UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE of ENGINEERING

TECSAR

Transformative Computer Systems and Architecture Research

Introduction to ML
Lecture 17: Convolutional Neural Networks (CNNs)

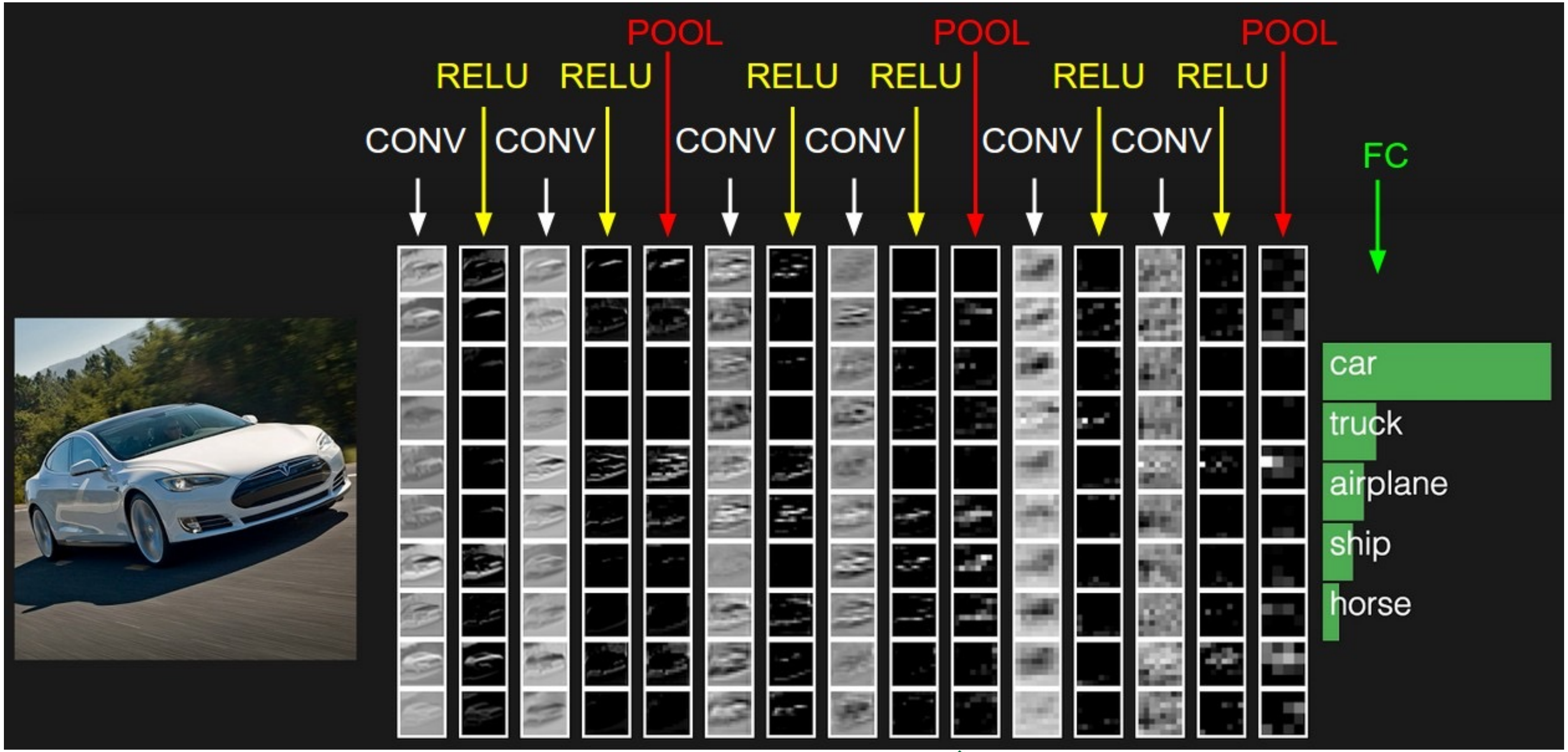
Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

CNNs


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

A Case for Depth and Pooling

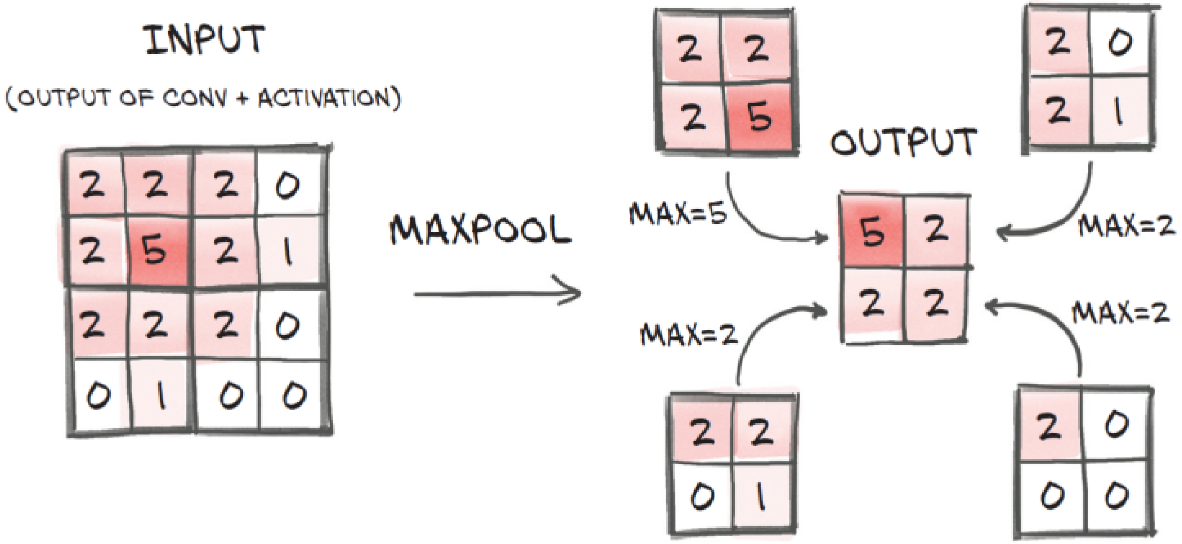
- By moving from fully connected layers to convolutions, we achieve locality and translation invariance.
- We use small kernels, 3×3, or 5×5: that's peak locality, and extract features in local neighborhoods.
- To capture the larger features, we stack one convolution layers and at the same time down-sampling the image between successive convolutions.

- Possibilities for stacking convolutions:

1. ***Average Pooling:*** The *average pooling* was a common approach early on but has fallen out of favor somewhat.
2. ***Max Pooling:*** This approach, called *max pooling,* is currently the most commonly used approach, but it has a downside of discarding the other three-quarters of the data.
3. ***Strided Convolution:*** *where only every Nth pixel is calculated.* The literature shows promise for this approach,.

Max Pooling

- Max pooling is provided by the `nn.MaxPool2d` module (as with convolution, there are versions for 1D and 3D data).
- It takes as input the size of the neighborhood over which to operate the pooling operation.

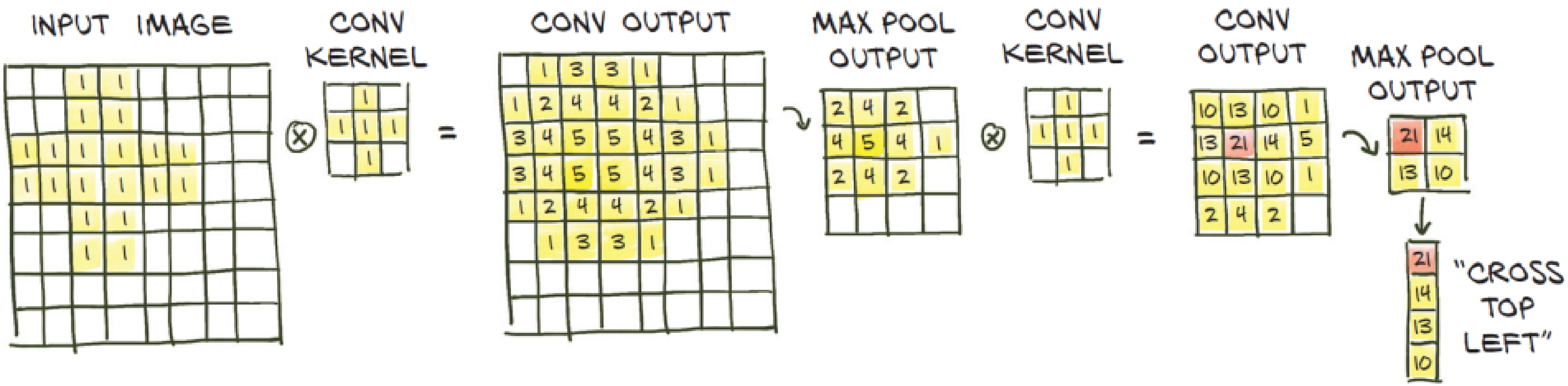



```
# In[21]:
pool = nn.MaxPool2d(2)
output = pool(img.unsqueeze(0))

img.unsqueeze(0).shape, output.shape

# Out[21]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 3, 16, 16]))
```

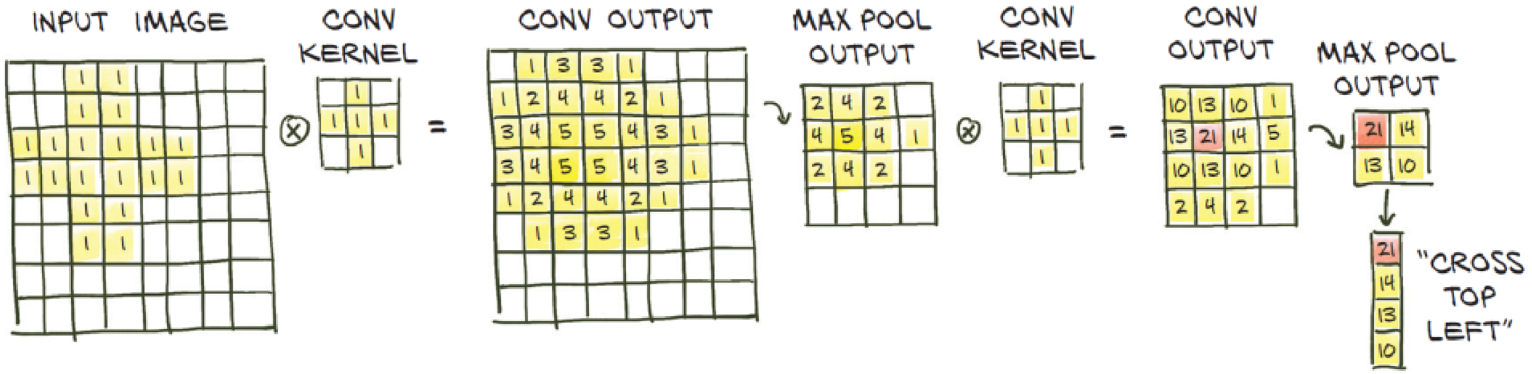
Combining Convolutions and Down Sampling



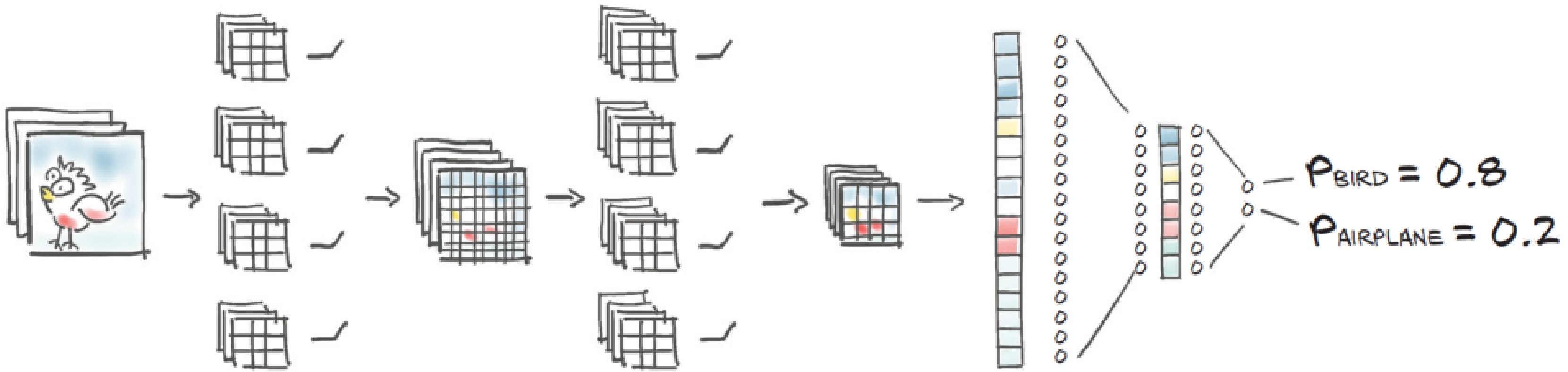
- Principle 1: the first set of kernels operates on small neighborhoods to extract low-level features
- Principle 1: the second set of kernels effectively operates on wider neighborhoods, producing features that are compositions of the previous features.
- This is a very powerful mechanism that provides convolutional neural networks with the ability to see into very complex scenes—much more complex than our 32 × 32 images from the CIFAR-10 dataset

Receptive Field

- The **receptive field** in Convolutional Neural Networks (**CNN**) is the region of the input space that affects a particular output of the network.
- When the second 3 × 3 convolution kernel produces 21 in its conv output, this is based on the top-left 3 × 3 pixels of the first max pool output. They, in turn, correspond to the 6 × 6 pixels in the top-left corner in the first conv output, which in turn are computed by the first convolution from the top-left 7 × 7 pixels. So the pixel in the second convolution output is influenced by a 7 × 7 input square. The first convolution also uses an implicitly “padded” column and row to produce the output in the corner; otherwise, we would have an 8 × 8 square of input pixels informing a given pixel (away from the boundary) in the second convolution’s output.
- In fancy language, we say that a given output neuron of the 3 × 3-conv, 2 × 2-max-pool, 3 × 3-conv construction has a *receptive field* of 8 × 8.



Building the CNN model




```
# In[23]:
model = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.Tanh(),
            nn.MaxPool2d(2),
            nn.Conv2d(16, 8, kernel_size=3, padding=1),
            nn.Tanh(),
            nn.MaxPool2d(2),
            # ...
            nn.Linear(8 * 8 * 8, 32),
            nn.Tanh(),
            nn.Linear(32, 2))
```

Warning: Something important is missing here!

- The size of the first linear layer is dependent on the expected size of the output of `MaxPool2d`: 8 × 8 × 8 = 512.

Model Complexity

https://pytorch.org/docs/stable/generated/torch.numel.html

```
# In[24]:
numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list

# Out[24]:
(18090, [432, 16, 1152, 8, 16384, 32, 64, 2])
```

- That’s very reasonable for a limited dataset of such small images.
- In order to increase the capacity of the model, we could increase the number of output channels for the convolution layers which would lead the linear layer to increase its size as well.

Our Network as nn.Module

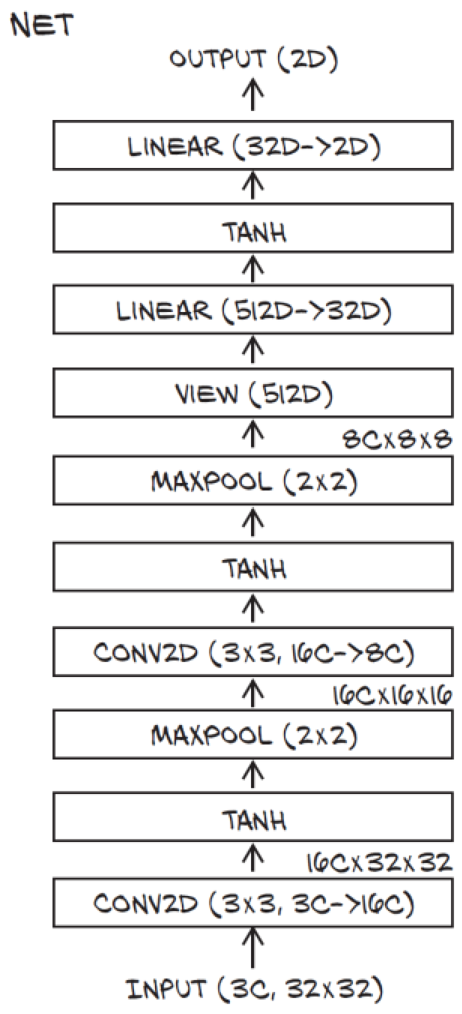
- When we want to build models that do more complex things than just applying one layer after another, we need to leave `nn.Sequential` for something that gives us added flexibility.
- PyTorch allows us to use any computation in our model by subclassing `nn.Module`.
- In order to subclass `nn.Module`, we need to define a `forward` function that takes the inputs to the module and returns the output.
- Typically, our computation will use other modules—premade like convolutions or customized.
- To include these *submodules*, we typically define them in the constructor `__init__` and assign them to `self` for use in the `forward` function.

Building Our Network as 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.



Building Our Network as nn.Module

- Here, we use a subclass of `nn.Module` to contain our entire model. We could also use subclasses to define new building blocks for more complex networks.
- The `Net` class is equivalent to the `nn.Sequential` model in terms of submodules; but by writing the `forward` function explicitly, we can manipulate the output of `self.pool3` directly and call `view` on it to turn it into a $B \times N$ vector.
- Note that we leave the batch dimension as –1 in the call to `view`, since in principle we don't know how many samples will be in the batch.

The Functional API

- PyTorch has *functional* counterparts for every `nn` module.
- By "functional" here we mean "having no internal state"—in other words, "whose output value is solely and fully determined by the value input arguments." -> **Basically, it dose not hold trainable parameter.**
- We can safely switch to the functional counterparts of pooling and activation, since they have no parameters.
- `torch.nn.functional` provides many functions that work like the modules we find in `nn`. But instead of working on the input arguments and stored parameters like the module counterparts, they take inputs and parameters as arguments to the function call.
- It makes sense to keep using `nn` modules for `nn.Linear` and `nn.Conv2d` so that `Net` will be able to manage their `Parameters` during training.

```
# In[28]:
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)

def forward(self, x):
    out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
    out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
    out = out.view(-1, 8 * 8 * 8)
    out = torch.tanh(self.fc1(out))
    out = self.fc2(out)
    return out
```

Training our CNN

- Recall that the core of our convnet is two nested loops: an outer one over the *epochs* and an inner one of the `DataLoader` that produces batches from our `Dataset`.

- In each loop, we then have to

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

Training our CNN: Training Loop

```
# In[30]:
import datetime

def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:

            outputs = model(imgs)

            loss = loss_fn(outputs, labels)

            optimizer.zero_grad()

            loss.backward()

            optimizer.step()

            loss_train += loss.item()

        if epoch == 1 or epoch % 10 == 0:
            print('{} Epoch {}, Training loss {}'.format(
                datetime.datetime.now(), epoch,
                loss_train / len(train_loader)))
```

Uses the datetime module included with Python (`import datetime`)

Our loop over the epochs, numbered from 1 to n_epochs rather than starting at 0 (`for epoch in range(1, n_epochs + 1):`)

Loops over our dataset in the batches the data loader creates for us (`for imgs, labels in train_loader:`)

Feeds a batch through our model … (`outputs = model(imgs)`)

… and computes the loss we wish to minimize (`loss = loss_fn(outputs, labels)`)

After getting rid of the gradients from the last round … (`optimizer.zero_grad()`)

… performs the backward step. That is, we compute the gradients of all parameters we want the network to learn. (`loss.backward()`)

Updates the model (`optimizer.step()`)

Sums the losses we saw over the epoch. Recall that it is important to transform the loss to a Python number with .item(), to escape the gradients. (`loss_train += loss.item()`)

Divides by the length of the training data loader to get the average loss per batch. This is a much more intuitive measure than the sum. (`loss_train / len(train_loader)`)

Training our CNN: Data Loader

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()   #                          Instantiates our network ...
optimizer = optim.SGD(model.parameters(), lr=1e-2)   #   ... the stochastic gradient descent optimizer we have been working with ...
loss_fn = nn.CrossEntropyLoss()   #        ... and the cross entropy loss we met in 7.10

training_loop(                             Calls the training loop we defined earlier
    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
```

Training our CNN: Measuring the 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.

Gives us the index of the highest value as output

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

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.

- Congratulations! This is quite a lot better than the fully connected model. We about halved the number of errors on the validation set.
- Also, we used far fewer parameters. This is telling us that the model does a better job of generalizing its task of recognizing the subject of images from a new sample, through locality and translation invariance.

Saving and Loading our Trained Model

- We achieved relatively good accuracy. Now, let's save our model!

```
# In[33]:
torch.save(model.state_dict(), data_path + 'birds_vs_airplanes.pt')
```

- The birds_vs_airplanes.pt file now contains all the parameters of `model`: which is, weights and biases for the two convolution modules and the two linear modules.
- when we deploy the model in production , we'll need to keep the `model` class handy, create an instance, and then load the parameters back into it:

```
# In[34]:
loaded_model = Net()
loaded_model.load_state_dict(torch.load(data_path
                                         + 'birds_vs_airplanes.pt'))
```

We will have to make sure we don't change the definition of Net between saving and later loading the model state.

Training on the GPU

- It is considered good style to move things to the GPU if one is available. A good pattern is to set the a variable `device` depending on `torch.cuda.is_available`:

```
# In[35]:
device = (torch.device('cuda') if torch.cuda.is_available()
          else torch.device('cpu'))
print(f"Training on device {device}.")
```

- We can amend the training loop by moving the tensors we get from the data loader to the GPU by using the `Tensor.to` method.
- Note that the code is exactly like our first version at the beginning of this section except for the two lines moving the inputs to the GPU.
- **The same amendment must be made to the `validate` function.**

Training on the GPU

```
# In[36]:
import datetime

def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0

        for imgs, labels in train_loader:
            imgs = imgs.to(device=device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            loss_train += loss.item()

        if epoch == 1 or epoch % 10 == 0:
            print('{} Epoch {}, Training loss {}'.format(
                datetime.datetime.now(), epoch,
                loss_train / len(train_loader)))
```

These two lines that move imgs and labels to the device we are training on are the only difference from our previous version.

Training on the GPU

```
# In[37]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = Net().to(device=device)
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[37]:
2020-01-16 23:10:35.563216 Epoch 1, Training loss 0.5717791349265227
2020-01-16 23:10:39.730262 Epoch 10, Training loss 0.3285350770137872
2020-01-16 23:10:45.906321 Epoch 20, Training loss 0.29493294959994637
2020-01-16 23:10:52.086905 Epoch 30, Training loss 0.26962305994550134
2020-01-16 23:10:56.551582 Epoch 40, Training loss 0.24709946277794564
2020-01-16 23:11:00.991432 Epoch 50, Training loss 0.22623272664892446
2020-01-16 23:11:05.421524 Epoch 60, Training loss 0.20996672821462534
2020-01-16 23:11:09.951312 Epoch 70, Training loss 0.1934866009719053
2020-01-16 23:11:14.499484 Epoch 80, Training loss 0.1799132404908253
2020-01-16 23:11:19.047609 Epoch 90, Training loss 0.16620008706761774
2020-01-16 23:11:23.590435 Epoch 100, Training loss 0.15667157247662544
```

Moves our model (all parameters) to the GPU. If you forget to move either the model or the inputs to the GPU, you will get errors about tensors not being on the same device, because the PyTorch operators do not support mixing GPU and CPU inputs.

- We also need to instantiate our model, move it to `device`, and run it as before
- Even for our small network here, we do see a sizable increase in speed.
- The advantage of computing on GPUs is more visible for larger models.

Training on the GPU

- There is a slight complication when loading network weights.
- PyTorch will attempt to load the weight to the same device it was saved from.
- It is a bit more concise to instruct PyTorch to override the device information when loading weights. This is done by passing the `map_location` keyword argument to `torch.load`:

```
# In[39]:
loaded_model = Net().to(device=device)
loaded_model.load_state_dict(torch.load(data_path
                                        + 'birds_vs_airplanes.pt',
                                        map_location=device))

# Out[39]:
<All keys matched successfully>
```