DIY Deep Learning: Advice on Weaving Nets

BERKELEY ARTIFICIAL INTELLIGENCE RESEARCH

with your host: Evan Shelhamer

Today

tour of collected advice

Q&A about deep learning in general and your projects in particular

Experience and Expertise

expertise noun

1. skill or knowledge

2. *knowing most of mistakes that can be made in a particular field*

DATA

know your data! if you don't, how will you know if your model does?

DATA

inspect the distribution of the inputs and targets

- inspect random selection of inputs and targets to have a general sense
- histogram input dimensions to see range and variability
- histogram targets to see range and imbalance
- select, sort, and inspect by type of target or whatever else

DATA

inspect the inliers, outliers, and neighbors

- visualize distribution and **outliers**, especially outliers, to uncover dataset issues

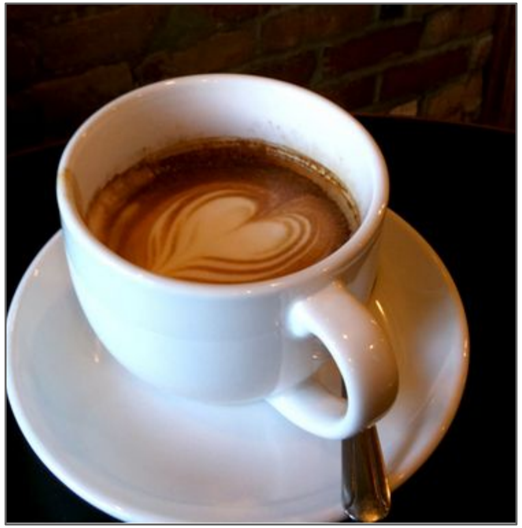
- look at **nearest neighbors**
  in the raw data, or pre-trained net for same domain, or randomly initialized net

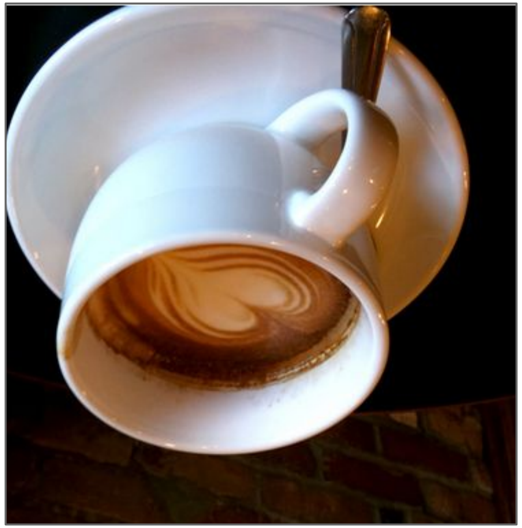
- examples:
  rare grayscale images in color dataset, huge images that should have been rescaled, corrupted class labels that had been cast to uint8

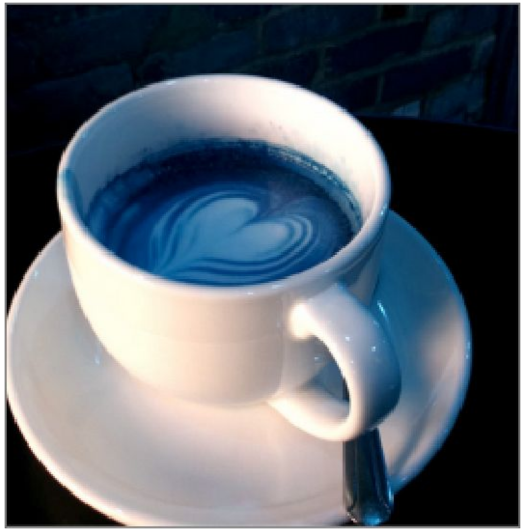
DATA

pre-processing: the data as it is loaded is not always the data as it is stored!

- inspect the data as it is given to the model by `output = model(data)`


original


DeCAF


Caffe

DATA

pre-processing:

- **summary statistics**: check the min/max and mean/variance
  to catch mistakes like forgetting to standardize

- **shape**: are you certain of each dimension and its size?
  sanity check with dummy data of prime dimensions: there are no common factors, so mistaken reshaping/flattening/permuting will be more obvious.
  example: a 64x64x64x64 array can be permuted without knowing.

- **type**: check for casting, especially to lower precision
  what's -1 for a byte? how does standardization change integer data?

DATA

pre-processing:

- start with less, and then add more… once everything works at all

- for instance: **just standardize** at first,
  and only augment once there is a model and it fits

DATA

- resample the data to decorrelate: that is, remember to **shuffle**

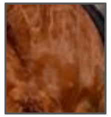
- consider selecting miniature train and test sets for development and testing: these should be chosen once and fixed throughout optimization + evaluation

  it's heartbreaking to wait an entire epoch and then and only then
  have your experiment-to-be crash

DATA

check if you can do the task, as it is given to the model

- take windowed data at receptive field size and stride



- (consider the task with and without pre-processing)

...if it's a reasonable perceptual task for a human

MODEL

keep it as simple as you can!

- sure, there are sophisticated models out there

- but they were made by either

  (1) going step-by-step, from simple to complex
  or
  (2) suffering, madness, and the gnashing of teeth

MODEL

keep it as simple as you can!

- do your first experiment with the simplest possible model w/ and w/o your idea
- at least you will be armed with a little understanding
- then follow-up with bidirectional search
  between simple models and state-of-the-art models
  with your own model in the middle

MODEL

resist snowflake syndrome! try defaults first

but my data/task/idea is special

...no it's not, or not until proven so (by say defeating a residual net)

- once while consulting I replaced 1 month+ of model development with LeNet and it was more accurate and >10,000x faster
- even if an off-the-shelf net does not prevail, it can serve as a baseline

MODEL

double-check the model actually is what you thought you defined

```
# walk the model for inspection
for name_, module_ in model.named_modules():
    if name_ == 'name' or isinstance(module_, nn.Conv2d):
        [...]
```

MODEL

simple baselines catch many issues and swiftly too

- learn a linear model on random features
  by fixing all of the parameters at their initialization except for the output

- zero out the data by `x.zero_()` and check that results are worse

OPTIMIZATION

we are still learning how to optimize deep nets

much progress is being made but it's nevertheless a dark forest

explore if you like, but balance exploration by exploitation of a known good setting, or the closest setting you can find

OPTIMIZATION

figure out optimization on **one/few/many points** in that order

- overfit to a single data point
- then fit a batch
- and finally try fitting the dataset (or a miniature edition of it)

to discover issues as soon as possible

OPTIMIZATION

sanity check the loss against a suitable reference value

- classification with cross-entropy loss: uniform distribution
- regression with squared loss: mean of targets (or even just zero)

and if your loss is constant, double-check for zero initialization of the weights

OPTIMIZATION

the **learning rate** and batch size are more-or-less the cardinal hyperparameters

- choose the learning rate on a small set (see Bottou)
- simplify your life and **use a constant rate**, that is, no schedule until everything else is figured out
- schedule according to epochs (== number of passes through the data), not number of iterations

OPTIMIZATION

bias learning in the right direction by setting output biases

- set to mean of targets for regression

- set to pr(positive) for imbalanced binary classification problem as seen in object detection (RetinaNet) and contour detection

OPTIMIZATION

clear gradients after each iteration or else they accumulate

remember `opt.zero_grad()`!

remarkably, with adaptive optimizers and enough time a model can still learn if the gradients are never cleared out… but it's really sensitive

OPTIMIZATION

accumulate gradients to decouple learning batch size from computational batch size

```
while iteration < max_iter:
    epoch += 1
    for data, target in loader:
        data = data.to(device)
        target = target.to(device)
        out = model(im)
        loss = loss_fn(out, target)
        loss.backward()
        grad_counter += 1
        if grad_counter == args.iter_size:
            opt.step()
            opt.zero_grad()
            iteration += 1
            grad_counter = 0
```

warning! watch out for normalization at the computational batch size

OPTIMIZATION

checkpoint features + gradients to trade space for time and fit large models

- checkpoint utility in PyTorch
- systems paper on checkpointing for deep nets
- reversible nets are a special architecture for decoupling memory from depth

note: this applies equally well to convolutional/recurrent/whatever nets

OPTIMIZATION

live on the edge and try extreme settings (but just a little bit)

> If optimization never diverges, your learning rate is too low.

in the style of the *Umeshism*

> If you’ve never missed a flight, you’re spending too much time in airports.

OPTIMIZATION

check for **accidental non-differentiability** from max/min/etc.

clipping the output or the loss is not the same as clipping the gradients!

```
y_hat = torch.min(y_hat, 0.5)
```

discards the gradient w.r.t. the parameters for y_hat > 0.5

OPTIMIZATION

check the sign of the loss

- surely ascent is not descent, but that can be hard to remember in the moment
- definitely for custom losses! the defaults were checked for you, not so your own

once asked a generative model to minimize the probability of the data,
and it sure did

EVALUATION

switch to **evaluation mode** by `model.eval()`

no, really

and check the mode by `model.training`

EVALUATION

know the output! metrics and summaries can obscure all kinds of issues

- inspect input-output pairs
  across min/max/quartiles
  of the loss and other metrics of interest

- keep an eye on the output and loss for a chosen set
  across iterations to have a sense of the learning dynamics
  (the chosen set could be the whole validation set)

- doesn't hurt to eyeball a few subsets chosen at random
  in case you catch anything surprising

EVALUATION

separate evaluation from optimization!

- save checkpoints at intervals and evaluate them offline

- there are the perils of nondeterminism, batch norm, etc. when mixing training and testing

- plus it only slows down training

EVALUATION

evaluate on the whole set: batch-wise statistics, even if smoothed, are too noisy

- if this is slow, then it's all the more reason to decouple evaluation from training

- if the evaluation is truly massive, consider a miniature set for routine evaluation and only evaluate on the full set at longer intervals

TUNING

try the scientific method
change one thing at a time

not the computer scientific method
change everything every time

...except perhaps when you really have no idea what's going on

(joke courtesy Dave Patterson)

TUNING

random search over grid search

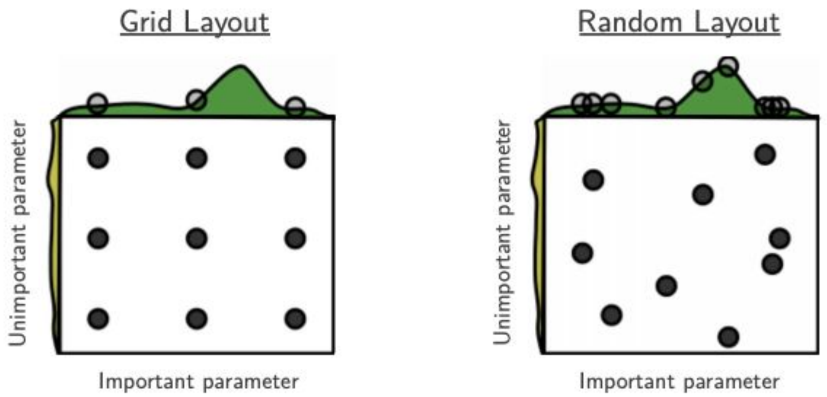


Figure 1: Grid and random search of nine trials for optimizing a function $f(x,y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.

Bergstra & Bengio, JMLR 2012.

TUNING

Coarse-to-fine search for hyperparameters

- wider search for shorter training
  (on the same data: remember to set the seed!)

- narrower search for longer training

- but be careful with greed! still need to tune for full training,
  because the best hyperparameters for short runs may not prevail in the end

EXECUTION

don't be finger-bound! script the optimization + evaluation of your models

every character you type is a chance to make a mistake

EXECUTION

log everything! especially arguments

```
import logging
def setup_logging(logfile):
   FORMAT = '[%(asctime)s.%(msecs)03d] %(message)s'
   DATEFMT = '%Y-%m-%d %H:%M:%S'
   logging.root.handlers = []  # clear logging from elsewhere
   logging.basicConfig(level=logging.INFO, format=FORMAT, datefmt=DATEFMT,
           handlers=[
               logging.FileHandler(logfile),
               logging.StreamHandler(sys.stdout)
           ])
   logger = logging.getLogger()
   return logger
```

```
args = parser.parse_args()
log = setup_logging('log')
log.info(f"args: {vars(args)}")
```

EXECUTION

check versioning status and restrain yourself from reckless changes

```
import subprocess
# record version for reproducibility
try:
   version = subprocess.check_output(
             ['git', 'describe', '--always']).decode().strip()
   log.info(f"version: {version}")
   is_dirty = subprocess.check_output(
              ['git', 'status', '--short']).decode()
   if is_dirty:
       log.warning("Working directory is dirty, and "
                   "might not be reproducible!")
except:
   log.warning("Could not determine version without git.")
```

if it's worth running the experiment, it's worth committing the conditions

EXECUTION

there's no writing like rewriting, and no coding like **debugging**

thankfully with PyTorch we're hacking imperatively in Python!
we can debug as usual with **pdb**

```
import pdb; pdb.set_trace()
```

https://docs.python.org/3/library/pdb.html
https://www.digitalocean.com/community/tutorials/how-to-use-the-python-debugger

TESTING

test correctness or settle for incorrectness, as there's very little middle ground

- anything untested might be wrong (if not now, then later)

- rely on a separate, simpler implementation as a reference
  like this convolution by loops

Caffe has hundreds of unit tests for a reason:
every one is a guard—but not guarantee—against error

TESTING

check gradients and do it thoroughly

- gradcheck is the checker bundled into PyTorch
- cs231n has gradient checking rules of thumb
- Tim Vieira has further tips for accurate and thorough checking

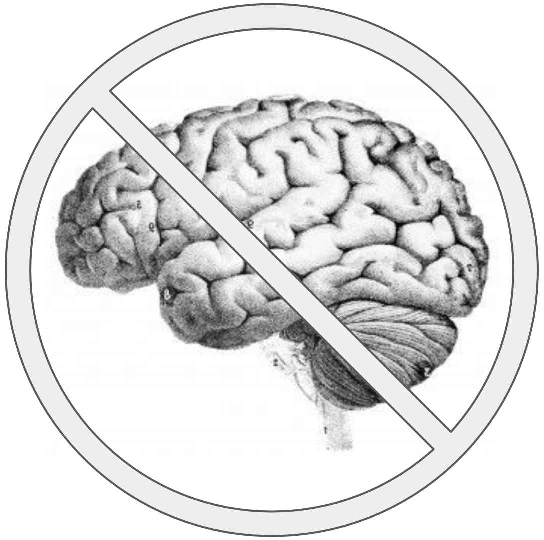
COMPUTATION

more hardware, more problems don't parallelize immediately

- make your model work on a single device first
- attempt to parallelize on a single machine
- only then go to a multi machine setup
- and check that iterations/time actually improves!

see Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour for good advice

HYPE



Not So "Neural"
these models are not how the brain works

we don't know how the brain works!

- this isn't a problem (save for neuroscientists)

- be wary of neural realism hype or "it just works because it's like the brain"

- **network**, not neural network
  unit, not neuron

So, is Deep Learning a Piece of Cake?

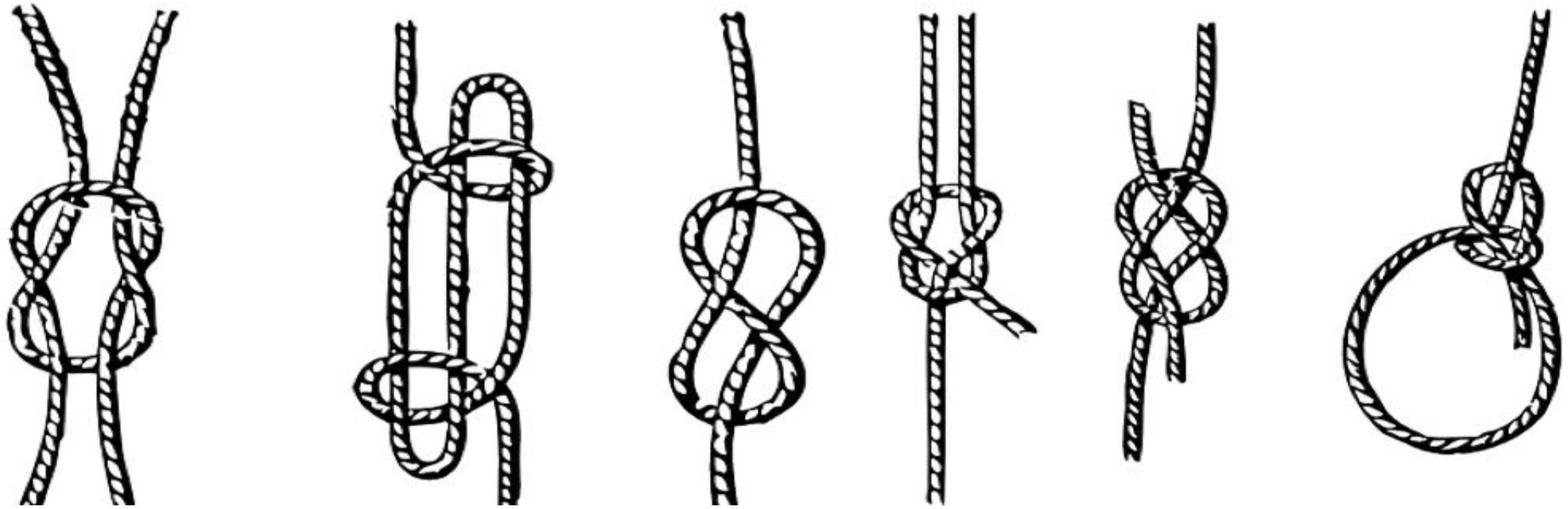
Not quite.

The tools are better every day, but tools are no substitute for thought.

While there are many layers, hopefully you're now more ready to cut through the complexity.

Next time you bake, you could try Andrej Karpathy's recipe:
http://karpathy.github.io/2019/04/25/recipe/

Thanks! Questions?



Good luck with your nets!