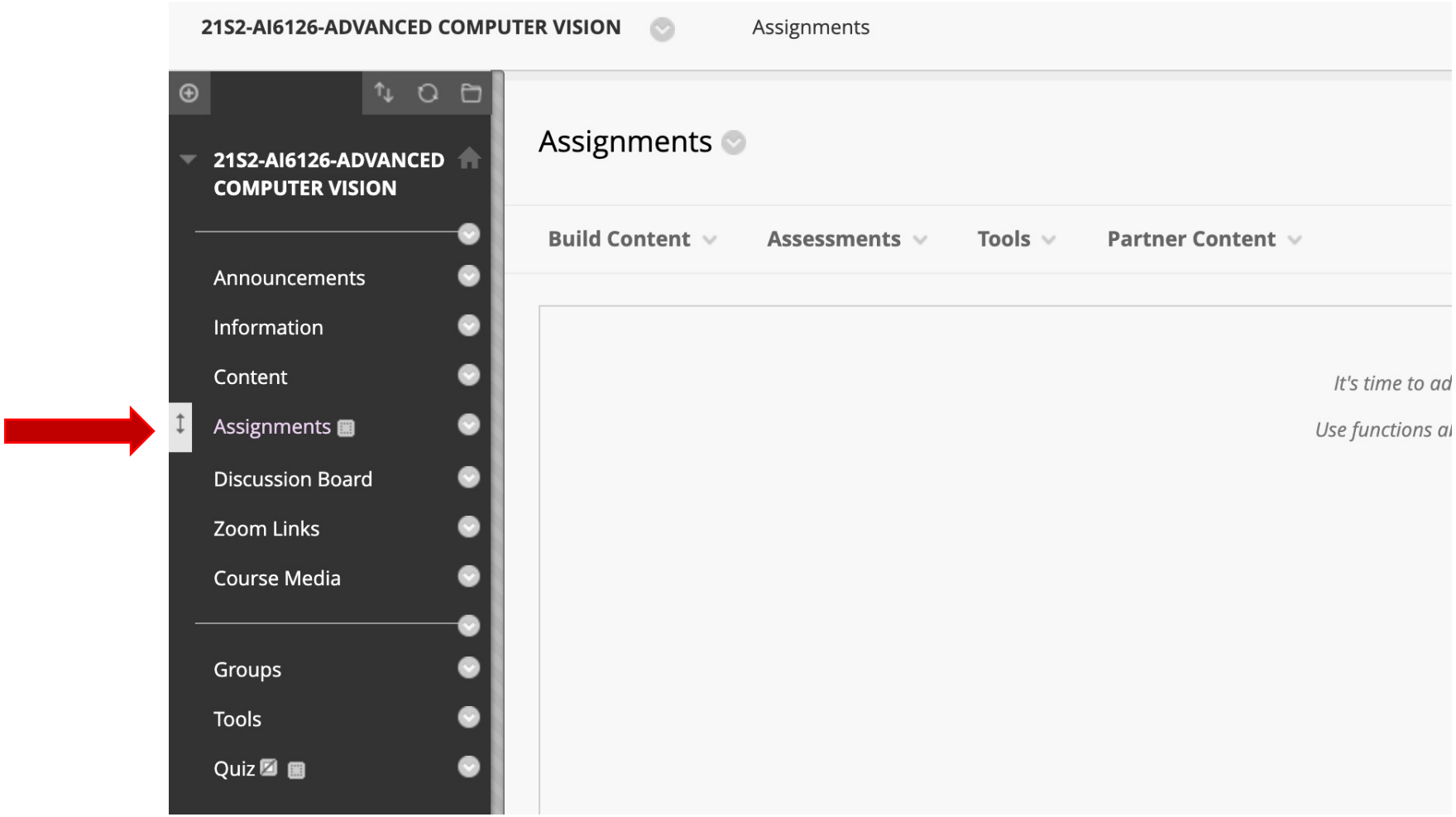




Load More

Assignment 1

- Assignment 1 will be released next week covering the first three lessons.



Quiz

- Date: 29 April (Friday)
- Time: 7:30 PM – 8:30 PM
- Venue: Online at NTULearn

- This quiz will be allocated 60 minutes and it is an open-book quiz, during which you will be required to answer 3 text-entry questions and 20 multiple-choice questions. There is no restriction on the use of writing material, printed notes or the Internet. But you are not allowed to communicate with other people via any means during the quiz.

  The questions will be based on all lectures.

  All questions will add up to 20 marks (0.5 mark for each question in MCQ) and 10 marks for questions that require text-based answers. The individual text-based questions may be of different marks as indicated in the quiz.

Feedback

Dear Student,

The Online Faculty Teaching Evaluation for Semester 2 AY21-22 is now open.

Please click here to provide the feedback. The link is valid from 24-JAN-2022 to 06-FEB-2022.

We would like to strongly encourage you to participate in this exercise. Giving teaching feedback to the School will benefit the student population as we work towards refining and improving our teaching pedagogy and course materials. Together we can develop a more nurturing and effective learning environment for every student.

Thank you and with Best Regards,

SFT Administrator

This is a computer generated letter. No signature is required.

Outline

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution

You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network

- Training basics
  - Weight initialization
  - Optimizers
  - Prevent overfitting

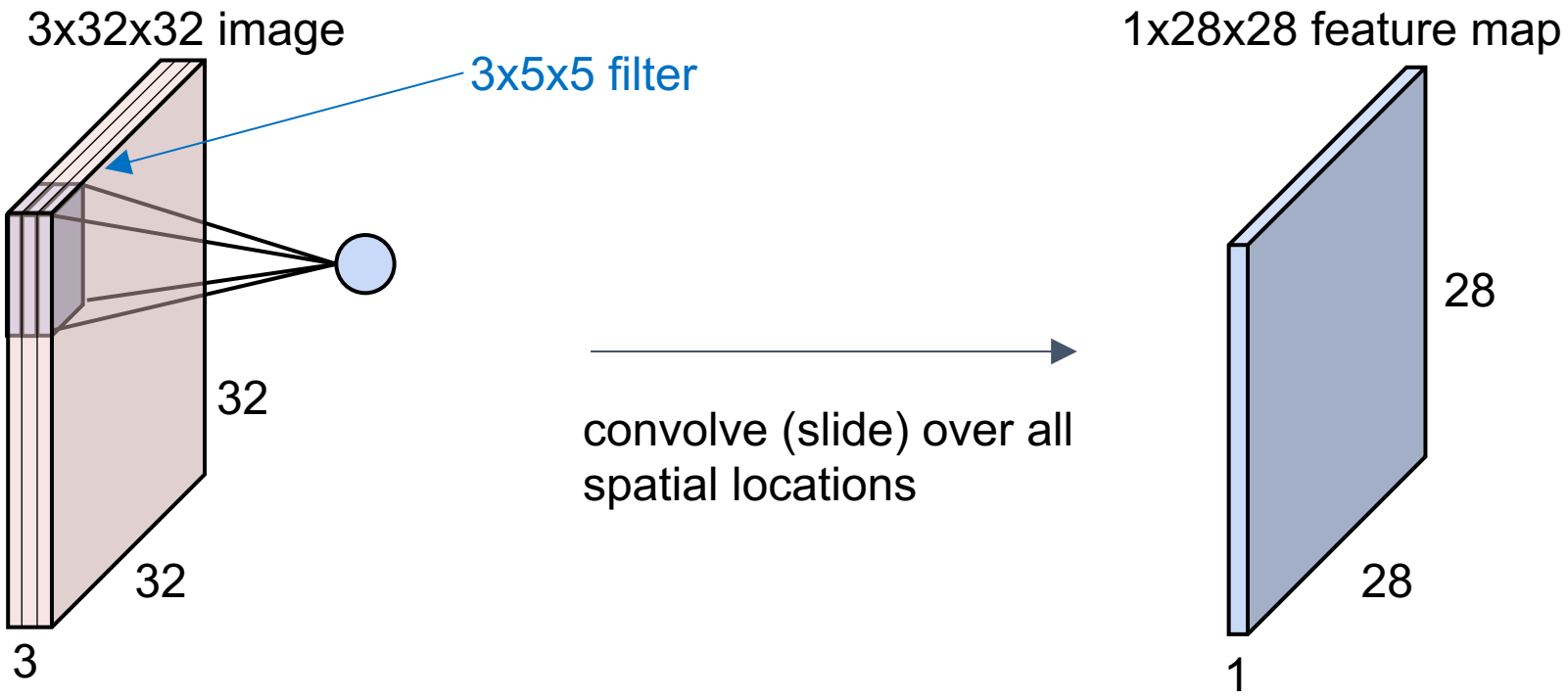
You learn two important techniques to prevent overfitting in neural networks

- Batch normalization

You learn an important technique to improve the training of modern neural networks

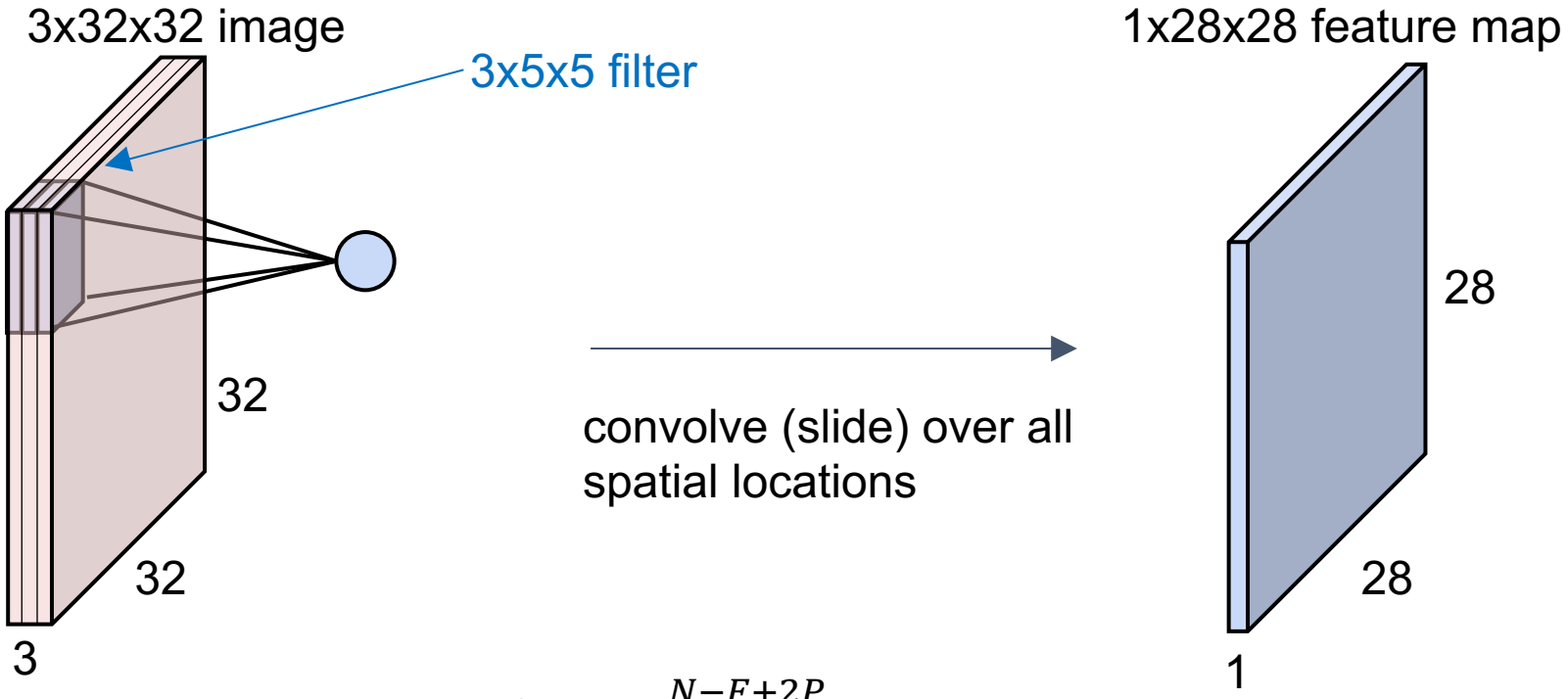
More on Convolution

Standard convolution



- 😀 How to calculate the spatial size of output? Lecture 2
- 😀 How to calculate the number of parameters? Lecture 2
- 😅 How to calculate the computations involved?

Recap: How to calculate the spatial size of output?

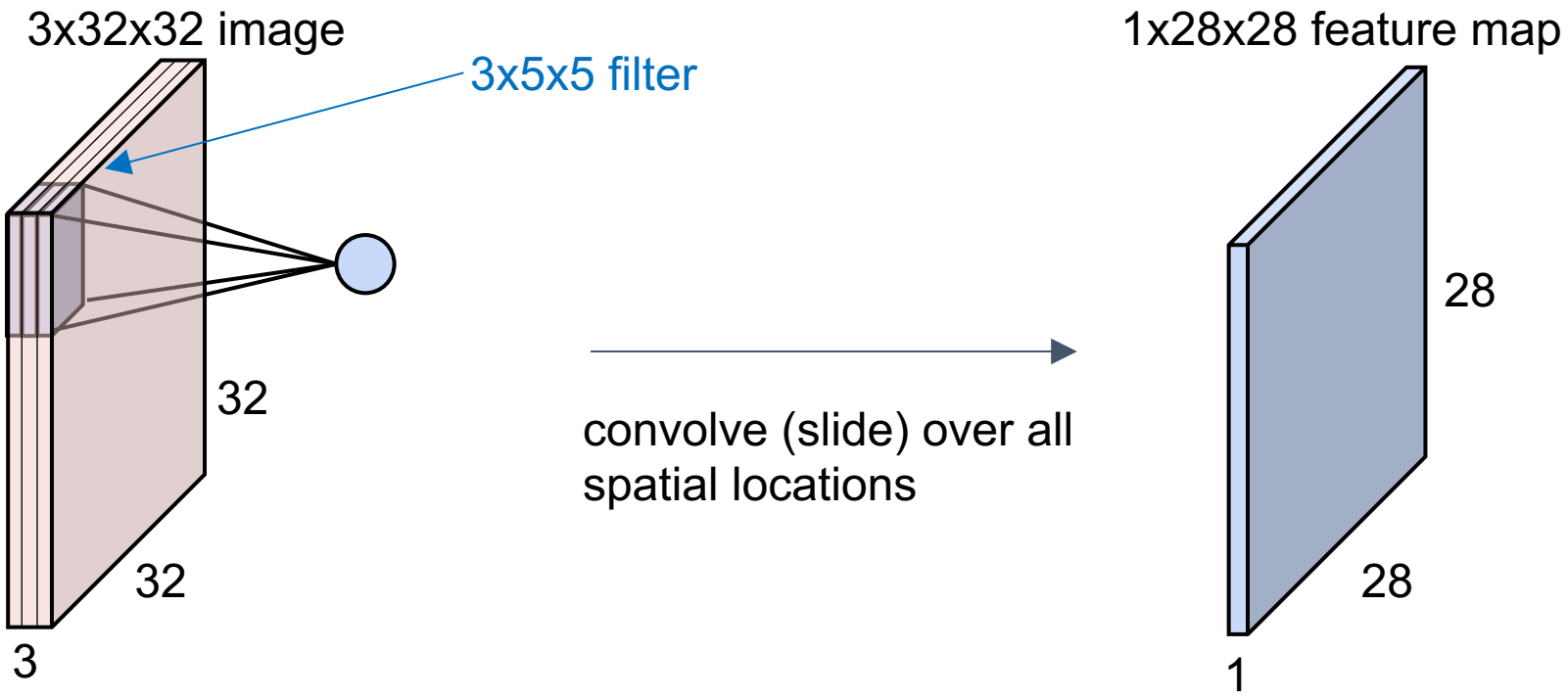


Output spatial size = $\frac{N-F+2P}{S} + 1$

In this example, $N = 32, F = 5, P = 0, S = 1$

Thus, output spatial size = $\frac{32-5+2(0)}{1} + 1 = 28$

Recap: How to calculate the number of parameters?

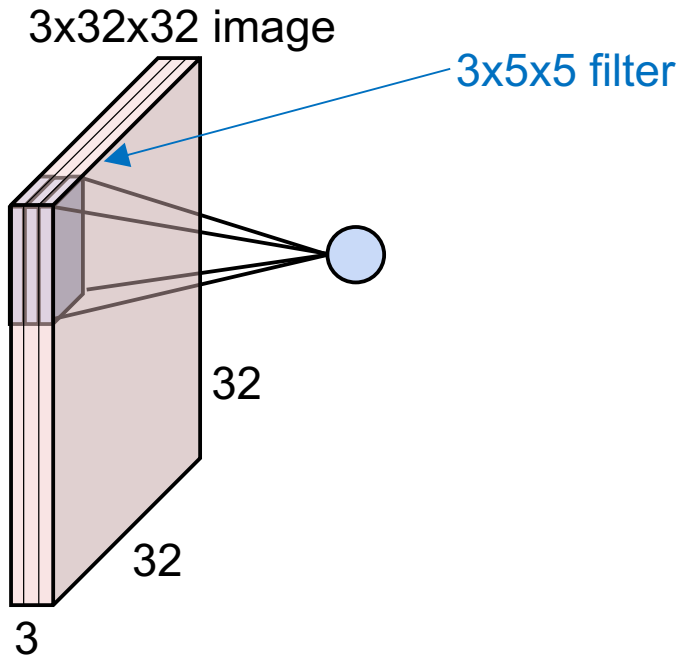


Let say we have ten 3×5×5 filters with stride 1, pad 0

Recap: How to calculate the number of parameters?

Input volume: $3 \times 32 \times 32$
Ten $3 \times 5 \times 5$ filters with stride 1, pad 0

Number of parameters in this layer: ?



Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)

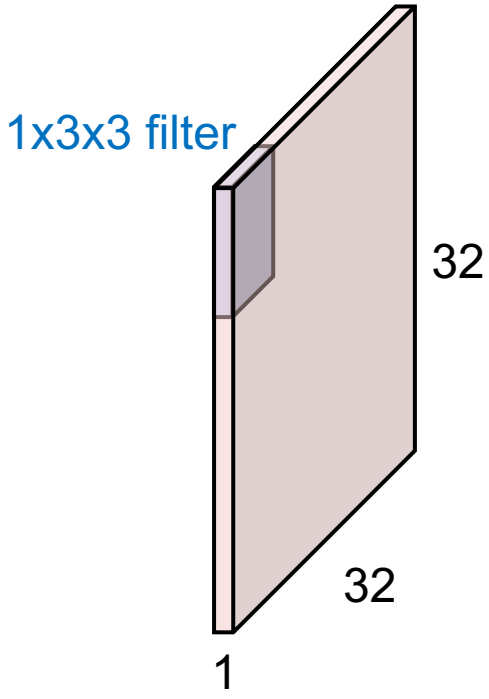
Ten filters so $76 \times 10 =$ **760 parameters**

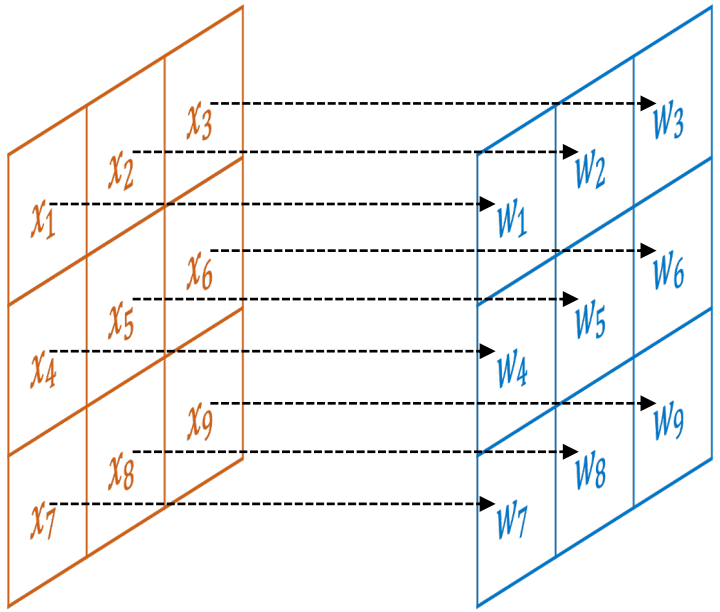
How to calculate the computations involved?

Let's focus on one input channel first

$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m)w(l,m) + b$$

Element-wise multiplication of input and weights





$x_1 w_1$

$x_2 w_2$

$x_3 w_3$

$x_4 w_4$

$x_5 w_5$

$x_6 w_6$

$x_7 w_7$

$x_8 w_8$

$x_9 w_9$

How many multiplication operations?

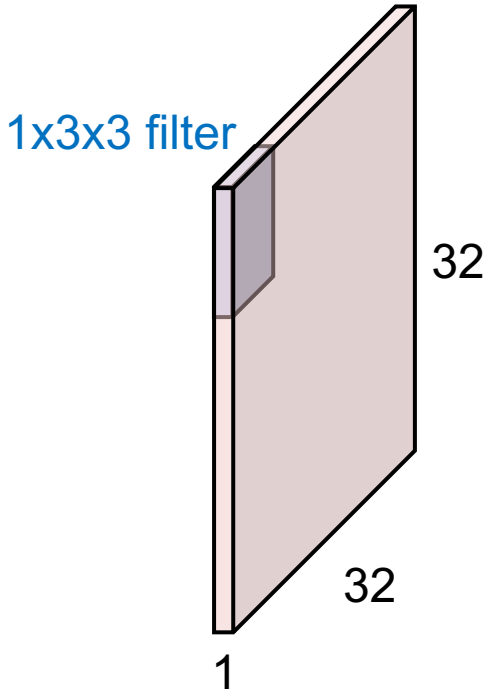
In general, there are F^2 multiplication operations, where F is the filter spatial size

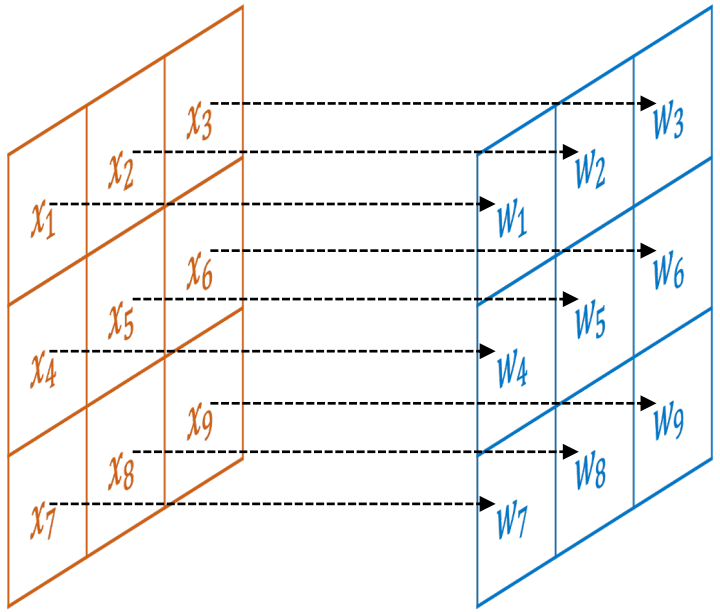
How to calculate the computations involved?

Let's focus on one input channel first

$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m)w(l,m) + b$$

Element-wise multiplication of input and weights





x_1w_1
+
x_2w_2
+
x_3w_3
+
x_4w_4
+
x_5w_5
+
x_6w_6
+
x_7w_7
+
x_8w_8
+
x_9w_9

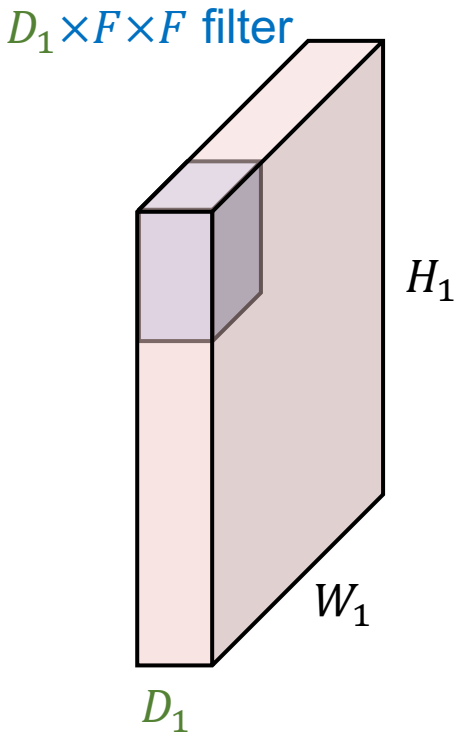
How many adding operations?

Adding the elements after multiplication, we need $n-1$ adding operations for n elements

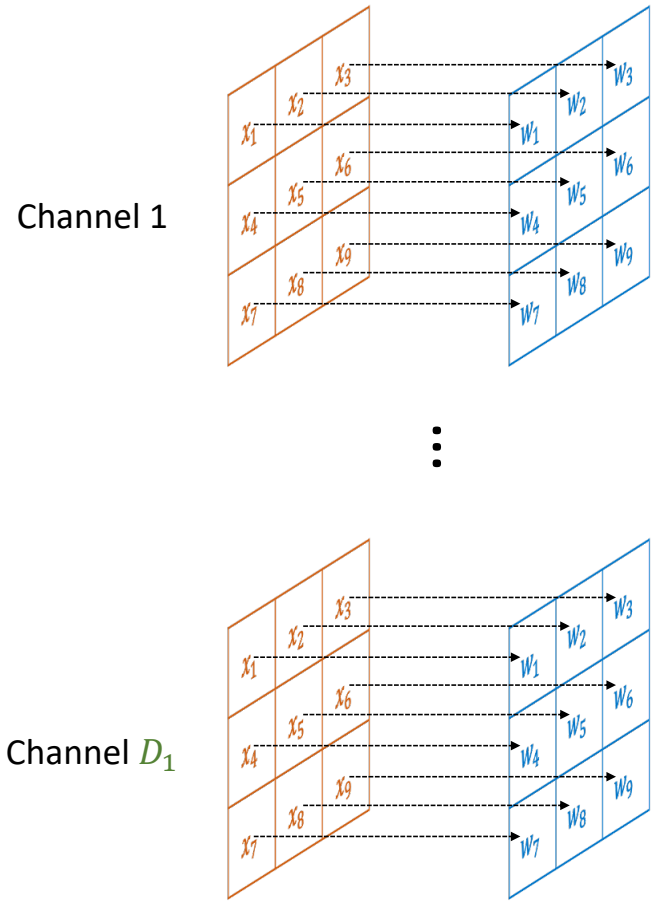
In general, there are F^2-1 adding operations, where F is the filter spatial size

How to calculate the computations involved?

Let's say we have D_1 input channels now



Element-wise multiplication of input and weights, for each channel



F^2 multiplication operations
$F^2 - 1$ adding operations

F^2 multiplication operations
$F^2 - 1$ adding operations

If we have only one filter, and apply it to generate output of one spatial location, the total operations

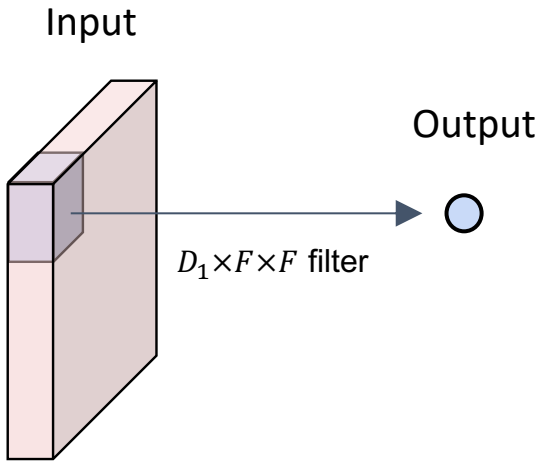
The total cost is

$$(D_1 \times F^2) + (D_1 \times F^2 - 1)$$

Let's not forget to add the bias

$$(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$$

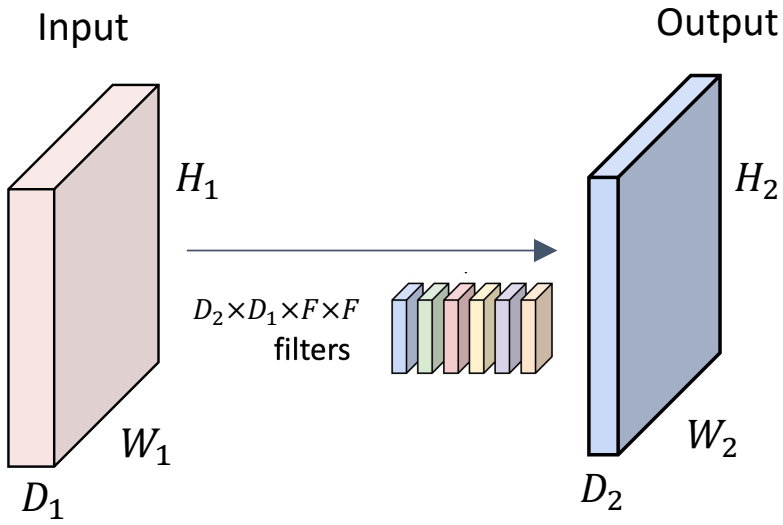
How to calculate the computations involved?



If we have only one filter, and apply it to generate output of one spatial location, the total operations

The total cost is

$$(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$$



If we have D_2 filter, and apply it to generate output of $H_2 \times W_2$ spatial location, the total operations

The total cost is

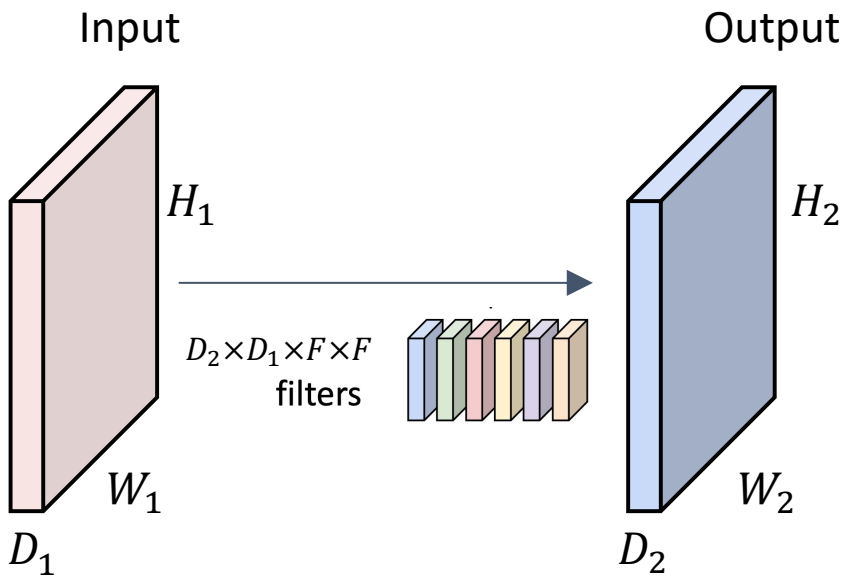
$$[(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$

$$= (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

Summary

FLOPs (floating point operations)

Not FLOPS (floating point operations per second)



Assume:

- Filter size F
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Produces a output volume of size $D_2 \times H_2 \times W_2$

The FLOPs of the convolution layer is given by

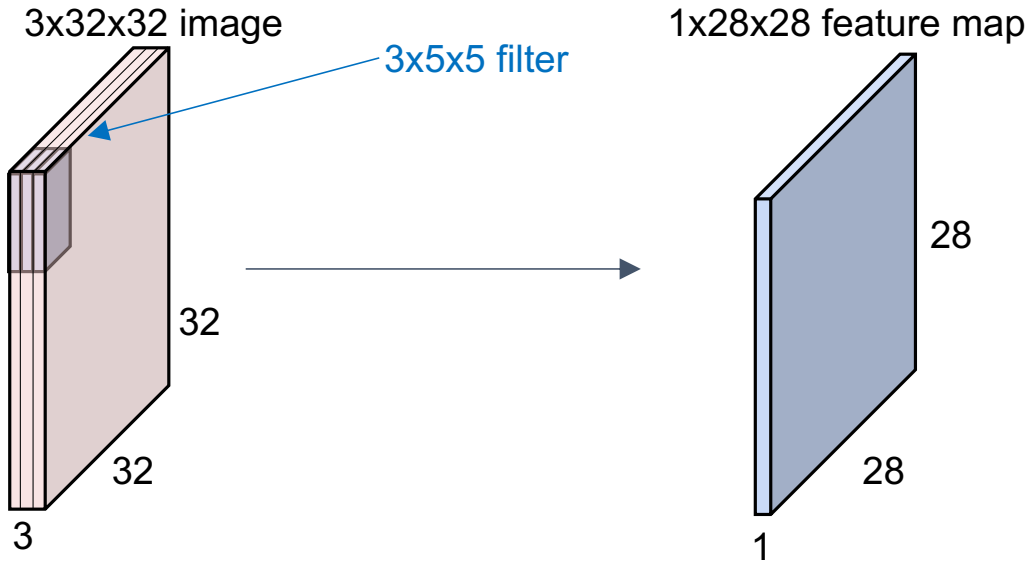
$$\text{FLOPs} = \underbrace{[(D_1 \times F^2)}_{\text{Elementwise multiplication of each filter on a spatial location}} + \underbrace{(D_1 \times F^2 - 1)}_{\text{Adding the elements after multiplication, we need } n-1 \text{ adding operations for } n \text{ elements}} + \underbrace{1}_{\text{Add the bias}}] \times D_2 \times H_2 \times W_2 = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

Try this

Assume:

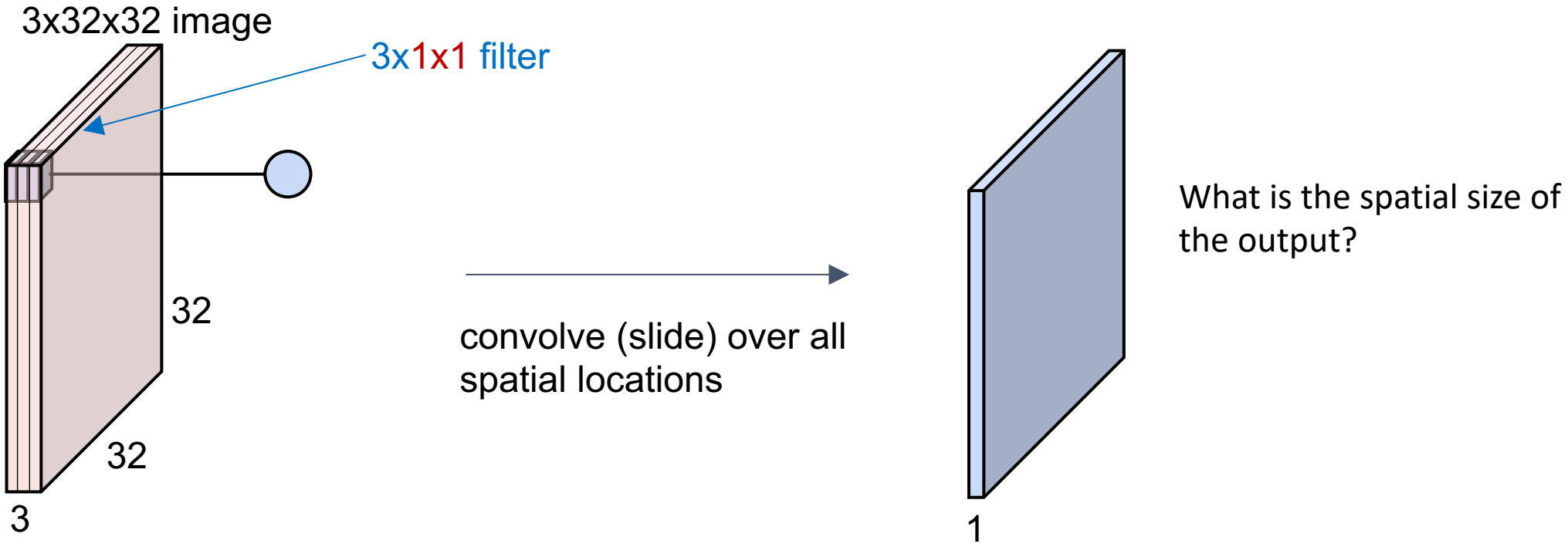
- One filter with spatial size of 5×5
- Accepts a volume of size $3 \times 32 \times 32$
- Produces a output volume of size $1 \times 28 \times 28$

The FLOPs of the convolution layer is given by?



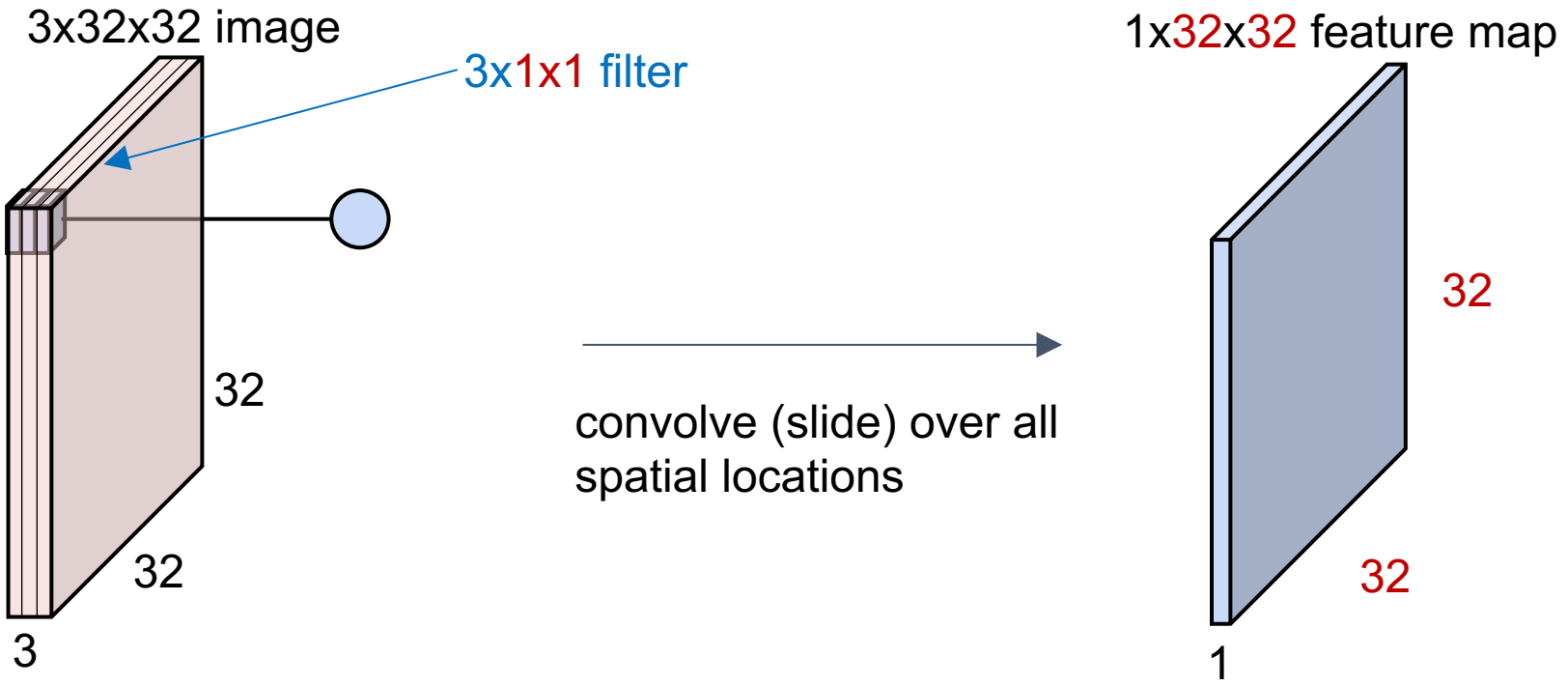
Pointwise convolution

- We have seen convolution with spatial size of 3×3, 5×5, 7×7
  - Can we have other sizes?
  - Can we have filter of spatial size 1×1?



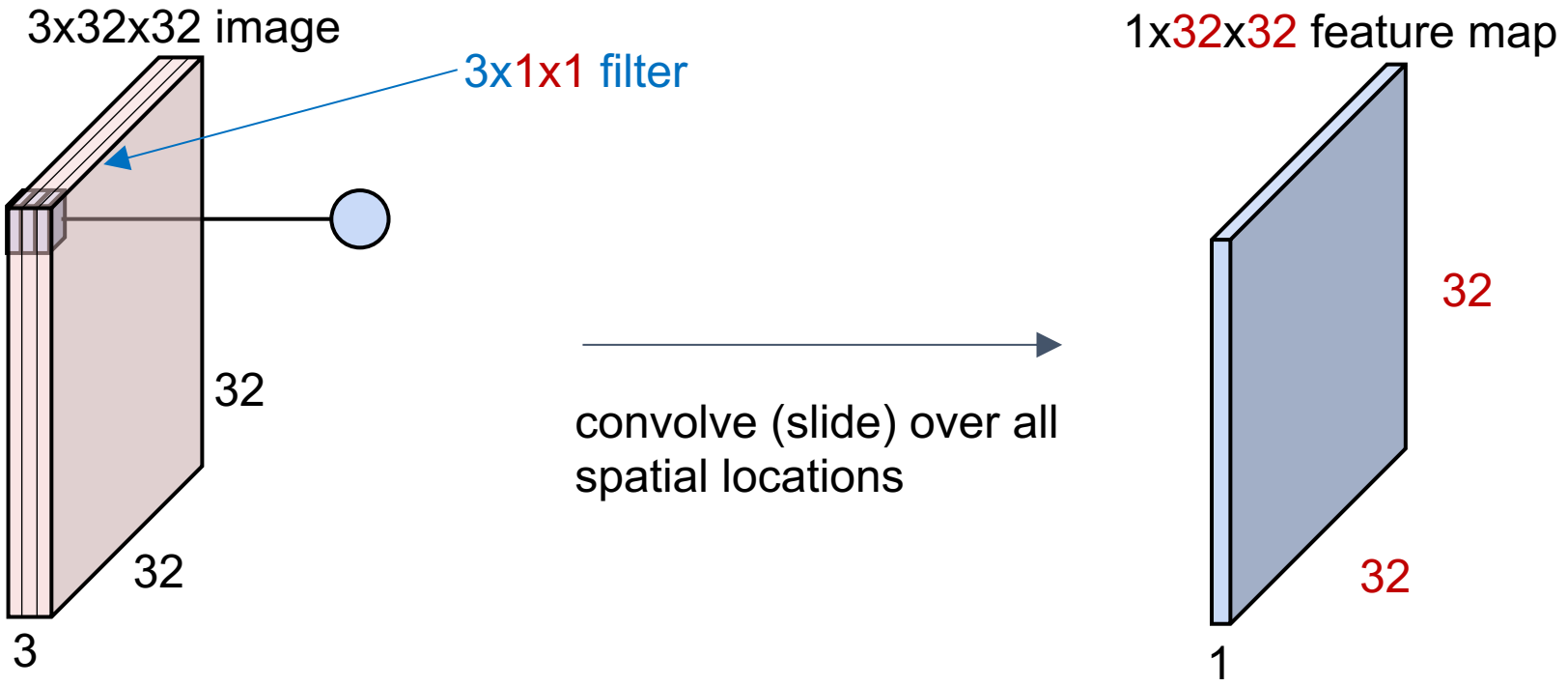
Pointwise convolution

- We have seen convolution with spatial size of 3×3, 5×5, 7×7
  - Can we have other sizes?
  - Can we have filter of spatial size 1×1?



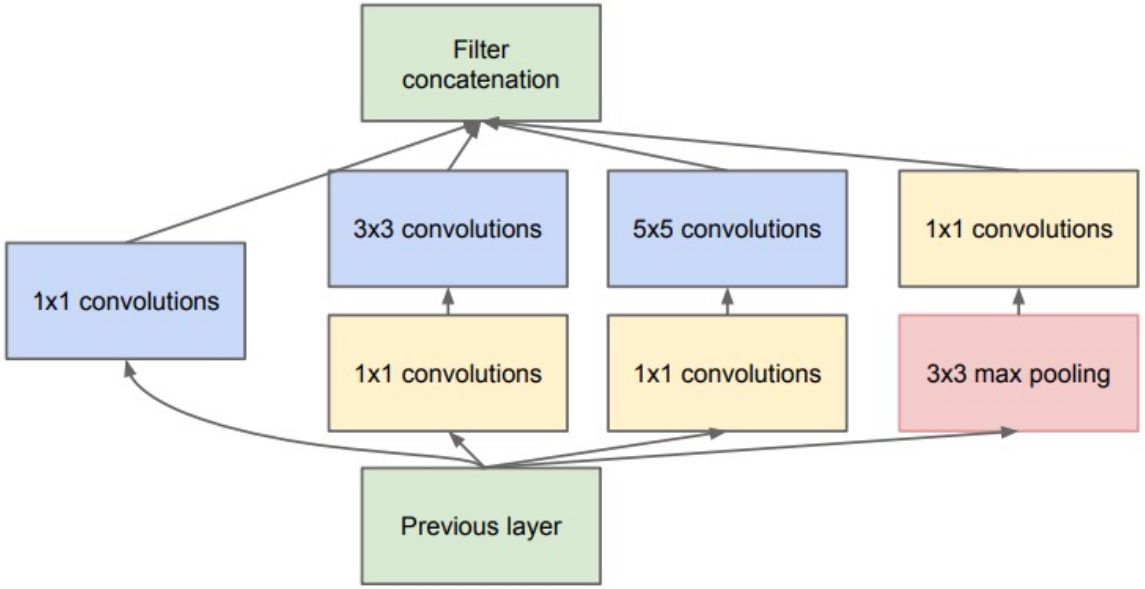
Pointwise convolution

- Why having filter of spatial size 1×1?
  - Change the size of channels
  - "Blend" information among channels by linear combination



Pointwise convolution

- A real-world example
  - 1×1 convolutions are used for compute reductions before the expensive 3×3 and 5×5 convolutions

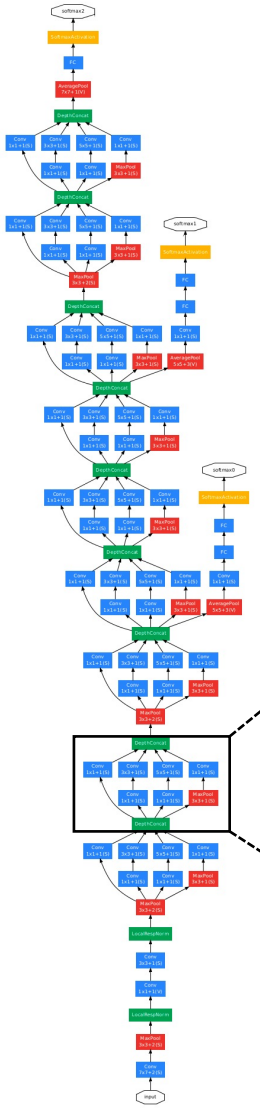


E.g., Given the output from the previous layer, reduce the number of channels of from $256\times32\times32$ to $128\times32\times32$ before the 5×5 convolutions

Referring to the FLOPs equation, reducing input channels help reduce the FLOPs

$$\text{FLOPs} = (2\times D_1\times F^2)\times D_2\times H_2\times W_2$$

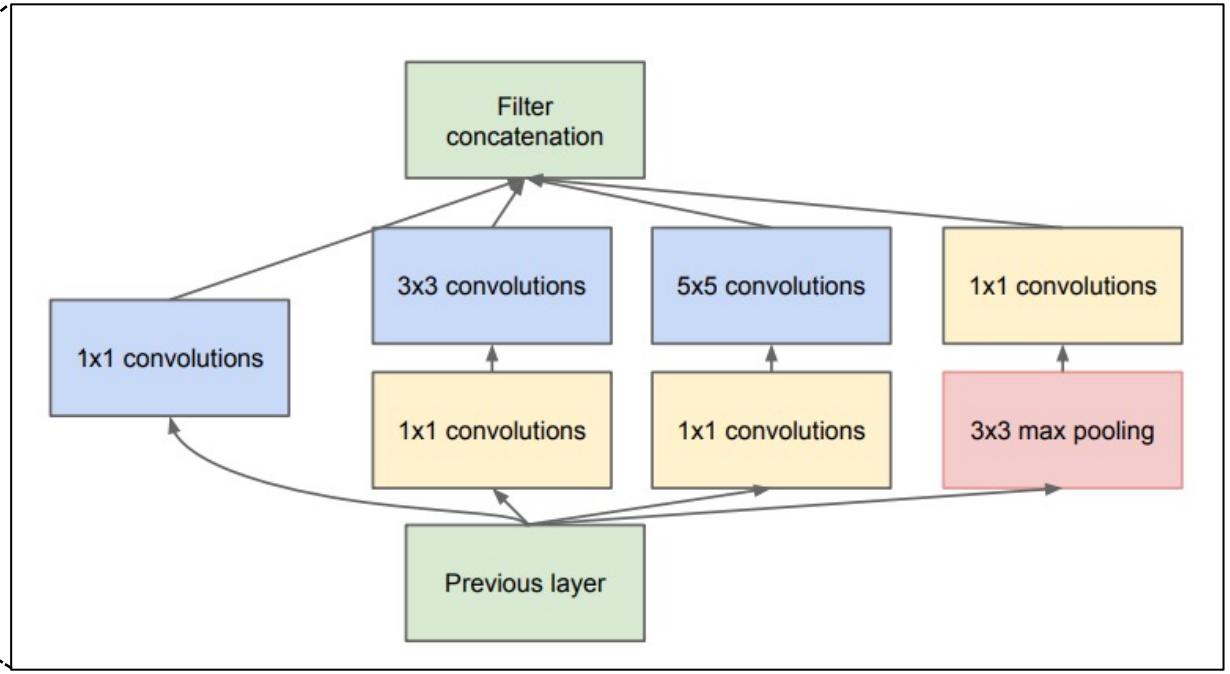
Pointwise convolution



This is known as inception structure used in GoogLeNet

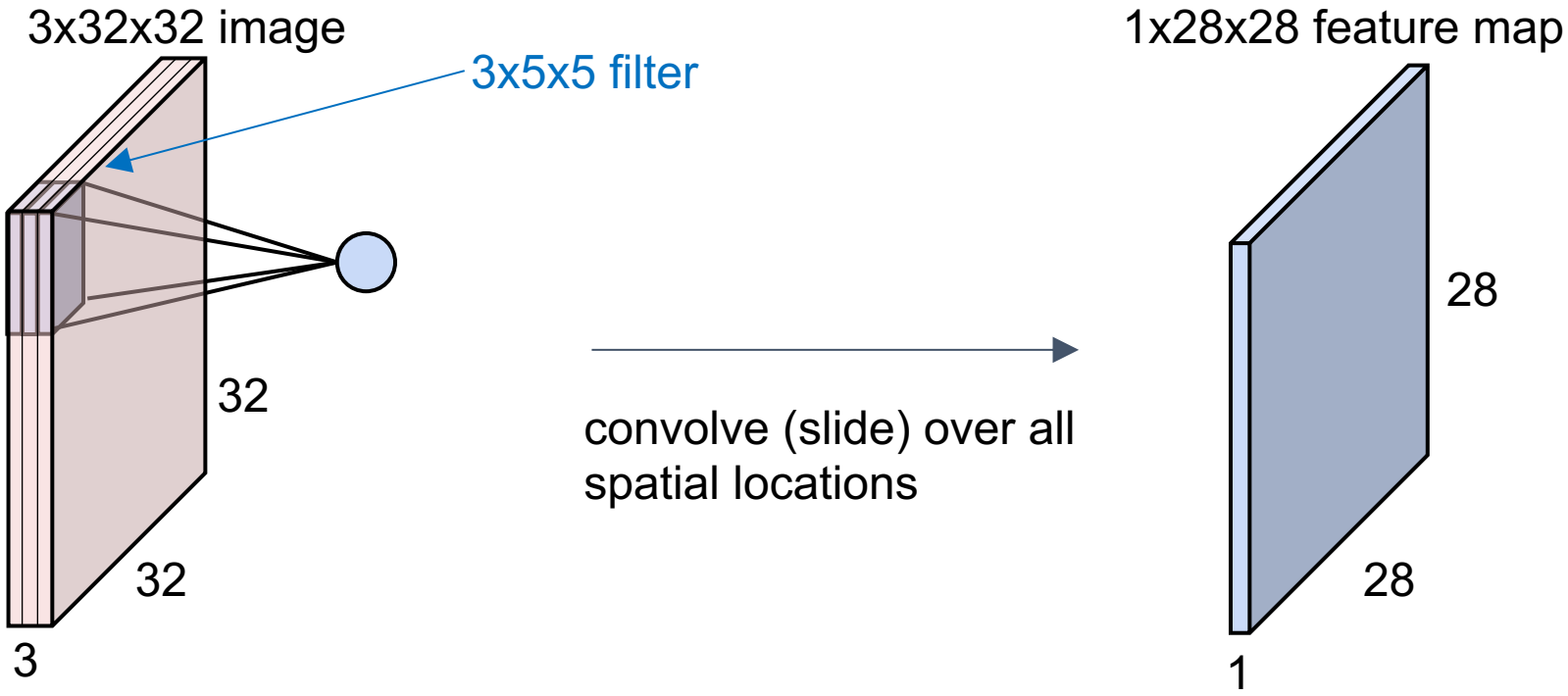
Proposed by Google in 2014, winning the ImageNet competition that year

4M parameters vs 60M parameters of AlexNet



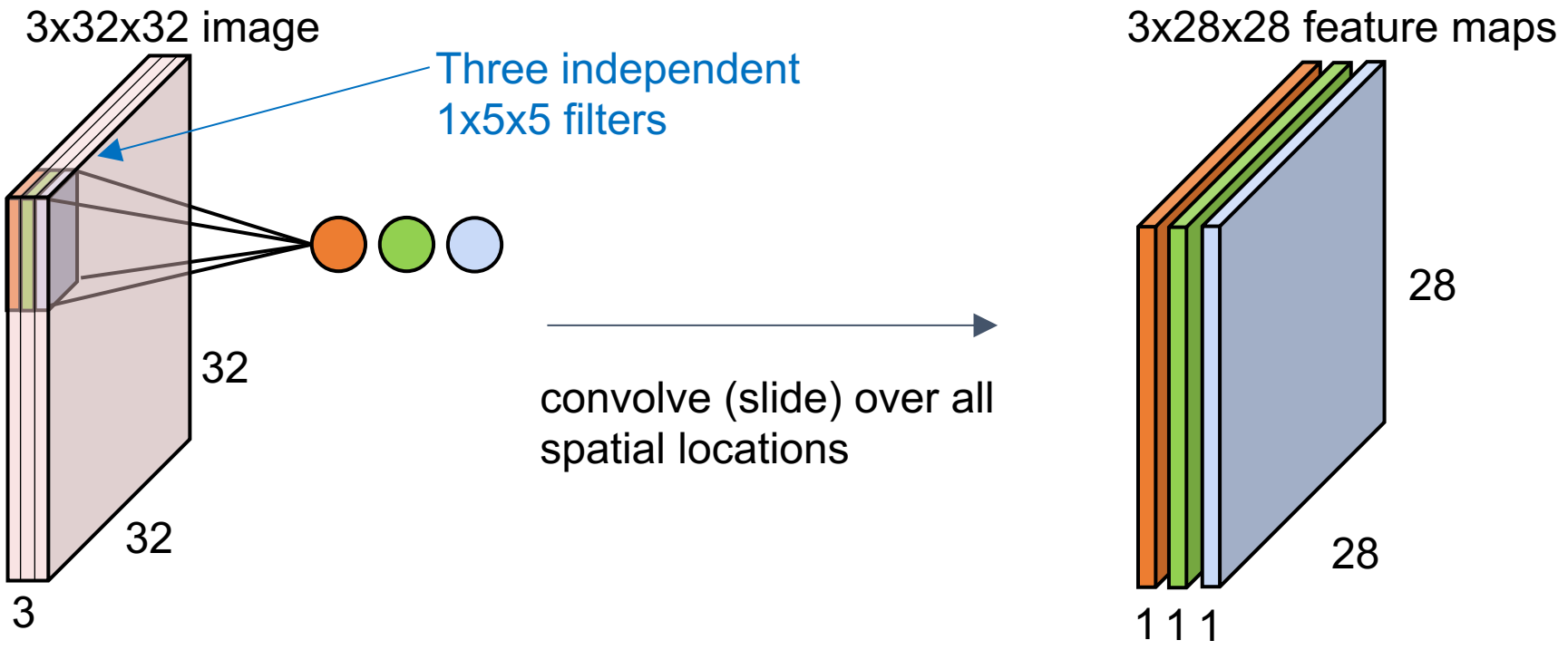
Standard convolution

- Standard convolution
  - The input and output are locally connected in spatial domain
  - In channel domain, they are fully connected

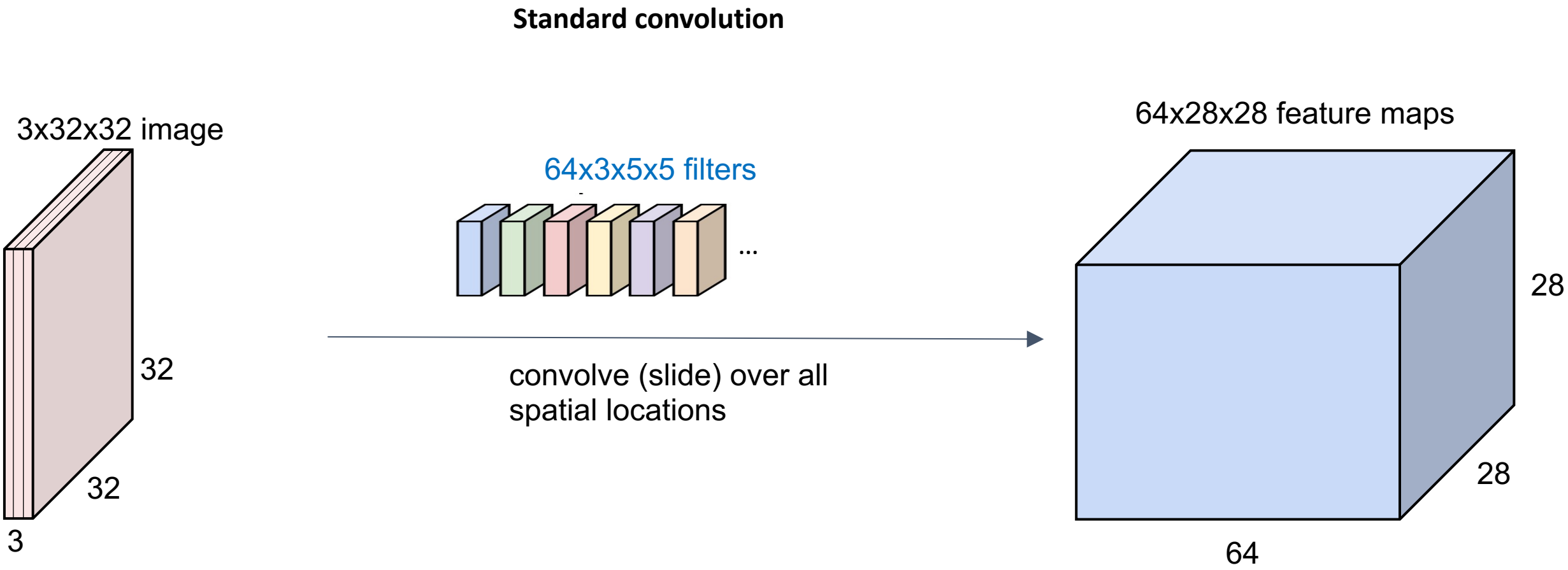


Depthwise convolution

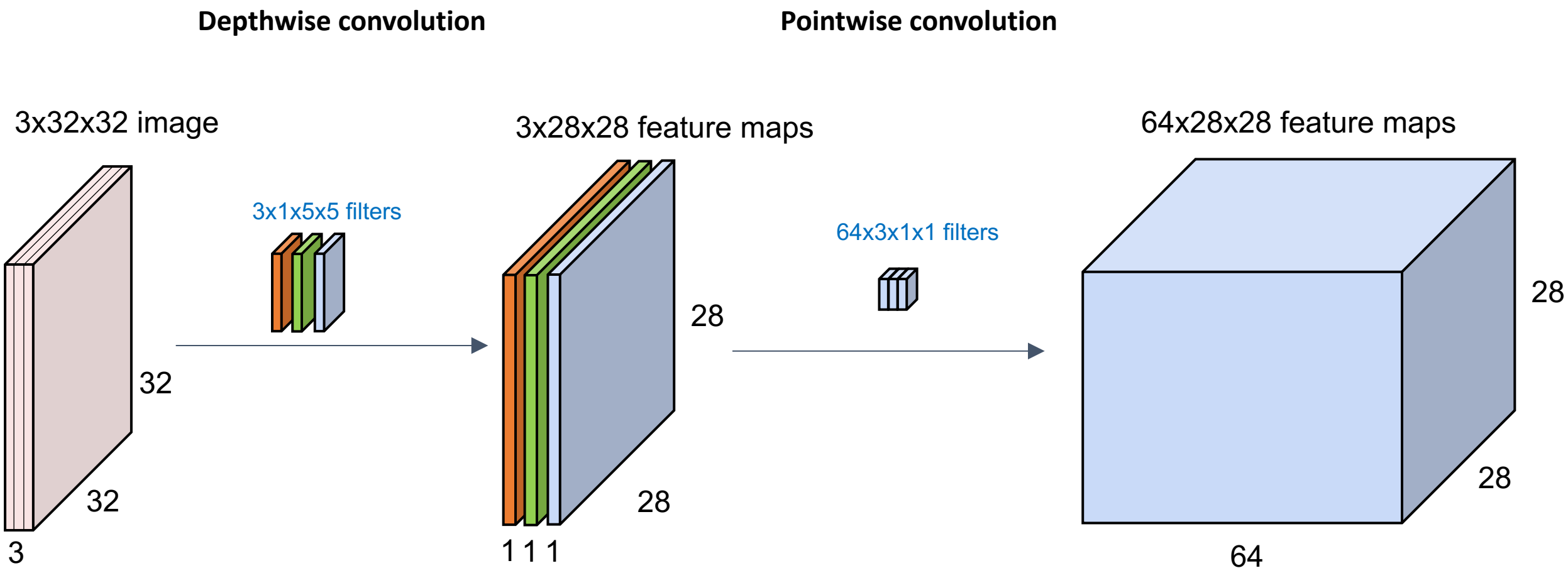
- Depthwise convolution
  - Convolution is performed independently for each of input channels



Standard convolution



Depthwise convolution + Pointwise convolution



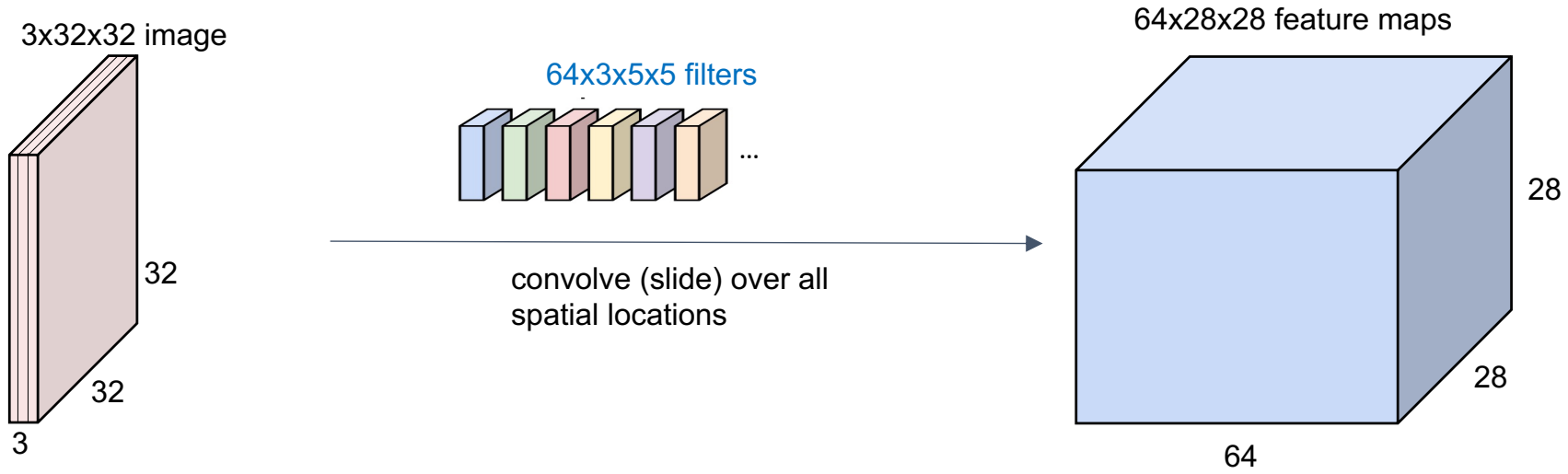
Depthwise convolution + Pointwise convolution

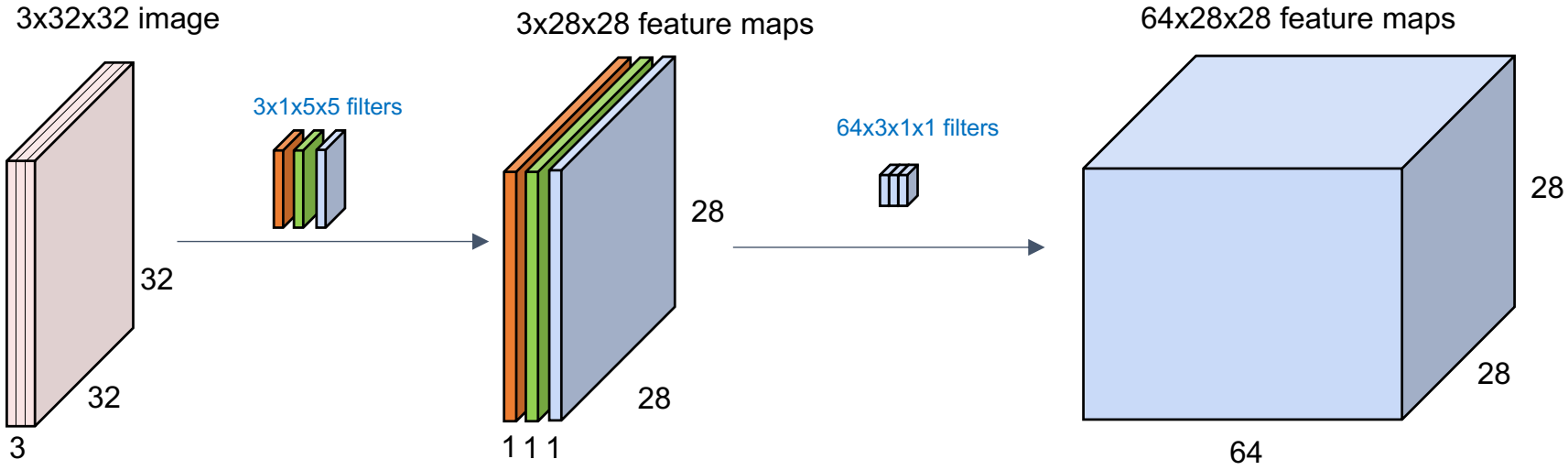
Replace standard convolution

with

Depthwise convolution + pointwise convolution

And we still get the same size of output volume!





Depthwise convolution + Pointwise convolution

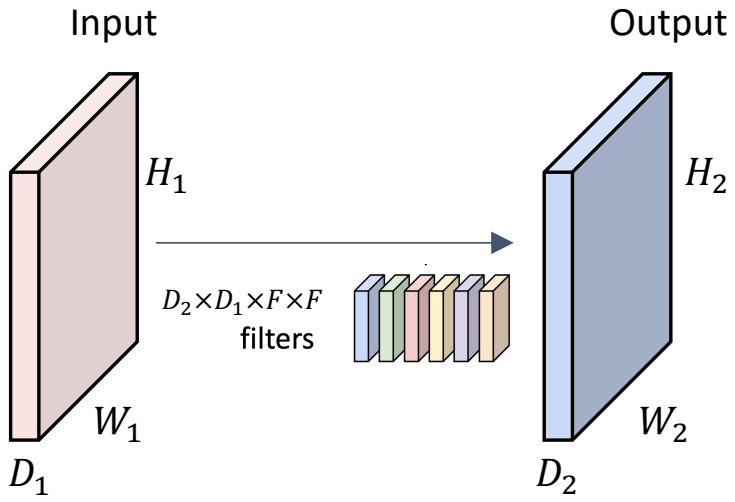
- Why replacing standard convolution with depthwise convolution + pointwise convolution?
  - The computational cost reduction rate is roughly 1/8–1/9 at only a small reduction in accuracy
  - Good if you want to deploy small networks on devices with CPU
  - Used in network such as MobileNet

G. Howard et al., MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, CVPR 2017

Depthwise convolution + Pointwise convolution

- Standard convolution cost

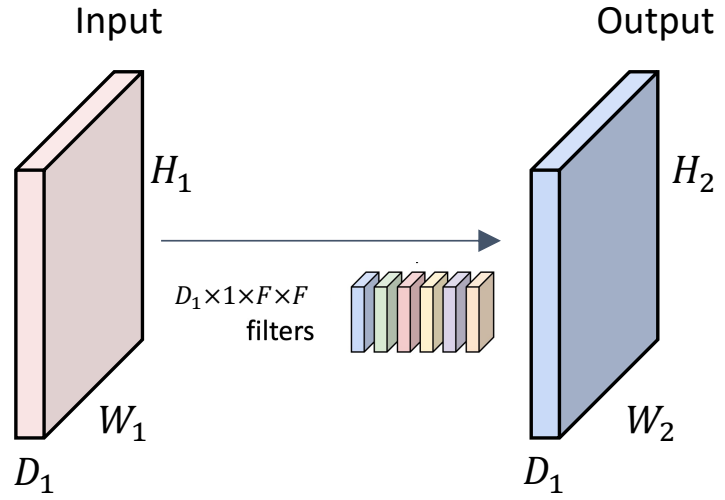
$$\text{FLOPs}_{\text{standard}} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$



- Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = [(1 \times F^2) + (1 \times F^2 - 1) + 1] \times D_1 \times H_2 \times W_2$$

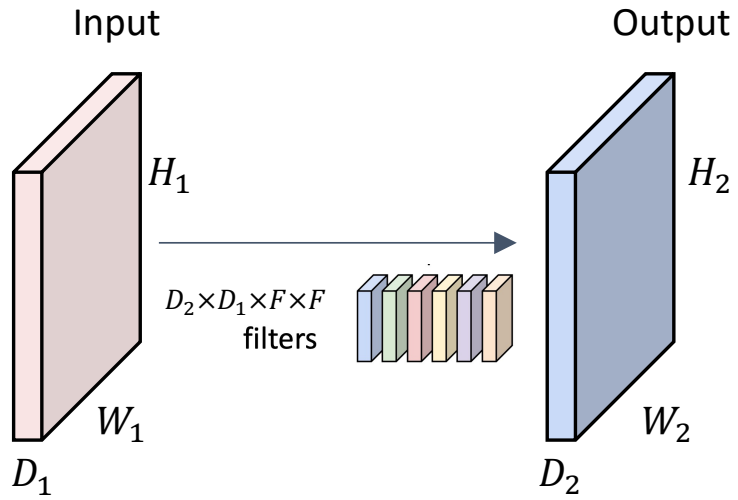
- Each filter is applied only to one channel
- The number of output channels equals to the number of input channels



Depthwise convolution + Pointwise convolution

- Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$



- Pointwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = [(D_1 \times 1) + (D_1 \times 1 - 1) + 1] \times D_2 \times H_2 \times W_2$$

  - The filter spatial size is 1×1

Depthwise convolution + Pointwise convolution

- Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

- Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = (2 \times F^2) \times D_1 \times H_2 \times W_2$$

- Pointwise convolution cost

$$\text{FLOPs}_{\text{pointwise}} = (2 \times D_1) \times D_2 \times H_2 \times W_2$$

Depthwise convolution + Pointwise convolution

- How much computation do you save by replacing standard convolution with depthwise+pointwise?

$$\text{Reduction} = \frac{\text{Cost of depthwise convolution} + \text{pointwise convolution}}{\text{Cost of standard convolution}}$$

$$\text{Reduction} = \frac{(2\times F^2)\times D_1\times H_2\times W_2}{(2\times D_1\times F^2)\times D_2\times H_2\times W_2} + \frac{(2\times D_1)\times D_2\times H_2\times W_2}{(2\times D_1\times F^2)\times D_2\times H_2\times W_2}$$

$$\text{Reduction} = \frac{1}{D_2} + \frac{1}{F^2}$$

D_2 is usually large. Reduction rate is roughly 1/8–1/9 if 3×3 depthwise separable convolutions are used

Summary

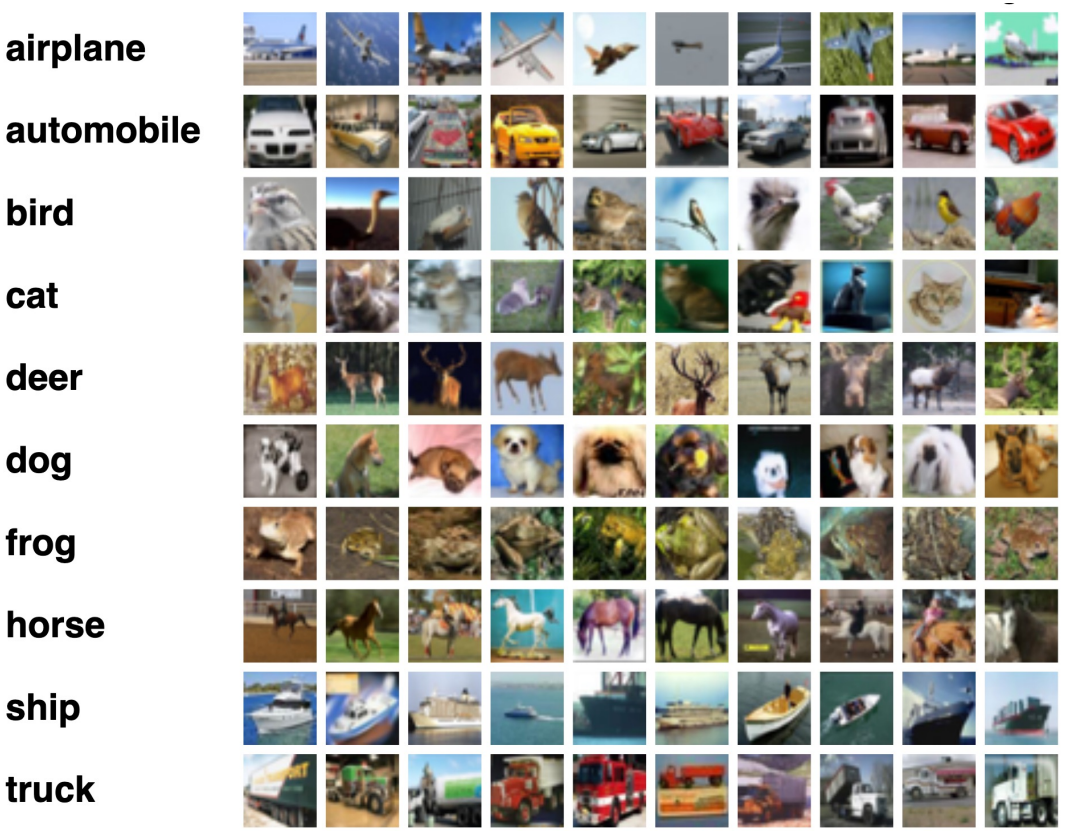
- More on convolutions
  - You learn how to calculate computation complexity of convolutional layer
  - Pointwise convolution can be used to change the size of channels. This can be used to achieve channel reduction and thus saving computational cost
  - Depthwise convolution + Pointwise convolution yields lower computations than standard convolution

Outline

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
- Training basics
  - Weight initialization
  - Optimizers
  - Prevent overfitting
- Batch normalization

Training Basics

Multi-class classification



CIFAR-10

- 10 classes
- 6000 images per class
- 60000 images - 50000 training images and 10000 test images
- Each image has a size of 3x32x32, that is 3-channel color images of 32x32 pixels in size

https://www.cs.toronto.edu/~kriz/cifar.html

Multi-class classification

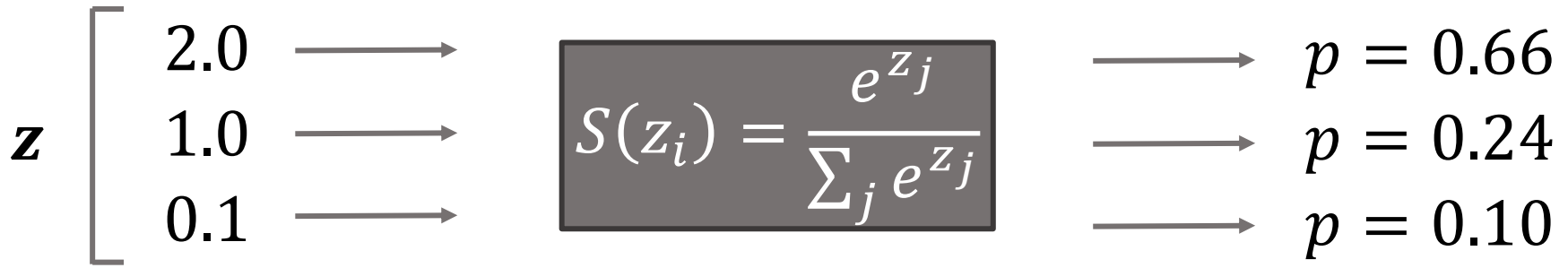
MNIST

- Size-normalized and centred 1x28x28 =784 inputs
- Training set = 60,000 images
- Testing set = 10,000 images

http://yann.lecun.com/exdb/mnist/

Softmax function

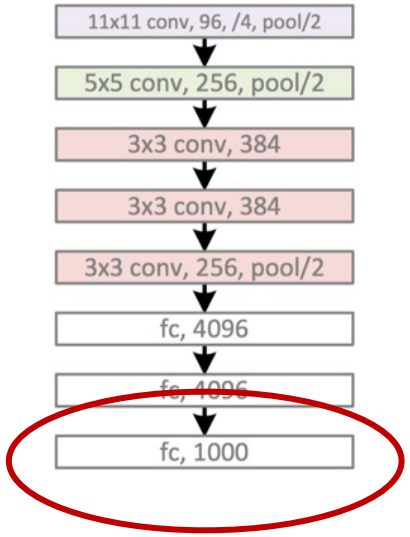
The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $\boldsymbol{z} \in \mathbb{R}^n$ and outputs a vector of output probability $\boldsymbol{p} \in \mathbb{R}^n$ through a softmax function at the end of the architecture.



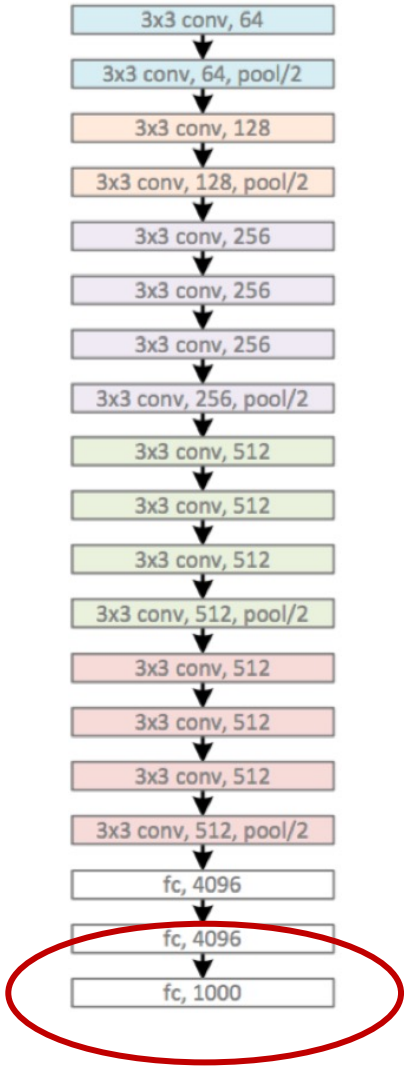
Numeric output of the last linear layer of a multi-class classification neural network

Softmax function

AlexNet



VGG



Where does the Softmax function fit in a CNN architecture?

Softmax's input is the output of the fully connected layer immediately preceding it, and it outputs the final output of the entire neural network. This output is a probability distribution of all the label class candidates.

Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label

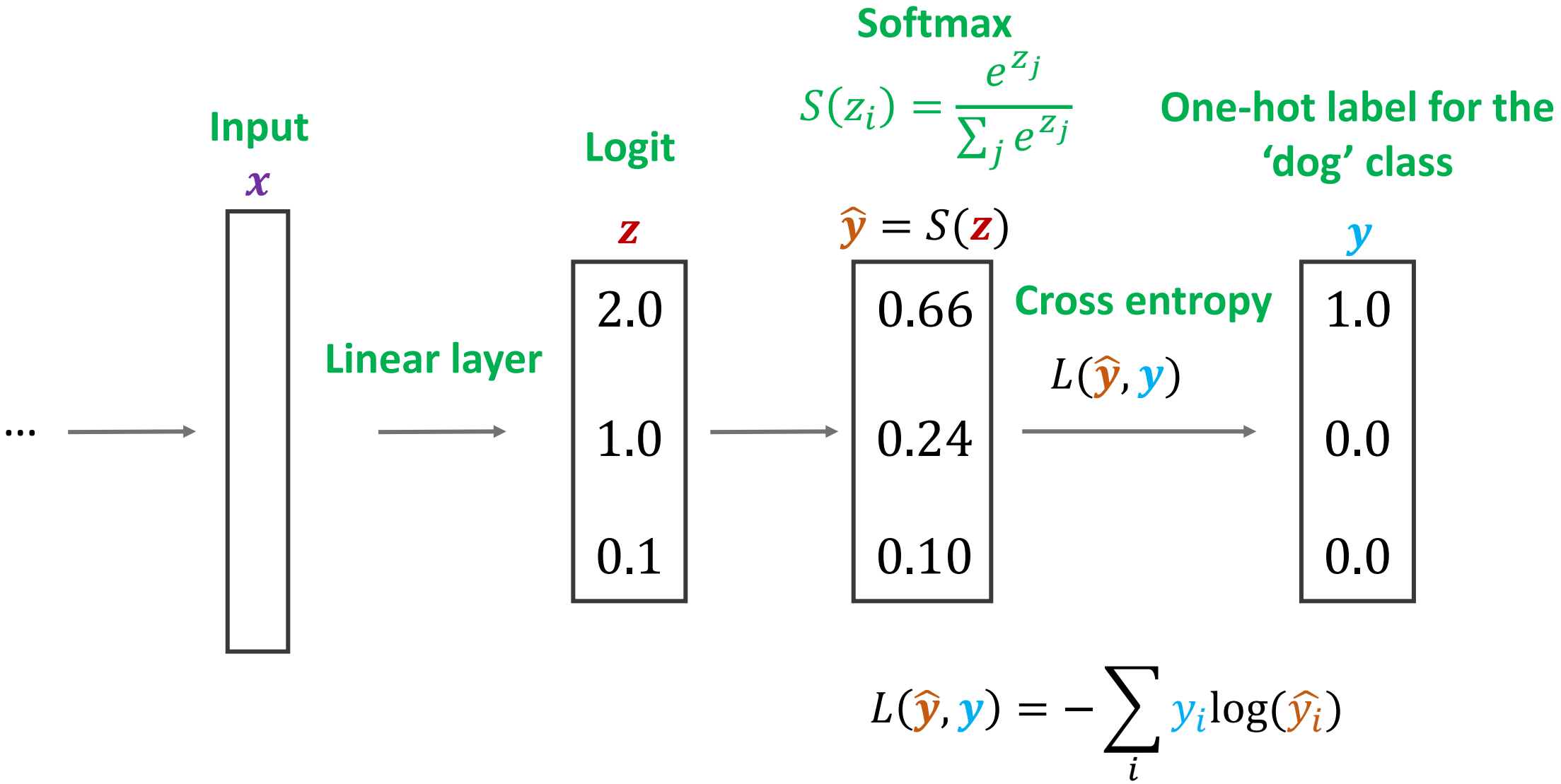
Cross entropy loss

One-hot encoded ground truth

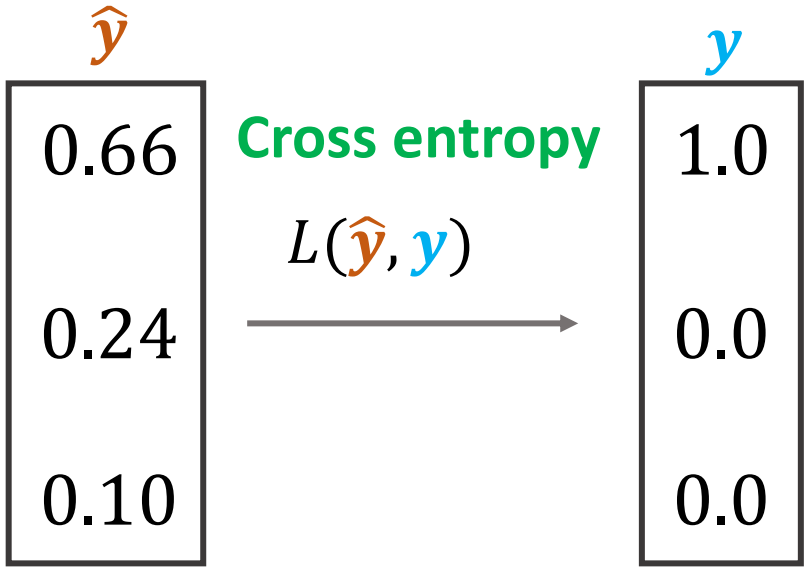
Example:

Class	One-hot vector
Dog	$[1 \quad 0 \quad 0]$
Cat	$[0 \quad 1 \quad 0]$
Bird	$[0 \quad 0 \quad 1]$

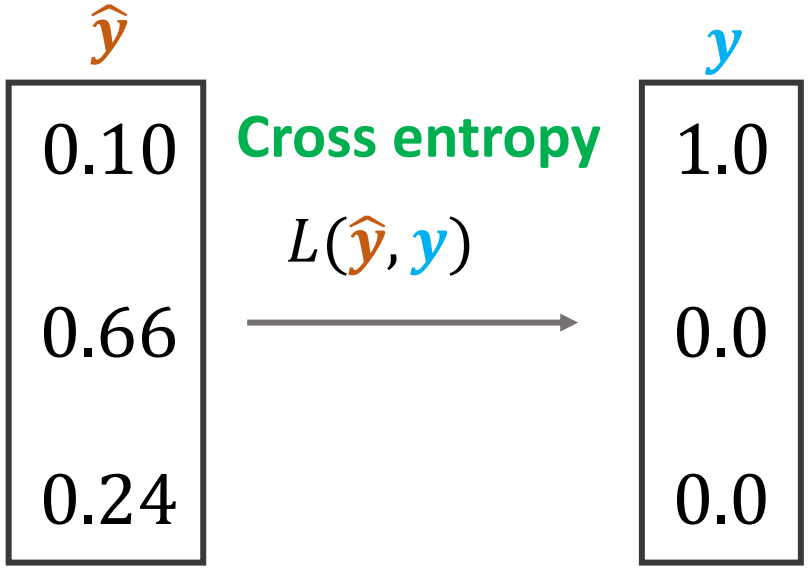
Cross entropy loss



Cross entropy loss



$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.66)) + (0 \times \log_2(0.24)) + (0 \times \log_2(0.10))]$$
$$= 0.6$$



$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.10)) + (0 \times \log_2(0.66)) + (0 \times \log_2(0.24))]$$
$$= 3.32$$

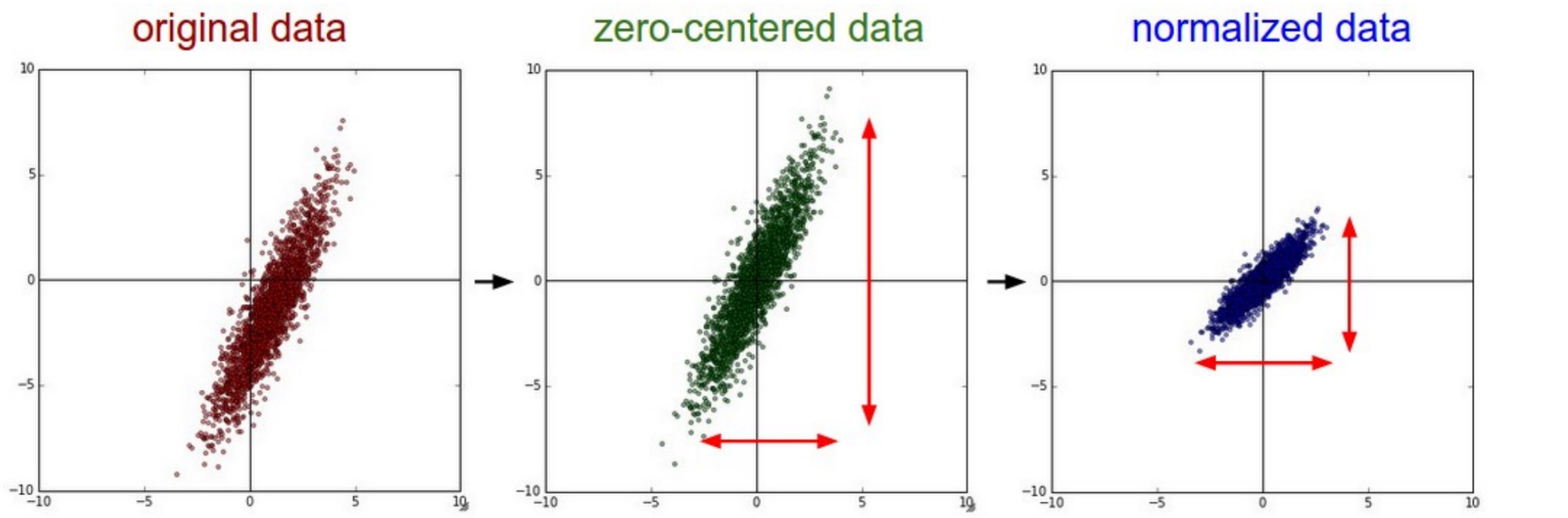
What is the min / max possible loss?

Epoch and mini-batch

- **Epoch** – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

- **Mini-batch** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on mini batches, where the number of data points in a batch is a hyperparameter that we can tune.

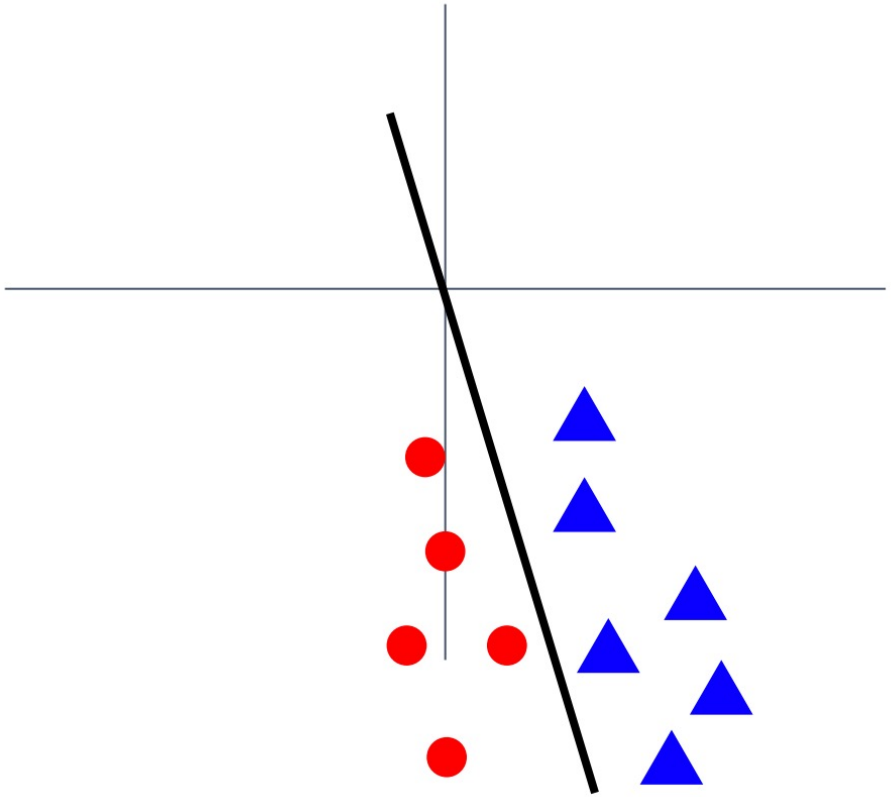
Data pre-processing



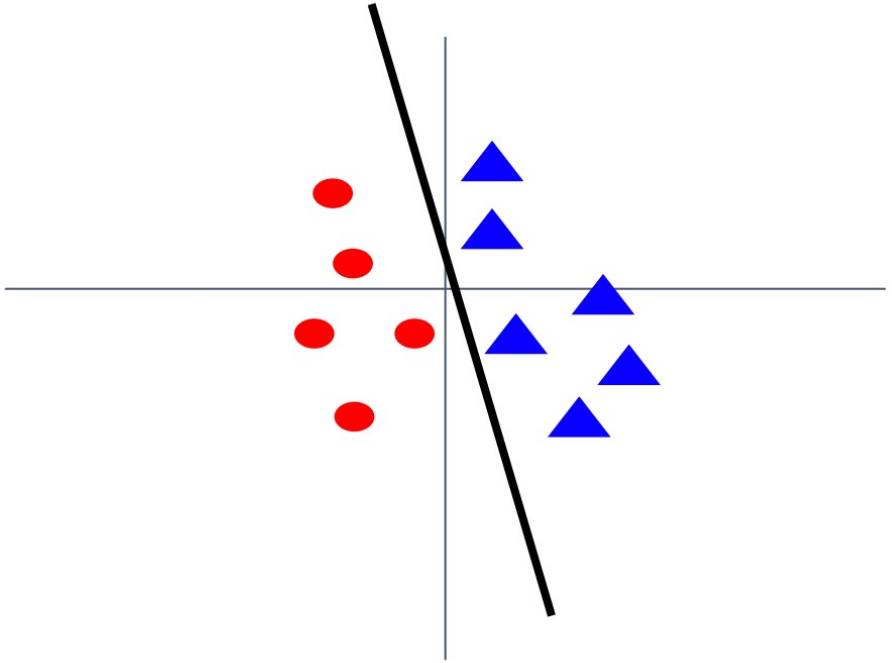
(Assume X [NxD] is data matrix,
each example in a row)

Data pre-processing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data pre-processing for images

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Outline

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
- Training basics
  - Weight initialization
  - Optimizers
  - Prevent overfitting
- Batch normalization

Weight Initialization

Network Initialization

- Q: What happens if we initialize all W=0, b=0?

Network Initialization

- Next idea: small random numbers (Gaussian with zero mean, std=0.01)

```
W = 0.01 * np.random.randn(Din, Dout)
```

Works okay for small networks, but problems with deeper networks.

Network Initialization

- The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network.

- If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Xavier Initialization

- Xavier initialization (torch.nn.init.xavier_normal_) sets a layer's weights to values chosen from a random uniform distribution

$$\mathcal{N}(0, \frac{2}{n^{[l-1]}+n^{[l]}})$$

where $n^{[l-1]}$ is the number of incoming neurons, or "fan-in," to the layer, and $n^{[l]}$ is the number of outgoing neurons from that layer, also known as the "fan-out."

- Maintain the variance of activations across layers
  - Proof – Glorot's paper or http://www.deeplearning.ai/ai-notes/initialization/#II

- Assume the activation function is Tanh

X. Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, AISTATS 2010

Kaiming Initialization

- Initialization specifically for ReLU (nn.init.kaiming_normal_)

$$\mathcal{N}\left(0, \frac{2}{n^{[l-1]}}\right)$$

where $n^{[l-1]}$ is the number of incoming neurons, or "fan-in," to the layer (receptive field x number of channels)

- Compared to Xavier
  - ReLU does not have zero mean, leading to conclusion different from Xavier
  - No clear difference in convergence for a 22-layer model
  - Xaviers completely stalls the learning for a 30-layer model (gradient diminished)

K. He, et al., Delving Deep into Rectifiers, ICCV 2015

Outline

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
- Training basics
  - Weight initialization
  - Optimizers
  - Prevent overfitting
- Batch normalization

Optimizers

Optimization

- A CNN as composition of functions

$$f_{\boldsymbol{w}}(\boldsymbol{x}) = f_L(\dots\ (f_2(f_1(\boldsymbol{x}; \boldsymbol{w}_1); \boldsymbol{w}_2)\ \dots; \boldsymbol{w}_L)$$

- Parameters

$$\boldsymbol{w} = (\boldsymbol{w_1}, \boldsymbol{w_2}, \dots \boldsymbol{w_L})$$

- Empirical loss function

$$L(\boldsymbol{w}) = \frac{1}{n}\sum_i l(y_i, f_{\boldsymbol{w}}(\boldsymbol{x_i}))$$

$$\boldsymbol{w}^* = \text{argmin}_{\boldsymbol{w}}\, L(\boldsymbol{w})$$

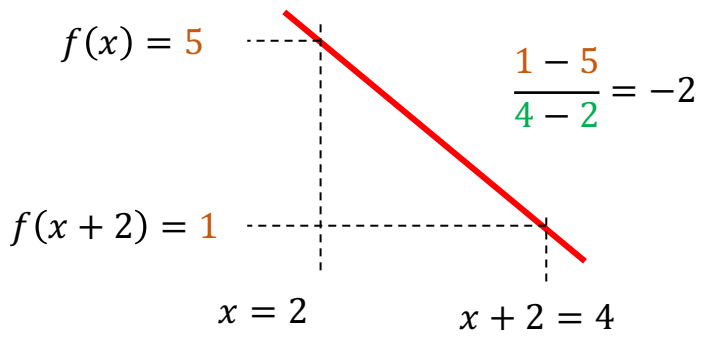
Random search is a bad idea

Follow the slope

Optimization

- In 1-dimension, the derivative of a function gives the slope:

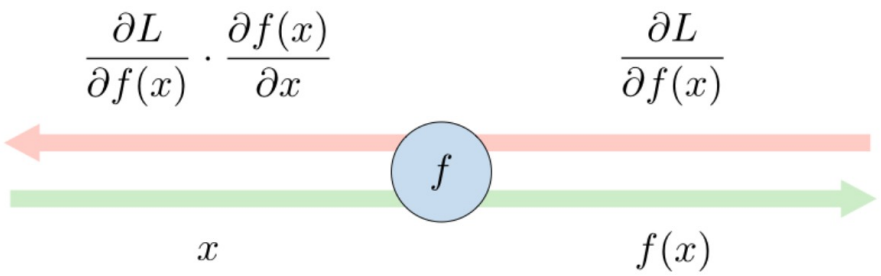
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension
- The direction of steepest descent is the **negative gradient**

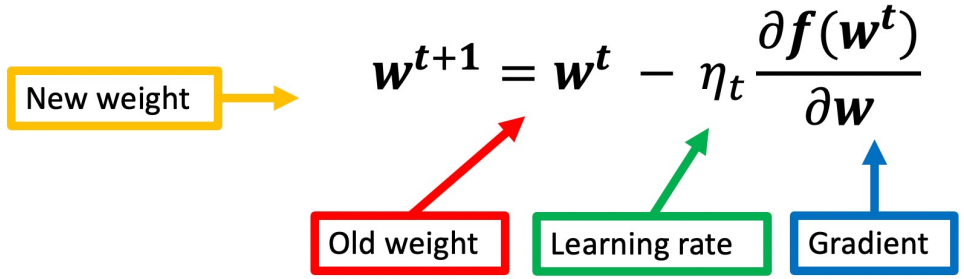
Training a Neural Network - Finding Optimal Weights

- **Backpropagation** – Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to each weight w is computed using the chain rule.



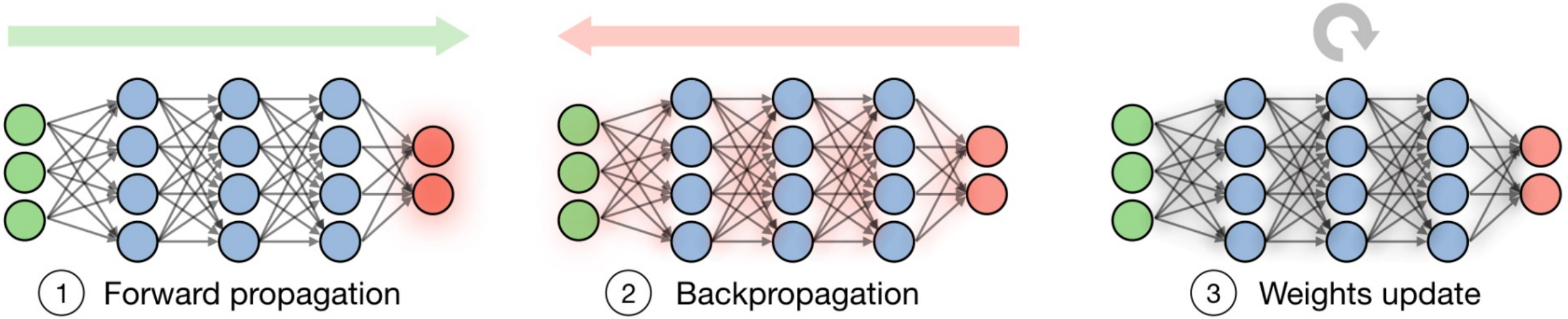
Intuitively, the chain rule states that knowing the instantaneous rate of change of *L* relative to *f(x)* and that of *f(x)* relative to *x* allows one to calculate the instantaneous rate of change of *L* relative to *x*.

- Using this method, each weight is updated with the rule



Training a Neural Network - Finding Optimal Weights

- **Updating weights** – In a neural network, weights are updated as follows:
  - Step 1: Take a batch of training data and perform forward propagation to compute the loss.
  - Step 2: Backpropagate the loss to get the gradient of the loss with respect to each weight.
  - Step 3: Use the gradients to update the weights of the network.



Gradient descent (GD)

- Iteratively step in the direction of the negative gradient
- Gradient descent

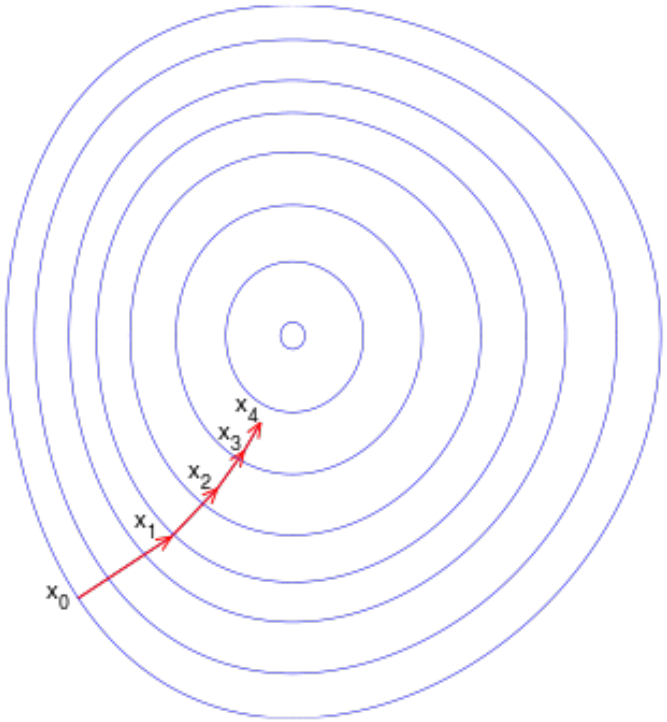
New weight

$$\boldsymbol{w^{t+1}} = \boldsymbol{w^{t}} - \eta_t \frac{\partial \boldsymbol{f}(\boldsymbol{w^{t}})}{\partial \boldsymbol{w}}$$

Old weight

Learning rate

Gradient



Gradient descent (GD)

- Batch Gradient Descent
  - Full sum is expensive when N is large

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

- Stochastic Gradient Descent (SGD)
  - Approximate sum using a minibatch of examples
  - 32 / 64 / 128 common minibatch size
  - Additional hyperparameter on batch size

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

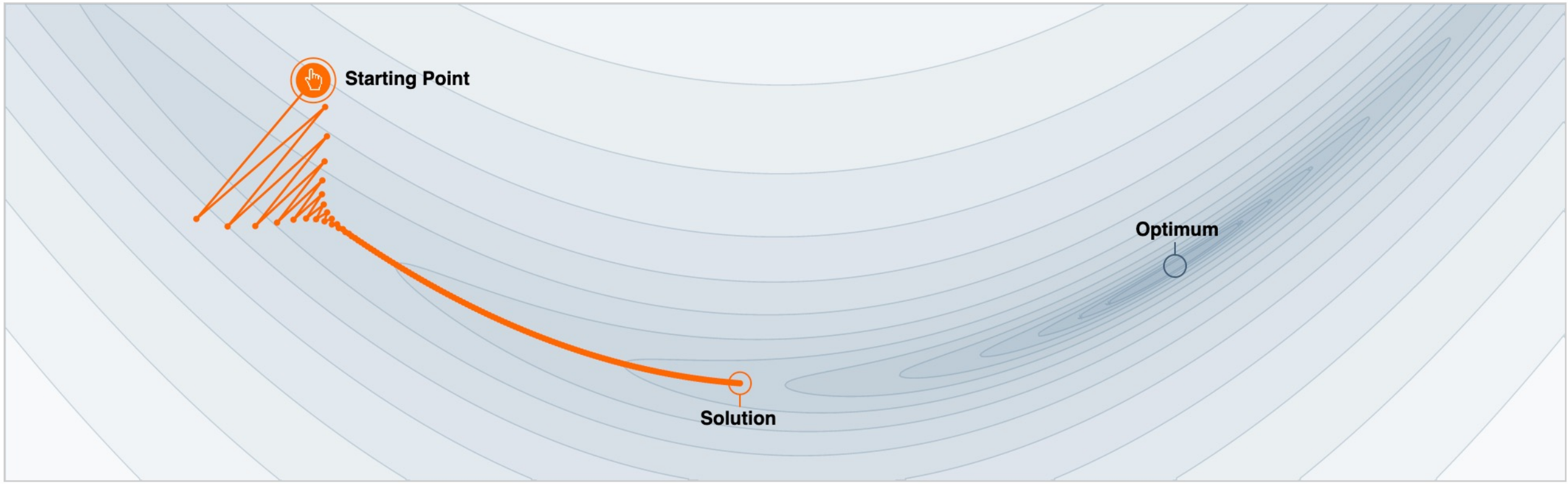
GD with Momentum

Deep neural networks have very complex error profiles. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. This problem is typical in deep learning architecture.

Momentum is one method of speeding the convergence along a narrow ravine.

GD with Momentum



Step-size α = 0.0030

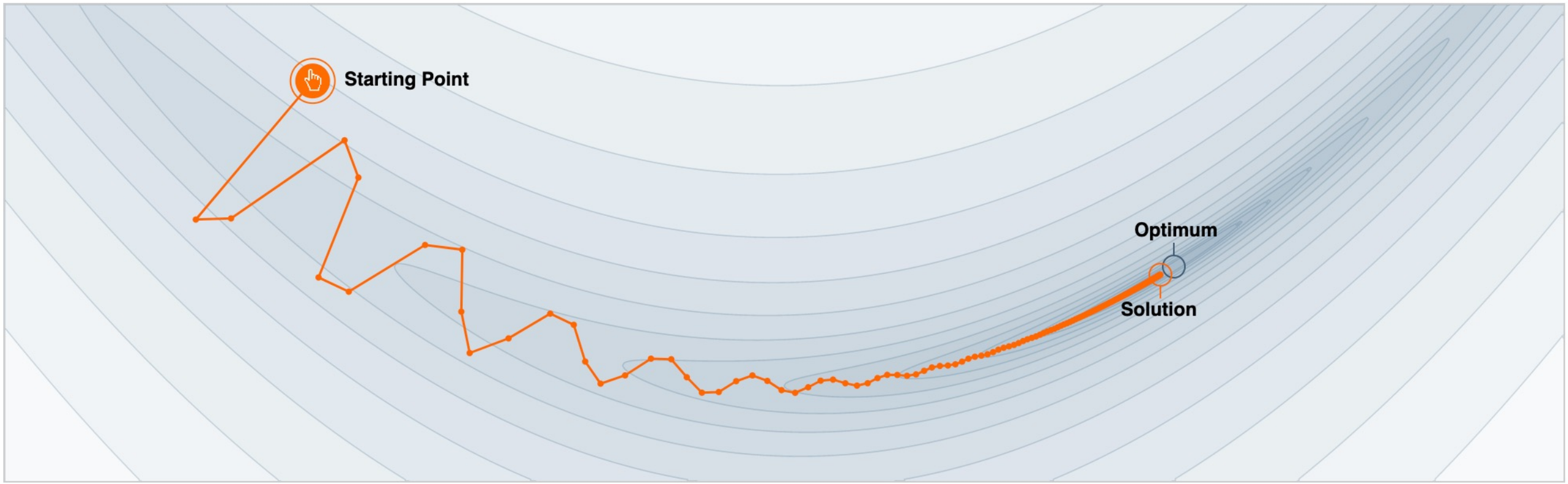
0 0.003 0.006

Momentum β = 0.0

0.00 0.500 0.990

Why Momentum Really Works: https://distill.pub/2017/momentum/

GD with Momentum



Step-size α = 0.0030

0 0.003 0.006

Momentum β = 0.80

0.00 0.500 0.990

Why Momentum Really Works: https://distill.pub/2017/momentum/

GD with Momentum

Momentum update is given by:

$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

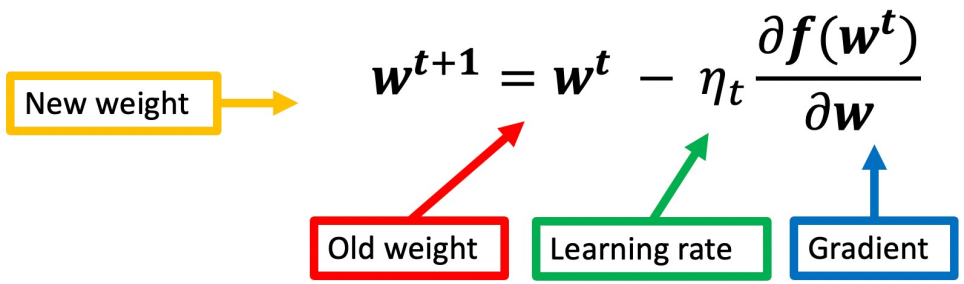
where $\boldsymbol{V}$ is known as the velocity term and has the same dimension as the weight vector $\boldsymbol{W}$.

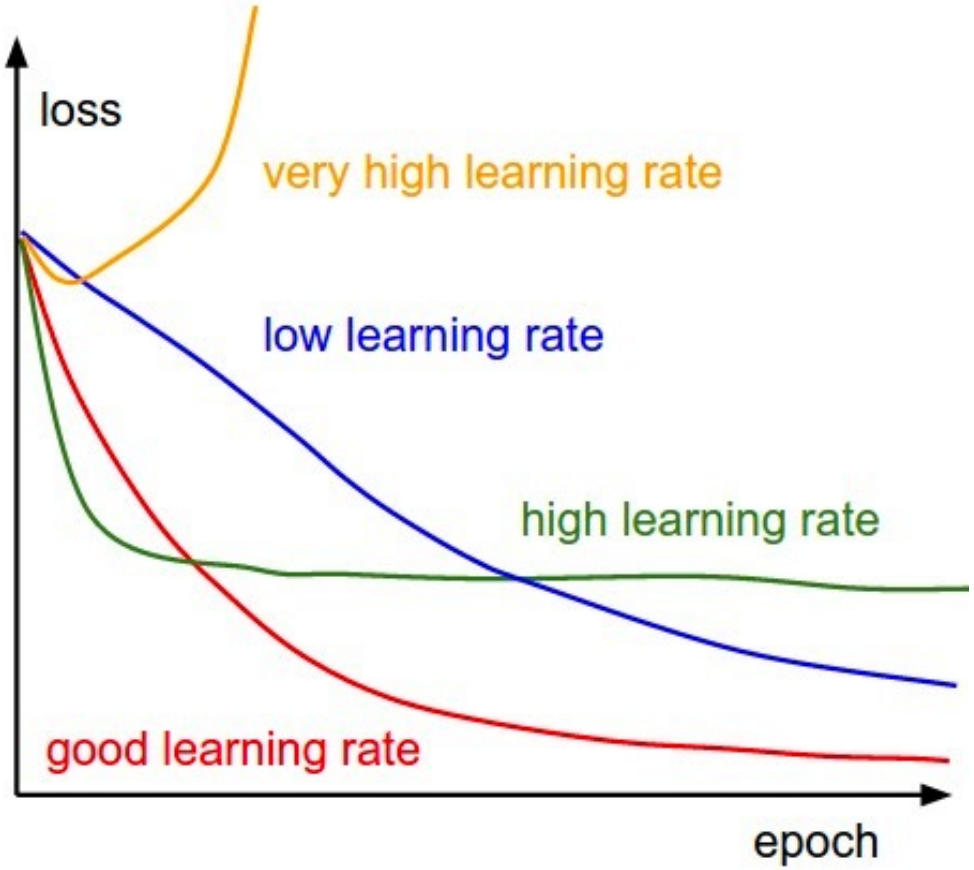
The momentum parameter $\gamma \in [0,1]$ indicates how many iterations the previous gradients are incorporated into the current update.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

Learning rate





Learning rate

The learning rate, often noted α or sometimes η, indicates at which pace the weights get updated. It can be fixed or adaptively changed.

The current most popular method is called Adam, which is a method that adapts the learning rate.

Learning rate

- **Adaptive learning rates**
  - Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution.
  - While Adam optimizer is the most commonly used technique, others can also be useful.

- **Algorithms with adaptive learning rates:**
  - AdaGrad torch.optim.Adagrad()
  - RMSprop torch.optim.RMSprop()
  - Adam torch.optim.Adam()

Adam: A Method for Stochastic Optimization, ICLR 2014

https://pytorch.org/docs/stable/optim.html

Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to start with a large learning factor and then gradually reducing it.

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and $\alpha(\infty) = 0$.

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.

Learning trajectory of a neural network minimizing non-convex cost function passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. This improves the learning rates, especially in the convex regions of error function.

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + (\nabla_{\boldsymbol{W}} J)^2$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

In other words, learning rate:

$$\tilde{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region.

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\sqrt{\varepsilon + \boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

The decay constant ρ controls the length of the moving average of gradients.

Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer combines RMSprop and momentum methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

Momentum term: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\nabla_{\boldsymbol{W}} J$

Learning rate term: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)(\nabla_{\boldsymbol{W}} J)^2$

$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1}$$

$$\boldsymbol{r} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{s}$$

Note that $\boldsymbol{s}$ adds the momentum and $\boldsymbol{r}$ contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001, \rho_1 = 0.9, \rho_2 = 0.999$, and $\epsilon = 10^{-8}$

Outline

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
- Training basics
  - Weight initialization
  - Optimizers
  - Prevent overfitting
- Batch normalization

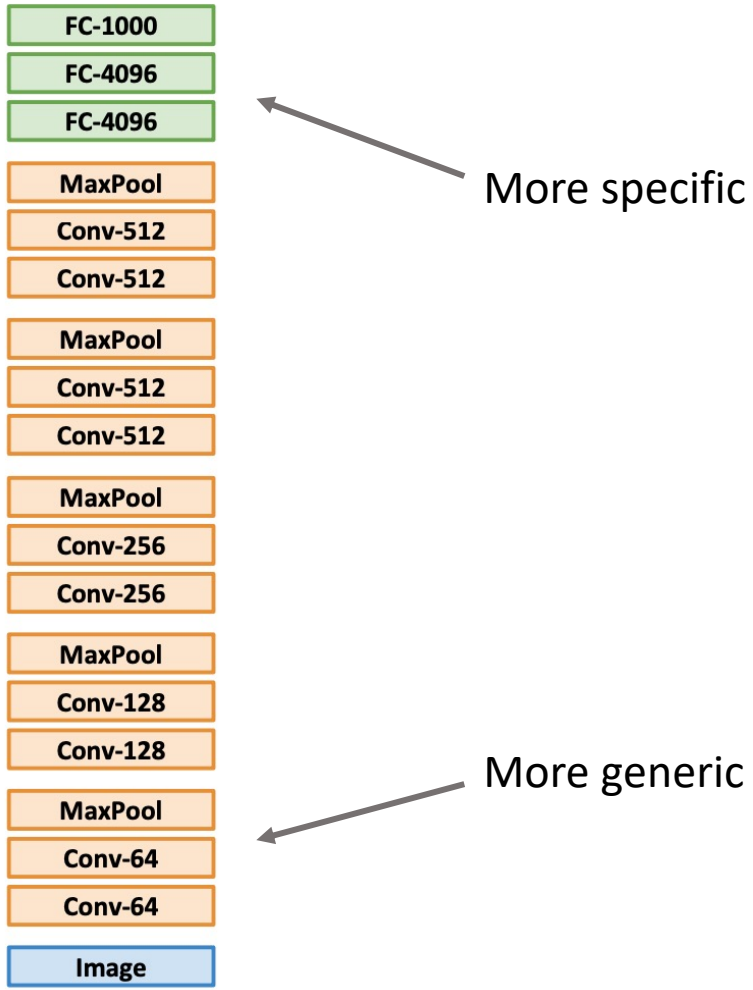
Prevent Overfitting

Why overfitting?

- This happens when our model is too complex and too specialized on a small number of training data.

- Increase the size of the data, remove outliers in data, reduce the complexity of the model, reduce the feature dimension

Transfer learning



In a network with an N-dimensional softmax output layer that has been successfully trained toward a supervised classification objective, each output unit will be specific to a particular class

When trained on images, deep networks tend to learn first-layer features that resemble either Gabor filters or color blobs. These first-layer features are general.

Yosinski et al., How transferable are features in deep neural networks? NIPS'14

Transfer learning

Pre-training + Fine-tuning

Step 1: Pre-training on large-scale dataset like ImageNet

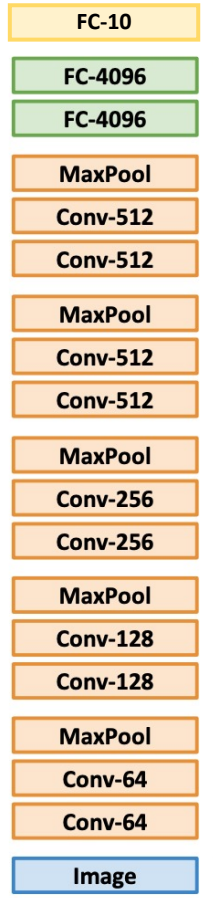
Transfer weights

Step 2: Use pre-trained network as initialization

Remove last layer

Freeze these

Step 3: Fine-tuning

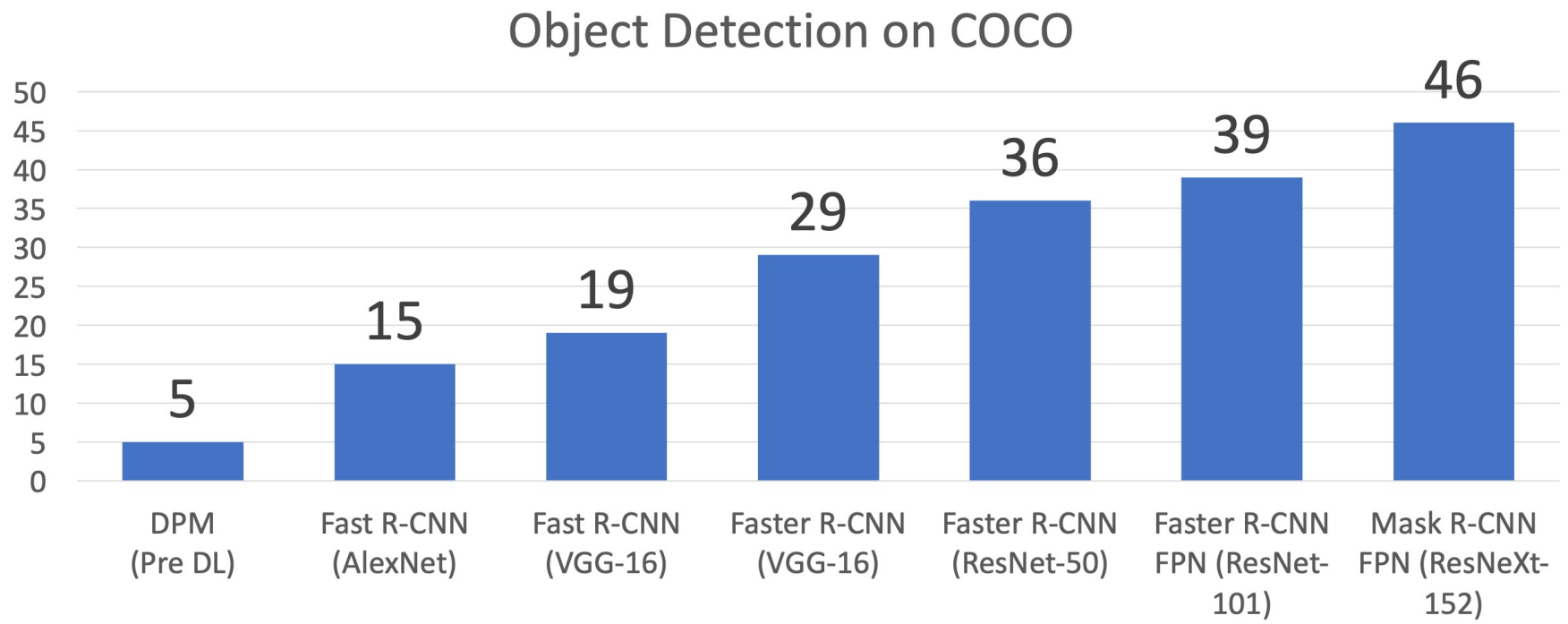


Add new layer correspond to the target class number

Train on new data

Transfer learning

Architecture matters

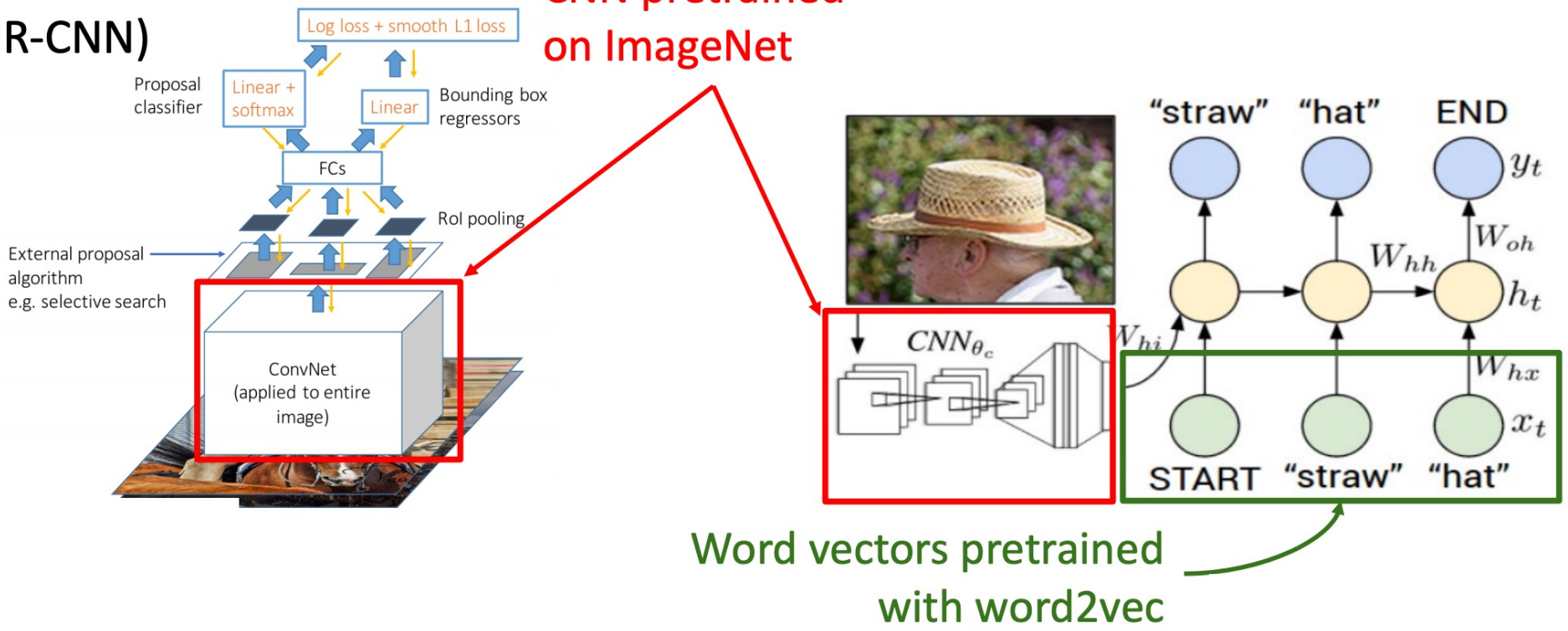


Ross Girshick, “The Generalized R-CNN Framework for Object Detection”, ICCV 2017 Tutorial on Instance-Level Visual Recognition

Transfer learning

Transfer learning is pervasive

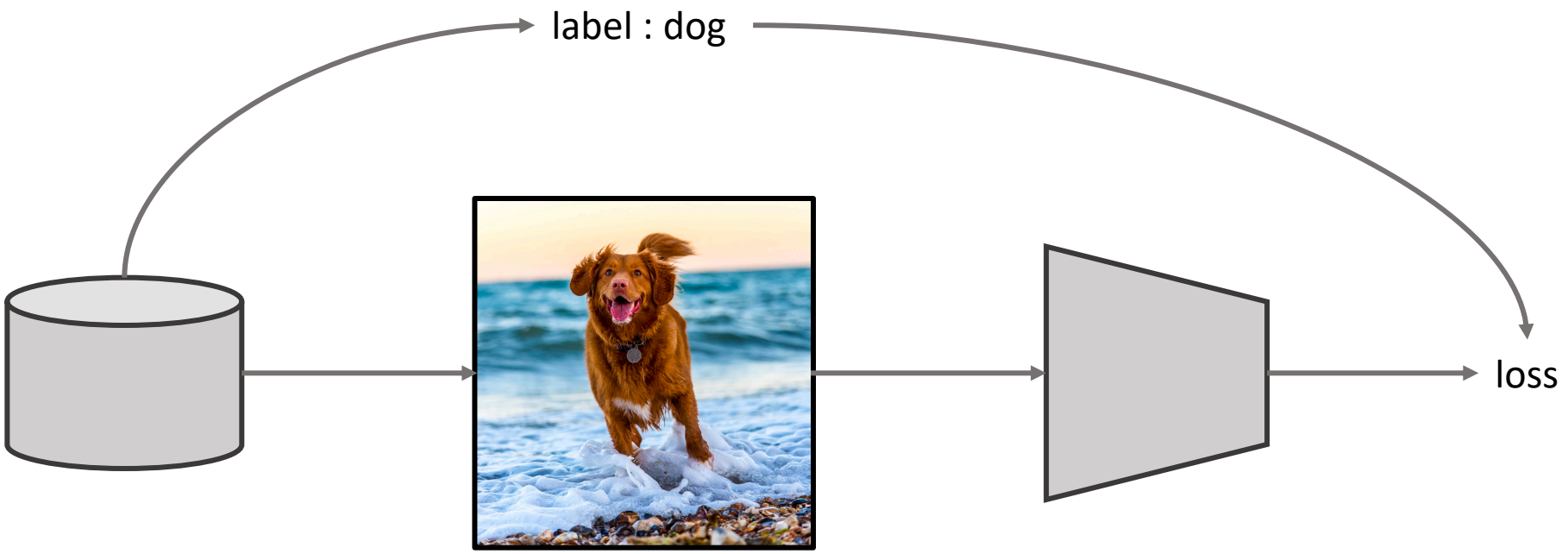
Object Detection (Fast R-CNN)



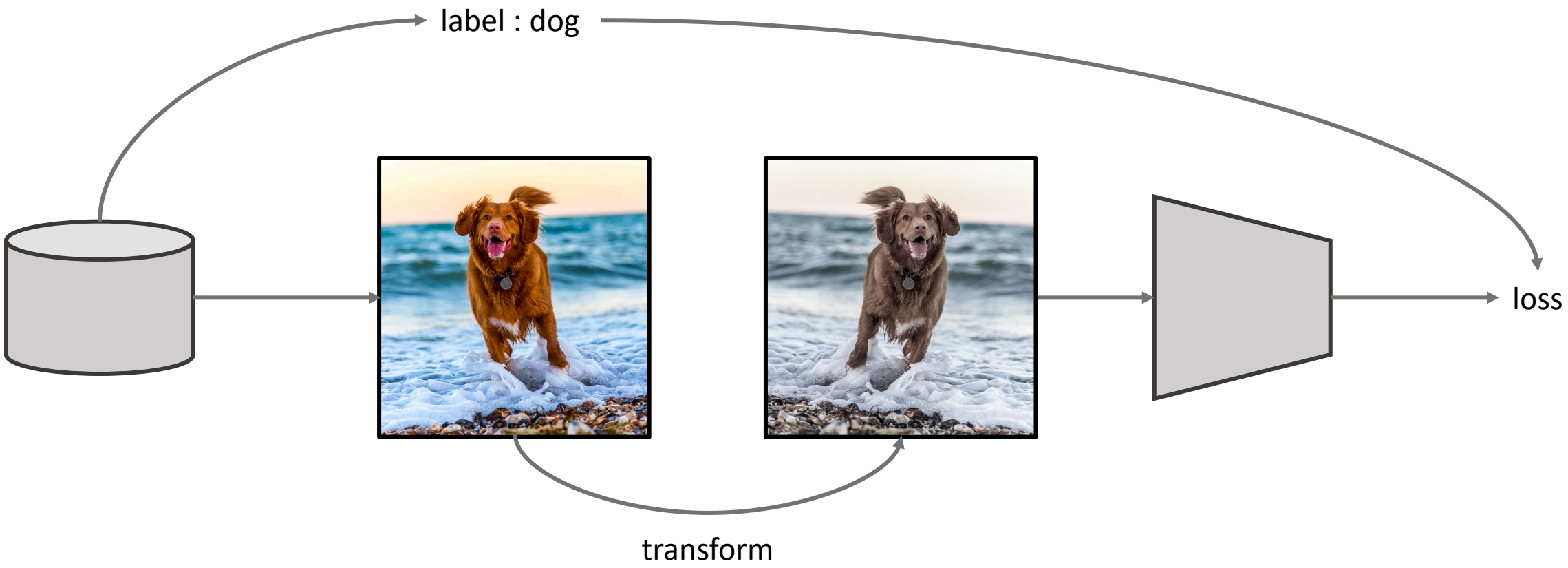
Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Data augmentation



Data augmentation



Data augmentation

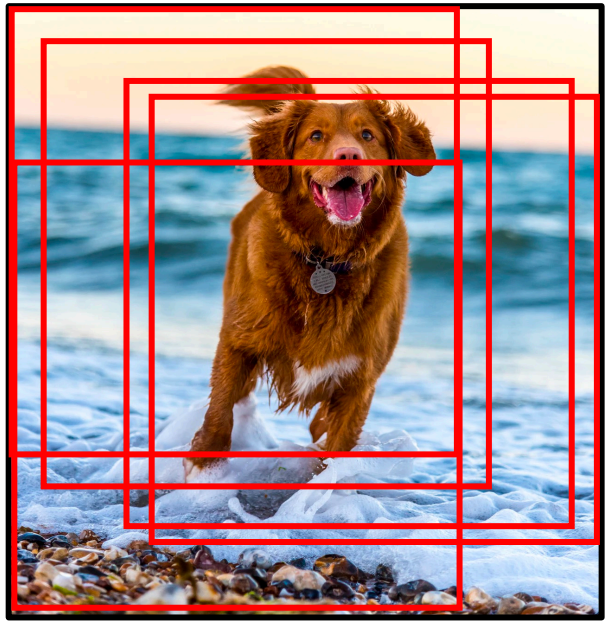
Horizontal flip

Data augmentation

Random crops and scales

Training:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data augmentation

Color jitter

Simple :

1. Randomize contrast and brightness

Complex:

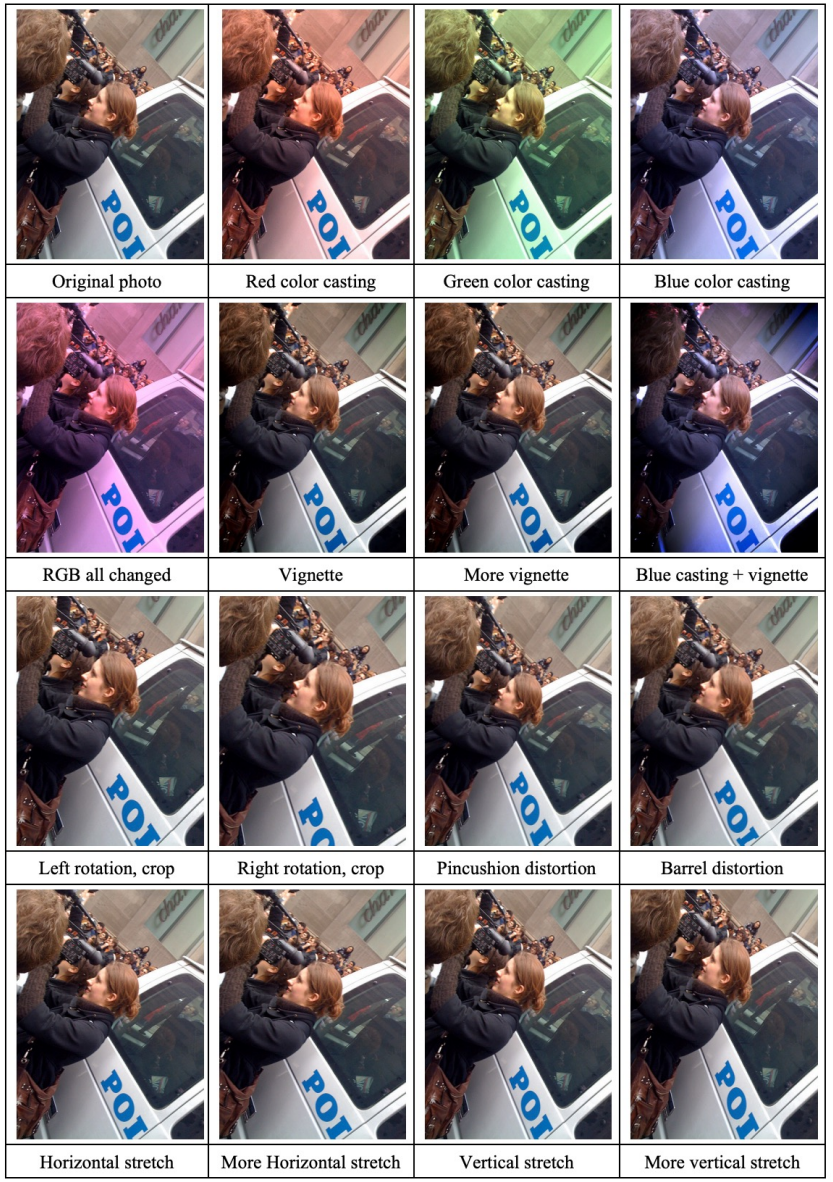
1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(Used in AlexNet, ResNet, etc.)

Data augmentation

There are many more data augmentation schemes

Random mix/combinations of:
translation - rotation - stretching - shearing, - lens distortions ….



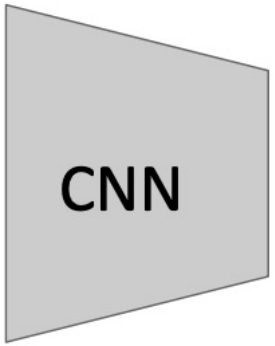
Ren Wu et al., Deep Image: Scaling up Image Recognition, https://arxiv.org/pdf/1501.02876v2.pdf?

Mixup

Training: Train on random blends of images

Testing: Use original images

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog



Target label:
cat: 0.4
dog: 0.6

Hongyi Zhang et al., mixup: Beyond Empirical Risk Minimization, ICLR 2018

Mixup

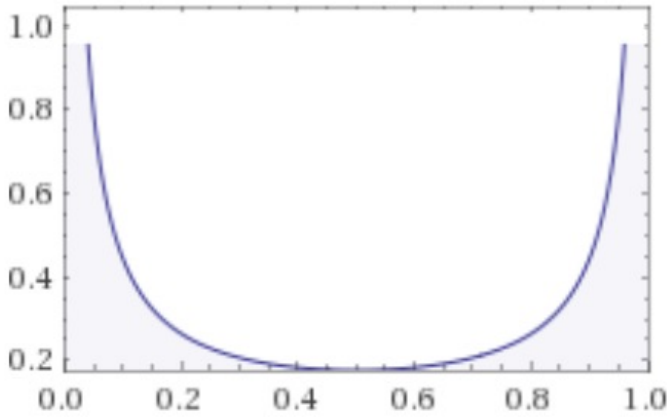
Training: Train on random blends of images

Testing: Use original images

Forming a convex combination between two training data instances, as well as changing the associated label to the corresponding convex combination of the original two labels

The blending weight λ is generated from a symmetric beta distribution with parameter α

There is a very high probability of picking values close to 0 or 1 (in which case the image is almost from 1 category) and then a somewhat constant probability of picking something in the middle



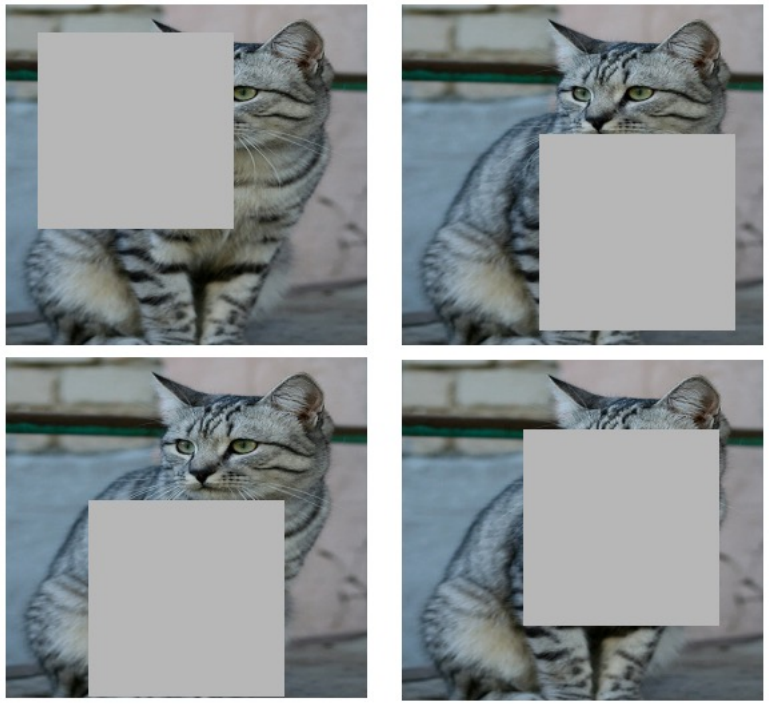
$Beta(\alpha, \alpha)$

Hongyi Zhang et al., mixup: Beyond Empirical Risk Minimization, ICLR 2018

Cutout

Training: Set random images regions to 0

Testing: Use the whole image



Researchers have shown that the feature removal strategies improve generalization and localization by letting a model attend not only to the most discriminative parts of objects, but rather to the entire object region

Difference from Dropout

- units are dropped out only at the input layer of a CNN, rather than in the intermediate feature layers
- drop out contiguous sections of inputs rather than individual pixels

Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”, arXiv 2017

Cutmix

Training: Patches are cut and pasted among training images

Testing: Use the whole image

	ResNet-50	Mixup [48]	Cutout [3]	CutMix
Image				
Label	Dog 1.0	Dog 0.5 Cat 0.5	Dog 1.0	Dog 0.6 Cat 0.4

Argue that the removal of informative pixels is not desirable

Ground truth labels are also mixed proportionally to the area of the patches

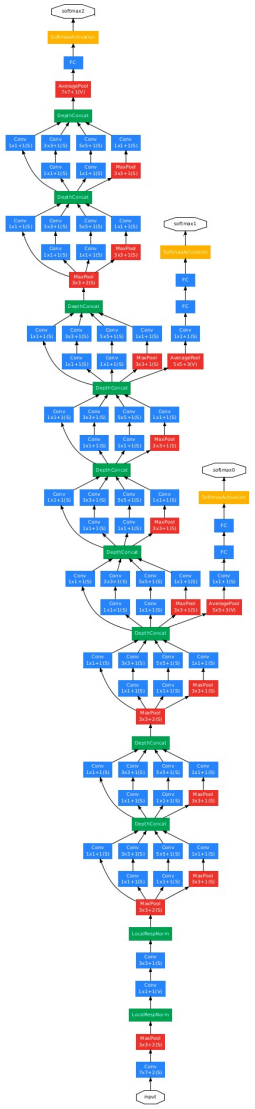
Yun et al., “CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features”, ICCV 2019

Outline

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
- Training basics
  - Weight initialization
  - Optimizers
  - Prevent overfitting
- Batch normalization

Batch Normalization

GoogLeNet



Proposed by Google in 2014, winning the ImageNet competition that year

Apart from the inception structure, there is another important technique called **batch normalization**

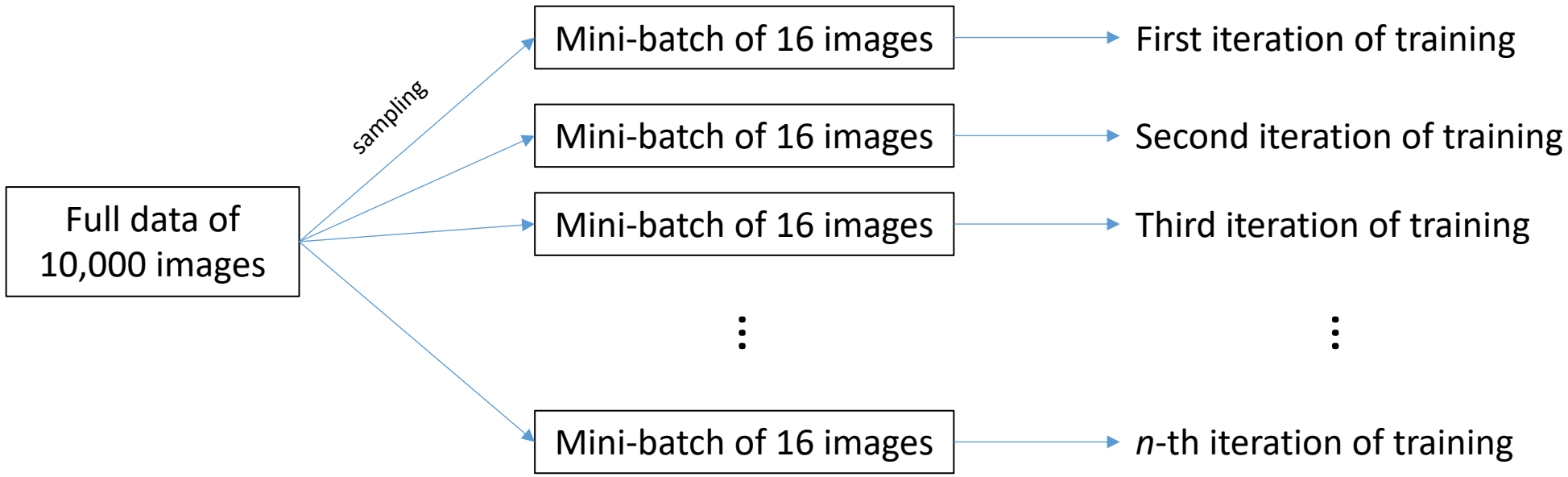
Problem: deep networks are very hard to train!

Main idea: “Normalize” the outputs of a layer so they have zero mean and unit variance

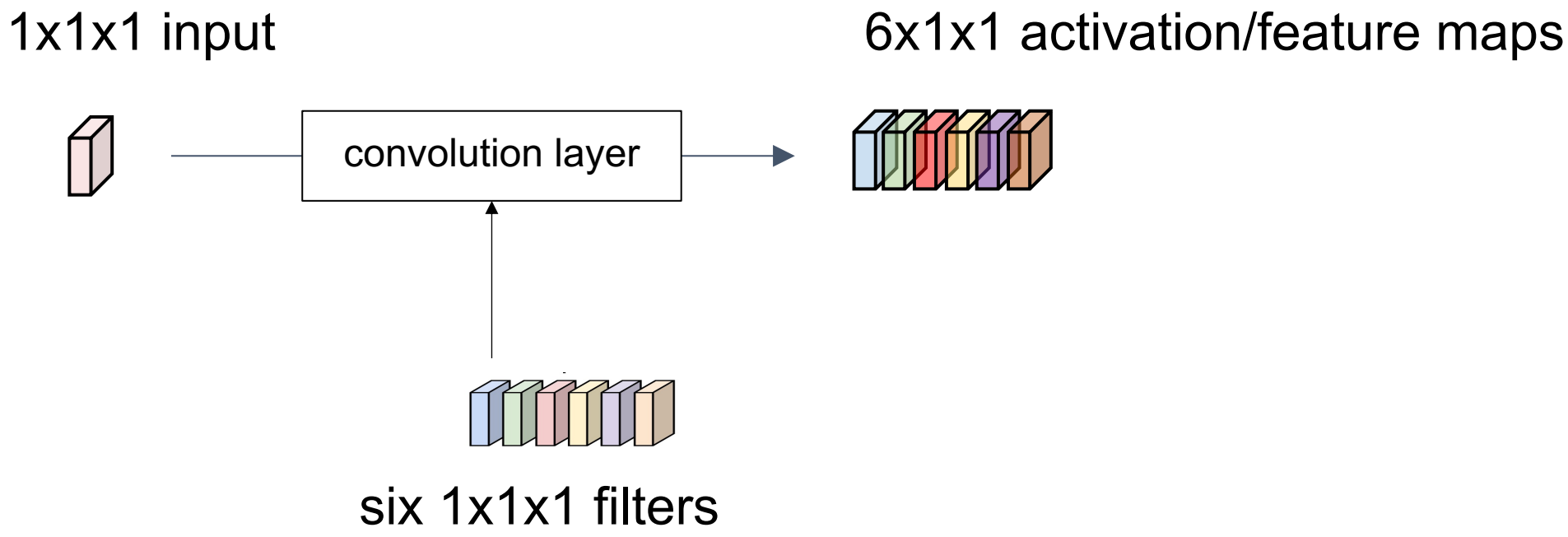
Effect: allowing higher learning rates and reducing the strong dependence on initialization

Mini-batch

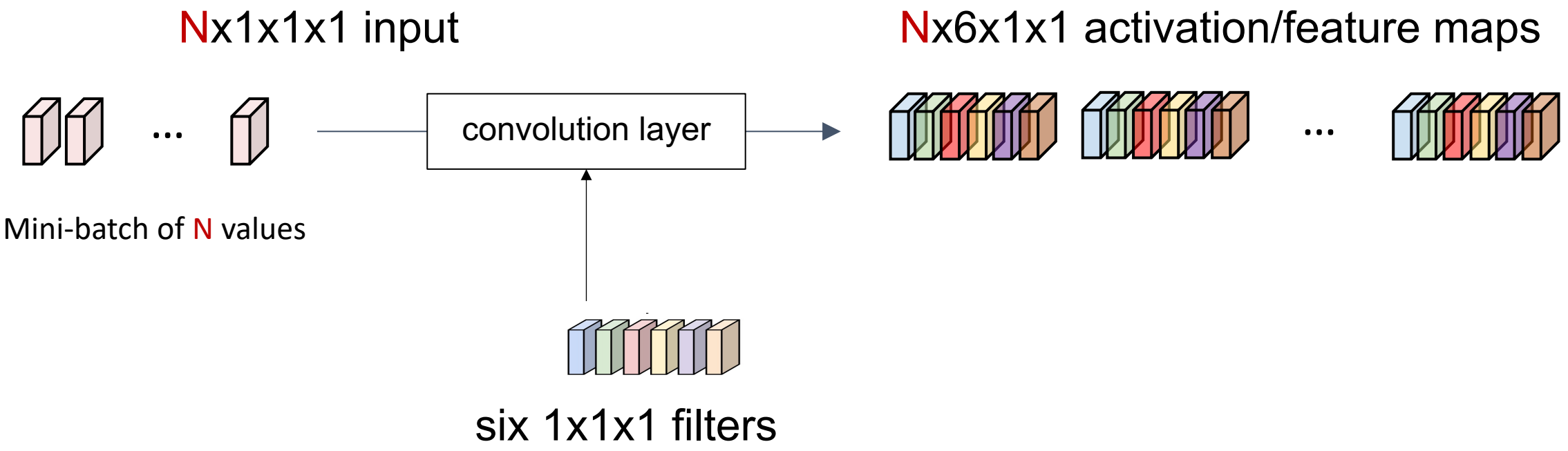
- What is mini-batch?
  - A subset of all data during one iteration to compute the gradient



Mini-batch


1x1x1 input
convolution layer
6x1x1 activation/feature maps
six 1x1x1 filters

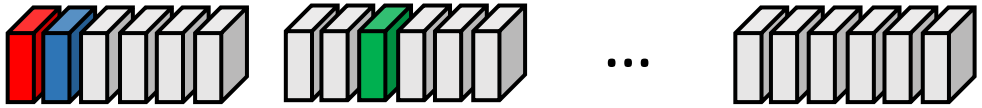
Mini-batch


Nx1x1x1 input
Nx6x1x1 activation/feature maps
convolution layer
Mini-batch of N values
six 1x1x1 filters

Batch Normalization

Let's arrange the activations of the mini batch in a matrix of $N \times D$

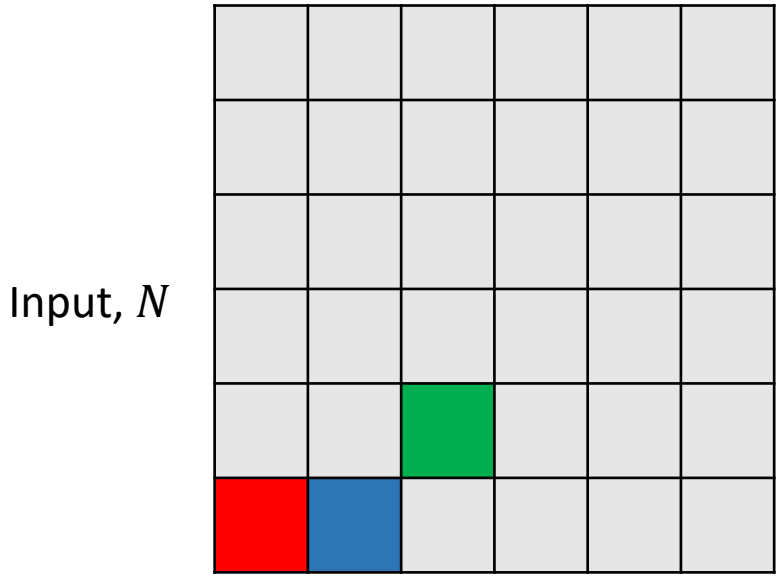
Nx6x1x1 activation/feature maps



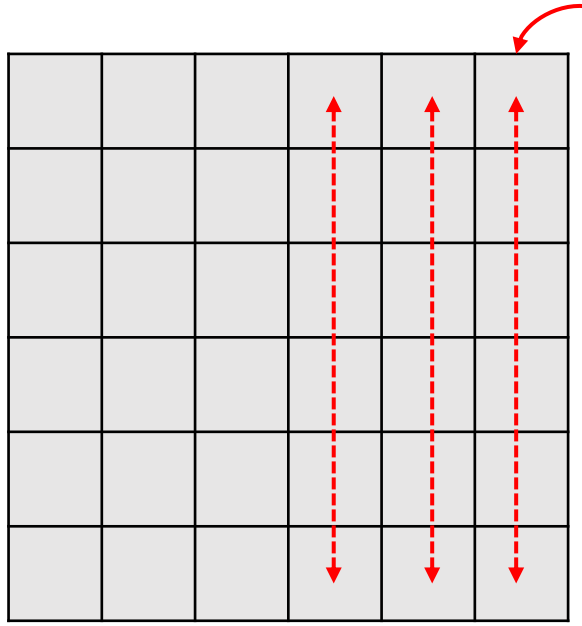
Activation of the first input in the mini-batch

Activation of the second input in the mini-batch

Activation of the N-th input in the mini-batch

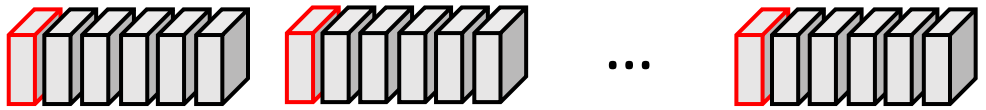


Batch Normalization

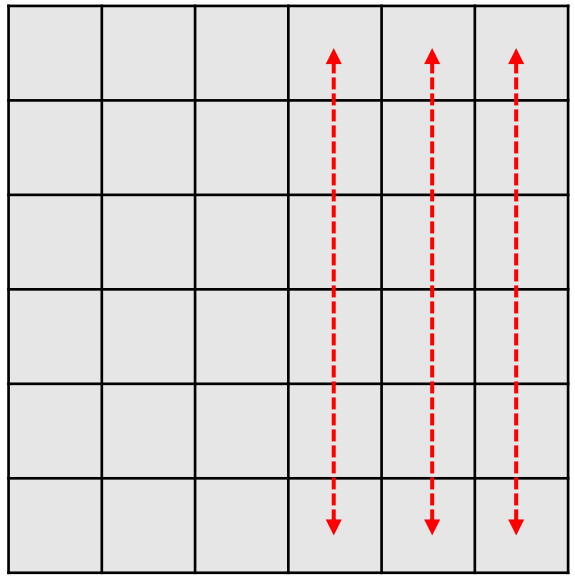


The goal of batch normalization is to normalize the values across each column so that the values of the column have zero mean and unit variance

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below



Batch Normalization - Training



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}\left(x_{i,j} - \mu_j\right)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the **normalized** values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization – Test Time

Input

Channel, D

Problem: Estimates depend on minibatch; can't do this at test-time

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization – Test Time

Input

Channel, D

Average of values seen during training

Per-channel mean, shape is $1 \times D$

Average of values seen during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization – Test Time

Input

Channel, D

Average of values seen during training

Per-channel mean, shape is $1 \times D$

Average of values seen during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization

Batch Normalization for
fully-connected networks

$x: N \times D$

Normalize ↓

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

Normalize ↓ ↓ ↓

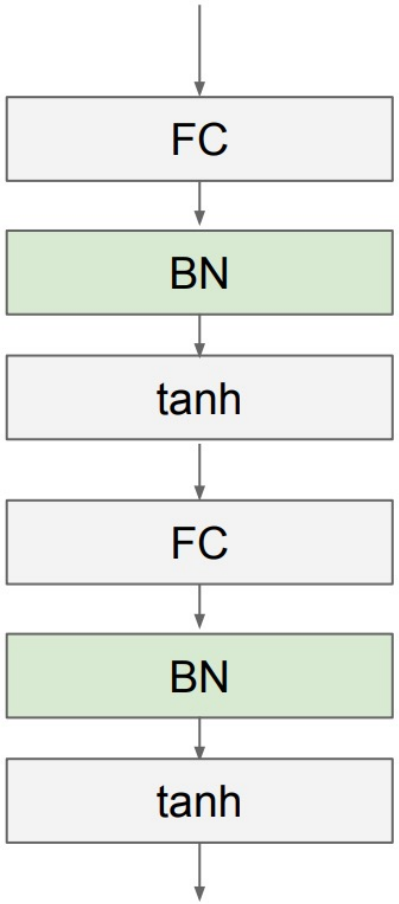
Normalize also on the spatial dimensions

$\mu, \sigma: 1 \times C \times 1 \times 1$

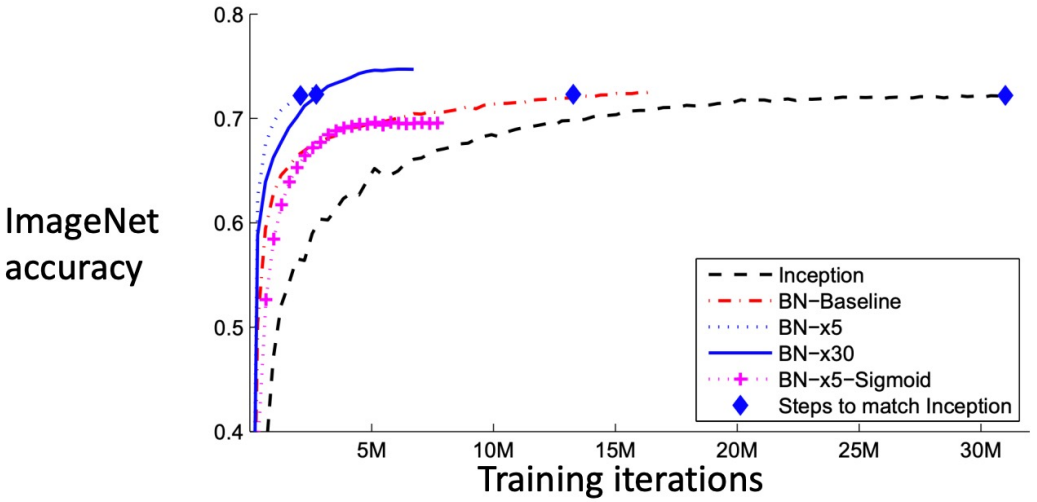
$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

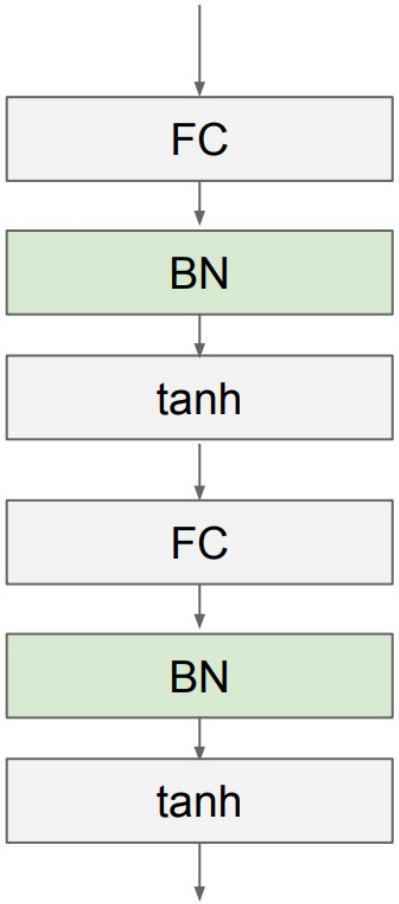
Batch Normalization



- It is usually done after a fully connected/convolutional layer and before a non-linearity layer
- Zero overhead at test-time: can be fused with conv
- Allows higher learning rates, faster convergence
- Networks becomes more robust to initialization



Batch Normalization



- It is usually done after a fully connected/convolutional layer and before a non-linearity layer
- Zero overhead at test-time: can be fused with conv
- Allows higher learning rates, faster convergence
- Networks becomes more robust to initialization
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a very common source of bugs!