CS839: AI for Scientific Computing
Advanced ML

Misha Khodak

University of Wisconsin-Madison

27 January 2026

Announcements

Enrollment:

- Finalized this week. Please keep checking your status.

Office hours:

- By appointment. Email me at khodak@wisc.edu.

Outline

- **Advanced neural architectures**
  - RNNs, Transformers, GNNs
- **Generative modeling**
  - density estimation, GANs, flow-based models, diffusion
- **Transfer learning**
  - pretraining, multi-task learning, foundation models

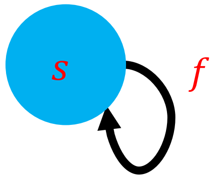
Outline

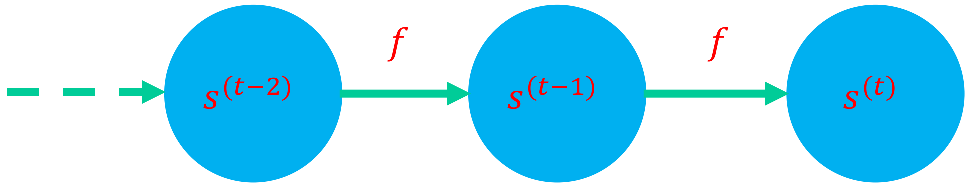
- **Advanced neural architectures**
  - RNNs, Transformers, GNNs
- **Generative modeling**
  - density estimation, GANs, flow-based models, diffusion
- **Transfer learning**
  - pretraining, multi-task learning, foundation models

Modeling Sequential Data

- Simplistic model:
  - $s^{(t)}$ state at time t. Transition function f

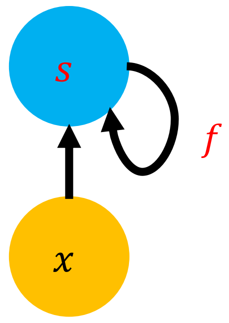
$$s^{(t+1)} = f(s^{(t)}; \theta)$$



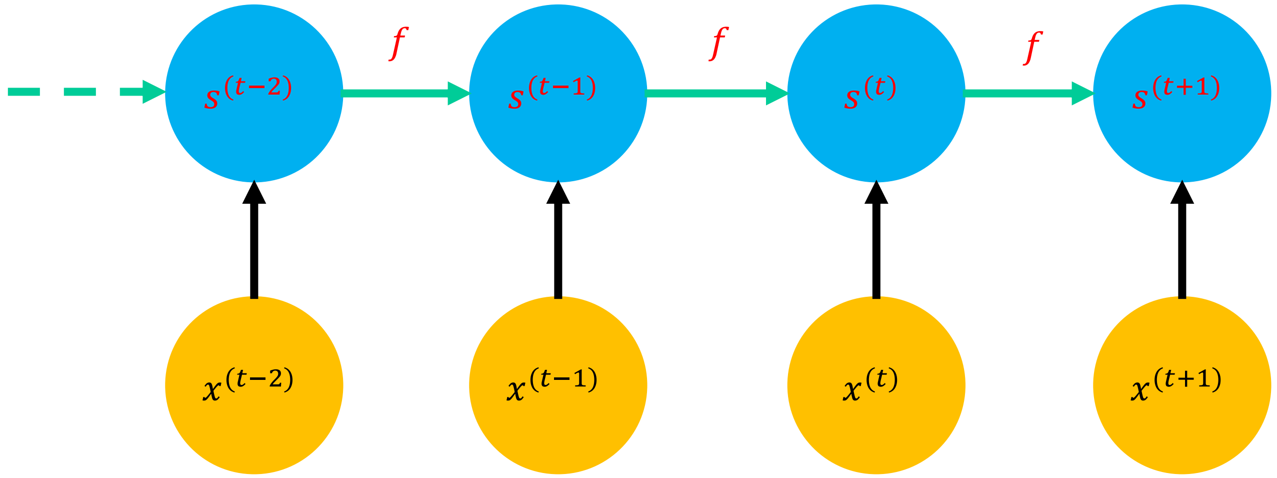


Modeling Sequential Data: External Input

- External inputs can also influence transitions
  - $s^{(t)}$ state at time t. Transition function f
  - $x^{(t)}$: input at time t

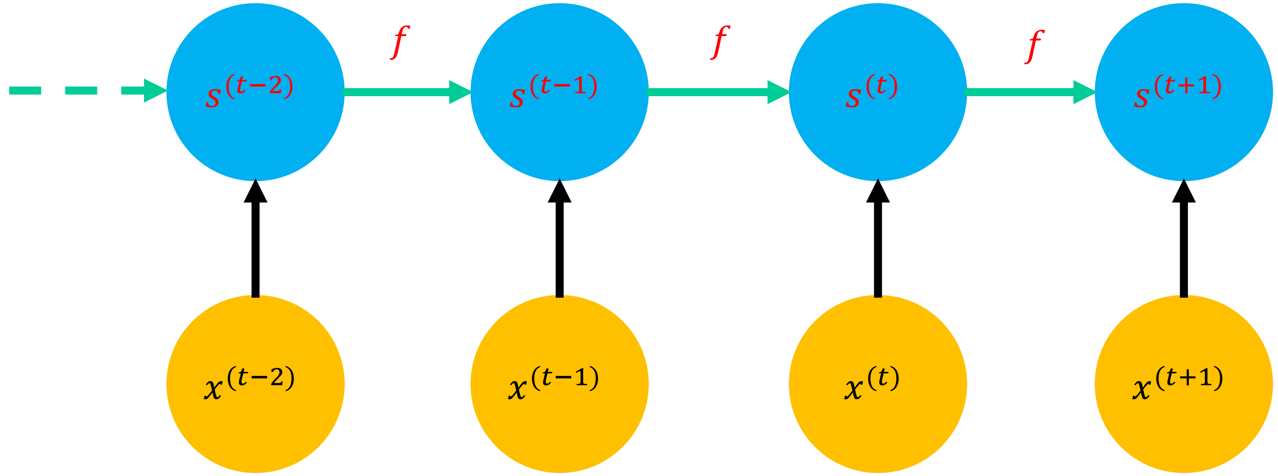


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$



Important: the same *f* and *θ* for all time steps

Recurrent Neural Networks

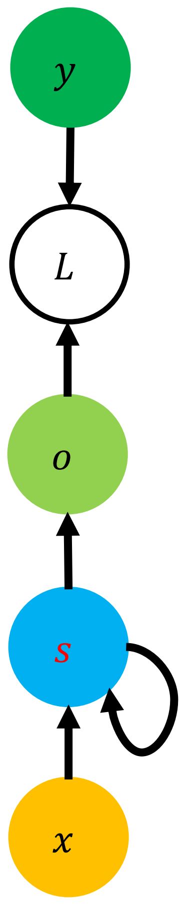


- Use the principle from the system above:
  - Same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the current hidden state and the **output** entry
- Training: loss typically computed at every time step

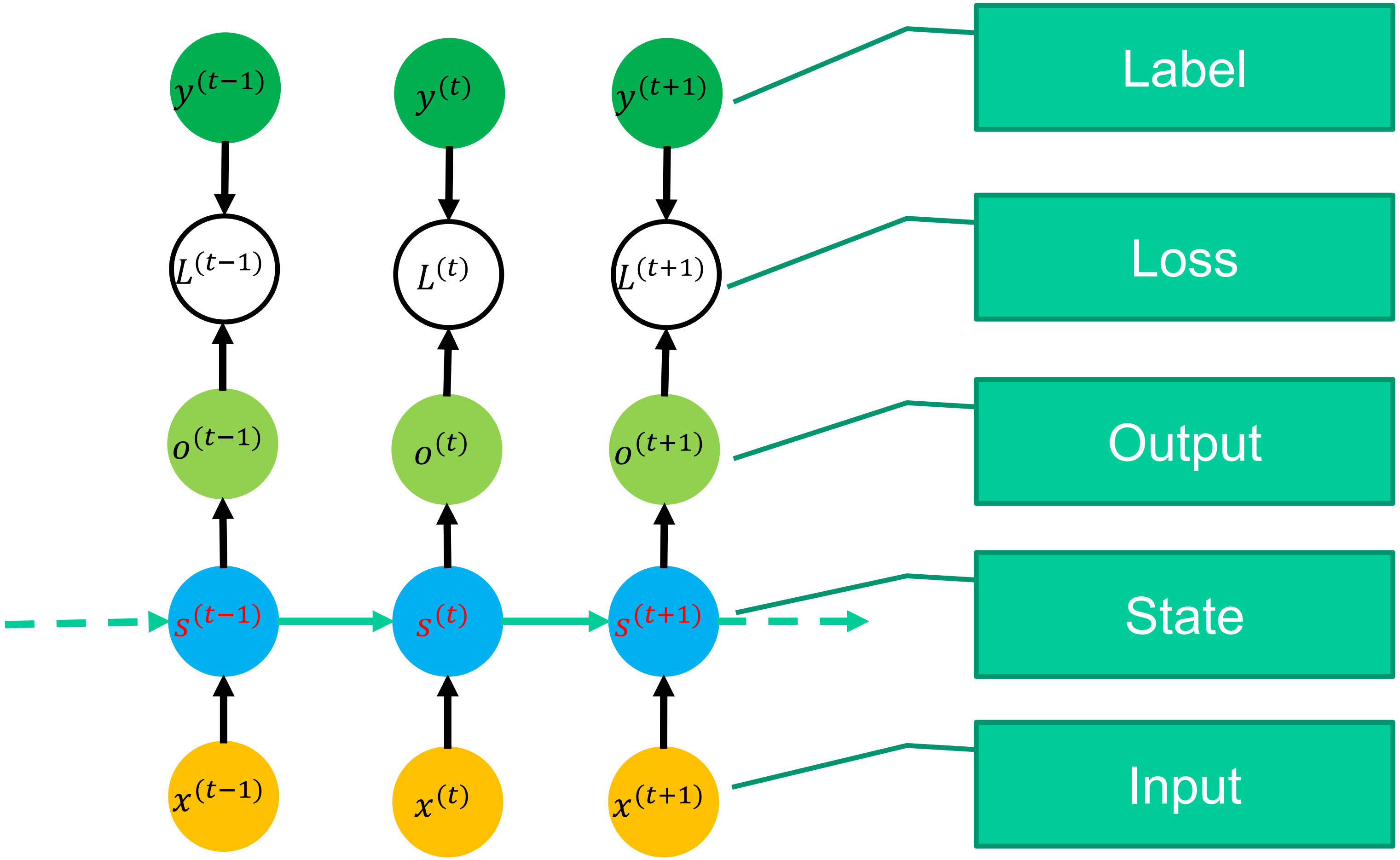
RNNs: Basic Components

- What do we need for our new network?
  - Input x
  - State s
  - Output o
  - Labels y & Loss function L
    - Still need to train!

Recurrent: state is plugged back into itself

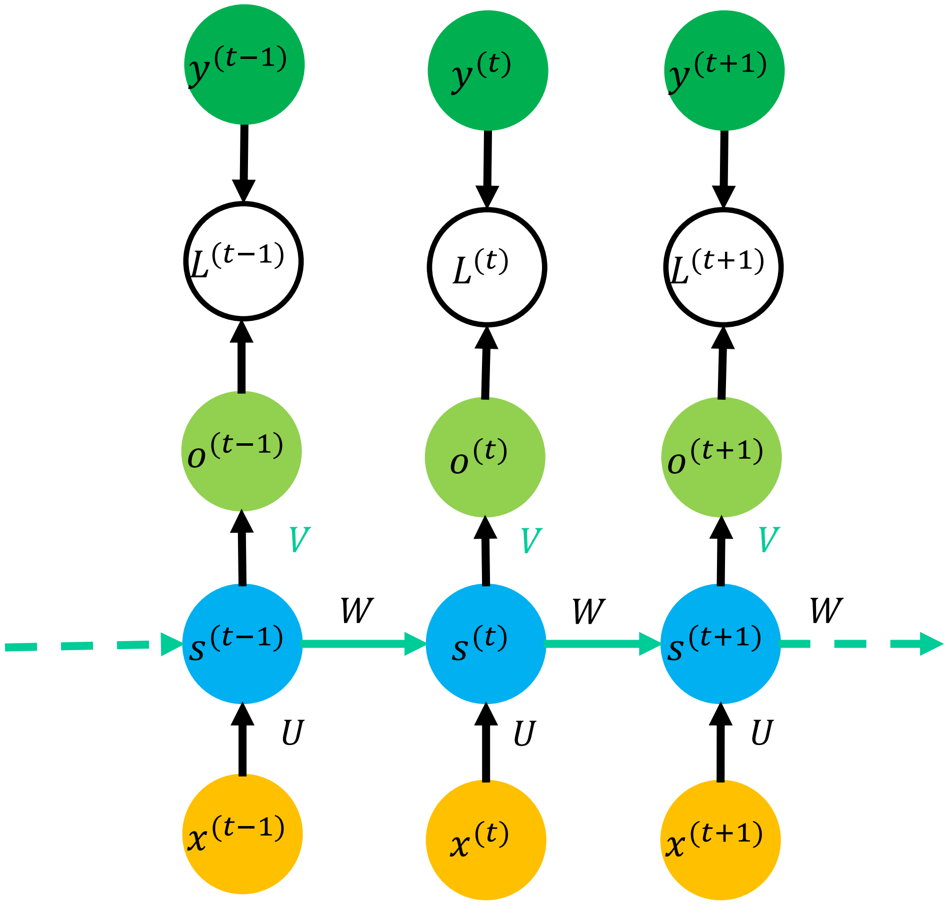


RNNs: Unrolled Graph



Simple RNNs

- Classical RNN variant:



$$
\begin{aligned}
a^{(t)} &= b + Ws^{(t-1)} + Ux^{(t)} \\
s^{(t)} &= \tanh\left(a^{(t)}\right) \\
o^{(t)} &= c + Vs^{(t)} \\
\hat{y}^{(t)} &= \mathrm{softmax}\left(o^{(t)}\right) \\
L^{(t)} &= \mathrm{CrossEntropy}(y^{(t)}, \hat{y}^{(t)})
\end{aligned}
$$

Properties

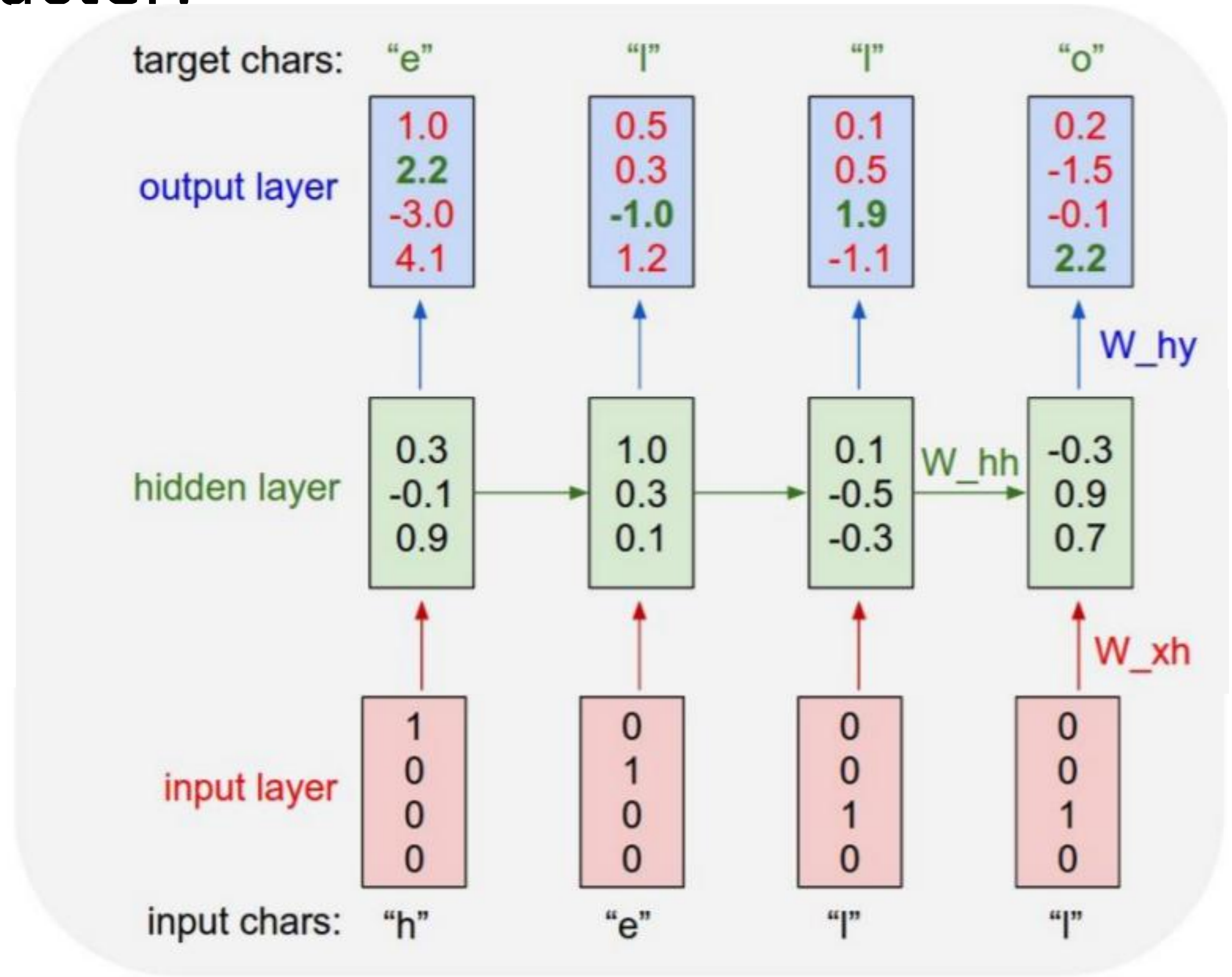
- **Hidden state**: a lossy summary of the past
- Shared functions / parameters
  - Reduce the capacity and good for **generalization**
- Uses the knowledge that sequential data can be processed in the same way at different time step
- Powerful (**universal**): any function computable by a Turing machine computed by such a RNN of a finite size
  - Siegelmann and Sontag (1995)

Example: Char. Level Language Model

- LM goal: predict next character:

- Vocabulary
  {h,e,l,o}

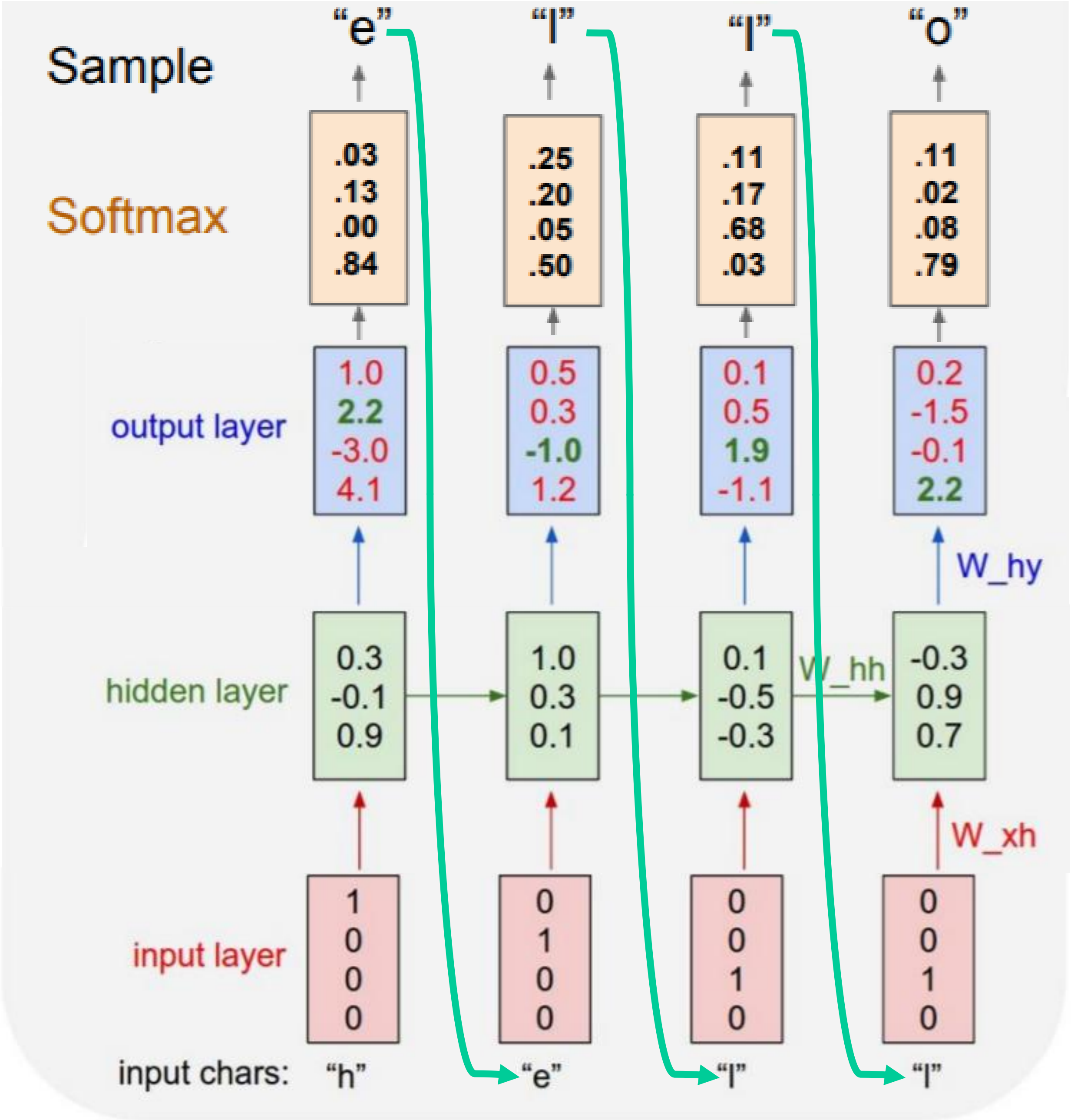
- **Training** sequence: 'hello'



Stanford CS231N

Example: Char. Level Language Model

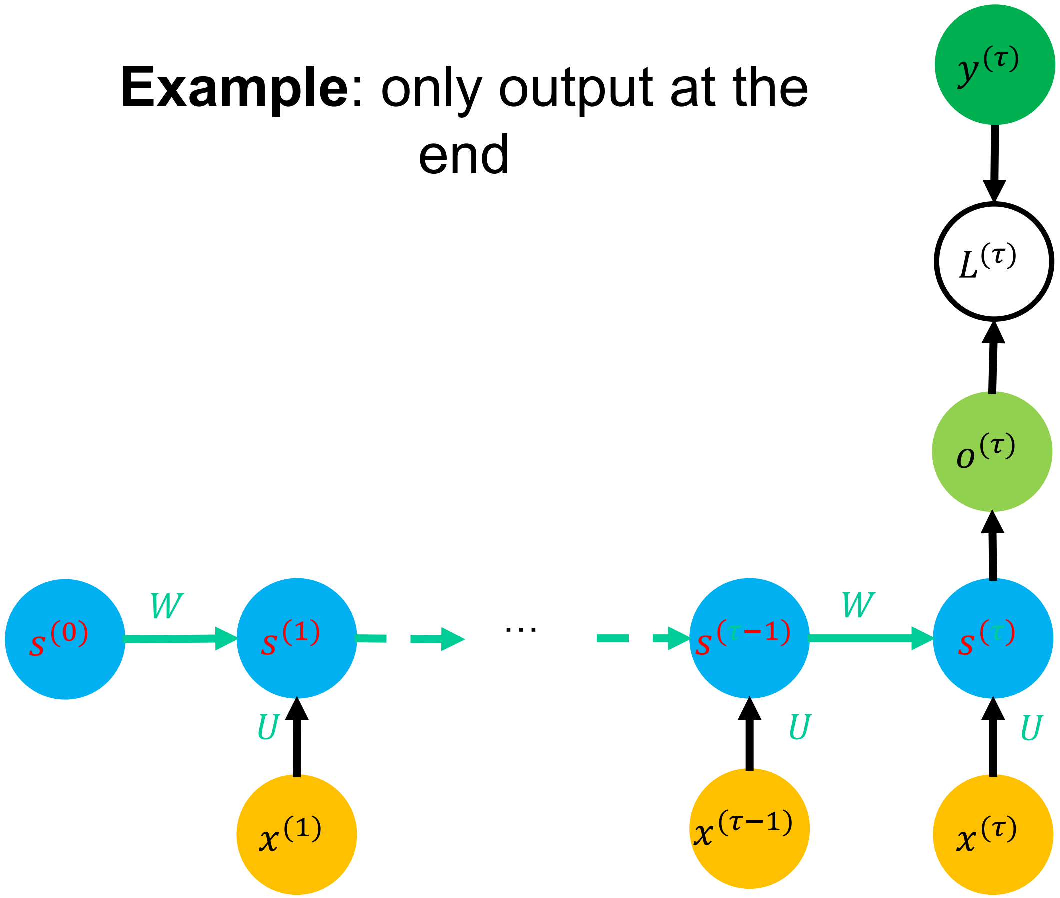
- LM goal: predict next char
- Vocabulary
  {h,e,l,o}
- **Training** sequence: ‘hello’
- Test time:
  - Sample chars and feed back into the model

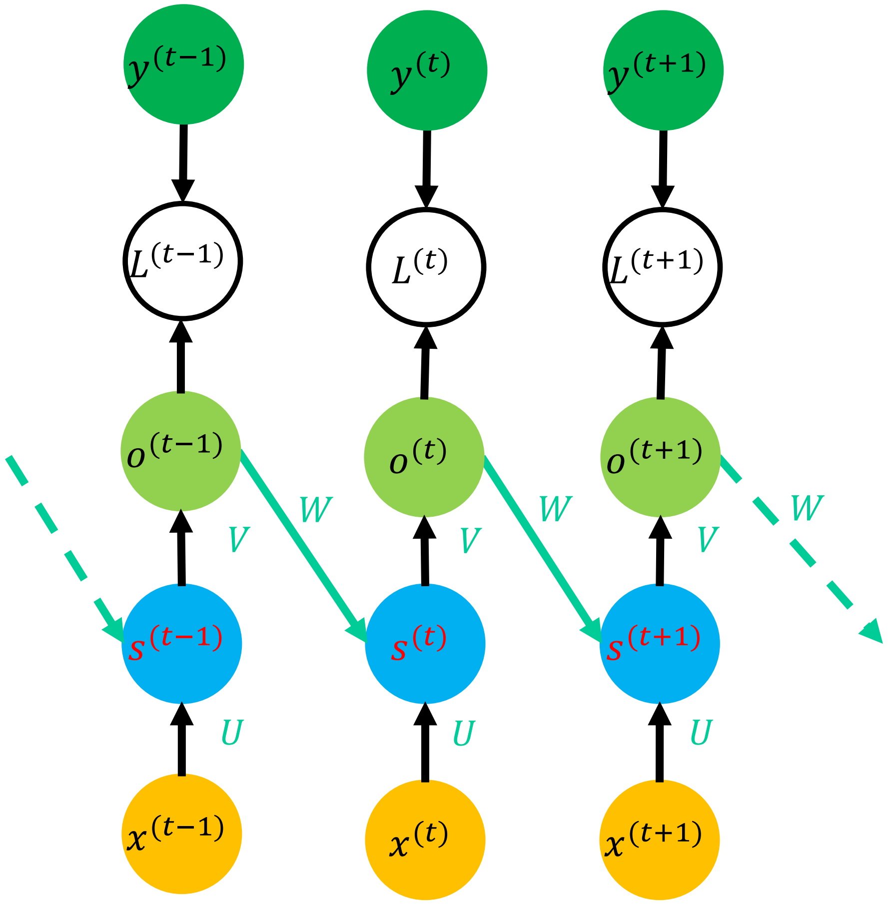


RNN Variants

Example: use the output at the previous step

Example: only output at the end

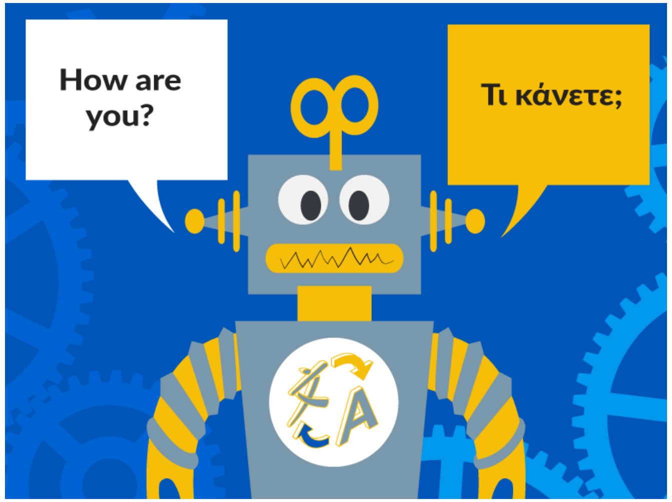




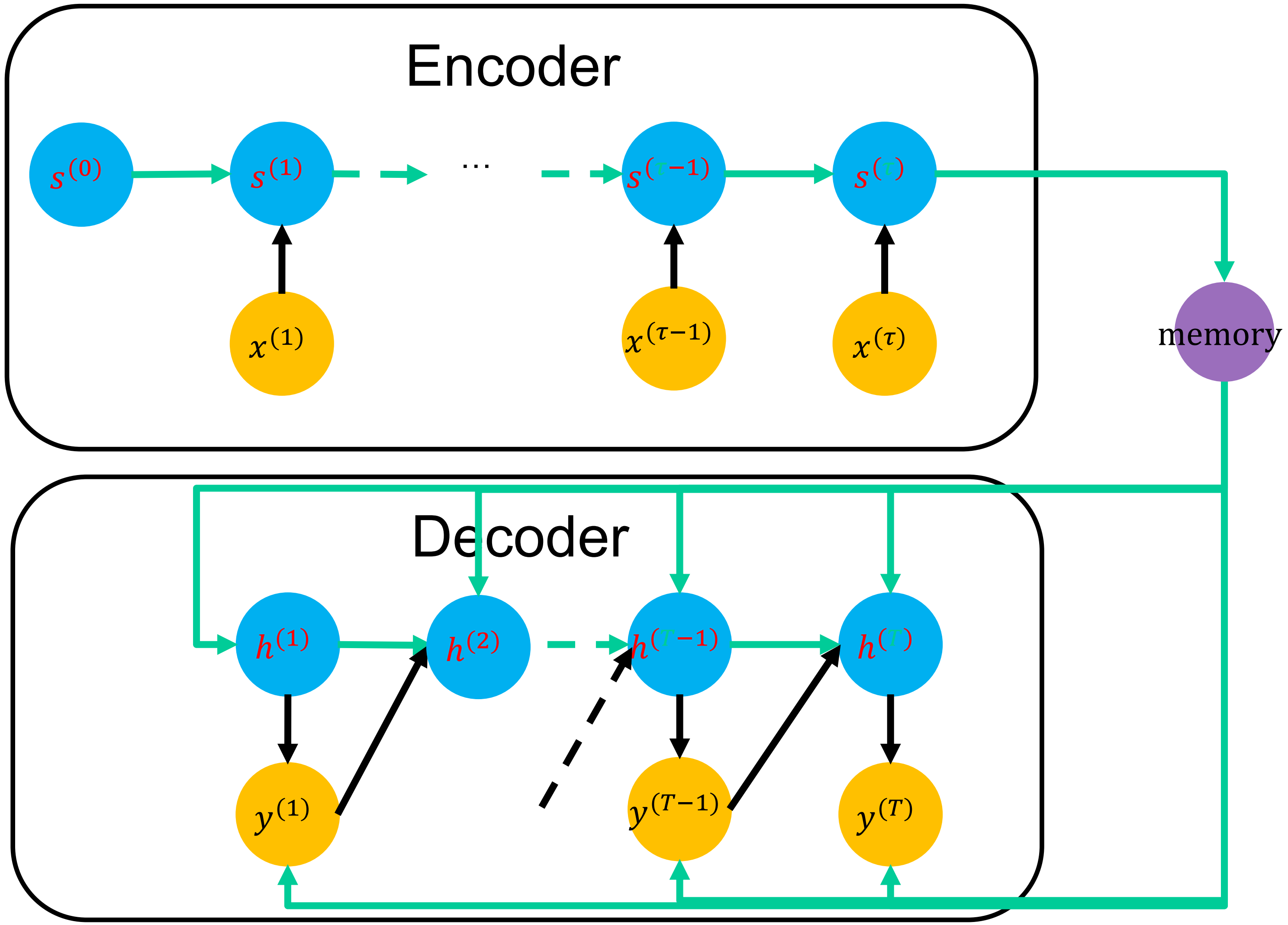
RNN Variants: Encoder/Decoder

- RNNs:
  - can map a sequence to one vector
  - or to sequences of same length

- What about mapping sequence to sequence of different length?
  - **Ex**: speech recognition, machine translation, question answering, **numerical simulation**



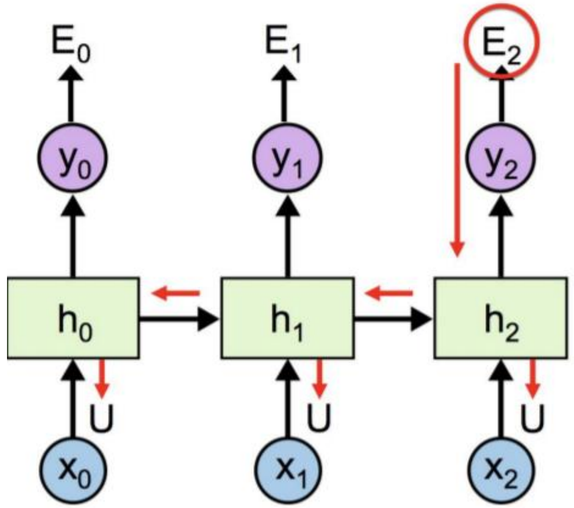
RNN Variants: Encoder/Decoder



Training RNNs

- How: Backpropagation Through Time
  - Idea: unfold the computational graph, and use backpropagation

- Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters



$$\frac{\partial E_2}{\partial U} = \frac{\partial E_2}{\partial h_2}\left(x_2^T + \frac{\partial h_2}{\partial h_1}\left(x_1^T + \frac{\partial h_1}{\partial h_0} x_0^T\right)\right)$$

RNN Problems

- What happens to gradients in backprop w. many layers?
  - In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily **explode or vanish**.
  - We can avoid this by initializing the weights very carefully.

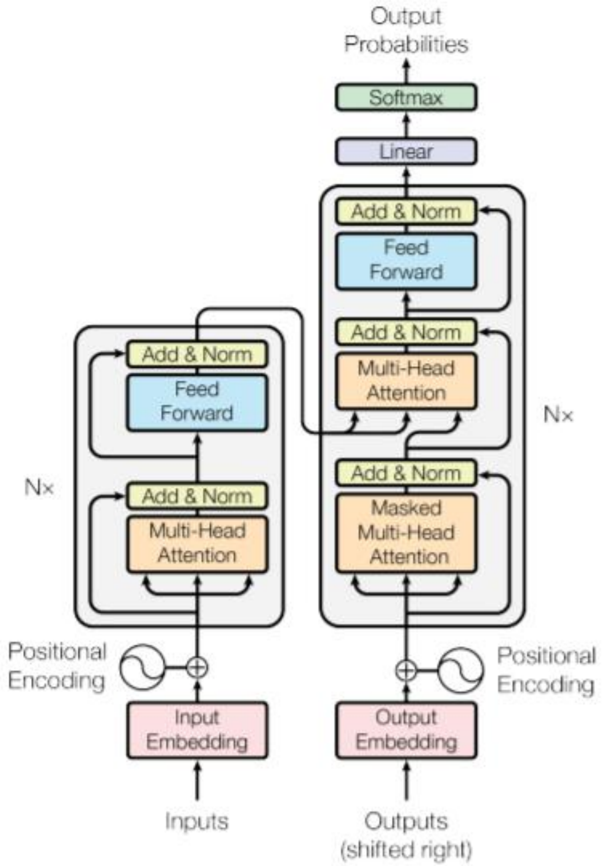
- Even with good initial weights, very hard to detect that current target output **depends** on an input from long ago.

- RNNs have difficulty dealing with long-range dependencies.

- **Most popular solution: LSTMs**

Transformers: Idea

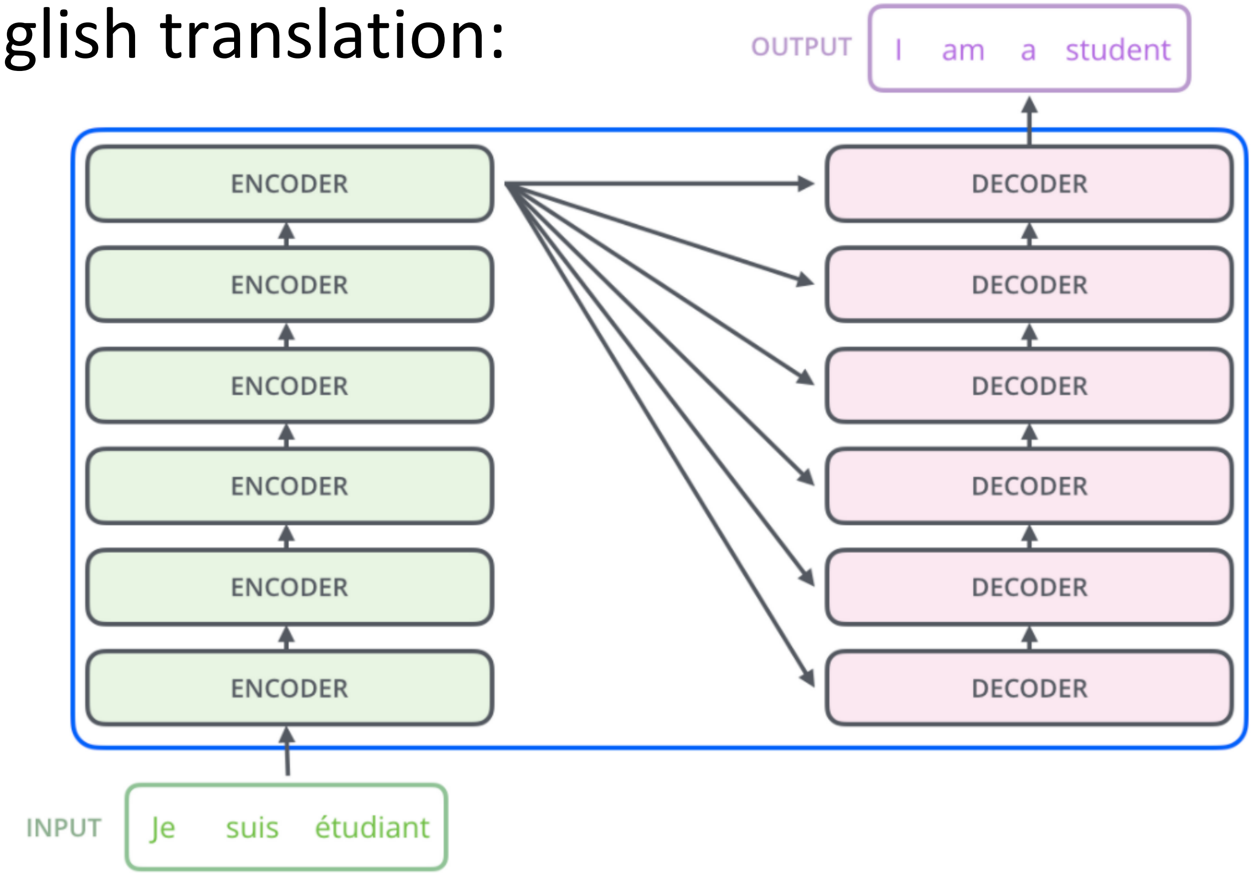
- Initial goal for an architecture: encoder-decoder
  - Get **rid of recurrence**
  - Replace with **self-attention**



Vaswani et al. '17

Transformers: Architecture

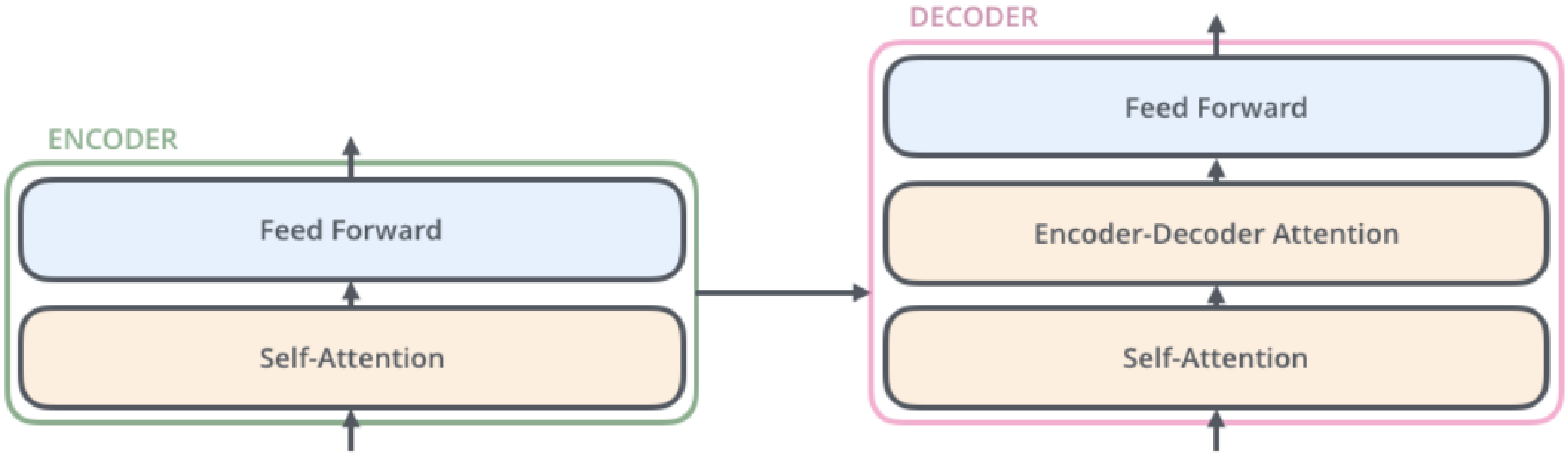
- Sequence-sequence model with **stacked** encoders/decoders:
  - For example, for French-English translation:



Excellent resource: https://jalammar.github.io/illustrated-transformer/

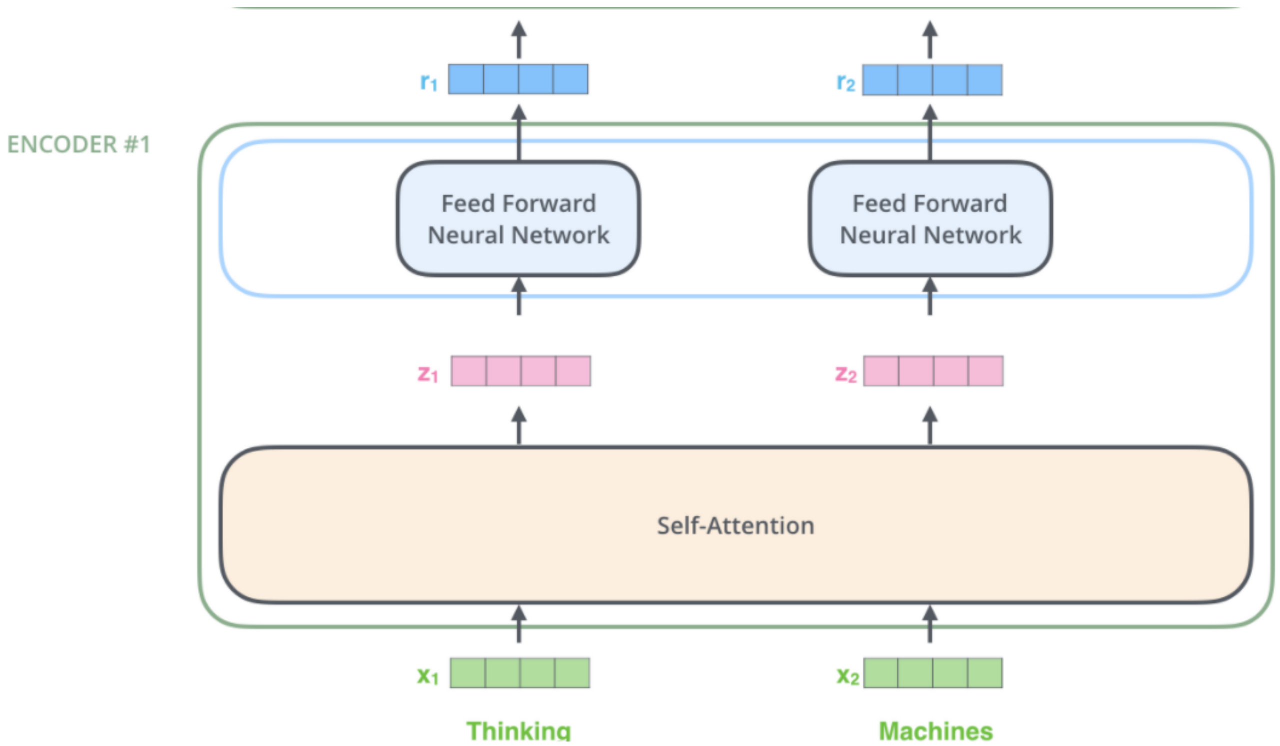
Transformers: Architecture

- Sequence-sequence model with **stacked** encoders/decoders:
  - What's inside each encoder/decoder unit?



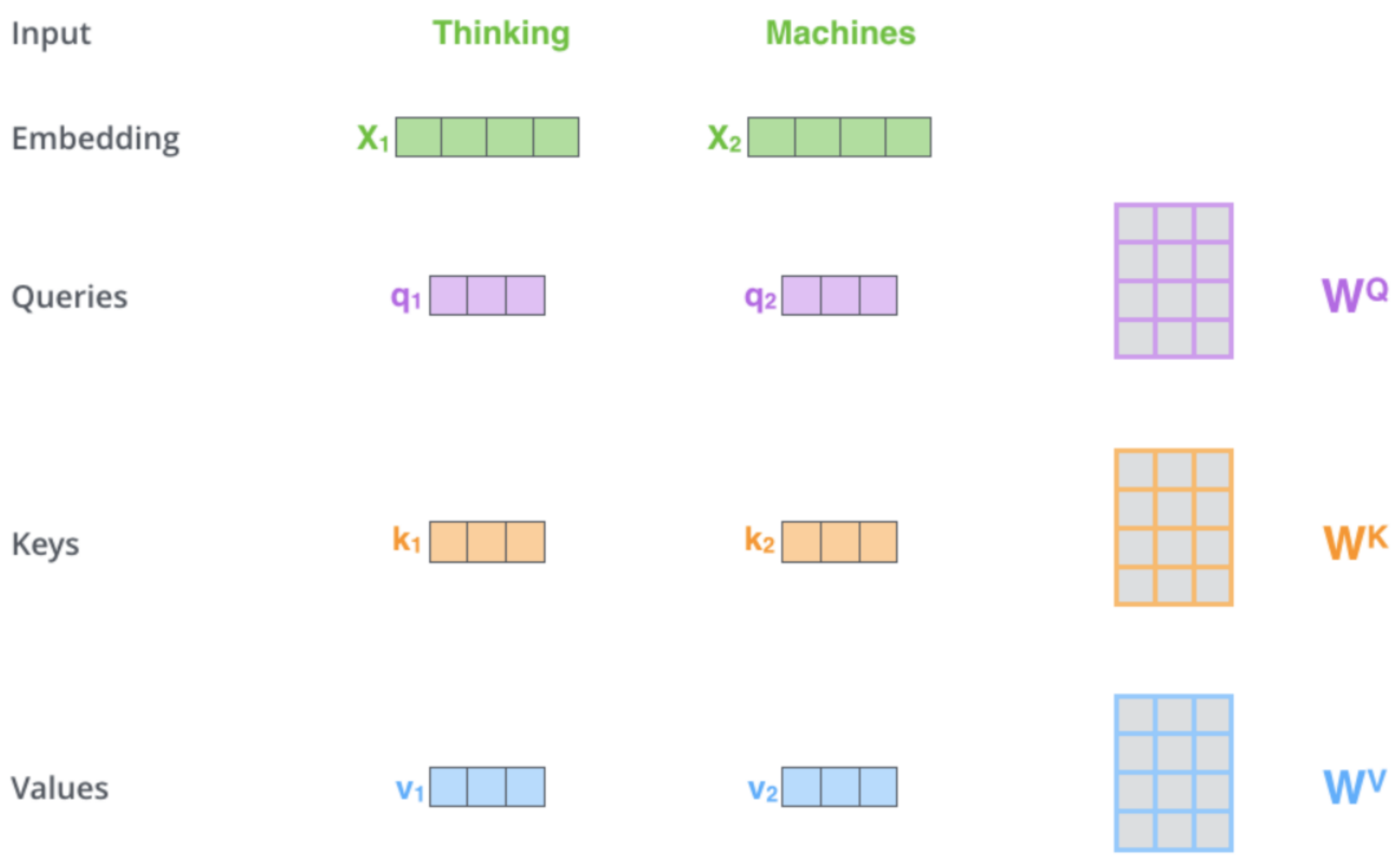
Transformers: Inside an Encoder

- Let's take a look at the encoder. Two components:
  - 1. **Self-attention** layer
  - 2. **Feedforward nets**



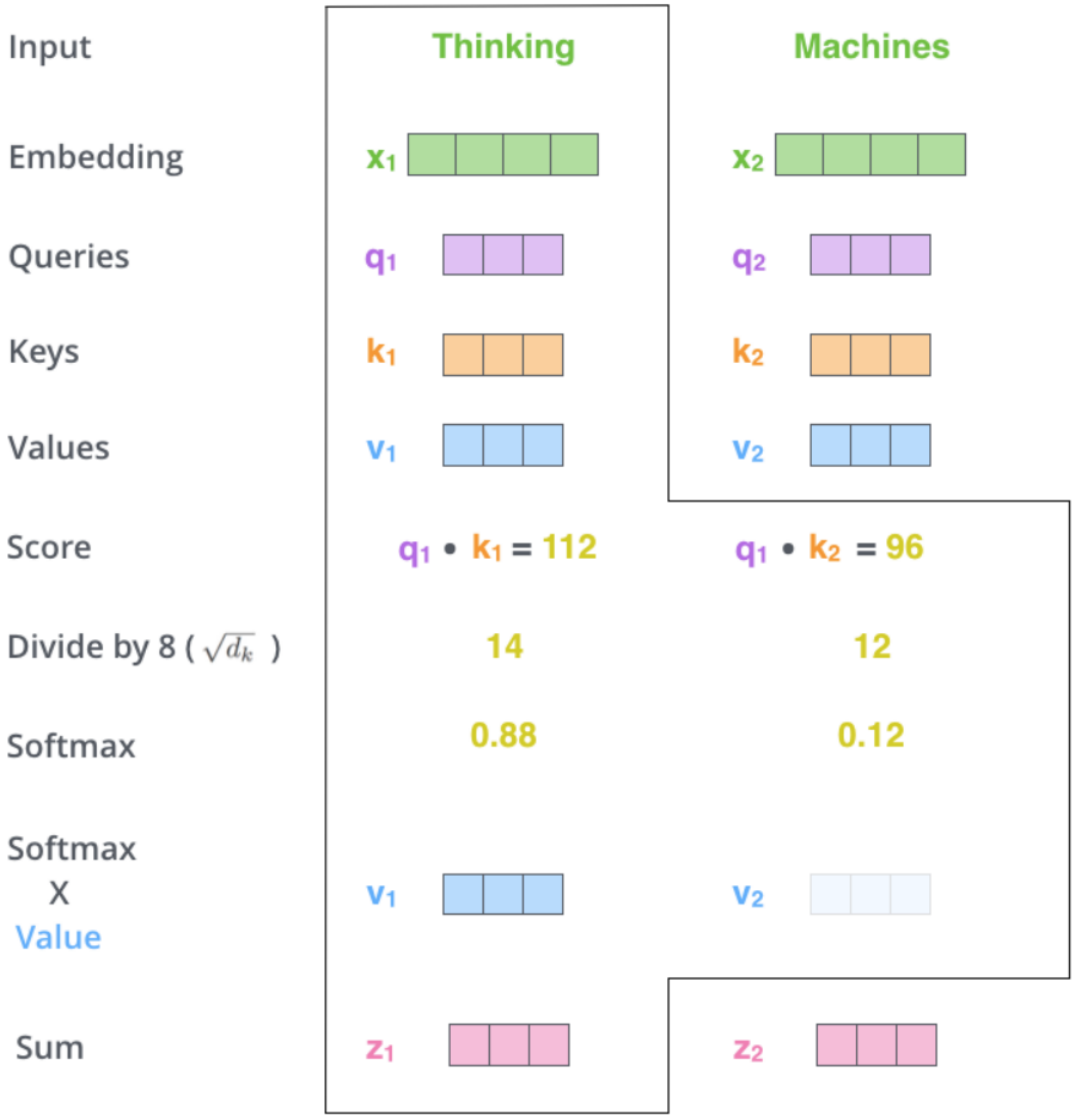
Transformers: Self-Attention

- Self-attention is the key layer in a transformer stack
  - Get 3 vectors for each embedding: **Query**, **Key**, **Value**



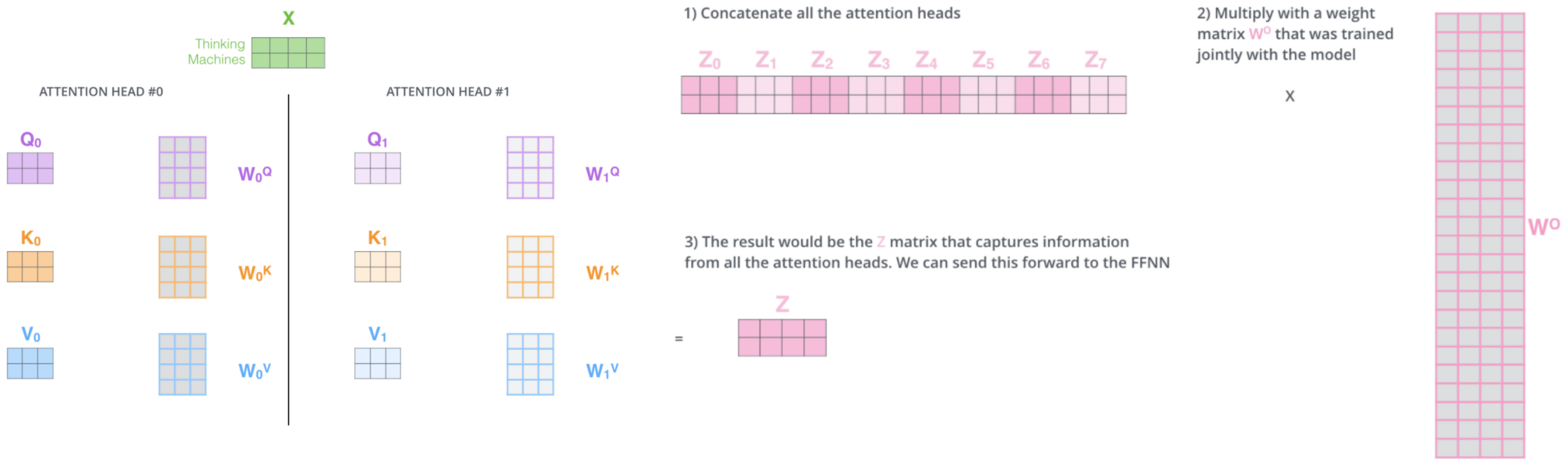
Transformers: Self-Attention

- Self-attention is the key layer in a transformer stack
  - Illustration. Recall the three vectors for each embedding: **Query**, **Key**, **Value**
  - The sum values are the outputs of the self-attention layer
  - Send these to feedforward NNs
- Highly parallelizable!



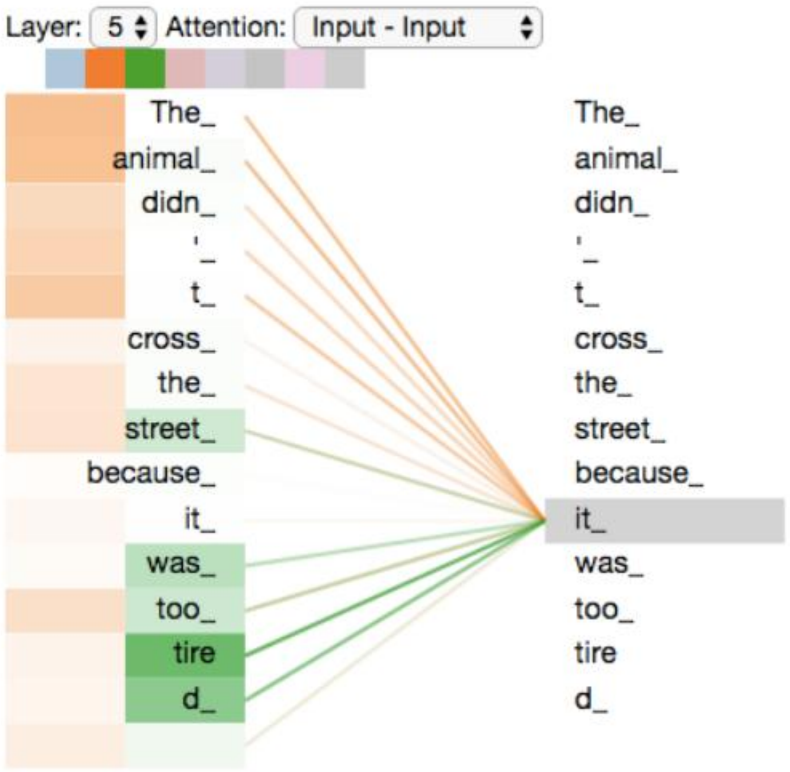
Transformers: Multi-Headed Attention

- We can do this multiple times in parallel
  - Called multiple heads
  - Need to combine the resulting output sums



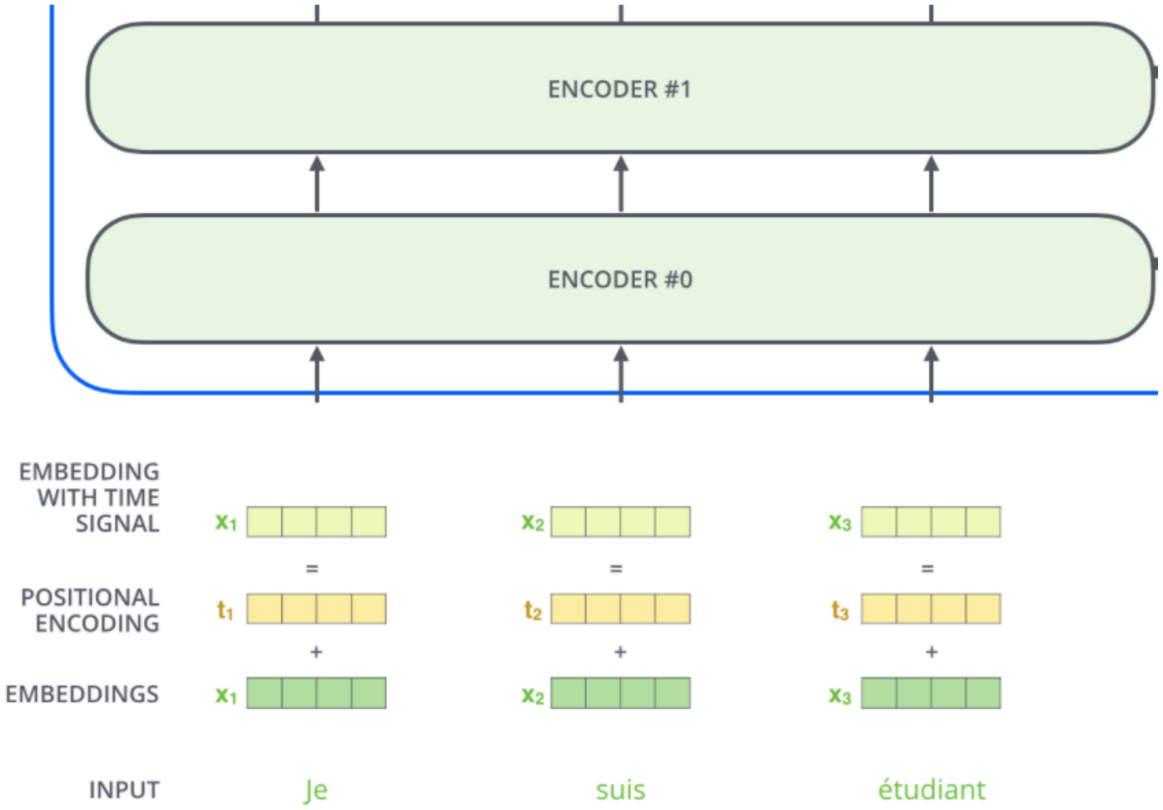
Transformers: Attention Visualization

- Attention tells us where to focus the information
  - Illustration for a sentence:



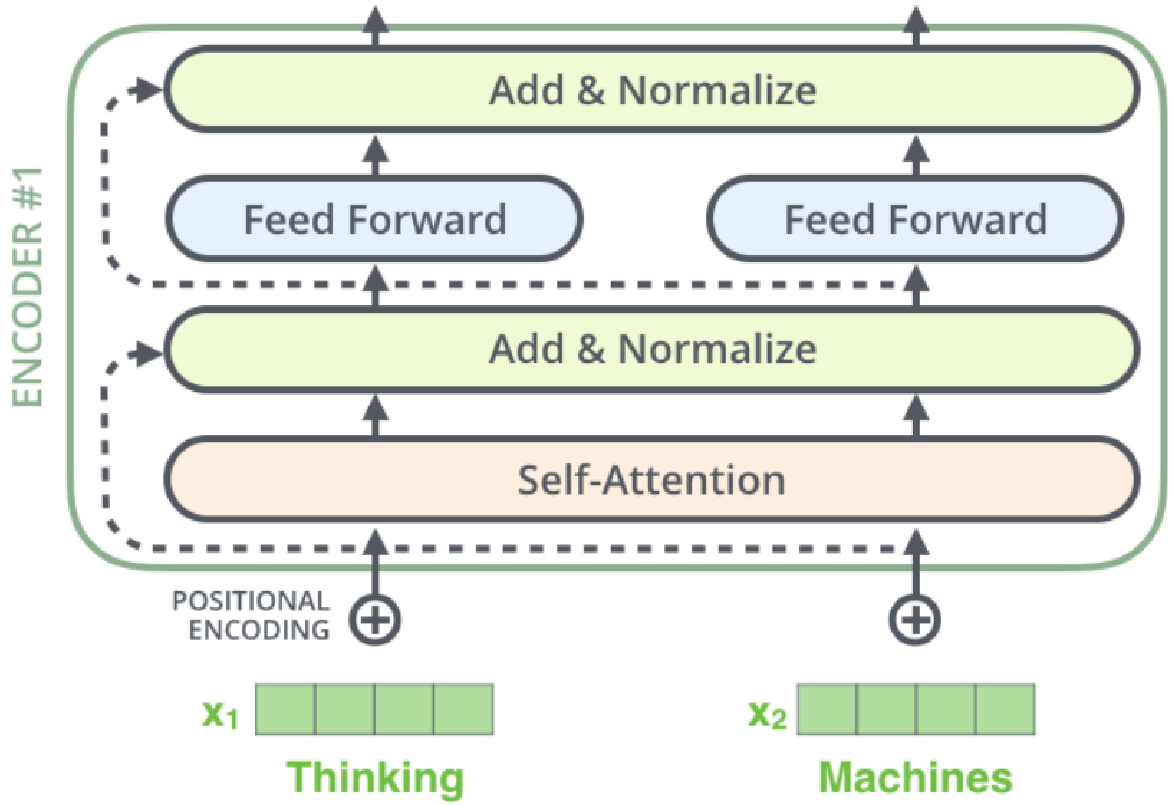
Transformers: Positional Encodings

- One thing we haven't discussed: the order of the symbols/elements in the sequence
  - Add a vector containing a special positional formula's embedding



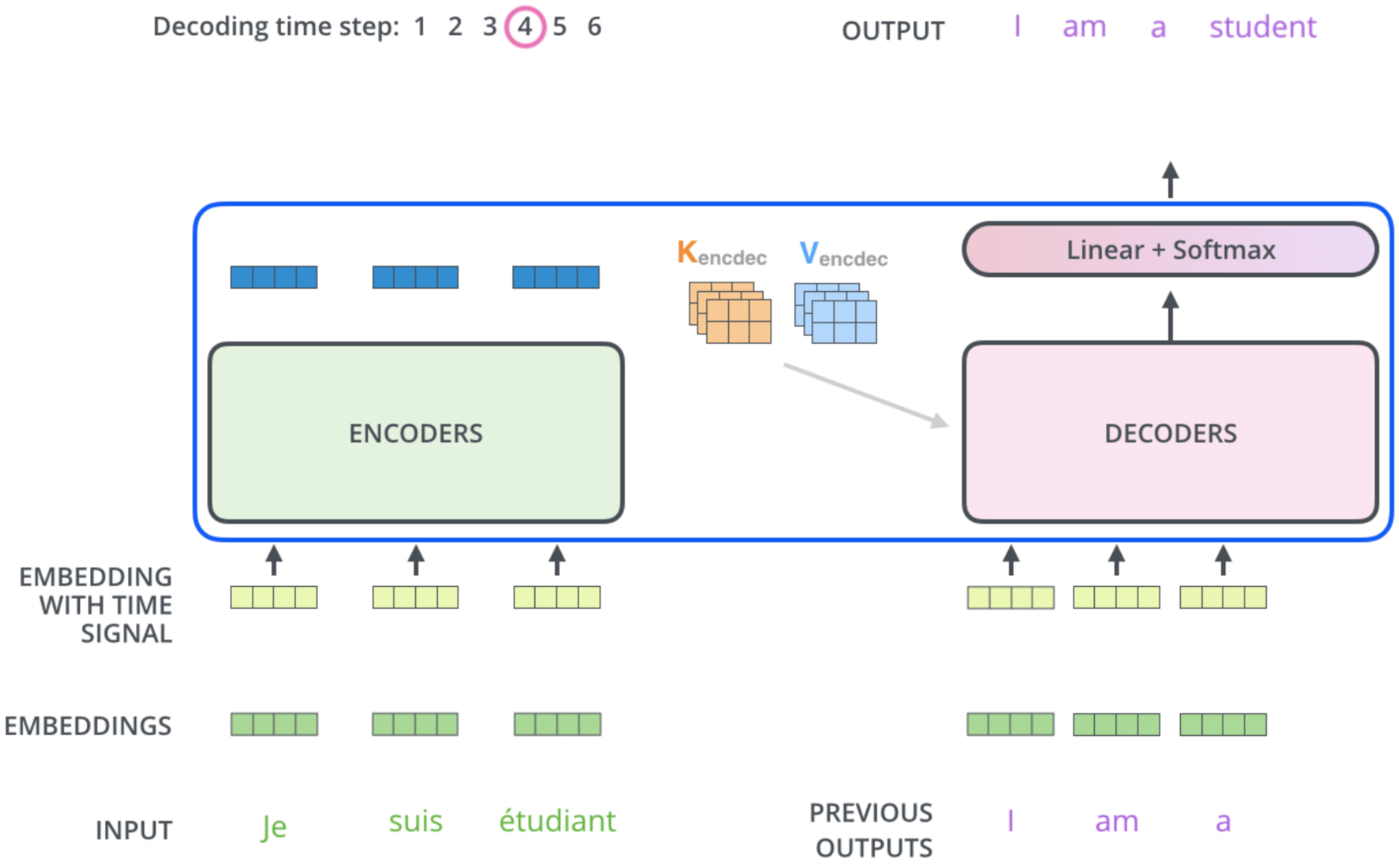
Transformers: More Tricks

- Recall a big innovation for ResNets: residual connections
  - And also layer normalizations
  - Apply to our encoder layers



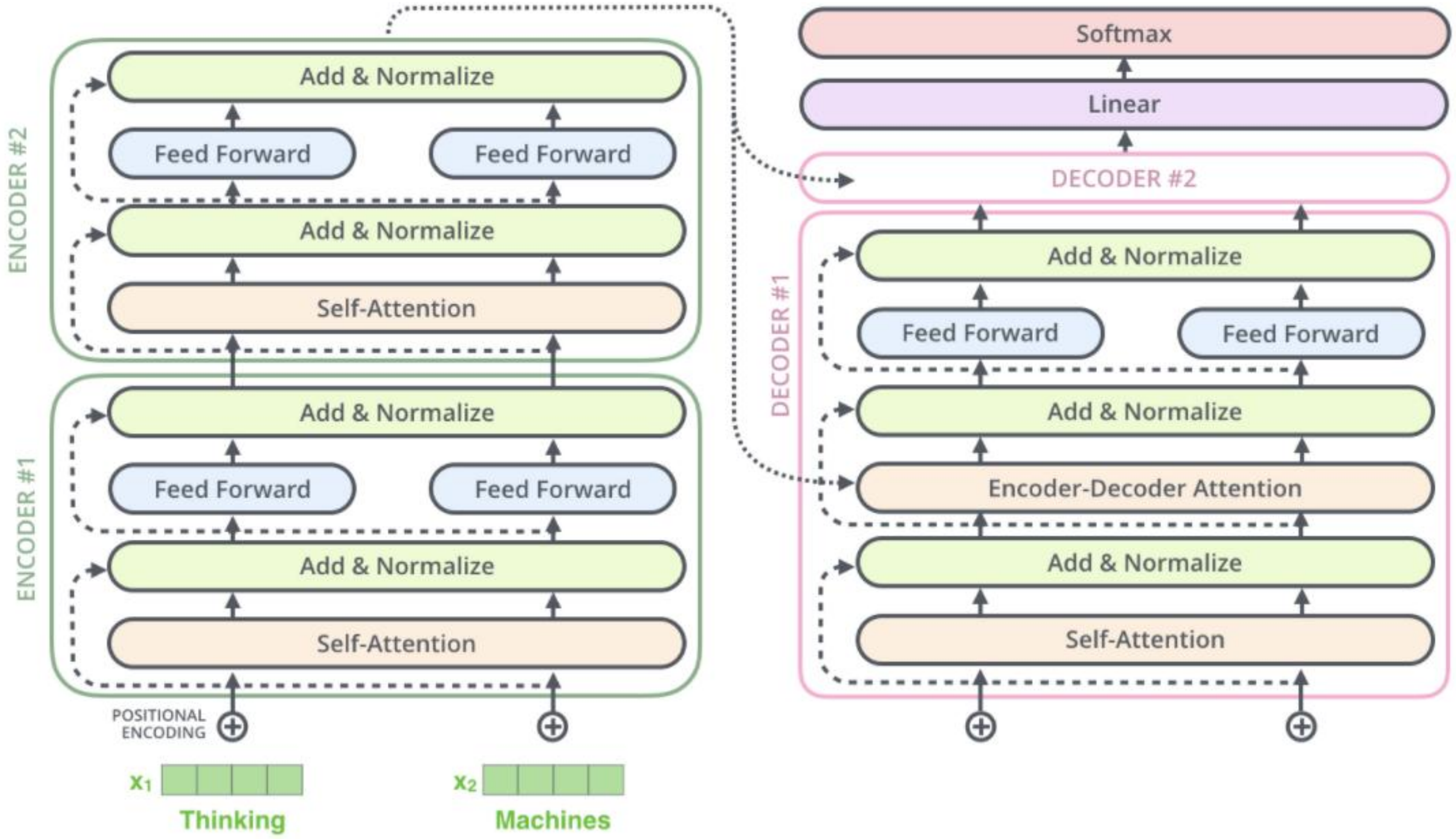
Transformers: Decoder

- Similar to encoders
- e.g. generating a translation



Transformers: Putting it All Together

- What does the full architecture look like?

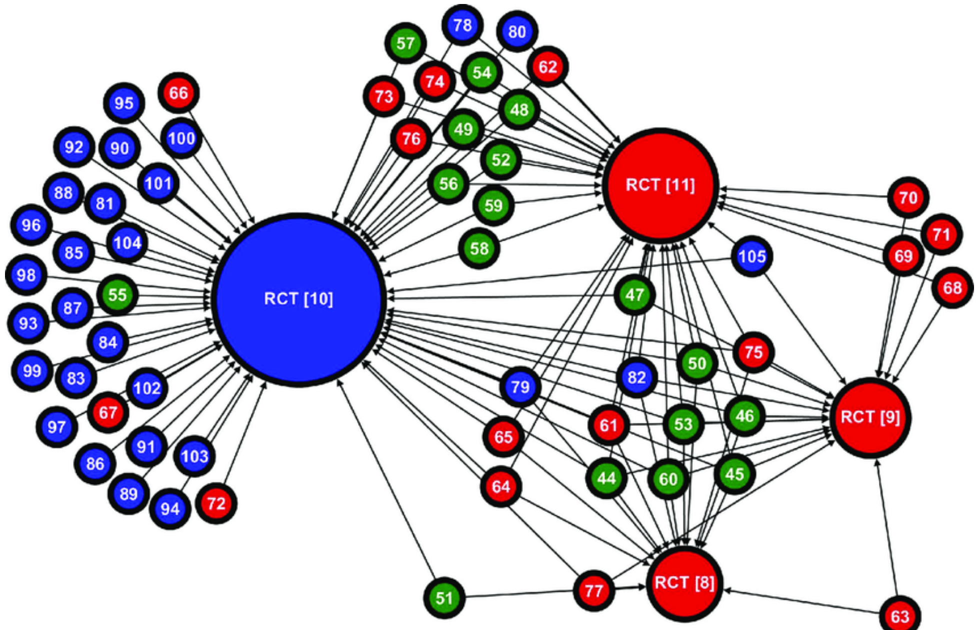


Graph Neural Networks: Motivations

- **Setting**: data that comes with some associated graph structure indicating similarity

- **Example:** citation networks.
    - Instances are scientific papers
    - Labels: subfield/genre
    - Graphs: if a paper cites another, there's an edge between them

- **Example:** meshes on which PDEs are solved



Leng

Graph Neural Networks: Approach

- **Idea**: want to use the graph information in our predictions.
- One popular network: graph convolutional network (GCN)

$$f(X, A) = \text{softmax}(\underbrace{A}_{\text{adjacency matrix}}\sigma(AX\underbrace{W^{(0)}}_{\text{layer 1 weights}})\underbrace{W^{(1)}}_{\text{layer 2 weights}})$$

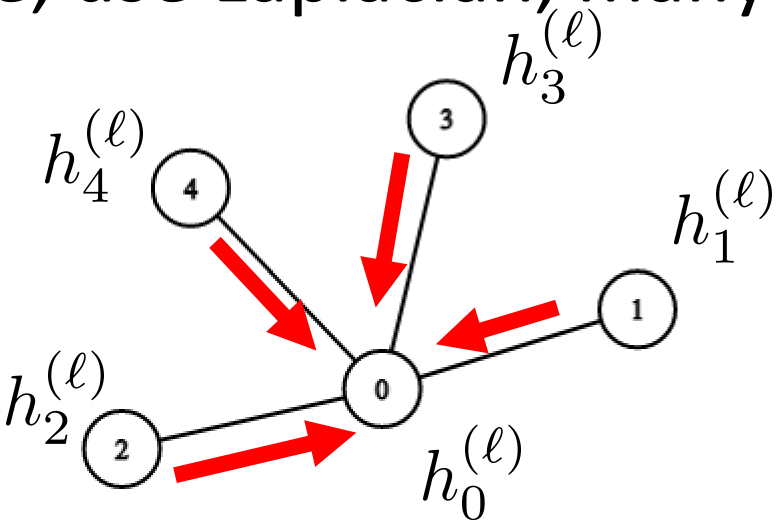
Kipf and Welling: "Semi-Supervised Classification with Graph Convolutional Networks"

Graph Convolutional Networks

- One popular network: graph convolutional network (GCN)

$$f(X, A) = \mathrm{softmax}(A\sigma(AXW^{(0)})W^{(1)})$$

- Just like a feedforward network, but also mix together nodes by multiplying by adjacency matrix
- Can also normalize, use Laplacian, many variations



Graph Convolutional Networks

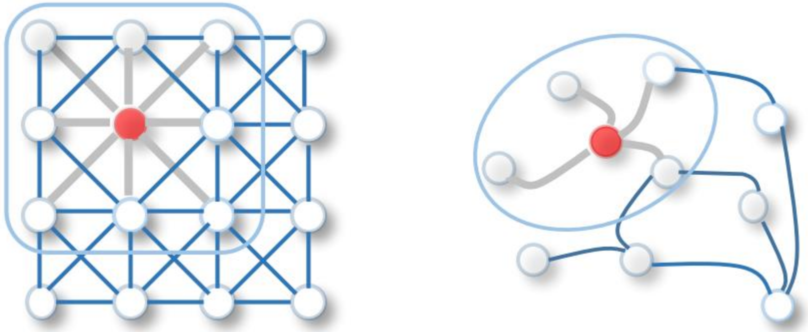
- One popular network: graph convolutional network (GCN)

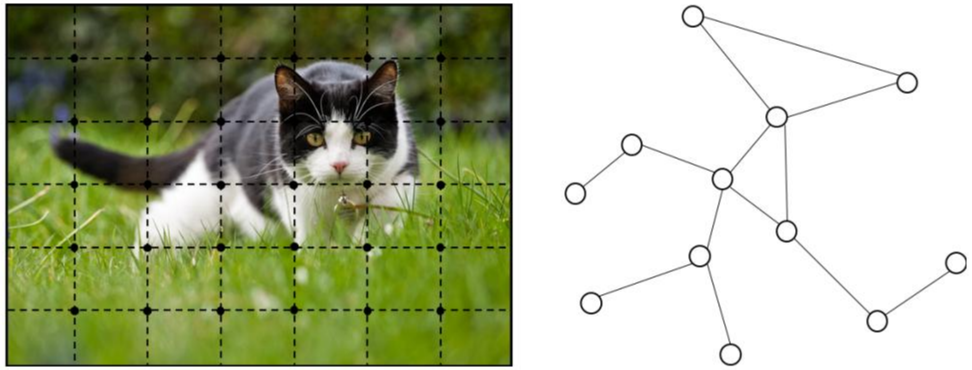
$$f(X, A) = \mathrm{softmax}(A\sigma(AXW^{(0)})W^{(1)})$$

Note the resemblance to CNNs:

- Pixels: arranged as a very regular graph
- Want: more general configurations (less regular)

Wu et al, A Comprehensive Survey on Graph Neural Networks





Zhou et al, Graph Neural Networks: A Review of Methods and Applications

Outline

- **Advanced neural architectures**
  - RNNs, Transformers, GNNs
- **Generative modeling**
  - density estimation, GANs, flow-based models, diffusion
- **Transfer learning**
  - pretraining, multi-task learning, foundation models

Goal: Learn a Distribution

- Want to estimate p_{data} from samples

$$x^{(1)}, x^{(2)}, \ldots, x^{(n)} \sim p_{\text{data}}(x)$$

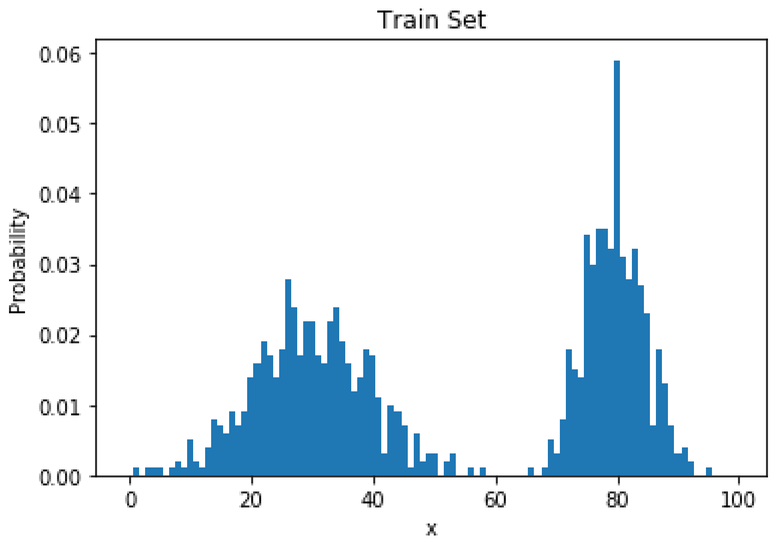
- Desired abilities:
  - **Inference**: compute p(x) for some x
  - **Sampling**: obtain a sample from p(x)

Goal: Learn a Distribution

- Want to estimate p_{data} from samples

$$x^{(1)}, x^{(2)}, \ldots, x^{(n)} \sim p_{\text{data}}(x)$$

- **One way**: build a histogram:
- Bin data space into k groups.
  - Estimate p_1, p_2, …, p_k
- Train this model:
  - Count times bin i appears in dataset



Histograms: Inference & Samples

- **Inference**: check our estimate of p_i
- **Sampling**: straightforward, select bin i with probability p_i, then select uniformly from bin i.

- But …
  - inefficient in high dimensions

Parametrizing Distributions

- Don't store each probability, store $p_\theta(x)$

- One approach: likelihood-based
  - We know how to train with **maximum likelihood**

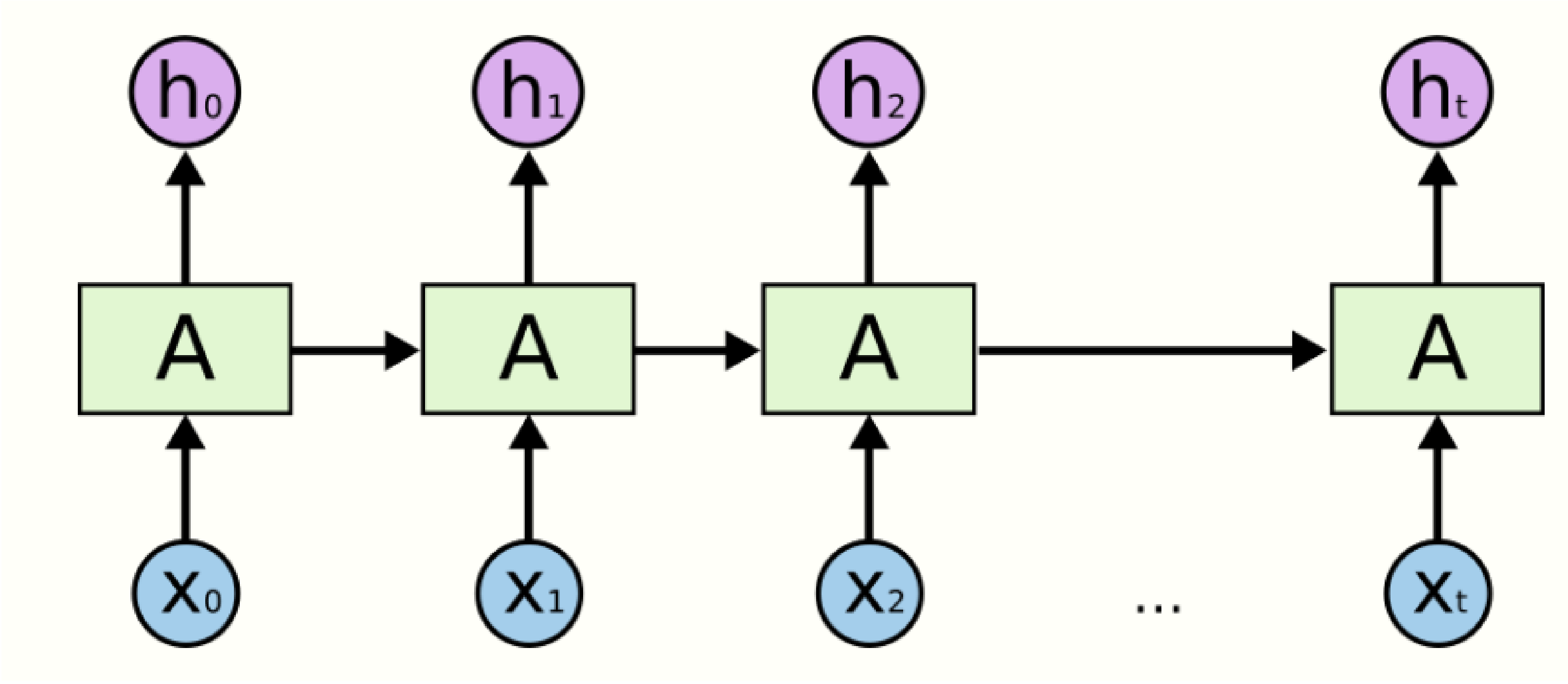
$$\arg\min_\theta - \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x^{(i)})$$

Parametrizing Distributions

- One approach: likelihood-based
  - We know how to train with **maximum likelihood**
  - Then, train with SGD
  - Just need to make some choices for $p_\theta(x)$
    - For example, recall Gaussian mixture models.
    - But many types of data have more complex underlying distributions.

Parametrizing Distributions: Autoregressive models

- e.g. recurrent neural networks, transformers.



Flow Models

- One way to specify $p_\theta(x)$
- Use a latent variable z with a “simple” (e.g Gaussian) distribution.
- Then use a “complex” transformation, $x = f_\theta(z)$.

Flow Models

- We will need to compute the inverse transformation and take its derivative as well (for training).
- So compose multiple “simple” transformations

$$x = f_{\theta_k}(f_{\theta_{k-1}}(\dots f_{\theta_1}(z)))$$

$$z = f_{\theta_1}^{-1}(f_{\theta_2}^{-1}(\dots f_{\theta_k}^{-1}(x)))$$

Flow Models

- Transform a simple distribution to a complex one via a chain of invertible transformations (the "flow")

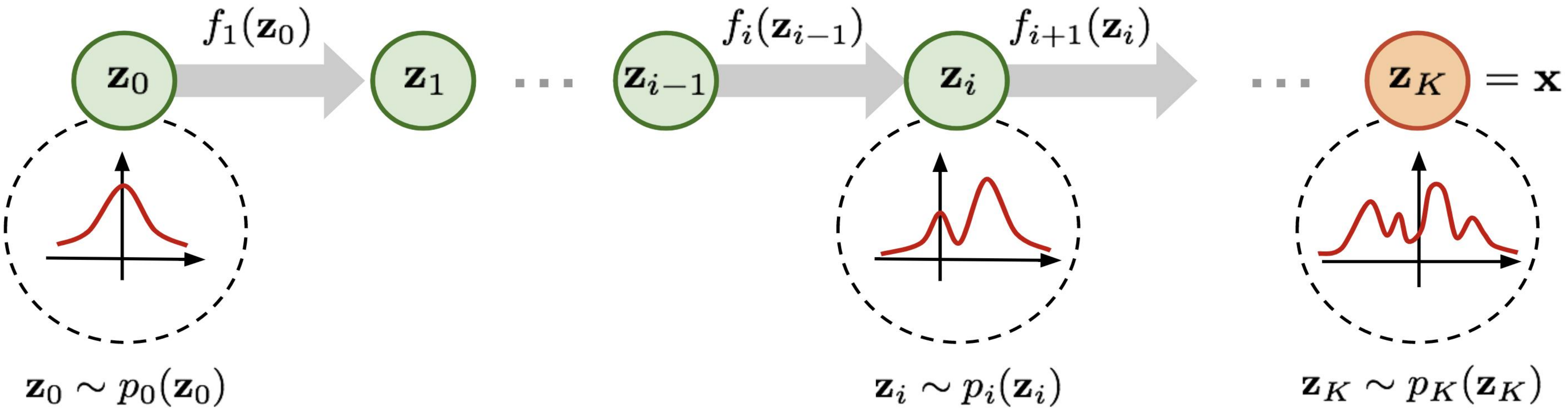
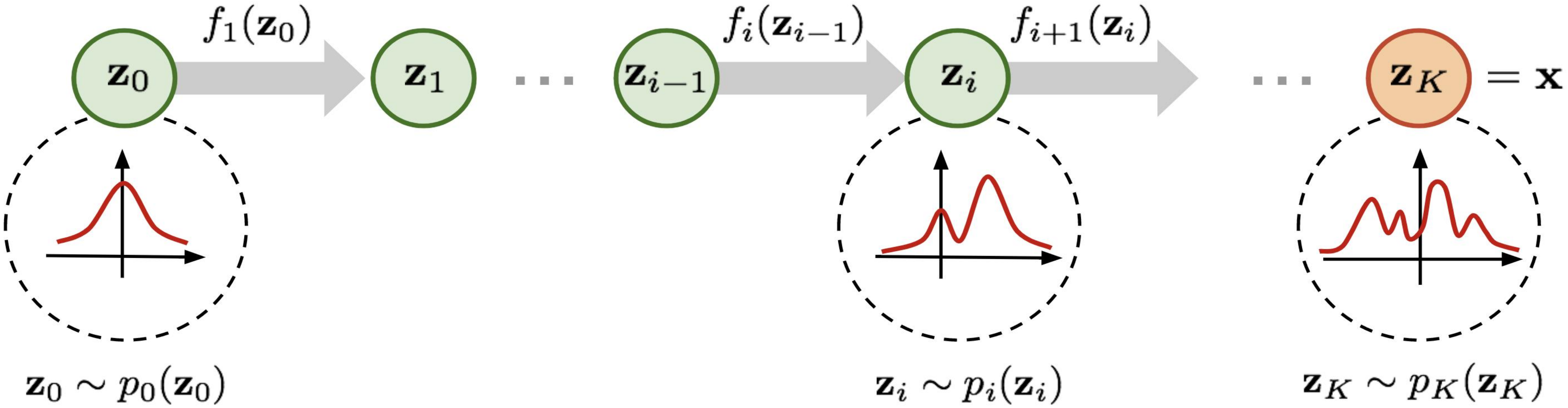


image from Lilian Weng

Flow Models: How to sample?

- Sample from z (the latent variable)---has a simple distribution that lets us do it: Gaussian, uniform, etc.
- Then run the sample z through the flow to get a sample x



Flow Models: How to train?

- Relationship between $p_x(x)$ and $p_z(z)$ (densities of x and z), given that $x = f_\theta(z)$?

$$p_x(x) = p_z(f_\theta^{-1}(x)) \left| \frac{\partial f_\theta^{-1}(x)}{\partial x} \right|$$

[change of variables]

Determinant of Jacobian matrix

Flow Models: Training

$$\max_\theta \sum_i \underbrace{\log\left(p_x(x^{(i)};\theta)\right)}_{\text{Maximum Likelihood}} = \max_\theta \left(\sum_i \log\left(p_z(f_\theta^{-1}(x^{(i)}))\right) + \log\left|\frac{\partial f_\theta^{-1}(x^{(i)})}{\partial x}\right| \right)$$

Latent variable version

Determinant of Jacobian matrix

GANs: Generative Adversarial Networks

- So far we've been modeling the density…
  - What if we just want to get high-quality samples?

- GANs do this.
  - Think of art forgery
  - Left: original
  - Right: forged version
  - Two-player game:
    - **Generator** wants to pass off the discriminator as an original
    - **Discriminator** wants to distinguish forgery from original

GANs: Basic Setup

- Let's set up networks that implement this idea:
  - **Discriminator** network
  - **Generator** network

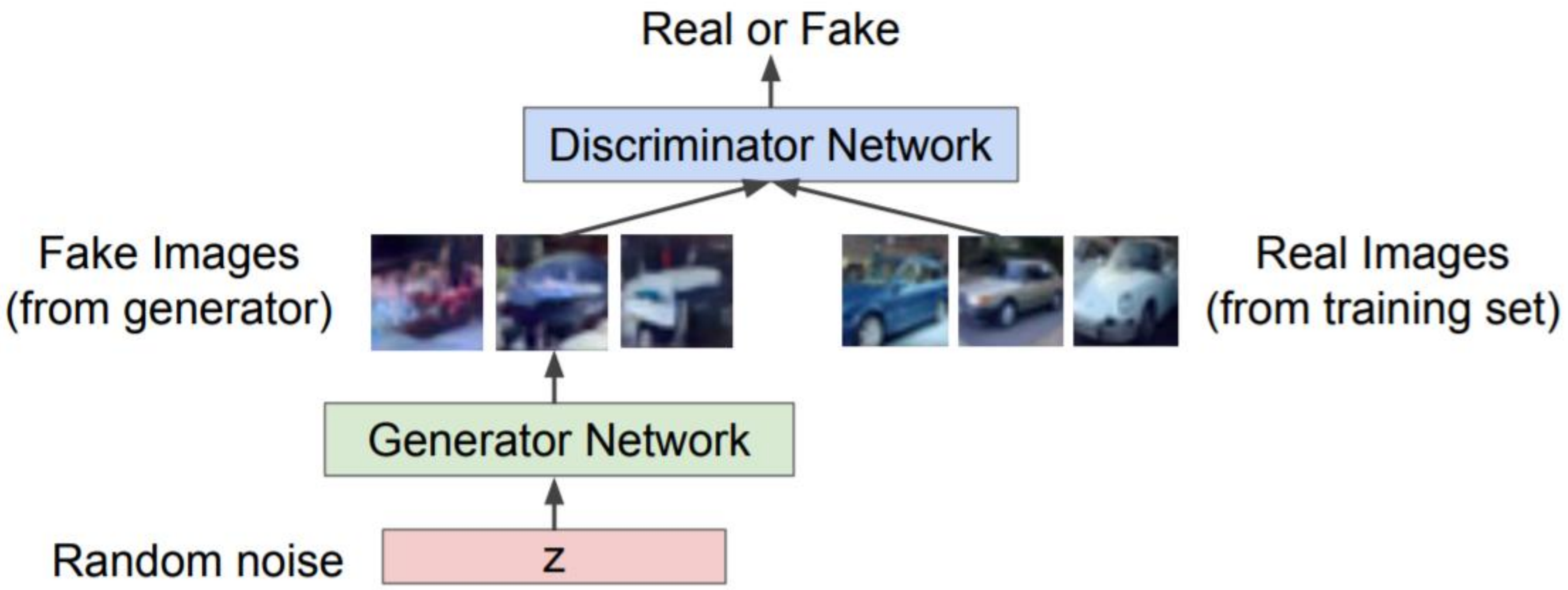


image from Stanford CS231n / Emily Denton

GAN Training: Discriminator

- How to train these networks? Two sets of parameters to learn: θ_d (discriminator) and θ_g (generator)

- Let's **fix** the generator. What should the discriminator do?
  - Distinguish fake and real data: binary classification.
  - Use the cross-entropy loss, we get

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Real data, want to classify 1

Fake data, want to classify 0

GAN Training: Generator & Discriminator

- How to train these networks? Two sets of parameters to learn: θ_d (discriminator) and θ_g (generator)

- This makes the discriminator better, but also want to make the generator more capable of fooling it:
  - Minimax game! Train jointly.

$$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Real data, want to classify 1

Fake data, want to classify 0

GAN Training: Alternating Training

- So we have an optimization goal:

$$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

- Alternate training:
  - **Gradient ascent**: *fix generator*, make the discriminator better:

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

  - **Gradient descent**: *fix discriminator*, make the generator better

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

GAN Training: Issues

- Training often not stable
- Many tricks to help with this:
  - Replace the generator training with

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

  - Better gradient shape
  - Choose number of alternating steps carefully

- Can still be challenging.

GAN Architectures

- **Discriminator**: image classification, use a **CNN**
- What should **generator** look like
  - Input: noise vector z.
  - Output: an image (i.e. a 3-channel x width x height volume)
  - Similar to a reversed CNN pattern…

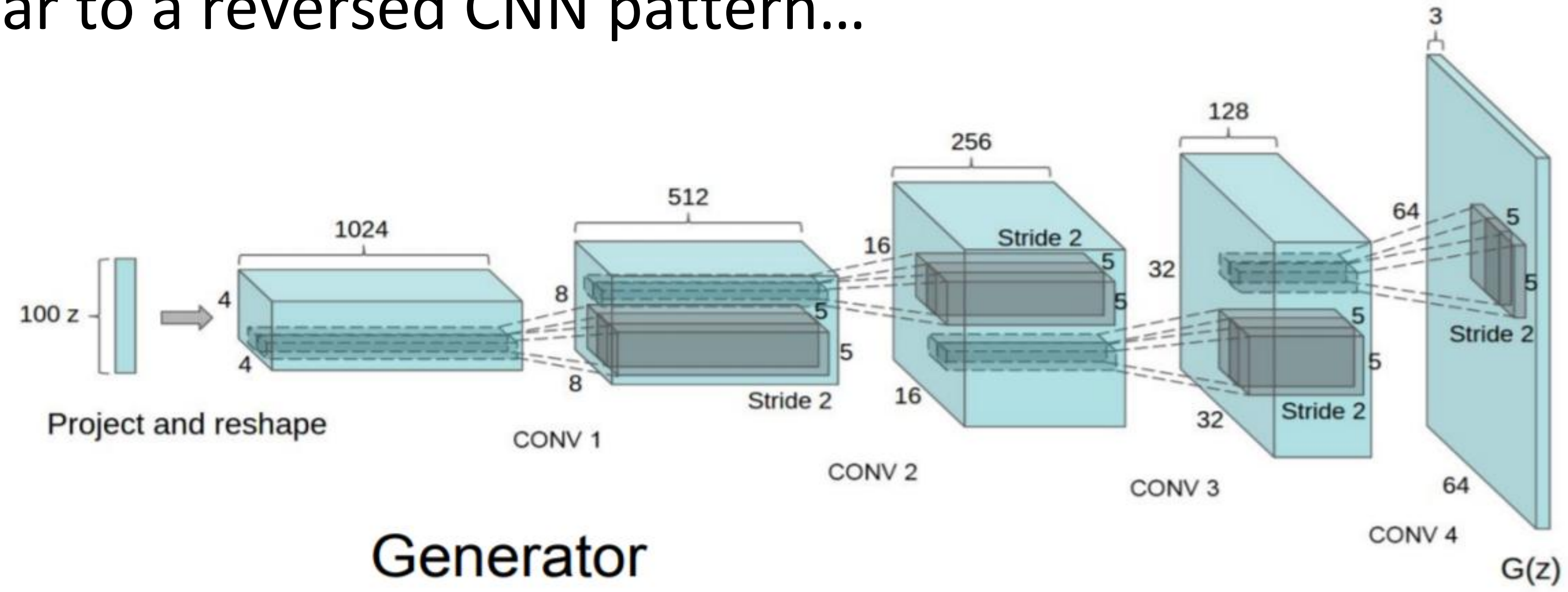
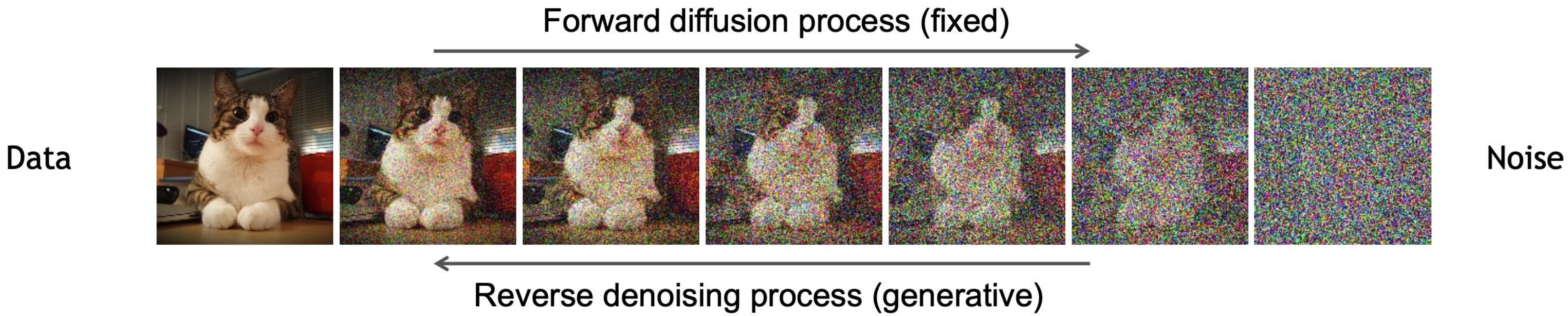


image from Radford et al '16

Diffusion Models

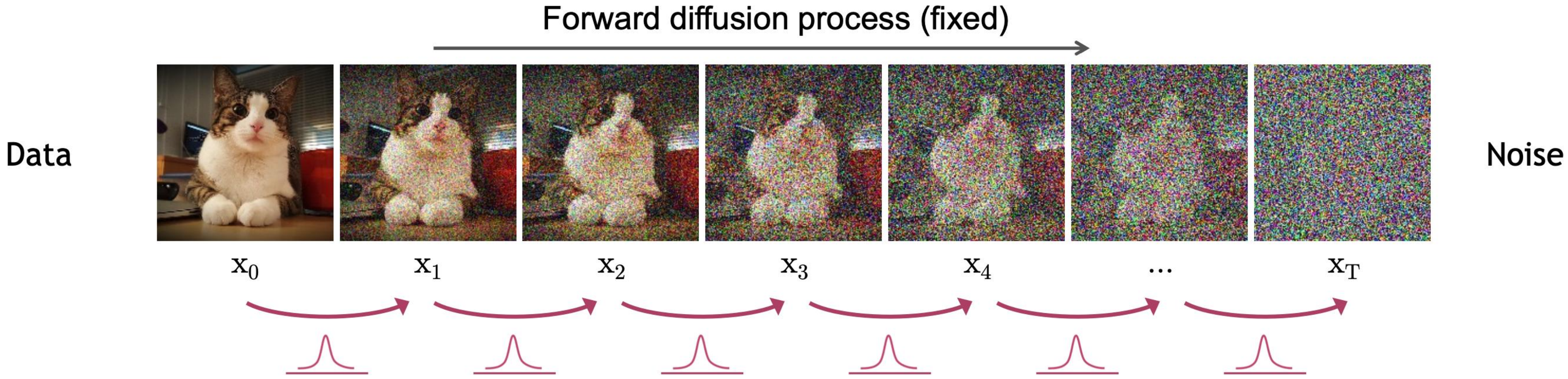
- **Learning to generate by denoising**
- Denoising diffusion models consist of two processes:
  - Forward diffusion process that gradually adds noise to input
  - Reverse denoising process that learns to generate data by denoising



Some slides were borrowed from
Denoising Diffusion-based Generative Modeling CVPR2022 tutorial
(https://cvpr2022-tutorial-diffusion-models.github.io/)

Diffusion Models

- The formal definition of the forward process in T steps:

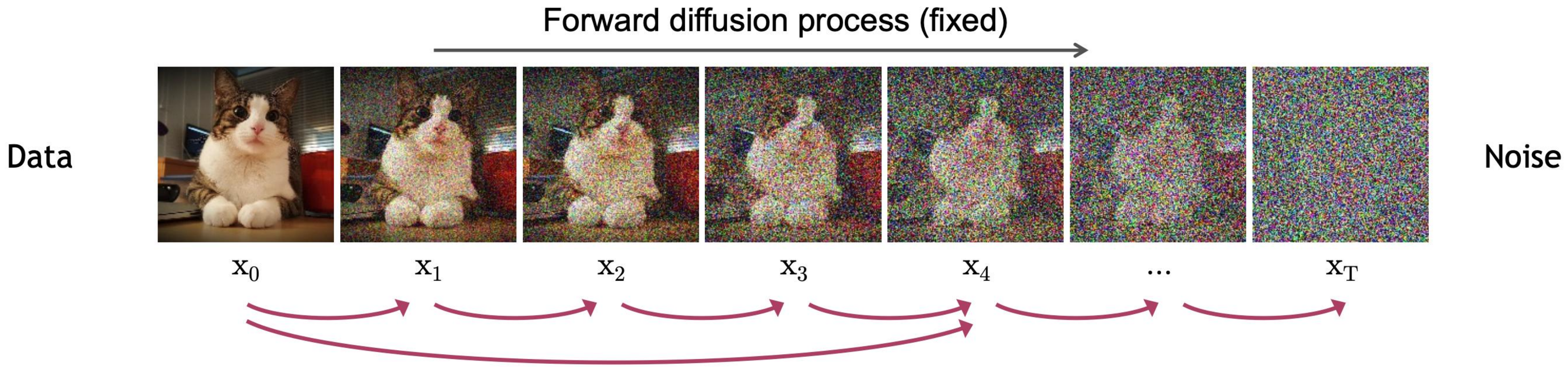


$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x_t}; \sqrt{1-\beta_t}\mathbf{x_{t-1}}, \beta_t\mathbf{I}) \quad \Rightarrow \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

(joint)

Diffusion Models

- Diffusion Kernel



Define $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$ $\Rightarrow$ $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}))$ (Diffusion Kernel)

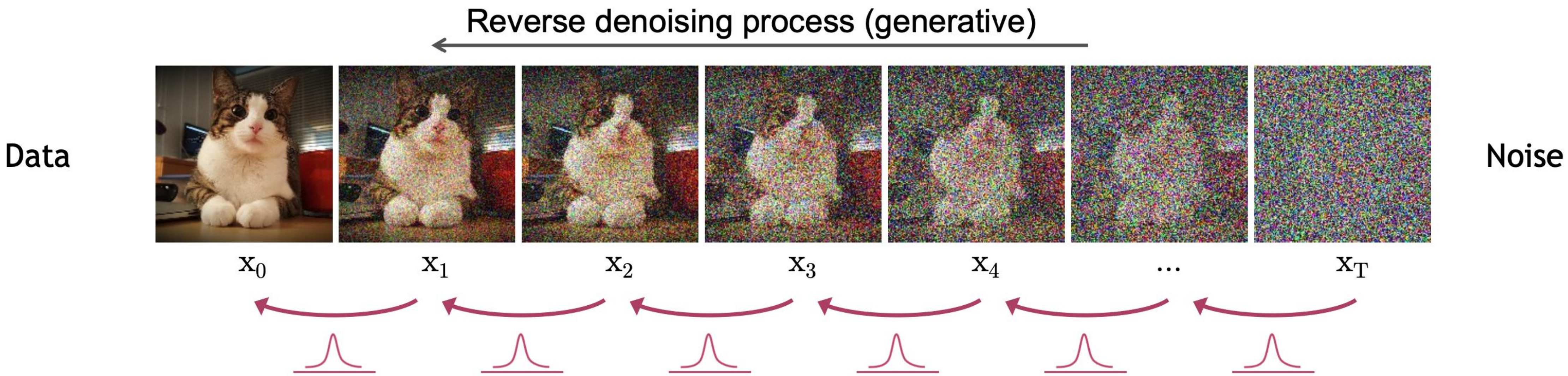
For sampling: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\,\epsilon$ where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

β_t values schedule (i.e., the noise schedule) is designed such that $\bar{\alpha}_T \to 0$ and $q(\mathbf{x}_T|\mathbf{x}_0) \approx \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I}))$

Diffusion Models

- Reverse Denoising Process

Formal definition of forward and reverse processes in T steps:



$$p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$$

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \underbrace{\mu_\theta(\mathbf{x}_t, t)}_{\substack{\text{Trainable network} \\ \text{(U-net, Denoising Autoencoder)}}}, \sigma_t^2 \mathbf{I})$$

$$\Rightarrow \quad p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

Outline

- **Advanced neural architectures**
  - RNNs, Transformers, GNNs
- **Generative modeling**
  - density estimation, GANs, flow-based models, diffusion
- **Transfer learning**
  - pretraining, multi-task learning, foundation models

Transfer learning

We typically assume labeled points $(x_1, y_1), \ldots, (x_n, y_n) \sim D$ drawn i.i.d. from the **target distribution** D

What if:

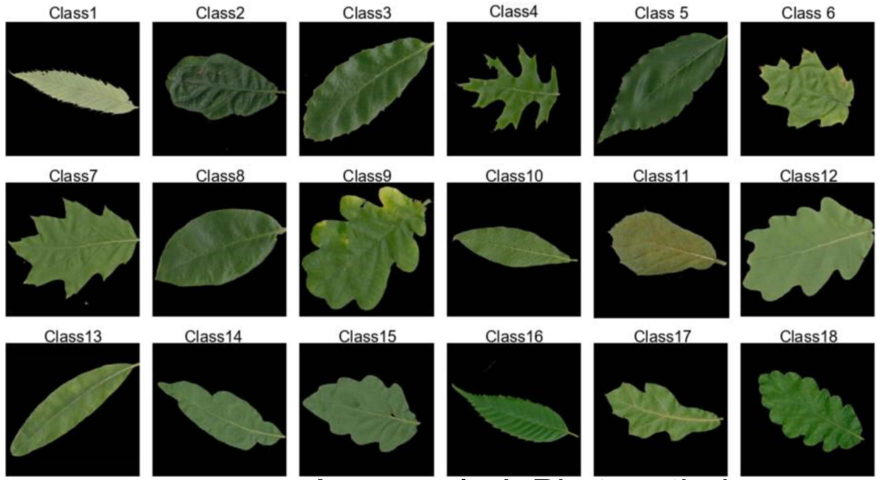
- n is too small to learn a sufficiently expressive model
- but we have access to more data $(x_1', y_1'), \ldots, (x_N', y_N') \sim D'$ from a **related distribution** D'?

Using data from a related distribution to improve performance on the target distribution is **transfer learning**

Canonical example: ImageNet

standard vision pipeline:

1. collect a bunch of data for your target task
2. download a large CNN (e.g. a big ResNet) trained on ImageNet and **replace its classification layer**
3. then
   I. either pass its **features** to a simpler model
   II. or **fine-tune** it directly on the task



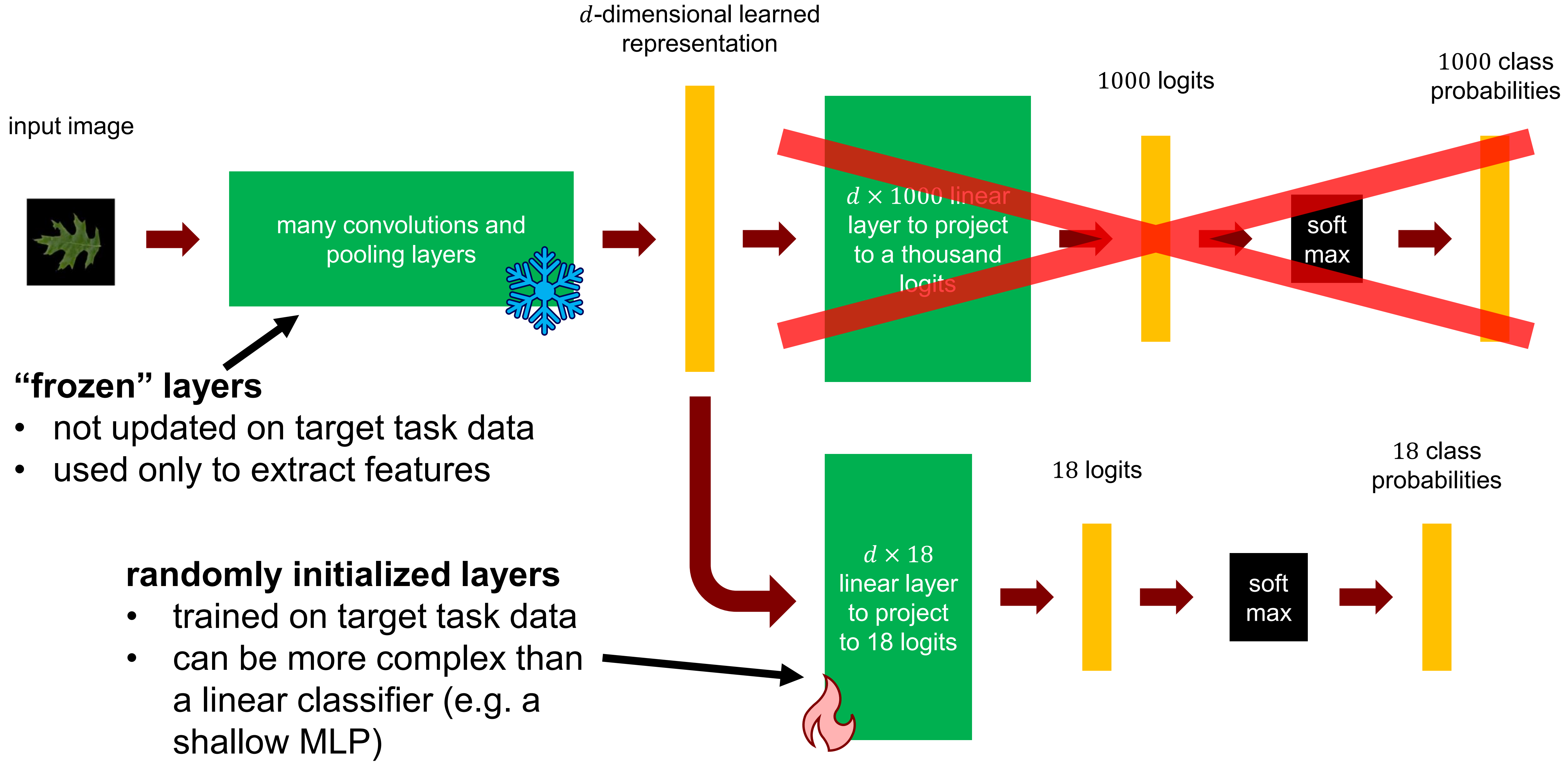
Arun et al. *J. Phytopathology*.

a few datapoints for a few classes

Kaggle

thousands of datapoints for each of a thousand classes

Approach I: feature extraction



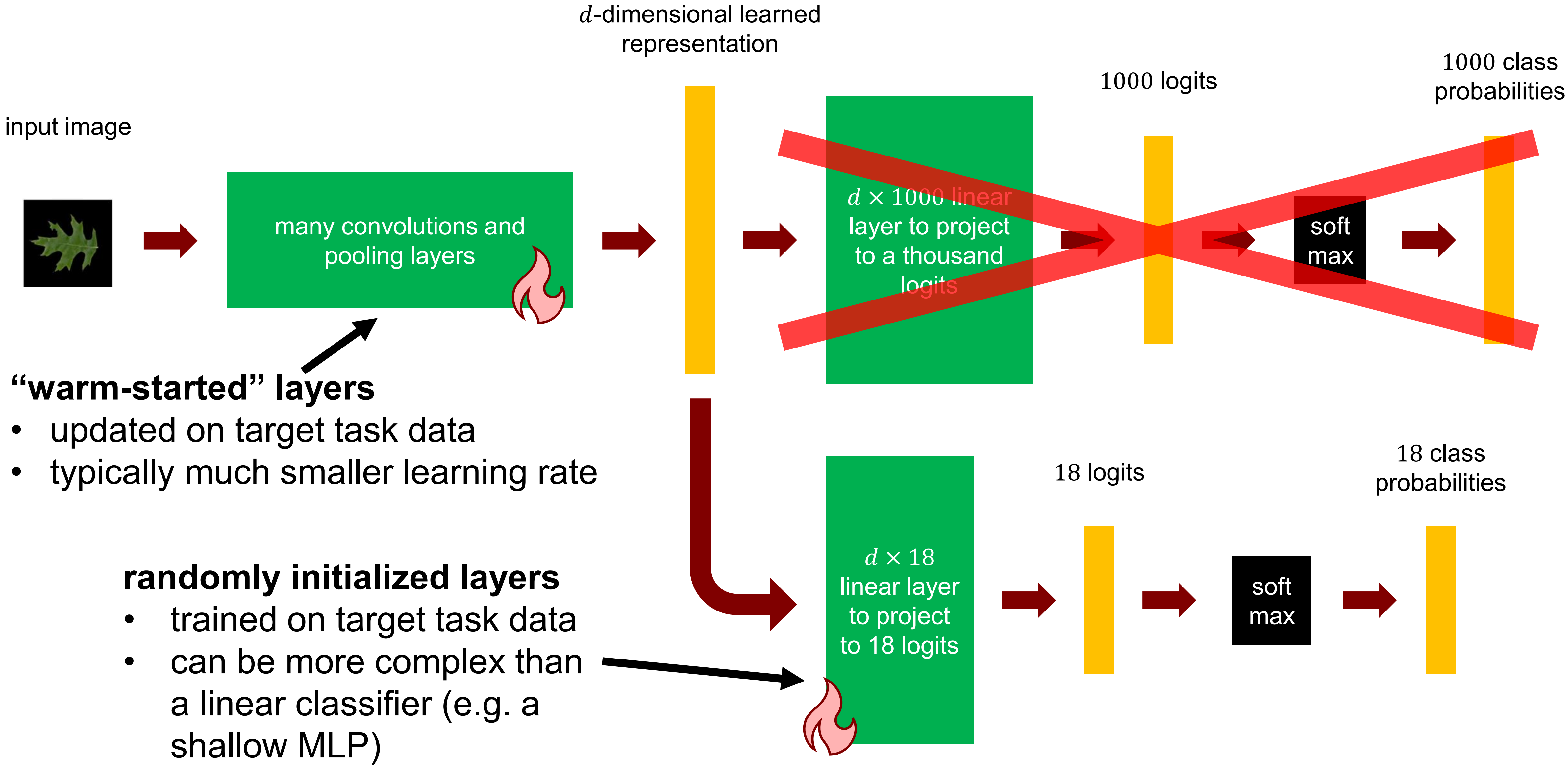
“frozen” layers

- not updated on target task data
- used only to extract features

randomly initialized layers

- trained on target task data
- can be more complex than a linear classifier (e.g. a shallow MLP)

Approach II: fine-tuning



"warm-started" layers
- updated on target task data
- typically much smaller learning rate

randomly initialized layers
- trained on target task data
- can be more complex than a linear classifier (e.g. a shallow MLP)

Transfer learning

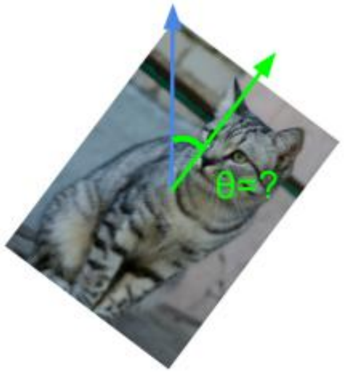
- Transfer learning has been hugely successful
- Numerous other potential approaches
- Big remaining question: **what if the related data lacks labels?**
  - we chop off the classification layers anyway, so we just need to extract some **representation** of the data
  - can do so using classical unsupervised learning (PCA, etc.)
  - or we can do it with **self-supervised learning (SSL)**

Self Supervision: Basic Idea

- Use domain-specific properties of the inputs (x) to create pseudo-labels (y) corresponding to **“pretext tasks”**
- Ex: predict stuff you already know

image completion

Stanford CS 231n



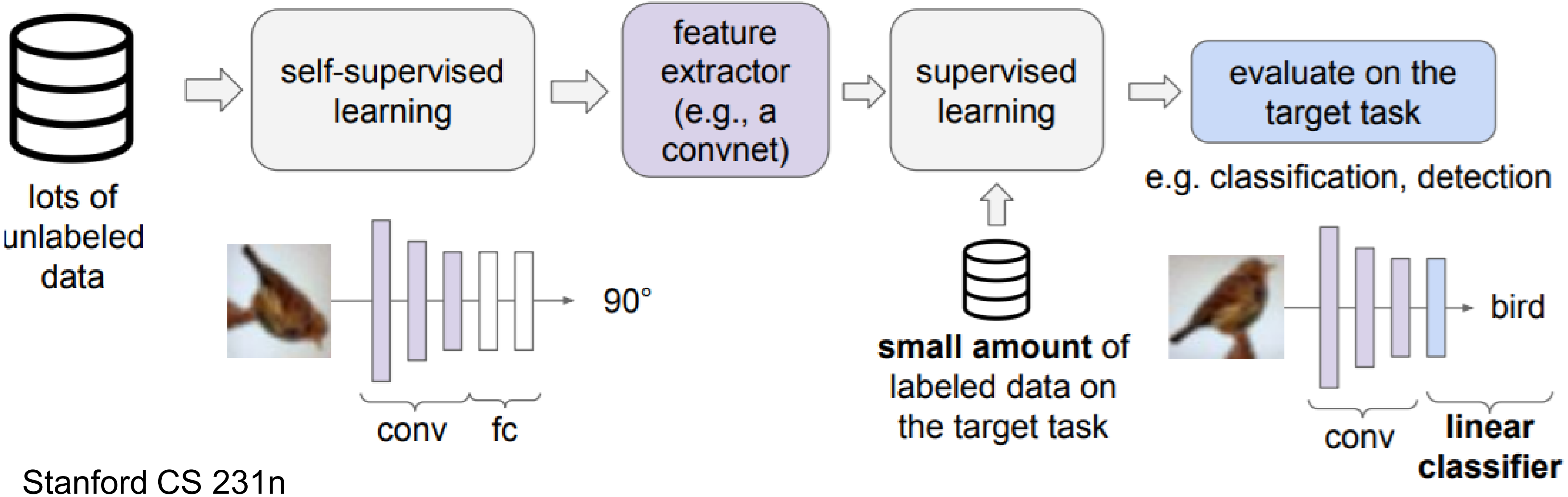
rotation prediction

“jigsaw puzzle”

colorization

Self Supervision: Using the Representations

- Don’t care specifically about our performance on pretext task
- Use the learned network as a feature extractor
- Once we have labels for a particular task, train on a small amount of data



Stanford CS 231n

Self Supervision: Pretext Tasks

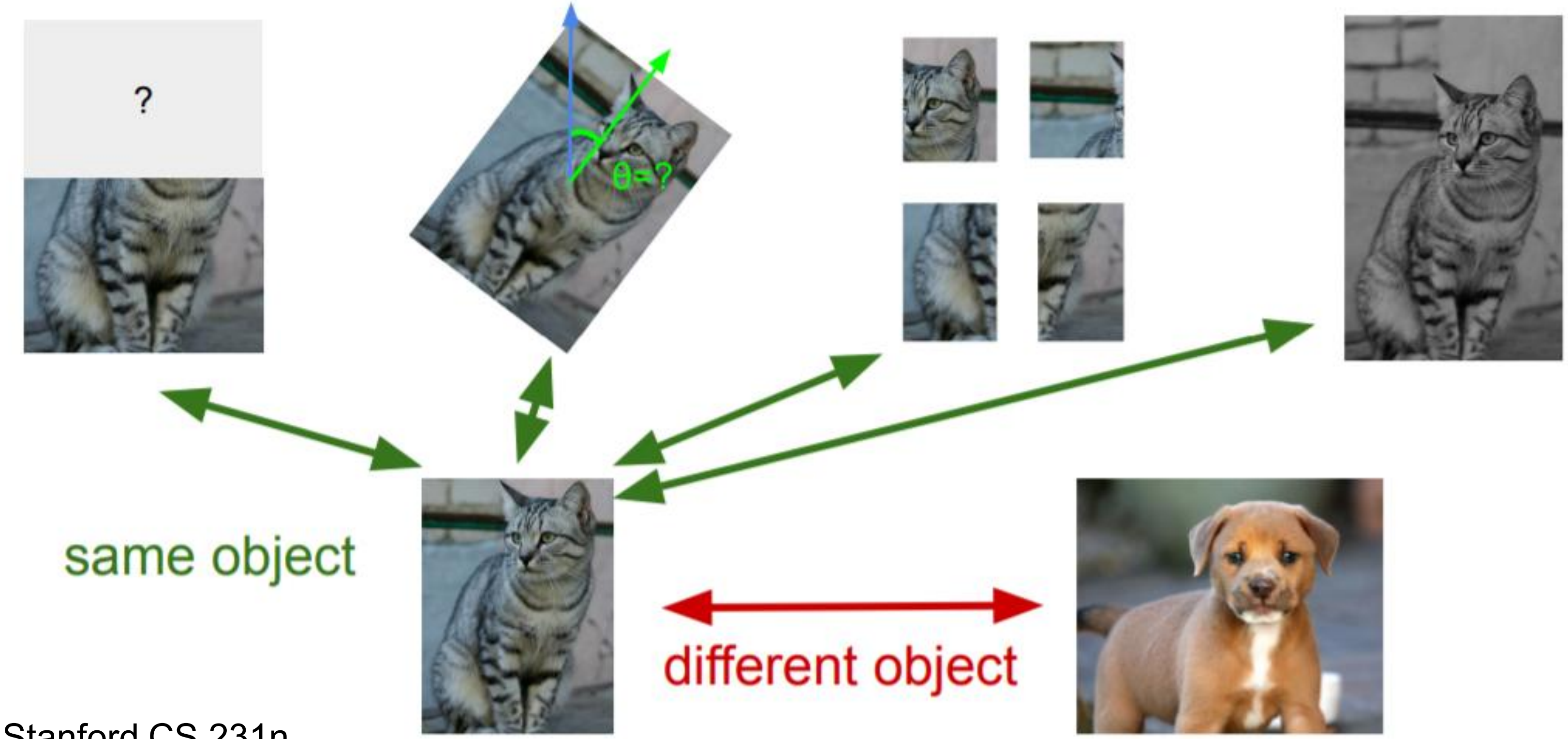
- Lots of options for pretext tasks
  - Predict rotations
  - Coloring
  - Fill in missing portions of the image
  - Solve puzzles

(a) (b) (c)

Noroozi and Favaro

Self Supervision: Contrastive Learning

- Type of SSL where we learn representations such that:
  - transformed versions of single sample are similar
  - different samples are different



Stanford CS 231n

Self-supervised learning: Summary

Procedure:

- **pretrain** a network to do well on a pretext task
- **transfer** the network to your target task

ChatGPT

Most well-known example: predict-the-next-word

Transfer learning from **multiple tasks**

What if instead of one related task with lots of data we have **many related tasks with similar amounts of data?**

$$(x_{1,1}, y_{1,1}), \ldots, (x_{1,n_1}, y_{1,n_1}) \sim D_1$$

$$\vdots$$

$$(x_{t,1}, y_{t,1}), \ldots, (x_{t,n_t}, y_{t,n_t}) \sim D_t$$

$$\vdots$$

Many setups:

- multi-task learning
- meta-learning
- continual learning
- lifelong learning
- …

We'll cover two of them: **multi-task** and **meta**-learning

Multi-task learning

Setup: **fixed number of related tasks**

Examples:

- predict the weather in nearby cities
- diagnose patients in different hospitals

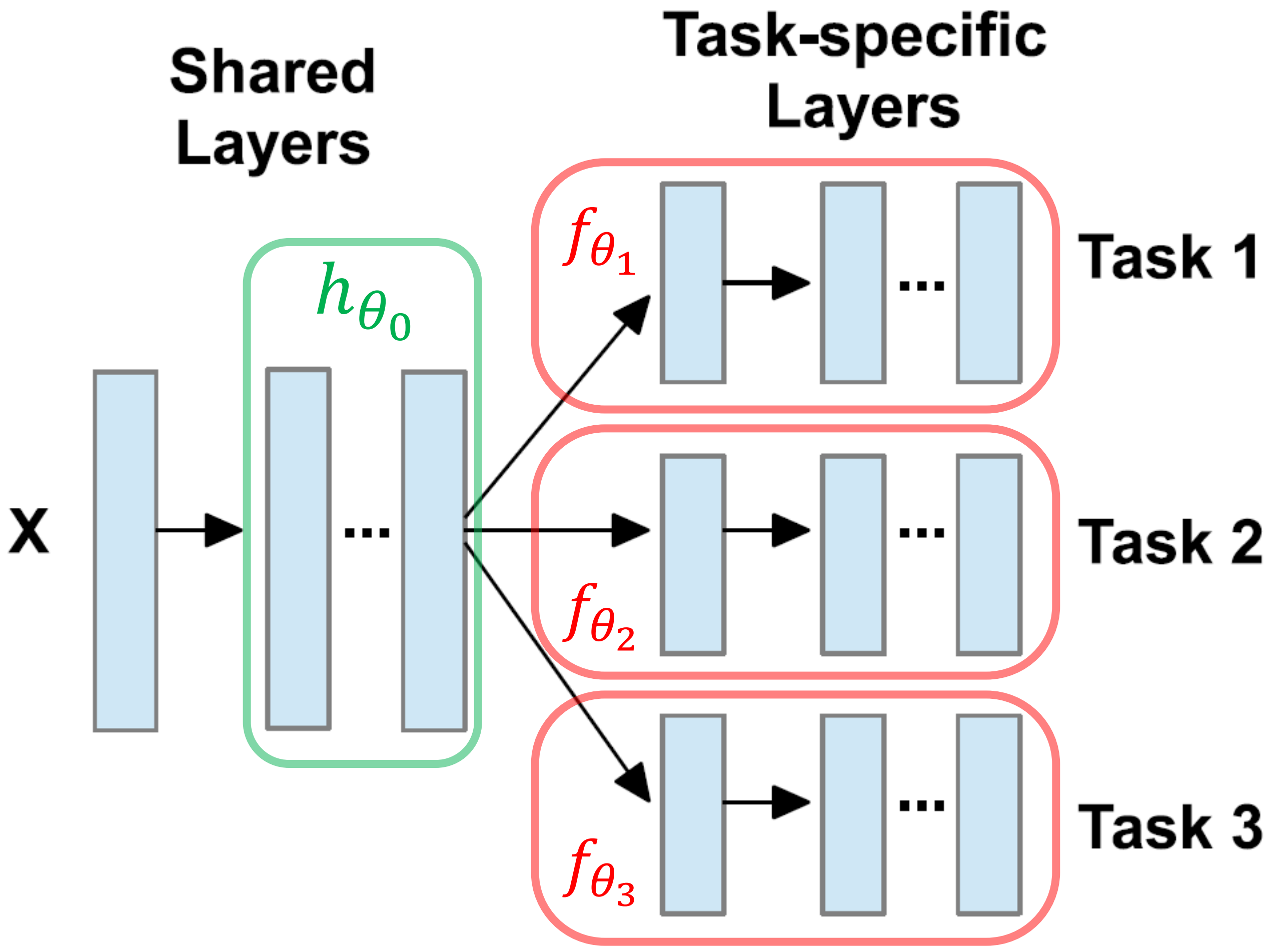
Key challenges:

- how to encode task-relationships?
- how to avoid conflicting tasks?

One common approach: **Layer-sharing**

- jointly train a multi-output network
- assumes existence of a good **shared representation** h_{θ_0}
- example objective:

$$\sum_{t=1}^{T}\sum_{i=1}^{n_t}\left(y_{t,i} - f_{\theta_t}\left(h_{\theta_0}(x_{t,i})\right)\right)^2$$



Thung & Wee. *Multimedia Tools & Applications*

Another common approach: **Regularization**

- jointly train separate networks
- regularize parameters to be closer together
- example objective:

$$\sum_{t=1}^{T}\sum_{i=1}^{n_t}\left(y_{t,i} - f_{\theta_t}(x_{t,i})\right)^2 + \sum_{t=1}^{T}\sum_{u=t+1}^{T}\lambda_{t,u}\|\theta_t - \theta_u\|^2$$

- allows hand-encoding of task-relationships via the regularization strengths $\lambda_{t,u}$

Meta-learning

$$(x_{1,1}, y_{1,1}), \dots, (x_{1,n_1}, y_{1,n_1}) \sim D_1$$

$$\vdots$$

$$(x_{T,1}, y_{T,1}), \dots, (x_{T,n_T}, y_{T,n_T}) \sim D_T$$

Setup:

- **meta-training** dataset of related tasks
- **at meta-test time** we get a new dataset $(x_1, y_1), \dots, (x_n, y_n) \sim D$
- **our goal:** low expected error on unseen examples $(x, y) \sim D$

Applications:

- auto-complete for new cellphone users (federated learning)
- image classification with limited labels (few-shot learning)
- robots in related environments (meta-RL)

What is a foundation model?

1. take a **massive neural network**
   - older / specialized models had 100M+ params
   - latest models have 1-100 billion or more

2. **pretrain** it on Internet-scale data

3. (optionally) **post-train** on large-scaled supervised data

4. use it for transfer learning for many different tasks

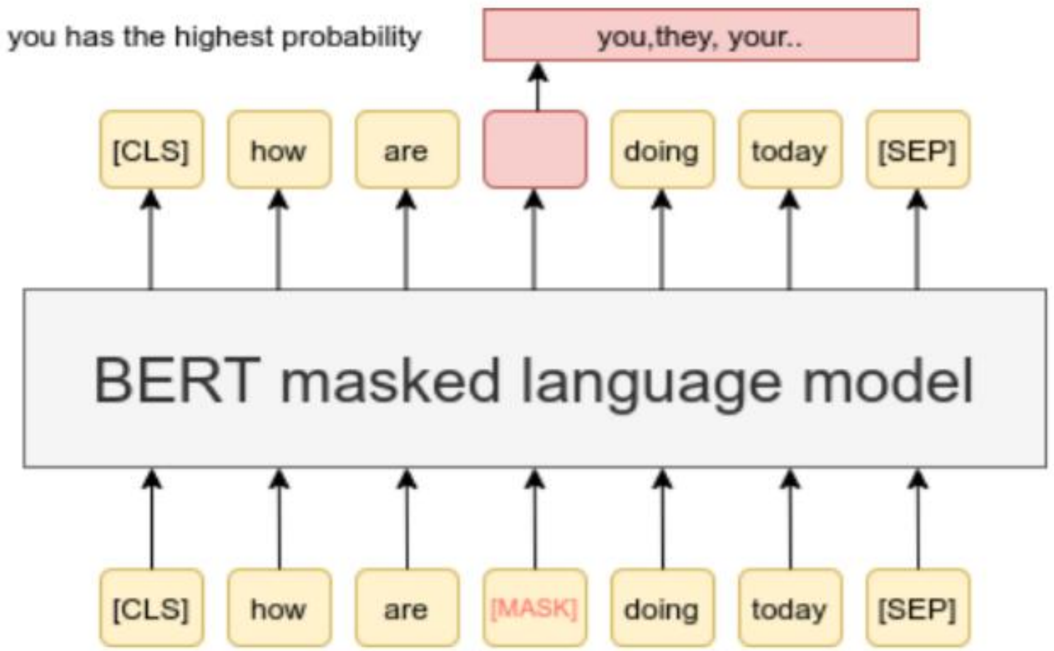
Early history

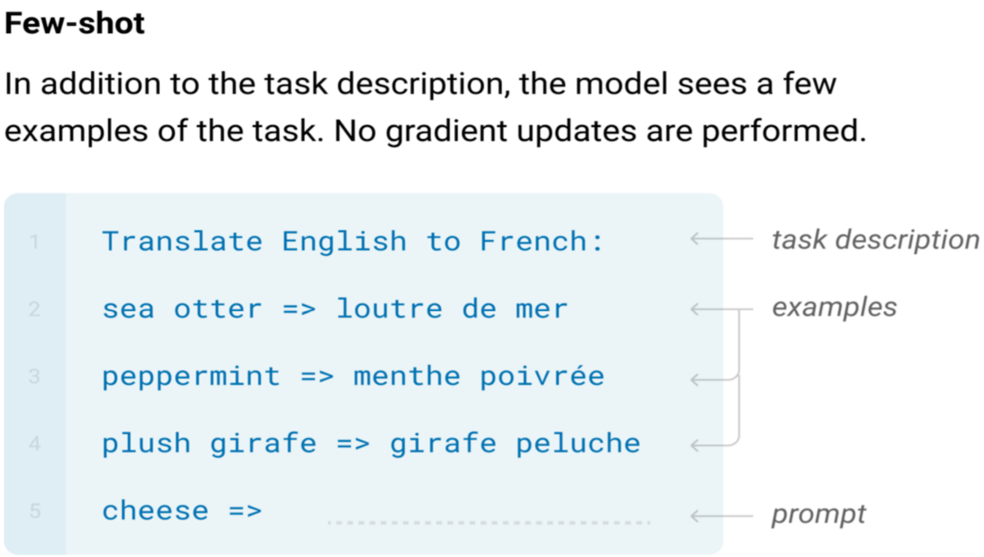
2017: BERT model (340M)

- Transformer trained on masked language modeling (pretext task)
- “solved” transfer learning for language

2017-present: GPT series

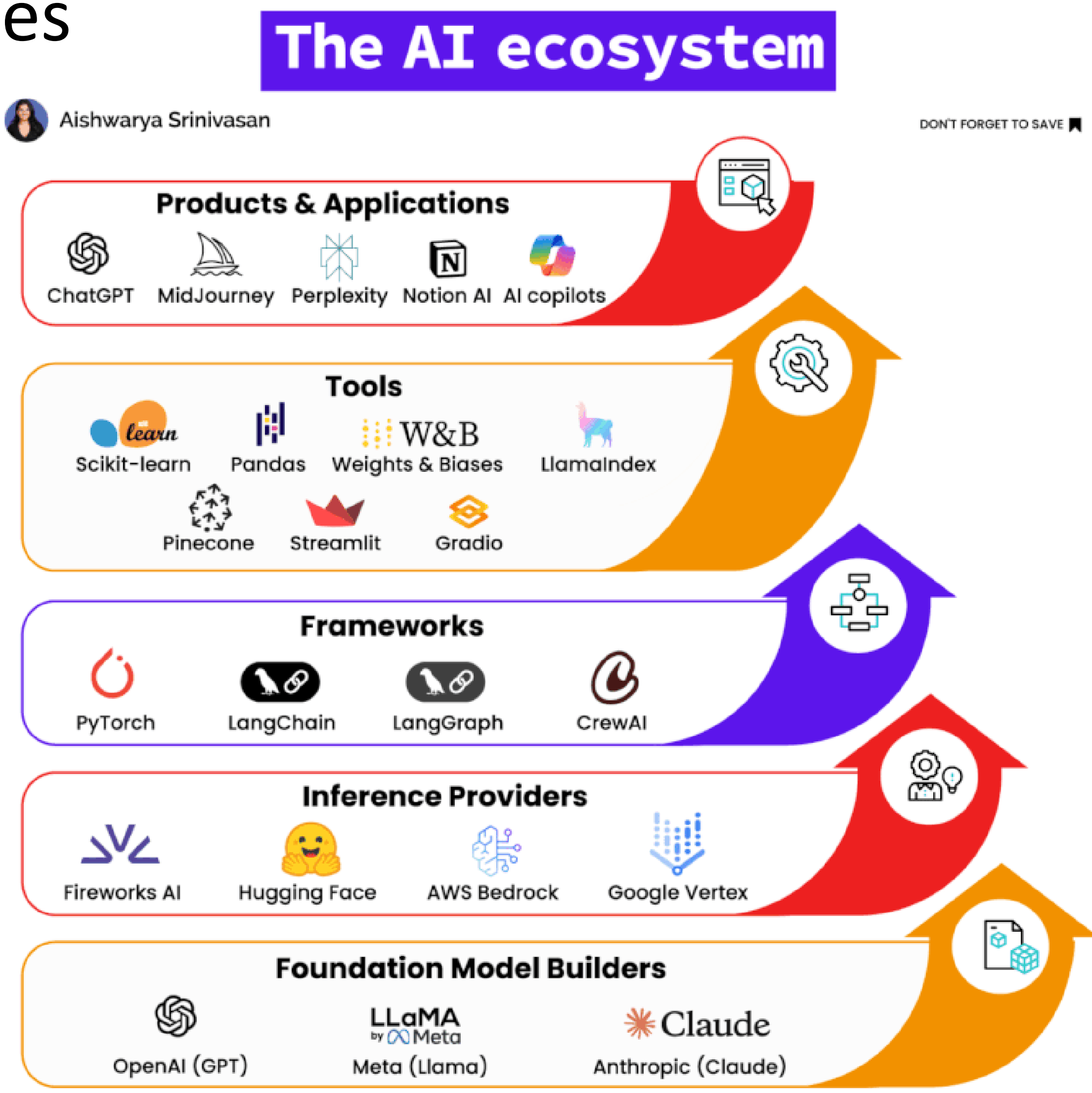
- Transformer trained on next-word prediction
- first observation of **in-context** learning capabilities in GPT-3
- ChatGPT post-trained on GPT-3.5





Post-ChatGPT

- many models with varying capabilities
- closed-source models typically outperform open-source models
- new challenges:
  - **massive compute costs**
  - privacy, security, safety
- new opportunities:
  - **in-context learning**
  - reasoning



Challenge: **Compute costs**

pretraining FMs limited to large orgs

- one training run requires 100s of GPUs
- need many training runs (to tune) and engineers (to manage training)

even fine-tuning is hard:

- SGD on GPT-3 (175B) uses 1.2TB VRAM
- NVIDIA GPUs max out below 200GB
- what can we do?

Model	Microarchitecture	Launch	Core	Core clock (MHz)	Shaders			Memory				
					Core config [c]	Base clock (MHz)	Max boost clock (MHz)[d]	Bus type	Bus width (bit)	Size (GB)	Clock MT/s)	Bandwidth (GB/s)
A100 GPU accelerator (PCIe card)[442][443]		May 14, 2020[444]	1× GA100-883AA-A1	—	6912:432 :160:432:0 (108)	765	1410	HBM2	5120	40 or 80	1215	1555
H100 GPU accelerator (PCIe card)[445]	Hopper	March 22, 2022[446]	1× GH100[447]	—	14592:456 :24:456:0 (114)	1065	1755 CUDA 1620 TC	HBM2E	5120	80	1000	2039
H100 GPU accelerator (SXM card)				—	16896:528 :24:528:0 (132)	1065	1980 CUDA 1830 TC	HBM3	5120	64 or 80 or 96	1500	3352
H200 GPU accelerator (PCIe card)[448]		November 18, 2024[449]	1× GH100	—		1365	1785	HBM3E	5120	141	1313	3360
H200 GPU accelerator (SXM card)				—		1590	1980	HBM3E	5120	141	1313	3360
H800 GPU accelerator (SXM card)		March 21, 2023[450]	1× GH100	—		1095	1755	HBM3	5120	80	1313	3360
L40 GPU accelerator[451]	Ada Lovelace	October 13, 2022	1× AD102[452]	—	18176:568 :192:568:142 (142)	735	2490	GDDR6	384	48	2250	864
L4 GPU accelerator[453][454]		March 21, 2023[455]	1x AD104[456]	—	7424:240 :80:240:0 (60)	795	2040	GDDR6	192	24	1563	300
B100 GPU accelerator[457]	Blackwell	November 2024	2× GB102	—	2× 16896:528 :24:528:0 (132)	1665	1837	HBM3E	2× 4096	2× 96	2000	2× 4100
B200 GPU accelerator[459]		2024	2× GB100	—		1665	1837	HBM3E	2× 4096	2× 96	2000	2× 4100

Parameter-efficient fine-tuning (PEFT)

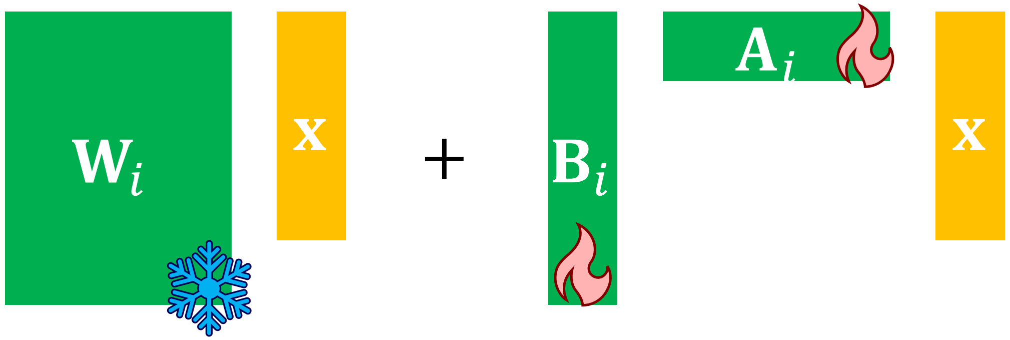
Most popular approach: **LoRA**

1. take an FM with **pretrained weight** matrices $\mathbf{W}_1, \dots, \mathbf{W}_N$
2. for each matrix $\mathbf{W}_i \in \mathbb{R}^{d \times k}$:
   - set $r \ll \min\{d, k\}$ and initialize **fine-tuning weights**:
     - $\mathbf{B}_i \in \mathbb{R}^{d \times r}$ to $\mathbf{B}_i = 0$
     - $\mathbf{A}_i \in \mathbb{R}^{r \times k}$ to $\mathbf{A}_i \sim \text{Gaussian}$
   - replace $\mathbf{W}_i$ by $\mathbf{W}_i + \mathbf{B}_i\mathbf{A}_i$
3. fine-tune on target task but
   - freeze $\mathbf{W}_i$
   - update $\mathbf{B}_i$ and $\mathbf{A}_i$

$$f_i(\mathbf{x}) = \mathbf{W}_i\mathbf{x} =$$

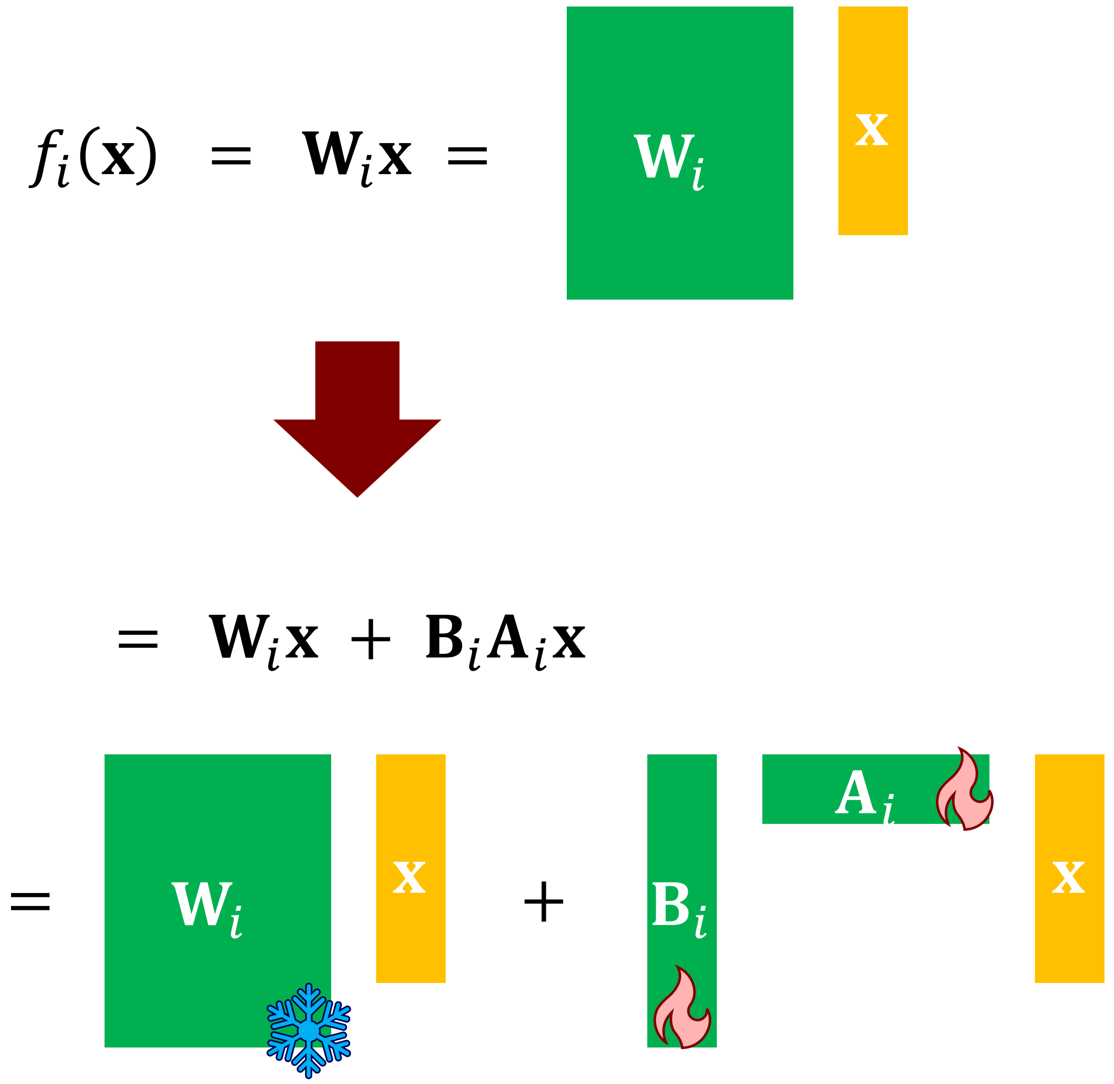
$$= \mathbf{W}_i\mathbf{x} + \mathbf{B}_i\mathbf{A}_i\mathbf{x}$$

$$=$$



How does LoRA save memory?

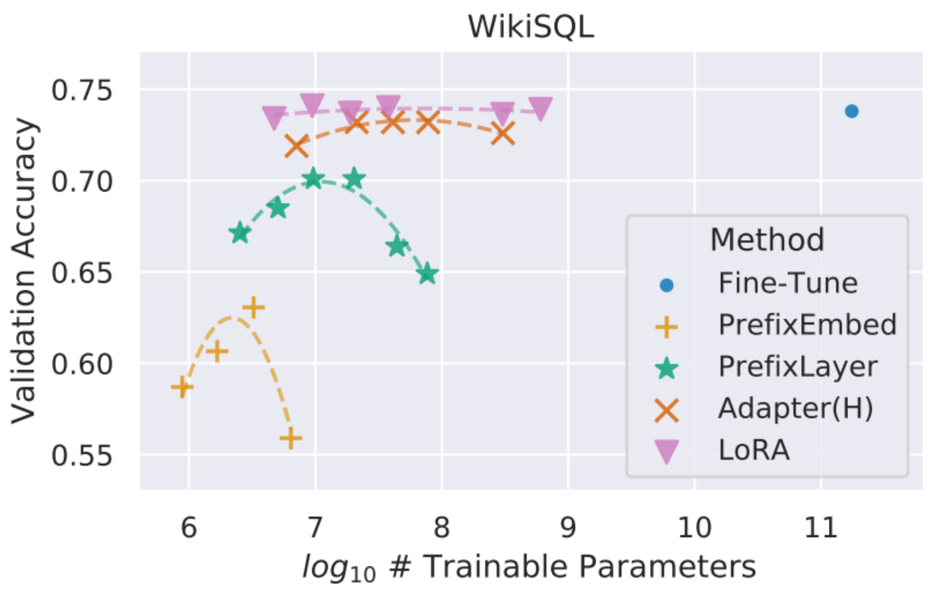
- original weights $\mathbf{W}_i \in \mathbb{R}^{d \times k}$ have dk trainable params
- new weights $\mathbf{B}_i \in \mathbb{R}^{d \times r}$ and $\mathbf{A}_i \in \mathbb{R}^{r \times k}$ have $(\mathrm{d} + \mathrm{k})r$
- typical values in GPT-3 175B:
  - $d \approx k \approx 10^4$
  - $r \leq 10$
- $\geq 10^4$x fewer trainable params!
- 3x less fine-tuning VRAM

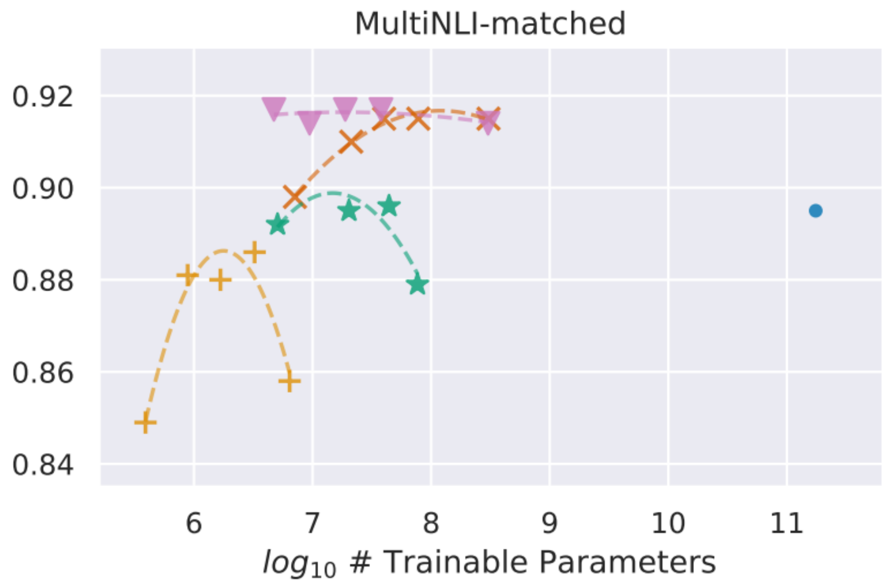


Does LoRA affect accuracy?

Yes, it constrains weights of the fine-tuned model:

- fine-tuned matrices $\mathbf{W}_i + \mathbf{B}_i\mathbf{A}_i$ at most a rank $r \ll \min\{d,k\}$ update away from pretrained matrices $\mathbf{W}_i$
- LoRA = **L**ow-**R**ank **A**daptation
- in practice do not need large r for good performance
- learning theory intuition?





Hu et al.

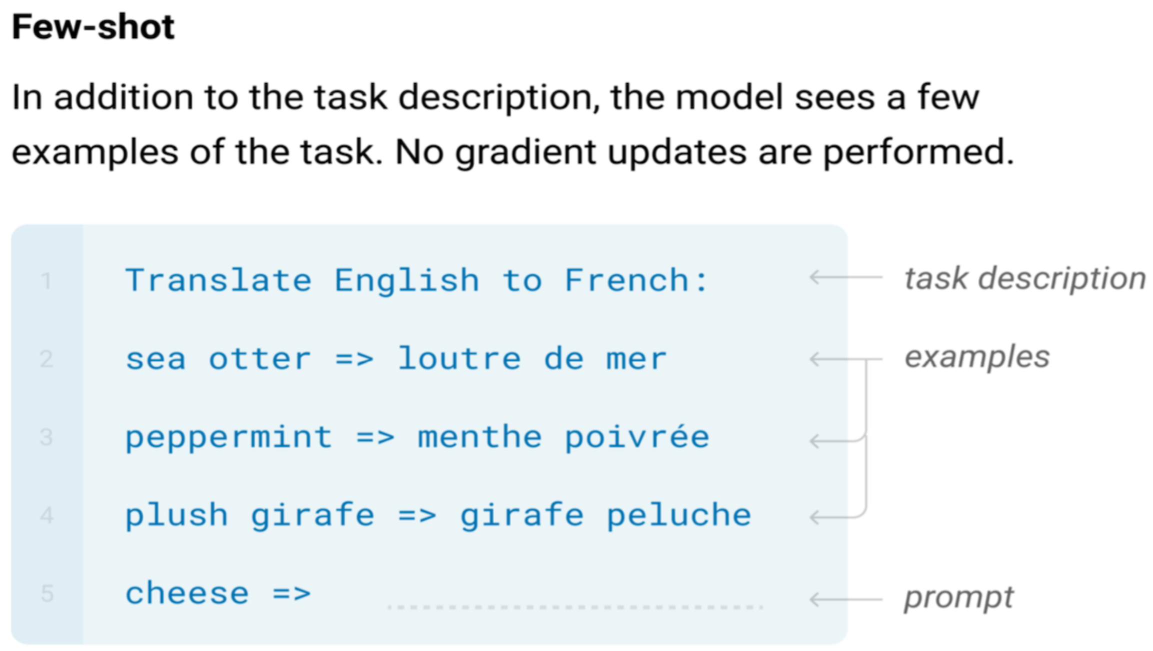
Opportunity: **In-context learning**

Observation: the perfect next-word predictor can be **prompted** to answer any question correctly

Idea: **in-context learning**

1. encode task instructions and data as a **context** sequence
2. make the FM generate the remainder of the sequence

Enables learning with target data **without updating the weights at all!**

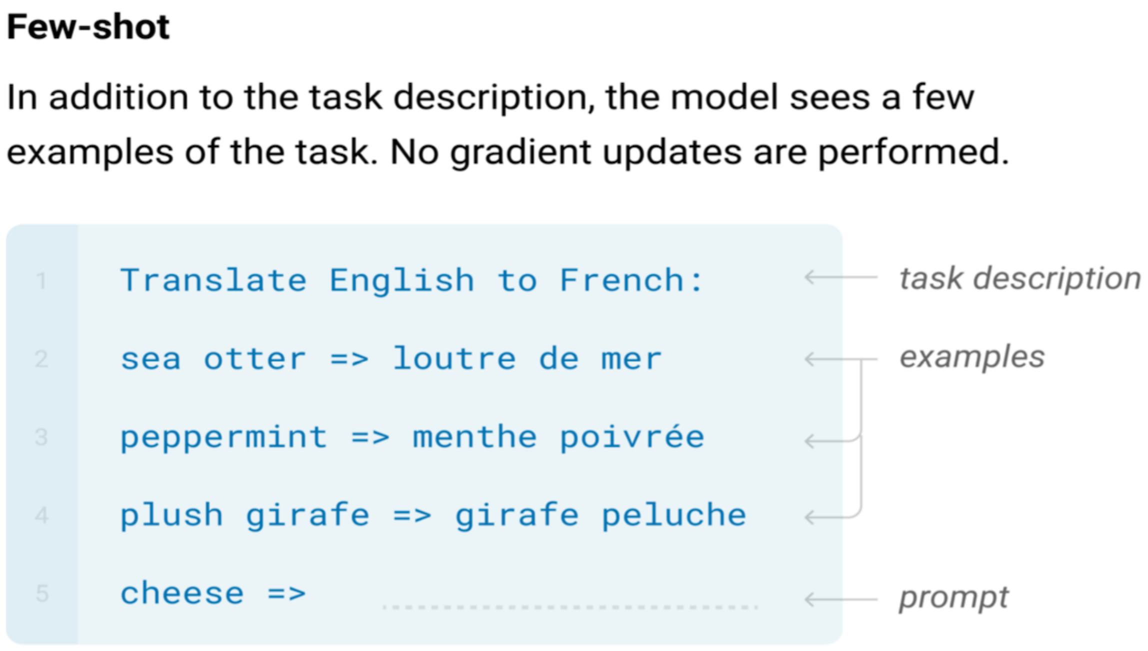


Brown et al.

Opportunity: **In-context learning**

Usefulness:

- handles tasks with diverse input and output structures
- directly incorporates pretraining knowledge
- enables multi-step reasoning



Brown et al.

Thanks Everyone!

Some of the slides in these lectures have been adapted/borrowed from materials developed by Mark Craven, David Page, Jude Shavlik, Tom Mitchell, Nina Balcan, Elad Hazan, Tom Dietterich, Pedro Domingos, Jerry Zhu, Yingyu Liang, Volodymyr Kuleshov, Fred Sala, Kirthi Kandasamy, Josiah Hanna, Tengyang Xie, Fei-Fei Li, Justin Johnson, Serena Yeung, Pieter Abbeel, Peter Chen, Jonathan Ho, Aravind Srinivas, Ruiqi Gao