CS428 Introduction to AI

Lecture 4: Convolutional Neural Networks

Outline

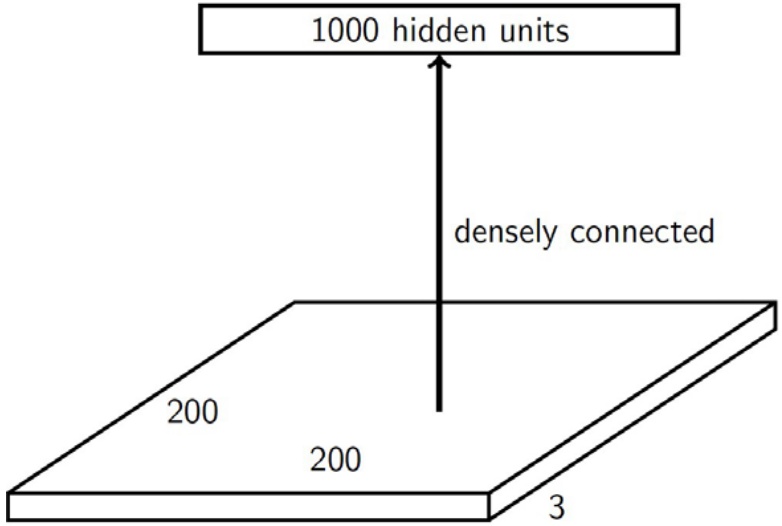
- Understanding convolution
- Building a convolutional neural network
- Creating custom nn.Module subclasses in PyTorch
- The difference between the module and functional APIs

Overview

- What makes computer vision hard?
- Vison needs to be robust to a lot of transformations or distortions:
  - change in pose/viewpoint
  - change in illumination
  - deformation
  - occlusion (some objects are hidden behind others)
- Many object categories can vary wildly in appearance (e.g. chairs)

Overview

Suppose we want to train a network that takes a 200×200 RGB image as input.



What is the problem with having this as the first layer?

- Too many parameters! Input size = $200 \times 200 \times 3 = 120K$.
- Parameters = $120K \times 1000 = 120$ million.

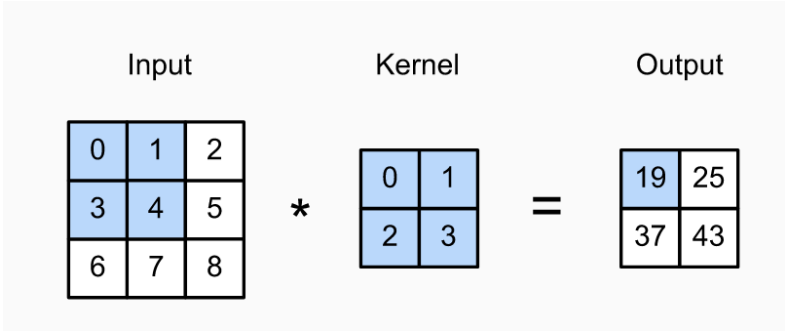
What happens if the object in the image shifts a little?

Overview

- We've already been vectorizing our computations by expressing them in terms of matrix and vector operations.
- Now we'll introduce a new high-level operation, convolution. Here the motivation isn't computational efficiency — we'll see more efficient ways to do the computations later. Rather, the motivation is to get some understanding of what convolution layers can do.
- The thing we convolve by is called a kernel, or filter.

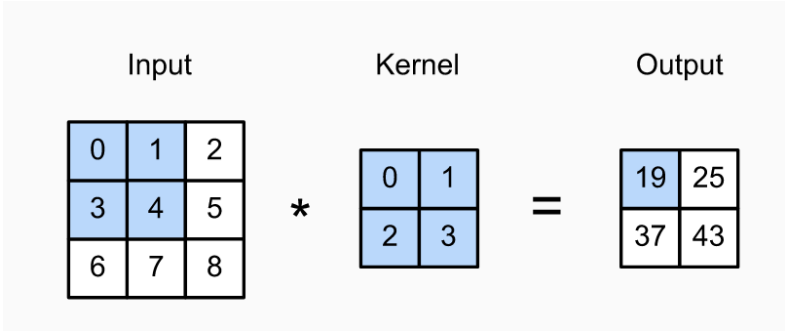
Convolution Operation

- The input is a two-dimensional tensor with a height of 3 and width of 3.
- We mark the shape of the tensor as 3×3.
- The height and width of the kernel are both 2. The shape of the kernel window (or convolution window) is given by the height and width of the kernel (here it is 2×2).



The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:
$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

Convolution Operation



- In the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the upper-left corner of the input tensor and slide it across the input tensor, both from left to right and top to bottom.
- When the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied elementwise and the resulting tensor is summed up yielding a single scalar value.
- This result gives the value of the output tensor at the corresponding location.

Conclusion 1: Tensor Transformation by One Convolution Operation

- input size: $n_h \times n_w$;
- kernel size: $k_h \times k_w$;
- output size: $(n_h - k_h + 1) \times (n_w - k_w + 1)$;

2-D Convolution

- The thing we convolve by is called a kernel, or filter.
- What does this convolution kernel do?

$*$

0	1	0
1	4	1
0	1	0

- blur an image

2-D Convolution

- The thing we convolve by is called a kernel, or filter.
- What does this convolution kernel do?

$*$

0	-1	0
-1	8	-1
0	-1	0

- sharpen an image

2-D Convolution

- The thing we convolve by is called a kernel, or filter.
- What does this convolution kernel do?

$*$

0	-1	0
-1	4	-1
0	-1	0

- detect edges

2-D Convolution

- The thing we convolve by is called a kernel, or filter.
- What does this convolution kernel do?

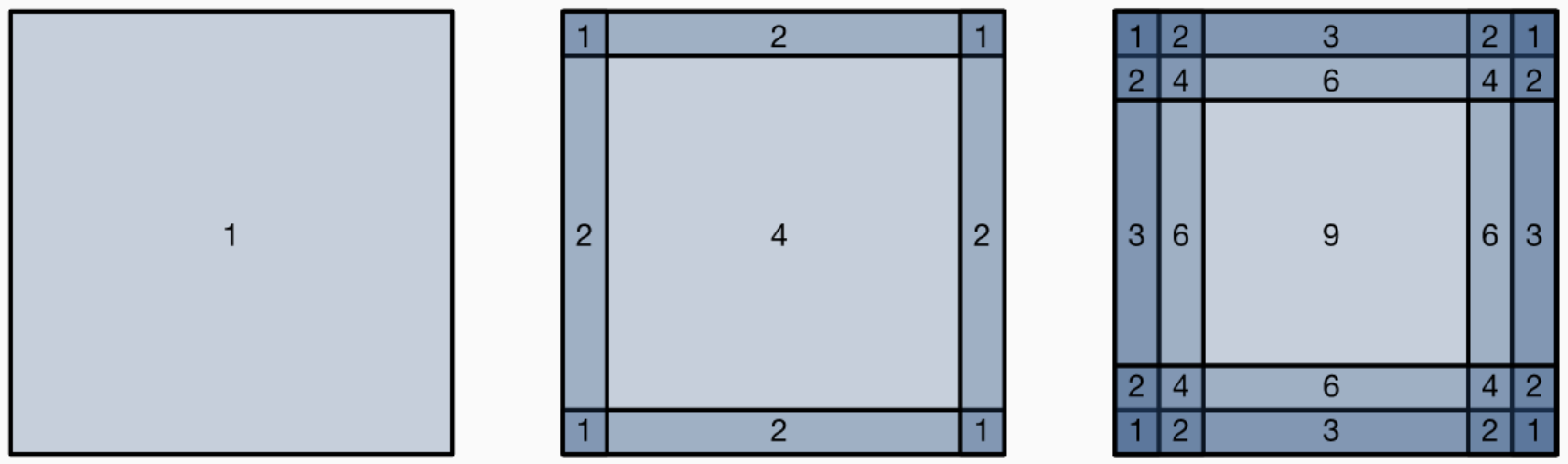
$*$

1	0	-1
2	0	-2
1	0	-1

- detect edges (This filter is known as a Sobel filter.)

Extension: Padding

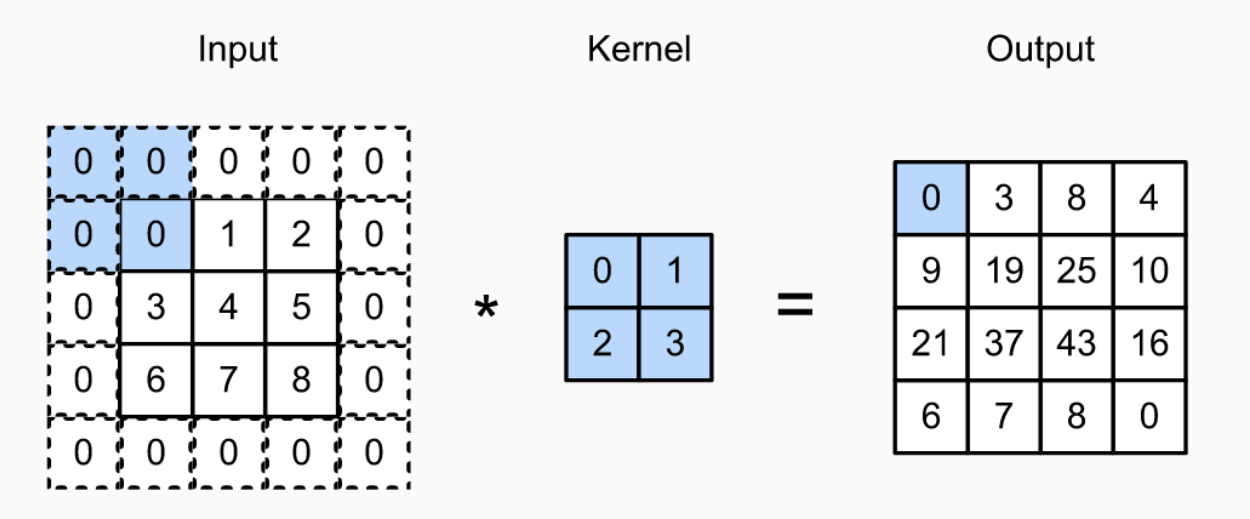
- One tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image.
- An example of pixel utilization for convolutions of size 1×1, 2×2, and 3×3, respectively.



- Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers.

Extension: Padding

- One straightforward solution to this problem is padding, which adds extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image.
- Typically, we set the values of the extra pixels to zero.
- Then, calculate by convolution operations.
- Here is an example with padding size 1×1. Note: 1 row and column are padded on either side, so a total of 2 rows or columns are added.



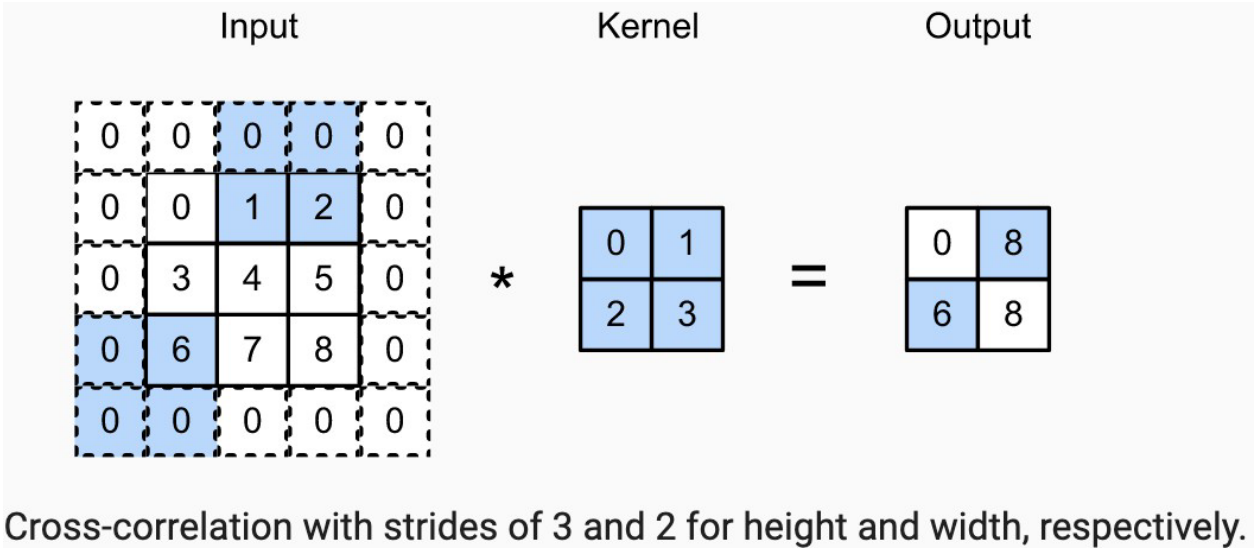
Extension: Padding

Conclusion 2: Tensor Transformation by One Convolution Operation with Padding

- input size: $n_h \times n_w$;
- kernel size: $k_h \times k_w$;
- padding size: $p_h \times p_w$;
- output size: $(n_h - k_h + 2 * p_h + 1) \times (n_w - k_w + 2 * p_w + 1)$;

Extension: Stride

- sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations.
- This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.
- We refer to the number of rows and columns traversed per slide as stride.



Cross-correlation with strides of 3 and 2 for height and width, respectively.

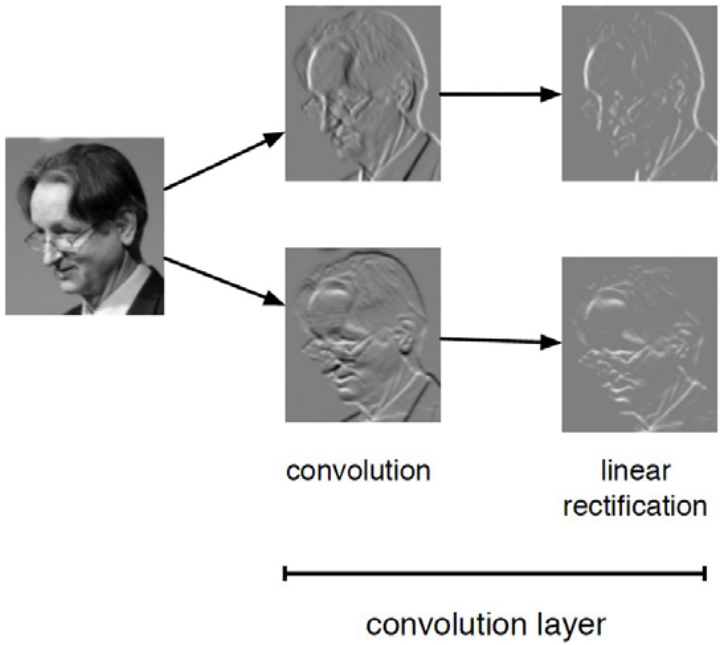
Extension: Stride

Conclusion 3: Tensor Transformation by One Convolution Operation with Padding and Stride

- input size: $n_h \times n_w$;
- kernel size: $k_h \times k_w$;
- padding size: $p_h \times p_w$;
- stride size: $s_h \times s_w$;
- output size:
  $\lfloor (n_h - k_h + 2 * p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2 * p_w + s_w)/s_w \rfloor$;

Activation Function

It's common to apply a linear rectification nonlinearity: $y_i = \max(z_i, 0)$

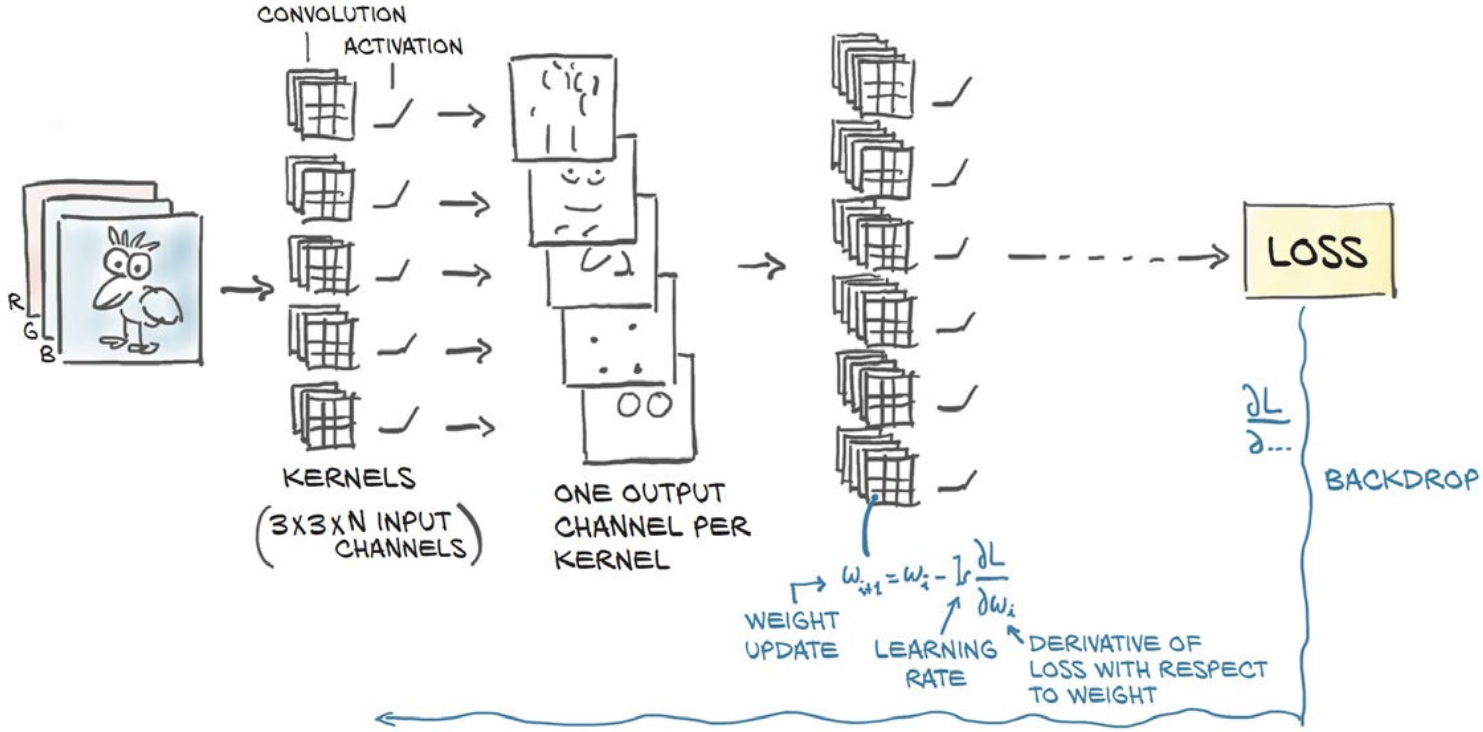


Why might we do this?

- Convolution is a linear operation. Therefore, we need a nonlinearity, otherwise 2 convolution layers would be no more powerful than 1.
- Two edges in opposite directions shouldn't cancel
- Makes the gradients sparse, which helps optimization

Convolutions by Convolutions

The process of learning with convolutions by estimating the gradient at the kernel weights and updating them individually in order to optimize for the loss.

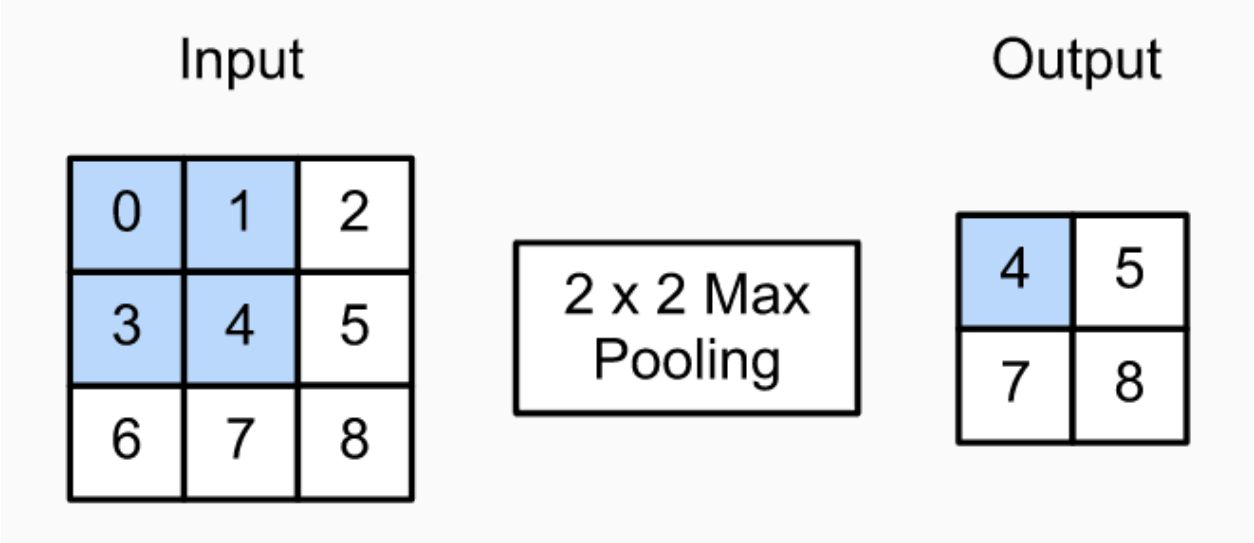


Pooling

- In many cases our ultimate task asks some global question about the image, e.g., does it contain a cat?
- Consequently, the units of our final layer should be sensitive to the entire input.
- By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.
- The deeper we go in the network, the larger the receptive field (relative to the input) to which each hidden node is sensitive.
- Reducing spatial resolution accelerates this process, since the convolution kernels cover a larger effective area.

Pooling

- Pooling layers serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.
- Here is an example of Max-pooling.

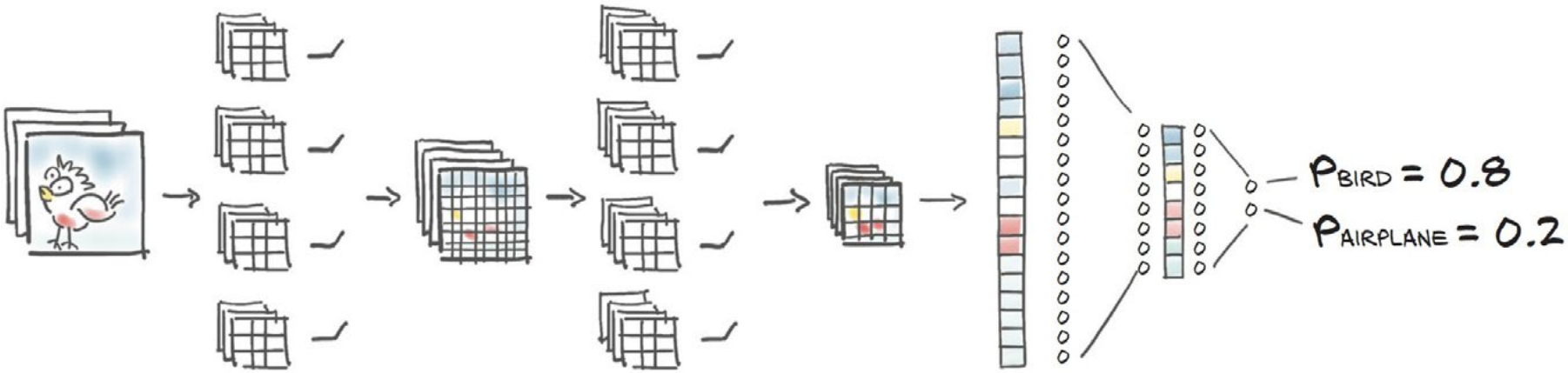


Conclusion 4: Tensor Transformation by One Pooling Operation

- input size: $n_h \times n_w$;
- pooling size: $pool_h \times pool_w$;
- we often assume the stride size keeps the same as the pooling size and padding size is 0.
- output size: $\lceil n_h/pool_h \rceil \times \lceil n_w/pool_w \rceil$;

Shape of a Typical Convolutional Network

An image is fed to a series of convolutions and max pooling modules and then straightened into a 1D vector and fed into fully connected modules.



Conclusion 5: Calculations of the Number of Parameters

For One Convolutional Layer

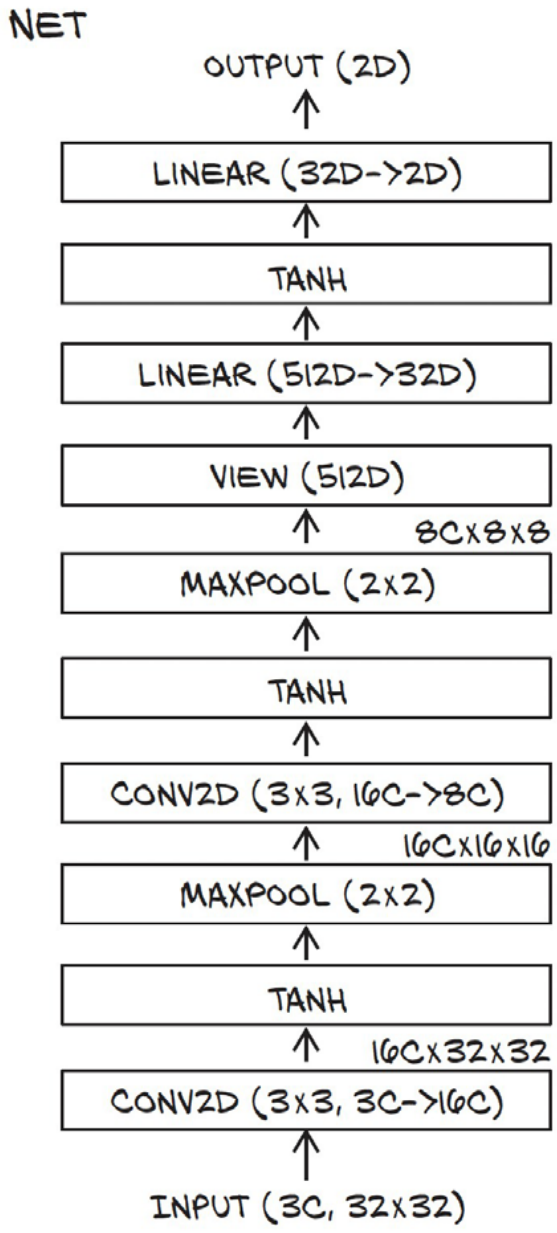
- input size: $n_h \times n_w$;
- kernel size: $k_h \times k_w$;
- padding size: $p_h \times p_w$;
- stride size: $s_h \times s_w$;
- # of input channels: c_{in};
- # of output channels (# of kernel filters): c_{out};
- # of parameters: $c_{out} \times (c_{in} \times k_h \times k_w + 1)$.

Why +1? $'\rightarrow$ We also have a bias associated with each kernel weight parameter.

For One Pooling Layer

# of parameters: 0

Subclassing nn.Module



Subclassing nn.Module

```
# In[26]:
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.Tanh()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.Tanh()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.Tanh()
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

This reshape is what we were missing earlier.

Subclassing nn.Module

- PyTorch allows us to use any computation in our model by subclassing nn.Module.
- In order to subclass nn.Module, at a minimum we need to define a forward function that takes the inputs to the module and returns the output.
- To include submodules, like `nn.Conv2d`, `nn.MaxPool2d`, `nn.Linear`, we typically define them in the constructor `__init__` and assign them to self for use in the forward function.
- They will, at the same time, hold their parameters throughout the lifetime of our module.

Training our Convnet in PyTorch

FOR N EPOCHS:

SPLIT DATASET IN MINIBATCHES

FOR EVERY MINIBATCH:

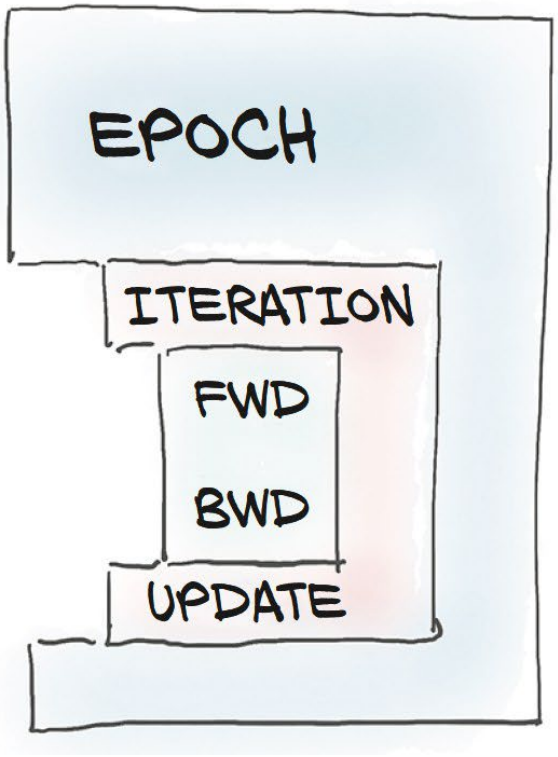
WITH EVERY SAMPLE IN MINIBATCH:

EVALUATE MODEL (FORWARD)

COMPUTE LOSS

ACCUMULATE GRADIENT OF LOSS (BACKWARD)

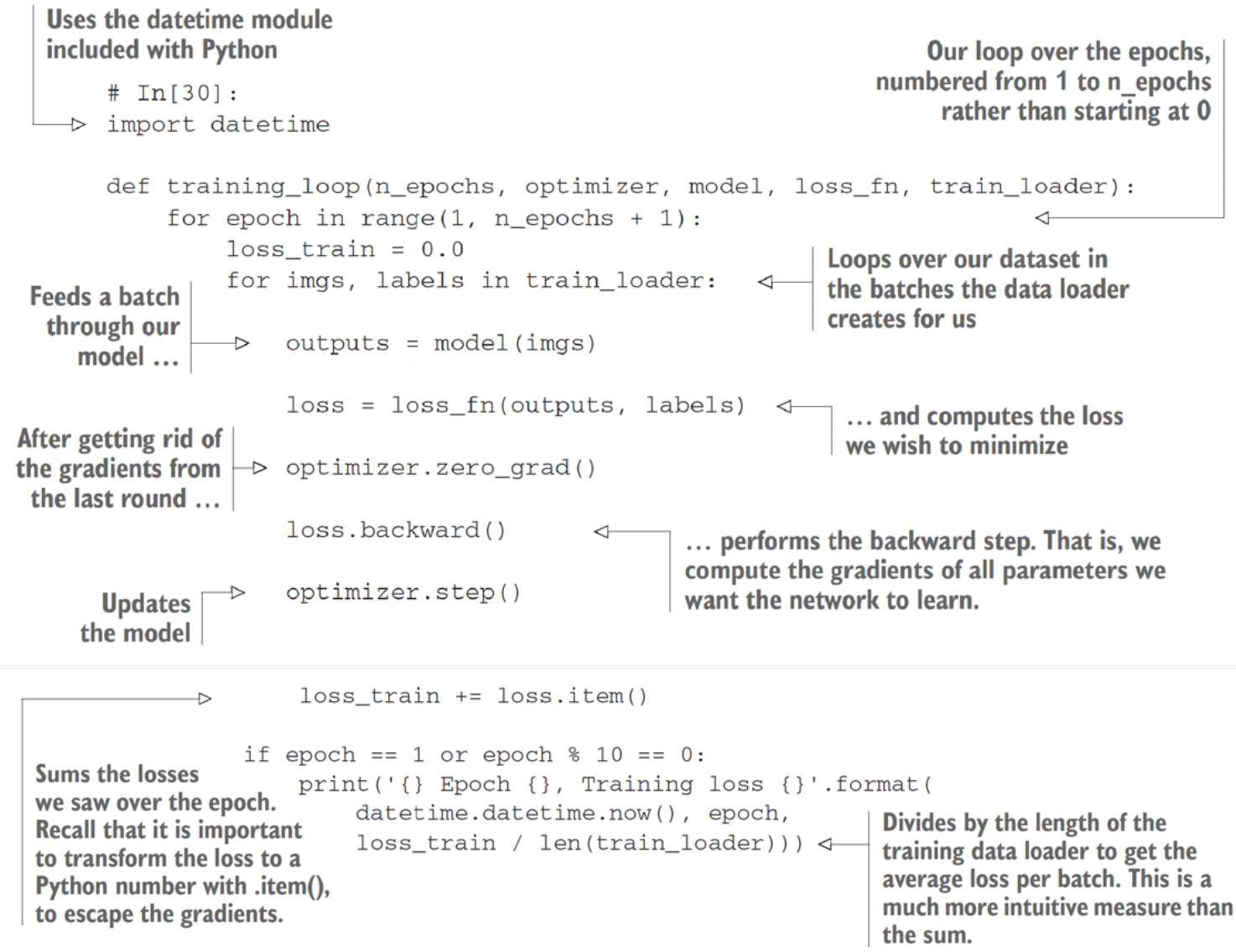
UPDATE MODEL WITH ACCUMULATED GRADIENT



Training our Convnet in PyTorch

1. Feed the inputs through the model (the forward pass).
2. Compute the loss (also part of the forward pass).
3. Zero any old gradients.
4. Call `loss.backward()` to compute the gradients of the loss with respect to all parameters (the backward pass).
5. Have the optimizer take a step in toward lower loss.

Training our Convnet in PyTorch



Training our Convnet in PyTorch

The DataLoader batches up the examples of our cifar2 dataset. Shuffling randomizes the order of the examples from the dataset.

```
# In[31]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = Net()  #
optimizer = optim.SGD(model.parameters(), lr=1e-2)  #
loss_fn = nn.CrossEntropyLoss()  #

training_loop(
    n_epochs = 100,
    optimizer = optimizer,
    model = model,
    loss_fn = loss_fn,
    train_loader = train_loader,
)

# Out[31]:
2020-01-16 23:07:21.889707 Epoch 1, Training loss 0.5634813266954605
2020-01-16 23:07:37.560610 Epoch 10, Training loss 0.3277610331109375
2020-01-16 23:07:54.966180 Epoch 20, Training loss 0.3035225479086493
2020-01-16 23:08:12.361597 Epoch 30, Training loss 0.28249378549824855
2020-01-16 23:08:29.769820 Epoch 40, Training loss 0.2611226033253275
2020-01-16 23:08:47.185401 Epoch 50, Training loss 0.24105800626574048
2020-01-16 23:09:04.644522 Epoch 60, Training loss 0.21997178820477928
2020-01-16 23:09:22.079625 Epoch 70, Training loss 0.20370126601047578
2020-01-16 23:09:39.593780 Epoch 80, Training loss 0.18939699422401987
2020-01-16 23:09:57.111441 Epoch 90, Training loss 0.17283396527266046
2020-01-16 23:10:14.632351 Epoch 100, Training loss 0.1614033816868712
```

Instantiates our network …

… the stochastic gradient descent optimizer we have been working with …

… and the cross entropy loss we met in 7.10

Calls the training loop we defined earlier

Measuring Accuracy

```
# In[32]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=False)
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64,
                                         shuffle=False)

def validate(model, train_loader, val_loader):
    for name, loader in [("train", train_loader), ("val", val_loader)]:
        correct = 0
        total = 0

        with torch.no_grad():
            for imgs, labels in loader:
                outputs = model(imgs)
                _, predicted = torch.max(outputs, dim=1)
                total += labels.shape[0]
                correct += int((predicted == labels).sum())

        print("Accuracy {}: {:.2f}".format(name , correct / total))

validate(model, train_loader, val_loader)

# Out[32]:
Accuracy train: 0.93
Accuracy val: 0.89
```

We do not want gradients here, as we will not want to update the parameters.

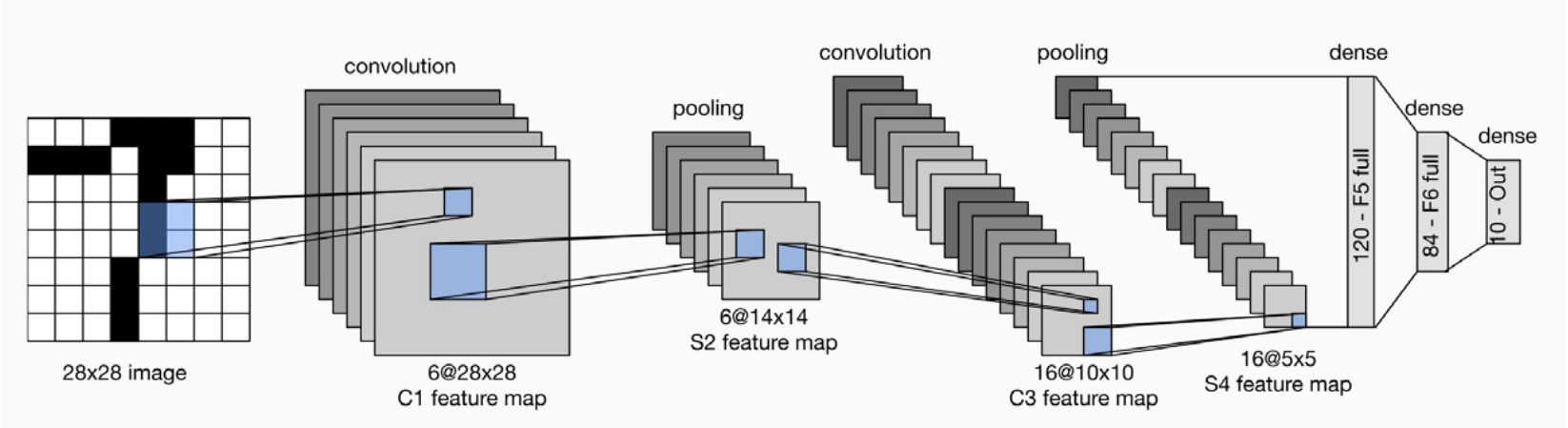
Counts the number of examples, so total is increased by the batch size

Gives us the index of the highest value as output

Comparing the predicted class that had the maximum probability and the ground-truth labels, we first get a Boolean array. Taking the sum gives the number of items in the batch where the prediction and ground truth agree.

Coding Time

Now, it's your turn: try to build the following neural network in PyTorch.



If you have done, you could also fill out the `training_loop` function in the Jupyter notebook.