Deep Learning

Fabian Bosshard

February 4, 2026

Contents

1 Polynomial Regression with Gradient Descent

1.1 Model and data

- Polynomial model $p(z) = \sum_{k=0}^{4} w_k z^k$.
- Feature vector $\boldsymbol{x} = [1, z, z^2, z^3, z^4]^T$ and prediction $\hat{y} = \boldsymbol{x}^T \boldsymbol{w}$.
- Data generation $y_i = p(z_i) + \varepsilon_i$ with $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$.
- Design matrix $\underline{\boldsymbol{X}}$ contains rows $\boldsymbol{x}_i^T$.

1.2 Gradient descent and implementation details

- Mean-squared error
$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \big(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\big)^2$$
- Gradient in matrix form
$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = -\frac{2}{N} \underline{\boldsymbol{X}}^T \big(\boldsymbol{y} - \underline{\boldsymbol{X}} \boldsymbol{w}\big)$$
- Update rule
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}\big(\boldsymbol{w}^{(t)}\big)$$
- Learning rate $\eta > 0$ sets the step size. Too small η gives slow convergence. Too large η yields oscillation or divergence. In code, $\eta = 0.01$ gave a smooth decrease and reached near noise-level MSE in reasonable steps.
- Bias handling in `nn.Linear`: since $\boldsymbol{x}$ already contains a constant feature, using `bias=False` makes w_0 act as the bias. If `bias=True` is used while keeping the constant feature, the model has two redundant bias terms.
- Training bookkeeping:
  - A **step** is one parameter update (one batch forward/backward plus `optimizer.step()`).
  - An **epoch** is one full pass over the training set.
  - For mini-batches, steps per epoch equal num_samples/batch_size.
- Optimizer calls:
  - `optimizer.zero_grad()` clears accumulated gradients.
  - `optimizer.step()` updates parameters using the stored gradients (here, the SGD rule).

1.3 Generalization, expected optimum loss, and common losses

- Validation set: held-out data (no gradient updates) to monitor generalization and tune hyperparameters without touching the test set. Overfitting appears as training loss decreasing while validation loss increases. Typical countermeasures are smaller models, explicit regularization such as weight decay (L_2), more data or augmentation, and early stopping.
- Expected MSE at the optimum: if the model recovers $\boldsymbol{w}^*$ and $y = \boldsymbol{x}^T \boldsymbol{w}^* + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ then
$$\mathrm{E}\big[\big(y - \boldsymbol{x}^T \boldsymbol{w}^*\big)^2\big] = \sigma^2$$
with $\sigma = 0.5$ giving an expected optimum MSE of about 0.25.
- Classification losses: binary cross-entropy for binary classification and cross-entropy for multi-class classification.
- Adaptive optimizers use per-parameter learning rates based on past gradients. Examples are AdaGrad, RMSProp, and Adam. SGD uses a fixed learning rate and is a simple baseline.

2 Convolutional Neural Networks

2.1 Dataset & Basics

- CIFAR-10: 50k train, 10k test, RGB images of size $3 \times 32 \times 32$, 10 classes.
- Normalization: per-channel mean and standard deviation so inputs have roughly mean 0 and variance 1.
- Train/val split: original test set split into validation and final test.

2.2 Parameters

$$\#\text{params} = (k_w k_h\, C_{\text{in}} + 1)\, C_{\text{out}}$$

(bias)

- Convolutional layers: filter weight tensors + optional bias vectors.
- Fully connected layers: weight matrices $\underline{\boldsymbol{W}}$ + bias vectors.
- BatchNorm: scale γ and shift β parameters.
- All trainable parameters are in `model.parameters()` (tensors on the chosen device).

$$n_{\text{out}} = \left\lfloor \frac{n_{\text{in}} + 2p - d(k-1) - 1}{s} + 1 \right\rfloor$$

(padding, kernel size, stride)

2.3 Architecture choices and training practice

Why convolutional layers instead of fully connected

- Local receptive fields exploit spatial locality (neurons see small patches).
- Weight sharing reuses each filter across spatial positions, giving far fewer parameters than a dense layer on raw pixels.
- Translation equivariance means a shift in the input induces a corresponding shift in feature maps.
- A fully connected model on $3 \times 32 \times 32$ inputs would have a huge number of parameters and would ignore spatial structure.

Pooling layers

- Role: downsample feature maps (e.g. max pool 2×2), increase the effective receptive field in deeper layers, reduce activations before fully connected layers, and provide some invariance to small translations.
- If pooling is removed: feature maps stay large, which increases parameters and memory, raises overfitting risk, and slows training. A common substitute is strided convolutions or other downsampling.

Dropout, BatchNorm, and other regularization

- Dropout during training zeros activations with probability p. It reduces co-adaptation and overfitting. Typically p is smaller in convolutional blocks and larger before the final dense output layer.
- Batch Normalization normalizes mini-batch activations to roughly zero mean and unit variance and then applies learnable scale and shift parameters γ and β. It stabilizes gradients, speeds up training, allows larger learning rates, and adds some regularization due to batch noise.
- Additional regularization includes weight decay (L_2 penalty), data augmentation, early stopping, and using a smaller CNN.

Data augmentation

- `RandomCrop(32, padding=4)` introduces small translations and zoom-like effects.
- `RandomHorizontalFlip` encourages left–right invariance.
- `RandomPerspective` adds mild geometric distortions.
- Overall, augmentation makes the effective training set larger and more diverse and improves validation and test generalization.

Activation functions

- ReLU uses $f(x) = \max(0, x)$ with a hard cutoff at 0.
- GeLU is smooth and down-weights negative values rather than discarding them.
- GeLU can yield smoother optimization and sometimes a small accuracy gain in vision settings.

Optimization settings in the experiment

- Base model: learning rate about 0.03, training for 5 epochs, batch size 32. This learns quickly (reaching at least 65% test accuracy) but tends to overfit if trained much longer without strong regularization.
- Improved model: learning rate about 0.01 together with BatchNorm and a scheduler, training for 40 epochs, batch size 64. Strong regularization (dropout, augmentation, weight decay) makes longer training beneficial.

Momentum, weight decay, and learning-rate scheduling

- Momentum (e.g. 0.9) keeps an exponential moving average of gradients, giving smoother updates, less oscillation, and faster progress along consistent directions.
- Weight decay (e.g. 10^{-6}) adds an L_2 penalty and encourages smaller weights, which reduces overfitting.
- `ReduceLROnPlateau` monitors validation loss and decreases the learning rate when validation performance stops improving. A common strategy is a larger learning rate early and a smaller one later for fine-tuning.

Overfitting and run-to-run variability

- If training loss decreasing while validation loss flattens or increases $\Rightarrow$ overfitting.
- Try using a deeper network with more channels and add augmentation, dropout, BatchNorm, weight decay, scheduler. This reduces the train-validation gap and substantially improves test accuracy.
- Different random seeds change initialization $\boldsymbol{w}$ and mini-batch order. Test accuracy can vary by a few percentage points, so comparisons should average over runs or at least account for variance.

3 Recurrent models (RNN and LSTM)

Sequence processing: Given an input sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, a recurrent layer processes it *step by step* while carrying a state forward in time.

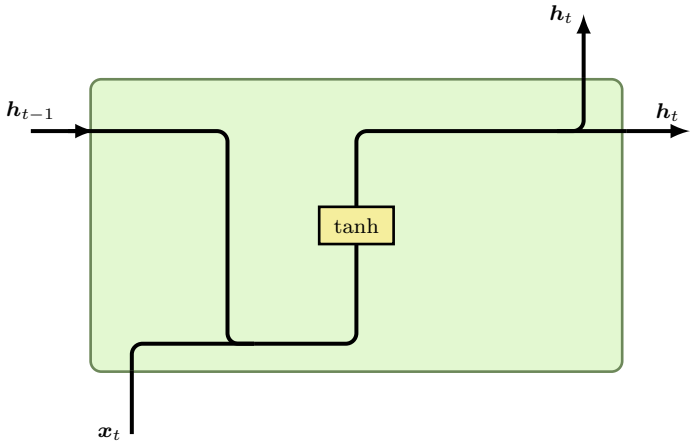
3.1 RNN

Hidden state (RNN): The *hidden state* $\boldsymbol{h}_t \in \mathbb{R}^h$ is the internal state of a recurrent layer at time t. It summarizes past information and is used (i) to produce outputs and (ii) to influence the next state.

Vanilla RNN update rule: A standard RNN layer is defined by one parameter set (shared across time):

$$\boldsymbol{h}_t = \tanh\big(\underline{\boldsymbol{W}}_{xh}\boldsymbol{x}_t + \underline{\boldsymbol{W}}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h\big)$$

where $\underline{\boldsymbol{W}}_{xh} \in \mathbb{R}^{h\times d}$, $\underline{\boldsymbol{W}}_{hh} \in \mathbb{R}^{h\times h}$, $\boldsymbol{b}_h \in \mathbb{R}^h$. At the first step one initializes $\boldsymbol{h}_0$ (typically $\mathbf{0}$, sometimes learned).

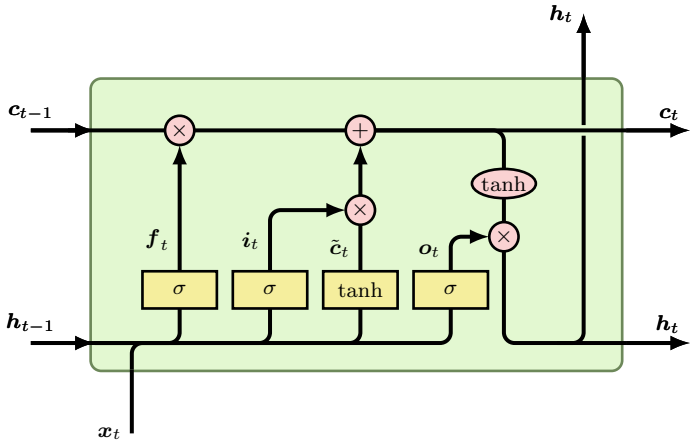


Unrolling and weight sharing: A *single* recurrent layer is *unrolled* over timesteps: the boxes drawn at different t are *copies of the same layer*. Thus, $\underline{\boldsymbol{W}}_{xh}, \underline{\boldsymbol{W}}_{hh}, \boldsymbol{b}_h$ are *shared across all timesteps*. (Weights are *not* shared across *stacked* layers.)

- issue: exploding gradient. solution: gradient clipping.
- issue: vanishing gradient. solution: LSTM.

3.2 LSTM

two states: An LSTM keeps $\boldsymbol{h}_t \in \mathbb{R}^h$ (hidden/output state; what is exposed), and $\boldsymbol{c}_t \in \mathbb{R}^h$ (cell state; long-term memory). Initialize $(\boldsymbol{h}_0, \boldsymbol{c}_0)$ (typically $(\mathbf{0}, \mathbf{0})$).



LSTM gates and update: Gates are elementwise values in $[0, 1]$ (via σ) that control *forget, write, expose*:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\underline{\boldsymbol{W}}_{xf}\boldsymbol{x}_t + \underline{\boldsymbol{W}}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{i}_t &= \sigma(\underline{\boldsymbol{W}}_{xi}\boldsymbol{x}_t + \underline{\boldsymbol{W}}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \qquad & \boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \\
\tilde{\boldsymbol{c}}_t &= \tanh(\underline{\boldsymbol{W}}_{xc}\boldsymbol{x}_t + \underline{\boldsymbol{W}}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \qquad & \boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \\
\boldsymbol{o}_t &= \sigma(\underline{\boldsymbol{W}}_{xo}\boldsymbol{x}_t + \underline{\boldsymbol{W}}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o)
\end{aligned}$$

Forgetting is $\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}$, *adding/writing* is $\boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$.

Why LSTM helps vanishing gradients (one line):

$$\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \operatorname{diag}(\boldsymbol{f}_t)$$

so if $\boldsymbol{f}_t \approx \mathbf{1}$, gradients can flow through many timesteps.

long sequences could require too much memory
→ truncated backpropagation through time (TBPTT): break sequences into chunks: fixed-length segments from the original sequences. TBPTT keeps dependencies within chunks while avoiding excessive memory usage.

3.3 Hidden state in an RNN

$\boldsymbol{h}_t$ is the internal state at time t; it summarizes past inputs and is used to compute (i) outputs and (ii) the next hidden state. It is updated recurrently: $\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$ with shared weights across time.

3.4 Padding for variable-length sequences

Batches require equal tensor sizes, so shorter sequences are padded (usually with a special PAD token) to the max length in the batch. In practice one uses a *mask* to ignore padded positions in loss/metrics/attention, or uses packed sequences (concept: process only true lengths, then unpack).

3.5 Stacked RNN

A stacked RNN has multiple recurrent layers. Layer ℓ takes input $\boldsymbol{h}_t^{(\ell-1)}$ (from the layer below at the same t) and its own previous state $\boldsymbol{h}_{t-1}^{(\ell)}$:

$$\boldsymbol{h}_t^{(\ell)} = f^{(\ell)}\big(\boldsymbol{h}_t^{(\ell-1)}, \boldsymbol{h}_{t-1}^{(\ell)}\big)$$

Why it helps: higher layers can learn more abstract / longer-range temporal features. Weights are shared across time *within a layer*, but *not shared across layers*.

3.6 Bidirectional RNN

A bidirectional RNN runs one RNN forward ($1 \to T$) and one backward ($T \to 1$), then combines states (often concatenation):

$$\boldsymbol{h}_t^{\text{bi}} = [\boldsymbol{h}_t^{\rightarrow};\ \boldsymbol{h}_t^{\leftarrow}]$$

Advantage when the whole sequence is available (tagging, classification). Not suitable for strictly causal/online prediction where future inputs are unknown.

3.7 Exploding gradients in RNNs

Backprop through time multiplies many Jacobians; norms can grow exponentially, causing unstable updates (very large gradients, NaNs/divergence).

3.8 Vanishing gradients (especially severe in RNNs)

The same repeated multiplication can make gradients shrink exponentially, so early timesteps get almost no learning signal. RNNs are particularly affected because the effective depth is the sequence length T. Mitigations: LSTM/GRU gating, careful initialization, normalization, residual connections, shorter BPTT windows.

3.9 Gradient clipping

Clip gradients to prevent exploding updates, e.g. clip global norm:

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

Used commonly for RNN/LSTM training stability. (Helps exploding, not vanishing.)

3.10 Activation functions and vanishing/exploding

Sigmoid/tanh saturate: derivatives become small ⇒ vanishing gradients in deep/time-unrolled networks. ReLU avoids saturation in the positive region (often helps in feedforward nets) but can still explode in recurrent settings. LSTMs use sigmoid/tanh but avoid catastrophic vanishing via the *additive* cell update and the forget gate.

3.11 Truncated backpropagation through time (TBPTT)

Instead of backpropagating through all T steps, backprop only through the last K steps, detaching the graph periodically. Why: reduces memory/time and stabilizes training. Tradeoff: harder to learn dependencies longer than K.

3.12 One-hot encoding and limitations

Represent a token w from a vocabulary of size V as $\boldsymbol{e}_w \in \{0,1\}^V$ with exactly one 1. Limitations: high-dimensional and sparse; no notion of similarity between words; expensive for large V.

3.13 `nn.Embedding`

An embedding layer is a learnable lookup table $\underline{\boldsymbol{E}} \in \mathbb{R}^{V \times d}$. Given token id w, it outputs the dense vector $\boldsymbol{x} = \underline{\boldsymbol{E}}_{w,:} \in \mathbb{R}^d$. Preferred over one-hot: efficient and learns semantic similarity (nearby vectors for related words).

3.14 Gating mechanisms (LSTM/GRU)

Gates are elementwise controllers in $[0,1]$ (computed via σ) that decide how much information to keep, write, and expose. They allow context-dependent time scales (keep some components for long, overwrite others quickly).

3.15 How LSTM gates mitigate vanishing gradients

Core cell update:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \quad \Rightarrow \quad \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \operatorname{diag}(\boldsymbol{f}_t)$$

If $\boldsymbol{f}_t \approx \mathbf{1}$ for relevant components, gradients can propagate through many timesteps; the model learns when to remember/forget.

4 TSP with Transformers (encoder-decoder heuristic)

4.1 Problem setup (Euclidean TSP)

- Cities are points $p_i = (x_i, y_i) \in \mathbb{R}^2$.
- Goal: shortest Hamiltonian cycle (visit each city once, return to start).
- Edge costs: Euclidean distance $w_{uv} = \|p_u - p_v\|_2$.
- TSP is NP-hard $\Rightarrow$ use heuristics / learned approximations.

4.2 Dataset structure (NetworkX graphs)

Each sample is a tuple (G, `opt_tour`) where G is a complete weighted graph and `opt_tour` is the optimal cycle as a node list (starting/ending at 0).

- Node attribute `pos`: `pos`$[i] = (x_i, y_i)$ = city coordinates $p_i \in \mathbb{R}^2$.
- Edge attribute `weight`: `weight`$(u, v) = \|p_u - p_v\|_2$.
- Boolean edge attribute `tour`: `True` iff edge (u, v) lies on the provided optimal tour.

4.3 Data pipeline (PyTorch view)

- Model input:
$$\underline{\boldsymbol{X}} \in \mathbb{R}^{n\times 2}, \qquad \underline{\boldsymbol{X}}_{i,:} = (x_i, y_i)$$
- Target tour (closed cycle):
$$\boldsymbol{y} \in \{0, \dots, n-1\}^{n+1}, \qquad y_0 = y_n = 0$$
- Batch shapes: $\underline{\boldsymbol{X}} \in \mathbb{R}^{B\times n\times 2}$, $\boldsymbol{y} \in \mathbb{Z}^{B\times(n+1)}$.

4.4 Sinusoidal positional encoding

For pos $\in \{0, \dots, T-1\}$ and $i \in \{0, \dots, \frac{d_{\text{model}}}{2} - 1\}$:

$$\text{PE}_{\text{pos},2i} = \sin\Big(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\Big) \qquad \text{PE}_{\text{pos},2i+1} = \cos\Big(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\Big)$$

Purpose: attention alone is permutation-equivariant; positional information breaks symmetry so the model can represent a sequence/tour.

4.5 Transformer heuristic for TSP (encoder–decoder)

- Encoder builds context vectors for all cities from (x_i, y_i) (often with an extra node-id embedding).
- Decoder generates the tour autoregressively: given prefix $(y_0, \dots, y_{t-1})$, predict y_t.
- Causal mask in decoder self-attention blocks access to future steps.

4.6 Training objective (teacher forcing)

Let $\boldsymbol{y} = [y_0, \dots, y_n]$ be the optimal closed tour. Use

$$\boldsymbol{y}^{\text{src}} = [y_0, \dots, y_{n-1}], \qquad \boldsymbol{y}^{\text{tgt}} = [y_1, \dots, y_n]$$

Train with token-level cross-entropy (next-node prediction):

$$\mathcal{L} = -\frac{1}{Bn}\sum_{b=1}^{B}\sum_{t=1}^{n} \log p_\theta\big(y_t^{(b)} \mid y_{<t}^{(b)}, \underline{\boldsymbol{X}}^{(b)}\big)$$

Best checkpoint selected by validation loss.

4.7 Inference and evaluation

- Greedy autoregressive decoding starts with prefix [0]. For $t = 1, \ldots, n-1$ choose the highest-scoring unvisited node
$$y_t = \underset{j \notin \{y_0, \ldots, y_{t-1}\}}{\arg\max} \ell_\theta\big(j \mid y_{<t}, \underline{\boldsymbol{X}}\big)$$
where $\ell_\theta(\cdot)$ denotes the decoder logits. Close the cycle by appending 0.
- Tour length is $L(\text{tour}) = \sum_{(u,v) \text{ on tour}} w_{uv}$.
- Gap to optimum is $\text{gap} = \big(L(\text{heuristic}) - L(\text{opt})\big)/L(\text{opt})$.
- Baselines include a random tour and greedy nearest-neighbor (NetworkX `greedy_tsp`).

4.8 Practical extensions and core Transformer concepts

Scaling to larger instances and improving decoding

- As written, the approach typically does not scale directly if $n = 20$ is hard-coded in `nn.Embedding(n,...)` or in the output head dimension.
- To support $n = 50$ and variable n:
  - Train or finetune on $n = 50$ or on variable n with padding and masks.
  - Replace a fixed-n output head with a pointer-style attention scoring mechanism. The decoder scores each city by compatibility with its current state, which supports variable n.
  - Improve decoding with beam search, sampling plus best-of-k, and post-optimization such as 2-opt or local search.
  - Active search or test-time finetuning on each instance can further reduce tour length.

Attention and multi-head attention Attention computes content-dependent weighted averages. Each token or node aggregates information from others using similarity scores, enabling long-range interactions without recurrence. Multi-head attention runs several attention heads in parallel using different learned projections and then concatenates the results. This is useful because different heads can capture different relations such as local geometry, global structure, and symmetries.

Positional encodings Without positional information, attention is permutation-equivariant. Positional encodings inject order so the decoder can represent the step t in a tour.

Feed-forward network A position-wise MLP adds nonlinearity and feature mixing within each token embedding
$$\text{FFN}(\boldsymbol{h}) = \underline{\boldsymbol{W}}_2\, \phi\big(\underline{\boldsymbol{W}}_1 \boldsymbol{h} + \boldsymbol{b}_1\big) + \boldsymbol{b}_2$$

LayerNorm For $\boldsymbol{h} = (h_1, \ldots, h_d) \in \mathbb{R}^d$, LayerNorm normalizes across the feature dimension. Define
$$\mu = \frac{1}{d}\sum_{i=1}^{d} h_i \qquad \sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(h_i - \mu)^2$$
then
$$\hat{h}_i = \frac{h_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \text{LN}(\boldsymbol{h}) = \boldsymbol{\gamma} \odot \hat{\boldsymbol{h}} + \boldsymbol{\beta}$$
with $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$. For a sequence tensor $\underline{\boldsymbol{H}} \in \mathbb{R}^{B \times T \times d}$, LayerNorm is applied independently to each token $\underline{\boldsymbol{H}}_{b,t,:}$. It is independent of batch size and works well for variable-length sequences and autoregressive inference. BatchNorm depends on batch statistics and is awkward or unstable in sequence models.

Queries, keys, values, scaling, and masking Given token embeddings $\underline{\boldsymbol{H}} \in \mathbb{R}^{B\times T\times d}$ and per-head dimension d_k,

$$\underline{\boldsymbol{Q}} = \underline{\boldsymbol{H}}\boldsymbol{W}_Q \qquad \underline{\boldsymbol{K}} = \underline{\boldsymbol{H}}\boldsymbol{W}_K \qquad \underline{\boldsymbol{V}} = \underline{\boldsymbol{H}}\boldsymbol{W}_V$$

with $\underline{\boldsymbol{Q}}, \underline{\boldsymbol{K}}, \underline{\boldsymbol{V}} \in \mathbb{R}^{B\times T\times d_k}$. Queries ask what to attend to, keys describe content, and values are aggregated. Dot products grow with d_k, so scaling by $\sqrt{d_k}$ avoids softmax saturation and tiny gradients. Causal masking blocks access to future positions so that step t depends only on $< t$.

Teacher forcing, tokenization, and training paradigms Teacher forcing feeds the ground-truth prefix $(y_0, \ldots, y_{t-1})$ during training to predict y_t. At inference time the model conditions on its own predictions, so errors can accumulate. Tokenization maps text to discrete symbols and affects sequence length, vocabulary size, and what can be represented efficiently. Pretraining uses a large-scale generic objective, usually self-supervised, to learn broad representations. Finetuning adapts the model to a specific task or domain using task data and objectives.

Decoding strategies Greedy decoding chooses arg max at each step and is fast but can be suboptimal. Sampling draws from the distribution, optionally with temperature, top-k, or top-p, and can improve diversity. Beam search keeps the top k partial sequences, often improving quality at increased compute cost.