fastbook ch.1 by @dk21

DEEP LEARNING APPLICATIONS: NLP, Vision, Medicine, Biology, Image generation, Recommendation systems, Robotics, Logistics, Finance

ML: discipline where we define program not by writing it entirely ourselves, but by learning from data

DL: subset of ML, using NN with multiple layers

TEACHING APPROACH: TOP-DOWN (VIA D. PERKINS, BASEBALL ANALOGY)

① Start with real, practical examples (teach the whole game)

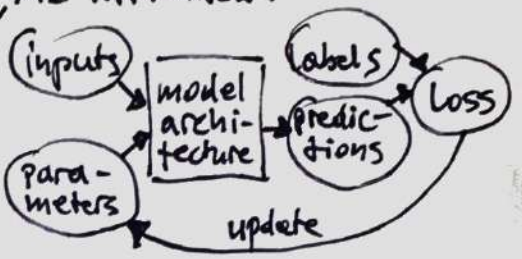
② Learn by doing: run/re-implement code. Do personal projects

③ Deep dive into theory later, as needed - to improve models

+ Simplify and remove barriers:
- fastai library
- Pytorch
- Jupyter

CLASSICAL PROGRAM: input → program → results

ML APPROACH:



ML DURING INFERENCE: input → model → results

LIMITATIONS OF ML:
- Need data with labels
- Can only learn patterns seen in ~~the~~ input data
- predictions vs recommended actions

WATCH OUT: FEEDBACK LOOPS! e.g. Youtube recommending viral anti-vax videos

Some history:

1943 - Mcculloch, Pitts: Artificial Neuron

Rosenblatt's device: Mark I Perceptron

Minsky, Papert: Perceptrons - book introducing multiple layer neural networks

1986 - Parallel Distributed Processing - book introducing most of current DL framework

1961/62: Samuel - Artificial Intelligence essay, introducing current ML approach, program beating humans in checkers

text | tabular | collab

```
from fastai.vision.all import *  # fastai library
path = untar_data(URLs.PETS)/'images'  # download dataset
def is_cat(x): return x[0].isupper()  # labelling function
dls = ImageDataLoaders.from_name_func(
  path, get_image_files(path), valid_pct=0.2, seed=42,
  label_func=is_cat, item_tfms=Resize(224))
  # load data, label, split into train/valid, transform
learn = cnn_learner(dls, resnet34, metrics=error_rate)
  # load architecture, pretrained model, define metric
learn.fine_tune(1)  # finetune ~ fit pretrained model
```

Different layers in NNs learn to recognize increasingly complex features.

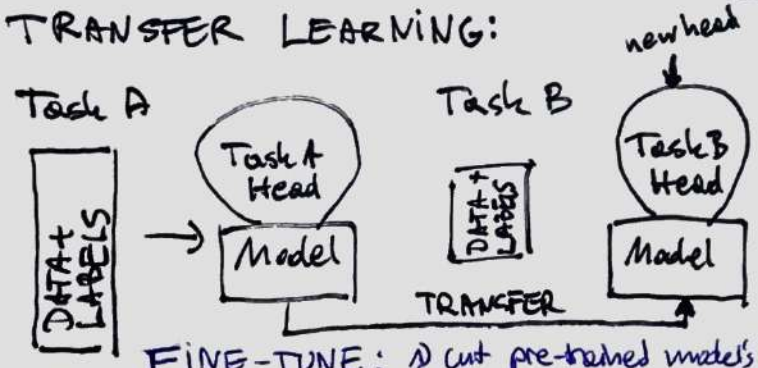
CNN - Convolutional Neural Network example:


L1: edges


L2: corners, shapes, colors, patterns

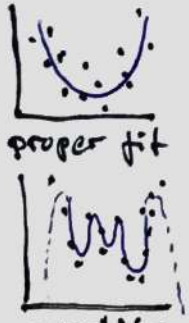

L3+: higher level components: faces, flowers, etc.

TRANSFER LEARNING:

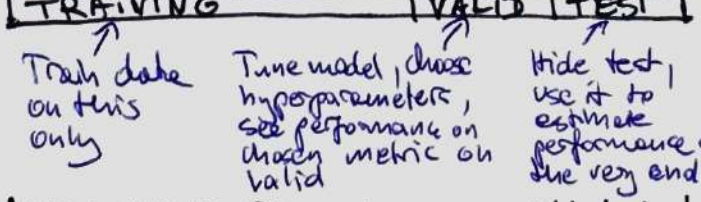


FINE-TUNE:
1) cut pre-trained model's head
2) Add new head specific to new task
3) Train new head only for 1 epoch
4) Fit entire model for more epochs (more tricks here)

OVERFITTING! Single most important and challenging issue! Model starts memorizing examples, rather than learning to generalize!



As a rule, split data into train/validation set, or maybe test set as well:



AVOID LEAKAGE: Time series, same subjects in train/valid etc...

LOSS
- Universal Approximation Theorem

METRIC
- CLASSIFICATION
- REGRESSION

EPOCH
PARAMETERS
ARCHITECTURE, CNN
SEGMENTATION
GPU, NOTEBOOK

OTHER TOPICS CONSTRAINED BY SPACE

fastbook ch 2 by @ dk21 | DL STATE 2020 | Computer Vision: ✓ Object recognition ✓ Object detection └ image captioning | Text: Classification, translation, summarization, Generation: compelling but not factually correct! question answering NER | Tabular: timeseries, typically ensemble w/ RF, GBM helps: high-cardinality cols, eg ZIP. | Recommendation sys.: Collaborative filtering customers as rows products as columns

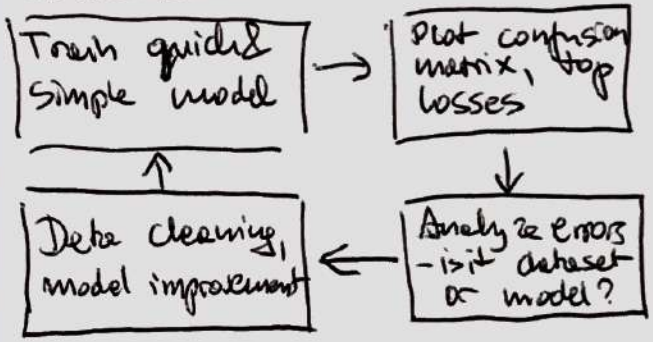
Underestimate DL capabilities

Overestimate DL capab.

KEEP AN OPEN MIND

1. Complete lots of small experiments and work on your own project
2. Consider data availability
3. Iterate E2E - all the way to final product
4. Start with sth that DL is good at.

WORKFLOW:

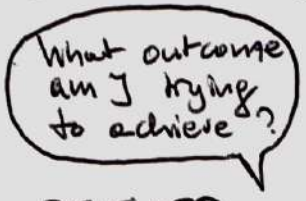


HOW TO AVOID DISASTER?

- Out of domain data: training vs production
- Domain shift: data changes over time

! Anticipate unforeseen consequences and feedback loops. What if this went really well?

DRIVETRAIN APPROACH: produce actionable outcomes, vs. predictions only



DEFINED OBJECTIVE | LEVERS | DATA | MODELS

! Randomized experiments needed to have good data, eg for recsys.

```
class DataLoaders(GetAttr): #Provides data for model
  def __init__(self, *loaders): self.loaders = loaders
  def __getitem__(self, i): return self.loaders[i]
  train, valid = add_props(lambda i, self: self[i])

bears = DataBlock( # template for creating data loaders (DB)
  blocks = (ImageBlock, CategoryBlock), # type of independent/dependent variable
  get_items = get_image_files, # function takes a path and returns images in that path
  splitter = RandomSplitter(valid_pct=0.3, seed=42), # split train/valid, fix seed
  get_y = parent_label # label images
  item_tfms = Resize(128)) # item tfms run on CPU, eg. image resizing
dls = bears.dataloaders(path) # provide source of data to (DB) - here path with images
```

RESIZE OPTIONS: crop, squish, pad, RandomResizedCrop ← recommended

DATA AUGMENTATION: create random variation in the input data, standard set provided in aug_transforms, can be done on GPU in batch:

```
bears = bears.new(item_tfms=RandomResizedCrop(128, min_scale=0.5), batch_tfms=aug_transforms())
learn.export(): Saves both model and parameters. load_learner(path/'export.pkl') = loads model
```

Voilà ipywidgets

INFERENCE, CPU vs GPU

map method L.map(function)

(space constrained)

GRADUAL ROLLOUT

1. Manual process
   - Run model in parallel
   - Humans check all predictions
2. Limited scope deployment
   - Careful human supervision
   - Time or geography limited
3. Gradual expansion
   - Good reporting systems needed
   - Consider what could go wrong!

TIP: Start writing, blog! Write for people one step behind you.

Lastpass ch3 notes by @dk21

ETHICS: the study of righ & wrong, how we define & recognize them, understand the connection between action & consequences

DATA ETHICS: complicated, context dependent → learn through examples, like a muscle → develop & practice it!

BUGS, RECOURSE, ACCOUNTABILITY

(Ex) Buggy algorithm cut healthcare benefits, impacting a vulnerable group

! Finger-pointing vs taking accountability bureaucracy as a way to evade responsibility

! Data often contains errors → mechanisms for audit and correction are crucial

(Ex) Police maintaining database of gang members with no mechanism to correct obvious errors

(Ex) US credit report system – very! difficult to correct errors

FEEDBACK LOOPS

(Ex) Conspiracy theories videos tend to get recommended more on YT | FB, People watching them tend to watch more online videos
YT | FB recommendation algorithms suggest more similar videos

spiral effect

WHY YOU – SHOULD CARE ?

EX. IBM products used in Nazi concentration camps – would you be ok to contribute to killing people?

EX. VW emmision scandal – engineers jailed!

- ML can create feedback loops & amplify bias
- People more likely to assume algorithms are objective and error-free
- Often used at scale, with no appeals process in place

Considering this will make you a better practitioner!

BIAS

Historical bias – people, processes, society are biased – taking real world data includes these biases

MEDICAL – doctor prescriptions differ for white vs black patients
SALES – different prices by race

(Ex) Searching google for a name that is historically black, you get ads for background checks (suggesting a criminal record)

! Systematic imbalance in the make-up of popular datasets, eg. ImageNet, word embeddings

(Ex) Translating doctor ~ man, nurse ~ woman.

Measurement bias – measuring wrong thing, in the wrong way, incorporating it into model incorrectly

(Ex) Factors predictive of stroke
- prior stroke
- cardiovascular disease
- accidental injury
- colonoscopy
- ...

these are correlated with people actually going to a doctor, being able to afford it, vs having a stroke

Aggregation bias – eg diabetes treatment based on linear, univariate models, small studies on homogeneous groups, when reality is non-linear, eg. diff. complications, symptoms across ethnicities

IDENTIFYING & ADDRESSING ETHICAL ISSUES

(1) Analyze a project you're working on
- Should we even be doing this?
- What bias is in the data?
- Can the code and data be audited?
- What are error-rates for different sub-groups?
- What is the accuracy of a simple, rule based alternative?
- What processes are in place to handle appeals or mistakes?
- How diverse is the team that built it?

(2) Processes to implement
EX. Regular ethical risk sweeps (≈ pen testing)
- Include perspectives of a variety of stakeholders
- What could bad actors do?
- Who will be directly and indirectly affected?
- apply ethical lenses! which option...

[RIGHTS] best reflects the rights of all stakeholders
[JUSTICE] treats people equally or proportionately?
[UTILITARIAN] will produce most good & least harm?
[COMMON GOOD] best serves community as a whole, not just some members
[VIRTUE] leads me to act as the sort of person I want to be

(3) The power of diversity
! Similar backgrounds ≈ similar blindspots!
⇒ innovation, more risks/solutions considered

(4) Role of policy – regulation is important!
(Ex) FB lack of action during Rohingya genocide, vs quick action to address GDPR
Advocacy is important – support the regulations that you a data scientist – believe we need!

Fastbook Ch.4 notes by @dk21 | FUNDAMENTAL TOOLS AND CONCEPTS FOR DEEP LEARNING

`new_list = [f(o) for o in a_list if o>0]`
list comprehension: sth to do for each element, filter

TENSOR shape - length of each axis
rank - number of axes = len(shape)

Measuring distance in space:
Mean absolute distance (L1 norm) → absolute differences
Root mean squared error (RMSE, L2 norm) → mean of square diffs, then root

numpy ARRAY: multidimensional table of data, with all items of the same type, any type. With array of arrays, arrays underneath may have different sizes → "jagged array". Operations on regular arrays written in optimized C - much faster than Python.

PyTorch TENSOR - like a numpy array, but has to use simple basic numeric type for all components. Can run on GPU.

BROADCASTING - critical for efficient code
Pytorch, when performing operation between tensors of different ranks, will automatically expand tensor with smaller rank to have the same size as the one with larger rank.

PyTorch **VIEW** change the shape of a tensor without changing contents, -1 parameter: make this axis as big as necessary to fit all data

@ - PyTorch matrix multiplication
batch @ weights + bias ⇒ fundamental operation in NN
$w * x + b$ (w: weights, b: biases) } parameters

Universal Approximation Theorem: this can represent any function

```
def train_model(model, epochs):
    for i in range(epochs):
        train_epoch(model)
        print(validate_epoch(model))
def train_epoch(model):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        opt.step()
        opt.zero_grad()
class BasicOptim:
    def __init__(self, params, lr):
        self.params, self.lr = list(params), lr
    def step(self, *args, **kwargs):
        for p in self.params:
            p.data -= p.grad.data * lr
#we use .data so PyTorch won't take gradient
#of this step
    def zero_grad(self, *args, **kwargs):
        for p in self.params: p.grad = None
def calc_grad(xb, yb, model):
    preds = model(xb)
    loss = mnist_loss(preds, yb)
    loss.backward()
def linear(xb): return xb@weights+bias
⇒ linear1 = nn.Linear(28*28, 1)
def mnist_loss(predictions, targets):
    predictions = predictions.sigmoid()
    return torch.where(targets==1, 1-predictions,
                       predictions).mean()
def init_params(size, std=1):
    return (torch.randn(size)*std).requires_grad_()
def simple_net(xb):
    res = xb@w1 + b1
    res = res.max(tensor(0.0))
    res = res@w2 + b2
    return res
```

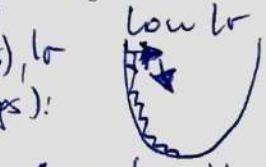
← this is a neural network, because we add a non-linearity (here: ReLU) between 2 classifiers
in practice, we want more layers

SGD Gradient descent process:
init → predict → loss → gradient → step → stop (repeat)
initialize to random values
function reflecting performance of the model
find direction to change weights and do it

Derivative of a function tells us how much a change in parameters will change result ($\frac{\text{rise}}{\text{run}}$)

GRADIENT: value of loss function's derivative at the point we're predicting. PyTorch can do it for us!

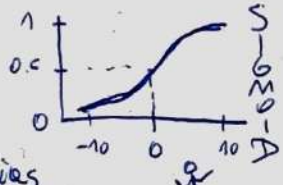
```
def f(x): return x**2
xt = tensor(3.).requires_grad_()
yt = f(xt)
yt.backward()
xt.grad #tensor(6.)
```



long time! explode vs converge!

STEPPING WITH A LEARNING RATE
Multiply the gradient by a small learning rate to decide how much to change parameters.

LOSS FUNCTION: represents how good is the performance of our model. Needs to react to small changes in weights (accuracy isn't good!)




```
def sigmoid(x): return 1/(1+torch.exp(-x))
```
→ ensure values between 0 and 1.

METRIC - drive human understanding
Loss - drive automated learning

To **step**: change the weights/biases - we need to calculate loss on 1 or more data items. 1 is not enough - not much info, not optimized. Whole data set would be too slow → **MINI-BATCH**
items = batch size (! important decision)
We need to vary examples during training - randomly shuffle dataset on every epoch.

DATA LOADER: takes Python collection, turns it into iterator over batches:
`dl = DataLoader(collection, batch_size=8, shuffle=True)`

DATASET: collection of tuples of independent and dependent variables. Simplest PyTorch dataset:
`dset = list(zip(X_train, y_train))`

Fastbook ch.5 notes by @jk21

DEEP DIVE INTO MECHANICS OF DL

Learn Regex! ⇒ Use eg in RegexLabeller

PRESIZING

```
item_tfms = Resize(460),
batch_tfms = aug_transforms(size=224, min_scale=0.75)
```

this adds RandomResizedCrop

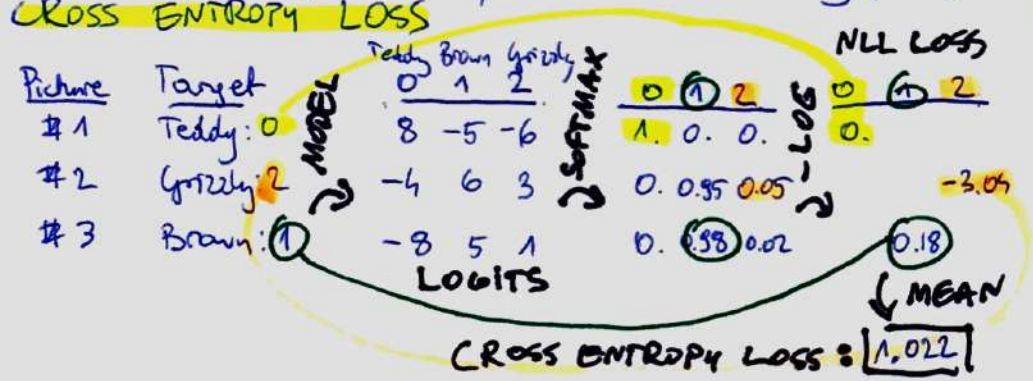
① Resize images to relatively large dimension (vs target training dimension)

② Compose all common aug operations (incl. resize to target size) into 1, and perform the combined operation on the GPU once at the end of processing
→ avoids data losses during augmentation
→ speeds up the process!

CHECKING AND DEBUGGING DATA BLOCK

show_batch → inspect data, DataBlock.summary(path)

CROSS ENTROPY LOSS



⇒ take the softmax, then negative log likelihood of that

Softmax: ensure final activations are between 0-1, and sum=1

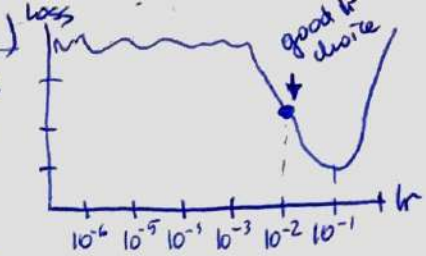
```
def softmax(x): return exp(x)/exp(x).sum(dim=1, keepdim=True)
```

- if one activation is slightly higher than others, exp will amplify this - Softmax really wants to pick one class - good if each image has definitive label
- may not be good at inference - will boost probs of example relative to other choices, independent of overall confidence - binary output columns, with sigmoid activation may be better?

Log likelihood: pick loss from column with correct label only, take -log of that to transform 0⇔1 scale to 0⇔+inf scale

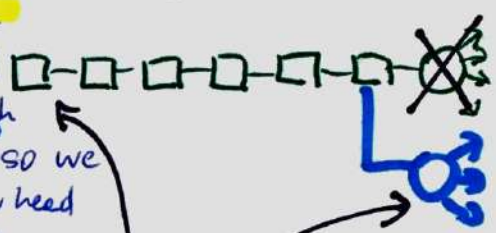
Pytorch: nn.CrossEntropyLoss ⇒ nn.LogSoftmax + nn.NLLLoss

Learning Rate Finder (Leslie Smith)



→ start with a very small learning rate
→ use that for 1 mini-batch, find the loss
→ increase the lr gradually, eg. double, per mini-batch, track the loss again
→ keep doing this until the loss gets worse
→ good choices: a) divide minimum loss lr by 10, or b) last point where the loss is clearly decreasing (steepest point)

Unfreezing and transfer learning



→ remove pretrained model's classification head
→ replace it with classif. head for new task
→ this will have random weights initially, so we freeze pretrained layers and only train new head
→ later, we unfreeze, check lr_finder again

Discriminative learning rates: pass slice(lr1, lr2) in fine_tune

→ train first layers with smaller lr, last layer with higher lr, range between - multiplicatively equidistant lr's.

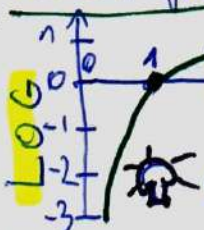
Selecting the number of epochs

1) Choose based on time available
2) Observe training/validation losses and val. metrics
3) Make decisions on metrics, not losses! - initially, validation loss will get worse, because it becomes overconfident, it's still ok if the metrics improve. Only later, model will start to memorize.

Early stopping - save model after each epoch, select the one with best metric
This is NOT GOOD with 1cycle training - epochs in the middle have higher lr, so unlikely to find best results. Better - if we overfit - retrain model from scratch with the number of epochs where we had best results.

Deeper Architectures: rule of thumb - more layers ⇒ more accurate model, but also risk of overfitting, longer to train, not always better!

We can speed this up with mixed precision training: learner.to_fp16()



log(a*b) = log(a) + log(b) ⇒ important property
- log increases linearly when underlying signal increases exponentially (multiplicatively)
- replace multiplication with addition ⇒ numerical stability

gradient of cross entropy(a,b) = softmax(a) - b, diff pred. vs target - linear, allows smoother training.

Fastbook ch.6 notes by @dke21

Multi/one-hot encoding: [0 1 0 0 1 0 1 0 0]

Multi-label classification: more than one type of object in an image - more common in practice to have some images with zero or more than 1 category match.

Use DataBlock API to construct DataLoaders object from Pandas df:
→ start by creating and testing Datasets
→ create DataLoaders after that's working

```
dblock = DataBlock(get_x = lambda r: r['fname'],
                   get_y = lambda r: r['labels'])
dsets = dblock.datasets(df)
dsets.train[0], len(dsets.train), len(dsets.valid)
```

! Lambda functions (defined inline) are good for iterating, but not compatible with serialization! Need verbose functions to export Learner after training.

```
def get_x(r): return path/'train'/r['fname']
def get_y(r): return r['labels'].split(' ')
def splitter(df):
    train = df.index[~df['is_valid']].tolist()
    valid = df.index[df['is_valid']].tolist()
    return train, valid
dblock = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                   splitter=splitter,
                   get_x=get_x,
                   get_y=get_y,
                   item_tfms=RandomResizedCrop(128, min_scale=0.35))
dls = dblock.dataloaders(df)
dls.show_batch(nrows=1, ncols=3)
```

BINARY CROSS ENTROPY (BCE)

```
def binary_cross_entropy(inputs, targets):
    inputs = inputs.sigmoid()
    return -torch.where(targets==1, inputs, 1-inputs).log().mean()
```

Picture	Target (Teddy, Brown, Grizzly)	Logits	Sigmoid	Loss
#1	1 0 0	8 -5 -6	1 0.01 0	0. 0.01 0.
#2	0 1 0	-4 6 3	0.02 1 0.95	0.02 0. 3.05
#3	0 1 1	-8 5 1	0 0.99 0.73	0 0.01 0.31

positive target: $-\log(\text{sigmoid})$
neg. target: $-\log(1-\text{sigmoid})$

(.mean()) BCE Loss = 0.3777

PyTorch: nn.BCELoss (without sigmoid) or nn.BCEWithLogitsLoss (with sigmoid)

```
loss_func = nn.BCEWithLogitsLoss()
loss = loss_func(acts, targets)
```

Accuracy - single label

```
def accuracy(inp, targ, axis=-1):
    pred = inp.argmax(dim=axis)
    return (pred==targ).float().mean()
```

Accuracy - multi label

```
def acc_multi(inp, targ, thres=0.5, sigmoid=True):
    if sigmoid: inp = inp.sigmoid()
    return ((inp>thresh)==targ.bool()).float().mean()
```

PARTIAL example (Python):
`acc_02 = partial(accuracy_multi, thresh=0.2)`

REGRESSION

Example - image regression, key point detection:

```
biwi = DataBlock(
    blocks=(ImageBlock, PointBlock),
    get_items=get_image_files,
    get_y=get_ctr,
    splitter=FuncSplitter(lambda o: o.parent.name=='13'),
    batch_tfms=[*aug_transforms(size=(240,320)),
    Normalize.from_stats(*imagenet_stats)]
```

! in fastai, this will also apply augmentation to point coordinates.

```
learn = cnn_learner(dls, resnet18, y_range=(-1,1))
dls.loss_func
→ MSELoss()
⇒ right function for regression!
```

! Pass y_range to learner to force outputs into range: `def sigmoid_range(x,lo,hi): return torch.sigmoid(x)*(hi-lo)+lo`

Flexible API + Transfer learning = POWER!

Rather than focusing on domains, focus on:

Independent Var	Dependent Var
Image	Text (caption)
text	Image (from capt.)
image + text + behavior	product - Purchase-prob.

+ Loss function!

Finding the best threshold:

```
xs = torch.linspace(0.05, 0.95, 29)
accs = [acc_multi(preds, targs, thresh=i, sigmoid=False) for i in xs]
plt.plot(xs, accs);
```

ok to choose hyperparam based on valid set, if the function looks smooth

Fastbook ch.7 notes by @dkl11 | Training a SOTA model

① If your dataset is big, experiment on a subset of it
→ iterate at faster speed
→ the more experiments you can do, the better
→ subset should be representative → generalize
(Ex) Imagenette : 10 classes from Imagenet

Normalization ⇒ mean = 0, stdev = 1

→ helps the model train
→ especially important when using pretrained models — distributed with stats used for normalization, use them for inference or transfer learning

check: `x,y = dls.one_batch()`
`x.mean(dim=[0,2,3]), x.std(dim=[0,2,3])`
(average over all axes except channel = 1)

fastai: add Normalize transform in batch_tfms

```
def get_dls(bs, size):
    dblock = DataBlock(blocks=(ImageBlock, CategoryBlock),
                       get_items=get_image_files,
                       get_y=parent_label,
                       item_tfms=Resize(460),
                       batch_tfms=[*aug_transforms(size=size,
                                                   min_scale=0.75),
                                   Normalize.from_stats(*imagenet_stats)])
    return dblock.dataloaders(path, bs=bs)
```

Progressive resizing

Gradually using larger & larger images as we train
similar to transfer learning : finetune after resizing
works also as data augmentation

```
dls = get_dls(128,128)
#create learner, fit_one_cycle
learn.dls = get_dls(64,224)
#learn.fine_tune(epochs,lr)
```

▽ May hurt performance in transfer learning if pretrained model/dataset are similar to our dataset.

Test Time Augmentation (TTA)

During inference or validation, creating multiple versions of each image using data augmentation, and then taking the average or maximum of the predictions.
(the default in fastai is center-cropping for validation = largest square centered)
→ problematic if relevant objects near edges
→ squish/stretch may be difficult to train
→ TTA solves these problems

`preds,targs = learn.tta()` # default is unaugmented center crop + 4 randomly augmented images, applied on valid dset.

Mixup

For each image img
1) select another dset image at random
2) pick a weight at random
3) take a weighted average of the img image and the selected image
4) take a weighted average of the img labels with the selected image's labels
(targets need to be one-hot encoded)

In fastai : callbacks are used to inject custom behavior in training loop

```
model = xresnet50()
learn = Learner(dls, model, loss_func=
    CrossEntropyLossFlat(), metrics=accuracy,
    cbs=Mixup) # callback.
```

→ more epochs are needed to train for good accuracy (eg. Imagenette LB - mixup in models ⩾ 80 epochs)
💡 can be used on activations inside models (NLP use cases)

check/read + research papers

Label Smoothing (LS)

Problem with OHE: overconfidence, labels are always 0 or 1 even if there is nuance or uncertainty. Leads to overfitting, probabilities at inference not meaningful.

LS: replace all 1s with a number bit less than 1
— 0s — " — more than 0
→ then train. Leads to:
- training more robust, even with noisy data
- models that generalize better

① Start with OHE
② Replace all 0s with $\frac{\text{epsilon}}{N}$ — usually 0.1
N — no. of classes
③ Replace 1 with $1 - \text{epsilon} + \frac{\text{epsilon}}{N}$

In practice we don't change, or one-hot encode labels, but apply this in the loss function.

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

→ more epochs are needed to train for good accuracy.

Summary

① Establish your environment for quick iteration/experimentation
- subset of dataset
- validation generalizes to test/prod.

② Start with a simple, strong baseline
③ Run many experiments:
- Augmentation
- Loss functions
- Inspect data/results for insights

Chapter 8 - Feedback

notes by Darek: @dk21

COLLABORATIVE FILTERING DEEP DIVE

= look at what products the current user has used or liked, find other users that have used or liked similar products, then recommend other products that those users have used or liked. Generalize: items vs products, eg. diagnoses, links etc.

LATENT FACTORS

common characteristics of items / user preferences

Example:

	action	scifi	old	
Movie is	0.9	0.98	-0.9	A
User likes	0.8	0.9	-0.6	B

Dot Product A.B = multiply vectors together, then sum up the result

We don't know latent factors → need to learn them!

Embedding

from scratch in PyTorch

multiplying by a one-hot encoded matrix, using the computational shortcut that it can be implemented by simply indexing directly.

	f1	f2	f3
U1	1	2	3
U2	3	1	2
U3	1	1	2
U4	2	2	3

← matrix F

$u = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ ← one-hot vector user3

$F[3,:] \Rightarrow F^T * u$

We index into the embedding matrix using an integer, but calculate the derivative as if we were multiplying the matrix with one-hot vector

Bootstrapping problem (= cold start)

→ pick some user to represent average taste

→ use a tabular model based on user metadata to construct initial embedding

! Representation bias & feedback loops are a risk - monitor, keep humans in the loop, gradual rollout...

```
dls = CollabDataLoaders.from_df(ratings, item_name='title', bs=64)
n_users = len(dls.classes['user'])
n_movies = len(dls.classes['title'])
n_factors = 50

def create_params(size):
    return nn.Parameter(torch.zeros(*size).normal_(0, 0.01))
# Parameter in PyTorch - wrapper, automatically calls requires_grad, and initializes

class DotProductBias(Module):
# in PyTorch we need to inherit from Module
    def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
        self.user_factors = create_params([n_users, n_factors])
        # self.user_factors = Embedding(n_users, n_factors)
        self.user_bias = create_params([n_users])
        # self.user_bias = Embedding(n_users, 1)
        self.movie_factors = create_params([n_movies, n_factors])
        self.movie_bias = create_params([n_movies])
        self.y_range = y_range
    def forward(self, x):
    # in PyTorch forward method is called whenever module is called, passing along any parameters included in the call
        users = self.user_factors[x[:,0]]
        movies = self.movie_factors[x[:,1]]
        res = (users * movies).sum(dim=1)
        res += self.user_bias[x[:,0]] + self.movie_bias[x[:,1]]
        return sigmoid_range(res, *self.y_range)
```

PCA - principal component analysis

pull out underlying directions in latent factor matrix

```
movie_w = learn.model.movie_factors[idxs].cpu().detach()
movie_pca = movie_w.pca(3)
fac0, fac1, fac2 = movie_pca.t()
x = fac0[movie_idxs]
y = fac2[movie_idxs]
```

→ plt scatter plot to visualize.

OOP in Python, ** leverages @delegates decorator

NN SPACE

Weight decay / L2 regularization

adding to our loss function the sum of all the weights squared, to encourage weights to stay small to prevent overfitting

wd / weight decay - parameter, how much we add loss:

```
loss_with_wd = loss + wd*(parameters**2).sum()
parameters.grad += wd*2*parameters - same but faster!
```

In fastai, we pass it in a call to fit:

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

Embedding distance

Movie similarity can be defined by the similarity of users that like those movies, distance between embedding vectors can define that similarity

```
movie_factors = learn.model.i_weight.weight
idx = dls.classes['title'].o2i['Movie title 1']
distances = nn.CosineSimilarity(dim=1)(movie_factors, movie_factors[idx][None])
idx = distances.argsort(descending=True)[1]
dls.classes['title'][idx]
```

Collab NN

- dot product was PMF (probabilistic matrix factorization) approach, there is an option to do it with deep learning!

```
class CollabNN(Module):
    def __init__(...)
    ...
    self.layers = nn.Sequential(
        nn.Linear(user_sz[1]+item_sz[1], n_act),
        nn.ReLU(),
        nn.Linear(n_act, 1))
    ...
    def forward(self, x):
        embs = self.user_factors(x[:,0]), self.item_factors(x[:,1])
        x = self.layers(torch.cat(embs, dim=1))
        return sigmoid_range(x, *self.y_range)
```

Fastbook Ch 9 notes by @tk21 | TABULAR MODELING ⇒ Data as a table; predict value in 1 column based on other cols

Variables

- continuous: numerical data, feed directly to model
- categorical: discrete levels (eg movie IDs), need to convert to numbers first
  - ordinal - categories with natural ordering

```
df['ord_cat'] = df.ord_cat.astype('category').cat.set_categories(order, ordered=True, inplace=True)
```

CAT: represent via one-hot-encoding or entity embedding:
→ reduces memory usage and speeds up NN vs OHE
→ reveals intrinsic properties of variables - similar values close to each other in embedding space

Decision Trees

TIP: avoid OHE categories for DTs | RFs

1) Loop through each column in dataset (greedy approach)
2) For each col, loop through each possible level of that col.
3) Try splitting data into 2 groups at that level
4) For regression: find avg value of dependent var. for each of 2 groups, see how close it is to the actual value of dep. variable for each of the items in that group
5) After looping thru all cols | levels, pick split point with the best predictions
6) For each group based on this split, repeat the process

Random Forests (Breiman 1994, 2001)

1) Randomly choose a subset of rows and subset of columns
2) Train a model using this subset (decision tree)
3) Save that model, return to step 1 several times
4) To make a prediction, predict using all saved models then take average of those predictions ⇐ BAGGING

Important - errors of individual models are not correlated, so the average of those errors is ZERO

Out of Bag Error: measure prediction error on the training set by only including in the calculation of a row's error the trees where that row was not included in training

BOOSTING

another approach to ensembling (vs BAGGING)

1) Train a small model that underfits your dataset
2) Calculate predictions in the training set for this model
3) Subtract predictions from targets = RESIDUALS
4) Go back to step 1, but use the residuals as targets
5) Continue until a stopping criterion: max no trees, valid error getting worse, etc.

GBMs, GBDTs, XGBoost → Risk of overfitting
→ very sensitive to hyperparameter choices

TABULAR MODELING WORKFLOW

1) Start with RF - easiest to train, resistant to hyperparam. choices, little preprocessing
2) Use RF model for feature selection, PDP analysis
3) Then, try NNs or GBMs, try adding embeddings of cat. variables to the data

MODEL INTERPRETATION

1) How confident are we in our predictions for a particular row?
2) For a particular row, what were the most important factors influencing prediction?
3) Which columns are the strongest predictors, which can we ignore?
4) Which columns are effectively redundant with each other?
5) How do predictions vary as we vary each column?

TREE VARIANCE (1)

Check std deviation of predictions across trees:

```
preds = np.stack([t.predict(valid_xs) for t in m.estimators_])
preds_std = preds.std(0)
```

high std ⇒ low confidence

REMOVING LOW IMPORTANCE VARIABLES (3)

Generally the 1st step to improving the model is simplifying it, so that it's easier to study, roll out, maintain, ↓ ⇒

Remove columns | variables of low importance
→ Retrain the model → check impact on accuracy

REMOVING REDUNDANT FEATURES (4)

cluster_columns(xs) - shows similarity of colums
We determine similarity by calculating the rank correlation - all values are replaced by their rank, then correlation is calculated
→ try removing each of potentially redundant features, one at a time then multiple from overlapping groups, observe OOB score | accuracy

TREE INTERPRETER + WATERFALL CHARTS (2)

What were the most impt factors for predicting with a particular row of data how did they influence that prediction? Calculated similarly to feature importances.
→ display feature contributions with waterfall chart
→ use it to provide useful information to users of your data product - reasons behind predictions

Extrapolation

- Decision Trees | RF can never predict values outside of the range of training data (eg. trend). NN can help. Also finding

Out-of-domain data: 1) combine train & valid sets
2) Use RF to predict if a row is in train or valid set
3) Get feature importances - for the columns that differ significantly between train/valid, try removing them and see how it impacts accuracy. It can improve it, and make the model more resilient.
→ Try to avoid using old data - it may no longer be predictive.

FEATURE IMPORTANCE (3)

m.feature_importances_
Loop through each tree, then recursively explore each branch - check what feature was used for that split, and how much the model improves as the result. The improvement weighted by number of rows in that group, is added to importance score for that feature. Sum across all branches of all trees, normalize scores so that they add to 1.

PARTIAL DEPENDENCE (5) PLOTS = PDP

PDP visualize how variables affect our predictions: if a row varied on nothing other than the feature in question, how would it impact the dependent variable?

- replace every value in YearMade col. with 1950
- calculate predicted sale price for every auction, take average over all auctions
- repeat for 1951, 1952 ..., 2020
- this isolates the effect of YearMade

from sklearn.inspection import plot_partial_dependence

The Book of Why

DATA LEAKAGE (6)

= giving model information about the target, which normally should not be available at the time of prediction. How to detect it?

- check if the accuracy is too good to be true
- look for impt predictors that don't make sense
- look for PDP plots that don't make sense in practice

Using a Neural Network

1) Decide which cols should be treated as cat vs cont
2) Create embeddings for categorical variables
3) Add normalization (in procs)
4) Consider adding y_range to regression models
5) Adjust hidden layer sizes to size of dataset

fastai: TabularPandas (pandas df + conveniences)
TabularProc
- Categorify
- FillMissing

tabular_learner
TabularModel → study source code

Other: dtreeviz
add_datepart

Fastbook ch 10 | NLP DEEP
notes by @dair1 | DIVE: RNNs

Wikitext 103 → Language model → Imdb → Language model → Imdb → Classifier

ULMFit approach: fine-tune pretrained LM on the target corpus prior to transfer learning to classification task.

Language Model: model trained to guess the next word in a text, after reading the words before

Self-supervised learning: training a model using labels that are embedded in the independent variable, rather than requiring external labels. Usually used for pretraining in transfer learning.

TEXT PREPROCESSING IN 3 STEPS

① TOKENIZATION

= converting a text into a list of tokens

a) **word-based**: split sentence on spaces and language-specific rules

b) **subword-based**: analyze a corpus of documents to find the most commonly occurring groups of letters. These substrings become the vocab.

c) **character-based**

fastai adds some functionality with the Tokenizer class, e.g. special tokens:

- xxbos : beginning of text
- xxmaj : next word begins with a capital
- xxunk : next word is unknown

☆: see rules: defaults.text_proc_rules

setup creates the vocab that is used in tokenization

```
spacy = WordTokenizer()
toks = first(spacy([txt]))
print(coll_repr(toks, 30))
tkn = Tokenizer(spacy)
print(coll_repr(tkn(txt), 31))
sp = SubwordTokenizer(vocab_sz=52)
sp.setup(txts)
```

[txt] - collection of text documents

displays first n elems of collection

adds special tokens etc

② NUMERICALIZATION

= mapping tokens to integers

1) make a list of all possible levels of a variable (the vocab)
2) replace each level with its index in the vocab

```
num = Numericalize()
num.setup(toks)
coll_repr(num.vocab, 20)
```

class methods

③ PUTTING TEXT INTO BATCHES for a language model

- we want LM to read text in order
- we use a model that maintains a state - remembers what it read previously when predicting what comes next

1) transform indiv. texts into a stream by concatenating them, shuffle docs order before each epoch
2) cut this stream into a certain number of batches (batch size) mini-streams preserve order of tokens
3) Each time step read seq_len from mini-streams

→ dependent variable is offset from the independent variable by 1 token // LM Data Loaders

⚠ Potential to generate disinformation campaigns, flood social media with fake content etc → see ch. 3 on ethics

Collating items in a batch

- use padding to make texts all the same size
- sort (ish) docs by length prior to each epoch - batch together docs with similar lengths, pad to length of longest doc in a batch

```
get_imdb = partial(get_text_files, folders=['train', 'test', 'unsup'])
dls_lm = DataBlock(                          # when fastai handles tokenization and numericalization
    blocks=TextBlock.from_folder(path, is_lm=True),
    get_items=get_imdb, splitter=RandomSplitter(0.1)
).dataloaders(path, path=path, bs=128, seq_len=80)
dls_lm.show_batch(max_n=2)
```

Words that are not in the vocab of pretrained lm will be added with random embeddings

```
# fine-tuning language model
learn = language_model_learner(dls_lm, AWD_LSTM,
    drop_mult=0.3, metrics=[accuracy, Perplexity()]).to_fp16()
```

loss function: cross entropy
perplexity metric: exponential of ← torch.exp(cross_entropy)

```
learn.fit_one_cycle(1, 2e-2)        # automatically frozen, this will only train embeddings
learn.save('1epoch')                # use fit_one_cycle to save/load intermediate model results
learn = learn.load('1epoch')
learn.unfreeze()
learn.fit_one_cycle(10, 2e-3)
learn.save_encoder('finetuned')     # ENCODER: model without task-specific final layers, like body in CNNs
TEXT = "I liked this movie because"
N_WORDS = 40
# TEXT GENERATION
pred = learn.predict(TEXT, N_WORDS, temperature=0.75)
```

```
# creating the classifier dataloaders        ! is_lm=False
dls_class = DataBlock(                       # pass vocab to use fine-tuned embeddings
    blocks=(TextBlock.from_folder(path, vocab=dls_lm.vocab), CategoryBlock),
    get_y=parent_label,
    get_items=partial(get_text_files, folders=['train', 'test']),
    splitter=GrandparentSplitter(valid_name='test')
).dataloaders(path, path=path, bs=128, seq_len=72)
learn = text_classifier_learner(dls_class, AWD_LSTM, drop_mult=0.5,
    metrics=accuracy).to_fp16()
learn = learn.load_encoder('finetuned')      # Gradual unfreezing + discriminative learning rates
learn.fit_one_cycle(1, 2e-2)
learn.freeze_to(-2)
learn.fit_one_cycle(1, slice(1e-2/(2.6**4), 1e-2))
... (freeze_to(-3) ...
learn.unfreeze()
learn.fit_one_cycle(2, slice(1e-3/(2.6**4), 1e-3))
```

Fastbook ch. 11 notes by @daj21

Data Munging with fastai's Mid-Level API

fastai is built on a layered API:

Top layer = applications - train a model in 5 lines of code, eg:
TextDataLoaders.from_folder()

Mid level API: - create new DataLoaders
- apply just part of transforms
- has the callback system, which allows to customize training loop any way we like
- has general optimizers

Transform: an object that behaves like a function, has optional setup method to initialize hidden state (eg vocab) and an optional decode that reverses the function. Special behavior - always gets applied over tuples (input, target)

Examples: Tokenizer, Numericalize

1) Create object
2) call setup method
3) apply to input by calling object as a function
4) decode result back to understandable representation

(usage, fastai encapsulates these steps in the Transform class)

Write your own Transform

1) Write a function /+ decorator
```
def f(x:int): return x+1
```
← function only gets applied to ints
```
tfm = Transform(f)
```
2) **Decorator** - Python syntax for passing a function to another function (or callable obj that behaves like a function)
```
@Transform
def f(x:int): return x+1
```
3) If we need setup or decode, then need to subclass Transform and implement encode

FROM: → → → →
```
path = untar_data(URLs.IMDB)
dls = DataBlock(
  blocks=(TextBlock.from_folder(path),
          CategoryBlock),
  get_y = parent_label,
  get_items = partial(get_text_files,
    folders=['train','test']),
  splitter = GrandParentSplitter(
    valid_name='test')
).dataloaders(path)
```

```
class NormalizeMean(Transform)
  def setups(self, items): self.mean = sum(items)/len(items)
  def encodes(self, x): return x - self.mean
  def decodes(self, x): return x + self.mean
tfm = NormalizeMean()
tfm.setup([1,2,3,4,5])
```
← note we call setup but implement setups

Pipeline: compose several Transforms together
```
tfms = Pipeline([tok, num])
```
define it by passing a list of transforms
```
t = tfms(txts)
tfms.decode(t)
```
decode result of encoding

TfmdLists and Datasets: Transformed Collections
```
cut = int(len(files)*0.8)
splits = [list(range(cut)), list(range(cut,len(files)))]
tls = TfmdLists(files, [Tokenizer.from_folder(path), Numericalize], splits=splits)
t = tls[0]
tls.decode(t)
tls.show(t)
```
TfmdLists = class that groups data as a set of raw items (fnames, df rows...) and Pipeline of Transforms. At initialization, call setup in each Transform in order. We index into it to get results of pipeline on raw elements. Can handle train/valid: `tls.valid[0]`. Convert it to dataloaders w/ dataloaders method

TO: (equivalent, but can be customized):
```
tfms = [[Tokenizer.from_folder(path), Numericalize],
        [parent_label, Categorize]]
files = get_text_files(path, folders=['train','test'])
splits = GrandparentSplitter(valid_name='test')(files)
dsets = Datasets(files, tfms, splits=splits)
dls = dsets.dataloaders(dl_type=SortedDL,
        before_batch=pad_input)
```

Datasets: apply 2 (or more) Pipelines in parallel to the same raw object and build a tuple with the result. → automatically do setup for us → when indexed, return a tuple with result of each pipeline!
```
x_tfms = [Tokenizer.from_folder(path), Numericalize]
y_tfms = [parent_label, Categorize()]
dsets = Datasets(files, [x_tfms, y_tfms], splits=splits)
x,y = dsets.valid[0]
```
→ convert it to dataloaders (here: need to pad input):
```
dls = dsets.dataloaders(bs=64, before_batch=pad_input)
```

DataLoader: collates the items from our datasets into batches. Important ways to customize:

1) after_item: applied on each item after grabbing it inside the dataset (≈ item_tfms in DataBlock)
2) before_batch: applied on list of items before they are collated. Ideal place to pad items to the same size.
3) after_batch: applied on the batch as a whole after its construction (≈ batch_tfms)

Application Example: Siamese Pair
Siamese model takes two images and has to determine if they are same class or not

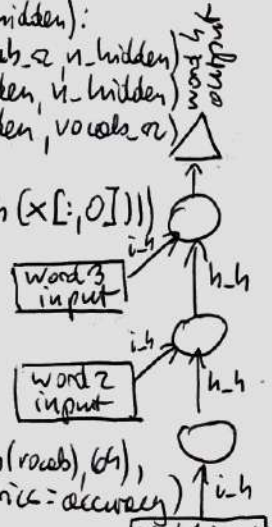
Fastbook ch 12 notes by @dk21 | # A language model from scratch!

TIP: When starting to work on a new problem, find the simplest dataset to try out methods quickly & easily, interpret results. Eg.: Human Numbers

#First language model:

```
class LMModel1(Module):
  def __init__(self, vocab_sz, n_hidden):
    self.i_h = nn.Embedding(vocab_sz, n_hidden)
    self.h_h = nn.Linear(n_hidden, n_hidden)
    self.h_o = nn.Linear(n_hidden, vocab_sz)
  def forward(self, x):
    h = F.relu(self.h_h(self.i_h(x[:,0])))
    h = h + self.i_h(x[:,1])
    h = F.relu(self.h_h(h))
    h = h + self.i_h(x[:,2])
    h = F.relu(self.h_h(h))
    return self.h_o(h)

learn = Learner(dls, LMModel1(len(vocab), 64),
    loss_func=F.cross_entropy, metrics=accuracy)
learn.fit_one_cycle(4, 1e-3)
```



compare vs naive baseline: always predict the most common token

RNN

① Refactor with a for-loop ⇒ Recurrent NN
= we can now apply to token sequences of different length.

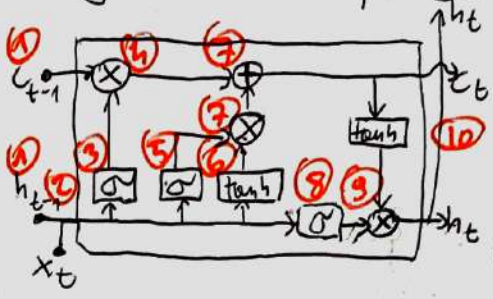
Hidden state = activations that are updated at each step of a recurrent NN

② Maintaining the State of RNN
- don't initialize hidden state to zero for each sample (move initialization of hs to __init__)
- BUT: that makes our NN as deep as the number of tokens (consider unrolled representation)
- calculating derivatives will be slow / memory intensive
- SOLUTION: tell Pytorch we don't want to calculate all gradients, just n steps. Use detach method to remove gradient history.

Backpropagation Through Time
Treating a NN with effectively one layer per time step, refactored using a loop, as one big model, and calculating gradients on it in the usual way. To avoid running out of memory and time, we use truncated BPTT, which detaches the history of computation steps in the hidden state every few time steps.

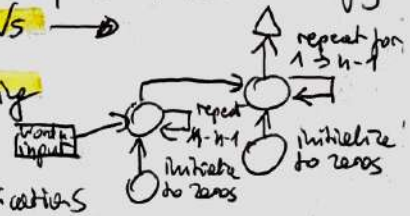
! Make sure the model sees text samples in correct order!

```
class LSTMCell(Module):
  def __init__(self, ni, nh):
    self.forget_gate = nn.Linear(ni+nh, nh)
    self.input_gate = nn.Linear(ni+nh, nh)
    self.cell_gate = nn.Linear(ni+nh, nh)
    self.output_gate = nn.Linear(ni+nh, nh)
  def forward(self, input, state):
①   h,c = state
②   h = torch.cat([h, input], dim=1)
③   forget = torch.sigmoid(self.forget_gate(h))
④   c = c * forget
⑤   inp = torch.sigmoid(self.input_gate(h))
⑥   cell = torch.tanh(self.cell_gate(h))
⑦   c = c + inp * cell
⑧   out = torch.sigmoid(self.output_gate(h))
⑨   h = out * torch.tanh(c)
⑩   return h, (h,c)
```



③ Creating more signal
= predict the next word after every input word
dependent variable = input shifted by 1
model will return outputs of shape bs*seq_len*vocab_sz.
targets are in shape bs*seq_len
flatten with tensor.view, use F.cross_entropy.

④ Multi layer RNNs ⑤ Exploding/vanishing activations



- many layers ⇒
- many matrix multiplications
- extremely large or small numbers
- floating point numbers are an approximation
- gradients end up as zero or infinity
- model stops training

SOL's = batch norm | residual connections | careful init, for RNN: GRU | LSTM layers!

LSTM adds another hidden state (cell state) to remember what happened earlier in a sequence (long - short term memory)

#Final language model

```
class LMModel2(Module):
  def __init__(self, vocab_sz, n_hidden, n_layers, p):
    self.i_h = nn.Embedding(vocab_sz, n_hidden)
    self.rnn = nn.LSTM(n_hidden, n_hidden, n_layers,
        batch_first=True)
    self.drop = nn.Dropout(p)
    self.h_o = nn.Linear(n_hidden, vocab_sz)
    self.h_o.weight = self.i_h.weight
    self.h = [torch.zeros(n_layers, bs, n_hidden)
        for _ in range(2)]
  def forward(self, x):
    raw, h = self.rnn(self.i_h(x), self.h)
    out = self.drop(raw)
    self.h = [h_.detach() for h_ in h]
    return self.h_o(out), raw, out
  def reset(self):
    for h in self.h: h.zero_()

learn = Learner(dls, LMModel2(len(vocab), 64, 2, 0.5),
    loss_func=CrossEntropyLossFlat(), metrics=accuracy,
    cbs=[ModelResetter, RNNRegularizer(alpha=2, beta=1)])
learn.fit_one_cycle(15, 1e-2, wd=0.1)
```

⑥ Regularizing an LSTM

DROPOUT: Randomly change some activations to zero at training time, so that all neurons actively work towards the output. This changes scale of activations, so we need to rescale them (divide by 1-p)

```
class Dropout(Module):
  def __init__(self,p): self.p=p # set by model.train/model.eval
  def forward(self,x):
    if not self.training: return x
    mask = x.new(*x.shape).bernoulli_(1-p) # tensor of random zeros and ones
    return x * mask.div_(1-p)
```

Activation Regularization | Temporal Act Reg (AR | TAR)
Like weight decay, we want to make final activations from LSTM as small as possible.
loss += alpha * activations.pow(2).mean()
Temporal AR: keep diffs between consecutive token activations small
loss += beta * (acts[:,1:] - acts[:,:-1]).pow(2).mean()
this + other tricks (weight tying) in AWD-LSTM

Fastbook Ch 13 notes by @dk21

CONVOLUTIONAL NEURAL NETWORKS IMPROVING TRAINING STABILITY (CNN)

Feature Engineering: creating new transformations of the input data in order to make it easier to model.
eg: add_datepart, images: edge detectors

Convolution: applying a kernel across an image (multiplication + addition)
→ activation map

```
top_edge = tensor([[-1,-1,-1],
                   [ 0, 0, 0],
                   [ 1, 1, 1]]).float()

def apply_kernel(row, col, kernel):
    return (img[row-1:row+2, col-1:col+2] * kernel).sum()
```

Pytorch ⇒ F.conv2d
Input tensor shape:
(minibatch, n_channels, n_rows (img height), n_cols (img width))
Weight tensor = kernels / filters of shape:
(out_channels, in_channels, n_rows (kernel height), n_cols (kernel width))

Padding – we use it to ensure that the output activation map is the same size as original image

Stride – we use it to decrease the size of our outputs, how much we move kernel at each step

CNN ⇒ we learn the values of the kernels (eg SGD)
→ we use convolutions in the network architecture

Channels / Features – size of 2nd axis of weight matrix, ie number of activations per grid cell after a convolution. Channels can also refer to input data (R color)

Receptive Field – area of an image that is involved in the calculation of a layer
– the more stride 2 convolutions we have before a layer, the larger the receptive field for an activation in that layer, also more semantically rich features, so we need more weights / parameters (channels)

Color Images: 3 channels: R / G / B: in one sliding window, on each channel we multiply elements of window by elements of corresponding filter, then sum the results and sum over all filters

kernel tensor shape = ch-in × ks × ks

```
def conv(ni, nf, ks=3, act=True):
    layers = [nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)]
    layers.append(nn.BatchNorm2d(nf))
    if act: layers.append(nn.ReLU())
    return nn.Sequential(*layers)

simple_cnn = sequential(
    conv(1,4),      # 14x14
    conv(4,8),      # 7x7
    conv(8,16),     # 4x4
    conv(16,32),    # 2x2
    conv(32,2, act=False),  # 1x1
    Flatten())
```

stride=2 decreases no. of activations, so we increase no of features at the same time

Flatten: same as tensor.squeeze, but as a module

nn.Conv2d – module equivalent of F.conv2d, more convenient when creating an architecture because it creates the weight matrix automatically when we instantiate it.

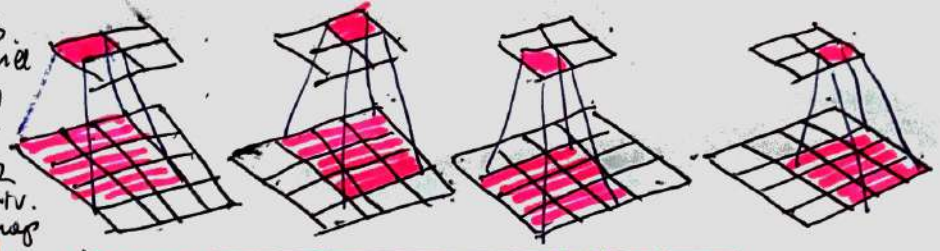
```
learn = Learner(dls, simple_cnn, loss_func=F.cross_entropy, metrics=accuracy)
learn.summary()
```

Stefano Giacomo: learn.activation_stats.color_dim(-2)
→ another way to see training stability
→ observe for change in non-zero activations


increasing collapse = bad


smooth increase = good

PYTHON
Nested list comprehension [[i,j] for j in range(1,5) for i in range(1,5)]
use unsqueeze to insert a unit axis into a tensor at specific position



Improving Training Stability

Let's use ActivationStats callback to look inside our models when they're training:

```
from fastai.callback.hook import *
... fit ... cbs=ActivationStats(with_hist=True)...
learn.activation_stats.plot_layer_stats(0)
```

→ model should have consistent / smooth mean and stdev of layer activations during training
→ activations near zero are especially problematic – computation is doing nothing, zeros propagate to further layers

(1) **Increase Batch Size**: – gradients are more accurate (from more data)
– BUT: fewer batches per epoch, so fewer chances to update weights

(2) **1 cycle training**: initial weights are not suited well to our task, so we don't want a high LR – training would diverge quickly. We also don't want high LR at the end, so we don't skip over the minimum ⇒ we separate learning rate schedule in 2 phases:
1) LR grows from min to max value (warm up)
2) LR decreases back to min value (annealing)
3) the momentum (impact of previous steps / direction in SGD) varies in the opposite direction of the LR

Super convergence (Leslie Smith) by training with higher LR's, we train faster.
we also overfit less, because we skip over the sharp local minima to end up in smoother part of the loss. ↳ more generalizable

(3) **Batch Normalization** nn.BatchNorm2d (nf)
Take average of the mean and std devs of the activations of a layer and using those to normalize the activations
potential problem: some acts need to be high to predict well
so – we add 2 learnable parameters, gamma and beta!
y – activation vector after normalization
BN returns gamma * y + beta
During training – use mean and stdev of the batch to normalize data. During validation – use running mean of statistics calculated during training → training is more smooth, also works as regularizer, model generalizes better

Fastbook ch 14 notes by @dkl21

ResNets

Before 2013 (incl. VGG): Sequence of conv layers, then flatten activation matrix into a vector

FCN - Fully Convolutional Network:
sequence of conv layers, some of them stride 2, followed by an adaptive average pooling layer, a flatten layer to remove the unit axes, and finally a linear layer.

```
def avg_pool(x): return x.mean((2,3))   # mean over x and y axes
```

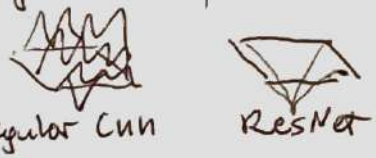
Pytorch: `nn.AdaptiveAvgPool2d`
→ converts a grid of activations into a single activation per image

```
def block(ni,nf): return ConvLayer(ni,nf,stride=2)
def get_model():
    return nn.Sequential(
        block(3,16),
        block(16,32),
        block(32,64),
        block(64,128),
        block(128,256),
        nn.AdaptiveAvgPool2d(1),
        Flatten(),
        nn.Linear(256, dls.c))
```

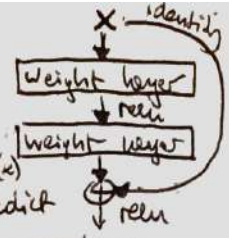
? Is this good for OCR, eg MNIST? No!
We slice image into small pieces, jumble them up, and decide whether on average each piece looks like our target class - ..
FCN - good for objects that don't have a single correct orientation or size, eg. photos
2015: Kaiming He et al. Deep Residual Learning...
Insight: Deeper NNs led to worse performance vs shallower nets, both them and test.

SCIENTIFIC DISCOVERY
1) start with experimental observation
2) watch out for unexpected results
3) try to figure it out, with tenacity

LOSS LANDSCAPE



Skip connection: Skip one every second convolution, so:
`x + conv2(conv1(x))`



Resnet block returns `y = x + block(x)`
We are asking the blocks to predict the difference between x & y = residual
Later trick: use zero for the initial value of gamma in the last batch norm layer of each block - we begin training with a true identity path through the ResNet blocks

How to handle different dims (grid size, channels)?

1x1 Convolution: Conv with a kernel size of 1, only doing a dot product over the channels of each input pixel, not combining pixels

```
def _conv_block(ni,nf,stride=stride):
    return nn.Sequential(
        ConvLayer(ni,nf,stride=stride),
        ConvLayer(ni,nf,act_cls=None,norm_type=
                  NormType.BatchZero))
```

```
class ResBlock(Module):
    def __init__(self,ni,nf,stride=1):
        self.convs = _conv_block(ni,nf,stride)
        self.idconv = noop if ni==nf else
                      ConvLayer(ni,nf,1,act_cls=None)
        self.pool = noop if stride==1 else
                    nn.AvgPool2d(2,ceil_mode=True)
    def forward(self,x):
        return F.relu(self.convs(x)+self.idconv(self.pool(x)))
```

This removes ReLu

`# noop - no operation, return unchanged input`

First layers: majority of the computation, few parameters. Eg 128 px image; stride 1: apply kernel to each of 128×128 pixels
3 × input fts × 32 output fts × 3×3 kernel = 864 params
Last layers: majority of parameters, few computations
256 inp fts × 512 output fts × 3×3 kernel = 1,179,648 params
apply kernel to grid size 4×4 or 2×2

→ Keep first layers as simple as possible (STEM)

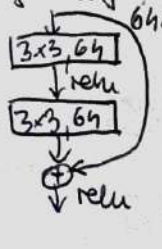
```
def _resnet_stem(*sizes):
    return [
        ConvLayer(sizes[i],sizes[i+1],3,stride=2 if i==0 else 1)
        for i in range(len(sizes)-1)
    ]+[nn.MaxPool2d(kernel_size=3,stride=2,padding=1)]
```

```
class ResNet(nn.Sequential):
    def __init__(self,n_out,layers,expansion=1):
        stem = _resnet_stem(3,32,32,64)
        self.block_szs = [64,64,128,256,512]
        for i in range(1,5): self.block_szs[i] *= expansion
        blocks = [self._make_layer(*o) for o in enumerate(layers)]
        super().__init__(*stem,*blocks,
            nn.AdaptiveAvgPool2d(1), Flatten(),
            nn.Linear(self.block_szs[-1],n_out))
    def _make_layer(self,idx,n_layers):
        stride = 1 if idx==0 else 2
        ch_in,ch_out = self.block_szs[idx:idx+2]
        return nn.Sequential(*[
            ResBlock(ch_in if i==0 else ch_out, ch_out,
                     stride if i==0 else 1)
            for i in range(n_layers)])
```

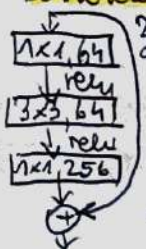
```
def _conv_block(ni,nf,stride):
    return nn.Sequential(
        ConvLayer(ni,nf//4,1),
        ConvLayer(nf//4,nf//4,stride=stride),
        ConvLayer(nf//4,nf,1,act_cls=None,norm_type=NormType.BatchZero))
```

```
# Resnet 18:
m = ResNet(dls.c,[2,2,2,2]
# ResNet 50: pass 4 as expansion parameter, start with 4x fewer channels, end with 4x more
m = ResNet(dls.c,[3,4,6,3],4)
dls = get_data(URLs.IMAGENETTE_320,presize=320,resize=224)
learn = get_learner(m)
learn.fit_one_cycle(20,3e-3) # + mixup, 100 epochs, ...
```

Regular layers:



Bottleneck Layer: 1×1 convs are much faster! This block with 256 dim executes faster, so we can add more filters in the same amount of time. 1×1 conv first diminishes, then restores the no. of channels.



Fastbook ch 15 notes by @dke21 | APPLICATION ARCHITECTURES DEEP DIVE

cnn_learner

- pass an architecture to use as the body of the network (usually ResNet)
- download and load pretrained weights
- for transfer learning, cut the head
ResNet: everything from adaptive Avg Pooling layer
→ use dictionary to determine head vs body:
model_meta[resnet50] {'cut': -2, 'split': <function ...>, 'stats': ([---],[---])}
- create new head: create_head(20,2)
Sequential(...)
AdaptiveConcatPool2d(
(ap): AdaptiveAvgPool2d(output_size=1)
(mp): AdaptiveMaxPool2d(output_size=1)
→ used by default in fastai cnn head
→ fastai adds 2 linear layers by default in cnn head (helps transfer learning across domains)
→ bn_final parameter: use batchnorm as the final layer to help the model scale output acts.

unet_learner

- used for segmentation, super-resolution, colorization, style-transfer
- output is an image | pixel grid | predicted label for each pixel (generative vision models)
- same approach as cnn - cut off head, replace it with a new one for the generation task
- How to increase grid size?

a) nearest neighbor interpolation (Pytorch layer) # so-so
b) transposed convolution | stride half convolution (insert zero padding between all the pixels in the input)
c) add skip connections - from the activations in the body of the ResNet (U-net)
- fastai Dynamic Unet - autogenerates architecture with the right size for our image size

Practical DL workflow:

1) Start with a small model and data sample, get to the point where you overfit
2) Add more data
3) Data augmentation
4) Generalizable architectures
5) Regularization
6) Reduce architecture complexity

Custom model: a Siamese network

```
class SiameseModel(Module):
  def __init__(self, encoder, head):
    self.encoder, self.head = encoder, head
  def forward(self, x1, x2):
    ftrs = torch.cat([self.encoder(x1),
                      self.encoder(x2)], dim=1)
    return self.head(ftrs)
encoder = create_body(resnet34, cut=-2)
head = create_head(512*4, 2, ps=0.5)
model = SiameseModel(encoder, head)
# convert boolean targets to integers in loss fn:
def loss_func(out, targ):
  return nn.CrossEntropyLoss()(out, targ.long())
# Custom splitter: how to split the model into parameter groups, so we can train only the head of the model in transfer lrn:
def siamese_splitter(model):
  return [params(model.encoder), params(model.head)]
learn = Learner(dls, model, loss_func=loss_func,
  splitter=siamese_splitter, metrics=accuracy)
# need to call freeze | unfreeze manually
```

NLP (AWD_LSTM)

- select the stacked RNN for the encoder in the language model (= activation for every word in the input)
- to create a classifier, we use "BPTT for text clsf":
→ divide a doc into fixed-length batches of size b
→ initialize each batch with final state of prev batch
→ keep track of hidden states for mean and max pooling
→ gradients are back-propagated to the batches whose hidden states contributed to the final prediction
→ classifier contains a for-loop over each batch of a sequence
→ we maintain state and store activations of each batch
→ at the end, we use avg and max concatenated pooling over RNN sequences
→ each text needs to be treated separately (because of various lengths, we add padding)
Later randomness, so texts of similar size are batched together)

Embedder

get activations from each embedding matrix

```
forward method:
if self.n_emb != 0:
  x = [e(x_cat[:,i]) for i,e in enumerate(self.embeds)]
  x = torch.cat(x,1) # concat embeddings into a single tensor
  x = self.emb_drop(x) # dropout
if self.n_cont != 0:
  x_cont = self.bn_cont(x_cont) # batch norm
  x = torch.cat([x,x_cont],1) if self.n_emb != 0 else x_cont
return self.layers(x)
```

Fastbook ch 16 notes by @dle21

THE TRAINING PROCESS
OPTIMIZERS & CALLBACKS

Baseline: `def get_learner(**kwargs):`
`return cnn_learner(dls, resnet34, pretrained=False, metrics=accuracy, **kwargs).to_fp16()`

SGD: `learn = get_learner(opt_func=SGD)` (turn off momentum)
`learn.fit_one_cycle(3, 0.03, moms=(0,0,0))`

With large beta, we might miss that the gradients changed direction and roll over small local minima

this is actually a better place!

New inputs will #in the beginning be similar to our training data, but not exactly the same - the model will generalize better if we end up in a wide flat area of the loss function

Generic Optimizer: 2 key methods:

```
def zero_grad(self):
    for p,*_ in self.all_params():
        p.grad.detach_() # remove history of gradient computations
        p.grad.zero_()

def step(self):
    for p,pg,state,hyper in self.all_params():
        for cb in self.cbs: # loop through callbacks
            state = _update(state, cb(p, **{**state, **hyper}))
        self.state[p] = state   # calls the callbacks to update params
```

```
# SGD callback:
def sgd_cb(p, lr, **kwargs): p.data.add_(-lr, p.grad.data)
# Pytorch add_ with 2 parameters -> multiply them before adding
opt_func = partial(Optimizer, cbs=[sgd_cb]) # SGD
```

Momentum

Iron ball rolling down a mountain will pass over bumps and holes, a ping pong ball is more likely to always follow direction of the gradient and get stuck (heavier = more momentum)

→ use moving average, instead of only the current gradient

```
weight.avg = beta * weight.avg + (1-beta) * weight.avg
new_weight = weight - lr * weight.avg
```

beta: how much momentum we use (default: 0.9, SGD: 0)
weight.avg ⇒ we need to store moving avgs for each param!

fit_one_cycle momentum: 0.95 → 0.85 → 0.95


```
def average_grad(p, mom, grad_avg=None, **kwargs):
    if grad_avg is None: grad_avg = torch.zeros_like(p.grad.data)
    return {'grad_avg': grad_avg*mom + p.grad.data}
def momentum_step(p, lr, grad_avg, **kwargs):
    p.data.add_(-lr, grad_avg)
opt_func = partial(Optimizer, cbs=[average_grad, momentum_step], mom=0.9)
```

RMSProp (Hinton)

Adaptive learning rate: each param gets its own lr, controlled by global lr:

- if gradients close to zero for a while → the loss is flat → need higher lr
- if gradients all over the place → risks of divergence → pick lower lr
- we determine this by using moving average of gradients squared (low avg → higher lr)

```
w.square_avg = alpha * w.square_avg + (1-alpha) * (w.grad**2)
new_w = w - lr * w.grad / math.sqrt(w.square_avg + eps)
```

current gradient; divided by square root of moving avg; numerical stability

```
def average_sqr_grad(p, sqr_mom, sqr_avg=None, **kwargs):
    if sqr_avg is None: sqr_avg = torch.zeros_like(p.grad.data)
    return {'sqr_avg': sqr_avg*sqr_mom + (p.grad.data**2)*(1-sqr_mom)}
def rms_prop_step(p, lr, sqr_avg, eps, grad_avg=None, **kwargs):
    denom = sqr_avg.sqrt().add_(eps)
    p.data.addcdiv_(-lr, p.grad, denom)
opt_func = partial(Optimizer, cbs=[average_sqr_grad, rms_prop_step], sqr_mom=0.99, eps=1e-7)
```

Adam: SGD with momentum + RMSProp together!
→ moving avg of gradient as direction
→ adaptive lr for each parameter

Difference: using unbiased moving average

```
w.avg = beta * w.avg + (1-beta) * w.grad   # i: ite.
unbias_avg = w.avg / (1 - (beta**(i+1)))   # ration
```

unbiased avg looks more like gradients after more steps similar to moving average.

```
w.sqr_avg = beta2 * w.sqr_avg + (1-beta2) * (w.grad**2)
new_w = w - lr * unbias_avg / sqrt(w.sqr_avg + eps)
```

	beta1	beta2	eps
default:	0.9	0.999	1e-8
fastai:	0.9	0.99	1e-5

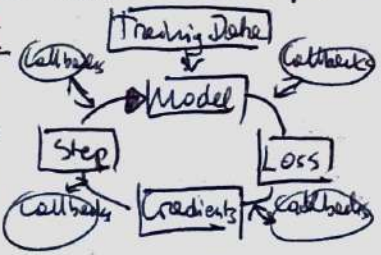
eps also limits max value of adjusted lr!

Decoupled Weight Decay: 2 formulations:

① `new_w = w - lr * w.grad - lr * wd * w`
② `w.grad += wd * w`

In SGD both are equivalent, with Adam or Momentum, ① is correct - used in fastai!

CALLBACKS



Make changes to the training process without changing the training loop!

Callback: a piece of code we can inject into another piece of code at a predefined point

Training loop:

```
for xb, yb in dl:
    try:
        self.split(b);                          self('begin_batch')
        self.pred = self.model(*self.xb);       after_pred
        self.loss = self.loss_func(self.pred, *self.yb);  after_loss
        if not self.training: return
        self.loss.backward();                   after_backward
        self.opt.step();                        after_step
        self.opt.zero_grad()
    except CancelBatchException:                after_cancel_batch
    finally:                                    after_batch
```

→ Callbacks can access attributes of Learner and modify them
→ Callbacks can raise an Exception, eg cancel training if the loss becomes inf or NaN
→ Callbacks can be called in particular order (specify: run_before, run_after)

Fastbook ch 17 notes by @dv21 | A Neural Net from the Foundations → Broadcasting

A neuron: $out = \sum_{i=1}^{n} x_i w_i + b$ (receive n inputs; weight for each input; adds an inner bias)

output = sum([x*w for x,w in zip(inputs, weights)]) + bias

We need to feed this into a non-linear activation function, eg:

def relu(x): return x if x>=0 else 0

* fully connected (dense, linear) layer: all inputs are linked to all neurons via a dot product.

x: inputs matrix (batch_size x n_inputs)
w: weights matrix (n_neurons x n_inputs)
b: biases vector (n_neurons)

y = x @ w.t() + b (batch_size x n_neurons) ← output of a fully connected layer

Matrix multiplication from scratch

```
def matmul(a,b):
  ar, ac = a.shape # n_rows x n_cols
  br, bc = b.shape
  assert ac==br
  c = torch.zeros(ar, bc)
  for i in range(ar):
    for j in range(bc):
      for k in range(ac):
        c[i,j] += a[i,k] * b[k,j]
  return c
```

(the three inner loops: see below)

→ with elementwise arithmetic:

```
c[i,j] = (a[i,:] * b[:,j]).sum()
```

→ with broadcasting:

```
c[i] = (a[i].unsqueeze(-1) * b).sum(dim=0)
```

Elementwise arithmetic:

Basic operators (+, -, *, /, <, >, ==) can be applied elementwise on tensors of the same size.

- If we want to know if all elements of tensor a are < tensor b, or if tensors are equal, we need to combine elementwise with torch.all: `(a<b).all()`, `(a==b).all()`
- reduction operations: all, sum, mean return rank 0 tensor (1 element) - convert to Python Boolean/number with .item(): `(a+b).mean().item()`

Broadcasting

Describes how tensors of different ranks are treated during arithmetic ops:
- rules to codify when tensors are compatible
- rules how to expand smaller tensor to match shape of a bigger tensor

① Broadcasting with scalar: expand scalar to shape of bigger tensor

② Broadcasting a tensor to a matrix

```
c = tensor([10.,20,30])
m = tensor([[1.,2,3],[4,5,6],[7,8,9]])
m.shape, c.shape # [3,3], [3]
```

m + c → c expanded to make 3 rows that match m

```
t = c.expand_as(m) → tensor([[10.,20,30],
                             [10,20,30],
                             [10,20,30]])
```

t.storage() → torch.FloatStorage of size 3 ⇒ PyTorch doesn't create copies

Expanded dimension gets a stride of 0 when PyTorch looks for the next row by adding stride, it doesn't move: `t.stride(), t.shape # (0,1), [3,3]`

Broadcasting: Pytorch executes `c.unsqueeze(0)` behind the scenes, so c is expanded on row side. We can change this: `c = c.unsqueeze(1)`, now it will expand on column side.

Comparing 2 tensors, PyTorch starts with the trailing dimensions, and goes backward, adding 1 on empty dimension. Compatible dims: equal, or one is 1.

Einstein summation

```
def matmul(a,b): return torch.einsum('ik,kj->ij', a, b)
```

- used for combining products and sums
- operands dimensions on the left, seperated by commas
- results dims on the right
- repeated indices are implicitly summed over.

Chain rule: $(g \circ f)'(x) = g'(f(x))\, f'(x)$

$y = g(u)$, $u = f(x)$ ⇒ $\frac{dy}{dx} = \frac{dy}{du} * \frac{du}{dx}$

SymPy: lib for symbolic computation.

```
from sympy import symbols, diff
sx, sy = symbols('sx sy')
diff(sx**2, sx) # 2 * sx
```

The forward and backward passes

```
x = torch.randn(200,100); y = torch.randn(200)
```

① We have to scale our weights so that activations have mean = 0 and std = 1. std < 0 or > 0 will lead to 0 or nan in our activations after many multiplications. → Initialization (Xavier (Glorot), Kaiming He)

```
w1 = torch.randn(100,50) / sqrt(100)  → / sqrt(2/100)
w2 = torch.randn(50,1)   / sqrt(50)   → / sqrt(2/50)
```

div by $1/\sqrt{n_{in}}$, n_{in} = number inputs (Glorot - tanh act.)
div by $1/\sqrt{2/n_{in}}$ → Kaiming He, for relu activation

Gradients and the backward pass (chain rule)

To compute all gradients for the update, we begin from the model output and work backward through the layers (⇒ backpropagation). We can automate it by having each function we implemented provide its backward step and populate gradients in an attribute of each tensor (like PyTorch .grad), eg:

```
class Lin():
  def __init__(self, w, b): self.w, self.b = w, b
  def __call__(self, inp):
    self.inp = inp
    self.out = inp @ self.w + self.b
    return self.out
  def backward(self):
    self.inp.g = self.out.g @ self.w.t()
    self.w.g = self.inp.t() @ self.out.g
    self.b.g = self.out.g.sum(0)
```

__call__ makes our class callable in Python

```
class Model():
  def __init__(self, w1, b1, w2, b2):
    self.layers = [Lin(w1,b1), Relu(), Lin(w2,b2)]
    self.loss = Mse()
  def __call__(self, x, targ):
    for l in self.layers: x = l(x)
    return self.loss(x, targ)
  def backward(self):
    self.loss.backward()
    for l in reversed(self.layers): l.backward()
```

```
model = Model(w1, b1, w2, b2) # instantiate
loss = model(x, y) # forward pass
model.backward() # backward pass.
```

Fastbook Ch 18 notes by @dlii21

CNN INTERPRETATION WITH CAM

Class Activation Map (CAM) Zhou et al.

Uses the output of the last convolutional layer (just before average pooling) together with the predictions to give us a heatmap visualization of why the model made its decision
- at each point/position of our final conv layer, we have as many filters as in the last linear layer
- we can compute the dot product of these activations with the final weights to get - for each location on our feature map - the score of the feature that was used to make a decision

Pytorch Hook: like fastai callback
Use it to get access to activations inside the model while it's training (inject code into the forward and backward calculations)

Example: cats vs dogs model

```
...
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1)
class Hook():
  def hook_func(self, m, i, o):
    self.stored = o.detach().clone()
hook_output = Hook()
hook = learn.model[0].register_forward_hook(
    hook_output.hook_func)
img = PILImage.create('images/cat.jpg')
x, = first(dls.test_dl([img]))
with torch.no_grad():
  output = learn.model.eval()(x)
act = hook_output.stored[0]
F.softmax(output, dim=-1)
x.shape -> [1, 3, 224, 224]
cam_map = torch.einsum('ck,kij->cij',
  learn.model[1][-1].weight, act)
cam_map.shape -> [2, 7, 7]
```

We put hook function in a class so it has a state we can access later, just store copy of the output
Instantiate hook and attach it to the last layer of the cnn body
put a cat pic in a batch
grab a batch and feed it through our model
access stored activations
verify prediction
Einsum does dot product of weight matrix (2x number of activations) with the activations (batch size x activations x rows x columns)
7x7 feature map for each img/class - high/low activations

Visualize CAM map

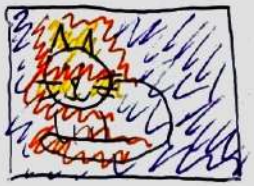
```
x_dec = TensorImage(
  dls.train.decode((x,))[0][0])
_,ax = plt.subplots()
x_dec.show(ctx=ax)
ax.imshow(cam_map[1].detach().cpu(),
  alpha=0.6, extent=(0,224,224,0),
  interpolation='bilinear', cmap='magma');
hook.remove()
```

We need to decode input x as it was normalized by DataLoader, also cast to TensorImage
remove when done to avoid leaking memory, OR:

Context manager: Python construct that calls __enter__ when the object is created in a **with** clause, and __exit__ at the end of the **with** clause
→ have the Hook class be a context manager, registering the hook when we enter it, and removing it when we exit

```
class Hook():
  def __init__(self, m):
    self.hook = m.register_forward_hook(
        self.hook_func)
  def hook_func(self, m, i, o):
    self.stored = o.detach().clone()
  def __enter__(self, *args): return self
  def __exit__(self, *args):
    self.hook.remove()

with Hook(learn.model[0]) as hook:
  with torch.no_grad():
    output = learn.model.eval()(x.cuda())
  act = hook.stored
```



Gradient-CAM (Selvaraju et al.)

CAM works only for the last layer.
Grad-CAM uses the gradients of the final activation for the desired class
(gradients of the output of the last layer with respect to input of that layer = layer weights, since it's a linear layer)
In deeper layers, gradients are no longer equal to weights, calculated by PT in the backward pass, but not stored. We can register a hook on the backward pass to store them.

```
class HookBwd():
  def __init__(self, m):
    self.hook = m.register_backward_hook(
        self.hook_func)
  def hook_func(self, m, gi, go):
    self.stored = go[0].detach().clone()
  def __enter__(self, *args): return self
  def __exit__(self, *args): self.hook.remove()
cls = 1  # cat = True
with HookBwd(learn.model[0][-2]) as hookg:
  with Hook(learn.model[0][-2]) as hook:
    output = learn.model.eval()(x.cuda())
    act = hook.stored
  output[0,cls].backward() *
  grad = hookg.stored
w = grad[0].mean(dim=[1,2], keepdim=True)
cam_map = (w * act[0]).sum(0)
```

Store gradients instead of activations!
We can use it on any layer - here [-2]

* We can't use output.backward() → gradients make sense only with respect to a scalar (normally our loss) and output here is a rank-2 tensor. But if we pick a single image and a single class, we can calculate the gradients of weights or activations wrt that single value. We'll use these gradients as weights.
Weights for Grad-CAM = average of our gradients across the feature map.

Fastbook ch.19 (1) notes by @dli21 | A fastai Learner from Scratch, p.1

DATA

untar_data
get_image_files : glob vs os.walk
Path.parent (pathlib)
L.val2idx , L.map
Python : ~v -> filter = [True, False]
a = a[list[filter] ; b = a[list[~filter]

DATASET

```
class Dataset:
    def __init__(self,fns): self.fns=fns
    def __len__(self) : return len(self.fns)
    def __getitem__(self,i):
        im = Image.open(self.fns[i]).resize((64,64))
             .convert('RGB')
        y = v2i[self.fns[i].parent.name]
        return tensor(im).float()/255, tensor(y)
```

→ this returns x,y as a tuple, that we need to collate into a mini batch

```
def collate(idxs,ds):
    xb,yb = zip(*[ds[i] for i in idxs])
    return torch.stack(xb), torch.stack(yb)
class DataLoader:
    def __init__(self,ds,bs=128,shuffle=False,n_workers=1):
        self.ds,self.bs,self.shuffle,self.n_workers = ds,bs,shuffle,n_workers
    def __len__(self): return (len(self.ds)-1)//self.bs+1
    def __iter__(self):
        idxs = L.range(self.ds)
        if self.shuffle: idxs = idxs.shuffle()
        chunks = [idxs[n:n+self.bs] for n in range(
                  0, len(self.ds), self.bs)]
        with ProcessPoolExecutor(self.n_workers) as ex:
            yield from ex.map(collate, chunks, ds=self.ds)
```

→ we need to do our processing in parallel - opening and decoding a JPEG image is a slow process, one CPU core is not enough to decode images fast enough to keep a GPU busy.

```
n_workers = min(16, defaults.cpus)
```

We'll need image stats for normalization:

```
stats = [xb.mean((0,1,2)), xb.std((0,1,2))]
class Normalize:
    def __init__(self,stats): self.stats=stats
    def __call__(self,x):
        if x.device != self.stats[0].device:
            self.stats = to_device(self.stats, x.device)
        return (x-self.stats[0])/self.stats[1]
norm = Normalize(stats)
def tfm_x(x): return norm(x).permute((0,3,1,2))
```

→ permute the axis order from PIL (HWC) to Pytorch (NCHW)

MODULE AND PARAMETER

```
class Module:
    def __init__(self):
        self.hook,self.params,self.children,self._training = None,[],[],False
    def register_parameters(self,*ps): self.params += ps
    def register_modules(self,*ms): self.children += ms
    @property
    def training(self): return self._training
    @training.setter
    def training(self,v):
        self._training = v
        for m in self.children: m.training = v
    def parameters(self):
        return self.params + sum([m.parameters()
            for m in self.children], [])  #recursion
    def __setattr__(self,k,v):
        super().__setattr__(k,v)
        if isinstance(v,Parameter): self.register_parameters(v)
        if isinstance(v,Module): self.register_modules(v)
    def __call__(self,*args,**kwargs):
        res = self.forward(*args,**kwargs)
        if self.hook is not None: self.hook(res,args)
        return res
    def cuda(self):
        for p in self.parameters: p.data = p.data.cuda()
class Parameter(Tensor):
    def __new__(self,x): return Tensor._make_subclass(
                                Parameter, x, True)
    def __init__(self,*args,**kwargs): self.requires_grad_()
```

* Parameter doesn't add any functionality, only calls 'requires_grad' for us. It's used as a marker to show what to include in 'parameters'.

SIMPLE CNN ⇒

```
class ConvLayer(Module):
    def __init__(self,ni,nf,stride=1,bias=True,act=True):
        super().__init__()
        self.w = Parameter(torch.zeros(nf,ni,3,3))
        self.b = Parameter(torch.zeros(nf)) if bias else None
        self.act,self.stride = act,stride
        init = nn.init.kaiming_normal_ if act else nn.init.xavier_normal_
        init(self.w)
    def forward(self,x):
        x = F.conv2d(x, self.w, self.b, stride=self.stride, padding=1)
        if self.act: x = F.relu(x)
        return x
class Linear(Module):
    def __init__(self,ni,nf):
        super().__init__()
        self.w = Parameter(torch.zeros(nf,ni))
        self.b = Parameter(torch.zeros(nf))
        nn.init.xavier_normal_(self.w)
    def forward(self,x): return x@self.w.t() + self.b
class Sequential(Module):
    def __init__(self,*layers):
        super().__init__()
        self.layers = layers
        self.register_modules(*layers)
    def forward(self,x):
        for l in self.layers: x = l(x)
        return x
class AdaptivePool(Module):
    def forward(self,x): return x.mean((2,3))
def simple_cnn():
    return Sequential(
        ConvLayer(3,16,stride=2),  #32
        ConvLayer(16,32,stride=2), #16
        ConvLayer(32,64,stride=2), #8
        ConvLayer(64,128,stride=2), #4
        AdaptivePool(),
        Linear(128,10)
    )
```

Fastbook ch.19 (2) notes by @dlu21 | A fastai Learner from Scratch, p.2

LOSS

```
def nll(input, target): return -input[range(target.shape[0]), target].mean()
1) def log_softmax(x): return (x.exp()/(x.exp().sum(-1, keepdim=True))).log()
2) def log_softmax(x): return x - x.exp().sum(-1, keepdim=True)).log()  # log(a/b) = log(a)-log(b)
3) def logsumexp(x):
       m = x.max(-1)[0]  # LogSumExp trick..
       return m + (x-m[:,None]).exp().sum(-1).log()
def logsoftmax(x): return x - x.logsumexp(-1, keepdim=True)  .. for numeric stability
def cross_entropy(preds, yb): return nll(log_softmax(preds), yb).mean()
```

CALLBACKS

```
class Callback(GetAttr): _default = 'learner'   (*)
class SetupLearnerCB(Callback):
    def before_batch(self):
        xb, yb = to_device(self.batch)
        self.learner.batch = tfm_x(xb), yb
    def before_fit(self): self.model.cuda()
class TrackResults(Callback):
    def before_epoch(self):
        self.accs, self.losses, self.ns = [],[],[]
    def after_epoch(self):
        n = sum(self.ns)
        print(self.epoch, self.model.training,
              sum(self.losses).item()/n, sum(self.accs).item()/n)
    def after_batch(self):
        xb, yb = self.batch
        acc = (self.preds.argmax(dim=1)==yb).float().sum()
        self.accs.append(acc)
        n = len(xb)
        self.losses.append(self.loss*n)
        self.ns.append(n)
cbs = [SetupLearnerCB(), TrackResults()]
learn = Learner(simple_cnn(), dls, cross_entropy, lr=0.1, cbs=cbs)
learn.fit(1)
```

LEARNER

```
class SGD:
    def __init__(self, params, lr, wd=0.): store_attr(self, 'params,lr,wd')
    def step(self):
        for p in self.params:
            p.data -= (p.grad.data + p.data*self.wd)*self.lr
            p.grad.data.zero_()
class DataLoaders:
    def __init__(self, *dls): self.train, self.valid = dls
class Learner:
    def __init__(self, model, dls, loss_func, lr, cbs, opt_func=SGD):
        store_attr(self, 'model,dls,loss_func,lr,cbs,opt_func')
        for cb in cbs: cb.learner = self
    def one_batch(self):
→       self('before_batch')
        xb, yb = self.batch
        self.preds = self.model(xb)
        self.loss = self.loss_func(self.preds, yb)
        if self.model.training:
            self.loss.backward()
            self.opt.step()
→       self('after_batch')
    def one_epoch(self, train):
        self.model.training = train
→       self('before_epoch')
        dl = self.dls.train if train else self.dls.valid
        for self.num, self.batch in enumerate(progress_bar(dl, leave=False)):
            self.one_batch()
→       self('after_epoch')
    def fit(self, n_epochs):
→       self('before_fit')
        self.opt = self.opt_func(self.model.parameters(), self.lr)
        self.n_epochs = n_epochs
        try:
            for self.epoch in range(n_epochs):
                self.one_epoch(True)
                self.one_epoch(False)
        except CancelFitException: pass
→       self('after_fit')
    def __call__(self, name):
        for cb in self.cbs: getattr(cb, name, noop)()
```

- every callback knows what learner it is used in → get info from learner → change things in learner
- when Learner calls self, it calls `__call__`, which uses `getattr(cb, name)` on each callback in self.cbs where the attribute (here: method) is defined.
- eg. self('before_fit') will call cb.before_fit() if defined

SCHEDULING THE LEARNING RATE

```
class LRFinder(Callback):
    def before_fit(self):
        self.losses, self.lrs = [],[]
        self.learner.lr = 1e-6
    def before_batch(self):
        if not self.model.training: return
        self.opt.lr *= 1.2
    def after_batch(self):
        if not self.model.training: return
        if self.opt.lr>10 or torch.isnan(self.loss):
            raise CancelFitException
        self.losses.append(self.loss.item())
        self.lrs.append(self.opt.lr)
class OneCycle(Callback):
    def __init__(self, base_lr): self.base_lr = base_lr
    def before_fit(self): self.lrs = []
    def before_batch(self):
        if not self.model.training: return
        n = len(self.dls.train)
        bn = self.epoch*n + self.num
        mn = self.n_epochs*n
        pct = bn/mn
        pct_start, div_start = 0.25, 10
        if pct < pct_start:
            pct /= pct_start
            lr = (1-pct)*self.base_lr/div_start + pct*self.base_lr
        else:
            pct = (pct-pct_start)/(1-pct_start)
            lr = (1-pct)*self.base_lr
        self.opt.lr = lr
        self.lrs.append(lr)
```

(*) GetAttr is a fastai class that implements Python's `__getattr__` and `__dir__`: if we try to access attribute that doesn't exist, it passes the request to `_default`, eg:
self.model.cuda → self.learner.model.cuda
(only works for getting attributes, not setting)

Fastbook ch.20 notes by @dk21

2020: Deep Dive into Machine Learning, Deep Learning, Data Science

active on Fastai Forum

active on Kaggle Forum

I started a blog: sicok.ai

I published a paper: 'Polbert: Attacking Polish NLP Tasks with Transformers'

NEW: to be announced :)

Fastbook notes

NEW: Kaggle competition notes

Huge thanks to Jeremy Howard, Sylvain Gugger and Rachel Thomas & fastai community for the knowledge and inspiration!

HELP

WRITE

Concluding Thoughts

GATHER

BUILD

Kaggle kernels, eg. OpenVaccine LSTM

Polbert: the first published Polish BERT model

Co-Organizing Polish NLP Meetup Group

2021: carry on, build meaningful & ethical data products and contribute to the community.

Kaggle competitions 2 silvers, targeting gold (Master level) in 2021

NEW: fastai community Meetup group

Poleval: winning Polish NLP competition task: Word Sense Disambiguation

Contribute to Hugging Face library (datasets)

ML Community of Practice @ work

Work projects: Image Similarity, Object detection, Classification, Random Forest - tabular

Dareh Mieszek
Warsaw, 1.1.2021