Pet Breeds

Pet Breeds verisetini untar ettikten sonra, dataloaders'ı elde etmek için aşağıdaki gibi datablock api kullanılıyor.

Farklı olan ilk yapı, labelları elde etme fonksiyonu, path/'images' yolundaki fotoğraflar resimlerin ismine göre labeling edilecek bu yüzden regular expressions kullanılmış.

Ayrıca item_tfms ve batch_tfms aşağıdaki gibi belirtilmiş. Burada özel bir yaklaşım uygulanıyor, önce tüm resimler 460x460 şeklinde büyük boyutlu olarak ele alınıyor sonra bunların içerisinden gpu yardımıyla aug_transforms ile random crops alınıyor ve batch transforms uygulanıyor. Bu yöntemle daha başarılı augmentation elde edebiliyoruz.

```
pets = DataBlock(blocks = (ImageBlock, CategoryBlock),
                 get_items=get_image_files,
                 splitter=RandomSplitter(seed=42),
                 get_y=using_attr(RegexLabeller(r'(.+)_\d+.jpg$'), 'name'),
                 item_tfms=Resize(460),
                 batch_tfms=aug_transforms(size=224, min_scale=0.75))
dls = pets.dataloaders(path/"images")
```

item_tfmts ve batch_tfms linelarını baz alarak:

These lines implement a fastai data augmentation strategy which we call *presizing*. Presizing is a particular way to do image augmentation that is designed to minimize data destruction while maintaining good performance.

Yani olay şu önce büyük bir size ile item_tfms uygularız böylece batch_tfms uygularken yapılacak intepolations daha kaliteli olur, eğer item_tfms'de 224 boyutunda olsaydı, image'I 45 derece rotate etmek istersek ne yapacaktık, boş kalan yerleri interpolate edecektik, ama kötü sonuçlar alırdık, önce büyük bir image elde edersek augmentation daha kaliteli olur.

Aşağıdaki satırla rastgele örneklere bakabiliriz, böylece datablock api ile doğru işlemler yapıp yapmadığımızı anlama fırsatımız olur.

```
dls.show_batch(nrows=1, ncols=3)
```

Ayrıca datablock api sonrasında summary methodunu çalıştırabiliriz, bu bize yapılan işlemleri detaylı bir şekilde gösterir ve bir hata varsa debug edebiliriz:

```
pets1 = DataBlock(blocks = (ImageBlock, CategoryBlock),
                 get_items=get_image_files,
                 splitter=RandomSplitter(seed=42),
                 get_y=using_attr(RegexLabeller(r'(.+)_\d+.jpg$'), 'name'))
pets1.summary(path/"images")
```

Modeli Hızlıca Eğit

Modeli eğitmeyi çok geciktirmek doğru değil, hızlıca basit bir model eğitip bir baseline performance elde etmek isteriz.

Zaten hatırlarsan, data cleaning yapmak için de bir model eğitmemiz gerek, aslında temel bir model eğiterek daha iyi bir model eğitme noktasında çok önemli bir adım atmış oluyoruz.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(2)
```

Yukarıda Loss Function Neden Belirtilmemiş?

Çünkü fastai dataset'e göre otomatik olarak sensible bir loss function seçer. Seçilen loss function'ı görmek için **learn.loss_func** diyebiliriz. Bu örnek için sonucun CrossEntropyLoss() çıktığını görürüz.

Activations and Labels

Aşağıdaki yapı ile dls içerisinde bir batch'i çekeriz ve x,y içerisine atarız. Dataloaders'ı oluştururken batch_size belirtilmediği için default olarak 64 alındı.

```
x,y = dls.one_batch()
```

O halde yukarıdaki yapıda x içerisinde 64 adet image tutacak, buna karşın y ise içerisinde 64 adet label tutacak. Burada label dediğimiz 0-36 arasında bir sayı, çünkü total 37 tane pet breed kategorisi var.

```
y
```

```
TensorCategory([ 0,  5, 23, 36,  5, 20, 29, 34, 33, 32, 31, 24, 12, 36,
8, 26, 30,  2, 12, 17,  7, 23, 12, 29, 21,  4, 35, 33,  0, 20, 26, 30,
3,  6, 36,  2, 17, 32, 11,  6,  3, 30,  5, 26, 26, 29,  7, 36,
        31, 26, 26,  8, 13, 30, 11, 12, 36, 31, 34, 20, 15,  8,  8, 23]
, device='cuda:5')
```

Her bir sayı bir breed'e karşılık geliyor, peki hangi sayı hangi breed'e karşılık geliyor? Bunu anlamak için:

```
dls.vocab
```

Yapısını kullanabiliriz, burada ilk isim 0'a, diğeri 1'e şeklinde giden bir correspondance vardır. Örneğin dls.vocab[16] diyerek 16 label'ına karşılık gelen pet preed ismini görebiliriz.

Şimdi aşağıdaki gibi learner'a bir dataloader vererek, prediction'ı görebiliriz. Verilen dataloader içerisinde 64 tane datapoint var, sonuçta 64 farklı prediction söz konusu. Ayrıca her bir prediction için model 37 farklı output üretiyor. Çünlü 37 farklı output unitimiz var, burada softmax kullanılacak ki üretilen outputların toplamı 1 etsin.

```
preds,_ = learn.get_preds(dl=[(x,y)])
preds[0]
```

Yukarıdaki gibi preds[0] dersek ilk datapoint için verilen 37 output'u görürüz, bunları toplarsak sonucun 1 yaptığını da görürüz, bunların arasından en büyüğü diyelim ki 17. Olsun o halde modelimiz ilk datapoint'i 17. Breed olarak tahmin etmiş demektir.

CrossEntropyLoss

Bunu eskiden de biliyoruz zaten temelde yapılan şey hatanın logaritmasını almak. Bunu daha convex bir cost function için yaptığımızı söylerdi Andrew.

Anladığım kadarıyla bunun uygulanışı, için softmax bile modele dahil değil, loss function içerisine dahil, yani model output olarak linear çıktıları üretiyor, sonrasında CrossEntropyLoss seçersek, çıktıların log_softmax'i alınıyor ki bu da önce softmax layerı koyar, sonra da sonuçların logaritmalarını alır, daha sonra da nll_loss uygulanır. Bu da anladığım kadarıyla sadece sonucun positive veya negative oluşuna göre log sonuçlarını seçer ve toplar.

Taking the mean of the positive or negative log of our probabilities (depending on whether it's the correct or incorrect class) gives us the *negative log likelihood* loss. In PyTorch, `nll_loss` assumes that you already took the log of the softmax, so it doesn't actually do the logarithm for you.

When we first take the softmax, and then the log likelihood of that, that combination is called *cross-entropy loss*. In PyTorch, this is available as `nn.CrossEntropyLoss` (which, in practice, actually does `log_softmax` and then `nll_loss`):

```
loss_func = nn.CrossEntropyLoss()
```

```
loss_func(acts, targ)
```

```
tensor(1.8045)
```

Model Interpretation

Modelimizi interpret etmek için doğrudan loss function'a bakmak çok anlamlı olmaz, çünkü loss function'ın asıl amacı SGD'nin verimli çalışmasıdır, bizim için anlamlı olacak şey ise metrics tir.

Örneğin aşağıdaki yapı ile önce learnerdan bir interpretation objesi yaratırız, daha sonra bu interp objesini kullanarak confusion matrix'i bastırabiliriz, bu matrix 37x37 bir matrix olacaktır çok iyi görünmeyecektir ama yine de bakmakta fayda var.

```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix(figsize=(12,12), dpi=60)
```

Class sayımız fazla olduğunda confusion matrix'ten daha iyi bir interpretation yöntemi mos_confused kullanmaktır bu method bize en fazla karıştırılan n tane tahmini verir. Yani örneğin model "stafford" tipli köpeklerden 10 tanesini "pit_bull" olarak tahmin etmiş yani 10 kere confused olmuş. O halde bunu en üstte görürüz, yani model hangi class'ı bir başka class'la sık sık karıştırmış.

Aşağıdaki satır ile modelin en çok karıştırdığı 5 class'ı, kaç kere karıştırdığını ve gerçek sınıflarını görebiliriz.

```
interp.most_confused(min_val=5)
```

Improving Our Model

Learning Rate

İlk olarak improve etmek istediğimiz şey, learning rate'imiz olacak. Bunu neden yapmak isteriz?

- Çünkü iyi bir learning rate ile daha hızlı ve converged bir eğitim yakalayabiliriz.
- Fine tune methodu default olarak base_lr = 0.002 kullanıyor, buna shift tab ile method içerisinden bakabiliriz.

Şimdi biz gidip learning rate 'i 5 kat artıralım, bakalım ne olacak:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1, base_lr=0.1)
```

epoch	train_loss	valid_loss	error_rate	time
0	2.778816	5.150732	0.504060	00:20

epoch	train_loss	valid_loss	error_rate	time
0	4.354680	3.003533	0.834235	00:24

Gördüğümüz gibi error rate bariz biçimde arttı, lr çok fazla olduğu için, gradients step converge edemedi diverge ediyor, training'e devam etsek muhtemelen artmaya devam edecek.

Bu sebeple doğru learning rate seçmek training için crucial, isteriz ki modeli diverge etmeyecek veya global minumum etrafında back and forth zıplatmayacak doğrudan büyük adımlarla global minumum'a converge ettirecek büyüklükte bir lr seçmek isteriz.

Bunu yapmak için learning rate finder'ı kullanacağız.

Learning Rate Finder

Learning rate finder'ı aşağıdaki gibi çalıştırıyoruz. Cnn_learner oluşturulduğunda otomatik olarak, body kısmı freezed olarak oluşturulur.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
lr_min,lr_steep = learn.lr_find()
```

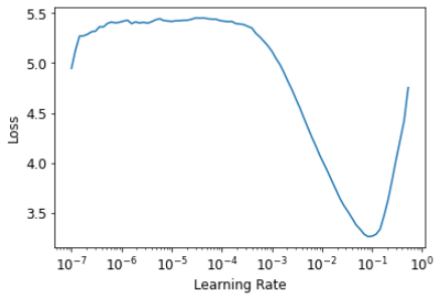
Yani learning rate'i bulurken, freezed architecture'a bakarız. Lr finder'ın çalışma mantığı şu, her minibatch için learning rate'i artırır ve eğitimi bu şekilde gerçekleştirir, diyelim ki her minibatch için lr değeri %25 artacak. İlk başta çok küçük bir lr seçilir, dolayısıyla başlarda loss çok improve etmez, bir noktadan sonra loss improve edecek, minimum'a ulaştıktan sonra ise fırlayıp gidecek.

İşte biz loss uçmadan önceki değerlerle ilgileniyoruz, şekilde bariz bir yokuş aşağı düşüş varsa, en yüksek slope'un olduğu noktayı lr olarak seçeriz, şekil stable gidip fırlıyorsa fırlamadan 10x falan öncesini alırız.

Learning Rate Finder Devam:

Freezed network için lr_finder çalıştırınca graph aşağıdaki gibi çıktı, burada steepest point 5*10^-3 olarak görülüyor.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
lr_min,lr_steep = learn.lr_find()
```




```
print(f"Minimum/10: {lr_min:.2e}, steepest point: {lr_steep:.2e}")
```

```
Minimum/10: 1.00e-02, steepest point: 5.25e-03
```

Burada 3e-3 değeri stage1 eğitimi için seçilmiş:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(2, base_lr=3e-3)
```

epoch	train_loss	valid_loss	error_rate	time
0	1.328591	0.344678	0.114344	00:20

epoch	train_loss	valid_loss	error_rate	time
0	0.540180	0.420945	0.127876	00:24
1	0.329827	0.248813	0.083221	00:24

Ama bak hatırlarsan eskiden yöntem şuydu, önce stage 1 için fit_one_cycle ile eğitim yapılırdı, burada sadece head eğitileceği için lr_finder'dan bulunan learning rate değeri tek başına kullanılabilirdi. Daha sonra stage 2 için lr finder tekrar çalıştırılırdı, yeni değer

bulunurdu, bu kez fit_one_cycle'a lr olarak bir slice verilirdi, slice'ın ilk elemanı stage2 için bulunan yeni lr olurdu, diğeri ise stage1 lr'ının 5x veya 10x gerisi olurdu, böylece head kısmı stage1'in 10'da 1'i bir lr ile fine_tune edilirdi, ayrıca body kısmı da bulunan yeni lr ile fine_tune edilirdi.

Yukarıda ise, stage1 için bulunan lr ile fine_tune methodu çağırılmış ki bu method aslında içinde hem stage1 hem stage2'yi içeriyor:

```
learn.fine_tune??
```

```
Source:
@patch
@log_args(but_as=Learner.fit)
@delegates(Learner.fit_one_cycle)
def fine_tune(self:Learner, epochs, base_lr=2e-3, freeze_epochs=1, lr_mult=100,
              pct_start=0.3, div=5.0, **kwargs):
    "Fine tune with `freeze` for `freeze_epochs` then with `unfreeze` from `epochs` using discriminative LR"
    self.freeze()
    self.fit_one_cycle(freeze_epochs, slice(base_lr), pct_start=0.99, **kwargs)
    base_lr /= 2
    self.unfreeze()
    self.fit_one_cycle(epochs, slice(base_lr/lr_mult, base_lr), pct_start=pct_start, div=div, **kwargs)
```

Olay şu zaten learner freeze olacak ama yine de tekrar freeze edilmiş, sonra fit_one_cyle ile bulunan stage1 lr'ı kullanılarak head kısmı initial eğitimini tamamlıyor.

what `fit_one_cycle` does is to start training at a low learning rate, gradually increase it for the first section of training, and then gradually decrease it again for the last section of training.

Daha sonra stage2'ye otomatik olarak geçiyor, ancak stage2 için head kısmı stage1 lr'ının yarısı ile eğitilirken, body kısmı ise 100'de 1'i gibi çok küçük değerlerle eğitiliyor. Yani burada bir stage2 lr'ı bulunmadan işlemler yapılmış.

Kendi Fine Tune Yöntemimizi Yazalım:

Alternatif bir yöntem şöyle olabilirdi, fine_tune methodunu kendimiz uygulayabilirdik eskisi gibi, önce default olarak freezed network, stage 1 lr'ı ile eğitilir, yani burada head initial eğitimini alıyor.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(3, 3e-3)
```

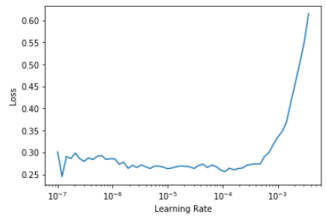
Daha sonra model unfreeze edilir, stage 2 için yeni learning rate bulunur, bu learning rate tüm network'ü eğitmek için kullanılacak learning rate'dir aşağıda 1e-5 noktası stage 2 lr olarak seçilir.

```
learn.unfreeze()
```

and run `lr_find` again, because having more layers to train, and weights that have already been trained for three epochs, means our previously found learning rate isn't appropriate any more:

```
learn.lr_find()
```

```
(1.0964782268274575e-05, 1.5848931980144698e-06)
```



Ardından fit_one_cycle ile tüm model aşağıdaki gibi eğitilir, burada slice'ın ilk parametresi ilk layer'ların eğitileceği değer olacak bu yeni lr baz alınarak seçilmeli burada biraz az seçilmiş sanki, ikinci parametre ise stage1 değerinin 5x veya 10x gerisi olarak seçilir.

```
learn.fit_one_cycle(12, lr_max=slice(1e-6,1e-4))
```

Kendi fine_tune'umuz ile yani stage1 – stage2 yaklaşımıyla gayet iyi sonuçlar elde ediyoruz, unutma fit_one_cycle training'in yarısı boyunca lr'ı artırır, yani verilen slice içinde lr'lar aslında max lr'lar fit_one_cycle lr'ı azdan başlayarak önce artırır, sonra da düşürür, bu yüzden error_rate'in epoch epoch gelişimine baktığımızda sürekli bir azalma görmeyebiliriz ama sonuçta iyi sonuçlar elde ederiz.

Modeli Nasıl Daha İyi Yapabiliriz?

Yukarıda discriminative learning rates kavramı ile, yani farklı layerları farklı lr ile eğiterek, ve modeli seçilen lr değerleri ile fine_tune ederek gayet iyi performanslar elde ettik.

Daha iyi sonuçlar için daha complex network'lar denenebilir yani resnet18 yerine resnet34,50,101,152 kullanılabilir, burada GPU memory tükenebilir şayet böyle olursa kernel'i restart yapmamız gerek.

Kernel > Restart

Daha az gpu kullanmak için batch_Size'ı düşüerebiliriz, bunu DataLoaders oluşturuken bs parametresi ile yapabiliriz.

Bunun yanında fp16 kullanabiliriz, bunun yaptığı şey, daha az bit kullanan numbers ile çalışmak, yani örneğin 8 byte'lık double's kullanıyorsa 4 byte'lık float kullanmaya başlar bu da gpu'dan tasarruf eder.

```
from fastai.callback.fp16 import *
learn = cnn_learner(dls, resnet50, metrics=error_rate).to_fp16()
learn.fine_tune(6, freeze_epochs=3)
```

Modeli daha iyileştirmek ile ilgili şuan burada değinilmeyen bir diğer nokta ise, input size'ı küçük başlayıp adım adım artırmak, bunu ilk kursta yapmıştık, bu da iyi sonuçlar doğurabilir, gerekirse bak.