Advanced Topic A13: *Gradient Pathologies in Deep Networks*

Data Science with Python – BSc Advanced Lectures

Joerg Osterrieder

10 Minutes

Motivation: Why Deep Networks Are Hard to Train

Depth creates representational power but destabilises learning

- A 10-layer network multiplies gradients across 10 Jacobians during backpropagation
- If each Jacobian has spectral norm < 1: gradients shrink exponentially (vanishing)
- If each Jacobian has spectral norm > 1: gradients grow exponentially (exploding)
- Vanishing gradients: early layers receive no useful signal; training stalls completely
- Exploding gradients: parameter updates become numerically unstable; loss diverges
- Both pathologies plagued early deep networks and RNNs on long sequences

Gradient pathologies were the key obstacle to training deep networks before 2015

Backpropagation and the Chain Rule

The gradient is a product of per-layer Jacobians

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial h_L} \cdot \frac{\partial h_L}{\partial h_{L-1}} \cdot \ldots \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial W_1}$$

- Each factor $\partial h_{l+1}/\partial h_l = \text{diag}(\sigma'(z_l)) \cdot W_l^T$ combines the activation derivative and the weight matrix
- **Sigmoid derivative**: $\sigma'(z) = \sigma(z)(1-\sigma(z)) \leq 0.25$ – guaranteed attenuation at every layer
- Deep sigmoid networks: gradient shrinks by a factor of at most 0.25 per layer; 10 layers gives $0.25^{10} \approx 10^{-6}$
- Weight matrices with large singular values amplify instead of attenuating: the product explodes

The chain rule product is the fundamental cause of both gradient pathologies in deep networks

Vanishing Gradients: The Saturation Problem

Saturating activations kill the gradient signal

- Sigmoid and tanh saturate: for large $|z|$, the derivative is near zero regardless of the weight
- A neuron that saturates has a gradient of approximately zero; it "dies" to backpropagation
- In an L-layer sigmoid network: the gradient at layer 1 has magnitude $\sim (0.25)^L$; for $L = 20$ this is $\sim 10^{-12}$
- Effect: early layers learn orders of magnitude more slowly than later layers
- Originally discovered by Hochreiter (1991) as the long-term dependency problem in RNNs

Vanishing gradients: the gradient at layer 1 decays as 0.25^L for sigmoid networks

Exploding Gradients: Unbounded Growth

Large weights cause gradient norms to diverge

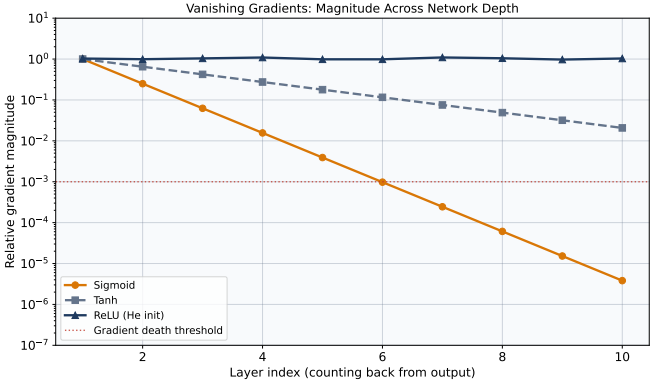
- If the weight matrices have singular values > 1: the Jacobian product grows exponentially with depth
- Exploding gradients produce NaN losses after a single bad update step
- More common in RNNs: the same weight matrix is multiplied T times (once per timestep)
- A weight matrix W with spectral norm $\rho > 1$ leads to gradient norm $\sim \rho^T$ after T steps
- Financial return series span hundreds of timesteps: an unstabilised RNN is almost certain to explode

Exploding gradients: gradient norm grows as ρ^T for an RNN with spectral radius $\rho > 1$

Gradient Magnitudes Across Layers

Empirical gradient norms shrink to zero with depth for sigmoid networks

- ReLU derivative is 1 (active) or 0 (dead): no saturation in the active regime
- ReLU networks with He initialisation maintain gradients much better than sigmoid/tanh
- Even ReLU can vanish for deep networks if many units are dead (negative pre-activations)

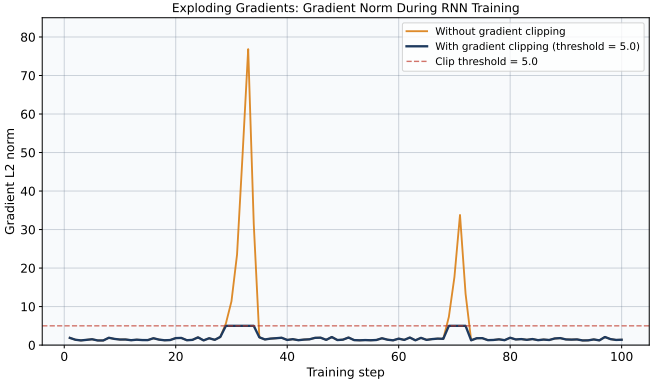


Sigmoid vanishes 10x faster per layer than tanh; ReLU is near-constant with proper initialisation

Exploding Gradient Norms During Training

Unclipped gradient norms spike orders of magnitude during training

- Gradient spikes cause parameter updates that undo many steps of learning
- With clipping: the norm is bounded, learning is stable
- Clipping does not prevent information from flowing; it only bounds the update magnitude



Gradient clipping to norm 1.0 is the standard fix for exploding gradients in RNNs and transformers

Solution: Weight Initialisation

Start with weights that keep gradient norms near 1

- **Xavier/Glorot** (2010): $W \sim \mathcal{U}(-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})})$; designed for sigmoid/tanh; keeps variance constant across layers
- **He/Kaiming** (2015): $W \sim \mathcal{N}(0, 2/n_{\text{in}})$; designed for ReLU; accounts for the factor-of-2 from dead units
- Bias terms initialised to zero; output layer uses Xavier regardless of activation
- Bad initialisation can cause vanishing/exploding even in a well-designed architecture
- PyTorch default: He uniform for convolutional layers; Kaiming uniform for linear layers

He init + ReLU is the canonical starting point for fully connected and convolutional networks

Solution: Batch Normalisation

Re-centre and re-scale pre-activations at every layer

$$\hat{z}^{(l)} = \frac{z^{(l)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}, \quad y^{(l)} = \gamma \hat{z}^{(l)} + \beta$$

- μ_B, σ_B^2: batch mean and variance; γ, β: learnable scale and shift
- Keeps pre-activations in the non-saturating regime: reduces vanishing gradients
- Acts as a regulariser (adds noise via batch statistics); often reduces need for dropout
- Limitation: behaviour differs between training and inference; problematic for small batches or RNNs
- Finance: batch norm complicates time-series models (temporal leakage via batch statistics)

Batch norm (Ioffe & Szegedy 2015) enabled training networks with 100+ layers

Solution: Residual Connections

Identity shortcuts bypass the gradient bottleneck

$$h_{l+1} = \mathcal{F}(h_l, W_l) + h_l$$

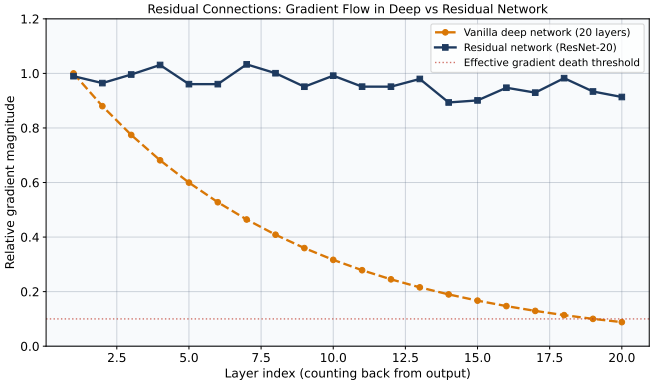
- The identity term h_l provides an additive gradient path: $\partial h_{l+1}/\partial h_l = \partial \mathcal{F}/\partial h_l + I$
- Even if $\partial \mathcal{F}/\partial h_l \approx 0$ (vanishing), the identity term keeps the total gradient at 1
- He et al. (2016) ResNet: trained a 152-layer network (previously impossible); won ImageNet by a large margin
- The residual block encourages the network to learn corrections to the identity rather than full transformations
- Dense connections (DenseNet): every layer connects to all subsequent layers; even stronger gradient flow

Residual connections (He et al. 2016): the single most impactful fix for deep network training

ResNets: Gradient Highways Across Depth

Identity shortcuts maintain gradient magnitude throughout the network

- Vanilla 20-layer networks: gradient at layer 1 is ∼1000x smaller than at layer 20
- ResNet-20: gradient stays near 1.0 throughout, enabling useful updates at every layer



Residual connections: gradient magnitude stays near 1 regardless of network depth

Solution: Gradient Clipping

Bound the gradient norm before applying the update

$$g \leftarrow g \cdot \min\left(1,\ \frac{\tau}{\|g\|_2}\right)$$

- If the gradient norm $\|g\|_2 > \tau$: rescale g to have norm exactly τ
- The gradient direction is preserved; only the step size is bounded
- Standard in RNN and transformer training: $\tau = 1.0$ in the original transformer paper
- Does not fix the root cause (weight initialisation or architecture); it is a training-time stabiliser
- PyTorch: `torch.nn.utils.clip_grad_norm_(parameters, max_norm=1.0)`

Gradient clipping: 3 lines of code that prevent explosive divergence in any recurrent model

Solution: Layer Normalisation and RMSNorm

Normalise across the feature dimension (not the batch)

- **Layer norm** (Ba et al. 2016): normalise over the feature dimension for each sample independently; batch size of 1 is valid; no temporal leakage; compatible with RNNs and transformers
- **RMSNorm** (Zhang & Sennrich 2019): $\hat{x}_i = x_i / \text{RMS}(x)$; removes the mean-centring step; computationally cheaper; used in LLaMA and most modern LLMs
- Pre-norm (normalise before attention/MLP) vs post-norm (after): pre-norm is more stable for deep transformers
- Combines with residual connections: the pre-norm residual transformer trains reliably at 100+ layers

Layer norm + residual: the standard recipe for training deep transformers (Vaswani et al. 2017 and beyond)

LSTM: Gating as a Learned Solution

The cell state provides an additive gradient path through time

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

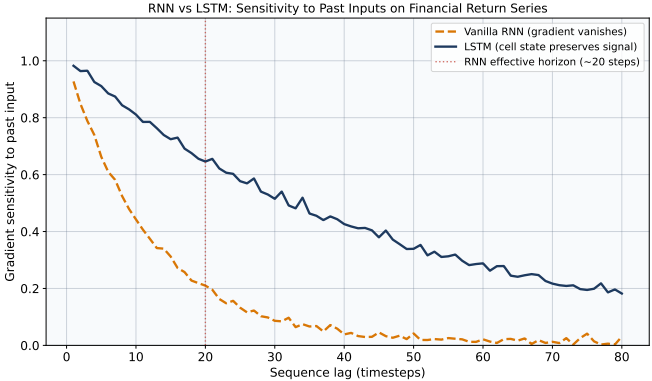
- Vanilla RNN: hidden state multiplied by W_h at every step; spectral radius determines stability
- LSTM (Hochreiter & Schmidhuber 1997): the cell state c_t accumulates via addition (not multiplication)
- Forget gate $f_t \in [0, 1]$: learned per-timestep decay rate; gradient flows through c_t without additional vanishing
- The gradient of the loss w.r.t. c_{t-1} is simply f_t: no repeated matrix multiplications
- GRU (Cho et al. 2014): simplified gating with fewer parameters; competitive with LSTM on most tasks

LSTM cell state = residual connection through time; gating controls what to remember and forget

Finance Application: RNNs on Return Series

Long-range dependencies in financial time series

- Volatility clustering: today's volatility predicts tomorrow's; dependencies span weeks to months
- Vanilla RNN: gradient vanishes across 20+ timesteps; cannot capture monthly seasonality
- LSTM: cell state preserves information across 100+ timesteps; viable for weekly return models



LSTM outperforms vanilla RNN on return prediction tasks with long memory (volatility, momentum)

Transformers: Attention Bypasses the Bottleneck

Direct connections between all positions eliminate sequential gradient chains

- RNN: gradient between positions i and j traverses $|i - j|$ matrix multiplications
- **Self-attention**: every position attends directly to every other; gradient path length is $O(1)$
- No recurrence: the entire sequence is processed in parallel; gradients are not multiplied across time
- Transformer pathologies: attention collapse (all attention on one token), rank collapse in deep transformers
- Solutions: pre-norm architecture, relative positional encodings, QK normalisation (recent LLMs)

Attention replaces the RNN sequence bottleneck with direct O(1) gradient paths between positions

Code Pointer

Diagnosing and fixing gradient pathologies in PyTorch

- **Monitor gradient norms:** `for p in model.parameters(): print(p.grad.norm())`
- **He initialisation:** `nn.init.kaiming_uniform_(layer.weight, nonlinearity='relu')`
- **Gradient clipping:** `torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)`
- **Residual block:** `out = F.relu(self.bn(self.conv(x))) + x`
- **Detect vanishing:** plot gradient norms per layer using `wandb` or `tensorboard`; any layer with norm $< 10^{-5}$ is effectively dead

Monitor gradient norms per layer during training: the single most informative diagnostic tool

Limitations and Trade-offs

Each fix introduces its own constraints

- Batch norm: invalid for batch size 1, time-series (temporal leakage), variable-length inputs; replaced by layer norm in NLP
- Residual connections: require matching dimensions or projection layers; add $\sim$10% parameter overhead
- Gradient clipping: hides the root cause of instability; a network that always clips is misconfigured
- LSTM: much slower than self-attention for long sequences; largely replaced by transformers for text
- Depth still matters: very deep networks (1000+ layers) can suffer from rank collapse even with residual connections and normalisation

No single fix is universal: match the solution to the architecture and the data modality

Summary and Open Questions

Three things to remember

- Vanishing gradients: sigmoid saturation multiplies the gradient by ≤ 0.25 per layer; fix with ReLU, He init, batch/layer norm, and residual connections
- Exploding gradients: spectral radius > 1 in RNN weights; fix with gradient clipping, careful initialisation, and spectral normalisation
- Residual connections (ResNets) and LSTM gating are the two most impactful architectural solutions; transformers solve the problem structurally via direct attention paths

Open questions:

1. Can we train very deep networks (1000+ layers) without rank collapse even with residual connections?
2. How do we detect and fix gradient pathologies in mixture-of-experts models with sparse routing?
3. Are there gradient-free training methods (forward-forward, perturbation-based) that bypass backpropagation entirely?

Further reading: He et al. (2016) ResNet; Hochreiter & Schmidhuber (1997) LSTM; Vaswani et al. (2017) Attention