Advanced Deep Learning

Peter Bloem
Data Science Center, UvA

see also **shorturl.at/jpHM4**

In the previous lectures, we've seen how to build a basic neural network, how to train it using backpropagation and we've met our first specialized layer: the Convolution. Putting all this together, you can build some pretty powerful networks already, but doing so is not a trivial process. Today, we'll look at some of the things you need to know about to actually design, build and test deep neural nets that work well, and work efficiently.

THE PLAN FOR TODAY

10:00 Introduction. What are the main interests in the room?

10:20 Part one: DL projects What kinds of tools does Deep Learning offer and how can we chain them together into a model?

10:40 Brainstorm for all. What are people doing, and what kind of model you would design for it.

11:00 Group brainstorm. Every group picks one group member's current project and they come up with a set of model designs. We discuss some of them.

11:20 Part two: Internals and data preparation

11:40 Worksheet. A worksheet of quick exercises to reinforce understanding of the deeper technical details.

12:00 Part three: big hunting How do you deal with bugs. How do you plan your project and design your code so that you avoid problems, or at least catch them early.

12:20 Build an example model. A worksheet for a simple autoregressive model. This shows what a pytorch project looks like if you code a lot of it from scratch.

12:40 Q&A. Final question and answer session.

We'll divide the generic deep learning development pipeline into these four basic stages, and give some general tips and pointers for each.

PART ONE: DEEP LEARNING PROJECTS

Let's start by looking at the practical business of building deep learning models. Either for research purposes, or for production settings. What does the process look like, and what do you need to keep in mind at each step?

THE GENERAL TIMELINE

Pick a task, get some data

Develop a model, tune hyperparameters

Debugging your model

Publish model, or push to production

We'll divide the generic deep learning development pipeline into these four basic stages, and give some general tips and pointers for each.

DATA, BEST PRACTICES

Withhold **test data** to gauge your model performance

Withhold **validation data** to develop your model and tune the hyperparameters (learning rate, batch size, etc).

Whatever is left over is your **training data**.

Benchmarks come with *canonical splits*. If not, you're responsible for splitting.

mlvu.github.io/lecture03

Most of the standard practices of data handling are carried over from basic machine learning, so we won't go into them here. **If you haven't taken a machine learning course, please make sure to read up on things like ROC curves, dataset splits, cross validation and so on.** We will only review briefly the most important principle: the notion of a train, validation and test set.

You measure the performance of your model on data that it hasn't seen before: the test set. If you overuse the test set during model development, you end up tuning the model to the test set anyway, so the rule is that in a given project you use the test set **only once**: to measure the final performance of your model that you will report. In order to make choices during model development (which hyperparameters, how many layers, etc) you check the performance on another set of withheld data called the validation set. Whatever data you have left over after withholding these two, you can use as training data.

The MNIST data that you've seen in the first assignment is an example of a benchmark that comes with a canonical test/train split, but not with validation data. That doesn't mean you're free to use the test set for validation**, it means you have to split the training data into a training and validation split yourself.**

HOW MUCH DATA DO YOU NEED?

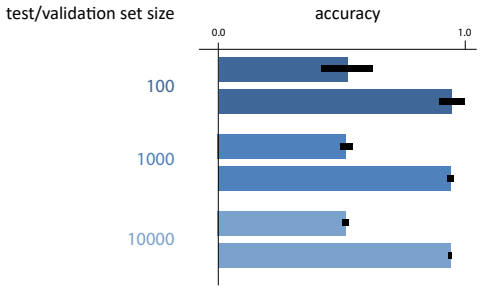
The size of the test set is more important than the size of the training set.

6

Deep learning has a reputation for being data hungry. Don't expect to get very far with a hugely deep model, if you don't have a decent amount of data. How much exactly? The first consideration is not how much training data you have, but how much test data (and, by extension, how much validation data)

Having a small training set may mean that your model doesn't learn, but having a small test set means that you can't even tell whether your model is learning or not.

CONFIDENCE INTERVALS



7

To understand the importance of test set size, let's look more closely at what we're doing when we compute the test set accuracy of a binary classifier. In effect, we're estimating a statistic: **the probability that a randomly drawn instance will be classified correctly**. The size of the test set is our *sample size for this estimate*. If we draw a confidence interval around our estimate, we can see the impact of this sample size.

The size of this confidence interval depends on two factors: the true accuracy* and the size of the test set. Here are some examples for different accuracies and test set sizes. (NB:)

This tells us that if the true success probability (accuracy) of a classifier is 0.5, and the test set contains 100 examples, our confidence interval has size 0.2. This means that even if we report 0.5 as the accuracy, we may well be wrong by as much as 0.1 either side. We can't reliably tell the difference between 40% and 60% accuracy.

Even if these confidence intervals are usually not reported, you can easily work them out (or look them up) yourself. **So, if you see someone say that classifier A is better than classifier B because A scored 60% accuracy and and B scored 59%, on a test set of 100 instances, you have a good reason to be sceptical.**

In practice, we usually end up trying to differentiate between scores like 99.4% and 99.5%. Clearly, for these kind of measurements to be meaningful, we need test and validation sets with at least 10 000 instances, and often more.

*We don't know the true accuracy, but it's accepted practice to substitute the estimate, since it is likely close enough to get a reasonable estimate of the confidence interval.

HOW MUCH DATA DO I NEED?

Split off a **test set** that allows for small confidence intervals

10 000 instances is a good aim

Split off a **validation set** of similar size

half the size of test is fine

The rest is your **training data**

If your dataset is just too small:

- Consider not using machine/deep learning
- Find lots of *unlabeled* data: self/semi-supervised learning
- For evaluation: combined 5x2 cross-validation F-testing (Alpaydin '99)

8

Sometimes your dataset comes with a canonical split. If it doesn't you'll need to make your own split.

For your final training run, you are allowed to train on both your training data and your validation data.

DO NOT USE YOUR TEST SET MORE THAN ONCE.

Just to reiterate: this is a really important principle. We can't go deeply into why this is so important, your bachelor ML course should have covered this*, but just remember this as a rule.

This is something that goes wrong a lot in common practice, and the tragic thing is that it's a mistake you cannot undo. Once you've developed your model and chosen your hyperparameters based on the performance on the test set, the only thing you can do is fess up in your report.

The safest thing to do is to split the data into different files and simply not look at the test file in any way.

*For a refresher,see **mlvu.github.io/lecture03**

THE GENERAL TIMELINE

~~Pick a task, get some data~~

Develop a model, tune hyperparameters

Debugging your model

Publish model, or push to production

10

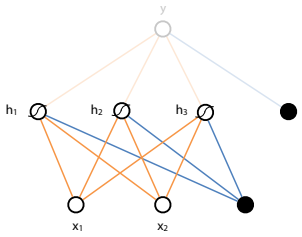
We'll divide the generic deep learning development pipeline into these four basic stages, and give some general tips and pointers for each.

Know the basic building blocks.

Know their inductive biases.

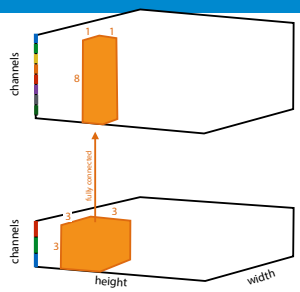
Understand what assumptions they make about the structure of your data.

BUILDING BLOCKS: MLP, FULLY CONNECTED LAYER



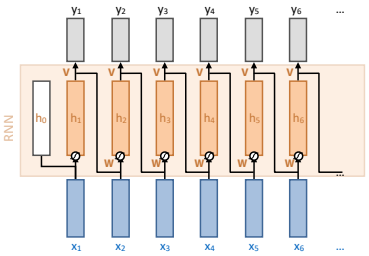
Inductive bias: none

BUILDING BLOCKS: CONVOLUTION



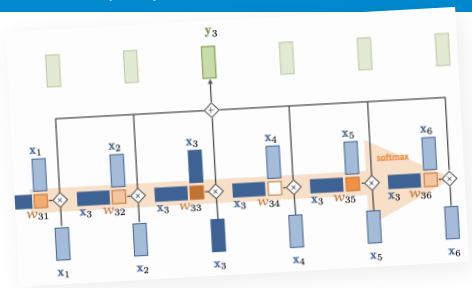
Inductive bias: translational equivariance, locality

BUILDING BLOCKS: RNN



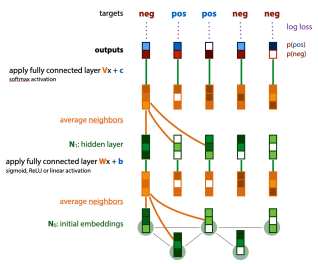
Inductive bias: translational equivariance, "causality", locality, long dependence

BUILDING BLOCKS: (SELF) ATTENTION



Inductive bias: permutation equivariance/depends on embeddings, long dependence, finite memory

GNNS



Inductive bias: relational structure, neighbor similarity

Also:

- Regularizers: L1, L2, Dropout
- Normalization: Batch, instance, group, layer
- Residual connections, gates

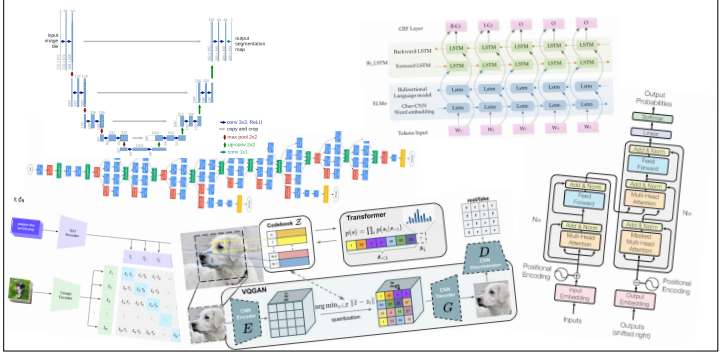
- Embedding layers

17

MLPs, Convolutions, RNNs, attention and GNNS are the three main groups that perform the computation of our neural networks.In addition to that there are many other operations that function as a kind of "control units". By themselves, they don't facilitate the bulk of the learning, but they help the learning go smoothly.

Embedding layers are a bit of an odd duck. They certainly are a key part of the learning, but they don't really perform much of a computation: it's more that they hel us represent our entities before we start learning on them.

COMBINING THE LEGO BLOCKS



GENERAL TIPS

Start with a setup you know works. Plan a careful route to your own design.

Baselines, baselines, baselines.

```
Competing models, linear models, majority class, random class
```

Scale up slowly: in features added, data size, in model size, in task hardness.

19

If you are designing a new network or architecture, or otherwise trying something new that may or may not work, **don't just implement your idea right away**. There's a vanishing chance that it'll work out of the gate, and you'll be stuck with a complicated mess that doesn't work either because you have a bug somewhere, *or* because your idea doesn't work. **You'll be stuck looking for a bug that you are not sure is even there**, which is an almost impossible task.

Instead, start with a model that you know *has* to work, and for which you know *why* it must work. Either make the model simpler or make the data simpler:

- Start with synthetic data that makes the learning problem trivial to solve. Slowly scale up to more realistic data.
- Start with a competing model for which the performance has been reported. Replicate the performance, and then transform this model into yours step by step.
- Divide and conquer. If your model consists of separate

features, implement them one by one., and check the performance impact of each. If it doesn't think hard about how you can break things up.

The main requirement for a successful deep learning project is not a good idea: it's a **good plan** consisting of small and careful increments.

A **baseline** is simply another model that you compare against. This can be a competing model for the same task, but also a stupidly simple model that helps you calibrate what a particular performance means. Baselines help you to interpret your results, but they are also incredibly helpful during development. For instance, if you start with a baseline, and you slowly turn it into the model you had in mind, then at each step you can always check if the performance drops to tell you whether you have a bug.

FOR EXAMPLE

"I want to build a 6 layer CNN for MNIST classification."

1. Linear model
2. 1 convolution, linear layer, no activation, no pooling.
3. 1 convolution, linear layer, activation, no pooling.
4. 1 convolution, linear layer, max pooling.
5. 2 convolutions, etc.

Here's an example. To build up to a 6 layer CNN, you might start slowly with a simple linear model (a one-layer NN). This will give you a baseline.

The next step is to introduce a stride-1 convolution before the linear layer. As you know, this doesn't change the expressivity of the network, but it allows you to check if the performance degrades. We know that this model is capable of performing the same as the linear model, so if performance drops, we know that there is either a bug or something is wrong with the gradient descent or initialization.

Then, we can introduce an activation. We know that activations don't usually hurt performance, but they might. If we can't get the network with activation to perform better than or as well as the previous network (even after much tuning of hyperparameters), we should try a different activation.

Then, we can introduce some pooling. This strictly reduces the size of the linear layer so it may hurt performance (we may make up this decrease by adding more convolutions). At this point, we're pretty sure that the rest of our code is sound, so if the performance drops, we know what the cause is.

This kind of approach may not always be necessary. On well-covered ground like CNNs, you can probably start with a more complex model right away and expect it to work. But that's only because somebody else has done this work before us. We know it has to work because many other people have got it to work before us. If we struggle, we can always take one of their versions, and eliminate the differences one by one.

Whenever you are breaking new ground, and building things from scratch, you need to start with a simple model and build up the complexity step by step.

IF YOU DON'T KNOW WHY IT *SHOULD* WORK,

YOU WON'T KNOW WHY IT *DOESN'T* WORK

The main principle behind scaling up slowly is that you need to have a sound reasoning for why the code you are about to run will work. If you don't, and the code doesn't work, you can't tell whether it's because of a bug somewhere, or because the ideas behind the code are simply not sound.

One of the best things you can do in building neural networks is to make sure that you know exactly what should happen when you execute your code. Inevitably, that then won't happen (we're programming after all), but then at least you can be sure that something is wrong with the code and not with your idea.

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Develop a model~~, tune hyperparameters

Debugging your model

Publish model, or push to production

22

We'll divide the generic deep learning development pipeline into these four basic stages, and give some general tips and pointers for each.

TUNING STRATEGIES: TRIAL AND ERROR

Usually good enough.

Easy to use model insights.

You know what your hyperparameters mean.

Difficult to do fairly.

Nobody tunes their baselines as much as their own model.

23

What about the rest of our hyperparameters? We also need to figure out how many layers our network needs, how to preprocess the data, and how set things like regularization parameters (which we will discuss later).

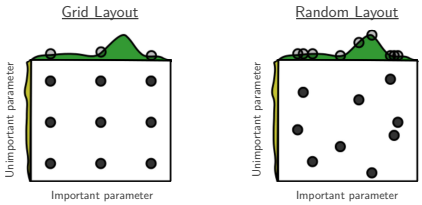
How should we proceed? Just try a bunch of settings that seem right and check our validation performance? Or should we follow some rigid strategy?

In practice, when it comes to tuning your own model, simple trial and error is usually fine, so long as you check the performance only on the validation set. It also usually works *better* than automatic approaches, because you know best what your hyperparameters mean and what their impact may be. You can reason about them in a way that automatic methods can't.

However, it is difficult to dispense the same amount of effort when you are trying to tune your baselines. If you want truly fair comparisons, **automatic hyperparameter tuning** might be a better approach.

AUTOMATIC TUNING: GRID SEARCH VERSUS RANDOM SEARCH

Grid search: define values for each parameters, try all possibilities.



NB: linear vs logarithmic scales: 0.1, 0.2, 0.3 or 0.0001, 0.001, 0.01, 0.1

image source: *Random search for hyper-parameter optimization*, Bergstra and Bengio JMLR 2012

If you want to tune your hyperparameters in a more rigorous or automated fashion, one option is **grid search**. This simply means defining a set of possible values for each hyperparameter and trying every single option exhaustively (where the options form a *grid* of points in your hyperparameter space).

This may seem like the best available options, but as this image shows, selecting hyperparameters randomly may lead to better results, because it gives you a larger range of values to try over different dimensions. If one parameter is important for the performance and another isn't you end up testing more difference values for the important parameter.

You should also think carefully about the *scale* of your hyperparameter. For many hyperparameters, the main performance boost comes not from the difference between 0.1 and 0.2 but from the difference between 0.01 and 0.1. In this case it's best to test the hyperparameters in a **logarithmic scale**. The learning rate is an example of this.

TUNING THE LEARNING RATE

Fix a batch size first

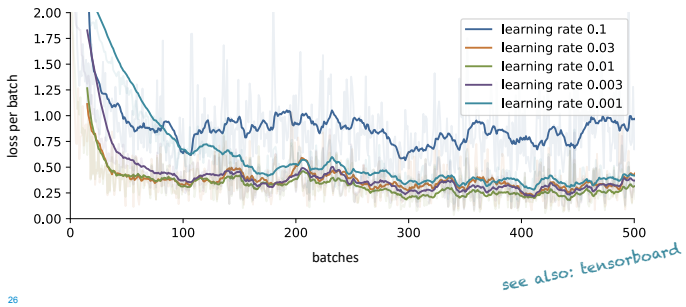
As big as fits in memory is usually reasonable. Better: measure the *throughput*, the amount of data seen by the model per second, and choose your batch size to optimize that.

Standard: try `0.1, 0.01, 0.001, 0.0001, 0.00001` for a few epochs each. Compare per-batch loss curves.

Your learning rate is *very dependent* on the batch size. You can tune both together, but generally, the bigger the batch size, the faster your training, so people tend to go for the largest batch size that will fit in memory and tune with that. There is some evidence that smaller batches often work better, but then you can train longer with a larger batch size, so the trade-off depends on the model.

If you're running large experiments, it can be good to measure throughput instead. This is the amount of data (eg. the number of characters) that your models sees per second. Often, GPUs work a lot faster if their memory isn't entirely full, so the batch size that will give you optimal throughput is the one that fills you memory to something like 80%.

CHECK YOUR (PER-BATCH) LOSS CURVES



The simplest way to find a reasonable learning rate is to take a small number of logarithmically spaced values, do a short training run with each, and plot their loss per batch (after each forward pass, you plot the loss for that batch).

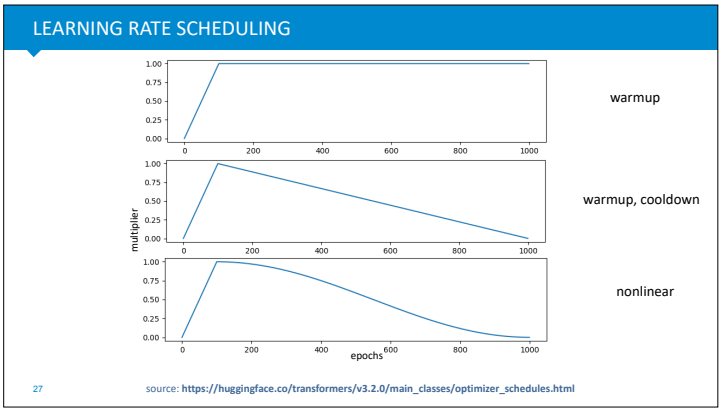
This is usually a noisy curve (especially if your batch size is small), so you'll need to apply some smoothing to see what's going on.

In this case, 0.1 is clearly too high. The loss bounces around, but never drops below a certain line. 0.03 seems to give us a nice quick drop, but we see that in the long run 0.01 and 0.003 drop below the orange line. 0.001 is too low: it converges very slowly, and never seems to make it into the region that 0.01 is in.

Note that the difference between the lower learning rates seems small, but these differences are not insignificant. Small improvements late in the process tend to correspond to much more meaningful learning than the big changes early on. It can be helpful to plot the vertical axis in a logarithmic scale to emphasize this.

One popular tool for automatically tracking your loss this

way is Tensorboard (which originated with tensorflow, but now works with both tensorflow and pytorch). There are also online tools like *weights and biases*.



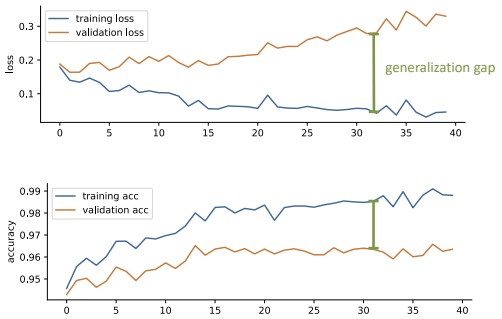
One trick that can help speed things up a little is to change the learning rate as learning progresses.

There are many schedules possible, and the precise differences aren't that important. The main tricks that are often tried are:

* Warming up at the start of training: starting at 0 and slowly increasing the learning rate. This is particularly useful with optimizers like Adam (which we'll discuss later) which gather secondary statistics about the gradients. It ensures that gradient descent doesn't make any big steps until these statistics are accurate estimates.
* Cooling down towards the end of learning (lowering the learning rate again). This can help with the problem where gradient descent is in an optimum, but bouncing from side to side instead of finding the minimum (this is known as oscillation). Momentum-based optimizers should recognize and dampen oscillations automatically, but sometimes a cooldown on the learning rate helps the process along.

In general, you should try learning without a schedule first, and look at your loss curves and gradient norms to see if you think a learning rate schedule might help.

PER-EPOCH LOSS CURVES



28

With a decent learning rate chosen, we can train for longer runs. At this point, a different kind of loss curve becomes important. For a smaller number of times during your training run (usually once per epoch), you should test your model on the whole validation set and on the whole training set*. You can then plot the average loss or accuracy (or any other metric) for both datasets.

What we see here is that if we look at just the training accuracy, we might be confident of getting almost 99% of our instances correct. Unfortunately, the validation accuracy is much lower at around 96%. Some parts of our performance are due to *overfitting* (remembering the data), and will not **generalize** to unseen data. For this reason the difference between the performance on the training and validation data is called the **generalization gap**.

* People often use the running loss form the past epoch in place of this, but it's usually a little more accurate to rerun the model on the whole training data (since the running loss comes from a changing model)

STABILIZING, SPEEDUPS

Learning rate warmup, cooldown

Gradient clipping: reduce gradient if it exceeds a threshold.

Either by element-wise clamping, or by normalizing the total norm

Momentum: more later

Regularization, batch normalization: more later

29

In general, a high learning rate is preferable to a low one, so long as you can keep the learning stable. These are some tricks to help learning stabilize, so that you can use higher learning rates, and train faster.

A learning rate warmup is a slow, linear increase in the learning rate, usually in the first few epochs. This can usually help a lot to stabilize learning. A cooldown at the end of learning can also help, but is less common.

Momentum and batch normalization can also help a lot. We will discuss these techniques later.

SIMPLICITY CAN BE MORE MEANINGFUL THAN *ACCURACY*

Remember however, that using too many tricks can hurt your *message*. People are much less likely to trust the performance of a model that seems to require very specific hyperparameters to perform well. If your model only performs with a specific learning rate of 0.004857, a five stage learning rate schedule, and a very particular early-stopping strategy, then it's unlikely that that model performance will be robust enough to translate to another dataset or domain (at least, not without a huge effort in re-tuning the hyperparameters).

Your performance is also unlikely to transfer from validation to test, and your audience may be skeptical that you chose such particular hyperparameters without occasionally sneaking a look at your test set.

However, if you report performance for a simple learning rate of 0.0001, with no early stopping, gradient clipping, or any other tricks, it's much more likely that your performance is *robust*.

In other words, hyperparameter tuning is not simply about playing around until you get a certain performance. It's about finding a **tradeoff between simplicity and**

performance.

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Develop a model, tune hyperparameters~~

Debugging your model

Publish model, or push to production

31

Next up, debugging. After you build a model, you are bound to have some bugs. Deep learning models are hard to debug, so it's good to know a few tricks.

WHY IS DEBUGGING DIFFICULT?

Neural networks fail *at runtime*

e.g. shape errors

Neural networks fail *silently*

especially due to broadcasting

Neural networks *may not fail at all*

More in part three.

32

If your language supports type checking, then a lot of the mistakes you make will be noticed at compile time. That is, before you even run the program. This is the best time to spot a bug. You know immediately that somethings gone wrong, and usually you get a very accurate pointer to where you made the mistake.

In deep learning, things are more hairy. Python doesn't have type checking, but even if it did, almost every object in our program is usually a tensor. Type information doesn't help us here. If our program fails, it'll usually be at runtime. If we're lucky we will at least get a pointer to where we made the mistake, but if we are using lazy execution, or the mistake happens in the backward pass, then the failure happens at a different point in the program to where we made the mistake.

Even worse, if all your tensor shapes broadcast together (more on this later), your neural network may fail without telling you. You'll have to infer from your loss becoming NaN or Inf that something is wrong. Maybe your loss stays real-valued, but it simply doesn't go down. In this case you have to decide whether your neural network is |(a)

correctly implemented, but not learning (b) has a bug that yo haven't spotted yet.

Finally, sometimes neural networks may not even fail at all, despite the fact that you've made a mistake. You may see the loss converge and the network train despite the fact you've not implemented the model you thought you'd implemented.

In short, deep learning programming is by nature more treacherous than regular programming, so it pays to be extra vigilant. The main way to learn this is to suffer through it a few times, but we'll try to minimize the amount you have to suffer by flagging up a couple of common mistakes and by giving a few useful tricks.

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Develop a model, tune~~ hyperparameters

~~Debugging your model~~

Publish model, or push to production

33

Once you're pretty sure you have a model that is doing something reasonable, you need to start tuning it, to see how much you can boost the performance.

AUTOMATIC TUNING

Useful for fair comparisons: each model gets the same amount of compute.

Are GANs Created Equal? A Large-Scale Study Lucic et al, NeurIPS 2018

On the State of the Art of Evaluation in Neural Language Models Melis et al, ICLR 2018

You CAN Teach an Old Dog New Tricks! On Training Knowledge Graph Embeddings Ruffinelli et al, ICLR 2020

Random search with *Sobol configurations* for discrete parameters.

Bayesian search for continuous hyperparameters.

https://ax.dev/

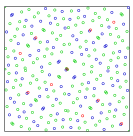


image source: By Jheald - Own work. Created in R.
CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=16106862

34

Here are some examples of papers that do this kind of automatic turning. One popular approach is to use a random layout for the discrete parameters, but to use something called **Sobol sequences** to make the sampled points a little more regular, and to then tune the continuous hyperparameters using Bayesian search.

Ax.dev is one popular platform for such experiments (but beware, they require *a lot* of compute).

THE GENERAL TIMELINE

~~Pick a task, get some data~~

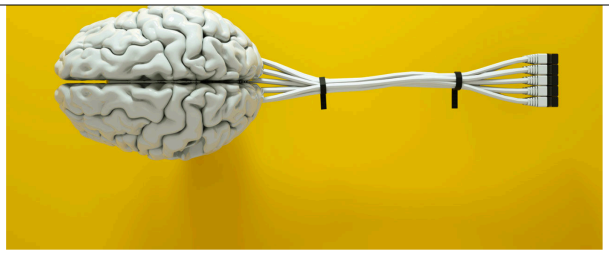
~~Debugging your model~~

~~Develop a model, tune hyperparameters~~

Publish model, or push to production

35

The last stage of our pipeline is to deploy our model. Either we turn it into a production model powering our software product, or we write a research paper about our findings.



JUST_SUPER/ISTOCK.COM

Eye-catching advances in some AI fields are not real

By Matthew Hutson | May. 27, 2020 , 12:05 PM

Artificial intelligence (AI) just seems to get smarter and smarter. Each iPhone learns your face

The practice of manual tuning is a large contributing factor to a small reproduction crisis in machine learning. A lot of results have been published over the years that turn out to disappear when the baselines and the models are given equal attention by an automatic hyperparameter tuning algorithm.

There are a few potential reasons for this:

- **Publication bias:** some people get lucky and find some high performing parameters that generalize to the test set but not beyond that.
- **Unrigorous experimentation:** nobody checks how careful you are in your experiments. If some experimenters are careless about never using their test set, they may get better results than their more careful colleagues. The publication system will then select the more careless researchers.
- **Regression towards the mean:** Selecting good results and then retesting will always yield lower average scores. If we we take all students with a 9 or higher for some exam and retest them, they will probably score below 9 on average. This is because performance is always partly due to ability and partly due to luck.

For now, the important message is that we should be mindful that when we compare hand-tuned models, there is room for error. We can't afford to do full hyperparameter sweeps for every experiment we publish, so the best option is probably to keep hand-tuning and occasionally publish a big replication study were all current models are pitted against each other in a large automated search.

https://www.sciencemag.org/news/2020/05/eye-catching-advances-some-ai-fields-are-not-real

PUBLISHING: ABLATION

Which features have the most impact?

1) Build the best model you can.
2) Remove features one-by-one.
3) Measure impact step by step.

Hyperparams				Dev Set Accuracy		
#L	#H	#A	LM (ppl)	MNLI-m	MRPC	SST-2
3	768	12	5.84	77.9	79.8	88.4
6	768	3	5.24	80.6	82.2	90.7
6	768	12	4.68	81.9	84.8	91.3
12	768	12	3.99	84.4	86.7	92.9
12	1024	16	3.54	85.7	86.9	93.3
24	1024	16	3.23	86.6	87.8	93.7

Table 6: Ablation over BERT model size. #L = the number of layers; #H = hidden size; #A = number of attention heads. "LM (ppl)" is the masked LM perplexity of held-out training data.

source: *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Devlin et al, 2018

Since the final model is usually a combination of many different innovations, it's good to figure out which of these is the most important for the performance.

The most common way to do this is an **ablation study**: you first pick the best model with all the bells and whistles, and then remove features one by one to evaluate the impact.

There's no standard way to design an ablation. The main principle is that you pick a full-featured model *first*, because the features likely need to interact with each other to improve the performance, and then you measure their performance.

ML IN PRODUCTION

Not to be underestimated

Be wary of:

- Distributional drift
- Cost of inference
  Is it worth paying $10^{-6}$$ for every product recommendation?
- Difference between *prediction* and *taking action*
  Feedback loops!

Finally, even if you have a cheap model that perfectly predicts what you want it to predict, when you put it into production you will be using that model to **take actions**. This means that its predictions will no longer be purely offline, as they were in the training setting. For instance, if you recommend particular items to your users, you are driving the whole system of all your users engaging with your website towards a particular mode of behavior. Your model was only trained to predict certain targets on data where the model itself was not active. In short, you have no idea where the model will drive the interactions between your users and your website.

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Debugging your model~~

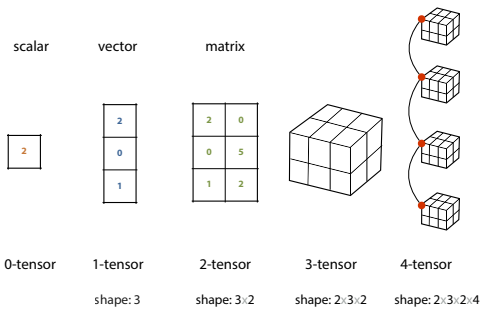
~~Develop a model, tune hyperparameters~~

~~Publish model, or push to production~~

PART TWO: **THE PYTORCH INTERNALS**

Let's start by looking at the practical business of building deep learning models. Either for research purposes, or for production settings. What does the process look like, and what do you need to keep in mind at each step?

TENSORS (AKA MULTIDIMENSIONAL ARRAYS)



VU

The basic datastructure of our system will be the tensor. A tensor is a generic name for family of datastructures that includes a scalar, a vector, a matrix and so on.

There is no good way to visualize a 4-dimensional structure. We will occasionally use this form to indicate that there is a fourth dimension along which we can also index the tensor.

We will assume that whatever data we work with (images, text, sounds), will in some way be encoded into one or more tensors, before it is fed into our system. Let's look at some examples.

A CLASSIFICATION TASK AS TWO TENSORS

word count	nr. of recipients	class
30	3	ham
340	4	ham
121	2	spam
11	1	spam
23	1	spam
455	1	spam
512	2	spam
2	12	ham
32	1	ham
432	1	ham
23	2	spam

x =

30	3
340	4
121	2
11	1
23	1
455	1
512	2
2	12
32	1
432	1
23	2

y =

0
0
1
1
1
1
0
0
0
1

VU

A simple dataset, with numeric features can simply be represented as a matrix. For the labels, we usually create a separate corresponding vector for the labels.

Any categoric features or labels should be converted to numeric features (normally by one-hot coding).

AN IMAGE AS A 3-TENSOR

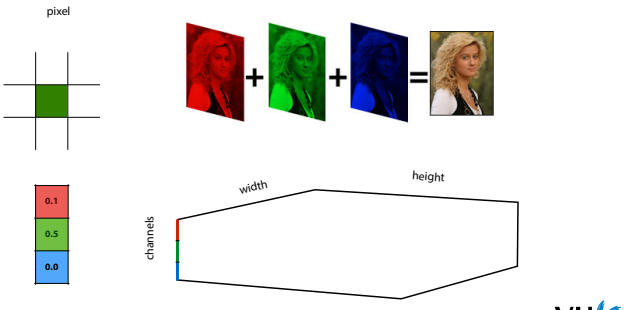
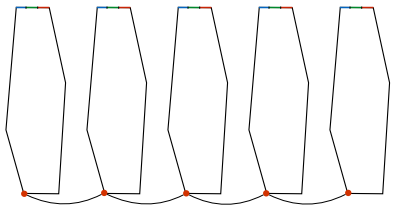


image source: http://www.adsell.com/scanning101.html

Images can be represented as 3-tensors. In an RGB image, the color of a single pixel is represented using three values between 0 and 1 (how red it is, how green it is and how blue it is). This means that an RGB image can be thought of as a stack of three color channels, represented by matrices.

This stack forms a 3-tensor.

A DATASET OF IMAGES



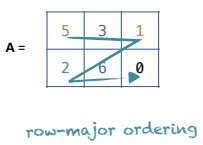
```
In [5]: from keras.datasets import cifar10
        (x_train, y_train), (x_test, y_test) = cifar10.load_data()
        x_train.shape
Out[5]: (50000, 32, 32, 3)
```

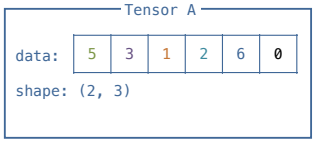

If we have a dataset of images, we can represent this as a 4-tensor, with dimensions indexing the instances, their width, their height and their color channels respectively. Below is a snippet of code showing that when you load the CIFAR10 image data in Keras, you do indeed get a 4-tensor like this.

There is no agreed standard ordering for the dimensions in a batch of images. Tensorflow and Keras use (batch, height, width, channels), whereas Pytorch uses (batch, channels, height, width).

(You can remember the latter with the mnemonic "**bach**elor **chow**".)

TENSORS IN MEMORY: ROW MAJOR ORDERING





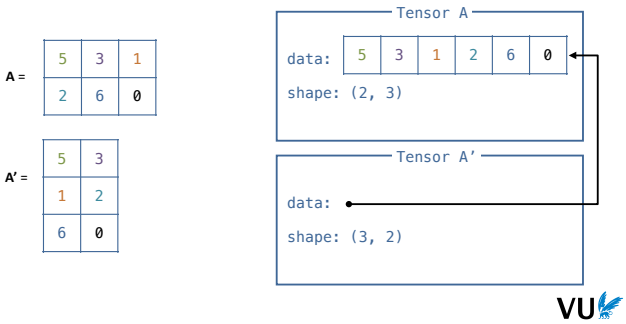
It's important to realize that even though we think of tensors as multidimensional arrays, in memory, they are necessarily laid out as a single line of numbers. The Tensor object knows the shape that these numbers should take, and uses that to compute whatever we need it to compute.

We can do this in two ways: scanning along the rows first and then the columns is called **row-major ordering**. This is what numpy and pytorch both do. The other option is (unsurprisingly) called column major ordering. In higher dimensional tensors, the dimensions further to the right are always scanned before the dimensions to their left.

Imagine looping over all elements in a tensor with shape (a, b, c, d) in four nested loops. If you want to loop over the elements in the order they are in memory, then the loop over d would be the innermost loop, then c, then b and then a.

This allows us to perform some operations very cheaply, **but we have to be careful**.

RESHAPE/VIEW

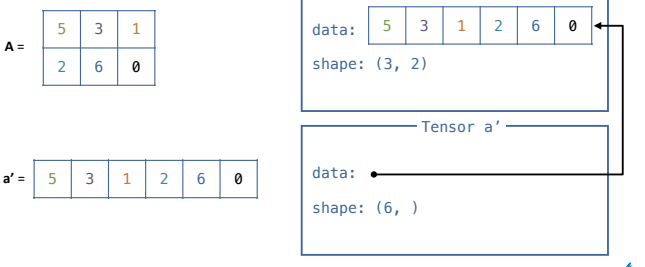


VU

Here is one such cheap operation: a reshape. This changes the shape of the tensor, but not the data.

To do this cheaply, we can create a new tensor object with the same data as the old one, but with a different shape.

RESHAPE/VIEW



VU

EXAMPLE: NORMALIZE COLORS

```
images = load_images(…)
n, h, w, c = images.size()

images = images.reshape(n*h*w, c)
images = images / images.max(dim=0)
images = images.reshape(n, h, w, c)

# this makes no sense, but it is allowed.
images = images.reshape(n*h, w*c)
images = images.reshape(h, n, c, w)
```


VU

Imagine that we wanted to scale a dataset of images in such way that over the whole dataset, each color channel has a maximal value of 1 (independent of the other channels).

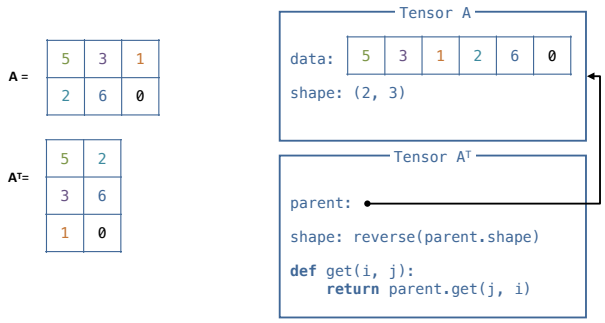
To do this, we need a 3-vector of the maxima per color channel. Pytorch only allows us to take the maximum in one direction, but we can reshape the array so that all the directions we're not interested in are collapsed into one dimension.

Afterwards, we can reverse the reshape to get our image dataset back.

We have to be careful: the last three lines in this slide form a perfectly valid reshape, but the c dimension in the result does *not* contain our color values.

In general, you're fine if you **collapse dimensions that are next to each other**, and uncollapse them *in the same order.*

TRANSPOSE

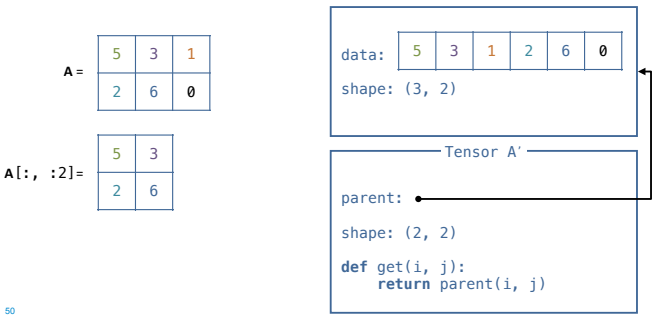


49

A transpose operation can also be achieved cheaply. In this case, we just keep a reference to the old tensor, and whenever a user requests the element (i, j) we return (j, i) from the original tensor.

NB: This isn't precisely how pytorch does this, but the effect is the same: we get a transposed tensor in constant time, by viewing the same data in memory in a different way.

SLICE



50

Even slicing can be accomplished by referring to the original data.

NON-CONTIGUOUS

Contiguous tensor: the data are directly laid out in row-major ordering for the shape with no gaps or shuffling required.

```
x = torch.randn(2, 3)
x = x[:, 1:]
x.view(6)
Traceback (most recent call last):
…
RuntimeError: view size is not compatible with input tensor's size and
stride (at least one dimension spans across two contiguous subspaces).
Use .reshape(...) instead.
```

`x.contiguous()`: make contiguous (by copying the data).

`x.view()`: reshape if possible without making contiguous.

`x.reshape()`: reshape, call `contiguous()` if necessary.

51

VU

For some operations, however, the data needs to be contiguous. That is, the tensor data in memory needs to be one uninterrupted string of data in row major ordering with no gaps. If this isn't the case, pytorch will throw an exception like this.

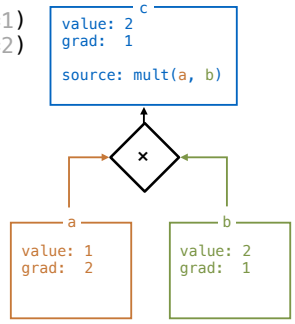
You can fix this by calling `.contiguous()` on the tensor. The price you pay is the linear time and space complexity of copying the data.

EXAMPLE: EAGER EXECUTION

```
a = TensorNode(value=1)
b = TensorNode(value=2)

c = a * b

c.backward()
```



VU

In eager mode deep learning systems, we create a node in our computation graph (a TensorNode) by specifying what data it should contain. The result is a tensor object that stores both the data, and the gradient over that data (which will be filled later).

Here we create the variables a and b. If we now apply an operation to these, for instance to multiply their values, the result is another variable c. Languages like python allow us to *overload* the * operator it looks like we're just computing multiplication, but behind the scenes, we are creating a computation graph that records all the computations we've done.

We compute the data stored in c by running the computation immediately, but we also store references to the variables that were used to create c, and the operation that created it. **We create the computation graph on the fly, as we compute the forward pass.**

Using this graph, we can perform the backpropagation from a given node that we designate as the loss node (node c in this case). We work our way down the graph computing the derivative of each variable with respect to c. At the start the TensorNodes do not have their grad's filled in, but at the end of the backward, all gradients have been computed.

Once the gradients have been computed, and the gradient descent step has been performed, we clear the computation graph. **It's rebuilt from scratch for every forward pass.**

PART THREE: BUG HUNTING

Let's start by looking at the practical business of building deep learning models. Either for research purposes, or for production settings. What does the process look like, and what do you need to keep in mind at each step?

WHY IS DEBUGGING DIFFICULT

Neural networks fail *at runtime*

e.g. shape errors

Neural networks fail *silently*

especially due to broadcasting

Neural networks *may not fail at all*

More in part three.

54

If your language supports type checking, then a lot of the mistakes you make will be noticed at compile time. That is, before you even run the program. This is the best time to spot a bug. You know immediately that somethings gone wrong, and usually you get a very accurate pointer to where you made the mistake.

In deep learning, things are more hairy. Python doesn't have type checking, but even if it did, almost every object in our program is usually a tensor. Type information doesn't help us here. If our program fails, it'll usually be at runtime. If we're lucky we will at least get a pointer to where we made the mistake, but if we are using lazy execution, or the mistake happens in the backward pass, then the failure happens at a different point in the program to where we made the mistake.

Even worse, if all your tensor shapes broadcast together (more on this later), your neural network may fail without telling you. You'll have to infer from your loss becoming NaN or Inf that something is wrong. Maybe your loss stays real-valued, but it simply doesn't go down. In this case you have to decide whether your neural network is |(a) correctly implemented, but not learning (b) has a bug that yo haven't spotted yet.

Finally, sometimes neural networks may not even fail at all, despite the fact that you've made a mistake. You may see the loss converge and the network train despite the fact you've not implemented the model you thought you'd implemented.

In short, deep learning programming is by nature more treacherous than regular programming, so it pays to be extra vigilant. The main way to learn this is to suffer through it a few times, but we'll try to minimize the amount you have to suffer by flagging up a couple of common mistakes and by giving a few useful tricks.

ASSERT

```
assert my_tensor.size() == (b, c, h, w)

assert not x.isnan().any(), 'tensor x contains a NaN value.'

assert len(x) == n, f'tensor x has dim {len(x)}, expected {n}.'
```

NB: Expect asserts to be *turned off* in production code.

55

The simplest way to stop a program from failing silently is to check whether the program is in the state you think it's in, and to **force it to fail** if it isn't

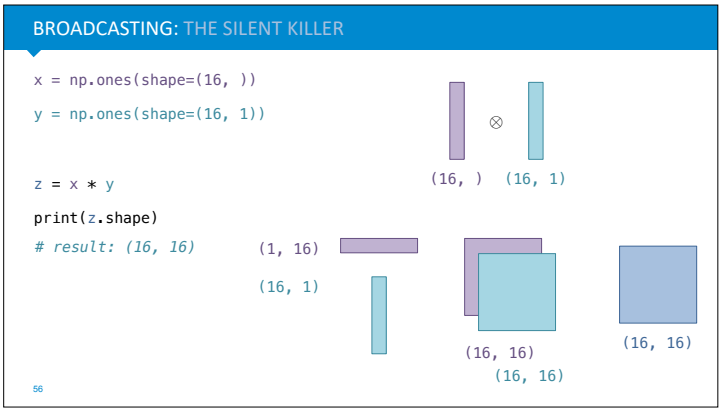
This is what the **`assert`** statement is for. It checks a condition and raises an exception if the condition is false. It may seem counterintuitive to make your own code fail, but a program that fails fast, *at the point where the mistake happens*, can save you days worth of debugging time.

In deep learning you will commonly add asserts for the dimensions of a particular tensor, or to check that a tensor contains no NaN or infinity values.

These are asserts in python, but the keyword exists in most languages. In python,the second (optional) argument is a string that is shown as a part of the error message when the assert fails. Using an f-string here allows you to add some helpful information.

Don't worry about the assert condition being expensive to compute. It's easy to run python in a way that the asserts are skipped (once you're convinced there are no bugs).

This does mean that you should only use asserts for things you expect to be **turned off in production**. So don't use them for, for instance, for validating user input.



Broadcasting is a mechanism that allows you to apply element-wise operations to tensors of different shapes.

It's very helpful for smoothly implementing things like scalar-by-matrix multiplication in an intuitive manner. It's also one of the more treacherous mechanisms in numpy and pytorch, because it's very easy to make mistakes if you don't fully understand it.

Here's an example of how it might go wrong. We have two tensors both representing vectors with 16 elements each. The first has shape (16,) and the second has shape (16, 1). Element-wise multiplying these, we might expect to get another tensor with 16 values with shape either (16,) or (16, 1).

What actually happens is that we get a matrix of 16 x 16. The (16,) vector is given an extra dimension to make the two match, and that extra dimension is added on the left. We now have a (16, 1) matrix and a (1, 16) matrix. Next, the singleton dimensions are expanded (the values repeated) to give us two (16, 16) matrices which are element-wise multiplied.

In short, don't expect broadcasting to always behave the way you expect. Know the exact rules for the mechanism, and be careful when you use it.

BROADCASTING

Applied to any element-wise operation on two or more tensors.

Sum, multiplication, division, even some slicing.

For example: **A** + **B**, with

```
shape(A) = (3, 4, 1)
shape(B) = (1, 3)
```

Align the shape tuples to the right:

```
(3, 4, 1)
   (1, 3)
```

<- danger

Add singletons to match # dimensions:

```
(3, 4, 1)
(1, 1, 3)
```

Expand singletons to match:

```
(3, 4, 3)
(3, 4, 3)
```

57

These are the rules of the broadcasting algorithm: the shapes are aligned on the right, extra singleton dimensions are added to make them match, and the singleton dimensions are expanded so that the shapes match. Once the shapes match, the element-wise operation is applied.

Here is the complete documentation: **https://numpy.org/doc/stable/user/basics.broadcasting.html** In pytorch, broadcasting works the same as it does in numpy.

The alignment step is the dangerous one. Here, two dimensions often get aligned with each other that you did not expect to be matched.

AVOIDING SHAPE ERRORS

Add the singleton dimensions yourself to be sure.

```
c = a[:, :, :] + b[None, : , :]
```

Keepdim

```
normalized = x / x.sum(dim=1, keepdim=True)
```

Open each method by getting the shapes of the inputs.

```
def forward(input):
  b, c, h, w = input.size()
```

Add copious **assert**s, *especially for tensor shapes*.

```
assert rowsums.size() == (b, c, h, 1)
```

58

One of the best ways to ensure that broadcasting works as expected is to manually add the singleton dimensions, so that both tensors have the same number of dimensions and you know how they will be aligned.

The None inside a slice adds a singleton dimension. Note that in the first example, "`a[:, :, :]`" is the same as just "`a`". However, this notation communicates to the reader what is happening.

Methods that eliminate a dimension (like summing out a dimension, or taking the maximum) come with a `keepdim` argument in pytorch (`keepdims` in numpy). If you set this to `True`, the eliminated dimension remains as a singleton dimension, which ensures that broadcasting will behave reliably.

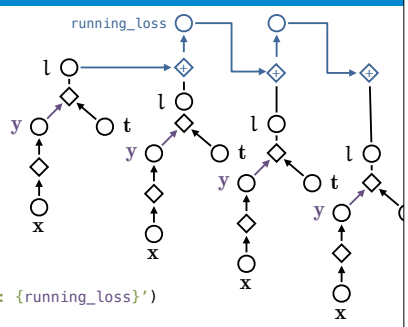
Opening a function with a line like the third one helps to make your code more readable: it tells the reader the dimensionality of the input tensor, and gives them four explicit shape variables to keep track of. These shape variables can then be used to easily assert stuff

---

MEMORY LEAK

```
for e in range(epochs):
   running_loss = 0.0
   for x, t in dataset:
      opt.zero_grad()
      y = model(x)
      l = loss(y, t)
      running_loss += l

   print(f'epoch {e} total loss: {running_loss}')
```



59

Here's another common mistake to be aware of: memory leaks from pointers to the computation graph. This one only happens in eager execution systems.

Remember that in eager execution, the computation graph is cleared and rebuilt for every forward pass. Usually, this clearing is done behind the scenes by the garbage collector. Once python sees that there are no references in the running code to any part of the graph, it will collect and delete the whole thing from memory.

This means that if you accidentally do have some variable that is active outside the training loop which references a node in the computation graph, the graph doesn't get cleared. Instead, you end up adding nodes to the same computation graph for every pass, constructing a massive graph until your memory runs out.

A common way this happens is if you keep a running loss as shown in the code. The line `running_loss += l` looks harmless, but remember what pytorch is doing under water. It's not just computing values, it's also adding nodes to the computation graph. Since running_loss is referenced outside the training loop, the graph never gets

cleared, and just grows and grows.

If your memory use balloons over several iterations (instead of returning to near 0 after each batch), this is probably what is happening. Check for any variables that exist outside the training loop which might be nodes in the computation graph.

MEMORY LEAK

```
for e in range(epochs):
    running_loss = 0.0
    for x, t in dataset:
        opt.zero_grad()
        y = model(x)
        l = loss(y, t)
        running_loss += l.item()
    print(f'epoch {e} total loss: {running_loss}')
```

see also x.detach() and x.data

60

In pytorch, the solution is to call `.item()`. This function works on any *scalar* tensor, and returns a simple float value of its data, unconnected to the computation graph.

In other situations you may be better served by the `.detach()` function, which creates a copy of your tensor node, detached from the current computation graph, and `.data`, which gives a view of the data as a basic tensor.

In modern pytorch, every tensor is always a computation graph node, so .detach() and .data do pretty much the same thing. The difference is a holdover from earlier versions, when Tensors needed to be turned into a computation graph node by wrapping them in a Variable object.

NaN LOSS

Something somewhere has become NaN, Inf or -Inf.

Try an absurdly low learning rate *and* a 0 learning rate

Localize the problem:

```
assert not x.isnan().any()
assert not x.isinf().any()
```

61

One of the most common problems is a loss that becomes NaN. This just means that some value somewhere in your network has become either NaN or infinite. Once that happens, everything else quickly fails as well, and everything in your network becomes NaN. To emphasize: this is not usually a problem with the loss, it's just that the loss is usually where you first notice it.

One of the causes of NaN loss is that your learning rate is too high. To eliminate this problem, try a run with a learning rate of 0.0 and 1e-12. If these are also NaN it's just a bug. If not, there are two options: (a) it's a bug that shows only for some parameters (b) your network works fine so long as you keep the learning rate reasonable.

To localize where the values of your forward pass first go wrong, you can add these kinds of asserts. Note that these operations are linear in the size of the tensor, so they have a non-negligable impact on performance. However, you can always turn off the asserts when you're sure the code is free of bugs (just run python with the -O flag)

NO LEARNING

Check a few learning rates.

Logarithmically: 1e-5, 3e-5, 1e-4, 3e-4, 1e-3, 3e-3, ...

Check your gradients.

```
x.retain_grad()
loss.backward()
print(x.grad.min(), x.grad.mean(), x.grad.max())
```

`grad == None` : backprop didn't reach it.

`grad == 0.0` : backprop visited, but the gradient died.

Next, you may find yourself with a network that runs without errors or NaN values, but that doesn't learn: the loss stays exactly or approximately the same.

The first thing to do is to try a few learning rates. Learning only happens for a relatively narrow band of learning rates, and the values changes from one setting to the next. If the problem happens for all learning rates, it's probably bug. The place to start then is **to check what gradients pytorch is computing**.

Pytorch does not hold on to gradients it doesn't need, by default. For debugging purposes, you can add a `retain_grad()` in the forward pass to a tensor. The gradient is then retained, and after the backward pass, you can print the gradient tensor (or some statistics).

If the gradient is None, then the backpropagation algorithm never reached it. This can happen if your computation graph is somehow disconnected (perhaps you used a non-differentiable operation somewhere). If the gradient has a value, but it's 0.0, then backpropagation reached the node, but the gradient died out. This can happen, for instance if you have ReLU activations somewhere with all inputs always negative, or a sigmoid activation with very large negative or positive inputs.

READ THE DOCUMENTATION

TORCH.NN.FUNCTIONAL.NLL_LOSS

torch.nn.functional.nll_loss(*input*, *target*, *weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*) [SOURCE]

The negative log likelihood loss.

See `NLLLoss` for details.

Parameters:

- **input** (*Tensor*) – (N, C) where *C = number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$ where $K \geq 1$ in the case of K-dimensional loss. *input* is expected to be log-probabilities.
- **target** (*Tensor*) – (N) where each value is $0 \leq \text{targets}[i] \leq C-1$, or $(N, d_1, d_2, ..., d_K)$ where $K \geq 1$ for K-dimensional loss.
- **weight** (*Tensor*, *optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size *C*