Training a State-of-the-Art Model

At a high level, a state-of-the-art ML model refers to a model that achieves the best performance on a particular task, dataset, and metric compared to other published models.

To train a SOTA model, we must apply a few major techniques. We will be using fastai library.

Normalization:

When training a model, it helps if your input data is normalized—that is, has a mean of 0 and a standard deviation of 1. But most images and computer vision libraries use values between 0 and 255 for pixels, or between 0 and 1; in either case, your data is not going to have a mean of 0 and a standard deviation of 1. Fortunately, normalizing the data is easy to do in fastai by adding the Normalize transform. This acts on a whole mini-batch at once, so you can add it to

the batch_tfms section of your data block. You need to pass to this transform the mean and standard deviation that you want to use; fastai comes with the standard ImageNet mean and standard deviation already defined. (If you do not pass any statistics to the Normalize transform, fastai will automatically calculate them from a single batch of your data.)

```
dblock = DataBlock(blocks=(ImageBlock, CategoryBlock),
                   get_items=get_image_files,
                   get_y=parent_label,
                   item_tfms=Resize(460),
                   batch_tfms=[*aug_transforms(size=size, min_scale=0.75),
                   Normalize.from_stats(*imagenet_stats)])

x,y = dls.one_batch()
x.mean(dim=[0,2,3]),x.std(dim=[0,2,3])
    (TensorImage([-0.0787, 0.0525, 0.2136], device='cuda:5'),
    TensorImage([1.2330, 1.2112, 1.3031], device='cuda:5'))
```

Normalization becomes especially important when using pretrained models. The pretrained model knows how to work with only data of the type that it has seen before. If the average pixel value was 0 in the data it was trained with, but your data has 0 as the minimum possible value of a pixel, then the model is going to be seeing something very different from what is intended!

This means that when you distribute a model, you need to also distribute the statistics used for normalization, since anyone using it for inference or transfer learning will need to use the same statistics. By the same token, if you're using a model that someone else has trained, make sure you find out what normalization statistics they used, and match them. the fastai library automatically adds the proper Normalize transform; the model has been pretrained with certain statistics in Normalize (usually coming from the ImageNet dataset), so the library can fill those in for you. Note that this applies to only pretrained models, which is why we need to add this information manually here, when training from scratch.

Progressive resizing:

Start training using small images, and end training using large images. Spending most of the epochs training with small images helps training complete much faster. Completing training using large images makes the final accuracy much higher. We call this approach progressive resizing. Progressive resizing has an additional benefit: it is another form of data augmentaion. Therefore, you should expect to see better generalization of your models that are trained with progressive resizing. Note that for transfer learning, progressive resizing may actually hurt performance.

This is most likely to happen if your pretrained model was quite similar to your transfer learning task and the dataset and was trained on similar-sized images, so the weights don't need to be changed much. In that case, training on smaller images may damage the pretrained weights.

On the other hand, if the transfer learning task is going to use images that are of different sizes, shapes, or styles than those used in the pretraining task, progressive resizing will probably help. As always, the answer to "Will it help?" is "Try it!"

Test Time augmentation:

We have been using random cropping as a way to get some useful data augmentation, which leads to better generalization, and results in a need for less training data. When we use random cropping, fastai will automatically use center-cropping for the validation set—that is, it will select the largest square area it can in the center of the image, without going past the image's edges.

This can often be problematic. For instance, in a multi-label dataset, sometimes there are small objects toward the edges of an image; these could be entirely cropped out by center cropping. Even for problems such as our pet breed classification example, it's possible that a critical feature necessary for identifying the correct breed, such as the color of the nose, could be cropped out.

One solution to this problem is to avoid random cropping entirely. Instead, we could simply squish or stretch the rectangular images to fit into a square space. But then we miss out on a very useful data augmentation, and we also make the image recognition more difficult for our model,

because it has to learn how to recognize squished and squeezed images, rather than just correctly proportioned images.

Another solution is to not center crop for validation, but instead to select a number of areas to crop from the original rectangular image, pass each of them through our model, and take the maximum or average of the predictions. In fact, we could do this not just for different crops, but for different values across all of our test time augmentation parameters. This is known as test time augmentation (TTA).

Depending on the dataset, test time augmentation can result in dramatic improvements in accuracy. It does not change the time required to train at all, but will increase the amount of time required for validation or inference by the number of test-time-augmented images requested. By default, fastai will use the unaugmented center crop image plus four randomly augmented images.

You can pass any DataLoader to fastai's tta method; by default, it will use your validation set:

```
preds,targs = learn.tta()

accuracy(preds, targs).item()

        0.8737863898277283
```

Mixup:

Mixup, introduced in the 2017 paper "mixup: Beyond Empirical Risk Minimization" by Hongyi Zhang et al., is a powerful data augmentation technique that can provide dramatically higher accuracy, especially when you don't have much data and don't have a pretrained model that was trained on data similar to your dataset.

Mixup works as follows, for each image:

1. Select another image from your dataset at random.

2. Pick a weight at random.

3. Take a weighted average (using the weight from step 2) of the selected image with

your image; this will be your independent variable.

4. Take a weighted average (with the same weight) of this image's labels with your

image's labels; this will be your dependent variable.

In pseudocode, we're doing this (where t is the weight for our weighted average):

```
image2,target2 = dataset[randint(0,len(dataset)]

t = random_float(0.5,1.0)
```

```
new_image = t * image1 + (1-t) * image2

new_target = t * target1 + (1-t) * target2
```

Mixing a church and a gas station

All you need to know is that you use the cbs parameter to Learner to pass callbacks.

Here is how we train a model with Mixup:

```
model = xresnet50()

learn = Learner(dls, model, loss_func=CrossEntropyLossFlat(),

metrics=accuracy, cbs=Mixup)

learn.fit_one_cycle(5, 3e-3)
```

What happens when we train a model with data that's "mixed up" in this way? Clearly, it's going to be harder to train, because it's harder to see what's in each image. And the model has to predict two labels per image, rather than just one, as well as figuring out how much each one is weighted. Overfitting seems less likely to be a problem, however, because we're not showing the same image in each epoch, but are instead showing a random combination of two images.

Mixup requires far more epochs to train to get better accuracy, compared to other augmentation approaches.

Label smoothing:

In the theoretical expression of loss, in classification problems, our targets are one-hot encoded (in practice, we tend to avoid doing this to save memory, but what we compute is the same loss as if we had used one-hot encoding). That means the model is trained to return 0 for all categories but one, for which it is trained to return 1.

Even 0.999 is not "good enough"; the model will get gradients and learn to predict activations with even higher confidence. This encourages overfitting and gives you at inference time a model that is not going to give meaningful probabilities: it will always say 1 for the predicted category even if it's not too sure, just because it was trained this way.

This can become very harmful if your data is not perfectly labeled. In the bear classifier we studied in Chapter 2, we saw that some of the images were mislabeled, or contained two different kinds of bears. In general, your data will never be perfect. Even if the labels were manually produced by humans, they could make mistakes, or have differences of opinions on images that are harder to label.

Instead, we could replace all our 1s with a number a bit less than 1, and our 0s with a number a bit more than 0, and then train. This is called label smoothing. By encouraging your model to be less confident, label smoothing will make your training more robust, even if there is mislabeled data. The result will be a model that generalizes better at inference.

To use this in practice, we just have to change the loss function in our call to Learner:

```
model = xresnet50()

learn = Learner(dls, model, loss_func=LabelSmoothingCrossEntropy(),

metrics=accuracy)

learn.fit_one_cycle(5, 3e-3)
```

As with Mixup, you won't generally see significant improvements from label smoothing until you train more epochs.