

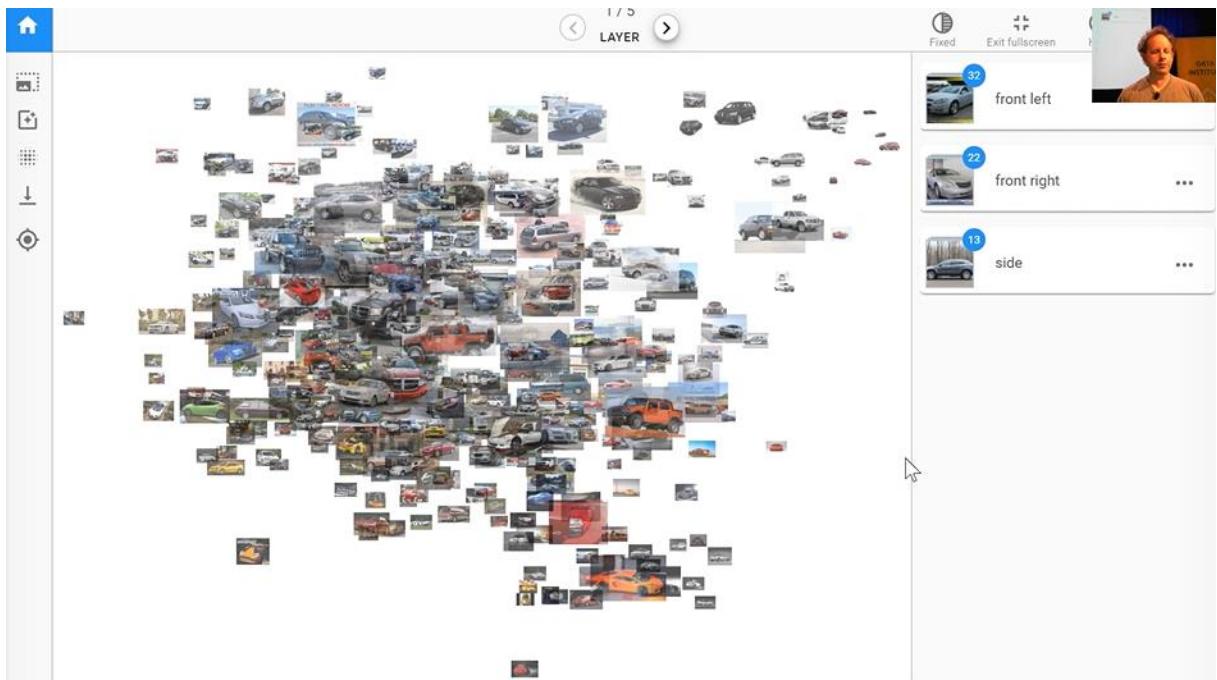
Bu kısımda L2 regularization'ın yanında diğer regularization methodlarını da göreceğiz. Dropout, Data augmentation ve Batch normalization'dan bahsedilecek.

Computer vision'a deep dive yapacağız. CNN'i göreceğiz. Ayrıca Data Ethics konusundan bahsedilecek.

Build Models with Unlabeled Data

platform.ai sitesi ile labelı olmayandataları kolayca labellayarak modeller eğitebiliriz.

Siteye gidip tek bil file içinde unlabeled dataset'i yükleyebiliriz. Aşağı yukarı 500 örnek model eğitmek için yeterli olur daha fazlası göz çıkarmaz tabi.



Diyelim ki bir araba datasetini aldık yükledik, yukarıdaki gibi bir görüntü elde ediyoruz. Ekranda görülenler tüm dataset değil, sadece bir random sample.

Bu resimler bir deep learning space'den 2D space'e projected hali. Yani resnet34 gibi bir imagenet model kullanılmış, yukarıda layerlar görünüyor şuanda ilk layerdayız yani burada diyelim ki ilk layerda 100 feature detect ediliyor yani dataset 100 dimensional bir uzaya dağılmış durumda ancak PCA gibi bir yöntemle ben bu dataseti 100D'den 2D'ye çekebilirim ve çekince işte yukarıdaki gibi bir dağılım elde etmişiz.

Sonuçta 2D uzayda her axis 1. Layer'in yakaladığı simple feature'ların karışımını temsil ediyor, yani birbirinden farklı konumda olan images farklı simple feature'lara sahip diyebiliriz.

Bu örnekte nihai amacımız, yüklenen dataseti farklı açılardan oluşan subclass'lara ayırmak olsun, yani datayı labellayacağız, ancak bunu tek tek yapmak yerine bu tooldan yararlanarak hızlıca yapmayı amaçlıyoruz.

Ancak imangenet üzerinde eğitilen bir model arabanın farklı açılarını yakalamakta pek de iyi olmayacağından değil mi? Çünkü imangenet spesific olarak arabalar – bisikletler – kuşlar vb. gibi ayrımlar yapmak için eğitildi, yani iki farklı açıdan çekilen arabanın ikisini de araba olarak kabul etmek için eğitildi zaten.

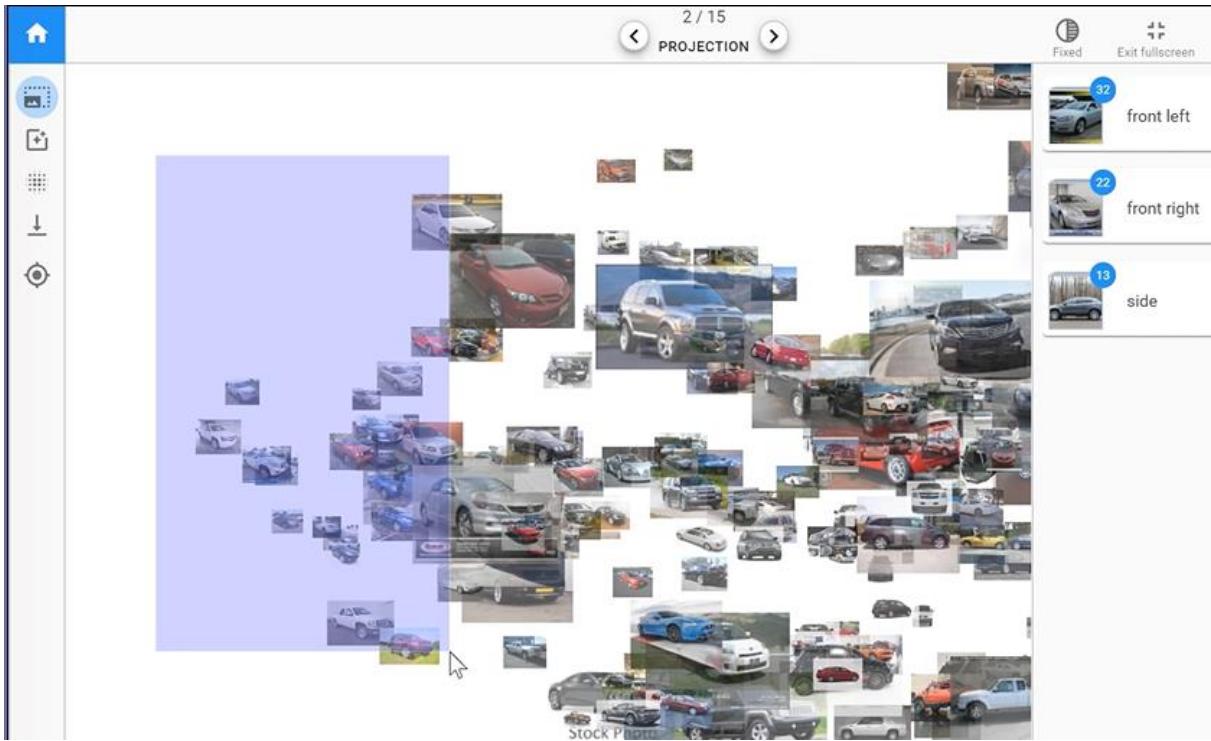
Yukarıdan layerları değiştirince projections da değişecektir, bu bağlamda son layer projectionları ile arabaların farklı dağılmasını beklemek pek mantıklı olmayacağından çünkü bu layerin amacı zaten arabaların hepsini birbirine yakın konumlara atamak ki diğer objelerden ayırt edilebilsin.

Benzer şekilde ilk layer da bizim için useless olacaktır çünkü, kenar köşe gibi basit feature'lar ile angle ayırt etmek zor olabilir.

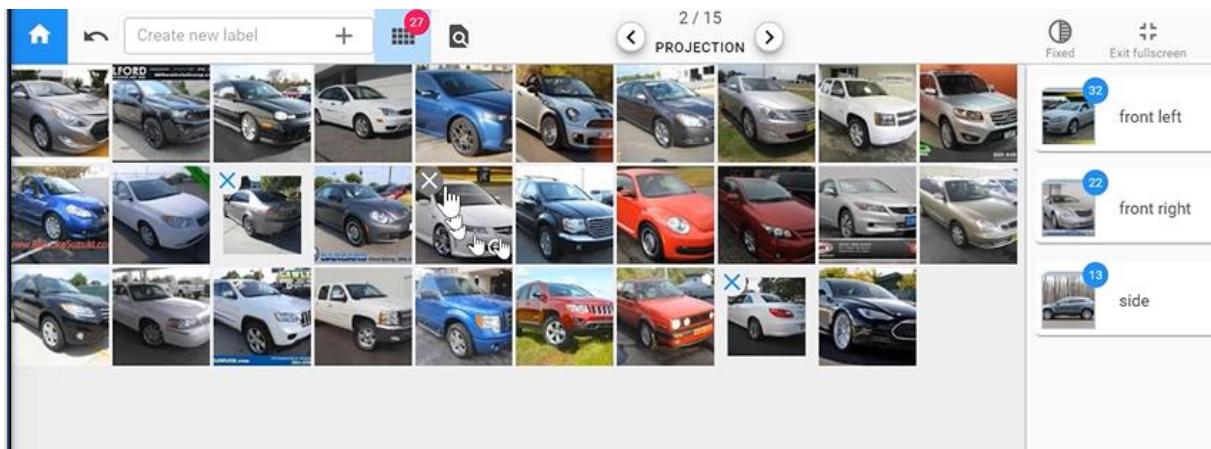
Bu sebeple farklı açıların farklı dağılımını yakalayabilmek için orta layer'a yani 3. Layer'a gidiyoruz, umuyoruz ki burada dağılım farklı olsun.

3. layer'a geldikten sonra yukarıdan projection'ı seçip 15 farklı projection arasında gezebiliriz böylece 3. Layer için kendi amacımıza uygun projection bulabiliyoruz, amacımız da şu farklı açılardan çekilmiş araba fotoğrafları ayrık şekilde dursun yani kümelenmiş olsun.

Bu noktada 2. Projection'a baktığımızda görüyoruz ki 2D projection'ın sol kısımlarındaki arabalar hep aynı angle'dan çekilmiş, işte bu noktada buraya zoom yapıyoruz ve soldan selection mode'a gelip bu fotoğrafları seçiyoruz.



Daha sonra seçilen fotoğrafları yakından inceleyip araya karışmış hatalı açıları atabiliyorum:



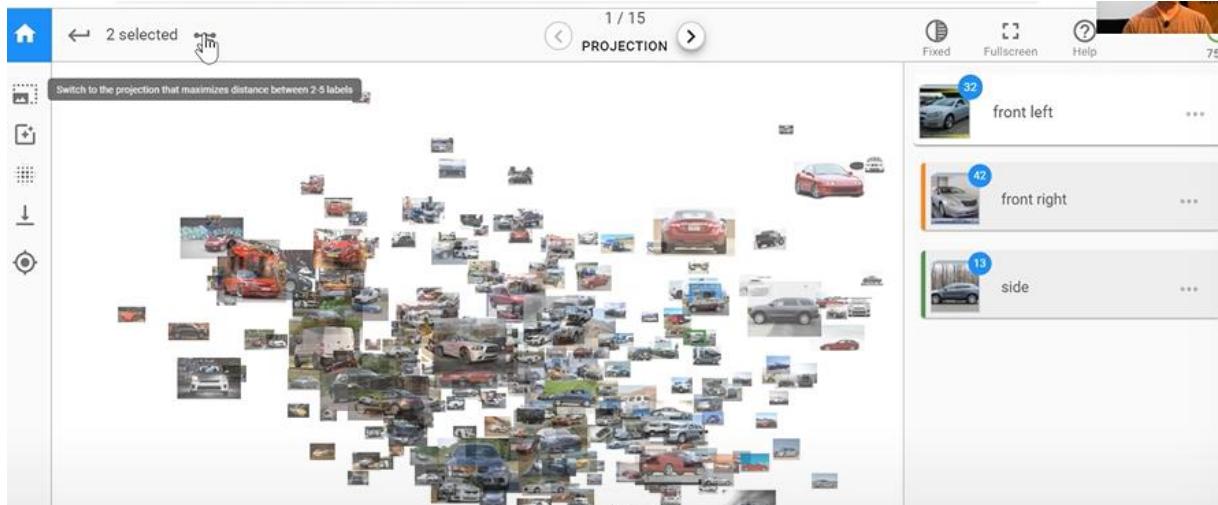
Bu seçilen fotoğraflarla yeni bir label oluşturabileceğim gibi fotoğrafları var olan bir labelin içine de taşıyabilirim.

Bu şekilde bir nevi hızlandırılmış labeling yapabiliyorum.

Ancak zamanla şunu görebiliriz ki her label'ı rahatlıkla tespit edemiyor olabilirim. Mesela bu örnekte sideview bulmakta zorlanıyoruz.

Bu problemi çözmek için yine aynı projection sayfasında sideview fotoğraflardan birkaçına tıplayıp yukarıdan find similar images seçeneğini işaretleyebilirim, böylece o projection space'de seçilen images'a benzeyen projectionlar tüm datasetten seçili önümüze gelecek.

Ayrıca eğer diyelim ki side view ile front right view'i birbirinden ayıran bir projection bulmak istiyorum diyelim, bunu yapmak için yaratılan label klasörlerinin ikisini de seçtikten sonra sol üste switch to the projection that maximizes distance between labels seçeneğini işaretliyorum:



Sonuçta öyle bir 2D projection bulmak istiyorum ki side views ile front right views biribirinden ayırik dursun, böylece yeni gelen images'dan kolaylıkla istenilen angle fotoğraflarını seçebilirim.

200-300 civarı fotoğrafı ayırip classlarını oluşturuktan sonra soldaki train model butonu ile kolay bir şekilde modelimizi eğitebiliriz.

Sonuçta hep predictions'ı hem de labelları indirebiliriz böylece bu dataseti kendimiz daha iyi bir eğitim için kullanabiliriz.

Tabular Model ile Kalan Regularization Methodlarını Anlamaya Çalışalım

Bu amaçla Rossmann Store Sales ismindeki kaggle competition dataset'ini kullanarak bir tabular model eğitilecek, bu kernel kaggle'da var zaten, ben buradan devam edeceğim.

Bu competition için error metric olarak RMSPE kullanılmış bunun formülü de RMSE'e çok benziyor ancak percentage olarak hesaplanıyor yani $y - \hat{y}$ tekrar y 'ye bölündükten sonra kare alınıyor.

$$\text{RMSPE} = \sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2},$$

Bu problem için 6-7 farklı table joint edilecek kullanılacak vesaire ancak bu derste bunun detayına girmeyeceğiz, ancak bunu ben aşağı yukarı biliyorum zaten.

Yani data preperation'i atlıyoruz. Bu preperation işlemi rossman_data_clean.ipynb içerisinde yapılmış istersen aç bak.

In [1]: %reload_ext autoreload
%autoreload 2

In [2]: from fastai import *
from fastai.tabular import *

▼ Rossmann

▼ Data preparation

To create the feature-engineered train_clean and test_clean from the Kaggle competition data, run rossman_data_clean.ipynb . One important step that deals with time series is this:

```
add_datepart(train, "Date", drop=False)  
add_datepart(test, "Date", drop=False)
```

Burada önemli bir nokta preperation içindeki yukarıdaki `add_datepart()` kısmı. Timeseries için RNN kullanılmasını bekliyor olabilirsiniz ama kullanılmayacak, RNN'in iyi olduğu alan görünüşe göre eğer elinizdeki tek bir data varsa ve o da time series ise. Ancak gerçek hayatı neredeyse hiçbir zaman durum bu değil, genelde elimizde store'la ilgili insalarla ilgili birsürü bilgi metadata veya başka sequence veriler olur.

Bu yüzden pratikte çoğunlukla timeseries veriyi ki bu durumda bu tarih verisi oluyor, bu veriyi alıp bir sürü farklı metadataya çevirip o şekilde tabular bir model eğitmek çok iyi sonuçlar veriyor.

İşte `add_datepart()`'ın yaptığı şey bu, date kısmını alıyor bırsürü categorical variable ekliyor bunlar, kaçinci hafta, hangi ay, hangi gün, vesaire gibi categorical variables:

Open	1	1	1	1	1
Promo	1	1	1	1	1
StateHoliday	False	False	False	False	False
SchoolHoliday	1	1	1	1	1
Year	2015	2015	2015	2015	2015
Month	7	7	7	7	7
Week	31 	31	31	31	31
Day	31	31	31	31	31
Dayofweek	4	4	4	4	4
Dayofyear	212	212	212	212	212
Is_month_end	True	True	True	True	True
Is_month_start	False	False	False	False	False
Is_quarter_end	False	False	False	False	False
Is_quarter_start	False	False	False	False	False
Is_year_end	False	False	False	False	False
Is_year_start	False	False	False	False	False
Elapsed	1438300800	1438300800	1438300800	1438300800	1438300800
StoreType	c	a	a	c	a
Assortment	a	a	a	c	a

Yani özetle dataset içindeki zaman serisi değişken'i temsil etmek için bırsürü veri ekliyoruz böylece mesela ayın 15'nde payday olduğu için harcamaların arttığı gibi bir bilgiyi modelin yakalayabilmesini sağlıyoruz, modelin böyle komplex bir feature'i başka türlü timestamp'den elde etmesi imkansız gibi bir şey.

Bu yöntemle bir çok time-series probleme regular tabular problems gibi yaklaşabiliriz, hepsine değil ancak birçoğuna böyle yaklaşabiliriz.

Sonuçta yukarıdaki gibi bir dataset'i elde ettik burada her column bir example, burada sadece ilk 5 column gösterilmiş head() ile baktığımız için, sonuçta tahmin edilmeye çalışılan şey de bu değişenlerden birisi Sales! Yani sales is the dependent variable.

In [4]:	train_df.head().T
Out[4]:	
	0 1 2 3 4
index	0 1 2 3 4
Store	1 2 3 4 5
DayOfWeek	5 5 5 5 5
Date	2015-07-31 00:00:00 2015-07-31 00:00:00 2015-07-31 00:00:00 2015-07-31 00:00:00 2015-07-31 00:00:00
Sales	5263 6064 8314 13995 4822
Customers	555 625 821 1498 559
Open	1 1 1 1 1
Promo	1 1 1 1 1
StateHoliday	False False False False False
SchoolHoliday	1 1 1 1 1
Year	2015 2015 2015 2015 2015
Month	7 7 7 7 7
Week	31 31 31 31 31
Day	31 31 31 31 31
Dayofweek	4 4 4 4 4

İlk bakacağımız şey PreProcessers:

Transforms'u görmüştük, transform dataset'ten minibatch alınırken her seferinde uygulanıyordu ve data augmentation sağlıyordu.

Preprocessors da transforms'a benziyor ancak biraz farklılar preprocessors trainingden önce tek bir kez çalışırlar, eğitim süresince çalışmaz.

Önemli nokta şu, trainings et üzerinde bir kez çalışmalarına rağmen, üretilen metadata validation ve test set ile paylaşılır, yani o dataset'e de uygulanır, yoksa training set ile diğer setlerin dağılımı değiştirdi bu istediğimiz bir şey değil.

Şimdi bunları anlamak için önce bir subset dataset yaratıyorum yani bir sample yaratıyorum, önce rastgele 2000 id çekiyorum, daha sonra 1000'lik bir dataset ve 1000'lik bir test set yaratıyorum ve seçtiğim bazı cont ve categorical verileri çekiyorum. şimdi bu datasetlerle oynayarak yapmak istediklerimi daha rahat deneyebilirim.

Experimenting with a sample

```
In [6]: idx = np.random.permutation(range(n))[:2000]
idx.sort()
small_train_df = train_df.iloc[idx[:1000]]
small_test_df = train_df.iloc[idx[1000:]]
small_cont_vars = ['CompetitionDistance', 'Mean_Humidity']
small_cat_vars = ['Store', 'DayOfWeek', 'PromoInterval']
small_train_df = small_train_df[small_cat_vars + small_cont_vars + ['Sales']]
small_test_df = small_test_df[small_cat_vars + small_cont_vars + ['Sales']]
```

```
In [7]: small_train_df.head()
```

Out[7]:

	Store	DayOfWeek	PromoInterval	CompetitionDistance	Mean_Humidity	Sales
280	281	5	NaN	6970.0	61	8053
584	586	5	NaN	250.0	61	17879
588	590	5	Jan,Apr,Jul,Oct	4520.0	51	7250

Sub training set'e bakınca görüyoruz ki, PromoInterval satırında NaN değerler var orijinal değerleri "Jan,Apr,Jul,Oct" gibi string değerler.

280	281	5	NaN	6970.0	61	8053
584	586	5	NaN	250.0	61	17879
588	590	5	Jan,Apr,Jul,Oct	4520.0	51	7250
847	849	5	NaN	5000.0	67	10829
896	899	5	Jan,Apr,Jul,Oct	2590.0	55	5952

```
In [8]: small_test_df.head()
```

Out[8]:

	Store	DayOfWeek	PromoInterval	CompetitionDistance	Mean_Humidity	Sales
428412	921	2	NaN	840.0	89	8343
428541	1050	2	Mar,Jun,Sept,Dec	13170.0	78	4945
428813	209	1	Jan,Apr,Jul,Oct	11680.0	85	4946
430157	414	6	Jan,Apr,Jul,Oct	6210.0	88	6952
431137	285	5	NaN	2410.0	57	5377

İlk bakacağımız preprocessör CATEGORIFY. Bunun yapacağı şey şu, belirtilen PromoInterval için kullanılan tüm farklı stringleri alacak bir liste yaratacak ve herbirine bir number atayacak. Training set üzerinde çağırınca kategorilerini burada yaratacak, daha sonra test set üzerinde çağrırmak için test=True parametresi ile çağırınca, training set üzerindeki kategorilerin aynısını test set'e uygulayacak.

```
In [9]: categorify = Categorify(small_cat_vars, small_cont_vars)
categorify(small_train_df)
categorify(small_test_df, test=True)
```

Uygulamadan sonra test set' bakarsak promo interval hala aynı görünüyor, ancak aslında değişim arkaplanda yapıldı pandas bize hala eski string değerleri gösteriyor.

```
In [10]: small_test_df.head()
Out[10]:
   Store  DayOfWeek  PromoInterval  CompetitionDistance  Mean_Humidity  Sales
428412    NaN         2        NaN             840.0            89     8343
428541  1050.0       2  Mar,Jun,Sept,Dec          13170.0            78     4945
428813    NaN         1  Jan,Apr,Jul,Oct          11680.0            85     4946
430157    414.0       6  Jan,Apr,Jul,Oct          6210.0            88     6952
431137    285.0       5        NaN             2410.0            57     5377
```



```
In [11]: small_train_df.PromoInterval.cat.categories
Out[11]: Index(['Feb,May,Aug,Nov', 'Jan,Apr,Jul,Oct', 'Mar,Jun,Sept,Dec'], dtype='object')
```

Değişimi anlamak için `PromoInterval.cat.categories`'e bakıyoruz ve `list of classes`'ı görüyoruz.

```
In [12]: small_train_df['PromoInterval'].cat.codes[:5]
Out[12]:
280    -1
584    -1
588     1
847    -1
896     1
dtype: int8
```

Bu şekilde de classlara karşılık gelen sayı kodlarını görebilirim, -1 demek `NaN` demek. Tabi bu değerler bir embedding matrixde kullanılacak bu yüzden -1'ler yerine 1 kullanılıyor.?

Yani `categorify`'ın yaptığı şey datayı kategorik veriye çevirmek ve her birine bir sayısal kod atamak. Daha sonra her bir kategorik veri için bir embedding vector oluşturulacak ve NN için bu embedding vector kullanılacak.

Benzer şekilde bir diğer preprocessor ile **FILLMISSING** Bunun yapacağı şey missingleri doldurup yerine isMissing column'u eklemek, böylece hem missingi doldurmuş oluyoruz ama hatalı bir doldurma için de modele isMissing column'u ile durumu yakalama şansı veriyoruz.

```
In [13]: fill_missing = FillMissing(small_cat_vars, small_cont_vars)
fill_missing(small_train_df)
fill_missing(small_test_df, test=True)

In [14]: small_train_df[small_train_df['CompetitionDistance_na'] == True]

Out[14]:
   Store  DayOfWeek  Promolnerval  CompetitionDistance  Mean_Humidity  Sales  CompetitionDistance_na
78375      622          5            NaN        2380.0             71    5390                True
161185      622          6            NaN        2380.0             91    2659                True
363369      879          4  Feb,May,Aug,Nov        2380.0             73    4788                True
```

Preprocessors'ı manual olarak kodlamamıza gerek yok, aşağıdaki gibi processorsı tanımladıktan sonra, databunch'imizi oluştururkenhatta ItemList'imizi oluştururken içерinse procs=procs satırı ile processors'ı direkt tanımlayabiliriz.

```
In [17]: procs=[FillMissing, Categorify, Normalize]

In [24]: data = (TabularList.from_df(df, path=path, cat_names=cat_vars, cont_names=cont_vars, procs=procs)
           .split_by_idx(valid_idx)
           .label_from_df(cols=dep_var, label_cls=FloatList, log=True)
           .databunch())
```

Böylece, missing değerleri otomatik olarak dolduracak categorify uygulayacak ve normalization(continious variables için) yapacak.

Tabular Örneğine Devam

Tabular databunch oluşturmak için categorical ve continuous variables'ı belirtmemiz gereklidir.

```
In [17]: procs=[FillMissing, Categorify, Normalize]

In [18]: cat_vars = ['Store', 'DayOfWeek', 'Year', 'Month', 'Day', 'StateHoliday', 'CompetitionMonthsOpen',
      'Promo2Weeks', 'StoreType', 'Assortment', 'PromoInterval', 'CompetitionOpenSinceYear', 'Promo2Si
      'State', 'Week', 'Events', 'Promo_fw', 'Promo_bw', 'StateHoliday_fw', 'StateHoliday_bw',
      'SchoolHoliday_fw', 'SchoolHoliday_bw']

cont_vars = ['CompetitionDistance', 'Max_TemperatureC', 'Mean_TemperatureC', 'Min_TemperatureC',
      'Max_Humidity', 'Mean_Humidity', 'Min_Humidity', 'Max_Wind_SpeedKm_h',
      'Mean_Wind_SpeedKm_h', 'CloudCover', 'trend', 'trend_DE',
      'AfterStateHoliday', 'BeforeStateHoliday', 'Promo', 'SchoolHoliday']

In [19]: dep_var = 'Sales'
df = train_df[cat_vars + cont_vars + [dep_var, 'Date']].copy()

In [20]: test_df['Date'].min(), test_df['Date'].max()
Out[20]: (Timestamp('2015-08-01 00:00:00'), Timestamp('2015-09-17 00:00:00'))

In [21]: cut = train_df['Date'][train_df['Date'] == train_df['Date'][len(test_df)]].index.max()
cut
Out[21]: 41395
```

Sadece stringleri değil başka değişkenleri de categorical olarak işleyebiliriz. Mesela day of month aslında 1-30 arasında bir number olmasına rağmen biz bunu categorical olarak aldık. Ancak ayın 1 ile 15 ile 30'u aslında farklı purchase behaviour gösterebilir. Bunu yakalayabilmek için bazı variable'ları categorical seçiyoruz.

Eğer cardinality çok yüksek değilse yani categorical'a çevirince mesela day of month için cardinality 30 bu yüzden yüksek olmayanları her zaman categorical olarak almakta fayda var.

Sonuçta birkaç stepten sonra tabular databunch'ı oluşturuyoruz.

Burada label_cls=FloatList şeklinde bir ibare var, bunu daha önce görmedik.

Dependent variable'a baktığımız zaman dtype:int64 olarak görünüyor, eğer bu değerler float olursa fastai problemin bir regression problemi olduğunu anlar, aksi halde bunu classification olarak kabul eder. Bu yüzden labeling yaparken list of floats kullanıyoruz ki fastai problemin regression olduğunu anlasın.

```
In [23]: df[dep_var].head()
Out[23]: 0    5263
          1    6664
          2    8314
          3   13995
          4    4822
Name: Sales, dtype: int64

In [24]: data = (TabularList.from_df(df, path=path, cat_names=cat_vars, cont_names=cont_vars, procs=procs)
           .split_by_idx(valid_idx)
           .label_from_df(cols=dep_var, label_cls=FloatList, log=True)
           .databunch())
In [28]: doc(FloatList)
```

Log=True ibaresi de y değerinin log'unu alır, bunu yapmamın sebebi loss olarak RMSE kullanabilmek, log alıp RMSE kullandığım zaman bu log almadan RMSPE kullanmakla aynı kapıya çıkıyor.

Databunch'ımızı oluşturuktan sonra modele geçebiliriz:

Model

```
In [29]: max_log_y = np.log(np.max(train_df['Sales'])*1.2)
y_range = torch.tensor([0, max_log_y], device=defaults.device)

In [26]: learn = tabular_learner(data, layers=[1000,500], ps=[0.001,0.01], emb_drop=0.04,
y_range=y_range, metrics=exp_rmspe)
```

Gördüğümüz gibi model için y_range tanımlıyoruz bunu yapınca modelin sonuna bir sigmoid ekliyoruz ve regression problemi için modeli tahmin sınırlarını öğrenme derdinden kurtarıyoruz.

Max değeri hespalarken 1.2 çarpanının sağladığı şey, sigmoidin üst sınırının gerçek maksimumdan %20 daha üstte olmasını sağlamak böylece modelimiz max değeri çıktı olarak verebilecek.

Tabular learner içine databunch'ımızı ve y_range'imizi ve metriğimizi vedığımızı zaten biliyoruz bunun yanında yukarıda görülen başka parametreler de var.

Layers'ı 1000'e 500 olarak tanımlıyoruz böylece intermediate weight matrix 50 000 weight'e sahip olacak bu datasetimiz için çok fazla bir rakam, yani overfit olacağı garanti, ancak overfiti engellemek için parametreden kışkırdık, onun yerine regularization kullanıyorduk işte diğer parametrelerin tanımladığı şey de bu, dropout regularization parametreleri tanımlanmış.

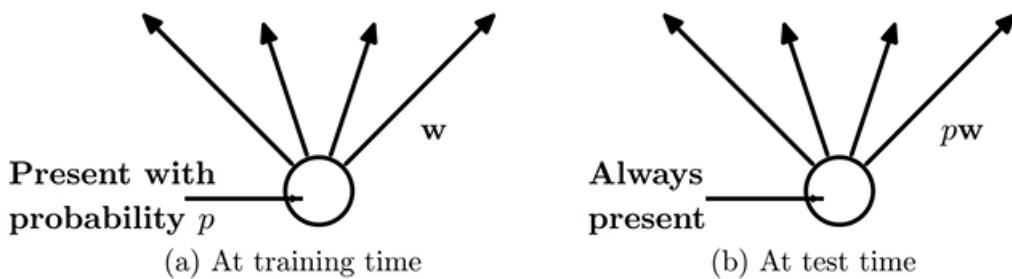
Weight decay regularization zaten otomatik olarak kullanıyor, istersek değerini değiştirebiliriz.

Daha fazla regularization istediğimiz için dropout kullanıyoruz.

Dakika 31:00 civarı dropoulttan bahsediyor özetle, random olarak unitleri deactivate ederek, unitlerin belirli feature'ları ezberleyerek sonucu bulmasını engelliyoruz.

Fastai learner için bir list olarak p values geçirebiliriz böylece ayrı ayrı layer'lar için dropout probability'sini geçirebiliriz yada tek bir değer gireriz ve tüm layerlar için bu değer kullanılır.

Normalde test time için dropout çalışmadığı için, training sırasında kullanıldandan daha fazla layer kullanılır dolayısıyla output'un daha yüksek çıkması beklenir, bunu engellemek için test time'da weights'i p ile çarpmak bir yöntem.



Ancak pytorch training sırasında yapılan bir trick ile bu durumu ortadan kaldırmış, yani test time'da weights'i p ile çarpmamıza gerek kalmamış. Çünkü onun yerine training time'daki prediction manipüle edilerek test time ile eşitlenmiş . Aşağıdaki kod sayesinde.

```
noise.bernoulli_(1 - p);
noise.div_(1 - p);
return multiply<inplace>(input, noise);
```

Emb_Drop=0.04 de embedding layerda kullanılan bir special dropout'ı işaret eder. Yani embedding aktivasyonlarından bazılarını random olarak kaldırıyoruz, sonuçta embedding layer da de bir layer. Bu sonuca experimental olarak ulaşmış sanıyorum.

Modelimizin içeriğine learn.model ile bakabiliriz:

```
In [27]: M learn.model
Out[27]: TabularModel(
    (embeds): ModuleList(
        (0): Embedding(1116, 50)
        (1): Embedding(8, 5)
        (2): Embedding(4, 3)
        (3): Embedding(13, 7)
        (4): Embedding(32, 17)
        (5): Embedding(3, 2)
        (6): Embedding(26, 14)
        (7): Embedding(27, 14)
        (8): Embedding(5, 3)
        (9): Embedding(4, 3)
        (10): Embedding(4, 3)
        (11): Embedding(24, 13)
        (12): Embedding(9, 5)
        (13): Embedding(13, 7)
        (14): Embedding(53, 27)
        (15): Embedding(22, 12)
        (16): Embedding(7, 4)
        (17): Embedding(7, 4)
        (18): Embedding(4, 3)
        (19): Embedding(4, 3)
        (20): Embedding(9, 5)
        (21): Embedding(9, 5)
        (22): Embedding(3, 2)
        (23): Embedding(3, 2)
    )
    (emb_drop): Dropout(p=0.04)
    (bn_cont): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (layers): Sequential(
        (0): Linear(in_features=229, out_features=1000, bias=True)
        (1): ReLU(inplace)
        (2): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (3): Dropout(p=0.001)
        (4): Linear(in_features=1000, out_features=500, bias=True)
        (5): ReLU(inplace)
        (6): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (7): Dropout(p=0.01)
        (8): Linear(in_features=500, out_features=1, bias=True)
    )
)
In [28]: M len(data.train_ds.cont_names)
Out[28]: 16
In [30]: M learn.lr_find()
LR Finder is complete, type {learner_name}.recorder.plot() to see the graph.
```

Beklediğimiz gibi birsürü embedding var her categorical variable için (cardinality, custom dimension) boyutlarında bir matrix vardır yani mesela 1116 tane store varmış ve herbirini temsil etmek için 50 parametre kullanılmış, bir store NN'e input olarak verileceğinde lookup ile bu 50 elemanlı vektör kullanılıyor sanıyorum ve her kullanımından sonra da bu vektörlerin weightleri update ediliyor en sonunda hem biz kategorik verilerimizi algoritmamızda kullanmış oluyoruz hem de her kategorik veriyi temsil eden feature'lar öğrenmiş oluyoruz böylece farklı kategorik verilerin arasındaki ilişkiler keşfedilenebiliyor.

16 tane continuous variable'ım var, bu yüzden yukarıdaki cont_names satırında 16'yı elde ettik.

Modelimin layerlarına bakarsak Embedding dropout'tan sonra bir 16 inputlu bir batchNorm layer görüyoruz çünkü 16 tane continuous variable'ım var.

Yani batchNormalization continuous variables için kullanılıyor.

BatchNorm Nedir?

Ne olduğu belirsiz bir şey. Biraz regularization gibi biraz training helper gibi bir şey.

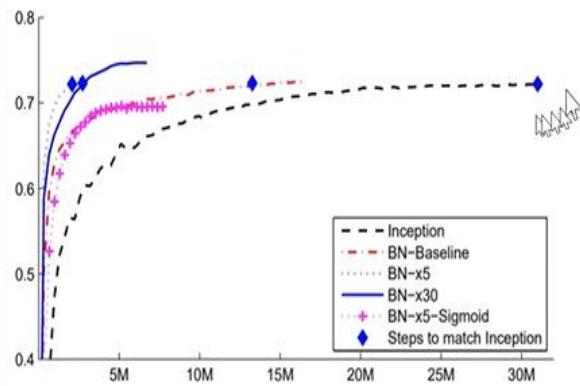
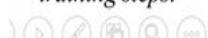


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;

Parameters to be learned: γ, β

Output: $\{y_i = BN_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch norm ile çok daha düşük training step ile yüksek accuracy'e ulaşabiliyoruz, ancak neden işe yaradığını açıklayanlar yanıldır. Yeni bir açıklamaya göre batch normalization'ın covariance shift'i azaltmadığı ortaya olmuş yani nasıl işe yaradığı biraz muğlak.

Aşağıda görüldüğü gibi batch norm ile eğitim yapılmınca loss çok daha az bumpy bu şu demek, batch norm yokken küçük bir parametre değişimi loss'da büyük zıplamalara yol açıyor veya bir başka deyişle loss function'ın çok engebeli olmasına yol açıyor bu yüzden learning rate büyütmemiyorduk ancak batch normalization ile daha temiz bir cost function elde ederek, learning rate'i büyütüyebiliyoruz ve böylece global minimum'a çok daha hızlı şekilde ulaşabiliyoruz.

How Does Batch Normalization Help Optimization?



Shibani Santurkar*
MIT
shibani@mit.edu

Dimitris Tsipras*
MIT
tsipras@mit.edu

Andrew Ilyas*
MIT
ailyas@mit.edu

Aleksander Madry
MIT
madry@mit.edu



“the positive impact of BatchNorm on training might be somewhat serendipitous”

What batchNorm does?

$$\hat{y} = f(\underbrace{w_1, w_2 \dots w_{\text{xxxx}},}_{\uparrow}, \vec{x}) \times g + b$$
$$L = \sum (y - \hat{y})^2$$

$-1 \rightarrow 1$

$1 \rightarrow 5 \rightarrow$

Diyelim ki, outputumun 1-5 arasında olmasını istiyorum ancak aktivasyon sonucu -1 ile 1 arası çıkıyor bunu değiştirmek için weightleri scale edemem. Onun yerine basitçe aktivasyonun çıkışını bir g ile çarparsam ve b eklersem -1 ile 1 arasında çıkış veren bir NN'i basit bir şekilde 1-5 arasında çıkış vermeye itebilirim ayarlaması gereken tek şey 2 parametre. Batch norm'un sağladığı şey bu.

Batch norm makes it easier to shifting outputs up and down and in and out.

Detaylar önemli değil ama **ÇOK İYİ ÇALIŞIYOR KESİNLİKLE KULLANMALIYIZ.**

Her continuous variable için bir batch norm layer kullanılıyor.

Modeli oluşturduk, ve detaylarından bahsettik.

Bu noktadan sonra lr_find() veya fit ile hep yaptığımız şeyleri yapabiliriz detaylar için kernel'e bakabilrsin.

SORU: Hangi regularization methodlarını ne zaman kullanmalıyız?

CEVAP: Weight Decay, Dropout, Data Augmentation ve Dropout'u düşündüğümüz zaman, batch norm'u her zaman kullanmak isteriz, data augmentation'dan az sonra bahsedeceğiz. Weight Decay ile Dropout arasında nasıl bir seçim yapılacak bilmiyoruz, kimse bilmiyor ikisini karıştıralım mı, ayrı ayrı mı kullanalım hangisi verimli kimsenin fikri yok. Ama sanki ikisinden de biraz kullanmak практиkte yararlı gibi duruyor biraz weight decayi neredeyse her zaman isteriz, birazcık dropout da fena olmuyor, bunları deneyip görmek lazım. Default değerler genelde gayet iyi çalışır ama istersen tabi deneme yapabilirsin.

Sonucta Tabular Data konusunu kapattik kapatirken de Dropout ve Batch Normalization'ı da aradan cikardik simdi yeni bir örnek ile **Data Augmentation**'dan bahsедelim:

Jeremy'e göre üzerinde en az çalışılan regularization metodu data augmentation buna karşın data augmentation'ın çok değerli olabileceğini söylüyor çünkü data augmentation'ın neredeyse hiç cost'u olmadığını söylüyor.

Data augmentation ile daha iyi bir generalization sağlayabiliriz ve bunu yaparken daha fazla eğitime gerek duymayız veya underfitting riski taşımayız.

Data augmentation'ı anlamak için daha önce kullanılan PETs datasetine geri dönüyoruz.

Lesson 6: pets revisited

```
In [1]: %reload_ext autoreload  
%autoreload 2  
%matplotlib inline  
  
from fastai import *  
from fastai.vision import *  
  
In [2]: bs = 64  
  
In [3]: path = untar_data(URLs.PETS) / 'images'
```

Data augmentation için get_transform satırı değerli. Burada bir çok transform'u belirttik tüm bunları anlamak için doc(get_transforms) ile documentation'a bakabiliriz.

Data augmentation

```
In [4]: tfms = get_transforms(max_rotate=20, max_zoom=1.3, max_lighting=0.4, max_warp=0.4,  
    p_affine=1., p_lighting=1.)  
  
In [5]: doc(get_transforms)  
  
In [6]: src = ImageItemList.from_folder(path).random_split_by_pct(0.2, seed=2)  
  
In [7]: def get_data(size, bs, padding_mode='reflection'):  
    return (src.label_from_re(r'([^\.]+)\.\d+.jpg$')  
        .transform(tfms, size=size, padding_mode=padding_mode)  
        .databunch(bs=bs).normalize(imagenet_stats))  
  
In [8]: data = get_data(224, bs, 'zeros')
```

```
In [9]: def _plot(i,j,ax):
    x,y = data.train_ds[3]
    x.show(ax, y=y)

plot_multi(_plot, 3, 3, figsize=(8,8))
```



Data augmentation ile free extra data kazanmış oluyoruz, farklı açıları farklı brightness değerleri rotation'ı ve daha bir çok şeyi hesaba katıyoruz böylece test datasını çok daha iyi bir şekilde genelleyebiliyoruz.

Data augmentation'ı diğer domain'lerde nasıl gerçekleştirebileceğimizi hala bilmiyoruz bunu anlamak büyük bir adım olurdu.

Şimdi Convolution'ı Anlamaya Çalışacağız:

Bir CNN yaratıyorum, biraz eğitiyorum unfreeze ediyorum sonra larget dataset için biraz daha eğitiyorum.

Train a model

```
In [12]: gc.collect()
learn = create_cnn(data, models.resnet34, metrics=error_rate, bn_final=True)
```

```
In [12]: learn.fit_one_cycle(3, slice(1e-2), pct_start=0.8)

Total time: 00:52
epoch  train_loss  valid_loss  error_rate
1      2.413196   1.091087   0.191475   (00:18)
2      1.397552   0.331309   0.081867   (00:17)
3      0.889401   0.269724   0.068336   (00:17)
```

```
In [13]: learn.unfreeze()
learn.fit_one_cycle(2, max_lr=slice(1e-6,1e-3), pct_start=0.8)

Total time: 00:44
epoch  train_loss  valid_loss  error_rate
1      0.695697   0.286645   0.064276   (00:22)
2      0.636241   0.295290   0.066982   (00:21)
```

```
In [14]: M data = get_data(352,bs)
learn.data = data

In [15]: M learn.fit_one_cycle(2, max_lr=slice(1e-6,1e-4))
Total time: 01:32
epoch  train_loss  valid_loss  error_rate
1      0.626780   0.264292  0.056834  (00:47)
2      0.585733   0.261575  0.048038  (00:45)

In [16]: M learn.save('352')
```

Sonuçta eğitilmiş bir CNN'im var.

Sırada CNN'in ne yaptığını anlmaya çalışacağız bunu yapmak için de aşağıda görüldüğü gibi bir heat map'in nasıl oluşturulduğunu anlayacağız bu heat map bana CNN'in classification'ı yaparken inputun hangi kısımlarına odaklandığını gösteren bir heat map.

```
In [37]: M show_heatmap(avg_acts)
```

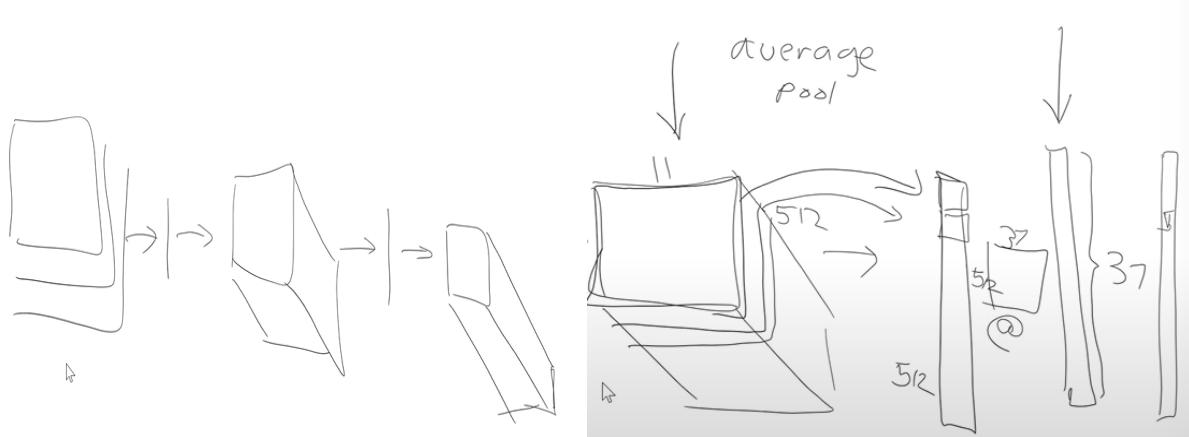


Bu heatmap'i sıfırdan elde edeceğiz. Bunun için neredeyse hiç fastai kullanmayacağız. Bu yüzden biraz yeni görünebilir, anlamayabilirsin dert değil, notebook'a git dene yanlış, geri gel tekrar dinle falan diyor, ben bi dinleyeceğim sadece, zaten CNN mantığını biliyorum Jeremy'den de dinlemiş olayım.

Dakika 1:07:00'da başlıyor. 1:34:00'a kadar zaten bildiğim CNN mantığını anlatıyor.

Peki HeatMap'i nasıl oluşturacağız?

Sonuçta olan şey şu RGB image input olarak giriyor convolutional layerlar sonucun $11 \times 11 \times 512$ boyutunda bir map elde ediliyor bunu fully connected layer'a çevirmek için average pooling kullanıyor 11×11 pooling yapıyor ve sonuçta 512 outputlu bir layer elde ediyor ve daha sonra fully connected layers ile 37 outputlu bir layer ile sonlanıyor.



Heat map'i elde etmek için kullanılan fikir şu: 512'lik fc layer'ın her bir elemanı neyi temsil eder? Amacımız pet breeds'i ayırmak olduğu için 512 tane pet feature'ı temsil eder, image'in tamamına bakılınca ilgili feature'ın detect edilme seviyesini temsil eder.

Ondan önceki layerda ise $11 \times 11 \times 512$ bir map tensorumuz var, bunun her 11×11 'lik kısmı bir filter'in sonucunu veriyor, 11×11 'lik map'in her bir unit'i de ilgili filter'in resmin ilgili kısmında o feature'ın ne kadar detect edildiğini gösteriyor.

E bizim amacımız hangi feature'ın ne kadar detect edildiği değil amacımız, genel olarak resmin hangi kısımlarında feature'ların daha çok detect edildiği olacak o zaman $11 \times 11 \times 512$ 'lik map'in average pooling'ini 11×11 'lik kısımlar için değil de 512lik kısım için alırsak sonuçta elimizde 11×11 bir tensor kalacak bunun her elemanı resmin ilgili bölgesinde ne kadar çok feature detect edildiğini temsil edecek. İşte heatMap'i böyle elde edebiliriz.

Bunun kodlamasını da gösteriyor, gerekirse bakabilirsin.

Ethics and Data Science

Önümüzdeki derste Generative Models'e bakacağımız yani bir text veya image veya video yaratabilecek modellere bakacağımız. Bunlara girmeden önce ethics'den bahsetmeyi doğru görmüş.