IMEN574/IDSC574 Programming for Data Science

15. **Deep Learning**

Spring 2026

Instructor: **Sangwoo Mo**

Lecture Outline

01

From Linear Models to Deep Networks

Why depth? MLP, activations, loss landscape, backpropagation

02

Making Depth Work

Vanishing gradients, init, norm, residuals, CNN, ResNet, dropout

03

The Transformer

Attention, positional encoding, Transformer block, GPT/BERT, ViT

04

Modern Optimization & Training

SGD→Adam, LR scheduling, training stability, full recipe

05

Scaling

Scaling laws, compute-optimal training, model scale progression

01

From Linear Models to Deep Networks

You know y = Wx + b and SGD. What changes when we go deep?

What You Already Know

The building blocks we'll extend

Linear Model

$$\hat{y} = Wx + b$$

Linear decision boundary
Hand-crafted features
Convex loss landscape
Closed-form or simple SGD

SGD

$$W \leftarrow W - \eta \nabla L(W)$$

Compute gradient on mini-batch
Update parameters
Converges for convex problems
Learning rate is the key knob

Deep learning keeps the same core loop (forward → loss → backward → update) but changes what the model can represent.

Why Go Deep?

Depth enables automatic, hierarchical feature learning

Shallow (Linear + Features)

You design the features:
TF-IDF, HOG, SIFT, n-grams ...

Performance bottleneck =
your feature engineering skill

Deep (Learned Features)

The network learns features:
Edges → Textures → Parts → Objects

Performance bottleneck =
data + compute + architecture

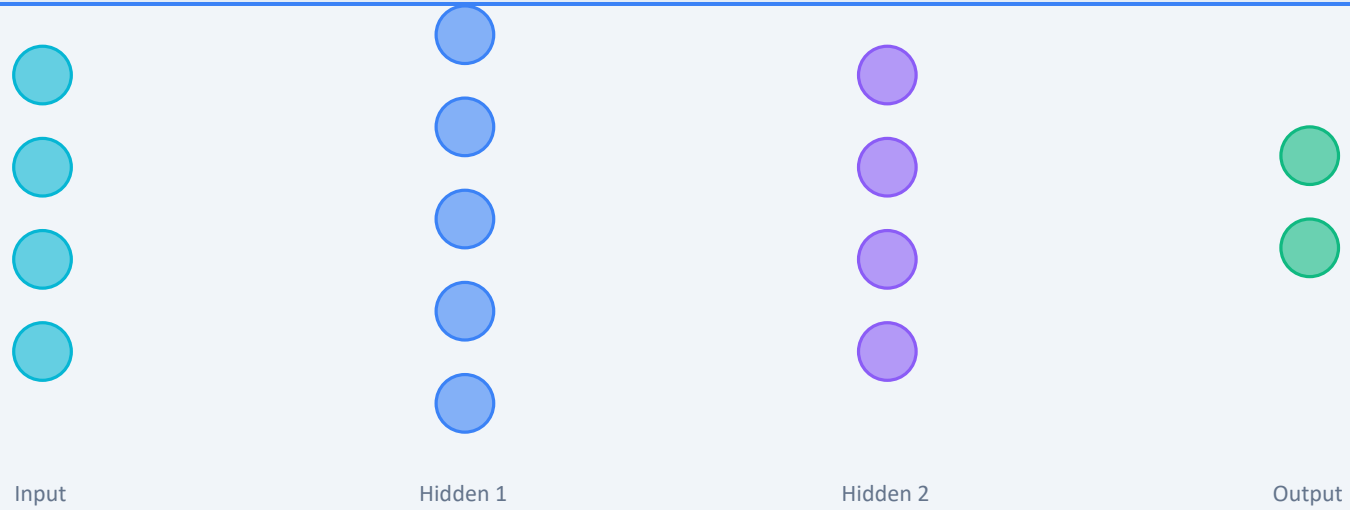
The Depth Efficiency Argument

A shallow network (1 hidden layer) can approximate any function (Universal Approximation Theorem), but may require exponentially many neurons. Depth gives exponential compression:
a deep network with N parameters can represent functions that a shallow network needs 2^p to match.

The Multi-Layer Perceptron (MLP)

Stack linear transforms with non-linear activations

$$h_1 = \sigma(W_1 x + b_1) \qquad h_2 = \sigma(W_2 h_1 + b_2) \qquad \hat{y} = W_3 h_2 + b_3$$



Without non-linearity σ, stacking layers collapses to a single linear transform: $W_2 W_1 x = W'x$. Activation functions are what make depth useful.

Activation Functions: What Matters Now

ReLU dominates. Know the variants and why.

ReLU

$\max(0, z)$

Default for most networks.
Simple, fast, sparse activations.
Dead neuron problem.

GELU

$z \cdot \Phi(z)$

Default in Transformers
(GPT, BERT, ViT).
Smooth approximation of ReLU.

SiLU / Swish

$z \cdot \sigma(z)$

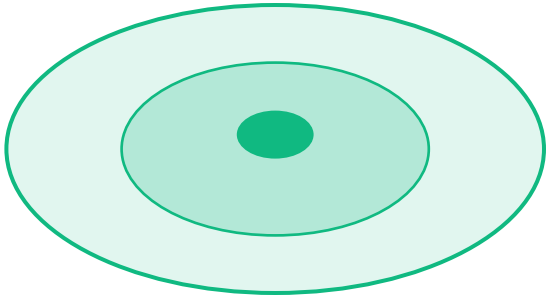
Used in modern CNNs
(EfficientNet) and LLMs.
Self-gated, smooth.

Sigmoid/Tanh are largely historical for hidden layers. Use ReLU as default; GELU for Transformers; SiLU for vision.

The Loss Landscape Changes

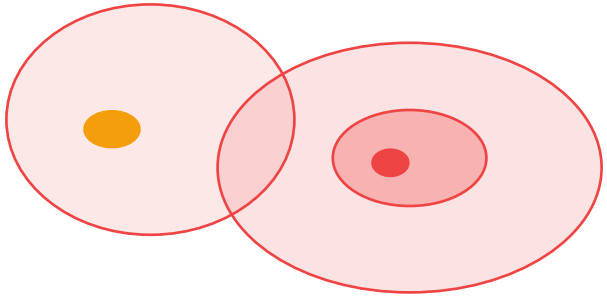
Linear models are convex. Deep networks are not.

Linear / Convex



One global minimum
SGD finds it

Deep / Non-Convex



Local minima, saddle points
SGD can get stuck

Implications for optimization:

Saddle points (not local minima) are the main obstacle in high dimensions. Momentum and adaptive LR (Adam) help escape them. This is why plain SGD struggles on deep networks but works fine on linear models.

Backpropagation & Computation Graphs

How gradients actually flow through a deep network

Forward Pass

$$
\begin{aligned}
&x \\
&\quad\downarrow\ W_1,\ b_1 \\
&z_1 = W_1 x + b_1 \\
&\quad\downarrow\ \sigma \\
&h_1 = \sigma(z_1) \\
&\quad\downarrow\ W_2,\ b_2 \\
&z_2 = W_2 h_1 + b_2 \\
&\quad\downarrow \\
&L = \mathrm{Loss}(z_2, y)
\end{aligned}
$$

Backward Pass (Chain Rule)

$$
\begin{aligned}
&\partial L/\partial z_2 = \partial \mathrm{Loss}/\partial z_2 \\
&\quad\uparrow \\
&\partial L/\partial W_2 = \partial L/\partial z_2 \cdot h_1^{\top} \\
&\partial L/\partial h_1 = W_2^{\top} \cdot \partial L/\partial z_2 \\
&\quad\uparrow \\
&\partial L/\partial z_1 = \partial L/\partial h_1 \cdot \sigma'(z_1) \\
&\partial L/\partial W_1 = \partial L/\partial z_1 \cdot x^{\top}
\end{aligned}
$$

Each layer multiplies by $\sigma'(z)$

Autograd (PyTorch, JAX) builds this graph automatically. You define forward pass; the framework computes all gradients via chain rule.

02

Making Depth Work

The tricks that made deep learning actually trainable

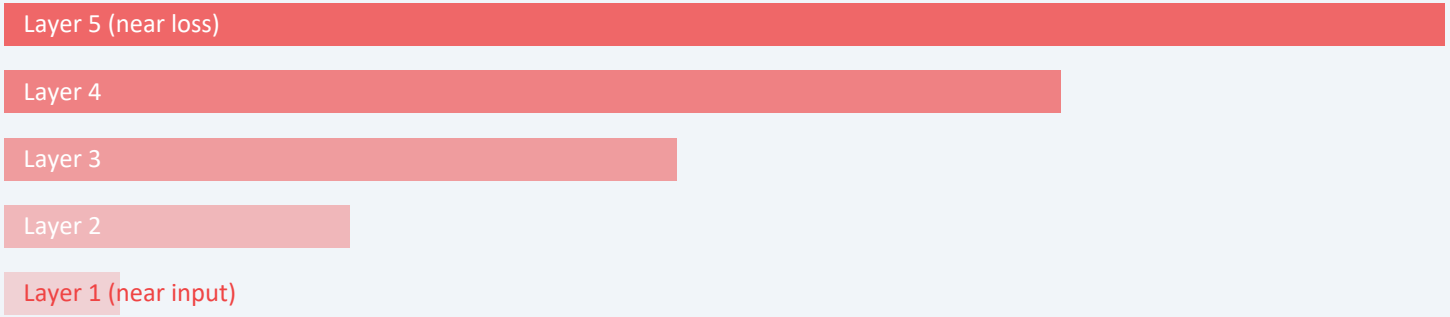
The Vanishing Gradient Problem

Why naïve deep networks don’t train

Backpropagation multiplies gradients through layers via chain rule:

$$\partial L/\partial W_1 = \partial L/\partial h_n \cdot \partial h_n/\partial h_{n-1} \cdot \ldots \cdot \partial h_2/\partial h_1 \cdot \partial h_1/\partial W_1$$

Gradient magnitude per layer (backward from loss):



If each $\partial h_i/\partial h_{i-1} < 1$, gradients shrink exponentially. With Sigmoid, this kicks in around 5+ layers. Three key solutions follow.

Weight Initialization

Set the starting point right, or training fails before it begins

Random (Naïve)

$W \sim N(0, 0.01)$

Activations shrink to 0
or explode in deep nets.
Don’t use this.

Xavier / Glorot

$W \sim N(0, 2/(n_{in}+n_{out}))$

Preserves variance across layers.
Designed for Sigmoid/Tanh.
Used with linear activations.

He / Kaiming

$W \sim N(0, 2/n_{in})$

Accounts for ReLU zeroing
half the activations.
Default for ReLU networks.

Goal: keep the variance of activations and gradients roughly constant across layers. Good init alone isn’t enough for very deep nets.

Normalization: BatchNorm vs LayerNorm

Stabilize activations during training

Batch Normalization

$$\hat{x} = (x - \mu_B) / \sqrt{(\sigma^2_B + \varepsilon)}$$

Normalize across the batch dimension.

Used in: CNNs (ResNet, EfficientNet)

Pros: Smooths loss landscape, enables higher learning rates.

Cons: Behaves differently at train vs inference. Batch-size dependent.

Layer Normalization

$$\hat{x} = (x - \mu_l) / \sqrt{(\sigma^2_l + \varepsilon)}$$

Normalize across the feature dimension.

Used in: Transformers (GPT, BERT, ViT)

Pros: No batch dependency. Identical behavior at train and inference.

Cons: Slightly less effective for CNNs than BatchNorm.

Rule of thumb: BatchNorm for CNNs, LayerNorm for Transformers. Both normalize then apply learnable scale γ and shift β.

Residual Connections

The single most important architectural innovation for depth (He et al., 2015)

$$h_l = F(h_{l-1}) + h_{l-1} \qquad \text{("skip connection")}$$

Why It Works

Gradient flows directly through the skip connection ($\partial h_l / \partial h_{l-1} = 1 + \ldots$), avoiding vanishing gradients.

The network only needs to learn the **residual** F(x), not the full mapping.

Impact

Before ResNet: ~20 layers max
After ResNet: 100–1000+ layers

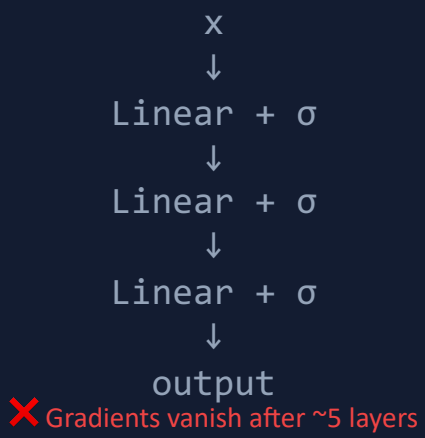
Used **everywhere** in modern DL: ResNet, Transformers, U-Net, Diffusion models, ...

The modern "Transformer block" = LayerNorm + Self-Attention + Residual + LayerNorm + FFN + Residual. All three ideas combined.

The Modern Building Block

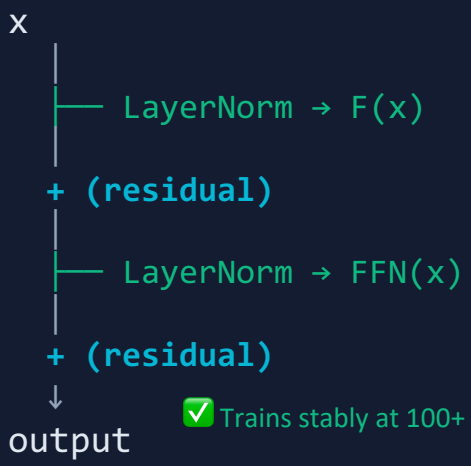
Init + Norm + Residual = trainable at any depth

Naïve Deep Network



❌ Gradients vanish after ~5 layers

Modern Block (Transformer-style)



✅ Trains stably at 100+ layers

Convolution & CNNs

Local patterns, weight sharing, and translation equivariance

The Convolution Operation

Slide a small filter (e.g. 3×3) across the input, computing dot products:

```
output(i,j) = Σ filter ⊙ input_patch(i,j)
```

Weight sharing: same filter everywhere → far fewer parameters than MLP

Translation equivariance: if the input shifts, the output shifts the same way

CNN Architecture Pattern

```
Input image  →  [Conv→BN→ReLU→Pool] × N

spatial dims shrink, channels grow

Flatten → FC → Output
```

Feature hierarchy:

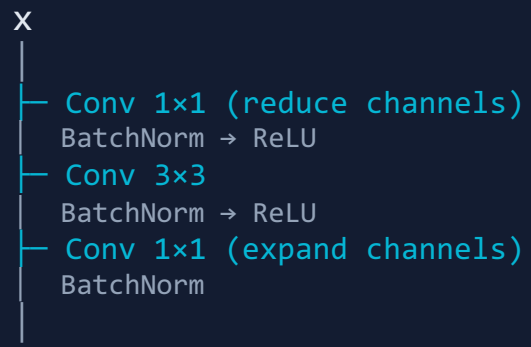
Early: edges, textures
Middle: parts, patterns
Late: objects, scenes

CNNs dominated vision (2012–2020). The inductive bias (locality + weight sharing) makes them data-efficient, but limits global reasoning.

ResNet: Putting It All Together

How residuals, normalization, and init enabled 150+ layer networks (He et al., 2015)

ResNet Bottleneck Block



Everything in Practice

He initialization — variance-preserving for ReLU

Batch Normalization — after every conv

Residual connections — skip around block

Bottleneck design — 1×1→3×3→1×1

ResNet-50: 25M params, 50 layers
Still a strong baseline in 2024.

Dropout & Regularization

Preventing overfitting in over-parameterized networks

Dropout (Srivastava et al., 2014)

Training: randomly zero out neurons with prob p

Inference: use all neurons, scale by (1−p)

$$h = \text{mask} \odot \sigma(Wx + b)$$

$$\text{mask} \sim \text{Bernoulli}(1-p)$$

Forces redundant representations.
Acts as an ensemble of sub-networks.
Typical p = 0.1–0.5.

Other Techniques

Weight Decay
Shrink weights each step (λ=0.01–0.1)

Data Augmentation
Crops, flips, color jitter (vision)
Back-translation, masking (NLP)

Early Stopping
Stop when val loss plateaus

Modern trend: large models are under-regularized by design (no dropout in GPT-3). Weight decay + data scale act as implicit regularizers.

Overfitting & Generalization

The gap between training performance and real-world performance

Diagnosing with Learning Curves

Train ↓ Val ↑ → Overfitting

Both high & flat → Underfitting
(model too small or LR too low)

Both low, small gap → Good fit

Always plot train & val loss.
Train loss alone tells you nothing.

The Modern Perspective

Double Descent

Beyond the interpolation threshold, larger models generalize BETTER (contradicts classical bias-variance tradeoff).

Scaling > Regularization

With enough data, bigger models generalize without explicit regularization.
GPT-3: no dropout, just weight decay.

Classical ML: fight overfitting with regularization. Modern DL: scale up data and model, regularization becomes less critical.

03

The Transformer

The architecture that unified NLP, vision, and beyond

Why Attention?

The problem with processing sequences

RNN Bottleneck

Sequential processing:

$h_t = f(h_{t-1}, x_t)$

Information must flow through every step. Long-range dependencies are lost. Cannot parallelize across time.

O(n) sequential steps = slow.

Attention Solution

Direct access to every position:

$\text{Attention}(Q, K, V)$

Every token can directly attend to every other token. No information bottleneck. Fully parallelizable.

O(1) sequential depth = fast on GPUs.

"Attention Is All You Need" (Vaswani et al., 2017) — replaced recurrence entirely with attention. Changed everything.

Self-Attention: The Core Idea

Each element computes a weighted sum of all others, based on relevance

For each token, we compute three vectors from the input:

Q (Query) = "What am I looking for?" **K (Key)** = "What do I contain?" **V (Value)** = "What do I provide?"

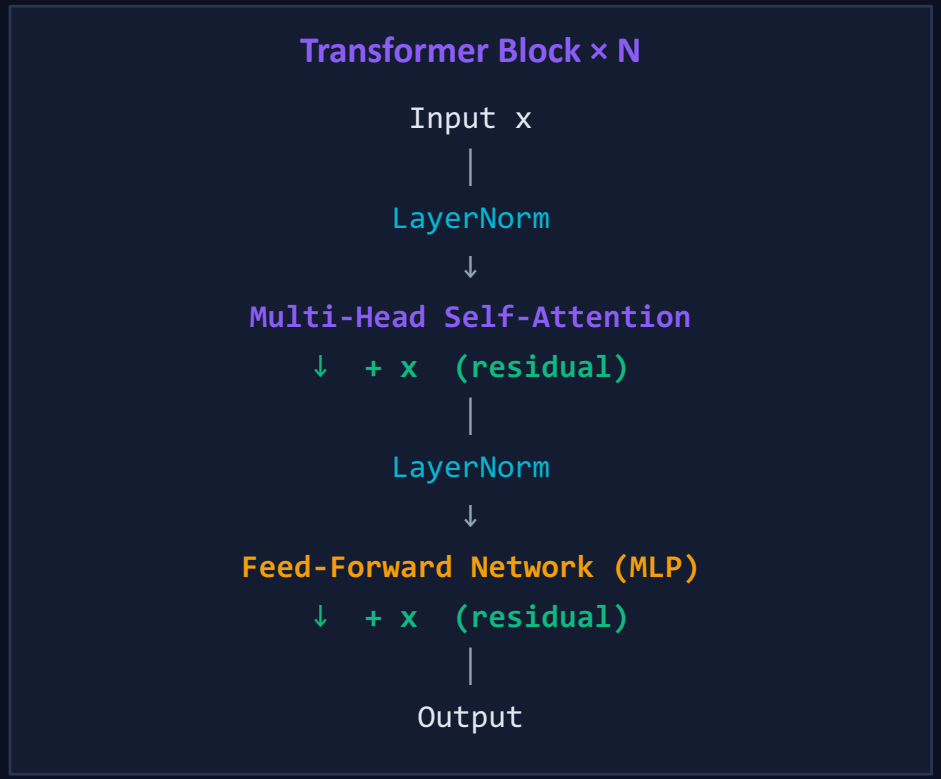
Attention score = how much Query matches each Key. Output = weighted sum of Values.

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k}) \cdot V$$

$\sqrt{d_k}$ scaling prevents dot products from growing too large (which pushes softmax into saturation).
Multi-Head Attention runs h independent attention heads in parallel, each learning different relationship patterns.

The Transformer Block

The repeating unit stacked N times



Key Properties

Permutation equivariant — must add position info

O(n^2) attention cost — quadratic in seq length

Pre-Norm (modern default) — LN before attention/FFN, more stable

Positional encoding — sinusoidal, learned, or RoPE

Positional Encoding

Attention is permutation-equivariant — we must inject position information

Problem: Self-attention treats input as a **set**, not a **sequence**. "The cat sat" and "sat the cat" produce identical outputs.

Solution: Add a position-dependent vector to each token embedding before feeding into the Transformer.

Sinusoidal (Original)

PE(pos, 2i) = sin(pos / 10000^{2i/d})
PE(pos, 2i+1) = cos(...)

Fixed, no learnable params.
Can extrapolate to unseen lengths.
Used in original Transformer.

Learned Embeddings

PE = Embedding(pos)

A learnable lookup table.
Simple and effective.
Used in GPT-2, BERT.
Cannot extrapolate beyond training length.

RoPE (Rotary)

Rotate Q and K vectors by position-dependent angles.

Relative position encoding.
Extrapolates well with NTK.
Modern default: LLaMA, Gemma, Mistral.

Transformers for Language

Two paradigms that reshaped NLP

Decoder-Only (GPT-style)

Autoregressive: predict next token

$$P(x_t \mid x_1, \ldots, x_{t-1})$$

Causal mask prevents attending to future tokens. Train on next-token prediction (language modeling).

GPT-2/3/4, LLaMA, Claude

Encoder-Only (BERT-style)

Bidirectional: see full context

$$P(x_t \mid x_1, \ldots, x_{t-1}, x_{t+1}, \ldots)$$

No causal mask. Train with masked language modeling (predict [MASK]). Better for understanding tasks.

BERT, RoBERTa, DeBERTa

The industry has largely converged on decoder-only (GPT-style) for generalist models. Encoder-only remains strong for embeddings and retrieval.

Vision Transformer (ViT)

Applying Transformers to images (Dosovitskiy et al., 2020)



Key Idea

Treat an image as a sequence of patches, just like tokens in text.

No convolutions needed — pure attention over patch embeddings. 196 patches for 224×224 / 16×16.

Implications

Same architecture for text and vision. Unifies modalities under one framework.

Needs more data than CNNs to train (less inductive bias), but scales better with compute.

ViT showed that the Transformer is not just for NLP. Today: vision, audio, proteins, robotics — one architecture to rule them all.

04

Modern Optimization

Adam, scheduling, and the tricks that actually matter

The Road to Adam

How each innovation addresses a specific failure of plain SGD

SGD

$$W \leftarrow W - \eta \nabla L$$

Oscillates in steep dimensions, slow in flat ones. Same LR for all parameters.

+ Momentum

$$v \leftarrow \beta v + \nabla L; \quad W \leftarrow W - \eta v$$

→ Adds inertia to escape local minima

Accumulates velocity → faster convergence. But still one LR for all params.

+ Adaptive LR (Adam)

$$m, v = \text{moments}; \quad W \leftarrow W - \eta \cdot m / \sqrt{v}$$

→ Different LR per parameter

Per-parameter learning rate from gradient history. Fast + adaptive.

Each step solves one problem: momentum adds inertia, adaptive LR handles heterogeneous gradients. Adam combines both.

Adam: The Default Optimizer

Adaptive moments for each parameter (Kingma & Ba, 2014)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$W_t = W_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$$

m_t = 1st moment (momentum)

v_t = 2nd moment (per-param LR)

$\hat{}$ = bias correction β_1=0.9, β_2=0.999, ε=1e-8

SGD + Momentum

Simple, well-understood.
Generalizes slightly better.
Needs careful LR tuning.

Adam

Fast convergence. Adapts per-parameter. Default for Transformers.
Most used in practice.

AdamW

Adam + decoupled weight decay.
Fixes L2 bug in Adam.
Standard for modern LLMs.

Learning Rate Scheduling

The LR trajectory matters as much as the peak value

Modern Recipe: Warmup + Cosine Decay

1. Warmup (first ~1-5% of steps): linearly increase LR from ~0 to peak. Stabilizes early training.

2. Peak LR (typical: 1e-4 to 3e-4 for Transformers): the main learning rate.

3. Cosine Decay (rest of training): smoothly anneal LR to ~0 following a cosine curve.

Why Warmup?

At initialization, Adam's moment estimates (m, v) are zero.
Large updates early on can destabilize training. Warmup gives the optimizer time to calibrate its estimates.

Weight Decay (λ)

$W \leftarrow (1 - \lambda\eta)\, W - \eta\, \nabla L$

Shrinks weights each step. Acts as implicit regularization. Typical $\lambda = 0.01–0.1$.

AdamW decouples this from the adaptive LR — crucial difference.

Training Stability Tricks

Small details that prevent large failures

Gradient Clipping

Cap the gradient norm before updating:
if ||∇L|| > threshold: ∇L ← threshold · ∇L / ||∇L||

Prevents catastrophic updates from bad batches or loss spikes.
Standard in all LLM training.

Mixed Precision (fp16/bf16)

Forward pass in half-precision (16-bit), master weights in fp32.

~2× throughput, ~2× memory savings.
bf16 preferred (no loss scaling needed).
Enabled by default in modern frameworks.

Gradient Accumulation

Simulate large batch sizes by accumulating gradients over N steps before one update.

Effective batch = N × micro-batch.
Critical when GPU memory is limited.

The Complete Training Recipe

A checklist for training any deep network in 2024+

① **Architecture**	Transformer (or MLP/CNN for the domain). Residual + LayerNorm.
② **Initialization**	He init (ReLU) or Xavier (GELU/SiLU). Use framework defaults.
③ **Optimizer**	AdamW (β_1=0.9, β_2=0.95–0.999, ε=1e-8).
④ **LR Schedule**	Linear warmup (1–5% steps) → cosine decay to 0.
⑤ **Regularization**	Weight decay λ=0.01–0.1. Dropout only if small model/data.
⑥ **Precision**	bf16 mixed precision. Gradient clipping (max_norm=1.0).
⑦ **Monitor**	Track train & val loss. Watch for divergence. Save checkpoints.

This recipe works for ~90% of cases. Tune LR and batch size first; architecture changes last.

05

Scaling

Why bigger models don’t just do “more of the same”

Scaling Laws

Predictable power-law relationships between compute, data, params, and loss

Key Finding (Kaplan et al., 2020; Hoffmann et al., 2022):

Loss follows a power law: $L(N) \propto N^{-\alpha}$ where N = parameters, data, or compute.

Chinchilla Optimal

For a fixed compute budget, there's an optimal ratio of model size to training data.

Chinchilla (70B, 1.4T tokens) matched Gopher (280B, 300B tokens) with 4× fewer parameters but 4× more data.

Rule: ~20 tokens per parameter.

Emergent Abilities

Some capabilities appear only beyond a certain scale (chain-of-thought, in-context learning, few-shot reasoning).

Small models: near-zero performance.
Large models: sudden jump.

Implies scale is not just quantitative.

Scale of Real Architectures

Same building blocks, 1000× more of them

Model	Year	Layers	Params	Key Innovation
ResNet-50	2015	50	25M	Residual connections
BERT-Base	2018	12	110M	Bidirectional pretraining
GPT-2	2019	48	1.5B	Autoregressive LM at scale
ViT-Large	2020	24	307M	Patches → Transformer
GPT-3	2020	96	175B	In-context learning emerges
LLaMA-2	2023	80	70B	Open-weight, efficient

The building blocks are the same (Transformer + Residual + LayerNorm + Adam). Scale is the variable.

Key Takeaways

1. Deep networks learn hierarchical features automatically — replacing hand-crafted pipelines.
2. Residual connections + normalization + proper init made depth actually trainable.
3. The Transformer (attention + residual + LayerNorm) is the universal architecture for text, vision, and beyond.
4. AdamW + warmup + cosine decay is the modern optimization recipe.
5. Scaling laws are predictable: more compute → lower loss, but balance model size with data.

The Modern Deep Learning Stack

Scaling	More compute, more data, predictable gains
Optimization	AdamW + warmup + cosine decay + weight decay
Architecture	Transformer blocks (Attention + FFN + Residual + LayerNorm)
Training Dynamics	He init + LayerNorm + residual connections + gradient clipping
Foundation	Backpropagation + automatic differentiation + GPU compute