02 Optimisation

Optimisers

modern optimisers

from SGD to Adam

SGD (1951)

$$\theta = \theta - \alpha \cdot \nabla L$$

Simple but slow, sensitive to learning rate

Momentum (1964)

$$v = \beta \cdot v + \nabla L$$
$$\theta = \theta - \eta \cdot v$$

- computes gradients at current point
- momentum is a coarse prediction of where SGD will move

Nesterov accelerated momentum

$$m_{t+1} = \beta \cdot m_t + (1-\beta) \sum_i \in \mathcal{B}_t \frac{\partial \ell_i \left[w_t - \eta \cdot m_t \right]}{\partial w}$$
$$w_{t+1} = w_t - \eta \cdot m_{t+1}$$

- Nesterov accelerated momentum computes the gradient at a predicted point of where the gradient **will** move after the modification
- frequently, though not always, better than standard momentum method

RMSprop (2012)

$$s = \beta \cdot s + (1-\beta) \cdot \nabla L^2$$
$$\theta = \theta - \alpha \cdot \nabla L / \sqrt{s}$$

- Adapts learning rate per parameter

Adam (2014)

$$
\begin{aligned}
m &= \beta_1 \cdot m + (1-\beta_1) \cdot \nabla L && \text{Momentum} \\
v &= \beta_2 \cdot v + (1-\beta_2) \cdot \nabla L^2 && \text{RMSprop} \\
\theta &= \theta - \eta \cdot \hat{m}/\sqrt{\hat{v}} && \text{Bias-corrected}
\end{aligned}
$$

- Best of both worlds!
- takes into account the variance

Why Adam Works

Per-Parameter Learning Rates

Problem: Different parameters need different learning rates

- Embeddings: Large gradients, need small LR
- Output layer: Small gradients, need large LR
- vanishing gradient problem too

Adam's Solution:

```
Effective LR = α / √(variance + ε)
```

- High variance → Lower LR (stable)
- Low variance → Higher LR (faster)

AdamW - the important fix

problem with Adam + L2

What we want:

$$\theta = \theta - \alpha(\nabla L + \lambda\theta) \qquad \text{Decoupled weight decay}$$

What Adam + L2 does:

$$m = \beta_1 m + (1-\beta_1)(\nabla L + \lambda\theta)$$

wrong couples decay with adaptive LR

AdamW fix

Apply weight decay AFTER Adam update

$$\theta = \theta - \alpha \cdot \hat{m}/\sqrt{\hat{v}} - \lambda\theta$$

- Transformers always use AdamW
- Better generalisation
- Proper regularisation

Learning Rate Schedules

warmup (Critical for Transformers)

```
if step < warmup_steps:
    lr = base_lr * step / warmup_steps
```

Why: Large initial gradients can destabilise

Cosine Annealing

```
lr = min_lr + (max_lr - min_lr) * 0.5 * (1 + cos(π * step / total_steps))
```

Why: Smooth decay, allows "restarts"

Inverse Square Root (GPT-style)

```
lr = base_lr * min(step^(-0.5), step * warmup_steps^(-1.5))
```

Why: Fast initial progress, slow refinement

---

generalisation

the generalisation puzzle

the mystery

What we observe:

- Models with 100B+ parameters training on "only" millions of examples
- Perfect memorisation of training data
- Yet they generalise!

Classical theory says: This shouldn't work!

$$\epsilon^2 < \frac{\log |H_\epsilon| + \log 1/\delta}{2m}$$

- where
  - ϵ is the difference between training and generalisation error
  - $|H_\epsilon|$ is the ϵ cover of the hypothesis class, usually $|H_\epsilon| \sim (1/\epsilon)^d$
  - d is the VC dimension
  - δ is confidence
  - m is the number of samples
- VC dimension → ∞ for deep networks
- Rademacher complexity → ∞
- PAC bounds → vacuous

the resolution:

- SGD has implicit bias
- Networks prefer simple functions
- Real data has structure

---

implicit regularisation

SGD finds special solutions

All these solutions have zero training loss:

```
θ₁ = [1000, -999, 0.1, ...]  # Complex
θ₂ = [0.1, 0.1, 0.1, ...]   # Simple
```

SGD prefers θ_2!

mechanisms:

1. **Noise**: SGD noise prevents overfitting
2. **Early stopping**: Implicit regularization
3. **Architecture**: CNNs bias toward local features
4. **Initialization**: Start near simple functions

Net effect: `Loss = Training_Loss + Implicit_Complexity_Penalty`

---

grokking - delayed generalization

the phenomenon

```
Epoch 1-1000:    Train = 100%, Test = 50% (memorization)
Epoch 1001-5000: Train = 100%, Test = 50% (still memorizing)
Epoch 5001:      Train = 100%, Test = 99% (sudden generalization!)
```

why grokking happens:

1. **Phase 1**: Memorize with complex pattern
2. **Phase 2**: Simplify internal representation
3. **Phase 3**: Simple pattern generalizes

Example: Modular arithmetic

- Memorize: Store all input-output pairs
- Generalize: Learn the algorithm

Lesson: Generalization can emerge long after fitting!

---

what really drives generalization?

the modern view

Not (just) explicit regularization:

- L2, dropout help but aren't necessary
- Unregularized networks still generalize

But rather:

1. **Inductive bias** from architecture
2. **Implicit bias** from optimization
3. **Data structure** (manifold hypothesis)
4. **Overparameterization** (lottery tickets)

the simplicity bias

Networks prefer simple functions that:

- Vary smoothly
- Have low frequency
- Decompose hierarchically

- Match data symmetries
- networks are lazy and want to compute as little as possible

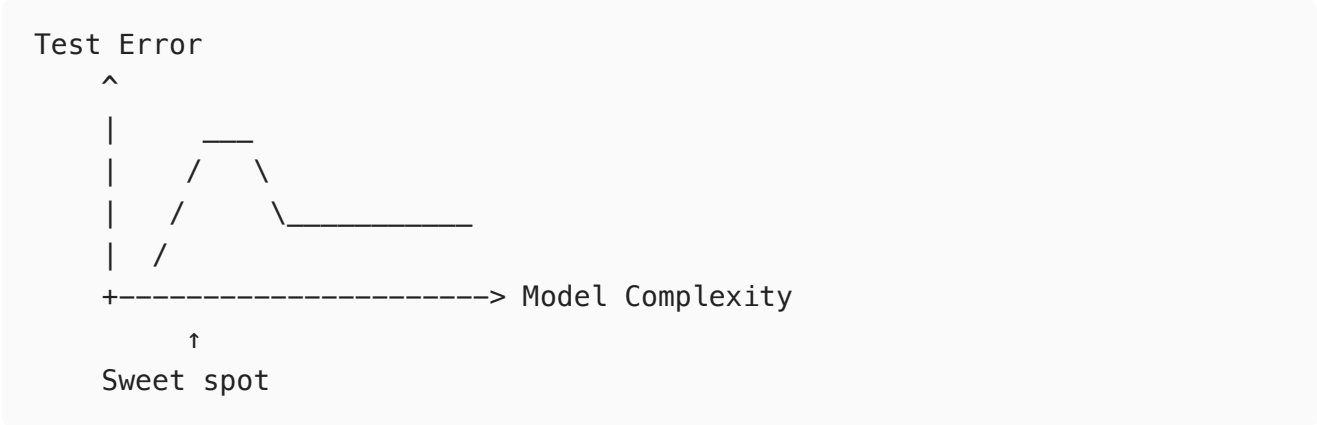
> 🎯 **Key**: Deep learning works because real-world data is simple in the right coordinate system!

---

double descent curve

classical vs modern wisdom

classical U-curve (a bias-variance tradeoff)



Belief: Optimal complexity balances bias and variance

the bias-variance tradeoff

$$
\begin{aligned}
Err(\hat{f}) &= E[(y - \hat{f})^2] \\
&= E[y^2] - 2E[y\hat{f}] + E[\hat{f}^2]
\end{aligned}
$$

where

$$
\begin{aligned}
E[y^2] &= E[(f + \epsilon)^2] = E[f^2] + 2E[f\epsilon] + \epsilon^2 \\
&= E[f^2] + 2E[f] \cdot 0 + \epsilon^2 \\
&= f^2 + \sigma^2
\end{aligned}
$$

where

$$
Var(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2
$$

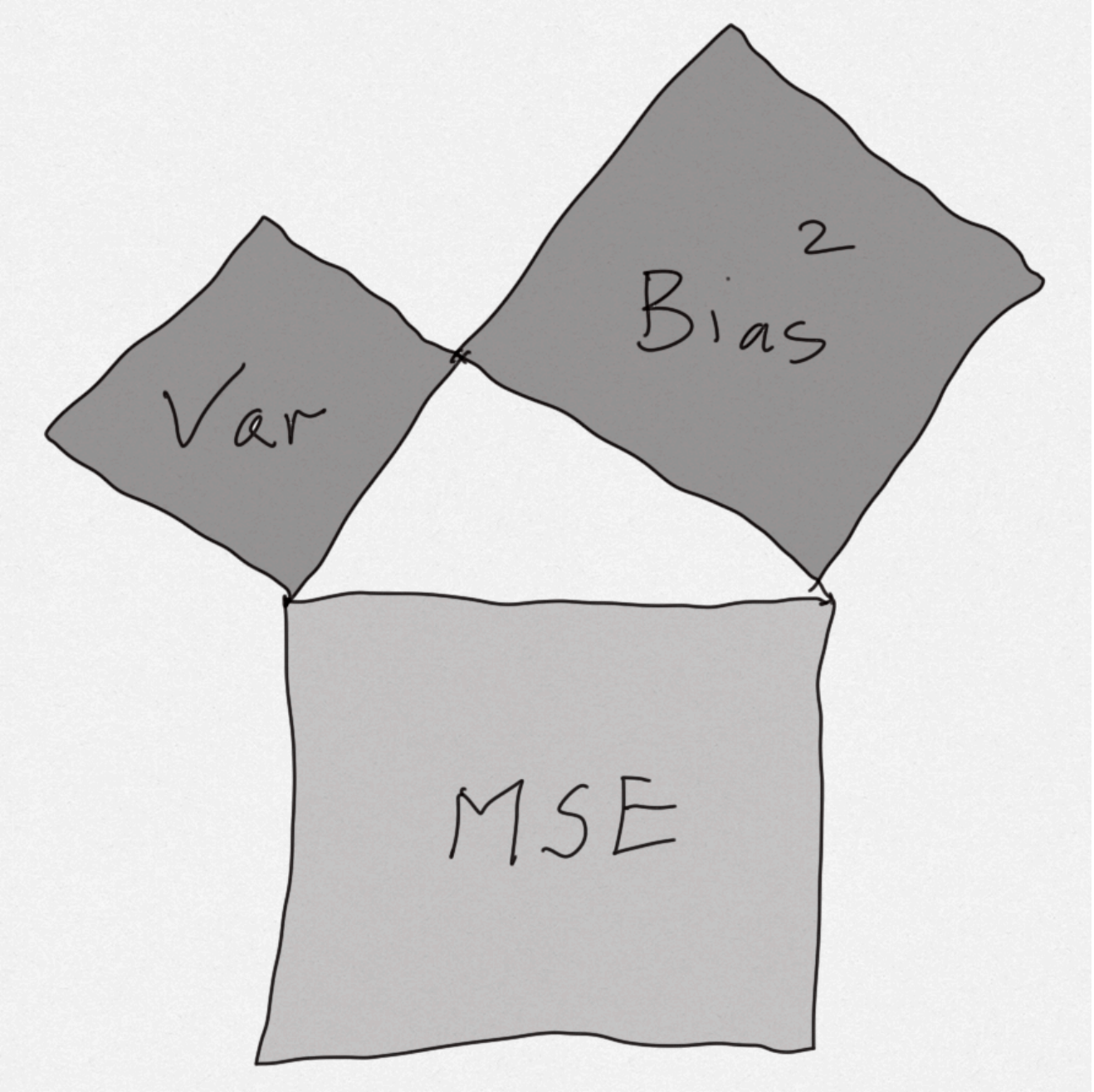
thus

$$
\begin{aligned}
E[\hat{f}^2] &= Var(\hat{f}) + E[\hat{f}]^2 \\
E[y\hat{f}] &= E[(f + \epsilon)\hat{f}] \\
&= E[f\hat{f}] + E[\epsilon\hat{f}] \\
&= fE[\hat{f}]
\end{aligned}
$$

thus

$$
\begin{aligned}
Err[\hat{f}] &= f^2 + \sigma^2 - 2fE[\hat{f}] + Var(\hat{f}) + E[\hat{f}]^2 \\
&= f^2 - 2fE[\hat{f}] + E[\hat{f}]^2 + \sigma^2 + Var(\hat{f}) \\
&= (f - E[\hat{f}])^2 + \sigma^2 + Var(\hat{f}) \\
&= Bias[\hat{f}]^2 + \sigma^2 + Var(\hat{f})
\end{aligned}
$$

- **bias** is the average error between average prediction and the true values
- **variance** (a system error) is the residual noisy error - the differences between average predictions
- minimising one may lead to the growth of the other (see drawing below)
  - multiple expert model may be a solution
- the system noise can be further broken down to stable pattern noise, occasion noise, and level noise



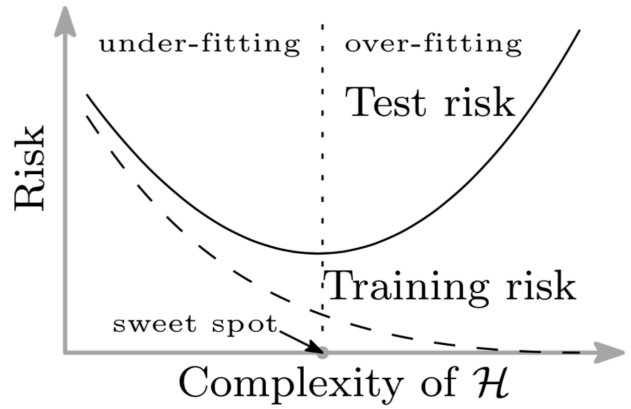
MLU-EXPLAIN: the bias-variance tradeoff

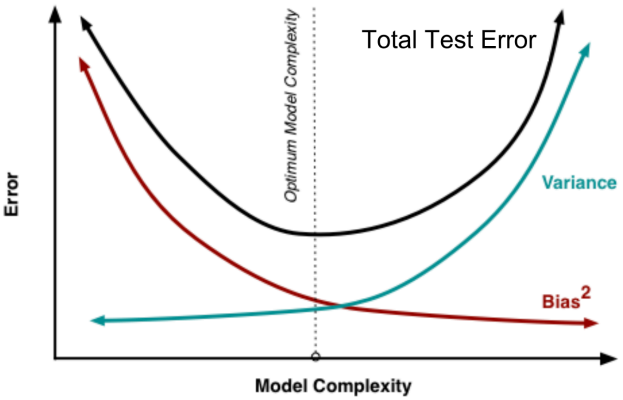
---

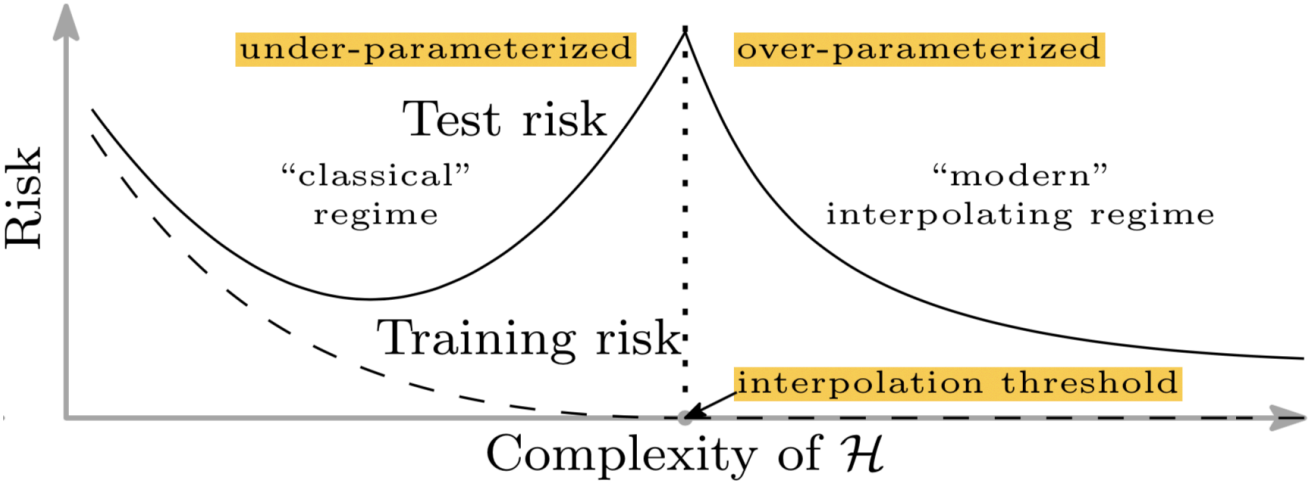
modern double descent

```
Test Error
    ^
    |     ___
    |    /   \
    |   /     \___    ___
    |  /          \  /   \___
    +-----------------------> Model Complexity
          ↑          ↑
    Classical  Interpolation
    optimum     threshold
```

in reality we can keep going past interpolation!







(after [Reconciling modern machine learning practice and bias-variance trade-off, Belkin et al. 2018]) Actually not very easy to copy in code

[MLU-EXPLAIN: double descent]
[MLU-EXPLAIN: double descent simple math]

---

the interpolation threshold

three regimes

1. under-parameterised (n_params < n_data)

- can't fit all training data
- classical bias-variance tradeoff
- test error follows U-curve

2. critical (n_params ≈ n_data)

- just enough to memorize
- worst generalisation!

- peak test error

3. over-parameterised (n_params >> n_data)

- many ways to fit data
- SGD chooses simple solution
- test error decreases again!

at interpolation threshold, only ONE solution fits → It's complex!

---

why double descent happens

mathematical intuition

underparameterized: Unique solution via least squares

$$\theta^* = (X^T X)^{(-1)} X^T y$$

critical regime: matrix barely invertible

$$\theta^* = (X^T X + \varepsilon\, I)^{(-1)} X^T y \qquad \# \ \varepsilon \to 0, ||\theta^*|| \to \infty$$

Overparameterized: Minimum norm solution

$$\theta^* = X^+ y = X^T (X X^T)^{(-1)} y \qquad \# \text{ Well-conditioned!}$$

key line: more parameters → more solutions → SGD picks simpler one

---

practical implications

do's:

- ✅ use large models when possible
- ✅ train past interpolation
- ✅ do not stop at 100% training accuracy
- ✅ use early stopping on validation loss, **not train**

don'ts:

- ❌ don't choose model size to match data size
- ❌ don't fear overparameterization
- ❌ don't trust classical statistical guidance

where double descent appears:

- Model size (width, depth)
- Training time (epoch-wise)
- Data size (sample-wise)
- Even label noise!

modern practice: go big and regularise implicitly via SGD

---

loss function visualisation

the geometry of optimisation

high-dimensional reality

the curse of visualisation:

- networks have millions of dimensions
- we can only see 2D/3D projections
- most intuition from low-dimensions is WRONG

what is different in high-dimensionality

- no local minima (mostly saddles)
- volume at surface (not center)
- all points equidistant
- connected solution manifolds

---

visualisation techniques

1. random projections

```
θ_proj = θ* + α·δ₁ + β·δ₂  # δᵢ ~ N(0,1)
```

Shows local geometry around minimum

2. PCA projections

```
θ_proj = θ* + α·PC₁ + β·PC₂
```

Shows dominant variation directions

3. linear interpolation

```
θ_interp = (1−α)·θ₁ + α·θ₂
```

Tests mode connectivity

4. filter normalisation (Li et al. 2018)

```
δ_normalized = δ * ||θ||/||δ||  # Scale-invariant
```

Handles different parameter scales

---

sharp vs flat minima

sharp minimum:

```
    \   /
     \ /
      V     ← Small perturbation → large loss increase
```

poor generalisation

flat minimum:

```
    \___/
          ← Large perturbation → small loss increase
```

good generalisation

width effect on landscape

- ***narrow network**: chaotic, isolated minima
- **wide network**: smooth, connected valleys

batch size effect on landscape

- small batch → noisy → finds flat minima
- large batch → smooth → finds sharp minima

---

mode connectivity

the discovery

- ***observation** different training runs find different minima
- **question** is it possible for to be connected?
  - **answer** YES! (perhaps with slight path elevation)

types of connectivity

linear mode connectivity

```
L(θ_interpolated) ≈ L(θ₁) ≈ L(θ₂)
```

nonlinear path

```
θ(t) = bezier_curve(θ₁, θ_curve, θ₂, t)
```

low loss path exists but curves

implication

The solution space is a connected manifold, not isolated points!

---

deep learning success story (dl-ss)

why deep learning works?

1. **Representation Learning**: learns what to look for
2. **over-parameterisation**: makes optimisation easier
3. **foundation models**: Amortises learning across tasks
4. **modern optimisers**: Adaptive, stable, fast
5. **implicit regularization**: SGD prefers simple solutions
6. **double descent**: More is actually better
7. **good geometry**: Loss landscapes are navigable

the surprising truth

> "We don't train networks in spite of having too many parameters, we train them BECAUSE we have too many parameters!"

Lottery Ticket Hypothesis

The Surprising Discovery

The Observation (Frankle & Carbin, 2019)

> "A randomly initialised, dense neural network contains a subnetwork that is initialised such that— when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations."

in simple terms

Your neural network is like a lottery with millions of tickets:

- **most tickets lose** (bad subnetworks)
- **a few tickets win** (good subnetworks)
- **training finds the winning tickets**
- **the winning ticket was there from the start!**

why this may matter?

- we can remove 90-95% of parameters
- but ONLY if we keep the right initialisation
- suggests structure emerges very early in training
    - structure, i.e. the winning subnetwork

---

the Lottery Ticket algorithm

step-by-step

```
def find_winning_ticket(network, data):
    # Step 1: Random Initialization
    θ₀ = random_initialize()

    # Step 2: Train to Completion
    θ_final = train(network, θ₀, data)

    # Step 3: Identify Important Weights (Pruning)
    # pruning can be e.g. magnitude
```

```
    mask = magnitude_prune(θ_final, sparsity=90%)

    # Step 4: Reset to Original Init with Mask
    θ_ticket = mask ⊙ θ₀  # Element-wise product

    return θ_ticket, mask
```

mathematical formulation:

Given network $f(x;\theta)$ with parameters $\theta \in \mathbb{R}^n$:

1. **initialize**: $\theta_0 \sim \mathcal{N}(0, \sigma^2)$
2. **train**: $\theta_T = \text{SGD}(f, \theta_0, \mathcal{D}, T)$
3. **prune**: $m_i = \mathbb{1}[|\theta_{T,i}| > \tau]$ where τ is the pruning threshold
4. **reset**: $\theta_{\text{ticket}} = m \odot \theta_0$

key finding

$$\mathcal{L}(f(x; m \odot \theta_0)) \approx \mathcal{L}(f(x; \theta_T)) \quad \checkmark \text{ Works if initialised to original values !}$$
$$\mathcal{L}(f(x; m \odot \theta_0')) \gg \mathcal{L}(f(x; \theta_T)) \quad \times \text{ Fails if reinitialised!}$$

---

why do lottery tickets exist?

the combinatorial argument

number of possible subnetworks

$$N_{\text{subnets}} = \binom{n}{k} \approx \frac{n^k}{k!}$$

where n = total parameters, k = parameters kept

probability at least one is good

$$P(\text{winning ticket exists}) = 1 - (1-p)^{N_{\text{subnets}}}$$

Concrete Example:

- Network with 1M parameters, keeping 10% (100K)
- Number of possible subnetworks: $\binom{10^6}{10^5} \approx 10^{250,000}$
- Even if $p = 10^{-100}$ per subnet, we still have winners!

The Initialization Lottery:

```
Dense Network (1M params)
    ↓
Contains ~10^250,000 subnetworks
    ↓
Most are bad, but some are golden!
    ↓
Training identifies which ones
```

---

pruning strategies

magnitude pruning (most common)

remove weights with smallest absolute values:

$$m_i = \begin{cases} 1 & \text{if } |\theta_i| \geq \tau_p \\ 0 & \text{if } |\theta_i| < \tau_p \end{cases}$$

where τ_p is the p-th percentile of $|\theta|$

structured vs unstructured

unstructured (Original LTH):

- prune individual weights
- maximum flexibility
- hard to accelerate

structured:

- prune entire neurons / channels / layers
- less flexible
- hardware-friendly

iterative magnitude pruning (IMP)

Better for high sparsity:

```
for round in range(num_rounds):
    train_model()
    prune_20_percent()  # Gradual
    reset_to_original_init()
```

---

when do tickets emerge?

early-bird tickets (You et al., 2020)

question: when, during training, does the winning structure emerge?

finding: very early, often within 10-20% of training!

the rewinding insight

instead of resetting to iteration 0, reset to iteration k:

$$\theta_{\text{ticket}} = m \odot \theta_k \quad \text{where } k \ll T$$

results:

- $k = 0$ (original init): Works up to ~90% sparsity
- $k = 0.1\,T$ (10% trained): Works up to ~98% sparsity
- $k = 0.5\,T$ (half trained): Almost any sparsity works

implication:

the "lottery ticket" isn't completely random—early training refines it!

---

lottery tickets at scale

BERT and Transformers

challenge: original LTH often fails for large models!

solution: Late rewinding

- Don't reset to iteration 0
- Reset to ~1000 steps in
- Now works for BERT, GPT-2

the Lottery Ticket Hypothesis 2.0

for large models:

1. **phase 1** (0-1000 steps): Find good basin
2. **phase 2** (1000+ steps): Find specific minimum
3. **pruning**: Identifies important structure
4. **rewinding**: Keep Phase 1, redo Phase 2

Practical Impact

- 90% sparse BERT matches dense performance
- 10x inference speedup possible
- transfer: Tickets found on one task work for others!

---

what Lottery Tickets tell us?

1. **initialisation contains solution**
   - networks don't learn from blank state
   - good subnetworks exist at initialisation
2. **parameterisation = more tickets**
   - larger networks → more subnetworks → higher chance of winners
   - may explains why bigger models train better
3. **structure emerges early**
   - important connections identified quickly
   - rest of training is refinement

Practical Applications:

✅ **network compression**: prune 90%+ parameters
✅ **faster training**: train only the winning ticket
✅ **architecture search**: tickets reveal important structures
✅ **transfer learning**: tickets generalise across tasks

Lottery Ticket Hypothesis

Neural Tangent learning - understanding deep learning through kernel methods

the core idea

the revolutionary insight (Jacot et al., 2018)

> "In the infinite width limit, neural network training dynamics become linear and are governed by a fixed kernel—the Neural Tangent Kernel"

what this means:

Finite Width Network (Normal):

- weights change during training

- features evolve and adapt
- non-linear dynamics
- hard to analyse

Infinite Width Network (NTK):

- features stay (approximately) fixed
- training = linear regression in feature space
- dynamics completely predictable
- becomes a kernel method!

the bridge:

Neural Networks ←-→ Kernel Methods

---

mathematical foundation

The Neural Tangent Kernel Definition

for neural network $f(x;\theta)$ with parameters θ:

$$\boxed{\Theta(x,x') = \left\langle \nabla_\theta f(x;\theta), \nabla_\theta f(x';\theta) \right\rangle = \sum_{i=1}^{p} \frac{\partial f(x)}{\partial \theta_i} \frac{\partial f(x')}{\partial \theta_i}}$$

intuition: how similarly does the network change at inputs x and x'?

training dynamics in function space

under gradient flow ($\dot{\theta} = -\nabla_\theta \mathcal{L}$):

$$\frac{df(x,t)}{dt} = -\sum_{i=1}^{n} (f(x_i,t) - y_i) \cdot \Theta(x, x_i)$$

in matrix form:

$$\boxed{\frac{d\vec{f}}{dt} = -\mathbf{\Theta} \cdot (\vec{f} - \vec{y})}$$

this is a LINEAR ODE! Solution:

$$f(x,t) = f(x,0) + \mathbf{\Theta}(x,X)^\top (\mathbf{I} - e^{-\mathbf{\Theta} t}) \mathbf{\Theta}^{-1} (\vec{y} - f(X,0))$$

---

the infinite width limit

what happens as width → ∞

consider a 2-layer network:

$$f(x) = \frac{1}{\sqrt{m}} \sum_{j=1}^{m} a_j \sigma(w_j^T x)$$

where m = width, a_j = output weights, w_j = hidden weights

key results:

1. **at initialization** (width → ∞):
   - $f(x) \to \mathcal{GP}(0, \Sigma)$ (Gaussian Process)
   - $\Theta(x, x') \to \Theta^*(x, x')$ (deterministic kernel)
2. **during training** (width → ∞):
   - $\Theta(x, x', t) \approx \Theta(x, x', 0)$ (kernel stays constant!)
   - features don't evolve
   - training = kernel regression

the width scaling:

$$\text{Feature learning} \sim \frac{1}{\sqrt{\text{width}}}$$

$$\text{Kernel stability} \sim 1 - \frac{1}{\sqrt{\text{width}}}$$

NTK for Different Architectures

Fully Connected Networks

$$\Theta_{\text{FC}}(x, x') = \sum_{l=0}^{L} \Sigma^{(l)}(x, x') \cdot \dot{\Sigma}^{(l)}(x, x')$$

where $\Sigma^{(l)}$ is the kernel at layer l

Convolutional Networks

$$\Theta_{\text{CNN}}(x, x') = \sum_{\text{patches}} \Theta_{\text{FC}}(\text{patch}(x), \text{patch}(x'))$$

CNNs = Locally connected NTK

Attention/Transformers

$$\Theta_{\text{Attn}}(x, x') = \sum_{i,j} \alpha_{ij}(x)\alpha_{ij}(x') \cdot \Theta_{\text{base}}(x_i, x'_j)$$

where α_{ij} are attention weights

Key Pattern:

Architecture → Kernel Structure → Inductive Bias

---

finite width effects

real networks are not infinite!

finite width (real world):

- NTK evolves during training
- features learn and adapt
- better generalization

the feature learning regime:

$$\Theta(t) = \Theta(0) + \underbrace{\Delta\Theta(t)}_{\text{Feature Learning}}$$

$$\Delta\Theta \propto \frac{1}{\sqrt{\text{width}}}$$

the transition

```
Narrow Networks          Wide Networks          Infinite Networks
(width < 100)            (width ~ 1000)         (width → ∞)
     ↓                        ↓                      ↓
Strong Feature           Moderate Feature        No Feature
Learning                 Learning                Learning
     ↓                        ↓                      ↓
Hard to Train            Sweet Spot              Kernel Regression
```

optimal width

not too narrow (hard optimization)
not too wide (poor feature learning)

---

why NTK matters

Theoretical Insights

✅ **explains trainability**: wide networks have better-conditioned NTK
✅ **predicts convergence**: can compute convergence rate from $\lambda_{\min}(\Theta)$
✅ **architecture design**: NTK reveals inductive bias
✅ **generalization bounds**: kernel theory applies

practical applications

1. **infinite-width networks as baselines**

```
# If infinite-width does well → architecture is good
# If finite-width much better → feature learning matters
```

2. **architecture search via NTK**
   - Compute NTK at initialization
   - Good condition number → trainable architecture
3. **understanding scale**
   - When is a network "wide enough"?
   - NTK condition number tells us

limitations

❌ real networks learn features (good thing!)
❌ NTK can't explain all phenomena (e.g., grokking)
❌ infinite width is impractical

NTK vs Lottery Tickets

two perspectives on the same phenomenon

Aspect	Lottery Ticket	Neural Tangent Kernel
Focus	Sparse structure	Training dynamics
Key insight	Good subnetworks exist	Wide → linear dynamics
Initialization	Contains solution	Determines kernel

Aspect	Lottery Ticket	Neural Tangent Kernel
Overparameterization	More tickets	Better conditioning
Feature learning	Implicit	Decreases with width
Practical use	Pruning, compression	Architecture design

the unified view

both explain why over-parameterisation helps:

Lottery Tickets: more parameters → more subnetworks → higher chance of winners

NTK: more parameters → smoother kernel → easier optimization

the paradox resolved

- **classical view**: parameterisation → Overfitting ❌
- **modern view**: parameterisation → Better optimisation → Better generalisation ✅

both LTH and NTK explain different aspects of why this works!

Lottery Ticket and NTK summary and key takeaways

Lottery Ticket Hypothesis

🎰 **core Message**: Your network already contains the solution at initialization

remember:

- magnitude pruning after training
- reset to original initialization
- 90%+ sparsity is achievable
- structure emerges early

practical Impact: Compression, efficiency, understanding

Neural Tangent Kernel

🧠 **core Message**: Wide networks train like linear models

remember:

- $\Theta(x, x') = \langle \nabla f(x), \nabla f(x') \rangle$
- infinite width → fixed kernel

- finite width → feature learning
- architecture → kernel → bias

Practical Impact: Theory, architecture design, understanding

the big picture

```
More Parameters
      ↓
Better Optimization Landscape
      ↓
    /   \
   /     \
  LTH    NTK
   ↓      ↓
Sparse  Linear
Structure Dynamics
```

both lead to: **easier training + better generalization!**

Some resources used

[Noise, a flaw in human judgement] Daniel Kahneman, Oliver Sibony, Cass R. Sunstein, 2021