# OVERVIEW

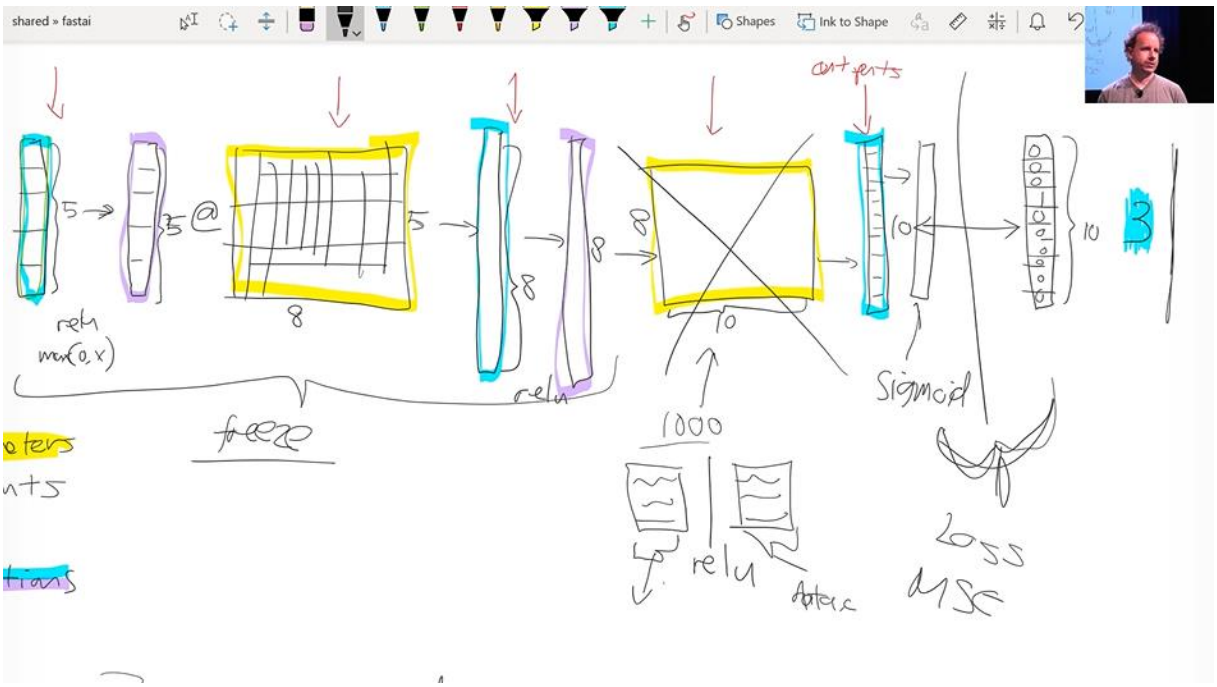
Bu derse kadar yukarı tırmandık ve uygulamaları gördül (Computer Vision, NLP, Tabular Data, Collaborative Filtering) şimdi ise geriye doğru gideceğiz, artık detayları anlayacağız yani modellerin nasıl çalıştığını.

Bunun için Colloborative Filtering ile başlayacağız, bu dersin sonunda daha önce yapamadığımız bir şey öğrenmeyeceğiz, onun yerine teoriyi anlayacağız.

Ki zaten ben andrew'in derslerini aldığım için muhtemelen anlatılanları ben biliyor olacağım, ancak yine de bakalım. Belki kod kısımlarını bilmiyor olabilirim, yine de bir bakış atalım.

Özellikle regularization'dan bahsedeceğiz. Önceki modellerimizi geliştirmek için bazı şeyler öğrenebiliriz.

# Geçen Dersin Sonunda Baktığımız Computer Vision Yapısını Bakalım:




Biz bir model için resnet34 kullandığımızda yapılan şey, resnet34'ün sonunda bulunan weight matrix'i atıyor, resnet34'ün sonunda 1000 dimensionlı bir weight matrix var diye düşün, bunun sebebi ImageNet dataseti için 1000 class tahmin edilmeye çalışılması dolayısıyla benim uygulamamda bu layer useless.

Bunu atarım yerine kendi datasetime göre, 2 yeni weight matrix koyar arasına da relu layer koyar.  İlkinin size'ı için bir default değer var, ancak ikinci weight matrix'in boyutu datasetime göre belirlenir. More specificly data.c değerine göre belirlenir, yani kaç tane class'ım olduğuna göre veya regression için kaç tane dependent variable için prediction yapmaya çalıştığıma göre.

Daha sonra eklenen layerların dışındakileri freeze ederiz ve rough training'i gerçekleştiririz, böylece sadece son eklenen weight matrixleri update ederiz.

Şunu unutma resnet34'ün ilk layerları simple features detect ederken sonlara doğru gittikçe, göz kafa tekerlek vesaire gibi çok daha spesifik feature'lar detect edilir bu yüzden uygulamamız için son layerları olduğu gibi kullanmak pek kullanışlı olmayabilir, yani ben bir medical diagnosis yapıyorsam modelimin son layerda insan kafası araması çok anlamsız.

Freeze işleminden sonra son layerları bir süre eğittikten sonra, modelin performansı iyileşecek ve diyeceğiz ki hmm bu iyi. Daha sonra artık tüm modeli train etmek istiyoruz, ancak şunu biliyoruz ki bu modelin başlardaki layerları zaten low level feature'ları efsane şekilde detect ediyor, ben bu weightleri çok bozmak istemiyorum, buna karşın sonlardaki layer'ı daha yeni ekledim çok kabaca bir eğitim yaptım eminim ki daha fazla eğitim payı var, ayrıca sona yakın diğer layerlar da yukarıda bahsettiğim gibi çok spesifik feature'lara bakıyor, bunları da daha fazla eğitip kendi uygulamamın feature'larını bulacak şekilde modifiye etmek isterim.

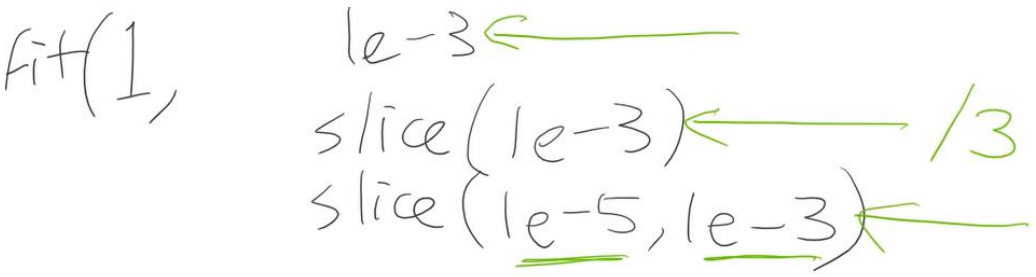
İşte tam da bu sebepten dolayı modelin farklı layerlarına farklı learning rate değerleri veriyoruz. Örneğin başlardakine 1e-5 verirken sonlara 1e-3 veriyoruz. Böylece başlardaki weightler fine-tuning'den çok etkilenmezken, sonlara gittikçe weightler daha radikal değişimlere uğrayacak.

Bu olaya Using Discriminative Learning Rates denir.

# How do we use discriminative learning rates in FASTAI?

Fit function'ı içine girilen ilk parametre number of epochs.

Daha sonra learning rate'i aşağıda görüldüğü gibi farklı şekillerde yazabiliriz. Tek bir değer verebiliriz veya slice içinde tek bir değer verebiliriz veya slice içinde 2 değer verebiliriz. Bunların hepsinin anlamı farklı.




- İlki tüm layerları aynı learning rate ile eğit demek.

- 2. Gösterimde son layers(son eklenen 2 layer) 1e-3 lr değeri ile eğitilirken kalan layerların hepsi bunun 3'te biri ile eğitilir.

- Son gösterimde ise final layers(son eklenen 2 layer) 1e-3 lr değeri ile eğitilirken ilk layerlar ise 1e-5 ile eğitilir arada kalan layerlar ise equally spaced learning rate alır, ancak buradaki equally space logaritmik olarak hesaplanıyor. Yani arada bir layer daha varsa 1e-4 ile eğitilecek.

Önemli bir nokta şu aslında her layer'a ayrı bir lr vermiyoruz bunun yerine her bir layer grubuna ayrı bir lr veriyoruz.

Layer grupları default olarak şöyle ayrılır, son eklenen 2 layer bir grup. Kalan layerların yarısı bir grup kalan yarısı da bir diğer grup yani totalde 3 grup. Bu ayarlar değiştirilebilir.

Dolayısıyla CNN için default olarak 3 layer grup var. Yani CNN için lr'ı 3. Seçenekteki gibi tanımlarsak, ilk layer grup 1e-5 ile eğitilecek 2. Grup 1e-4 ile ve son eklenen grupta 1e-3 ile eğitilecek.

# Collaborative Filtering Örneğine Dönersek

learn.fit_one_cyle(3, 5e-3) kodunu kullandık çünkü collaborative filtering için tek bir layer var, içinde birkaç farklı parça var ama sonuçta tek layer.

Burada bir kavramdan bahsedelim, layerlar her zaman tam olarak matrix multiplications'dan ve activation'dan oluşmuyor. Ancak buna baya yakın, linear functions'dan oluşuyor. Şimdiye kadar matrix multiplications diye tanımladığımız fonksiyonları "Affine Functions" olarak tanımlamak daha doğru. Yani bundan sonra affine functions dediğimizde bunu matrix multiplication gibi düşünebilirsin, ancak mesela convolution işlemi tam olarak bir matrix multiplication değil direkt 2 matrixin çarpımı şeklinde gösterilemez bu yüzden bunlara affine function diyeceğiz. Yani affine function dediğimizde bir linear function gelsin aklına bu da matrix multiplication'a baya yakın bir şey.

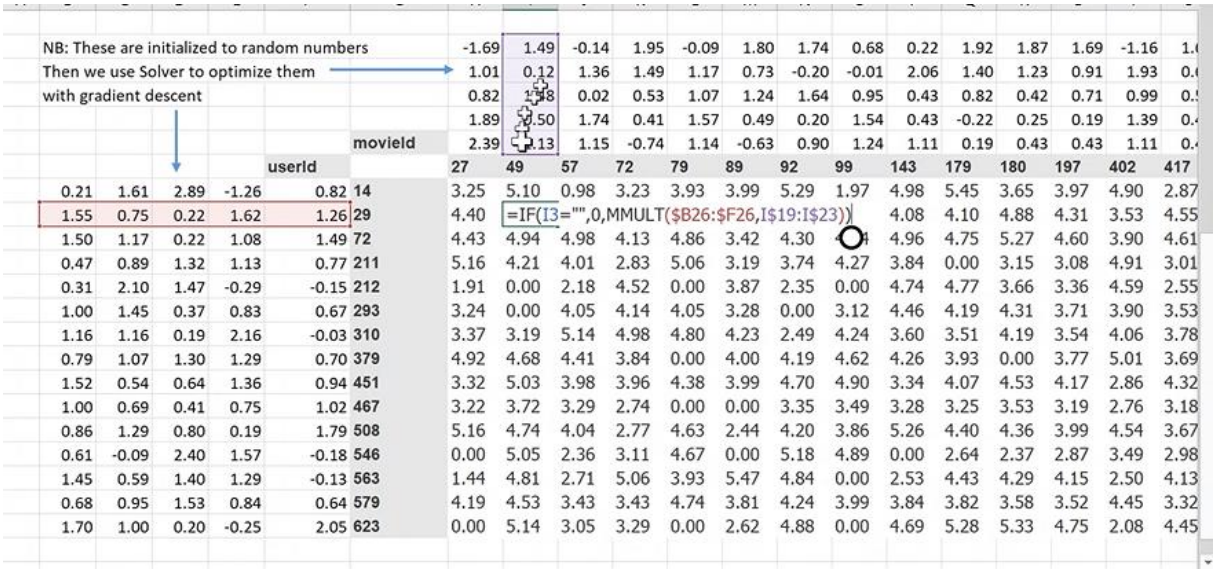
# Embedding

Dediğimizde one-hot encoded matrix ile çarpmanın short ve memory efficient bir way'i olarak düşünebiliriz. Matrix lookup yapmaya deniyor. Tam anlayamadım ama, gerekirse döner bakarım veya araştırırım.

Embedding'i → matrix multiplied by one hot encoded matrix ile mathematically identical bir array lookup aklımıza gelebilir.

Sanıyorum bu yapıyı, veriyi userId(x axes) vs movıId(y axes) array yapısından diğer gösterime çekerken kullanıyoruz, yani her ikisi de tek bir satırda ve karşılık gelen weightler de aynı satırda yazıyor.

# Spesific User and Movie Weights and their meanings?



| NB: These are initialized to random numbers | | | | | | -1.69 | 1.49 | -0.14 | 1.95 | -0.09 | 1.80 | 1.74 | 0.68 | 0.22 | 1.92 | 1.87 | 1.69 | -1.16 | 1.( |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Then we use Solver to optimize them → | | | | | | 1.01 | 0.12 | 1.36 | 1.49 | 1.17 | 0.73 | -0.20 | -0.01 | 2.06 | 1.40 | 1.23 | 0.91 | 1.93 | 0.( |
| with gradient descent | | | | | | 0.82 | 1.18 | 0.02 | 0.53 | 1.07 | 1.24 | 1.64 | 0.95 | 0.43 | 0.82 | 0.42 | 0.71 | 0.99 | 0.5 |
| | | | | | | 1.89 | 0.50 | 1.74 | 0.41 | 1.57 | 0.49 | 0.20 | 1.54 | 0.43 | -0.22 | 0.25 | 0.19 | 1.39 | 0.4 |
| | | | | | movieId | 2.39 | 0.13 | 1.15 | -0.74 | 1.14 | -0.63 | 0.90 | 1.24 | 1.11 | 0.19 | 0.43 | 0.43 | 1.11 | 0.4 |
| | | | | userId | | 27 | 49 | 57 | 72 | 79 | 89 | 92 | 99 | 143 | 179 | 180 | 197 | 402 | 417 |
| 0.21 | 1.61 | 2.89 | -1.26 | 0.82 | 14 | 3.25 | 5.10 | 0.98 | 3.23 | 3.93 | 3.99 | 5.29 | 1.97 | 4.98 | 5.45 | 3.65 | 3.97 | 4.90 | 2.87 |
| 1.55 | 0.75 | 0.22 | 1.62 | 1.26 | 29 | 4.40 | =IF(I3="",0,MMULT($B26:$F26,I$19:I$23)) | | | | | | | 4.08 | 4.10 | 4.88 | 4.31 | 3.53 | 4.55 |
| 1.50 | 1.17 | 0.22 | 1.08 | 1.49 | 72 | 4.43 | 4.94 | 4.98 | 4.13 | 4.86 | 3.42 | 4.30 | 4.27 | 4.96 | 4.75 | 5.27 | 4.60 | 3.90 | 4.61 |
| 0.47 | 0.89 | 1.32 | 1.13 | 0.77 | 211 | 5.16 | 4.21 | 4.01 | 2.83 | 5.06 | 3.19 | 3.74 | 4.27 | 3.84 | 0.00 | 3.15 | 3.08 | 4.91 | 3.01 |
| 0.31 | 2.10 | 1.47 | -0.29 | -0.15 | 212 | 1.91 | 0.00 | 2.18 | 4.52 | 0.00 | 3.87 | 2.35 | 0.00 | 4.74 | 4.77 | 3.66 | 3.36 | 4.59 | 2.55 |
| 1.00 | 1.45 | 0.37 | 0.83 | 0.67 | 293 | 3.24 | 0.00 | 4.05 | 4.14 | 4.05 | 3.28 | 0.00 | 3.12 | 4.46 | 4.19 | 4.31 | 3.71 | 3.90 | 3.53 |
| 1.16 | 1.16 | 0.19 | 2.16 | -0.03 | 310 | 3.37 | 3.19 | 5.14 | 4.98 | 4.80 | 4.23 | 2.49 | 4.24 | 3.60 | 3.51 | 4.19 | 3.54 | 4.06 | 3.78 |
| 0.79 | 1.07 | 1.30 | 1.29 | 0.70 | 379 | 4.92 | 4.68 | 4.41 | 3.84 | 0.00 | 4.00 | 4.19 | 4.62 | 4.26 | 3.93 | 0.00 | 3.77 | 5.01 | 3.69 |
| 1.52 | 0.54 | 0.64 | 1.36 | 0.94 | 451 | 3.32 | 5.03 | 3.98 | 3.96 | 4.38 | 3.99 | 4.70 | 4.90 | 3.34 | 4.07 | 4.53 | 4.17 | 2.86 | 4.32 |
| 1.00 | 0.69 | 0.41 | 0.75 | 1.02 | 467 | 3.22 | 3.72 | 3.29 | 2.74 | 0.00 | 0.00 | 3.35 | 3.49 | 3.28 | 3.25 | 3.53 | 3.19 | 2.76 | 3.18 |
| 0.86 | 1.29 | 0.80 | 0.19 | 1.79 | 508 | 5.16 | 4.74 | 4.04 | 2.77 | 4.63 | 2.44 | 4.20 | 3.86 | 5.26 | 4.40 | 4.36 | 3.99 | 4.54 | 3.67 |
| 0.61 | -0.09 | 2.40 | 1.57 | -0.18 | 546 | 0.00 | 5.05 | 2.36 | 3.11 | 4.67 | 0.00 | 5.18 | 4.89 | 0.00 | 2.64 | 2.37 | 2.87 | 3.49 | 2.98 |
| 1.45 | 0.59 | 1.40 | 1.29 | -0.13 | 563 | 1.44 | 4.81 | 2.71 | 5.06 | 3.93 | 5.47 | 4.84 | 0.00 | 2.53 | 4.43 | 4.29 | 4.15 | 2.50 | 4.13 |
| 0.68 | 0.95 | 1.53 | 0.84 | 0.64 | 579 | 4.19 | 4.53 | 3.43 | 3.43 | 4.74 | 3.81 | 4.24 | 3.99 | 3.84 | 3.82 | 3.58 | 3.52 | 4.45 | 3.32 |
| 1.70 | 1.00 | 0.20 | -0.25 | 2.05 | 623 | 0.00 | 5.14 | 3.05 | 3.29 | 0.00 | 2.62 | 4.88 | 0.00 | 4.69 | 5.28 | 5.33 | 4.75 | 2.08 | 4.45 |

Yukarıda görüldüğü gibi her bir user için ve movie için 5 ayrı parametre öğreniliyor, sonuçta bu parametreler somut feature'lara denk gelmeli ki modelimiz gerçek değerleri elde edebilsin, yani eğitim sonucunda model feature'lar öğrenmiş oluyor.

Mesela her bir user weight'i mesela kullanıcının romantik, aksiyon, nolan filmleri vesaire gibi film özelliklerini ne kadar sevdiğini temsil ederken benzer şekilde her bir movie weight de movienin içinde bu özellikleri barındırıp barındırmadığını öğrenmiş oluyor.

Böylece bu iki vektörün çarpımı, personal taste vektörü ile – movie features vektörünün çarpımı oluyor böyle bir çarpımın user rate'i vermesi hiç de anlaşılmaz değil.

Ancak bazı filmler vardır ki komedi olmamasına rağmen film o kadar iyidir ki sadece komedi severler bile o filmi izleyince dayanamayıp iyi puan vermişlerdir, işte bu modelle bu filmleri yakalayamayız, benze şekilde bazı kullanıcılar vardır ki diğerlerine göre bariz daha yüksek veya alçak puanlar verir, modelimizin bu movie ve user bias'i kapatabilmesi için gereken tek şey bias term. Her bir user embedding vektörü ve movie embedding vektörüne bir de bias term eklendiği zaman bariz daha iyi bir model elde ederiz.

Yani bias term ile şöyle bir statement'i formülüze edebiliyoruz, komedi sever userlar komedi filmlerini çok severler unless sözkonusu movie maide'nin altın günü değilse veya söz konusu user 234 Id'li hiçbir şeyi beğenmeyen user değilse gibi.

Bias ile her movie için overall bir iyi/kötü film score'u belirlemiş oluyoruz ayrıca user için de genel olarak yüksek mi alçak mı puanlar verdiğini score'lamış oluyoruz böylece hesabımız çok daha accurate oluyor.

# Some Questions

1. **When we load a pre-trained model, can we explore the activation grids to see what they might be good at recognising?**

Evet bu yapılabilir ve nasıl yapılacağını haftaya göreceğiz.

2. **Fit methodunun ilk parametresi nedir?**

Number of epochs, yani bu sayı 10 ise eğitim sırasında modele her inputu 10 kez göstermiş olacağız eğer komplex bir modelimiz varsa (high number of parameters) ve yüksek bir learning rate'imiz varsa modelimizin overfit etme ihtimali artar. Model training error'u 0 yapacak global minimum'a/veya çok yakınına converge eder ancak buna karşın validation error artar.

Eğer epoch number=1 ise modelin overfit etmesi imkansız diyor.

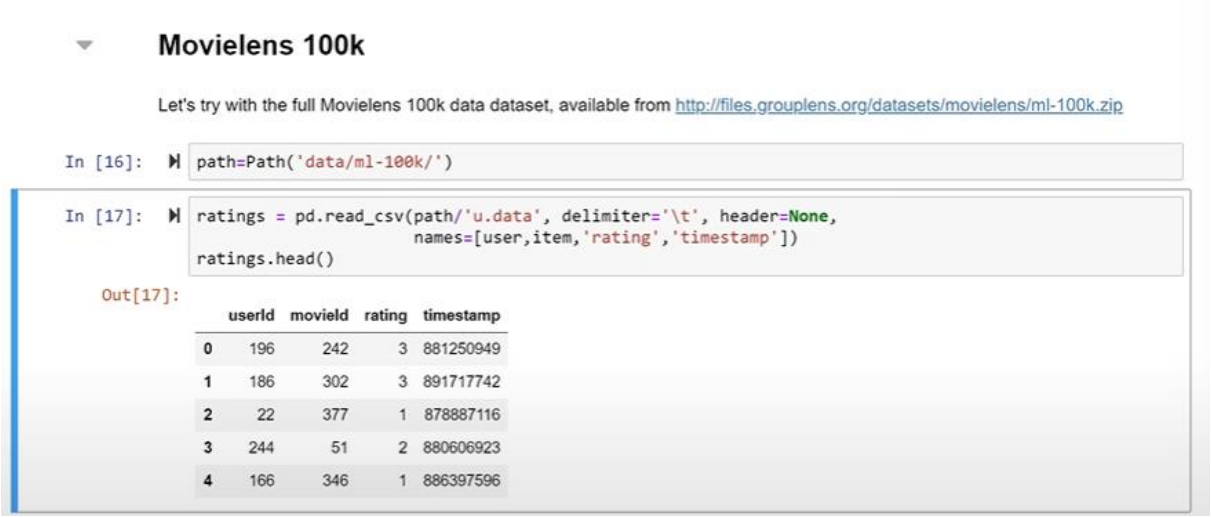
3. **What is a affine function?**

An affine function is a linear function. Birşeyleri çarpıp topluyorsak buna affine function deriz. Bundan daha komplex bir tanıma ihtiyacımız yok.

Bir affine functionı diğerinin üstüne koyarsak, elimizde yeni bir affine functiondan fazlası olmaz. Linear layerların toplamı gibi düşün. Yani linear bir layer'ın aktivasyonunu affine functions ile tanımlarız, zaten bildiğim şeyler.

Affine-Relu-Affine-Relu-Affine şeklinde layerları birleştirirsem işte o zaman complex feature'ları detect edebilirim, ve bir deep NN'imiz var diyebiliriz.

# Şimdi Colloborative Filtering Notebook'una Geri Dönelim:



## Movielens 100k

Let's try with the full Movielens 100k data dataset, available from http://files.grouplens.org/datasets/movielens/ml-100k.zip

In [16]:
```
path=Path('data/ml-100k/')
```

In [17]:
```
ratings = pd.read_csv(path/'u.data', delimiter='\t', header=None,
                      names=[user,item,'rating','timestamp'])
ratings.head()
```

Out[17]:

| | userId | movieId | rating | timestamp |
|---|---|---|---|---|
| 0 | 196 | 242 | 3 | 881250949 |
| 1 | 186 | 302 | 3 | 891717742 |
| 2 | 22 | 377 | 1 | 878887116 |
| 3 | 244 | 51 | 2 | 880606923 |
| 4 | 166 | 346 | 1 | 886397596 |

Performans değerlendirmesi yapmak istiyorsak, original MovieLens datasını kullanmalıyız biz de bunu yapacağız, ancak bu data biraz eski olduğu için güncel verilerden ayrı bazı işlemler yapacağız.

Öncelikle data bir csv file'da yer alıyor ama örneğin delimeter olarak comma yerine tab kullanmışlar bu yüzden read_csv içinde bunu belirttik, ayrıca header row koymamışlar bunu da belirtiyoruz, ayrıca son olarak da header olmadığı için column name'leri kendimiz söylüyoruz.

Bu noktadan sonra elde edilen ratings datasının head kısmına yukarıdaki gibi bakabiliriz.

Ayrıca hangi movie Id'nin hangi filme denk geldiğini aşağıdaki gibi başka bir csv file'dan okuyabilirim.

```
In [18]: movies = pd.read_csv(path/'u.item',  delimiter='|', encoding='latin-1', header=None,
                              names=[item, 'title', 'date', 'N', 'url', *[f'g{i}' for i in range(19)]])
         movies.head()
```

Out[18]:

| | movieId | title | date | N | url | g0 | g1 | g2 | g3 | g4 | ... | g9 | g10 | g11 | g12 | g13 | g14 | g15 | g1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | Toy Story (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Toy%20Story%2... | 0 | 0 | 0 | 1 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 2 | GoldenEye (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?GoldenEye%20(... | 0 | 1 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 3 | Four Rooms (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Four%20Rooms%... | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 3 | 4 | Get Shorty (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Get%20Shorty%... | 0 | 1 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 4 | 5 | Copycat (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Copycat%20(1995) | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Burada encoding kısmını latin-1 olarak belirtiyoruz yoksa unicodeDecodeError alırız, eskiden utf-8 unicode standardı yoktu, artık var.

Her bir genre için farklı bir sütun var, bir film birden fazla genre'ya dahil olabiliyor.

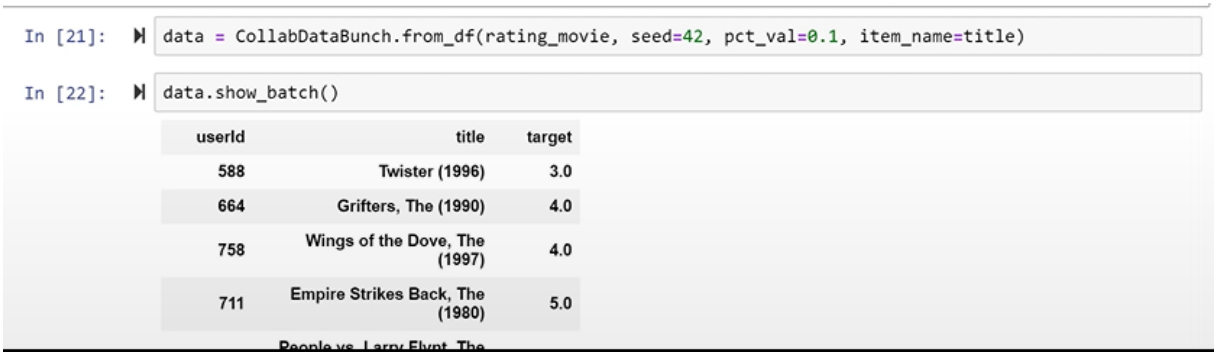
İlk başta elde ettiğim rating dataframe'ine bir de title başlığı eklemek istiyor olabilirim, bunun için pandas'ın merge fonksiyonu kullanacağım:

```
In [20]: rating_movie = ratings.merge(movies[[item, title]])
         rating_movie.head()
```

Out[20]:

| | userId | movieId | rating | timestamp | title |
|---|---|---|---|---|---|
| 0 | 196 | 242 | 3 | 881250949 | Kolya (1996) |
| 1 | 63 | 242 | 3 | 875747190 | Kolya (1996) |
| 2 | 226 | 242 | 5 | 883888671 | Kolya (1996) |
| 3 | 154 | 242 | 3 | 879138235 | Kolya (1996) |
| 4 | 306 | 242 | 5 | 876503793 | Kolya (1996) |

Bu noktadan sonra bu dataframe'i kullanarak kendime collaborative filtering'de kullanmak üzere bir databunch elde edeceğim.



```
In [21]: data = CollabDataBunch.from_df(rating_movie, seed=42, pct_val=0.1, item_name=title)

In [22]: data.show_batch()
```

| userId | title | target |
|---|---|---|
| 588 | Twister (1996) | 3.0 |
| 664 | Grifters, The (1990) | 4.0 |
| 758 | Wings of the Dove, The (1997) | 4.0 |
| 711 | Empire Strikes Back, The (1980) | 5.0 |

from_df ile rating_movie dataframe'inden 10%'luk bir validation set ayıyoruz.

Ayrıca default olarak CollabDataBunch sırasıyla userId, movieId, rating, 3 columnluk bir dataframe kullandığımızı varsayıyor fakat burada item olarak movie Id yerine title_column'unu kullanmak istiyoruz bu yüzden item_name=title ifadesini de ekledik.

Show batch dediğimizde databunch'ın userId ve tittle'ı input olarak belirlediğini buna karşılık rating'i de target/output olarak belirlediğini görebiliyoruz.

## Collaborative Filtering Learner

Uygulamam için gerekli databunch'ı elde ettiğime göre geriye kalan şey learner'ı oluşturmak ve eğitmek.

Bu noktada iyi bir skor elde etmek için elimizden geleni yapacağız bunun için öncelikle son katmana 0-5 arasında bir sigmoid eklenecek, daha doğrusunu sigmoidi collab_learner otomatik olarak ekliyor biz sadece sınırını belirtiyoruz, böylece modelimizin kaçla kaç arasında çıktı vermesi gerektiğini öğrenmesi gerekmeyecek enerjisini daha anlamlı şeylere harcayacak.

Fakat farkındaysa y_range'i 0 ile 5 arası yerine 0-5.5 arası verdik bunun sebebi sigmoid'in maximuma yakınsaması ancak tam 5 değerini hiç almaması buna karşılık bir sürü film 5 rating'ini almış durumda, bu yüzden rangi'i 5.5'e çekiyoruz ki modelimiz rahatlıkla 5 outputunu verebilsin.

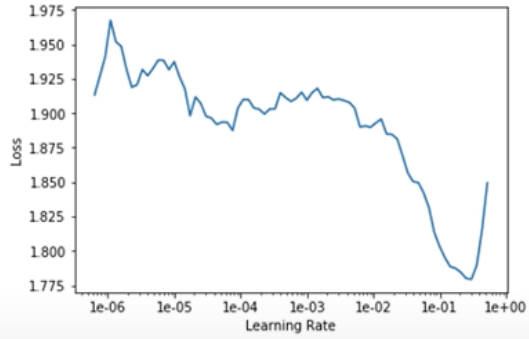
```
798                Sabrina (1954)        4.0

In [23]:  y_range = [0,5.5]

In [24]:  learn = collab_learner(data, n_factors=40, y_range=y_range, wd=1e-1)

In [17]:  learn.lr_find()
          learn.recorder.plot(skip_end=15)

          LR Finder is complete, type {learner_name}.recorder.plot() to see the graph.
```

Yapılan bir diğer trick ise weight decay, wd termini eklemek. Bu konuya da çok yakında geleceğiz.

Ayrıca kaç faktör istediğimizi de belirtiyoruz, number of factors = width of the embedding matrix yani her bir user için veya movie için kaç tane feature olsun onu belirliyoruz. Bu feature'lara LATENT FACTORS deniyor. Yani n_factors yerine embedding_size ismini kullanmak daha mantıklı olabilirdi ancak bu kullanım daha yaygın.

Learner'ı oluşturduktan sonra sırada lr_find() var burada Jeremy'nin rule of thumb'ına göre genelde steepest slope'u buluyorduk, bir başkasının rule of thumb'ı ise minimumu bulup 10x geri gelmekti ancak burada 5e-3 seçilmiş ikisi de değil, jeremy diyor ki steepest slope'u buldum 10x ilerisi 10x gerisini denedim en iyi sonucu bu verdi, o yüzden 5e-3 seçtim.

```
In [17]:  learn.lr_find()
          learn.recorder.plot(skip_end=15)

          LR Finder is complete, type {learner_name}.recorder.plot() to see the graph.
```




```
In [25]:  learn.fit_one_cycle(5, 5e-3)

          Total time: 00:33
          epoch  train_loss  valid_loss
          1      0.938132    0.928146    (00:06)
          2      0.862458    0.885790    (00:06)
          3      0.753191    0.831451    (00:06)
```

Sonuçta 5 epoch ile 0.813588 gibi bir MSE valid loss elde ettik bu değer gayet iyi.

# İyi bir sonuç elde ettik, notebook'un devamında bazı interpretations yapılmış, bunlara bakacağız:

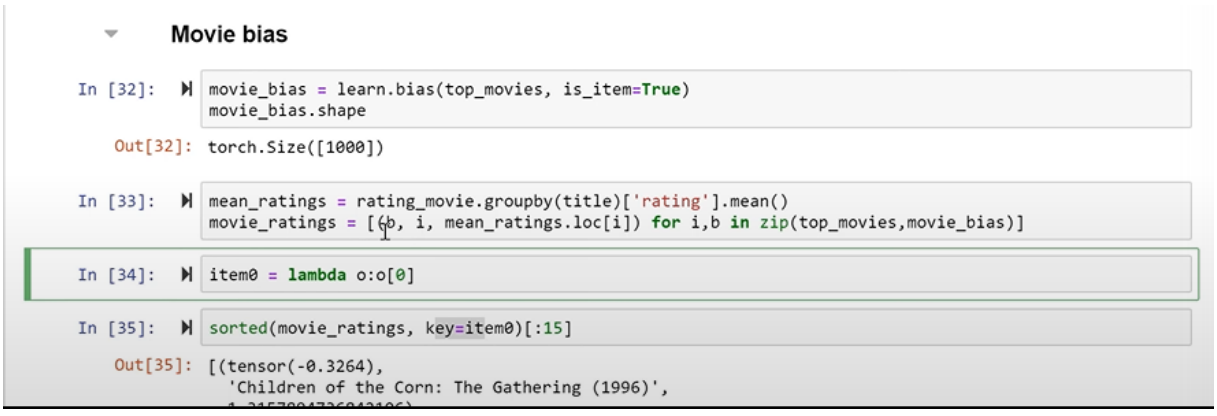
```
In [26]: learn.model

Out[26]: EmbeddingDotBias(
           (u_weight): Embedding(944, 40)
           (i_weight): Embedding(1654, 40)
           (u_bias): Embedding(944, 1)
           (i_bias): Embedding(1654, 1)
         )

In [27]: g = rating_movie.groupby(title)['rating'].count()
         top_movies = g.sort_values(ascending=False).index.values[:1000]
         top_movies[:10]

Out[27]: array(['Star Wars (1977)', 'Contact (1997)', 'Fargo (1996)', 'Return of the Jedi (1983)', 'Liar Lia
         r (1997)',
                'English Patient, The (1996)', 'Scream (1996)', 'Toy Story (1995)', 'Air Force One (1997)',
                'Independence Day (ID4) (1996)'], dtype=object)

Movie bias

In [32]: movie_bias = learn.bias(top_movies, is_item=True)
         movie_bias.shape

Out[32]: torch.Size([1000])
```

Öncelikle en fazla rating alan  movie'leri bulalım, yukarıda görüldüğü gibi bunlar Star Wars, Contact, Fargo gibi filmler. Top_movies içine 1000 tane top rated movie'yi yerleştirdik.

Daha sonra, learn.bias ile train edilmiş learner objesine top_movies içindeki item'ların bias değerlerini elde etmek istiyorum.

İs_item=True dediğimizde movies için bias değerlerini alırız, false dediğimizde user için bias değerlerini alırız.

Sonuçta bu bias 1000'lik bir tensor olacak çünkü top_movies içinde 1000 adet film var her bir item'a bir bias denk geliyor.



```
Movie bias

In [32]: movie_bias = learn.bias(top_movies, is_item=True)
         movie_bias.shape

Out[32]: torch.Size([1000])

In [33]: mean_ratings = rating_movie.groupby(title)['rating'].mean()
         movie_ratings = [(b, i, mean_ratings.loc[i]) for i,b in zip(top_movies,movie_bias)]

In [34]: item0 = lambda o:o[0]

In [35]: sorted(movie_ratings, key=item0)[:15]

Out[35]: [(tensor(-0.3264),
           'Children of the Corn: The Gathering (1996)',
```

Daha sonra titles'ı rating mean'ine göre grupladık etc… en nihayetinde filmleri bias değerlerine göre sıraladık. Örneğin en düşük bias'ten yukarı doğru 15 filmi ve bias değerlerini gösterebiliriz. Burada bu filmler, feature'larının yanında kötü filmler olmasıyla dikkat çekiyor, yani feature'ları iyi de olsa bu filmin bias'i çok düşük bu yüzden bir çok userdan iyi rating alamıyor.

```
 1.3157894736842106),
(tensor(-0.3241),
 'Lawnmower Man 2: Beyond Cyberspace (1996)',
 1.7142857142857142),
(tensor(-0.2799), 'Island of Dr. Moreau, The (1996)', 2.1578947368421053),
(tensor(-0.2761), 'Mortal Kombat: Annihilation (1997)', 1.9534883720930232),
(tensor(-0.2703), 'Cable Guy, The (1996)', 2.339622641509434),
(tensor(-0.2484), 'Leave It to Beaver (1997)', 1.8409090909090908),
(tensor(-0.2413), 'Crow: City of Angels, The (1996)', 1.9487179487179487),
(tensor(-0.2395), 'Striptease (1996)', 2.2388059701492535),
(tensor(-0.2389), 'Free Willy 3: The Rescue (1997)', 1.7407407407407407),
(tensor(-0.2346), 'Barb Wire (1996)', 1.9333333333333333),
(tensor(-0.2325), 'Grease 2 (1982)', 2.0),
(tensor(-0.2294), 'Beverly Hills Ninja (1997)', 2.3125),
(tensor(-0.2223), "Joe's Apartment (1996)", 2.2444444444444445),
(tensor(-0.2218), 'Bio-Dome (1996)', 1.903225806451613),
(tensor(-0.2117), "Stephen King's The Langoliers (1995)", 2.413793103448276)]
```

```
In [36]: sorted(movie_ratings, key=lambda o: o[0], reverse=True)[:15]
```

```
Out[36]: [(tensor(0.6105), "Schindler's List (1993)", 4.466442953020135),
 (tensor(0.5817), 'Titanic (1997)', 4.2457142857142856),
 (tensor(0.5685), 'Shawshank Redemption, The (1994)', 4.445229681978798),
 (tensor(0.5451), 'L.A. Confidential (1997)', 4.161616161616162),
 (tensor(0.5350), 'Rear Window (1954)', 4.3875598086124405),
 (tensor(0.5341), 'Silence of the Lambs, The (1991)', 4.28974358974359),
 (tensor(0.5330), 'Star Wars (1977)', 4.3584905660377355),
 (tensor(0.5227), 'Good Will Hunting (1997)', 4.262626262626263),
 (tensor(0.5114), 'As Good As It Gets (1997)', 4.196428571428571),
```

Benzer şekilde tersine sıralarsak da yine bias değeri yüksek olan filmler, feature'larının yanısıra net iyi filmler, yani ben romantik bir film sevmeyen biri olsam da titanic iyi film, model benim titatiniğe vereceğim rating'i hesaplarken işte bu bias'i devreye sokuyor.

Burada bias'in düşük ancak rating'in iyi olduğu filmler şunu gösteriyor, bu feature'lar ile aslında bias olmasa filmin daha iyi rating alması bekleniyor, belki kadrosu çok iyi, yönetmeni çok iyi, bu yüzden model bias haricinde yüksek bir rating hesaplıyor ancak sonuç böyle çıkmıyor bu yüzden bias'i düşük, yani bias'i düşük olanlar genelde kötü filmler olmakla birlikte, sanıyorum aslında bias'i düşük olması demek feature'lara bakıldığında beklenenden düşük rating alması demek.

Benzer şekilde bias'i yüksek olanlarda feature'lara bakıldığında beklenenden yüksek score alması demek. Yani eğer bir film için bias=0 ise o filmin feature'ları cuk oturmuş demektir. Ama mesela titaniğin bias'i yüksek demekki filmin feature'ları dikkate alındığında filmden bu performans beklenmiyor, o yüzden model bu açığını bias değerini yükselterek kapatıyor.

Bias değerlerine bakabildiğimiz gibi, weight değerlerine de bakabiliriz. Yine top movies'i temsil eden weightlere bakacağız, n_factors değerini 40 atamıştık yani her bir movie için 40 feature olacak yani 40 weight, e 1000 tane top movie seçtik o zaman 1000x40'lık bir torch tensor outputu bekleriz.

**Movie weights**

```
In [37]: movie_w = learn.weight(top_movies, is_item=True)
         movie_w.shape
Out[37]: torch.Size([1000, 40])

In [38]: movie_pca = movie_w.pca(3)
         movie_pca.shape
Out[38]: torch.Size([1000, 3])
```

Ancak 40 feature'ı intuitive olarak algılamak oldukça zor, yani bir film için 40 farklı feature'ı ben belirleyemem bunun için fastai'ın pytorch tensorleri için çalışan PCA methodunu kullancağız, bildiğin pca dimension sayısını azaltmak için kullanılıyor, elimde 40 feature'lı 1000 örnek vardı, bunu pca ile 3 featura'a çektim böylece 1000 örneği 3 boyutlu uzayda vesaire yazdırabilirim, ve bu feautre'lara baktığım zaman daha intuitive bir anlayış geliştirebilirim.

Feature'lardan birine bakalım, bu feature'ı max ve min olan 10 örnek aşağıdaymış, tam olarak neyi temsil ettiği belli değil ama sanki kült filmleri temsil ediyor gibi, yani bir filmin kült olup olmadığını ölçüyor olabilir.

```
In [40]: sorted(movie_comp, key=itemgetter(0), reverse=True)[:10]
Out[40]: [(tensor(1.0834), 'Chinatown (1974)'),
          (tensor(1.0517), 'Wrong Trousers, The (1993)'),
          (tensor(1.0271), 'Casablanca (1942)'),
          (tensor(1.0193), 'Close Shave, A (1995)'),
          (tensor(1.0093), 'Secrets & Lies (1996)'),
          (tensor(0.9771), 'Lawrence of Arabia (1962)'),
          (tensor(0.9724), '12 Angry Men (1957)'),
          (tensor(0.9660), 'Some Folks Call It a Sling Blade (1993)'),
          (tensor(0.9517), 'Ran (1985)'),
          (tensor(0.9460), 'Third Man, The (1949)')]

In [41]: sorted(movie_comp, key=itemgetter(0))[:10]
Out[41]: [(tensor(-1.2521), 'Jungle2Jungle (1997)'),
          (tensor(-1.1917), 'Children of the Corn: The Gathering (1996)'),
          (tensor(-1.1746), 'Home Alone 3 (1997)'),
          (tensor(-1.1325), "McHale's Navy (1997)"),
          (tensor(-1.1266), 'Bio-Dome (1996)'),
          (tensor(-1.1115), 'D3: The Mighty Ducks (1996)'),
          (tensor(-1.1062), 'Leave It to Beaver (1997)'),
          (tensor(-1.1051), 'Congo (1995)'),
          (tensor(-1.0934), 'Batman & Robin (1997)'),
          (tensor(-1.0904), 'Flipper (1996)')]
```

Bu feature'ların ne olduğunu biz bilmiyoruz bunu unutma bu yüzden, yeni bir film çıktığında cold start problem söz konusu çünkü, o feature'ları biz yerleştiremiyoruz, modelin feature'ları bulması için önce userların feedback'ine

ihtiyacı var. Benzer şekilde yeni user'ın feature'larını öğrenmek için de user'ın ratinglerine ihtiyaç var.

## Collab Learner Function Under the Hood:

```
In [24]: learn = collab_learner(data, n_factors=40, y_range=y_range, wd=1e-1)
```

```
def collab_learner(data, n_factors:int=None, use_nn:bool=False, metrics=None,
                   emb_szs:Dict[str,int]=None, wd:float=0.01, **kwargs)->Learner:
    "Create a Learner for collaborative filtering."
    emb_szs = data.get_emb_szs(ifnone(emb_szs, {}))
    u,m = data.classes.values()
    if use_nn: model = EmbeddingNN(emb_szs=emb_szs, **kwargs)
    else:      model = EmbeddingDotBias(n_factors, len(u), len(m), **kwargs)
    return CollabLearner(data, model, metrics=metrics, wd=wd)
```

Gördüğümüz gibi collablearner input olarak databunch'ı alıyor etc. sonuçta bir EmbeddingDotBias modeli yaratıyor ve bu modeli CollabLearner'a besleyerek return ediyor o halde, EmbeddingDotBias'in içine bakalım:

```
class EmbeddingDotBias(nn.Module):
    "Base model for callaborative filtering."
    def __init__(self, n_factors:int, n_users:int, n_items:int, y_range:Tuple[float,float]=None):
        super().__init__()
        self.y_range = y_range
        (self.u_weight, self.i_weight, self.u_bias, self.i_bias) = [embedding(*o) for o in [
            (n_users, n_factors), (n_items, n_factors), (n_users,1), (n_items,1)
        ]]

    def forward(self, users:LongTensor, items:LongTensor) -> Tensor:
        dot = self.u_weight(users)* self.i_weight(items)
        res = dot.sum(1) + self.u_bias(users).squeeze() + self.i_bias(items).squeeze()
        if self.y_range is None: return res
        return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]
```

Gördüğümüz gibi bu class bir nn.Module'dan inherit edilmiş, tüm Pytorch modelleri ve layerları bir nn.Module'dur!

Forward propagation görüldüğü gibi bu class'ın içinde tanımlanan forward methodu ile yapılıyor.

Mesela ne yapıyor, self.u_weight(users) ile items weighti çarpıyor peki bu weightler nerede tanımlandı? Hem yukarıda init içine bakarsak self.u_weight'in bir embedding olduğunu görüyoruz.

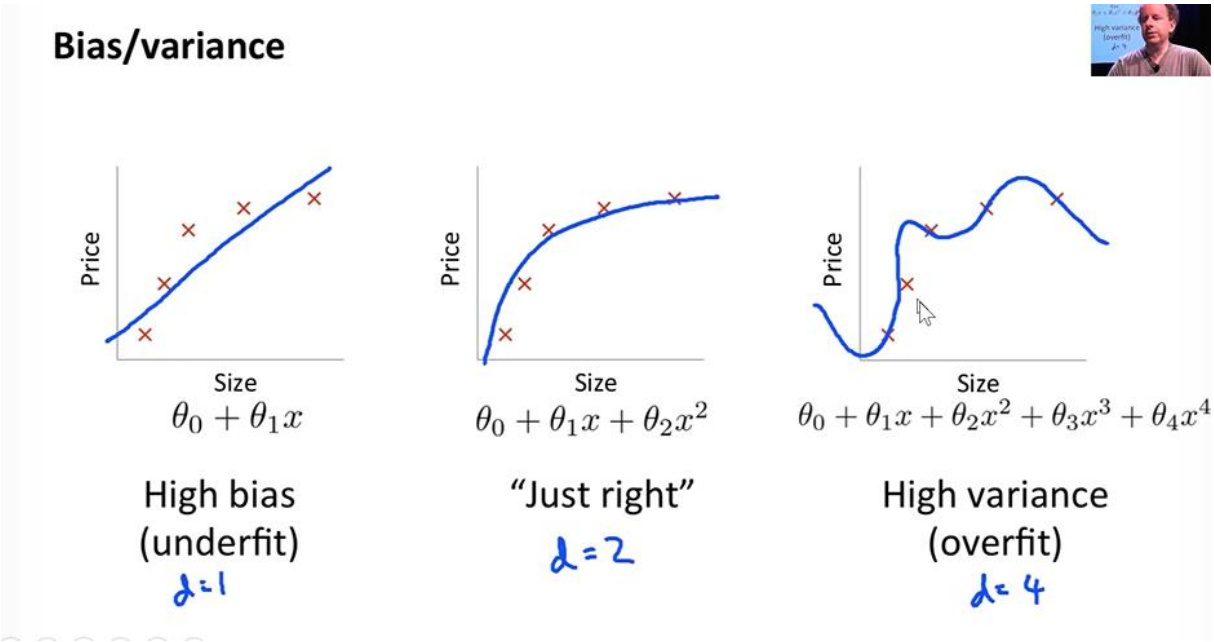
Burada yaratılan embeddingler aslında yine bir pytorch nn.Module olacak.

Buralar 1:05:00 civarı oluyor daha detayı gerekirse bakabilirsin.

# Collab Learner Function Under the Hood, Weight Decay:

Weight decay is a type of regularization.

Bunu anlamadan önce Andrew'in dersinden bias/variance kavramına bi bakış atalım.




Models with more parameters tends to look like the 3rd one above.

Traditional statistics'in önerdiği şey ise, madem ki çok parametreli modeller overfitting'e sebep oluyor o zaman biz de çok parametreli modelleri kullanmayalım. Çünkü böyle modeller datayı generalize edemiyor. Sadece training dataya mükemmel oturuyor, kalanlarda saçmalıyor.

Ancak bu yanlış bi algı, tamam ana amacımız fonksiyonumuzun daha az komplex olması, ancak daha az parametre kullanmak bunu elde etmenin yollarından yalnızca bir tanesi. Diyelim ki benim 10k parametrem var ama 9999 tanesi 10^-9 değerini almış yani bunlar ha var ha yok aslında 1 parametrem varmış gibi oluyor, o halde kompleks fonskiyonu daha az komplex yapmak için illaha da daha az parametre kullanmak zorunda değiliz bunun için başka yollar var (REGULARIZATION)

Peki neden daha fazla parametre kullanmak istiyorum? Because more parameters mean more non-linearities, more interactions, more curvy bits, and real life is full of curvy bits, real life not look like this (yukarıdaki ilk linear şekili gösteriyor)

Ancak aynı zamanda, modelin gerekenden fazla curvy olmasını da istemiyoruz or more interacting than necessary işte bu yüzden LET'S USE LOTS OF PARAMETERS AND THEN PENALIZE COMPLEXITY.

Bunu yapmanın bir yolu L2 regularization yani, parametrelerin karesini toplayarak loss function'a dahil etmek böylece fonksiyonun çok kompleks olmasını engellemiş olacağız. Ancak burada şöyle bir risk doğuyor, dengeyi nasıl tutturacağız, tüm weightlerin karelerinin toplamını minimize edersek, bunlar sıfıra converge etmesin? O zaman elimizde model falan kalmaz. Bu dengeyi nasıl yakalıyoruz?

Cost functionı şöyle düşün: Prediction Loss + Regularization Loss. Biz bu toplamı minimize etmek istiyoruz, eğer regularization loss'u minimize etmek için tüm weightleri = 0 dersek, bu kez prediction loss uçacaktır, ancak yine de öyle bir durumla karşılaşabiliriz ki regularization loss çok büyüktür bu yüzden, bunu sıfıra çekmek için weightleri sıfır yapmak en iyi optimiziasyon olabilir çünkü diğer ihitimalde prediction loss 0.1 gelecek anacak regularization loss 150 gelecek mesela. Onun yerine prediction loss 100 yapıp regularization loss 0 almak modelin işine gelecek.

İşte bu dengeyi kurmak için regularization term'in başına weight decay ekliyoruz bir başka deyişle REGULARIZATION PARAMETER = wd. Bu term sayesinde, demin bahsedilen weightleri aşırı düşürme olayını engellemeyi hedefliyoruz. Wd=0 yaparsak regularization ortadan kalkar model overfitte yatkın hale gelir, benzer şekilde wd eğer çok büyürse bu kez de model gerekli komplexityi sağlayamaz ve underfit eder.

Peki wd değeri ne olmalı? Genelde bu değer için 0.1 yani 1e-1 seçmek mantıklıdır.

Buna rağmen FASTAI defualt'u olarak wd değeri 0.01 olarak seçilmiştir, bunun sebebi şu: genelde 0.1 mükemmel çalışsa da bazı nadir durumlarda 0.1 wd değeri, modeli underfit ettirerek, ne kadar süre train etsek de iyi bir bir performans almamızı engeller, buna karşın 0.01 değeri genelde gerekenden küçük de olsa, ortaya çıkacak overfitting etkisini early stopping ile çözebileceğimiz için, bu değer default olarak atanmış, yani daha verimli bir yaklaşım yerine daha güvenli bir yaklaşım izlenmiş.

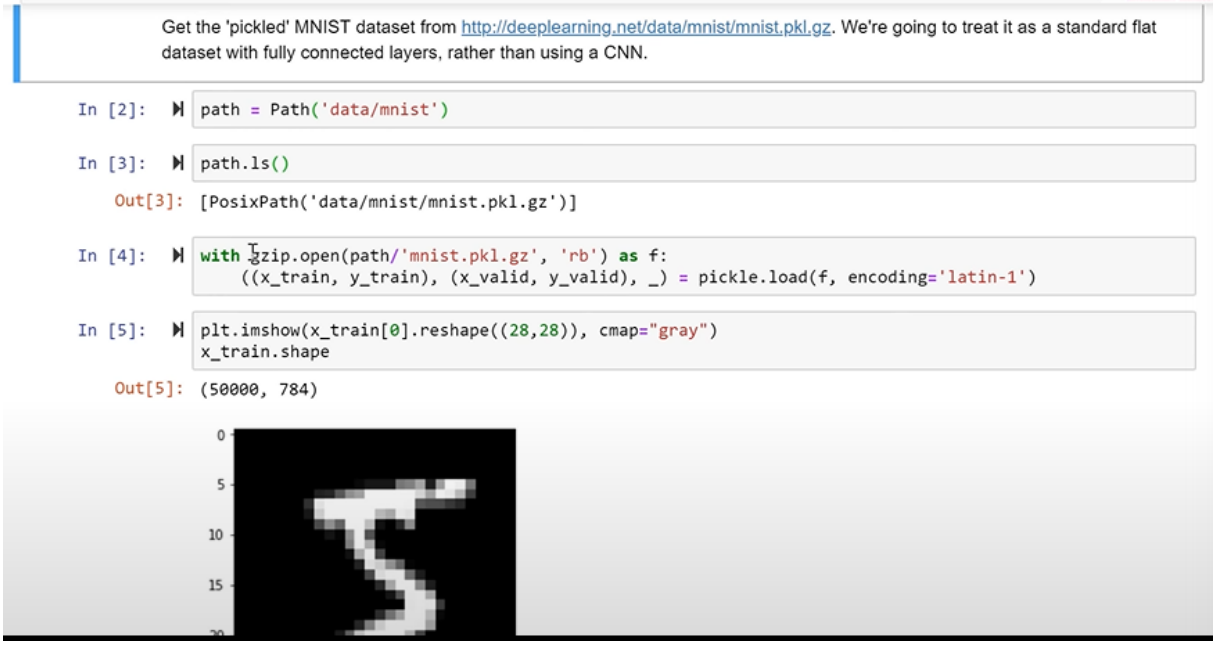
Ancak bu noktadan sonra bunu bildiğimize göre, learners için 0.1 wd değerini denemekte fayda var. Genelde işe yarar.

Şunu da belirtmek gerek, bu wd değerini her zaman learnerlar içerisinde görmeyebiliriz, ancak bütün learnerlar eninde sonunda Learner constructor'ını çağıracak ve bu constructor'ın bir wd değeri vardır.

Yani her türlü learner'ı yaratırken wd değerini girebiliriz, spesific learner için bu değer görünmese de onun çağıracağı Learner constructor'ı içinde wd vardır.

## Şimdi MNIST dataset ile uygulama yapalım:

Bu uygulamayı standard fully connected layers ile yapacağız, ConvNet ile değil. Deeplearning.net MNIST datasetini python pickle file olarak sağlıyor böylece dataset images direkt numpy arrays olarak elde ediliyor.




Gzip dosyasından train ve validation setleri çekiyoruz.

Sonuçta x_train'e baktığımızda 50k,784 bir numpy array görüyoruz bu da şu demek 50k tane 28x28 px images'dan oluşan bir training setimiz var plt.imshow ile bunu yukarıdaki gibi yazdırabiliriz.

Biz modelimiz için inputların flattened versioyununu kullanacağız yani bir input 28x28 lik bir matrix değil 784'lük bir vektör olarak ele alınacak.

```
In [6]: x_train,y_train,x_valid,y_valid = map(torch.tensor, (x_train,y_train,x_valid,y_valid))
        n,c = x_train.shape
        x_train.shape, y_train.min(), y_train.max()

Out[6]: (torch.Size([50000, 784]), tensor(0), tensor(9))
```

Train ve validation setlerim şuanda numpy arrays formunda bunları map ile tensore çeviriyoruz.

2. derste bias için 1'lerden oluşan bi column oluşturmuştuk, mse functiın tanımlamıştık, matrix multiplication ile y_hat hesaplamıştık vesaire, bunları burada yapmayacağız onun yerine pytorch fonksiyonlarını kullanacağız.

In lesson2-sgd we did these things ourselves:

```
x = torch.ones(n,2)
def mse(y_hat, y): return ((y_hat-y)**2).mean()
y_hat = x@a
```

Now instead we'll use PyTorch's functions to do it for us, and also to handle mini-batches (which we didn't do last time, since our dataset was so small).

Ayrıca geçen sefer minibatch konusunu hiç dahil etmemiştik çünkü datasetimiz çok küçüktür bu kez mini-batches oluşturacağız.

Pytorch fonksiyonu olan TensorDataset ile iki tensoru alıyoruz ve sonuçta bir dataset oluşturuyor, hatırlarsan dataset indexleyebileceğin ve x ve y value get edebileceğin bir classtı.

```
In [7]: bs=64
        train_ds = TensorDataset(x_train, y_train)
        valid_ds = TensorDataset(x_valid, y_valid)
        data = DataBunch.create(train_ds, valid_ds, bs=bs)

In [8]: x,y = next(iter(data.train_dl))
        x.shape,y.shape

Out[8]: (torch.Size([64, 784]), torch.Size([64]))
```

Daha sonra databunch.create ile train_ds ve valid_ds datasetlerinden dataloaders yaratılacak, dataloadersın yaptığı şey de şu: dataloaders'a 2. Elemanı veya 5. Elemanı ver demiyorsun da next elemanı ver diyorsun o da sana bs sayısı kadar elemanlı bir mini-batch veriyor.

Bu batchleri yukarıdaki son kod ile görebiliriz, data.train_dl ile next ve iter kullanarak sıradaki mini-batch'i x ve y içine alabiliyorum shape'lere baktığım zaman 64 adet MNIST resmini ve karşılık gelen labelını aldığımı görebilirim.

Şimdi bi logistic regressor yaratmak istiyoruz(yalnız tam olarak logistic regressor değil 10 tane output var, daha çok 10 tane logistic regressor yaratmak istiyoruz gibi veya bildiğimiz basit bir NN, hatta bu bence linear regression modeli çünkü output'a bir sigmoid tanımlanmadı, belki loss function tanımlayınca sigmoid tanımlanmış olabilir ama tam çözemedim. Evet daha sonra açıkladı loss'u cross entropy tanımlayınca pytorch loss'u hesaplamadan önce bir Softmax layer ekliyor!!!) bunu daha önce elimizle tensor işlemleri tanımlayarak ve loss function tanımlayarak yapmıştık şimdi bunun yerine pytorch'un nn.Module'unu kullanacağız.

Aşağıda görüldüğü gibi nn.Module'dan bir Mnist_Logistic sublcass'ı oluşturuyoruz, consturctor olarak supoer constructor'ı çağırdık, ayrıca yanına Mnist_Logistic nn.Module'ümüzün bir de lin adında bir attribute'u olsun istiyoruz. Bu lin attribute'u da bir nn.Linear module olacak, bunun yaptığı ney bunun yaptığı 784'lük bir input ve 10'luk bir output için bir linear model kurmak.

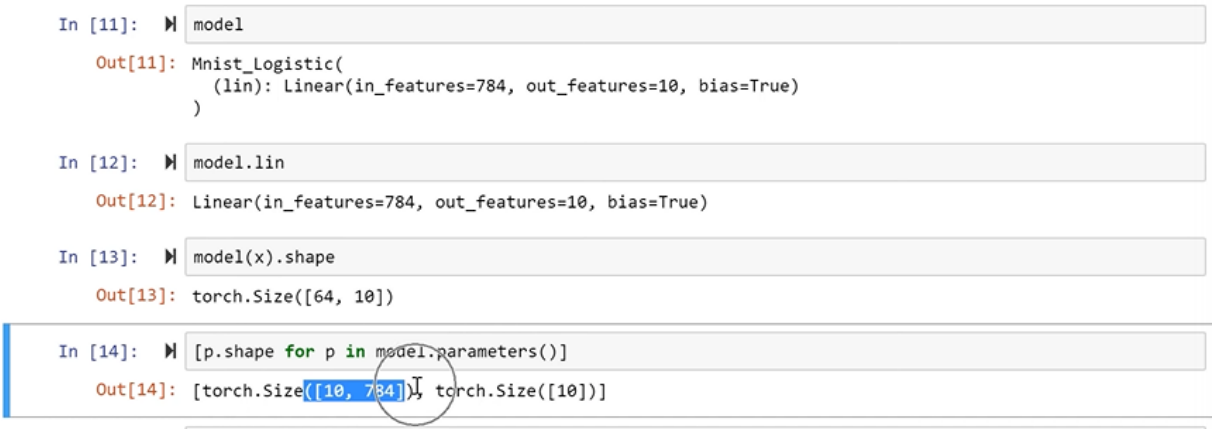
```
In [9]:  class Mnist_Logistic(nn.Module):
             def __init__(self):
                 super().__init__()
                 self.lin = nn.Linear(784, 10, bias=True)

             def forward(self, xb): return self.lin(xb)

In [10]: model = Mnist_Logistic().cuda()

In [11]: model
```

Ayrıca logistic regression modelimiz içindeki forward methodu ile xb minibatch'ini input olarak veriyorum ve output olarak xb minibatch'ine karşılık gelen output'u return etmesini bekliyorum.

Sonuçta bu şekilde logistic regression modelimizi oluşturmuş oluyoruz. Model = Mnist_Logistic().cuda() işlemi ile weight matrixleri GPU'ya yerleştiriyoruz.

Şimdi modelimize baktığımız zaman nn.Module class'ı sayesinde direkt modlimizin yapısını görebiliyoruz, 784 tane inputu alıp 10 tane output veren biaslı bir linear model görüyoruz.

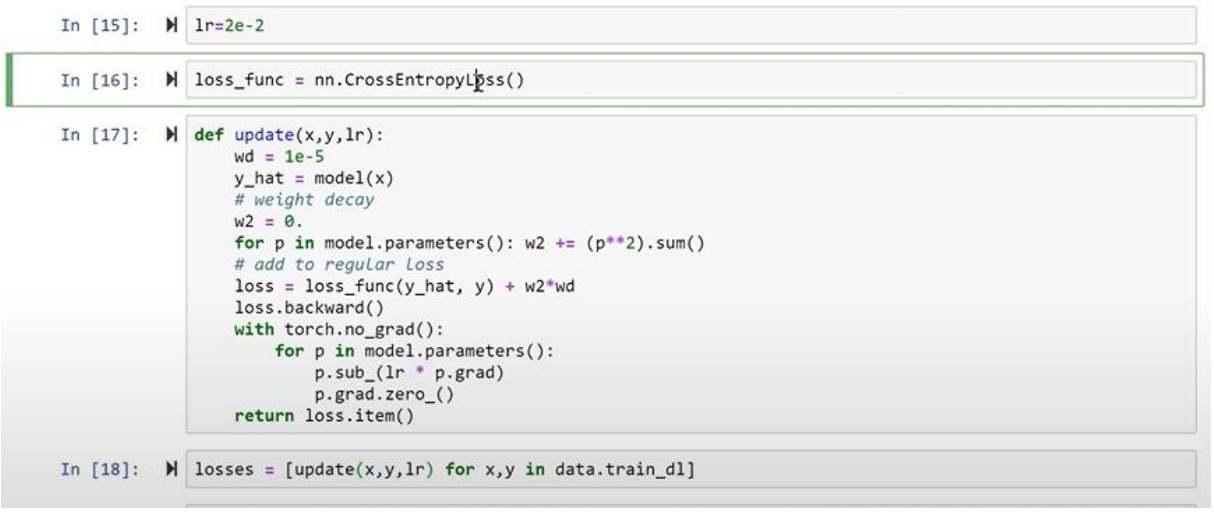
model(x).shape dediğimizde forward propagation sonucunu görüyoruz, yani model her forward propagation hesapladığında 64 tane örnek için output hesaplayacak her output 10'luk bir vektör olduğu için 64x10 boyutu beklenen.




Bunun yanında modelimin parametrelerini de yukarıdaki gibi görebiliyorum. Parametre olarak 10x784 ve 10 boyutunda 2 farklı tensor görebiliyorum, bunlar neyi temsil ediyor? 784 inputu 10 outputa bağlayacak 10x784 boyutunda bir weight matrix'i temsil ediyor diğer 10'luk tensor de her bir output uniti için eklenen bias term'den fazlası değil.

Şimdi bir lr tanımlıyorum, ayrıca nn.CrossEntropyLoss() ile regression için gerekli loss function'ımı tanımlıyorum.

Daha sonra bir update fonksiyonu içerisinde gradient descent step'ini tanımlayacağız.

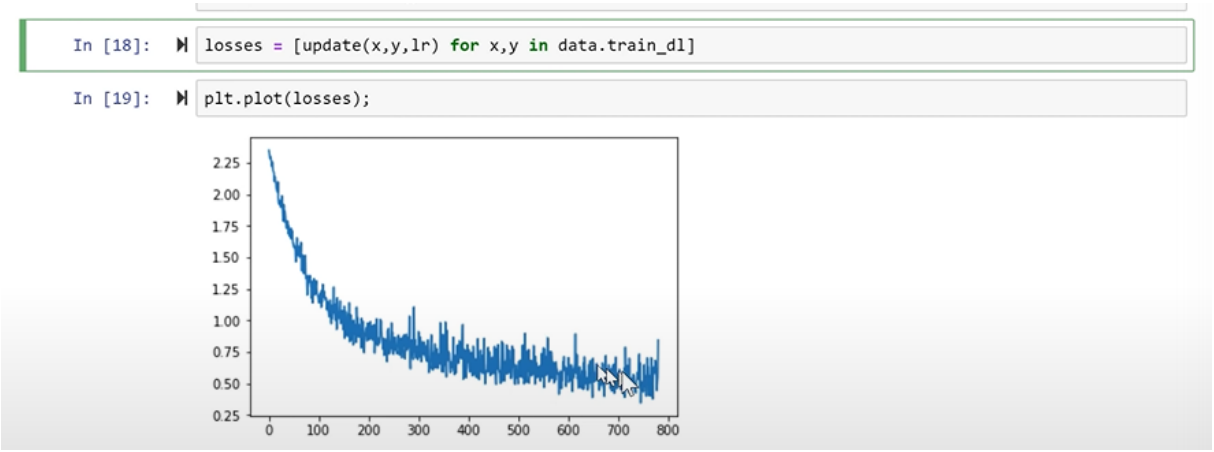



y_hat'i hesaplamak için model(x) dememiz yeterli direkt forward propagation sonucu y_hat'e atılıyor daha sonra, tanımlanan loss function'a y_hat ve y verilerek loss hesaplanıyor.

Daha sonra loss.backward ile model parametrelerinin gradients'i hesaplanıyor ve with torch.no_grad(): kısmında da daha önce yaptığımız gibi weight update yapılıyor.

Ayrıca yukarıda wd ve w2 parametreleri de dikkatini çeksin. W2'nun yaptığı şey her bir mini-batch update için parametrelerin karesini tutmak. Böylece bu değeri wd parametresi ile birlikte loss'a dahil ediyoruz. Yani L2 regularization'ı uygylamış oluyoruz, veya weight decay uygulamış oluyoruz.

Aşağıda görüldüğü gibi, dataloader'dan sıradaki minibatch'i alarak her minibatch için update methodunu çağırıyorum, ve 50k'lık datasetim bir kez gezilene kadar yani bir epoch boyunca eğitimim yapılmış oluyor, update fonksiyonunda loss'u return ettiğim için sonuçta loss'u plot ettirebiliyorum.




Görüyorsun ki loss azaldıkça yani hedefe yaklaştıkça zikzakların genliği artıyor bu yüzden learning rate'i giderek düşürmek mantıklı olabilir.

# Şimdi L2 Regularization'ın Nasıl Çalıştığını Anlamaya Çalışalım:

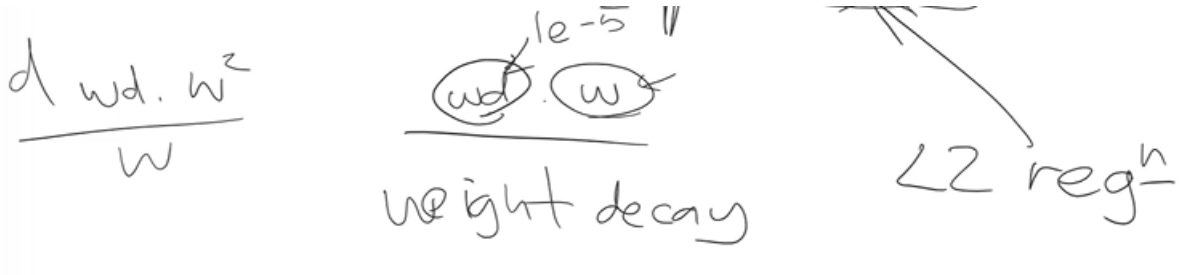
Loss functiona eklenen weightlerin karesi term'i sonuçta bulunan parametrelere etki ediyor çünkü, gradient step sırasında bu term'in türevi de hesaba katılıyor.

Bu term gradient step sırasında hesaba nasıl katılıyor buna bakalım, böylece L2 regularization'ın aslında ne yaptığını anlamış olacağız.

Gradients'i hesaplarken, ilk termin gradients'i+2. Termin gradients'i şeklinde hesaplayabiliriz, ilk kısmı zaten biliyoruz 2. Kısmı yani regularization term'e odaklanalım.


2. term'in gradient'ine baktığımız zaman, summation'ı elimine edersek weight'e göre türev alındığında sonuça 2wd*w kalır yani bir constant*w kalıyor yani her gradient update sırasında weightlerden kendi değerinin mesela 10'da birini çıkarmasını sağlamış oluyoruz bu yüzden buna weight decay deniyor. Gradient update yaparken hem prediction hatasından gelen update yapılıyor hem de kendi değerinin wd oranına göre bir kısmı update'e katılıyor.

Andrew'in ML derslerinde de bundan bahsedilmişti sanki oradan bakabilirsin, daha açıklayıcıdır. Kabaca bu.



# Şimdi MNIST dataset uygulamasına kaldığımız yerden devam:

784 inputlu 10 unitli bir modeli nasıl tanımlayacağımızı ve update fonksiyonu gördüğümüze göre şimdi bu yapıya bir layer daha ekleyip bir NN oluşturalım.

```
In [20]:
class Mnist_NN(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin1 = nn.Linear(784, 50, bias=True)
        self.lin2 = nn.Linear(50, 10, bias=True)

    def forward(self, xb):
        x = self.lin1(xb)
        x = F.relu(x)
        return self.lin2(x)
```

```
In [21]:
model = Mnist_NN().cuda()
```

```
In [22]:
losses = [update(x,y,lr) for x,y in data.train_dl]
```

Önce 784'e 50'lik bir linear module tanımlayarak 50'lik bir hidden layer tanımladım daha sonra 50'ye 10'luk bir linear model ile 10'luk output istediğimi belirttim.

Şimdi forward içinde linear hesap yapılıyor sonra relu aktivasyonu ekleniyor son oalrak da 2. Linear aktivasyon hesaplanıyor. Yine bir sigmoid göremiyorum, bu model bana hala 10 outputlu bir regression unit gibi görünüyor.

Ayrıca optimizasyon algoritması gibi daha farklı python fonksiyonu da kullanabiliriz. Update fonksiyonu içine aşağıda görüldüğü gibi Adam optimizer tanımlayabiliriz, bundan öncekilerde olarak SGD kullanışmıştı.

Burada yapılan, pytorch'a diyoruz ki model.parameters() parametrelerini Adam kullanarak optimize et artık p'leri tek tek for p in model.parameters() loopu ile update etmek yerine opt.step dediğimizde bir step atılır.

```
In [25]:
def update(x,y,lr):
    opt = optim.Adam(model.parameters(), lr)
    y_hat = model(x)
    loss = loss_func(y_hat, y)
    loss.backward()
    opt.step()
    opt.zero_grad()
    return loss.item()
```

```
In [26]:
losses = [update(x,y,0.001) for x,y in data.train_dl]
```

```
In [27]:
plt.plot(losses);
```

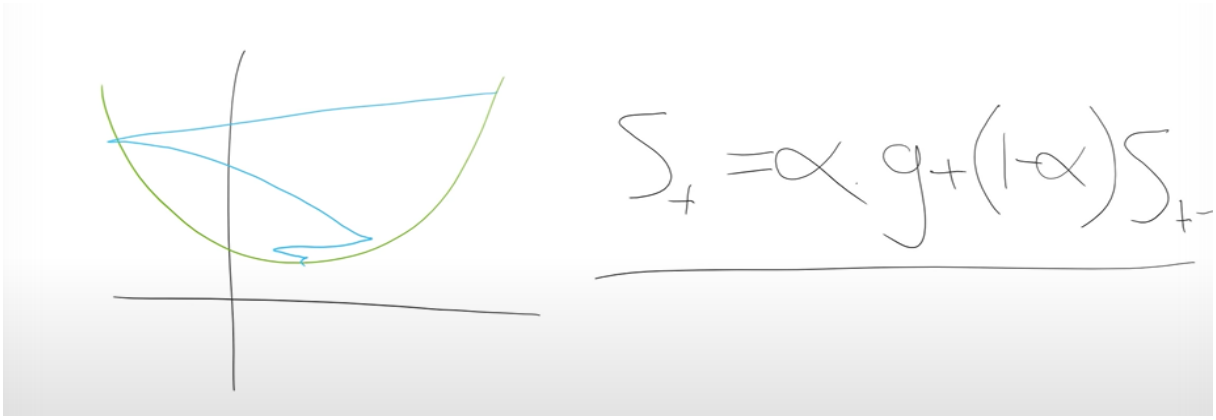
Bu optim.Adam(…) içerisine lr veya weight decay de ekleyebiliriz.

Adam ile learning rate'i ayarladıktan sonra SGD'ye göre daha hızlı bir convergence yakalayabiliriz.

Andrew'in derslerinden zaten Adam'ın ne olduğunu biliyorsun ama Jeremy'den de dinle:

# Adam Optimization

Bildiğimiz gibi Momentum exponentially weighted averages fikrinden yararlanarak optimizasyonu daha iyi bir hale getirir, sıradaki stepi atarken daha önce atılan steplerin ortalamasını da hesaba katar böylece alakasız yönlerde atılan adımların ortalaması 0 gelir ve atılan step direkt olarak hedefe doğru olur. Daha hızlı bir convergence beklenir.




Momemntum'u aşağıdaki şekilde tanımlayabiliriz, ayrıca bir de RMSProp var. Adam ise bu ikisinden de yararlanıyordu.

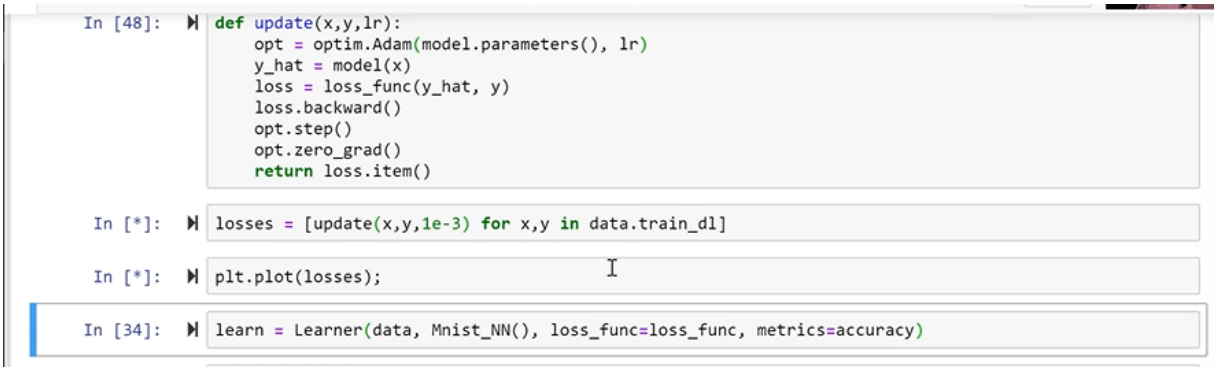
```
In [41]:  def update(x,y,lr):
              opt = optim.SGD(model.parameters(), lr, momentum=0.9)
              y_hat = model(x)
```

Sonuçta Adam ikisindende yararlanarak dynamic learning rate kullanır. Yani atılan adımlara göre diğer atılan adımların boyutu değişir bu da sanki learning rate i değiştirmek gibi. Ancak bu demek değil ki  o zaman learning rate tanımlamamıza gerek kalmadı.

# Biz İşlerimizi Pytorch ile Halletmek Yerine FASTAI kullanıyoruz

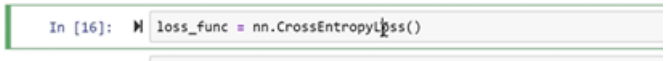
Update metodu tanımlamak onun içinde optimizer, loss, y_hat tanımlamak gradient step attırmak vesaire yerine fastai Learner'larını kullanıyoruz.

Bunun için bildiğimiz gibi aşağıdaki gibi datayı, modeli, loss function'ı ve metrics'i vererek kolayca bir learner tanımlayabiliriz ve daha sonra bu learner ile herşeyi yapabiliriz.




Modeli nasıl tanımladığımızı tekrar hatırlarsak, basit bir nn.Module instance'ı.


Loss function da benzer şekilde:

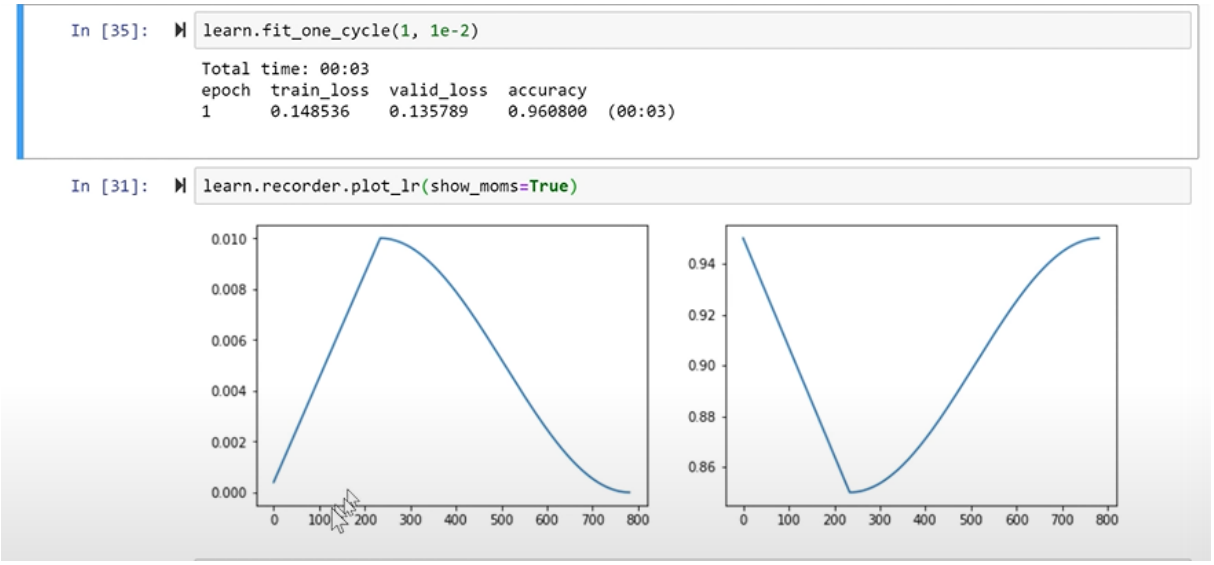



Bunun yanında eminim ki çok kolay bir şekilde custom loss functions da tanımlanabiliyordur.

Learner'ı yarattıktan sonra learn.lr_find() gibi komutlarla lr finder'ı ve daha farklı toolları kolayca learner objesi üzerinden kullanabiliyoruz, ve inan bana bu toolar çok çok büyük performans avantajı sağlıyor.

# Fit_one_cylce ne yapıyor?

Recorder.plot_lr ile learning rate'i minibatch başına plot ettiriyoruz. Learner'lar default olarak adam kullanıyor, ve adam için de bildiğin gibi bir learning rate kullanılıyor. Burada bir epoch eğitim yapılmış ve her batch için learning rate kaydedilmiş sonucu yazdırınca aşağıdaki iki grafiği görüyoruz.




Soldaki grafik learning rate'i gösterirken sağdaki momentum'u gösteriyor.

Learning rate çok küçük bir değerle başlıyor, sürecin yarısında artıyor kalan yarısında azalıyor. Çünkü başlarda cost function'ı düşününce nerede olduğumuzu bilmiyoruz ama çoğunlukla çok engebeli bir arazideyiz bu yüzden eğer büyük bir lr değeri ile başlarsak bu engebeli arazide büyük gradientler ile karşılaşacağız ve optimizasyon algoritması bizim alakasız yerlere uçmamıza neden olacak bu yüzden önce yavaş adımlarla ilerliyoruz ki şu engebeli ne olduğu belli olmayan araziden kurtulalım.

Daha sonra zamanla daha temiz açık alanlara ovalara geleceğiz işte bu noktada hızlanmak isteriz çünkü ovaya baktığımızıda aldığımız slope ile uzun bir adım atmak bizi global minimum'a yaklaştıracak. Global minimum'a yaklaştıkça yine adımlarımızı küçültmek isteriz, bu yüzden lr da giderek azalır ve sonuçta global minimum'un dibinde bir yerde durmak isteriz.

Peki sağdaki grafikte neden momentum ile learning rate ters hareket ediyor. Bunun sebebi de şu momentumu bizim ovada sağa sola gitmemizi engelliyor olarak düşün yani eğer dağı tırmanıyorsak, bi sağa bi sola küçük lr ile küçük küçük adımlar atarsak işler çok yavaş ilerler bu noktada momentum'u devreye sokmak

isteriz ve sağa sola atılan adımlardan kurtuluruz, bunun sonucunda ileri atılan adım da büyür ama zaten küçük adımlar atıyoruz biraz hızlanabiliriz.

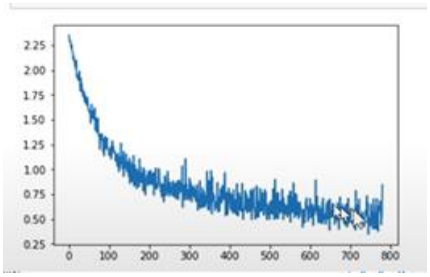
Ancak ovalarda büyük adımlar atarken momentumun büyük olması işimize gelmez çünkü ovada atılan büyük adımı daha da büyütürsek sıkıntı çıkarabilir.

Biliyorum momentum örneğe tam oturmadı bunun için momentum'a tekrar bakıp bu örneği güncellemek lazım, iyi bir intuiton olduğunu düşünüyorum

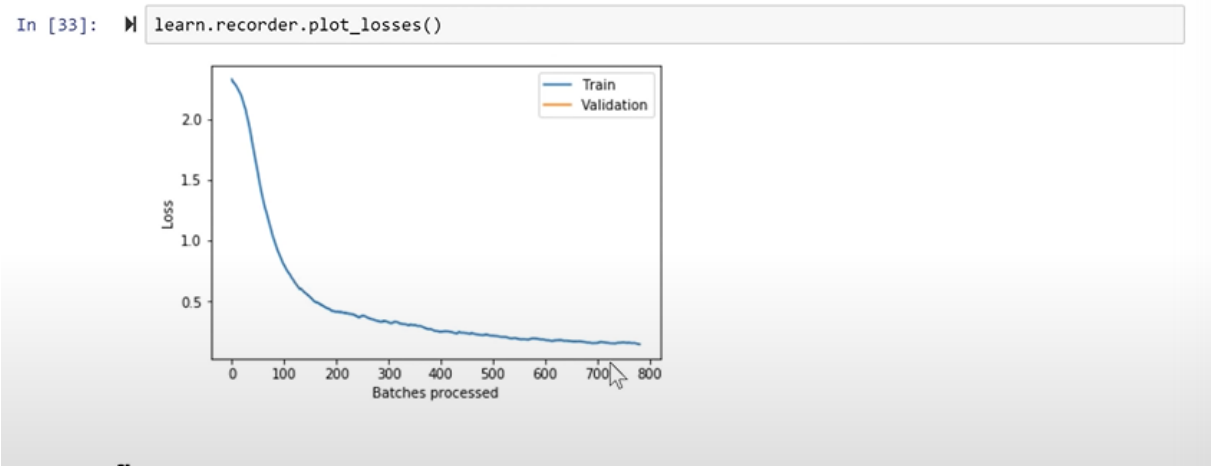
Bu ONE CYCLE FIT optimizasyonu inanılmaz derecede hızlandırıyor belki 10 kat hızlı converge sağlıyor inanılmaz bir şey, son derece yeni bir olgu.

## FASTAI Loss plotları neden Pytorch Modellerinin Plotlarına göre Daha Smooth?

Aynı işlemleri yaptık, adam kullandık vesaire ama pytorch ile yapılan örnekte loss function aşağıdaki gibi görünüyordu:




Buna karşılık aynı modeli FASTAI ile bir learner ile kurunca ve plot_losses dediğimizde aşağıdaki gibi çok daha smooth bir loss plot elde ettik:



Neden böyle? Çünkü fastai gradients update'leri yaparken loss'ları exponentially averages kullanarak kaydediyor bu yüzden daha smooth bir loss görüyoruz. Bu yüzden zigzaglardan kurtuluyoruz. Daha kolay okunabilir bir chart elde ediyoruz.

Tabi dezavantajı da var o da chart birkaç batch geriden geliyor, yani tam olarak güncel değeri değilde birkaç minibatch önce alınan değeri gösteriyor.

# Cross Entropy Loss and Softmax

Single label multiclass classification için genelde softmax output layer ve cross entropy loss kullanılır, ikisinin de ne olduğunu zaten Andrew'in derslerinden biliyorsun. Temelde cross entropy loss'un hesaplama mantığı şu eğer model bir class'ı çok emin şekilde tahmin etmiş ve bilmişse bu örnek için çok küçük bir loss hesaplansın (log ile), eğer emin şekilde yanlış tahmin yaptıysa büyük bir loss gelsin eğer emin olmayan bir şekilde doğru veya yanlış tahmin yaptıysa moderate bir loss gelsin.

| Cat | Dog | Pred(Cat) | Pred(Dog) | X-Entropy | |
|---|---|---|---|---|---|
| 1 | 0 | 0.5 | 0.5 | 0.30 | 0.30 |
| 1 | 0 | 0.98 | 0.02 | 0.01 | 0.01 |
| 0 | 1 | 0.9 | 0.1 | 1.00 | 1.00 |
| 0 | 1 | 0.5 | 0.5 | 0.30 | 0.30 |
| 1 | 0 | 0.9 | 0.1 | 0.05 | 0.05 |
| | | | | 1.66 | |

Elbette bu loss'un hesaplanması için bir şart var bu da pred(class1) + pred(class2) + pred(class K) toplamının 1 olması lazım ki cross entropy anlamlı sonuçlar versin.

İşte her bir class tahmininin toplamının 1 olduğundan emin olmak için de output layer olarak softmax kullanılır. Bunun da yaptığı şeyi bir önceki aktivasyonların exponensiyelini almak ve toplamlarına bölmektir. Bu sayede tüm output unitler 0-1 arasında bir değer verirken toplamları da 1 olmak zorundadır.

| | output | exp | softmax |
|---|---|---|---|
| cat | -2.17 | 0.11 | 0.01 |
| dog | -1.63 | 0.20 | 0.02 |
| plane | -2.75 | 0.06 | 0.01 |
| fish | 2.39 | 10.91 | 0.90 |
| building | -0.16 | 0.86 | 0.07 |
| | | 12.14 | 1.00 |

Pytorch loss olarak cross entropy tanımlarsak otomatik olarak loss hesabı için önce softmax layerı varmış gibi loss hesap yapıyor, bu yüzden yukarıdaki logistic regressor örneğinde softmax aktivasyonu tanımlanmadı.

Bu yüzden eğer modelimizin predictionunu 0-1 arasında beklerken yukarıdak i gibi -2.17, -1.63, 2.39 gibi beklenmedik değerler geliyorsa bunun sebebi, softmax'in loss içinde tanımlanmış olması olabilir prediction içinde tanımlanmamış. Bu yüzden softmax eklemek isteyebiliriz.

Fastai için prediction yaparken genelde bu hesaba katılıyor ve sonuç softmax eklenerek veriliyor ancak eğer custom loss function kullanırsak böyle bir problemle karşılaşabiliriz.

Hayır ben şunu anlamadım neden, modeli yaratırken relu tanımladığımız gibi bir softmax layer tanımlayıp bu karışıklıktan kolayca kurtulmuyoruz ki?