Convolutional Neural Networks 1

Data Science in Electron Microscopy

Philipp Pelz

2024

https://github.com/ECLIPSE-Lab/SS24_DataScienceForEM

Intro

- small tour of modern CNN architectures.
- idea of deep neural networks is quite simple (stack together a bunch of layers), performance can vary wildly across architectures and hyperparameter choices.
- The neural networks described in this chapter are the product of intuition, a few mathematical insights, and a lot of trial and error.
- batch normalization and residual connections described in this chapter have offered two popular ideas for training and designing deep models, both of which have since also been applied to architectures beyond computer vision.
- AlexNet (Krizhevsky et al., 2012), the first large-scale network deployed to beat conventional computer vision methods on a large-scale vision challenge;
- the VGG network (Simonyan and Zisserman, 2014), which makes use of a number of repeating blocks of elements;
- the network in network (NiN) that convolves whole neural networks patch-wise over inputs (Lin et al., 2013);
- DenseNet (Huang et al., 2017) for a generalization of the residual architecture.

Deep Convolutional Neural Networks (AlexNet)

- Pre-CNNs classical pipelines looked more like this:

1. Obtain an interesting dataset. In the early days, these datasets required expensive sensors. For instance, the Apple QuickTake 100 of 1994 sported a whopping 0.3 Megapixel (VGA) resolution, capable of storing up to 8 images, all for the price of $1,000.
2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the serendipitous discoveries of lucky graduate students.
3. Feed the data through a standard set of feature extractors such as the SIFT (scale-invariant feature transform) :cite:`Lowe.2004`, the SURF (speeded up robust features) :cite:`Bay.Tuytelaars.Van-Gool.2006`, or any number of other hand-tuned pipelines. OpenCV still provides SIFT extractors to this day!
4. Dump the resulting representations into your favorite classifier, likely a linear model or kernel method, to train a classifier.

```
from d2l import torch as d2l
import torch
from torch import nn
```

Representation Learning

- Another way to cast the state of affairs is that the most important part of the pipeline was the representation.
- And up until 2012 the representation was calculated mostly mechanically. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper.
- SIFT :cite:`Lowe.2004`, SURF :cite:`Bay.Tuytelaars.Van-Gool.2006`, HOG (histograms of oriented gradient) :cite:`Dalal.Triggs.2005`, bags of visual words :cite:`Sivic.Zisserman.003`, and similar feature extractors ruled the roost.
- Another group of researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber, had different plans.
- They believed that features themselves ought to be learned. Moreover, they believed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters.

- In the case of an image, the lowest layers might come to detect edges, colors, and textures, in analogy to how the visual system in animals processes its input. In particular, the automatic design of visual features such as those obtained by sparse coding :cite:olshausen1996emergence remained an open challenge until the advent of modern CNNs.
- It was not until :citet:dean2012large,le2013building that the idea of generating features from image data automatically gained significant traction.
- The first modern CNN :cite:Krizhevsky.Sutskever.Hinton.2012, named *AlexNet* after one of its inventors, Alex Krizhevsky, is largely an evolutionary improvement over LeNet. It achieved excellent performance in the 2012 ImageNet challenge.

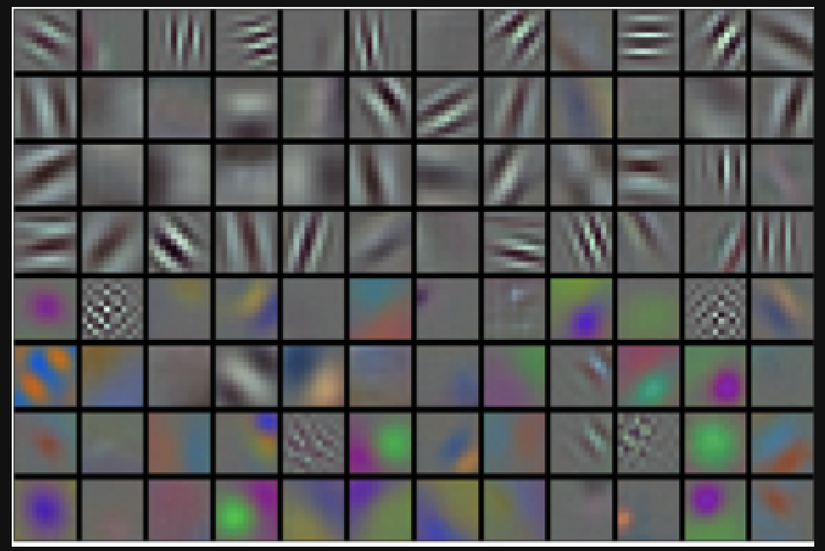
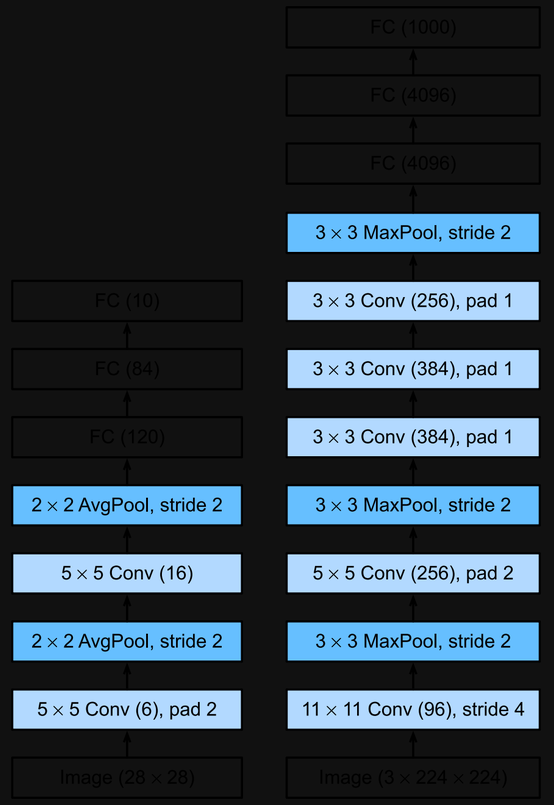


Image filters learned by the first layer of AlexNet. Reproduction courtesy of :citet:Krizhevsky.Sutskever.Hinton.2012.

AlexNet

- AlexNet network showed, for the first time, that **features obtained by learning can transcend manually-designed features*
- Note that we provide a slightly streamlined version of AlexNet removing some of the design quirks that were needed in 2012 to make the model fit on two small GPUs.



From LeNet (left) to AlexNet (right).

Architecture

- AlexNet's first layer, the convolution window shape is 11×11.
- Since the images in ImageNet are eight times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels with more visual detail.
- Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×5, followed by 3×3.
- In addition, after the first, second, and fifth convolutional layers, the network adds max-pooling layers with a window shape of 3×3 and a stride of 2.
- Moreover, AlexNet has ten times more convolution channels than LeNet.
- After the last convolutional layer, there are two huge fully connected layerswith 4096 outputs.

Activation Functions

- changed the sigmoid activation function –> ReLU activation function.
- ReLU activation function makes model training easier when using different parameter initialization methods. gradient of the ReLU activation function in the positive interval is always 1 (:numref:`subsec_activation-functions`).
- if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

Capacity Control and Preprocessing

- AlexNet controls the model complexity of the fully connected layer by dropout (:numref:`sec_dropout`), while LeNet only uses weight decay.
- the training loop of AlexNet added image augmentation, such as flipping, clipping, and color changes.
- makes the model more robust and the larger sample size effectively reduces overfitting.

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

- **construct a single-channel data example** with both height and width of 224 (**to observe the output shape of each layer**). It matches the AlexNet architecture in `fig_alexnet`.

```
1 AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d output shape:     torch.Size([1, 96, 54, 54])
ReLU output shape:   torch.Size([1, 96, 54, 54])
MaxPool2d output shape:  torch.Size([1, 96, 26, 26])
Conv2d output shape:     torch.Size([1, 256, 26, 26])
ReLU output shape:   torch.Size([1, 256, 26, 26])
MaxPool2d output shape:  torch.Size([1, 256, 12, 12])
Conv2d output shape:     torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:     torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:     torch.Size([1, 256, 12, 12])
ReLU output shape:   torch.Size([1, 256, 12, 12])
MaxPool2d output shape:  torch.Size([1, 256, 5, 5])
Flatten output shape:    torch.Size([1, 6400])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
```

Training

- AlexNet was trained on ImageNet in :citet:`Krizhevsky.Sutskever.Hinton.2012`, we use Fashion-MNIST here since training an ImageNet model to convergence could take hours or days even on a modern GPU.
- One of the problems with applying AlexNet directly on [**Fashion-MNIST**] is that its (**images have lower resolution**) (28×28 pixels) (**than ImageNet images.**)
- To make things work, (**we upsample them to** 224×224). This is generally not a smart practice, as it simply increases the computational complexity without adding information. Nonetheless, we do it here to be faithful to the AlexNet architecture.
- We perform this resizing with the `resize` argument in the `d2l.FashionMNIST` constructor.

- Now, we can [**start training AlexNet.**] Compared to LeNet in :numref:`sec_lenet`, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher image resolution, and the more costly convolutions.

```
model = AlexNet(lr=0.01)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
# trainer.fit(model, data)
```

Discussion

- AlexNet's structure bears a striking resemblance to LeNet, with a number of critical improvements, both for accuracy (dropout) and for ease of training (ReLU).
- What is equally striking is the amount of progress that has been made in terms of deep learning tooling.
- Reviewing the architecture, we see that AlexNet has an Achilles heel when it comes to efficiency: the last two hidden layers require matrices of size 6400×4096 and 4096×4096, respectively.
- This is one of the reasons why AlexNet has been surpassed by much more effective architectures that we will cover in the following sections.
- Nonetheless, it is a key step from shallow to deep networks that are used nowadays.
- even though the number of parameters by far exceeds the amount of training data in our experiments (the last two layers have more than 40 million parameters, trained on a datasets of 60 thousand images), there is hardly any overfitting: training and validation loss are virtually identical throughout training.
- due to the improved regularization, such as Dropout, inherent in modern deep network designs.

Exercises

1. Following up on the discussion above, analyze the computational properties of AlexNet.
    1. Compute the memory footprint for convolutions and fully connected layers, respectively. Which one dominates?
    2. Calculate the computational cost for the convolutions and the fully connected layers.
    3. How does the memory (read and write bandwidth, latency, size) affect computation? Is there any difference in its effects for training and inference?
2. You are a chip designer and need to trade off computation and memory bandwidth. For example, a faster chip requires more power and possibly a larger chip area. More memory bandwidth requires more pins and control logic, thus also more area. How do you optimize?
3. Why do engineers no longer report performance benchmarks on AlexNet?
4. Try increasing the number of epochs when training AlexNet. Compared with LeNet, how do the results differ? Why?

1. AlexNet may be too complex for the Fashion-MNIST dataset, in particular due to the low resolution of the initial images.
    1. Try simplifying the model to make the training faster, while ensuring that the accuracy does not drop significantly.
    2. Design a better model that works directly on 28×28 images.
2. Modify the batch size, and observe the changes in throughput (images/s), accuracy, and GPU memory.
3. Apply dropout and ReLU to LeNet-5. Does it improve? Can you improve things further by preprocessing to take advantage of the invariances inherent in the images?
4. Can you make AlexNet overfit? Which feature do you need to remove or change to break training?

Networks Using Blocks (VGG)

- AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks.
- In the following sections, we will introduce several heuristic concepts commonly used to design deep networks.
- Progress in this field mirrors that of VLSI (very large scale integration) in chip design where engineers moved from placing transistors to logical elements to logic blocks :cite:`Mead.1980`.
- design of neural network architectures has grown progressively more abstract: individual neurons to whole layers, and now to blocks, repeating patterns of layers.

- A decade later, this has now progressed to researchers using entire trained models to repurpose them for different, albeit related, tasks. Such large pretrained models are typically called *foundation models* :cite:bommasani2021opportunities.
- Back to network design. The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, in their eponymously-named *VGG* network :cite:Simonyan.Zisserman.2014.
- It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

```
from d2l import torch as d2l
import torch
from torch import nn
```

VGG Blocks

- The basic building block of CNNs is a sequence of the following:
- i. a convolutional layer with padding to maintain the resolution,
- ii. a nonlinearity such as a ReLU,
- iii. a pooling layer such as max-pooling to reduce the resolution.
- One of the problems with this approach: spatial resolution decreases quite rapidly.
- imposes a hard limit of $\log_2 d$ convolutional layers on the network before all dimensions (d) are used up. For instance, in the case of ImageNet, it would be impossible to have more than 8 convolutional layers in this way.
- key idea of :citet:`Simonyan.Zisserman.2014` was to use *multiple* convolutions in between downsampling via max-pooling in the form of a block.
- primarily interested in whether deep or wide networks perform better.
- For instance, the successive application of two 3×3 convolutions touches the same pixels as a single 5×5 convolution does.

- At the same time, the latter uses approximately as many parameters ($25 \cdot c^2$) as three 3×3 convolutions do ($3 \cdot 9 \cdot c^2$).
- In a rather detailed analysis they showed that deep and narrow networks significantly outperform their shallow counterparts. This set deep learning on a quest for ever deeper networks with over 100 layers for typical applications.
- Stacking 3×3 convolutions has become a gold standard in later deep networks (a design decision only to be revisited recently by :citet:`liu2022convnet`). Consequently, fast implementations for small convolutions have become a staple on GPUs :cite:`lavin2016fast`.
- Back to VGG: a VGG block consists of a *sequence* of convolutions with 3×3 kernels with padding of 1 (keeping height and width) followed by a 2×2 max-pooling layer with stride of 2 (halving height and width after each block).

- In the code below, we define a function called `vgg_block` to implement one VGG block.
- The function below takes two arguments, corresponding to the number of convolutional layers `num_convs` and the number of output channels `num_channels`.

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

VGG Network

- VGG Network can be partitioned into two parts:
  - first consisting mostly of convolutional and pooling layers
  - second consisting of fully connected layers that are identical to those in AlexNet.
- key difference : convolutional layers are grouped in nonlinear transformations that leave the dimensonality unchanged, followed by a resolution-reduction step, as depicted in :numref:`fig_vgg`.

fig_vgg

- convolutional part of the network connects several VGG blocks from :numref:`fig_vgg` (also defined in the `vgg_block` function) in succession. This grouping of convolutions is a pattern that has remained almost unchanged over the past decade, although the specific choice of operations has undergone considerable modifications.
- The variable `arch` consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments required to call the `vgg_block` function.
- As such, VGG defines a *family* of networks rather than just a specific manifestation. To build a specific network we simply iterate over `arch` to compose the blocks.

```python
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

- original VGG network had 5 convolutional blocks, among which the first two have one convolutional layer each and the latter three contain two convolutional layers each.
- The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512.

- Since this network uses 8 convolutional layers and 3 fully connected layers, it is often called VGG-11.

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:    torch.Size([1, 25088])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 10])
```

- halve height and width at each block, finally reaching a height and width of 7 before flattening the representations for processing by the fully connected part of the network.

Training

- **VGG-11 is computationally more demanding than AlexNet we construct a network with a smaller number of channels.**
- more than sufficient for training on Fashion-MNIST.
- **model training** process is similar to that of AlexNet in `sec_alexnet`.
- observe the close match between validation and training loss, suggesting only a small amount of overfitting.

```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
# trainer.fit(model, data)
```

Summary

- might argue that VGG is the first truly modern convolutional neural network.
- VGG introduced key properties such as blocks of multiple convolutions and a preference for deep and narrow networks.
- first network that is actually an entire family of similarly parametrized models, giving the practitioner ample trade-off between complexity and speed.

Exercises

1. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory.
   1. Compare the number of parameters needed for AlexNet and VGG.
   2. Compare the number of floating point operations used in the convolutional layers and in the fully connected layers.
   3. How could you reduce the computational cost created by the fully connected layers?
2. When displaying the dimensions associated with the various layers of the network, we only see the information associated with 8 blocks (plus some auxiliary transforms), even though the network has 11 layers. Where did the remaining 3 layers go?
3. Use Table 1 in the VGG paper :cite:`Simonyan.Zisserman.2014` to construct other common models, such as VGG-16 or VGG-19.
4. Upsampling the resolution in Fashion-MNIST by a factor of 8 from 28×28 to 224×224 dimensions is very wasteful. Try modifying the network architecture and resolution conversion, e.g., to 56 or to 84 dimensions for its input instead. Can you do so without reducing the accuracy of the network? Consider the VGG paper :cite:`Simonyan.Zisserman.2014` for ideas on adding more nonlinearities prior to downsampling.

Network in Network (NiN)

- LeNet, AlexNet, and VGG all share a common design pattern: extract features exploiting *spatial* structure via a sequence of convolutions and pooling layers and post-process the representations via fully connected layers.
- design poses two major challenges.
- the fully connected layers at the end of the architecture consume tremendous numbers of parameters.
- For instance, even a simple model such as VGG-11 requires a monstrous 25088×4096 matrix, occupying almost 400MB of RAM in single precision (FP32).
- significant impediment to computation, in particular on mobile and embedded devices.
- After all, even high-end mobile phones sport no more than 8GB of RAM. At the time VGG was invented, this was an order of magnitude less (the iPhone 4S had 512MB).

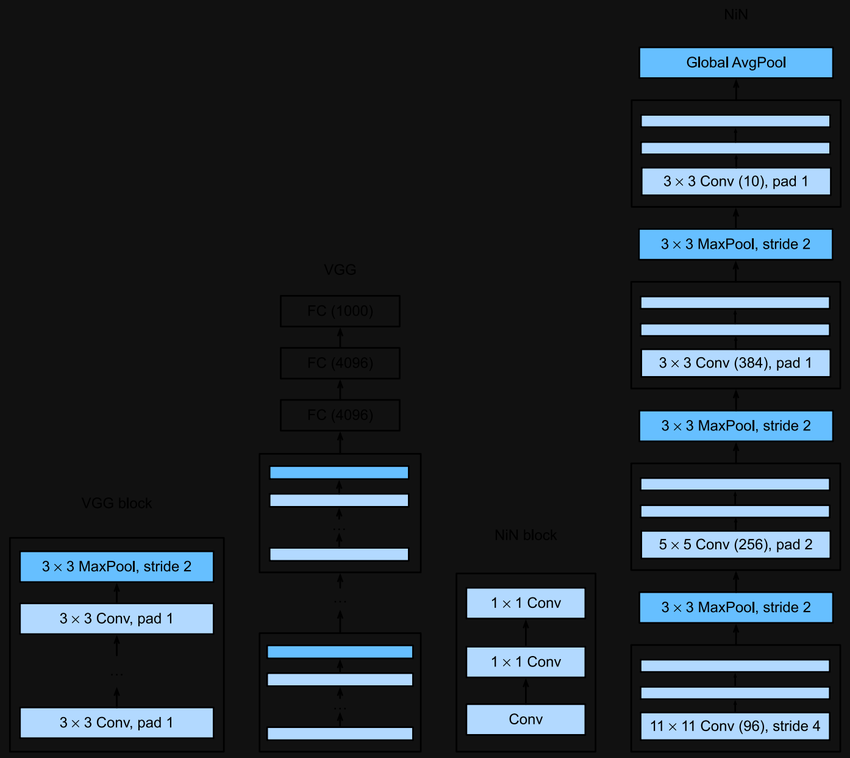
- As such, it would have been difficult to justify spending the majority of memory on an image classifier.
- equally impossible to add fully connected layers earlier in the network to increase the degree of nonlinearity: doing so would destroy the spatial structure and require potentially even more memory.
- *network in network* (*NiN*) blocks :cite:`Lin.Chen.Yan.2013` offer an alternative, capable of solving both problems in one simple strategy.
- proposed based on a very simple insight:
- i. use 1×1 convolutions to add local nonlinearities across the channel activations
- ii. use global average pooling to integrate across all locations in the last representation layer.
- Note that global average pooling would not be effective, were it not for the added nonlinearities. Let’s dive into this in detail.

```
from d2l import torch as d2l
import torch
from torch import nn
```

NiN Blocks

- Recall :numref:`subsec_1x1`. In it we discussed that the inputs and outputs of convolutional layers consist of four-dimensional tensors with axes corresponding to the example, channel, height, and width.
- Also recall that the inputs and outputs of fully connected layers are typically two-dimensional tensors corresponding to the example and feature.
- The idea behind NiN is to apply a fully connected layer at each pixel location (for each height and width).
- The resulting 1×1 convolution can be thought as a fully connected layer acting independently on each pixel location.
- :numref:`fig_nin` illustrates the main structural differences between VGG and NiN, and their blocks.

- Note both the difference in the NiN blocks (the initial convolution is followed by 1×1 convolutions, whereas VGG retains 3×3 convolutions) and in the end where we no longer require a giant fully connected layer.



:width:600px :label:fig_nin

```python
def nin_block(out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.LazyConv2d(out_channels, kernel_size, strides, padding), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU())
```

NiN Model

- NiN uses the same initial convolution sizes as AlexNet (it was proposed shortly thereafter).
- The kernel sizes are 11×11, 5×5, and 3×3, respectively, and the numbers of output channels match those of AlexNet. Each NiN block is followed by a max-pooling layer with a stride of 2 and a window shape of 3×3.
- The second significant difference between NiN and both AlexNet and VGG is that NiN avoids fully connected layers altogether.
- Instead, NiN uses a NiN block with a number of output channels equal to the number of label classes, followed by a *global* average pooling layer, yielding a vector of logits.
- This design significantly reduces the number of required model parameters, albeit at the expense of a potential increase in training time.

```python
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nin_block(96, kernel_size=11, strides=4, padding=0),
            nn.MaxPool2d(3, stride=2),
            nin_block(256, kernel_size=5, strides=1, padding=2),
            nn.MaxPool2d(3, stride=2),
            nin_block(384, kernel_size=3, strides=1, padding=1),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout(0.5),
            nin_block(num_classes, kernel_size=3, strides=1, padding=1),
            nn.AdaptiveAvgPool2d((1, 1)),
```

- create a data example to see **the output shape of each block**

```
1 NiN().layer_summary((1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 96, 54, 54])
MaxPool2d output shape:  torch.Size([1, 96, 26, 26])
Sequential output shape:     torch.Size([1, 256, 26, 26])
MaxPool2d output shape:  torch.Size([1, 256, 12, 12])
Sequential output shape:     torch.Size([1, 384, 12, 12])
MaxPool2d output shape:  torch.Size([1, 384, 5, 5])
Dropout output shape:    torch.Size([1, 384, 5, 5])
Sequential output shape:     torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d output shape:  torch.Size([1, 10, 1, 1])
Flatten output shape:    torch.Size([1, 10])
```

Training

- use Fashion-MNIST to train the model

```
model = NiN(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
# trainer.fit(model, data)
```

Summary

- NiN has dramatically fewer parameters than AlexNet and VGG.
- stems primarily from the fact that it needs no giant fully connected layers.
- Instead, uses global average pooling to aggregate across all image locations after the last stage of the network body.
- obviates the need for expensive (learned) reduction operations and replaces them by a simple average.
- averaging operation does not harm accuracy.
- averaging across a low-resolution representation (with many channels) also adds to the amount of translation invariance that the network can handle.
- choosing fewer convolutions with wide kernels and replacing them by 1×1 convolutions aids the quest for fewer parameters further.
- affords for a significant amount of nonlinearity across channels within any given location.
- Both 1×1 convolutions and global average pooling significantly influenced subsequent CNN designs.

Exercises

1. Why are there two 1×1 convolutional layers per NiN block? Increase their number to three. Reduce their number to one. What changes?
2. What changes if you replace the 1×1 convolutions by 3×3 convolutions?
3. What happens if you replace the global average pooling by a fully connected layer (speed, accuracy, number of parameters)?
4. Calculate the resource usage for NiN.
    1. What is the number of parameters?
    2. What is the amount of computation?
    3. What is the amount of memory needed during training?
    4. What is the amount of memory needed during prediction?
5. What are possible problems with reducing the $384 \times 5 \times 5$ representation to a $10 \times 5 \times 5$ representation in one step?
6. Use the structural design decisions in VGG that led to VGG-11, VGG-16, and VGG-19 to design a family of NiN-like networks.

Batch Normalization

- Training deep neural networks is difficult.
- Getting them to converge in a reasonable amount of time can be tricky.
- In this section, we describe *batch normalization*, a popular and effective technique that consistently accelerates the convergence of deep networks :cite:`Ioffe.Szegedy.2015`.
- Together with residual blocks—covered later in :numref:`sec_resnet`—batch normalization has made it possible for practitioners to routinely train networks with over 100 layers.
- A secondary (serendipitous) benefit of batch normalization lies in its inherent regularization.

```
from d2l import torch as d2l
import torch
from torch import nn
```

Training Deep Networks

- When working with data, we often preprocess before training.
- Choices regarding data preprocessing often make an enormous difference in the final results.
- Recall our application of MLPs to predicting house prices (:numref:`sec_kaggle_house`).
- first step when working with real data: standardize our input features to have zero mean $\boldsymbol{\mu} = 0$ and unit variance $\boldsymbol{\Sigma} = \mathbf{1}$ cross multiple observations :cite:`friedman1987exploratory`.
- one frequently rescales it such that the diagonal is unity, i.e., \Sigma_ ii} = 1.
- another strategy is to rescale vectors to unit length, possibly zero mean *per observation*.
- can work well, e.g., for spatial sensor data. These preprocessing techniques and many more are beneficial to keep the estimation problem well controlled.
- standardizing vectors also has the nice side-effect of constraining the function complexity of functions that act upon it.
- Intuitively, this standardization plays nicely with our optimizers since it puts the parameters *a priori* at a similar scale.

- natural to ask whether a corresponding normalization step *inside* a deep network might not be beneficial
- useful way of understanding it and its cousin, layer normalization :cite:Ba.Kiros.Hinton.2016 within a unified framework.
- Second, for a typical MLP or CNN, as we train, the variables in intermediate layers (e.g., affine transformation outputs in MLP) may take values with widely varying magnitudes:
  - both along the layers from input to output, across units in the same layer, and over time due to our updates to the model parameters.
  - inventors of batch normalization postulated informally that this drift in the distribution of such variables could hamper the convergence of the network.
- Intuitively, we might conjecture that if one layer has variable activations that are 100 times that of another layer, this might necessitate compensatory adjustments in the learning rates.
- Adaptive solvers such as AdaGrad :cite:Duchi.Hazan.Singer.2011, Adam :cite:Kingma.Ba.2014, Yogi :cite:Zaheer.Reddi.Sachan.ea.2018, or Distributed Shampoo :cite:anil2020scalable aim to address this from the viewpoint of optimization, e.g., by adding aspects of second-order methods.

- alternative is to prevent the problem from occurring, simply by adaptive normalization.
- deeper networks are complex –> overfitting.
- regularization becomes more critical. A common technique for regularization is noise injection.
- has been known for a long time, e.g., with regard to noise injection for the inputs :cite:`Bishop.1995`. It also forms the basis of dropout in :numref:`sec_dropout`.
- batch normalization conveys all three benefits: preprocessing, numerical stability, and regularization.
- Batch normalization is applied to individual layers, or optionally, to all of them:
- In each training iteration, we first normalize the inputs (of batch normalization) by subtracting their mean and dividing by their standard deviation, where both are estimated based on the statistics of the current minibatch.
- Next, we apply a scale coefficient and an offset to recover the lost degrees of freedom. It is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name.

- Note that if we tried to apply batch normalization with minibatches of size 1, we would not be able to learn anything.
- That is because after subtracting the means, each hidden unit would take value 0.
- As you might guess, since we are devoting a whole section to batch normalization, with large enough minibatches, the approach proves effective and stable.
- One takeaway here is that when applying batch normalization, the choice of batch size is even more significant than without batch normalization, or at least, suitable calibration is needed as we might adjust it.
- Denote by $\mathcal{B}$ a minibatch and let $\mathbf{x} \in \mathcal{B}$ be an input to batch normalization (BN). In this case the batch normalization is defined as follows:
- $$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}.$$

  :eqlabel:`eq_batchnorm`

- In :eqref:`eq_batchnorm`, $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ is the sample mean and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ is the sample standard deviation of the minibatch $\mathcal{B}$.
- After applying standardization, the resulting minibatch has zero mean and unit variance. The choice of unit variance (vs. some other magic number) is an arbitrary choice. We recover this degree of freedom by including an elementwise *scale parameter* $\boldsymbol{\gamma}$ and *shift parameter* $\boldsymbol{\beta}$ that have the same shape as $\mathbf{x}$. Both are parameters that need to be learned as part of model training.
- The variable magnitudes for intermediate layers cannot diverge during training since batch normalization actively centers and rescales them back to a given mean and size (via \hat \boldsymbol{\mu}}_\mathcal{B} and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$).
- Practical experience confirms that, as alluded to when discussing feature rescaling, batch normalization seems to allow for more aggressive learning rates.

- We calculate $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ in :eqref:`eq_batchnorm` as follows:

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.$$

- Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might be very small or even vanish.
- The estimates $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ counteract the scaling issue by using noisy estimates of mean and variance.
- You might think that this noisiness should be a problem. Quite to the contrary, this is actually beneficial.

- This turns out to be a recurring theme in deep learning. For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting:
- this variation appears to act as a form of regularization. :citet:`Teye.Azizpour.Smith.2018` and :citet:`Luo.Wang.Shao.ea.2018` related the properties of batch normalization to Bayesian priors and penalties, respectively.
- In particular, this sheds some light on the puzzle of why batch normalization works best for moderate minibatches sizes in the $50 \sim 100$ range.
- This particular size of minibatch seems to inject just the “right amount” of noise per layer, both in terms of scale via $\hat{\boldsymbol{\sigma}}$, and in terms of offset via $\hat{\boldsymbol{\mu}}$:
- a larger minibatch regularizes less due to the more stable estimates, whereas tiny minibatches destroy useful signal due to high variance.

- Exploring this direction further, considering alternative types of preprocessing and filtering may yet lead to other effective types of regularization.
- Fixing a trained model, you might think that we would prefer using the entire dataset to estimate the mean and variance.
- Once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside?
- During training, such exact calculation is infeasible because the intermediate variables for all data examples change every time we update our model.
- However, once the model is trained, we can calculate the means and variances of each layer's variables based on the entire dataset.
- Indeed this is standard practice for models employing batch normalization and thus batch normalization layers function differently in *training mode* (normalizing by minibatch statistics) and in *prediction mode* (normalizing by dataset statistics).
- In this form they closely resemble the behavior of dropout regularization of :numref:`sec_dropout`, where noise is only injected during training.

Batch Normalization Layers

- Batch normalization implementations for fully connected layers and convolutional layers are slightly different.
- One key difference between batch normalization and other layers is that because batch normalization operates on a full minibatch at a time, we cannot just ignore the batch dimension as we did before when introducing other layers.

Fully Connected Layers

- When applying batch normalization to fully connected layers, the original paper inserted batch normalization after the affine transformation and *before* the nonlinear activation function. Later applications experimented with inserting batch normalization right *after* activation functions :cite:`Ioffe.Szegedy.2015`.
- Denoting the input to the fully connected layer by $\mathbf{x}$, the affine transformation by $\mathbf{W}\mathbf{x} + \mathbf{b}$ (with the weight parameter $\mathbf{W}$ and the bias parameter $\mathbf{b}$), and the activation function by ϕ, we can express the computation of a batch-normalization-enabled, fully connected layer output $\mathbf{h}$ as follows:

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})).$$

- Recall that mean and variance are computed on the *same* minibatch on which the ransformation is applied.

Convolutional Layers

- Similarly, with convolutional layers, we can apply batch normalization after the convolution and before the nonlinear activation function. The key difference from batch normalization in fully connected layers is that we apply the operation on a per-channel basis *across all locations*.
- This is compatible with our assumption of translation invariance that led to convolutions: we assumed that the specific location of a pattern within an image was not critical for the purpose of understanding.
- Assume that our minibatches contain m examples and that for each channel, the output of the convolution has height p and width q.
- For convolutional layers, we carry out each batch normalization over the $m \cdot p \cdot q$ elements per output channel simultaneously.
- Thus, we collect the values over all spatial locations when computing the mean and variance and consequently apply the same mean and variance within a given channel to normalize the value at each spatial location.
- Each channel has its own scale and shift parameters, both of which are scalars.

Layer Normalization

- Note that in the context of convolutions the batch normalization is well-defined even for minibatches of size 1: after all, we have all the locations across an image to average
- consequently, mean and variance are well defined, even if it is just within a single observation.
- This consideration led :citet:`Ba.Kiros.Hinton.2016` to introduce the notion of *layer normalization*.
- It works just like a batch norm, only that it is applied to one observation at a time. Consequently both the offset and the scaling factor are scalars. Given an n-dimensional vector $\mathbf{x}$ layer norms are given by

$$\mathbf{x} \to \mathrm{LN}(\mathbf{x}) = \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}},$$

where scaling and offset are applied coefficient-wise and given by

$$\hat{\mu} \stackrel{\mathrm{def}}{=} \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } \hat{\sigma}^2 \stackrel{\mathrm{def}}{=} \frac{1}{n}\sum_{i=1}^{n} (x_i - \hat{\mu})^2 + \epsilon.$$

- As before we add a small offset $\epsilon > 0$ to prevent division by zero.
- One of the major benefits of using layer normalization is that it prevents divergence.
- After all, ignoring ϵ, the output of the layer normalization is scale independent.
- That is, we have $\mathbf{LN}(\mathbf{x}) \approx \mathbf{LN}(\alpha\mathbf{x})$ for any choice of $\alpha \neq 0$.
- This becomes an equality for $\left|\alpha\right| \to \infty$ (the approximate equality is due to the offset ϵ for the variance).

- Another advantage of the layer normalization is that it does not depend on the minibatch size.
- It is also independent of whether we are in training or test regime.
- In other words, it is simply a deterministic transformation that standardizes the activations to a given scale.
- This can be very beneficial in preventing divergence in optimization. We skip further details and recommend the interested reader to consult the original paper.

Batch Normalization During Prediction

- mentioned earlier, batch normalization typically behaves differently in training mode and prediction mode.
- First, the noise in the sample mean and the sample variance arising from estimating each on minibatches are no longer desirable once we have trained the model.
- Second, we might not have the luxury of computing per-batch normalization statistics.
- For example, we might need to apply our model to make one prediction at a time.
- Typically, after training, we use the entire dataset to compute stable estimates of the variable statistics and then fix them at prediction time.
- Consequently, batch normalization behaves differently during training and at test time. Recall that dropout also exhibits this characteristic.

Implementation from Scratch

- To see how batch normalization works in practice, we implement one from scratch below.

```python
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    ## Use is_grad_enabled to determine whether we are in training mode
    if not torch.is_grad_enabled():
        ## In prediction mode, use mean and variance obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            ## When using a fully connected layer, calculate the mean and
            ## variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            ## When using a two-dimensional convolutional layer, calculate the
            ## mean and variance on the channel dimension (axis=1). Here we
            ## need to maintain the shape of X, so that the broadcasting
            ## operation can be carried out later
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
```

- We can now [**create a proper `BatchNorm` layer.**] Our layer will maintain proper parameters for scale `gamma` and shift `beta`, both of which will be updated in the course of training.
- Additionally, our layer will maintain moving averages of the means and variances for subsequent use during model prediction.
- Putting aside the algorithmic details, note the design pattern underlying our implementation of the layer.
- Typically, we define the mathematics in a separate function, say `batch_norm`.
- We then integrate this functionality into a custom layer, whose code mostly addresses bookkeeping matters, such as moving data to the right device context, allocating and initializing any required variables, keeping track of moving averages (here for mean and variance), and so on.
- This pattern enables a clean separation of mathematics from boilerplate code.
- Also note that for the sake of convenience we did not worry about automatically inferring the input shape here, thus we need to specify the number of features throughout.
- By now all modern deep learning frameworks offer automatic detection of size and shape in the high-level batch normalization APIs (in practice we will use this instead).

```python
class BatchNorm(nn.Module):
    ## num_features: the number of outputs for a fully connected layer or the
    ## number of output channels for a convolutional layer. num_dims: 2 for a
    ## fully connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        ## The scale parameter and the shift parameter (model parameters) are
        ## initialized to 1 and 0, respectively
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        ## The variables that are not model parameters are initialized to 0 and
        ## 1
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.ones(shape)
```

- used `momentum` to govern the aggregation over past mean and variance estimates.
- This is somewhat of a misnomer as it has nothing whatsoever to do with the *momentum* term of optimization in :numref:`sec_momentum`.
- Nonetheless, it is the commonly adopted name for this term and in deference to API naming convention we use the same variable name in our code, too.

[LeNet with Batch Normalization]

- see how to apply `BatchNorm` in context, below we apply it to a traditional LeNet model (:numref:`sec_lenet`).
- Recall that batch normalization is applied after the convolutional layers or fully connected layers but before the corresponding activation functions.

```
class BNLeNetScratch(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5), BatchNorm(6, num_dims=4),
            nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), BatchNorm(16, num_dims=4),
            nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(), nn.LazyLinear(120),
            BatchNorm(120, num_dims=2), nn.Sigmoid(), nn.LazyLinear(84),
            BatchNorm(84, num_dims=2), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

- As before, we will [**train our network on the Fashion-MNIST dataset**]. This code is virtually identical to that when we first trained LeNet.

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNetScratch(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
# trainer.fit(model, data)
```

- Let's [**have a look at the scale parameter `gamma` and the shift parameter `beta`**] learned from the first batch normalization layer.

```
model.net[1].gamma.reshape((-1,)), model.net[1].beta.reshape((-1,))
```

```
(tensor([1., 1., 1., 1., 1., 1.], grad_fn=<ViewBackward0>),
 tensor([0., 0., 0., 0., 0., 0.], grad_fn=<ViewBackward0>))
```

[Concise Implementation]

- Compared with the `BatchNorm` class, which we just defined ourselves, we can use the `BatchNorm` class defined in high-level APIs from the deep learning framework directly.
- The code looks virtually identical to our implementation above, except that we no longer need to provide additional arguments for it to get the dimensions right.

```
class BNLeNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5), nn.LazyBatchNorm2d(),
            nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.LazyBatchNorm2d(),
            nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(), nn.LazyLinear(120), nn.LazyBatchNorm1d(),
            nn.Sigmoid(), nn.LazyLinear(84), nn.LazyBatchNorm1d(),
            nn.Sigmoid(), nn.LazyLinear(num_classes))
```

- Below, we [**use the same hyperparameters to train our model.**] Note that as usual, the high-level API variant runs much faster because its code has been compiled to C++ or CUDA while our custom implementation must be interpreted by Python.

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
# trainer.fit(model, data)
```

Discussion

- Intuitively, batch normalization is thought to make the optimization landscape smoother.
- However, we must be careful to distinguish between speculative intuitions and true explanations for the phenomena that we observe when training deep models.
- Recall that we do not even know why simpler deep neural networks (MLPs and conventional CNNs) generalize well in the first place.
- Even with dropout and weight decay, they remain so flexible that their ability to generalize to unseen data likely needs significantly more refined learning-theoretic generalization guarantees.

- explanation for why it works:
  - by reducing *internal covariate shift*.
  - presumably by *internal covariate shift* the authors meant something like the intuition expressed above—the notion that the distribution of variable values changes over the course of training.
- However, there were two problems with this explanation:
- i. This drift is very different from *covariate shift*, rendering the name a misnomer. If anything, it is closer to concept drift.
- ii. The explanation offers an under-specified intuition but leaves the question of *why precisely this technique works* an open question wanting for a rigorous explanation.

- Throughout this book, we aim to convey the intuitions that practitioners use to guide their development of deep neural networks.
- However, we believe that it is important to separate these guiding intuitions from established scientific fact.
- Eventually, when you master this material and start writing your own research papers you will want to be clear to delineate between technical claims and hunches.
- Following the success of batch normalization, its explanation in terms of *internal covariate shift* has repeatedly surfaced in debates in the technical literature and broader discourse about how to present machine learning research.
- In a memorable speech given while accepting a Test of Time Award at the 2017 NeurIPS conference, Ali Rahimi used *internal covariate shift* as a focal point in an argument likening the modern practice of deep learning to alchemy.
- Subsequently, the example was revisited in detail in a position paper outlining troubling trends in machine learning :cite:`Lipton.Steinhardt.2018`.
- Other authors have proposed alternative explanations for the success of batch normalization, some claiming that batch normalization's success comes despite exhibiting behavior that is in some ways opposite to those claimed in the original paper :cite:`Santurkar.Tsipras.Ilyas.ea.2018`.

- We note that the *internal covariate shift* is no more worthy of criticism than any of thousands of similarly vague claims made every year in the technical machine learning literature.
- Likely, its resonance as a focal point of these debates owes to its broad recognizability to the target audience.
- Batch normalization has proven an indispensable method, applied in nearly all deployed image classifiers, earning the paper that introduced the technique tens of thousands of citations. We conjecture, though, that the guiding principles of regularization through noise injection, acceleration through rescaling and lastly preprocessing may well lead to further inventions of layers and techniques in the future.

- On a more practical note, there are a number of aspects worth remembering about batch normalization:
- During model training, batch normalization continuously adjusts the intermediate output of the network by utilizing the mean and standard deviation of the minibatch, so that the values of the intermediate output in each layer throughout the neural network are more stable.
- Batch normalization for fully connected layers and convolutional layers are slightly different. In fact, for convolutional layers, layer normalization can sometimes be used as an alternative.
- Like a dropout layer, batch normalization layers have different behaviors in training mode and prediction mode.
- Batch normalization is useful for regularization and improving convergence in optimization. On the other hand, the original motivation of reducing internal covariate shift seems not to be a valid explanation.
- For more robust models that are less sensitive to input perturbations, consider removing batch normalization :cite:`wang2022removing`.

Exercises

1. Should we remove the bias parameter from the fully connected layer or the convolutional layer before the batch normalization? Why?
2. Compare the learning rates for LeNet with and without batch normalization.
    1. Plot the increase in validation accuracy.
    2. How large can you make the learning rate before the optimization fails in both cases?
3. Do we need batch normalization in every layer? Experiment with it?
4. Implement a “lite” version of batch normalization that only removes the mean, or alternatively one that only removes the variance. How does it behave?

1. Fix the parameters `beta` and `gamma`. Observe and analyze the results.
2. Can you replace dropout by batch normalization? How does the behavior change?
3. Research ideas: think of other normalization transforms that you can apply:
    1. Can you apply the probability integral transform?
    2. Can you use a full rank covariance estimate? Why should you probably not do that?
    3. Can you use other compact matrix variants (block-diagonal, low-displacement rank, Monarch, etc.)?
    4. Does a sparsification compression act as a regularizer?
    5. Are there other projections (e.g., convex cone, symmetry group-specific transforms) that you can use?

Densely Connected Networks (DenseNet)

- *DenseNet* (dense convolutional network) is to some extent the logical extension of this cite:Huang.Liu.Van-Der-Maaten.ea.2017.
- DenseNet is characterized by both the connectivity pattern where each layer connects to all the preceding layers and the concatenation operation (rather than the addition operator in ResNet) to preserve and reuse features from earlier layers.
- To understand how to arrive at it, let's take a small detour to mathematics.

```
from d2l import torch as d2l
import torch
from torch import nn
```

From ResNet to DenseNet

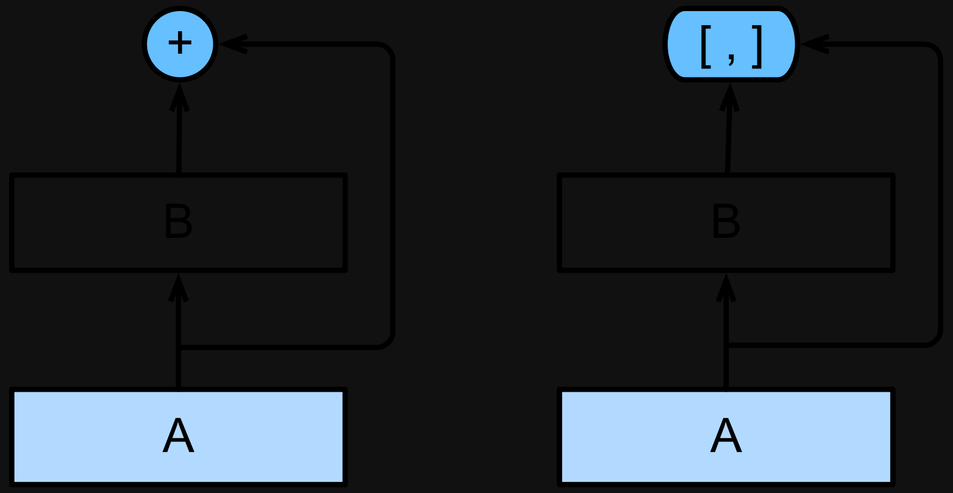
- Recall the Taylor expansion for functions. For the point $x = 0$ it can be written as

$$f(x) = f(0) + x \cdot \left[f'(0) + x \cdot \left[\frac{f''(0)}{2!} + x \cdot \left[\frac{f'''(0)}{3!} + \ldots\right]\right]\right].$$

- The key point is that it decomposes a function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}).$$

- That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one.
- What if we wanted to capture (not necessarily add) information beyond two terms? One such solution is DenseNet :cite:`Huang.Liu.Van-Der-Maaten.ea.2017`.



fig_densenet_block

- As shown in fig_densenet_block, the key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* (denoted by $[,]$) rather than added.
- As a result, we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions:

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots].$$

- all these functions are combined in MLP to reduce the number of features again. In terms of implementation this is quite simple:
- rather than adding terms, we concatenate them.
- name DenseNet arises from the fact that the dependency graph between variables becomes quite dense.
- last layer of such a chain is densely connected to all previous layers. The dense connections are shown in `fig_densenet`.

:label:`fig_densenet`

- main components that compose a DenseNet are *dense blocks* and *transition layers*. The former define how the inputs and outputs are concatenated, while the latter control the number of channels so that it is not too large, since the expansion $\mathbf{x} \rightarrow \left[\mathbf{x}, f_1(\mathbf{x}), f_2\left(\left[\mathbf{x}, f_1\left(\mathbf{x}\right)\right]\right), \ldots\right]$ can be quite high-dimensional.

[Dense Blocks]

- DenseNet uses the modified “batch normalization, activation, and convolution” structure of ResNet (see the exercise in :numref:`sec_resnet`). First, we implement this convolution block structure.

```
def conv_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1))
```

- *dense block* consists of multiple convolution blocks, each using the same number of output channels. In the forward propagation, however, we concatenate the input and output of each convolution block on the channel dimension. Lazy evaluation allows us to adjust the dimensionality automatically.

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            ## Concatenate input and output of each block along the channels
            X = torch.cat((X, Y), dim=1)
        return X
```

- In the following example, we [**define a `DenseBlock` instance**] with 2 convolution blocks of 10 output channels.
- When using an input with 3 channels, we will get an output with $3 + 10 + 10 = 23$ channels. The number of convolution block channels controls the growth in the number of output channels relative to the number of input channels. This is also referred to as the *growth rate*.

```
blk = DenseBlock(2, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 23, 8, 8])
```

[Transition Layers]

- Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A *transition layer* is used to control the complexity of the model. It reduces the number of channels by using an 1×1 convolution. Moreover, it halves the height and width via average pooling with a stride of 2.

```python
def transition_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

- [**Apply a transition layer**] with 10 channels to the output of the dense block in the previous example. This reduces the number of output channels to 10, and halves the height and width.

```
blk = transition_block(10)
blk(Y).shape
```

```
torch.Size([4, 10, 4, 4])
```

DenseNet Model

- Next, we will construct a DenseNet model. DenseNet first uses the same single convolutional layer and max-pooling layer as in ResNet.

```
class DenseNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

- Then, similar to the four modules made up of residual blocks that ResNet uses, DenseNet uses four dense blocks.
- Similar to ResNet, we can set the number of convolutional layers used in each dense block. Here, we set it to 4, consistent with the ResNet-18 model in :numref:`sec_resnet`.
- Furthermore, we set the number of channels (i.e., growth rate) for the convolutional layers in the dense block to 32, so 128 channels will be added to each dense block.
- Similar to ResNet, a global pooling layer and a fully connected layer are connected at the end to produce the output.

```python
@d2l.add_to_class(DenseNet)
def __init__(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
             lr=0.1, num_classes=10):
    super(DenseNet, self).__init__()
    self.save_hyperparameters()


    self.net = nn.Sequential(self.b1())
    for i, num_convs in enumerate(arch):
        self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                          growth_rate))
        ## The number of output channels in the previous dense block
        num_channels += num_convs * growth_rate
        ## A transition layer that halves the number of channels is added
        ## between the dense blocks
        if i != len(arch) - 1:
            num_channels //= 2
            self.net.add_module(f'tran_blk{i+1}', transition_block(
                num_channels))
```

[Training]

Since we are using a deeper network here, in this section, we will reduce the input height and width from 224 to 96 to simplify the computation.

```
model = DenseNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
# trainer.fit(model, data)
```

Summary and Discussion

- main components that compose DenseNet are dense blocks and transition layers. For the latter, we need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.
- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.
- concat –> heavy GPU memory consumption.

Exercises

1. Why do we use average pooling rather than max-pooling in the transition layer?
2. One of the advantages mentioned in the DenseNet paper is that its model parameters are smaller than those of ResNet. Why is this the case?
3. One problem for which DenseNet has been criticized is its high memory consumption.
    1. Is this really the case? Try to change the input shape to 224×224 to see the actual GPU memory consumption empirically.
    2. Can you think of an alternative means of reducing the memory consumption? How would you need to change the framework?
4. Implement the various DenseNet versions presented in Table 1 of the DenseNet paper :cite:`Huang.Liu.Van-Der-Maaten.ea.2017`.
5. Design an MLP-based model by applying the DenseNet idea. Apply it to the housing price prediction task in :numref:`sec_kaggle_house`.