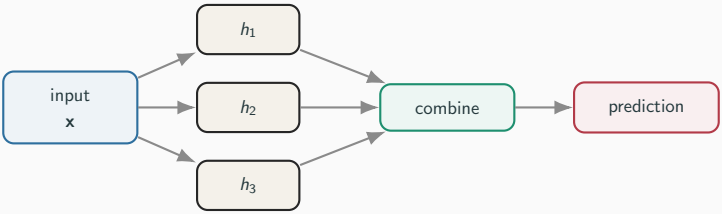
Ensemble Methods; Self-Supervised Learning and Transformers

Can many mediocre models beat one great one?
Can a model learn with no labels at all?

CSCI 567 — Machine Learning • Lecture 08 • Oct 30, 2026

Two questions today: combine many predictors (Part I), and learn without labels (Part II)



Every model so far was one $\mathbf{w}$ or θ, fit once and used alone; Part I combines B of them, each wrong in *different* ways. Part II asks what all of them left open: every one was fit to human-supplied labels, the scarce resource.

Part I: averaging things that disagree cancels their errors. **Part II**: invent the labels from the data itself. They meet at the end: today's largest language models are Transformers whose MLPs are mixtures of experts.

Outline

Part I: ensembles

1. **Wisdom of crowds**: averaging divides the variance by B; correlation caps the gain.
2. **Bagging**: bootstrap resamples, parallel fits, an average; measured on deep trees.
3. **Boosting**: sequential fits to the current mistakes; AdaBoost to zero training error.
4. **Gradient boosting and XGBoost**: fit the next learner to the negative gradient.
5. **Ensembling neural networks**: deep ensembles, model soups, mixture of experts.

Part II: learning without labels

6. **Self-supervised learning**: pretext tasks; composing with transfer learning.
7. **Language modelling**: next-token prediction; Markov models and RNNs as motivation.
8. **Attention**: query/key/value, worked example, scaling, masking, heads, positions.
9. **Transformers**: the block, the architecture, tokenization, encoder and decoder.
10. **Two worked instances**: BERT and SimCLR.

1. Wisdom of Crowds

The statistical case for averaging

Averaging B independent estimators divides the variance by B

Take B estimators $\hat{f}_1, \ldots, \hat{f}_B$ of the same quantity (say $f^\star(\mathbf{x}_0)$, Lecture 03's notation), each unbiased with variance σ^2, and for now independent. Average them: $\bar{f} := \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b$.

$$\mathrm{Var}(\bar{f}) = \frac{1}{B^2}\sum_{b=1}^{B}\sum_{b'=1}^{B} \mathrm{Cov}(\hat{f}_b, \hat{f}_{b'}) = \frac{1}{B^2}\sum_{b=1}^{B} \mathrm{Var}(\hat{f}_b) = \frac{\sigma^2}{B},$$

because independence kills every cross term $b \neq b'$ and only the B diagonal terms remain.

> Averaging keeps the mean (still unbiased) and divides the variance by B: enough independent noisy estimators average out to the truth, however noisy each one is.

This is the whole statistical content of "a large independent crowd's average guess beats almost any expert", and of why an ensemble can beat every one of its members. The catch is the word *independent*: B trees fit to the same n points are not independent draws, and they tend to make the same mistakes on the same hard examples.

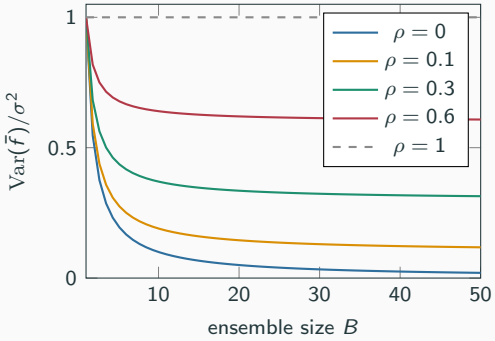
Correlated estimators: the variance has a floor

Keep $\mathrm{Var}(\hat{f}_b) = \sigma^2$ but let every pair be correlated, $\mathrm{Cov}(\hat{f}_b, \hat{f}_{b'}) = \rho\sigma^2$ for $b \neq b'$. There are $B(B-1)$ such pairs, so

$$\mathrm{Var}(\bar{f}) = \frac{1}{B^2}\left[B\sigma^2 + B(B-1)\rho\sigma^2\right]$$
$$= \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.$$

The second term still vanishes as $B \to \infty$. The first does not depend on B at all.

Floor. However many correlated estimators are averaged, $\mathrm{Var}(\bar{f}) \geq \rho\sigma^2$. At $\rho = 1$ (identical copies) averaging buys nothing; at $\rho = 0$ it is the previous slide.



The design principle. Averaging helps only to the extent the members *disagree*; every method below is a way to push ρ down without pushing σ^2 up.

2. Bagging

Bootstrap aggregating: parallel, variance-reducing

Bagging (Breiman, 1996): perturb the data B times, fit B times, average

Given the training set $S = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ and *any* base learner:

> for $b = 1, \dots, B$: draw a **bootstrap resample** S_b of size n from S *with replacement*, and fit the base learner to S_b to get h_b. Then combine: $f_B(\mathbf{x}) = \frac{1}{B}\sum_b h_b(\mathbf{x})$ for regression, a majority vote $\arg\max_y \sum_b \mathbb{1}[h_b(\mathbf{x}) = y]$ for classification.

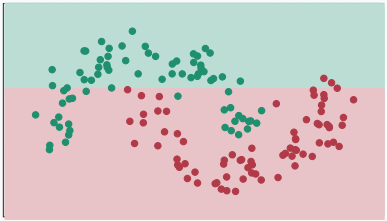
How different is a resample? A given point is left out of one draw with probability $1 - \frac{1}{n}$, so out of all n draws with

$$\mathbb{P}[i \notin S_b] = \left(1 - \frac{1}{n}\right)^n \xrightarrow{n\to\infty} \frac{1}{e} \approx 0.368 \qquad (n = 100: \ 0.99^{100} = 0.366).$$

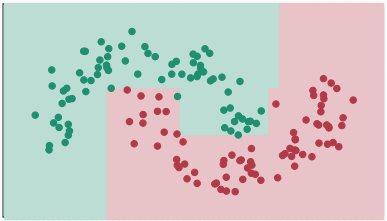
Each S_b contains about 63% of the distinct points, some repeated, and misses the other 37% entirely (the **out-of-bag** points, a free validation set for h_b). So the h_b are trained on genuinely different data, yet all drawn from the same n points: Section 1's correlated regime, with ρ small but not zero.

The running example: a decision tree, as a stump and grown deep

one split (a stump): 78.7% train accuracy



grown deep: 100% train accuracy

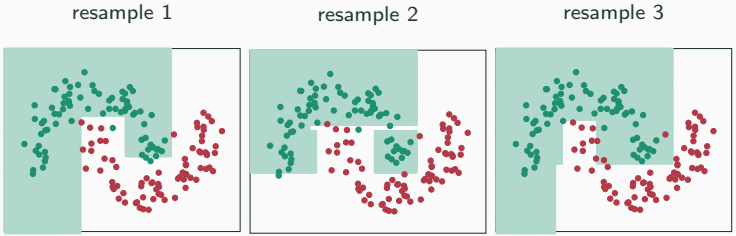


A **decision tree** splits the data at the feature and threshold that most reduce label impurity, then repeats inside each child until the leaves are pure or small. Its prediction is piecewise constant: the shaded regions on the left, for 150 points of two interleaving moons (Colab Part 1).

A **stump** stops after one split, one straight cut that cannot separate interleaving moons; a **deep** tree keeps splitting until it fits the training set exactly: low bias.

Bagging works with any base learner; trees are the example because a deep tree is *high-variance* (next slide). Bagging stable least-squares regression changes little.

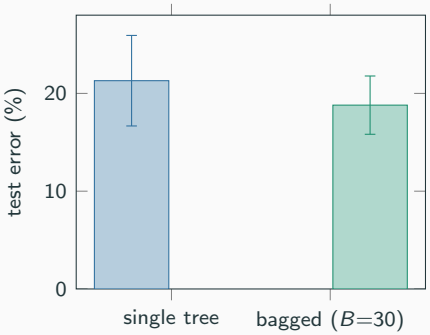
Why a deep tree is high-variance: the same learner on three bootstrap resamples



Green: the region a deep tree labels $+1$ after training on one bootstrap resample of the same 150 points (all points drawn for reference). The three trees agree with the full-data tree on 93%, 87%, and 96% of the plane.

A tree is greedy and cascading: the root split is chosen from all n points, and every split below sees only what its ancestors routed to it. Change a few training points and an early split can flip, after which the whole subtree restructures, so $\mathrm{Var}(h(\mathbf{x}_0))$ across training sets is large: three draws from the same data give three visibly different boundaries, differing most along the edges of the moons. That disagreement is what averaging cancels: a vote over many such trees keeps the shape they share and smooths away what only one of them believes.

Measured: bagging lowers both the mean and the variance of the test error



Mean ± 1 standard deviation over $R = 100$ training-set redraws (Colab Part 2).

Setup. Two moons, deliberately small and noisy ($n = 40$, noise 0.35). Redraw the training set 100 times; on each draw fit one deep tree and one bagged ensemble of 30 deep trees; score both on one fixed test set of 1000 points. Each redraw is one sample of S in Lecture 03's bias-variance sense.

	single tree	**bagged**
mean test error	21.3%	18.8%
variance across redraws	2.1×10^{-3}	0.9×10^{-3}

Both halves of Section 1's prediction appear: the mean error drops by 12% relative and the variance by 2.4×. Bagging makes the result more consistent across training sets, which is what "reduces variance" means.

Random forests (Breiman, 2001): push ρ down further by restricting features

Bootstrap resampling alone leaves every tree free to consider every feature at every split, and when one feature dominates, every tree makes the same early split anyway. The correlation ρ stays higher than it needs to be.

bagging alone

every split considers all features; the same dominant feature wins early in every tree

random forest

every split considers a fresh random subset of features (about $\sqrt{d}$ for classification)

This lowers ρ directly at a small cost in each tree's accuracy, the lever the floor $\rho\sigma^2$ identified and bagging alone does not pull. Random forests are a strong tabular default with almost no tuning; a tree-specific refinement, not developed further here. Bagging in one line: parallel, independent fits on perturbed data, averaged; it reduces variance and does nothing for bias, so a bagged ensemble of stumps is still a poor model.

3. Boosting

Sequential, bias-reducing

Boosting: fit learners in sequence, each one to the ensemble's current mistakes



Boosting builds the ensemble one member at a time: for $b = 1, \ldots, B$, look at where the current ensemble f_{b-1} does poorly, fit h_b to those points, and add it in. The members are **weak learners**, each only a little better than chance (a one-split stump), and dependent by construction, so this is not wisdom of crowds: boosting attacks **bias**, targeting the ensemble's remaining systematic errors round after round.

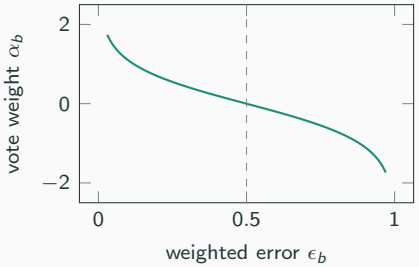
	bagging	**boosting**
fit	parallel, independent	sequential, dependent
reduces	variance	bias
base learner	high-variance (deep tree)	high-bias (stump)

AdaBoost (Freund and Schapire, 1997): reweight the points, weight the votes

Binary labels $\mathbf{y}_i \in \{-1, +1\}$; a distribution $p_b(i)$ over the training points, starting uniform, $p_1(i) = \frac{1}{n}$. For $b = 1, \dots, B$:

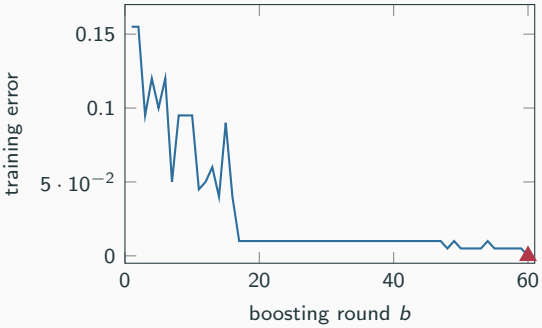
1. fit the weak learner h_b to minimize the *weighted* error under p_b;
2. its weighted error: $\epsilon_b = \sum_i p_b(i)\, \mathbb{1}[h_b(\mathbf{x}_i) \neq \mathbf{y}_i]$;
3. its vote weight: $\alpha_b = \frac{1}{2} \ln \frac{1-\epsilon_b}{\epsilon_b}$;
4. reweight: $p_{b+1}(i) \propto p_b(i) \exp\big(-\alpha_b\, \mathbf{y}_i\, h_b(\mathbf{x}_i)\big)$, normalized: up on what h_b got wrong ($\mathbf{y}_i h_b < 0$), down on the rest, so h_{b+1} targets what is still missed.

Output the weighted vote
$f_B(\mathbf{x}) = \operatorname{sign}\big(\sum_b \alpha_b h_b(\mathbf{x})\big)$.



The vote weight is calibrated to the learner's accuracy: small ϵ_b gives a loud vote, a coin flip ($\epsilon_b = \frac{1}{2}$) is ignored, and worse than random gets a negative weight (invert it).

Measured: stumps that are 79% accurate alone reach 0% training error together



AdaBoost with stumps on two moons, $n = 200$ (Colab Part 3). Red: exactly 0 at round 60.

A lone stump gets 15.5% of the training points wrong, and nothing with one straight cut can do better on interleaving moons. AdaBoost's weighted vote of stumps is at 1.0% by round 21 and exactly 0% by round 60.

A sequence of weak learners, each barely better than chance on the reweighted hard examples, combines into a perfect fit of the training set: bias reduction, made concrete.

Nothing here reduces variance: every stump sees the full dataset, only reweighted.

4. Gradient Boosting and XGBoost

Boosting for any loss, and the tabular-ML workhorse

Gradient boosting (Friedman, 2001): boosting for any loss

AdaBoost's reweighting is tied to classification with an exponential-style update. Friedman's generalization makes "correct the ensemble's mistakes" precise for *any* differentiable loss $\ell(\mathbf{y}, f)$, regression or classification:

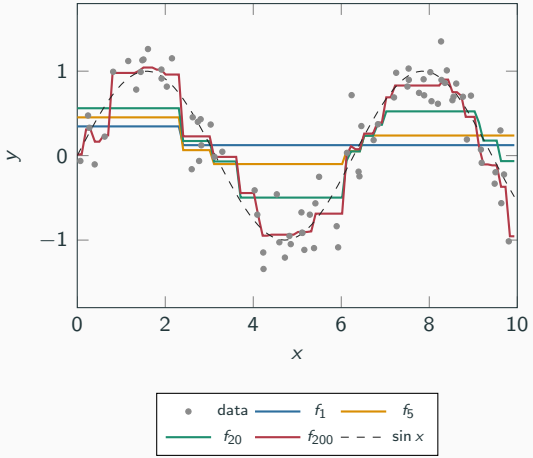
Start from a constant f_0 (the label mean). For $b = 1, \ldots, B$: compute, for every training point, how the loss wants the current prediction to move,

$$r_i^{(b)} := - \left. \frac{\partial \, \ell(\mathbf{y}_i, f)}{\partial f} \right|_{f = f_{b-1}(\mathbf{x}_i)},$$

fit the weak learner h_b to the targets $r_i^{(b)}$ (an ordinary regression), and update $f_b = f_{b-1} + \nu \, h_b$ with a small shrinkage $\nu \in (0, 1]$.

For squared loss the target is the **residual** $r_i^{(b)} = \mathbf{y}_i - f_{b-1}(\mathbf{x}_i)$: "fit the next tree to what is left over" is steepest descent on the training loss, one function-valued step per round. For logistic or absolute loss only the formula for $r_i^{(b)}$ changes.

Seen directly: stumps fit to residuals build a curve one step at a time



Setup. 80 points, $y = \sin x +$ noise; weak learner a regression stump (one split, two constants); squared loss, $\nu = 0.3$.

round	train MSE
1	0.50
5	0.37
20	0.17
200	0.04

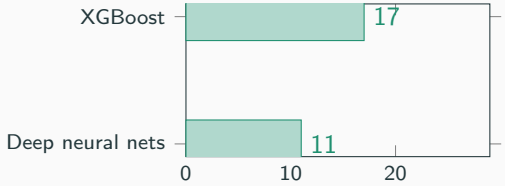
One stump is a step function; 200 of them, each fit to what the previous ones left over, trace the sine. By round 200 the fit is below the noise level (0.09), so it has begun to fit noise: B and ν are the regularization knobs, set by early stopping on validation data.

XGBoost (Chen and Guestrin, 2016): gradient-boosted trees, engineered

A regularized, second-order, scalable implementation of gradient boosting with trees:

- an explicit **regularizer** on tree complexity (leaf count, leaf values) in the objective;
- a **second-order** (Newton-style) loss approximation each round, scoring splits with the Hessian as well as the gradient;
- **systems** work (sparsity-aware splits, cache-aware layout, parallel tree building) for very large datasets.

> On tabular data, gradient-boosted trees, not neural networks, remain the dominant practical method.



2015 Kaggle-winning solutions (of 29)

The paper's own count: of the 29 winning solutions published on Kaggle's blog in 2015, 17 used XGBoost (8 alone, the rest in combination, often with a neural network); the next most used method, deep neural networks, appeared in 11. Every top-10 team in KDD Cup 2015 used it.

5. Ensembling Neural Networks

Three genuinely different answers

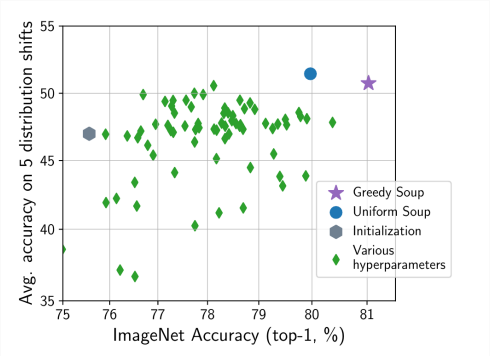
Deep ensembles (Lakshminarayanan et al., 2017): bagging, with free diversity

Bagging and boosting were stated for any base learner; Lecture 07 made a neural network one. The most direct instance trains several networks *independently* and combines their predictions, exactly bagging's recipe applied to networks.

> **No bootstrap needed.** Training a network is already non-convex and stochastic (Lecture 07's saddles, mini-batches, random initialization), so retraining the *same* architecture on the *same* full dataset from a different seed already gives a meaningfully different network: diversity, and hence a low ρ, for free.

What it is used for. Accuracy improves as the correlation floor predicts, but the paper's main point is **predictive uncertainty**: how much the members disagree on an input is a useful, well-calibrated signal of how much to trust the prediction, competitive with far more expensive approximate Bayesian networks. The cost is B trainings and B forward passes per prediction, which is what the next idea removes.

Model soups (Wortsman et al., 2022): average the weights, not the predictions



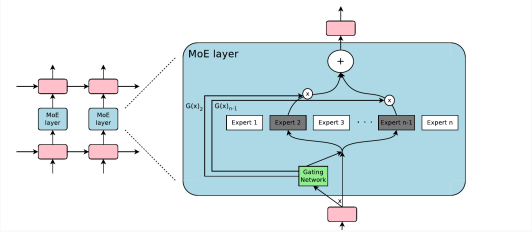
Wortsman et al. (2022), Figure 1. Green diamonds: individual fine-tuned models. Blue: the uniform soup, all of them averaged. Star: the greedy soup, models added one at a time only if they help.

Fine-tune B copies of the *same* pretrained network on the same task with different hyperparameters, then average the **weights**, $\bar{\theta} = \frac{1}{B} \sum_b \theta_b$.

$f_{\bar{\theta}}$ is one network: an ensemble-like gain at the cost of a **single** forward pass. Best soup: 90.94% ImageNet top-1 (ViT-G), a new state of the art at publication.

Why, and when. Models sharing a pretraining lineage sit in one basin of the loss landscape, and the line between their weights stays low-loss; two networks from different seeds do not, and averaging them is usually worse than either.

Mixture of experts (Shazeer et al., 2017): a learned, sparse combination



Shazeer et al. (2017), Figure 1: an MoE layer inside a stacked recurrent model. The gating network picks 2 of n experts (shaded) for this input; only those are computed.

Every combination rule so far is fixed after training. MoE replaces a feedforward layer by a bank of expert sub-networks plus a small *learned* gate that reads the input and picks a few experts, the only ones evaluated.

	bagging/boosting	**MoE**
rule	fixed	learned
per input	every member	a few, gated

Not bagging relabeled: one large model whose capacity far exceeds its per-example compute; over 1000× a dense layer's capacity at a minor efficiency loss (language modeling, translation).

Part II

6. Self-Supervised Learning

Inventing labels from unlabeled data

Every model so far needed a label; a pretext task manufactures one from x alone

labeled data: a human grades an image, transcribes audio, judges text. Expensive, so small.

unlabeled data: the open internet's text, images, audio. Enormous, almost free.

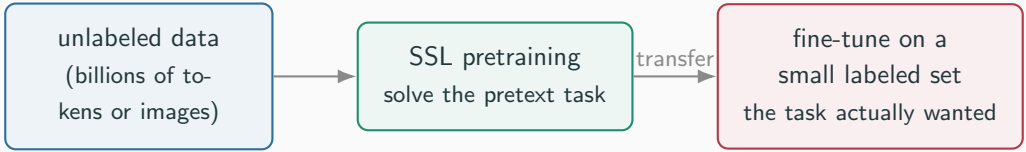
Lecture 06 found structure in $\mathbf{x}$ alone (clusters, principal directions), and Lecture 07 learned the feature map $\phi(\mathbf{x})$ itself, but every network in Lecture 07 was still fit on labeled pairs $(\mathbf{x}_n, \mathbf{y}_n)$. The question for Part II: can a model learn a useful $\phi(\mathbf{x})$ with *no* labels at all?

Self-supervised learning (SSL). Given only unlabeled $\mathbf{x}$, construct a **pretext task** $g : \mathbf{x} \mapsto (\mathbf{x}_{\text{in}}, \mathbf{x}_{\text{out}})$, both carved out of $\mathbf{x}$ itself. Fit a model to solve it; the feature map learned along the way is the payoff, reused for the downstream task actually wanted.

Self-supervised learning vs. transfer learning: not competitors, they compose

transfer learning: still needs *some* labels. Train on a labeled *base* task, reuse the learned feature map on a related target task.

self-supervised learning: needs *no* labels for the base task at all. The pretext task invents its own.



The standard modern pipeline is exactly this composition: a model pretrained on a pretext task is then fine-tuned, with a transfer-learning step on top, on a small labeled set. BERT (one pretraining run, eleven downstream tasks) and SimCLR (a linear classifier on frozen features) both follow it, and Section 10 reports what each bought.

7. Language Modelling

The leading instance of SSL

Next-token prediction is a pretext task; Markov models estimate it by counting

Language modelling: text is a token sequence $x_1, \ldots, x_T$ (roughly, words or word-pieces); the pretext task is $x_1, \ldots, x_t \mapsto x_{t+1}$. The label is the next word already in the raw text, so the model is fit by maximizing $\mathbb{P}(x_{t+1} \mid x_1, \ldots, x_t)$ at every position of every document.

The classical baseline. A **Markov model** of order k looks only at the last k tokens and estimates by counting:

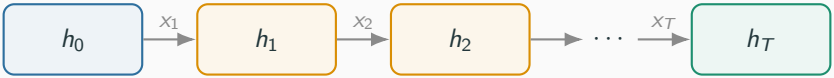
$$\mathbb{P}(x_{t+1} \mid x_1, \ldots, x_t) \approx \mathbb{P}(x_{t+1} \mid x_{t-k+1:t}) \approx \frac{\text{count}(x_{t-k+1:t+1})}{\text{count}(x_{t-k+1:t})}.$$

Data sparsity: most k-token contexts are rare or never seen, and raising k makes this exponentially worse. **No generalization**: "the cat sat" and "the dog sat" are unrelated count-table entries. One cause for both: the context is a raw, discrete lookup key, not a learned continuous representation in which similar contexts behave similarly.

RNNs replace the count table with a learned recurrence, and pay for it twice

A **recurrent neural network (RNN)** keeps a learned **hidden state** h_t, updated one token at a time, meant to summarize all of $x_1, \ldots, x_t$ with no fixed order k:

$$h_t = g\big(W_{hh} h_{t-1} + W_{xh} x_t + b_h\big), \qquad h_0 := \mathbf{0}.$$



Problem 1, the sequential bottleneck: h_t needs h_{t-1}, so a length-T sequence costs T strictly sequential steps however much parallel hardware is available.

Problem 2, vanishing gradients: backpropagation through time chains $t-1$ Jacobians $\partial h_{s+1}/\partial h_s$, Lecture 07's argument with sequence length as depth. LSTM and GRU gates patch this one and leave Problem 1 untouched; attention removes the recurrence, and so both problems, at once.

8. Attention

What replaces the recurrence

Attention: every token looks at every other token at once, through learned query, key, and value projections

Let $X \in \mathbb{R}^{T \times D}$ hold the sequence, row t being token t's D-dimensional vector. Three learned projections $W_Q, W_K, W_V \in \mathbb{R}^{D \times D_K}$ give every token three roles:

query $Q = XW_Q$: what this token is *looking for*. **key** $K = XW_K$: what this token *offers*, for queries to match against. **value** $V = XW_V$: what this token *contributes*, once matched.

Scaled dot-product attention: $P = \mathrm{softmax}\big(QK^\top/\sqrt{D_K}\big) \in \mathbb{R}^{T \times T}$ with the softmax taken **row-wise**, so row t of P is a distribution over all T tokens; output $= PV \in \mathbb{R}^{T \times D_K}$.

Row t of the output, $\sum_s P_{ts} v_s$, averages every value with weights from token t's query against every key; all T rows are computed in parallel.

Worked example: two tokens, computed by hand (and in the Colab notebook)

$T = 2$, $D_K = 4$, value dimension 2; the projections have already been applied:

$q_1 = (1, 2, 0, 1),\ q_2 = (0, 1, 2, 1), \quad k_1 = (1, 1, 0, 1),\ k_2 = (2, 0, 1, 0), \quad v_1 = (1, 0),\ v_2 = (0, 2).$

Token 1's row.

$q_1^\top k_1 = 1{+}2{+}0{+}1 = 4, \quad q_1^\top k_2 = 2$

scale by $1/\sqrt{4} = \frac{1}{2}$: $(2,\ 1)$

$p_{1,1} = \frac{e^2}{e^2 + e^1} \approx 0.731,\ p_{1,2} \approx 0.269$

Token 2's row.

$q_2^\top k_1 = 2, \quad q_2^\top k_2 = 2$

scale: $(1,\ 1)$, an exact tie

$p_{2,1} = p_{2,2} = 0.5$

Weighted sums of the values: $\text{out}_1 = 0.731\,(1, 0) + 0.269\,(0, 2) \approx (0.731,\ 0.538)$ and $\text{out}_2 = 0.5\,(1, 0) + 0.5\,(0, 2) = (0.5,\ 1.0)$. Token 1 places 73% of its attention on itself and 27% on token 2; token 2 splits evenly. The full matrices are $P = \left(\begin{smallmatrix} 0.731 & 0.269 \\ 0.5 & 0.5 \end{smallmatrix}\right)$ and $PV = \left(\begin{smallmatrix} 0.731 & 0.538 \\ 0.5 & 1.0 \end{smallmatrix}\right)$, matched by the notebook's from-scratch implementation to full floating-point precision.

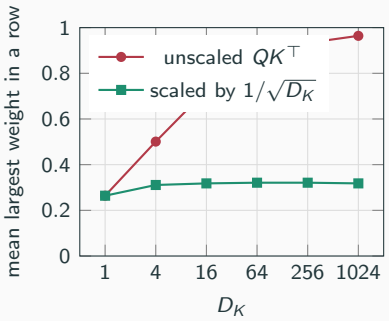
Why divide by $\sqrt{D_K}$: unscaled dot products grow with the dimension and saturate the softmax

If the entries of q and k are independent with mean 0 and variance 1, then

$$q^\top k = \sum_{i=1}^{D_K} q_i k_i \quad \text{has mean 0 and variance } D_K,$$

so raw scores have typical size $\sqrt{D_K}$ (8 at Vaswani et al.'s $D_K = 64$). Scores far apart in magnitude push the softmax to a near one-hot output whose gradient is nearly zero: Lecture 07's saturating-activation problem, now in the attention weights.

> *"We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients."* (Vaswani et al., 2017, §3.2.1)



$T = 10$ random $\mathcal{N}(0, 1)$ keys, 2000 draws per D_K. Unscaled, the largest weight climbs from 0.26 to 0.96 (one token takes almost everything); scaled, it stays near 0.32 at every D_K.

Causal masking: a language model must not see the future

Language modelling predicts x_{t+1} from $x_1, \ldots, x_t$ only, so token t must never attend to tokens $t+1, \ldots, T$. Add a mask *before* the softmax:

$$P = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{D_K}} + M\right), \qquad M_{ts} = \begin{cases} 0 & s \le t \\ -\infty & s > t \end{cases}, \qquad M = \begin{pmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{pmatrix} \ (T = 3).$$

Since $e^{-\infty} = 0$, every masked entry gets exactly zero weight, and the unmasked entries keep their relative weighting.

On the running example. Mask token 1's view of token 2 ($M_{1,2} = -\infty$): its scaled score row goes from $(2, 1)$ to $(2, -\infty)$, the softmax of which is exactly $(1, 0)$, so $\text{out}_1 = v_1 = (1, 0)$. Token 2's row has $s \le t$ for both entries and stays at $(0.5, 0.5)$.

This one $T \times T$ matrix is the only difference between a bidirectional encoder and a causal decoder (Section 9): the same block, with and without M.

Multi-head attention: H attention patterns learned in parallel

One (W_Q, W_K, W_V) triple gives one attention *pattern* per layer, one way for tokens to relate. Run H independent copies (**heads**), concatenate their outputs, and mix with one more learned matrix W_O:

$$\text{head}_h = \text{Attention}(XW_{Q,h},\, XW_{K,h},\, XW_{V,h}), \qquad \text{MultiHead}(X) = [\text{head}_1, \ldots, \text{head}_H]\, W_O.$$

The numbers (Vaswani et al., 2017): model dimension $D = 512$, $H = 8$ heads, per-head $D_K = D/H = 64$. Splitting the budget across heads keeps the total cost comparable to one full-width head.

The analogy: a conv layer's multiple output channels (Discussion 07). Several *different* patterns learned in parallel, not one averaged pattern; one head can track syntax while another tracks coreference.

Attention is permutation-equivariant: it sees a set of tokens, not a sequence

Claim. With no positional information, permuting the input rows permutes the output rows identically: for a $T \times T$ permutation matrix R (so $R^\top R = I$), $\mathrm{Attention}(RX) = R\,\mathrm{Attention}(X)$.

Proof. $Q = XW_Q$ is linear in X, so $Q' = RQ$, and likewise $K' = RK$, $V' = RV$; rows and columns of the score matrix are then permuted by the same R, and a row-wise softmax does not care how the entries within a row are indexed:

$$Q'K'^\top = R\,(QK^\top)\,R^\top \;\Rightarrow\; P' = RPR^\top \;\Rightarrow\; P'V' = RPR^\top RV = RPV. \qquad \blacksquare$$

Why this is a problem. "she prefers cats to dogs" and "she prefers dogs to cats" contain the exact same set of tokens and mean opposite things. Attention alone is blind to word order.

The notebook confirms this on a random 6-token sequence (discrepancy 2.8×10^{-17}).

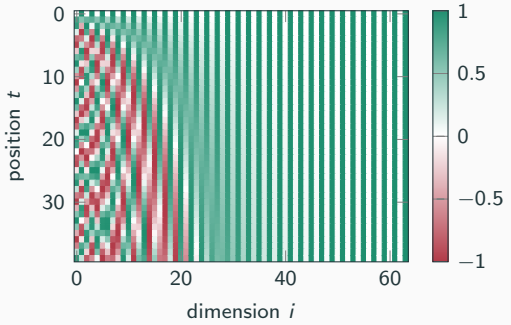
Positional encoding puts the order back: add a position-specific vector to each token

Before the first attention layer, replace each token's embedding by $\tilde{\mathbf{x}}_t = \mathbf{x}_t + PE_t$, a vector that depends on the *position* t, not the content. Reordering the words is then no longer a row permutation, and the proof no longer applies. Vaswani et al. use sinusoids with wavelengths from 2π to $10000 \cdot 2\pi$:

$$PE_{(t,2i)} = \sin\big(t/10000^{2i/D}\big),$$

$$PE_{(t,2i+1)} = \cos\big(t/10000^{2i/D}\big).$$

A learned table does about as well; sinusoids were chosen because PE_{t+k} is a linear function of PE_t for fixed k, and the formula extends to unseen lengths.



The 40×64 encoding for $D = 64$. Low dimensions oscillate fast along t, high dimensions barely move: each position gets a unique pattern, and nearby positions get similar ones.

9. Transformers

Assembling attention into a reusable block

The Transformer block: attention between tokens, an MLP within each token, residual and layer norm around both

Given layer l's token representations $X^{(l)} \in \mathbb{R}^{T \times D}$, one block computes

$$X^{(l)}_{\text{attn}} = \text{LayerNorm}\Big(X^{(l)} + \text{MultiHead}(X^{(l)})\Big), \qquad X^{(l+1)} = \text{LayerNorm}\Big(X^{(l)}_{\text{attn}} + \text{MLP}(X^{(l)}_{\text{attn}})\Big),$$

with the MLP applied identically and independently to every row t:

$$\text{MLP}(\mathbf{x}_t) = W_2^\top \,\text{ReLU}(W_1^\top \mathbf{x}_t + b_1) + b_2.$$

Two new pieces. Multi-head attention mixes information *between* tokens: each token's new representation is a learned combination of every other's. The **per-token MLP** mixes *within* a token only.

Two pieces reused from Lecture 07 and Discussion 07. A **residual connection** around each sub-layer, the skip-connection argument verbatim; and **layer normalization** on every token's row. Output shape equals input shape, so blocks stack.

The architecture from the paper, and its numbers

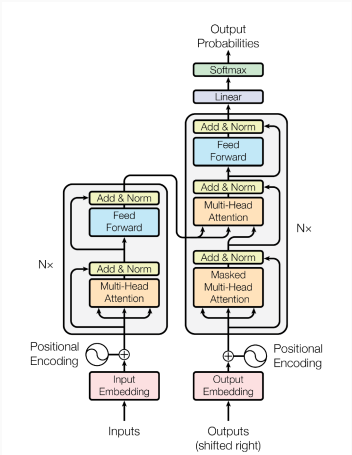


Figure 1: The Transformer - model architecture.

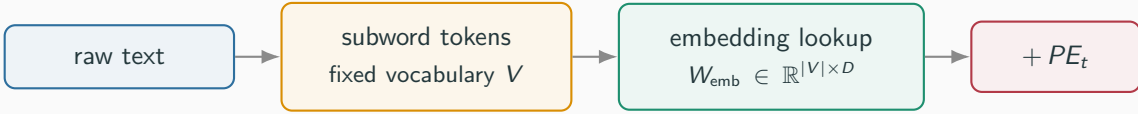
Vaswani et al. (2017), Figure 1.

Left stack, the encoder: $L = 6$ blocks of exactly the previous slide, bidirectional self-attention (no mask), so every token sees the whole input.

Right stack, the decoder: $L = 6$ blocks with *masked* self-attention (Section 8's M), plus a third sub-layer whose queries come from the decoder and whose keys and values come from the encoder's output (**cross-attention**).

The base model: $D = 512$, $H = 8$ heads of $D_K = 64$, MLP inner width 2048, $6 + 6$ blocks, 65M parameters, trained for translation. Every box in the figure is a piece already defined.

Text in: subword tokens and an embedding table; three ways to wire the same block



Common words become one token; rare words split into smaller, still meaningful pieces, so no input word is ever out of vocabulary. Row x_t of the embedding table is token t's initial vector; the table is learned with the rest.

Variant	Attention	Use case
Encoder-only	bidirectional	classification, tagging (BERT)
Decoder-only	causal, with mask M	generation one token at a time (GPT)
Encoder-decoder	both, plus cross-attention	translation-style input-to-output tasks

All three use the same block; only which tokens each position may attend to changes.

Why this replaced the RNN: parallel and one step between any two tokens, at a quadratic cost in T

	Cost per layer	Sequential operations	Max path length
Self-attention	$O(T^2 D)$	$O(1)$	$O(1)$
Recurrent	$O(TD^2)$	$O(T)$	$O(T)$

Vaswani et al. (2017), Table 1; T sequence length, D representation dimension.

Sequential operations is Section 7's Problem 1: an RNN needs T dependent steps per layer, attention needs one, so training parallelizes across the whole sequence. **Max path length** is Section 7's Problem 2: a gradient from token T reaches token 1 through T Jacobians in an RNN and through one attention matrix here, so no long multiplicative chain can vanish.

The trade: cost per layer is quadratic in T instead of linear; cheap at T in the thousands, binding in the hundreds of thousands. Both RNN problems go away by removing the recurrence, not patching it.

10. Two Worked SSL Instances

BERT (language) and SimCLR (vision)

BERT (Devlin et al., 2019): an encoder-only Transformer pretrained on masked language modelling

A complete instance of Section 6's recipe: an encoder-only Transformer, pretrained on two pretext tasks built from unlabeled text, then fine-tuned.

Task 1, masked language modelling (MLM): pick 15% of the input tokens at random and predict the *original* token at each picked position from the final hidden vector there (cross-entropy). No causal mask, so the prediction uses context from both directions: the "bidirectional" in the name.

Task 2, next sentence prediction (NSP): pack two sentences; 50% of the time the second really follows the first (`IsNext`), 50% it is random (`NotNext`). Predicted from the `[CLS]` token's final vector.

The 80/10/10 **split**: of the picked tokens, 80% become `[MASK]`, 10% a random token, 10% stay unchanged, since `[MASK]` never appears at fine-tuning time.

BERT: the input format, two sizes, and what one pretraining run bought

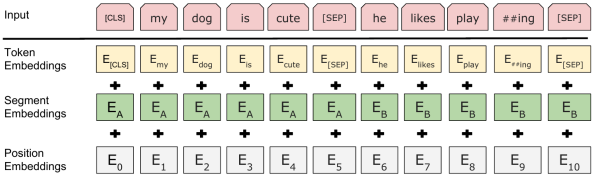


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

Devlin et al. (2019), Figure 2. `[CLS]` first (whole-sequence representation), `[SEP]` between the two packed sentences; every input vector is the sum of a token, a segment, and a position embedding.

	Layers	D	Heads / params
$\text{BERT}_{\text{BASE}}$	12	768	12 / 110M
$\text{BERT}_{\text{LARGE}}$	24	1024	16 / 340M

Fine-tuned with one task-specific output layer: new state of the art on **eleven** NLP tasks (GLUE 80.5, up 7.7 points; SQuAD v1.1 F1 93.2), all from *one* self-supervised pretraining run.

SimCLR (Chen et al., 2020): contrastive learning, the same recipe on images with no text and no masking

Pretext task: two augmented views of one image should get agreeing representations.

Trivial collapse: a constant encoder $f(x) \equiv c$ makes every pair agree and learns nothing.

The fix, negatives: also push *different* images apart. In a minibatch of N images ($2N$ views), every view but a view's own partner is a negative, free of charge.

Crop, color distortion, and blur give the **positive pair** $\tilde{x}_i, \tilde{x}_j$; encoder f (a ResNet), projection head g (a small MLP, discarded after training), loss on $z = g(f(\tilde{x}))$.

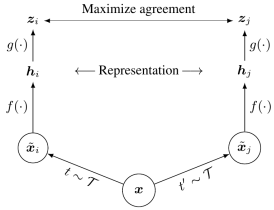


Figure 2. A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

Chen et al. (2020), Figure 2.

The NT-Xent loss, and what a linear classifier on the frozen features reaches

For a positive pair (i, j) among the $2N$ views, with cosine similarity $\mathrm{sim}(u, v) = u^\top v/(\|u\|\,\|v\|)$ and temperature τ:

$$\ell_{i,j} = -\log \frac{\exp\big(\mathrm{sim}(z_i, z_j)/\tau\big)}{\sum_{k=1}^{2N} \mathbb{1}[k \neq i]\, \exp\big(\mathrm{sim}(z_i, z_k)/\tau\big)}.$$

A softmax cross-entropy over the $2N - 1$ candidates: the positive partner must win against every negative in the batch. "These two views come from the same image" is a label invented from an unlabeled image: Section 6's template once more, and BERT and SimCLR are one recipe twice.

76.5%
top-1, ImageNet

a **linear classifier** on SimCLR's frozen, self-supervised ResNet-50 features, matching a fully supervised ResNet-50 trained with labels throughout; fine-tuned on 1% of the labels, SimCLR beats AlexNet with 100× fewer labels.

11. Wrap-up

Two ideas, and where they meet

The whole arc, Part I: one statistical idea, four instances

1. **Wisdom of crowds**: averaging B independent estimators divides the variance by B; correlation caps the gain at the floor $\rho\sigma^2$, so every method's real job is to make its members disagree.
2. **Bagging**: bootstrap resamples, parallel fits, an average; reduces variance, so it helps high-variance learners such as deep trees (21.3% $\rightarrow$ 18.8%, variance down 2.4$\times$).
3. **Boosting**: sequential fits to the current mistakes; reduces bias. AdaBoost's reweighting takes stumps from 15.5% to 0% training error.
4. **Gradient boosting**: fit the next learner to the loss's negative gradient (the residual for squared loss); XGBoost won 17 of 29 Kaggle 2015 solutions.
5. **Neural networks**: deep ensembles (bagging, diversity from seeds), model soups (average weights, one forward pass), mixture of experts (a learned, sparse combination).

The whole arc, Part II: one recipe for learning without labels, and the machinery it needed

6. **Self-supervised learning**: invent a pretext task from unlabeled data; the learned feature map is the payoff. Composes with transfer learning as pretrain-then-fine-tune.
7. **Language modelling**: next-token prediction is a pretext task. Markov models fail on sparsity; RNNs fix that and pay with a sequential bottleneck and vanishing gradients.
8. **Attention**: a learned query/key/value match, scaled by $1/\sqrt{D_K}$ (largest weight $0.96 \to 0.32$ at $D_K = 1024$), masked causally, H heads, blind to order until positional encoding is added.
9. **Transformer**: attention plus a per-token MLP, wrapped in Lecture 07's residual connections and layer norm; $O(1)$ sequential steps and path length at $O(T^2 D)$ cost per layer; encoder-only, decoder-only, or both.
10. **BERT and SimCLR**: one recipe twice; eleven NLP tasks from one masked-LM run, 76.5% ImageNet top-1 from a linear probe on contrastive features.

Coming up

Discussion 08: Project check-in. Progress against plan, blockers, and a refined scope, three weeks before the report review.

HW3 is due Monday 11/2.

Next lecture: what happens as language models like Part II's get *large*: scaling laws and compute-optimal training, in-context learning, and the debate over whether emergent abilities are real discontinuities or measurement artifacts.

Two Colab notebooks, NumPy only. `bagging_boosting_demo.ipynb` is the source of every "measured" Part I slide. `attention_transformer_demo.ipynb` implements Part II: attention matching the worked example to full precision, causal masking, two heads, a full Transformer block, and the permutation-equivariance test with and without positional encoding.

References, Part I

- Breiman. *Bagging Predictors.* Machine Learning, 24(2), 1996, pp. 123–140.
- Breiman. *Random Forests.* Machine Learning, 45(1), 2001, pp. 5–32.
- Freund, Schapire. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting.* JCSS, 55(1), 1997, pp. 119–139.
- Friedman. *Greedy Function Approximation: A Gradient Boosting Machine.* The Annals of Statistics, 29(5), 2001, pp. 1189–1232.
- Chen, Guestrin. *XGBoost: A Scalable Tree Boosting System.* KDD 2016, pp. 785–794.
- Lakshminarayanan, Pritzel, Blundell. *Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles.* NeurIPS 2017; arXiv:1612.01474.
- Wortsman et al. *Model Soups: Averaging Weights of Multiple Fine-Tuned Models Improves Accuracy Without Increasing Inference Time.* ICML 2022; arXiv:2203.05482.
- Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.* ICLR 2017; arXiv:1701.06538.

Part I is fully original; every citation verified against the primary source (`references/sources/lecture08-*.md`). "Measured" slides reproduce the Colab notebook.

References, Part II

- Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin. *Attention Is All You Need.* NeurIPS 2017, pp. 5998–6008; arXiv:1706.03762.
- Devlin, Chang, Lee, Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* NAACL-HLT 2019, pp. 4171–4186; arXiv:1810.04805.
- Chen, Kornblith, Norouzi, Hinton. *A Simple Framework for Contrastive Learning of Visual Representations.* ICML 2020, PMLR 119, pp. 1597–1607; arXiv:2002.05709.
- West (slide credit; Flammarion, Jaggi, Kosson, Yüksel). *Transformers; Text Representation Learning; Self-Supervised Learning.* EPFL CS-433 Machine Learning, lectures 09a, 12a, 13a, Fall 2025.

The attention derivation, the permutation-equivariance argument, the Transformer-block assembly, and the SSL framing are adapted from EPFL's `ML_course`, used with permission. The $1/\sqrt{D_K}$ motivation is quoted from Vaswani et al.; the saturation and positional-encoding figures are computed for this deck (`figdata-ssl.tex`). Markov models and the RNN-motivates-attention argument are standard material. Full citation details: `references/sources/epfl-l09-self-supervised-transformers.md`, `lecture09-10-transformers-scaling-citations.md`, `lecture09-additional-verified-facts.md`.