Lecture 8: Hacker’s guide to DL

Speaker: Phillip Isola
8. Hacker’s guide to DL

• Data

• Model

Lots of slides adapted from Evan Shelhamer’s “DIY Deep Learning: Advice on Weaving Nets”

• Optimization

• Evaluation, Experimentation, and Debugging

Further acknowledgements:
Builds on advice from Andrej Karpathy (http://karpathy.github.io/2019/04/25/recipe/), feedback from Isolab members and MIT community, slides from Dylan Hadfield-Menell, twitter feedback (https://twitter.com/phillip_isola/status/1576965425384263680?s=20&t=3eLg6JBYVSkacUtNlz83pA)
Data

Machine learning == Programming via data

# training data
input1 = [5,2,1,3,4]; output1 = [1,2,3,4,5]
input2 = [1,8,4]; output2 = [1,4,8]
input3 = [6,5,4,3]; output3 = [3,4,5,6]
...

Compiler:

Program:
QuickSort

Compiler:

void quickSort(int input[], l, h) {
  if l < h {
    int pi = partition(input, l, h)
    ...}
}

Compiler:

# training data
input1 = [5,2,1,3,4]; output1 = [1,2,3,4,5]
input2 = [1,8,4]; output2 = [1,4,8]
input3 = [6,5,4,3]; output3 = [3,4,5,6]
...
“Become friends with every pixel”
“Become friends with every pixel”

Look at your data
Look at the input

Look at the output

DeGrave, Janizek, Lee, 2020
Data

Look at the data!

inspect the distribution of 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

[slide adapted from Evan Shelhamer]
Data

Inspect the inliers, outliers, and neighbors:

• visualize distribution and data, especially **outliers**, to uncover dataset issues

• look at **nearest neighbors**

• examples:

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

[slide adapted from Evan Shelhamer]
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 $\text{output} = \text{model}(\text{data})$

[slide adapted from Evan Shelhamer]
Data

pre-processing:

• **standardize:**

\[
x_k \leftarrow \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}} \quad \forall k
\]

• Squashes all your data dimensions into the same standard range

• This makes it so that, a priori, no one dimension is valued more than any other

• Important when different measurements have vastly different scales or units
Data

pre-processing:

- **summary statistics**: check the min/max and mean/variance to catch mistakes like loading values in the range \([0,255]\) when the model expects values in the range \([0,1]\).

- **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?

[slide adapted from Evan Shelhamer]
A lot of your code will just be reshaping tensors

What does `reshape(X, (X.shape[0]*X.shape[1]))` do? Is it column order or row order?

Tools like einops can make it much easier to avoid mistakes

https://github.com/arogozhnikov/einops/tree/master/docs

```python
# or compose a new dimension of batch and width
rearrange(ims, 'b h w c -> h (b w) c')
```
Data augmentation

Training data

\( x \), \( y \)

- “Fish”
- “Grizzly”
- “Chameleon”

\( \vdots \)

- “Fish”
- “Fish”
- “Fish”
- “Fish”

- Mirror
- Crop
- Crop
- Darken
Idea: Train on randomly perturbed data, so that test set just looks like another random perturbation.

This is called **domain randomization** or **data augmentation**.
Domain randomization

Training data

Test data

[Sadeghi & Levine 2016]

Above example is from [Tobin, Fong, Ray et al. 2017]
Domain gap between $p_{\text{source}}$ and $p_{\text{target}}$ will cause us to fail to generalize.
High train accuracy can mean problem is too easy

Add more data to make problem harder
Data-Centric AI Competition

- Competition over the summer to improve a baseline model through data manipulations
- Submissions were due Sept. 4, 2021
- Goal to classify handwritten Roman numerals

Leaderboard

<table>
<thead>
<tr>
<th>Rank</th>
<th>Submission</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Iter3_002</td>
<td>0.85826</td>
</tr>
<tr>
<td></td>
<td>Divakar Roy</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>baseline-cleaned-Naugmented</td>
<td>0.85744</td>
</tr>
<tr>
<td></td>
<td>Innotescus</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>syn-ann-sub_24</td>
<td>0.85455</td>
</tr>
<tr>
<td></td>
<td>Synaptic-AnN</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>syn-ann-sub-21-gray</td>
<td>0.85083</td>
</tr>
<tr>
<td></td>
<td>Synaptic-AnN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jktubs_98_20_10_EDGE</td>
<td>0.84959</td>
</tr>
<tr>
<td></td>
<td>Jens Kramer</td>
<td></td>
</tr>
</tbody>
</table>

Model Keep it as simple as possible!

do your first experiment with the simplest possible model w/ and w/o your idea

Why keep it simple?

• easy to build, debug, share

• tractable to understand, make robust, build theories around

• *simple models also work better* (Occam’s razor, Solomonoff Induction)

• if you focus on simplicity you will have an unfair advantage
Model

start with a standard and popular model (popularity matters more than performance)

if you have an image classification problem, you might try:

```python
model = torch.hub.load('pytorch/vision:v0.9.0', 'resnet18')
```

if you have text problem, you might try:

```python
>>> from transformers import pipeline, set_seed
>>> generator = pipeline('text-generation', model='gpt2')
```

find popular models and code here: https://paperswithcode.com/
Model

stand on the shoulders of giants

use pretrained models

(but be aware of their flaws and limitations)
double-check the model actually is what you thought you defined

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

[slide adapted from Evan Shelhamer]
Transform your problem into a "solved" problem

- Case study: transforming image colorization to image classification

[c.f. the strategy of "polynomial-time reduction"]
Image colorization

Input \( \mathbf{x} \) \quad \begin{array}{c}
\text{Training data} \\
\{ \mathbf{x}, \mathbf{y} \}
\end{array} \quad \begin{array}{c}
\{ \mathbf{x}, \mathbf{y} \}
\end{array} \quad \begin{array}{c}
\{ \mathbf{x}, \mathbf{y} \}
\end{array} \quad \vdots
\end{array}
\quad \quad \quad \quad \quad \begin{array}{c}
Output \mathbf{y}
\end{array}

\arg \min_{f \in \mathcal{F}} \sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))

[Zhang, Isola, Efros, ECCV 2016]
Grayscale image: $\mathbf{L}$ channel

$x \in \mathbb{R}^{H \times W \times 1}$

Color information: $ab$ channels

$y \in \mathbb{R}^{H \times W \times 2}$

[Zhang, Isola, Efros, ECCV 2016]
Grayscale image: \textbf{L channel}

\[ \mathbf{x} \in \mathbb{R}^{H \times W \times 1} \]

Color information: \textbf{ab channels}

\[ \mathbf{y} \in \mathbb{R}^{H \times W \times 2} \]

[Zhang, Isola, Efros, ECCV 2016]
Colorization $\rightarrow$ Classification
**Colors $\rightarrow$ Classes**

\[ y \in \mathbb{R}^{H \times W \times 2} \]

One-hot representation of $K$ discrete classes

\[ y \in \mathbb{R}^{H \times W \times K} \]

One hot codes:

- Blue $\rightarrow [0,0,1, \ldots]$
- Yellow $\rightarrow [1,0,0, \ldots]$
- Pink $\rightarrow [0,1,0, \ldots]$
rockfish
Image classification → Pixel classification

→  

yellow
Image classification $\rightarrow$ Pixel classification

yellow
Image classification → Pixel classification
Model

• Formulate your problem as softmax regression (a.k.a. classification)

  • cross-entropy loss, 1-hot labels

• Why?

  1. No restriction on shape of predictive distribution [up to quantization] (this is not the case for least-squares regression, which assumes Gaussian predictions)

  2. Discrete classes are easy to label

  3. All labels are equidistant under 1-hot representation
Recipe for deep learning in a new domain

1. Transform your data into numbers (one-hot vectors)

2. Transform your goal into an numerical measure (cross-entropy loss)

3. Use a generic optimizer (Adam) and an standard architecture (transformer) to solve the learning problem
Don’t use batch norm

- Introducing a strong dependency on batch size (now batch size becomes an even more critical hyperparameter)

- Different behavior at train and test time

- Makes distributed computing hard — requires communication between all elements in a batch

- Use layer norm instead
(Disclaimer that this is my personal opinion)

Batchnorm can be a useful tool, but in my experience it's usually more trouble than it's worth. Below are some things that make working with batchnorm a headache. You can work around all these issues... or you can just not use batchnorm 😞.

1. Behavior at train time and test time is different. Forgot to set model.eval()? You will have a bug. More generally, differences in train vs test behavior make it harder to anticipate test behavior from training behavior. You might be in for surprises.

2. It only works if batch size is sufficiently big. Suppose your batch size is 1. Then if you have an activation vector \( z \) and subtract the mean over the batch, you get \( z - \bar{z} = 0 \). The model just zeroed out the activation vector and your net won’t work. Worse, the variance is undefined for batch size 1 and that could cause bugs too. For small batch size the variance could be a very poor estimate of the true variance of the activations and cause numerical and optimization issues. I had this bug in the original version of the pix2pix paper, and it made the baseline work worse than it should have (which might not have been a bad thing for making the paper popular...). See change log in appendix here: [https://arxiv.org/abs/1611.07004](https://arxiv.org/abs/1611.07004)

3. Suppose your batch size is large, but all activations in a batch happen to have the same value. Again the variance is undefined and the activations get zeroed out by subtracting the mean. [This was the “bug” that the SPADE paper tried to fix: [https://arxiv.org/abs/1903.07291](https://arxiv.org/abs/1903.07291)]

4. Want to train a really big model in parallel across many machines? With SGD and no batchnorm you can run forward and back propagation independently on subbatches distributed across the machines, then only communicate between them once to aggregate the gradients. With batchnorm, the activations for one subbatch will depend on the activations for another subbatch – you need to communicate all these activations between the machines. Naively you need to do this after each layer of your network, to compute the inputs to the next layer, which is a huge communication overhead, but in practice people use various (bug-prone) speedups.

5. More generally, batchnorm introduces a dependency between different elements of your batch. Different training examples are not processed iid. This not only gives implementation headaches but also makes the theoretical analysis harder. This might be one reason why the theory of why batchnorm works is still not really resolved.

6. There are lots of tricks and hacks to fix the above issues – these tricks add more potential bugs and complexity. Sometimes the fix is worse than the original problem. In distributed training sometimes people run batchnorm independently on each machine. That fixes the communication problem, but now you have a maybe worse problem: your results change dramatically depending on how many machines are in your cluster (since the number of machines determines the size of the subbatches when you distribute across machines and subbatch size has a big effect on how well batchnorm works).

Feel free to add to this list if anyone has more – I think the list could go on and on.
Remember that often the easiest way to get better performance is:

1) **Scale** your data: more (diverse) training examples

2) **Scale** your model: more layers, more channels

3) **Scale** your compute: train for longer

In the current era, I would say these are the top three factors that determine success, roughly in order

... but working at small scale forces efficiency, and then when you do scale up, you get more bang for your buck
Once you get your system working, you are only halfway done

Second half is to remove everything nonessential

“Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away”
— Antoine de Saint Exupéry
Optimization

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

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

first make sure you can fit train set, then consider generalization to test set

[slide adapted from Evan Shelhamer]
Optimization

**sanity check the loss** against a suitable reference value

- classification with cross-entropy loss: uniform distribution
- get to know log loss numbers:
  - \(-0.69 = \ln(0.5)\) [chance on binary classification]
  - \(-2.3 = \ln(0.1)\) [chance on 10-way classification]
- 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

[slide adapted from Evan Shelhamer]
Optimization

Most important hyperparameters: **learning rate** and **batch size**

- first, **use a constant rate**; don’t schedule until everything else is figured out

- schedule according to number of iterations of SGD, not epochs

- use biggest batch size that will fit in memory

- always retune lr when *anything* changes in your model (most changes to model change scale of gradients, which changes the **effective lr**)

  may look like your model is training much faster, but really you just scaled the effective lr
Optimization

Remove the concept of “epochs” from your brain

• There are no epochs in the wild
• Don’t tie lr schedule to epochs!
  • makes it hard to compare learning curves between experiments (huge waste of time/money in modern research)
  • be careful with cosine lr (looks like it is converging when it is not)

<table>
<thead>
<tr>
<th>Method</th>
<th>Architecture</th>
<th>Param.</th>
<th>Head</th>
<th>Epochs</th>
<th>Top-1</th>
<th>Top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstDis [73]</td>
<td>ResNet-50</td>
<td>24</td>
<td>Linear</td>
<td>200</td>
<td>56.5</td>
<td>-</td>
</tr>
<tr>
<td>Local Agg. [83]</td>
<td>ResNet-50</td>
<td>24</td>
<td>Linear</td>
<td>200</td>
<td>58.8</td>
<td>-</td>
</tr>
<tr>
<td>CMC [66]</td>
<td>ResNet-50*</td>
<td>12</td>
<td>Linear</td>
<td>240</td>
<td>60.0</td>
<td>82.3</td>
</tr>
<tr>
<td>MoCo [28]</td>
<td>ResNet-50</td>
<td>24</td>
<td>Linear</td>
<td>200</td>
<td>60.6</td>
<td>-</td>
</tr>
<tr>
<td>PIRL [49]</td>
<td>ResNet-50</td>
<td>24</td>
<td>Linear</td>
<td>800</td>
<td>63.6</td>
<td>-</td>
</tr>
<tr>
<td>CPC v2 [31]</td>
<td>ResNet-50</td>
<td>24</td>
<td>-</td>
<td>-</td>
<td>63.8</td>
<td>85.3</td>
</tr>
<tr>
<td>SimCLR [10]</td>
<td>ResNet-50</td>
<td>24</td>
<td>MLP</td>
<td>1000</td>
<td>69.3</td>
<td>89.0</td>
</tr>
<tr>
<td>InfoMin Aug. (Ours)</td>
<td>ResNet-50</td>
<td>24</td>
<td>MLP</td>
<td>200</td>
<td>70.1</td>
<td>89.4</td>
</tr>
<tr>
<td>InfoMin Aug. (Ours)</td>
<td>ResNet-50</td>
<td>24</td>
<td>MLP</td>
<td>800</td>
<td><strong>73.0</strong></td>
<td><strong>91.1</strong></td>
</tr>
</tbody>
</table>

[Tian, Sun, Poole, et al., 2020]
Optimization

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

- can then accumulate gradients across checkpoints
- can resume training if your computer crashes
- have a “paper trail” to debug later

[slide adapted from Evan Shelhamer]
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 *Umeshism*

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

[slide adapted from Evan Shelhamer]
Use exponentially moving averages (EMA)

- Replace a quantity with a weighted average of its previous values, with weight exponentially decaying over time

- Useful for many quantities in deep learning, including gradients (where it is known as momentum), weights, data, activations, targets, etc.

- (Basically for any variable in DL, try replacing it with its EMA version and it may be better)

\[
\theta_{EMA}^t \leftarrow \beta \theta_{EMA}^{t-1} + (1 - \beta) \theta^t-1
\]
Optimization

optimizers:

• Adam (or AdamW) is good for prototyping (generally just works)

• SGD may be slightly better for performance (but requires more tuning of hyperparameters)


• Clip gradients to improve stability
Evaluation

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

no, really

and check the mode by model.training

[slide adapted from Evan Shelhamer]
Evaluation

Look at the output

Loss

epoch
Evaluation

**Iterate** on metrics and evaluation

- Visualize your data and predictions
- Describe the predictions and associated issues, one per item
- Group issues together into categories/failure modes
- Instrument your code to track the categories (if you can)

[slide adapted from Dylan Hadfield-Menell]
Tensorboard and WandB are your friends!

- When in doubt, log it
- If you’re logging it, make it easy to see the results

[slide adapted from Dylan Hadfield-Menell]
The developer-first MLOps platform
Build better models faster with experiment tracking, dataset versioning, and model management

SIGN UP
REQUEST DEMO

https://wandb.ai/site
Tuning to tune hyperparameters (architectural design choices, optimizer parameters, etc) **random search** may be better than grid search.

Figure 1: Grid and random search of nine trials for optimizing a function $f(x,y) = g(x) + h(y)$ 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.

[https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf]  [slide adapted from Evan Shelhamer]
Tuning

Cayenne pepper is all you need?

No! Each spice has its use. But combination matters. And don’t over spice.
Experimentation and debugging

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

every character you type is a chance to make a mistake

also scripting makes the work reproducible!

use config files (e.g., yaml) to manage experiments; log all arguments

[slide adapted from Evan Shelhamer]
Experimentation and debugging

debug with the default python debugger: **pdb**

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


[slide adapted from Evan Shelhamer]
Compute

**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 set
- and check that iterations/time actually improves

see [Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour](https://arxiv.org/abs/1512.03385) for good advice
8. Hacker’s guide to DL

• Data
• Model
• Optimization
• Evaluation, Experimentation, and Debugging

Lots of slides adapted from Evan Shelhamer’s “DIY Deep Learning: Advice on Weaving Nets”

Further acknowledgements:
Builds on advice from Andrej Karpathy (http://karpathy.github.io/2019/04/25/recipe/), feedback from Isolab members and MIT community, slides from Dylan Hadfield-Menell, twitter feedback (https://twitter.com/phillip_isola/status/1576965425384263680?s=20&t=3eLg6JBYVSkacUtNlz83pA)