Day4 - Debugging ML code

02476 Machine Learning Operations

Nicki Skafte Detlefsen, Associate Professor, DTU Compute

January 2026

Debugging is a hard but necessary discipline

Debugging ML code is even harder

Bugs in ML code can be non-ml specific and ML specific

🐛 Classic bugs: Code does not run

Use traditional debugger to find these

🐛 ML specific bugs: My model is not converging

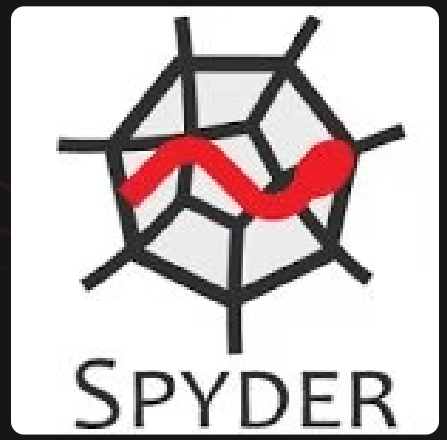
Need the correct approach for debugging

Lets start with the classics...

First step: Get a good working envionment

💡Jupyter notebooks are great at what they are meant for: exploring ideas and combining code + text into standalone document...

💡However, it can be a pain debugging code in notebooks...



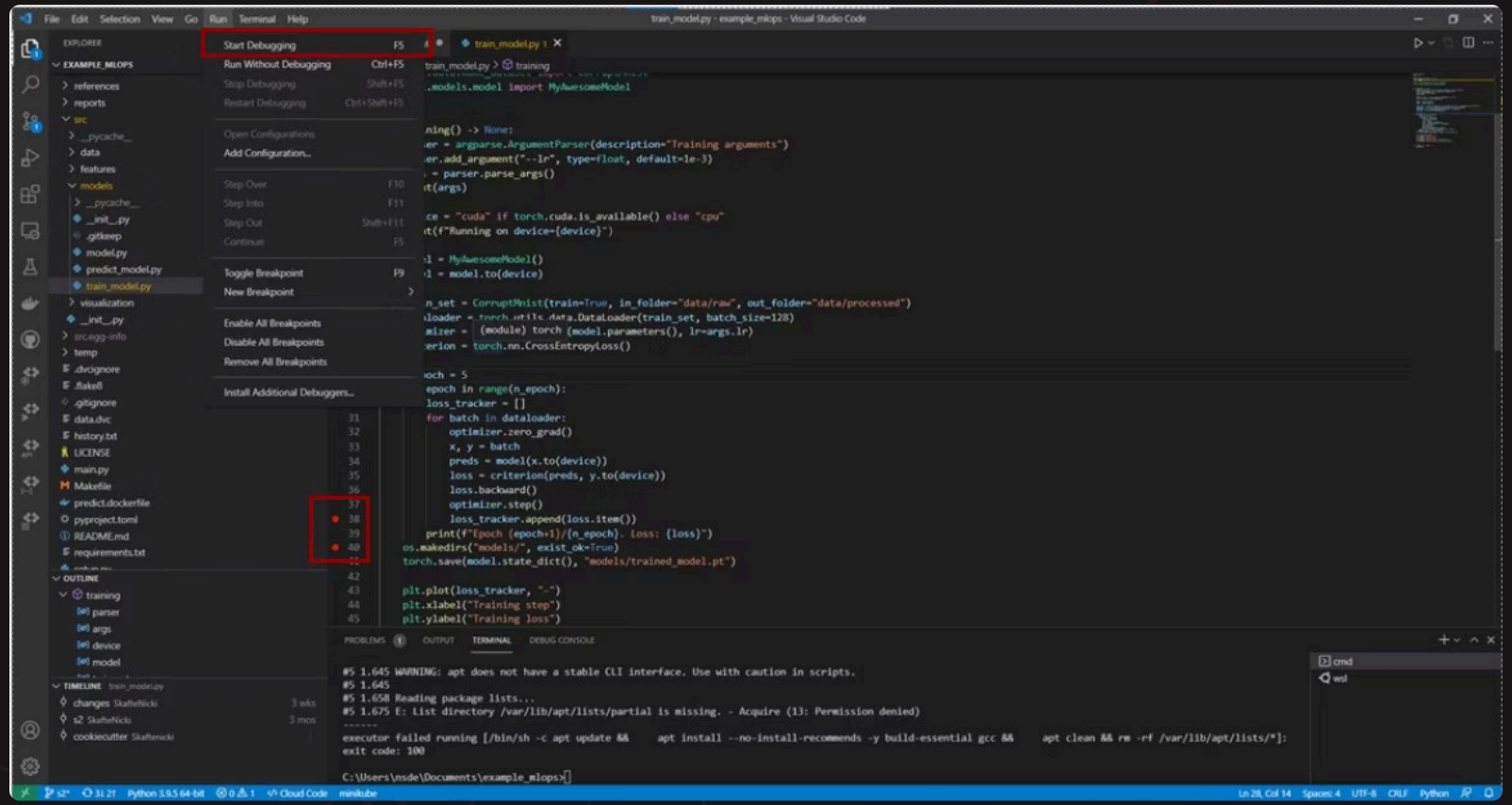
Python debugging in general

Print statements	`print("x.shape = {}".format(x.shape))`

Stop at interesting points in your code and interact	`from IPython import embed; embed()`
	`... # do your stuff interactively here`
	`exit()  # exit ipython to let your code continue`

Use build in Python debugger	`import pdb; pdb.set_trace()`
Use breakpoints	

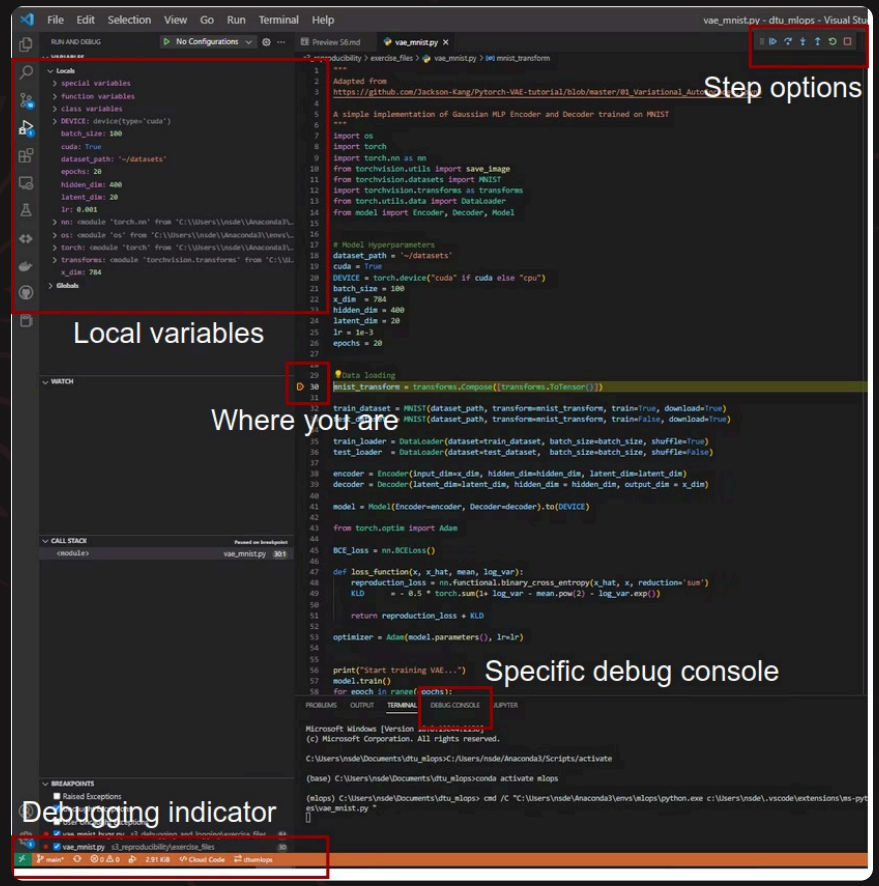
VS code debugging



VS code debugger

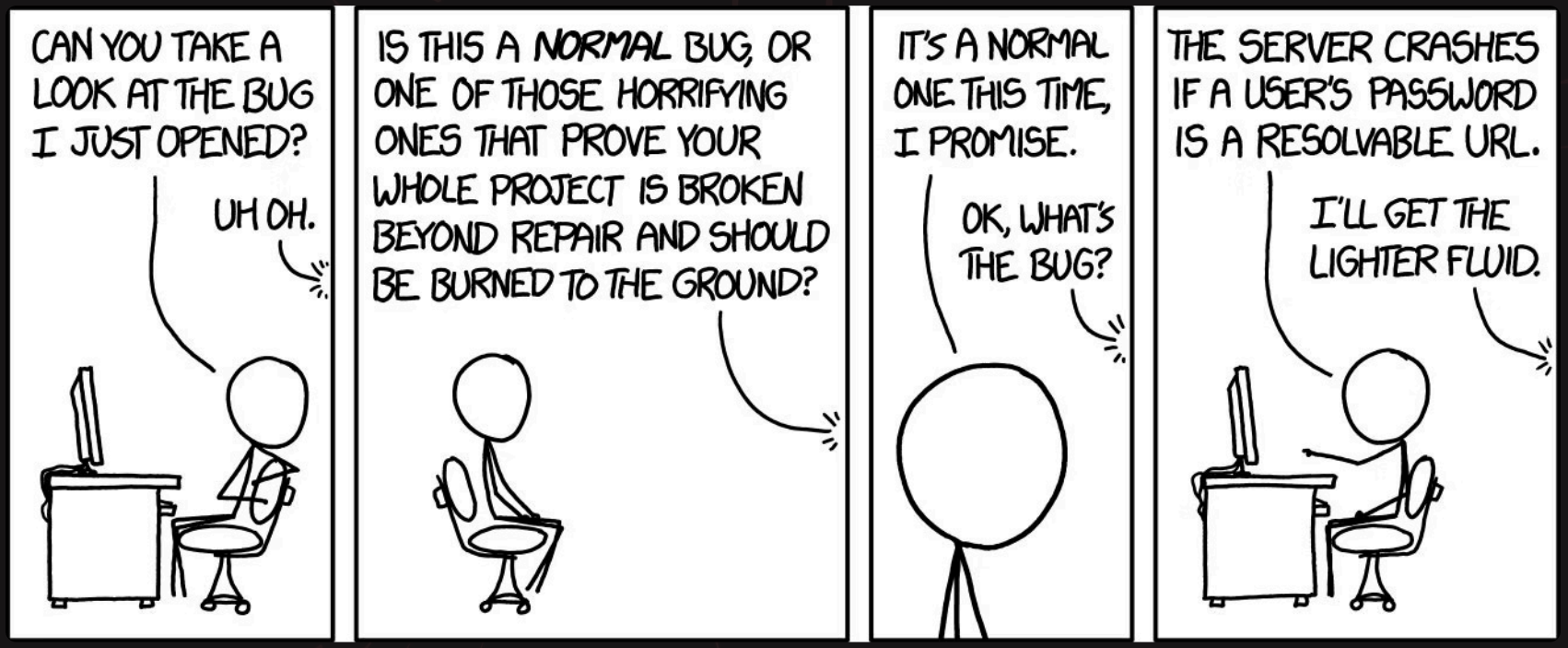
Step options:

- 💻F5: next breakpoint
- 💻F10: next line
- 💻F11: step into
- 💻Shift-F11: step out
- 💻CTRL-Shift-F11: Restart
- 💻Shift-F5: stop



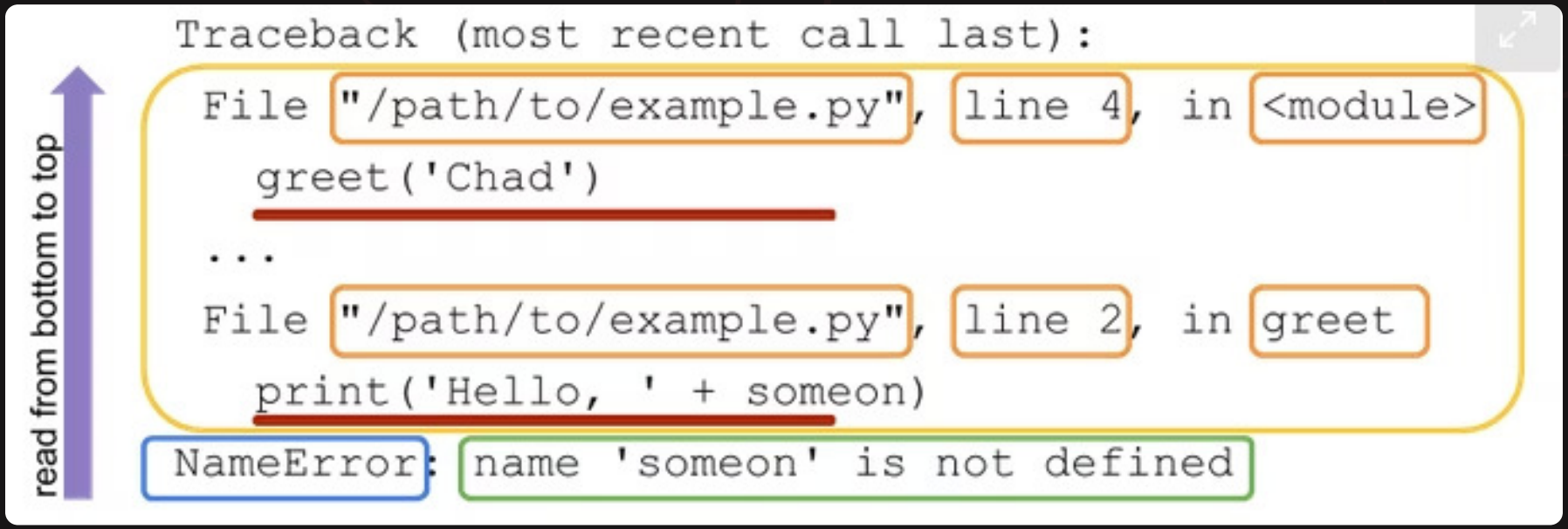
Back to these ML specific bugs

⚠️When everything is running, but results are wrong⚠️ = Silent Failures



Finding these buggers comes with experience. A potpourri of my findings over the last couple of years and **others**.

1. Read the freaking stack traces



Sometimes the real error is in the beginning, sometimes in the middle but in most cases it is the end line.

2. Check your data

Your starting point should always be the data in ML!

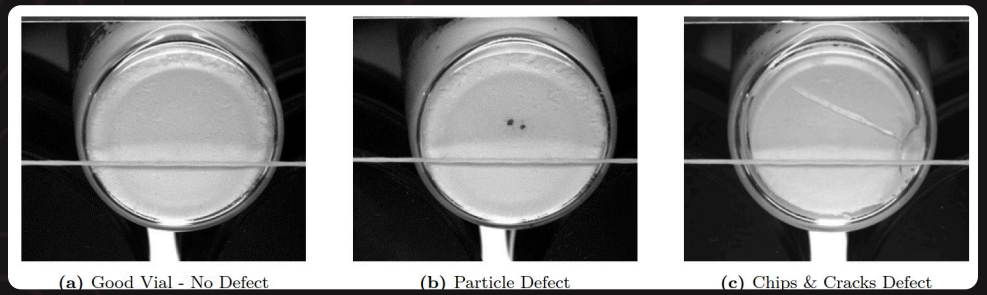
Check

✅Examine summary statistics (data normalized?)

✅Look at label distributions (is it shuffled?)

✅Visualize a few samples (are they corrupted in anyway?)

If you are working on datasets you know, you may skip these steps.



(a) Good Vial - No Defect (b) Particle Defect (c) Chips & Cracks Defect

Data was manually collected by introducing particles into vials and then running them through the system

> What are the potential problem with automatic labeling of batches?

3. Look out for data leakage

Data leakage occurs when information from val or test is used to create the model

💡 Normalization trap: calculating mean/std on the entire dataset before splitting (includes batch normalization)

💡 Time-series leakage: using future to predict past

💡 Duplicate samples: having the same sample in both training and val/test

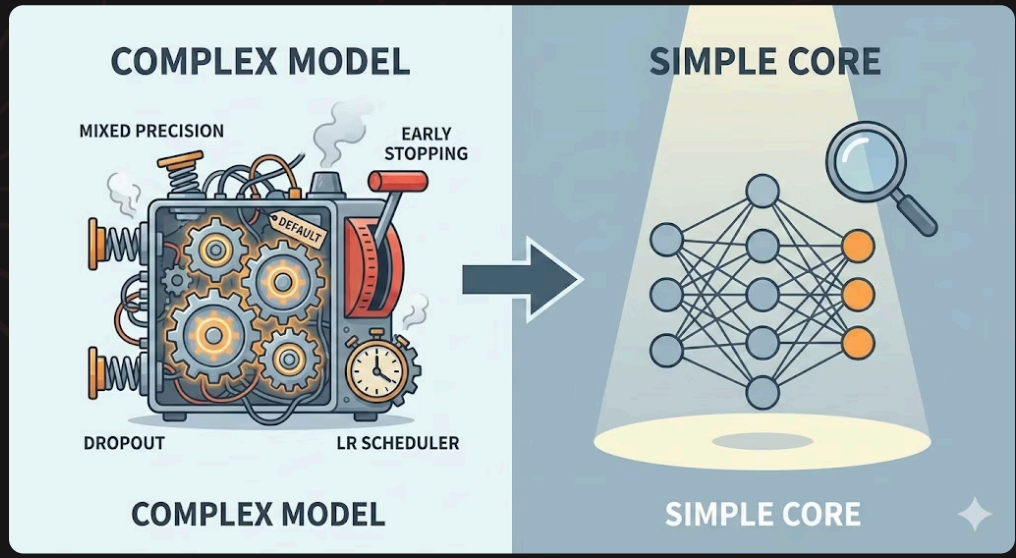
💡 Group leakage: multiple samples from same user, some in training, some in val/test

4. Start as simple as you can

Remove all the fancy stuff

- Mixed precision
- Regularization like dropout
- Early stopping
- Learning rate schedulers
- ...

One or more of these may be enabled by default if you are starting from someone's else codebase



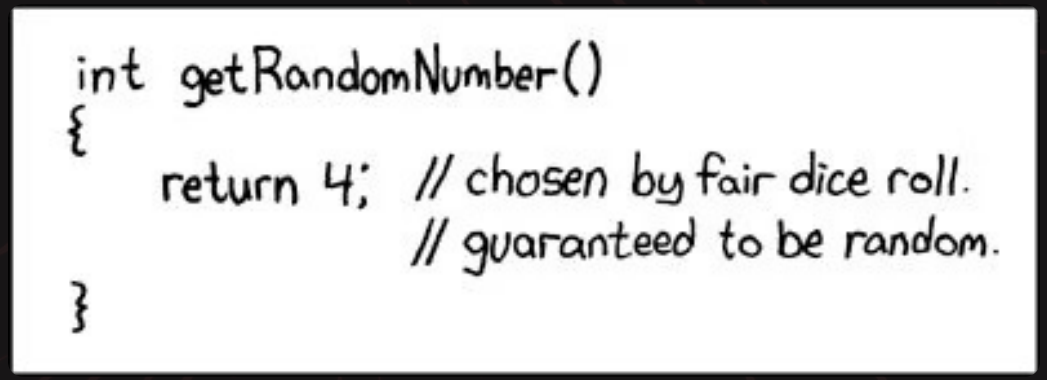
5. Make everything deterministic

Every machine learning run is by default random. Try fixing:

💡 Seed everything and use same seed everywhere

💡 Remove all data augmentation

💡 Use only a single batch of data where you have a feeling of the outcome



6. Investigate the math

Debug the math:

💡Go through your code, line by line

💡There should be a one-to-one match between equations and lines of code

💡Refactor if it is not clear

Check dimensions and annotate if necessary or use **typing software**

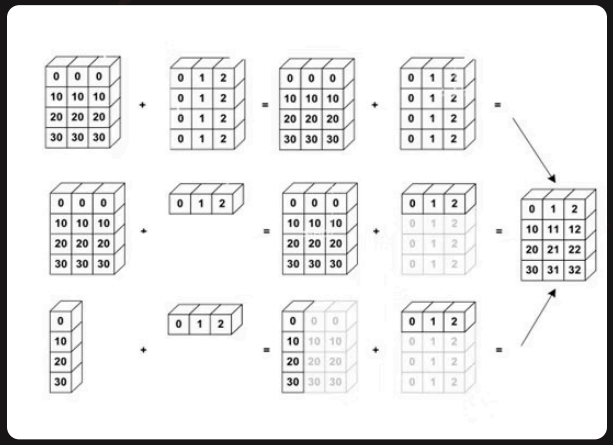
```
# add shape comments
A = torch.randn(N, D)  # NxD
x = torch.randn(D)  # D
Ax = A.mv(x)  # N
```

6. Investigate the math

💡 Lookout for broadcasting!

💡 Broadcasting in python is both a blessing and a curse

💡 It can create problems (real life example)



```
import torch
preds = torch.randn(100,)
target = torch.randn(100,1)
loss = (preds - target).abs().pow(2.0).sum()
```

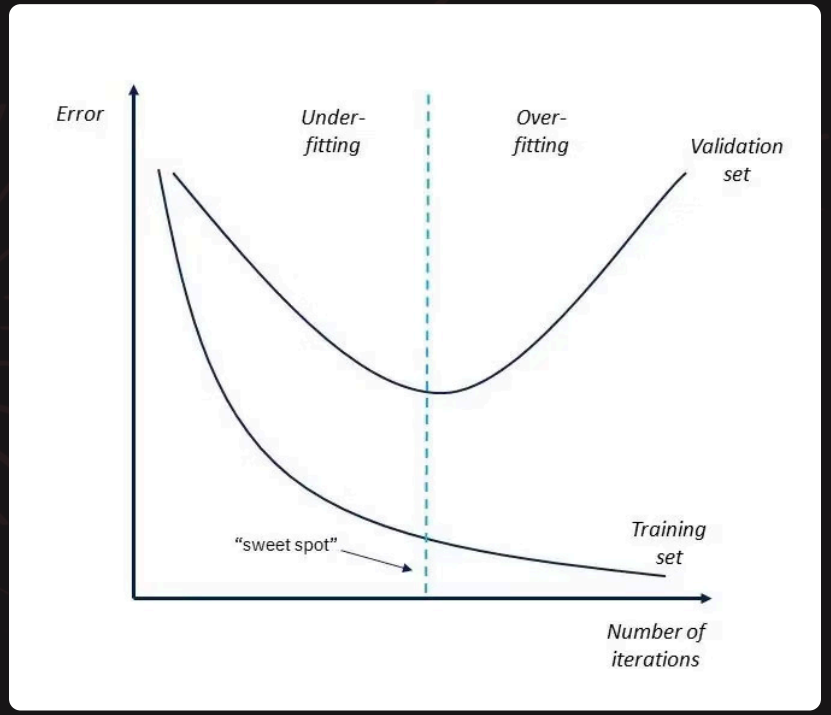
What is the problem here?

7. Overfitting is good?

Overfitting is usually seen as an bag thing bug...

💡Models should be able to memorize one batch

💡Train on one batch, if loss is 0 move on to larger models/large data else debug



8. Really look at your loss

Assuming your model is training without errors, but your loss is not behaving as it should.

💡 Are you printing/logging the results correctly?

💡 Did you remember loss.backward, optimizer.step and optimizer.zero_grad ?

💡 What about your learning rate and batch size?

For your loss, if possible, calculate in log-space

```
a = 1
for x in data:
  a = a * x

log_a = 0
for x in data:
  log_a = log_a + log(x)
```

9. Visualizations are your friend

Log and visualize everything

💡 Training loss is really decreasing right?

💡 Can you add additional metrics?

💡 Log dynamic changing hyperparameters (learning rate with lr schedulers)

💡 Plot data, predictions, reconstructions etc. over time

10. Add complexity over time

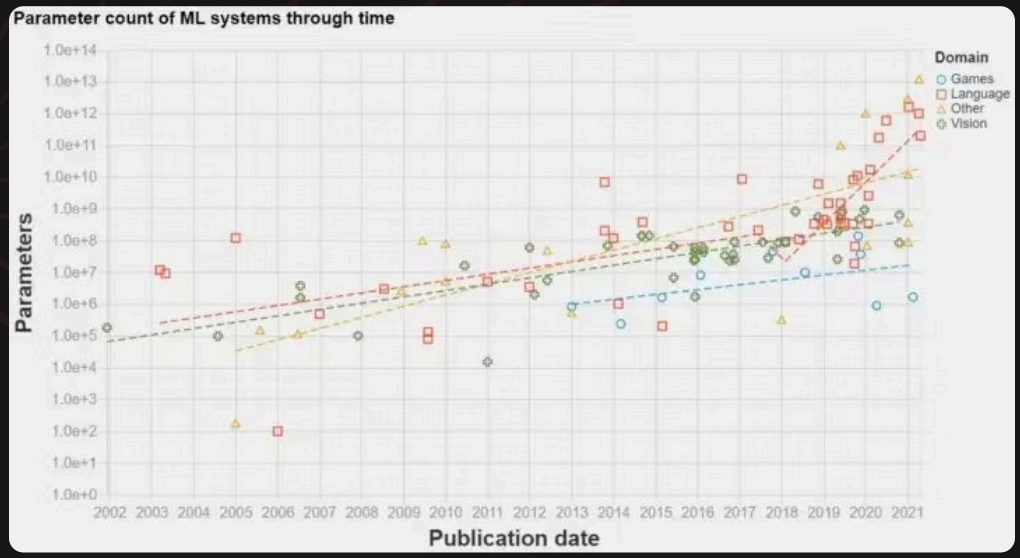
If you are still good, then

💡Get a baseline that just works (=train on more data data)

💡Stop overfitting:

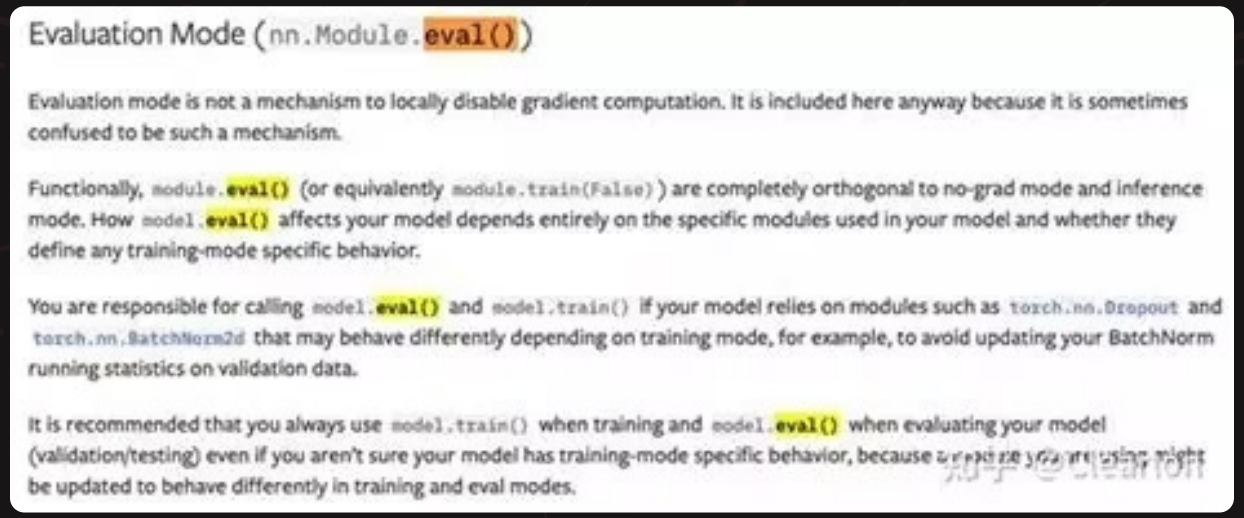
- Add back regularization
- Add back data augmentations

💡Tune hyper parameters



11. Model mode

Enabling `model.eval` vs. `model.train` at the right time



Evaluation Mode (`nn.Module.eval()`)

Evaluation mode is not a mechanism to locally disable gradient computation. It is included here anyway because it is sometimes confused to be such a mechanism.

Functionally, `module.eval()` (or equivalently `module.train(False)`) are completely orthogonal to no-grad mode and inference mode. How `model.eval()` affects your model depends entirely on the specific modules used in your model and whether they define any training-mode specific behavior.

You are responsible for calling `model.eval()` and `model.train()` if your model relies on modules such as `torch.nn.Dropout` and `torch.nn.BatchNorm2d` that may behave differently depending on training mode, for example, to avoid updating your BatchNorm running statistics on validation data.

It is recommended that you always use `model.train()` when training and `model.eval()` when evaluating your model (validation/testing) even if you aren't sure your model has training-mode specific behavior, because a module you are using might be updated to behave differently in training and eval modes.

Summary of steps in ML debugging

1. Read your stacktrace
2. Check your data
3. Check for data leakage
4. Start as simple as you can
5. Make everything deterministic
6. Investigate the math
7. Overfit to your data
8. Look at your loss
9. Visualize everything
10. Add complexity in steps
11. Check model modes are correct

Exercise: Why is my model 'perfect'? 🤔

A student is training a binary classifier. The training loss goes to exactly 0.00 after 1 epoch, and validation accuracy is 100%. However, when they test it on a new hidden dataset, the accuracy is 50% (random guessing).

Discussion Question (2 minutes): In pairs, identify three potential ML-specific bugs from our list (Data, Math, or Overfitting) that could cause this. Which one is the most likely 'silent' culprit?

Meme of the day

https://skaftenicki.github.io/dtu_mlops/s4_debugging_and_logging/