loss functions

under F

In [ ]:
```
F.l1_loss(a_3,mean7),  # mean absolute arror
F.mse_loss(a_3,mean7).sqrt() #mse, a_3, mean7 arrays of images hai
```

In [ ]:
```
F.nll_loss(sm_acts, targ, reduction='none')
```

In [ ]:
```
```

dls.classes give a dict with all entries of each class present (user and rating for movies)

In [ ]:
```
```

Generally it's a good idea to specify `low_memory=False` unless Pandas actually runs out of memory and returns an error. The `low_memory` parameter, which is `True` by default, tells Pandas to only look at a few rows of data at a time to figure out what type of data is in each column. This means that Pandas can actually end up using different data type for different rows, which generally leads to data processing errors or model training problems later.

In [ ]:
```
df = pd.read_csv('Tabular_Data/TrainAndValid.csv', low_memory=False)
```

In [ ]:
```
```

observe how to change the values in df using .loc

In [ ]:
```
xs.loc[xs['YearMade']<1900, 'YearMade'] = 1950
```

In [ ]:
```
```

In [ ]:
```
```

set as ordered categories

In [ ]:
```
sizes = 'Large','Large / Medium','Medium','Small','Mini','Compact'
type(sizes) # tuple!
df['ProductSize'] = df['ProductSize'].astype('category')
df['ProductSize'].cat.set_categories(sizes, ordered=True, inplace=True)
```

In [ ]:
```
```

Handling Dates

provide categorical data that we *suspect* will be useful.

In [ ]:
```
df = add_datepart(df, 'saledate')
# 'saleYear , saleMonth , saleWeek , saleDay , saleDayofweek , saleDayofyear ,
# saleIs_month_end , saleIs_month_start , saleIs_quarter_end , saleIs_quarter_start ,
# saleIs_year_end , saleIs_year_start , saleElapsed'
```

In [ ]:

```
```

Saving

Python's pickle system to save nearly any Python object:

In [ ]:

```
# say to is tabularpandas
save_pickle('Tabular_Data/to.pkl',to)
# To read this back later, you would type:
to = (path/'to.pkl').load()
```

In [ ]:

```
```

nunique() is used to find number of unique elements for each col in df

two variables (features) having similar description, both with similar very cardinalities, suggests that they may contain similar, redundant information.

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

Learning Rate finder

His idea was to **start with a very, very small learning rate,** something so small that we would never expect it to be too big to handle. We use that for one mini-batch, find what the losses are afterwards, and then increase the learning rate by some percentage (e.g., doubling it each time). Then we do another mini-batch, track the loss, and double the learning rate again. We keep doing this until the loss gets worse, instead of better. This is the point where we know we have gone too far. We then select a learning rate a bit lower than this point. Our advice is to pick either:

- **One order of magnitude less than where the minimum loss was achieved (i.e., the minimum divided by 10)**
- **The last point where the loss was clearly decreasing**

In [ ]:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
lrs = learn.lr_find(suggest_funcs=(minimum, steep, valley, slide))
lrs
# SuggestedLRs(minimum=0.00831763744354248, steep=0.005248074419796467,
# valley=0.0008317637839354575, slide=8.31763736641733e-06)
```

In [ ]:

```
<img src="images/learning_rate.png" width="400">
```

We can see on this plot that in the **range 1e-6 to 1e-3, nothing really happens and the model doesn't train. Then the loss starts to decrease until it reaches a minimum, and then increases again.** We don't want a learning rate greater than 1e-1 as it will give a training that diverges like the one before (you can try for yourself), **but 1e-1 is already too high: at this stage we've left the period where the loss was decreasing steadily.**

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

unfreezing, transfer learning

`fit_one_cycle` is the suggested way to train models without using `fine_tune` . 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.

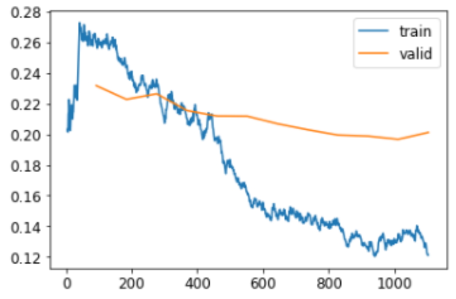
In [ ]:
```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(3, 3e-3)
learn.unfreeze()
# because having more layers to train, and weights that have already been trained
# for three epochs, means our previously found learning rate isn't appropriate any more:
learn.lr_find()
learn.fit_one_cycle(6, lr_max=1e-5)
```

discriminative learning rates

The deepest layers of our pretrained model might not need as high a learning rate as the last ones, so we should probably use different learning rates for those

In [ ]:
```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(3, 3e-3)
learn.unfreeze()
learn.fit_one_cycle(12, lr_max=slice(1e-6,1e-4))
# epoch train_loss      valid_loss      error_rate      time
# 0     1.147984        0.335110        0.105548        01:41...
# 2     0.344369        0.238139        0.079838        01:40
# epoch train_loss      valid_loss      error_rate      time
# 0     0.259971        0.231559        0.077808        02:12...
# 11    0.121182        0.201167        0.060217        02:12
```

In [ ]:
```
learn.recorder.plot_loss()
```



As you can see, the training loss keeps getting better and better. But notice that eventually the validation loss improvement slows, and sometimes even gets worse! **This is the point at which the model is starting to over fit.** In particular, the model is becoming overconfident of its predictions.

In [ ]:
```
from fastai.callback.fp16 import *
```

```
learn = cnn_learner(dls, resnet50, metrics=error_rate).to_fp16()
learn.fine_tune(6, freeze_epochs=3)
# epoch train_loss      valid_loss      error_rate      time
# 0     1.300184        0.307617        0.100812        02:21...
# 2     0.414568        0.290107        0.091340        02:21
# epoch train_loss      valid_loss      error_rate      time
# 0     0.287533        0.269856        0.085250        03:10 ...
# 5     0.059884        0.203740        0.053451        03:10
```

In [ ]:

```

```

weight decay

technique of normalisation

when overfitting, and data isn't images

type of jo andrew ng wale course me tha loss func

Weight decay, or L2 regularization, consists in adding to your loss function the sum of all the weights squared. Why do that? Because when we compute the gradients, it will add a contribution to them that will encourage the weights to be as small as possible.

In [ ]:

```
learn.fit_one_cycle(5, 5e-3, wd=0.1) # yaha hai
```

In [ ]:

```

```

argsort(x)

Returns the indices that would sort an array.

In [ ]:

```
x = np.array([3, 1, 2])
np.argsort(x)
# array([1, 2, 0])
```

In [ ]:

```

```

normalisation

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.

`Normalize` transform. 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

normalization becomes especially important when using pretrained models. The pretrained model only knows how to work with 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 to what is intended!

if you're using a model that someone else has trained, make sure you find out what normalization statistics they used, and match them.

In [ ]:

```
def get_dls(bs, size):
    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)])
    # only extra normalise function is added
    return dblock.dataloaders(path, bs=bs)
```

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.

In [ ]:
```
dls = get_dls(32, 128)  #size is 128 of images
```

In [ ]:
```
learn = Learner(dls, xresnet50(n_out=dls.c), loss_func=CrossEntropyLossFlat(),
                metrics=accuracy)
learn.fit_one_cycle(4, 3e-3)
epoch   train_loss      valid_loss      accuracy        time
# 0     1.568923        2.229021        0.414862        02:18 ...
# 3     0.669132        0.597463        0.811426        02:18
```

In [ ]:
```
# SIZE BADHA DOYA
learn.dls = get_dls(32,224)
learn.fine_tune(5, 1e-3)
# epoch train_loss      valid_loss      accuracy        time
# 0     0.742987        0.741256        0.758028        05:46
# epoch train_loss      valid_loss      accuracy        time
# 0     0.621787        0.571918        0.823002        05:45 ...
# 4     0.430894        0.405990        0.869679        05:46
```

In [ ]:
```
```

test time augmentation

random cropping - can often be problematic - it's possible that some critical feature necessary for identifying the correct breed, such as the color of the nose, could be cropped out.

avoid random cropping entirely - **creating multiple versions of each image, using data augmentation, and then taking the average or maximum of the predictions for each augmented version of the image.**

In [ ]:
```
preds,targs = learn.tta()
accuracy(preds, targs).item()
```

TTA gives us good a boost in performance, with no additional training required. However, it does make inference slower—if you're averaging five images for TTA, inference will be five times slower.

In [ ]:
```
```

mixup

is a very 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**

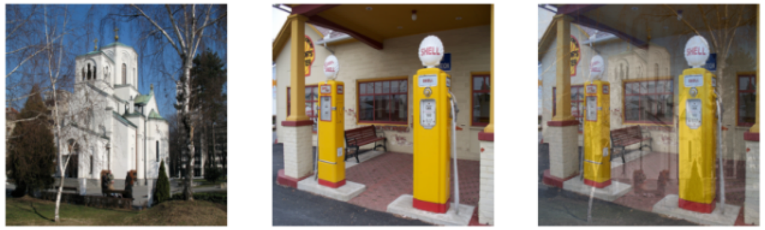
```
image2,target2 = dataset[randint(0,len(dataset)] #taking one at random
t = random_float(0.5,1.0)
new_image = t * image1 + (1-t) * image2
new_target = t * target1 + (1-t) * target2 #ye to interpolation type se dikh raha hai
```

For this to work, our targets need to be one-hot encoded.

In [ ]:

```python
church = PILImage.create(get_image_files_sorted(path/'train'/'n03028079')[0])
gas = PILImage.create(get_image_files_sorted(path/'train'/'n03425413')[0])
church = church.resize((256,256))
gas = gas.resize((256,256))
tchurch = tensor(church).float() / 255.
tgas = tensor(gas).float() / 255.

_,axs = plt.subplots(1, 3, figsize=(12,4))
show_image(tchurch, ax=axs[0]);
show_image(tgas, ax=axs[1]);
show_image((0.3*tchurch + 0.7*tgas), ax=axs[2]);
```



The third image is built by adding 0.3x first + 0.7x second. model predict church or gas station -- 30% church and 70% gas station, since that's what we'll get if we take the linear combination of the one-hot-encoded targets For instance, suppose we have 10 classes and "church" is represented by the index 2 and "gas station" is reprsented by the index 7, the one-hot-encoded representations are:

```
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0] and [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
```

so our final target is:

```
[0, 0, 0.3, 0, 0, 0, 0, 0.7, 0, 0]
```

This all done for us inside fastai by adding a **callback** to our Learner

In [ ]:

```python
model = xresnet50(n_out=dls.c)
learn = Learner(dls, model, loss_func=CrossEntropyLossFlat(),
                metrics=accuracy, cbs=MixUp()) #yaha!!!!!!!!!
learn.fit_one_cycle(5, 3e-3)
```

harder to train, requires far more epochs to train to get better accuracy it can be applied to types of data other than photos. In fact, some people have even shown good results by using Mixup on activations inside their models, not just on inputs—this allows Mixup to be used for NLP and other data types too.

One issue with this, however, is that Mixup is "accidentally" making the labels bigger than 0, or smaller than 1. a way to handle this more directly is to use label smoothing.

In [ ]:

```

```

In [ ]:

```

```

Label Smoothing

Instead, we could replace all our 1s with a number a bit less than 1, and our 0s by 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.

In [ ]:

```
model = xresnet50(n_out=dls.c)
learn = Learner(dls, model, loss_func=LabelSmoothingCrossEntropy(),
                metrics=accuracy)
learn.fit_one_cycle(5, 3e-3)
```

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

In [ ]:

```

```