Chapter 3: Convolutional Neural Networks

Mathematical Foundations of Deep Neural Networks

Fall 2022

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Fully connected layers

Advantages of fully connected layers:

- Simple.
- Very general, in theory. (Sufficiently large MLPs can learn any function, in theory.)

Disadvantage of fully connected layers:

- Too many trainable parameters.
- Does not encode shift equivariance/invariance and therefore has poor inductive bias. (More on this later.)

Shift equivariance/invariance in vision

Many tasks in vision are equivariant/invariant with respect shifts/translations.

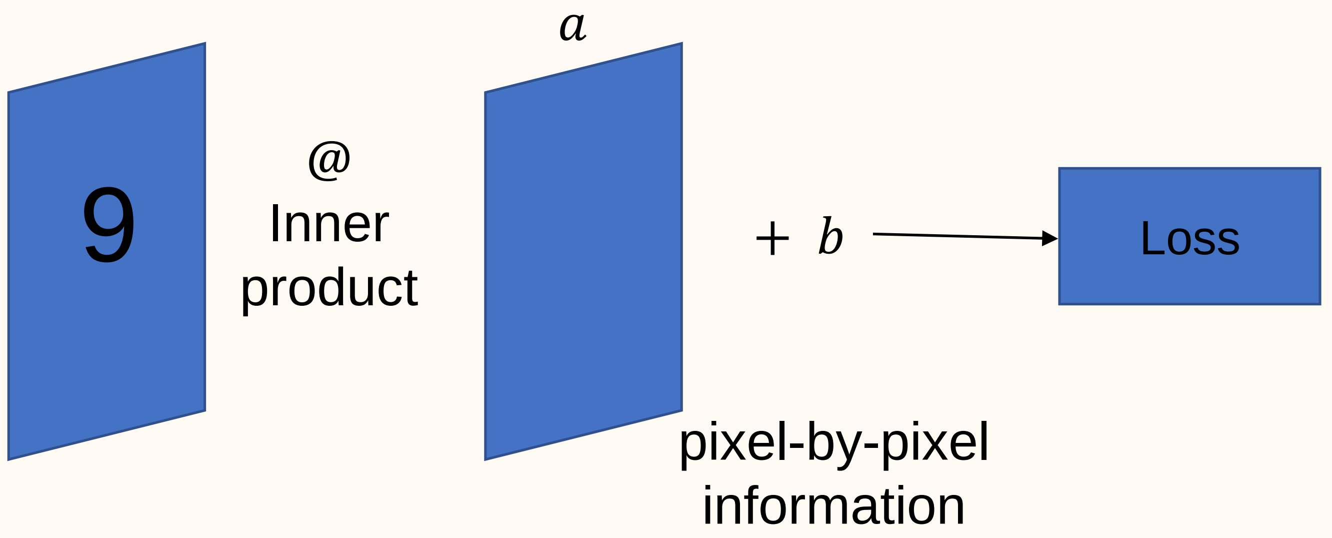
Cat

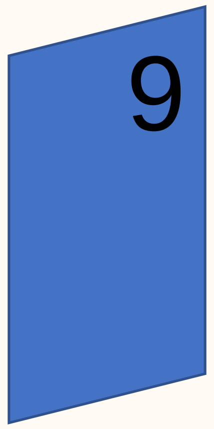
Still a Cat

Roughly speaking, equivariance/invariance means shifting the object does not change the meaning (it only changes the position).

Shift equivariance/invariance in vision

Logistic regression (with a single fully connected layer) does not encode shift invariance.

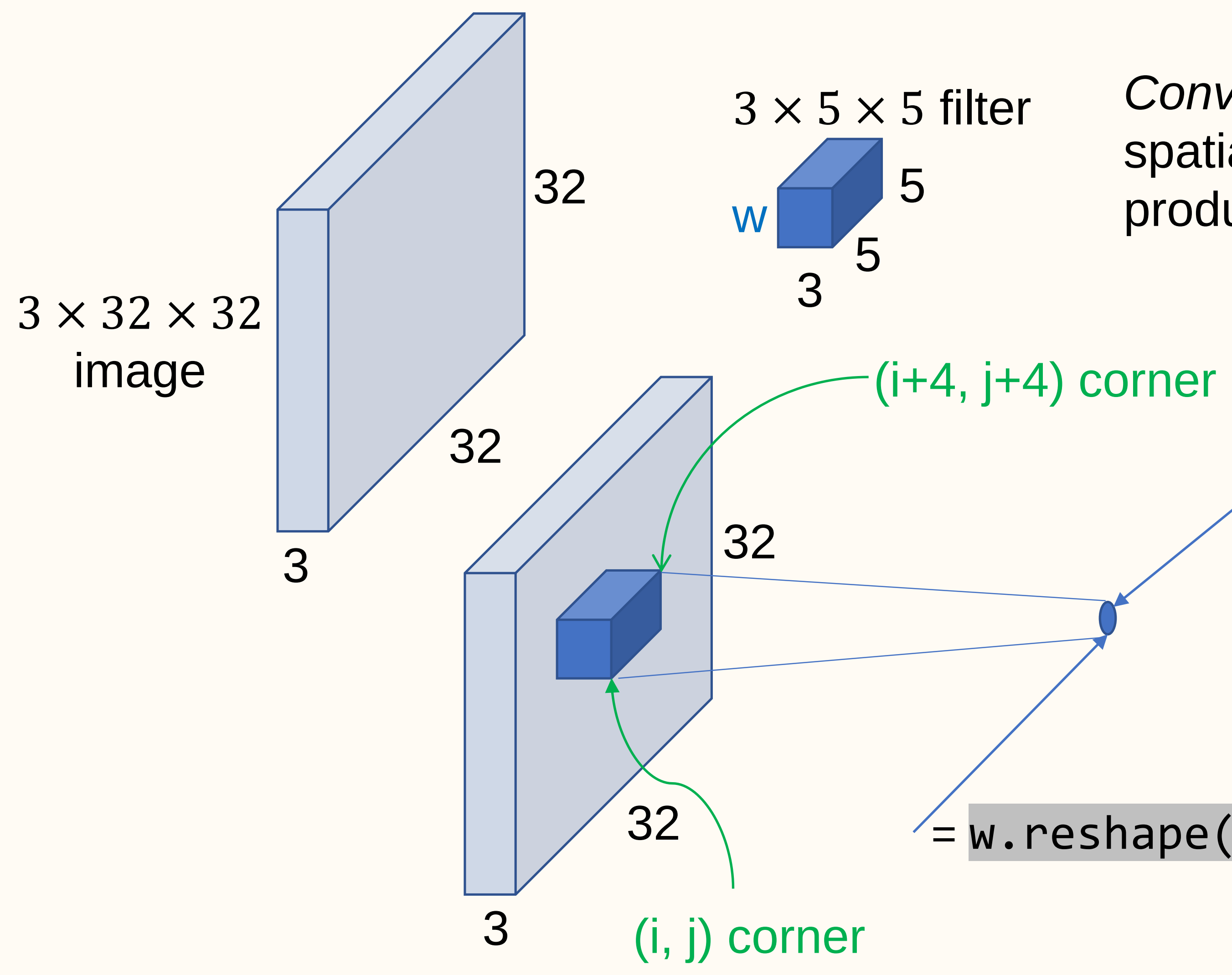




Translating digit changes output significantly

Since convolution is equivariant with respect to translations, constructing neural network layers with them is a natural choice.

Convolution



Convolve a filter with an image: slide the filter spatially over the image and compute dot products.

Take a $3 \times 5 \times 5$ chunk of the image and take the inner product with w and add bias b.

= `w.reshape(-1)@image[:,i:i+5,j:j+5].reshape(-1)+b`

Multiple filters

image
convolved
with
1 filter

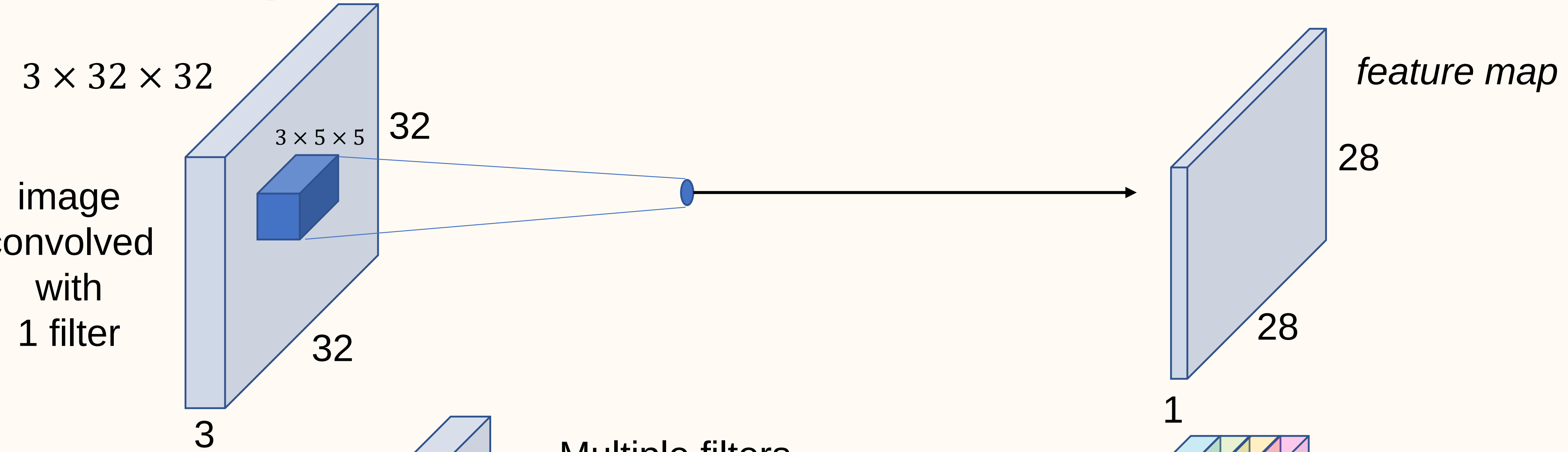
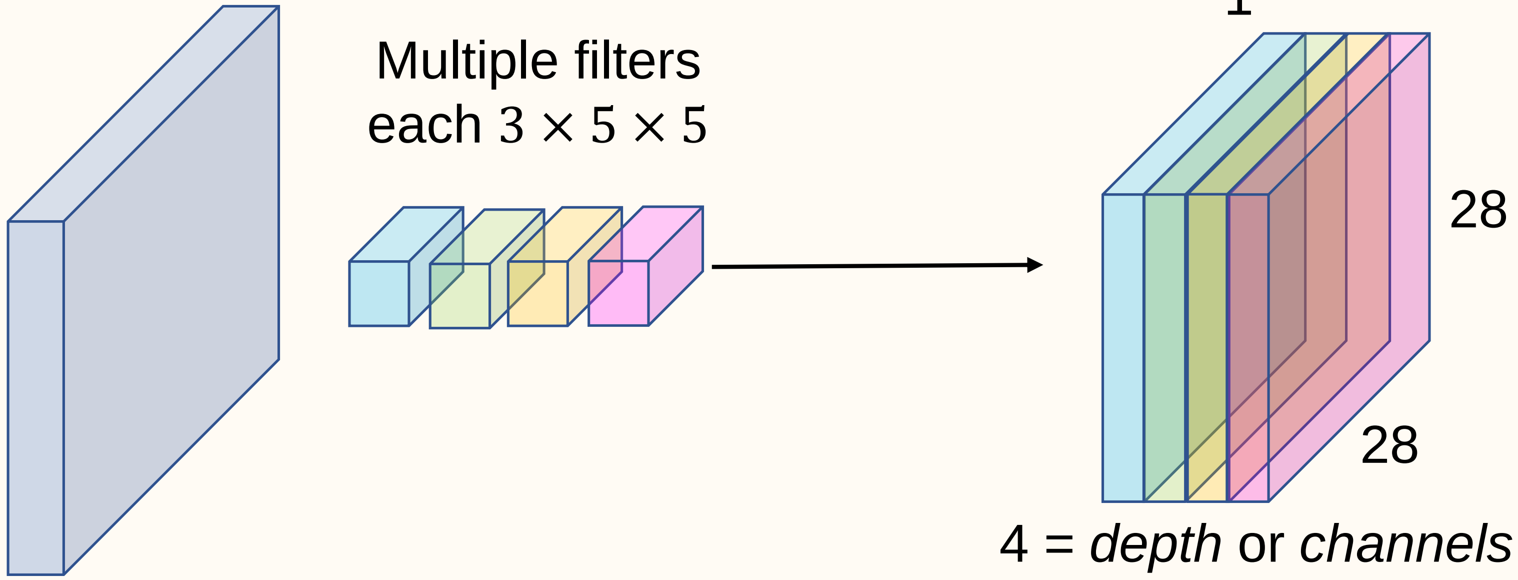


image
convolved
with
4 filters



4 = *depth* or *channels* of feature map

2D convolutional layer: Formal definition

Input tensor: $X \in \mathbb{R}^{B\times C_{\text{in}}\times m\times n}$, B batch size, C_{in} # of input channels, m, n # of vertical and horizontal indices.

Output tensor: $Y \in \mathbb{R}^{B\times C_{\text{out}}\times (m-f_1+1)\times(n-f_2+1)}$, B batch size, C_{out} # of output channels.

With filter $w \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times f_1\times f_2}$, bias $b \in \mathbb{R}^{C_{\text{out}}}$, $k = 1, \dots B$, $\ell = 1, \dots, C_{\text{out}}$, $i = 1, \dots, m - f_1 + 1$, and $j = 1, \dots, n - f_2 + 1$:

$$Y_{k,\ell,i,j} = \sum_{\gamma=1}^{C_{\text{in}}} \sum_{\alpha=1}^{f_1} \sum_{\beta=1}^{f_2} w_{\ell,\gamma,\alpha,\beta} X_{k,\gamma,i+\alpha-1,j+\beta-1} + b_\ell$$

Operation is independent across elements of the batch. The vertical and horizontal indices are referred to as *spatial dimensions*. If `bias=False`, then $b = 0$.

Notes on convolution

Mind the indexing. In math, indices start at 1. In Python, indices start at 0.

1D conv is commonly used with 1D data, such as audio.

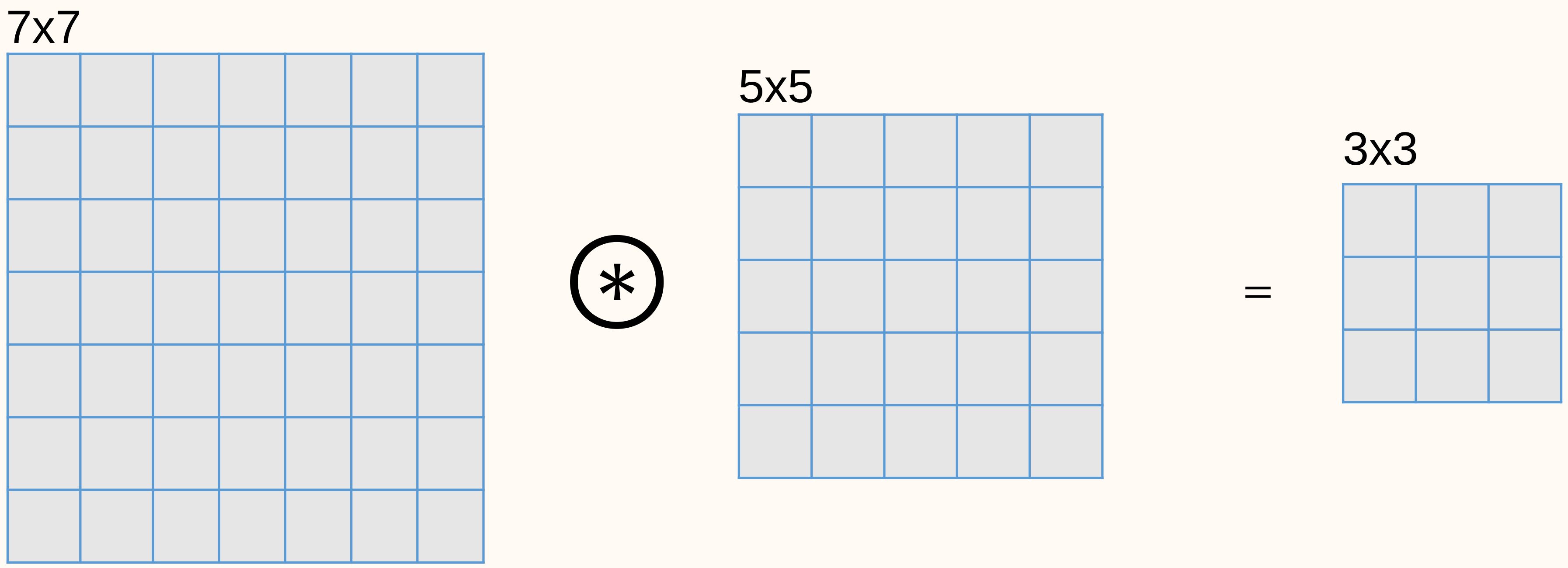
3D conv is commonly used with 3D data, such as video.

1D and 3D conv are defined analogously.

Zero padding

($C \times 7 \times 7$ image) $\circledast$ ($C \times 5 \times 5$ filter) = ($1 \times 3 \times 3$ feature map).

Spatial dimension 7 reduced to 3.

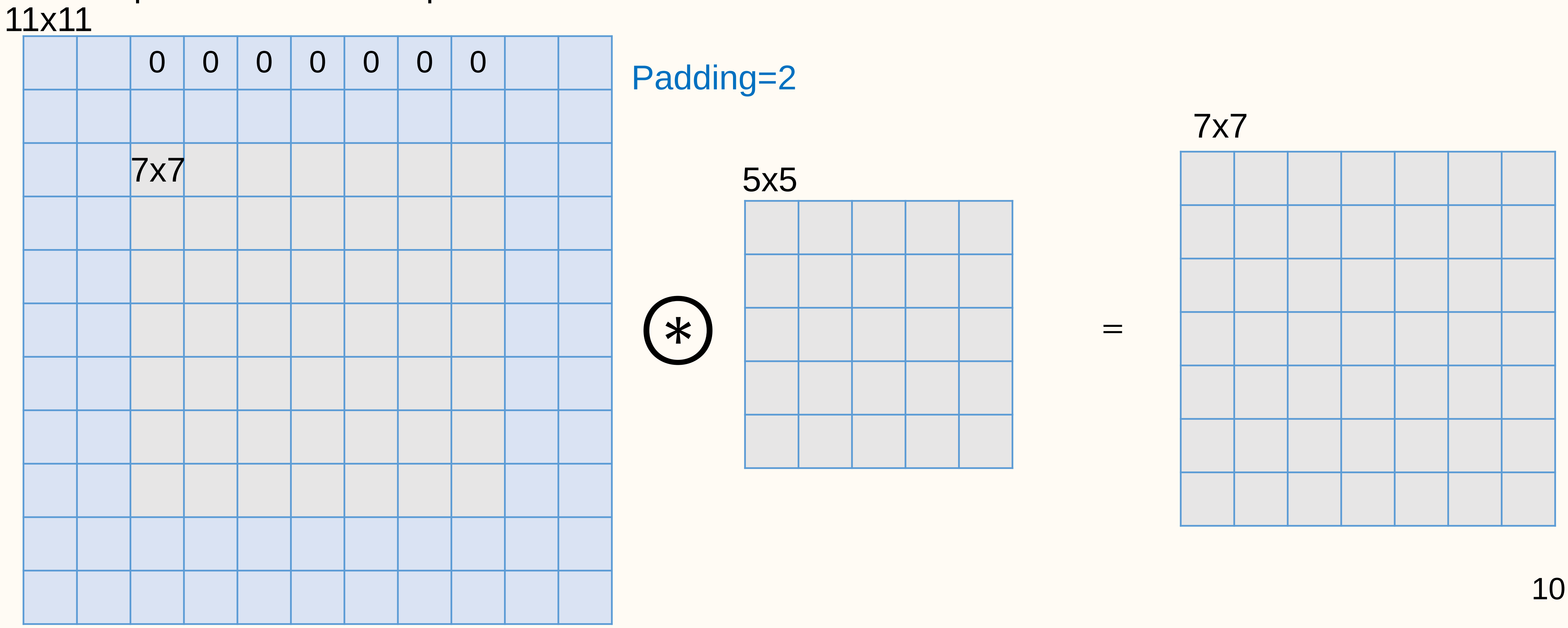


We write $\circledast$ to denote convolution.

Zero padding

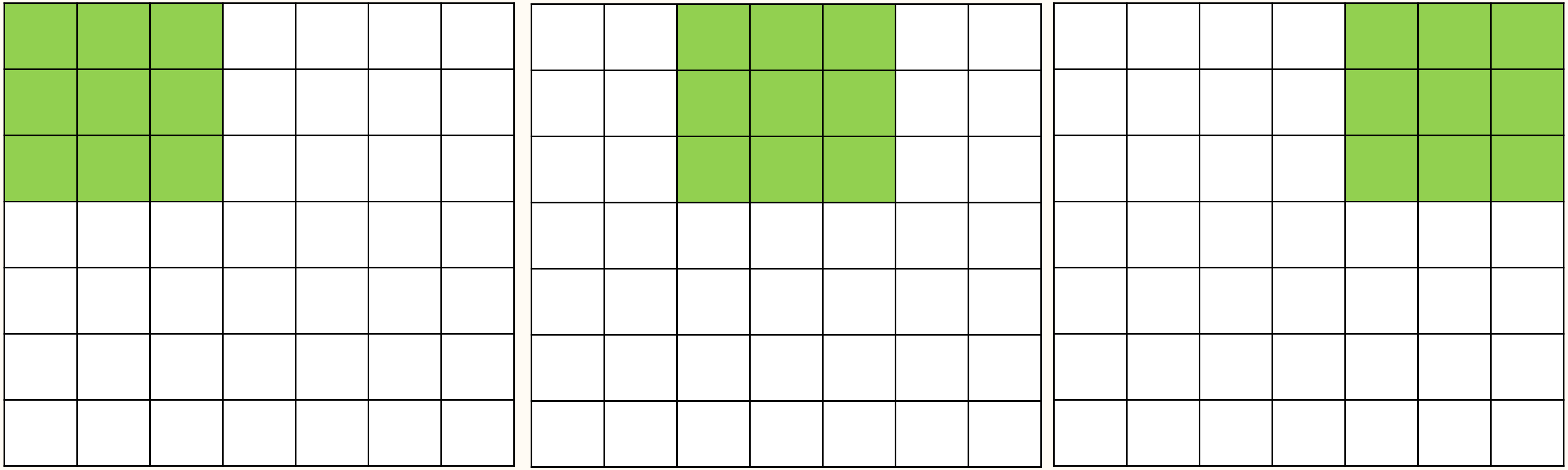
($C \times 7 \times 7$ image with zero padding = 2) $\circledast$ ($C \times 5 \times 5$ filter) = ($1 \times 7 \times 7$ feature map).

Spatial dimension is preserved.



Stride

(7x7 image) ⊛ (3x3 filter with stride 2) = (output 3x3).
(With stride 1, output is 5x5.)



If stride 3, dimensions don't fit.
7x7 image with zero padding of 1 becomes 9x9 image.
(7x7 image, padding of 1) ⊛ (3x3 filter) with stride 3 does fit.

Convolution summary

Input $C_{\text{in}} \times W_{\text{in}} \times H_{\text{in}}$

Conv layer parameters

- C_{out} filters
- F spatial extent ($C_{\text{in}} \times F \times F$ filters)
- S stride
- P padding

Output $C_{\text{out}} \times W_{\text{out}} \times H_{\text{out}}$

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - F + 2P}{S} + 1 \right\rfloor$$

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - F + 2P}{S} + 1 \right\rfloor$$

$\lfloor \cdot \rfloor$ denotes the floor (rounding down) operation. To avoid the complication of this floor operation, it is best to ensure the formula inside evaluates to an integer.

Number of trainable parameters:

$$\underbrace{F^2 C_{\text{in}} C_{\text{out}}}_{\text{filters}} + \underbrace{C_{\text{out}}}_{\text{biases}}$$

Make sure you are able to derive these formulae yourself.

Aside: Geometric deep learning

More generally, given a group $\mathcal{G}$ encoding a symmetry or invariance, one can define operations “equivariant” with respect $\mathcal{G}$ and construct *equivariant neural networks*.

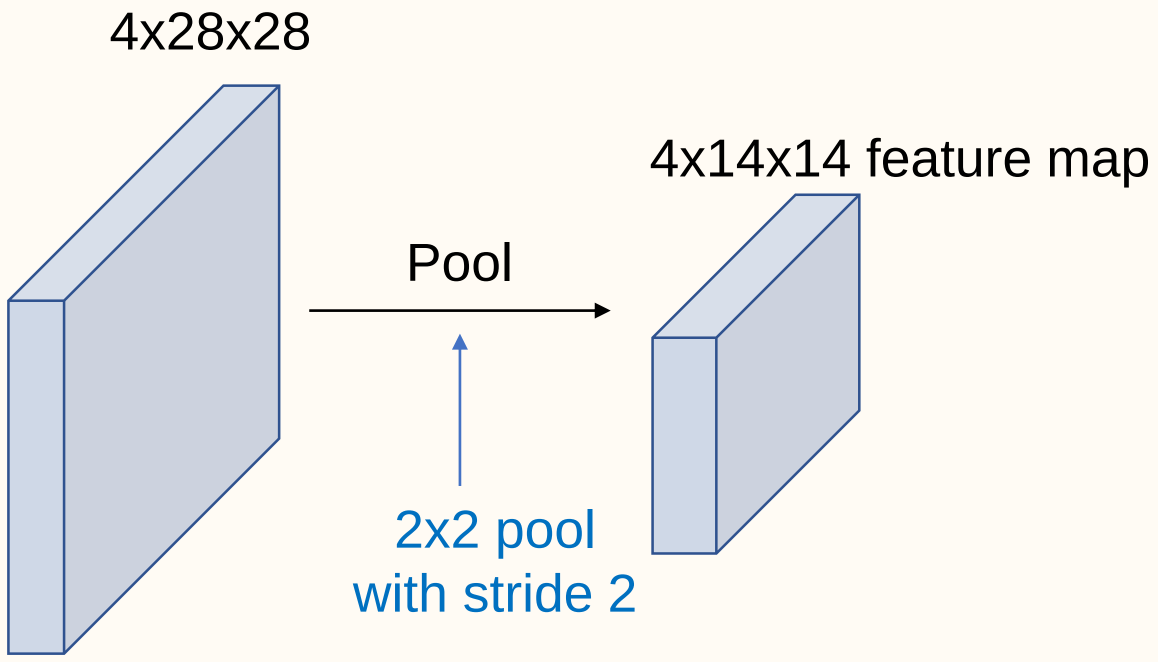
This is the subject of *geometric deep learning*, and its formulation utilizes graph theory and group theory.

Geometric deep learning is particularly useful for non-Euclidean data. Examples include as protein molecule data and social network service connections.

Pooling

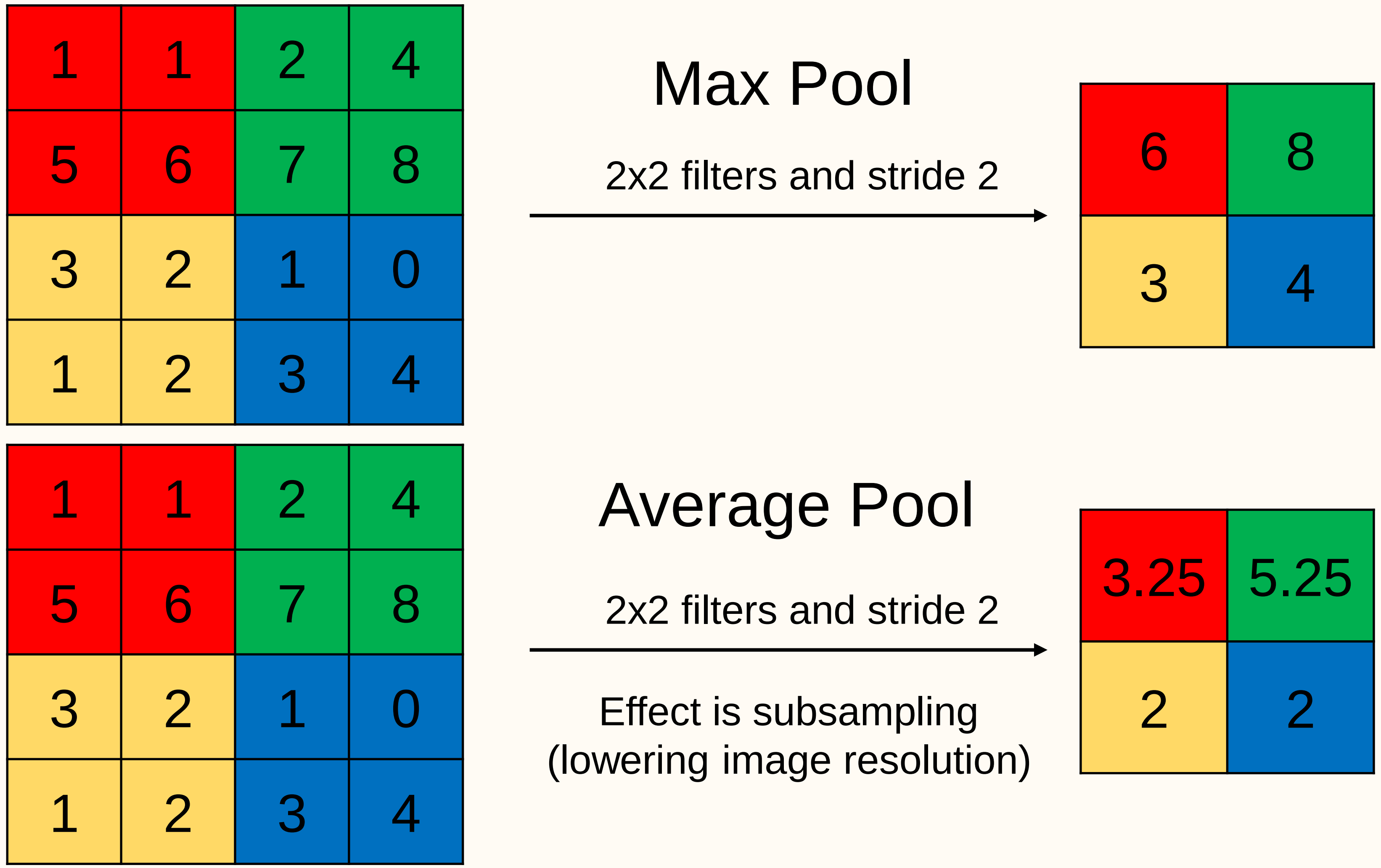
Primarily used to reduce spatial dimension. Similar to conv.

Operates over each channel independently.



Pooling

For each channel



Effect is subsampling (lowering image resolution)

Not an instance of conv.

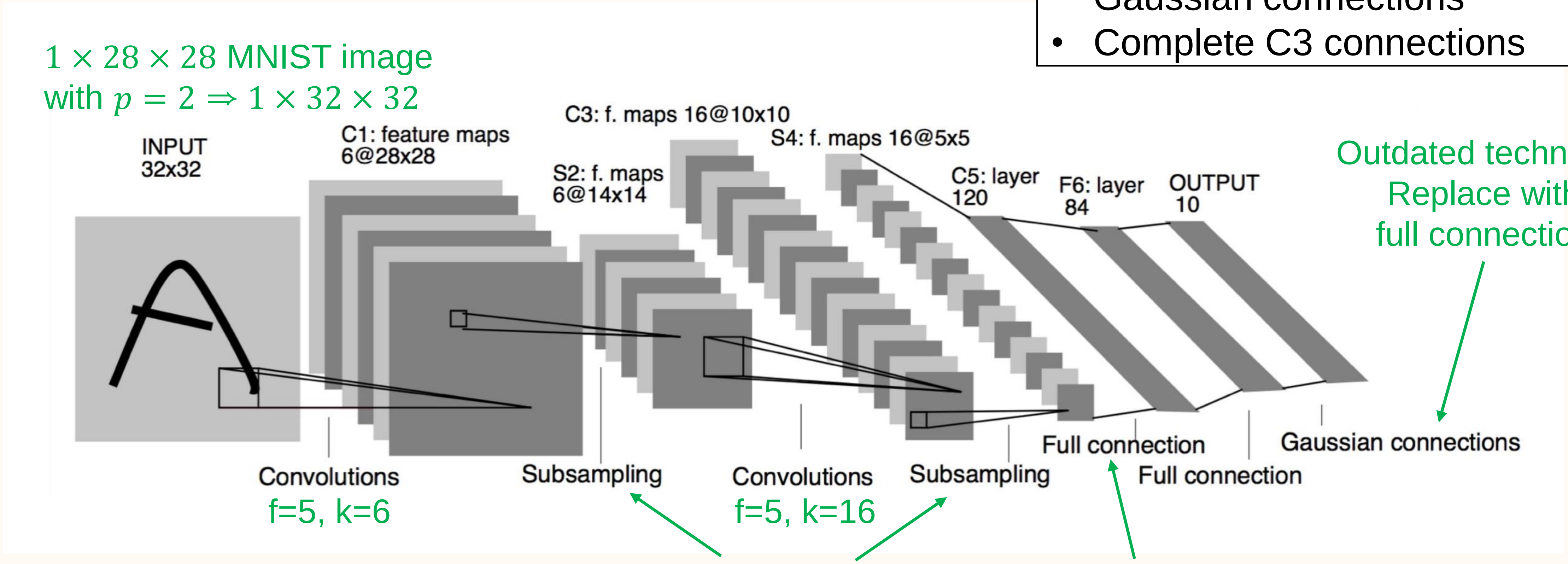
`torch.nn.MaxPool2D`
`torch.nn.AvgPool2D`

Instance of conv. with fixed (untrainable) weights.

LeNet5

Modern instances of LeNet5 use

- σ = ReLu
- MaxPool instead of AvgPool
- No σ after S2, S4 (Why?)
- Full connection instead of Gaussian connections
- Complete C3 connections



$1 \times 28 \times 28$ MNIST image with $p = 2 \Rightarrow 1 \times 32 \times 32$

f=5, k=6

f=5, k=16

Outdated technique Replace with full connection

σ = modification of tanh
Applied after C1, S2, C3, S4, C5, F6

Something like average pool f=2, s=2

full connection from $16 \times 5 \times 5$ to 120 ⇔conv. with f=5, k=120

LeNet5

PyTorch demo

Architectural contribution: LeNet

One of the earliest demonstration of using a deep CNN to learn a nontrivial task.

Laid the foundation of the modern CNN architecture.

Weight sharing

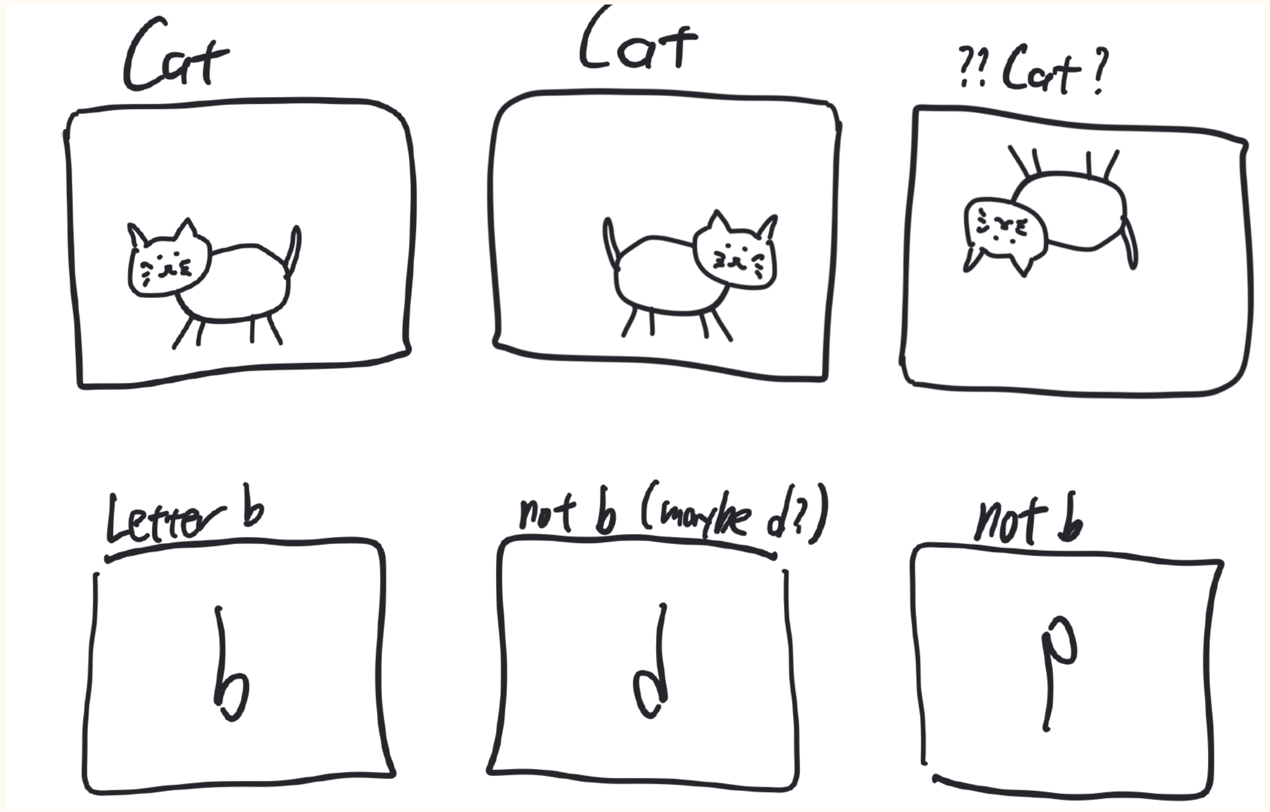
In neural networks, *weight sharing* is a way to reduce the number of parameters by reusing the same parameter in multiple operations. Convolutional layers are the primary example.

$$A_w = \begin{bmatrix} w_1 & \cdots & w_r & 0 & \cdots & & & 0 \\ 0 & w_1 & \cdots & w_r & 0 & \cdots & & 0 \\ 0 & 0 & w_1 & \cdots & w_r & 0 & \cdots & 0 \\ \vdots & & & \ddots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & w_1 & \cdots & w_r & 0 \\ 0 & & \cdots & 0 & 0 & w_1 & \cdots & w_r \end{bmatrix}$$

If we consider convolution with filter w as a linear operator, the components of w appear may times in the matrix representation. This is because the same w is reused for every patch in the convolution. Weight sharing in convolution may now seem obvious, but it was a key contribution back when the LeNet architecture was presented.

Some models (not studied in this course) use weight sharing more explicitly in other ways.

Data augmentation



Invariances
- Translation
- Horizontal flip
- ~~Vertical flip~~
- Color change (?)

Invariances
- Translation
- ~~Horizontal flip~~
- ~~Vertical flip~~
- Color change

Translation invariance encoded in convolution, but other invariances are harder to encode (unless one uses geometric deep learning). Therefore encode invariances in data and have neural networks learn the invariance.

Data augmentation

Data augmentation (DA) applies transforms to the data while preserving meaning and label.

Option 1: Enlarge dataset itself.

- Usually cumbersome and unnecessary.

Option 2: Use randomly transformed data in training loop.

- In PyTorch, we use Torchvision.transforms.

PyTorch demo

Spurious correlation

Hypothetical: A photographer prefers to take pictures with cats looking to the left and dogs looking to the right. Neural network learns to distinguish cats from dogs by which direction it is facing. This learned correlation will not be useful for pictures taken by another photographer.

This is a *spurious correlation,* a correlation between the data and labels that does not capture the “true” meaning. Spurious correlations are not robust in the sense that the spurious correlation will not be a useful predictor when the data changes slightly.

Removing spurious correlations is another purpose of DA.

Data augmentation

We use DA to:

- Inject our prior knowledge of the structure of the data and force the neural network to learn it.
- Remove spurious correlations.
- Increase the effective data size. In particular, we ensure neural network never encounters the exact same data again and thereby prevent the neural network from performing exact memorization. (Neural network can memorize quite well.)

Effects of DA:

- DA usually worsens the training error (but we don’t care about training error).
- DA often, but not always, improves the test error.
    - If DA removes a spurious correlation, then the test error can be worsened.
- DA usually improves robustness.

Data augmentation on test data?

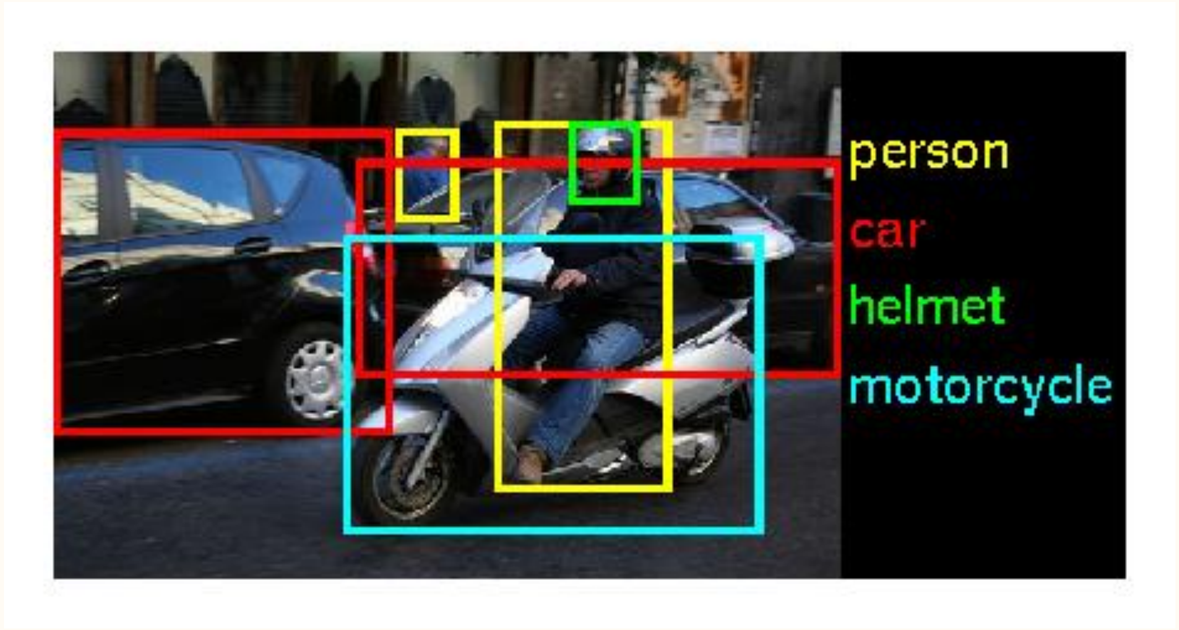
DA is usually applied only on training data.

DA is usually not applied on test data, because we want to ensure test scores are comparable. (There are many different DAs, and applying different DAs on test data will make the metric different.)

However, one can perform “test-time data augmentation” to improve predictions without changing the test. More on this later.

ImageNet dataset

ImageNet contains more 14 million hand-annotated images in more than 20,000 categories.



Many classes, higher resolution, non-uniform image size, multiple objects per image.

History

- Fei-Fei Li started the ImageNet project in 2006 with the goal of expanding and improving the data available for training AI algorithms.
- Images were annotated with Amazon Mechanical Turk.
- The ImageNet team first presented their dataset in the 2009 Conference on Computer Vision and Pattern Recognition (CVPR).
- From 2010 to 2017, the ImageNet project ran the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
- In the 2012 ILSVRC challenge, 150,000 images of 1000 classes were used.
- In 2017, 29 teams achieved above 95% accuracy. The organizers deemed task complete and ended the ILSVRC competition.

ImageNet-1k

Commonly referred to as “the ImageNet dataset”. Also called ImageNet2012

However, ImageNet-1k is really a subset of full ImageNet dataset.

ImageNet-1k has 150,000 images of 1000 roughly balanced classes.

List of categories:

https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a

ImageNet-1k

Data has been removed from the ImageNet website. Downloading peer-to-peer via torrent is now the most convenient way to access the data.

Privacy concerns: Although dataset is about recognizing objects, rather than humans, many human faces are in the images. Troublingly, identifying personal information is possible.

NSFW concerns: Sexual and non-consensual content.

Creating datasets while protecting privacy and other social values is an important challenge going forward.

V. U. Prabhu and A. Birhane, Large image datasets: A pyrrhic win for computer vision?, *WACV* , 2020.
K. Yang, J. Yau, L. Fei-Fei, J. Deng, and O. Russakovsky, A Study of Face Obfuscation in ImageNet, *arXiv*, 2021.

Top-1 vs. top-5 accuracy

Classifiers on ImageNet-1k are often assessed by their *top-5 accuracy*, which requires the 5 categories with the highest confidence to contain the label.

In contrast, the *top-1 accuracy* simply measures whether the network's single prediction is the label.

For example, AlexNet had a top-5 accuracy of 84.6% and a top-1 accuracy of 63.3%.

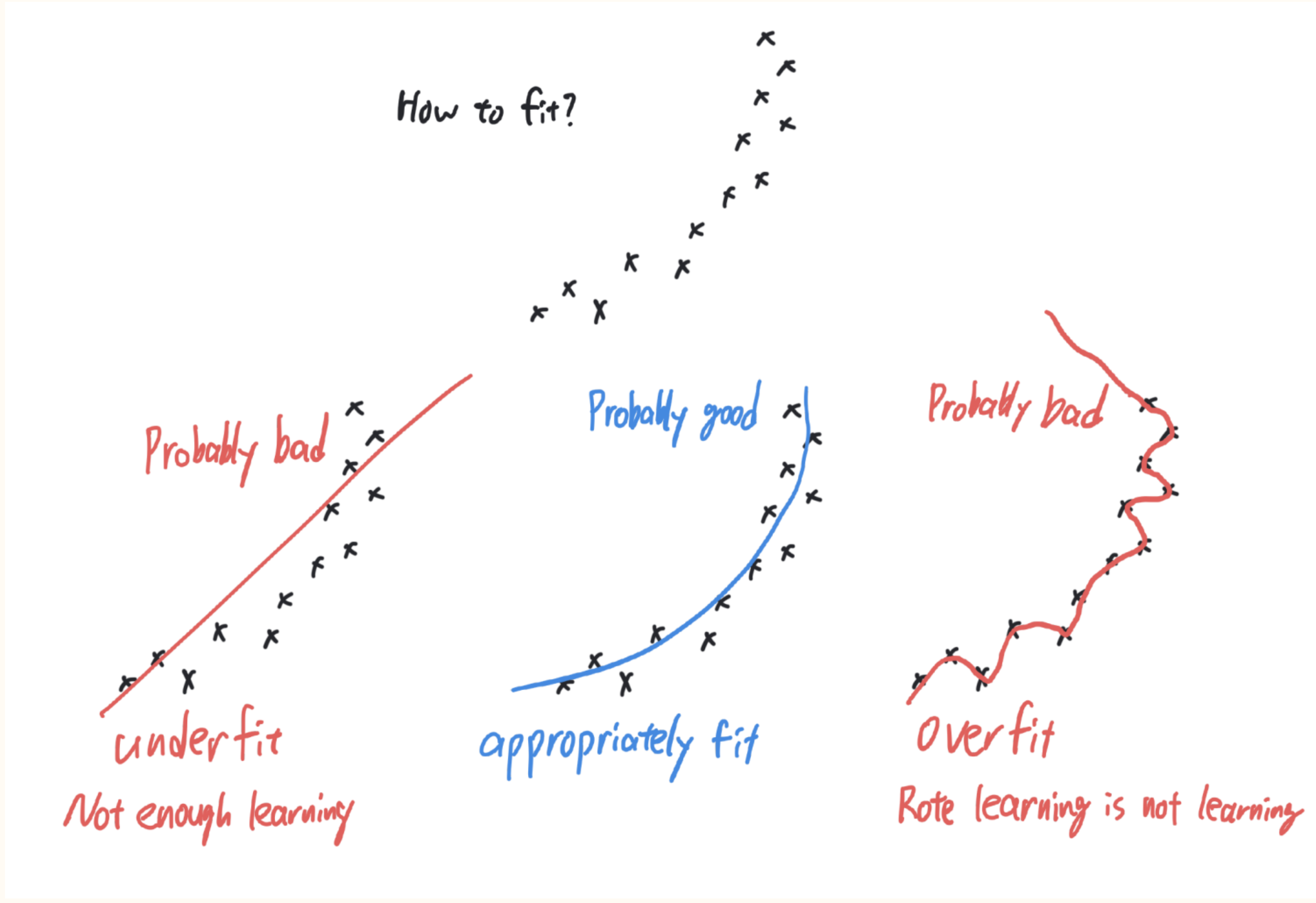
Nowadays, accuracies of classifiers has improved, so the top 1 accuracy is becoming the more common metric.

You can find the state-of-the-art results on ImageNet-1k classification at: https://paperswithcode.com/sota/image-classification-on-imagenet

Classical statistics: Over vs. underfitting

Given separate train and test data

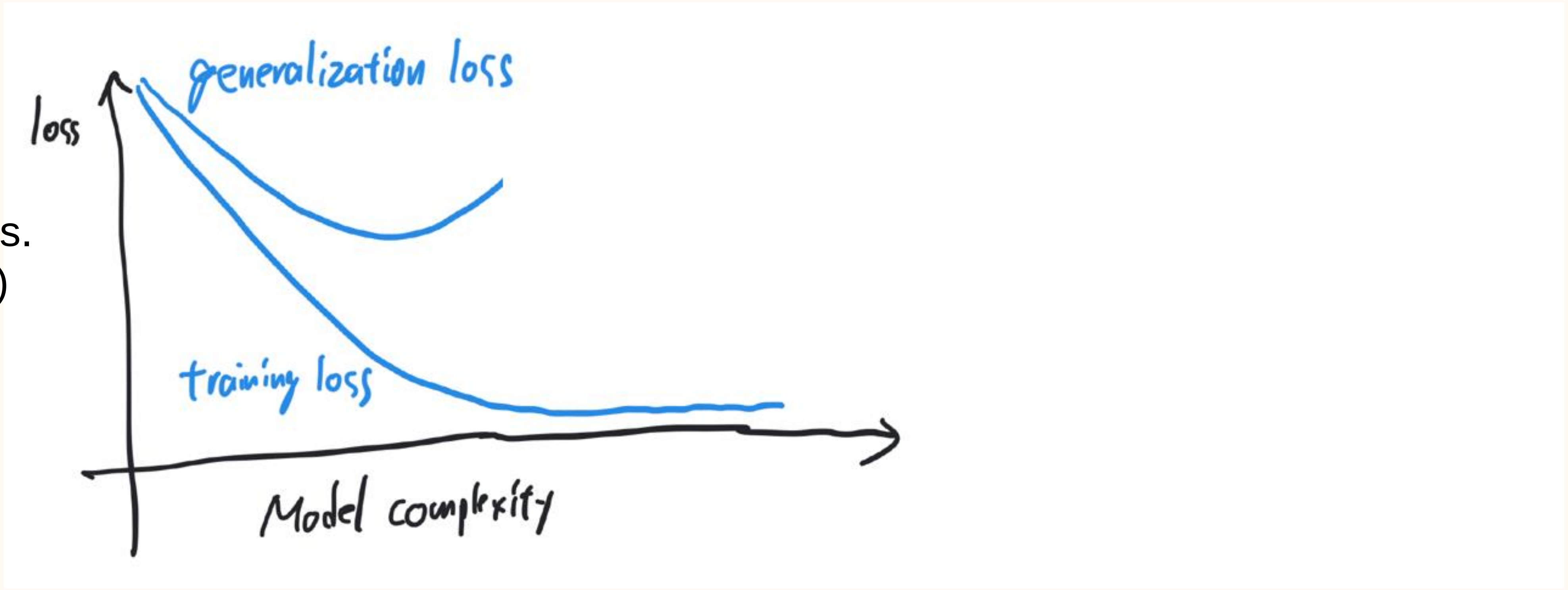
- When (training loss) ≪ (testing loss) you are overfitting. What you have learned from the training data does not carry over to test data.
- When (training loss) ≈ (testing loss) you are underfitting. You have the potential to learn more from the training data.



Classical statistics: Over vs. underfitting

The goal of ML is to learn patterns that generalize to data you have not seen. From each datapoint, you want to learn enough (don't underfit) but if you learn too much you overcompensate for an observation specific to the single experience.

In classical statistics, underfitting vs. overfitting (bias vs. variance tradeoff) is characterized rigorously.

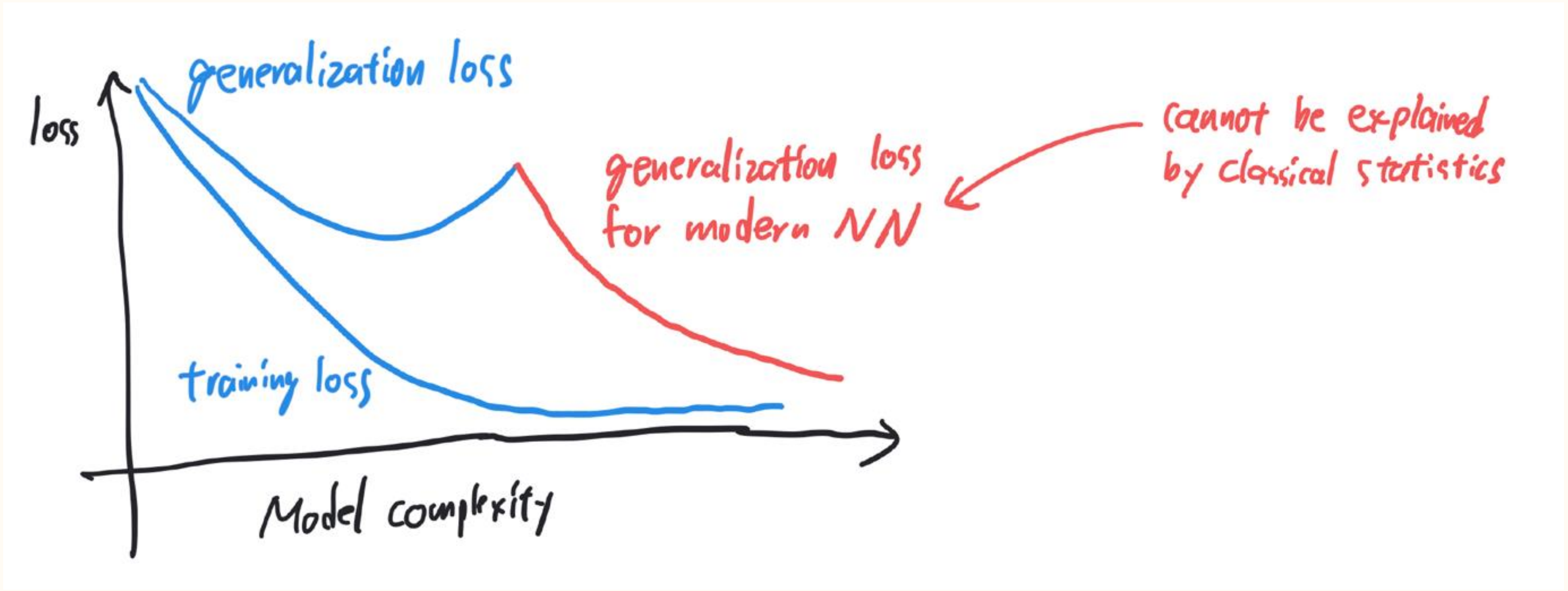


Modern deep learning: Double descent

In modern deep learning, you can overfit, but the state-of-the art neural networks do not overfit (or "benignly overfit") despite having more model parameters than training data.

We do not yet have clarity with this new phenomenon.

When overfitting happens and when it does not is unclear.


generalization loss
loss
cannot be explained by classical statistics
generalization loss for modern NN
training loss
Model complexity

Double descent on 2-layer neural network on MNIST

Belkin et al. experimentally demonstrates the double descent phenomenon with an MLP trained on the MNIST dataset.

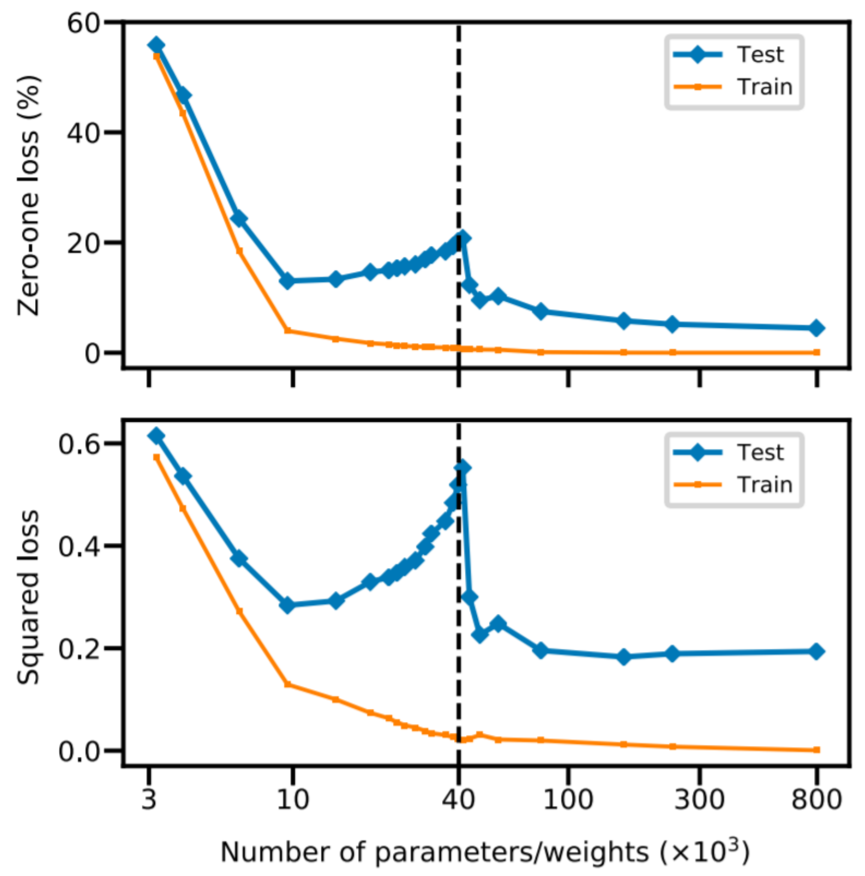
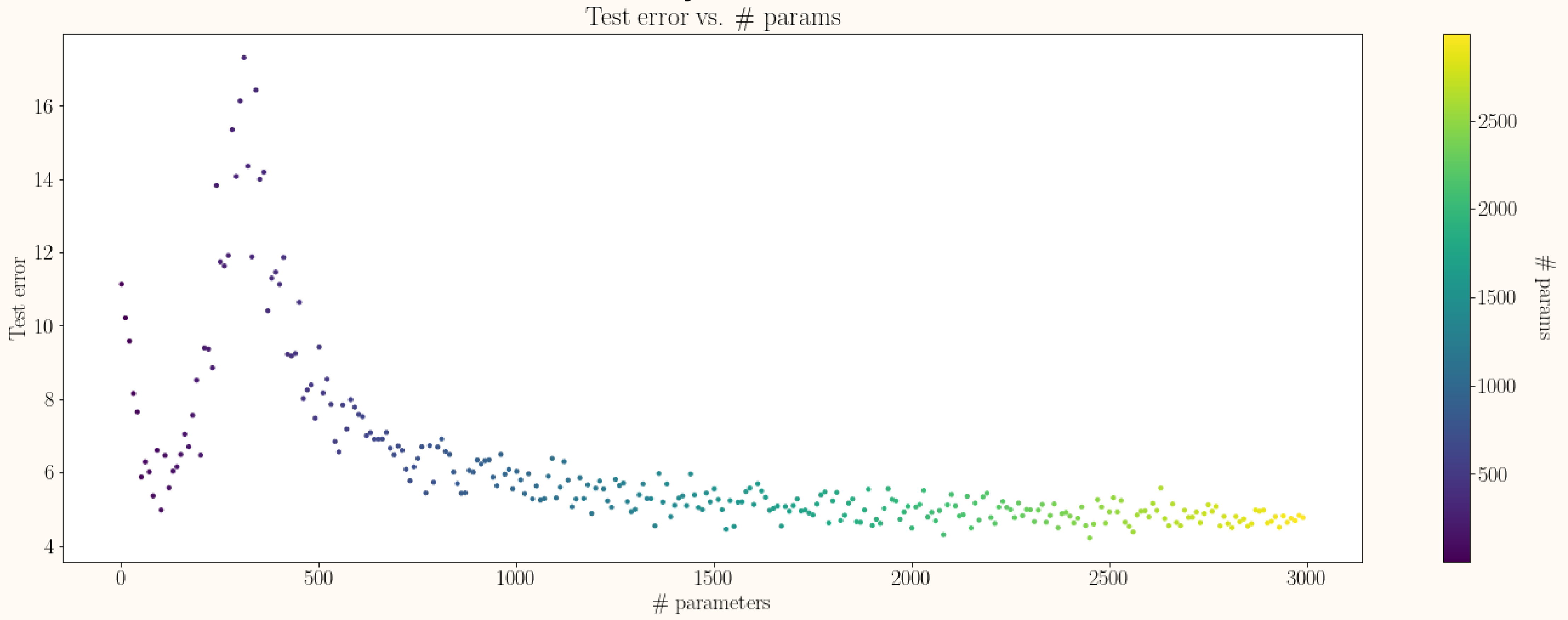


Fig. 3. Double-descent risk curve for a fully connected neural network on MNIST. Shown are training and test risks of a network with a single layer of H hidden units, learned on a subset of MNIST ($n = 4 \cdot 10^3$, $d = 784$, $K = 10$ classes). The number of parameters is $(d+1) \cdot H + (H+1) \cdot K$. The interpolation threshold (black dashed line) is observed at $n \cdot K$.

Double descent example: 2-layer ReLU NN with fixed hidden layer weights

PyTorch Demo



P. Nakkiran, P. Venkat, S. Kakade, and T. Ma, Optimal regularization can mitigate double descent, *ICLR*, 2021.

How to avoid overfitting

Regularization is loosely defined as mechanisms to prevent overfitting.

When you are overfitting, regularize with:

- Smaller NN (fewer parameters) or larger NN (more parameters).
- Improve data by:
    - using data augmentation
    - acquiring better, more diverse, data
    - acquiring more of the same data
- Weight decay
- Dropout
- Early stopping on SGD or late stopping on SGD

How to avoid underfitting

When you are underfitting, use:

- Larger NN (if computationally feasible)
- Less weight decay
- Less dropout
- Run SGD longer (if computationally feasible)

Weight decay $\cong \ell^2$-regularization

ℓ^2*-regularization* augments the loss function with

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad \frac{1}{N}\sum_{i=1}^{N}\ell(f_\theta(x_i),y_i)+\frac{\lambda}{2}\|\theta\|^2$$

SGD on the augmented loss is usually implemented by changing SGD update rather than explicitly changing the loss since

$$\begin{aligned}\theta^{k+1} &= \theta^k - \alpha\left(g^k + \lambda\theta^k\right)\\ &= (1-\alpha\lambda)\theta^k - \alpha g^k\end{aligned}$$

Where g^k is stochastic gradient of original (unaugmented) loss.

In classical statistics, this is called *ridge regression* or *maximum a posteriori (MAP) estimation with Gaussian prior*.

Weight decay $\cong \ell^2$-regularization

In Pytorch, you can use SGD + weight decay by:

augmenting the loss function

```
for param in model.parameters():
  loss += (lamda/2)*param.pow(2.0).sum()
torch.optim.SGD(model.parameters(), lr=... , weight_decay=0)
```

or by using `weight_decay` in the optimizer

```
torch.optim.SGD(model.parameters(), lr=... , weight_decay=lamda)
```

For plain SGD, weight decay and ℓ^2-regularization are equivalent. For other optimizers, the two are similar but not the same. More on this later.

Dropout

Dropout is a regularization technique that randomly disables neurons.

Standard layer,

$$h_2 = \sigma(W_1 h_1 + b_1).$$

Dropout with drop probability p defines

$$h_2 = \sigma(W_1 h_1' + b_1)$$

with

$$(h_1')_j = \begin{cases} 0 & \text{with probability } p \\ \dfrac{(h_1)_j}{1-p} & \text{otherwise.} \end{cases}$$

Note, h_1' is defined so that $\mathbb{E}[h_1'] = h_1$.

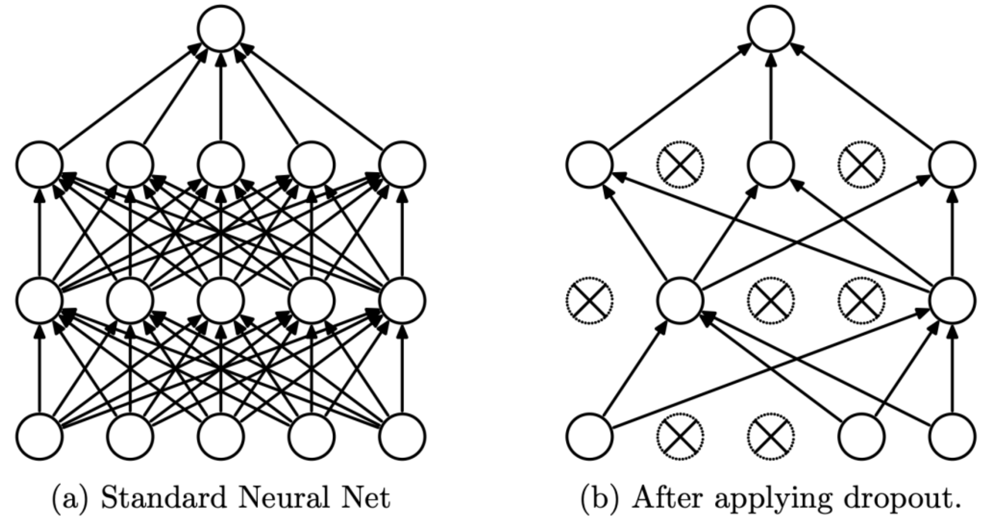


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Why is dropout helpful?

“A motivation for dropout comes from a theory of the role of sex in evolution (Livnat et al., 2010).”

Sexual reproduction, compared to asexual reproduction, creates the criterion for natural selection mix-ability of genes rather than individual fitness, since genes are mixed in a more haphazard manner.

“Since a gene cannot rely on a large set of partners to be present at all times, it must learn to do something useful on its own or in collaboration with a small number of other genes. … Similarly, each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes.

Why is dropout helpful?

The analogy to evolution is very interesting, but it is ultimately a heuristic argument. It also shifts the burden to the question: “why is sexual evolution more powerful than asexual evolution?”

However, dropout can be shown to be loosely equivalent to ℓ^2-regularization. However, we do not yet have a complete understanding of the mathematical reason behind dropout’s performance.

S. Wang and C. Manning, Fast dropout training, *ICML*, 2013.
S. Wager, S. Wang, and P. Liang, Dropout training as adaptive regularization, *NeurIPS*, 2013.

Dropout in PyTorch

Dropout simply multiplies the neurons with a random $0 - \frac{1}{1-p_{\text{drop}}}$ mask.

A direct implementation in PyTorch:

```
def dropout_layer(X, p_drop):
  mask = (torch.rand(X.shape) > p_drop).float()
  return mask * X / (1.0 - p_drop)
```

PyTorch provides an implementation of dropout through `torch.nn.Dropout`.

Dropout in training vs. test

Typically, dropout is used during training and turned off during prediction/testing.

(Dropout should be viewed as an additional onus imposed during training to make training more difficult and thereby effective, but it is something that should be turned off later.)

In PyTorch, activate the training mode with

```
model.train()
```

and activate evaluation mode with

```
model.eval()
```

dropout (and batchnorm) will behave differently in these two modes.

When to use dropout

Dropout is usually used on linear layers but not on convolutional layers.

- Linear layers have many weights and each weight is used only once per forward pass. (If $y = \mathrm{Linear}_{A,b}(x)$, then A_{ij} only affect y_i.) So regularization seems more necessary.
- A convolutional filter has fewer weights and each weight is used multiple times in each forward pass. (If $y = \mathrm{Conv2D}_{w,b}(x)$, then $w_{ijk\ell}$ affects $y_{i,:,:}$.) So regularization seems less necessary.

Dropout seems to be going out of fashion:

- Dropout's effect is somehow subsumed by batchnorm. (This is poorly understood.)
- Linear layers are less common due to their large number of trainable parameters.

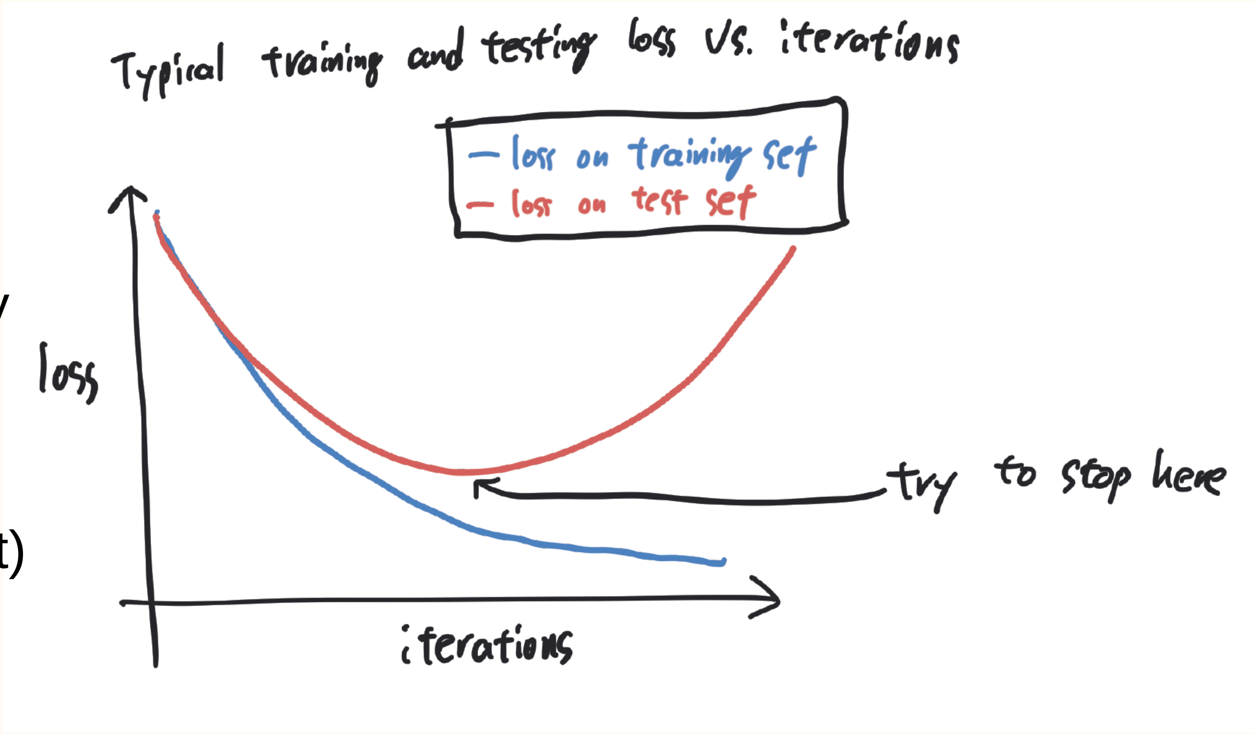
There is no consensus on whether dropout should be applied before or after the activation function. However, Dropout-σ and σ-Dropout are equivalent when σ is ReLU or leaky ReLU, or, more generally, when σ is nonnegative homogeneous.

SGD early stopping

Early stopping of SGD refers to stopping the training early even if you have time for more iterations.

The rationale is that SGD fits data, so too many iterations lead to overfitting.

A similar phenomenon (too many iterations hurt) is observed in classical algorithms for inverse problems.

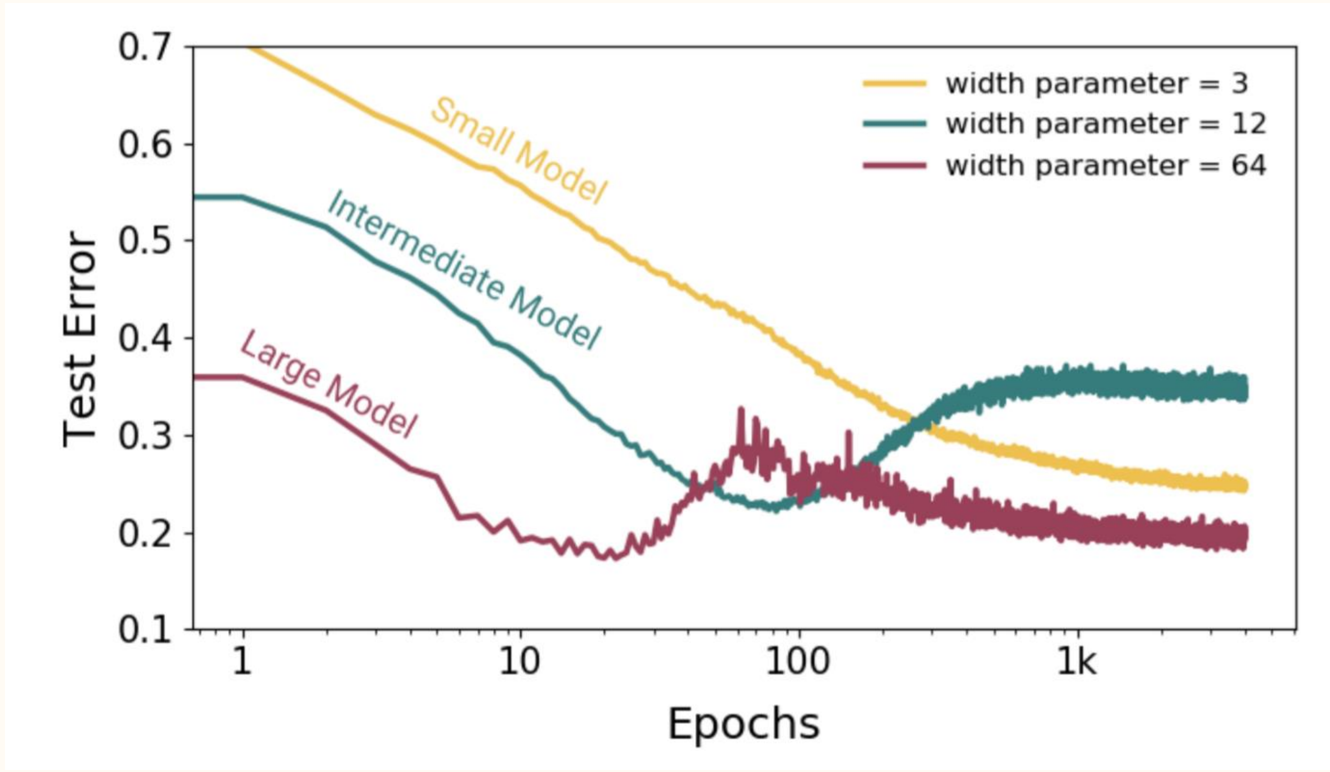


Epochwise double descent

Recently, however, an *epochwise double descent* has been observed.

So perhaps one should stop SGD early or very late.

We do not yet have clarity with this new phenomenon.



More data (by data auagmentation)

With all else fixed, using more data usually* leads to less overfitting.

However, collecting more data is often expansive.

Think of data augmentation (DA) as a mechanism to create more data for free. You can view DA as a form of regularization.

*It seems that more data is not always useful. More on this when we discuss the double descent phenomenon.

Summary of over vs. underfitting

In modern deep learning, the double descent phenomenon has brought a conceptual and theoretical crisis regarding over and underfitting. Much of the machine learning practice is informed by classical statistics and learning theory, which do not take the double descent phenomenon into account.

Double descent will bring fundamental changes to statistics, and researchers need more time to figure things out. Most researchers, practitioners and theoreticians, agree that not all classical wisdom is invalid, but what part do we keep, and what part do we replace?

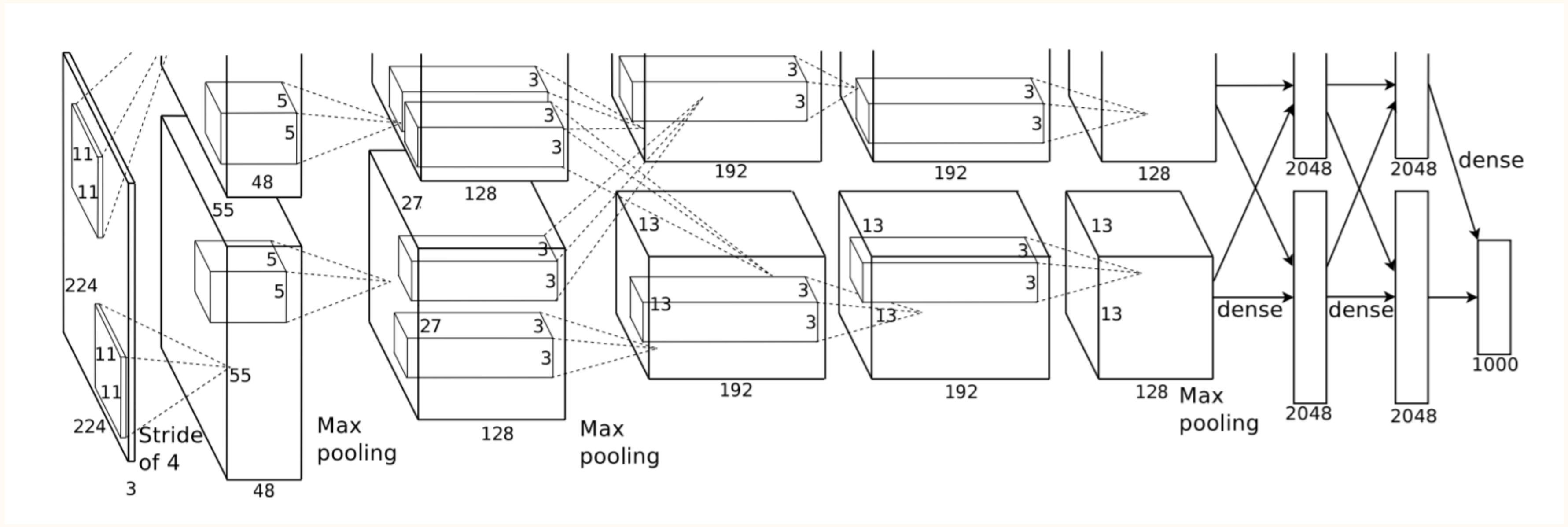
In the meantime, we will have to keep in mind the two contradictory viewpoints and move forward in the absence of clarity.

AlexNet

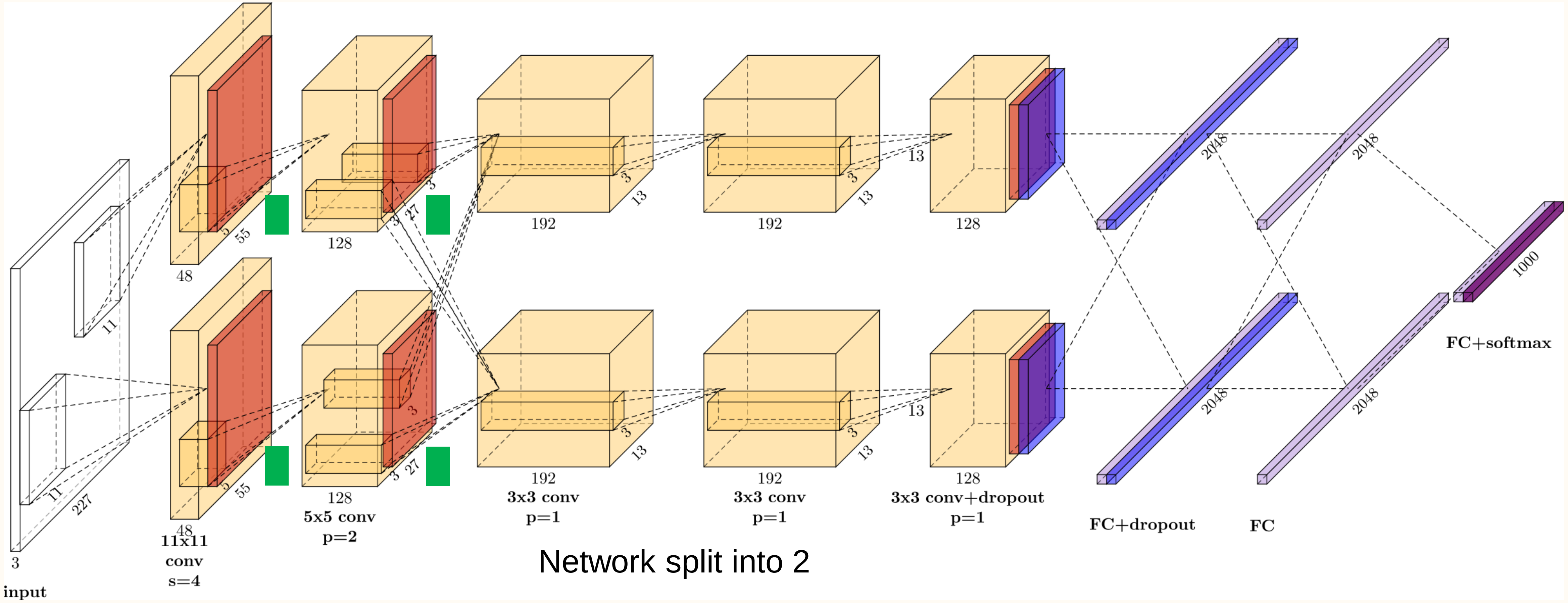
Won the 2012 ImageNet challenge by a large margin: top-5 error rate 15.3% vs. 26.2% second place.

Started the era of deep neural networks and their training via GPU computing.

AlexNet was split into 2 as GPU memory was limited. (A single modern GPU can easily hold AlexNet.)



AlexNet for ImageNet



🟨 Convolution+ReLU

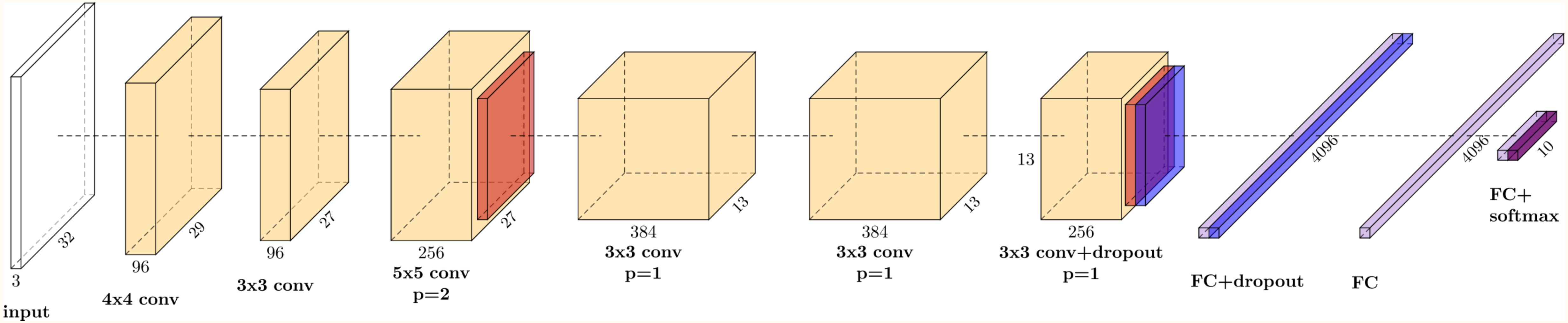
🟦 Dropout (0.5)

🟩 ~~Local response normalization (preserves spatial dimension&channel #s)~~ (outdated technique)

🟥 Max pool $f = 3$, $s = 2$ (overlapping max pool)

🟪 Fully connected layer+ReLU

AlexNet CIFAR10



🟨 Conv.-ReLU

Max pool $f = 3$, $s = 2$ (overlapping max pool)

Network not split into 2

No local response normalization

Architectural contribution: AlexNet

A scaled-up version of LeNet.

Demonstrated that deep CNNs can learn significantly complex tasks. (Some thought CNNs could only learn simple, toy tasks like MNIST.)

Demonstrated GPU computing to be an essential component of deep learning.

Demonstrated effectiveness of ReLU over sigmoid or tanh in deep CNNs for classification.

SGD-type optimizers

In modern NN training, SGD and variants of SGD are usually used. There are many variants of SGD.

The variants are compared mostly on an experimental basis. There is some limited theoretical basis in their comparisons. (Cf. Adam story.)

So far, all efforts to completely replace SGD have failed.

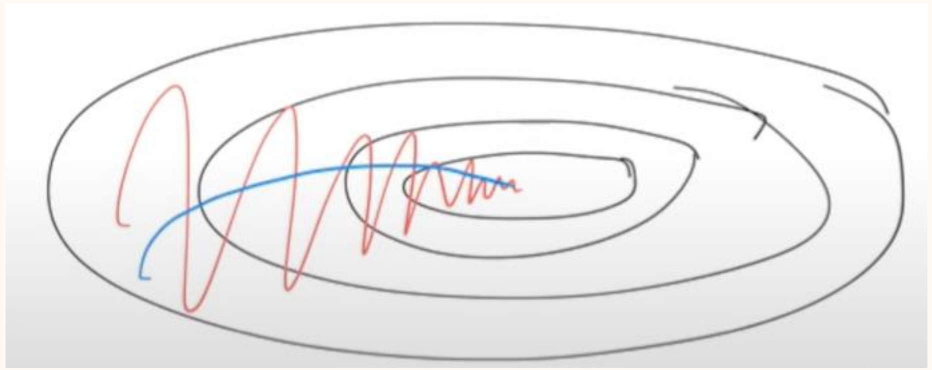
SGD with momentum

SGD:

$$\theta^{k+1} = \theta^k - \alpha g^k$$

SGD with momentum:

$$v^{k+1} = g^k + \beta v^k$$
$$\theta^{k+1} = \theta^k - \alpha v^{k+1}$$



$\beta = 0.9$ is a common choice.

When different coordinates (parameters) have very different scalings (i.e., when the problem is ill-conditioned, momentum can help find a good direction of progress.

RMSProp

RMSProp:

$$m_2^{k+1} = \beta_2 m_2^k + (1-\beta_2)\left(g^k \circledast g^k\right)$$
$$\theta^{k+1} = \theta^k - \alpha g^k \oslash \sqrt{m_2^{k+1} + \epsilon}$$

$\beta_2 = 0.99$ and $\epsilon = 10^{-8}$ are common values. $\circledast$ and $\oslash$ are elementwise mult. and div.

m_2^k is a running estimate of the 2nd moment of the stochastic gradients, i.e., $\left(m_2^k\right)_i \approx \mathbb{E}\left(g^k\right)_i^2$.

$\alpha \oslash \sqrt{m_2^{k+1} + \epsilon}$ is the learning rate scaled elementwise. Progress along steep and noisy directions are dampened while progress along flat and non-noisy directions are accelerated.

Adam (Adaptive moment estimation)

Adam:

$$m_1^{k+1} = \beta_1 m_1^k + (1-\beta_1) g^k, \;\; m_2^{k+1} = \beta_2 m_2^k + (1-\beta_2)\left(g^k \circledast g^k\right)$$

$$\tilde{m}_1^{k+1} = \frac{m_1^{k+1}}{1-\beta_1^{k+1}}, \quad \tilde{m}_2^{k+1} = \frac{m_2^{k+1}}{1-\beta_2^{k+1}}$$

$$\theta^{k+1} = \theta^k - \alpha \tilde{m}_1^{k+1} \oslash \sqrt{\tilde{m}_2^{k+1} + \epsilon}$$

- β_1^{k+1} means β_1 to the $(k+1)$th power.
- $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ are common values. Initialize with $m_1^0 = m_2^0 = 0$.
- m_1^k and m_2^k are running estimates of the 1st and 2nd moments of g^k.
- $\tilde{m}_1^k$ and $\tilde{m}_2^k$ are bias-corrected estimates of m_1^k and m_2^k.
- Using $\tilde{m}_1^k$ instead of g^k adds the effect of momentum.

Bias correction of Adam

To understand the bias correction, consider the hypothetical $g^k = g$ for $k = 0,1, \ldots$. Then

$$m_1^k = \left(1 - \beta_1^k\right) g$$

and

$$m_2^k = \left(1 - \beta_2^k\right)(g \circledast g)$$

while $m_1^k \to g$ and $m_2^k \to (g \circledast g)$ as $k \to \infty$, the estimators are not exact despite there being no variation in g^k.

On the other hand, there is bias-corrected estimators are exact:

$$\tilde{m}_1^k = g$$

and

$$\tilde{m}_2^k = (g \circledast g)$$

The cautionary tale of Adam

Adam's original 2015 paper justified the effectiveness of the algorithm through experiments training deep neural networks with Adam. After all, this non-convex optimization is what Adam was proposed to do.

However, the paper also provided a convergence proof under the assumption of convexity. This was perhaps unnecessary in an applied paper focusing on non-convex optimization.

The proof was later shown* to be incorrect! Adam does not always converge in the convex setup, i.e., the algorithm, rather than the proof, is wrong.

Reddi and Kale presented the AMSGrad optimizer, which does come with a correct convergence proof, but AMSGrad tends to perform worse than Adam, empirically.

*S. J. Reddi, S. Kale, and S. Kumar, On the convergence of Adam and beyond, *ICLR*, 2018.

How to choose the optimizer

Extensive research has gone into finding the "best" optimizer. Schmidt et al.[*] reports that, roughly speaking, that Adam works well most of the time.

So, Adam is a good default choice. Currently, it seems to be the best default choice.

However, Adam does not always work. For example, it seems to be that the widely used EfficientNet model can only be trained[†] with RMSProp.

However, there are some setups where the LR of SGD is harder to tune, but SGD outperforms Adam when properly tuned.[#]

[*] R. M. Schmidt, F. Schneider, and P. Hennig, Descending through a crowded valley — benchmarking deep learning optimizers, *ICML*, 2021.
[†] M. Tan and Q. V. Le, EfficientNet: Rethinking model scaling for convolutional neural networks, *ICML*, 2019.
[#] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, The marginal value of adaptive gradient methods in machine learning, *NeurIPS*, 2017.

How to tune parameters

Everything should be chosen by trial and error. The weight parameters and β, β_1, β_2 and the weight decay parameter λ, and the optimizers should be chosen based on trial and error.

The LR (the stepsize α) of different optimizers are not really comparable between the different optimizers. When you change the optimizer, the LR should be tuned again.

Roughly, large stepsize, large momentum, small weight decay is faster but less stable, while small stepsize, small momentum, and large weight decay is slower but more stable.

Using different optimizers in PyTorch

In PyTorch, the `torch.optim` module implements the commonly used optimizers.

Using SGD:

```
torch.optim.SGD(model.parameters(), lr=X)
```

Using SGD with momentum:

```
torch.optim.SGD(model.parameters(), momentum=0.9, lr=X)
```

Using RMSprop:

```
torch.optim.RMSprop(model.parameters(), lr=X)
```

Using Adam:

```
torch.optim.Adam(model.parameters(), lr=X)
```

Exercise: Try Homework 3 problem 1 with Adam but without the custom weight initialization.

Learning rate scheduler

Sometimes, it is helpful to change (usually reduce) the learning rate as the training progresses. PyTorch provides *learning rate* schedulers to do this.

```
optimizer = SGD(model.parameters(), lr=0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9) # lr = 0.9*lr
for _ in range(...):
  for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
  scheduler.step()  # .step() call updates (changes) the learning rate
```

Diminishing learning rate

One common choice is to specify a diminishing learning rate via a function (a lambda expression). Choices like `C/epoch` or `C/sqrt(iteration)`, where `C` is an appropriately chosen constant, are common.

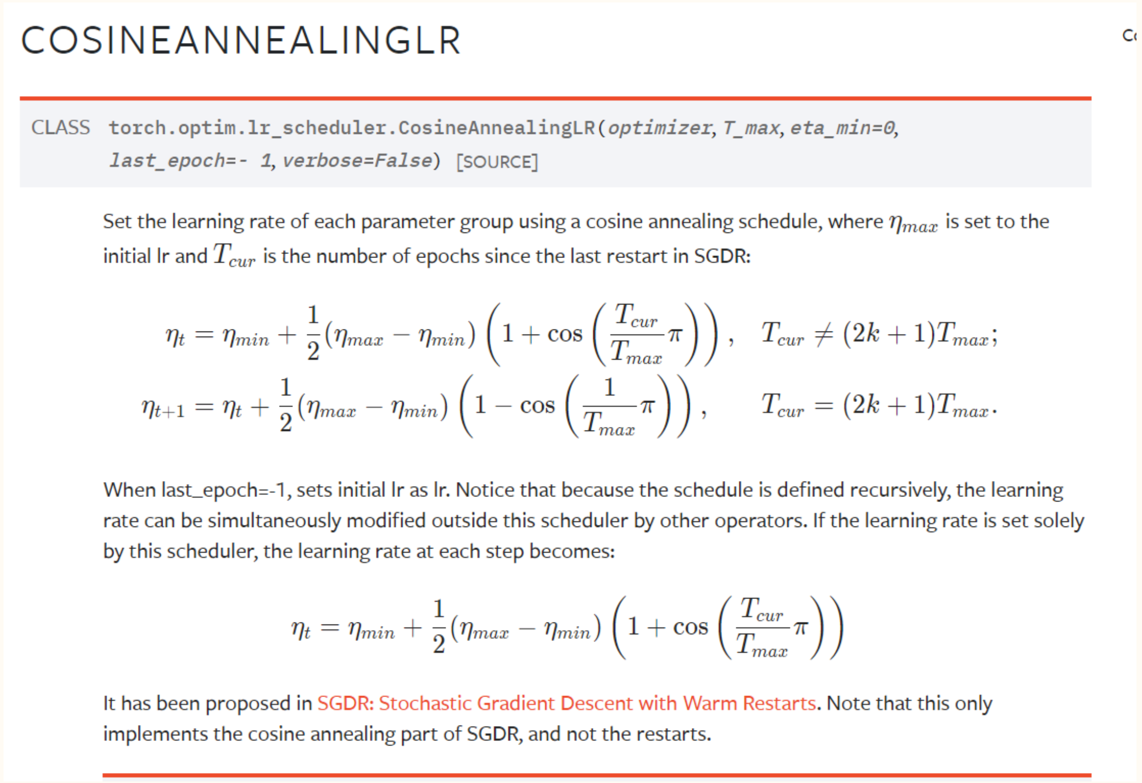
```
# lr_lambda allows us to set lr with a function
scheduler = LambdaLR(optimizer, lr_lambda = lambda ep: 1e-2/ep)
for epoch in range(...):
  for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
  scheduler.step()  # lr=0.01/epoch
```

Cosine learning rate

The cosine learning rate scheduler, which sets the learning rate with the cosine function, is also commonly used.

It is also common to use only a half-period of the cosine rather than having the learning rate oscillate.

The 2nd case in the specification means k and its purpose is to prevent the learning rate from becoming 0.



COSINEANNEALINGLR

CLASS `torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max, eta_min=0, last_epoch=- 1, verbose=False)` [SOURCE]

Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial lr and T_{cur} is the number of epochs since the last restart in SGDR:

$$\begin{aligned} \eta_t &= \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), & T_{cur} \neq (2k+1)T_{max}; \\ \eta_{t+1} &= \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), & T_{cur} = (2k+1)T_{max}. \end{aligned}$$

When last_epoch=-1, sets initial lr as lr. Notice that because the schedule is defined recursively, the learning rate can be simultaneously modified outside this scheduler by other operators. If the learning rate is set solely by this scheduler, the learning rate at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts. Note that this only implements the cosine annealing part of SGDR, and not the restarts.

Wide vs. sharp minima

As alluded to in hw1:

- Large step makes large and rough progress towards regions with small loss.
- Small steps refines the model by finding sharper minima.

Also small steps better suppress the effect of noise. Mathematically, one can show that SGD with small steps becomes very similar to GD with small steps.[#]

However, using small steps to converge to sharp minima may not always be optimal. There is some empirical evidence that wide minima have better test error than sharp minima.[*]

[#]D. Davis, D. Drusvyatskiy, S. Kakade and J. D. Lee, Stochastic subgradient method converges on tame functions, *Found. Comput. Math.*, 2020.
[*]Y. Jiang, B. Neyshabur, H. Mobahi, D. Krishnan, and S. Bengio, Fantastic generalization measures and where to find them, *ICLR*, 2020.

Weight initialization

Remember, SGD is

$$\theta^{k+1} = \theta^k - \alpha g^k$$

where $\theta^0 \in \mathbb{R}^p$ is an initial point. Using a good initial point is important in NN training.

Prescription by LeCun et al.: “Weights should be chosen randomly but in such a way that the [tanh] is primarily activated in its linear region. If weights are all very large then the [tanh] will saturate resulting in small gradients that make learning slow. If weights are very small then gradients will also be very small.” (Cf. Vanishing gradient homework problem.)

“Intermediate weights that range over the [tanh’s] linear region have the advantage that (1) the gradients are large enough that learning can proceed and (2) the network will learn the linear part of the mapping before the more difficult nonlinear part.”

Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp, In: G. Montavon, G. B. Orr, and K.-R. Müller. (eds), *Neural Networks: Tricks of the Trade,* 1998.

Quick math review

Using the 1st order Taylor approximation,

$$\tanh(z) \approx z$$

Write $X \sim \mathcal{N}(\mu, \sigma^2)$ to denote that X is a Gaussian (normal) random variable with mean μ and standard deviation σ.

If X and Y are independent mean-zero random variables, then

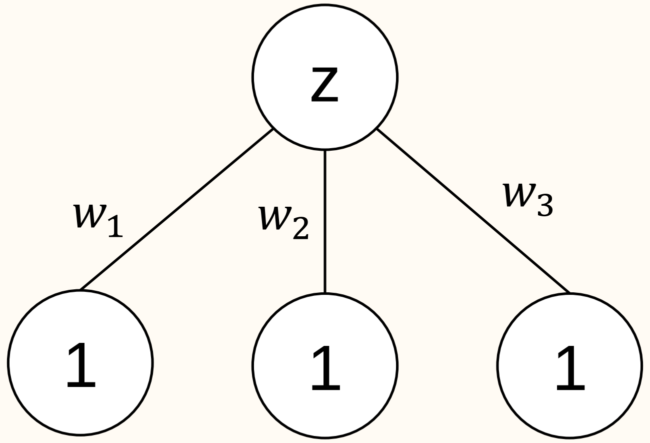
$$\mathbb{E}[XY] = 0$$

$$\mathrm{Var}(XY) = \mathrm{Var}(X)\mathrm{Var}(Y)$$

If X and Y are uncorrelated, i.e., if $\mathbb{E}[(X - \mu_X)(Y - \mu_Y)] = 0$, then $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$. (Uncorrelated R.V. need not be independent.)

Weight initialization

Consider



$$z = w_1 + w_2 + w_3$$

If $w_i \sim \mathcal{N}(0, \sigma^2)$ (zero-mean variance σ^2 Gaussian) then $\mathrm{Var}(\mathrm{z}) = 3\sigma^2$.

If $\sigma = \frac{1}{\sqrt{3}}$, then $\mathrm{Var}(\mathrm{z}) = 1$.

LeCun initialization

Consider the layer

$$y = \tanh(\tilde{y})$$
$$\tilde{y} = Ax + b$$

where $x \in \mathbb{R}^{n_{\text{in}}}$ and $y, \tilde{y} \in \mathbb{R}^{n_{\text{out}}}$. Assume x_j have mean $= 0$ variance $= 1$ and are uncorrelated. If we initialize $A_{ij} \sim \mathcal{N}(0, \sigma_A^2)$ and $b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$, IID, then

$\tilde{y}_i = \sum_{j=1}^{n_{\text{in}}} A_{ij} x_j + b_i$ has mean $= 0$ variance $= n_{\text{in}} \sigma_A^2 + \sigma_b^2$

$y_i = \tanh(\tilde{y}_i) \approx \tilde{y}_i$ has mean ≈ 0 variance $\approx n_{\text{in}} \sigma_A^2 + \sigma_b^2$

If we choose

$$\sigma_A^2 = \frac{1}{n_{\text{in}}}, \quad \sigma_b^2 = 0,$$

(so $b = 0$) then we have y_i mean ≈ 0 variance ≈ 1 and are uncorrelated.

Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp, In: G. Montavon, G. B. Orr, and K.-R. Müller. (eds), *Neural Networks: Tricks of the Trade,* 1998.

LeCun initialization

By induction, with an L-layer MLP,

- if the input to has mean $= 0$ variance $= 1$ and uncorrelated elements,
- the weights and biases are initialized with $A_{ij} \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$ and $b_i = 0$, and
- the linear approximations $\tanh(z) \approx z$ are valid,

then we can expect the output layer to have mean ≈ 0 variance ≈ 1.

Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp, In: G. Montavon, G. B. Orr, and K.-R. Müller. (eds), *Neural Networks: Tricks of the Trade,* 1998.

Xavier initialization

Consider the layer

$$y = \tanh(\tilde{y})$$
$$\tilde{y} = Ax + b$$

where $x \in \mathbb{R}^{n_{\text{in}}}$ and $y, \tilde{y} \in \mathbb{R}^{n_{\text{out}}}$. Consider the gradient with respect to some loss $\ell(y)$. Assume $\left(\frac{\partial \ell}{\partial y}\right)_i$ have mean $= 0$ variance $= 1$ and are uncorrelated. Then

$$\frac{\partial y}{\partial x} = \operatorname{diag}\big(\tanh'(Ax+b)\big)\,A \approx A$$

if $\tanh(\tilde{y}) \approx \tilde{y}$ and

$$\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial y} A$$

If we initialize $A_{ij} \sim \mathcal{N}(0, \sigma_A^2)$ and $b_i \sim \mathcal{N}\big(0, \sigma_b^2\big)$, IID, and assume that $\frac{\partial \ell}{\partial y}$ and A are independent*, then

$$\left(\frac{\partial \ell}{\partial x}\right)_j = \sum_{i=1}^{n_{\text{out}}} \left(\frac{\partial \ell}{\partial y}\right)_i A_{ij} \text{ has mean} \approx 0 \text{ and variance} \approx n_{\text{out}} \sigma_A^2$$

If we choose

$$\sigma_A^2 = \frac{1}{n_{\text{out}}}$$

then $\left(\frac{\partial \ell}{\partial x}\right)_j$ have mean ≈ 0 variance ≈ 1 and are uncorrelated.

Xavier Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, *AISTATS*, 2010.
*False, but we assume it nevertheless.

Xavier initialization

$\frac{\partial \ell}{\partial y}$ and A are not independent; $\frac{\partial \ell}{\partial y}$ depends on the forward evaluation, which in turn depends on A. Nevertheless, the calculation is an informative exercise and its result seems to be representative of common behavior.

If $y = \tanh(Ax + b)$ is an early layer (close to input) in a deep neural network, then the randomness of A is diluted through the forward and backward propagation and $\frac{\partial \ell}{\partial y}$ and A will be nearly independent.

If $y = \tanh(Ax + b)$ is an later layer (close to output) in a deep neural network, then $\frac{\partial \ell}{\partial y}$ and A will have strong dependency.

Xavier initialization

Consideration of forward and backward passes result in different prescriptions.
The Xavier initialization uses the harmonic mean of the two:

$$\sigma_A^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}, \qquad \sigma_b^2 = 0$$

In the literature, the alternate notation fan_{in} and fan_{out} are often used instead of n_{in} and n_{out}.
The fan-in and fan-out terminology originally refers to the number of electric connections entering and exiting a circuit or an electronic device.

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

(Kaiming) He initialization

Consider the layer

$$y = \mathrm{ReLU}(Ax + b)$$

We cannot use the Taylor expansion with ReLU.

However, a similar line of reasoning with the forward pass gives rise to

$$\sigma_A^2 = \frac{2}{n_{\mathrm{in}}}$$

And a similar consideration with backprop gives rise to

$$\sigma_A^2 = \frac{2}{n_{\mathrm{out}}}$$

In PyTorch, use `mode='fan_in'` and `mode='fan_out'` to toggle between the two modes.

Kaiming He, X. Zhang, S. Ren, and J. Sun, Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification, *ICCV*, 2015.

Discussions on initializations

In the original description of the Xavier and He initializations, the biases are all initialized to 0. However, the default initialization of `Linear`* and `Conv2d`# layers in PyTorch uses initialize the biases randomly. A documented reasoning behind this choice (in the form of papers or GitHub discussions) do not seem to exist.

Initializing weights with the proper scaling is sometimes necessary to get the network to train, as you will see with the VGG network. However, so long as the network gets trained, the choice of initialization does not seem to affect the final performance.

Since initializations rely on the assumption that the input to each layer has roughly unit variance, it is important that the data is scaled properly. This is why PyTorch dataloader scales pixel intensity values to be in $[0,1]$, rather than $[0,255]$.

*https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html
#https://pytorch.org/docs/stable/_modules/torch/nn/modules/conv.html

Initialization for conv

Consider the layer

$$\begin{aligned} y &= \tanh(\tilde{y}) \\ \tilde{y} &= \mathrm{Conv2D}_{w,b}(x) \end{aligned}$$

where $w \in \mathbb{R}^{C_{\mathrm{out}} \times C_{\mathrm{in}} \times f_1 \times f_2}$ and $b \in \mathbb{R}^{C_{\mathrm{out}}}$. Assume x_j have mean $= 0$ variance $= 1$ and are uncorrelated*. If we initialize $w_{ijk\ell} \sim \mathcal{N}(0, \sigma_w^2)$ and $b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$, IID, then

$$\begin{aligned} &\tilde{y}_i \quad\quad \text{has mean} = 0 \text{ variance} = (C_{\mathrm{in}} f_1 f_2)\sigma_w^2 + \sigma_b^2 \\ &y_i \approx \tilde{y}_i \text{ has mean} \approx 0 \text{ variance} \approx (C_{\mathrm{in}} f_1 f_2)\sigma_w^2 + \sigma_b^2 \end{aligned}$$

If we choose

$$\sigma_w^2 = \frac{1}{C_{\mathrm{in}} f_1 f_2}, \quad \sigma_b^2 = 0,$$

(so $b = 0$) then we have y_i mean ≈ 0 variance ≈ 1 and are correlated.

*False, but we assume it nevertheless.

Initialization for conv

Outputs from a convolutional layer are correlated. The uncorrelated assumption is false. Nevertheless, the calculation is an informative exercise and its result seems to be representative of common behavior.

Xavier and He initialization is usually used with

$$n_{\mathrm{in}} = C_{\mathrm{in}} f_1 f_2$$

and

$$n_{\mathrm{out}} = C_{\mathrm{out}} f_1 f_2$$

Justification of $n_{\mathrm{out}} = C_{\mathrm{out}} f_1 f_2$ requires working through the complex indexing or considering the “transpose convolution”. We will return to it later.

ImageNet after AlexNet

AlexNet won the 2012 ImageNet challenge with 8 layers.

ZFNet won the 2013 ImageNet challenge also with 8 layers but with better parameter tuning.

Research since AlexNet indicated that depth is more important than width.

VGGNet ranked 2nd in the 2014 ImagNet challenge with 19 layers.

GoogLeNet ranked 1st in the 2014 ImageNet challenge with 22 layers.

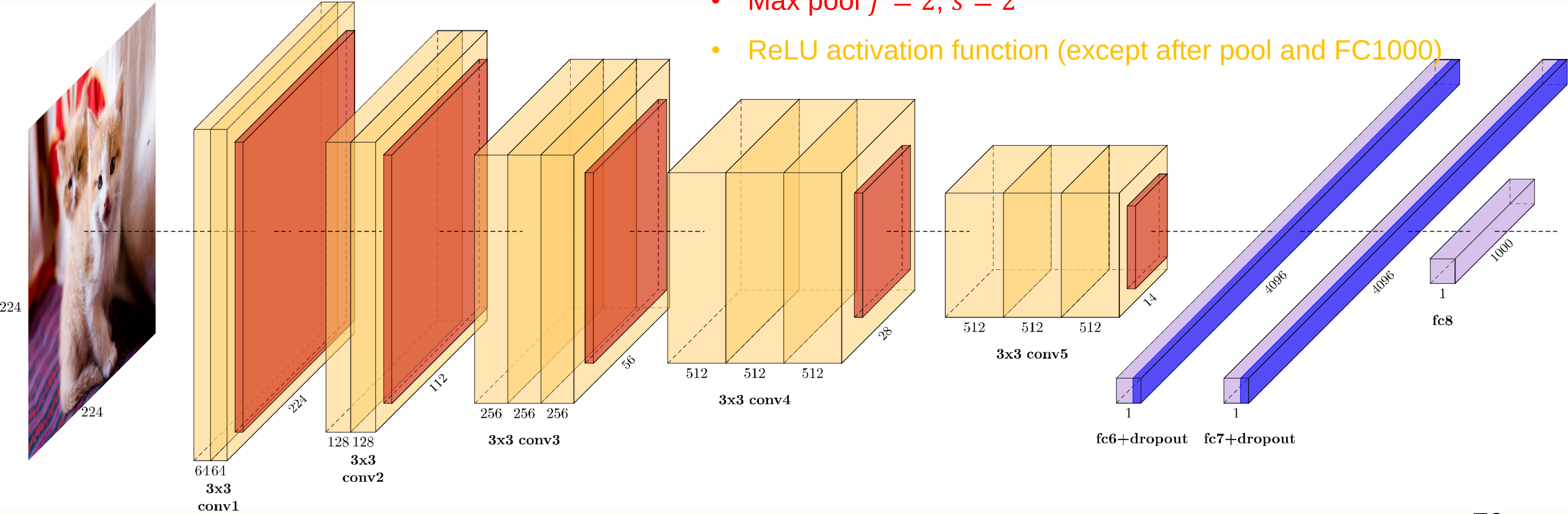
M. D. Zeiler and R. Fergus, Visualizing and understanding convolutional networks, *ECCV*, 2014.

VGGNet

By the Oxford Visual Geometry Group

VGG16

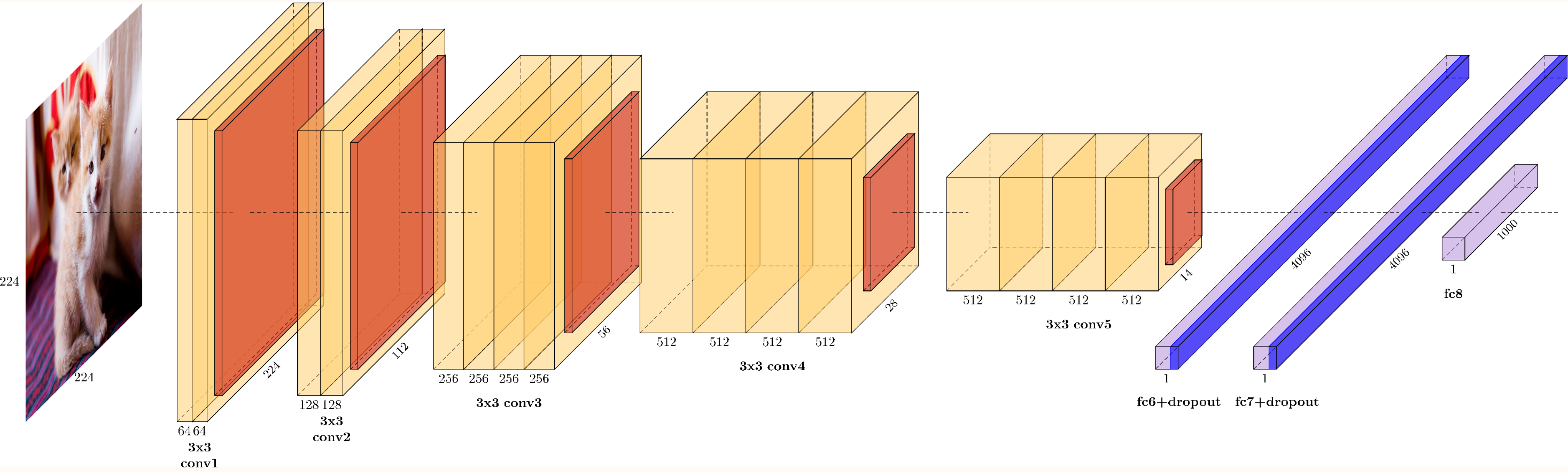
- 16 layers with trainable parameters
- 3x3 conv. $p = 1$ (spatial dimension preserved)
- No local response normalization
- Weight decay 5×10^{-4}
- Dropout(0.5) used
- Max pool $f = 2$, $s = 2$
- ReLU activation function (except after pool and FC1000)



VGGNet

VGG19

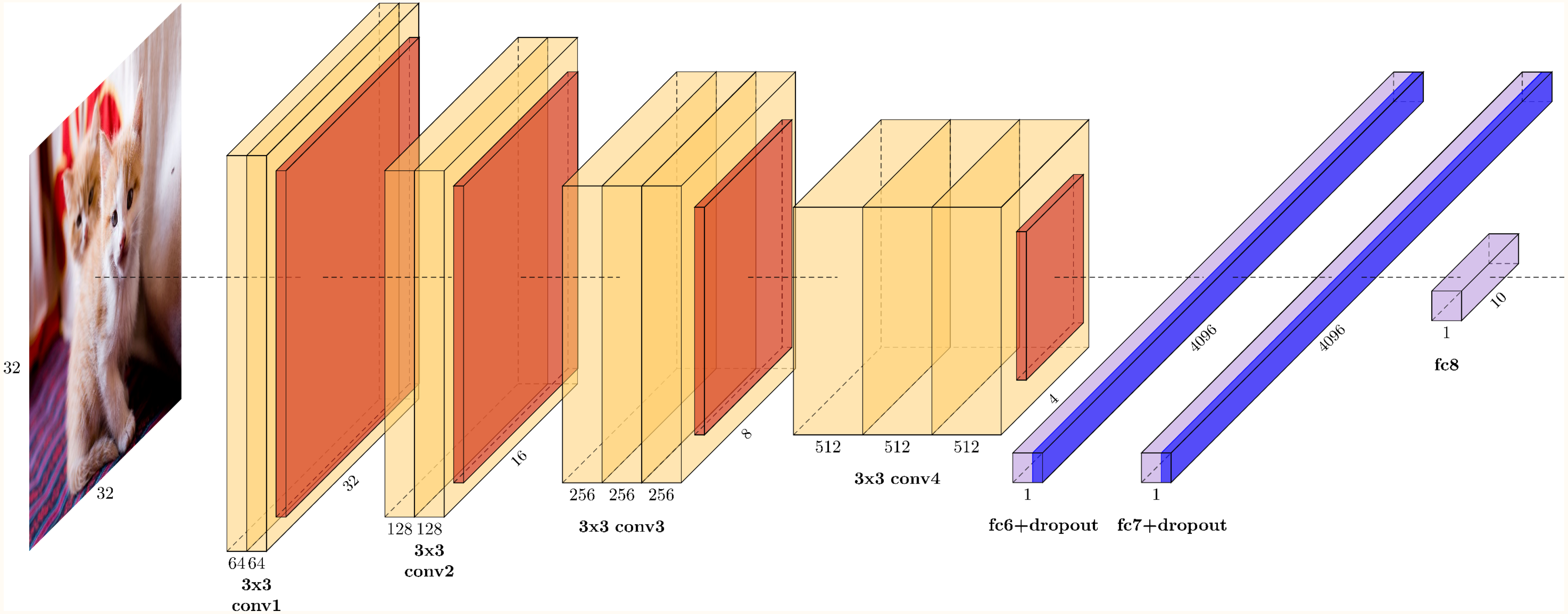
- 19 layers with trainable parameters
- Slightly better than VGG16



VGGNet-CIFAR10

13-layer modification of VGGNet for CIFAR10

- 3x3 conv. $p = 1$
- Max pool $f = 2$, $s = 2$



VGGNet training

Training VGGNet was tricky. A shallower version was first trained and then additional layers were gradually added.

Our VGGNet-CIFAR10 is much easier to train since there are fewer layers and the task is simpler. However, good weight initialization is still necessary

Batchnorm (not available when VGGNet was published) makes training VGGNet much easier. With Batchnorm, the complicated initialization scheme of training a smaller version first becomes unnecessary.

PyTorch demo

Architectural contribution: VGGNet

Demonstrated simple deep CNNs can significantly improve upon AlexNet.

In a sense, VGGNet represents the upper limit of the simple CNN architecture. (It is the best simple model.) Future architectures make gains through more complex constructions.

Demonstrated effectiveness of stacked 3x3 convolutions over larger 5x5 or 11x11 convolutions. Large convolutions (larger than 5x5) are now uncommon.

Due to its simplicity, VGGNet is one of the most common test subjects for testing something on deep CNNs.

Backprop $\subseteq$ autodiff

Autodiff (automatic differentiation) is an algorithm that automates gradient computation. In deep learning libraries, you only need to specify how to evaluate the function.
Backprop (back propagation) is an instance of autodiff.

Gradient computation costs roughly $5\times$ the computation cost* of forward evaluation.

To clarify, backprop and autodiff are not

- finite difference or
- symbolic differentiation.

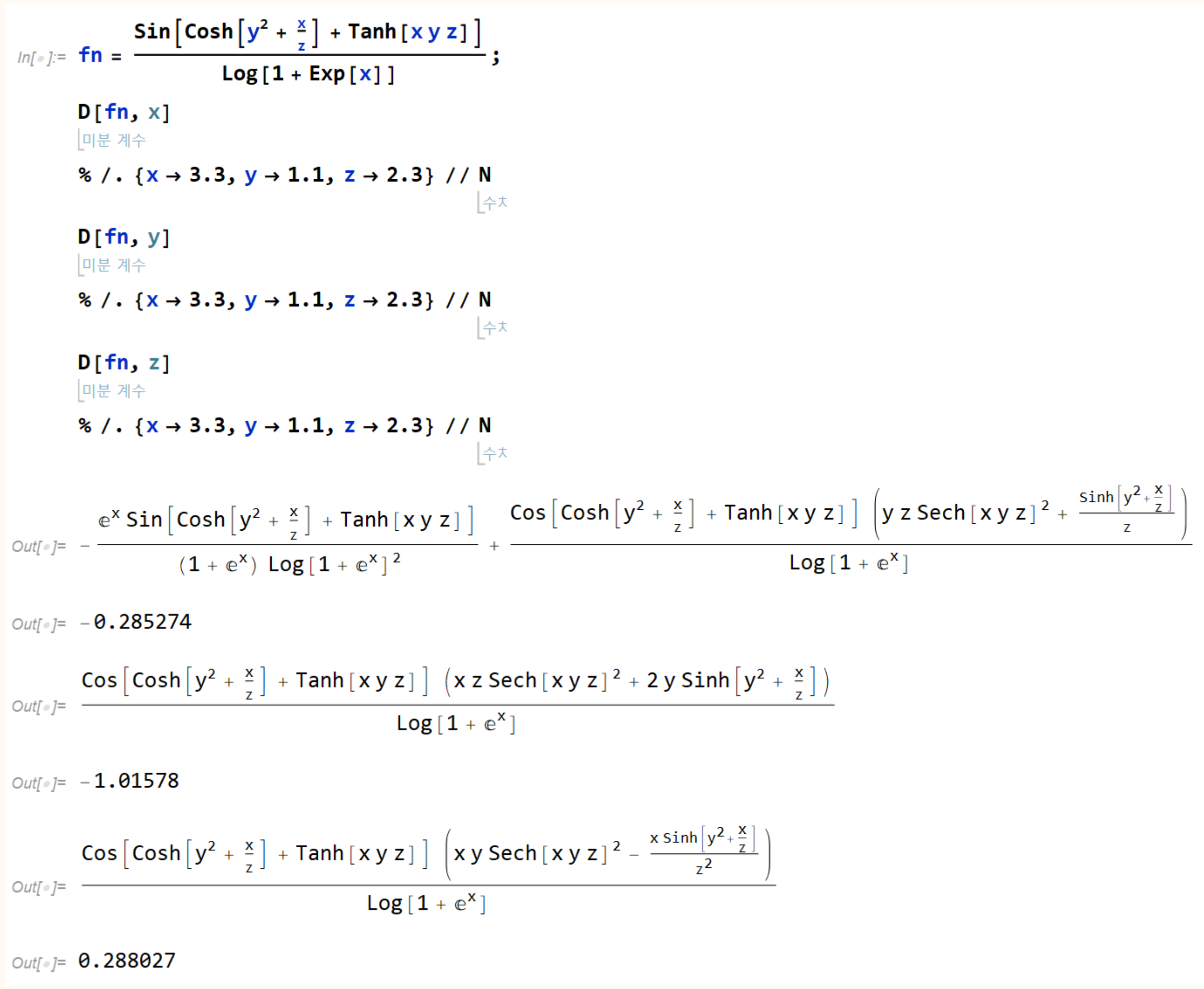
Autodiff $\approx$ chain rule of vector calculus

*Depends on computational structure of function. 5X difference is mostly true for neural networks used in deep learning.

Autodiff example

This complicated gradient computation is simplified by autodiff.

PyTorch demo



In[∘]:=
$$fn = \frac{\mathrm{Sin}\left[\mathrm{Cosh}\left[y^2 + \frac{x}{z}\right] + \mathrm{Tanh}[x\,y\,z]\right]}{\mathrm{Log}[1 + \mathrm{Exp}[x]]};$$

```
D[fn, x]
미분 계수
% /. {x → 3.3, y → 1.1, z → 2.3} // N
수ㅊ
D[fn, y]
미분 계수
% /. {x → 3.3, y → 1.1, z → 2.3} // N
수ㅊ
D[fn, z]
미분 계수
% /. {x → 3.3, y → 1.1, z → 2.3} // N
수ㅊ
```

Out[∘]=
$$-\frac{e^x\,\mathrm{Sin}\left[\mathrm{Cosh}\left[y^2 + \frac{x}{z}\right] + \mathrm{Tanh}[x\,y\,z]\right]}{(1 + e^x)\,\mathrm{Log}[1 + e^x]^2} + \frac{\mathrm{Cos}\left[\mathrm{Cosh}\left[y^2 + \frac{x}{z}\right] + \mathrm{Tanh}[x\,y\,z]\right]\left(y\,z\,\mathrm{Sech}[x\,y\,z]^2 + \frac{\mathrm{Sinh}\left[y^2 + \frac{x}{z}\right]}{z}\right)}{\mathrm{Log}[1 + e^x]}$$

Out[∘]= -0.285274

Out[∘]=
$$\frac{\mathrm{Cos}\left[\mathrm{Cosh}\left[y^2 + \frac{x}{z}\right] + \mathrm{Tanh}[x\,y\,z]\right]\left(x\,z\,\mathrm{Sech}[x\,y\,z]^2 + 2\,y\,\mathrm{Sinh}\left[y^2 + \frac{x}{z}\right]\right)}{\mathrm{Log}[1 + e^x]}$$

Out[∘]= -1.01578

Out[∘]=
$$\frac{\mathrm{Cos}\left[\mathrm{Cosh}\left[y^2 + \frac{x}{z}\right] + \mathrm{Tanh}[x\,y\,z]\right]\left(x\,y\,\mathrm{Sech}[x\,y\,z]^2 - \frac{x\,\mathrm{Sinh}\left[y^2 + \frac{x}{z}\right]}{z^2}\right)}{\mathrm{Log}[1 + e^x]}$$

Out[∘]= 0.288027

The power of autodiff

Autodiff is an essential yet often an underappreciated feature of the deep learning libraries. It allows deep learning researchers to use complicated neural networks, while avoiding the burden of performing derivative calculations by hand.

Most deep learning libraries support 2nd and higher order derivative computation, but we will only use 1st order derivatives (gradients) in this class.

Autodiff includes forward-mode, reverse-mode (backprop), and other orders. In deep learning, reverse-mode is most commonly used.

Autodiff by Jacobian multiplication

Consider $g = f_L \circ f_{L-1} \circ \cdots \circ f_2 \circ f_1$, where $f_\ell : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$ for $\ell = 1, \cdots, L$.

Chain rule: $Dg = \underset{n_L \times n_{L-1}}{Df_L} \; \underset{n_{L-1} \times n_{L-2}}{Df_{L-1}} \; \cdots \; \underset{n_2 \times n_1}{Df_2} \; \underset{n_1 \times n_0}{Df_1}$

Forward-mode: $Df_L(Df_{L-1}(\cdots(Df_2 Df_1)\cdots))$

Reverse-mode: $(((Df_L\, Df_{L-1})\, Df_{L-2})\cdots)\, Df_1$

Reverse mode is optimal* when $n_L \le n_{L-1} \le \cdots \le n_1 \le n_0$. The number of neurons in each layer tends to decrease in deep neural networks for classification. So reverse-mode is often close to the most efficient mode of autodiff in deep learning.

*Can be proved with dynamic programming. Cf. "matrix chain multiplication".

General backprop

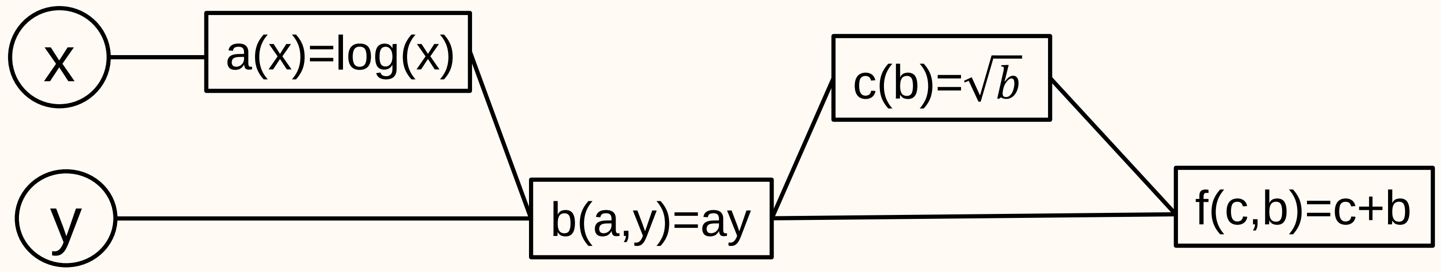
Backprop in PyTorch:

1. When the loss function is evaluated, a computation graph is constructed.
2. The computation graph is a directed acyclic graph (DAG) that encodes dependencies of the individual computational components.
3. A topological sort is performed on the DAG and the backprop is performed on the reversed order of this topological sort. (The topological sort ensures that nodes ahead in the DAG are processed first.)

The general form combines a graph theoretic formulation with the principles of backprop that you have seen in the homework assignments.

Computation graph example

Consider $f(x, y) = y \log x + \sqrt{y \log x}$. Evaluate f with the computation graph:



The chain rule:
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\frac{\partial x}{\partial x} + \frac{\partial b}{\partial y}\cancelto{0}{\frac{\partial y}{\partial x}}\right) + \frac{\partial f}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\frac{\partial x}{\partial x} + \frac{\partial b}{\partial y}\cancelto{0}{\frac{\partial y}{\partial x}}\right)$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\cancelto{0}{\frac{\partial x}{\partial y}} + \frac{\partial b}{\partial y}\frac{\partial y}{\partial y}\right) + \frac{\partial f}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\cancelto{0}{\frac{\partial x}{\partial y}} + \frac{\partial b}{\partial y}\frac{\partial y}{\partial y}\right)$$

But in what order do you evaluate the chain rule expression?

Computation graph

Let $y_1, \dots, y_L$ be the output values (neurons) of the computational nodes. Assume $y_1, \dots, y_L$ follow a linear topological ordering, i.e., the computation of y_ℓ depends on $y_1, \dots, y_{\ell-1}$ and does not depend on $y_{\ell+1}, \dots, y_L$.

Define the graph $G = (V, E)$, where $V = \{1, \dots, L\}$ and $(i, \ell) \in E$, i.e., $i \to \ell$, if the computation of y_ℓ directly depends on y_i. Write the computation of $y_1, \dots, y_L$ as

$$y_\ell = f_\ell([y_i : \text{for } i \to \ell])$$

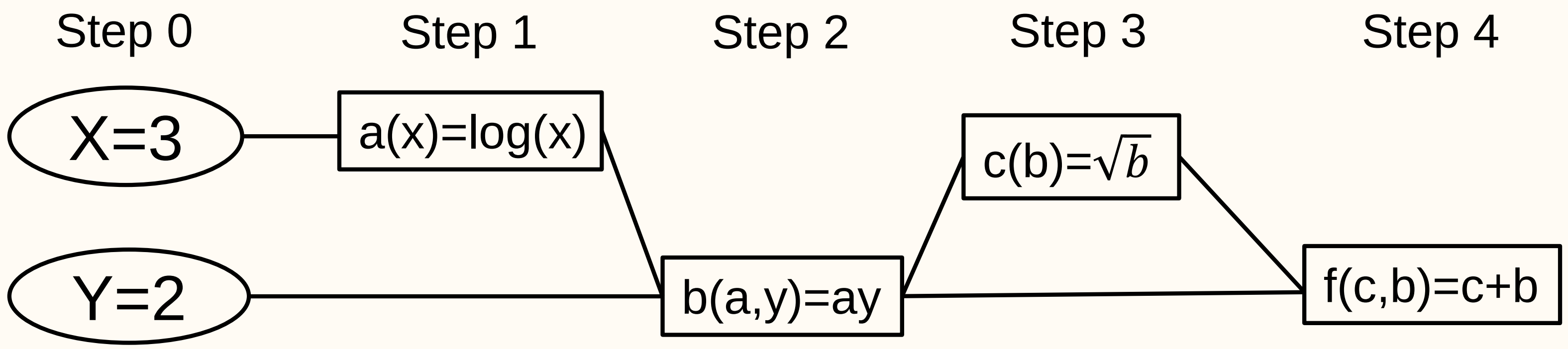
Forward pass on computation graph

In the forward pass, sequentially compute $y_1, \ldots, y_L$ via

$$y_\ell = f_\ell([y_i : \text{for } i \to \ell])$$

```
# Use 1-based indexing
# y[1] given
for l = 2,...,L
  inputs = [y[i] for j such that (i->l)]
  y[l] = f[l].eval(inputs)
end
```

Forward-mode autodiff



0. $x = 3, y = 2,\ \frac{\partial x}{\partial x} = 1,\ \frac{\partial x}{\partial y} = 0,\ \frac{\partial y}{\partial x} = 0,\ \frac{\partial y}{\partial y} = 1$

1. $a = \log x = \log 3,\ \frac{\partial a}{\partial x} = \frac{1}{x} \cdot \frac{\partial x}{\partial x} = \frac{1}{3},\ \frac{\partial a}{\partial y} = 0$

2. $b = ya = 2\log 3,\ \frac{\partial b}{\partial x} = \frac{\partial y}{\partial x} a + y\frac{\partial a}{\partial x} = \frac{2}{3},\ \frac{\partial b}{\partial y} = \frac{\partial y}{\partial y} a + y\frac{\partial a}{\partial y} = a = \log 3$

3. $c = \sqrt{b} = \sqrt{2\log 3},\ \frac{\partial c}{\partial x} = \frac{1}{2\sqrt{b}} \frac{\partial b}{\partial x} = \frac{1}{3\sqrt{2\log 3}},\ \frac{\partial c}{\partial y} = \frac{1}{\sqrt{b}} \frac{\partial b}{\partial y} = \frac{1}{2}\sqrt{\frac{\log 3}{2}}$

4. $f = c + b = \sqrt{2\log 3} + 2\log 3,\ \frac{\partial f}{\partial x} = \frac{\partial c}{\partial x} + \frac{\partial b}{\partial x} = \frac{1}{3}\left(2 + \frac{1}{3\sqrt{2\log 3}}\right),\ \frac{\partial f}{\partial y} = \frac{\partial c}{\partial y} + \frac{\partial b}{\partial y} = \frac{1}{2}\sqrt{\frac{\log 3}{2}} + \log 3$

Computation does not involve x or derivatives of x

Computation only depends on node b

Computation only depends on nodes b and c

Backprop on computation graph

```
# Use 1-based indexing
# y[1],...,y[L] already computed

g[:] = 0 // .zero_grad()

g[L] = 1 // dy[L]/dy[L]=1
for l = L,...,2
  for i such that (i->l)
    g[i] += g[l]*f[l].grad(i)
  end
end
```

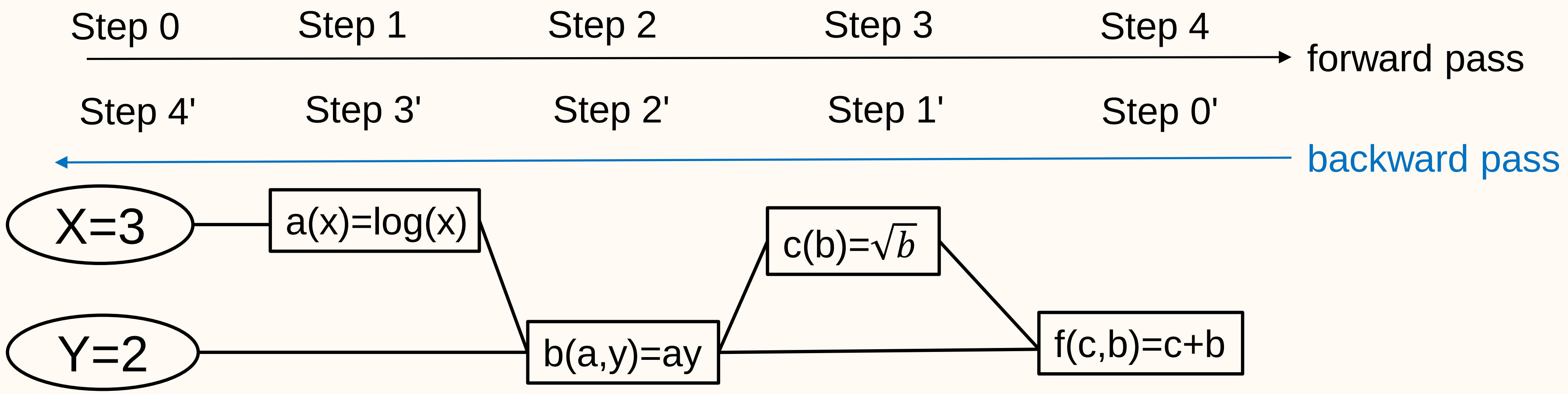
To perform backprop[#], use

$$\frac{\partial y_L}{\partial y_i} = \sum_{\ell : i \to \ell} \frac{\partial y_L}{\partial y_\ell} \frac{\partial f_\ell}{\partial y_i}$$

to sequentially compute $\frac{\partial y_L}{\partial y_L}, \frac{\partial y_L}{\partial y_{L-1}}, \dots, \frac{\partial y_L}{\partial y_1}$.

[#]When y_i is not a leaf node in the computation graph, there is a slight ambiguity in the meaning of $\frac{\partial y_L}{\partial y_i}$. Roughly, define $\frac{\partial y_L}{\partial y_i}$ by letting y_i be a variable independent of $y_1, \dots, y_{i-1}$ (imagine detaching all edges entering y_i) and then taking the derivative.

Reverse-mode autodiff (backprop)



0. $x = 3,\ y = 2$
1. $a = \log 3$
2. $b = 2\log 3$
3. $c = \sqrt{2\log 3}$
4. $f = \sqrt{2\log 3} + 2\log 3$

0'. $\frac{\partial f}{\partial f} = 1$

1'. $\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}1 = 1$

2'. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} = \frac{1}{2\sqrt{b}}1 + 1 = \frac{1}{2\sqrt{2\log 3}} + 1$

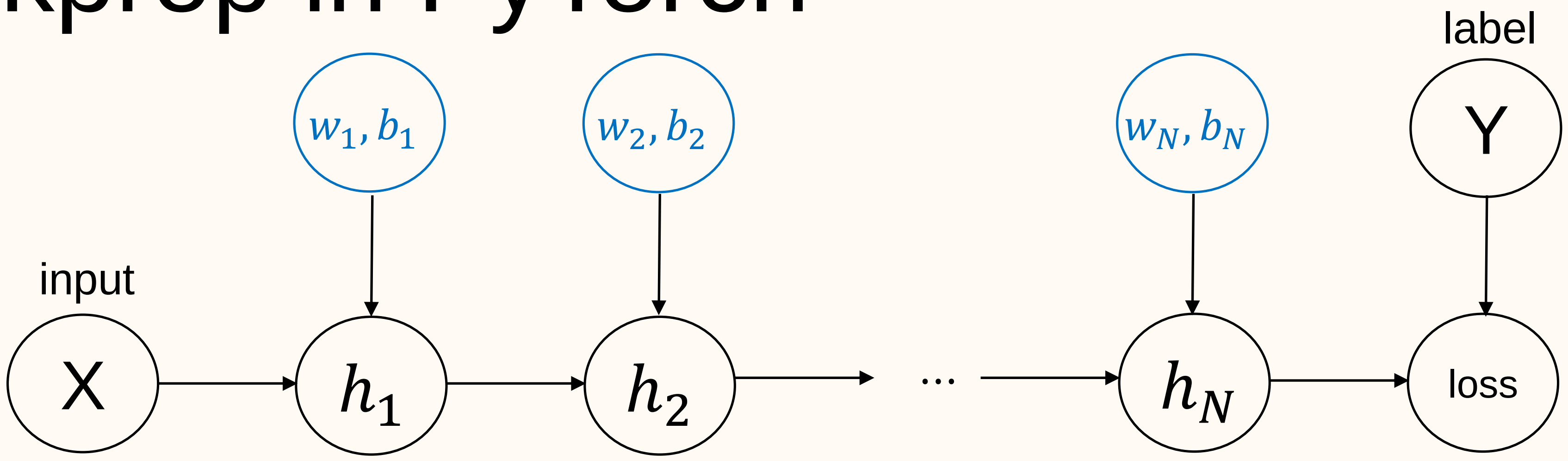
3'. $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\ \frac{\partial b}{\partial a} = \frac{\partial f}{\partial b}\ y = 2 + \frac{1}{\sqrt{2\log 3}}$

4'. $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\ \frac{\partial a}{\partial x} = \frac{\partial f}{\partial a}\frac{1}{x} = \frac{1}{3}\left(2 + \frac{1}{\sqrt{2\log 3}}\right)$

$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\ \frac{\partial b}{\partial y} = \frac{\partial f}{\partial b}\ a = \frac{1}{2}\sqrt{\frac{\log 3}{2}} + \log 3$

Backward pass depends on node values computed in forward pass.

Backprop in PyTorch



In NN training, parameters and fixed inputs are distinguished. In PyTorch, you (1) clear the existing gradient with `.zero_grad()` (2) forward-evaluate the loss function by providing the input and label and (3) perform backprop with `.backward()`.

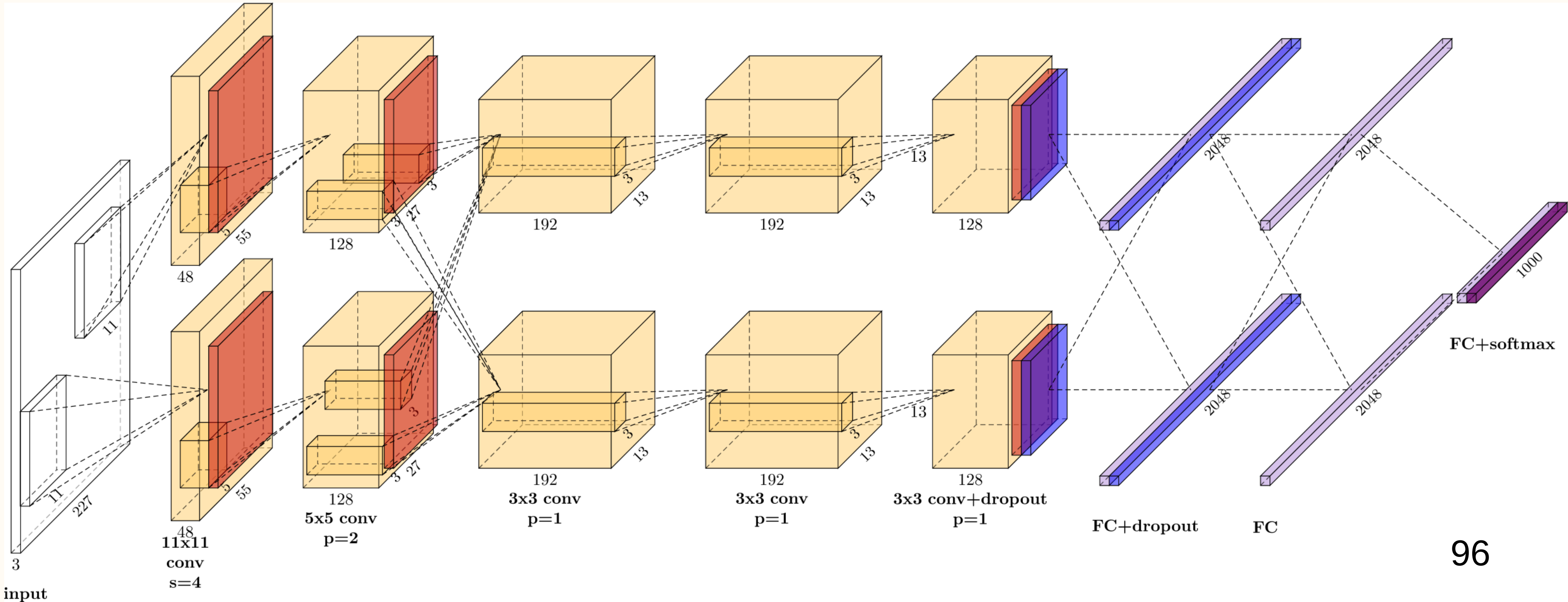
The forward pass stores the intermediate neuron values so that they can later be used in backprop. In the test loop, however, we don't compute gradients so the intermediate neuron values are unnecessary. The `torch.no_grad()` context manager allows intermediate node values to discarded or not be stored. This saves memory and can accelerate the test loop.

Linear layers have too may parameters

AlexNet: Conv layer params: 2,469,696 (4%)

Linear layer params: 58,631,144 (96%)

Total params: 61,100,840

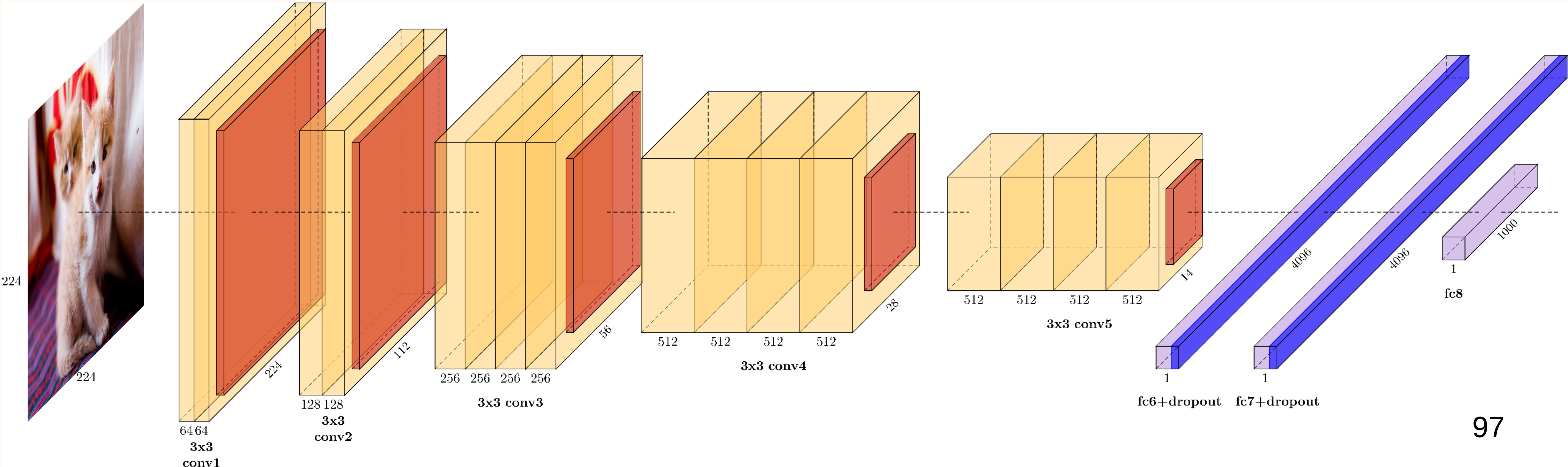


Linear layers have too may parameters

VGG19: Conv layer params: 20,024,384 (14%)

Linear layer params: 123,642,856 (86%)

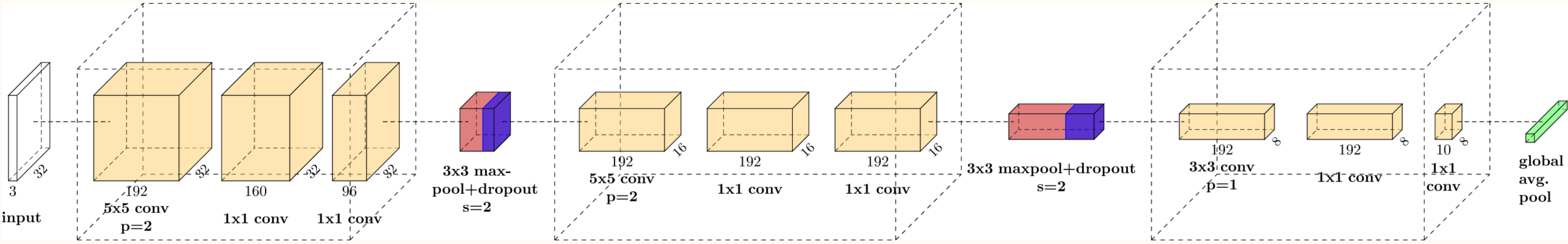
Total params: 143,667,240



Network in Network (NiN) Network

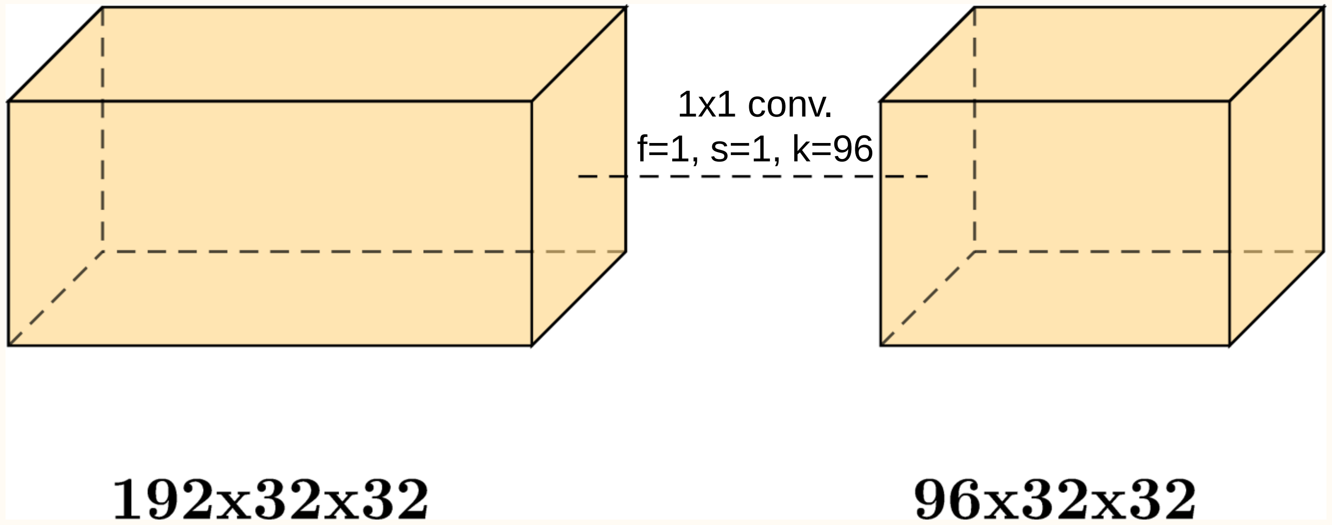
NiN for CIFAR10.

- Remove linear layers to reduce parameters. Use global average pool instead.
- Weight decay 1×10^{-5}.
- Dropout(0.5). (dropout after pool is not consistent with modern practice.)
- Maxpool $f = 3$, $s = 2$. Use `ceil_mode=True` so that $\frac{32-3}{2} + 1 = 15.5$ is rounded up to 16. Default behavior of PyTorch is to round down.



1x1 convolution

A 1×1 convolution is like a fully connected layer acting independently and identically on each spatial location.



- 96 filters act on 192 channels separately for each pixel
- $96 \times 192 + 96$ parameters for weights and biases

Regular conv. layer

Input: $X \in \mathbb{R}^{C_0 \times m \times n}$

- Select an $f \times f$ patch $\tilde{X} = X[:, i : i + f, j : j + f]$.
- Inner product $\tilde{X}$ and $w_1, \dots, w_{C_1} \in \mathbb{R}^{C_0 \times f \times f}$ and add bias $b_1 \in \mathbb{R}^{C_1}$.
- Apply σ. (Output in $\mathbb{R}^{C_1}$.)

Repeat this for all patches. Output in $X \in \mathbb{R}^{C_1 \times (m-f+1) \times (n-f+1)}$.

Repeat this for all batch elements.

“Network in Network”

Input: $X \in \mathbb{R}^{C_0 \times m \times n}$

- Select an $f \times f$ patch $\tilde{X} = X[\, i : i + f, j : j + f]$.
- Inner product $\tilde{X}$ and $w_1, \ldots, w_{C_1} \in \mathbb{R}^{C_0 \times f \times f}$ and add bias $b_1 \in \mathbb{R}^{C_1}$.
- Apply σ. (Output in $\mathbb{R}^{C_1}$.)
- Apply $\mathrm{Linear}_{A_2, b_2}(x)$ where $A_2 \in \mathbb{R}^{C_2 \times C_1}$ and $b_2 \in \mathbb{R}^{C_2}$.
- Apply σ. (Output in $\mathbb{R}^{C_2}$.)
- Apply $\mathrm{Linear}_{A_3, b_3}(x)$ where $A_3 \in \mathbb{R}^{C_3 \times C_2}$ and $b_3 \in \mathbb{R}^{C_3}$.
- Apply σ. (Output in $\mathbb{R}^{C_3}$.)

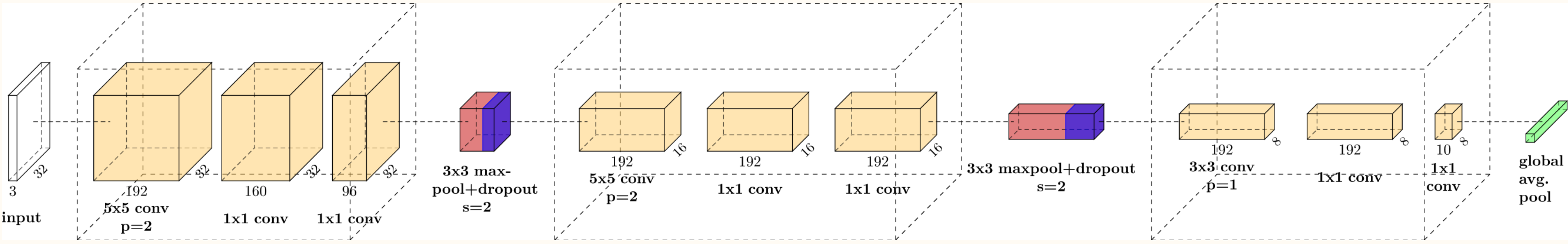
Repeat this for all patches. Output in $X \in \mathbb{R}^{C_3 \times (m-f+1) \times (n-f+1)}$. Repeat this for all batch elements.

Why is this equivalent to (3x3 conv)-(1x1 conv)-(1x1 conv)?

Global average pool

When using CNNs for classification, position of object is not important.

The global average pool has no trainable parameters (linear layers have many) and it is translation invariant. Global average pool removes the spatial dependency.



Architectural contribution: NiN Network

Used 1x1 convolutions to increase the representation power of the convolutional modules.

Replaced linear layer with average pool to reduce number of trainable parameters.

First step in the trend of architectures becoming more abstract. Modern CNNs are built with smaller building blocks.

GoogLeNet (Inception v1)

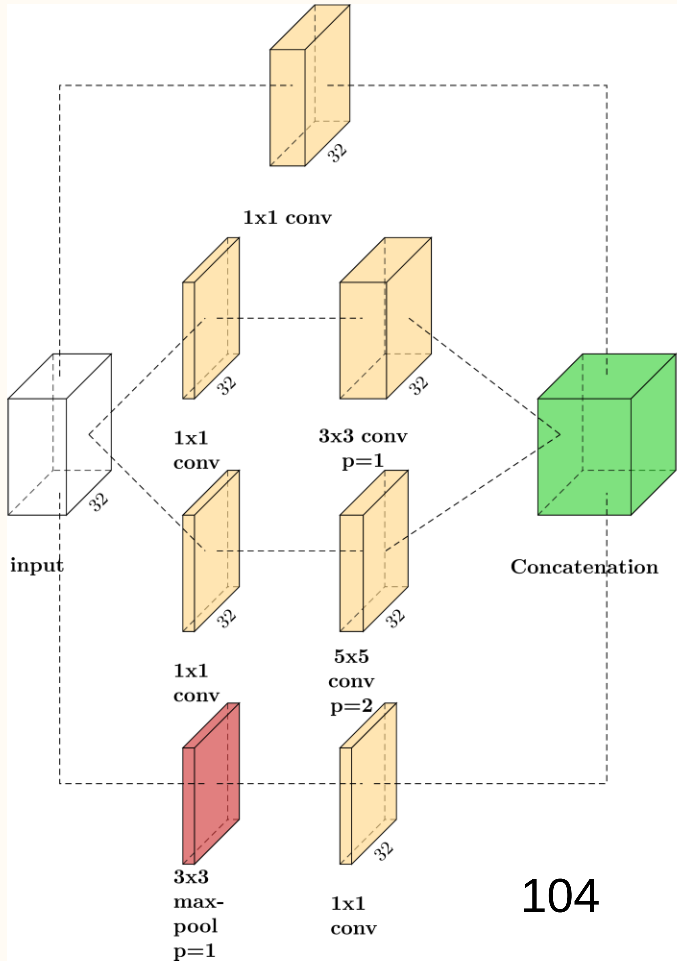
Utilizes the *inception module*. Structure inspired by NiN and name inspired by 2010 Inception movie meme.

Used 1×1 convolutions.

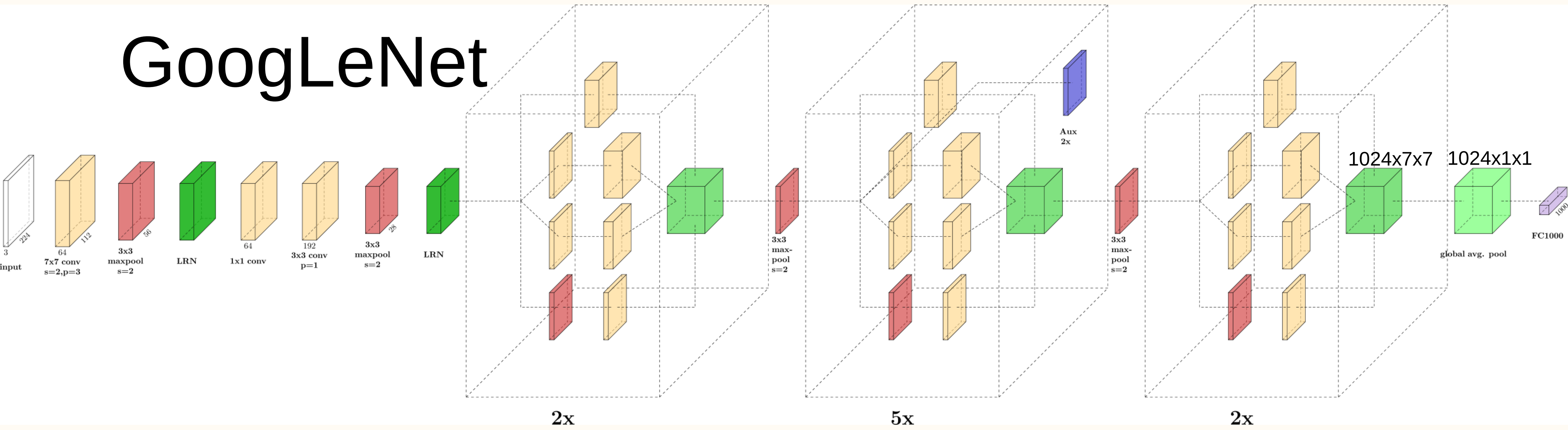
- Increased depth adds representation power (improves ability to represent nonlinear functions).
- Reduce the number of channels before the expensive 3×3 and 5×5 convolutions, and thereby reduce number of trainable weights and computation time. (Cf. hw5)

The name GoogLeNet is a reference to the authors' Google affiliation and is an homage to LeNet.

Inception module



GoogLeNet



k=256 480 512 512 512 528 832 832 1024

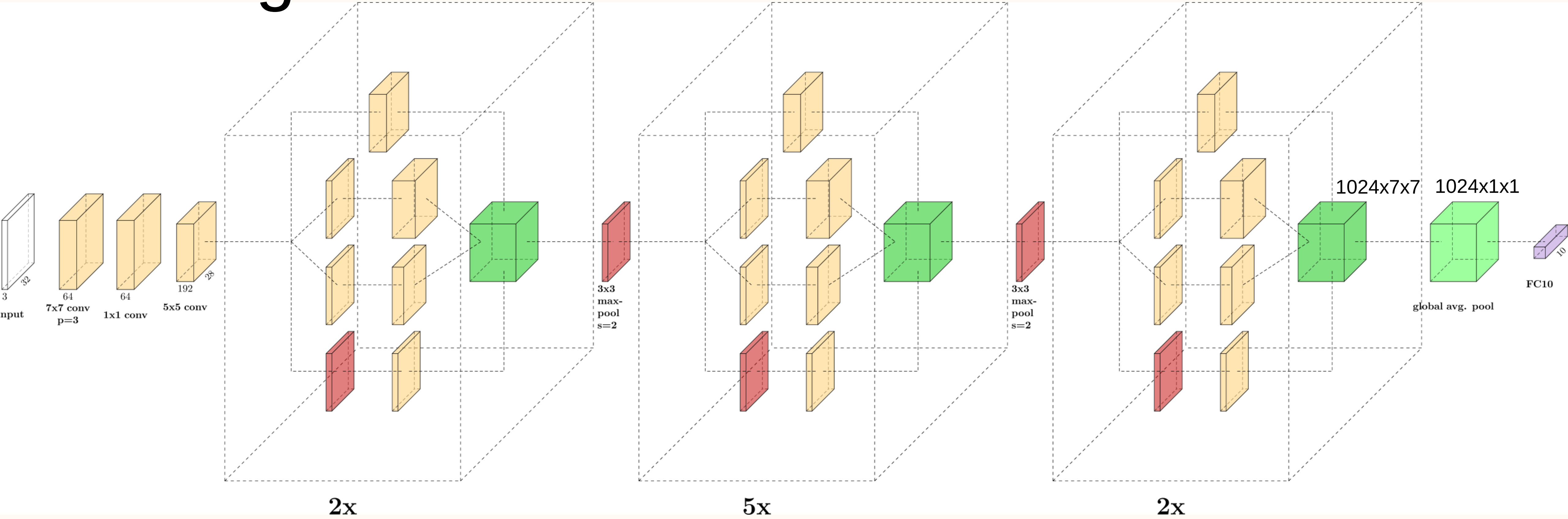
k=64	k=128	k=192	k=160	k=128	k=112	k=256	k=256	k=384
96 128	128 192	96 208	112 224	128 256	144 288	160 320	160 320	192 384
16 32	32 96	16 48	24 64	24 64	32 64	32 128	32 128	48 128
32	64	64	64	64	64	128	128	128

🟩 Local Response Normalization

🟥 Maxpool $f = 3$, $s = 2$. Use `ceil_mode=True`.

🟦 Two auxiliary classifiers used to slightly improve training. No longer necessary with batch norm.

GoogLeNet for CIFAR10



k=256	480	512	512	512	528	832	832	1024
k=64	k=128	k=192	k=160	k=128	k=112	k=256	k=256	k=384
96 128	128 256	96 208	112 224	128 256	144 288	160 320	160 320	192 384
16 32	24 64	16 48	24 64	24 64	32 64	32 128	32 128	48 128
32	64	64	64	64	64	128	128	128

Architectural contribution: GoogLeNet

Demonstrated that more complex modular neural network designs can outperform VGGNet’s straightforward design.

Together with VGGNet, demonstrated the importance of depth.

Kickstarted the research into deep neural network architecture design.

Batch normalization

The first step of many data processing algorithms is often to normalize data to have zero mean and unit variance.

- Step 1. Compute $\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} X_i$, $\widehat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}(X_i - \hat{\mu})^2$

$$\hat{X}_i = \frac{X_i - \hat{\mu}}{\sqrt{\widehat{\sigma^2} + \varepsilon}}$$

- Step 2. Run method with data $\hat{X}_1, \dots, \hat{X}_N$

Batch normalization (BN) (sort of) enforces this normalization layer-by-layer. BN is an indispensable tool for training very deep neural networks. Theoretical justification is weak.

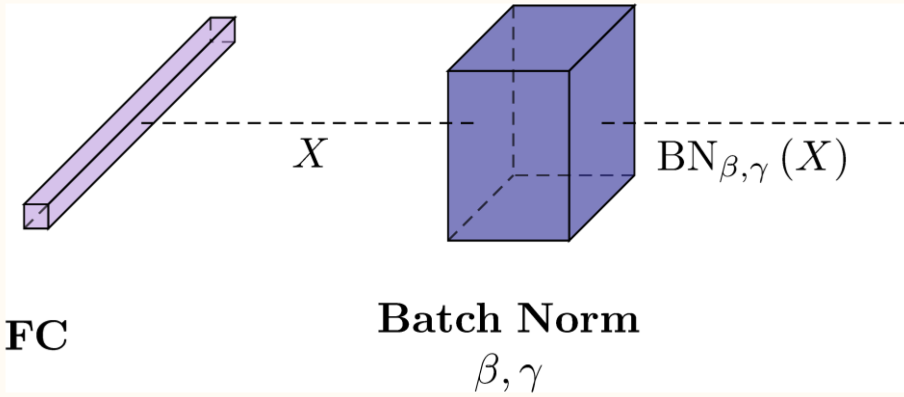
BN for linear layers

Underlying assumption: Each element of the batch is an IID sample.

Input: X (batch size) $\times$ (# entries)

output: $\mathrm{BN}_{\beta,\gamma}(X)$. $\text{shape}\left(\mathrm{BN}_{\beta,\gamma}(X)\right) = \text{shape}(X)$

$\mathrm{BN}_{\beta,\gamma}$ for linear layers acts independently over neurons.



$$\hat{\mu}[:] = \frac{1}{B}\sum_{b=1}^{B} X[b,:]$$

$$\hat{\sigma}^2[:] = \frac{1}{B}\sum_{b=1}^{B} (X[b,:] - \hat{\mu}[:])^2$$

$$\mathrm{BN}_{\gamma,\beta}(X)[b,:] = \gamma[:]\frac{X[b,:] - \hat{\mu}[:]}{\sqrt{\hat{\sigma}^2[:] + \varepsilon}} + \beta[:] \qquad b = 1, \ldots, B$$

where operations are elementwise. BN normalizes each output neuron. The mean and variance are explicitly controlled through learned parameters β and γ. In Pytorch, `nn.BatchNorm1d`.

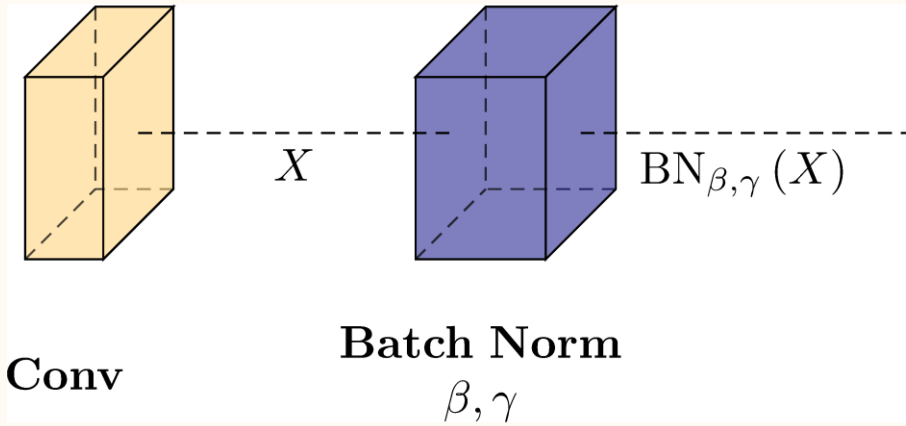
BN for convolutional Layers

Underlying assumption: Each element of the batch, horizontal pixel, and vertical pixel is an IID sample.*

Input: X $(\text{batch size}) \times (\text{channels}) \times (\text{vertical dim}) \times (\text{horizontal dim})$

output: $\text{BN}_{\beta,\gamma}(X)$. $\text{shape}\left(\text{BN}_{\beta,\gamma}(X)\right) = \text{shape}(X)$

$\text{BN}_{\beta,\gamma}$ for conv. layers acts independently over channels.



$$\hat{\mu}[:] = \frac{1}{BPQ} \sum_{b=1}^{B} \sum_{i=1}^{P} \sum_{j=1}^{Q} X[b,:,i,j]$$

$$\hat{\sigma}^2[:] = \frac{1}{BPQ} \sum_{b=1}^{B} \sum_{i=1}^{P} \sum_{j=1}^{Q} (X[b,:,i,j] - \hat{\mu}[:])^2$$

$$\text{BN}_{\gamma,\beta}(X)[b,:,i,j] = \gamma[:] \frac{X[b,:,i,j] - \hat{\mu}[:]}{\sqrt{\hat{\sigma}^2[:] + \varepsilon}} + \beta[:] \qquad \begin{aligned} b &= 1,\dots,B \\ i &= 1,\dots,P \\ j &= 1,\dots,Q \end{aligned}$$

BN normalizes over each convolutional filter. The mean and variance are explicitly controlled through learned parameters β and γ. In Pytorch, `nn.BatchNorm2d`.

*Assuming translation invariance, one can argue that different pixels have identical distributions. The independence assumption, however, is clearly false.

BN during testing

$\hat{\mu}$ and $\hat{\sigma}$ are estimated from batches during training. During testing, we don't update the NN, and we may only have a single input (so no batch).

There are 2 strategies for computing final values of $\hat{\mu}$ and $\hat{\sigma}$:

1. After training, fix all parameters and evaluate NN on full training set to compute $\hat{\mu}$ and $\hat{\sigma}$ layer-by-layer. Store this computed value. (Computation of $\hat{\mu}$ and $\hat{\sigma}$ must be done sequentially layer-by-layer. Why?)
2. During training, compute running average of $\hat{\mu}$ and $\hat{\sigma}$. This is the default behavior of PyTorch.

In PyTorch, use `model.train()` and `model.eval()` to switch BN behavior between training and testing.

Discussion of BN

BN does not change the representation power of NN; since β and γ are trained, the output of each layer can have any mean and variance. However, controlling the mean and variance as explicit trainable parameters makes training easier.

With BN, the choice of batch size becomes a more important hyperparameter to tune.

BN is indispensable in practice. Training of VGGNet and GoogLeNet becomes much easier with BN. Training of ResNet requires BN.

BN and internal covariate shift

BN has insufficient theoretical justification.

The original paper by Ioffe and Szegedy hypothesized that BN mitigates internal covariate shift (ICS), the shift in the mean and variance of the intermediate layer neurons throughout the training, and that this mitigation leads to improved training.

$$\text{BN} \Rightarrow \text{(reduced ICS)} \Rightarrow \text{(improved training)}$$

However, Santukar et al. demonstrated that when experimentally measured, BN does not mitigate ICS, but nevertheless improves the training.

$$\text{BN} \not\Rightarrow \text{(reduced ICS)}$$

Nevertheless

$$\text{BN} \Rightarrow \text{(improved training performance)}$$

S. Ioffe and C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, *ICML*, 2015.
S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry, How does batch normalization help optimization?, *NeurIPS*, 2018.

BN and internal covariate shift

Santukar et al. argues that

$$\text{BN} \Rightarrow (\text{smoother loss landscape}) \Rightarrow (\text{improved training performance})$$

While this claim is more evidence-based than that of Ioffe and Szegedy, it is still not conclusive. It is also unclear why BN makes the loss landscape smoother, and it is not clear whether the smoother loss landscape fully explains the improved training performance.

This story is a cautionary tale: we should carefully distinguish between speculative hypotheses and evidence-based claims, even in a primarily empirical subject.

S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry, How does batch normalization help optimization?, *NeurIPS*, 2018.

BN has trainable parameters

BN is usually not considered a trainable layer, much like pooling or dropout, and they are usually excluded when counting the “depth” of a NN. However, BN does have trainable parameters. Interestingly, if one randomly initializes a CNN, freezes all other parameters, and only train BN parameters, the performance is surprisingly good.

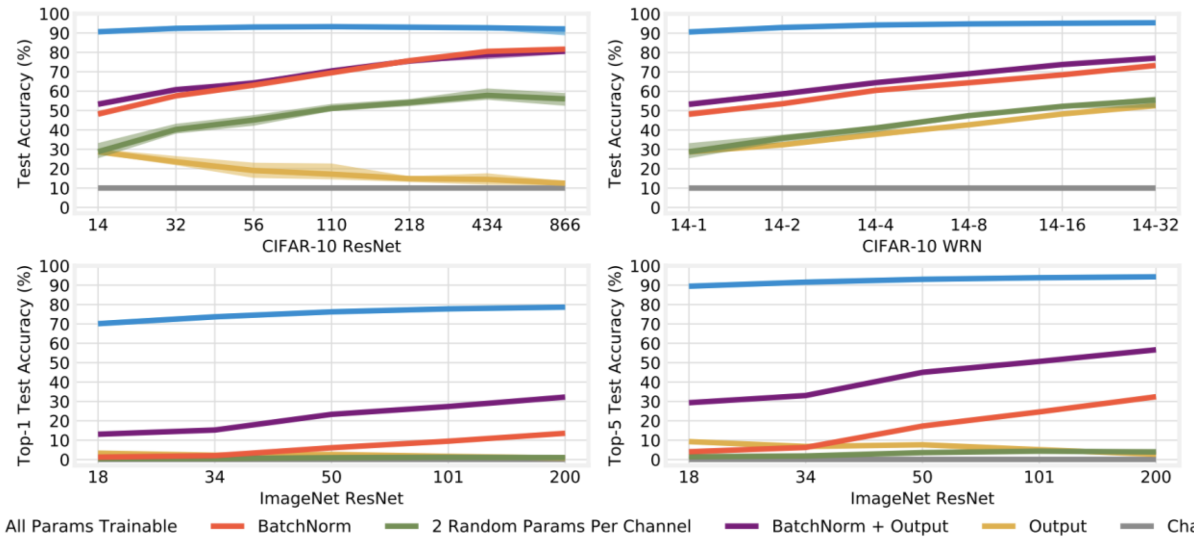


Figure 2: Accuracy of ResNets for CIFAR-10 (top left, deep; top right, wide) and ImageNet (bottom left, top-1 accuracy; bottom right, top-5 accuracy) with different sets of parameters trainable.

J. Frankle, D. J. Schwab, and A. S. Morcos, Training BatchNorm and only BatchNorm: On the expressive power of random features in CNNs, *NeurIPS SEDL Workshop*, 2019.

Discussion of BN

BN seems to also act as a regularizer, and for some reason subsumes effect Dropout. (Using dropout together with BN seems to worsen performance.) Since BN has been popularized, Dropout is used less often.[*]

After training, functionality of BN can be absorbed into the previous layer when the previous layer is a linear layer or a conv layer. (Cf. homework 6.)

The use of batch norm makes the scaling of weight initialization less important irrelevant.

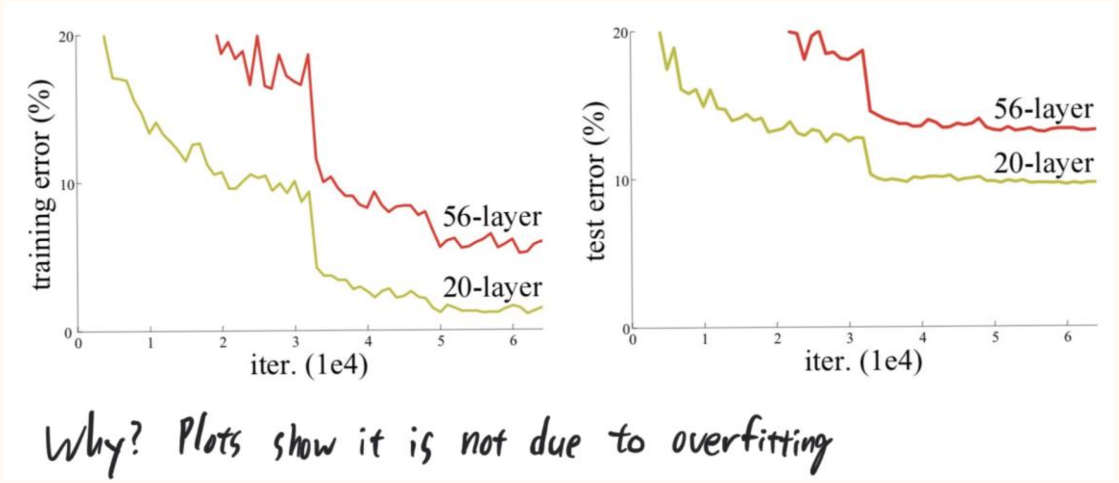
Use `bias=false` on layers preceding BN, since β subsumes the bias.

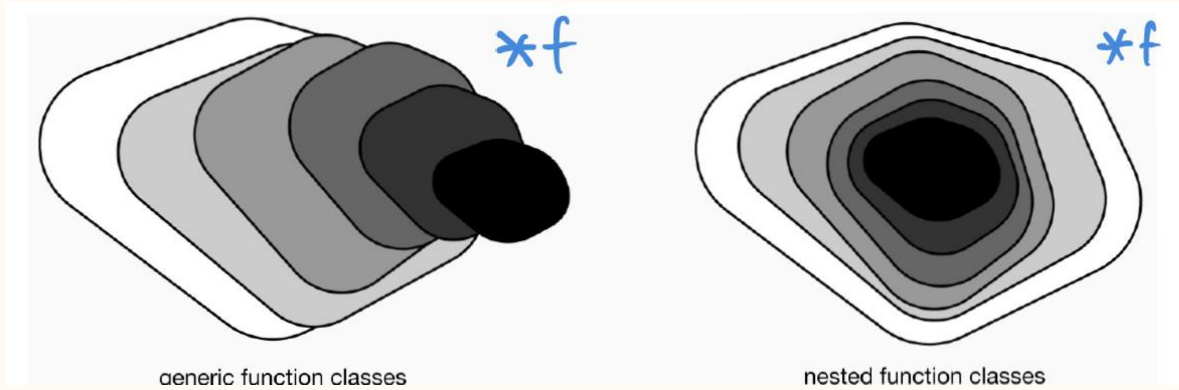
[*]X. Li, S. Chen, X. Hu and J. Yang, Understanding the disharmony between dropout and batch normalization by variance shift, *CVPR*, 2019.

Residual Network (ResNet)

Winner of 2015 ImageNet Challenge

Observation: Excluding the issue of computation cost, more layers it not always better





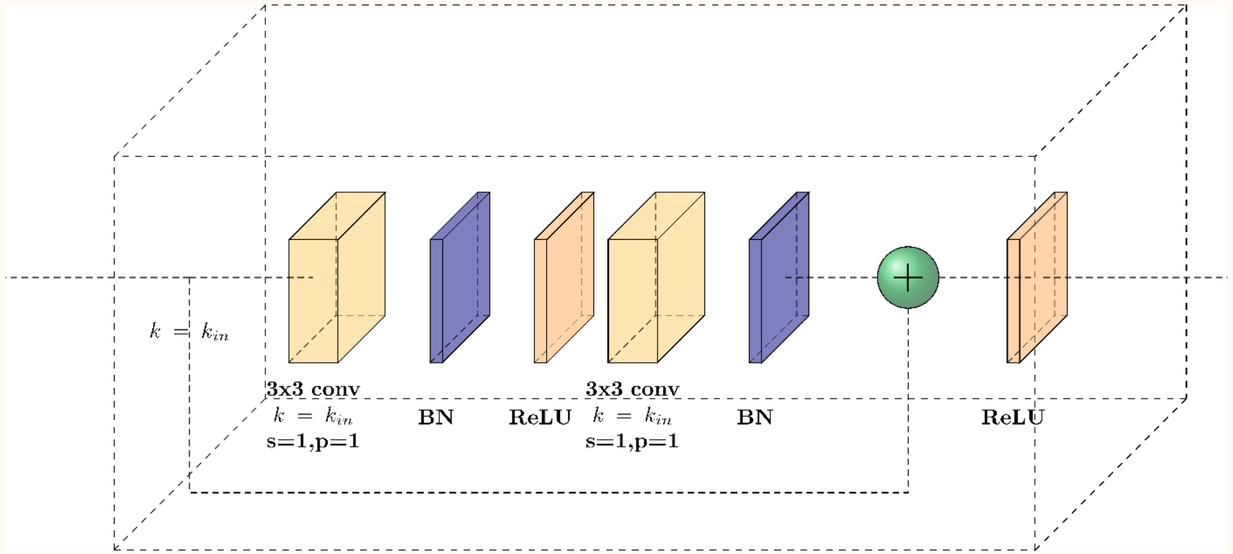
Hypothesis 1: Deeper networks are harder to train.
Is there a way to train a shallow network and embed it in a deeper network?

Hypothesis 2: The deeper networks may be worse approximations of the true unknown function. Find an architecture representing a strictly increasing function class as a function of depth.

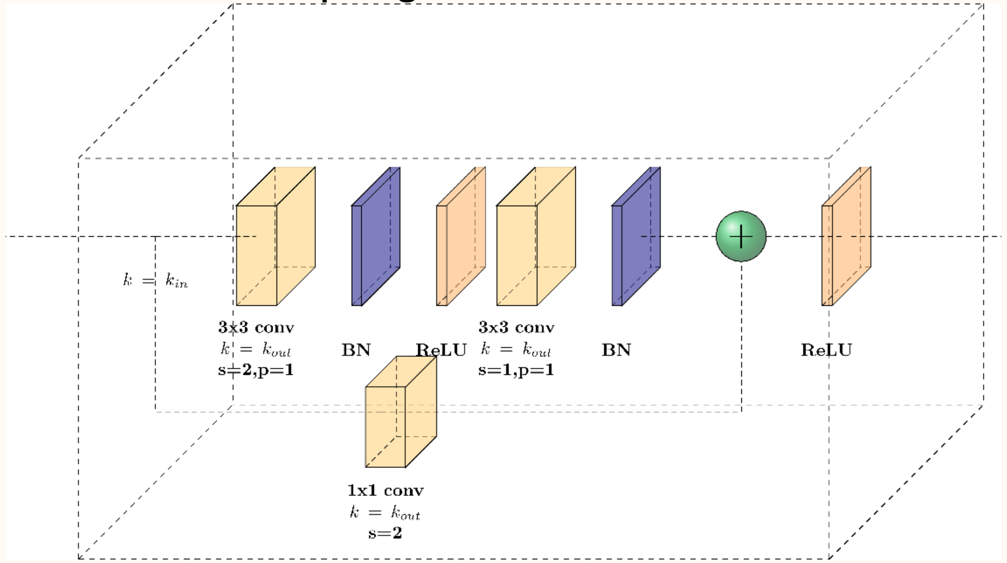
Residual blocks

Use a residual connection so that [all weights=0] correspond to [block=identity]*

regular residual block

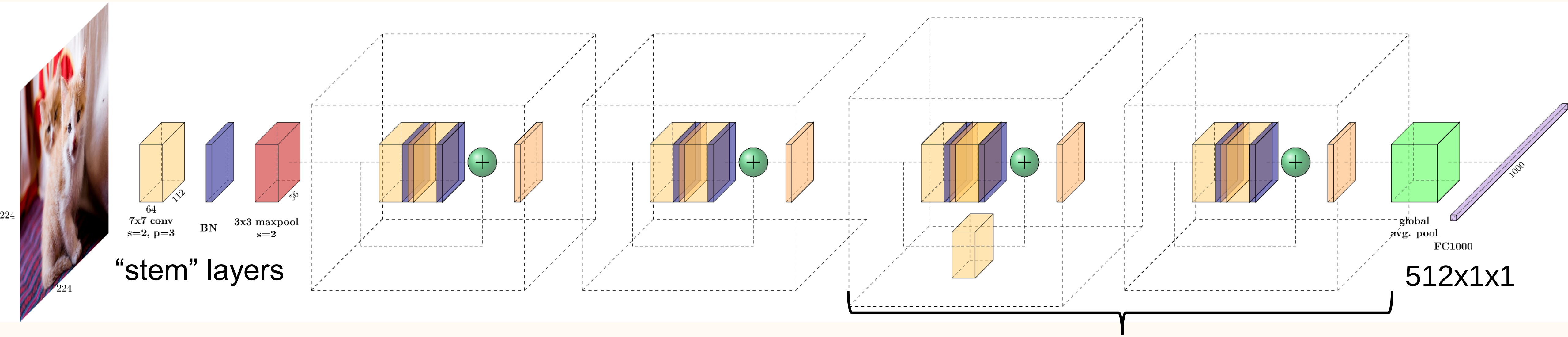


downsampling residual block



Regular block must preserve spatial dimension and number of channels.
Downsampling block halves the spatial dimension and changes the number of channels.

ResNet18



“stem” layers

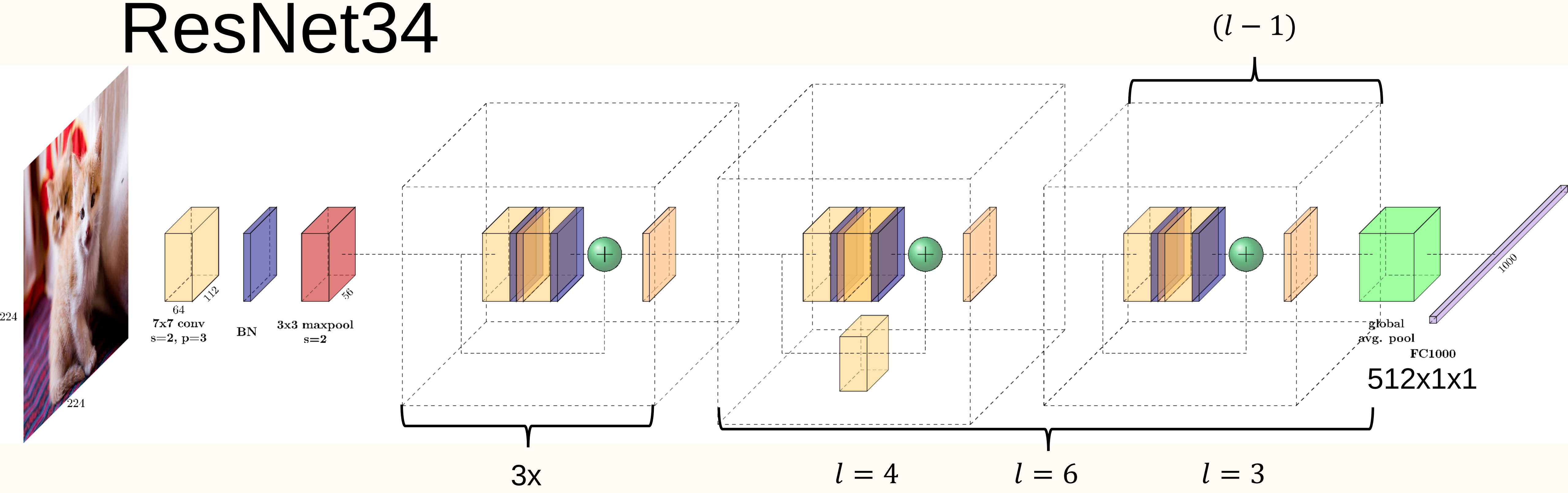
512x1x1

3x

64x56x56 → 128x28x28 → 256x14x14 → 512x7x7

Layer count excludes BN even though BN has trainable parameters.

ResNet34



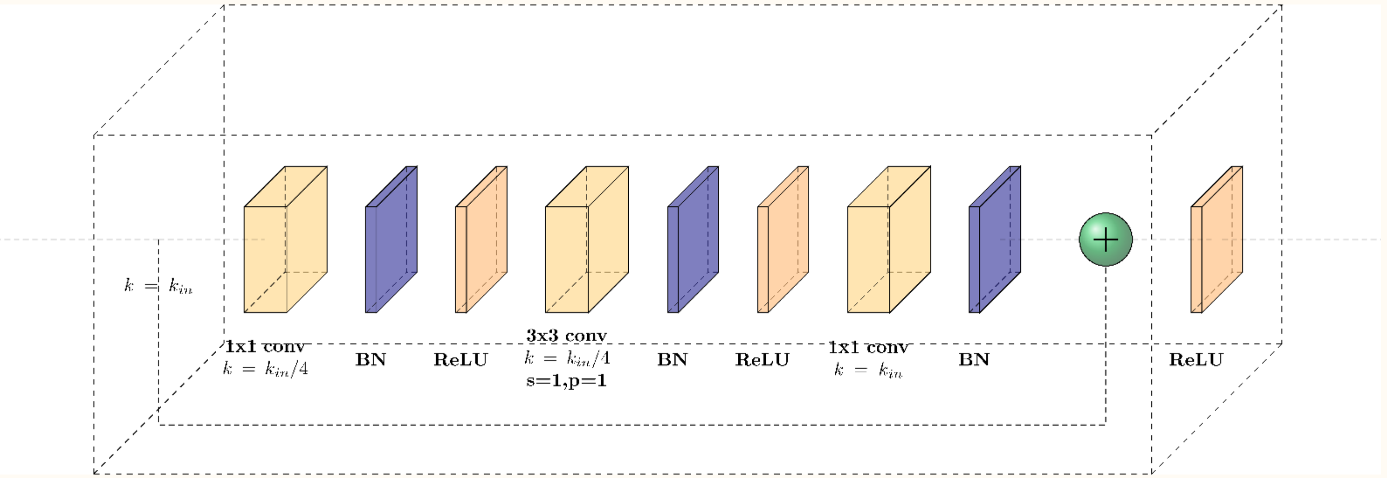
64x56x56 → 128x28x28 → 256x14x14 → 512x7x7

A trained ResNet18 architecture can be exactly fitted into a ResNet34: copy over the parameters and set parameters of the additional blocks to be 0. The additional blocks with only serve to apply an additional ReLU, but this makes no difference as ReLU is idempotent.

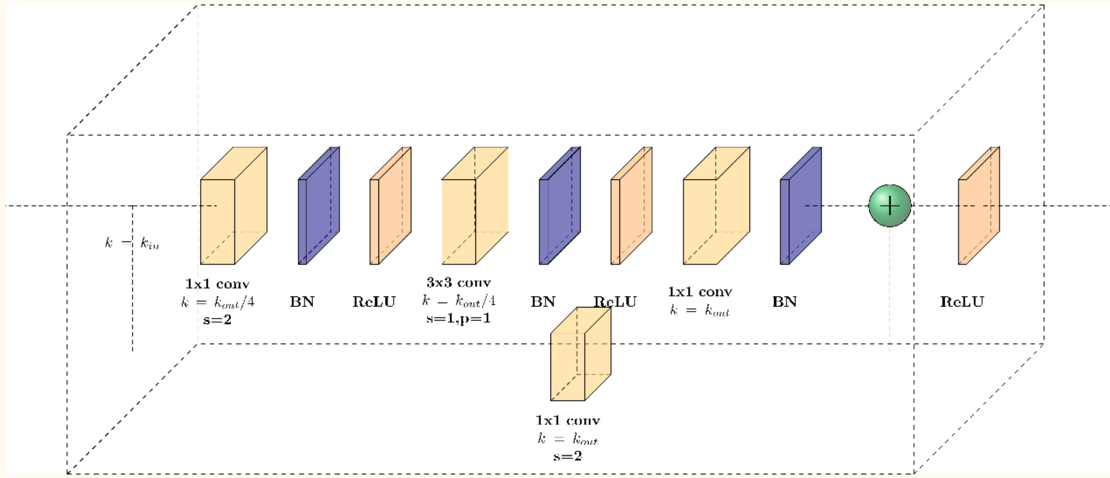
ResNet blocks for deeper ResNets

ResNet in fact goes deeper. For the deeper variants, computation cost becomes more significant. To remedy this cost, use 1×1 conv to reduce number of channels, perform costly 3x3 convolution, and use 1x1 conv to restore the number of channels. This bottleneck” structure is adapted from GoogLeNet.

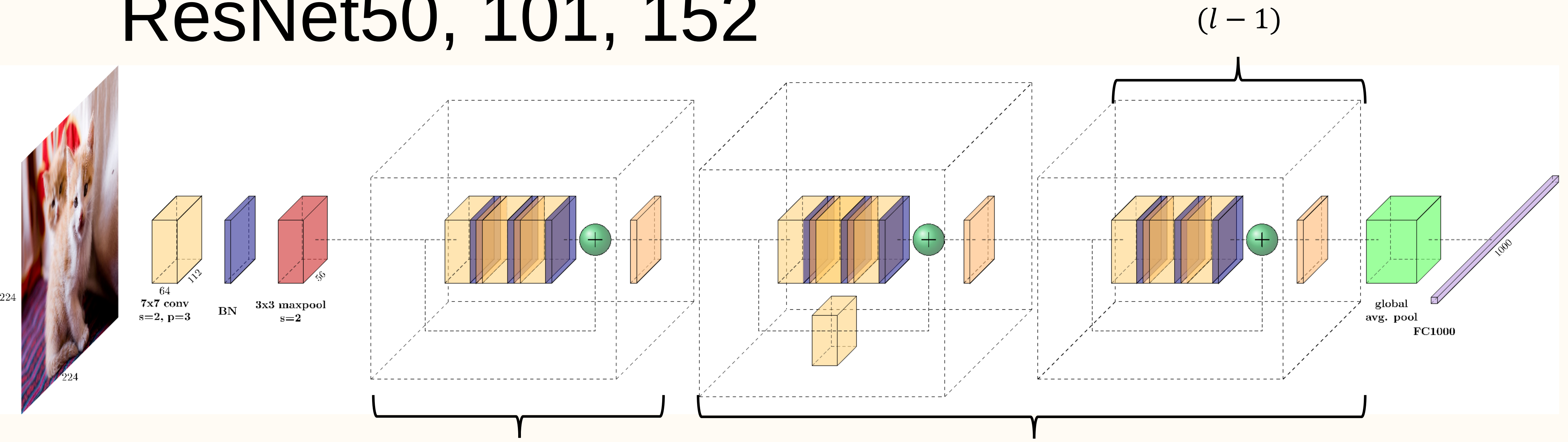
regular residual block with bottleneck



downsampling residual block with bottleneck



ResNet50, 101, 152

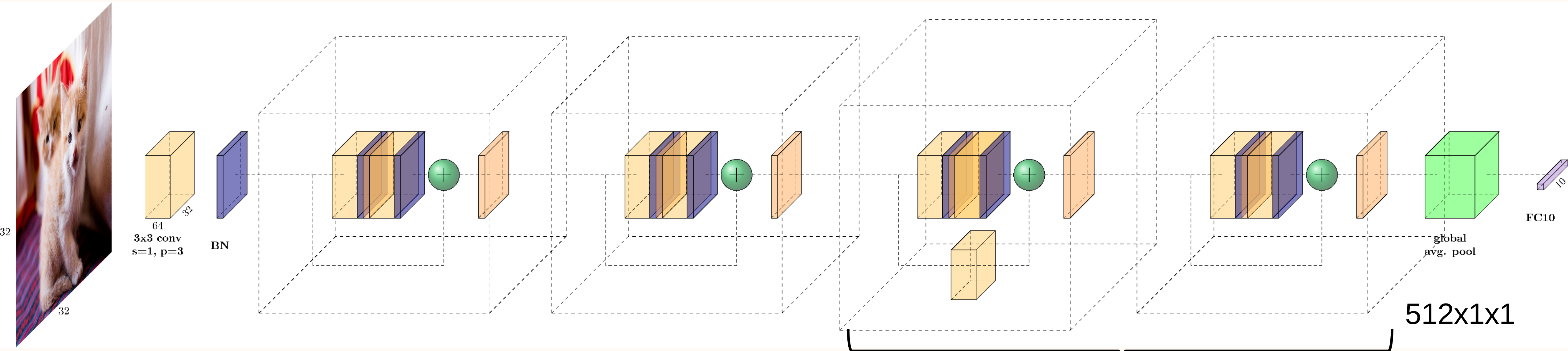


3x

ResNet50:	$l = 4$	$l = 6$	$l = 3$
ResNet101:	$l = 4$	$l = 23$	$l = 3$
ResNet152:	$l = 8$	$l = 36$	$l = 3$

64x56x56 → 128x28x28 → 256x14x14 → 512x7x7

ResNet18 for cifar10



3x

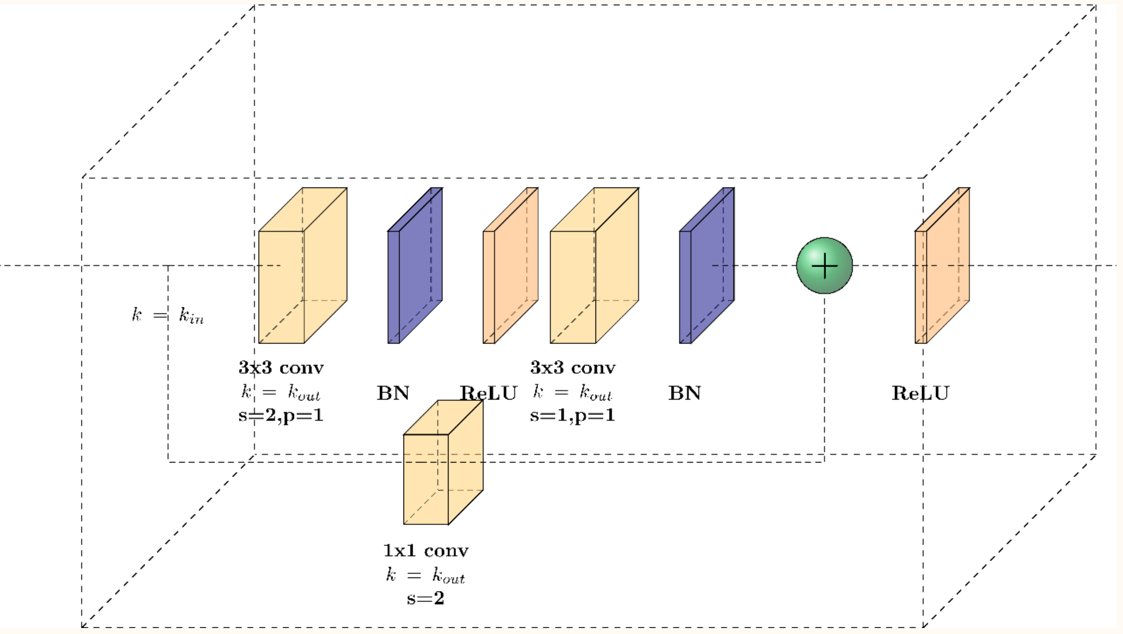
64x32x32 → 128x16x16 → 256x8x8 → 512x4x4

ResNet{34,50,101,152} for CIFAR10. The intermediate layers are the same as before.

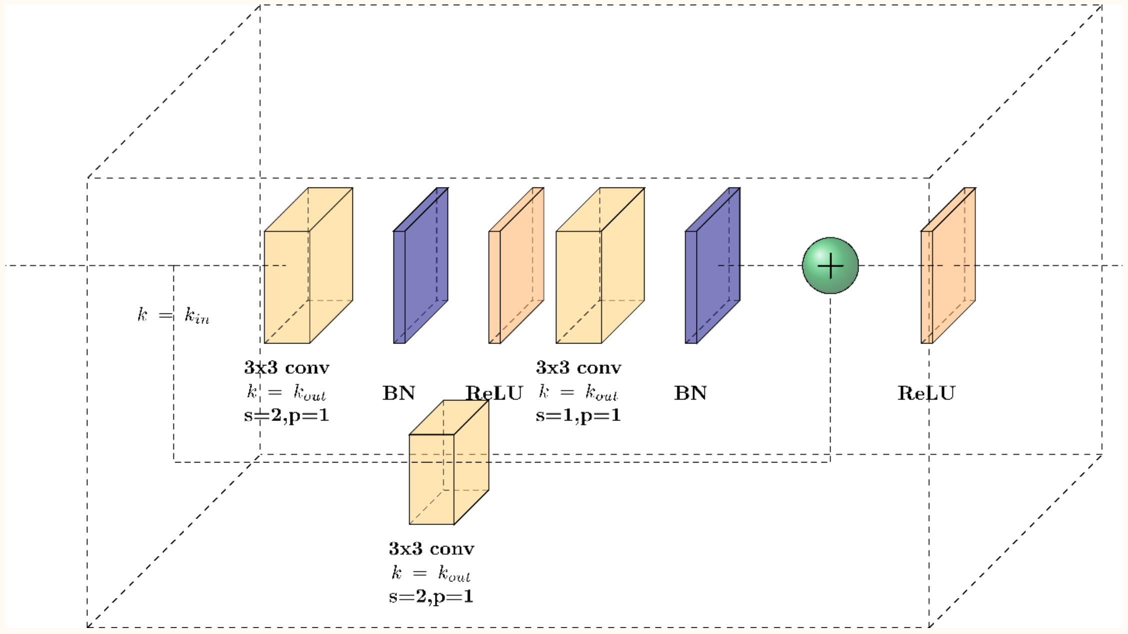
ResNet v1.5

In the bottleneck blocks performing downsampling, the use of 1x1 conv with stride 2 is suboptimal as the operation simply ignores 75% of the neurons. ResNet v1.5 replaces them with 3x3 conv with stride 2.

downsampling residual block v1



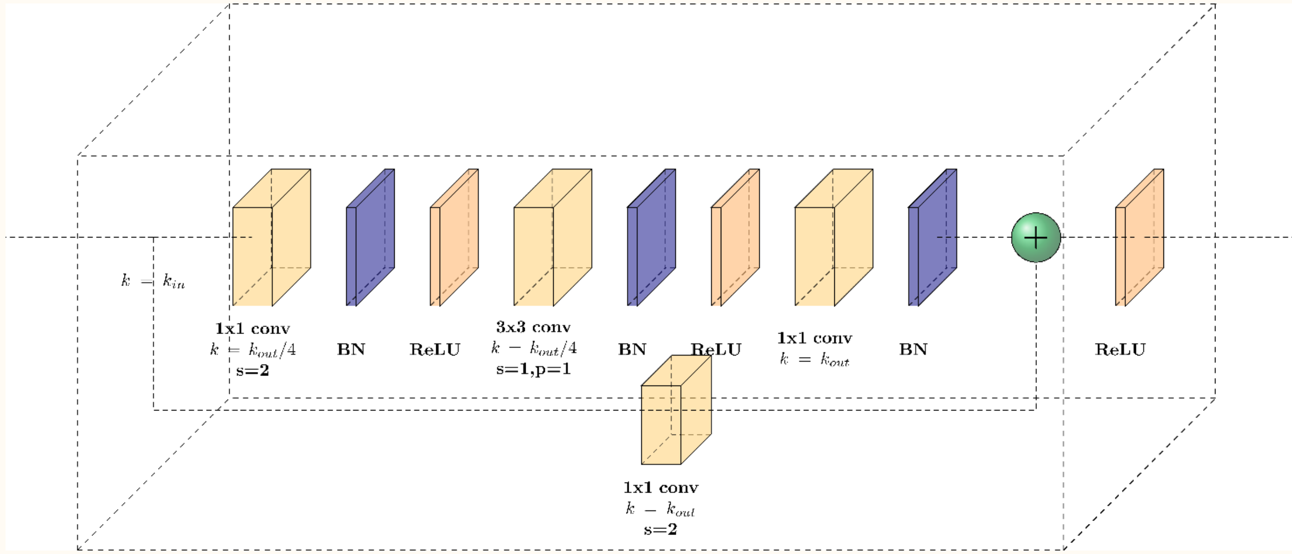
downsampling residual block v1.5



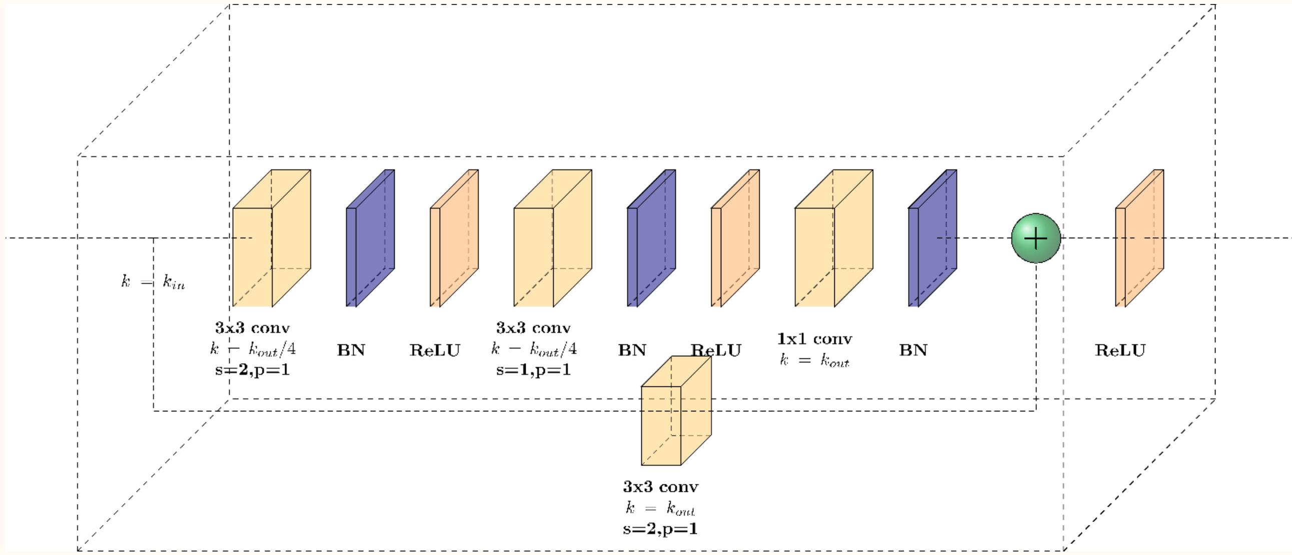
https://ngc.nvidia.com/catalog/resources/nvidia:resnet_50_v1_5_for_pytorch

ResNet v1.5

The fix is more important for the deeper downsampling residual blocks.



downsampling residual block with bottleneck v1

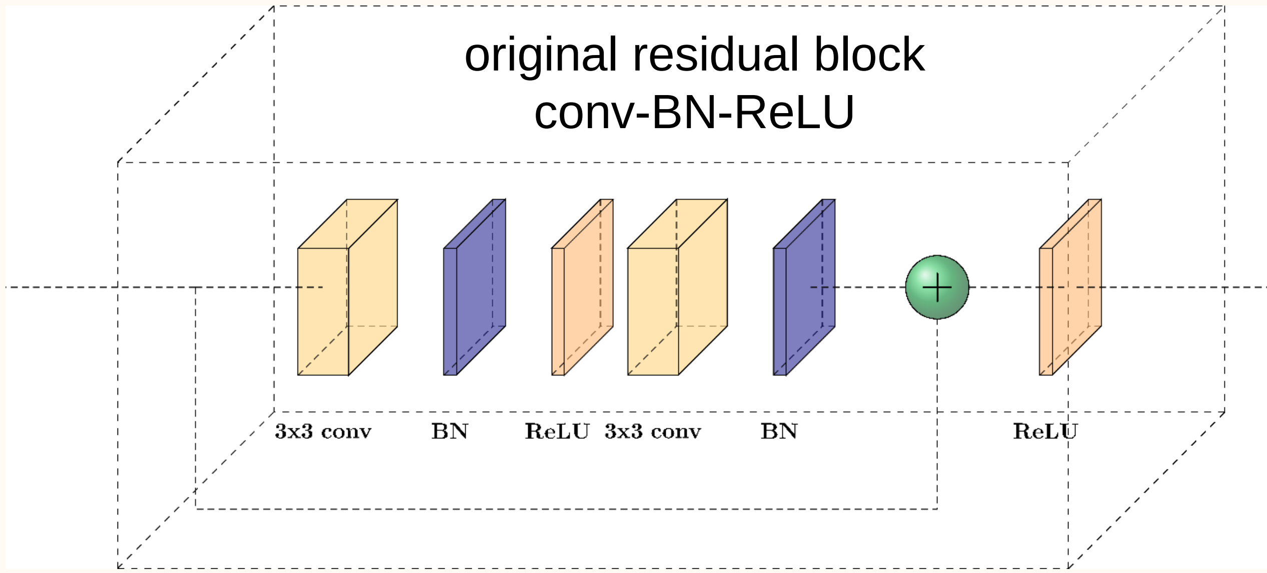


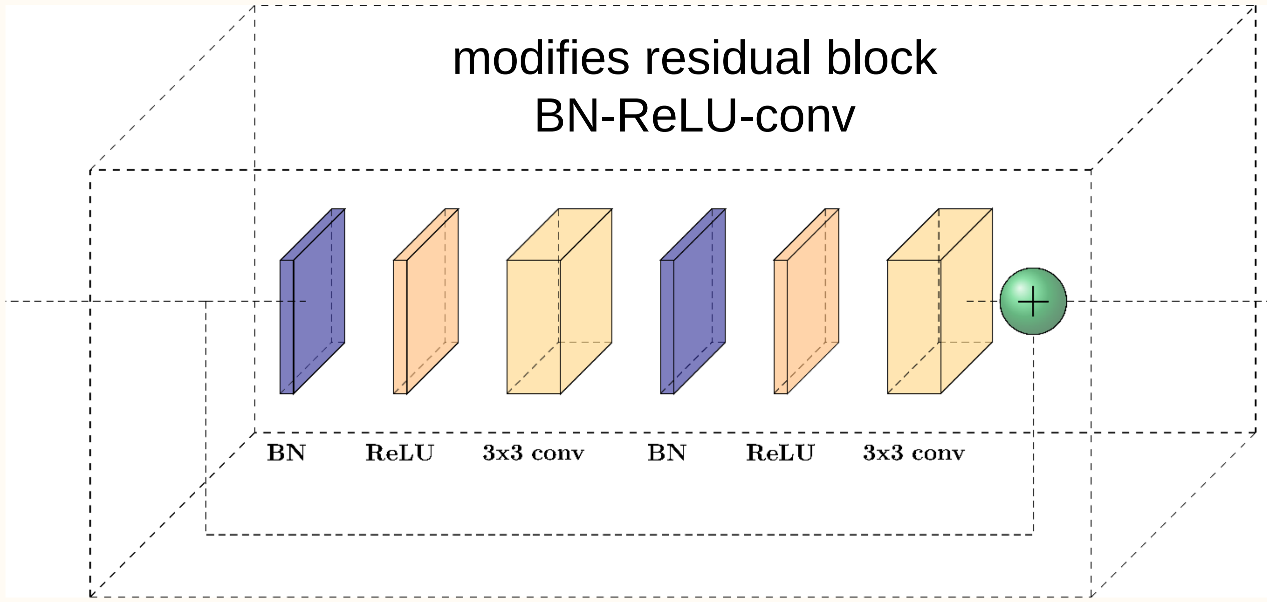
downsampling residual block with bottleneck v1.5

ResNet v2

Permutations of the ordering of conv, BN, and ReLU were tested. BN-ReLU-conv had the best performance.

Perform all operations before the residual connection so that the identity mapping can be learned.





Architectural contribution: ResNet

Introduced residual connections as a key architectural component.

Demonstrated that extremely deep neural networks can be trained with residual connections and BN. ResNet152 concluded the progression of depth. ImageNet challenge winners:

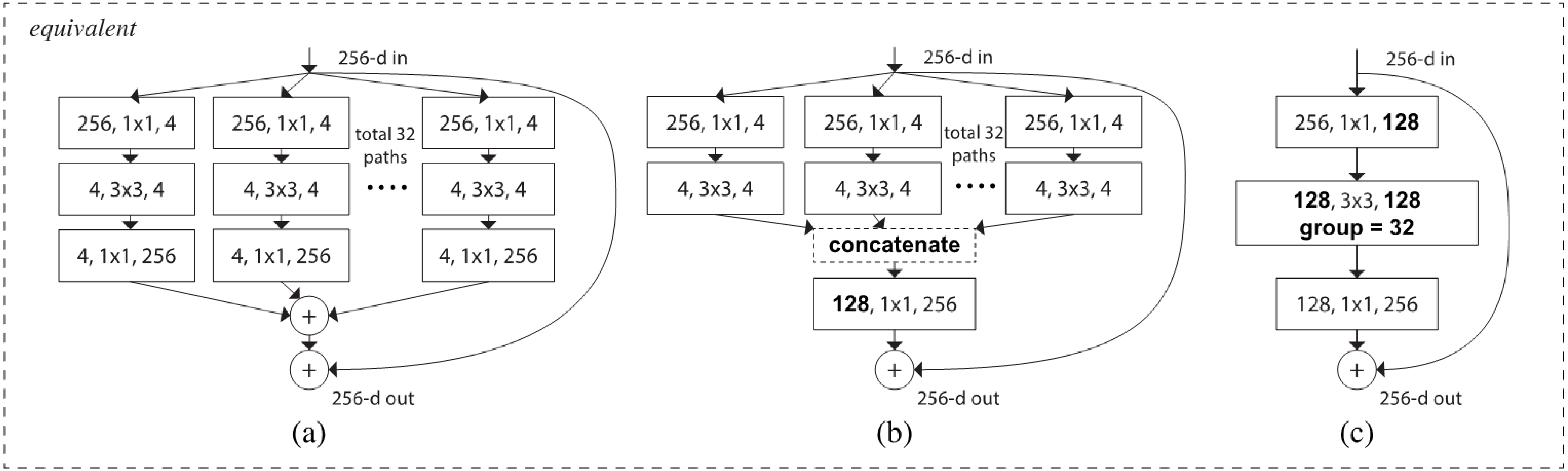
- 2012. AlexNet with 8 layers.
- 2013. ZFNet with 8 layers.
- 2014. GoogLeNet with 22 layers.
- 2015. ResNet152 with 152 layers.
- 2016. Shao et al.* with 152 layers.
- 2017. SENet with 152 layers.

Residual connections and BN are very common throughout all of deep learning.

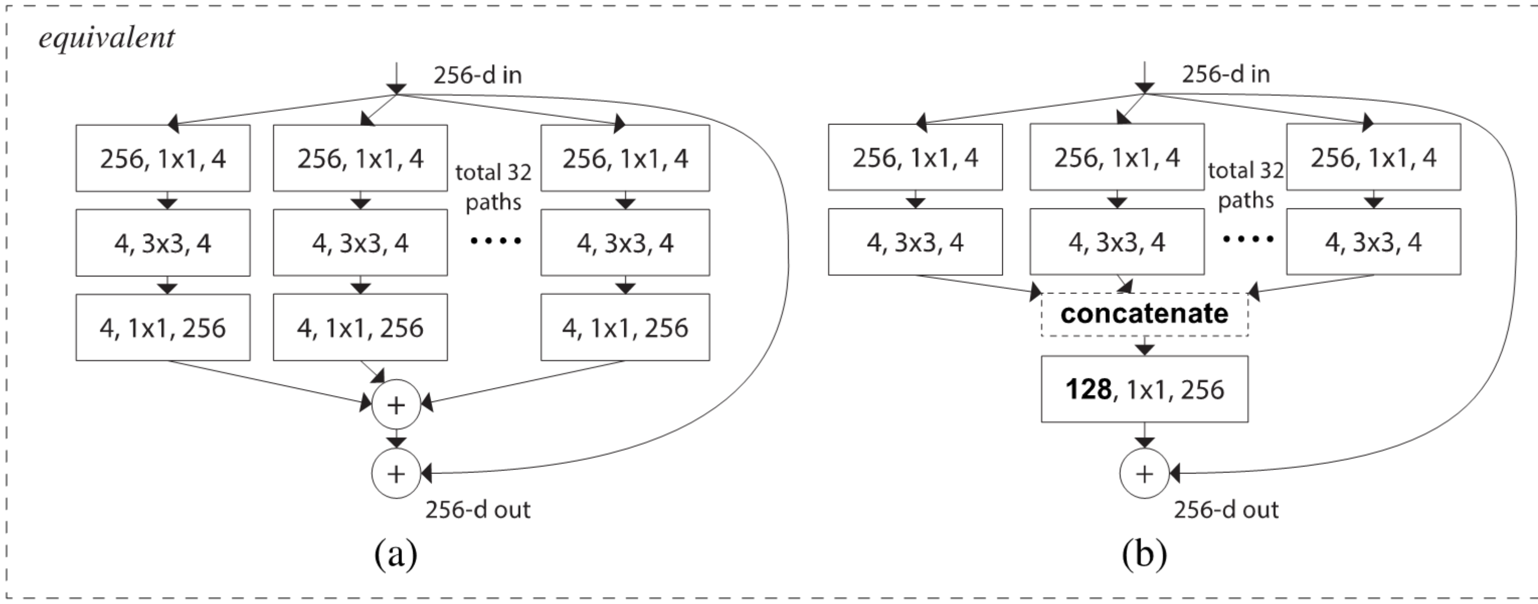
*J. Shao, X. Zhang, Z. Ding, Y. Zhao, Y. Chen, J. Zhou, W. Wang, L. Mei, and C. Hu, *Trimps-Soushen*, 2016. An ensemble model without a novel architectural component. No paper or report was written. Video presentation: https://youtu.be/NaoVOOhVC3w

ResNext

2016 ImageNet challenge 2nd place. Introduced *cardinality* as another network parameter, in addition to width (number of channels) and depth. Cardinality is the number of independent paths in the *split-transform-merge* structure.



ResNext



Blocks (a) and (b) almost equivalent due to the by the following observation.

$$256\left(\overbrace{\begin{bmatrix} A_1 \end{bmatrix}}^{4}\right.\begin{bmatrix} x_1 \end{bmatrix}\}4 + 256\left(\overbrace{\begin{bmatrix} A_2 \end{bmatrix}}^{4}\right.\begin{bmatrix} x_2 \end{bmatrix}\}4 \overset{32\text{ times}}{+\cdots+} 256\left(\overbrace{\begin{bmatrix} A_{32} \end{bmatrix}}^{4}\right.\begin{bmatrix} x_{32} \end{bmatrix}\}4$$

$$= 256\left(\overbrace{\begin{bmatrix} A_1 & A_2 & \overset{32\text{ times}}{\cdots} & A_{32} \end{bmatrix}}^{4\times 32=128}\right.\left.\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{32} \end{bmatrix}\right)128$$

Difference: Block (a) has 32 bias terms which are added to serve the role of the single bias term of block (b).

Ensemble learning

Let (X, Y) be a data-label pair. Let $m_1, \dots, m_K$ be models estimating the Y given X.

An *ensemble* is a model

$$M = \theta_1 m_1 + \cdots + \theta_K m_K$$

where $\theta_1, \dots, \theta_K \in \mathbb{R}$. Often $\theta_1 + \cdots + \theta_K = 1$ and $\theta_i \geq 0$ for $i = 1, \dots, K$. (So M is often a nonnegative weighted average $m_1, \dots, m_K$.)

If $\theta_1, \dots, \theta_K$ is chosen well, then

$$\mathbb{E}_{(X,Y)}[\|M(X) - Y\|^2] \leq \min_{i=1,\dots,K} \mathbb{E}_{(X,Y)}[\|m_i(X) - Y\|^2]$$

(The ensemble can be worse if $\theta_1, \dots, \theta_K$ is chosen poorly.)

2016 ImageNet Challenge ensemble

Trimps–Soushen* won the 2016 ImageNet Challenge with an ensemble of

- Inception-v3[1]
- Inception-v4[2]
- Inception-Resnet-v2[2]
- ResNet-200[3]
- WRN-68-3[4]

*J. Shao, X. Zhang, Z. Ding, Y. Zhao, Y. Chen, J. Zhou, W. Wang, L. Mei, and C. Hu, Trimps-Soushen, 2016.
[1]C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, Rethinking the inception architecture for computer vision, *CVPR*, 2016.
[2]C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, Inception-v4, Inception-ResNet and the impact of residual connections on learning, *AAAI*, 2017.
[3]K. He, X. Zhang, S. Ren, and J. Sun, Identity mappings in deep residual networks, *ECCV*, 2016.
[4]S. Zagoruyko and N. Komodakis, Wide residual networks, *BMVC*, 2016.

Dropout ensemble interpretation

Let m be a model with dropout applied to K neurons. The there are 2^K possible configurations, which we label $m_1, \ldots, m_{2^K}$. These models share weights.

Dropout can be viewed as randomly selecting one of these models and updating it with an iteration of SGD.

Turning off dropout at test time can be interpreted and making predictions with an ensemble of these 2^K, since each neuron is scaled so that the neuron value has the same expectation as when dropout is applied.

However, this is not a very precise connection, and I am unsure as to how much to trust it.

K. Hara, D. Saitoh, and H. Shouno, Analysis of dropout learning regarded as ensemble learning, *ICANN*, 2016.

Test-time data augmentation

Test-time data augmentation is an ensemble technique to improve the prediction.
(This is not a regularization or data augmentation technique)

Given a single model M and input X, make predictions with

$$\frac{1}{K}\sum_{i=1}^{K} M\big(T_i(X)\big)$$

where $T_1, \dots, T_K$ are random data augmentations.

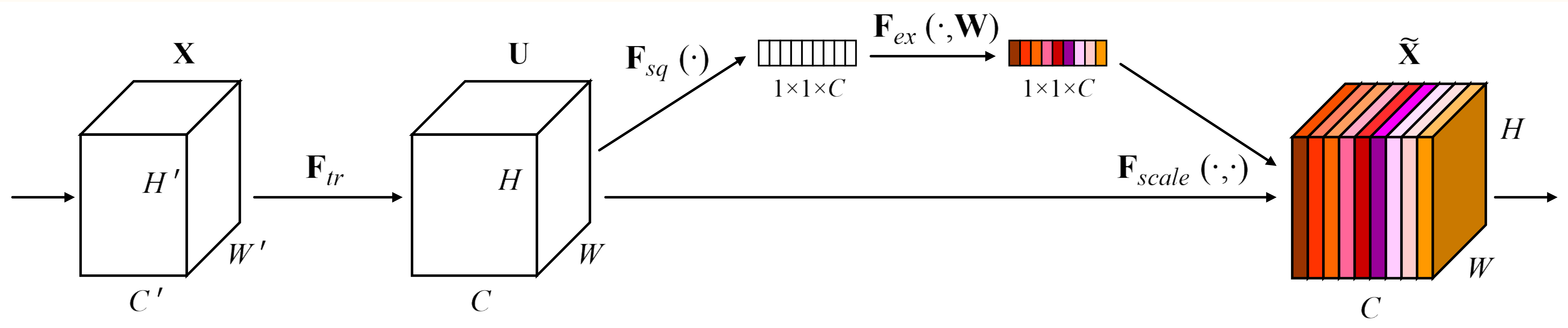
The original AlexNet paper uses test-time data augmentation with random crops and horizontal reflections: “At test time, the network makes a prediction by extracting five … patches … as well as their horizontal reflections …, and averaging the predictions made by the network’s softmax layer on the ten patches.” Most ImageNet classifiers use similar tricks.

A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, *NeurIPS*, 2012.

SENet

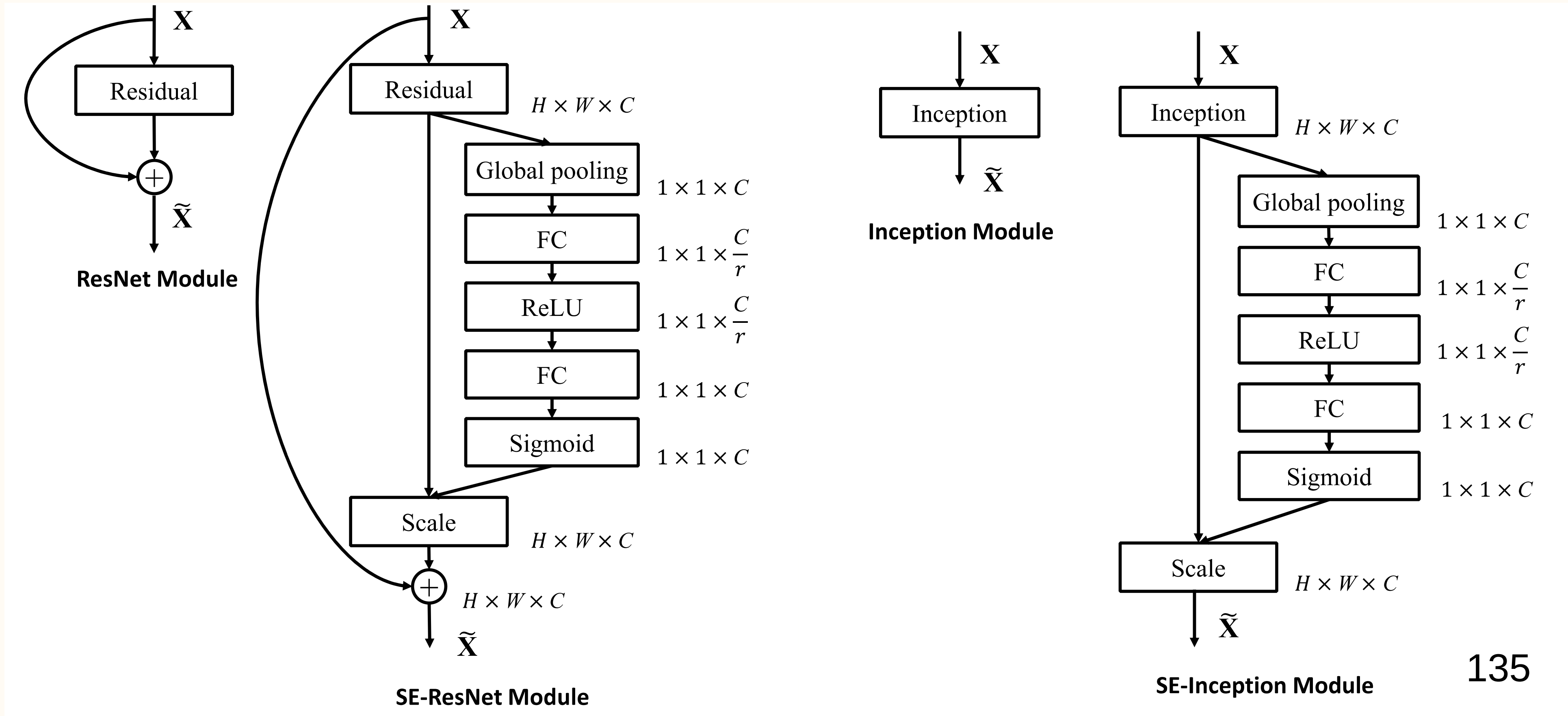
2017 ImageNet challenge 1st place. Introduced the *squeeze-and-excitation* mechanism, which is referred to *attention* in more modern papers.

Attention multiplicatively reweighs channels.



Squeeze-and-excitation

Squeeze is a global average pool. Excitation is a bottleneck structure with 1x1 convolutions and outputs weights in $(0{,}1)$ by passing through sigmoid. Finally, scale each channel.



Conclusion

We followed the ImageNet challenge from 2012 to 2017 and learned the foundations of the design and training of deep neural networks.

With the advent of deep learning, research in computer vision shifted from “feature engineering” to “network engineering”. Loosely speaking, the transition was from what to learn to learn to how to learn.

A natural progression may be to continue studying the more recent neural network architectures, beyond the 2017 SENet. However, we will stop here to move on to learning about other machine learning tasks.