fastbook (/github/fastai/fastbook/tree/master)
/ 18_CAM.ipynb (/github/fastai/fastbook/tree/master/18_CAM.ipynb)

In [ ]:

```
#hide
! [ -e /content ] && pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

In [ ]:

```
#hide
from fastbook import *
```

[[chapter_cam]]

CNN Interpretation with CAM

Now that we know how to build up pretty much anything from scratch, let's use that knowledge to create entirely new (and very useful!) functionality: the *class activation map*. It gives us some insight into why a CNN made the predictions it did.

In the process, we'll learn about one handy feature of PyTorch we haven't seen before, the *hook*, and we'll apply many of the concepts introduced in the rest of the book. If you want to really test out your understanding of the material in this book, after you've finished this chapter, try putting it aside and recreating the ideas here yourself from scratch (no peeking!).

CAM and Hooks

The class activation map (CAM) was introduced by Bolei Zhou et al. in "Learning Deep Features for Discriminative Localization" (https://arxiv.org/abs/1512.04150). It uses the output of the last convolutional layer (just before the average pooling layer) together with the predictions to give us a heatmap visualization of why the model made its decision. This is a useful tool for interpretation.

More precisely, at each position of our final convolutional layer, we have as many filters as in the last linear layer. We can therefore compute the dot product of those activations with the final weights to get, for each location on our feature map, the score of the feature that was used to make a decision.

We're going to need a way to get access to the activations inside the model while it's training. In PyTorch this can be done with a *hook*. Hooks are PyTorch's equivalent of fastai's callbacks. However, rather than allowing you to inject code into the training loop like a fastai `Learner` callback, hooks allow you to inject code into the forward and backward calculations themselves. We can attach a hook to any layer of the model, and it will be executed when we compute the outputs (forward hook) or during backpropagation (backward hook). A forward hook is a function that takes three things—a module, its input, and its output—and it can perform any behavior you want. (fastai also provides a handy `HookCallback` that we won't cover here, but take a look at the fastai docs; it makes working with hooks a little easier.)

To illustrate, we'll use the same cats and dogs model we trained in <>:

In [ ]:

```
path = untar_data(URLs.PETS)/'images'
def is_cat(x): return x[0].isupper()
dls = ImageDataLoaders.from_name_func(
    path, get_image_files(path), valid_pct=0.2, seed=21,
    label_func=is_cat, item_tfms=Resize(224))
learn = vision_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1)
```

epoch	train_loss	valid_loss	error_rate	time
0	0.145994	0.019272	0.006089	00:14

epoch	train_loss	valid_loss	error_rate	time
0	0.053405	0.052540	0.010825	00:19

To start, we'll grab a cat picture and a batch of data:

In [ ]:

```
img = PILImage.create(image_cat())
x, = first(dls.test_dl([img]))
```

For CAM we want to store the activations of the last convolutional layer. We put our hook function in a class so it has a state that we can access later, and just store a copy of the output:

In [ ]:

```
class Hook():
    def hook_func(self, m, i, o): self.stored = o.detach().clone()
```

We can then instantiate a `Hook` and attach it to the layer we want, which is the last layer of the CNN body:

In [ ]:

```
hook_output = Hook()
hook = learn.model[0].register_forward_hook(hook_output.hook_func)
```

Now we can grab a batch and feed it through our model:

In [ ]:

```
with torch.no_grad(): output = learn.model.eval()(x)
```

And we can access our stored activations:

In [ ]:

```
act = hook_output.stored[0]
```

Let's also double-check our predictions:

In [ ]:

```
F.softmax(output, dim=-1)
```

Out[ ]:

```
tensor([[0.0010, 0.9990]], device='cuda:0')
```

We know `0` (for `False`) is "dog," because the classes are automatically sorted in fastai, bu we can still double-check by looking at `dls.vocab`:

In [ ]:

```
dls.vocab
```

Out[ ]:

```
(#2) [False,True]
```

So, our model is very confident this was a picture of a cat.

To do the dot product of our weight matrix (2 by number of activations) with the activations (batch size by activations by rows by cols), we use a custom `einsum`:

In [ ]:

```
x.shape
```

Out[ ]:

```
torch.Size([1, 3, 224, 224])
```

In [ ]:

```
cam_map = torch.einsum('ck,kij->cij', learn.model[1][-1].weight, act)
cam_map.shape
```

Out[ ]:

```
torch.Size([2, 7, 7])
```

For each image in our batch, and for each class, we get a 7×7 feature map that tells us where the activations were higher and where they were lower. This will let us see which areas of the pictures influenced the model's decision.

For instance, we can find out which areas made the model decide this animal was a cat (note that we need to `decode` the input `x` since it's been normalized by the `DataLoader`, and we need to cast to `TensorImage` since at the time this book is written PyTorch does not maintain types when indexing—this may be fixed by the time you are reading this):

In [ ]:

```
x_dec = TensorImage(dls.train.decode((x,))[0][0])
_,ax = plt.subplots()
x_dec.show(ctx=ax)
ax.imshow(cam_map[1].detach().cpu(), alpha=0.6, extent=(0,224,224,0),
              interpolation='bilinear', cmap='magma');
```

The areas in bright yellow correspond to high activations and the areas in purple to low activations. In this case, we can see the head and the front paw were the two main areas that made the model decide it was a picture of a cat.

Once you're done with your hook, you should remove it as otherwise it might leak some memory:

In [ ]:

```
hook.remove()
```

That's why it's usually a good idea to have the `Hook` class be a *context manager*, registering the hook when you enter it and removing it when you exit. A context manager is a Python construct that calls `__enter__` when the object is created in a `with` clause, and `__exit__` at the end of the `with` clause. For instance, this is how Python handles the `with open(...) as f:` construct that you'll often see for opening files without requiring an explicit `close(f)` at the end. If we define `Hook` as follows:

In [ ]:

```
class Hook():
    def __init__(self, m):
        self.hook = m.register_forward_hook(self.hook_func)
    def hook_func(self, m, i, o): self.stored = o.detach().clone()
    def __enter__(self, *args): return self
    def __exit__(self, *args): self.hook.remove()
```

we can safely use it this way:

In [ ]:

```
with Hook(learn.model[0]) as hook:
    with torch.no_grad(): output = learn.model.eval()(x.cuda())
    act = hook.stored
```

fastai provides this `Hook` class for you, as well as some other handy classes to make working with hooks easier.

This method is useful, but only works for the last layer. *Gradient CAM* is a variant that addresses this problem.

Gradient CAM

The method we just saw only lets us compute a heatmap with the last activations, since once we have our features, we have to multiply them by the last weight matrix. This won't work for inner layers in the network. A variant introduced in the paper "Grad-CAM: Why Did You Say That? Visual Explanations from Deep Networks via Gradient-based Localization" (https://arxiv.org/abs/1611.07450) in 2016 uses the gradients of the final activation for the desired class. If you remember a little bit about the backward pass, the gradients of the output of the last layer with respect to the input of that layer are equal to the layer weights, since it is a linear layer.

With deeper layers, we still want the gradients, but they won't just be equal to the weights anymore. We have to calculate them. The gradients of every layer are calculated for us by PyTorch during the backward pass, but they're not stored (except for tensors where `requires_grad` is `True`). We can, however, register a hook on the backward pass, which

PyTorch will give the gradients to as a parameter, so we can store them there. For this we will use a `HookBwd` class that works like `Hook`, but intercepts and stores gradients instead of activations:

In [ ]:

```
class HookBwd():
    def __init__(self, m):
        self.hook = m.register_backward_hook(self.hook_func)
    def hook_func(self, m, gi, go): self.stored = go[0].detach().clone()
    def __enter__(self, *args): return self
    def __exit__(self, *args): self.hook.remove()
```

Then for the class index `1` (for `True`, which is "cat") we intercept the features of the last convolutional layer as before, and compute the gradients of the output activations of our class. We can't just call `output.backward()`, because gradients only make sense with respect to a scalar (which is normally our loss) and `output` is a rank-2 tensor. But if we pick a single image (we'll use `0`) and a single class (we'll use `1`), then we *can* calculate the gradients of any weight or activation we like, with respect to that single value, using `output[0,cls].backward()`. Our hook intercepts the gradients that we'll use as weights:

In [ ]:

```
cls = 1
with HookBwd(learn.model[0]) as hookg:
    with Hook(learn.model[0]) as hook:
        output = learn.model.eval()(x.cuda())
        act = hook.stored
    output[0,cls].backward()
    grad = hookg.stored
```

The weights for our Grad-CAM are given by the average of our gradients across the feature map. Then it's exactly the same as before:

In [ ]:

```
w = grad[0].mean(dim=[1,2], keepdim=True)
cam_map = (w * act[0]).sum(0)
```

In [ ]:

```
_,ax = plt.subplots()
x_dec.show(ctx=ax)
ax.imshow(cam_map.detach().cpu(), alpha=0.6, extent=(0,224,224,0),
              interpolation='bilinear', cmap='magma');
```

The novelty with Grad-CAM is that we can use it on any layer. For example, here we use it on the output of the second-to-last ResNet group:

In [ ]:

```
with HookBwd(learn.model[0][-2]) as hookg:
    with Hook(learn.model[0][-2]) as hook:
        output = learn.model.eval()(x.cuda())
        act = hook.stored
    output[0,cls].backward()
    grad = hookg.stored
```

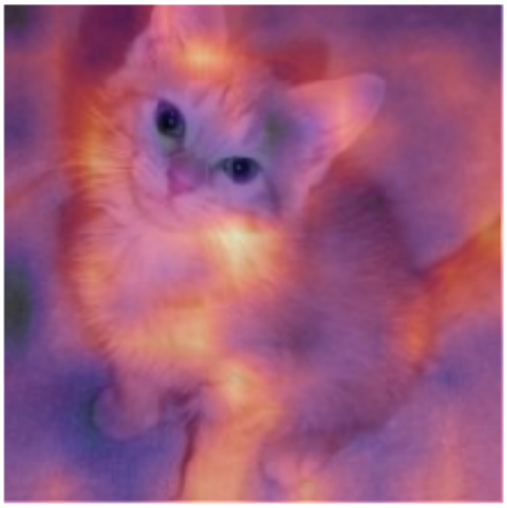
In [ ]:

```
w = grad[0].mean(dim=[1,2], keepdim=True)
cam_map = (w * act[0]).sum(0)
```

And we can now view the activation map for this layer:

In [ ]:

```
_,ax = plt.subplots()
x_dec.show(ctx=ax)
ax.imshow(cam_map.detach().cpu(), alpha=0.6, extent=(0,224,224,0),
              interpolation='bilinear', cmap='magma');
```



Conclusion

Model interpretation is an area of active research, and we just scraped the surface of what is possible in this brief chapter. Class activation maps give us insight into why a model predicted a certain result by showing the areas of the images that were most responsible for a given prediction. This can help us analyze false positives and figure out what kind of data is missing in our training to avoid them.

Questionnaire

1. What is a "hook" in PyTorch?
2. Which layer does CAM use the outputs of?
3. Why does CAM require a hook?
4. Look at the source code of the `ActivationStats` class and see how it uses hooks.
5. Write a hook that stores the activations of a given layer in a model (without peeking, if possible).
6. Why do we call `eval` before getting the activations? Why do we use `no_grad` ?
7. Use `torch.einsum` to compute the "dog" or "cat" score of each of the locations in the last activation of the body of the model.

8. How do you check which order the categories are in (i.e., the correspondence of index->category)?
9. Why are we using `decode` when displaying the input image?
10. What is a "context manager"? What special methods need to be defined to create one?
11. Why can't we use plain CAM for the inner layers of a network?
12. Why do we need to register a hook on the backward pass in order to do Grad-CAM?
13. Why can't we call `output.backward()` when `output` is a rank-2 tensor of output activations per image per class?

Further Research

1. Try removing `keepdim` and see what happens. Look up this parameter in the PyTorch docs. Why do we need it in this notebook?
2. Create a notebook like this one, but for NLP, and use it to find which words in a movie review are most significant in assessing the sentiment of a particular movie review.

In [ ]: