Foundations of Deep Learning & Introduction to Transformers

Deep Learning Part I,
Module 4 of 15.S60 IAP Software Tools 2026

Catherine Ning, cat2510@mit.edu

January 2026

Plan for this session

Part I: Foundations (50 min)

- Expressivity: MLPs, depth, inductive bias
- Optimization: loss, backprop, SGD
- Generalization
- Convolutional neural network (CNN)
- Encoder–decoder (bridge to transformers)

(5 min break)

Part II: Transformers (50 min)

- Understanding next-token prediction
- Transformer architecture: overview + self-attention

Useful Resources

- Foundations of Computer Vision: visionbook.mit.edu
- Understanding Deep Learning: udlbook.github.io/udlbook
- Personal recommendation for visuals: 3blue1brown.com/lessons/attention

The goal of learning

The goal of learning is to extract lessons from past experience in order to solve future problems.

Concretely: given data, we choose a model (a *useful* representation) that scores well under an *objective*.

OR lens: learning is an optimization problem, but the decision variable is a *function* (or its parameters) mapping inputs to outputs $f_\theta : \mathcal{X} \to \mathcal{Y}$.

Three lines of work

A learning algorithm has three ingredients:

1. **Expressivity:** what mappings $f_\theta : \mathcal{X} \to \mathcal{Y}$ can we learn/approximate?
2. **Optimization:** how do we find a good f efficiently?
3. **Generalization:** why should the f we found work on new data?

Deep learning at a glance

Deep learning is about learning a parameterized function $f_\theta : \mathcal{X} \to \mathcal{Y}$.

Neural nets give a rich hypothesis space:

1. **Depth:** a deep net is a composition of functions
2. **Expressivity:** can approximate many functions (UAT is a starting point)
3. **Differentiability:** enables gradient-based optimization (backprop)
4. **Inductive bias:** architecture + training assumptions to bias toward certain function spaces

Multilayer Perceptrons (MLPs)

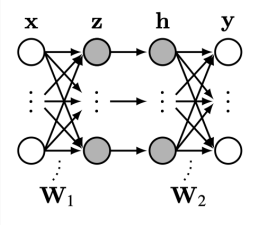
Perceptron $\rightarrow$ hidden layer

A perceptron is a linear model with a nonlinearity:

$$h(x) = \sigma(w^\top x + b)$$

Stacking them yields a **two-layer** neural network:

$$f(x) = W_2\, \sigma(W_1 x + b_1) + b_2$$



Why stack layers? (Depth)

A *deep* neural network computes a composition of L functions (layers):

$$f(x) = f_L \circ f_{L-1} \circ \cdots \circ f_1(x)$$

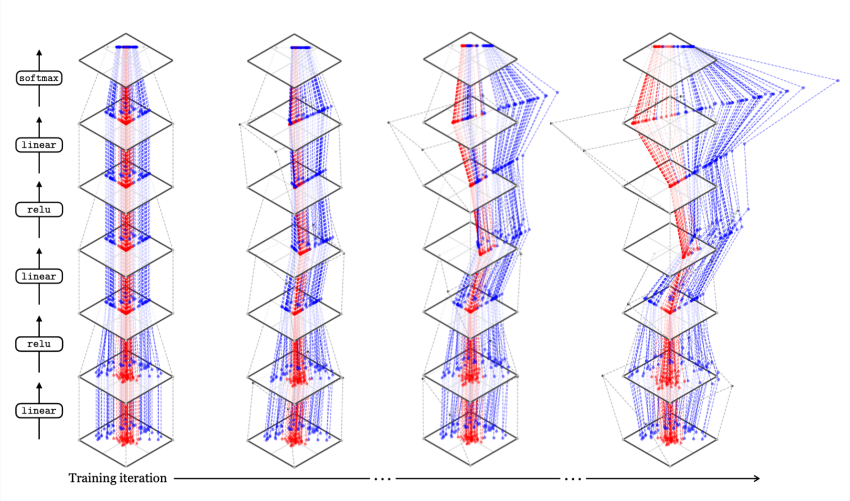
Each layer:

- mixes features linearly
- applies a fixed nonlinearity

Depth $\rightarrow$ Feature Transformations:

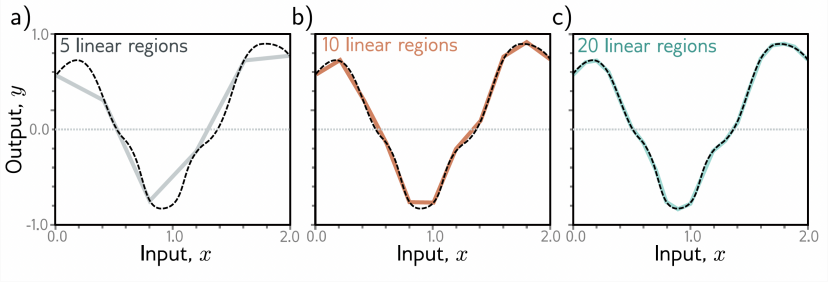
- Early layers extract simple features
- Later layers recombine them into more abstract ones
- Intermediate representations are reused across tasks

Feature Transformations with Depth + Iterations



Expressivity: universal approximation (informal)

UAT: A neural network, given enough neurons and layers, can approximate any continuous function to any desired level of accuracy.



Significance: UAT provides the theoretical bedrock for the power of neural networks and why they can model complex real-world relationships.

Why inductive bias is necessary

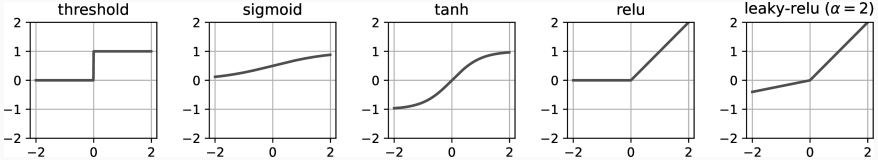
Finite data $\Rightarrow$ infinitely many functions fit the training set.

Learning requires choosing *which* function to prefer.

Inductive bias = architectural and algorithmic assumptions that tilt training toward some solutions, e.g.:

- architecture (MLP vs CNN vs Transformer)
- regularization (weight decay, dropout)
- data augmentation
- optimization dynamics (SGD, Adam)

Activation functions as inductive bias



Activations bias both:

- **function class** (piecewise linear vs smooth, saturation)
- **optimization** (vanishing gradients, conditioning)

From representation to learning

So far:

- neural nets define a hypothesis class $\{f_\theta\}$
- inductive bias says which f_θ are likely under training

Now: how do we choose θ from data? (optimization)

Optimization & Training

Motivation

We want to solve an empirical risk minimization problem:

$$\min_{\theta} \ \mathcal{L}(\theta) \ := \ \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(x_i), y_i).$$

- For deep nets, f_θ is a composition of nonlinear functions $\Rightarrow \mathcal{L}(\theta)$ is high-dimensional and nonconvex so no closed-form solution.
- Hence we use *local information* (gradient descent) to improve θ step-by-step.

One iteration = one local improvement:

$$\theta_{t+1} \ = \ \theta_t \ - \ \eta \, \nabla_\theta \mathcal{L}(\theta_t).$$

Gradient descent (GD)

GD updates parameters using the full dataset gradient:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \Big[\frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta) \Big]$$

Pros: stable; Cons: expensive when n is large.

In deep learning: n and model size are huge! SGD trades exactness for speed by using a random sample of the data.

Stochastic gradient descent (SGD)

SGD uses an unbiased gradient estimate from a minibatch $\mathcal{B}$ sampled uniformly:

$$g_t = \nabla_\theta \Big[\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \ell_i(\theta_t) \Big], \qquad \theta_{t+1} = \theta_t - \eta g_t$$

Properties:

- Each iteration is cheaper the smaller the batch size.
- Variance of g_t introduces noise in the trajectory.
- The noise is not a nuisance; it can help escape poor local stationary points.

Backprop: Computational graph perspective

Neural nets decompose computation into modular blocks.

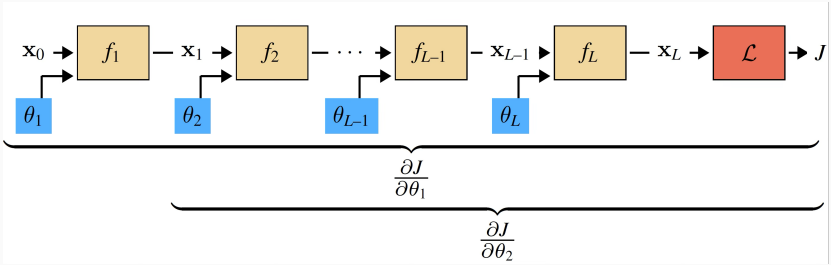


Figure 1: Nodes are operations; edges are tensors.

Backpropagation

To compute gradients, we apply the chain rule through the graph.

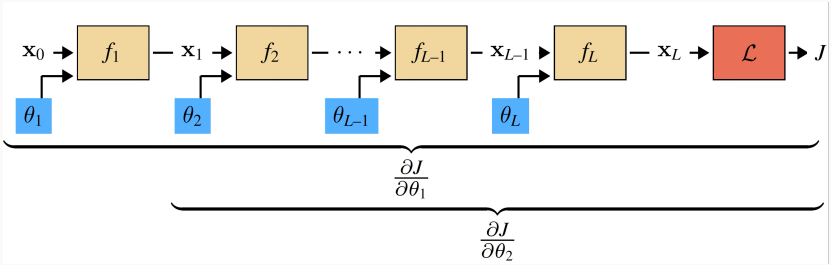


Figure 2: Backward pass reuses intermediate quantities from the forward pass.

Backprop propagates error signals backward, assigning "blame" to each parameter.

Why backprop scales

Shared subproducts are reused:

$$\frac{\partial J}{\partial \theta_1} = \boxed{\frac{\partial J}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\cdots\frac{\partial x_3}{\partial x_2}}\frac{\partial x_2}{\partial x_1}\frac{\partial x_1}{\partial \theta_1}$$

$$\frac{\partial J}{\partial \theta_2} = \boxed{\frac{\partial J}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\cdots\frac{\partial x_3}{\partial x_2}}\frac{\partial x_2}{\partial \theta_2}$$

Thus:

$$\text{Cost(backward)} \approx \text{Cost(forward)}.$$

Summary

A single training iteration (mini-batch SGD):

1. Sample a mini-batch $\mathcal{B}$.
2. **Forward pass:** compute predictions $\hat{y} = f_\theta(x)$ for $x \in \mathcal{B}$.
3. Compute batch loss $\mathcal{L}_\mathcal{B}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} \ell(f_\theta(x), y)$.
4. **Backward pass (backprop):** compute $\nabla_\theta \mathcal{L}_\mathcal{B}(\theta)$ via chain rule on the computation graph.
5. **Update:** $\theta_{t+1} \leftarrow \theta_t - \eta \, \nabla_\theta \mathcal{L}_\mathcal{B}(\theta)$.

Modern optimizers

Common tweaks to SGD:

- **Momentum:** smooth the gradient with an exponential moving average (often used for vision models)
- **Adam:** per-parameter adaptive step sizes using first/second moments (default baseline for transformers)
- **Weight decay:** penalize large norms (often improves generalization)

Generalization

Generalization in overparameterized models

Modern nets often operate where:

- the model can fit the training data exactly
- many distinct θ achieve near-zero training loss

Generalization depends on *which* interpolating solution is selected.

Empirically, gradient-based training often favors:

- solutions close to initialization
- lower complexity solutions
- flatter regions of the loss landscape ...

Diagnostics for generalization

We monitor:

- training vs validation loss
- robustness under distribution shift
- sensitivity to hyperparameters (LR, batch size, weight decay)
- effects of augmentation / early stopping

Reference: Zhang et al. (2017), “Understanding deep learning requires rethinking generalization”.

A key inductive bias: Convolution (CNNs)

Why CNNs for image processing?

In images, nearby pixels are correlated, and objects should be recognizable regardless of location.

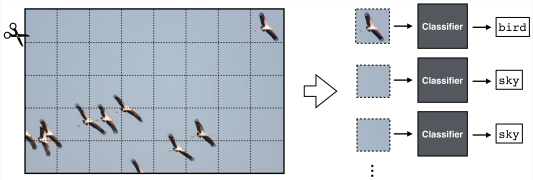


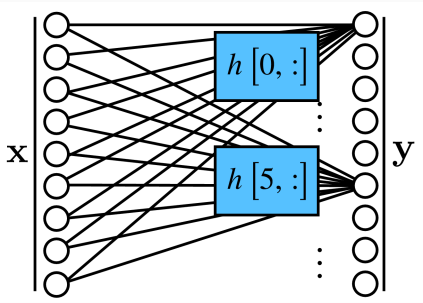
Figure 3: The key idea of CNNs is to chop up the input image into little patches, and then process each patch independently and identically.

Dense Mixing

Recall generic MLP layer:

$$h = \sigma(Wx + b)$$

Dense matrix W: every output mixes all input features.



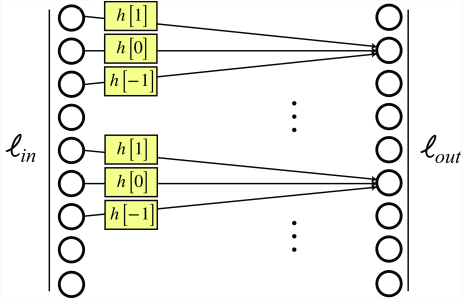
Convolutional neural net

CNN layer: replace dense mixing with constrained linear operation "convolution" ($*$):

$$h = \sigma(K * x + b)$$

Properties:

- local connectivity (small receptive fields)
- weight sharing (same kernel at every location)
- translation equivariance (shift input $\Rightarrow$ shift features)

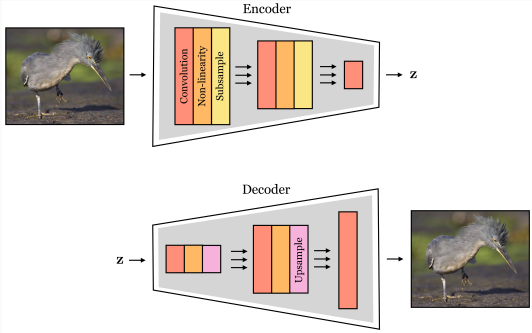


Encoder–Decoder architectures

Encoder–decoder as a design pattern

Core idea: split learning into two modules:

$$x \xrightarrow{\text{encoder}} z \xrightarrow{\text{decoder}} \hat{y}$$



The bottleneck in early seq2seq models

Encoder: maps input $\rightarrow$ intermediate representation z (features).

Decoder: maps representation $\rightarrow$ output (prediction or generation).

Bottleneck in NLP / text generation: z has limited information

- long inputs lose detail
- long-range dependencies are hard

Answer: Attention $\Rightarrow$ Transformers.

5-minute break

Stretch, refill coffee, feel free to come chat to me :).