fastbook ch.1
by @dk21

DEEP LEARNING APPLICATIONS: NLP, VISION, MEDICINE, BIOLOGY, IMAGE GENERATION, RECOMMENDATION SYSTEMS, ROBOTICS, LOGISTICS, FINANCE

CLASSICAL PROGRAM: inputs → program → results

→ ML APPROACH: inputs → model architecture ← parameters; model architecture → predictions → loss ← labels; loss → update → parameters

ML DURING INFERENCE: inputs → model → results

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

LIMITATIONS OF ML:
- Need data with labels
- Can only learn patterns seen in 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

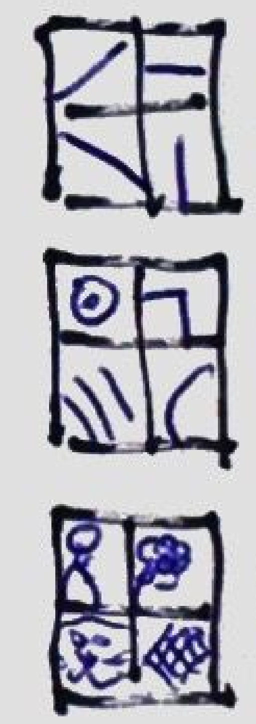
1961/62: Samuel - Artificial Intelligence essay, introducing current ML approach, program beating humans in checkers

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 the metric
learn.fine_tune(1)   # finetune ~ fit pretrained model
```

Different layers in NNs learn to recognize increasingly complex features.

CNN - Convolutional Neural Network example:

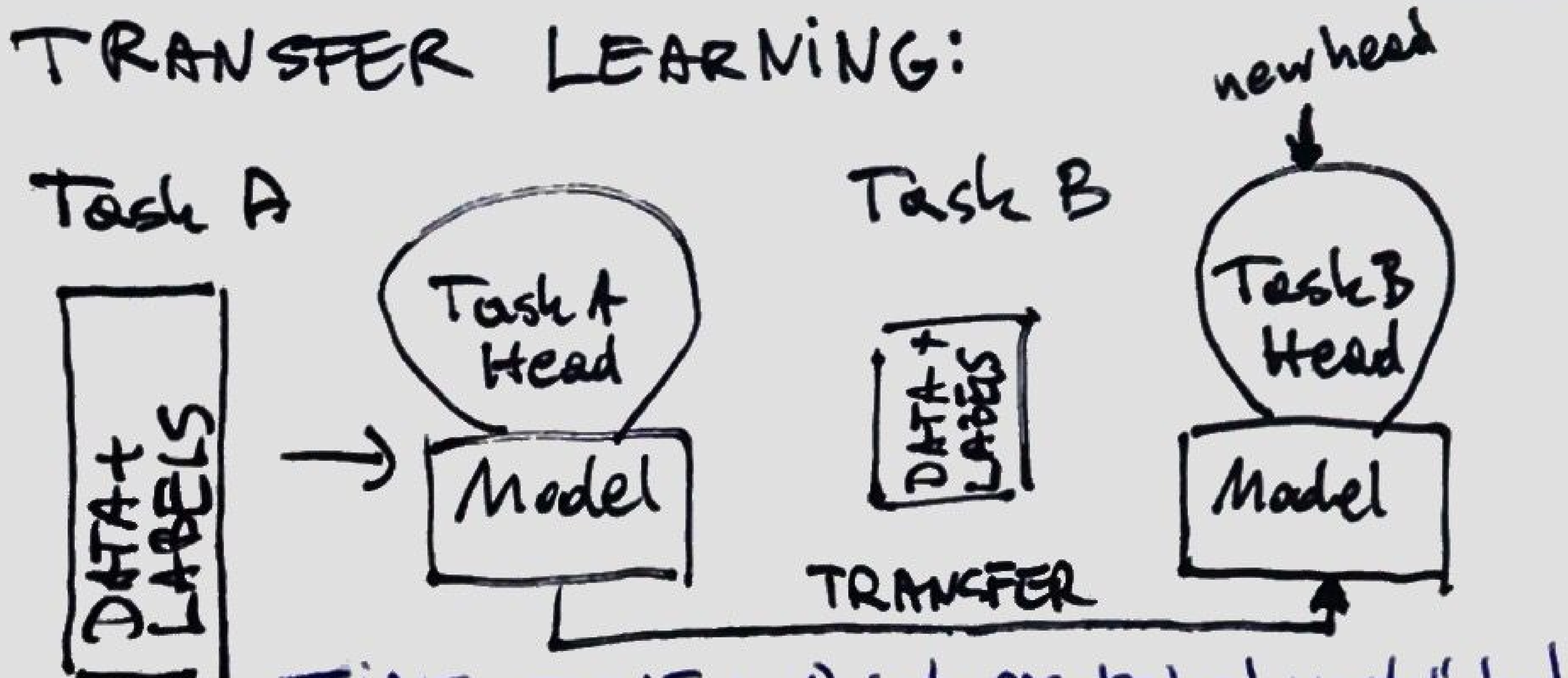


L1: edges

L2: corners, shapes, colors, patterns

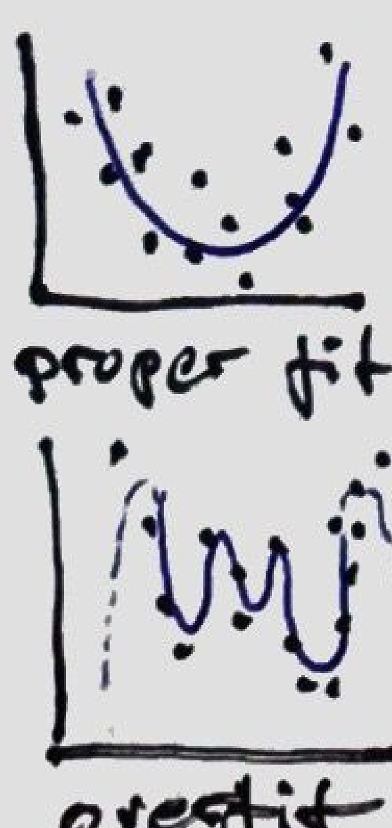
L3+: higher level components: faces, flowers, etc.

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:

TRAINING	VALID	TEST
Train data on this only	Tune model, choose hyperparameters, see performance on chosen metric on valid	Hide test, use it to estimate performance at the very end

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

LOSS
METRIC
Universal Approximation Theorem
CLASSIFICATION
REGRESSION
EPOCH PARAMETERS
ARCHITECTURE, CNN
SEGMENTATION
GPU, NOTEBOOK

OTHER TOPICS CONSTRAINED BY SPACE

fastbook ch 2 by @ ..

DL STATE 2020:

Computer Vision:
✓ object recognition
✓ object detection
└ image captioning

Text: Classification, translation, summarization, question answering, NER
Generation: compelling but not factually correct!
└ Other data types: eg protein chains

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.

DRIVETRAIN APPROACH: produce actionable outcomes, vs. predictions only

What outcome am I trying to achieve? — DEFINED OBJECTIVE

What inputs can we control? — LEVERS

What data can we collect? — DATA

How the levers influence the objective — MODELS

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

INFERENCE, CPU vs GPU

Voilà ipywidgets

map method
L.map(function)

space constrained

WORKFLOW:

Train quick & simple model → Plot confusion matrix, top losses → Analyze errors - is it dataset or model? → Data cleaning, model improvement → Train quick & simple model

```
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

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?

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.

fastbook ch3
notes by @dk21

ETHICS: the study of right & 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

Spiral effect

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

! 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] treat 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 blind spots!
   ⇒ 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

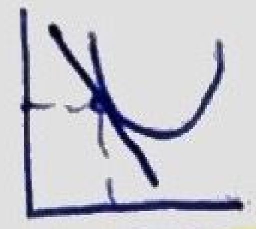
@ - PyTorch matrix multiplication
batch@weights + bias ⇒ fundamental operation in NN
W * x + b
weights, biases } parameters

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)
init: initialize to random values
loss: function reflecting performence of the model
gradient, step: find direction to change weights and do it



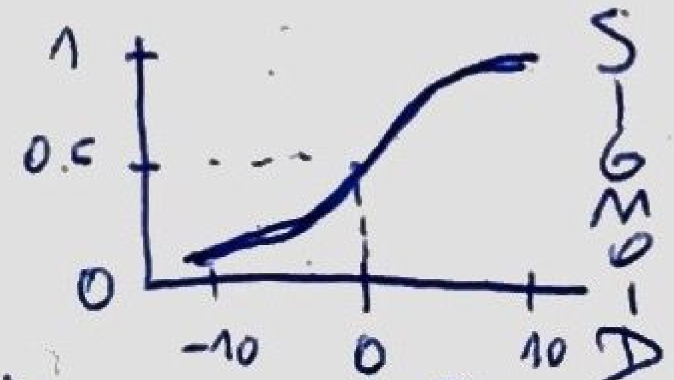
Derivative of a function tells us how much a change in parameters will change results (rise / run)

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.)
```

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

Fastbook ch.5
notes by @dk21

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

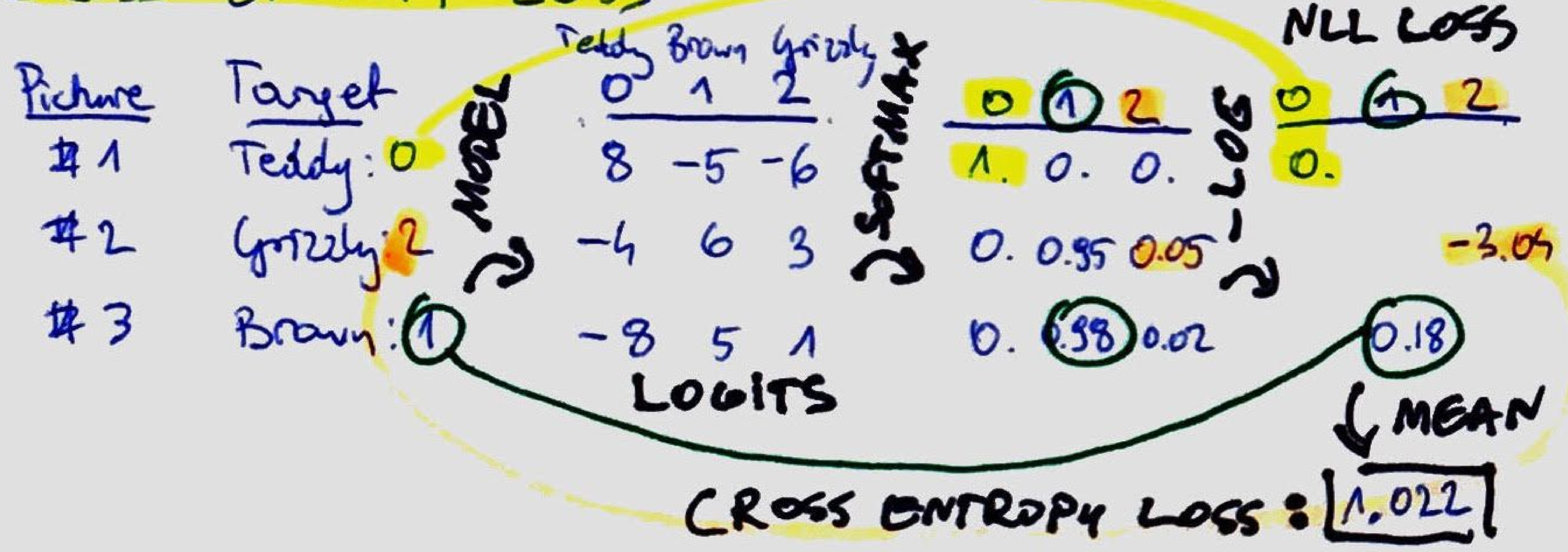
① Resize images to relatively large dimension (vs target training dimension)

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

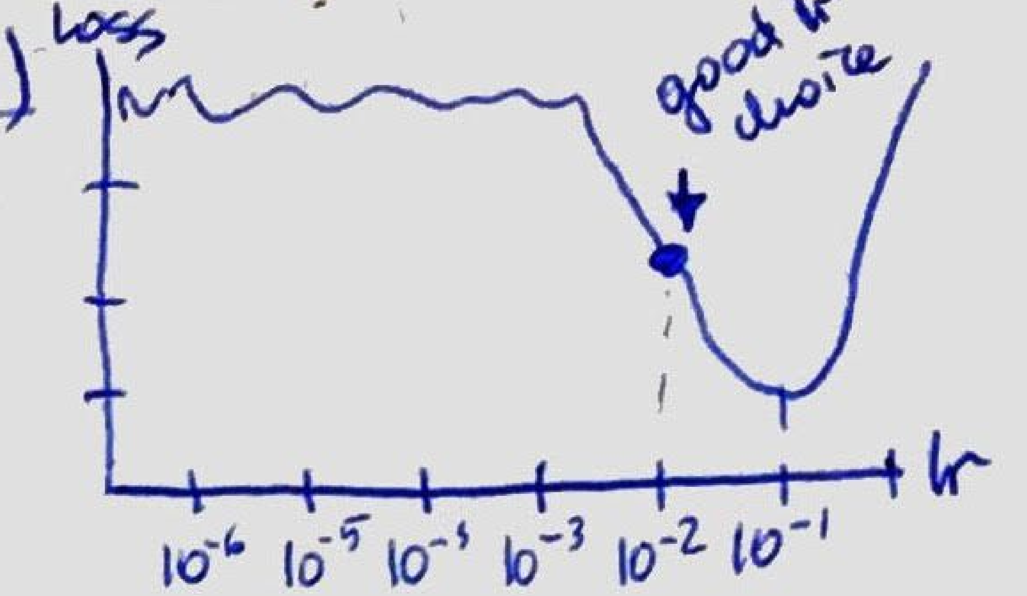
def softmax(x): return exp(x)/exp(x).sum(dim=1, keepdim=True)

- if one activation is slightly higher then 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

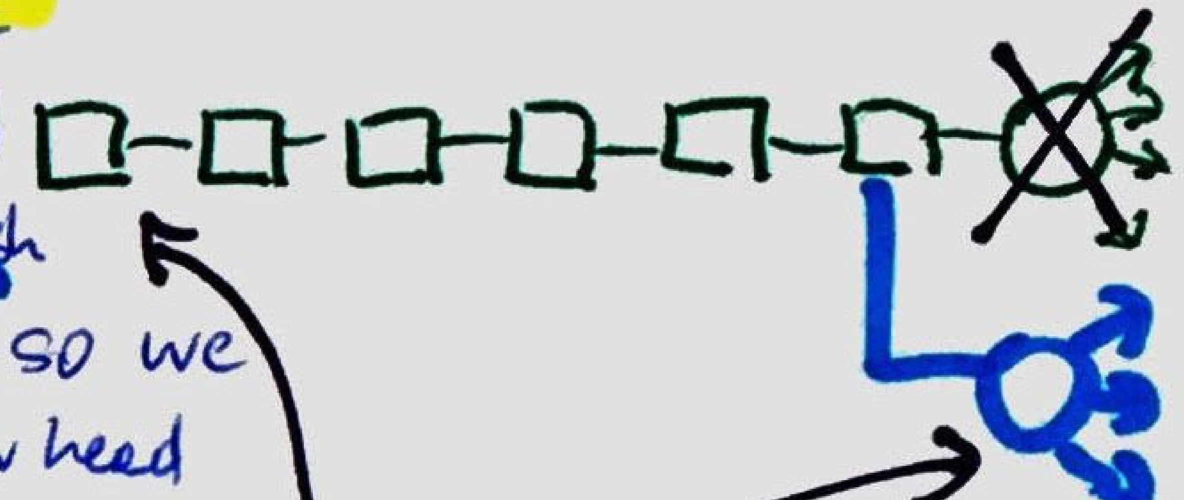
Pytorch: nn.CrossEntropyLoss ⇒ nn.LogSoftmax + nn.NLLLoss

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

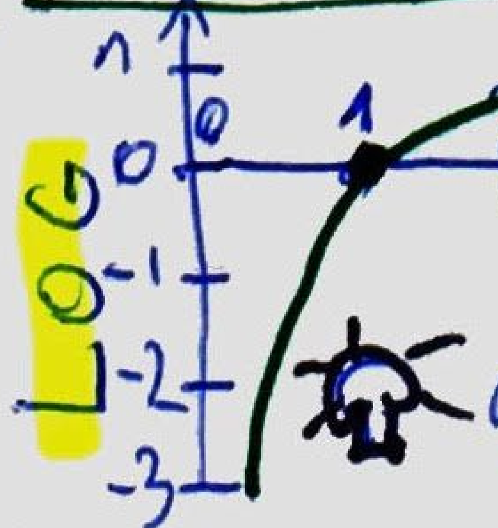
Discriminative learning rates: pass slice(lr1, lr2) in fastai
→ train first layers with smaller lr, last layer with higher lr, range between – multiplicatively equidistant lr's.

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 @dkk21

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 Data Block 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 targets: -log(sigmoid)
neg. targets: -log(1-sigmoid)

↓ mean()

BCE LOSS = 0.3777

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

```
loss_func = nn.BCEWithLogitsLoss()
loss = loss_func(activs, 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, thresh=0.5,
              sigmoid=True):
    if sigmoid: inp = inp.sigmoid()
    return ((inp > thresh) == targ.bool()).
           float().mean()
```

PARTIAL example (Python):

```
acc_02 = partial(accumulti, 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!

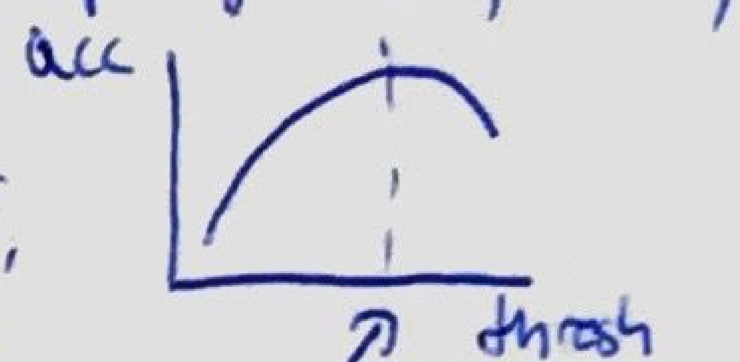
Rather than focusing on domains, focus on:

Independent var	Dependent var
Image	Text (caption)
text	Image (from capt.)
image + text + tabular	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 @dk21 | 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 - Fastbook
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

$u = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ ← OHE 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 OHE vector

Bootstrapping problem (- cold start)

→ pick some user to represent avg 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):
  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):
    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)
```

in PyTorch we need to inherit from Module

in PyTorch forward method is called whenever module is called, passing along any parameters included in the call

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, **kwargs, @delegates decorator

NO 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 to 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 @tt21

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'].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, now use the residuals as targets
5) Continue until a stopping criterion: max no trees, valid error getting worse, etc.

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

TABULAR MODELING WORKFLOW

1) Start with RF - easiest to train, resilient 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 columns
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 dsets
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 3

= 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 + convenience)
TabularProc
 - Categorify
 - FillMissing

tabular_learner
TabularModel
⇒ study source code

Other:
dtreeviz
add_datepart

Fastbook ch.10 notes by @di21 | NLP DEEP 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

[txt] - collection of text documents

```
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)
```

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 // **LMDataLoader**

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(
    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)
```

when TextBlock passed fastai handles tokenization and numericalization

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)
learn.save('1epoch')
learn = learn.load('1epoch')
learn.unfreeze()
learn.fit_one_cycle(10, 2e-3)
learn.save_encoder('finetuned')
```

automatically frozen, this will only train embeddings

use fit-one-cycle to save/load intermediate model results

ENCODER = model without task-specific final layers, like body in CNNs

TEXT GENERATION

```
TEXT = 'I liked this movie because'
N_WORDS = 40
pred = learn.predict(TEXT, N_WORDS, temperature=0.75)
```

creating the classifier dataloaders

! is_lm = False

```
dls_clas = DataBlock(
    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)
```

pass vocab to use fine-tuned embeddings

```
learn = text_classifier_learner(dls_clas, AWD_LSTM, drop_mult=0.5,
    metrics=accuracy).to_fp16()
learn = learn.load_encoder('finetuned')
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))
```

Gradual unfreezing + discriminative learning rates

Fastbook ch. 11 notes by @dh21

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 optimizer

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 - sth 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) automatically applies transforms

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 initiation, 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. Impt 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: SiamesePair
Siamese model takes two images and has to determine if they are same class or not