Lecture 4: Tools of the trade

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

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.

OUTLINE

part one: Deep Learning in practice

part two: Why does any of this work at all?

part three: Understanding optimizers

part four: The bag of tricks

VU

PART ONE: DEEP LEARNING IN PRACTICE

VU

THE GENERAL TIMELINE

Pick a task, get some data

Debugging your model

Develop a model, tune hyperparameters

Publish model, or push to production

VU

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.

5

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 review briefly the notion of a train, validation and test set.

The MNIST data 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?

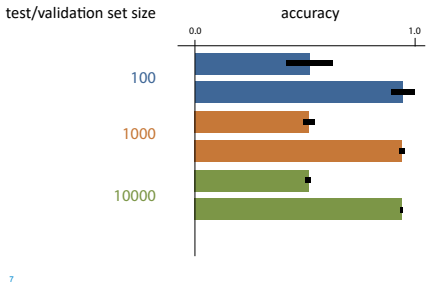
The size of the test set is more important than the size of the training set.

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 the size of the test set.

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 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.

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.

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 score 59%, on a test set of 100 instances, you have a good reason to be sceptical.**

*We don't know the true accuracy, but it's accepted practice to subtitute 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 ideal

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)

VU

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.

This is an 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.

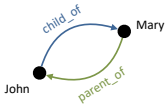
This is something that goes wrong a lot in common practice, and the tragic thing is that it's a mistake you cannot undo. Once you've developed your model and chosen your hyperparameters based on the performance on the test set, the only thing you can do is fess up in your report.

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

TEST SET LEAKAGE

Examples:

- Spam detection: emails shuffled in time dimension.
- Link prediction: graphs with inverse links.
- Preprocessing before splitting.
  - normalization, running averages



https://en.wikipedia.org/wiki/Leakage_(machine_learning)

VU

When you accidentally use information from your test set in a way that the model normally doesn't have access to, this is called a **test set leak**. This is very common, and a deep learning practitioner should be on the look out for mistakes like these.

There's no fool-proof recipe for avoiding leakage. This is one of those problems you have to train yourself to recognize, and be continually on the lookout for.

TEST SET LEAKAGE: GPT-3

Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

A major methodological concern with language models pretrained on a broad swath of internet data, particularly large models with the capacity to memorize vast amounts of content, is potential contamination of downstream tasks by having their test or development sets inadvertently seen during pre-training. To reduce such contamination, we searched for and attempted to remove any overlaps with the development and test sets of all benchmarks studied in this paper. Unfortunately, a bug in the filtering caused us to ignore some overlaps, and due to the cost of training it was not feasible to retrain the model. In Section 4 we characterize the impact of the remaining overlaps, and in future work we will more aggressively remove data contamination.

2.3 Training Process

As found in [KMH+20, MKAT18], larger models can typically use a larger batch size, but require a smaller learning rate. We measure the gradient noise scale during training and use it to guide our choice of batch size [MKAT18]. Table 2.1 shows the parameter settings we used. To train the larger models without running out of memory, we use a mixture of model parallelism within each matrix multiply and model parallelism across the layers of the network. All models were trained on V100 GPU's on part of a high-bandwidth cluster provided by Microsoft. Details of the training process and hyperparameter settings are described in Appendix B.

"Language Models are Few-Shot Learners" Brown et al 2020.

THE GENERAL TIMELINE

~~Pick a task, get some data~~

Debugging your model

Develop a model, tune hyperparameters

Publish model, or push to production

VU

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

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*

VU

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. Unfortunately, even if your language has type checking, it doesn't help much if everything is a tensor. The analogue of a type error in tensors is perhaps a shape error: combining two tensors that can't be broadcast together. Unfortunately, there is no compile-time shape checking in most deep learning platforms (keep an eye on hasktorch if you're interested in this).

Even worse, if all your shapes broadcast together, your neural network may fail without telling you. You'll have to infer from your loss becomes 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 made a

ASSERT

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

assert not contains_nan(x), '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.

VU

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.

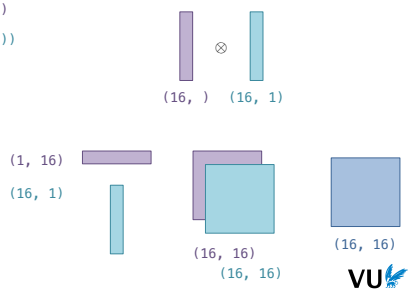
The second (optional) argument is a string that is shown as a part of the error message. Using an f-string here allows you to add some helpful information to the error messages.

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, validating user input.

BROADCASTING: THE SILENT KILLER

```
x = np.ones(shape=(16, ))
y = np.ones(shape=(16, 1))

z = x * y
print(z.shape)
# result: (16, 16)
```



15

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, because it's very easy to make mistakes if you don't fully understand 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)
```

16

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 might 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.

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

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

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

17

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 dimensionality and you know how they will be aligned. Note that in the first example, "`a[:, :, :]`" is the same as just "`a`". However, this notation communicates to the reader what is happening.

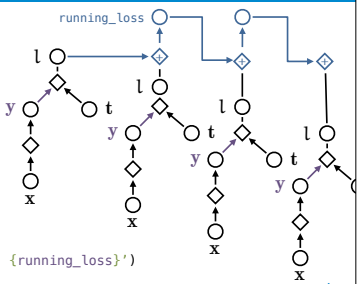
Methods that elminate a dimension (like summing out a dimension, or taking the maximum) come with a `keepdim` argument in pytorch (`keepdims` in numpy). If you set this to `True`, the eliminated dimension remains as a singleton dimension, which ensures that broadcasting will behave reliably.

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}')
```



VU

After each forward and backward, we rely on the python garbage collector to clear up the old computation graph. It can only do this if nothing else refers to it anymore.

If your memory use balloons over several iterations (instead of returning to near 0 after each batch), you might be referencing a node in your computation outside the training loop.

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

VU

In pytorch, the solution is to call .item(). This function works on any *scalar* tensor, and returns a simple float value of it's 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 already a computation graph node, so detach() and .data do pretty much the same thing).

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()
```


VU

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.

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.

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.

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.

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 you can print some statistics for it in the backward pass

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Debugging your model~~

Develop a model, tune hyperparameters

Publish model, or push to production

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.

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.

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.

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.

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.

VU

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 can check the search algorithm.

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.

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 make sure that you know exactly should happen when you execute your code. Inevitably, that then won't happen (we're programming after all), but then at least you'll know that something is wrong, and you'll know where to start looking.

Other tricks

Your model should be able to overfit on a single batch

VU

TUNING THE LEARNING RATE

Fix a batch size first

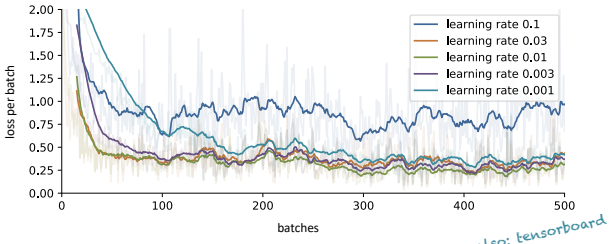
As big as fits in memory is usually reasonable. A little smaller may be better but slower.

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

VU

Your learning rate is *very dependent* on the batch size. You can tune both together, but generally, the big the batch size, the faster your training, so people tend to go for the largest bacth 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.

CHECK YOUR (PER-BATCH) LOSS CURVES



see also: tensorboard

VU

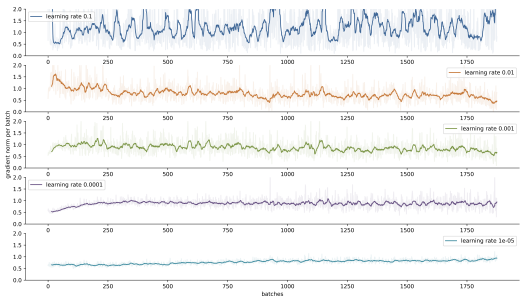
The simplest way to find a reasonable learning rate is to take a small number of logarithmically spaced values, do a short training run with each, and plot their loss per batch.

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 boundes 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: is converges very slowly, and never seems to make it into the region that 0.01 is in.

One popular tool for automatically tracking your loss this way is tensorboard (which originated with tensorflow, but now works with both tensorflow and pytorch).

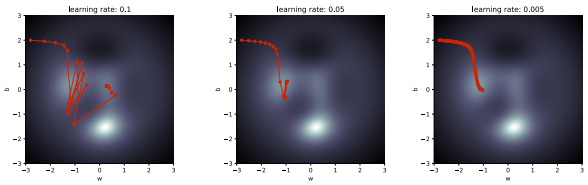
CHECK YOUR GRADIENT NORMS



29

Another helpful thing to plot is the gradient norm (basically an indicator of the sive of step you're taking). If the variance of your gradients goes to zwero, your probably getting stuck in a local minimum. If the variance is very large, your model is probably bouncing over areas of low loss.

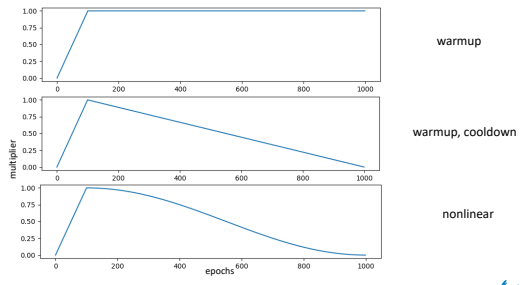
INTERPRETING VARIANCE (IN LOSS OR NORM)



high variance at the start | high variance at the end | low variance throughout

30

LEARNING RATE SCHEDULING



warmup

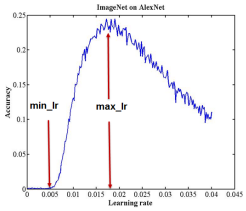
warmup, cooldown

nonlinear

31 source: https://huggingface.co/transformers/v3.2.0/main_classes/optimizer_schedules.html

PROTIP: RANGE TESTING

Ramp up learning rate exponentially *during a single training run*.

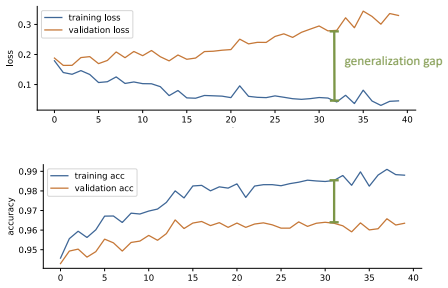


(a) Typical learning rate range test result where there is a peak to indicate max_lr.

32 image source: *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates* Smith et al 2018

One popular trick is to do a *range test*. This is a single training run, where you ramp up the learning rate by small, but exponentially increasing steps *each batch*. It will give you a plot like this. Where the accuracy peaks, is a good estimate of your maximum learning rate. You can then warm up to this learning rate slowly (and possibly cool back down again).

PER-EPOCH LOSS CURVES



33

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 validation set and on the training set. You can then plot the average loss and any other metric you're interested in 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, 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.

STABILIZING, SPEEDUPS

Learning rate warmup, cooldown

Gradient clipping: reduce gradient if it exceeds a threshold.

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

Momentum: more later

Regularization, batch normalization: more later

34

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.

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**.

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

36

VU

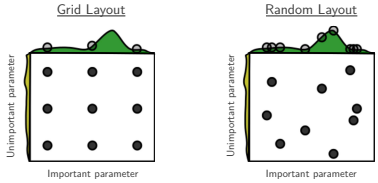
A lot has been written about tuning learning rates, but what about the rest of our hyperparameters? How should we proceed? Just try a bunch of settings that seem right, or should we follow some rigid strategy?

In practice, when it comes to tuning your own model, simple trial and error is usually fine, so long as you do it 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.

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

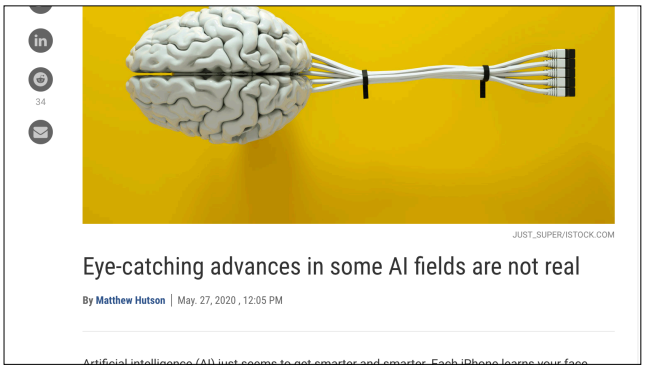
37

VU

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 from 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 form 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. Learning rate is an example.



JUST_SUPER/ISTOCK.COM

Eye-catching advances in some AI fields are not real

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

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 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

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/

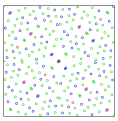


image source: By Jheald - Own work. Created in R.
CC BY-SA 3.0, **https://commons.wikimedia.org/w/index.php?curid=16106862**

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 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

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

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~~

VU

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

Lecture 4: Tools of the trade

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

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.

PART TWO: WHY DOES ANY OF THIS WORK AT ALL?

VU

NEURAL NETWORKS ARE GETTING BIG

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	**125M**	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	**350M**	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	**760M**	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	**1.3B**	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	**2.7B**	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	**6.7B**	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	**13.0B**	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	**175.0B**	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

2.1 Model and Architectures

We use the same model and architecture as GPT-2 [RWC+19], including the modified initialization, pre-normalization, and reversible tokenization described therein, with the exception that we use alternating dense and locally banded sparse attention patterns in the layers of the transformer, similar to the Sparse Transformer [CGRS19]. To study the dependence of ML performance on model size, we train 8 different sizes of model, ranging over three orders of magnitude from 125

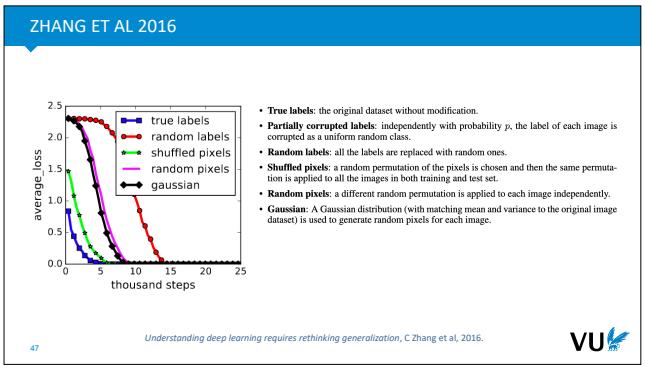
These are the sizes of some versions of GPT-3: a large language model trained by OpenAI. The largest has 175 billion parameters.

This model was trained by nothing more than gradient descent and backpropagation. It's worth pausing to think about that. Imagine if everybody on earth had 20 opinions that you could each individuall sway one way or the other, and you wanted to sway each nd every one of them in order to make one UN resolution pass. Backpropagation is what would allow you, theoretically to reason backwards from the outcome of the vote through the organizational structure of the UN, the governments of the members states, the determination of their

makeup by elections (or other means) down to the individual people and how their opinions go into making up their vote (or their choice whether or not to protest the government).

We don't quite have the full picture for how this is possible, but we are beginning to gather up some of the ingredients.

Language Models are Few-Shot Learners, Brown et al, 2020.
https://arxiv.org/abs/2005.14165



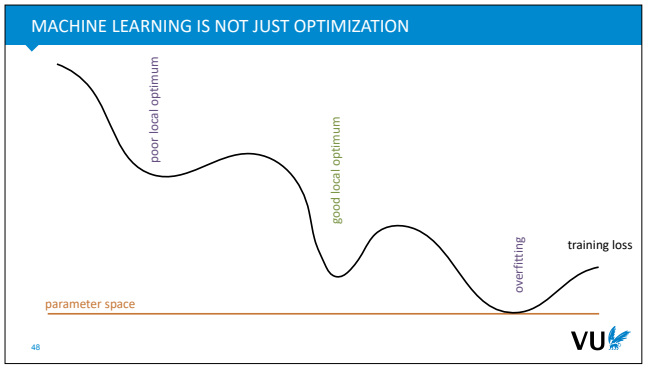
At the start of the deep learning revolution, most researchers were not immediately struck by how odd it was that we can train neural networks with millions of parameters. We were pleasantly surprised, of course, but the main focus was on simply seeing how far this method could be pushed.

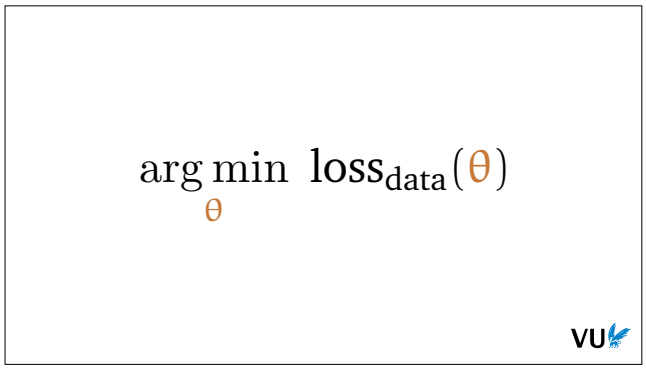
The first clear signal of just how odd this breakthrough was, came from a very simple experiment, performed by Zhang et al. They took a simple convolutional neural network, trained it on MNIST, and then they repeated the experiments with the dataset randomized. They did this in several ways: randomized the labels, replacing the images with gaussian noise, etc. But in each case, they made it impossible to predict from the image what the label could be, while still keeping the possibility of *remembering* the label for a given image.

Of course, the validation loss plummeted to chance level: anything that could be used to predict the label was removed. However, the training loss still went to zero. **The model learned to predict the training data perfectly.** Better than it did when the task was realistic.

Why is this so strange? Note that during training, the model only has access to the training loss. It has access to two points in the parameter space: the generalizing solution that it chooses on normal data, and the overfitting solution that it chooses if we randomize the labels. **And the second solution has lower loss.** Why doesn't the model choose this solution on normal data? Why does it end up with the (from its perspective) worse solution which generalizes well, instead of the perfect solution that doesn't generalize at all? How does it know?

The first clue is in the fact that the generalizing solution is

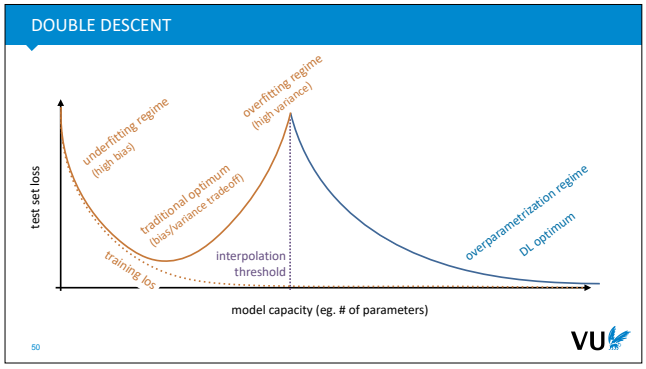




So, next time you see a formula like this, to explain the basic optimization task of ML, you should know that this is a slightly poor definition of our problem. We want to solve this optimization problem, and we use techniques from optimization to do so. But in a way, we don't want to solve it *too* well.

In the classical view, we would keep to this optimization objective, and solve it perfectly, but we would cripple the model class so that the overfitting models were no longer in the search space. In deep learning we expand the model class far beyond the overfitting models. At this point, the only optimization algorithms we have available will only be able to reach a certain part of this space. If we're lucky, these parts contain the generalizing local optima and the overfitting global optima are pushed out of the range of the optimization algorithm.

In short, modern machine learning is often the strange business of letting an optimization algorithm loose on a problem like this, but then crippling it, implicitly or explicitly so that it doesn't find the very best solution, because that wouldn't lead to generalization.



The traditional view of machine learning has always been that the aim was to find the sweet spot between underfitting and overfitting. An underfitting model is too simple to capture all the relations between the input and the output. For instance, if we try to fit a parabola with a line, we get errors due to underfitting.

If we give our model too much capacity, it can overfit: it can remember all the features of the training set that won't be reproduced in the test set. The model starts latching on to noise that isn't relevant for the task: it starts to overfit. One error is called high bias, because the mistakes are the same for every training run. The other is called high variance, because the mistakes average out

over multiple training runs (with fresh data each time), but because we only have one dataset, this doesn’;t help us much.

The point at which the model capacity is large enough to fully remember the dataset (or at least enough of it to remember all the labelings perfectly), is called the interpolation threshold. This is the point beyond which the model should always perfectly overfit without any generalization. That is, assuming that the model search optimizes perfectly.

The traditional solution was always to find the best tradeoff between these two extremes of model capacity. We cripple our model just enough that it find the generalizing solutions because the overfitting solutions are simply not there in the parameter space.

“Double descent” refers to the surprising finding (due to experiments like those of Zhang) that there is life beyond the interpolation threshold. If we push on beyond the regime where we can just about remember the data, into the regime where the capacity is vastly greater than that required to remember the data, we see that some model classes start to naturally prefer generalizing solutions over non-generalizing solutions.

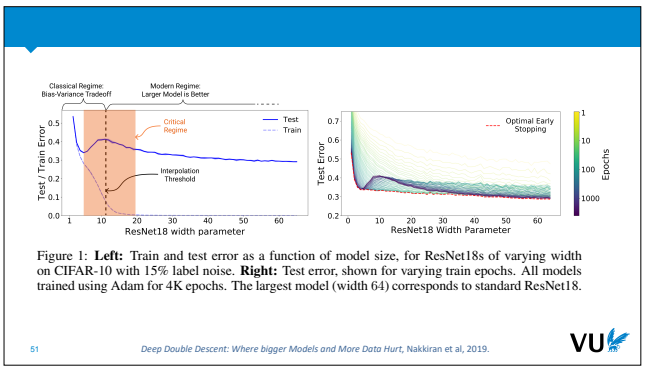
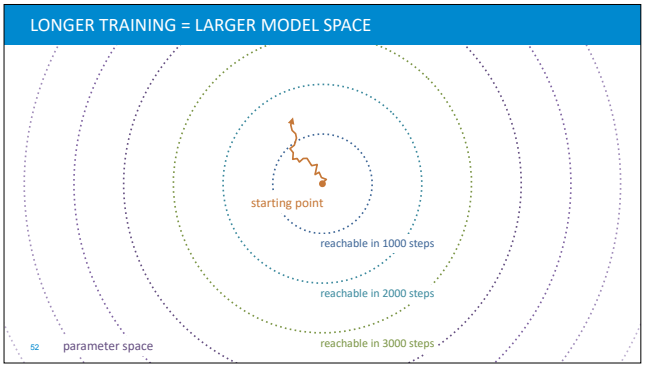


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

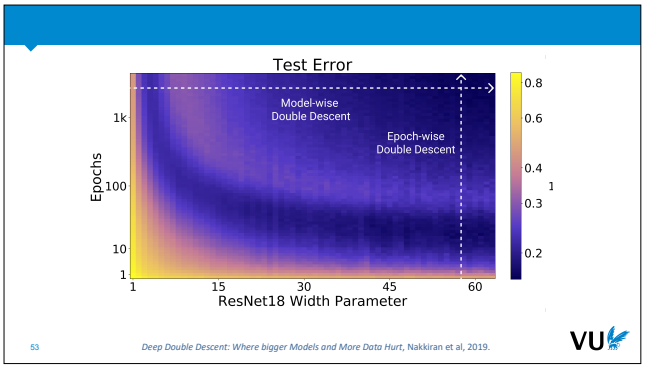
51 *Deep Double Descent: Where bigger Models and More Data Hurt*, Nakkiran et al, 2019.

Here is what such curves actually look like for deep convolutional networks. The curve on the right shows that

https://openai.com/blog/deep-double-descent/



We can also think of the length of training as a cap on the model complexity. Since gradient descent usually has a maximum step size, there is a limit on how far it can reach in a finite number of updates. The longer we train, the bigger our potential model class.



53 *Deep Double Descent: Where bigger Models and More Data Hurt*, Nakkiran et al, 2019.

This means that we can see a double descent phenomenon not just in model complexity (horizontal) but also in training time (vertical).

TAKEAWAYS

The best solutions are suboptimal, *local* minima for the training error.

Finding the *global* optimum is disastrous

Gradient descent has implicit regularization: some parameters are preferred over others, a priori.

More on *explicit* regularization later

Initialization is of *crucial* importance.

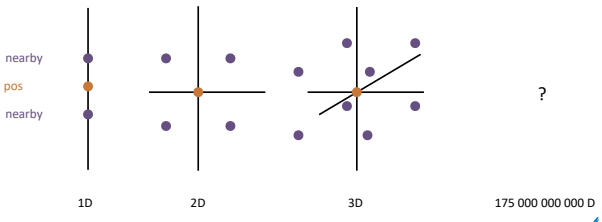
More on this later

54

Note that we're talking about the minima of the training loss *on the dataset*. If we define the loss as a function of the data *distribution* (which we estimate by sampling data), then the global optimum is still the ideal, but the lowest loss we can find for the data is actually a poor estimate, since we get a completely different loss on a new sample.

The mechanism behind the double descent phenomenon seems to be that the more your overparametrize the model, the stronger the effect of the imoplicit regularization of gradient descent becomes.

THE BLESSING OF HIGH DIMENSIONALITY



55

To paraphrase Geoffrey Hinton: "In a sixteen dimensional supermarket, the pizzas can be next to the anchovies, the beer, the cheese and the pasta".

There's not much rigorous work on this, but here's a blog post that goes a little deeper: https://moultano.wordpress.com/2020/10/18/why-deep-learning-works-even-though-it-shouldnt/

OBSERVATION

Network pruning is the practice of removing near-zero connections from a trained neural network.

Pruning works *exceptionally* well.

Often, 85 – 95% of weights can be safely removed.

So, we are training all these vast neural networks, and then it turn out that after training we can remove 90% of the weights.

On the other hand, we can't just train a network with 10% of the weight from the beginning. In that case the performance is noticeably lower. So, we need the large networks to end up with a good model, but the good model is also highly redundant.

How do we explain this?

LOTTERY TICKETS

Traditional view:

- Initialization picks a random model.
- GD teaches each weight what to to.

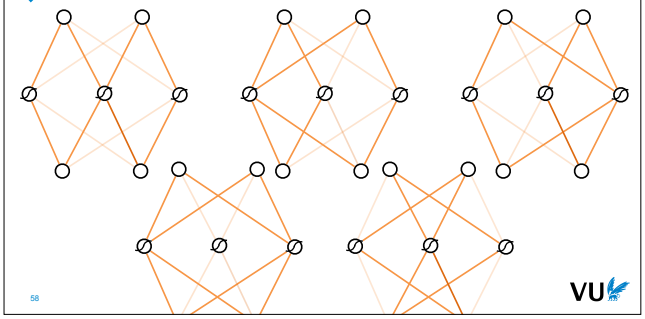
Lottery ticket view:

- Initialization creates combinatorial explosion of *subnetworks*.
- Some of these, by chance perform well.
- GD *selects* these subnetworks and disables others.
- GD finetunes for extra performance.

The lottery ticket hypothesis is a suggestion for the mechanism behind training big neural networksnetworks. The idea is that we aren't actually training the whole of this network, with all these millions or billions of weights each acting in concert.

What is actually happening, is that because we're initializing such

COMBINATORIAL EXPLOSION OF SUBNETWORKS.



Here are just five of the subnetworks of a simple (2, 3, 2) network that might compute a useful function if we set the greyed out weights to zero. There are many more, and the number explodes as the width of the network grows.

In a network with N weights, there are 2^N ways to select a subnetwork. Some of these won't connect the input to the output, but most will.

EXPONENTIAL GROWTH

2^N: subnetworks in a neural net with N weights.

2^{33}: People on Earth
2^{76}: Grains of sand in the Sahara
2^{83}: Molecules in a glass of water
2^{272}: Atoms in the visible universe
2^{408}: Number of possible games of chess
…
$2^{61\,000\,000}$: Number of subnetworks in AlexNet (2012)
$2^{175\,000\,000\,000}$: Number of subnetworks in GPT-3 (2020)

Here we define a *subnetwork* as a network defined by setting some of the parameters to 0.

LOTTERY TICKET HYPOTHESIS

The initialization of a large neural network contains subnetworks (lottery tickets) that, if isolated, already solve the task to near state-of-the-art performance, before any gradient descent is applied.

The power of gradient descent is not in training the model, but in eliminating the dead weight.

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin 2019.

EXPERIMENT 1

1. Train a large Neural Network
2. *Prune* the train neural network to a succesful subnetwork
   basically: kill any weights near 0
3. Revert the pruned network to its precise initialization weights
4. Retrain the pruned network

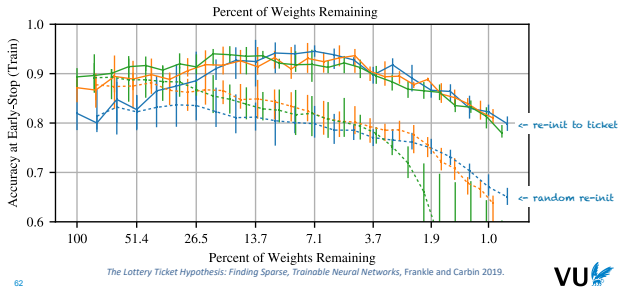
Result:

A small network trained to the performance of a large network.

If we revert to random weights, performance plummets.

VU

UNDER ITERATION



The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin 2019.

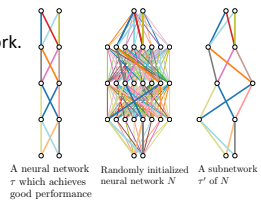
VU

If we iterate this process, we can prune to even lower values. At each error bar in this plot, the network is pruned by a small amount, reset to the initialization values, and retrained. This is compared with resetting to a new random initialization, represented by the dashed line.

EXPERIMENT 2:

1. Initialize a large neural network.
2. Keep the weights fixed.
3. Search for a mask that selects a subnetwork.
   use SGD and gradient estimation (see lecture 9)

Result: The lottery ticket by itself achieves near-SOTA performance.



What's Hidden in a Randomly Weighted Neural Network? Ramanujan et al. 2020

VU

MORE CONCLUSIONS

Re-initializing, but retaining the sign of the original weight is enough to retain performances (Zhou et al 2019).

Initializing with constant values with random sign (+/-) also yields lottery tickets.

VU

https://eng.uber.com/deconstructing-lottery-tickets/

RECAP

Zhang et al: Neural Networks can memorize, but don't.

Double descent: Some models perform best when massively overparametrized.

Lottery ticket hypothesis: The real power of deep learning comes from the combinatorial explosion of subnetworks, more than the ability of SGD to train the model.

Open questions: The last word has not been spoken on these issues.

VU

Lecture 4: Tools of the trade

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

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.

PART THREE: **UNDERSTANDING OPTIMIZERS**

VU

Optimizers are algorithms that take the basic idea of (stochastic) gradient descent and tweak it a little bit, to improve its performance.

JUSTIFYING *STOCHASTIC* GRADIENT DESCENT

$$\arg\min_{\theta} \text{loss}_{\text{data}}(\theta)$$

$$\arg\min_{\theta} \mathbb{E}_{\text{data}\sim p}\text{loss}_{\text{data}}(\theta)$$

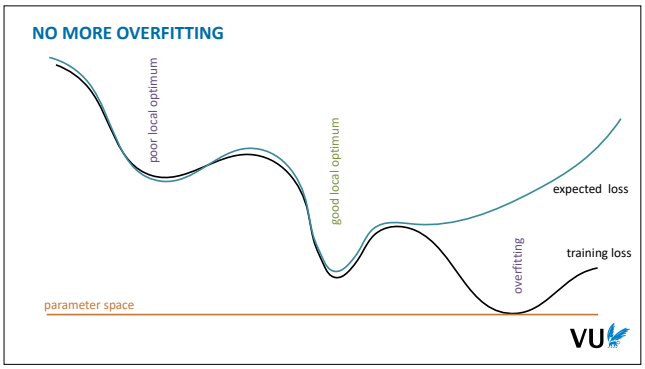
VU

In the first video, we discussed the problem that the optimization that we perform in machine learning is not the optimization that we actually want to solve. This put us in the awkward position of wanting to solve a problem well, but not too well.

We can solve this problem by restating the problem probabilistically. If we assume our data (either an instance, a batch or the whole dataset) is drawn from some distribution p, then what we really want to minimize is the expected loss under that distribution.

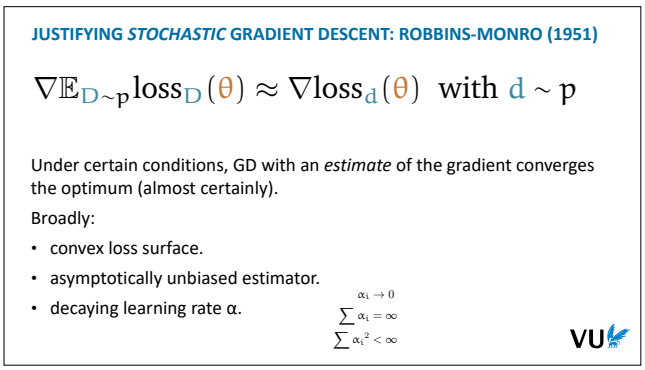
The problem is that this is not something we can compute. The best we can to is take a monte carlo

estimate (average a bunch of samples).



We can define overfitting as optimizing for details that are specific to the sample, but not to the distribution. In other words, memorizing noise in the data that will be complete different for another sample from the data distribution.

This means that if we could somehow optimize for the *expected* loss, we would eliminate all overfitting. The problem is that the expected loss is not a function we have access to (it would require an infinite sum over an infinite amount of data).



Luckily there is a result that tells us we can optimize for a loss we can't compute: the Robbins-Monro algorithm. We estimate the loss and/or its gradient based on some sample, and perform simple gradient

These constraints almost never hold in deep learning, but for some we can assume that they hold locally. For instance, once we get into an area of parameter space where the loss surface is locally convex (and the other constraints hold), we can be sure that SGD will converge to that local optimum. And that local optimum is a local optimum for the loss surface of the expected loss, rather than the data loss.

In most cases, we don't care too much about maintaining the convergence guarantees (even if we decay the learning rate, we rarely do it like this). Ultimately, we prove the quality of our learning empirically, not theoretically

THE LIMITS OF SGD

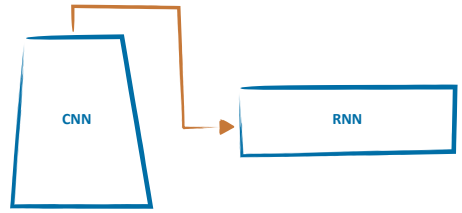


image model, `lr = 0.03`

language model, `lr = 0.00001`

VU

To understand the limits of plain gradient descent as we've seen it, imagine tuning two models: a CNN to analyze images, and a language model RNN to generate sentences (we haven't looked at RNNs yet, but we will soon).

Clearly these are very different models, with different requirements. We can expect them to require very different learning rates. So what should we do if we want to connect a CNN to a language model, and train the whole thing end to end? This is for instance, how you would build a simple image captioning network.

The more heterogeneous a network becomes, the more difficult it is to train it with a single learning rate. Instead, we need a mechanism that is more **adaptive**. Ideally, one that starts with a base learning rate, but adapts that learning rate at every step of training for every parameter individually.

Second-order optimization, conditioning

aka Newton's method

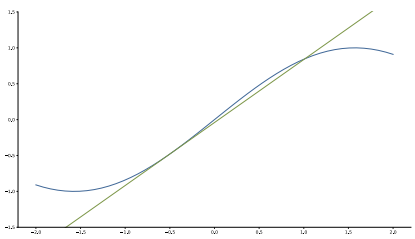
Momentum

Adam

RAdam, LookAhead, LAMB

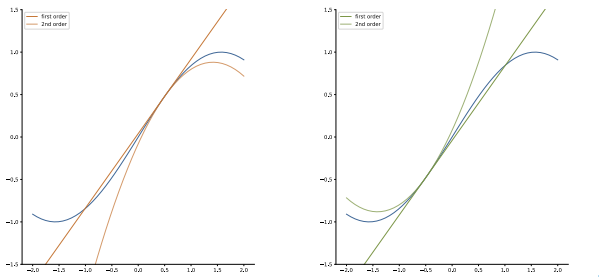
VU

GRADIENT: A LINEAR APPROXIMATION



VU

NON-LINEAR APPROXIMATION



74

If we make our approximating function non-linear, for instance a polynomial of order 2, we can get a better local approximation to our function. For these two points, the linear approximations give pretty similar slopes, but the 2nd order approximations give a much clearer (and very different) hints of how to find the nearest optimum. Essentially, the approximation on the left tells us to take a big step to the left, since the function is (locally) decreasing increasingly fast.

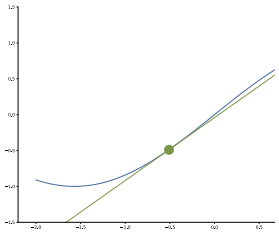
The approximation on the right tells us that we should start reducing our step size, because the minimum is coming up (which means the rate of decrease is decreasing).

BEST LINEAR APPROXIMATION

$$f_a(x) = x\ s + b$$
$$f_a(x) = x\ f'(a) + b$$

$$f_a(a) = a\ f'(a) + b$$
$$b = f(a) - a\ f'(a)$$

$$f_a(x) = x\ f'(a) + f_a(a) - a\ f'(a)$$
$$= f(a) + f'(a)(x - a)$$



75

Let's start with the linear approximation. We're usually only interested in the slopt of this function, but we can work out the complete function as well; we just have to solve for the bias term b.

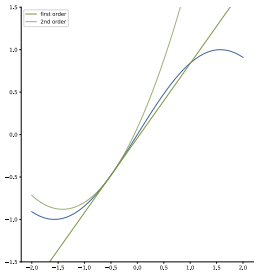
BEST SECOND ORDER APPROXIMATION

$$f_a(x) = c_1 + c_2(x - a) + c_3(x - a)^2$$
$$x = a \ \mapsto \ c_1 = f(a)$$

$$f'_a(x) = c_2 + 2c_3(x - a)$$
$$x = a \ \mapsto \ c_2 = f'(a)$$

$$f''_a(x) = 2c_3$$
$$x = a \ \mapsto \ c_3 = \frac{1}{2}f''(a)$$

$$f_a(x) = f(a) + f'(a)(x - a) + \frac{1}{2}f''(a)(x - a)^2$$



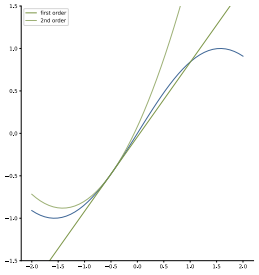
76

NEWTON'S METHOD (1D)

$$f_a(x) = f(a) + f'(a)(x - a) + \frac{1}{2}f''(a)(x - a)^2$$

$$f'_a(a) = f'(a) + f''(a)(x - a) = 0$$
$$x - a = -\frac{f'(a)}{f''(a)}$$
$$x = a - \frac{f'(a)}{f''(a)}$$

$$x \leftarrow x - \alpha\frac{f'(a)}{f''(a)}$$



77

So where does this approximation tell us to go? If we follow the direction of steepest descent, as with the first order optimization, we'll just end up following the gradient.

In this case, however, our approximation is a parabola, with a minimum of its own (unlike the 1st order approximation, which as its minimum at infinity). We can simply work out where this minimum is, and take a step towards it.

To work out the minimum, we take the derivative of our approximation wrt x (note that the purple factors are just constants), we set it equal to 0 and solve for x.

In the result, x is where we want to be and a is where we started. Therefore to move towards our goal, we should subtract from the first derivative of f at a, divided by the second derivative. We add a step size multiplier so we can control for the quality of our approximation.

The step size determining how far we trust the approximation. If we know the approximation is exact, we can just jump directly to the minimum.

NEWTON'S METHOD (ND)

$$x \leftarrow x - \alpha \frac{f'(x)}{f''(x)}$$

$$\theta \leftarrow \theta - \alpha \left[\nabla^2 l(x)\right]^{-1} \nabla l(x)$$

parameter vector — Hessian matrix — gradient of params

78

VU

In n dimensions, we can follow the same process (we'll save you the working out, but it's pretty straightforward with a little vector calculus). The result looks a lot like the 1d case, except that the second derivative is a now a **matrix**. This matrix is called the Hessian: element (i, j) of the Hessian is the derivative wrt to paramater j of the derivative wrt to the parameter i .

The problem in our case is that the number of elements in the Hessian matrix is the square of the number of parameters. If we have a million parameters, the Hessian has 10^{12} elements. Not only is that too big to fit in most memory, we'd have to do an additional backward pass for each element, and then compute its inverse. This is too much work for a single update step.

It does however, give us a good idea of what we're looking for.

IS IT PRACTICAL FOR US?

Newton's method requires:

- NxN matrix
- Accurate estimation (10K batch size)
- Extra backward pass for each element of the gradient (N in total).
- Inversion of that matrix.

Newton's method helps us understand and analyse our problems.

79

VU

Even for a relatively simple neural network of 100K parameters, this is already completely infeasible.

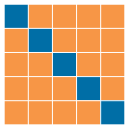
WHAT DOES NEWTON'S METHOD SOLVE?

Parameter interactions: partial derivatives assume independent updates

provided by the off-diagonal elements of the Hessian.

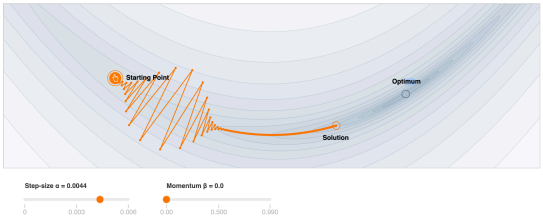
Curvature information: are we nearing a local maximum?

provided by the diagonal elements of the Hessian.



When we compute a partial derivative for a parameter, we assume that all other parameters are constant. We get the optimal update under that assumption (which is the same as assuming linearity; no interaction between the arguments of the function). But then we update all parameters in one step, and

PATHOLOGICAL CURVATURE

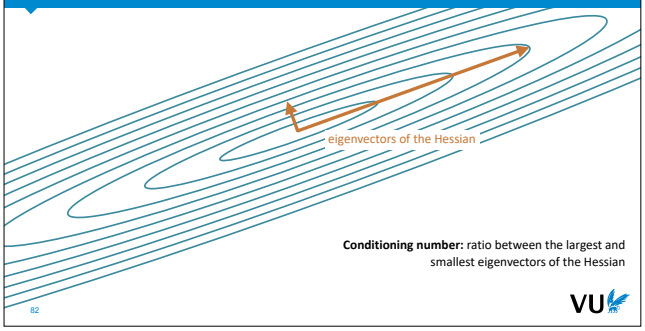


source: https://distill.pub/2017/momentum/

Here's how gradient descent behaves on a surface with a tricky curvature. If we had access to the Hessian, it could tell us two things.

First, in plain gradient descent, we compute partial derivatives: these tell us what happens if we

CONDITIONING



Conditioning number: ratio between the largest and smallest eigenvectors of the Hessian

SO, HOW CAN WE SOLVE THESE PROBLEMS?

Requirements:

- Require one backward pass, use only the gradient.
- only kN extra memory use.
- only O(N) extra computation.

MOMENTUM

$$\mathbf{m} \leftarrow \gamma \mathbf{m} + \mathbf{w}^{\nabla}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{m}$$

$$\gamma : 0.5, 0.9, 0.99$$

VU

THREE VIEWS ON MOMENTUM

- Heavy ball
- Gradient acceleration
- Exponential moving average

VU

HEAVY BALL MOMENTUM

The gradient acts not like a direction, but like a *force*.

- force adds to the velocity
- velocity adds to the position

$$\mathbf{m} \leftarrow \gamma \mathbf{m} + \mathbf{w}^{\nabla}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{m}$$

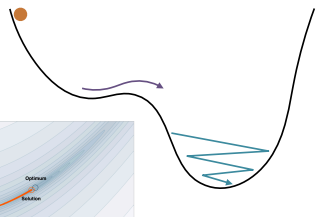
VU

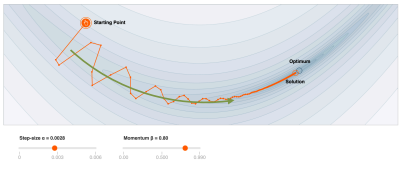
HEAVY BALL MOMENTUM

rolls out of local minima

dampens oscillations

accelerates repeating directions





VU

GRADIENT ACCELERATION

imagine all gradients point in in the same direction **d**:

$\mathbf{w} \leftarrow \mathbf{w} + \alpha\mathbf{d}$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\gamma\mathbf{d} + \mathbf{d})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\gamma^2\mathbf{d} + \gamma\mathbf{d} + \mathbf{d})$

...

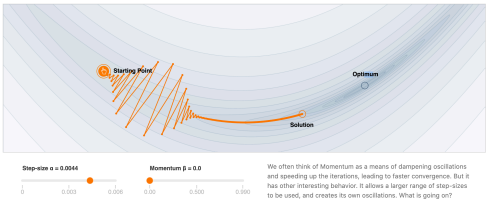
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\mathbf{d}\sum_{n=0}^{\infty}\gamma^n = \mathbf{w} + \alpha\frac{1}{1-\gamma}\mathbf{d}$$

$\gamma = 0.99 \;\mapsto\; 100\times$ acceleration

VU

EXPONENTIAL MOVING AVERAGE

Averaging gradients helps to stabilize.



VU

We could store up a bunch of

EXPONENTIAL MOVING AVERAGE

$\mathbf{x}_1, \ldots, \mathbf{x}_n$

$$\begin{aligned}\text{EMA}_n &= \kappa\mathbf{x}_n + (1-\kappa)\text{EMA}_{n-1} \quad \text{with } \text{EMA}_0 = 0\\ &= \kappa\mathbf{x}_n + (1-\kappa)\left(\kappa\mathbf{x}_{n-1} + (1-\kappa)\text{EMA}_{n-2}\right)\\ &= \kappa\mathbf{x}_n + \kappa(1-\kappa)\mathbf{x}_{n-1} + (1-\kappa)^2\text{EMA}_{n-2}\\ &= \kappa\mathbf{x}_n + \kappa(1-\kappa)\mathbf{x}_{n-1} + \kappa(1-\kappa)^2\mathbf{x}_{n-2} + (1-\kappa)^3\text{EMA}_{n-3}\end{aligned}$$

$\gamma = 1 - \kappa$

$\text{EMA}_n/(1-\gamma) = \mathbf{x}_n + \gamma\mathbf{x}_{n-1} + \gamma^2\mathbf{x}_{n-2} + \gamma^3\mathbf{x}_{n-3} + \ldots$

VU

An exponential moving average is a weighted average of a sequence that is easy to compute on the fly. When we see element x_n come in, we simply take the average of x_n and our previous average, with the new value counting for some proportion kappa.

In other words, the momentum **m** can be seen as an exponential moving average of the recently observed gradients (it's scaled by a constant factor, but then we mulitply it by an arbitary learning rate anyway).

MINIBATCHING IS ALSO AVERAGING

B : minibatch of instances $\mathbf{x}$

$$\nabla_{\mathbf{w}}\frac{1}{|\mathrm{B}|}\sum_{\mathbf{x}\in\mathrm{B}}\text{loss}_{\mathbf{x}}(\mathbf{w}) = \frac{1}{|\mathrm{B}|}\sum_{\mathbf{x}\in\mathrm{B}}\nabla_{\mathbf{w}}\text{loss}_{\mathbf{x}}(\mathbf{w})$$

VU

What does this mean? It means that doing one step of gradient descent for the loss over a minibatch, is equivalent to doing single-instance gradient descent, but summing the gradient that we get and holding off on the descent step until we've seen all gradients in the batch.

With this perspective, we can think of momentum as giving us the best of both worlds: because we use an exponential moving average, we can do an update after each instance (or small batch), but we also get the stabilizing and accelerating effects of large batch training.

MOMENTUM

- N extra memory
- N extra operations
- One extra hyperparameter to tune (γ)

N: number of weights

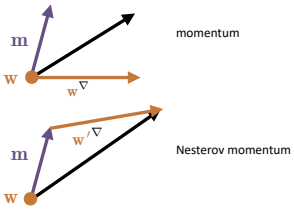
- Potential *quadratic* speedup in convergence.
- Per-parameter tuning of behavior (each param gets its own momentum)
- Much more to be said: **https://distill.pub/2017/momentum/**

VU

NESTEROV MOMENTUM

Compute gradient where you *will be*, not where you are.

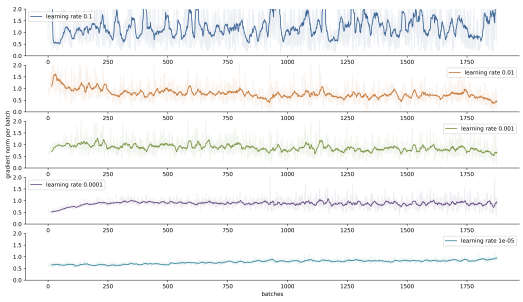
$$\mathbf{w}' \leftarrow \mathbf{w} + \alpha\mathbf{m}$$
$$\mathbf{m} \leftarrow \gamma\mathbf{m} + \mathbf{w}'^{\nabla}$$
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\mathbf{m}$$



see also: https://cs231n.github.io/neural-networks-3/#sgd

VU

REMEMBER THE *VARIANCE*

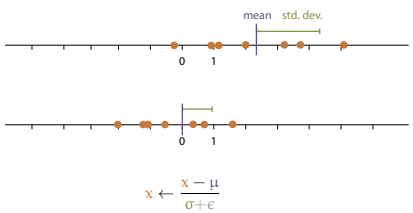


'U

But it's not just the average of our gradients that can tell us something. Remember in the first lecture of our video we noted that if the variance of our gradients is high, we're probably bouncing around some bowl in the loss surface (at least for that parameter), so we want to lower the learning rate. If the variance is low, we've settled on a local minimum, and (if that minimum isn't good enough) we had better boost the learning rate a little.

This tells us that it's not just helpful to use the average of a bunch of suggestions for the gradient, we may get a boost from normalizing the variance as well.

NORMALIZATION



$$x \leftarrow \frac{x - \mu}{\sigma + \epsilon}$$

VU

ADAM: EXPONENTIAL MOVING *NORMALIZATION*

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1)\, \mathbf{w}^{\nabla}$$

$$\mathbf{v} \leftarrow \beta_2 \mathbf{v} + (1 - \beta_2)\, \left(\mathbf{w}^{\nabla}\right)^2 \quad \text{<- element-wise}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\mathbf{m}}{\sqrt{\mathbf{v}} + \epsilon}$$

VU

In Adam, we compute not only an exponential moving average (as momentum does), but also an exponential moving average of the (non-centralized) second moment of each parameter (think variance).

BIAS CORRECTION

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1)\, \mathbf{w}^{\nabla}$$

$$\mathbf{v} \leftarrow \beta_2 \mathbf{v} + (1 - \beta_2)\, \left(\mathbf{w}^{\nabla}\right)^2$$

$$\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^{\,t}} \quad \text{<- steps so far}$$

$$\mathbf{v} \leftarrow \frac{\mathbf{v}}{1 - \beta_2^{\,t}}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\mathbf{m}}{\sqrt{\mathbf{v}} + \epsilon}$$

VU

In the first steps of our iteration, moving averages are very biased towards our initial values. These are arbitrariluy set to 0, underestimating the gradient. For **m** this leads to slightly smaller steps than desired, which is not a problem, but for **v** this can lead to larger steps by a serious amount.

To correct for this, the authors introduced the following bias correction. Note that this correction weighs heavily in the first steps and then dies out quickly.

ADAM

- 2N extra memory
- 2N extra operations
- Two extra hyperparameter to tune (β_1, β_2)
  defaults are usually fine, and the learning rate becomes *much* easier to tune.

- No convergence guarentees.
- Per-parameter tuning of behavior
- Currentlys the default optimizer for most DL settings

VU

PRACTICAL ADVICE

Newton's method doesn't work for deep learning, but it's great in other settings.

Start with Adam, with learning rates between 0.1 and 0.00001.
defaults are usually fine for β_1, β_2

Consider plain SGD with (Nesterov) momentum.

NEW KIDS ON THE BLOCK: RECTIFIED ADAM

Learning rate warmup is often an important trick.

Adam must underestimate the early-training variance.

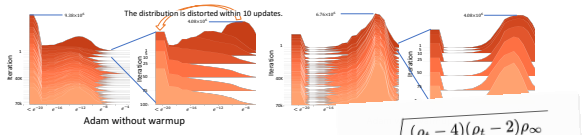


Figure 2: The absolute gradient histogram of the Transform during the training (stacked along the y-axis). X-axis is at height is the frequency. Without warmup, the gradient distribu

$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}.$$

On the variance of the adaptive learning rate and beyond. Liu et al, ICLR 2020

VU

NEW KIDS ON THE BLOCK: LOOKAHEAD (2019)

LookAhead

Two models: w, v. Train w normally (by any optimizer), periodically push w towards v.

When Nesterov meets gradient accumulation.

VU

Note that lookahead can be combined with any other optimization method. Compare this to momentum and gradient accumulation: it's very similar in that it essentially averages a number of gradients. However, within that averaging it applies a kind of Nesterov trick: it uses these gradients to do a lookahead of where it's likely to go, and evaluates each new gradient based on the old one.

NEW KIDS ON THE BLOCK: LARS(2017), LAMB(2020)

LARS: Tune the learning rate *per layer*.

$\mathbf{w}_l$: weights of layer l

$$r_l = \frac{\|\mathbf{w}_l\|}{\|\mathbf{w}_l^{\nabla}\|}$$

LAMB: Tune per layer with LARs, update with Adam.

And many other tweaks

Large batch optimization for deep learning: training BERT in 76 minutes, You et al. ICLR 2020

VU

Lecture 4: Tools of the trade

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

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.

PART FOUR: **THE BAG OF TRICKS**

VU

initialization, normalization

- Glorot, He
- Batch Norm, group norm, layer norm

regularization

- L1, L2, weight decay
- Dropout, priors

other tricks

- data augmentation, transfer learning

VU

INITIALIZATION

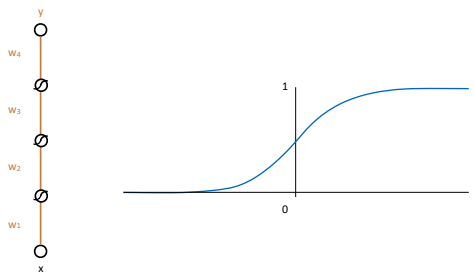
If the gradients are zero at the first batch, training never starts

If they're near zero, training starts very slowly

If the gradiesnt blow up, we get NaN

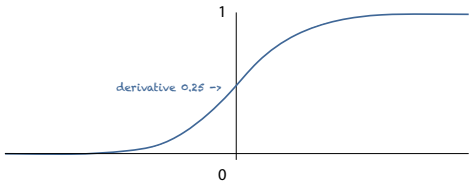
Initial weights should be randomly chosen in a way that keeps gradients *consistent* throughout the network.

VU



VU

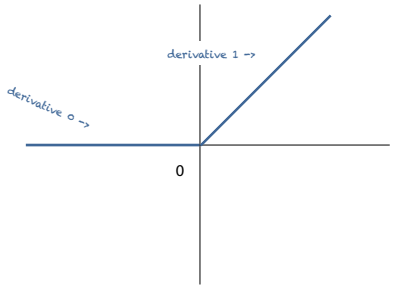
SIGMOID



Even if the value going in to the sigmoid is close enough to zero, we still end up with a derivative of only one quarter. This means that propagating the gradient down the network, it will still go to zero with many layers.

We could fix this by squeezing the sigmoid, so its derivative is 1, but it turns out there is a better and faster solution that doesn't have any of these problems.

RELU



The rely activation preserves the derivatives for the nodes whose activations it lets through. It kills it for the nodes that produce a negative value, of course, but so long as your network is properly initialised, about half of the values in your batch will always produce a positive input for the ReLU.

There is still the risk that during training, your network will move to a configuration where a neuron always produces negative input for every instance in your data. If that happens, you end up with a dead neuron: its gradient will always be zero and no weights below that neuron will change anymore (unless they also feed into a non-dead neuron).

We can avoid dead neurons by normalization: if all the inputs look standard-normally distributed, then half of them will always get a gradient.

GOOD INITIALIZATION

Make sure your input data is normalized: $\mathbf{0}$ mean, covariance $\mathbf{I}$

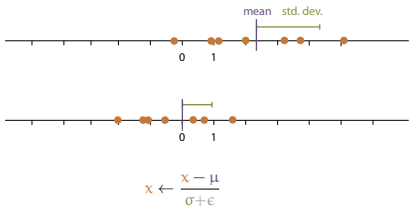
uniform over [0, 1] is usually fine too

Initialize your layer weights so that if the input has mean $\mathbf{0}$, covariance $\mathbf{I}$, then the output does too. Same for the **backward** function.

bias is easy: just init to 0 or close to zero.

- Glorot Initialization (aka Xavier init)
- He initialization (aka Kaiming init)

NORMALIZATION



$$x \leftarrow \frac{x - \mu}{\sigma + \epsilon}$$

We can take some in

GLOROT: ASSUME LINEAR LAYERS

$$\mathbf{y} = \mathbf{W}\mathbf{x} \quad \text{with } \mathbf{W} \in \mathbb{R}^{n \times m}$$
$$\mathbf{x}^{\nabla} = \mathbf{W}^{\mathsf{T}}\mathbf{y}^{\nabla}$$

assume $\mathrm{Var}(x_i) = 1$
choose $\mathrm{Var}(W_{ij}) = c, \mathrm{Exp}(W_{ij}) = 0$
require $\mathrm{Var}(x_i) = 1$

$$\begin{aligned}\mathrm{Var}(y_i) &= \mathrm{Var}\sum_k W_{ik}x_k = \sum_k \mathrm{Var}(W_{ik}x_k) \\ &= \sum_k \left(\mathrm{Var}\, W_{ik} \cdot \mathrm{Var}\, x_k + (\mathrm{Exp}\, W_{ik})^2 \mathrm{Var}\, x_k + (\mathrm{Exp}\, x_k)^2 \mathrm{Var}\, W_{ik}\right) \\ &= m \cdot c \\ \mathrm{Var}(x_i^{\nabla}) &= n \cdot c \\ \mathrm{Var}\, W_{ij} \approx \frac{1}{n} \approx \frac{1}{m} &\mapsto \frac{1}{\frac{1}{2}(n+m)}\end{aligned}$$

$$W_{ij} \sim \mathrm{N}\left(0, \sqrt{\frac{2}{n+m}}\right)$$
$$W_{ij} \sim \mathrm{U}\left(-\sqrt{\frac{6}{n+m}}, \sqrt{\frac{6}{n+m}}\right)$$

Note that Glorot init doesn't take the effect of the

HE INITIALIZATION

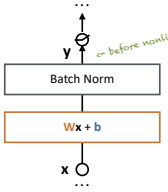
If we use a ReLU activation,we expect to lose *half* our outputs, so we need to change c to *double* the output variance.

$$W_{ij} \sim \mathrm{N}\left(0, \frac{2}{\sqrt{n+m}}\right)$$
$$W_{ij} \sim \mathrm{U}\left(-\sqrt{\frac{12}{n+m}}, \sqrt{\frac{12}{n+m}}\right)$$

If we multiply the standard deviation by sqrt(2), we double the variance.

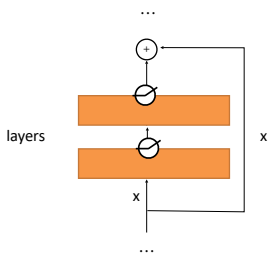
BATCH NORMALISATION

$x_1, \ldots, x_m$: output *batch* of previous layer
$y_1, \ldots, y_m$: *batch* result
γ, β : learnable parameter vectors



$$\mu = \frac{1}{m}\sum x_i \quad \text{mean over batch}$$
$$\sigma = \frac{1}{m}\sum (x_i - \mu)^2 \quad \text{variance over batch}$$
$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma + \epsilon}} \quad \text{standardize}$$
$$y^i = \gamma^{\mathsf{T}}\hat{x}_i + \beta \quad \text{rescale}$$

RESIDUAL CONNECTIONS



DANGER: LEAKAGE

During inference, **we should only look at one instance at a time**.

Using batch information is looking forward in the test data.

Solution:

- Take the training set mean and standard deviation.
- Use EMA for the mean and standard deviation.

This means your network needs to know if it's *training* or *predicting*.

LAYER, INSTANCE, GROUP NORMALIZATION

Same as batch norm, but over different subsets of the batch tensor.

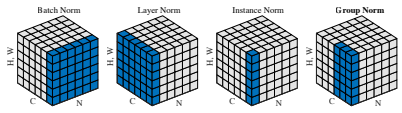


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Batch norm tends to work best **if**

- you have a large enough batches
- your instances are i.i.d.

image source: Group Normalization, Wu and He, 2018

initialization, normalization

- Glorot, He
- Batch Norm, group norm, layer norm

regularization

- L1, L2, weight decay
- Dropout, priors

other tricks

- data augmentation, transfer learning

REGULARIZATION

Encoding a preference for certain parameters over others, *independent of the data* (a priori).

Implicit regularization: initialization, choice of optimizer, etc.

Explicit regularization:

- penalty terms
- priors
- dropout

VU

PENALTY TERM: LP REGULARIZER

$$\text{loss}_{\text{reg}} = \text{loss} + \lambda\|\theta\|$$

parameter space

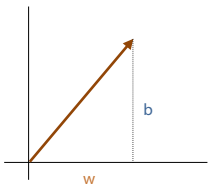
VU

The l2 regularizer considers models with small parameters to be simpler (and therefore preferable). It adds a a penalty to the loss for models with larger weights.

To implement the regularizer we simply compute the l2 norm of all the weights of the model (flattened into a vector). We then add this to the loss multiplied by hyper parameter lambda. Thus, models with bigger weights get a higher loss, but if it's worth it (the original loss goes down enough), they can still beat the simpler models.

Theta is a vector containing all parameters of the model (it's also possible to regularise only a subset).

VECTOR NORM



$$\theta = \begin{pmatrix} w \\ b \end{pmatrix}$$

$$\|\theta\| = \sqrt{w^2 + b^2}$$

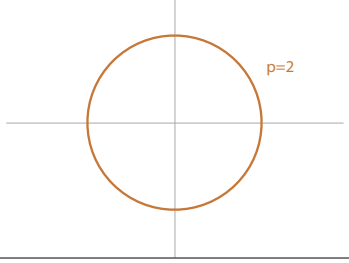
$$\|\theta\|_p = \sqrt[p]{w^p + b^p}$$

VU

We can generalise the l2 norm to an lp norm by replacing the squares with some other number p.

L2 NORM

$$\|\theta\|_p = \sqrt[p]{w^p + b^p}$$

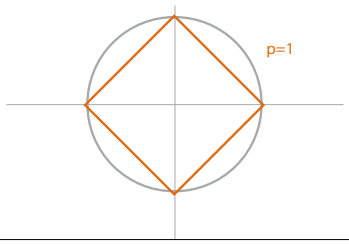


VU

For the l2 norm, the set of all points that have distance 1 to the origin form a circle.

L1 NORM

$$\|\theta\|_p = \sqrt[p]{w^p + b^p}$$

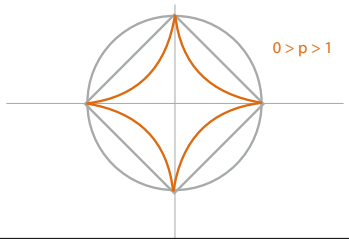


123

VU

For the l1 norm, the form a diamond.

LP NORM

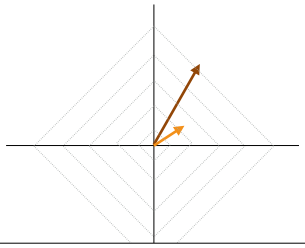


124

VU

L1 REGULARIZER

$$\text{loss} \leftarrow \text{loss} + \lambda||\theta||^1$$



125

VU

This means that for low p values (p=1 is most common), we get

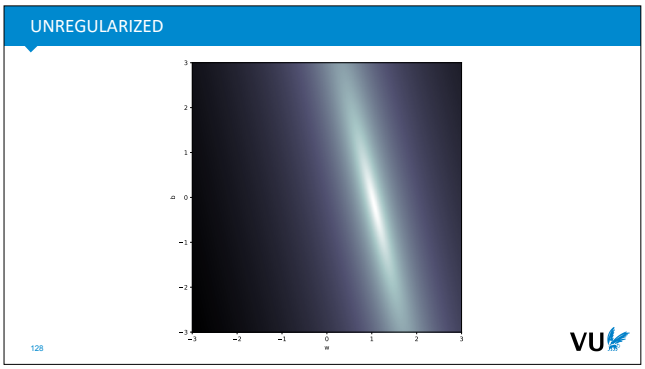
more sparse weight matrices: that is, if a weight is close to 0, it's more likely to get set entirely to zero.

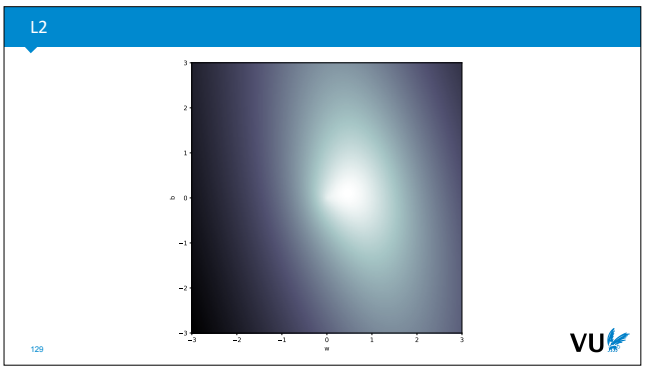
126

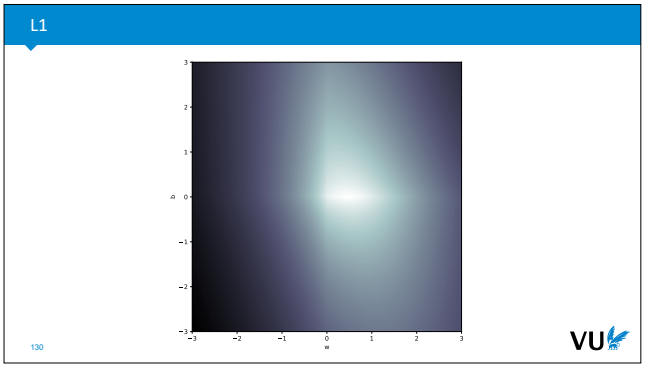
VU

Here's an analogy. Imagine you have a bowl, and you roll a marble down it to find the lowest point. Applying l2 loss is like tipping the bowl slightly to the right. You shift the lowest point in some direction (like to the origin).

L1 loss is like using a square bowl. It has grooves along the dimensions, so that when you tip the bowl, the marble is likely to end up in one of the grooves.







L2 regularization: often uses squared norm $\mathbf{w}^\mathsf{T}\mathbf{w}$ as penalty term

For computational simplicity, and ease of analysis.

L1 regularization: promotes sparsity

VU

WEIGHT DECAY

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla_{\mathbf{w}}(\text{loss}(\mathbf{w}) + \lambda\|\mathbf{w}\|^2)$$
$$= \mathbf{w} - \alpha\nabla\text{loss}(\mathbf{w}) - \lambda\nabla\sum_i w_i^2$$
$$= \mathbf{w} - \alpha\nabla\text{loss}(\mathbf{w}) - \lambda 2\mathbf{w}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\mathbf{w}^\nabla$$
$$\mathbf{w} \leftarrow \gamma\mathbf{w}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla\text{loss}(\mathbf{w})$$
$$\mathbf{w} \leftarrow \mathbf{w} - \lambda 2\mathbf{w} = (1 - \lambda 2)\mathbf{w}$$

VU

WEIGHT DECAY

Equivalent to (squared norm) L2 regularization, **but only with vanilla SGD**.

Cheap to compute: no extra nodes in the computation graph required.

With different optimizers, weight decay must be implemented differently.

cf Adam and AdamW

VU

PRIORS AND REGULARIZERS

$$\arg\max_{\mathbf{w}} \mathrm{p}_{\mathbf{w}}(\mathbf{x})\mathrm{p}(\mathbf{w})$$
$$= \arg\min_{\mathbf{w}} -\log\ \mathrm{p}_{\mathbf{w}}(\mathbf{x})\mathrm{p}(\mathbf{w})$$
$$= \arg\min_{\mathbf{w}} -\log\ \mathrm{p}_{\mathbf{w}}(\mathbf{x}) - \log\mathrm{p}(\mathbf{w})$$

base loss — penalty

$$-\log \mathrm{N}(\mathbf{w} \mid \mathbf{0}, \mathbf{I}) = -\log\left[\frac{1}{\sqrt{(2\pi)^k \mid \mathbf{I}\mid}}\exp\left(-\frac{1}{2}(\mathbf{w}-\mathbf{0})^\mathsf{T}\mathbf{I}^{-1}(\mathbf{w}-\mathbf{0})\right)\right] \mapsto -\log \mathbf{w}^\mathsf{T}\mathbf{w}$$

VU

PENALTY WEIGHT

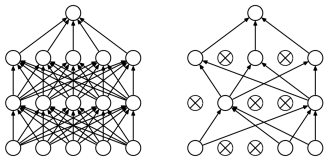
$$\arg\max_{\mathbf{w}} \; p_{\mathbf{w}}(\mathbf{x})\, p^{\alpha}(\mathbf{w}) \qquad \text{with } p^{\alpha}(\mathbf{w}) = \frac{p(\mathbf{w})^{\alpha}}{\int_{\mathbf{v}} p(\mathbf{v})^{\alpha}}$$

$$= \arg\min_{\mathbf{w}} -\log p_{\mathbf{w}}(\mathbf{x}) - \log p(\mathbf{w})^{\alpha} + \log \int_{\mathbf{v}} p(\mathbf{v})^{\alpha}$$

$$= \arg\min_{\mathbf{w}} -\log p_{\mathbf{w}}(\mathbf{x}) - \alpha \log p(\mathbf{w})$$

VU

DROPOUT



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

source: Dropout: A Simple Way to Prevent Neural Networks from Overfitting Srivastava et al, JMLR 2014

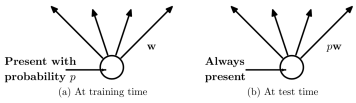
VU

Dropout is another regularization technique for large neural nets. During training, we simply remove hidden and input nodes (each with probability p).

This prevents co-adaptation. Memorization (aka overfitting) often depends on multiple neurons firing together in specific combinations. Dropout prevents this.

image source: http://jmlr.org/papers/v15/srivastava14a.html

DROPOUT



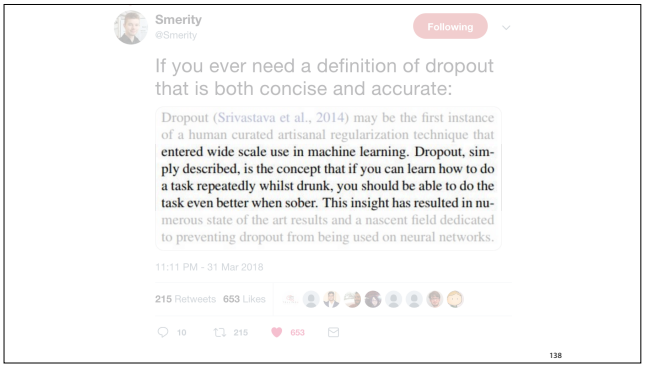
(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

VU

Once you've finished training and you starting using the model, you turn off dropout. Since this increases the size of the activations, you should correct by a factor of p.

Frameworks like Keras know when you're using the model to train and when you're using it to predict, and turn dropout on and off automatically.



https://twitter.com/Smerity/status/980175898119778304

initialization, normalization

- Glorot, He
- Batch Norm, group norm, layer norm

regularization

- L1, L2, weight decay
- Dropout, priors

other tricks

- data augmentation, transfer learning

VU

DATA AUGMENTATION

Simple random manipulations of your input

most common in image tasks

Rotation, flipping, adding noise, masking portions.

- Forces your network to learn the invariance that it doesn't posess naturally.
- Reduces overfitting: never the same input twice.

But: some invariances can harm your performance.

label: 9 label: 9

VU

TRANSFER LEARNING

Some models extract features that work well for other domains.

1. Train a large model to classify ImageNet or predict tokens in NL
   Inception, ResNet, VGG, MobileNet, GPT-2, BERT
2. Remove the last layer
3. Add a new classification layer, train only this layer.

Only the last layer requires gradients

state of the art performance, at the cost of a linear model

VU

LECTURE RECAP

The basic process of training a model. Designing implementing, debugging, tuning, publishing.

Why does deep learning work at all? Randomization, double descent, lottery tickets.

Optimizers. Newton's, momentum, Adam.

The toolbox: initialization, normalization, regularization.

VU

THANK YOU FOR YOUR ATTENTION

dlvu@peterbloem.nl