

10-701: Introduction to Machine Learning

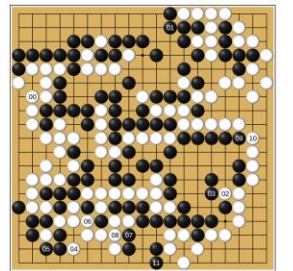
Lecture 18 – Pretraining, Fine-tuning & In-Context Learning

Hoda Heidari

* Slides adopted from F24 offering of 10701 by Henry Chai.

Two big Q's

1. What can we do if the reward and/or transition functions/distributions are unknown?
 - Use online learning to gather data and learn $Q^*(s, a)$
2. How can we handle infinite (or just very large) state/action spaces?
 - Throw a neural network at it!

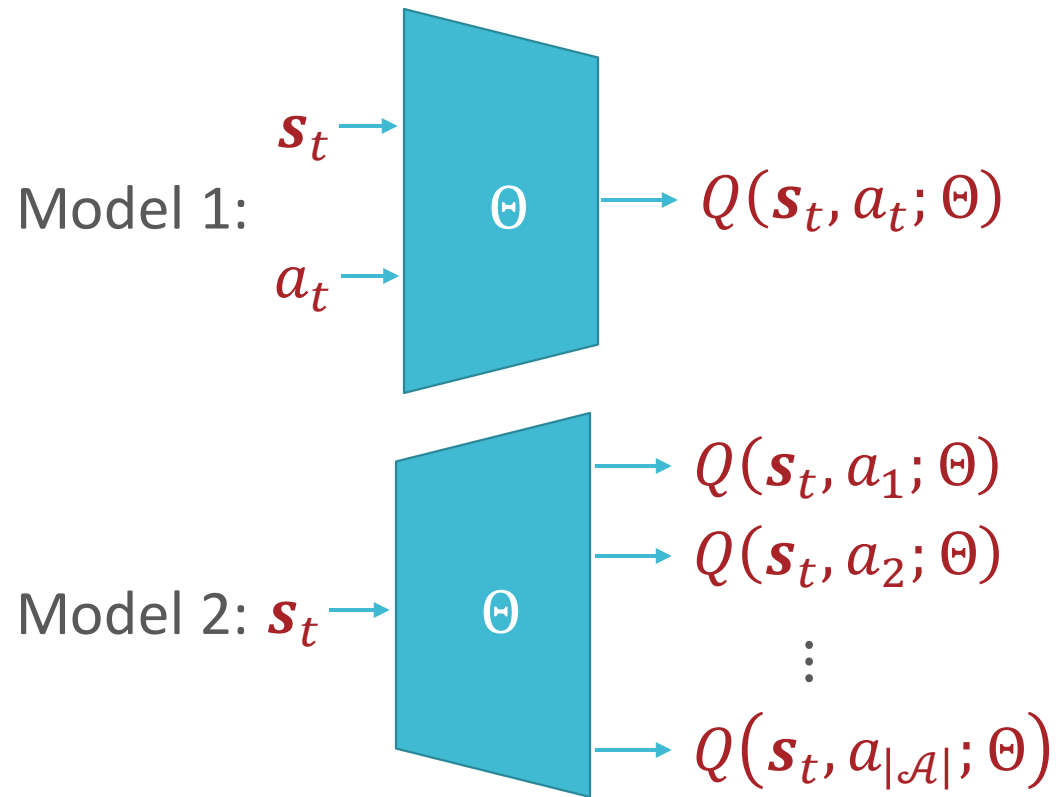


Deep Q-learning

- Use a parametric function, $Q(s, a; \Theta)$, to approximate $Q^*(s, a)$
 - Learn the parameters using *stochastic* gradient descent (SGD)
 - Training data $(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1})$ gathered online by the agent/learning algorithm

Deep Q-learning: Model

- Represent states using some feature vector $\mathbf{s}_t \in \mathbb{R}^M$
e.g. for Go, $\mathbf{s}_t = [1, 0, -1, \dots, 1]^T$
- Define a *differentiable* function that approximates Q



Deep Q-learning: Loss Function

- “True” loss

$$\ell(\Theta) = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \underbrace{\left(\overbrace{Q^*(s, a)}^{2. \text{ Don't know } Q^*} - Q(s, a; \Theta) \right)^2}$$

1. \mathcal{S} too big to compute this sum

1. Use stochastic gradient descent: just consider one state-action pair in each iteration

2. Use temporal difference learning:

- Given current parameters $\Theta^{(t)}$ the temporal difference target is

$$Q^*(s, a) \approx r + \gamma \max_{a'} Q(s', a'; \Theta^{(t)}) := y$$

- Set the parameters in the next iteration $\Theta^{(t+1)}$ such that $Q(s, a; \Theta^{(t+1)}) \approx y$

$$\ell(\Theta^{(t)}, \Theta^{(t+1)}) = \left(y - Q(s, a; \Theta^{(t+1)}) \right)^2$$

Deep Q-learning

Algorithm 4: Online learning (parametric form)

- Inputs: discount factor γ , an initial state s_0 ,
learning rate α
- Initialize parameters $\Theta^{(0)}$
- For $t = 0, 1, 2, \dots$
 - Gather training sample (s_t, a_t, r_t, s_{t+1})
 - Update $\Theta^{(t)}$ by taking a step opposite the gradient

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \alpha \nabla_{\Theta} \ell(\Theta^{(t)}, \Theta)$$

where

$$\nabla_{\Theta} \ell(\Theta^{(t)}, \Theta) = 2(y - Q(s, a; \Theta)) \nabla_{\Theta} Q(s, a; \Theta)$$

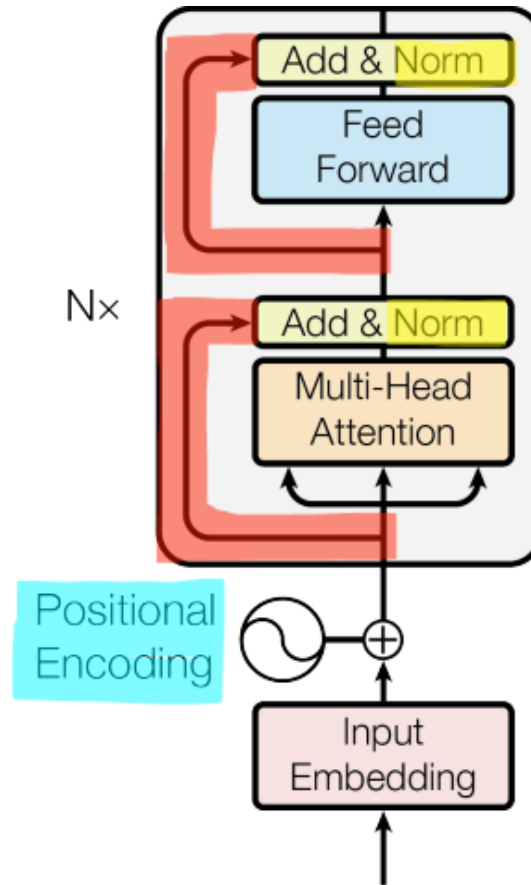
Deep Q-learning: Experience Replay

- SGD assumes i.i.d. training samples but in RL, samples are *highly* correlated
- Idea: keep a “replay memory” $\mathcal{D} = \{e_1, e_2, \dots, e_N\}$ of the N most recent experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ (Lin, 1992)
 - Also keeps the agent from “forgetting” about recent experiences
- Alternate between:
 1. Sampling some e_i uniformly at random from \mathcal{D} and applying a Q-learning update (repeat T times)
 2. Adding a new experience to \mathcal{D}
- Can also sample experiences from \mathcal{D} according to some distribution that prioritizes experiences with high error (Schaul et al., 2016)

Key Takeaways

- We can use (deep) Q-learning when the reward/transition functions are unknown and/or when the state/action spaces are too large to be modelled directly
 - Also guaranteed to converge under certain assumptions
 - Experience replay can help address non-i.i.d. samples

Okay, one massive detour later, how on earth do we go about training these things?



- In addition to multi-head attention, transformer architectures use
 1. Positional encodings
 2. Layer normalization
 3. Residual connections
 4. A fully-connected feed-forward network

Recall: Mini-batch Stochastic Gradient Descent...

- Input: training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$,
step size γ , and batch size B
 1. Randomly initialize the parameters $\boldsymbol{\theta}^{(0)}$ and set $t = 0$
 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from \mathcal{D} , $\{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient of the loss w.r.t. the sampled *batch*,
$$\nabla J^{(B)}(\boldsymbol{\theta}^{(t)})$$
 - c. Update $\boldsymbol{\theta}$: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \nabla J^{(B)}(\boldsymbol{\theta}^{(t)})$
 - d. Increment t : $t \leftarrow t + 1$
- Output: $\boldsymbol{\theta}^{(t)}$

Mini-batch Stochastic Gradient Descent is a lie!

- Input: training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$, step size γ , and batch size B
 1. Randomly initialize the parameters $\boldsymbol{\theta}^{(0)}$ and set $t = 0$
 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from \mathcal{D} , $\{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient of the loss w.r.t. the sampled batch,
 $\nabla J^{(B)}(\boldsymbol{\theta}^{(t)})$
 - c. Update $\boldsymbol{\theta}$: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \nabla J^{(B)}(\boldsymbol{\theta}^{(t)})$
 - d. Increment t : $t \leftarrow t + 1$
- Output: $\boldsymbol{\theta}^{(t)}$

Mini-batch Stochastic Gradient Descent is a lie! just the beginning!

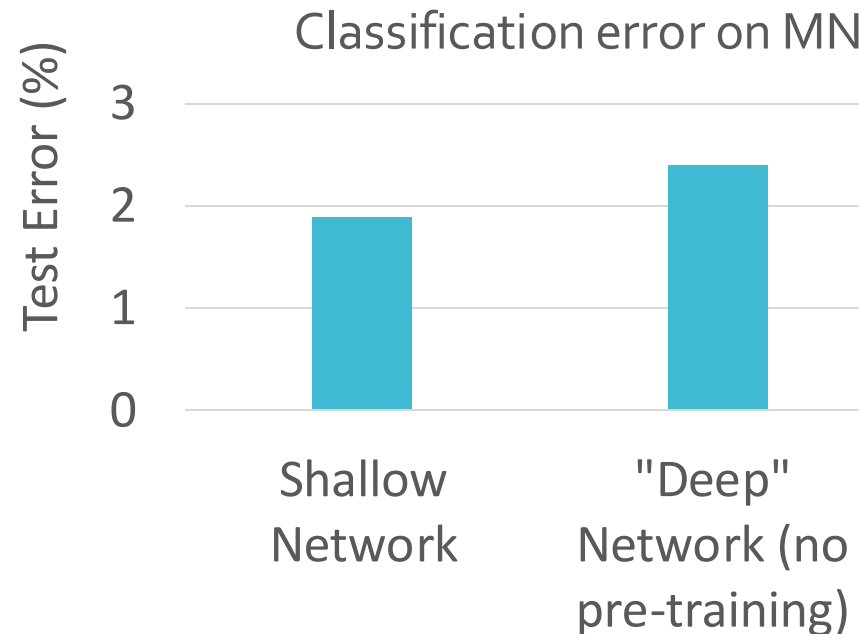
- Input: training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$, step size γ , and batch size B
 1. Randomly initialize the parameters $\boldsymbol{\theta}^{(0)}$ and set $t = 0$
 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from \mathcal{D} , $\{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient of the loss w.r.t. the sampled batch, $\nabla J^{(B)}(\boldsymbol{\theta}^{(t)})$
 - c. Update $\boldsymbol{\theta}$: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \nabla J^{(B)}(\boldsymbol{\theta}^{(t)})$
 - d. Increment t : $t \leftarrow t + 1$
- Output: $\boldsymbol{\theta}^{(t)}$

Traditional Supervised Learning

- You have some task that you want to apply machine learning to
- You have a labelled dataset to train with
- You fit a deep learning model to the dataset

Reality

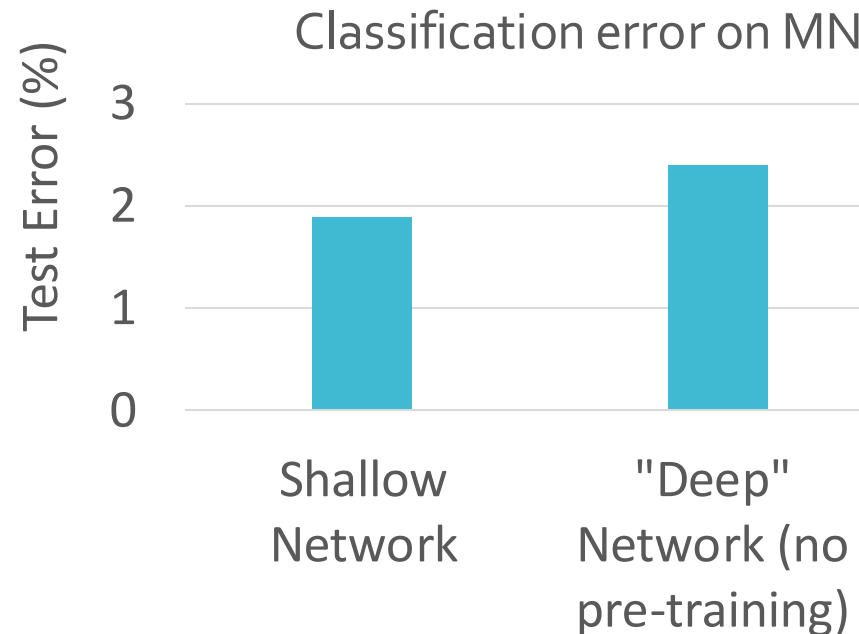
- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a tiny labelled dataset to train with
- You fit a massive deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high



- “gradient-based optimization starting from random initialization appears to often get stuck in poor solutions for such deep networks.”

Reality

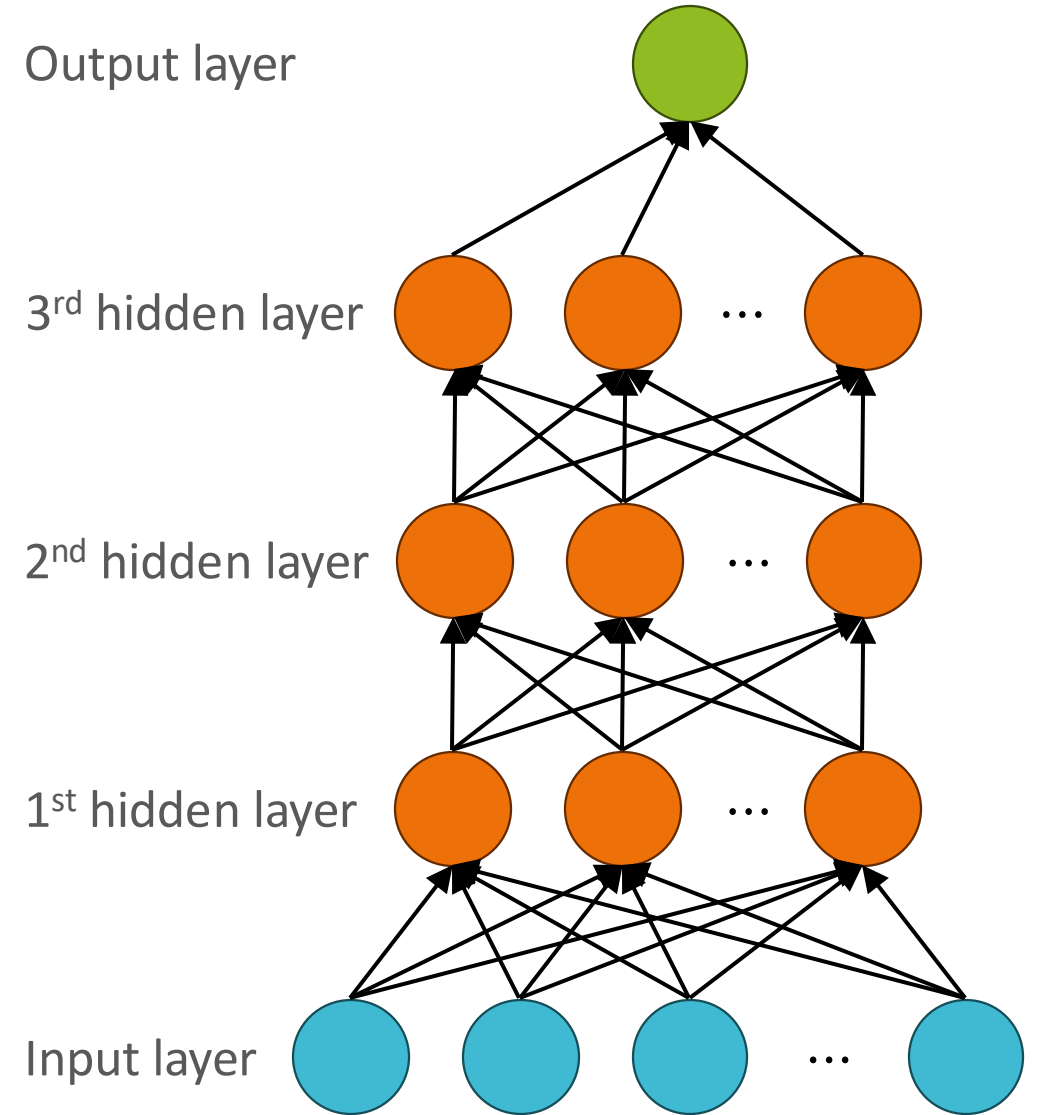
- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a tiny labelled dataset to train with
- You fit a massive deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high



- Idea: if shallow networks are easier to train, let's just decompose our deep network into a series of shallow networks!

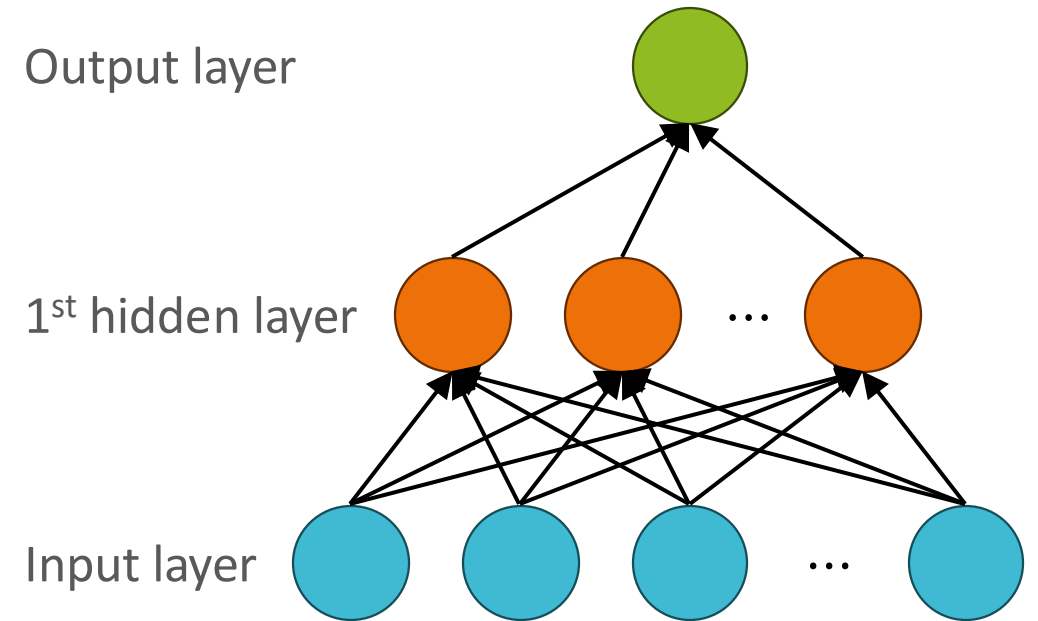
Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset
- Start at the input layer and move towards the output layer
- Once a layer has been trained, fix its weights and use those to train subsequent layers



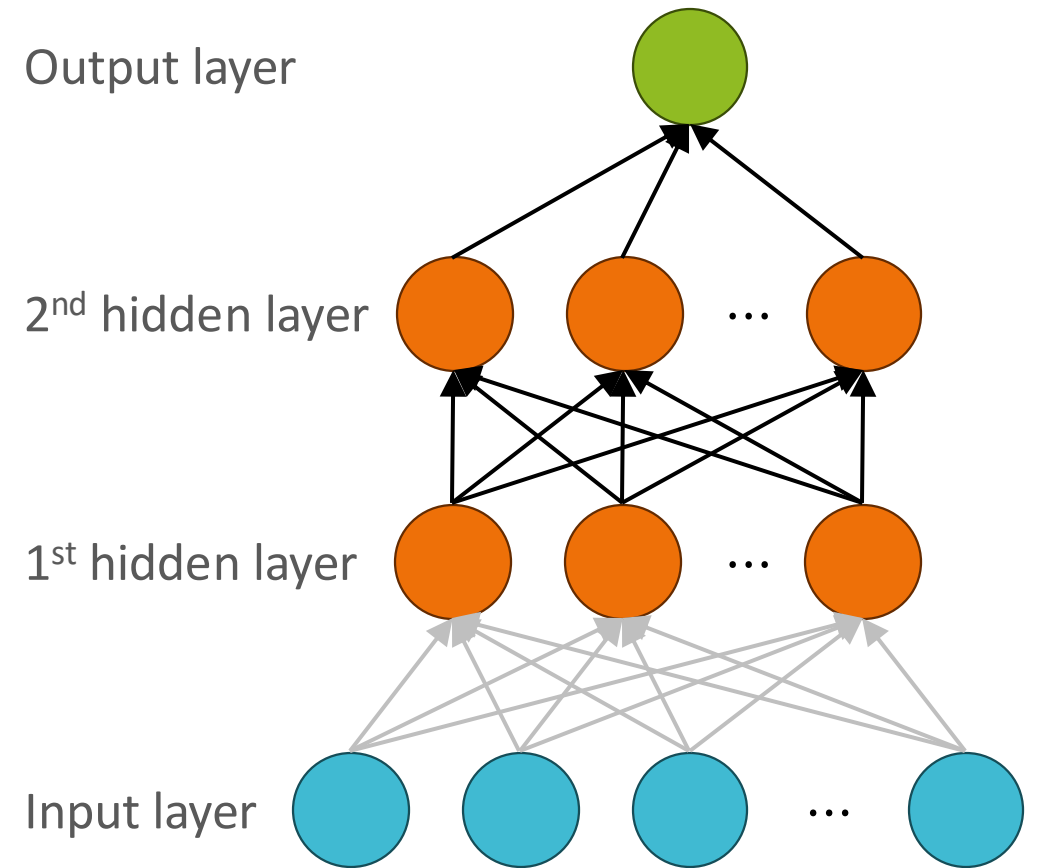
Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset
- Start at the input layer and move towards the output layer
- Once a layer has been trained, fix its weights and use those to train subsequent layers



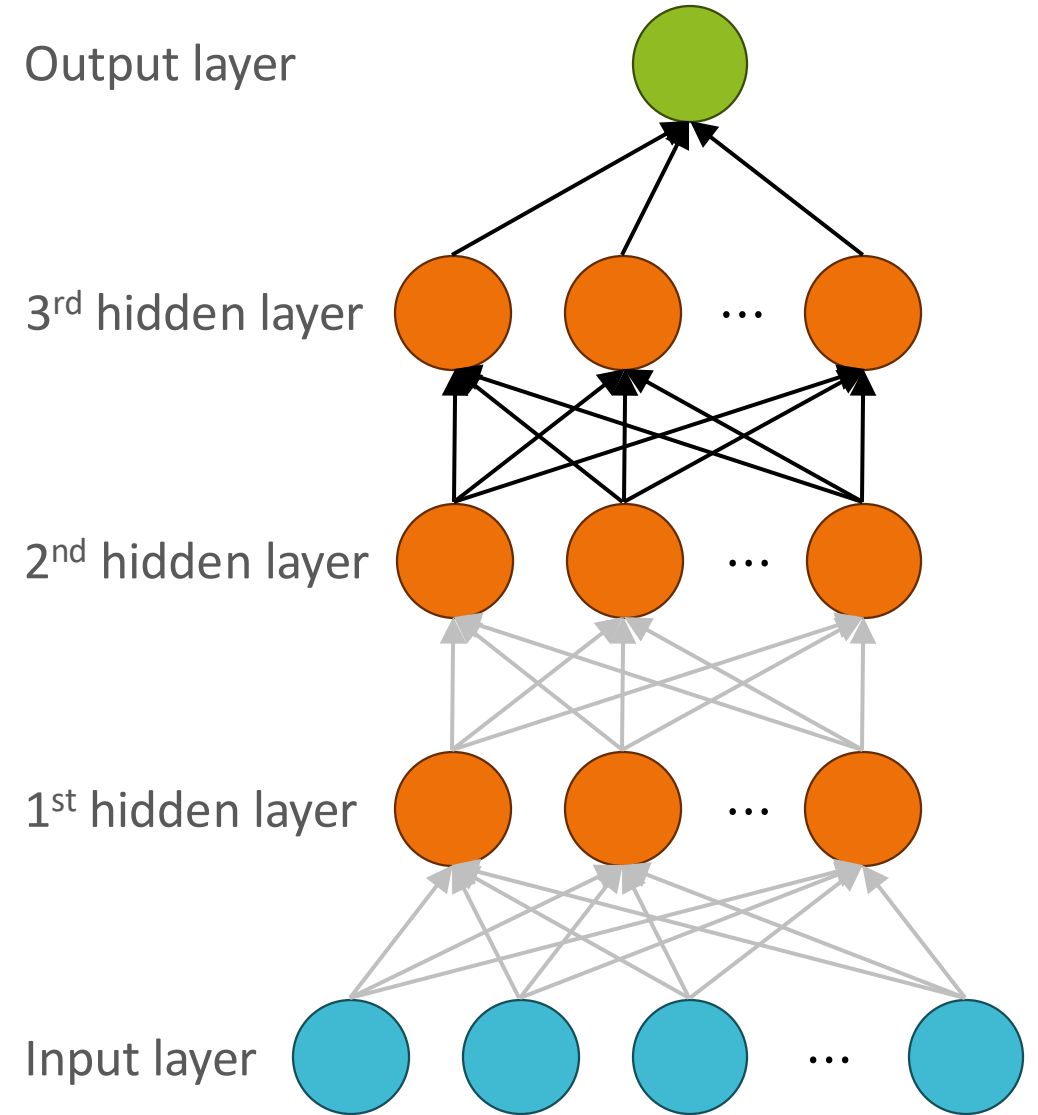
Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset
- Start at the input layer and move towards the output layer
- Once a layer has been trained, fix its weights and use those to train subsequent layers



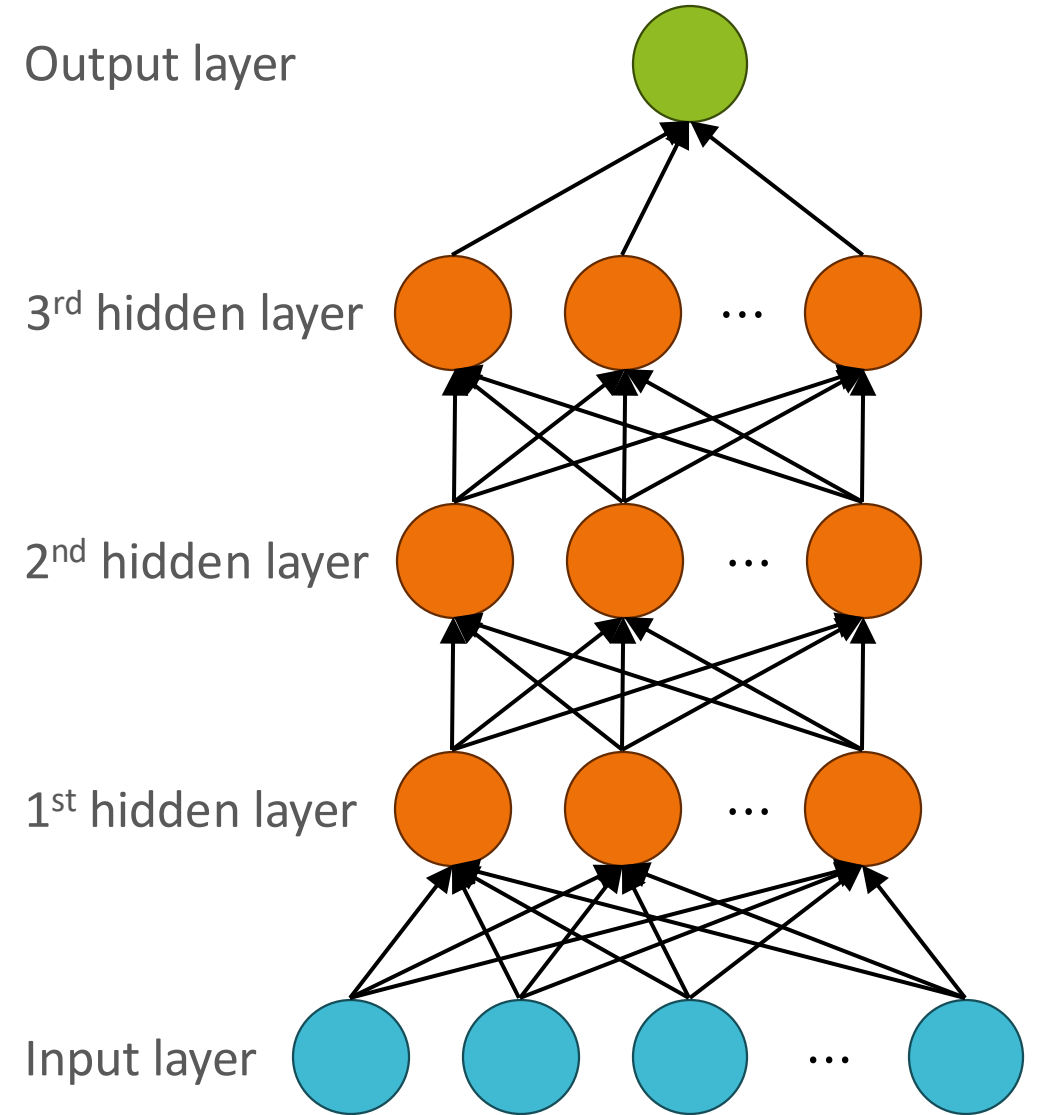
Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset
- Start at the input layer and move towards the output layer
- Once a layer has been trained, fix its weights and use those to train subsequent layers



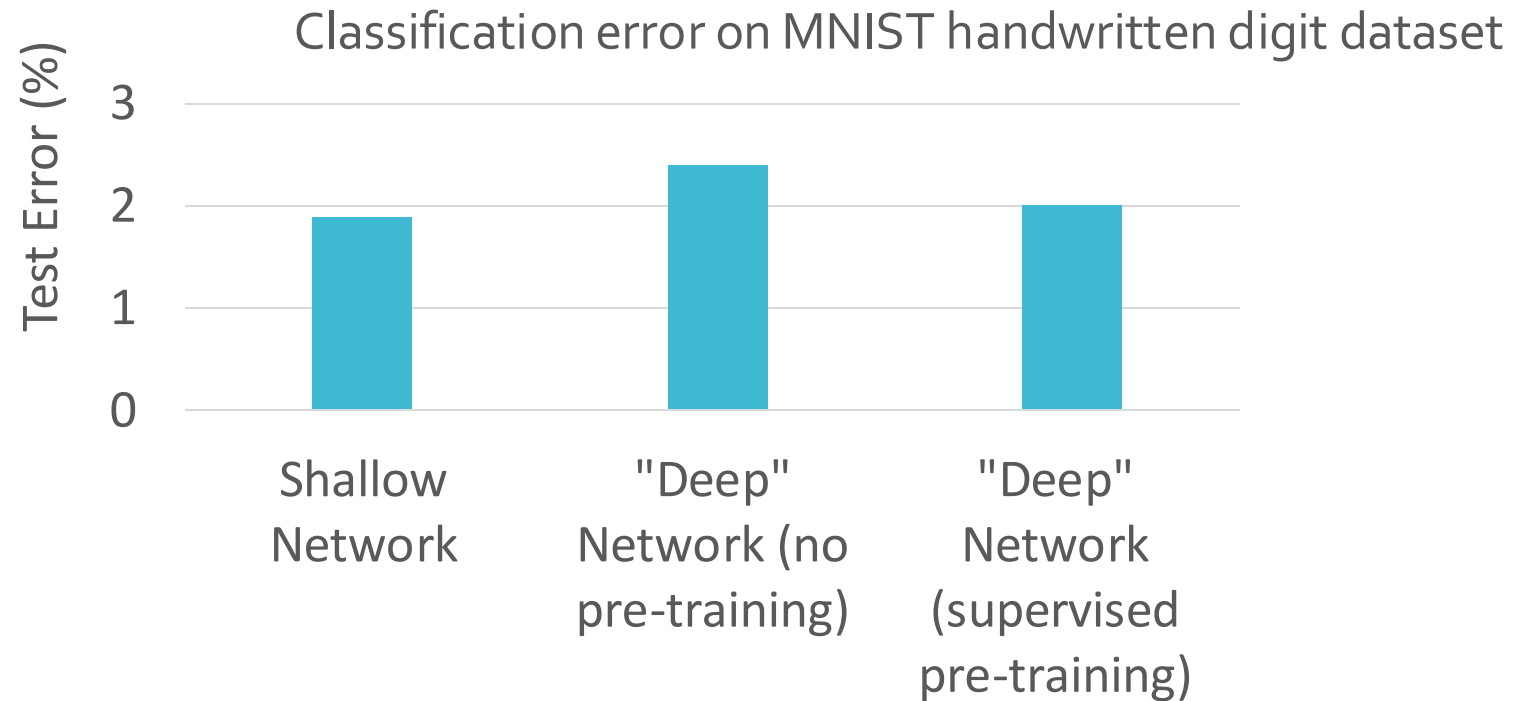
Fine-tuning (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset
- Use the pre-trained weights as an initialization and *fine-tune* the entire network e.g., via SGD with the training dataset



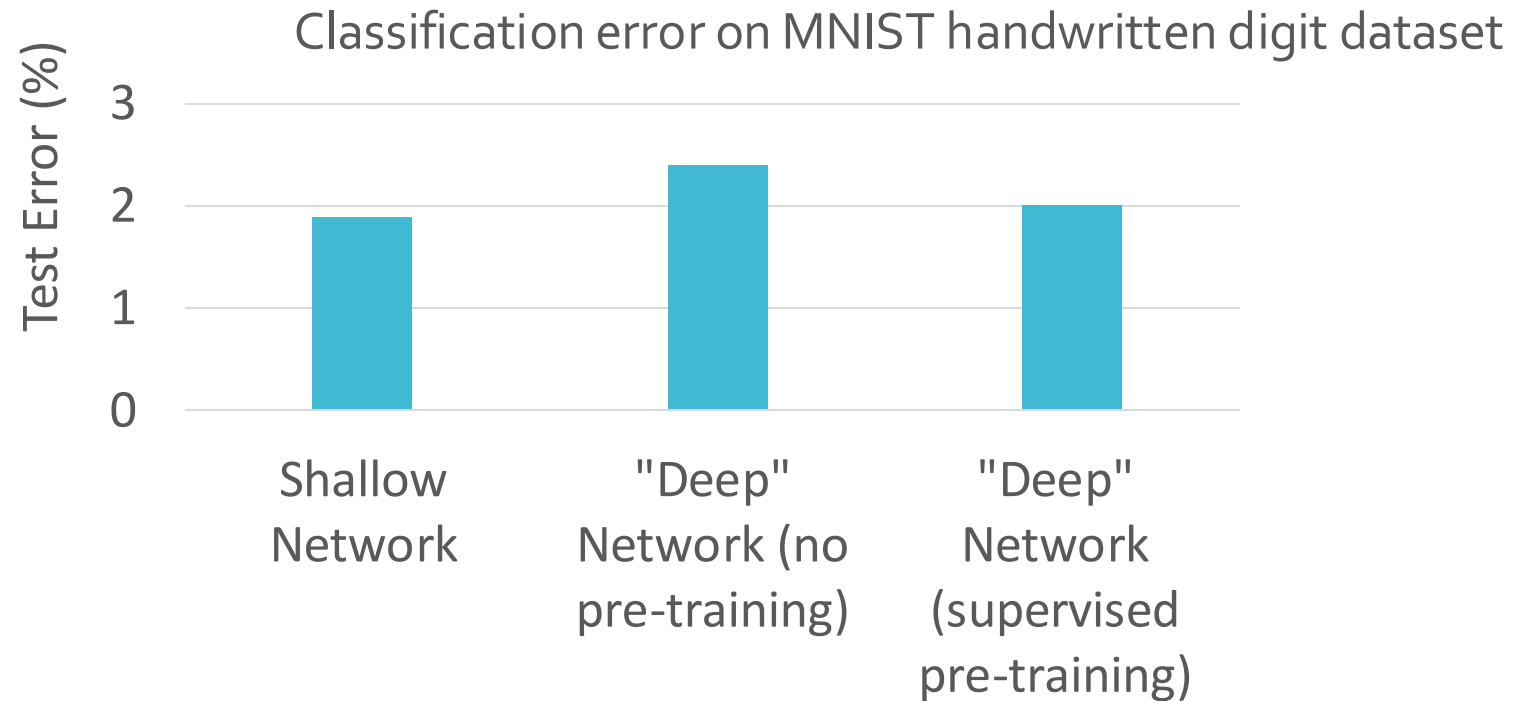
Supervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset
- Use the pre-trained weights as an initialization and *fine-tune* the entire network e.g., via SGD with the training dataset



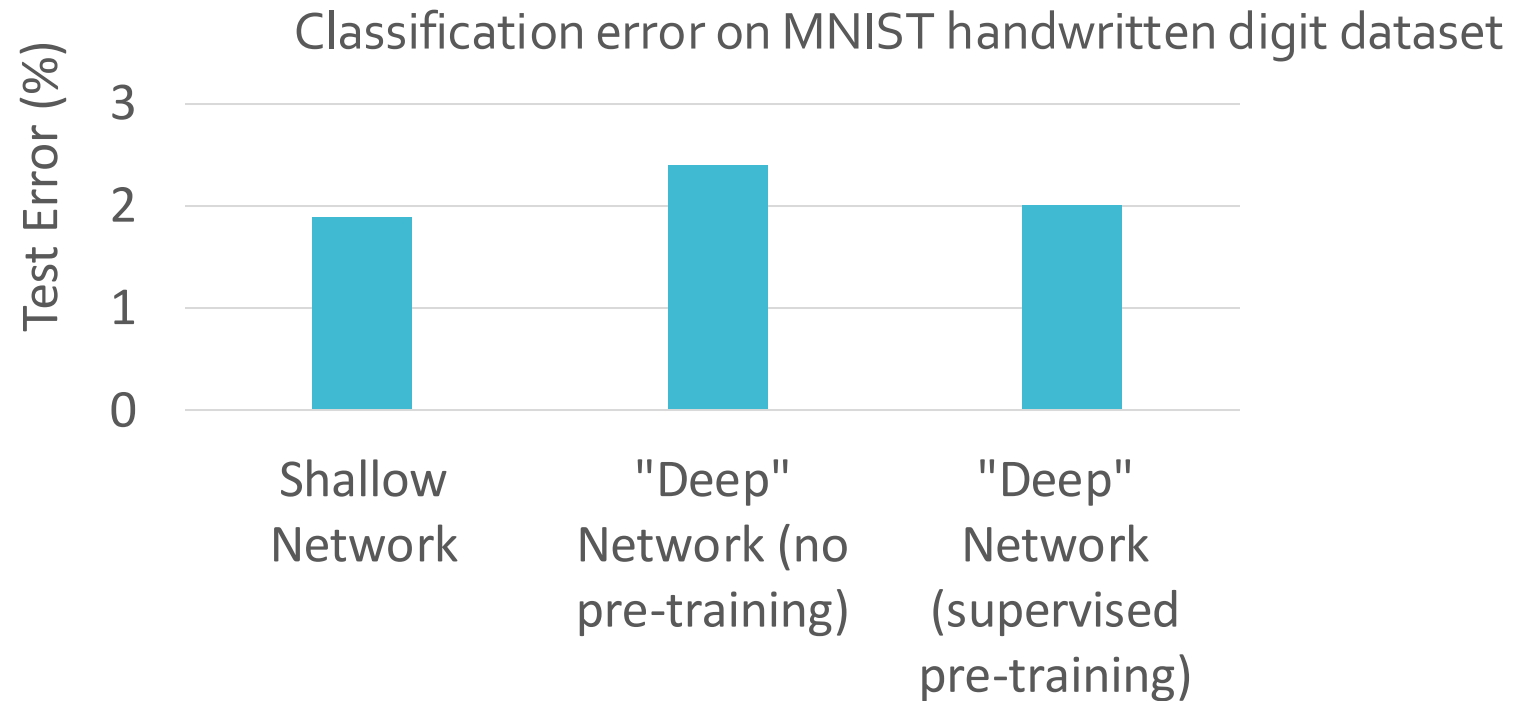
Supervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset *to predict the labels*
- Use the pre-trained weights as an initialization and *fine-tune* the entire network e.g., via SGD with the training dataset



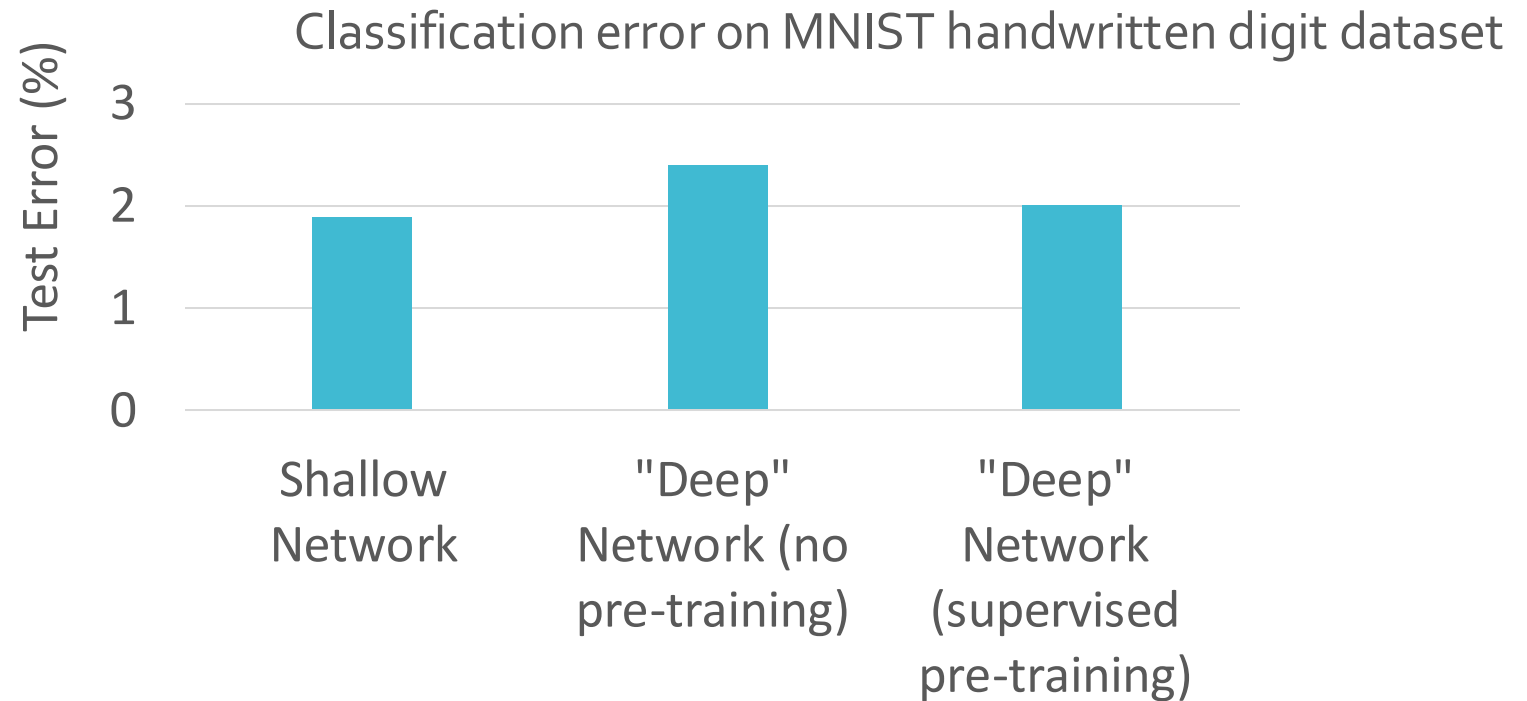
Is this the only thing we could do with the training data?

- Train each layer of the network iteratively using the training dataset *to predict the labels*
- Use the pre-trained weights as an initialization and *fine-tune* the entire network e.g., via SGD with the training dataset



Unsupervised Pre-training (Bengio et al., 2006)

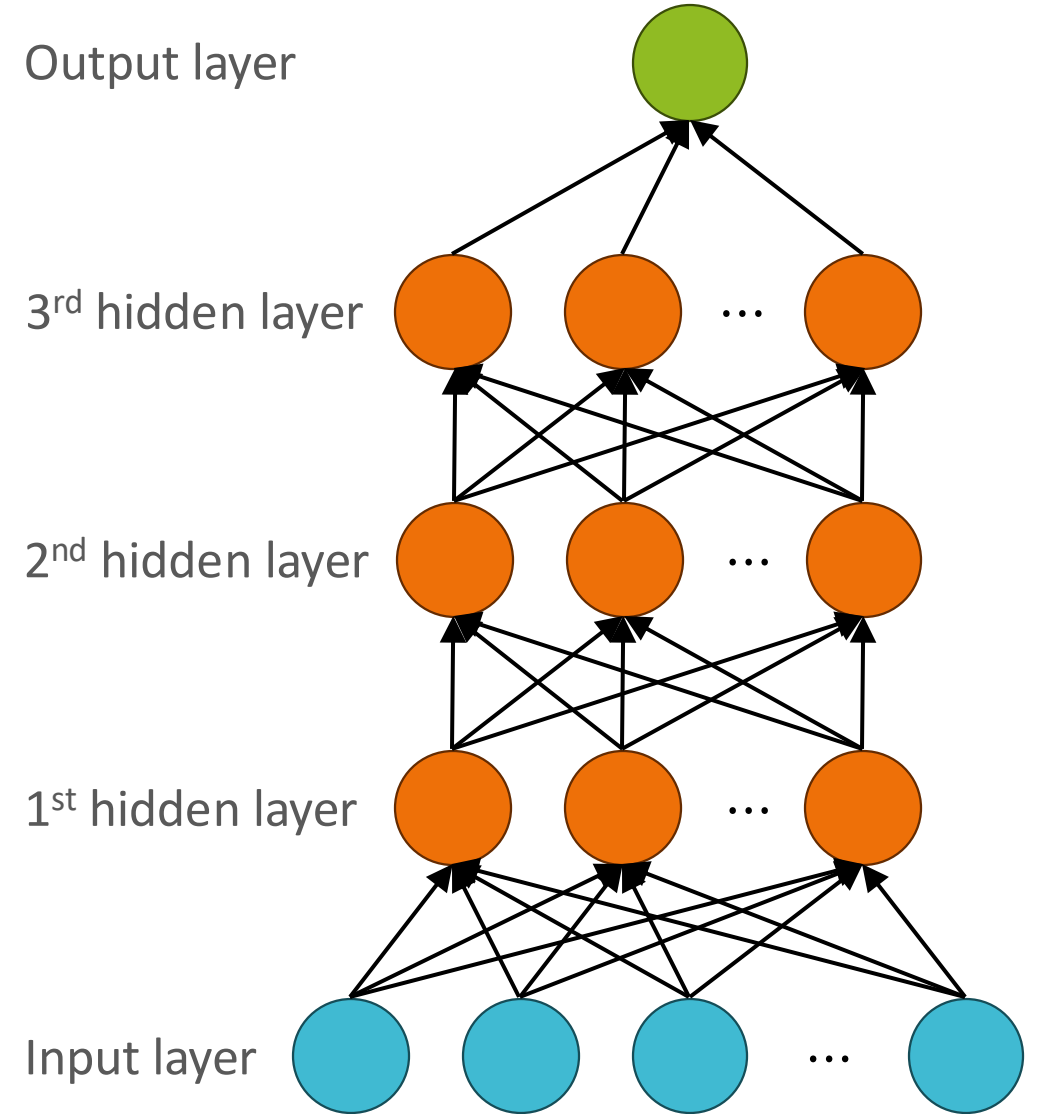
- Train each layer of the network iteratively using the training dataset *to learn useful representations*
- Idea: a good representation is one preserves a lot of information and could be used to recreate the inputs



Unsupervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset by minimizing the *reconstruction error*

$$\|\mathbf{x} - h(\mathbf{x})\|_2$$

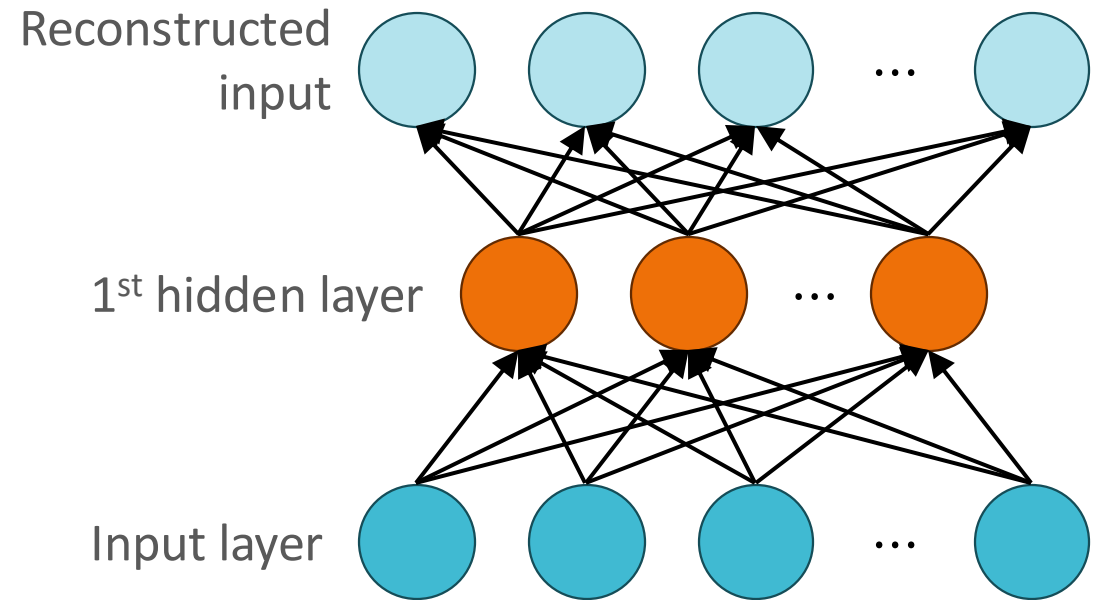


Unsupervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset by minimizing the *reconstruction error*

$$\|\mathbf{x} - h(\mathbf{x})\|_2$$

- This architecture/objective defines an *autoencoder*

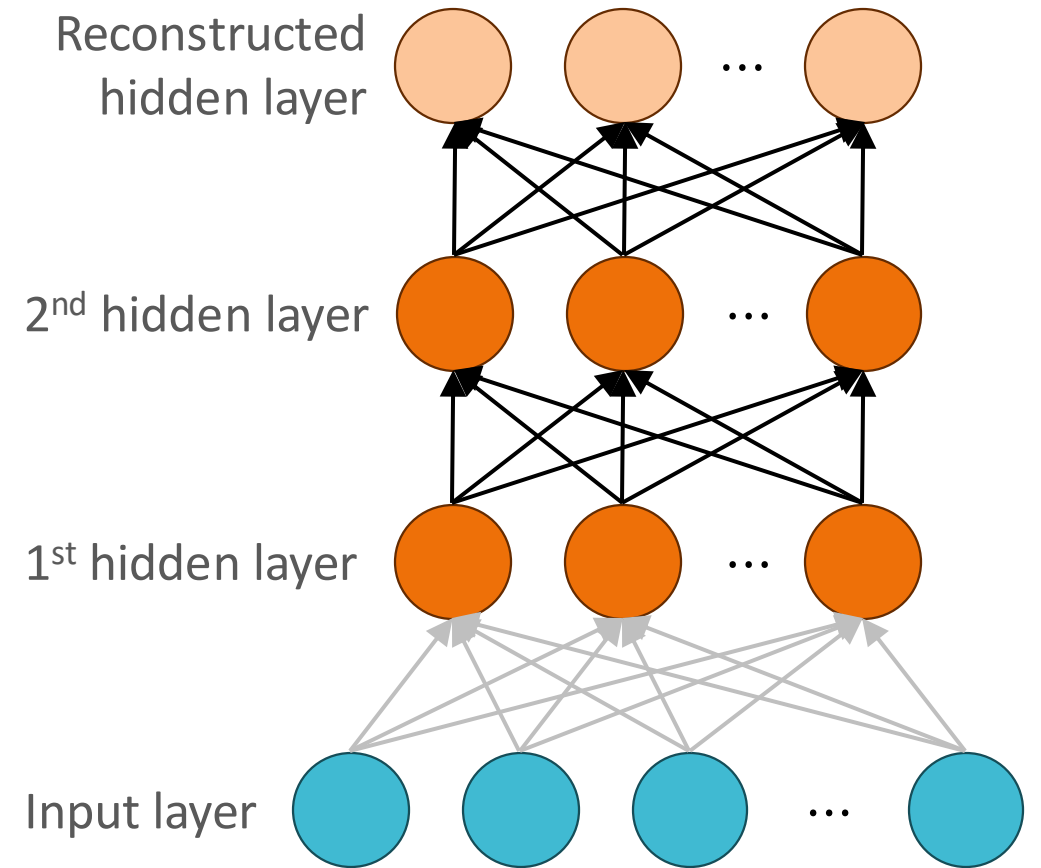


Unsupervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset by minimizing the *reconstruction error*

$$\|\mathbf{x} - h(\mathbf{x})\|_2$$

- This architecture/objective defines an *autoencoder*

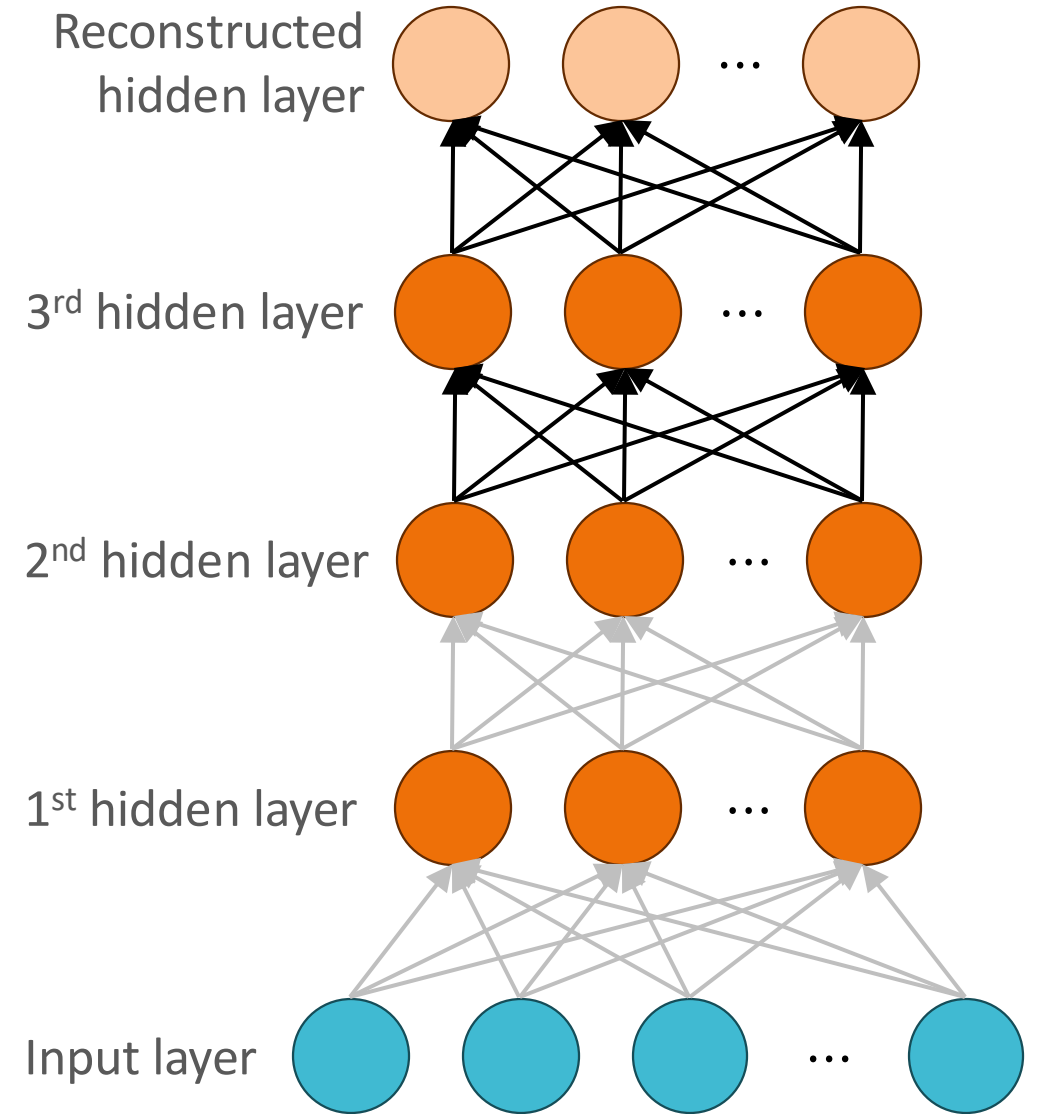


Unsupervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset by minimizing the *reconstruction error*

$$\|\mathbf{x} - \mathbf{h}(\mathbf{x})\|_2$$

- This architecture/objective defines an *autoencoder*

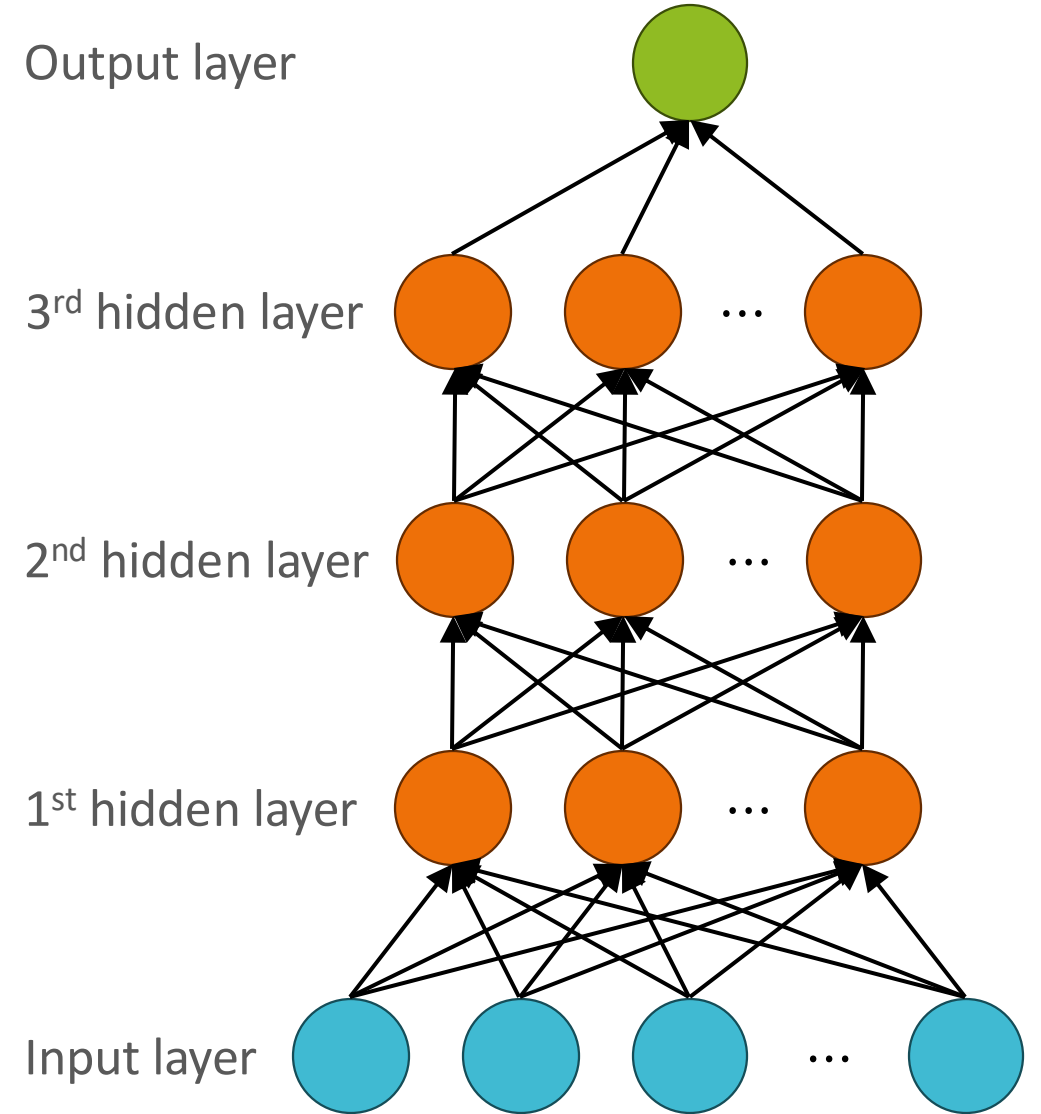


Fine-tuning (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset by minimizing the *reconstruction error*

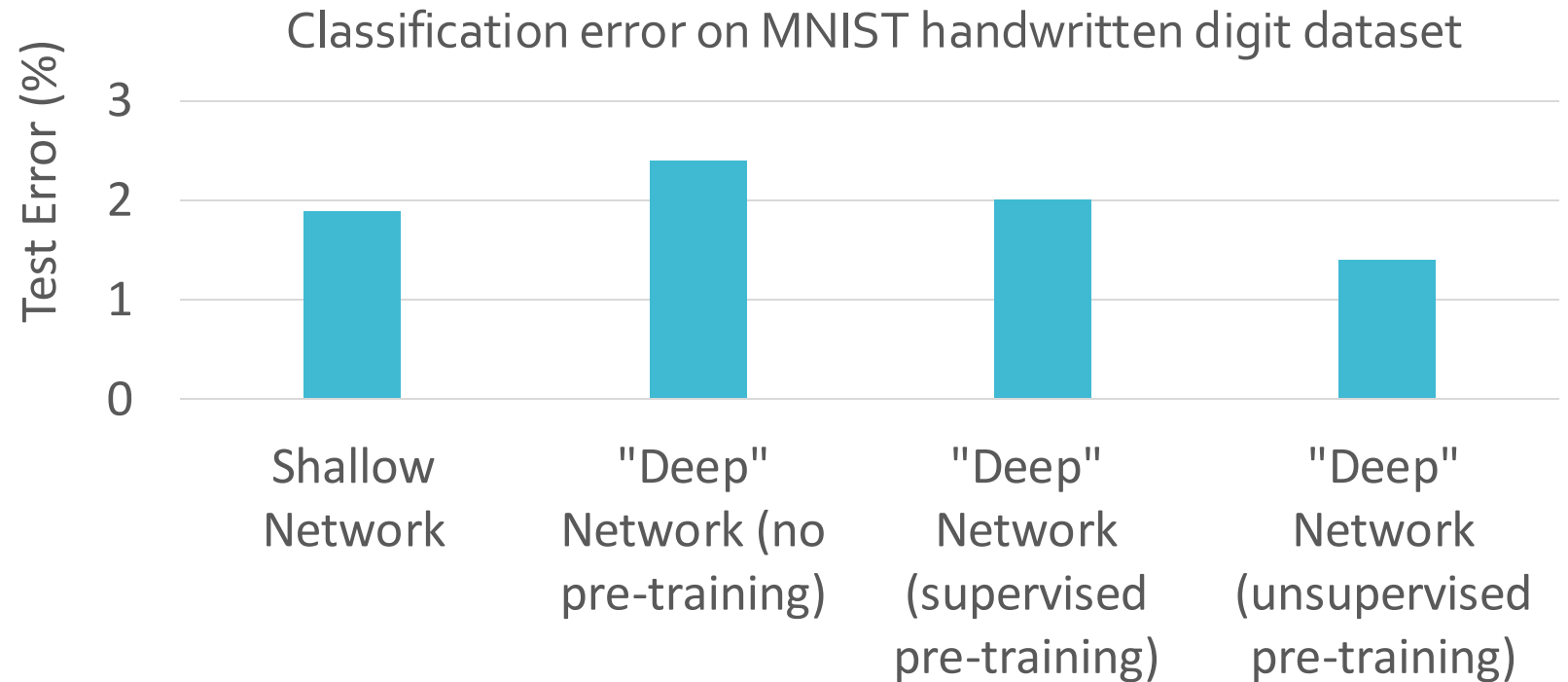
$$\|\mathbf{x} - \mathbf{h}(\mathbf{x})\|_2$$

- When fine-tuning, we're effectively swapping out the last layer and fitting all the weights to the training dataset



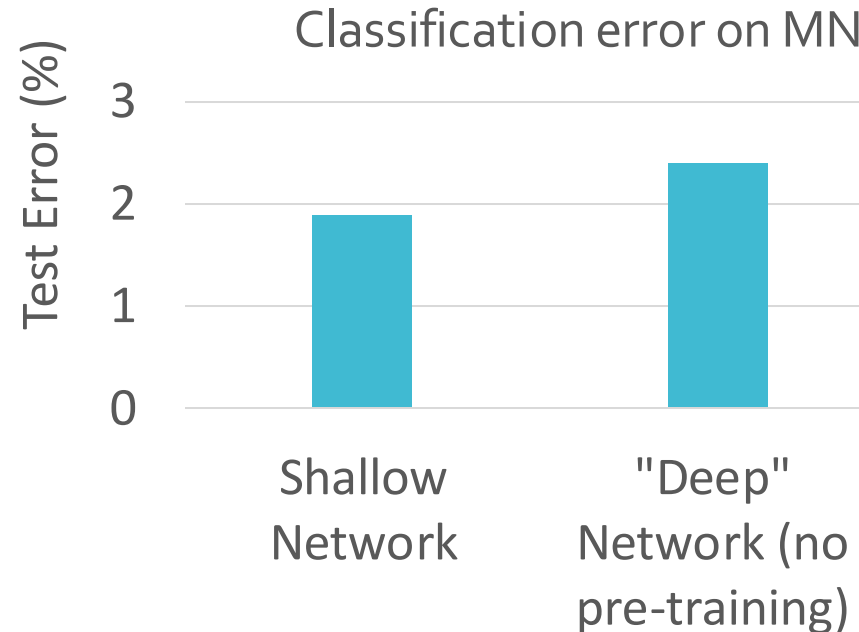
Unsupervised Pre-training (Bengio et al., 2006)

- Train each layer of the network iteratively using the training dataset by minimizing the *reconstruction error*
- Idea: a good representation is one preserves a lot of information and could be used to recreate the inputs



Another dose of Reality

- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a **tiny** labelled dataset to train with
- You fit a **massive** deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high



- Problem: what if you don't even have enough data to train a single layer/fine-tune the pre-trained network?

Another dose of Reality

- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a **tiny** labelled dataset to train with
- You fit a **massive** deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high
- Key observation: you can pre-train on basically any labelled or unlabelled dataset!
 - Ideally, you want to use a *large* dataset *related* to your goal task

Another dose of Reality

- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a **tiny** labelled dataset to train with
- You fit a **massive** deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high
- Key observation: you can pre-train on basically any labelled or unlabelled dataset!
 - GPT-3 pre-training data:

Dataset	Quantity (tokens)	Weight in training mix
Common Crawl (filtered)	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

Another dose of Reality

- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a **tiny** labelled dataset to train with
- You fit a **massive** deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high
- Key observation: you can pre-train on basically any labelled or unlabelled dataset!
- Okay that's great for pre-training and all, but what if
 - A. the concept of labelled data doesn't apply to your task i.e., not every input has a "correct" label e.g., chatbots?
 - B. you don't have enough data to fine-tune your model?

Reinforcement Learning from Human Feedback (RLHF)

- Insight: for many machine learning tasks, there is no universal ground truth, e.g., there are lots of possible ways to respond to a question or prompt.
- Idea: use human feedback to determine how good or bad some prediction/response is!
- Issue: if the input space is huge (e.g., all possible chat prompts), to train a good model, we might need tons and tons of (potentially expensive) human annotation...
- Idea: use a small number of annotations to learn a “reward” function!

Reinforcement Learning from Human Feedback (RLHF)

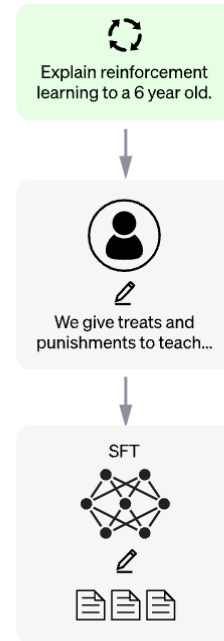
Step 1

Collect demonstration data and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3.5 with supervised learning.



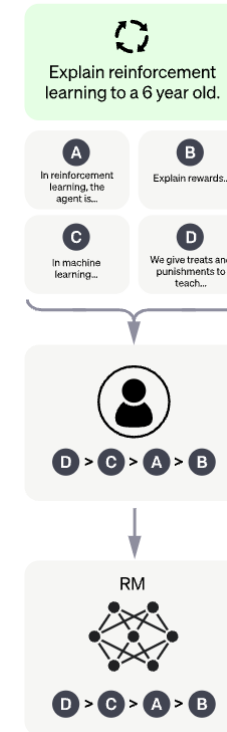
Step 2

Collect comparison data and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

Optimize a policy against the reward model using the PPO reinforcement learning algorithm.

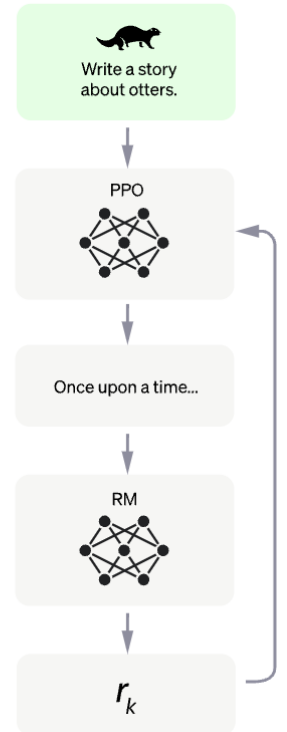
A new prompt is sampled from the dataset.

The PPO model is initialized from the supervised policy.

The policy generates an output.

The reward model calculates a reward for the output.

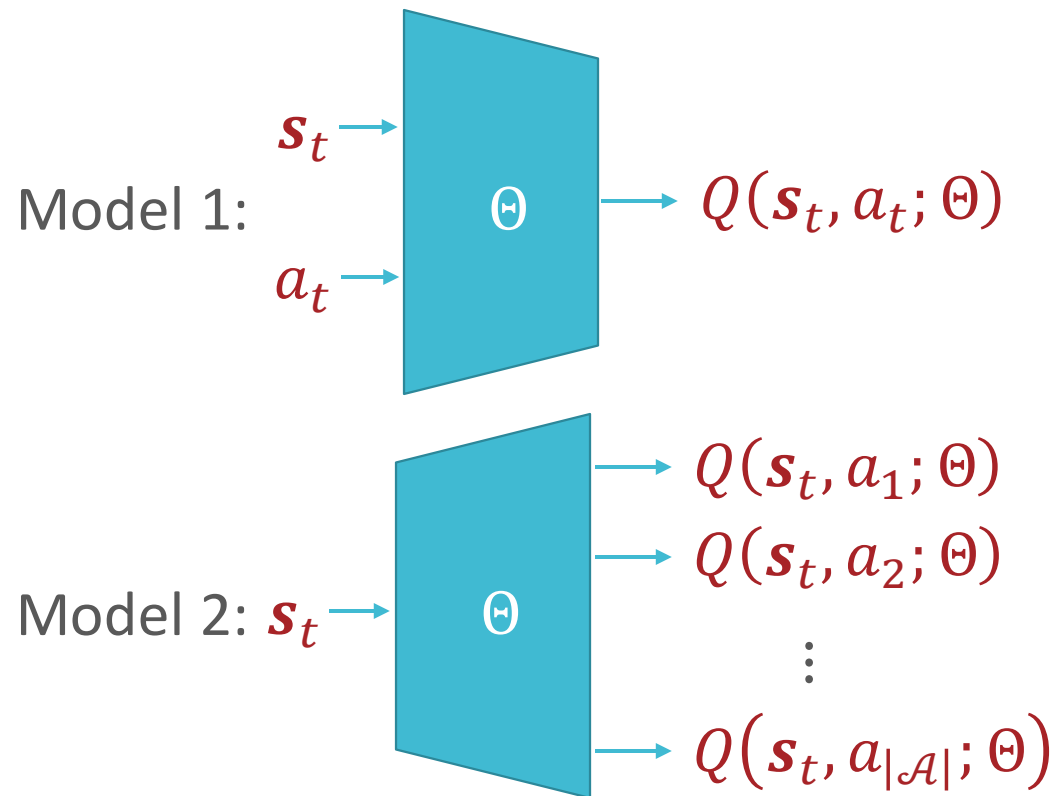
The reward is used to update the policy using PPO.



- RLHF is a special form of fine-tuning that uses *proximal policy optimization* (PPO).

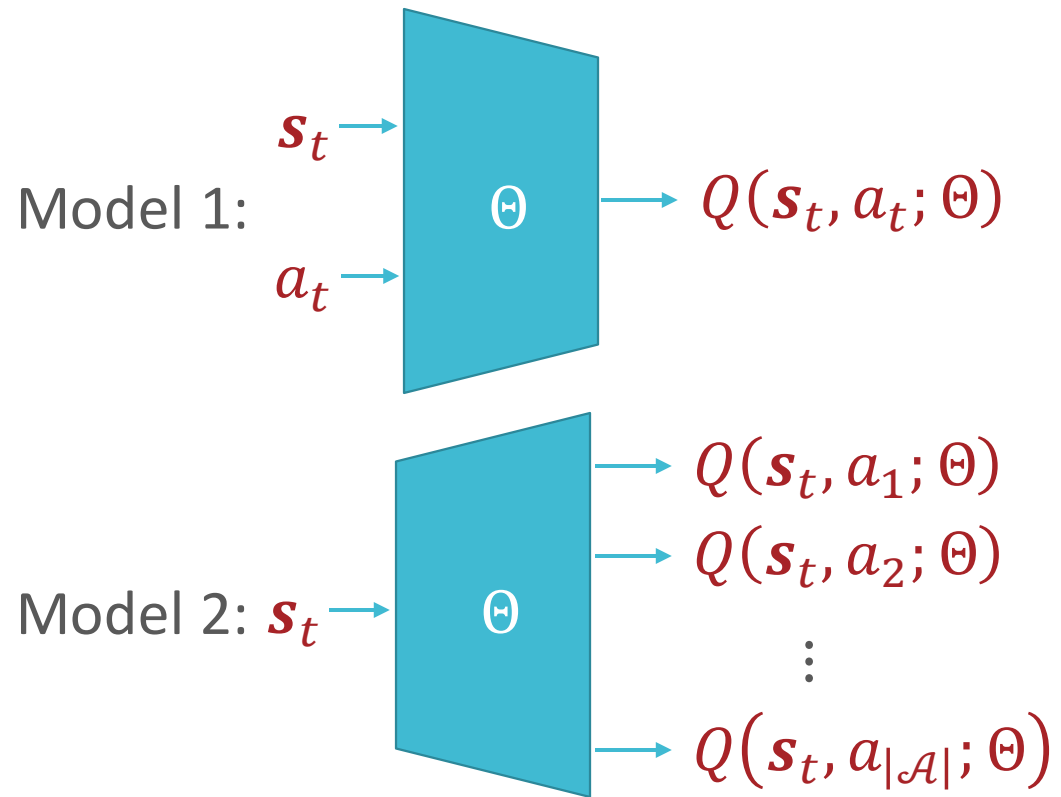
Recall: Deep Q-learning

- Represent states using some feature vector $\mathbf{s}_t \in \mathbb{R}^M$
e.g. for Go, $\mathbf{s}_t = [1, 0, -1, \dots, 1]^T$
- Define a *differentiable* function that approximates Q



What if instead of optimizing the Q-function, we could optimize the policy directly?

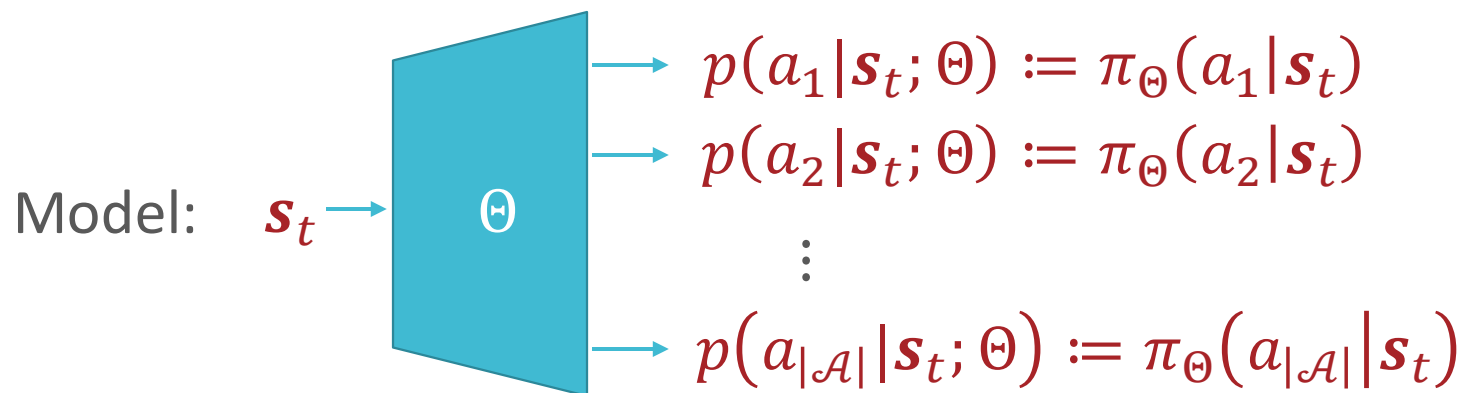
- Represent states using some feature vector $\mathbf{s}_t \in \mathbb{R}^M$
e.g. for Go, $\mathbf{s}_t = [1, 0, -1, \dots, 1]^T$
- Define a *differentiable* function that approximates Q



Parametrized Stochastic Policies

- Represent states using some feature vector $\mathbf{s}_t \in \mathbb{R}^M$
e.g. for Go, $\mathbf{s}_t = [1, 0, -1, \dots, 1]^T$
- Define a *differentiable* function that specifies a *stochastic* policy π_{Θ}
- Minimize the negative expected total reward w.r.t. Θ

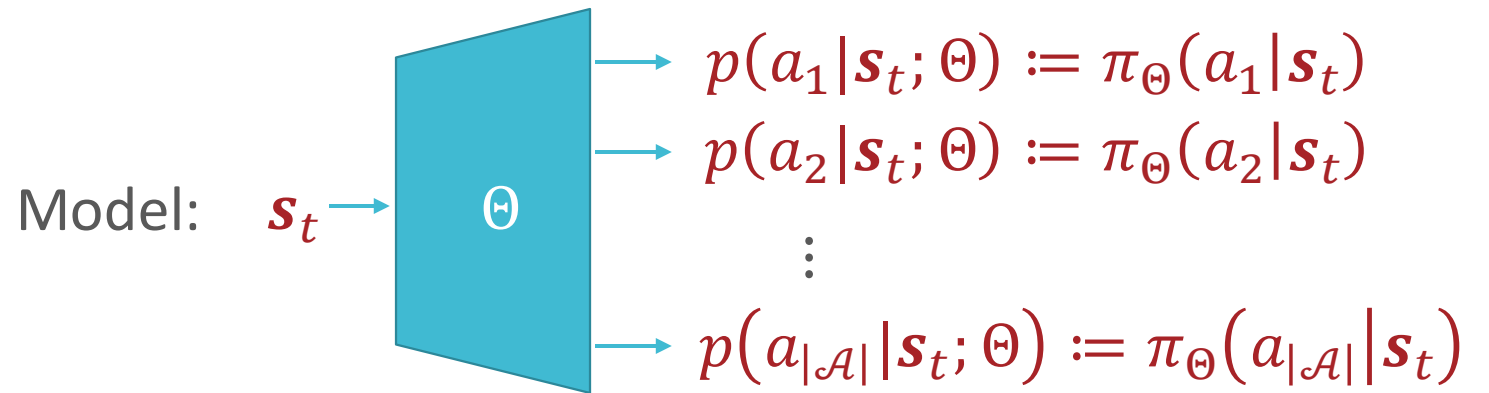
$$\ell(\Theta) = -\mathbb{E}_{\pi_{\Theta}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$



Okay... but
how on earth
do we
compute the
gradient of this
thing?

- Represent states using some feature vector $\mathbf{s}_t \in \mathbb{R}^M$
e.g. for Go, $\mathbf{s}_t = [1, 0, -1, \dots, 1]^T$
- Define a *differentiable* function that specifies a stochastic policy π_{Θ}
- Minimize the negative expected total reward w.r.t. Θ

$$\ell(\Theta) = -\mathbb{E}_{\pi_{\Theta}} \left[\mathbb{E}_{p(s'|s, a)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \right]$$



Trajectories

- A *trajectory* $\mathbf{T} = \{\mathbf{s}_0, a_0, \mathbf{s}_1, a_1, \dots, \mathbf{s}_T\}$ is one run of an agent through an MDP ending in a terminal state, \mathbf{s}_T
- Our stochastic policy and the transition distribution induce a distribution over trajectories

$$\begin{aligned} p_{\Theta}(\mathbf{T}) &= p(\{\mathbf{s}_0, a_0, \mathbf{s}_1, a_1, \dots, \mathbf{s}_T\}) \\ &= p(\mathbf{s}_0) \prod_{t=0}^{T-1} p(s_{t+1} | s_t, a_t) \pi_{\Theta}(a_t | \mathbf{s}_t) \end{aligned}$$

- Requires a distribution over initial states $p(\mathbf{s}_0)$ e.g., uniform over all states, fixed or deterministic, etc...
- If all runs end at a terminal state, then we can rewrite the negative expected total reward as

$$\ell(\Theta) = -\mathbb{E}_{p_{\Theta}(\mathbf{T}=\{\mathbf{s}_0, a_0, \dots, \mathbf{s}_T\})} \left[\sum_{t=0}^{T-1} \gamma^t R(\mathbf{s}_t, a_t) \right] := -\mathbb{E}_{p_{\Theta}(\mathbf{T})}[R(\mathbf{T})]$$

Likelihood
Ratio
Method
a.k.a.
REINFORCE
(Williams,
1992)

$$\begin{aligned}\nabla_{\Theta} \ell(\Theta) &= \nabla_{\Theta} (-\mathbb{E}_{p_{\Theta}(T)}[R(T)]) = \nabla_{\Theta} \left(-\int R(T) p_{\Theta}(T) dT \right) \\ &= -\int R(T) \nabla_{\Theta} p_{\Theta}(T) dT \\ &= -\int R(T) \nabla_{\Theta} \left(p(\mathbf{s}_0) \prod_{t=0}^{T-1} p(s_{t+1} | s_t, a_t) \pi_{\Theta}(a_t | \mathbf{s}_t) \right) dT\end{aligned}$$

- Issues:
 - The transition probabilities $p(s_{t+1} | s_t, a_t)$ are unknown a priori
 - Computing $\nabla_{\Theta} p_{\Theta}(T)$ involves taking the gradient of a product

Likelihood Ratio Method a.k.a. REINFORCE (Williams, 1992)

$$\begin{aligned}\nabla_{\Theta} \ell(\Theta) &= \nabla_{\Theta} (-\mathbb{E}_{p_{\Theta}(T)}[R(T)]) = \nabla_{\Theta} \left(-\int R(T) p_{\Theta}(T) dT \right) \\ &= -\int R(T) \nabla_{\Theta} p_{\Theta}(T) dT \\ &= -\int R(T) \nabla_{\Theta} \left(p(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t) \pi_{\Theta}(a_t|s_t) \right) dT\end{aligned}$$

- Insight:

$$\begin{aligned}\nabla_{\Theta} p_{\Theta}(T) &= \frac{p_{\Theta}(T)}{p_{\Theta}(T)} \nabla_{\Theta} p_{\Theta}(T) = p_{\Theta}(T) \nabla_{\Theta} (\log p_{\Theta}(T)) \\ \log p_{\Theta}(T) &= \log p(s_0) + \sum_{t=0}^{T-1} \log p(s_{t+1}|s_t, a_t) + \log \pi_{\Theta}(a_t|s_t) \\ \nabla_{\Theta} (\log p_{\Theta}(T)) &= \sum_{t=0}^{T-1} \nabla_{\Theta} \log \pi_{\Theta}(a_t|s_t) \quad \leftarrow \text{No longer depends on } p(s_{t+1}|s_t, a_t)!\end{aligned}$$

Likelihood Ratio Method a.k.a. REINFORCE (Williams, 1992)

$$\begin{aligned}\nabla_{\Theta} \ell(\Theta) &= \nabla_{\Theta} (-\mathbb{E}_{p_{\Theta}(T)}[R(T)]) = \nabla_{\Theta} \left(-\int R(T) p_{\Theta}(T) dT \right) \\ &= -\int R(T) \nabla_{\Theta} p_{\Theta}(T) dT = -\int R(T) \nabla_{\Theta} (\log p_{\Theta}(T)) p_{\Theta}(T) dT \\ &= -\mathbb{E}_{p_{\Theta}(T)} [R(T) \nabla_{\Theta} (\log p_{\Theta}(T))] \\ &\approx -\frac{1}{N} \sum_{n=1}^N R(T^{(n)}) \nabla_{\Theta} (\log p_{\Theta}(T^{(n)}))\end{aligned}$$

(where $T^{(n)} = \{\mathbf{s}_0^{(n)}, a_0^{(n)}, \mathbf{s}_1^{(n)}, a_1^{(n)}, \dots, \mathbf{s}_{T^{(n)}}^{(n)}\}$ is a sampled trajectory)

$$= -\frac{1}{N} \sum_{n=1}^N \left(\sum_{t=0}^{T^{(n)}-1} \gamma^t R(\mathbf{s}_t^{(n)}, a_t^{(n)}) \right) \left(\sum_{t=0}^{T^{(n)}-1} \nabla_{\Theta} \log \pi_{\Theta}(a_t^{(n)} | \mathbf{s}_t^{(n)}) \right)$$

Policy Gradient Methods

- Practical considerations:
 - Policy gradient methods are *on-policy*: they require using the current (potentially bad) policy to sample (a lot of) trajectories...
 - *Trust region methods* (Schulman et al., 2015) impose a constraint on how far the policy distribution can shift from one iteration to the next (in terms of a KL divergence)
 - *Proximal policy optimization* (Schulman et al., 2017) limits how much the magnitude of the objective function can change from one iteration to the next via clipping

In-context Learning

- Problem: given their size, effectively fine-tuning LLMs can require lots of labelled data points.
- Idea: leverage the LLM's context window by passing a few examples to the model as input, *without performing any updates to the parameters*
- Intuition: during training, the LLM is exposed to a *massive* number of examples/tasks and the input conditions the model to “locate” the relevant concepts

Few-shot, One-shot & Zero-shot (in-context) Learning

- Idea: leverage the LLM's context window by passing a few examples to the model as input, *without performing any updates to the parameters*

The three settings we explore for in-context learning

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
1 Translate English to French: ← task description
2 sea otter => loutre de mer ← examples
3 peppermint => menthe poivrée ←
4 plush girafe => girafe peluche ←
5 cheese => ..... ← prompt
```

Traditional fine-tuning (not used for GPT-3)

Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



Few-shot, One-shot & Zero-shot (in-context) Learning

- Idea: leverage the LLM's context window by passing a ~~few~~ one examples to the model as input, *without performing any updates to the parameters*

The three settings we explore for in-context learning

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
1 Translate English to French: ← task description
2 sea otter => loutre de mer ← example
3 cheese => ..... ← prompt
```

Traditional fine-tuning (not used for GPT-3)

Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



Few-shot, One-shot & Zero-shot (in-context) Learning

- Idea: leverage the LLM's context window by passing a ~~few~~ ~~one~~ zero(!) examples to the model as input, *without performing any updates to the parameters*

The three settings we explore for in-context learning

Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
1 Translate English to French: ← task description
2 cheese => ..... ← prompt
```

Traditional fine-tuning (not used for GPT-3)

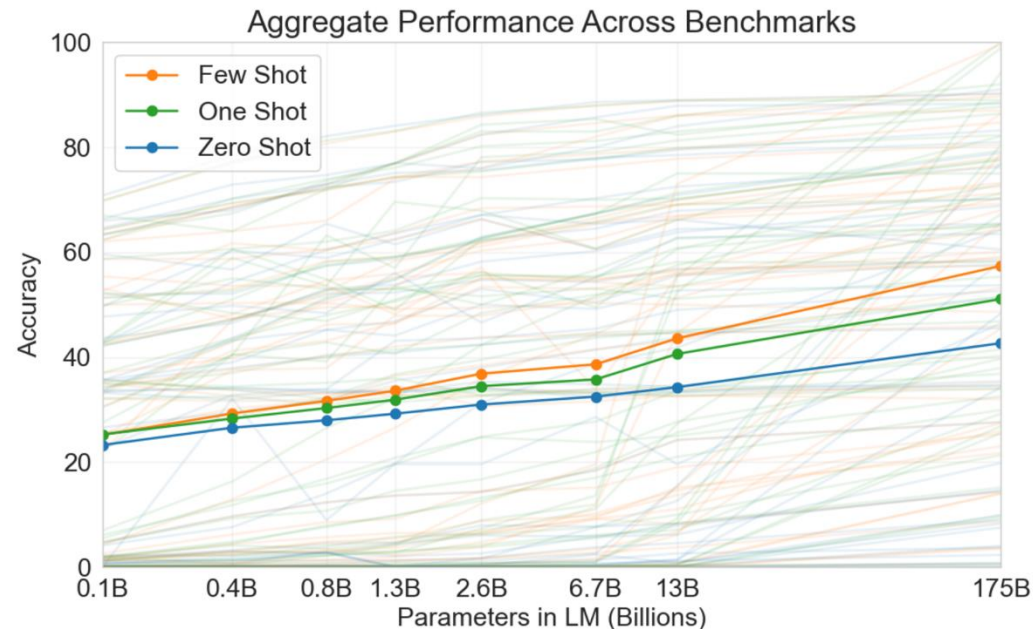
Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



Few-shot, One-shot & Zero-shot (in-context) Learning

- Idea: leverage the LLM's context window by passing a ~~few~~ one zero(!) examples to the model as input, *without performing any updates to the parameters*



- Key Takeaway: LLMs can perform well on novel tasks without having to fine-tune the model, sometimes even with just one or zero labelled training data points!

Key Takeaways

- Instead of random initializations, modern deep learning typically initializes weights via pretraining, then fine-tunes them to the specific task
 - Supervised vs. unsupervised fine-tuning
 - Pretraining need not occur on the task of interest
- For tasks with subjective outputs, models can be fine-tuned using reinforcement learning with human feedback
 - Uses (proximal) policy optimization to optimize a parametrized policy directly
- Some tasks can be performed by a pretrained LLM without any fine-tuning via in-context learning