

LESSON 1 – CNN

16. prosince 2018 13:33

FASTAI library

- open source
- postaveno na základě PyTorch (což není jen DL framework ale library která umožňuje psát libovolné gpu-accelerated modely from scratch)
- nicméně pro věci třeba na mobile to nejspíše chce použít tensorflow/keras

[cloud setup – paperspace and fastai](#)

LESSON 1

preparation

- importy
 - předpřipravená data a import
 - [imports – image recognition](#)
- test gpu, cuda dnn
 - torch.cuda.is_available()
 - torch.backends.cudnn.enabled
- files = os.listdir(f'{PATH}valid/cats')[:5]
 - python os library
 - důležité: složky mají definovanou strukturu kterou model očekává ['sample', 'valid', 'models', 'train', 'test1']
- img = plt.imread(f'{PATH}valid/cats/{files[0]}')
- plt.imshow(img);
 - pyplot image read and show
- img.shape
 - shape of image 3 dimension
 - **rank 3 tensor**

model

- **CNN type**
- **pre-trained in ImageNet (1.2 million images and 1000 classes)**
- **resnet34 model - <https://github.com/KaimingHe/deep-residual-networks>**
- training - **všechno FASTAI metody (postaveno na PyTorch)**
 - arch = resnet34
 - architektura modelu
 - data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch,sz))
 - data pro model
 - fastai metody pro konverzi obrázků a potřebné rozlišení (rz)
 - learn = ConvLearner.pretrained(arch,data,precompute=True)
 - model
 - learn.fit(0.01,2)
 - training ve dvou cyklech/jakoby epochách (2x projde celou sadu obrázků)
 - výsledek 98,9% (10 sekund běželo !)
 - epoch trn_loss val_loss accuracy
 - 0 0.047585 0.032429 0.988
 - 1 0.041195 0.029213 0.989
 - training entropy loss
 - validation entropy loss

predictions a evaluate

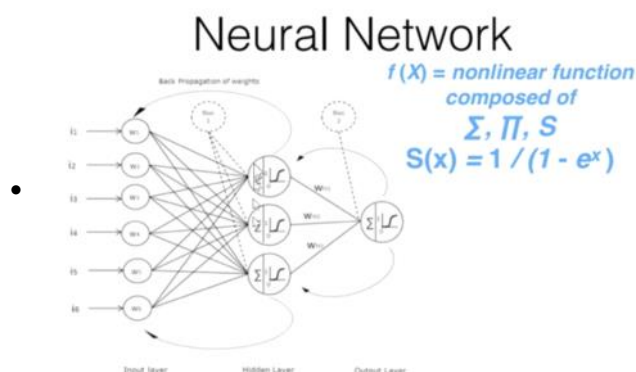
- data.val_y
- data.classes
- log_preds = learn.predict()
 - get predictions
- preds = np.argmax(log_preds,axis=1)
- probs = np.exp(log_preds[:,1])
 - jak pytorch tak fastai knihovny vrací většinou log

- python tak handluje fast numerical program array computation etc
- indexování polí `[:,1]` - bere všechny záznamy z daného pole a ve vnořeném poli druhou hodnotu je to `np.array`
- plotting functions

ploting functions

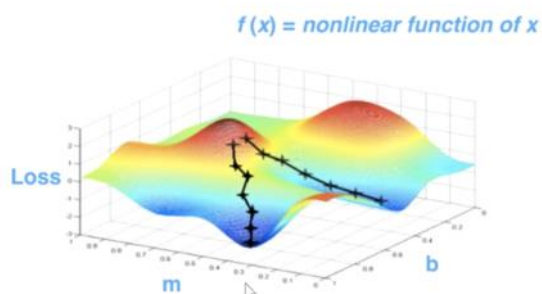
MACHINE AND DEEP LEARNING INTRO

- deep learning is a kind of machine learning, parameters
 - infinitely flexible function
 - **neural network**
 - all-purpose parameter fitting
 - **gradient descent**
 - fast and scalable
 - **GPU based computing**

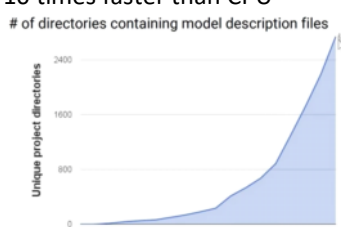


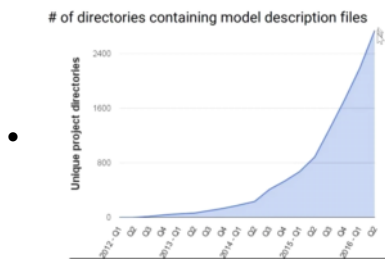
- ML: arthur samuels 1960, ibm mainframe, predicted that vast majority of sw will be written by machines
- **neural network**
 - contains number of simple linear interspersed with num. of nonlinear layers
 - universal approximation theorem - this kind of function can solve any given problem with arbitrarily close accuracy as long enough parameters is enclosed

Gradient Descent



- one or multiple hidden layers - superlinear when adding hidden layers (at some cases) -> deep learning
- **gradient descent**
 - get the set of parameters I have to solve the function -> find slightly different set of parameters to solve the function better
 - finding local minima - but for NN mostly only comparable spaces exists - so the local are also close to global
- **GPU based computing**
 - 10 times faster than CPU





IDEAS where GL brings results and number of areas/packages using DL in GOOGLE

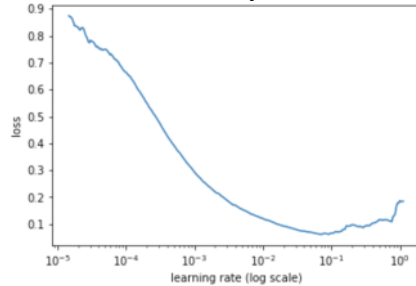


CNN

- convolution - linear operation
 - <http://setosa.io/ev/image-kernels/>
 - takes set of pixels (3x3 or any other 2x2 5x5.. but 3x3 is very usual in DL) and **multiplies them with kernel** matrix (same dimensions)
 - then **results are summed** in one value
 - thus values in kernel determines resolution output that can be focussed on different aspects - sharpnes, blur atd.
- then element-wise **nonlinear function**
 - eg Sigmoid - s-shaped function
 - nowadays mostly used is **ReLU - rectified linear unit**
 - algorithm: take any negative numbers and turn to zero, leave all positive as they are
 - creates arbitrarily complex shapes
- gradient descent (loss function)
 - eg for quadratic function
 - take point
 - calculate quadratic function to understand in which direction to minimize eg which way is down
 - take small step down and check result of original function
 - $x_{n+1} = x_n + \text{derivat}(dy/dx_n) * L$
 - L = learning rate which is a step size (must be small number)
 - repeat til result is minimized
- <https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf>
 - vizualizace toho co CNN rozezná při přidávání vrstev
 - při páté vrstvě rozeznala unicycle wheels a oči ptáků
 - při přidávání vrstev roste rozeznávaný detail (zhruba)**

choosing a learning rate

- one of the most difficult parameters to set, because it significantly affects model performance
- possible approach
 - gradually double learning rate starting from very tiny number eg 10^{-6} - stop when loss function starts increasing
 - vyber hodnotu při které nastal největší nárůst - jinými slovy kde byl dropping loss function největší
- fastai library
 - based on <http://arxiv.org/abs/1506.01186>
 - `learn = ConvLearner.pretrained(arch,data,precompute=True)`
 - `lrf = learn.lr_find()`
 - začne trénovat model ale zastaví se (před 100 procenty) při růstu loss func.
 - v atributu `learn.sched` je uložen learning rate scheduler



- průběh lze pak plotnout `learn.sched.plot()`
- vybraná je 10^{-2} tzn 0.01 (loss is still clearly improving at 10^{-2})

choosing number of epochs

- pokud se trénuje ve velkém množství epoch tak dochází k **overfittingu**
- takže je třeba sledovat accuracy a jak začne růst tak omezit počet epoch

choosing a batch size

- finding z testování
 - u malých datových vzorků je dobré změnit batch size, aby byl každý obrázek procházen "jednotlivě"
 - `bs=2` - 1 dávala error, mají to být mociny 2
- nicméně obecně by měly batch sizes větší
 - jedná se o vzorek na základě něhož se updatují weights (tzn by to měla být reprezentace celého vzorku)
 - <https://forums.fast.ai/t/please-explain-why-batch-size-matters/9045/2>
- `data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch,sz),bs=bs)`

cloud setup – paperspace and fastai

17. prosince 2018 8:46

- registration <https://www.paperspace.com/console>
 - DNN on GPU
 - only Nvidia because of CUDA
- git pull
 - or git clone <https://github.com/fastai/fastai.git>
 - to update fastai stuff
- conda env update
 - update python + conda environment
- https://github.com/reshamas/fastai_deeplearn_part1/blob/master/tools/paperspace.md
 - komplet konfigurace připojení k paperspace (vytvoření konfiguračního file k připojení ve složce ~/.ssh
 - nyní stačí **ssh paperspace**
- jupyter notebook
 - je to na portu 8888

imports – image recognition

17. prosince 2018 8:58

```
# Put these at the top of every notebook, to get automatic reloading and inline
plotting
%reload_ext autoreload
%autoreload 2
%matplotlib inline

from fastai.imports import *
from fastai.transforms import *
from fastai.conv_learner import *
from fastai.model import *
from fastai.dataset import *
from fastai.sgdr import *
from fastai.plots import *

PATH = "../../../data/satellite-imagery/"
sz=64
arch = resnext50 #původně 34
bs=64
```

ploting functions

16. prosince 2018 14:05

```
def rand_by_mask(mask): return np.random.choice(np.where(mask)[0],
min(len(preds), 4), replace=False)
def rand_by_correct(is_correct): return rand_by_mask((preds ==
data.val_y)==is_correct)
def plots(ims, figsize=(12,6), rows=1, titles=None):
    f = plt.figure(figsize=figsize)
    for i in range(len(ims)):
        sp = f.add_subplot(rows, len(ims)//rows, i+1)
        sp.axis('Off')
        if titles is not None: sp.set_title(titles[i], fontsize=16)
        plt.imshow(ims[i])
def load_img_id(ds, idx): return
np.array(PIL.Image.open(PATH+ds.fnames[idx]))

def plot_val_with_title(idxs, title):
    imgs = [load_img_id(data.val_ds,x) for x in idxs]
    title_probs = [probs[x] for x in idxs]
    print(title)
    return plots(imgs, rows=1, titles=title_probs, figsize=(16,8)) if
len(imgs)>0 else print('Not Found.')
def most_by_mask(mask, mult):
    idxs = np.where(mask)[0]
    return idxs[np.argsort(mult * probs[idxs])[:4]]


def most_by_correct(y, is_correct):
    mult = -1 if (y==1)==is_correct else 1
    return most_by_mask(((preds == data.val_y)==is_correct) & (data.val_y
== y), mult)

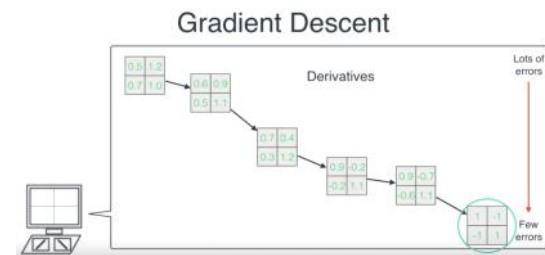
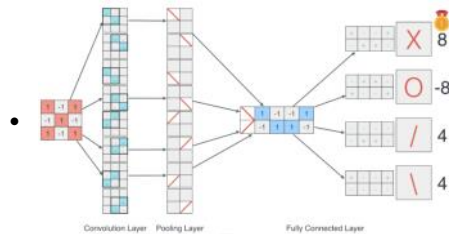
most_uncertain = np.argsort(np.abs(probs -0.5))[:4]
plot_val_with_title(most_uncertain, "Most uncertain predictions")
```

_other - Luis Serrano: A friendly introduction to Convolutional Neural Networks and Image Recognition

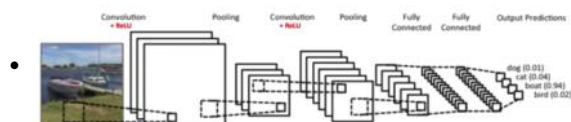
16. prosince 2018 13:35

<https://www.youtube.com/watch?v=2-OI7ZB0MmU>

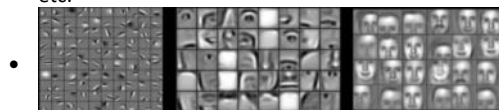
- **derivatives** taken during gradient descent optimization
- **convolutional layer** - breaks pictures into smaller pieces
- **filters** are applied - to check presence of patterns - resulting in high score
- 
- **pooling layer** - checks if the result is under or over threshold
- **fully connected layer** - classifies afterwards



actual image classifier works the same



- can have multiple convolutional and pooling layers
- must be pretrained on lot of images to be to assign numbers numbers on each filter each convolution etc.



LESSON 2 – CNN

16. prosince 2018 13:30

Learning rate (L)

- **Gradient descent** - if its steeper than ts probably far away
- to large = accuracy spinning to infinity
- **approach** - start small L - gradually increase - get the lowest point and go back circa one magnitude
 - cílem je najít ne tu nejnižší ale tu která dává největší přírůstky
- hyperparameter are optimized at background
- learning rate finder je dobré spouštět vždy když se něco mění na způsobu trénování tzn na začátku a pak např při unfreezing layers apod.

$$10^{-2} = 0.01 = 1e-2$$

$$10^{-1} = 0.1 = 1e-1$$

Best way to make model better is to give it more data

- to avoid Overfitting - **training loss is much better then validation loss = model is not generalist**

Data augmentation

- most important thing to make model better
- generování alternativních obrázků ze stávajících dat např. zrcadlové obrácení apod.
 - nicméně je třeba vždy klídat kontext - např písmena nebudu chtít převracet vertikálně protože by to změnilo jejich význam
 - tato znalost vychází z DOMAIN KNOWLEDGE
- fastai
 - `tfms = tfms_from_model(resnet34,sz,aug_tfms=transforms_side_on, max_zoom=1.1)`
 - `aug_tfms=transforms_side_on, max_zoom=1.1` (pak např `top_down`)
 - převrátit vertikálně
 - částečně rotovat ale jen trochu (ne převrátit)
 - přizoomovat částečně
- data augmentation nic nedělá pokud je `precomp.=True` tzn we have to set `learner.precompute=False` and fit more epochs

Precomputed & unfreezing

- *frozen*
 - *pracujeme pouze s posledními vrstvami (dvěma) co jsme přidali kvůli našim classes (které startují random)*
- *precomputed=True*
 - *nemá žádné accuracy důvody - je to pouze performance věc*
 - *počítáme jak jsou inputované obrázky blízke již existujícím activations v dřívějších vrstvách*
- *precompute=False*
 - *augmentation funguje*
 - *protože to v posledních vrstvách rekalkuluje všechny activations*
- *unfreeze*
 - *dochází k rekalkulaci filters v celé CNN*

Gradient Descent with Restarts (SGDR)

Plot data augs

def get_augs():

data = ImageClassifierData.from_paths(PATH, bs=2, tfms=tfms, num_workers=1)

x,_ = next(iter(data.aug_dl))

return data.trn_ds.denorm(x)[1]

ims = np.stack([get_augs() for i in range(6)])

plots(ims, rows=2)

Uložení a load modelu

- `learn.save('224_lastlayer')`
 - ukládá model do data/models
- `learn.load('224_lastlayer')`
 - zpětné načtení
 - je nutno dát `precompute=False` předtím
- info: pokud se pracuje s precomputed modelem tak se ukládají precomputed activations (nebo resized images apod.) do data/tmp
 - někdy je dobré tmp složku promazat pokud nastal problém (half completed, incompatible etc.) protože tam mohou být corrupted files
 - ale jinak by tam neměl být problém
- <https://forums.fast.ai/t/how-to-change-dropout-in-partially-trained-model/16999/11>
 - Why does the printout of the model only shows the head? Because you used `precompute=True` in your first learner, so the fastai library precomputes all the activations of your backbone model (here resnet34), stores them in a tmp directory, and only considers the last layers as the model to speed up things.
 - At no point did you say `precompute=False` (which you should before unfreezing) so when you save the model, it only saves the custom head.
 - When you come back later to your notebook and try to load the model with a new learner where you don't specify `precompute=True` (it's False by default), it doesn't work because it expects more weights.
 - If you want to retrieve your saved models, create a learner with `precompute=True`, load the model, then type `learn.precompute = False`, then resave your model.

Differential Learning Rates

Další optimalizace (ne modelu ale vstupních dat)

- v INFERENCE TIME gpu potřebuje všechny vstupy v konzistentních dimenzích/rozměrech (ideálně čtverec) protože pokud obrázky chodí v různých tvarech tak to snižuje performance
 - takže při validaci to bere čtverec uprostřed z obrázků které jsou široké
- řeší se to **test-time augmentation (TTA)**
 - při validaci/inference se vezmou čtyři random data augmentatione z obrázku a původní obrázek
 - napředikují se a vezme se průměr
 - snižuje error rate o dalších 10-20 procent
 - fastai
 - `log_preds, y = learn.TTA()`
 - `probs = np.mean(np.exp(log_preds), 0)`
 - `accuracy_np(probs, y)`

ANALYSIS OF RESULTS

- confusion matrix - sklearn
 - `cm = confusion_matrix(targs,preds)`
 - `plot_confusion_matrix(cm,data.classes)`

Recap to create world-class classification model

Other tips:

- get data metoda aby bylo možno rychle pracovat s daty
 - `def get_data(sz,bs):`
 - `tfms = tfms_from_model(arch,sz,aug_tfms=transforms_side_on,max_zoom=1.1)`
 - `data = ImageClassifierData.from_csv(PATH,folder='train',csv_fname=f"{PATH}labels.csv",`
 - `bs=bs,tfms=tfms,val_idxs=val_idxs,test_name='test',`
 - `suffix='.jpg',num_workers=4)`
 - `return data if sz>300 else data.resize(340,'tmp')`
- CUDA out of memory error
 - nelze recoverovat nutno restartovat kernel
 - a je nutno si pohrát s batch size bs a velikostí obrázků sz
- velikost validation setu
 - pokud je dostatečně velký vzorek (1000+) tak 20%
- batch size
 - standardně používá 64
 - z hlediska velikosti je nutno vážit přesnost optimalizace které je u nižších bs nižší a performance kde je nižší zase lepší
- accuracy
 - je hodně také závislá na počtu clases tzn 80 na dvou třídách je něco zásadně jiného než 80% na 120
- num of workers
 - ?
- **increase image size**
 - `learner.set_data(get_data(299,bs))`
 - pokud je něco vytrénováno na smaller size datech tak mohu na modelu zvětšit data - pokračování tréninku na větších obrázcích
 - důležitá metoda pro optimalizaci a zabránění overfittingu (jsou to defacto jiné obrázky ale s koncepčně stejnou informací)
 - neznámá
 - `learner.freeze()`
 - pro jistotu ale už to freezed bylo předtím
- 10000 obrázků je na deeplearning standards málo dat
- standardní velikosti obrázků pro convDL
 - 224
 - 299
- `trn_loss val_loss 0.345079 0.160794`
 - **validation loss je stále podstatně nižší než training loss - tzn stále **under** fituju**
 - tzn cyclelen nemá nikdy prostor najít nejlepší hodnotu protože se vždy restartuje
 - tzn je vhodné navýšit `cycle_mult` - aby měl delší čas na nalezení
 - **až se dostanem na úroveň kdy je training a validation loss stejná**
- `learn.sched.plot_loss()`
 - plotting lossu po fitování

- `learn.summary()`
 - summary modelu

ARCHITECTURES

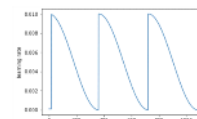
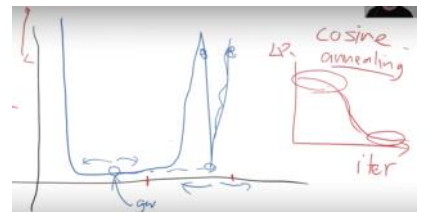
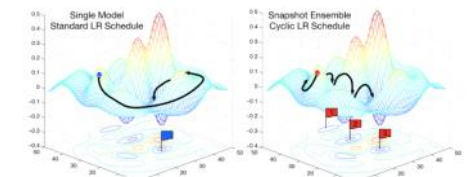
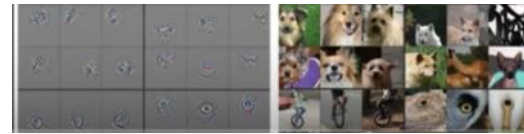
- jedna z možností jak rychle vylepšit model je používat lepší pretrained architektury
- architektury znamenají jak je CNN poskládaná, jak komunikují filtry, kolik je layers, sizes of kernels, počet filterů apod
- u resnetů číslo značí rozsah
- resnet34
 - je dobrý starting a finishing point
- resnext50
 - tip na další pokus
 - nicméně většinou je dvakrát pomalejší a 2-4 více paměti

Gradient Descent with Restarts (SGDR)

19. prosince 2018 10:17

GRADIENT DESCENT WITH RESTARTS

- later layers of CNN have **activations**
- **activation is a number** that says
 - this feature (eyeball of bird) is in this location with this probability/confidence
- we have pretrained network that was already train to recognize certain things (resnet - 1.5m images in imagenet)
- if we take second last layer (with all activations) we create **precomputed** layer
 - precomputed=True
 - by default learner sets all layers except the last one to freeze
- but data augmentation doesnt do anything if precomp is True we have to set learner.precompute=False and fit more epochs
 - learn.precompute=False
 - learn.fit(1e-2, 3, cycle_len=1)
 - **cycle_len parameter causes reset of learning rate after defined number of cycles** - overall number of cycles is second parameter (epoch = num of cyc * cyc len)
 - form of **gradient descent with restarts (SGDR)**, a variant of **learning rate annealing**
 - annealing - learning rate is gradually decreasing to get closer to optimal weights however..
 - protože je možné že se v danou chvíli nacházíme na nesprávném místě wégth space (malé změny ve váhách způsobují velké rozdíly v loss) - proto resetem LR dosáhneme toho že se můžeme přesunout do jiné části space, která je jak accurate tak stable
 - **annealing** má různé způsoby buď manuální nebo skrze nějakou funkci - častá ee **cosine** nejdříve snižují pomalu, pak rychle a pak opět pomalu
 - LR se mění every single minibatch
 - SGDR souvisí i s learning rate finder - proto se nevolí ta nejnižší hodnota - aby to přeskakovalo do jiných částí modelu a dalo se optimalizovat

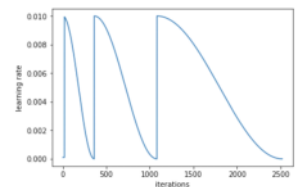
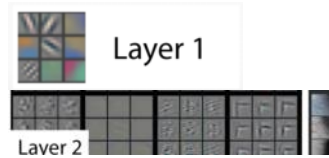


Differential Learning Rates a cycle_mul

19. prosince 2018 10:18

Fine-tuning and differential learning rate annealing

- **vše co bylo dosud děláno nehýbalo s precomputed vrstvami modelu !!!**
- **precomputed activations a weights zůstaly všechny stejné (v pretrained convolutional layers)**
- **přidávaly se jen nové vrstvy nakonec kvůli klasifikaci nových obrázků**
- pokud bych například chtěl rozpoznávat něco super specifického (ledovce?) tak bude asi nutné jít úplně na začátek k dřívějším vrstvám a přepočítat aby to optimálně fungovalo, nicméně
 - první vrstvy (layer 1, 2 apod)
 - se většinou nemusí moc měnit
 - řeší hodně základní tvary tzn je to univerzální
 - většinou se tedy jedná o finetuning pozdějších vrstev
- postup - fastai:
 - je nutné **unfreezeout layers**
 - frozen layer je ten co je untrained nebo neupdatovaný
 - `learn.unfreeze()`
 - a pak z důvodů výše **nastavit různé learning rates pro různé vrstvy (differential learning rates)**
 - `lr = np.array([1e-4, 1e-3, 1e-2])`
 - resnet má architekturu rozdělenou do bloků - ty to tři parametry dělí architekturu na tři části a pak jim přiřazují jiné LRs
 - první číslo jsou první vrstvy se základními tvary = hodně nízká LR
 - druhé jsou střední vrstvy s kombinacemi tvarů a sophisticated features
 - poslední číslo jsou závěrečné rozpoznávací na obrázky co jsme přidali
 - **je nutné být opatrný aby nedošlo ke zničení pretrained informací - proto nízká lr a spíše neměnit**
 - a znovu fitnout
 - `learner.fit(lrs=lr, n_cycle=2, cycle_len=1, cycle_mult=2)`
 - `cycle_len` násobí počet epoch v cyklech a po daném počtu provede restart
 - ještě jedna feature **cycle_mult=2**
 - prodlužuje délku cyklu 2x po každé cyklu
 - je to způsob jak změnit dobrý model na výborný
 - často fungující kombinace podle fast ai jsou
 - **ncyc 3, cyclen 1, cycmult 2**
 - **ncyc 3, cyclen 2**



Recap to create world-class classification model

16. prosince 2018 13:31

podtržené jsou prioritní

1. *conditions*: usually sz=224, bs=64 (ale záleží na velikosti vstupních obrázků)
2. Enable data augmentation, and precompute=True
3. Use lr_find() to find highest learning rate where loss is still clearly improving
4. Train last layer from precomputed activations for 1-2 epochs
5. Train last layer with data augmentation (i.e. precompute=False) for 2-3 epochs with cycle_len=1
6. *optional*: zkuste increase image size trick a přetrénovat s cycle_len=1 případně podruhé i s cycle_mult=2
7. Unfreeze all layers
 - a. *pokud jsou data velmi blízka imagenetu tak to už pak dál nedává smysl a nepřinese zlepšení*
8. Set earlier layers to 3x-10x lower learning rate than next higher layer
9. Use lr_find() again
10. Train full network with cycle_mult=2 until over-fitting
11. *optional*: pokud jsme z training setu vybírali validation set tak pak rozšířit na celou množinou a vytrénovat naplno
12. *optional*: use TTA (pokud je is_test=True tak je predikce nad TEST setem jinak VALIDATION setem)

```
In [1]: from fastai.conv_learner import *
PATH = "data/dogscats/"
sz=224; bs=64

In [10]: tfms = tfms_from_model(resnet50, sz, aug_tfms=transforms_side_on, max_zoom=1.1)
data = ImageClassifierData.from_paths(PATH, tfms=tfms, bs=bs)
learn = ConvLearner.pretrained(resnet50, data)
%time learn.fit(1e-2, 3, cycle_len=1)

...

In [11]: learn.unfreeze()
learn.bn_freeze(True)
%time learn.fit([1e-5, 1e-4, 1e-2], 1, cycle_len=1)

...

In [12]: %time log_preds, y = learn.TTA()
metrics.log_loss(y, np.exp(log_preds)), accuracy(log_preds, y)

CPU times: user 13.6 s, sys: 5.86 s, total: 19.5 s
Wall time: 23.8 s
```

LESSON 3 – CNN

16. prosince 2018 13:18

- **%time** - magic comand který uvede jak dlouho příkaz běžel
- **tmux** - linux feature s více obrazovkami
- **Keras**
 - knihovna využívaná spolu TensorFlow (TF je backend), Microsoft CNTK etc
 - v lekcí návod jak naimplementovat dogsbreeds na kerasu - nicméně je nutno mít správnou verzi CUDA (asi 9.0)
- **individual/single image prediction** (fastai/courses/dl1/lesson1-breeds.ipynb)
 - `trn_tfms, val_tfms = tfms_from_model(arch, sz)`
 - `im = val_tfms(open_image(PATH + fn))` # open_image() returns numpy.ndarray
 - `preds = learn.predict_array(im[None])`
 - `np.argmax(preds)`

Other notes links:

https://medium.com/@hiromi_suenaga/deep-learning-2-part-1-lesson-3-74b0ef79e56

CNN background theory

Satellite imagery:

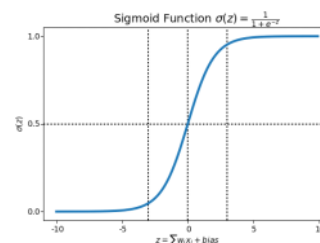
- načtení dat
 - `data = ImageClassifierData.from_csv(PATH, folder='train-jpg', csv_fname=f'{PATH}train_v2.csv', bs=bs, tfms=tfms, val_idxs=val_idxs, test_name='test-jpg', suffix='.jpg', num_workers=4)`
 - `x, y = next(iter(data.val_dl))`
- **dl** - data loader
- koncepty převzaté z pytorch
- returns minibatch (after transformation) - generátor
 - iter - convert to iterable
 - next - grab batch
- **ds** - data set
 - returns single object pack
- `plt.imshow(data.val_ds.denorm(to_np(x)))[0]*1.4)`
 - images are just a set of numbers in matrices -> *1.4 results in higher brightness
- most imagenets are trained on 224x224 or 299x299
 - satellite data are 256 - but start with resize to 64
 - and then 30proc plus data = `data.resize(int(sz*1.3), 'tmp')`
 - nicméně kromě prvních layers (edges and structures) není z imagenet použitelné pro daný data set téměř nic

get_data

metrics

MULTILABEL CLASSIFICATION

- SOFT max is not suitable for multilabel classification
- FASTAI library automatically checks whether there is more labels per object and selects suitable classifier
- Activation function for multilabel classification is **SIGMOID**
 - **Sigmoid je velmi blízký softmaxu ale nekalkuluje value/sum of values ale value / (1+value)**
 - If its below 0 then it will be below 0.5
 - Sigmoid je S funkce mezi 0 a 1
 - Používáno v logistic regression



Other hints

- Proč nezačínat rovnou s differential learning rates ale nejdřív předtrénovat fully connected layer a pak až unfreeze je ten že poslední vrstvy začínají zcela s random weights je tedy vhodné je nejdříve přetrénovat než se začne hýbat v weights původních pretrained vrstev
- SGD (stochastic gradient descent) vždy mění pouze **weights** (to je definice tréninku)
 - Weights v kernels v convolutional layers
 - A weights ve fully connected
- Activations are all calculated
- `learn.summary()`
 - Vypíše všechny vrstvy modelu, počty kernels ve vrstvách, jejich dimeze (tensorů) atd.

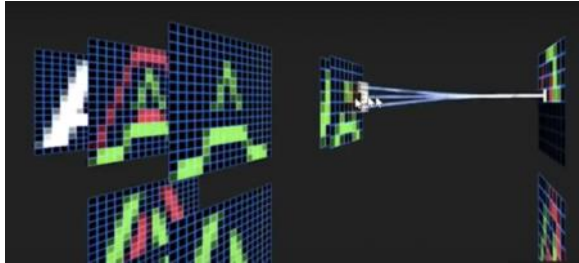
TWO TYPES OF DATA IN DEEP LEARNING

- First - where all the units in the objects are the same
 - These are UNSTRUCTURED
 - Audio, images, natural language text – waves, pixels, words
- Second – where all the units can be different (eg. column with num of page views, sex, zip code)
 - These are STRUCTURED
 - Profit and loss statement, fb user information

CNN background theory

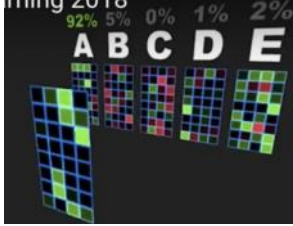
19. prosince 2018 10:14

SIMULATION EXCEL: <https://github.com/fastai/fastai/blob/master/courses/dl1/excel/conv-example.xlsx>



CNN THEORY (50:00):

Prediction sequence (simple architecture, pretrained model):

- **input LAYER takes the picture**
 - in this example only "black and white" ie one channel (normaly is would already start with 3 channles RGB)
- **second (first HIDDEN) LAYER**
 - size 2: because it has 2 filters/kernels
 - first KERNEL FILTER (1/1) creating the result of first kernel then second KERNEL again multiplying (2/1)
 - filters are designed to detect specific patterns ie TOP EDGES
 - filters are result of TRAINING
 - filters are stored as TENSORS (multidim matrices)
 - then ReLU
 - rectified liner unit - adding nonlinearity
 - basicaly throw away the negatives
 - 
 - in excel it is **MAX(0;convresult)**
 - outcome is an ACTIVATION
 - which is a calculated number
- **second HIDDEN LAYER**
 - combining previous results together with two 2x3x3 kernels (again stored as tensor)
 - applied ACTIVATION FUNCTION $\max(0, \text{filterresult} + \text{filterresult2})$ - but its just one filter with more dimensions
 - activation function = take one number in and calculates resultin number (activation)
- **then MAX PULL**
 - replacing a 2x2 or 3x3 by a single digit (nonoverlapping)
 - thus half the size of the image
- **ten FULLY CONNECTED LAYER**
 - present in older architectures or for structured data
 - because of risk of overfitting and poor performance
 - vgg has in result hundreds of milions of weights
 - not used in resnet etc.
 - take every single one of activations and give every a WEIGHT (weight matrix big as the entire input)
 - and do a sumproduct of the two-dimensional tensor (all kernels)
 - OUTPUT is one NUMBER/value
 - but is reality there would be that much numbers as clases
 - note: no relu is used so it could be negative also

	output	exp	softmax
cat	-1.83	0.16	0.00
dog	2.85	17.25	0.09
plane	3.86	47.54	0.26
fish	4.08	59.03	0.32
building	4.07	58.78	0.32
		182.75	1.00

- **SOFTMAX classifier**
 - what it does: probs are between 0 and 1 and in sum all the probs are 1
 - it is also an activation function but used ONLY in last layer
 - softmax
 - needs nonnegative numbers thus **EXP them (e to the power of each)**
 - add them up
 - PROB is division of each with the sum
 - because of the nature of exp one number tends to be always significantly higher than the others
 - but in NOT SUITABLE for multilabel classification
 - result of softmax is then compared to ONE HOT ENCODED LABELS (vector)
 - then we add up differences to calculate ERROR
 - note: pytorch is actually automatically onehot encoding indexes
 - In fastai actually logsoftmax is used (due to minor mathematical precision)
 - Sequential(
 - ... (7): **LogSoftmax()**

ARCHITECTURE = how many filters in layer one and how big (two 3x3), layer two (two 2x3x3 filters), then max pooling 2x2 (result is maximum of 4 numbers -> half resolution), ten fully connected layer

- general goal is maximize result and its quality (ie accuracy) but minimize complexity of the NN (due to performance, transparency etc)

get_data

16. prosince 2018 12:57

AUGUMENTATIONS SIDE ON + RESIZING

```
def get_data(sz,bs):
    tfms = tfms_from_model(arch,sz,aug_tfms=transforms_side_on,max_zoom=
1.1)
    data =
ImageClassifierData.from_csv(PATH,folder='train',csv_fname=f"{PATH}
labels.csv",

bs=bs,tfms=tfms,val_idx=val_idx,test_name='test',
                                suffix='.jpg',num_workers=4)
    return data if sz>300 else data.resize(340,'tmp')
```

AUGUMENTATIONS TOP DOWN

```
def get_data(sz,bs):
    tfms = tfms_from_model(arch,sz,aug_tfms = transforms_top_down,
max_zoom=1.05)
    data = ImageClassifierData.from_csv(PATH,folder='train-
jpg',csv_fname=f"{PATH}
train_v2.csv",bs=bs,tfms=tfms,val_idx=val_idx,test_name='test-jpg',
suffix='.jpg',num_workers=4)
    return data
```

`data = data.resize(int(sz*1.3),'tmp')`

Lesson 3 (1:40)

- Resizing due to speed/performance purposes because tfms/imageclassifierdata doesn't resize until training – so this is done upfront
 - <https://forums.fast.ai/t/lesson-3-couldnt-understand-data-resize-for-multi-label-classification-for-planet-data-set/12279/5?u=stanislav>
- 1.3 coefficient is for space for data augmentation purposes (rotations etc)
 - <https://forums.fast.ai/t/lesson-3-couldnt-understand-data-resize-for-multi-label-classification-for-planet-data-set/12279/14?u=stanislav>

GET VALIDATION IDX

```
labels_df = pd.read_csv(f"{PATH}labels.csv")
n = int(len(labels_df)*0.2)
val_idx=np.array(labels_df.sample(n).index)
# OR
label_csv = f'{PATH}labels.csv'
n = len(list(open(label_csv))) - 1 # header is not counted (-1)
val_idx = get_cv_idx(n) # random 20% data for validation set
```

Metrics

16. prosince 2018 13:08

Doposud se používala metrics = Accuracy

Nicméně může se vytvořit jakákoli metrika např kaggle vyžaduje f2 někde

Lesson 3 (1:41)

Funkce F-beta

- beta = how much do you weight false negatives vs false positives
- f2 means f with beta 2

<https://github.com/fastai/fastai/blob/master/courses/dl1/planet.py>

```
from sklearn.metrics
import fbeta_score
import warnings
```

```
def f2(preds, targs, start=0.17, end=0.24, step=0.01):
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        return max([fbeta_score(targs, (preds>th), 2, average='samples')
                    for th in np.arange(start,end,step)])
```

```
metrics = [f2]
learn = ConvLearner.pretrained(arch,data,ps=0.5,metrics=metrics)
```

- Do metrics nutno poslat funkci v poli a pak po každé epoše bude vypsána daná metrika
- Nicméně metrics model neoptimalizuje vždy optimalizuje loss function (cross entropy, negative log likely hood)
- Metrics jsou pouze parametr který se vypíše

LESSON 4 – STRUCTURED+NLP

18. prosince 2018 10:41

<https://forums.fast.ai/t/deeplearning-lec4notes/8146>
Other notes

ConvLearner(dropout, xtra, fo)

Structured data and embeddings

Categorical variables -> one hot encode

Continuous variables -> feed in to fully connected layers just as they are

Structured data preparation

Feature encoding/creation (using fastai.structured, sklearn.preprocessing)

- selection of subset of features that will be used for prediction the output
- converting categorical variables into embeddings, normalizing continuous features to standard normal

Selection of subset of features that will be used for prediction the output

- This is modelling decision
- If the variables are categorical by default the need to be as cat. in the model if continuous then we can decide
- Usually if continuous have floating point number they will remain continuous
- Rule of thumb if the cardinality is not too high (ie overall number of rows close to cardinality number) then its better to treat them as CATEGORICAL (because of rich representation as EMBEDDINGS later...)

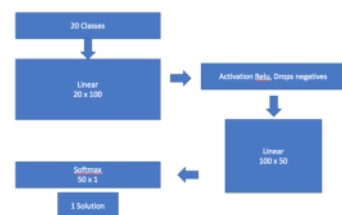
Categorical:

- Treating variables as categorical means that the NN will treat them as completely separated
- Some variables eg Year, Day of Week are generally continuous but treating them as categorical yields better results
- CARDINALITY of categorical variable
 - Number of levels it has

Feature creation (PROC DF)

DEEP LEARNING (FULLY CONNECTED NN)

- Example of simple FULLY CONNECTED NN
 - Approach
 - INPUT (Rank 1 tensor) -> linear layer -> activation layer (ReLU) -> linear layer -> OUTPUT
 - More linear and activation layers can be added
 - Dropout can be added
 - Softmax is not present in this case because it is not classification – we are predicting Sales
 - But could be added
 - Example for continual variables
 - 20 classes – ie Rank1 tensor (20 columns)
 - Goes to linear layer – must have 20 rows (due to matrix multiplication, arbitrary number of columns ie 100)
 - Gets multiplied -> matrix product -> rank 1 tensor with 100 columns
 - Again linear layer with 100 rows (number of columns in relation to number of dependent variables, could be 1) -> matrix product
 - Softmax classifier
 - Embeddings
 - Categorical variables
 - We create matrix with number of rows as it is levels for the selected cat.variable (cardinality) with arbitrary number of columns
 - Rule of thumb for number of columns: $\min(50, (c+1)//2)$
 - Take cardinality of cat.var. divide it by two but don't set number bigger than 50
 - Columns get randomly populated with numbers
 - During training via loss function backprop values get optimized via SGD
 - These are **EMBEDDING MATRICES**
 - After training the numbers in columns are reflecting best setup ie for individual days of week – to predict Sales
 - But they have no general MEANING – they are just optimization parameters
 - This rich (multidimensional) representation allow to capture "nonlinear" relationships
 - Note: ie good to capture seasonality aspects in timeseries if used in conjunction with `add_datepart(...)`, just need to make sure that the capturing column is there and NN will learn to use it
 - Afterwards we just append these to the tensor and the approach is the same (lin->activ->...)
 - Category and Embeddings size
 - `cat_sz = [(c, len(joined_samp[c].cat.categories)+1) for c in cat_vars]`
 - plus 1 is for the unknown (0)
 - `cat_sz`
 - `emb_szs = [(c, min(50, (c+1)//2)) for _, c in cat_sz]`
 - `emb_szs`
 - Metric = root-mean-squared percent error is the metric Kaggle used for Rossmann competition.
 - `def exp_rmse(y_pred, targ):`
 - `targ = inv_y(targ)`
 - `pct_var = (targ - inv_y(y_pred))/targ`
 - `return math.sqrt((pct_var**2).mean())`
 - `max_log_y = np.max(y1)`
 - `y_range = (0, max_log_y*1.2)`
 - Data for Model
 - `md = ColumnarModelData.from_data_frame(PATH, val_idx, df, y1.astype(np.float32), cat_fds=cat_vars, bs=128)`
 - ColumnarModelData returns data object with very similar API as ImageClassifierData



Submissions are evaluated on the Root Mean Square Percentage Error (RMSPE). The RMSPE is calculated as

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

where y_i denotes the sales of a single store on a single day and \hat{y}_i denotes the corresponding prediction. Any day and store with 0 sales is ignored in scoring.

Gotta! meant $\ln(a/b)$
 $\ln(a) - \ln(b)$

- PATH – only where do you want to store model etc (doesn't relate to getting data)
- Val_idx – indexes of rows we want to use for validation
- Df – dataframe with features
- Yl – dependent variable
- cat_fds=cat_vars which fields to treat as categorical
 - Because everything is number now!
 - Which are new set of weights in the model
- Learner/Model
 - `m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars), 0.04, 1, [1000,500], [0.001,0.01], y_range=y_range)`
 - emb_szs – embedding sizes get passed to learner
 - `len(df.columns)-len(cat_vars)` number of continuous variables
 - 0.04 dropout for embedding matrix
 - 1 how many outputs we want to create
 - [1000,500] - number of activations in first and second linear layer
 - [0.001,0.01] - dropout for the first and second linear layer
 -

Notes: when pinterest switched their recommendation system from gradient boosting machines to deep learning model much less feature engineering was needed – overall the model is much much simpler.

Approach recap:

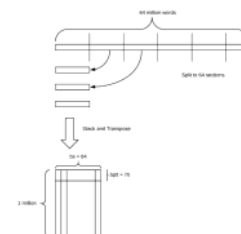
- Step 1
 - List the categorical variable names
 - List the continual variable names
 - Put it into dataframe
- Step 2
 - Create list of indexes to be put to validation set
- Step 3
 - Create ColumnarModelData object
- Step 4
 - Create list of embedding matrices sizes
- Step 5
 - Get learner
- Step 6
 - Fit

NLP

- Language modelling = model that predicts next word or words
- Goal create a pretrained language model to be used (after adding couple more layers to the end) for classification
 - Like finetuning in image classification
- Applications of NLP classif.
 - Fin markets - getting through all new reuters articles to find events that caused and can cause new drop
 - Customer retention - recognize customer queries that tend to be associated with leaving/cancelling contracts in next month

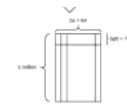
Approach

- Tokenization with spacy
 - Download en package for spacy
 - `python -m spacy download en`
 - `[sent.string.strip() for sent in spacy_tok(review[0])]`
- Preprocessing of data – with torchtext Field
 - `TEXT = data.Field(lower=True, tokenize="spacy")`
 - Lowercase, tokenize with spacy
- Model Data object
 - input torchtext field object, and the paths to our training, test, and validation sets
 - `bs=64; bptt=70`
 - Bs = mini batch size
 - Bptt (backprop through time) = how many words are processing at a time in each row of the mini-batch... how long sentence we will stick to GPU (70 tokens or less)
 - More importantly: defines how many 'layers' we will backprop through. Making this number higher will increase time and memory requirements, but will **improve the model's ability to handle long sentences**
 - `FILES = dict(train=TRN_PATH, validation=VAL_PATH, test=VAL_PATH)`
 - `md = LanguageModelData.from_text_files(PATH, TEXT, **FILES, bs=bs, bptt=bptt, min_freq=10)`
 - min_freq – at one moment there are words replaced by integers/indexes
 - Min frq sets threshold on minimum number of appearances of the word, otherwise call it unknown (don't think of it as a word)
 - After creation of model TEXT.vocab gets automatically populated with list of words appearing in the text + unique index
 - Indexing through vocabulary:
 - `TEXT.vocab.itos[:12]`
 - Int to string
 - Sorted by frequency, first two are special ones (unknown, padding)
 - `TEXT.vocab.stoi['the']`
 - Index of word
 - `TEXT.numericalize([md.trn_ds[0].text[:12]])`
 - List of words to int
 - Model processing
 - Full text is concatenated into one piece

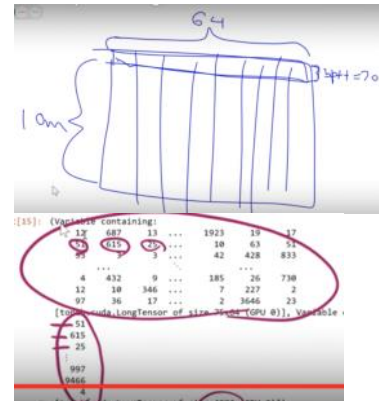


- `TEXT.numericalize([md.trn_ds[0].text[:12]))`
 - List of words to int

- Model processing
 - Full text is concatenated into one piece
 - Split into batches according to bs
 - These are stacked below each other and then transposed ie there is #bs number of columns
 - ie each column contains many many sentences (can get millions of words long)
 - Bptt is one step/bit
 - Result: one bit is always width of 64/bs and height up to 70/bptt
 - Plus the same shuffled by one word below (we are predicting next word)
 - PyTorch trick: bptt get randomly slightly changes, so each epoch the model get served slightly different text
 - Overall goal: predict next word thus in `next(iter(md.trn_dl))` comes current bit 75x64 plus next ie moved-down by one (due to minor technical reason flattend out but with the same contents)



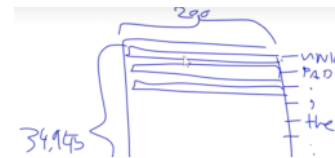
https://medium.com/@hiromi_suenaga/deep-learning-2-part-1-lesson-4-2048a26d58aa



Note: bag of words is and outdated concept because it is missing the sequence of words

RNN with LSTM

- Train
 - Parameters
 - `em_sz = 200` # size of each embedding vector
 - Each word in the vocabulary gets embedding vector of the size `em_sz`
 - Matrix: `em_sz x md.nt`
 - Vocabulary is basically high cardinality variable (an actually the only one in the model)
 - Rule of thumb for embedding size: 50-600
 - `nh = 500` # number of hidden activations per layer
 - `nl = 3` # number of layers
 - Data size
 - # batches/bits - `len(md.trn_dl)`
 - # unique tokens in the vocab - `md.nt`
 - At least 10/min_freq appearances – otherwise <unk>
 - # size of the training set - `len(md.trn_ds)`
 - One because its one chunk of text
 - # sentences - `len(md.trn_ds[0].text)`
 - Adam optimizer
 - Decrease Momentum than the default one (tech detail) `opt_fn = partial(optim.Adam, betas=(0.7, 0.99))`
 - Learner
 - `learner = md.get_model(opt_fn, em_sz, nh, nl,`
 - `dropouti=0.05, dropout=0.05, wdrop=0.1,`
 - `dropoute=0.02, dropouth=0.05)`
 - Using AWD LSTM LANGUAGE MODEL
 - Using lot of dropouts – see paper <https://arxiv.org/abs/1708.02182>
 - General rule for dropout optimization:
 - Overfitting = increase all of these dropouts
 - Under fitting = decrease all of these dropouts
 - `learner.reg_fn = partial(seq2seq_reg, alpha=2, beta=1)`
 - `learner.clip=0.3`
 - gradient clipping for the case LR is too high
 - `lr_find()`
 - Fiting
 - Standard multiple rounds with ongoing saving (`save/load_encoder`)
 - `learner.fit(3e-3, 4, wds=1e-6, cycle_len=1, cycle_mult=2)`
 - `learner.fit(3e-3, 1, wds=1e-6, cycle_len=10)`
 - `learner.fit(3e-3, 1, wds=1e-6, cycle_len=20)`
 - *Until not under or overfitting*



Word2Vec, GloVe – pretrained matrices (embeddings) for various tasks that can be downloaded and included in the model, but it doesn't help much

- Better is to use the whole pretrained model

Metrics

- Language modeling accuracy is generally measured using the metric perplexity, which is simply `exp()` of the loss function we used.
- #metric PERPLEXITY - exp of loss
- `math.exp(4.165)`

SENTIMENT CLASSIFICATION

(using pretrained model, finetune it and use it for classification):

- Exactly the same vocab must be used ie loading it
 - `TEXT = pickle.load(open(f'{PATH}models/TEXT.pkl', 'rb'))`
- `IMDB_LABEL = data.Field(sequential=False)`
- `splits = torchtext.datasets.IMDB.splits(TEXT, IMDB_LABEL, 'data/')`
 - Sequential = field should be tokenized, we just want to store sign positive/negative label
 - Splits
 - Torchtext method that creates train, val, test datasets
 - IMDB is builtin torchtext thus using here **but own datasets can be created**
 - `lang_model-arxiv.ipynb` to see how to define your own fastai/torchtext datasets
- `md2 = TextData.from_splits(PATH, splits, bs)`
 - Fastai load text data from splits directly
- `m3 = md2.get_model(opt_fn, 1500, bptt, emb_sz=em_sz, n_hid=nh, n_layers=nl,`
- `dropout=0.1, dropouti=0.4, wdrop=0.5, dropoute=0.05,`

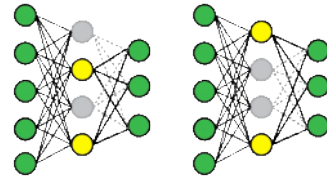
- dropouth=0.3)
- `m3.reg_fn = partial(seq2seq_reg, alpha=2, beta=1)`
- `m3.load_encoder(f'adam3_10_enc')`
 - Create model
 - But load our pretrained model into it
- FINETUNING
 - `m3.clip=25.`
 - `lrs=np.array([1e-4,1e-4,1e-4,1e-3,1e-2])`
 - Increase clipping for the SGDR to work better
 - Use differential learning rates
 - `m3.freeze_to(-1)`
 - `m3.fit(lrs/2, 1, metrics=[accuracy])`
 - `m3.unfreeze()`
 - `m3.fit(lrs, 1, metrics=[accuracy], cycle_len=1)`
 - Freezing training, unfreezing training

ConvLearner (dropout, xtra_fc)

19. prosince 2018 11:00

DROPOUT (IMAGE CLASSIFICATION)

- `learn = ConvLearner.pretrained(arch, data, precompute=True, ps=0.5)`
- `learn`
 - `Sequential`
 - (0): `BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True)`
 - (1): `Dropout(p=0.5)`
 - (2): `Linear(in_features=4096, out_features=512, bias=True)`
 - *Matrix multiply in 4096 activations (rows) out 512 activations (columns)*
 - (3): `ReLU()`
 - *Replace negatives with zeros*
 - (4): `BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True)`
 - (5): `Dropout(p=0.5)`
 - (6): `Linear(in_features=512, out_features=120, bias=True)`
 - *Second matrix mult. 512->120*
 - (7): `LogSoftmax()`
 - *Softmax classification*
- `learn`
 - Vypsání posledních vrstev (které spadají do `precompute=True`)
 - Vstupem pro ně je poslední convolution layer pretrained resnet modelu
- DROPOUT = with p probability I go through all of the activations/cells and I delete them (randomly)- ie removing of half of them
 - Each minibatch we throw away different ones
 - It prevents overfitting with generalization
 - It is not used in validation/inference time – obviously to use the best model we could
 - that's why often validation loss in early stages of training is better than training loss
 - Processing dropoutu řeší pytorch na pozadí po promazání nahrazuje průměry
- Usually OK sizes
 - First layer 0.5 second 0.25
- **When using bigger models (eg moving from resnet34 to resnext50) with more parameters its better to increase dropout**



MINIMUM PRECOMPUTED LAYER SETUP

- `learn = ConvLearner.pretrained(arch, data, ps=0., precompute=True, xtra_fc=[])`
 - No `ReLU`
 - Only one linear layer converting inout to 120 classes
 - And softmax
- `Sequential`
 - (0): `BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True)`
 - (1): `Linear(in_features=4096, out_features=120, bias=True)`
 - (2): `LogSoftmax()`
 -)

DIFFERENT DROPUTS

- `learn = ConvLearner.pretrained(arch, data, ps=[0.,0.25], precompute=True, xtra_fc=[512])`

Model

4. ledna 2019 8:34

Information about the model and architecture:

- `Learn.summary()`
- `Learn.model`
 - Pytorch model itself
 - Defined as Python property
 - `@property`
 - `def model(self): return self.models.model`

Structured data preparation

17. prosince 2018 16:24

Sources:

- [Entity Embeddings of Categorical Variables](#)
- <https://github.com/fastai/fastai/blob/master/courses/dl1/lesson3-rossman.ipynb>

Approach:

1. Get data to dataframes
2. Replacement of wrong or unmatching values
3. Datepart splitting to get individual datetime components (fastai.structured)
4. Adding "general" values valid for all to new columns
5. Joining all dataframes to one (resp train and test)
6. Replacing missing values for signal values (nonexisting in dataset) or valid (eg zeros) depending on underlying data meaning
7. Calculating number of days till or after event
8. Calculating rolling sums or averages (week window), joining to original one dataframe
9. Feature encoding/creation (using fastai.structured, sklearn.preprocessing)
 - a. Selection of subset of features that will be used for predictin the output
 - b. converting categorical variables into embeddings, normalizing continuous features to standard normal

Ongoing feather saving

```
from fastai.structured import *
from fastai.column_data import *
    Fastai.structured is not dependent on PyTorch – can be used individually (built on top of pandas DF)
    https://www.fast.ai/2018/04/29/categorical-embeddings/

add_datepart(df, "A", drop=False)
    • Splits date to multiple individual categories
    • AYear AMonth AWeek ADay ADayofweek ADayofyear Als_month_end Als_month_start Als_quarter_end
      Als_quarter_start Als_year_end Als_year_start AElapsed
    • Should always expand date-time data into these additional fieldsthat to capture trend/cyclical behaviour
      whoch is not possible at standard level eg yyyy-mm-dd

def join_df(left, right, left_on, right_on=None, suffix='_y'):
    if right_on is None: right_on = left_on
    return left.merge(right, how='left', left_on=left_on, right_on=right_on,
suffixes=("", suffix))
    • Left outer join method
    • If left rows doesn't hoave corresponding value -> null/nan
    • check for Null values post-join

df['year'] = df.year.fillna(1900).astype(np.int32)
    • Fillin missing data with "signal value"
    • Value otherwise not present in data set
```

Common when working with time series data to extract data that explains relationships across rows as opposed to columns, e.g.:

- Running averages
- Time until next event
- Time since last event
- In this case
 - days past state&school holiday, promo
 - days to state&school holiday, promo
 - running number of days of state&school holiday, promo in (b) week BEFORE
 - running number of days of state&school holiday, promo in (f) week FOLLOWING

```
def get_elapsed(fld, pre):
    day1 = np.timedelta64(1, 'D')
    last_date = np.datetime64()
    last_store = 0
    res = []

    for s,v,d in zip(df.Store.values,df[fld].values, df.Date.values):
        if s != last_store:
            last_date = np.datetime64()
            last_store = s
        if v: last_date = d
        res.append(((d-last_date).astype('timedelta64[D]') / day1))
    df[pre+fld] = res
```

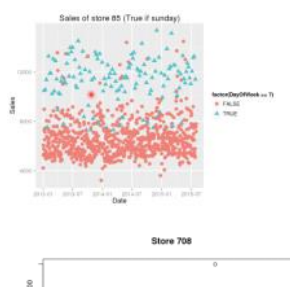
```
fld = 'SchoolHoliday'
df = df.sort_values(['Store', 'Date'])
get_elapsed(fld, 'After')
df = df.sort_values(['Store', 'Date'], ascending=[True, False])
get_elapsed(fld, 'Before')
```

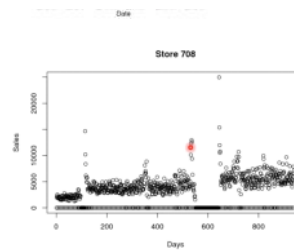
Window functions in pandas to calculate rolling quantities [Pandas](#)

[Feature creation \(PROC DF\)](#)

LESSON 6

- **apply_cats**(joined_test, joined)
 - is used to make sure that the test set and the training set have the same categorical codes.
- Keep track of **mapper**
 - which contains the mean and standard deviation of each continuous column, and apply the same mapper to the test set.





- Don't touch the data until you have analyzed what's going on
- Here in rossman there are huge leaps right before closer and right after reopening and because they omitted the zero sales data
- This is going to harm the model – not break it
 - Because the model will be trying to predict the jumps without knowing full context
 - Training to fit something which it doesn't have the data for

feature creation (PROC_DF)

19. prosince 2018 13:31

CATEGORY AND FLOAT TYPE

```
for v in cat_vars:
    joined[v] = joined[v].astype('category').cat.as_ordered()
apply_cats(joined_test, joined)
    • Conversion on categorical columns for Pandas
    • Will get stored differently internally
    • Plus aplikace na test set

for v in contin_vars:
    joined[v] = joined[v].fillna(0).astype('float32')
    joined_test[v] = joined_test[v].fillna(0).astype('float32')
    • Conversion of floating continuous values on float32 because of PyTorch
```

SAMPLE SET

```
idxs = get_cv_idxes(n, val_pct=150000/n)
joined_samp = joined.iloc[idxs].set_index("Date")
samp_size = len(joined_samp); samp_size
    • Creation of sample
    • To work with smaller data set
```

PROC_DF

```
df, y, nas, mapper = proc_df(joined_samp, 'Sales', do_scale=True)
yl = np.log(y)
```

- Proc_df = process dataframe – preparation for NN
- Separation of predicted/dependent variable ie Sales
- Scaling of continuoal variables
- Mapper – keeps track of what means and standrad deviations were used in scaling so it could be done for test set
- Processing of missing values
 - Categorical
 - where missing gets 0 other values become 1 2 3 etc
 - Continual
 - replacing missing value with mean/avg and creates new bollean columns if it was missing or not
 - Nas – object with missin values

VALIDATION SET

- In time series data, cross-validation is not random. Instead, our holdout data is generally the most recent data, as it would be in real application. This issue is discussed in detail in [this post](#) on our web site.
- One approach is to take the last 25% of rows (sorted by date) as our validation set.
 - train_ratio = 0.75
 - # train_ratio = 0.9
 - train_size = int(samp_size * train_ratio); train_size
 - val_idx = list(range(train_size, len(df)))
- An even better option for picking a validation set is using the exact same length of time period as the test set uses
 - val_idx = np.flatnonzero(
 - (df.index<=datetime.datetime(2014,9,17)) &
 - (df.index>=datetime.datetime(2014,8,1)))

Imports - NLP

21. prosince 2018 7:33

```
%reload_ext autoreload
%autoreload 2
%matplotlib inline

from fastai.learner import *

import torchtext
from torchtext import vocab, data
from torchtext.datasets import language_modeling

    • pytorch nlp library

from fastai.rnn_reg import *
from fastai.rnn_train import *
from fastai.nlp import *
from fastai.lm_rnn import *

import dill as pickle
import spacy
```

LESSON 5 – COLAB FILTERING

24. prosince 2018 9:17

Other lecture notes:

<https://forums.fast.ai/t/deeplearning-lecture5/8416>

https://medium.com/@hiromi_suenaga/deep-learning-2-part-1-lesson-5-dd904506bee8

Collaborative filtering

- Movielens dataset – user x movie rating
- Basic idea: recommendation based on
 - User aspect – which other users rate similarly like the predicted one (and what)
 - Movie aspect - which other movies are rated by similar people
- Collaborative filter in excel
(https://github.com/fastai/fastai/blob/master/courses/dl1/excel/collab_filter.xlsx)
 - Dot product.. In this case matrix product of vectors
 - Vectors are randomly initialized
 - The vectors are actually embeddings
 - Dimensionality is not easy to guess, but it need to cover inherent complexity of the related problem but at the same time keep reasonable performance
 - Metric: RMSE – root mean sq error
 - Gradient descent
 - View -> solver -> Min -> GRG nonlinear
 - Shallow learning
 - Using tools from DL (g.descent,matrix product) but without multiple linear and nonlinear layers
 - From linear algebra point of view it is: Probabilistic matrix factorization
 - with latent factors (not knowing actual meaning)
- Embeddings
 - Values in embeddings are latent factors

Dot Product

$$\begin{bmatrix} a & b \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = [ax + by]$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw + bx & ax + bz \\ cw + dx & cx + dz \end{bmatrix}$$

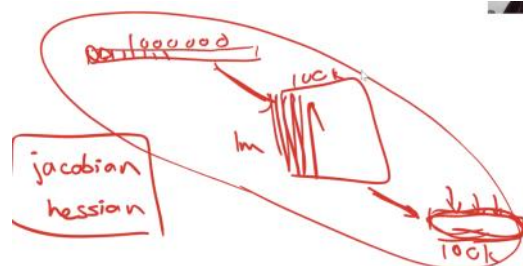
Fastai model

Practical approach to create this first (because its fast) and then start building from scratch with required tweaks in the architecture parameters etc.

- `cf_data = CollabFilterDataset.from_data_frame(PATH,ratings,'userId','movieId','rating')`
- `learn = cf_data.get_learner(n_factors,val_idxs,bs,opt_fn=optim.Adam)`
 - Adam optimizer is used for SGD
- `learn.fit(1rs,2,wds=wd,cycle_len=1,cycle_mult=2)`
 - Results in MSE
 - For RMSE (standard deviation of residuals) need to be squarerooted
 - `math.sqrt(0.738589)`

[Collab filtering from scratch – pytorch](#)

[Deep dive into training loop and optimizers](#)



Collab filtering from scratch – pytorch

26. prosince 2018 6:52

WITH DOT PRODUCT SIMPLE

CREATE TENSOR (fastai)

```
a = T([[1.,2],[3,4]])  
b = T([[2.,2],[10,10]])
```

- Operations with tensor
- $a*b$
- $(a*b).sum(1)$ - along first axis
- on the backward pass when our gradients are calculated and our weights are updated, taking the gradient of element-wise products (and sums) is cheaper than taking the gradients of full matrix products

```
class DotProduct(nn.Module):  
    def forward(self,u,m): return (u*m).sum(1)  
    • Create own pytorch module  
    • Special method forward

- related to forward pass in NN (calculation of activations in next layer)
- we don't have to care about backward at the moment

```

```
model=DotProduct()  
model(a,b)  
    • Instantiation
```

WITH DOT PRODUCT COMPLEX

to normalize indexes because they could be a starting at 1m and to make them contiguous

```
u_uniq = ratings.userId.unique()  
user2idx = {o:i for i,o in enumerate(u_uniq)}  
ratings.userId = ratings.userId.apply(lambda x: user2idx[x])
```

```
m_uniq = ratings.movieId.unique()  
movie2idx = {o:i for i,o in enumerate(m_uniq)}  
ratings.movieId = ratings.movieId.apply(lambda x: movie2idx[x])
```

```
n_users = len(u_uniq)  
n_movies = len(m_uniq)
```

creating custom module inheriting from pytorch Module

- `super().__init__()` call constructor from inherited class to get the behavior and to create fully functional pytorch module
- `self.u.weight.data.uniform_(0,0.05)` weights initialization, 0-0.05 is a calculated extent
- `self.u` contains the embedding
 - `u.weight` is a variable it like a tensor but it does automatic differentiation (derivatives)
 - to pull the tensor out of a variable `weight.data` is called
- `_` ...means in pytorch to perform inplace
 - So overall this line creates random number appropriate to size of the tensor, returns it but actually fill in the matrix inplace
 - Saves some typing
- `user,movies=cats[:,0],cats[:,1]` takes first and second column from `cat`
 - This construction along with `forward(self,cats,conts):` is used
- `u,m = self.u(users),self.m(movies)`
 - Lookup to users and movies embedding to get weights for multiplication
 - will work with whole minibatch not just one item
 - Pytorch can do batches so that we don't have to loop through
 - Also if we were to manually loop the gpu acceleration would not be utilized

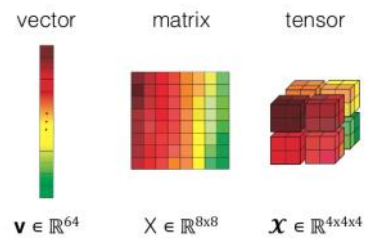
```
class EmbeddingDot(nn.Module):  
    def __init__(self,n_users,n_movies):  
        super().__init__()  
        self.u = nn.Embedding(n_users,n_factors)  
        self.m = nn.Embedding(n_movies,n_factors)  
  
        self.u.weight.data.uniform_(0,0.05)
```

```
    def forward(self,cats,conts):  
        user,movies=cats[:,0],cats[:,1]  
        u,m = self.u(users),self.m(movies)  
        return (u*m).sum(1) #dot product
```

Build data for model

```
x = ratings.drop(['rating','timestamp'],axis=1)  
y = ratings['rating']  
data = ColumnarModelData.from_data_frame(PATH,val_idxs,x,y,['userId','movieId'],64)
```

tensor = multidimensional array



Weights initialization formula

<https://www.jefkine.com/deep/2016/08/08/initialization-of-deep-networks-case-of-rectifiers/>

- To get a reasonable numbers resulting from first cycles

Instantiate model

- All pytorch stuff
- Optimizer
 - Optim = module that gives optimizer – thing that's going to update weights
 - Model.parameters() - weights etc that will be updated from nn.Module
 - Learning rate
 - Weight decay
 - Momentum

```
wd = 1e-5
model = EmbeddingDot(n_users,n_movies).cuda()
opt = optim.SGD(model.parameters(),1e-1,weight_decay=wd,momentum=0.9)
```

Fit method from fastai

- But it takes as input standard pytorch model and optimizer
- So if we download model from somewhere and we don't want to use fastai stuff, this is the easiest way – we don't have to write our training loop

```
fit(model,data,3,opt,F.mse_loss)
set_lrs(opt,0.01)
    • Set lower learning rates – fastai method
    • Fit again
```

WITH DOT PRODUCT

PLUS BIAS

- Adding constant for a user and constant for a movie to incorporate "inherent" characteristics for the movie and user itself
- Will be added to the dot product and also optimized by sgd
- It is actually the same code as it is in fastai/column_data.py

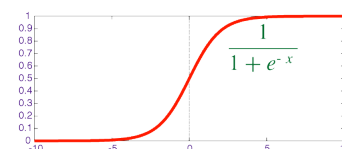
```
min_rating,max_rating = ratings.rating.min(),ratings.rating.max()
```

```
def get_emb(ni,nf):
    e = nn.Embedding(ni,nf)
    e.weight.data.uniform_(-0.01,0.01)
    return e
```

```
class EmbeddingDotBias(nn.Module):
    def __init__(self,n_users,n_movies):
        super().__init__()
        (self.u,self.m,self.ub,self.mb) = [get_emb(*o) for o in [
            (n_users,n_factors),(n_movies,n_factors),(n_users,1),(n_movies,1)
        ]]
```

```
    def forward(self,cats,conts):
        users,movies=cats[:,0],cats[:,1]
        um = (self.u(users)*self.m(movies)).sum(1)
        res = um + self.ub(users).squeeze() + self.mb(movies).squeeze()
        res = F.sigmoid(res) * (max_rating - min_rating) + min_rating
        return res
```

- **Broadcasting**
 - res = um + self.ub(users).squeeze() + self.mb(movies).squeeze()
- **Get the ratings between one and five**
 - res = F.sigmoid(res) * (max_rating - min_rating) + min_rating
 - Using sigmoid function between 0 and 1 and get is to required level



```
wd = 1e-5
model = EmbeddingDotBias(n_users,n_movies).cuda()
opt = optim.SGD(model.parameters(),1e-1,weight_decay=wd,momentum=0.9)
```

- **.cuda()** we are not using standard fastai learner etc thus we have to send it manually to GPU ie create cuda kernel

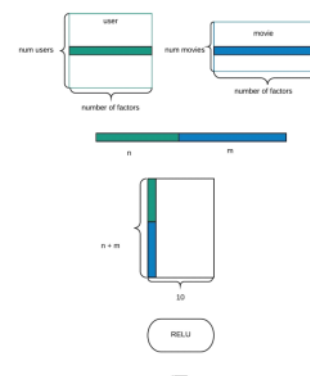
```
fit(model,data,3,opt,F.mse_loss)
set_lrs(opt,1e-2)
fit(model,data,3,opt,F.mse_loss)
```

WITH MINI NEURAL NET

Creating NN with one hidden layer (LINEAR) plus one embedding layer at the start and one non-linear function (ReLU).

- Input + embedding layer
- Hidden layer (fully-connected linear 100x10)
- ReLU
- Output layer (fully-connected linear 10x1)

```
class EmbeddingNet(nn.Module):
    def __init__(self,n_users,n_movies,nh=10):
        super().__init__()
```



```

class EmbeddingNet(nn.Module):
    def __init__(self, n_users, n_movies, nh=10):
        super().__init__()
        (self.u, self.m, self.ub, self.mb) = [get_emb(*o) for o in [
            (n_users, n_factors), (n_movies, n_factors), (n_users, 1), (n_movies, 1)
        ]]
        self.lin1 = nn.Linear(n_factors*2, nh)
        self.lin2 = nn.Linear(nh, 1)

```



- Initialization
- Creation of embeddings
- Creation of linear layers (rows, columns)

```

def forward(self, cats, conts):
    users, movies = cats[:, 0], cats[:, 1]

    x = F.dropout(torch.cat([self.u(users), self.m(movies)], dim=1), 0.75)
    x = F.dropout(F.relu(self.lin1(x)), 0.75)

    return F.sigmoid(self.lin2(x)) * (max_rating - min_rating + 1) +
    min_rating - 0.5

```

- Concat of user movies embeddings (torch.cat) on first dimension (rows)
 - Apply dropout
- Putting through linear layer (self.lin1)
 - Then ReLU
 - Activation function (from torch.F) - taking one activation in spitting one activation out
 - Then dropout
- After that sigmoid for classification
 - Nonlinear activation function in the last layer

```

wd=5e-4
model=EmbeddingNet(n_users, n_movies).cuda()
opt=optim.SGD(model.parameters(), 1e-2, weight_decay=wd, momentum=0.9)

```

```

fit(model, data, 3, opt, F.mse_loss)
set_lrs(opt, 1e-3)
fit(model, data, 3, opt, F.mse_loss)

```

- F.mse_loss – loss function that will be optimized during SGD
 - Mean squared error

Deep dive into training loop and optimizers

1. ledna 2019 20:30

Using excel <https://github.com/fastai/fastai/blob/master/courses/dl1/excel/graddesc.xlsm> (to be read from right to left)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Reset	intercept	1	learn	0.0001				de/db=2(ax+b-y)					
2	Run	slope	1						e=(ax+b-y)^2 de/da=x^2(ax+b-y)					
3	x	y	intercept	slope	y_pred	err^2	errb1	est de/db	errd1	est de/da	de/db	de/da	new a	new b
4	14	58	1	1	15	15	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00	1.12	1.01
5	86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.60	-209.27	-17,997.56	2.92	1.03
6	28	86	1.029527	2.9202	82.7939	10.28	10.22	-6.40	8.56	-171.70	-6.41	-179.54	2.94	1.03
7	51	132	1.030169	2.9381	150.874	356.22	356.60	37.76	375.73	1,951.14	37.75	1,925.13	2.75	1.03
8	28	86	1.026394	2.7456	77.9031	65.56	65.40	-16.18	61.10	-445.58	-16.19	-453.42	2.79	1.03
9	29	88	1.028013	2.7909	81.9653	36.42	36.30	-12.06	33.00	-341.60	-12.07	-350.01	2.83	1.03

- Goal: predict intercept and slope of function that was originally used for creation of input data

Basic SGD and Momentum

- Steps
 - We need to find which way is down for SGD
 - Calculate error
 - One option is to literally add little increments to intercept and slope and watch changes in error
 - Called: finding the derivative through finite differencing
 - Derivative is then change in dependent variable/change in independent variable
 - Error after change = squared (new value after 0.01 change – original value)
 - est de/db = error after change in intercept/change in b intercept
 - est de/da = error after change in slope/change in a slope
 - There is a big derivative with respect to slope (which is reasonable because slope is multiplying the value thus has bigger impact)
 - Hint: when thinking about derivative think about this small changes by 0.01
 - **Problem with finite differencing is in highdimensional spaces**
 - There isn't a gradient (like one) but as many as there are dimensions in the input vector and in the output vector etc (it's a matrix of gradients) -> to calculate one column in the hidden layer weight matrix
 - So the calculations would be huge and memory intensive
 - Solution:
 - Do it analytically instead of finite differencing – meaning using defined derivation rules
 - Most important for this case is a **CHAIN RULE**
 - Because NN is just a chain of functions
 - $e(f(g(x)))$...
 - lin->relu->lin->relu...linear->sigmoid/softmax
 - So if I want to calculate derivative somewhere in the chain all following need to be calculated -> that's why it is called **backpropagation**
 - Take the derivatives of all the functions and multiply them together
 - $f(g(x))$ is $f'(g(x)) \cdot g'(x)$
 - $\frac{d}{dx} f(g(t)) = \frac{df}{dg} \frac{dg}{dt} = f'(g(t))g'(t)$
 - Get the derivation functions (here from wolframalpha) of the error function $e=(ax+b-y)^2$
 - $d/da((ax+b-y)^2)$
 - $\frac{\partial}{\partial a} ((ax+b-y)^2) = 2x(ax+b-y)$
 - $d/db((ax+b-y)^2)$

- $\frac{\partial}{\partial b}((a x + b - y)^2) = 2(a x + b - y)$

- calculate derivatives
- Use derivatives to calculate new values
 - original value – LEARNING RATE * derivative
 - [Learning rate](#)
- This is example of online gradient descent = minibatch is of size 1
- Epoch = running through all training data
 - RMSE as accuracy metric of epoch = root of sum of all minibatch errors
- Problem: it goes very slow -> Solution: momentum (its right direction -> go faster)

	A	B	C	D	E	F	G	H	I	J	K	L	
1	Reset	b	1.0000	learn					beta	0.98	0.02		
2	Run	a	1.0000	0.0001									
3	x	y	b	a	pred	de/db	de/da	new b	new a	-11.97	-174.9	err^2	
4		14	58	1.0000	1.0000	15.0000	-86.00	-1204.00	1.00	1.02	-13.45	-195.4	1849
5		86	202	1.0013	1.0195	88.6822	-226.64	-19490.66	1.00	1.08	-17.71	-581.4	12840.93
6		28	86	1.0031	1.0777	31.1782	-109.64	-3070.02	1.01	1.14	-19.55	-631.1	3005.435
7		51	132	1.0051	1.1408	59.1855	-145.63	-7427.08	1.01	1.22	-22.07	-767	5301.954
8		28	86	1.0073	1.2175	35.0972	-101.81	-2850.56	1.01	1.30	-23.67	-808.7	2591.096
9		29	88	1.0096	1.2984	38.6623	-98.68	-2861.59	1.01	1.38	-25.17	-838.5	2591.096
10		72	174	1.0122	1.3833	100.6130	-146.77	-10567.72	1.01	1.49	-27.6	-1044	5385.646

- **Momentum**= linear interpolation between the current mini-batch's derivative and the step (and direction) we took after the last mini-batch (cell K9)
- It smooths out the surface for descent (ie it will keep longer in the direction that was sufficiently outlined)
- will reduce number of epoch required for training
- Actual number determines which of the input values (new derivative x previous step) will be weighted more

Adam

- Much better performance than SGD but problems with final accuracy (due to some issue with weight decays) -> should be improved though by AdamW
- MOMENTUM OF GRADIENT
 - uses same momentum as SGD
- MOMENTUM OF GRADIENT SQUARED
 - but utilizes another momentum / linear interpolation of the gradient **squared**
 - Referred to as exponentially weighted average
 - *Linear interpolation = parameter multiplication by 0.x value (alpha) and other by 1-alpha*
 - *Appears usually in the papers*
- New value = original – LR * momentum/rooted(gradient squared)
 - LR is initially set quite high due to the division (1)
- Result
 - When variance in gradient is high learning rate will be divided by bigger number to be more precise (result of the squaring)
 - When variance is low we feel confident about the step thus dividing by bigger number
 - -> **ADAPTIVE LEARNING RATE**
 - Adaptive learning rate — keep track of the average of the squares of the gradients and use that to adjust the learning rate. So there is just one learning rate, but effectively every parameter at every epoch is getting a bigger jump if the gradient is constant; smaller jump otherwise.

AdamW and Weight decay

- When you have lot of parameters (even more then data points) REGULARIZATION is very important to avoid overfitting
 - We have previously used dropout(random delete of activation in a hope to learn more resilient weights)
 - There is another one called WEIGHT DECAY (L2 REGULARIZATION)
 - It is **modifying the LOSS FUNCTION** by adding squared parameters, multiplied by wd rate (eg 0.0005) - hyperparameter

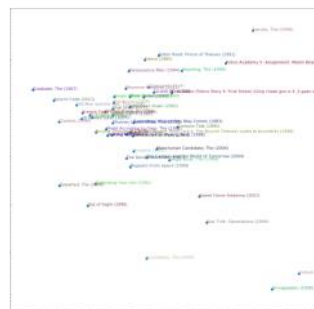
- This results in higher number this during the optimization there will be lower will to increase the weights/parameters
- Thus the improvement must be large enough to compensate this penalty
- Problem:
 - We don't want to actually put it to loss function because it would appear directly also in the G as G^2 momentum and have counter effect
 - decreasing the amount of weight decay when there is high variance in gradients, and increasing the amount of weight decay when there is little variation
 - Solution: thus in AdamW it is put to the point when we are calculating new values of parameters
- Generally the effect of WD is: do not increase any of the weight unless the improvement is worth it

1. jedna 2019 21:26

- ## Colab filter and Embeddings interpretation

- ## CPU vs. GPU in production

- Jeremy hint: processes everything in numpy until the point he needs GPU or its derivatives -> then switch to Pytorch -> but as soon as possible back (also for compatibility with other libs OpenCV, Pandas etc.)



Rossmann Model

Rossmann Embeddings

7. ledna 2019 8:49

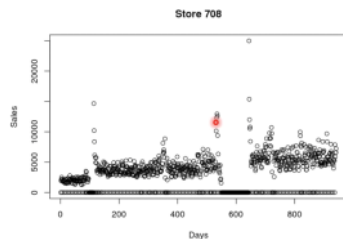
Rossmann Embedding Interpretation

Rossmann Embeddings Interpretation

- Entity embedding of the categorical entities = one hot encoding of the layer followed by matrix multiplication
- Then put it through linear/dense layer
- Embeddings can be used as an engineered feature afterwards also with standard ML methods (KNN, RF, GBM etc.) to improve performance
 - Truth is that standard ML libs still learn much faster than NN

Entity Embeddings

Unsupervised learning and AutoEncoders



Don't touch your data unless you, first of all, analyze to see what you are doing is okay — no assumptions.

- Here in rossman there are huge leaps right before closure and right after reopening and because they omitted the zero sales data
 - <https://www.kaggle.com/thie1e/exploratory-analysis-rossmann>
- This is not going to break the model but harm it
 - Because the model will be trying to predict the jumps without knowing full context
 - ie traing to fit something which it doesn't have the data for

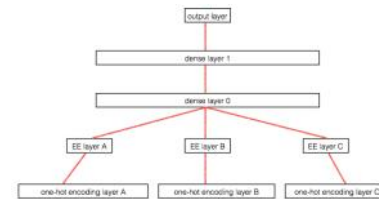
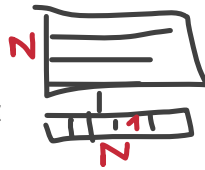


FIG. 1. Illustration that entity embedding layers are equivalent to extra layers on top of each one-hot encoded input.



Note: unsupervised learning and AutoEncoders

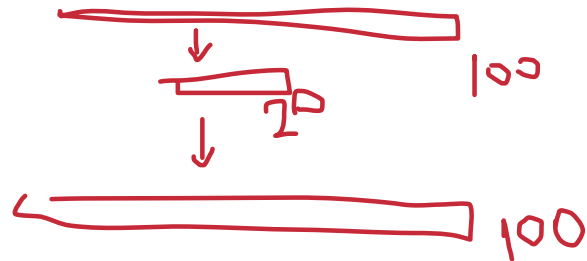
7. ledna 2019 7:34

SkipGrams vs EE: example Word2Vec – 11 word sentence – copy - middle word deleted and replaced by random word – labels 1 for the original 0 for the copy

- Then they build an ML model upon that, predicting the fake word
- But they were interested in the embeddings
- Downsides
 - Shallow learning does not capture too many details
 - Less predictive – thus Jeremy pushes to move from W2V to deep representation (full pretrained model)
- Upside
 - Embeddings in this way has very linear characteristics, can be added together etc.
 - Quick, can be trained on large data sets

Jeremy note: when people say unsupervised training today then usually mean fake labeled task, generally in this case we need tasks where the learned relationships are close to the ones we care about

- In computer vision.. often used task is do some weird/overdone augmentation and then train NN to recognize which one was augmented
- Ultimately crappy task is **AUTO ENCODER**.. (it won claims competition)
 - Trying to recreate input (eg with 100 variables) but with much less activations in hidden layers eg 20
 - Take a single policy, run it through neural net, and have it reconstruct itself (make sure that intermediate layers have less activations than the input variable). Basically, it is a task whose input = output which works surprisingly well as a fake task.
 - Any standard learner can be used for AC – the task is just INPUT = OUTPUT



Rossmann Model

úterý 8. ledna 2019 6:58

```
md = ColumnarModelData.from_data_frame(PATH, val_idx, df, yl.astype(np.float32), cat_flds=cat_vars,
bs=128,
                                test_df=df_test)
m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars),
0.04, 1, [1000,500], [0.001,0.01], y_range=y_range)
```

```
class ColumnarModelData(ModelData):
    def __init__(self, path, trn_ds, val_ds, bs, test_ds=None, shuffle=True):
        test_dl = DataLoader(test_ds, bs, shuffle=False, num_workers=1) if test_ds is not None
        super().__init__(path, DataLoader(trn_ds, bs, shuffle=shuffle, num_workers=1),
            DataLoader(val_ds, bs*2, shuffle=False, num_workers=1), test_dl)

    def get_learner(self, emb_szs, n_cont, emb_drop, out_sz, szs, drops,
y_range=None, use_bn=False, **kwargs):
        model = MixedInputModel(emb_szs, n_cont, emb_drop, out_sz, szs, drops, y_range, use_bn, **kwargs)
        return StructuredLearner(self, StructuredModel(to_gpu(model)), opt_fn=optim.Adam, **kwargs)
```

- **CONSTRUCTOR:** get_learner wraps the model (MixedInputModel – PyTorch model) and data together into StructuredLearner (with predefined optimizer etc.)

```
class MixedInputModel(nn.Module):
    def __init__(self, emb_szs, n_cont, emb_drop, out_sz, szs, drops,
y_range=None, use_bn=False):
        super().__init__()
        self.embs = nn.ModuleList([nn.Embedding(c, s) for c,s in emb_szs])
        for emb in self.embs: emb_init(emb)
        n_emb = sum(e.embedding_dim for e in self.embs)

        szs = [n_emb+n_cont] + szs
        self.lins = nn.ModuleList([nn.Linear(szs[i], szs[i+1]) for i in range(len(szs)-1)])
        self.bns = nn.ModuleList([nn.BatchNorm1d(sz) for sz in szs[1:]])
        for o in self.lins: kaiming_normal(o.weight.data)
        self.outp = nn.Linear(szs[-1], out_sz)
        kaiming_normal(self.outp.weight.data)

        self.emb_drop = nn.Dropout(emb_drop)
        self.drops = nn.ModuleList([nn.Dropout(drop) for drop in drops])
        self.bn = nn.BatchNorm1d(n_cont)
        self.use_bn, self.y_range = use_bn, y_range
```

- MixedInputModel
 - Argument: embedding sizes (n of rows and columns in each embedding)
 - Inside of **MixedInputModel** you see how it is creating Embedding
 - Iterates through passed in tuples and creates Embeddings for each
 - Pytorch cant register stuff as parameters if is just list
 - That's why there is **nn.ModuleList** - which is used to register a list of layers
 - Argument: number of fully connected layers and activations [1000,500]
 - Getlearner iterates through and creates nn.ModuleList with **nn.Linear** layers inside
 - Then initialization – kaiming_normal
 - Argument: dropout for each layer [0.001,0.01]
 - Getlearner iterates through and creates nn.ModuleList with **nn.Dropout** for each

layer

```
def forward(self, x_cat, x_cont):
    x = [e(x_cat[:,i]) for i,e in enumerate(self.embs)]
    x = torch.cat(x, 1)
    x2 = self.bn(x_cont)
    x = self.emb_drop(x)
    x = torch.cat([x, x2], 1)
    for l,d,b in zip(self.lins, self.drops, self.bns):
        x = F.relu(l(x))
        if self.use_bn: x = b(x)
        x = d(x)
    x = self.outp(x)
    if self.y_range:
        x = F.sigmoid(x)
        x = x*(self.y_range[1] - self.y_range[0])
        x = x+self.y_range[0]
    return x
```

- Def forward
 - Goes through each of the embedding layers
 - Calls it (as a function) with i-th categorical variable
 - Concatenates them all together
 - Put through dropout
 - Go through each of the linear layers
 - Call it
 - Apply ReLU
 - Apply Dropout
 - Apply final linear layer
 - Size of the output (1)
 - If passed in normalize to defined range
 - Apply sigmoid
 - Multiply with range

SGD recap

úterý 8. ledna 2019 6:58

Gradient descent is an algorithm that minimizes functions.

- Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function.
- This iterative minimization is achieved by taking steps in the negative direction of the function gradient.

Nearly all of deep learning is powered by one very important algorithm: **stochastic gradient descent (SGD)**

- SGD can be seen as an approximation of gradient descent (GD).
- In GD you have to run through all the samples in your training set to do a single iteration. In SGD you use only one or a subset of training samples to do the update for a parameter in a particular iteration.
- The subset used in every iteration is called a **batch or minibatch**.

NN – is just set of nested functions inside of functions

Example gradient descent with PyTorch:

- SGD example will be performed on random data generated by function $y=ax+b$
 - $y=\text{lin}(a,b,x)+0.1*\text{np.random.normal}(0,3,n)$
- Loss function -> MSE
 - `def mse(y_hat,y): return ((y_hat-y)**2).mean()`
 - `def mse_loss(a,b,x,y): return mse(lin(a,b,x),y)`
- Wrap x and y in pytorch tensors – for derivation purposes
 - `x,y = V(x),V(y)`
- create random weights a and b, wrap them in variables
 - `a=V(np.random.randn(1),requires_grad=True)`
 - `b=V(np.random.randn(1),requires_grad=True)`
- Full GD will be performed in 10000 epochs
- First calculation loss with our loss function
 - Forward pass: compute predicted y using operations on Variables
`loss = mse_loss(a,b,x,y)`
- Every 1000 epochs print operative result
- Loss
 - Is both single number but also pytorch variable because we have passed variables into it
- `loss.backward()`
 - Will perform grad computation where `requires_grad==True`
- Update a and b using grad descent
 - a.data and b.data are Tensors, a.grad and b.grad are Variables and a.grad.data and b.grad.data are Tensors
 - `a.data -= learning_rate * a.grad.data`
 - `b.data -= learning_rate * b.grad.data`
- If we had multiple loss functions/multiple output layers all contributing to gradient
 - We would be calling `loss.backward` on each of them and it adds result to the gradients
 - PyTorch will add them together. So you need to tell when to set gradients back to zero (`zero_()` in the `_` means that the variable is changed in-place).
 - We have to tell it when to set the gradients to zero (`_` means inplace in pytorch)
 - `a.grad.data.zero_()`

- `b.grad.data.zero_()`

```
learning_rate = 1e-3
for t in range(10000):
    # Forward pass: compute predicted y using operations on Variables
    loss = mse_loss(a,b,x,y)
    if t % 1000 == 0: print(loss.data[0])

    # Computes the gradient of loss with respect to all Variables with requires_grad=True.
    # After this call a.grad and b.grad will be Variables holding the gradient
    # of the loss with respect to a and b respectively
    loss.backward()

    # Update a and b using gradient descent; a.data and b.data are Tensors,
    # a.grad and b.grad are Variables and a.grad.data and b.grad.data are Tensors
    a.data -= learning_rate * a.grad.data
    b.data -= learning_rate * b.grad.data

    # Zero the gradients
    a.grad.data.zero_()
    b.grad.data.zero_()
```

- This code is normally wrapped in the `optim.SGD.step` class

Example gradient descent only with NumPy:

- First guess of a and b weights
 - `a_guess, b_guess = -1., 1.`
 - `mse_loss(a_guess, b_guess, x, y)`
- Calculate derivations of loss function with respect to a and b and update weights
 - `lr=0.01`
 - `def upd():`
 - `global a_guess, b_guess`
 - `y_pred = lin(a_guess, b_guess, x)`
 - `dydb = 2 * (y_pred - y)`
 - `dyda = x*dydb`
 - `a_guess -= lr*dyda.mean()`
 - `b_guess -= lr*dydb.mean()`
- Perform epochs
 - `for i in range(300):`
 - `upd()`
 - `print(mse_loss(a_guess, b_guess, x, y))`

Example gradient descent only with NumPy:

```
1 x, y = gen_fake_data(50, 3., 8.)

1 a_guess, b_guess = -1., 1.
2 mse_loss(a_guess, b_guess, x, y)

1 lr=0.01
2 def upd():
3     global a_guess, b_guess
4     y_pred = lin(a_guess, b_guess, x)
5     dydb = 2 * (y_pred - y)
6     dyda = x*dydb
7     a_guess -= lr*dyda.mean()
8     b_guess -= lr*dydb.mean()

1 for i in range(300):
2     upd()
3     print(mse_loss(a_guess, b_guess, x, y))
```

PyTorch implementation:

- Will automatically cover the forloop in forward method
- Create linear layers (in/out)
- `Nn.RNN` class

RNNs

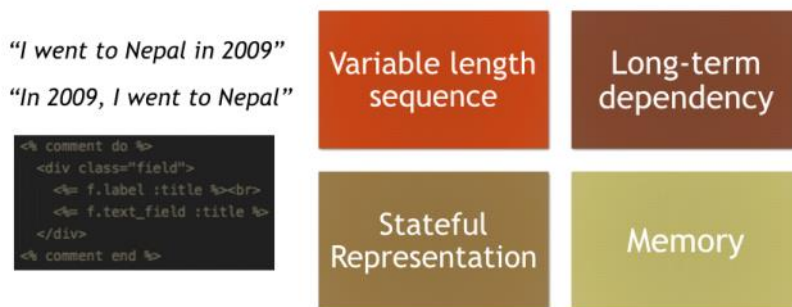
13. ledna 2019 17:03

RNN showcase

- Let's learn how to write philosophy like Nietzsche. This is similar to a language model we learned in lesson 4, but this time, we will do it one character at a time.
- Build on plain PyTorch

We need to:

Why we need RNNs



- Preserve long term dependency
 - remembering we are inside of some sentence with some beginning
 - It's harder (but possible) also with CNN
- Keep the state/memory
- Operate with variable length sequence

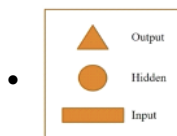
Some examples:

- [SwiftKey](#)
- [Andrej Karpathy LaTeX generator](#)

Example architectures:

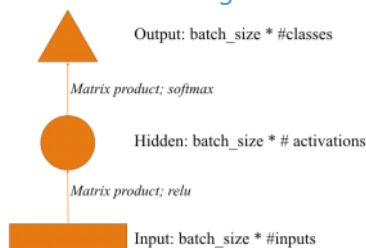
Shape = tensor of activations (calculated by some operation)

- **activation is a number**



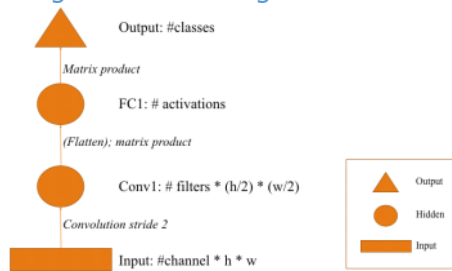
Arrow = layer operation (or more)

Basic NN with single hidden layer



Note: batchsize dimensions and activation functions (usually relu for hidden and softmax for output) will be omitted in diagrams (but it's there)

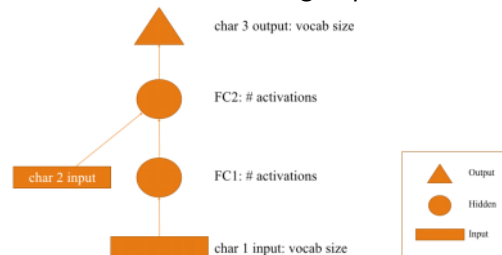
Image CNN with single dense hidden layer



- Flattening: currently most usual is “adaptive max pooling” — where we average across the height and the width and turn it into a vector

RNNs

The one we are recreating -> prediction of char 3 using sequence of chars 1 & 2



Standard 2 hidden layer NN with 3 matrix products, with only exception of later second input.

General Architecture

- Input: 1 char
 - can be one hot encoded ie vect of all chars in vocab and its position
 - or int and pretend its one hot encoded using EMBEDDING layer -> our case (matematically identical)
- We will put it through fully connected layer and get a set of activations and put it through another FC layer
- Input 2: now we feed the char 2
- We will get two matrix products (with same dimensionalities) and add them up
 - To get another set of activations FC 2
- All matrix product have RELU as well
- Finall matrix product will be finalized with SOFTMAX
 - To get predicted set of characters

Character based model vs. word based model [1:22:30]

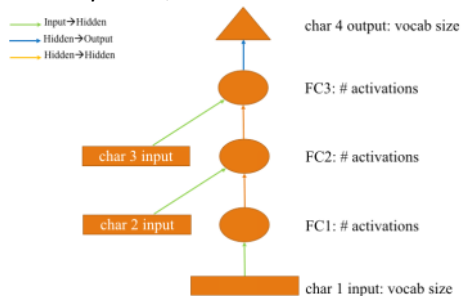
- Generally, you want to combine character level model and word level model (e.g. for translation).
- Character level model is useful when a vocabulary contains unusual words — which word level model will just treat as “unknown”. When you see a word you have not seen before, you can use a character level model.
- There is also something in between that is called Byte Pair Encoding (BPE) which looks at n-gram of characters.

Implementation

Basic FC NN

- VOCAB AND IDXs
 - Get vocabulary (set of chars) and insert null char
 - `chars = sorted(list(set(text)))`
 - `chars.insert(0, "\0")`
 - Get indicies for every character and character for every indicie

- `char_indicies = dict((c,i) for i,c in enumerate(chars))`
 - `indicies_char = dict((i,c) for i,c in enumerate(chars))`
- Grab index for each char in text
 - `idx = [char_indicies[c] for c in text]`
- PREPARE INPUTS
 - Get indexes of every character in steps of 3 – we will be predicting 4th character from the first 3 (4th char is the same as first in next step)
 - `cs = 3`
 - `c1_dat = [idx[i] for i in range(0,len(idx)-1-cs,cs)]`
 - `c2_dat = [idx[i+1] for i in range(0,len(idx)-1-cs,cs)]`
 - `c3_dat = [idx[i+2] for i in range(0,len(idx)-1-cs,cs)]`
 - `c4_dat = [idx[i+3] for i in range(0,len(idx)-1-cs,cs)]`
 - create inputs
 - `x1 = np.stack(c1_dat[:-2])`
 - `x2 = np.stack(c2_dat[:-2])`
 - `x3 = np.stack(c3_dat[:-2])`
 - create output
 - `y = np.stack(c4_dat[:-2])`
 - `y.shape -> 200k of sequences in this case`
- CREATE AND TRAIN MODEL
 - Number of hidden activations
 - `n_hidden_act = 256`
 - Number of latent factors ie size of embedding matrix
 - `n_fac = 42`
 - NOTE: change in the model:
 - All the arrows with the same color will use the same weight matrix.
 - The idea here is that a **character would not have different meaning (semantically or conceptually)** depending on whether it is the first, the second, or the third item in a sequence, so treat them the same -> we have arbitrarily chosen the step size



- So actually it is:
 - 1 linear layer – INPUT
 - 1 linear layer – HIDDEN
 - 1 linear layer – OUT
 - 1 embedding layer – for rich representation of characters
- Define model and forward method
 - `class Char3Model(nn.Module):`
 - `def __init__(self,vocab_size,n_fac,n_hidden_act):`
 - `super().__init__()`
 - `self.e = nn.Embedding(vocab_size,n_fac)`
 - `self.l_in = nn.Linear(n_fac,n_hidden_act)`
 - `self.l_hidden = nn.Linear(n_hidden_act,n_hidden_act)`
 - `self.l_out = nn.Linear(n_hidden_act,vocab_size)`
 - `def forward(self, c1, c2, c3):`
 - `in1 = F.relu(self.l_in(self.e(c1)))`
 - `in2 = F.relu(self.l_in(self.e(c2)))`

- `in3 = F.relu(self.l_in(self.e(c3)))`
 - Stick input 1 through embedding -> lin layer -> relu
- `h = V(torch.zeros(in1.size()).cuda())`
- `h = F.tanh(self.l_hidden(h+in1))`
- `h = F.tanh(self.l_hidden(h+in2))`
- `h = F.tanh(self.l_hidden(h+in3))`
 - Get set of activations on the orange arrow (after hidden layer)
 - Cumulation for all 3 inputs
 - First line is only there to make the three lines identical to make it easier to put in a for loop later (it's a set of zeros)
- `return F.log_softmax(self.l_out(h))`

SIZES RECAP: Generally -> the `.e()` will make it size `n_fac` -> `.l_in()` size `n_hidden` -> `.l_hidden()` size `n_hidden` (trick: **square matrix**) -> sum with another set of activations -> ... -> `.l_out()` size `vocab_size`

○ Create DATA

- We will use `ColumnarModelData` to make things easier (still good to you when doing things raw) so we don't have to fiddle with dataloaders
- Whatever you pass in parameter `xs=` (here `np.stack([x1,x2,x3])`) will be passed in `forward()` method
- `md = ColumnarModelData.from_arrays(PATH,[-1],np.stack([x1,x2,x3],axis=1),y,bs=512)`
 - Tiny data so we can use bigger batchsize

○ Create MODEL

- Standard Pytorch model will be used -> so we have call `.cuda()` to stick it to GPU
 - `m = Char3Model(vocab_size,n_fac,n_hidden_act).cuda()`

INSIGHT: whats going on in model -> grab next batch (len = 3, every of size = 512) -> extract tensors with `xs` and `y` -> use model as function by passing to it variabilized version of our inout tensors -> of (in var `t`) minibatch of size 512 and 85 which is probability of each of the vocab items (and log of them)

- `it = iter(md.trn_dl)`
- `*xs,yt = next(it)`
- `t=m(*V(xs))`

○ Create OPTIMIZER

- Standard pytorch otpim library
- `opt = optim.Adam(m.parameters(),1e-2)`
 - Pass in list of the things to optimize

○ FIT

- Using fastai method so we don't have to write the cycles
 - `fit(m,md,1,opt,F.nll_loss)`
 - `set_lrs(opt,0.001)`
 - We don't have learning features (`lr_find`, `cycles`)
 - So we have to do manual **ANNEALING** -> set LR a bit lower
 - `fit(m,md,1,opt,F.nll_loss)`

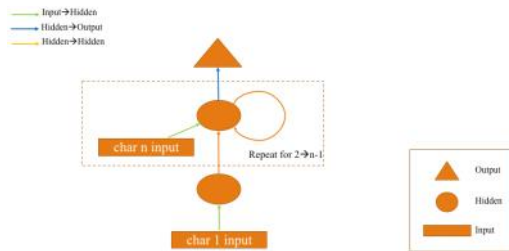
• INFERENCE/TEST

- Get char indicies for 3 letter input and create a tensor out of them -> get prediction form model `m` -> convert to np array and get the biggest value as prediction
 - `def get_next(inp):`
 - `idxs = T(np.array([char_indicies[c] for c in inp]))`
 - `p = m(*VV(idx))`
 - `i = np.argmax(to_np(p))`

- return chars[i]

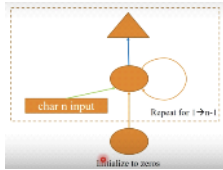
Basic RNN

We will put inside a for loop -> to make it RECURRENT nn



- Completely the same picture (but generalized) as previous but different way of writing
- Hidden2hidden layer repeat for given set of characters

It can be simplified further by only initializing with zeros (no special care to first char)



- Prepare data
 - Size of our unrolled RNN
 - cs=8
 - Create inputs
 - c_in_dat = [[idx[j+i] for i in range (cs)] for j in range(len(idx)-cs+1)]
 - xs = np.stack(c_in_dat,axis=0)
 - Create outputs
 - c_out_dat = [idx[j+cs] for j in range(len(idx)-cs+1)]
 - y = np.stack(c_out_dat)
 - val_idx = get_cv_idxs(len(idx)-cs+1)
 - md = ColumnarModelData.from_arrays(PATH,val_idx,xs,y,bs=512)
- Create and train model
 - Most important part is the for loop, which is ensuring the iteration through all inputs
 - class CharLoopModel(nn.Module):


```
def __init__(self,vocab_size,n_fac,n_hidden_act):
    super().__init__()
    self.e = nn.Embedding(vocab_size,n_fac)
    self.l_in = nn.Linear(n_fac,n_hidden_act)
    self.l_hidden = nn.Linear(n_hidden_act,n_hidden_act)
    self.l_out = nn.Linear(n_hidden_act,vocab_size)
def forward(self,*cs):
    bs = cs[0].size(0)
    h = V(torch.zeros(bs,n_hidden_act).cuda())
    for c in cs:
        inp = F.relu(self.l_in(self.e(c)))
        h = F.tanh(self.l_hidden(h+inp))
    return F.log_softmax(self.l_out(h))
```
 - INIT and FIT
 - m = CharLoopModel(vocab_size,n_fac,n_hidden_act).cuda()
 - opt = optim.Adam(m.parameters(),1e-2)
 - fit(m,md,1,opt,F.nll_loss)


Use of hyperbolic tanh

- Used in the state to state transitions because it stops the number flying of too high or too low
 - Hidden state transition matrices
 - Generally it is an offset sigmoid
 - In other cases ReLU is now mostly used

```
def forward(self, c1, c2, c3):
    in1 = F.relu(self.l_in(self.e(c1)))
    in2 = F.relu(self.l_in(self.e(c2)))
    in3 = F.relu(self.l_in(self.e(c3)))

    h = V(torch.zeros(in1.size()).cuda())
    h = F.tanh(self.l_hidden(h+in1))
    h = F.tanh(self.l_hidden(h+in2))
    h = F.tanh(self.l_hidden(h+in3))

    return F.log_softmax(self.l_out(h))
```



Because of the recurrence the network is quite deep now

- So the training gets longer

Basic RNN with concatenation of inputs instead of adding

- Reasons for concat is that input state and the hidden state are qualitatively different (Input is the encoding of a character, and h is an encoding of series of character)

MODEL

- class CharLoopConcatModel(nn.Module):
def __init__(self, vocab_size, n_fac, n_hidden_act):
 super().__init__()
 self.e = nn.Embedding(vocab_size, n_fac)
 self.l_in = nn.Linear(n_fac+n_hidden_act, n_hidden_act)
 Because we are concatenating we need to update the sizes
 self.l_hidden = nn.Linear(n_hidden_act, n_hidden_act)
 self.l_out = nn.Linear(n_hidden_act, vocab_size)
def forward(self, *cs):
 bs = cs[0].size(0)
 h = V(torch.zeros(bs, n_hidden_act).cuda())
 for c in cs:
 inp = torch.cat((h, self.e(c)), 1)
 Size: n_fac+n_hidden
 inp = F.relu(self.l_in(inp))
 Size: n_hidden
 h = F.tanh(self.l_hidden(h+inp))
 Size: n_hidden
 return F.log_softmax(self.l_out(h))

- HEURISTIC for Design of NN Architecture: if there are different types of information that we want to combine we should always consider concats because adding this together even if they are the same shape cause losing of information

PyTorch reimplementaion

- Class nn.RNN
 - Will take care of the forloop cycle in forward
 - Will create linear input layer
- class CharRNN(nn.Module):
def __init__(self, vocab_size, n_fac):
 super().__init__()
 self.e = nn.Embedding(vocab_size, n_fac)

```
self.rnn = nn.RNN(n_fac,n_hidden_act)
self.l_out = nn.Linear(n_hidden_act,vocab_size)
```

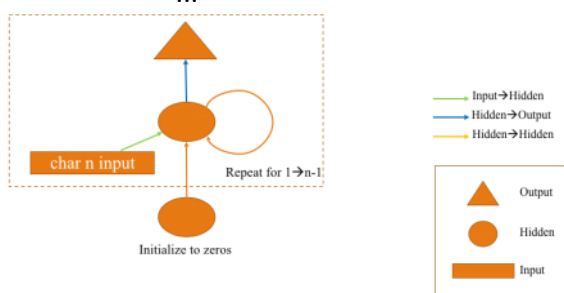
```
def forward(self, *cs):
    bs = cs[0].size(0)
    h = V(torch.zeros(1,bs,n_hidden_act))
    inp = self.e(torch.stack(cs))
    out,h = self.rnn(inp,h)

    return F.log_softmax(self.l_out(out[-1]))
```

- Initiated with zeros (as before)
- Pytorch returns the states (hidden etc.) appended to long tensor
 - That's why we reference out[-1]
- Dimensionality of h is rank 3 tensor (whereas in case before it was just rank 2 tensor)
 - Reason for that is there could be another RNN going backwards – to improve results (better relationships) -> BI-DIRECTIONAL RNN
 - Also we can have RNN feed to another RNN -> MULTILAYER RNN

Multi-output RNN

- Predict output after each character input
- Reason: it is inefficient to iterate through almost the same sequences (8 chars -> 7 same 1 new -> ...)
- So the input will be the whole text just split into sequences of 8
- Result: for an input char 0 to 7, the output would be the predictions for char 1 to 8
 - Doesnot improve accuracy but the training will be more efficient
 - `xs[:cs,:cs]`
`array([[40, 42, 29, 30, 25, 27, 29, 1],`
 `[1, 1, 43, 45, 40, 40, 39, 43],`
 `[33, 38, 31, 2, 73, 61, 54, 73],`
 `...`
 - `ys[:cs,:cs]`
`array([[42, 29, 30, 25, 27, 29, 1, 1],`
 `[1, 43, 45, 40, 40, 39, 43, 33],`
 `[38, 31, 2, 73, 61, 54, 73, 2],`
 `...`



MODEL

- Only one change from the previous and it is to keep all the outputs not just out[-1] ie the last one
- `class CharSeqRNN(nn.Module):`
`def __init__(self,vocab_size,n_fac):`
 `super().__init__()`
 `self.e = nn.Embedding(vocab_size,n_fac)`
 `self.rnn = nn.RNN(n_fac,n_hidden_act)`
 `self.l_out = nn.Linear(n_hidden_act,vocab_size)`

`def forward(self, *cs):`
 `bs = cs[0].size(0)`

```

h = V(torch.zeros(1,bs,n_hidden_act))
inp = self.e(torch.stack(cs))
out,h = self.rnn(inp,h)

return F.log_softmax(self.l_out(out))

```

- One difference is that when using NLL loss (negative log likelihood)
 - **standard NLL loss function expects two rank 2 tensors (two mini-batches of vectors)**
 - But we have 8 timesteps (eg characters cs) and for each 84 probabilities (vocab size) and for each of 512 items in minibatch -> ranks 3 tensor
 - We're gonna solve it by flattening our inputs and targets in custom NLL loss function
 - `def NLL_loss_seq(inp,targ):`
`sl,bs,nh = inp.size()`
`targ = targ.transpose(0,1).contiguous().view(-1)`
`return F.nll_loss(inp.view(-1,nh),targ)`
 - First axis is the sequence length (sl) = 8 (how many time steps)
 - Batch size second
 - Hidden state third
 - `targ.transpose(0,1).contiguous().view(-1)`
 - Pytorch when we transpose it doesn't actually transpose the inputs but keeps the information as metadata – call contiguous to actually perform the transposition
 - View(-1) flattening into a single vector (-1 = as long as it needs to be)

FIT

- `fit(m,md,4,opt,NLL_loss_seq)`
 - Fit is lowest level fastai abstraction that implements training loop
 - Rest is standard pytorch stuff (except for md that handles data loaders – wraps up training validation and test set)

[Gradient explosion](#)

Gradient explosion and identity matrix

13. ledna 2019 18:18

`self.rnn(inp, h)`

- is a loop applying the same matrix multiply again and again
- If that matrix multiply tends to increase the activations each time, we are effectively doing that to the power of 8
- Thus the activation are gonna shoot up (or very low in case of decreasing)

Solution is identity matrix

$$I_1 = [1], I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \dots, I_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

- introduced by Geoffrey Hinton et. al. in 2015 ([A Simple Way to Initialize Recurrent Networks of Rectified Linear Units](#))
- it has property -> anything times identity = same value



The diagram shows a vector a (represented as a column of circles) multiplied by a 3x3 identity matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The result is the same vector a . A red arrow points from the result back to the vector a .

- `?m.rnn`
 - Attributes: ... `weight_hh_l[k]`: the learnable hidden-hidden weights of the k-th layer, of shape `(hidden_size x hidden_size)`
- Thus we can do `m.rnn.weight_hh_l0.data.copy_(torch.eye(n_hidden_act))`
 - Puts in to hidden weights the identity matrix (`torch.eye`) of the size of hidden activations

Result: better behaved initialization -> faster training with higher learning rates

LESSON 7 – RESNETS

14. ledna 2019 6:51

OTHER NOTES

https://medium.com/@hiromi_suenaga/deep-learning-2-part-1-lesson-7-1b9503aff0c

Racap of RNNs and GRU/LSTM

[RNNs and LSTM](#)

CNNs and ResNets

- Cifar10 dataset wget <http://pirdie.com/media/files/cifar.tgz>
- Having a smaller dataset (number and size) is usually much closer to reality and can offer much quicker testing of algorithms

```
classes = os.listdir(TRN_PATH)
stats = (np.array([ 0.4914 , 0.48216, 0.44653]), np.array([ 0.24703, 0.24349, 0.26159]))
```

- Because we will be training from scratch we don't have the option that pretrained models have
- We have to calculate mean per channel (on all images) and standard deviation on all images

```
def get_data(sz,bs):
    tfms = tfms_from_stats(stats,sz,aug_tfms=[RandomFlip()],pad=sz//8)
    return ImageClassifierData.from_paths(PATH,val_name='test',tfms=tfms,bs=bs)
```

- tfms — For CIFAR 10 data augmentation, people typically do horizontal flip and black padding around the edge and randomly select 32 by 32 area within the padded image.
 - `aug_tfms=[RandomFlip()]` - creating list of specific augmentations to use
 - `pad=sz//8` - adds black padding around the picture and rotates the picture accordingly (4px in this case)

FULLY CONNECTED - SimpleNet

- From [this notebook](#)

```
class SimpleNet(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(layers[i], layers[i+1]) for i in range(len(layers)-1)])
```

```
def forward(self, x):
    x = x.view(x.size(0), -1)
    for l in self.layers:
        l_x = l(x)
        x = F.relu(l_x)
    return F.log_softmax(l_x, dim=-1)
```

- layers in pytorch must be collected inside of ModuleList
- layers[i], layers[i+1] number of features in number of out
- When going through 1) perform linear operation 2) relu

```
learn = ConvLearner.from_model_data(SimpleNet([32*32*3, 40, 10]), data)
```

- step up one level of API higher — rather than calling fit function, we create a learn object from a custom model
- `ConfLearner.from_model_data`
 - Creates model from standard PyTorch model
 - and model data object

```
learn.summary()
OrderedDict([('Linear-1',
  OrderedDict([('input_shape', [-1, 3072]),
    ('output_shape', [-1, 40]),
    ('trainable', True),
    ('nb_params', 122920)])),
  ('Linear-2',
  OrderedDict([('input_shape', [-1, 40]),
    ('output_shape', [-1, 10]),
    ('trainable', True),
    ('nb_params', 410)]))])
```

```
learn.lr_find()
%time learn.fit(lr, 2, cycle_len=1)
```

learn

```
Model architecture: SimpleNet(
  (layers): ModuleList(
    (0): Linear(in_features=3072, out_features=40, bias=True)
    (1): Linear(in_features=40, out_features=10, bias=True)
  )
)
```

- One hidden layer, one output layer
- With 122k parameters [122880, 40, 400, 10] achieved 47% accuracy

FULLY CONVOLUTIONAL - ConvNet

- Replacing fully connected model with convolutional model
- Fully connected layer is doing dot product with weight matrix (with item for every item on the input and output)
 - That's why there is a lot of parameters (3072*40) for the first layer
 - And still the accuracy is not good → not using them efficiently (working with every single pixel without any generalization)
- Convolution — going through 3 by 3 pixel (putting them through filter/kernel i.e. sumproduct of 3x3 and 3x3) and finding patterns

```
class ConvNet(nn.Module):
    def __init__(self, layers, c):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Conv2d(layers[i], layers[i+1], kernel_size=3, stride=2) for i in range(len(layers)-1)])
```

```
self.pool = nn.AdaptiveAvgPool2d(1)
self.out = nn.Linear(layers[-1],c)
```

```
def forward(self,x):
    for l in self.layers:
        x = F.relu(l(x))
    x = self.pool(x)
    x = x.view(x.size(0),-1)
    return F.log_softmax(self.out(x),dim=-1)
```

```
learn = ConvLearner.from_model_data(ConvNet([3,20,40,80],10),data)
```

- Similar code as before instead of linear layers, there are convolutional layers
 - Goal is with every layer to make the next smaller
 - In the past usually the maxpooling was used, now the stride 2 conv
 - **Stride 2 convolution - stride=2 will use every other 3 by 3 area which will halve the output resolution in each dimension (i.e. it has the same effect as 2 by 2 max-pooling)**
 - Going through every second 3x3 (to the left) and every second row (down)
 - Effect: halving the resolution with each layer
- kernel_size=3, the size of the filter

```
OrderedDict([('Conv2d-1',
  OrderedDict([('input_shape', [-1, 3, 32, 32]),
    ('output_shape', [-1, 20, 16, 16]),
    ('trainable', True),
    ('nb_params', 560)])),
  ('Conv2d-2',
  OrderedDict([('input_shape', [-1, 20, 16, 16]),
    ('output_shape', [-1, 40, 8, 8]),
    ('trainable', True),
    ('nb_params', 7240)])),
  ('Conv2d-3',
  OrderedDict([('input_shape', [-1, 40, 8, 8]),
    ('output_shape', [-1, 80, 4, 4]),
    ('trainable', True),
    ('nb_params', 2880)])),
  ('AdaptiveAvgPool2d-4',
  OrderedDict([('input_shape', [-1, 80, 4, 4]),
    ('output_shape', [-1, 80, 1, 1]),
    ('nb_params', 0)])),
  ('Linear-5',
  OrderedDict([('input_shape', [-1, 80]),
    ('output_shape', [-1, 10]),
    ('trainable', True),
    ('nb_params', 810)]))])
```

- 15x15 -> 7x7 -> 3x3 -> 80x1x1 -> 10 classes

- An pooling layer before the output – average pooling
 - Standard for state of the art
 - In last layer – but rather than doing maxpool we do adaptive pooling
 - Not saying the area to pool (ie 2x2 or 3x3...) we define the size of the output (here 1 by 1) -> find the single largest cell and use it as an activation -> 1x1 by number of features tensor
- ```
x = x.view(x.size(0),-1)
Returns minibatch by number of features
self.out = nn.Linear(layers[-1],c)
Splits out number of classes
```

|                     |                                                  |
|---------------------|--------------------------------------------------|
| Fully convolutional | every layer is convolutional except for the last |
|---------------------|--------------------------------------------------|

```
learn.lr_find(end_lr=100)
```

- Validation loss kept getting better under standard settings thus increasing the final lr to 100

## FULLY CONVOLUTIONAL – ConvNet2 (refactored)

```
class ConvLayer(nn.Module):
 def __init__(self,ni,nf):
 super().__init__()
 self.conv = nn.Conv2d(ni, nf, kernel_size=3, stride=2, padding=1)
 def forward(self,x):
 return F.relu(self.conv(x))
```

- Creating a separate nn.Module class for the convolutional layer
  - PyTorch definition of NN and NN Layer are literally the same
  - Thus layer can be used as NN and NN can be used as layer
- Padding=1
  - Adds line of zeros around the whole picture
  - So that also the first pixel can be in the middle of the kernel
  - Avoid to go from 32 by 32 to 16 by 16 but actually 15 by 15 (matters especially for the small sizes 3x3 where you don't want to through away the whole piece)

```
class ConvNet2(nn.Module):
 def __init__(self,layers,c):
 super().__init__()
 self.layers = nn.ModuleList(
 [ConvLayer(layers[i],layers[i+1]) for i in range(len(layers)-1)])
 self.out = nn.Linear(layers[-1],c)
 def forward(self,x):
 for l in self.layers:
 x = F.relu(l(x))
 x = F.adaptive_max_pool2d(x,1)
 x = x.view(x.size(0),-1)
 return F.log_softmax(self.out(x),dim=-1)
```

- Same as before only using new class for creation of layers
- And to save coding not creating POOLING as object
  - But because it has no weights it can be just a function call

## BatchNorm

- Problem with previous NN is when scaling up – adding more layers
  - With larger lr it goes NaN with smaller it takes very long time
  - Not resilient
- To avoid this BatchNorm to be used
- Problem is that it is very hard to keep activations when processing in reasonable scale because either weight matrix will likely be causing some potentially big growth (causing gradient explosion) or the opposite (gradient vanishing)
- We usually start with 0 mean std 1 by normalizing the inputs – but we want to normalize every

layer

[http://onlinestatbook.com/lms/normal\\_distribution/standard\\_normal.html](http://onlinestatbook.com/lms/normal_distribution/standard_normal.html)  
Standard Normal Distribution

```
class BnLayer(nn.Module):
 def __init__(self, ni, nf, stride=2, kernel_size=3):
 super().__init__()
 self.conv = nn.Conv2d(ni, nf, kernel_size=kernel_size, stride=stride,
 bias=False, padding=1)
 self.a = nn.Parameter(torch.zeros(nf, 1, 1))
 self.m = nn.Parameter(torch.ones(nf, 1, 1))
```

- New added value a for each channel (thast why bias is false)
- New multiplier m for each channel
  - Nf number of filters x 1 x 1 – zeros and ones for the first layer
  - But because it is nn.Parameter PyTorch is allowed to learn these as weights

```
def forward(self, x):
 x = F.relu(self.conv(x))
 x_chan = x.transpose(0, 1).contiguous().view(x.size(1), -1)
 if self.training:
 self.means = x_chan.mean(1)[None, None]
 self.stds = x_chan.std(1)[None, None]
 return (x - self.means) / self.stds * self.m + self.a
```

- Mean of each channel/filer, Std of each channel/filter
  - For input per channel for later layers per filter
- Then subtract the means and divide by standard deviations
  - (x-self.means)/self.stds just this wouldnt be because of SGD nature enough (in next minibatch it would undo it again)
  - Thats why there are a and m as nn.Parameters
    - Self.m - if it wants to **scale** the layer up, it does not have to scale up every single value in the matrix. It can just scale up this single trio of numbers self.m ,
    - Self.a - if it wants to **shift** it all up or down a bit, it does not have to shift the entire weight matrix, they can just shift this trio of numbers self.a
    - Intuitively: allowing the SGD to to shift and scale the layers using far fewer parameters than would have been necessary if it were to actually shift and scale the entire set of convolutional filters
    - Result: can add more layer and work with higher LR's
- Another thing that batchnorm does is that it is REGULARIZING
  - Dropout and weight decay can be decreased or removed
  - Because every minibatch will have defferent mean and std than the previous
  - ie self.means = x\_chan.mean(1)[None, None], self.stds = x\_chan.std(1)[None, None] these things keep changing and so its adding regularization effect because its noise
    - Note: *actually in real version, it does not use this batch's mean and standard deviation but takes an exponentially weighted moving average standard deviation and mean.*
  - Adding noise have regularization effect

**Importat: if self.training**

- True if when applied on training set
- False when applied on validation set
- Important because when you go through validation set you dont want to be changing meaning of the model
- !!! There are parts of NN that are sensitive to what the mode of the network is whether it is in training mode or evaluation/test mode
  - Operations you dont want to be performing in test time
  - in PyTorch, there are two such layer: dropout and batch norm
    - nn.Dropout already does the check
- This is also related to **FREEZING of layers**
  - Even in training these parameters should not be updated for every layer because with a pre-trained network, that is a terrible idea
  - If you have a pre-trained network for specific values of those means and standard deviations in batch norm, if you change them, it changes the meaning of those pre-trained layers.
  - In fast.ai, always by default, it will not touch those means and standard deviations if your layer is frozen. As soon as you un-freeze it, it will start updating them unless you **set learn.bn\_freeze=True** (means never tochu the means and stds)
    - Tend to work better with pretrained models especially if the data are close to the ones it was pretrained on

|                |                                                                             |
|----------------|-----------------------------------------------------------------------------|
| Ablation study | Turning different pieces of the NN on and off to see what makes what impact |
|----------------|-----------------------------------------------------------------------------|

- In the original batchnorm paper this wasn't done so it is considered they hadn't put the batchnorm in the right place
  - Originally put after ReLU
  - But there is probably a better place – Jeremy didn't said where

```
class ConvBnNet(nn.Module):
 def __init__(self, layers, c):
 super().__init__()
 self.conv1 = nn.Conv2d(3, 10, kernel_size=5, stride=1, padding=2)
 self.layers = nn.ModuleList(
 [BnLayer(layers[i], layers[i+1]) for i in range(len(layers)-1)])
 self.out = nn.Linear(layers[-1], c)
```

```
def forward(self, x):
 x = self.conv1(x)
 for l in self.layers:
 x = l(x)
 x = F.adaptive_max_pool2d(x, 1)
 x = x.view(x.size(0), -1)
 return F.log_softmax(self.out(x), dim=-1)
```

- Rest is the same
- One difference - single convolutional layer added at the start to get closer to current approaches



- Bigger kernel size 5 and stride 1 to produce richer input for the first layer
- Its standard approach in most of up to date architectures even 7x7 or 11x11
  - With quite a lot of filters like 32
- Since padding = kernel\_size — 1 / 2 and stride=1, the input size is the same as the output size — just got more filters.
- Way of creating richer starting point for the sequence of convolutional layers

```
learn = ConvLearner.from_model_data(ConvBnNet([10,20,40,80,100],10),data)
```

Fit

## Deep BatchNorm

- We cant add more stride 2 layers because it halves the size of the image with every layer -> there is no room left -> **we have to add stride 1 layers**

```
class ConvBnNet2(nn.Module):
```

```
def __init__(self, layers, c):
 super().__init__()
 self.conv1 = nn.Conv2d(3, 10, kernel_size=5, stride=1, padding=2)
 self.layers = nn.ModuleList(
 [BnLayer(layers[i], layers[i+1])
 for i in range(len(layers)-1)])
 self.layers2 = nn.ModuleList(
 [BnLayer(layers[i+1], layers[i+1], 1)
 for i in range(len(layers)-1)])
 self.out = nn.Linear(layers[-1], c)
```

```
def forward(self, x):
 x = self.conv1(x)
 for l, l2 in zip(self.layers, self.layers2):
 x = l(x)
 x = l2(x)
 x = F.adaptive_max_pool2d(x, 1)
 x = x.view(x.size(0), -1)
 return F.log_softmax(self.out(x), dim=-1)
```

```
learn = ConvLearner.from_model_data(ConvBnNet2([10,20,40,80,160],10),data)
```

Fit

- It is twice as deep but we end up with the same 2x2 output
- But it didnt help much for the accuracy
  - It is too deep even for the batch norm to handle
  - It gets harder to train this network properly (but it is possible)

## ResNet

- Our final stage

```
class ResnetLayer(BnLayer):
```

```
def forward(self, x): return x + super().forward(x)
```

- Small but major change in the layer — **RESNET BLOCK**
- Predictions = inputs + convolution of inputs
  - $y = x + f(x)$
  - $f(x) = y - x$
  - fitting function between the difference of  $y$  (thing I am trying to calculate) and  $x$  (thing I have most recently calculated) -> **RESIDUAL**
    - = error of what I have calculated so far
    - = try to find set of convolutional weights that would fill the amount I was off by
- Based on **BOOSTING** theory
  - have a function which tries to predict the error (i.e. how much we are off by)
  - then we add a prediction of how much we were wrong by to the input, then add another prediction of how much we were wrong by that time, and repeat that layer after layer
  - **zooming into the correct answer (via predicting residuals/error)**

```
class Resnet(nn.Module):
```

```
def __init__(self, layers, c):
 super().__init__()
 self.conv1 = nn.Conv2d(3, 10, kernel_size=5, stride=1, padding=2)
 self.layers = nn.ModuleList(
 [BnLayer(layers[i], layers[i+1])
 for i in range(len(layers)-1)])
 self.layers2 = nn.ModuleList(
 [ResnetLayer(layers[i+1], layers[i+1], 1)
 for i in range(len(layers)-1)])
 self.layers3 = nn.ModuleList(
 [ResnetLayer(layers[i+1], layers[i+1], 1)
 for i in range(len(layers)-1)])
 self.out = nn.Linear(layers[-1], c)
```

```
def forward(self, x):
 x = self.conv1(x)
 for l, l2, l3 in zip(self.layers, self.layers2, self.layers3):
 x = l3(l2(l(x)))
 x = F.adaptive_max_pool2d(x, 1)
 x = x.view(x.size(0), -1)
 return F.log_softmax(self.out(x), dim=-1)
```

- Then add bunch of layers and make it 3 times deeper, ad it still trains beautifully just because of  $x + \text{super().forward}(x)$  .
- Three layers per group — one conv batchnormed, two residual

## Actual/full architecture of Resnet block

- Resnet was very important development because it allows much deeper NN
- In reality it has mostly two convolutional layers before the residual prediction
- The BnLayer is called **BOTTLENECK BLOCK/LAYER**

- Changing the geometry of the architecture
- Actual resnet uses different bottleneck block then pure convolution – here it is simplification

```
%time learn.fit(lr,4,cycle_len=1,wds=wd)
```

Fit

- also added weight decay

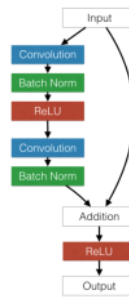
### Resnet extension

- Increased sizes
- Added dropout
- Reasonable approximation/starting point for modern architectures

```
class Resnet(nn.Module):
 def __init__(self, layers, c, p=0.5):
 super().__init__()
 self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=1, padding=2)
 self.layers = nn.ModuleList(
 [BnLayer(layers[i], layers[i+1])
 for i in range(len(layers)-1)])
 self.layers2 = nn.ModuleList(
 [ResnetLayer(layers[i+1], layers[i+1], 1)
 for i in range(len(layers)-1)])
 self.layers3 = nn.ModuleList(
 [ResnetLayer(layers[i+1], layers[i+1], 1)
 for i in range(len(layers)-1)])
 self.drop = nn.Dropout(p)
 self.out = nn.Linear(layers[-1], c)

 def forward(self, x):
 x = self.conv1(x)
 for l, l2, l3 in zip(self.layers, self.layers2, self.layers3):
 x = l2(l(l(x)))
 x = F.adaptive_max_pool2d(x, 1)
 x = x.view(x.size(0), -1)
 x = self.drop(x)
 return F.log_softmax(self.out(x), dim=-1)
```

```
learn = ConvLearner.from_model_data(Resnet([16,32,64,128,256], 10, 0.2), data)
```



### Summary

- Jeremy got to 82percent which is 2012/13 state of the art
- Nowadays its 97proc but it is all based on the same approaches but with some tweaks in resnet, better data augumentation, better regularization but noting seriously different
- In NLP, “**transformer architecture**” recently appeared and was shown to be the state of the art for translation, and it has a simple ResNet structure in it. This general approach is called “**skip connection**” (i.e. the idea of skipping over a layer) and appears a lot in computer vision, but nobody else much seems to be using it even through there is nothing computer vision specific about it

|                                        |                                                                                                                                                                                                                           |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Resnets trailing/numbering (34,50 etc) | Parameters reflecting number of resnet blocks etc.<br><a href="https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py">https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py</a> |
| PreAct Resnet                          | Currently most effective architecture                                                                                                                                                                                     |

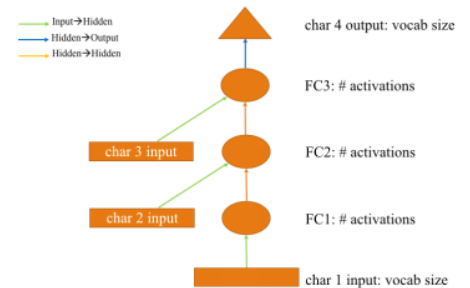
[ResNet and DogsCats](#)

# RNNs and LSTM

14. ledna 2019 7:09

## Recap RNNs

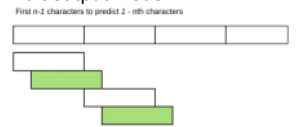
- Arrows one or more layer operations (linear matrix mult, nonlin relu/tanh)
- Same color same weight matrix
- Difference of RNN is that inputs are also coming into deeper levels (2 and 3)
- By using `nn.LINEAR` we are getting automatically
  - Weight matrix
  - Bias vector
- BASIC - Forward
  - Input
    - Get embeddings
    - Put it through linear (matrix mult)
    - Apply ReLU
  - Hidden
    - Hidden linear
    - Apply Tanh
    - Because there is no orange arrow in the first we have put an empty matrix to keep all of the hidden operations the same
- LOOP - Forward
  - Exactly the same but with refactor code to iterate the same lines of code in hidden layer
- Nn.RNN
  - Another refactoring with using `nn.RNN()` that replaced the loop with one single line of code



## General approach (quite wasteful)



## Multioutput model



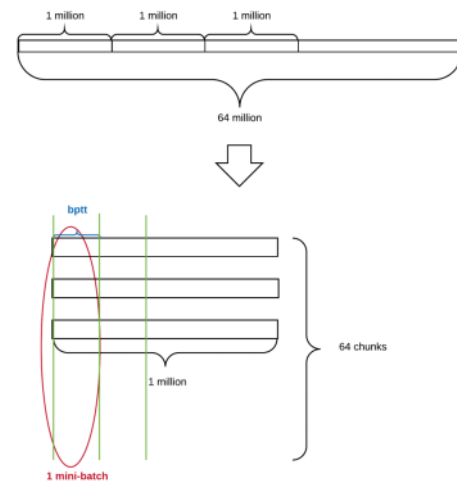
## Stateful RNN

- There is one issue though and it's this line of code `h = V(torch.zeros(1,bs,n_hidden_act))`
    - It is called everytime during the forward method (with every minibatch)
      - Resetting it with bunch of zeros
      - thus when we are predicting the first character it has nothing to predict it from
  - `class CharSeqStatefulRnn(nn.Module):`
    - `def __init__(self, vocab_size, n_fac, bs):`
      - `self.vocab_size = vocab_size`
      - `super().__init__()`
      - `self.e = nn.Embedding(vocab_size, n_fac)`
      - `self.rnn = nn.RNN(n_fac, n_hidden)`
      - `self.l_out = nn.Linear(n_hidden, vocab_size)`
      - `self.init_hidden(bs)`
    - `def forward(self, cs):`
      - `bs = cs[0].size(0)`
      - `if self.h.size(1) != bs: self.init_hidden(bs)`
      - `outp, h = self.rnn(self.e(cs), self.h)`
      - `self.h = repackage_var(h)`
      - `return F.log_softmax(self.l_out(outp), dim=-1).view(-1, self.vocab_size)`
    - `def init_hidden(self, bs): self.h = V(torch.zeros(1, bs, n_hidden))`
  - Important parts are
    - `self.init_hidden(bs)` - initialization with zeros but only in constructor
- [ISSUE/WRINKLE #1](#)
- `self.h = repackage_var(h)`
    - if we were to simply do `self.h = h`, and we trained on a document that is a million characters long, then the size of unrolled version of the RNN has a million layers (ellipses)
      - million layer fully connected network is going to be very memory intensive because in order to do a chain rule, we have to multiply one million layers while remembering all one million gradients every batch
      - `def repackage_var(h):`
        - `return Variable(h.data) if type(h) == Variable else tuple(repackage_var(v) for v in h)`

- Remembering only last state and forgetting the history
- Approach (**BACKPROP THROUGH TIME = BPTT**)
  - Call it in forward eg do 8 characters -> backpropagate -> keep track of hidden layers values (keep hidden state and history)
  - At the end of those end -> throw away history of operations (keep state but forget history)
  - BPTT = how many layers to backprop through (in NLP NN in lesson 4 it was set to 70)
  - Reason NOT to backprop through too many layers:
    - In case of any gradient instability (explosion, vanishing) -> the more layers we have the harder the network gets to train (slower, less resilient)
  - Longer value of BPTT:
    - Allows to capture longer memory in more state

#### ISSUE/WRINKLE #2

- How are we gonna create the minibatches as we do not want to process one section at a time, but a bunch in parallel at a time
- We are gonna process BS amount of chunks at a time which are separated at the length of BPTT
  - For a document that is 64 million characters long, each "chunk" will be 1 million characters.
  - We stack them together and now split them by bptt — 1 mini-batch consists of 64 by bptt matrix.
  - PREDICTION: we predict everything in the 64 chunks offset by 1
- Ideal BPTT size is more tight to performance
  - When training the tensor that gets processed is  $bptt \times bs \times embeddings \times hidden\ state$
  - So it's mostly about ram and gpu + stability of training
  - Otherwise as high as possible



TEXT = data.Field(lower=True, tokenize=list)

- Preprocessing of data
- List tokenization and case lowering
- Important attribute after creation is TEXT.vocab
  - TEXT.vocab.itos list of unique items in the vocabulary
  - TEXT.vocab.stoi is a reverse mapping from each item to number

bs=64; bptt=8; n\_fac=42; n\_hidden=256

- Standard parameters
- N\_hidden size of each of the created hidden states
- TorchText actually randomize a bit the bptt parameter so that it is more real, 5% of the time, it will cut it in half

FILES = dict(train=TRN\_PATH, validation=VAL\_PATH, test=VAL\_PATH)

- Dictionary of data sets

md = LanguageModelData.from\_text\_files(PATH, TEXT, \*\*FILES, bs=bs, bptt=bptt, min\_freq=3)

len(md.trn\_dl), md.nt, len(md.trn\_ds), len(md.trn\_ds[0].text)

- Number of minibatches, number of tokens, ..., length of a corpus

#### ISSUE/WRINKLE #3

- if self.h.size(1) != bs: self.init\_hidden(bs)
  - Check whether the width is the same as batch size (it must be due to multiplication) it get initialized with zeros again
  - Size of last mini batch depends on length of corp / bs x bptt and resulting remains
  - Happens at the end of each epoch

#### ISSUE/WRINKLE #4

- return F.log\_softmax(self.l\_out(outp), dim=-1).view(-1, self.vocab\_size)
  - Pytorch functions can't work with rank 3 tensor so we have to flatten it out to rank 2 with view(-1) -> these are the predictions
  - For the actuals pytorch actually flattens it automatically
  - F.log\_softmax(self.l\_out(outp), dim=-1)
    - Which axis we want to do the softmax on (which axis to sum)
    - -1 over the last one

Tanh repeat:

- Tahn = sigmoid\*2-1
- tanh is forcing the value to be between -1 and 1 (to keep the outcome in reasonable levels)
- Since we are multiplying by this weight matrix again and again, we would worry that relu (since it is unbounded) might have more gradient explosion problem


RNNCell:

- Can customize it to use different nonlinearity etc
- Then we have to write own forloop
- self.rnn = nn.RNNCell(n\_fac, n\_hidden)
- ...
- outp = []
  - o = self.h
  - for c in cs:
 o = self.rnn(self.e(c), o)
  - outp.append(o)
  - outp = self.l(torch.stack(outp))

```

for c in cs:
 o = self.rnn(self.e(c), o)
 outp.append(o)
outp = self.l_out(torch.stack(outp))

```

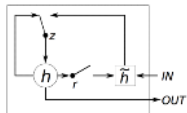
$$\frac{e^x}{1+e^x} = \text{sigmoid}$$


sigmoid

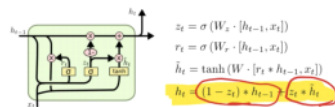
## GRU Gated Recurrent Units

- In practice few people use RNNCell
- More used is GRU
  - <http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-gru-lstm-rnn-with-python-and-theano/>
  - <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

$$\begin{aligned}
 z &= \sigma(x_t U^z + s_{t-1} W^z) \\
 r &= \sigma(x_t U^r + s_{t-1} W^r) \\
 \tilde{h} &= \tanh(x_t U^h + (s_{t-1} \circ r) W^h) \\
 s_t &= (1 - z) \circ \tilde{h} + z \circ s_{t-1}
 \end{aligned}$$



- Two important parts (gates)
  - reset gate - r - determines how to combine the new input with the previous memory
    - How much of hidden state to throwaway and replace by new input
  - and an update gate - z - defines how much of the previous memory to keep
    - How much to update my hidden state (or leave it the same)
    - Linear interpolation (to what degree use new hid state and previous hid state)



- Gates are not actual switches as drawn on pictures but these weight determining how much to update

- Z+R
  - Are basically mini NNs/models inside of NN (with on WEIGHT MATRIX) – can learn eg when there is full stop to forget all previous states

```

class CharSeqStatefulGRU(nn.Module):
 def __init__(self, vocab_size, n_fac, bs):
 self.vocab_size = vocab_size
 super().__init__()
 self.e = nn.Embedding(vocab_size, n_fac)
 self.rnn = nn.GRU(n_fac, n_hidden)
 self.l_out = nn.Linear(n_hidden, vocab_size)
 self.init_hidden(bs)

 def forward(self, cs):
 bs = cs[0].size(0)
 if self.h.size(1) != bs: self.init_hidden(bs)
 outp, h = self.rnn(self.e(cs), self.h)
 self.h = repackage_var(h)
 return F.log_softmax(self.l_out(outp), dim=-1).view(-1, self.vocab_size)

 def init_hidden(self, bs):
 self.h = (V(torch.zeros(1, bs, n_hidden)))

```

## LSTM

- Long short term memory
- from fastai import sgdr
- n\_hidden=512

```

class CharSeqStatefulLSTM(nn.Module):
 def __init__(self, vocab_size, n_fac, bs, nl):
 super().__init__()
 self.vocab_size, self.nl = vocab_size, nl
 self.e = nn.Embedding(vocab_size, n_fac)
 self.rnn = nn.LSTM(n_fac, n_hidden, nl, dropout=0.5)
 self.l_out = nn.Linear(n_hidden, vocab_size)
 self.init_hidden(bs)

 def forward(self, cs):
 bs = cs[0].size(0)
 if self.h[0].size(1) != bs: self.init_hidden(bs)
 outp, h = self.rnn(self.e(cs), self.h)
 self.h = repackage_var(h)
 return F.log_softmax(self.l_out(outp), dim=-1).view(-1, self.vocab_size)

 def init_hidden(self, bs):

```

```
self.h = ((V(torch.zeros(1,bs,n_hidden))),
 (V(torch.zeros(1,bs,n_hidden))))
```

- Have one more state which is the CELL STATE
  - Thus for the initialization here has to be tuple of zero matrices
- Added dropout and doubled number of hidden states
  - Goal: learn more but be more resilient

#### Using callbacks (specifically SGDR) without Learner class

```
m = CharSeqStatefulLSTM(md.nt,n_fac,512,2).cuda()
lo = LayerOptimizer(optim.Adam,m,1e-2,1e-5)
```

- Fastai class
- Takes in optimizer, model, lr and optional weightdecay
- Key reason why LayerOptimizer exists is to do differential learning rates and differential weight decay
- If you want to use callbacks or SGDR in code you are not using the Learner class, you need to use this
- Lo.opt
  - Returns the optimizer
- fit

```
on_end = lambda sched, cycle: save_model(m,f'{PATH}models/cyc_{cycle}')
cb = [CosAnneal(lo,len(md.trn_dl),cycle_mult=2,on_cycle_end=on_end)]
fit(m,md,2*4-1,lo.opt,F.nll_loss,callbacks=cb)
```

- Cosine Annealing callback
- Need as parameters: layeroptimizer, number of batches
  - does cosine annealing by changing learning rate inside the lo object
- Optional: what to do at the end of cycle – here we save the model
  - Another callback
- We can do callback at a start of a training, epoch or a batch, or at the end of a training, an epoch, or a batch

#### TEST

```
def get_next(inp):
 idxs = TEXT.numericalize(inp)
 p = m(VV(idxs.transpose(0,1)))
 r = torch.multinomial(p[-1].exp(),1)
 return TEXT.vocab.itos[to_np(r)[0]]
```

- Interesting is to test the character based model at different loss stages – difference between 1.3 and 1.2 is huge (at higher there is random punctuation etc.)
- If anything like this appears – train more

# \_comparison of CNN architectures

18. ledna 2019 9:10

| Architecture                  | Layers | Summary                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | Accuracy after 6 epochs | Time per 2 epochs |
|-------------------------------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|-------------------|
| Fully connected (SimpleNet)   | 2      | <pre>OrderedDict([('Linear-1',   OrderedDict([('input_shape', [-1, 3072]),     ('output_shape', [-1, 40]),     ('trainable', True),     ('nb_params', 122920)])),   ('Linear-2',   OrderedDict([('input_shape', [-1, 40]),     ('output_shape', [-1, 10]),     ('trainable', True),     ('nb_params', 410)]))])</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | 0.47                    | 59.3 s            |
| Fully convolutional (ConvNet) | 4      | <pre>OrderedDict([('Conv2d-1',   OrderedDict([('input_shape', [-1, 3, 32, 32]),     ('output_shape', [-1, 20, 15, 15]),     ('trainable', True),     ('nb_params', 560)])),   ('Conv2d-2',   OrderedDict([('input_shape', [-1, 20, 15, 15]),     ('output_shape', [-1, 40, 7, 7]),     ('trainable', True),     ('nb_params', 7240)])),   ('Conv2d-3',   OrderedDict([('input_shape', [-1, 40, 7, 7]),     ('output_shape', [-1, 80, 3, 3]),     ('trainable', True),     ('nb_params', 28880)])),   ('AdaptiveAvgPool2d-4',   OrderedDict([('input_shape', [-1, 80, 3, 3]),     ('output_shape', [-1, 80, 1, 1]),     ('nb_params', 0)])),   ('Linear-5',   OrderedDict([('input_shape', [-1, 80]),     ('output_shape', [-1, 10]),     ('trainable', True),     ('nb_params', 810)]))])</pre> | 0.5661                  | 1min 2s           |
| Batch normed CNN (ConvBnNet)  | 6      | + bn                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 0.6698                  | 1min 6s           |
| Deep BatchNorm                | 10     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | 0.6876                  | 1min 9s           |
| Resnet                        | 14     | + wd                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 0.6739                  | 1min 22s          |
| Resnet extension              | 15     | + wd<br>+ dropout                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | 0.6727                  | 1min 28s          |

# ResNet and DogsCats

20. ledna 2019 11:32

- More manual approach to create the model

```
from fastai.imports import *
```

```
from fastai.transforms import *
from fastai.conv_learner import *
from fastai.model import *
from fastai.dataset import *
from fastai.sgdr import *
from fastai.plots import *
```

```
arch=resnet34
m=arch(True)
```

- Calling arch function to get model
- True to get pretrained model on ImageNet
- M contains the model

```
ResNet(
 (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (maxpool): MaxPool2d(kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), dilation=(1, 1), ceil_mode=False)
 (layer1): Sequential(
 (0): BasicBlock(
 (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
)
 (1): BasicBlock(
 (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
)
 (2): BasicBlock(
 (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
)
)
 (layer2): Sequential(
 (0): BasicBlock(
 (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
 (downsample): Sequential(
 (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
 (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
)
)
 (1): BasicBlock(
 (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
)
)
)
```

ADDITION  
RESNET B.

BOTTLENECK



```

)
(2): BasicBlock(
 (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
)
(3): BasicBlock(
 (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
)
)
(layer3): Sequential(
 (0): BasicBlock(
 (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (downsample): Sequential(
 (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
)
)
 (1): BasicBlock(
 (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
)
 (2): BasicBlock(
 (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
)
 (3): BasicBlock(
 (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
)
 (4): BasicBlock(
 (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
)
 (5): BasicBlock(
 (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True)
)
)
)
(layer4): Sequential(
 (0): BasicBlock(
 (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
 (downsample): Sequential(
 (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
 (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
)
)
 (1): BasicBlock(
 (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

```

```

(bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
(relu): ReLU(inplace)
(conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
)
(2): BasicBlock(
 (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
 (relu): ReLU(inplace)
 (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
)
)
(avgpool): AvgPool2d(kernel_size=7, stride=7, padding=0, ceil_mode=False, count_include_pad=True)
(fc): Linear(in_features=512, out_features=1000, bias=True)
)

```

- We need to get rid of **this** – predicting 1000 features just as it is on ImageNet
  - When using ConvLearner from pretrained – it actually deletes these layers
  - Fastai replaces with:
    - Adaptive average pooling
    - Adaptive max pooling
    - And concatenates the together

```

m = nn.Sequential(*children(m)[: -2],
 nn.Conv2d(512, 2, 3, padding=1),
 nn.AdaptiveAvgPool2d(1), Flatten(),
 nn.LogSoftmax())

```

- Get all layers except last two
- Add one more conv layer with 2 outputs
- SoftMax
- It is gonna be model without fully connected layer at the end
  - But it is sort of replaced by the convlayer
  - 2 filters x 7 x 7 and once we do the average pooling it is gonna be just **two numbers**
  - Different way of producing/limiting final predictions (as opposed to FC)

```

tfms = tfms_from_model(arch, sz, aug_tfms=transforms_side_on, max_zoom=1.1)
data = ImageClassifierData.from_paths(PATH, bs, tfms)
learn = ConvLearner.from_model_data(m, data)

```

**learn.freeze\_to(-4)**

- Freezing up until the added conv layer
- Test m[-4].trainable - **TRUE**

```

learn.fit(1e-2, 1)
learn.fit(1e-2, 1, cycle_len=1)

```

- Got to 99 percent

[Class Activation Maps \(CAM\)](#)

# CAM (Class Activation Maps)

neděle 20. ledna 2019 14:47

Using CAM to find which parts of the image turned out to be important

---

Created by producing this matrix

```
array([[0.06796, 0.09529, 0.04571, 0.02226, 0.01783, 0.02682, 0.],
 [0.18074, 0.25366, 0.21349, 0.17134, 0.12896, 0.07998, 0.02269],
 [0.32468, 0.53808, 0.58196, 0.51675, 0.36953, 0.20348, 0.06563],
 [0.4753 , 0.82032, 0.93738, 0.84066, 0.58015, 0.30568, 0.09089],
 [0.53621, 0.90491, 1. , 0.87108, 0.60574, 0.33723, 0.11391],
 [0.44609, 0.70038, 0.72927, 0.60194, 0.42627, 0.26258, 0.10247],
 [0.21954, 0.3172 , 0.30497, 0.21987, 0.15255, 0.08741, 0.00709]], dtype=float32)
```

- Where the big numbers correspond to cat
- This matrix equals to the value of feature matrix feat times py vector
  - Py vector is predictions vector/tensor
  - Feat is the features tensor produced by the added conv layer

Get the data

```
class SaveFeatures():
 features=None
 def __init__(self,m): self.hook = m.register_forward_hook(self.hook_fn)
 def hook_fn(self, module, input, output): self.features = to_np(output)
 def remove(self): self.hook.remove()
```

```
x,y = next(iter(data.val_dl))
x,y = x[None,1],y[None,1]
vx=Variable(x.cuda(),requires_grad=True)
```

```
dx=data.val_ds.denorm(x)[0]
plt.imshow(dx)
```

- Getting the features is done via hook
  - "Hook" is the mechanism that lets us ask the model to return the matrix
  - register\_forward\_hook asks PyTorch that every time it calculates a layer it runs the function given — sort of like a callback that happens every time it calculates a layer
  - In this case it saved the value of particular layer

Get the py and feat vectors

```
sf = SaveFeatures(m[-4]) #adding hook
py = m(Variable(x.cuda()))
sf.remove()
py = np.exp(to_np(py)[0]);py

feat = np.maximum(0,sf.features[0]) #grab values
feat.shape
```

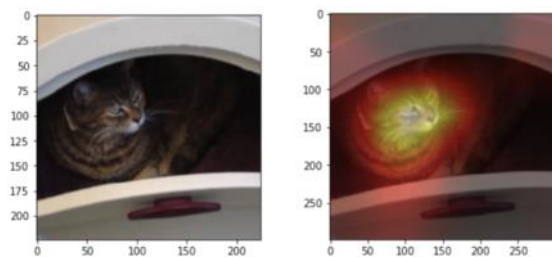
- Py is just array([1., 0.], dtype=float32)
- Feat has shape (2, 7, 7)
  - Which is result of the last conv layer we have added `nn.Conv2d(512,2,3,padding=1)`
- It will end up picking everything from the first channel and nothing from the second from the feat tensor (due to 1/0 in the py tensor) — which is lined up with features connected to cats
  - In other words the average pool took the 7x7 tensor and averaged out which parts are the most catlike

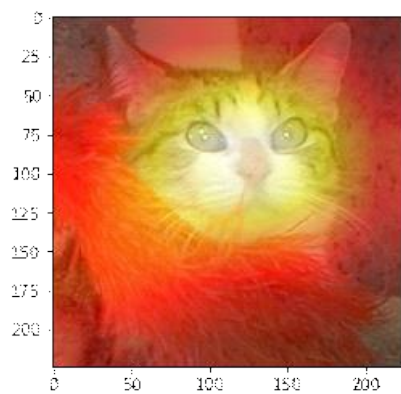
```
f2 = np.dot(np.rollaxis(feat,0,3),py)
f2=f2.min()
f2/=f2.max()
f2
```

Plotting

```
plt.imshow(dx)
plt.imshow(scipy.misc.imresize(f2,dx.shape),alpha=0.5,cmap='hot')
```

- Taking the "average cattiness" matrix and resizing it to match the original picture
- And just overlay it on top





## USECASE

- If we have very big picture we can calculate this matrix with small convnet -> crop it out -> and rerun it just on this part
  - To focus on the relevant area
  - Zoom in

## \_NEXT

neděle 20. ledna 2019

15:28

“If you intend to come to Part 2, you are expected to master all the techniques er have learned in Part 1”. Here are something you can do:

1. Watch each of the video at least 3 times.
2. Make sure you can re-create the notebooks without watching the videos — maybe do so with different datasets to make it more interesting.
3. Keep an eye on the forum for recent papers, recent advances.
4. Be tenacious and keep working at it! SO DO THOSE THAT SUCCEED!