Öncelikle multiclassification'a bakacağız.

- Ancak burada kasıt, sadece birden fazla class olması değil. Bir örneğin birden fazla class'a dahil olabilmesinden bahsediyoruz.
- Yani bir örnek için birden fazla label söz konusu.
- Bu durumda, daha önceki örnekte olduğu gibi her class'ı farklı bir klasöre kaydetmek fikri işlevsiz kalacak. Bunun yerine her örneğin labelları bir csv file içinde tutulabilir.
- Kaggle örneğinde de görülen bu: https://www.kaggle.com/yildizer/fast-ai-v3-lesson-3-planet/edit

1.Devam etmeden önce, verileri nasıl fastai'ın kullanabileceği formata çevirebileceğimizi daha iyi kavramaya çalışalım:

- Kaggle planet örneği için labeling aşağıdaki gibi bir csv file yardımı ile yapılmış.

```
df = pd.read_csv(path/'train_v2.csv/train_v2.csv')
df.head()
```

:

	image_name	tags
0	train_0	haze primary
1	train_1	agriculture clear primary water
2	train_2	clear primary
3	train_3	clear primary
4	train_4	agriculture clear habitation primary road

- Bir model eğitebilmek için öncelikle böyle bir csv file içinde tutulan bilgi'den fastai'ın kullanabileceği bir dataform elde etmeliyiz.
- Fastai için model eğitiminde kullanılan dataform'u ise DataBunch+ formu.İlgili image'lardan bir databunch elde ettikten sonra diğer işlemleri yapmak gayet kolay olacaktır.

- Şimdiye kadar, bu databunch'ı oluşturmak için factory methods kullandık, şu source'dan şu options ile şu type bir data elde etmek istiyorum demiştik. ImageDataBunch methodu ile yapmıştık bunu sanırım.
- Bu factory methods genelde iyi çalışır. Ancak bazı durumlarda, we want more flexibility.

- Çünkü bu süreçte yapılması gereken çok seçim var, source nerede, nasıl bir yapıda, labels nasıl bir yapıda, validation set'i nasıl ayıracağız, transofrmation'ı nasıl yapacağız vesaire…
- İşte bu yüzden fastai data block API'ı kulanıyor, bu API ile yukarıda bahsedilen seçimlerin hepsini ayrı ayrı yapabiliyoruz. Yani dataset'imi oluşturma aşamasında her türlü seçimi yapmama olanak sağlıyor.

```
tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0.)
```

We use parentheses around the data block pipeline below, so that we can use a multiline statement without needing to add '\'.

```
np.random.seed(42)
src = (ImageItemList.from_csv(path, 'train_v2.csv', folder='train-jpg', suffix='.jpg'
       .random_split_by_pct(0.2)
       .label_from_df(sep=' '))
```

```
data = (src.transform(tfms, size=128)
        .databunch(num_workers=0).normalize(imagenet_stats))
```

- Örneğin kaggle'daki planet datasını elde edebilmek için yukarıda görülen işlemleri yapıyoruz.
- Önceki sayfada görülen train_v2.csv şeklinde bir ImageItemList'imiz var bu dosya için path'i verdik, .from_csv ile csv'den okuma yapacağımızı belirttik, image'ların folderını ve suffix'ini belittik, random olarak %20'sini validation set olarak al dedik, csv file'da kullanılan seperator'ı yani space'i belirttik. Böylece bir src elde ettik.
- Daha sonra bu src'a yukarıda belirtilen transform'u uyguladık, daha sonra bunlardan bir databunch elde ettik ve son olarak dataset2i ilgili statistics'i kullanarak normalize ettik.

Şimdi fastai'ın kullanabileceği veri yapısını oluşturmak kısmını daha iyi kavrayabilmek için öncelikle, bilmemiz gereken fastai ve pytorch class'larından bahsedeceğiz. Burada bahsedilecek yapıları sürekli göreceğiz ve bunları kavradığımız zaman, istediğimiz yapıdaki datasetimizi fastai için kullanılır hale getirebiliriz.

İlk bakacağımız class Dataset class'ı

```
class Dataset(object):
    """An abstract class representing a Dataset.

    All other datasets should subclass it. All subclasses should override
    ``__len__``, that provides the size of the dataset, and ``__getitem__``,
    supporting integer indexing in range from 0 to len(self) exclusive.
    """

    def __getitem__(self, index):
        raise NotImplementedError

    def __len__(self):
        raise NotImplementedError
```

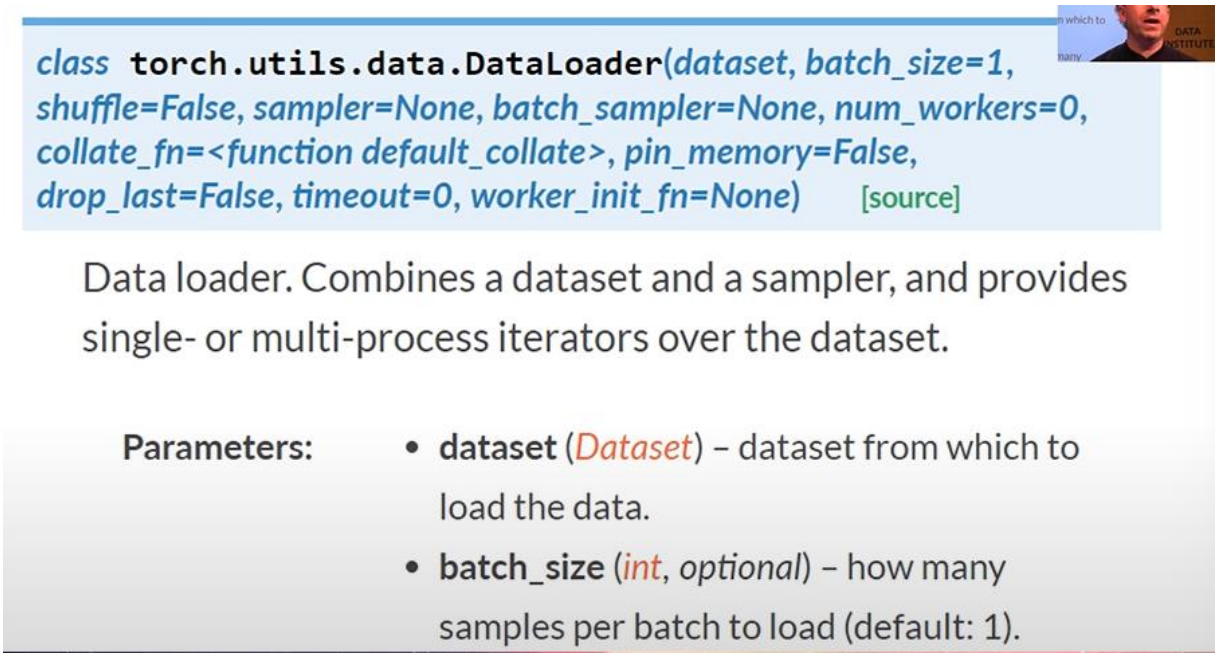
- Bu class pytorch'un bir parçası. Gördüğümüz gibi bu class'ın yaptığı bir iş yok.Çünkü bu bir abstract class. Diper bütün subclass'ların bu methodları override etmesi gerektiğini söylüyor.
- Bu class'ın iki tane methodu olduğunu görüyoruz, __method__ yapısına python'da "dunder"? deniyor, bunlar special magical methods that do some special behaviour.
- __getitem__'ın sağladığı şey, yaratılan dataset objesini "[ ]" ile indexleyebilmek, yani bir dataset objesi d1'i d1[3] yapısını kullanarak indexleyebiliriz.
- İkinci method __len__ ise obje d1'in lenght'ini döndürmeyi sağlar. Yani len(d1) dediğimizde length dönecektir.
- Görüldüğü gibi bu methodlar abstract class için tanımlı değil. Bunların subclass'larda tanımlanması bekleniyor.

Fastai bir çok dataset subclass'a sahip her biri başka çeşit datalar için özelleşmiş. Ancak hepsi bu abstract class'tan türediği için hepsi indexlenebilir ve hepsinin uzunluğu var.

Önceki haftalarda image classification için özelleşmiş datasets'i gördük, bu datasets için getitem ile istediğimiz image'ı ve label'ı return edebiliyor, veya datasetimizin büyüklüğünü döndürebiliyorduk.

Ancak bir dataset objesi yaratmak bir model train edebilmemiz için yeterli değildir. Bu yapının yanında bize gereken bir diğer şey şu, tek bir seferde birden fazla items'ı(images) elde edebilmeliyiz ki GPU'umuz paralel olarak farklı işleri yürütebilsin. Yani bir minibatch yaratabilmemiz lazım.

Bir minibatch yaratabilmek için DataLoader adındaki bir başka pytorch class'ından yararlanırız.



class **torch.utils.data.DataLoader**(*dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, collate_fn=<function default_collate>, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None*) [source]

Data loader. Combines a dataset and a sampler, and provides single- or multi-process iterators over the dataset.

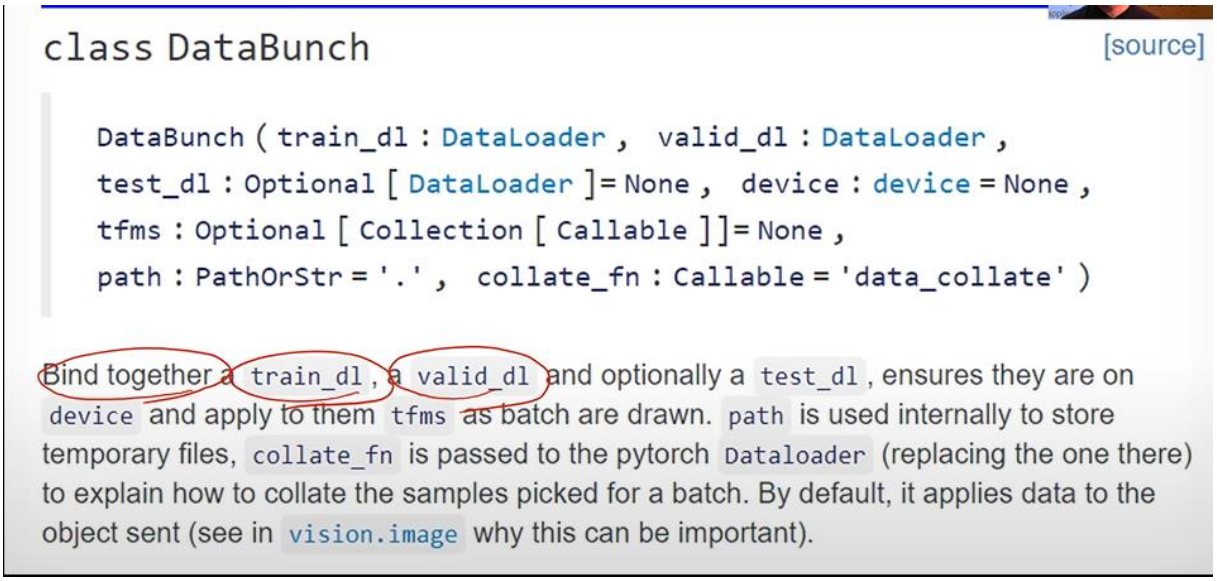
Parameters:

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int*, *optional*) – how many samples per batch to load (default: 1).

- Dataloader constructor'ına bir dataset alır, girilen batch_size'a göre bu dataset içindeki itemları rastgele olarak alır ve bir minibatch oluşturur.
- DataLoader is something that grabs individual items, combines them into a mini batch pops them on the GPU for modeling.

Ancak bunlar hala bir model eğitmek için yeterli değil. Çünkü modelimizi validate(doğrulamak) için bir yolumuz yok. Elimizde sadece training set varsa modelimizin iyi mi kötü mü olduğunu doğrulayamayız.

Bu sebeple, bir üst katmana çıkıp bir başka class tanımlıyoruz ki bu da bir fastai class'ı olan DataBunch class'ı.



class DataBunch [source]

```
DataBunch ( train_dl : DataLoader ,  valid_dl : DataLoader ,
test_dl : Optional [ DataLoader ]= None ,  device : device = None ,
tfms : Optional [ Collection [ Callable ]]= None ,
path : PathOrStr = '.' ,  collate_fn : Callable = 'data_collate' )
```

Bind together a `train_dl`, a `valid_dl` and optionally a `test_dl`, ensures they are on `device` and apply to them `tfms` as batch are drawn. `path` is used internally to store temporary files, `collate_fn` is passed to the pytorch `Dataloader` (replacing the one there) to explain how to collate the samples picked for a batch. By default, it applies data to the object sent (see in `vision.image` why this can be important).

- Yukarıda görüldüğü gibi DataBunch class'ının yaptığı iş temelde bir training dataloader'ı ile validation dataloader'ını bind etmek.

- Bu seviyeye geldiğimizde yani ham verimizden bir DataBunch elde edebildiğimizde artık bu veriyi bir learner'a gönderebiliriz yani bir model eğitmeye başlayabiliriz demektir.

2.Model eğitebilmek için ham veriyi databunch'a çevirmemiz gerektiğini anladık:

- Şimdi data block API kısmına geri dönersek:

```
tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0.)
```

We use parentheses around the data block pipeline below, so that we can use a multiline statement without needing to add '\'.

```
np.random.seed(42)
src = (ImageItemList.from_csv(path, 'train_v2.csv', folder='train-jpg', suffix='.jpg
       .random_split_by_pct(0.2)
       .label_from_df(sep=' '))
```

```
data = (src.transform(tfms, size=128)
        .databunch(num_workers=0).normalize(imagenet_stats))
```

- Yukarıda oluşturulan src aslında bir dataset. Dataset indexleri içinde image'ı ve label'ı içerir. Bu dataset'i oluşturmak için yukarıdaki gibi image'ların nerede olduğunu, hangi image'a hangi labels'ın karşılık geldiğini gibi şeyleri söyledik ve sonucunda src isminde bir image dataset elde ettik. Tabi burada random_split ile aslında 2 farklı dataset elde etmiş oldum, training ve validation.
- Daha sonra bu src dataset'ini belirtilen şekilde transform edip, .databunch ile tek seferde hem dataloaders'ı hem de databunch'ı elde etmiş olduk.

3. Şimdi bu data block API'ı daha iyi anlamak için bazı örneklere bakalım, çünkü bu API kısmını kavradığımız zaman, istediğimiz dataset'i nasıl databunch'a çevirebileceğimizi kavramış olacağız.

Bu örnekleri fastai documentation > data block üzerinden inceleyebiliriz. https://docs.fast.ai/data_block.html

- `data = ImageDataBunch.from_folder(path, ds_tfms=tfms, size=64)`
- İlk örneğe baktığımız zaman, MNIST_TINY datasetini görüyoruz, yani el hand written digits. Bu dataset'i yükledikten sonra, doğrudan yukarıdaki factory method ile databunch objemizi oluşturabiliyorduk. Ancak bu methodun çalışması için datasetimiz ImageNet style hazırlanmış olmalı.

- ImageNet style, with the `train` and `valid` directories, each containing one subdirectory per class, where all the labelled pictures are.
- Biz bu işlemi datablock API ile de yapabiliriz, üstelik datablock API ile sadece bu tipteki datasetleri değil, her türlü dataset'I databunch'a çevirebilriz.

Aşağıda gerekli kodu görüyoruz, öncelikle dataset'in list of image files olduğunu belirtiyoruz. Bu yüzden ilgili path'deki bir folder içerisinde bulunan datayı anlatıyoruz: ImageList.from_folder(path)

Ulaşılan datanın nasıl split edileceğini belirtiyoruz bu durumda by folder name.

Ulaşılan datanın nasıl label edileceğini belirtiyoruz bu da bu durumda by folder name.

Optional olarak test set ekleyebiliyoruz.

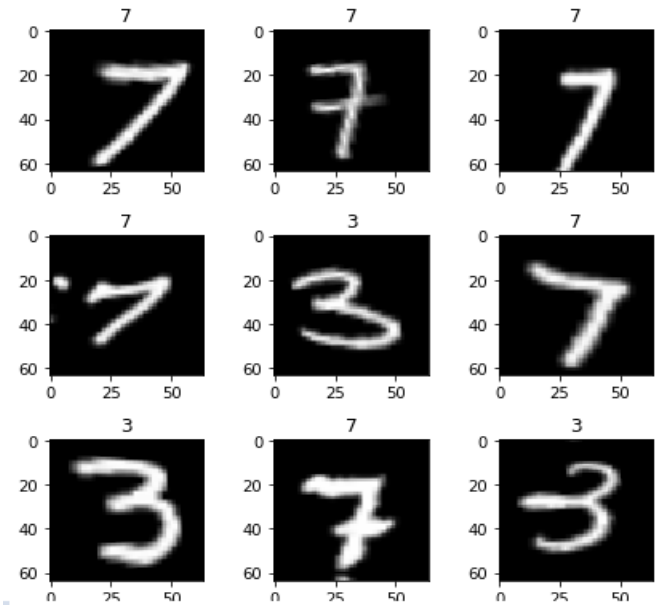
Daha sonra daha önce tfms olarak belirtilen transformations'ı uygula diyoruz, bu işlem data augmentation için kullanılabilir, ayrıca image size'ını vesaire de burada ayarlayabiliyoruz.

Son olarak bu bilgilere göre alınan datayı databunch'a çeviriyoruz.

```
data = (ImageList.from_folder(path) #Where to find the data? -> in path and its subfolde
rs
        .split_by_folder()              #How to split in train/valid? -> use the folders
        .label_from_folder()            #How to label? -> depending on the folder of the
filenames
        .add_test_folder()              #Optionally add a test set (here default name is
test)
        .transform(tfms, size=64)       #Data augmentation? -> use tfms with a size of 6
4
        .databunch())                   #Finally? -> use the defaults for conversion to
ImageDataBunch
```

Now we can look at the created DataBunch:

```
data.show_batch(3, figsize=(6,6), hide_axis=False)
```



Normalde yukarıdaki methodların içine bazı parametreler vererek, bu databunch oluşturma işini özelleştirebiliyoruz ama MNIST dataset için default values yeterli olacaktır.

Sonuçta elde edilen “data” bir dataBunch objecttir.

- data.train_ds[0] diyerek training set’in ilk elemanına ulaşabilirim bu bana ilk image’ı ve karşılık gelen label’ı döndürecektir.
- Veya yukarıdaki gibi data.show_batch() ile rastgele örnekleri visualize edebilirim.
- data_valid_ds.classes ile ilgili dataset’in classlarını return edebilirim.

Aynı linkin devamında başka data örnekleri için data block API’ın nasıl kullanıldığı açıklanmış gerekirse incele. Örneğin CAMVID dataset’i için labeling bir function kullanarak yapılmış vesaire, ama temel mantık aynı ve documentationda bilgiler yer alıyor.

4. Şimdi kaggle planet örneği ile yani v3 lesson 3 ile devam edelim, bu kodu kaggle'da çalıştıramadım şimdilik açıklamaları izleyeceğim ve buraya not edeceğim.

Öncelikle transform'dan bahsedeceğiz. Methoda shift+tab uzun basarsak detayları görürüz.

tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0.)

Transform methodu farklı işler yapar, örneğin random olarak bazı image'ları horizontally flip eder. Yukarıdaki methodda flip_vert=True dedik çünkü uydu fotoğrafları için vertical flip de yapılabilir, kedi köpek vesaire için bu mantıksız olurdu.

Daha bir çok özelleştirme yapılabilir, detaylarına bakabilirsin bu setting planet datası için iyi çalışıyor.

Enteresan noktalar biri max_warp=0 kısmı. Perspective warping'i fastai dışında uygulayan çok library yok. Çünkü genelde yavaş olur. Bunun yaptığı şey. Resmi farklı perspektiflerden çekilmiş gibi çoğaltıyor. Yani ben kediyi karşıdan çekince farklı görünecek yukardan farklı, aşağıdan farklı. Bu özellik sayesinde bu değişimleri hesaba katabiliyoruz. Tabi bu özellik uydu fotoğrafları için gerekli değil.

Sonuçta transform ile data augmentation yapabiliyoruz. Bundan daha sonra bahsedilecek.

Planet datasından dataBunch'ı elde ettikten sonra, kalan işlemler lesson1'e çok benzer, temelde bir model eğiteceğiz o kadar. Burada farklı olan tek şey, multi-label classification yaptığımız için, farklı metrics kullanacağız. Önceden tek bir unit hot olabiliyordu bu yüzden en yüksek çıkış vereni 1 kabul ediyorduk gerisini 0 kabul ediyorduk. Ancak burada durum biraz farklı, bir treshold koyacağız (default 0.5) ve bu treshold'un üstünü veren outputs'u 1 kabul edeceğiz böylece birden fazla output'a 1 olma şansı veriyoruz.

Ayrıca competition için Fbeta metrics kullanılıyor, bu f_score oluyor sanırım, temelde amacı, bildiğin gibi precision-recall vb metricleri birleştirip tek bir metric elde etmek.

Öncelikle architecture'ı tanımlıyorum:

arch = models.resnet50

Ayrıca create_cnn içinde farklı2 ayrı metrics tanımlıyorum ***metrics=[acc_02, f_score]*** şunu unutma, metrics'in model performansına hiçbir etkisi yok, yani metriği değiştirerek modeli daha iyi veya kötü hale getiremeyiz metric sadece bizim eğitim sürecini algılayabilmemizi sağlıyor.

Burada iki metrik tanımladık çünkü accuracy'e kendim bakmak istiyorum, f_score'a ise bakıyorum çünkü kaggle competition bu kriteri baz alacağını söylemiş.

Ancak dediğim gibi buradaki accuracy'de biraz farklı, çünkü multi-label söz konusu. Önceki örneklerde olan şuydu, mesela 10 class'lı bir digit classification yapıyorsak, modelin 10 tane outputu oluyordu, bunların sadece bir tanesi hot olabileceği için argmax ile en yükseğini buluyorduk ve bunu 1 kabul ediyorduk. Daha sonra bu 10'luk vektörde ground truth'u yani ilgili label'ı karşılaştırıp accuracy hesabı yapıyorduk.

acc_02 = partial(accuracy_thresh, thresh=0.2)

f_score = partial(fbeta, thresh=0.2)

learn = create_cnn(data, arch, metrics=[acc_02, f_score], model_dir='/tmp/models')

Fakat multi-label classification durumu için, önceki accuracy yöntemini kullanamayız, bunun yerine başka bir accuracy hesabı tanımlayacağız.

Daha önceden hatırlayabileceğimiz gibi, databunch objesinin data.c isminde bir attribute'u vardı. Temelde bu attribute modelin kaç outputu olduğunu gösteriyordu. Yani bir classifier için, data.c attribute'u class sayısına eşit olacaktır. Yani len(data.classes) ile data.c birbirine eşit olacaktır. Örneğin 10 digit classifier için 10 adet output unit olacak her bir unitin çıkışı 10 digitten birinin probability'sini temsil edecek.

Ancak multi-class classification için, data.c'den sadece bir tanesi hot olmayacak birden fazla hot olabilir. Bu yüzden, her bir output'u bir treshold ile karşılaştıracağız ve eğer tresholdu geçerse, bu output'u hot kabul edeceğiz. Planet örneği için 0.2 tresholdu seçilmiş, bunu deneme yanılma yoluyla bulabiliriz.

Böyle bir accuracy fonksiyonu zaten tanımlı, önceden kullanılan accuracy yerine burada accuracy_tresh(inp,targ,tresh=0.2) fonksiyonunu kullanacağız. Bu fonksiyon bir inp, target ve treshold alır, ilgili treshold'a göre input'un hotlarını belirler ve target ile karşılaştırarak bir accuracy döndürür.

Cnn_learner içine tanımlanan metrics eğitim sırasında otomatik olarak çağırılır, biz bunları elle çağırmayız. Biz her seferinde treshold=0.2 ile çağıralacak bir accuracy_tresh fonksiyonu istiyoruz. Bu yüzden yeni bir fonksiyon tanımlayacağız.Aslında neden metric kısmında accuracy_tresh(tresh=0.2) şeklinde çağıramıyoruz onu kavrayamadım.Sanıyorum bunun sebebi şu metrics=[acc_02,f_score] dediğimizde acc_02 fonskiyonu çağırılmıyor, sadece parametre olarak veriliyor, yani modele diyoruz ki eğitim sırasında acc_02 fonskiyonu çağır, yani burada acc_02(tresh=0.2)'yi parametre olarak veremeyiz yalnızca fonksiyonun kendisini parametre olarak verebiliyoruz. Bu sebeple de yeni bir fonksiyon yaratıyoruz.

Ama yeni bir fonksiyon yaratmamız gerektiğini söylüyor ve bunun için iki farklı yol var ilki şu:

```
def acc_02(inp,targ): return accuracy_tresh(inp,targ,treshold=0.2)
```

Böylece acc_02 input ve target ile çağırıldığında aslında accuracy_tresh() çağırılmış olacak ve treshold her zaman 0.2 olacak.

Ancak bir başka fonsiyona çok benzeyen bir başka fonskiyon yaratacaksak, ancak bunu hep sabit bir parametre ile çağıracaksak, bunun daha kısa ve common bir yolu var o da partial functions.

acc_02 = partial(accuracy_thresh, thresh=0.2)

f_score = partial(fbeta, thresh=0.2)

learn = create_cnn(data, arch, metrics=[acc_02, f_score], model_dir='/tmp/models')

O yüzden yukarıdaki acc_02 partial function'ı tam da bu işlevi yerine getirir.

Bu yapı fastai'da çok kullanılır, çünkü çoğu durumda we need to pass in some slightly customized versions of functions.

Sonuç olarak learner objemizi yukarıdaki gibi yarattık, artık kalan adımlar lesson 1-2 ile birebir aynı. Yani datayı elde ettiğimiz aman geriye kalan adımlar fastai ile gayet pratik ve başarılı şekilde halloluyor.

5. Dakika 42 civarı soru cevap:

- Deploy edilen modele kullanıcının yüklediği fotoğraflarda yanlış classification yapılırsa bu örnekleri modelimizi geliştirmek için kullanabilir miyiz?
- Elbette. Öncelikle, modelin yanılıp yanılmadığını user'dan feedback olarak almalıyız. Bu feedback'i kaydetmeliyiz. Daha sonra belki gün sonunda veya haftanın sonunda elde edilen bu verileri kullanarak hali hazırda eğitilmiş modelimize fine-tuning uygulayabiliriz.
- Burada yapılacak şey, aynen stage2 gibi olacak. Tüm modeli birazcık daha eğiteceğiz. Tabii ki yeni yaratılan databunch kullanılacak.
- Bu noktada şu var, misclassified örnekler sıradan örneklerden biraz daha farklı olacak bu yüzden higher learning rate kullanmak veya daha fazla epoch kullanmak mantıklı olabilir.

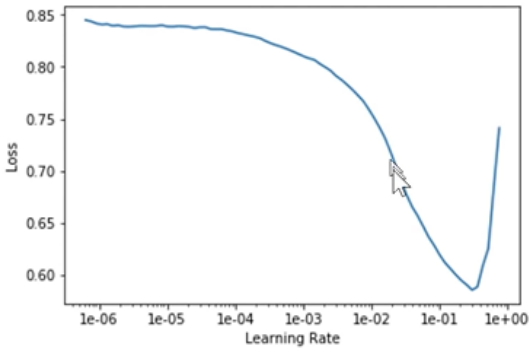
- Data block API'ı kullanırken blockları belli bir sırada mı yazmalıyız?
- Evet documentation'da belirtilen order'ı kullanmalıyız. Documentation'ı incelersek yeterli olacaktır.

- Video'yu framelerine ayırmak ve daha sonra modele beslemek için ne kullanabiliriz?
- Uygulamaya bağlı, eğer bu işlemi webde yapacaksanız, bunun için web API's var.
- Client-side için ise open-cv kullanılabilir?

6. Planet Multi-label classification örneğine geri dönüyoruz:

Unfreeze öncesi lr finder'ı çalıştırırsak nerdeyse her zaman aşağıdakine benzer bir şekil elde ederiz. Burada steepest slope'u seçeriz. Aşağıdaki örnek için 10^-2 civarı seçilmiş.

```
In [18]: learn.lr_find()
         LR Finder complete, type {learner_name}.recorder.plot() to see the graph.

In [19]: learn.recorder.plot()
```

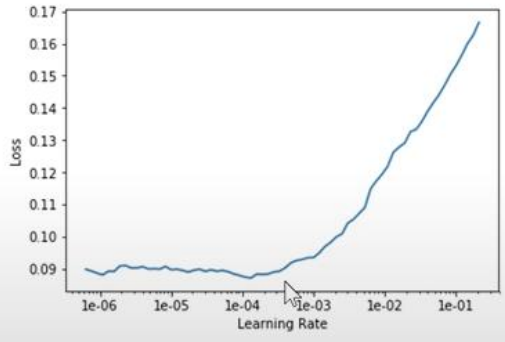


Then we can fit the head of our network.

```
In [25]: lr = 0.01
```

Stage-1 eğitimini tamamladıktan sonra bunu kaydederiz ve stage-2 için tekrar lr finder'ı çağırırız. Bu kez çok daha farklı bir shape elde ederiz:

```
In [111]: learn.unfreeze()

In [112]: learn.lr_find()
          learn.recorder.plot()
          LR Finder complete, type {learner_name}.recorder.plot() to see the graph.
```




```
In [113]: learn.fit_one_cycle(5, slice(1e-5, lr/5))
```

Burada seçim yapmak daha zor görünüyor. Jeremy diyor ki ben genelde böyle şekiller için loss uçmadan hemen önceki kısımdan 10x geri gelirim diyor. Yani burada 10^-4'ten 10x geri gelince 10^-5.

Bu rakamı yukarıda görüldüğü gibi, slice'ın ilk parametresi olarak gireriz. İkinci parametre içinse yine stage-1 değerinin 10x veya 5x gerisine gelinir. Bu örnekle lr/5 ile 5x gerisine gelmiş.

Burada tam olarak ne olduğundan daha sonra bahsedeceğiz buna Discriminitive Learning Rates deniyormuş.

Bu işlemler sonucunda, f2 score'umuz 0.92'lerde yani gayet iyi ancak daha bunu slightly better bir hale getirirsek çok daha iyi olacak. Nasıl yapabiliriz?

Şimdi bu örneğin başlarına geri dönersek, databunch'ı oluşturmak için transform kısmında size=128 verdiğimizi hatırlayabiliriz. Yani image'lar 128x128 size'ında eğitim yaptık

```
tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0.)
```

We use parentheses around the data block pipeline below, so that we can use a multiline statement without needing to add '\'.

```
np.random.seed(42)
src = (ImageItemList.from_csv(path, 'train_v2.csv', folder='train-jpg', suffix='.jpg
       .random_split_by_pct(0.2)
       .label_from_df(sep=' '))
```

```
data = (src.transform(tfms, size=128)
        .databunch(num_workers=0).normalize(imagenet_stats))
```

Aslında kaggle dataset'in örnekleri 256x256 boyutunda, bizim size=128 kullanmamızın bir sebebi, daha hızlı bir şekilde experiment yapabilmek. Small images kullanırsak experiment yapmak çok daha hızlı ve kolay olacaktır.

Size=128'in bir diğer sebebi ise şu: Bu şekilde artık 128x128'lik images üzerinde gayet iyi çalışan bir model elde ettik. Şimdi ise 256x256'lık images üzerinde iyi çalışan bir model elde etmek istiyorum. Ancak bu modeli sıfırdan eğitmek yerine neden transfer learning kullanmayayım ki? 128x128'lik images üzerinde iyi çalışan modeli alıp yeni dataset üzerinde fine-tune edebilirim. Böylece sıfırdan yeni model eğitmiş olmam.

Bu yöntem yeni bir yöntem, ve Jeremy bunun çok iyi çalıştığını söylüyor. Progressive size increase deniyormuş galiba. 128x128'lik images üzerinde modelimizi önce bir eğitiyoruz, belki biraz daha eğitsek artık overfitting problemi ile karşılaşacağız tam bu noktada, dataset'i değiştiriyoruz bunlar belki aynı dataset ama 256x256 boyutuna geldiği için convNet için aslında tamamen yeni bir dataset bu yüzden modelimiz önceki dataset'e ettiği overfitten kurtuluyor.

Bu olayı yapabilmek için 256'lık images ile yeni bir databunch oluşturmalıyız. Bu yüzden kodun başında databunch'ı oluştururken aşağıda görüldüğü gibi iki parçada oluşturduk, önce src dataset'ini oluşturduk daha sonra databunch'a çevirdik. Bunun sebebi Progressive Size Increase'i yapabilmekti.

```
tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0.)
```

We use parentheses around the data block pipeline below, so that we can use a multiline statement without needing to add '\'.

```
np.random.seed(42)
src = (ImageItemList.from_csv(path, 'train_v2.csv', folder='train-jpg', suffix='.jpg
       .random_split_by_pct(0.2)
       .label_from_df(sep=' '))
```

```
data = (src.transform(tfms, size=128)
        .databunch(num_workers=0).normalize(imagenet_stats))
```

Bunu nasıl yaptığımıza bakalım:

```
Total time: 05:48
epoch  train_loss  valid_loss  accuracy_thresh  fbeta
1      0.096917    0.089857    0.964909         0.923028  (01:09)
2      0.095722    0.087677    0.966341         0.924712  (01:09)
3      0.088859    0.085950    0.966813         0.926390  (01:09)
4      0.085320    0.083416    0.967663         0.927521  (01:09)
5      0.081530    0.082129    0.968121         0.928895  (01:09)
```

```
In [114]: learn.save('stage-2-rn50')
```

```
In [17]: data = (src.datasets(ImageMultiDataset)
                 .transform(tfms, size=256)
                 .databunch().normalize(imagenet_stats))

         learn.data = data
         data.train_ds[0][0].shape
Out[17]: torch.Size([3, 256, 256])
```

```
In [116]: learn.freeze()
```

```
In [79]: learn.lr_find()
         learn.recorder.plot()
```

Yukarıda görüldüğü gibi 128x128 images ile stage-2'yi tamamladıktan sonra, daha önce elde edilen src datasetinden size=256 parametresi ile yeni bir databunch elde ettik.

Elde edilen yeni dataBunch'ımı learner objemin data'sına atıyorum.

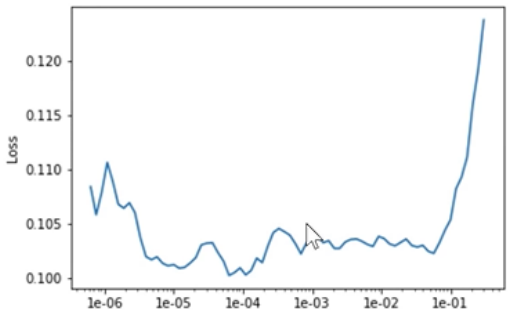
learn.data = data

Şimdi, train datasetimin x axesinin 0. örneğine bakıyorum, ki 0. örneğin y axes'i de bunun labels'ından başka bir şey değil. Bu örneğin shape'ine baktığım zaman 256x256'lık bir image olduğunu görüyorum.

Bu noktadan sonra, daha önceki adımları aynen tekrarlayabilirim. Önce learner'ı freeze ediyorum ve sadece head'i eğiteceğim. Bunun için iyi bir lr seçmeliyim, lr finder'ı kullanıyorum.

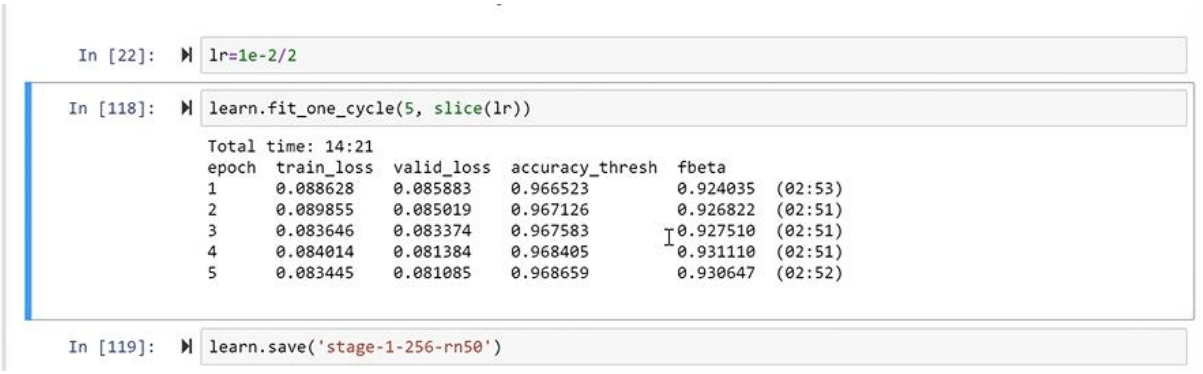
```
In [116]: learn.freeze()
```

```
In [79]: learn.lr_find()
         learn.recorder.plot()
LR Finder complete, type {learner_name}.recorder.plot() to see the graph.
```



Farkındaysan, yukarıdaki lr plot, stage-1 plotu gibi çıkmadı. Çünkü benim elimdeki model zaten size=128 input için gayet iyi çalışıyor yani bu bir nevi stage2 sayılır. Yukarıdaki şekil için de daha önce yaptığımız gibi, ani yükselişin başlangıcından 10x gerisini falan alıyoruz.

Yani lr=1e-2/2 alabiliriz.

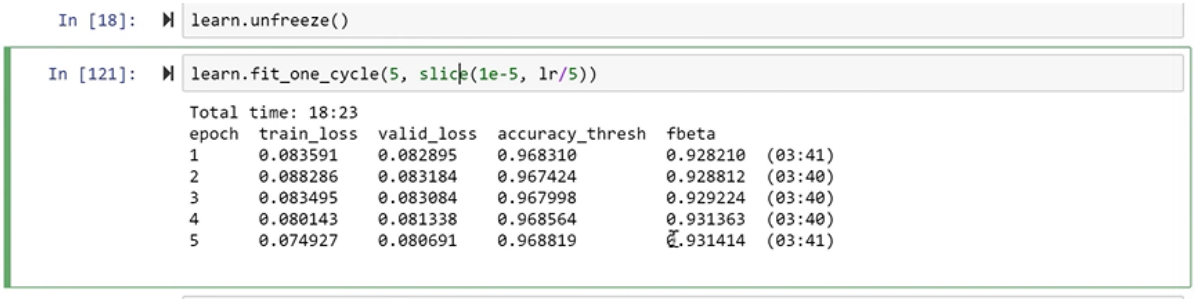


```
In [22]: lr=1e-2/2

In [118]: learn.fit_one_cycle(5, slice(lr))

Total time: 14:21
epoch  train_loss  valid_loss  accuracy_thresh  fbeta
1      0.088628    0.085883    0.966523         0.924035  (02:53)
2      0.089855    0.085019    0.967126         0.926822  (02:51)
3      0.083646    0.083374    0.967583         0.927510  (02:51)
4      0.084014    0.081384    0.968405         0.931110  (02:51)
5      0.083445    0.081085    0.968659         0.930647  (02:52)

In [119]: learn.save('stage-1-256-rn50')
```

Bu lr ile head kısmını fine-tune ediyorum.

Görüdğümüz gibi 0.92'lerden 0.93 f2 score'unu elde ettik bile.

Bunun üzerine bir de unfreeze edip tüm modeli fine-tune edersek tadından yenmez.



```
In [18]: learn.unfreeze()

In [121]: learn.fit_one_cycle(5, slice(1e-5, lr/5))

Total time: 18:23
epoch  train_loss  valid_loss  accuracy_thresh  fbeta
1      0.083591    0.082895    0.968310         0.928210  (03:41)
2      0.088286    0.083184    0.967424         0.928812  (03:40)
3      0.083495    0.083084    0.967998         0.929224  (03:40)
4      0.080143    0.081338    0.968564         0.931363  (03:40)
5      0.074927    0.080691    0.968819         0.931414  (03:41)
```

Bu score ile kaggle'da top20'ye giriyoruz resmen. İnanılmaz bir şey.

Böylece Plane MultiClass Classification örneğini tamamlamış oluyoruz.

Şimdi CamVid örneğine bakabiliriz. Bu örneği kernel üzerinde incelemeyi düşünüyorum. Notlarımı da oraya alacağım.