Lecture 4: Tools of the trade

Peter Bloem
Deep Learning

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

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

PART ONE: DEEP LEARNING IN PRACTICE

THE GENERAL TIMELINE

Pick a task, get some data

Debugging your model

Develop a model, tune hyperparameters

Publish model, or push to production

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).

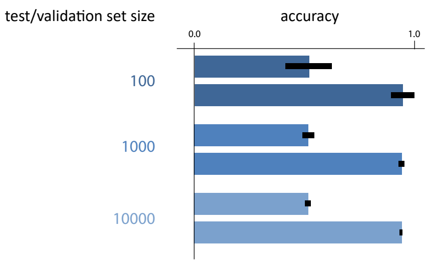
Whatever is left over is your **training data**.

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

HOW MUCH DATA DO YOU NEED?

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

CONFIDENCE INTERVALS



HOW MUCH DATA DO I NEED?

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

10 000 instances is a good aim

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

half the size of test is fine

The rest is your **training data**

If your dataset is just too small:

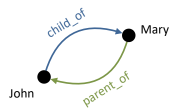
- Consider not using machine/deep learning
- Find lots of *unlabeled* data: self/semi-supervised learning
- For evaluation: combined 5x2 cross-validation F-testing (Alpaydin '99)

DO NOT USE YOUR TEST SET MORE THAN ONCE.

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)

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.

THE GENERAL TIMELINE

~~Pick a task, get some data~~

Debugging your model

Develop a model, tune hyperparameters

Publish model, or push to production

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*

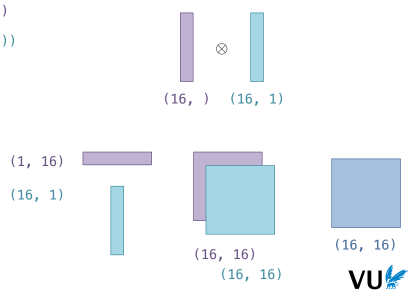
ASSERT

```
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.

BROADCASTING: THE SILENT KILLER

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

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



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)
```

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.

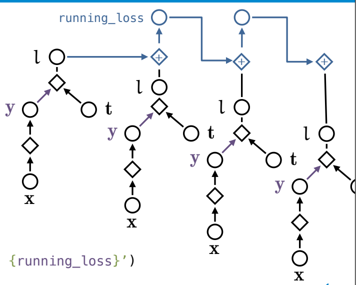
```
b, c, h, w = input.size()
```

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

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


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}')
```



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

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()
```

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.

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.

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.

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

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

Other tricks

Your model should be able to overfit on a single batch

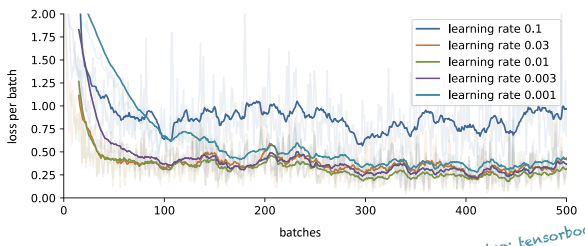
TUNING THE LEARNING RATE

Fix a batch size first

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

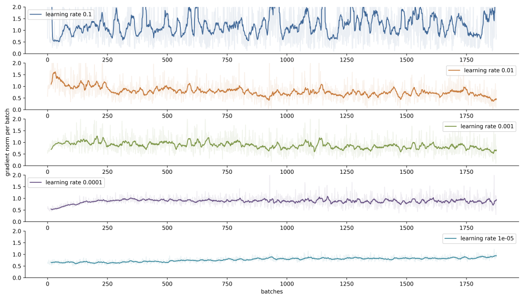
Standard: try `0.1, 0.01, 0.001, 0.0001, 0.00001` for a few epochs each. Compare per-batch loss curves.

CHECK YOUR (PER-BATCH) LOSS CURVES

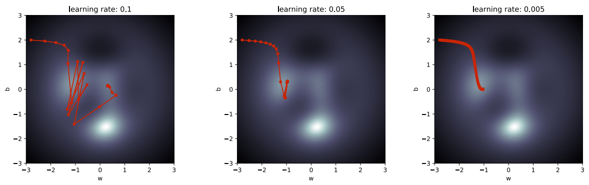


see also: tensorboard

CHECK YOUR GRADIENT NORMS



INTERPRETING VARIANCE (IN LOSS OR NORM)

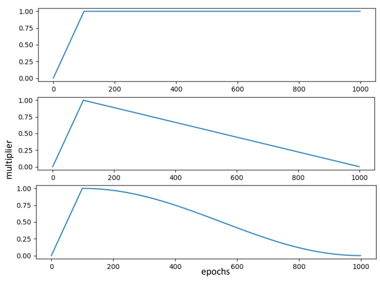


high variance at the start

high variance at the end

low variance throughout

LEARNING RATE SCHEDULING



warmup

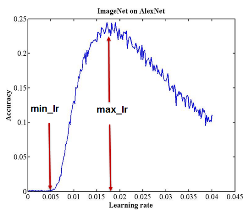
warmup, cooldown

nonlinear

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

PROTIP: RANGE TESTING

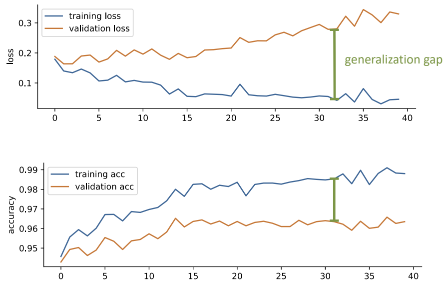
Ramp up learning rate exponentially *during a single training run*.



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

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

PER-EPOCH LOSS CURVES



STABILIZING, SPEEDUPS

Learning rate warmup, cooldown

Gradient clipping: reduce gradient if it exceeds a threshold.

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

Momentum: more later

Regularization, batch normalization: more later

SIMPLICITY CAN BE MORE MEANINGFUL THAN ACCURACY

TUNING STRATEGIES: TRIAL AND ERROR

Usually good enough.

Easy to use model insights.

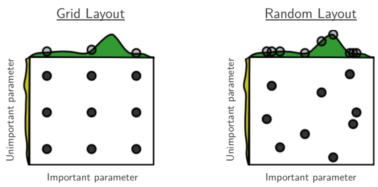
You know what your hyperparameters mean.

Difficult to do fairly.

Nobody tunes their baselines as much as their own model.

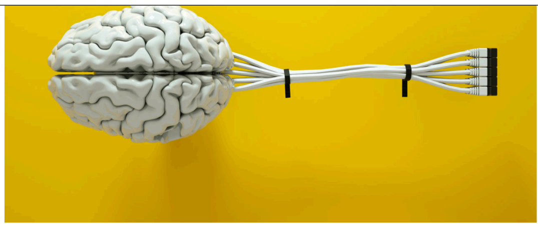
AUTOMATIC TUNING: GRID SEARCH VERSUS RANDOM SEARCH

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



JUST_SUPER/ISTOCK.COM

Eye-catching advances in some AI fields are not real

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

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

---

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/

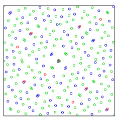


image source: By Jheald - Own work. Created in R.
CC BY-SA 3.0, **https://commons.wikimedia.org/w/index.php?curid=16106862**

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Debugging your model~~

~~Develop a model, tune hyperparameters~~

Publish model, or push to production

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

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!

THE GENERAL TIMELINE

~~Pick a task, get some data~~

~~Debugging your model~~

~~Develop a model, tune hyperparameters~~

~~Publish model, or push to production~~

Lecture 4: Tools of the trade

Peter Bloem
Deep Learning

dlvu.github.io

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

NEURAL NETWORKS ARE GETTING BIG

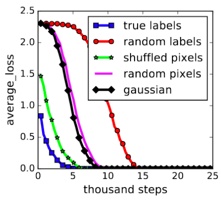
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	**125M**	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	**350M**	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	**760M**	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	**1.3B**	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	**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

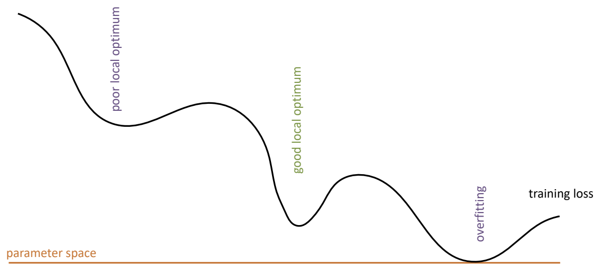
ZHANG ET AL 2016



- **True labels**: the original dataset without modification.
- **Partially corrupted labels**: independently with probability p, the label of each image is corrupted as a uniform random class.
- **Random labels**: all the labels are replaced with random ones.
- **Shuffled pixels**: a random permutation of the pixels is chosen and then the same permutation is applied to all the images in both training and test set.
- **Random pixels**: a different random permutation is applied to each image independently.
- **Gaussian**: A Gaussian distribution (with matching mean and variance to the original image dataset) is used to generate random pixels for each image.

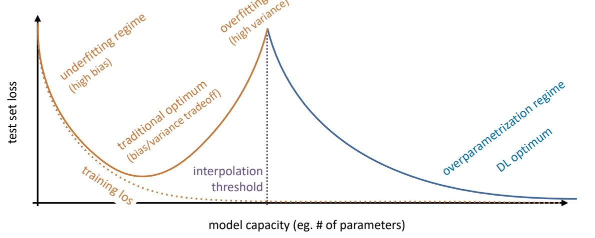
Understanding deep learning requires rethinking generalization, C Zhang et al, 2016.

MACHINE LEARNING IS NOT JUST OPTIMIZATION



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

DOUBLE DESCENT



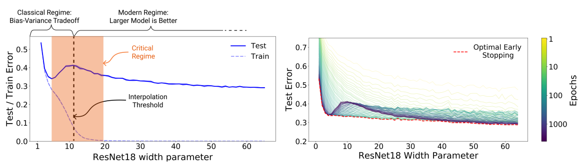
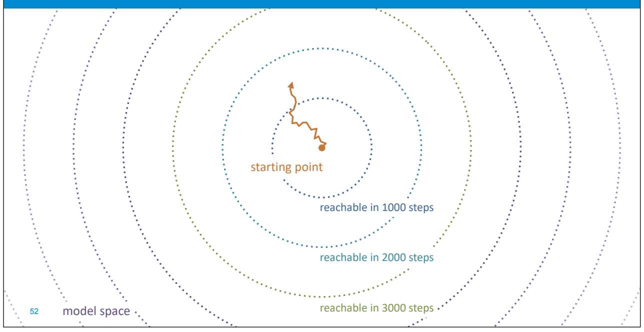
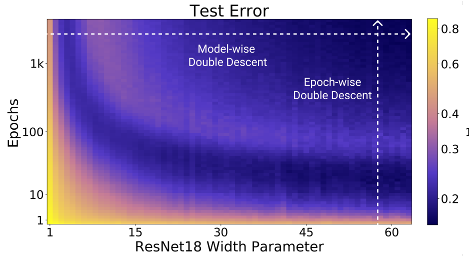


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

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

LONGER TRAINING = LARGER MODEL SPACE





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

TAKEAWAYS

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

Finding the *global* optimum is disastrous

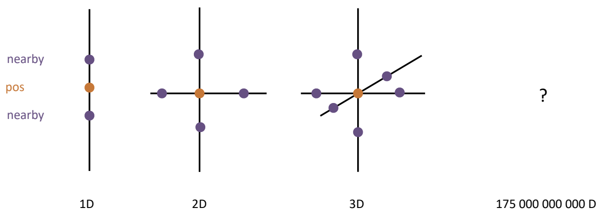
Gradient descent has implicit regularization: some parameters are preferred over others, a priori.

More on *explicit* regularization later

Initialization is of *crucial* importance.

More on this later

THE BLESSING OF HIGH DIMENSIONALITY



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.

LOTTERY TICKETS

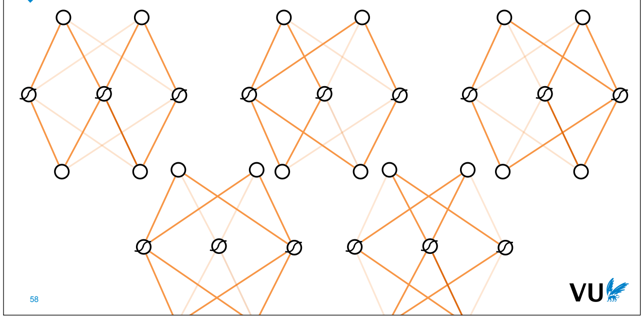
Traditional view:

- 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.

COMBINATORIAL EXPLOSION OF SUBNETWORKS.



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)

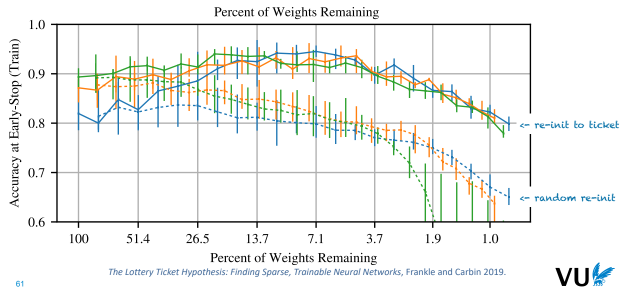
EXPERIMENT 1

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.

UNDER ITERATION

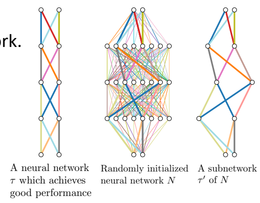


The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin 2019.

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 RL lecture)

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



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

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.

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 et al 2019.

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.

Lecture 4: Tools of the trade

Peter Bloem
Deep Learning

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

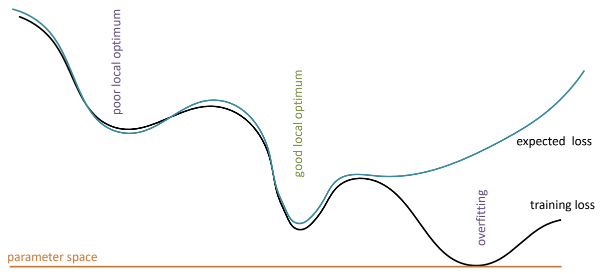
PART THREE: UNDERSTANDING OPTIMIZERS

JUSTIFYING *STOCHASTIC* GRADIENT DESCENT

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

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

NO MORE OVERFITTING



JUSTIFYING *STOCHASTIC* GRADIENT DESCENT: ROBBINS-MONRO (1951)

$$\nabla \mathbb{E}_{D \sim p} \text{loss}_D(\theta) \approx \nabla \text{loss}_d(\theta) \quad \text{with } d \sim p$$

Under certain conditions, GD with an *estimate* of the gradient converges the optimum (almost certainly).

Broadly:

- convex loss surface.
- asymptotically unbiased estimator.
- decaying learning rate α.

$\alpha_i \to 0$
$\sum \alpha_i = \infty$
$\sum \alpha_i^2 < \infty$

THE LIMITS OF SGD

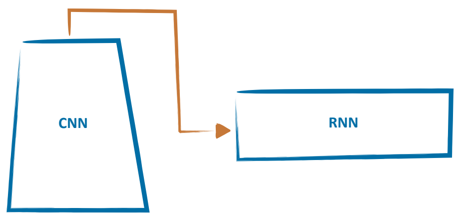


image model, `lr = 0.03`

language model, `lr = 0.00001`

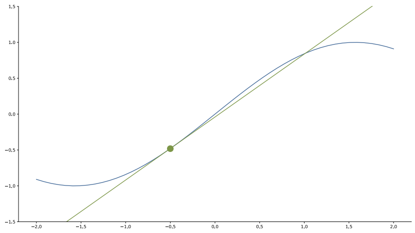
Second-order optimization, conditioning
aka Newton's method

Momentum

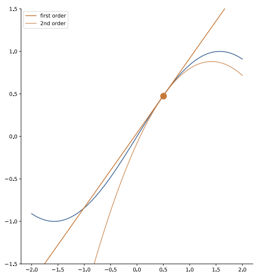
Adam

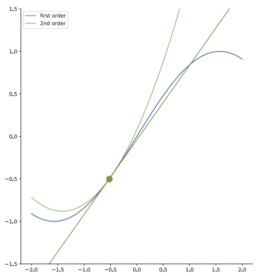
RAdam, LookAhead, LAMB

GRADIENT: A LINEAR APPROXIMATION



NON-LINEAR APPROXIMATION



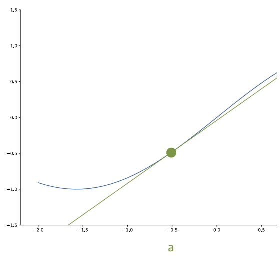


BEST LINEAR APPROXIMATION

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

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

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



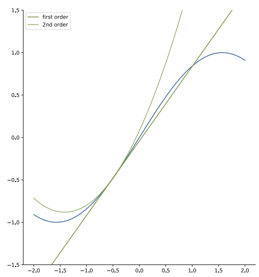
BEST SECOND ORDER APPROXIMATION

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

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

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

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



NEWTON'S METHOD (1D)

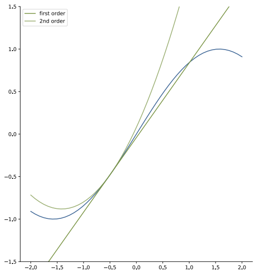
$$f_a(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2$$

$$f_a'(a) = f'(a) + f''(a)(x-a) = 0$$

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

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

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



NEWTON'S METHOD (ND)

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

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

$$f(\mathbf{x}) \approx \underbrace{f(\mathbf{a})}_{\text{scalar}} + \underbrace{\nabla f(\mathbf{a})}_{\substack{\text{vector}\\\text{(gradient)}}}(\mathbf{x}-\mathbf{a}) + \frac{1}{2}(\mathbf{x}-\mathbf{a})^T \underbrace{\nabla^2 f(\mathbf{a})}_{\substack{\text{matrix}\\\text{(Hessian)}}}(\mathbf{x}-\mathbf{a})$$

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

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.

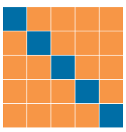
WHAT DOES NEWTON'S METHOD SOLVE?

Parameter interactions: partial derivatives assume independent updates

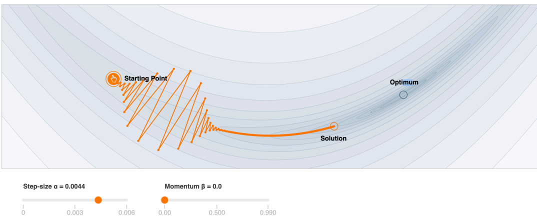
provided by the off-diagonal elements of the Hessian.

Curvature information: are we nearing a local optimum?

provided by the diagonal elements of the Hessian.



PATHOLOGICAL CURVATURE

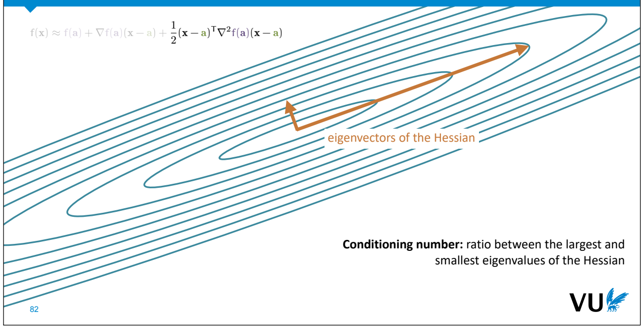


source: https://distill.pub/2017/momentum/

VU

CONDITIONING

$f(\mathbf{x}) \approx f(\mathbf{a}) + \nabla f(\mathbf{a})(\mathbf{x}-\mathbf{a}) + \frac{1}{2}(\mathbf{x}-\mathbf{a})^T \nabla^2 f(\mathbf{a})(\mathbf{x}-\mathbf{a})$



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

VU

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.

VU

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

HEAVY BALL MOMENTUM

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

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

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

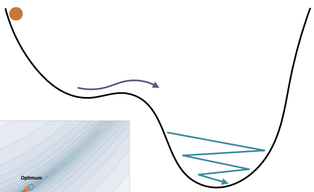
$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\mathbf{m}$$

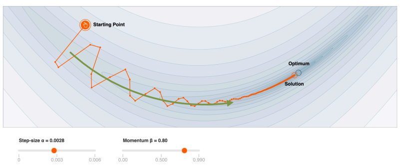
HEAVY BALL MOMENTUM

rolls out of local minima

dampens oscillations

accelerates repeating directions





GRADIENT ACCELERATION

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

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

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

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

...

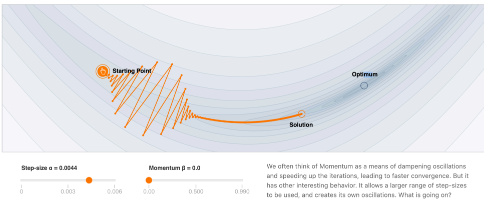
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\mathbf{d}\sum_{n=0}^{\infty}\gamma^n$$

$$= \mathbf{w} + \alpha\frac{1}{1-\gamma}\mathbf{d}$$

$$\gamma = 0.99 \mapsto 100 \times \text{ acceleration}$$

EXPONENTIAL MOVING AVERAGE

Averaging gradients helps to stabilize.



MOMENTUM AS A WEIGHTED SUM

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{g}_1$$
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\gamma \mathbf{g}_1 + \mathbf{g}_2)$$
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\gamma^2 \mathbf{g}_1 + \gamma \mathbf{g}_2 + \mathbf{g}_3)$$
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\gamma^3 \mathbf{g}_1 + \gamma^2 \mathbf{g}_2 + \gamma \mathbf{g}_3 + \mathbf{g}_4)$$
$$\ldots$$
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\gamma^n \mathbf{g}_1 + \ldots + \gamma \mathbf{g}_{n-1} + \mathbf{g}_n)$$

MOMENTUM VS. EXPONENTIAL MOVING AVERAGE

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

$$\gamma = 1 - \kappa$$

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

MINIBATCHING IS ALSO AVERAGING

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

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

MOMENTUM

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

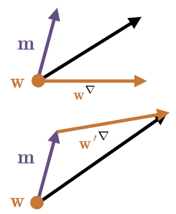
N: number of weights

- 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/**

NESTEROV MOMENTUM

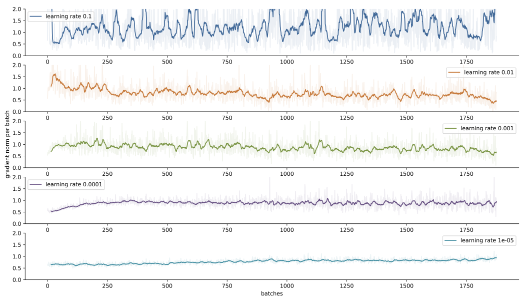
Compute gradient where you *will be*, not where you are.

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

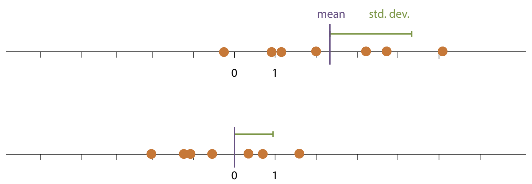


see also: https://cs231n.github.io/neural-networks-3/#sgd

REMEMBER THE *VARIANCE*



NORMALIZATION



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

ADAM: EXPONENTIAL MOVING *NORMALIZATION*

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

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

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

BIAS CORRECTION

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

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

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

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

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

ADAM

- 2N extra memory
- 2N extra operations
- Two extra hyperparameters 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
- Currently the default optimizer for most DL settings

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 trying plain SGD with (Nesterov) momentum.

Warning: Adam converges slowly for simple problems
SGD is much faster for linear problems.

NEW KIDS ON THE BLOCK: RECTIFIED ADAM

Learning rate warmup is often an important trick.

Adam must underestimate the early-training variance.

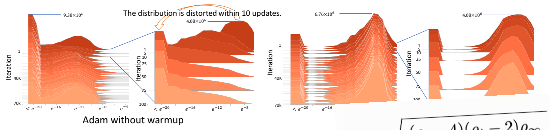


Figure 2: The absolute gradient histogram of the Transform... during the training (stacked along the y-axis). X-axis is ab... height is the frequency. Without warmup, the gradient distribu... steps.

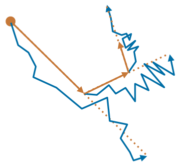
$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}.$$

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

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.

Lecture 4: Tools of the trade

Peter Bloem
Deep Learning

dlvu.github.io

VRIJE UNIVERSITEIT AMSTERDAM

PART FOUR: THE BAG OF TRICKS

initialization, normalization

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

regularization

- L1, L2, weight decay
- Dropout, priors

other tricks

- data augmentation, transfer learning

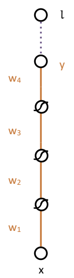
INITIALIZATION

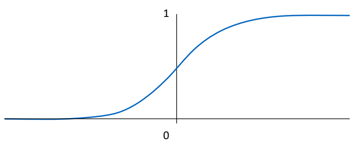
If the gradients are zero at the first batch, training never starts

If they're near zero, training starts very slowly

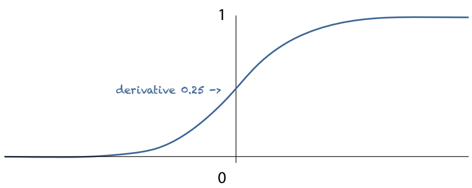
If the gradients blow up, we get NaN

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

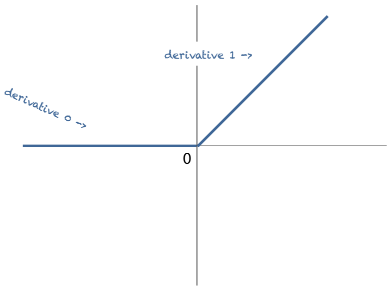




SIGMOID



RELU



GOOD INITIALIZATION

Make sure your input data is normalized: **0** mean, covariance **I**

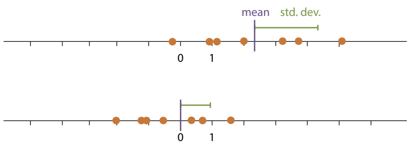
uniform over [0, 1] is usually fine too

Initialize your layer weights so that if the input has mean **0**, covariance **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



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

--P

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

$$\mathbf{x}^{\nabla} = \mathbf{W}^{T}\mathbf{y}^{\nabla}$$

assume $\text{Var}(x_i) = 1$

choose $\text{Var}(W_{ij}) = c, \text{Exp}(W_{ij}) = 0$

require $\text{Var}(y_i) = 1$

$$\text{Var}(y_i) = \text{Var} \sum_k W_{ik}x_k = \sum_k \text{Var}(W_{ik}x_k)$$

$$= \sum_k (\text{Var } W_{ik} \cdot \text{Var } x_k + (\text{Exp } W_{ik})^2 \text{Var } x_k + (\text{Exp } x_k)^2 \text{Var } W_{ik})$$

$$= m \cdot c$$

$$\text{Var}(x_i^{\nabla}) = n \cdot c$$

$$\text{Var } W_{ij} \approx \frac{1}{n} \approx \frac{1}{m} \mapsto \frac{1}{\frac{1}{2}(n+m)}$$

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

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

Understanding the difficulty of training deep feedforward neural networks, Glorot and Bengio PLMR 2010

HE INITIALIZATION (GAIN FACTOR)

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

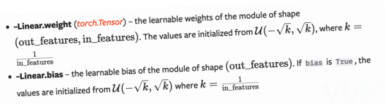
$$W_{ij} \sim N\left(0, \frac{2}{\sqrt{n+m}}\right)$$

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

NB: Glorot averages n and m by default, He takes n by default.

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He et al ICCV 2015

```
from torch import nn

model = nn.Sequential
    nn.Linear(784, 1024),
    nn.ReLU(),
    nn.Linear(1024, 10),
    nn.Softmax(dim=1)
)
```



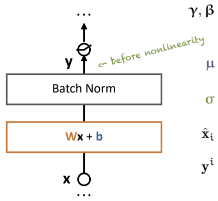
- **~Linear.weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

BATCH NORMALISATION

$x_1, \dots, x_m$: output *batch* of previous layer

$y_1, \dots, y_m$: *batch* result

γ, β : learnable parameter vectors



$$\mu = \frac{1}{m} \sum x_i \quad \text{mean over batch}$$

$$\sigma = \frac{1}{m} \sum (x_i - \mu)^2 \quad \text{variance over batch}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma + \epsilon}} \quad \text{standardize}$$

$$y^i = \gamma^T \hat{x}_i + \beta \quad \text{rescale}$$

BATCH NORM: 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.
- Compute using EMA

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

LAYER, INSTANCE, GROUP NORMALIZATION

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

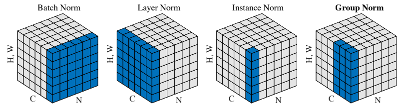


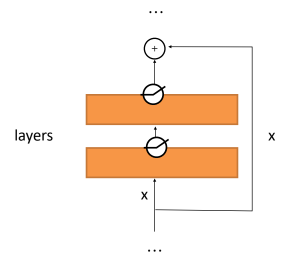
Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

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

RESIDUAL CONNECTIONS



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).

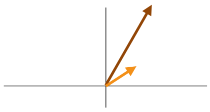
Implicit regularization: initialization, choice of optimizer, etc.

Explicit regularization:

- penalty terms
- priors
- dropout

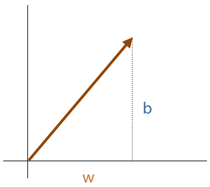
PENALTY TERM: LP REGULARIZER

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



parameter space

VECTOR NORM



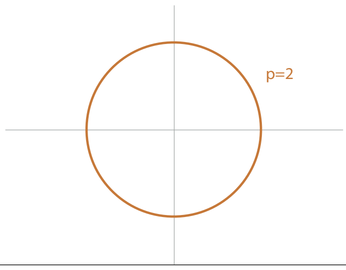
$$\theta = \begin{pmatrix} w \\ b \end{pmatrix}$$

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

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

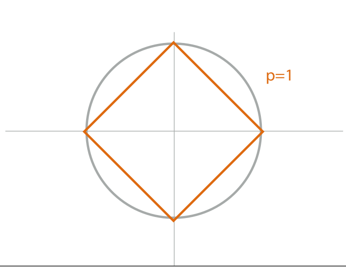
L2 NORM

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

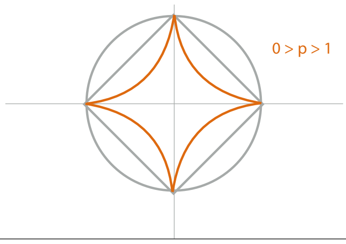


L1 NORM

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

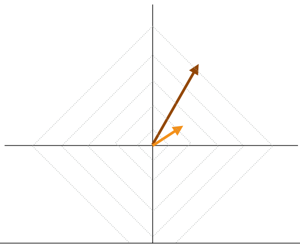


LP NORM

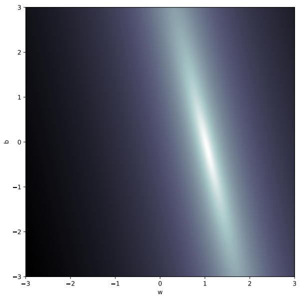


L1 REGULARIZER

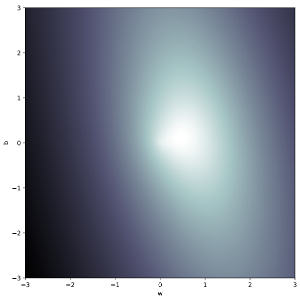
$$\text{loss} \leftarrow \text{loss} + \lambda||\theta||^1$$



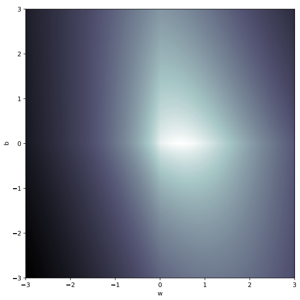
UNREGULARIZED



L2



L1



L2 regularization: often uses squared norm $w^T w$ as penalty term

For computational simplicity, and ease of analysis.

L1 regularization: promotes sparsity

WEIGHT DECAY

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

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

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

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

$$\mathbf{w} \leftarrow \gamma\mathbf{w}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla\text{loss}(\mathbf{w})$$

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

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

PRIORS AND REGULARIZERS

$$\arg\max_{\mathbf{w}} \ p_{\mathbf{w}}(\mathbf{x})p(\mathbf{w})$$

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

$$= \arg\min_{\mathbf{w}} \underbrace{-\log \ p_{\mathbf{w}}(\mathbf{x})}_{\text{base loss}} \underbrace{-\log p(\mathbf{w})}_{\text{penalty}}$$

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

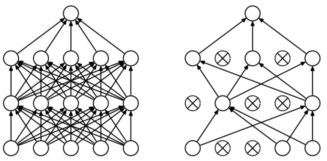
PENALTY WEIGHT

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

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

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

DROPOUT

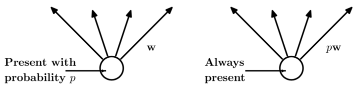


(a) Standard Neural Net (b) After applying dropout.

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

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.

Smerity
@Smerity
Following

If you ever need a definition of dropout that is both concise and accurate:

Dropout (Srivastava et al., 2014) may be the first instance of a human curated artisanal regularization technique that entered wide scale use in machine learning. Dropout, simply described, is the concept that if you can learn how to do a task repeatedly whilst drunk, you should be able to do the task even better when sober. This insight has resulted in numerous state of the art results and a nascent field dedicated to preventing dropout from being used on neural networks.

11:11 PM - 31 Mar 2018

215 Retweets 653 Likes

10 215 653

initialization, normalization

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

regularization

- L1, L2, weight decay
- Dropout, priors

other tricks

- data augmentation, transfer learning

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 possess naturally.
- Reduces overfitting: never the same input twice.

But: *some* invariances can harm your performance.

label: 9 label: 9

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

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.

THANK YOU FOR YOUR ATTENTION

dlvu@peterbloem.nl