Chpt8.5 Model design

Sep 7, 2021

Admittedly, our birds versus airplanes dataset wasn't that complicated: the images were very small, and the object under investigation was centered and took up most of the viewport. If we moved to ImageNet, we would find larger, more complex images, where the right answer would depend on multiple visual clues, often hierarchically organized. For instance, when trying to predict whether a dark brick shape is a remote control or a cell phone, the network could be looking for something like a screen.

Adding memory capacity: width

Given our feed-forward architecture, there are a couple of dimensions we'd likely want to explore before getting into further complications. The first dimension is the *width* of the network: the number of neurons per layer, or channels per convolution. We can make a model wider very easily by specifying a larger number of output channels in the first conv and increase the subsequent layers accordingly.

```
n_chans1 = 32

class NetWidth(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                               padding=1)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 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* self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

The numbers specifying channels and features for each layer are directly related to the number of parameters in a model; all other things being equal, they increase the *capacity* of the model. Let's

check how many parameters our model has now:

```
# 38386
sum(p.numel() for p in model.parameters())
```

There are a few more tricks we can play at the model level to control overfitting. Let's review the most common ones.

8.5.2 Helping our model to converge and generalize: Regularization

Training a model involves two critcal steps: optimization, when we need the loss to decrease on the training set; and generalizaiton, when the model has to work not only on the training set but also on data it has not seen before. The mathematical tools aimed at easing these two steps are sometimes subsumed under the label *regularization*.

1. Keeping the parameters in check: weight penalties

The first way to stablize generalization is to add a regularization term to the loss. It is a penalty on larger weight values. This makes the loss have a smoother topography, and there's relatively less to gain from fitting individual samples. The most popular regularization terms of this kind are L2 regularization, wchih is the sum of squares of all weights in the model. L2 regularization is also referred to as *weight decay*. Adding L2 regularization to the loss function is equivalent to decreasing each weight by an amount proportional to its current value.

In Pytorch, we could implement regularization pretty easily by adding a term to the loss. After computing the loss, whatever the loss function is, we can iterate the parameters of the model, sum their respective square or abs (L1), and backpropagate:

```
def training_loop_l2reg(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)

            l2_lambda = 0.001
            l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
            loss = loss + l2_lambda * l2_norm

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

```
17
18              loss_train += loss.item()
19          if epoch == 1 or epoch % 10 == 0:
20              print('{} Epoch {}, Training loss {}'.format(datetime.datetime.now(),
21                epoch,
22                loss_train / len(train_loader)))
```

2. Not relying too much on a single input: Dropout

Summary:

1. Using convolution produces networks with fewer parameters, exploiting locality and featuring translation invariance.

2. Stacking multiple convolutions with their activations one after the other, and using max pooling in between, has the effect of applying convolutions to increasingly smaller feature images, thereby effectively accounting for spatial relationships across larger portions of the input image as depth increases.

3. Any nn.Module subclass can recursively collect and return its and its children's parameters. This technique can be used to count them, feed them into the optimizer, or inspect their values.