

10-301/601: Introduction to Machine Learning

Lecture 19: Pretraining & Fine-tuning

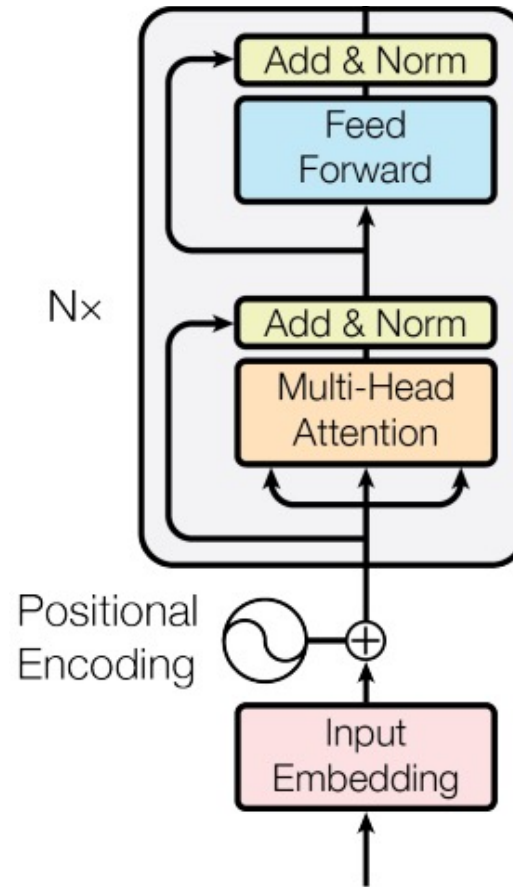
Matt Gormley & Henry Chai

3/24/25

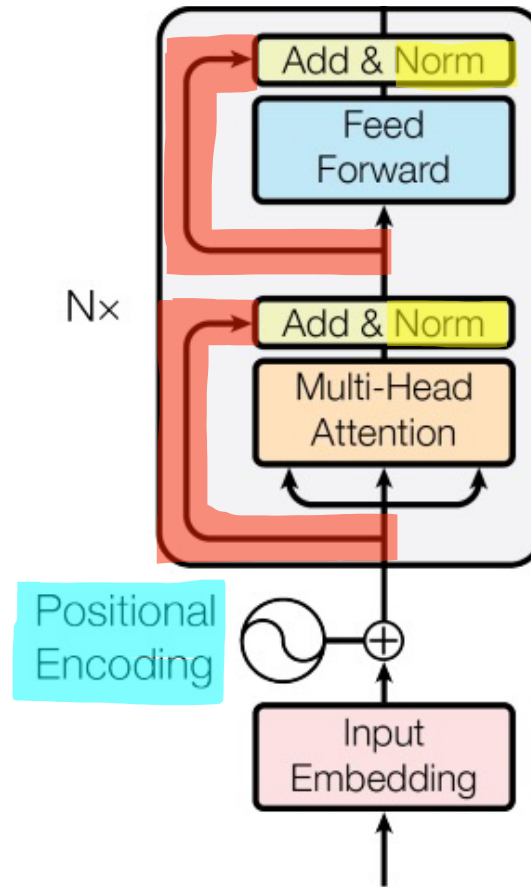
Front Matter

- Announcements
 - Exam 2 on 3/26 (Wednesday) from 7 – 9 PM
 - All topics from Lecture 8 to Lecture 16 (inclusive) + **MLE/MAP portion of Lecture 17** are in-scope
 - Exam 1 content may be referenced but will not be the primary focus of any question
 - Please review the seating chart on Piazza and make sure you have a seat / know where you're going

Transformers



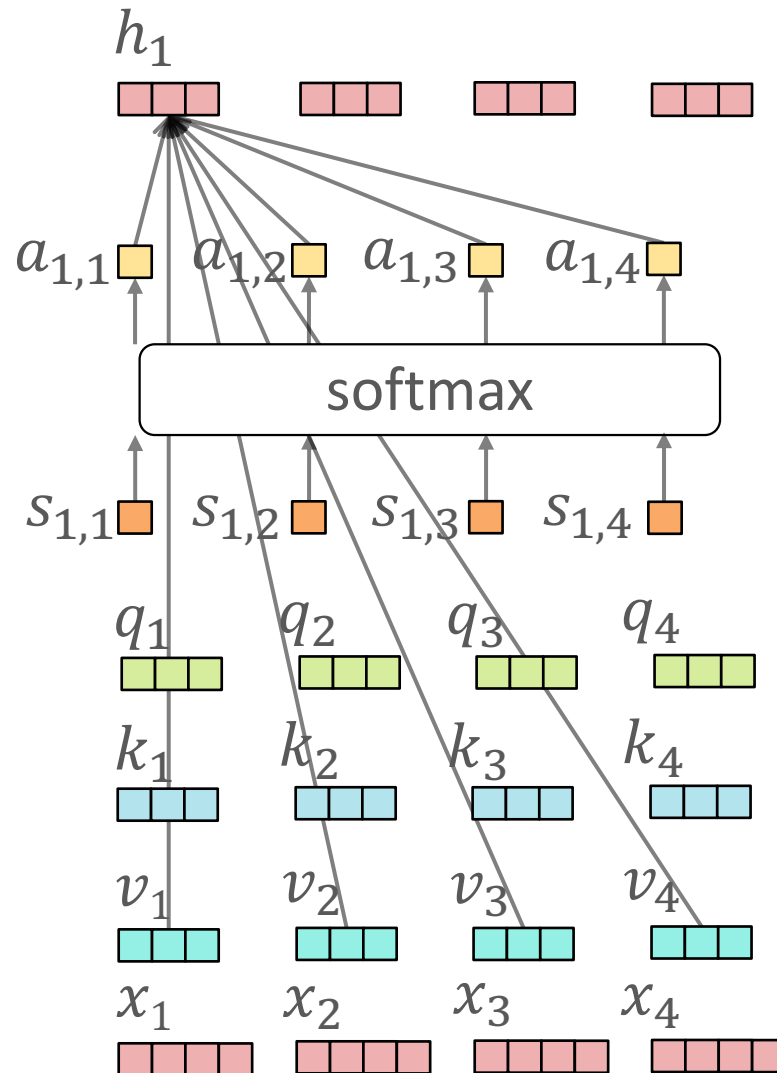
Transformers



- 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: Scaled Dot-product Attention

- Approach: compute a representation for each token in the *input sequence* by attending to all the input tokens



$$h_1 = \sum_{j=1}^4 \text{softmax}(s_{1,j}) v_j$$

attention weights

scores: $s_{1,j} = \frac{k_j^T q_1}{\sqrt{d_k}}$

queries: $q_t = W_Q x_t$

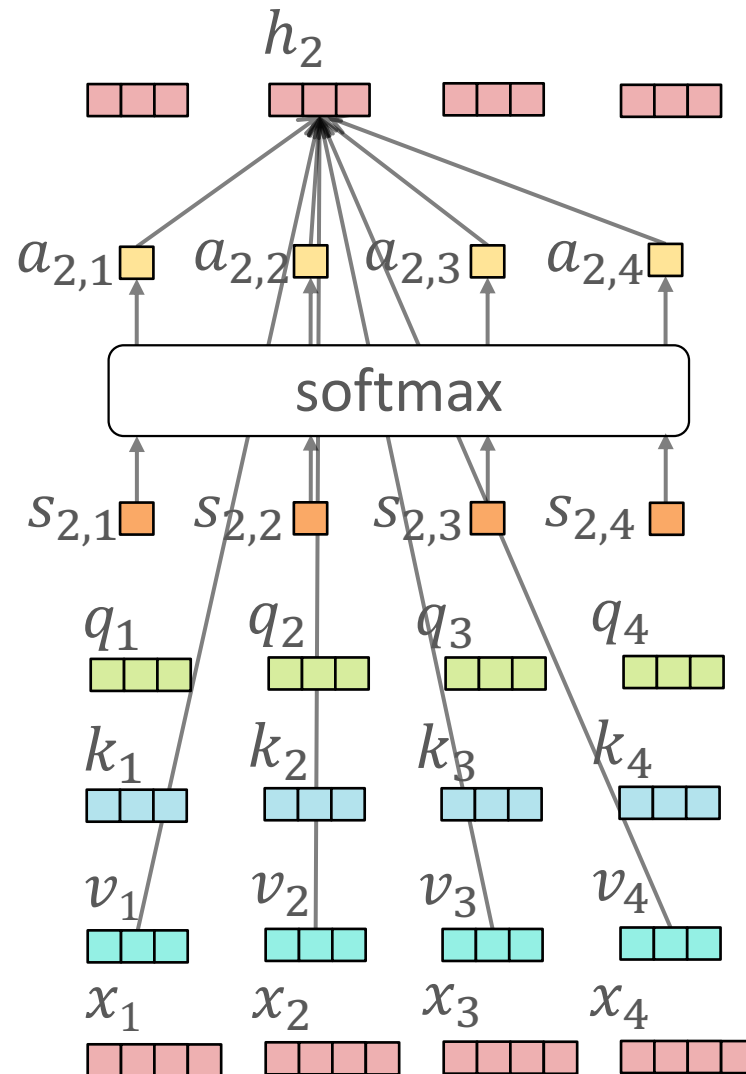
keys: $k_t = W_K x_t$

values: $v_t = W_V x_t$

input tokens

Recall: Scaled Dot-product Attention

- Approach: compute a representation for each token in the *input sequence* by attending to all the input tokens



$$h_2 = \sum_{j=1}^4 \text{softmax}(s_{2,j}) v_j$$

attention weights

scores: $s_{2,j} = \frac{k_j^T q_2}{\sqrt{d_k}}$

queries: $q_t = W_Q x_t$

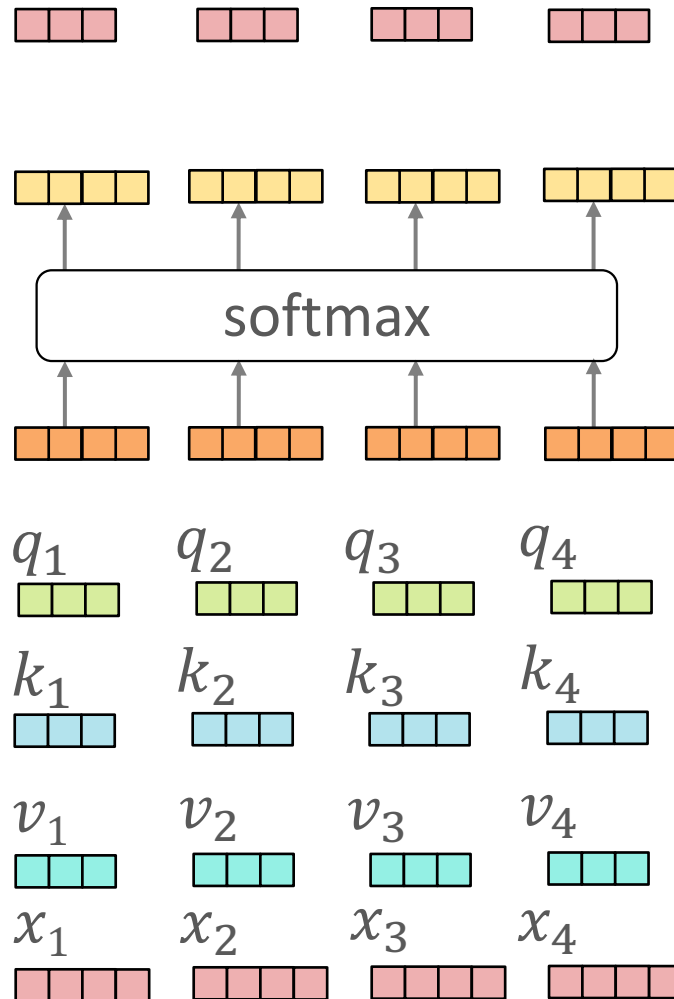
keys: $k_t = W_K x_t$

values: $v_t = W_V x_t$

input tokens

Scaled Dot-product Attention: Matrix Form

- Issue: if all tokens attend to every token in the sequence, then how does the model infer the order of tokens?



$$H = \text{softmax}(S)V \in \mathbb{R}^{N \times d_v}$$

attention weights

scores: $S = \frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{N \times N}$

queries: $Q = XW_Q \in \mathbb{R}^{N \times d_k}$

keys: $K = XW_K \in \mathbb{R}^{N \times d_k}$

values: $V = XW_V \in \mathbb{R}^{N \times d_v}$

design matrix: $X \in \mathbb{R}^{N \times D}$

Positional Encodings

- Issue: if all tokens attend to every token in the sequence, then how does the model infer the order of tokens?
- Idea: add a position-specific embedding p_t to the token embedding x_t

$$\tilde{x}_t = x_t + p_t$$

- Positional encodings can be
 - *fixed* i.e., some predetermined function of t or *learned* alongside the token embeddings
 - *absolute* i.e., only dependent on the token's location in the sequence or *relative* to the query token's location

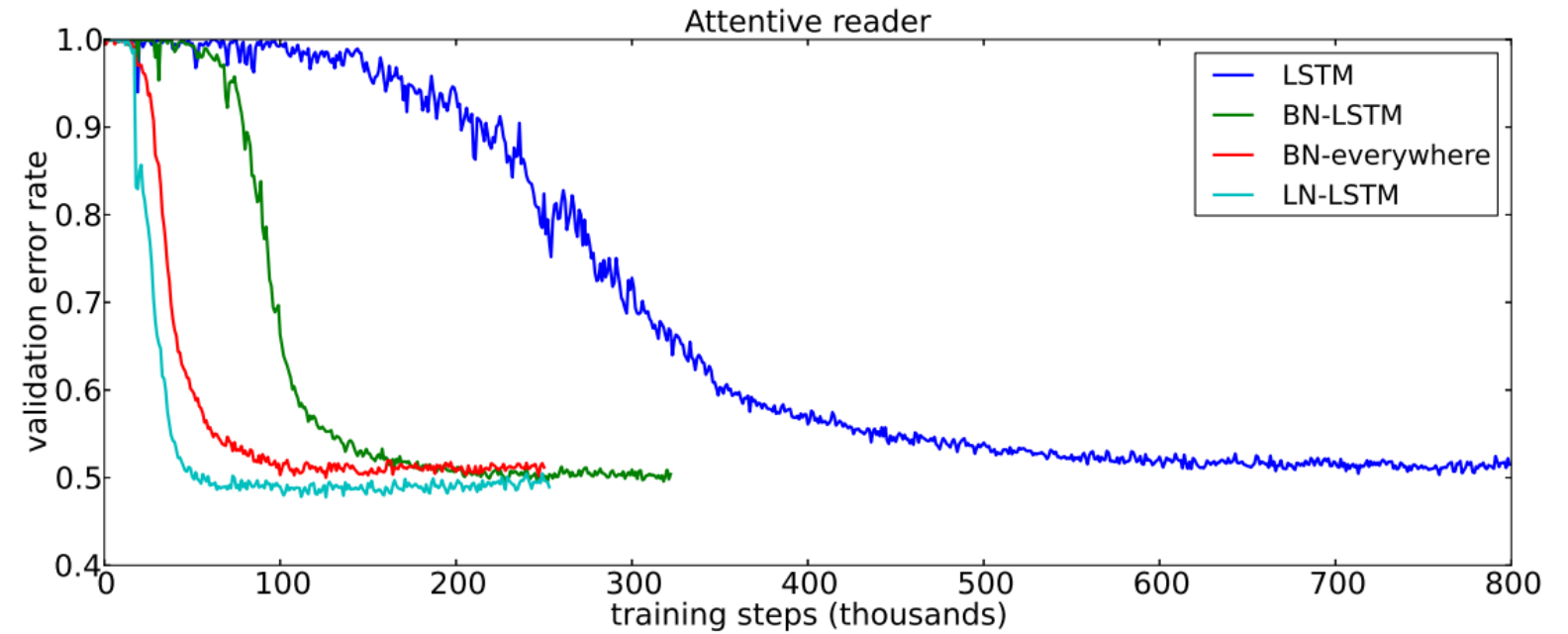
Layer Normalization

- Issue: for certain activation functions, the weights in later layers are **highly sensitive** to changes in the earlier layers
 - Small changes to weights in early layers are amplified so weights in deeper layers have to deal with massive dynamic ranges → slow optimization convergence
- Idea: normalize the output of a layer to always have the same (learnable) mean, β , and variance, γ^2

$$H' = \gamma \left(\frac{H - \mu}{\sigma} \right) + \beta$$

where μ is the mean and σ is the standard deviation of the values in the vector H

Layer Normalization



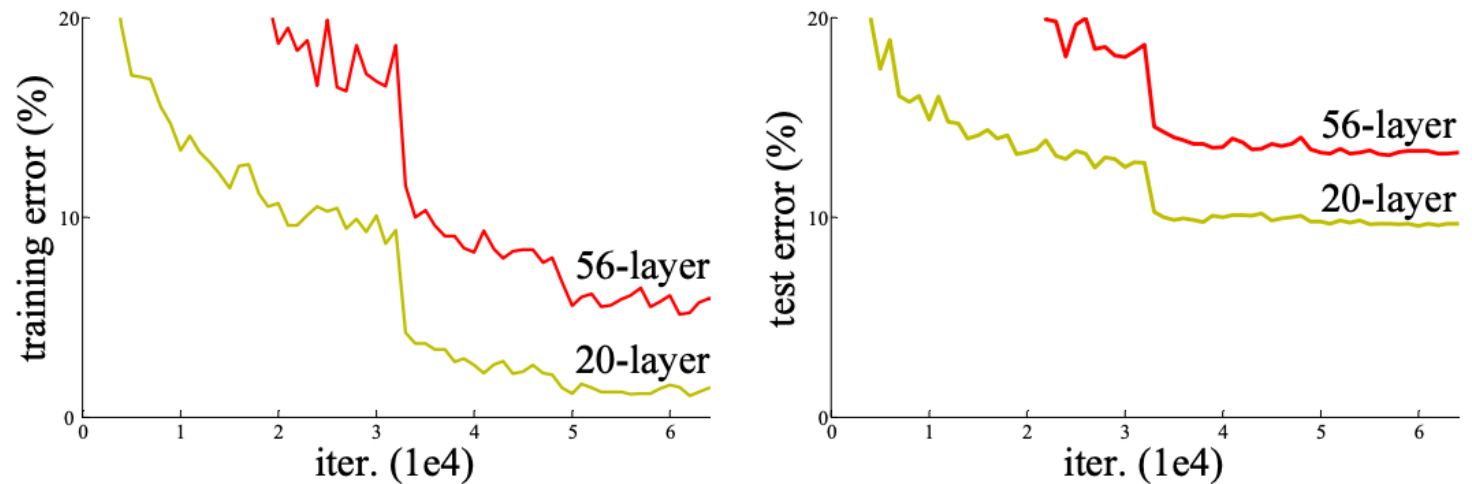
- Idea: normalize the output of a layer to always have the same (learnable) mean, β , and variance, γ^2

$$H' = \gamma \left(\frac{H - \mu}{\sigma} \right) + \beta$$

where μ is the mean and σ is the standard deviation of the values in the vector H

Residual Connections

- Observation: early deep neural networks suffered from the “degradation” problem where adding more layers actually made performance worse!



- Wait but this is ridiculous: if the later layers aren't helping, couldn't they just learn the identity transformation???
- Insight: neural network layers actually have a hard time learning the identity function

Residual Connections

- Observation: early deep neural networks suffered from the “degradation” problem where adding more layers actually made performance worse!
- Idea: add the input embedding back to the output of a layer

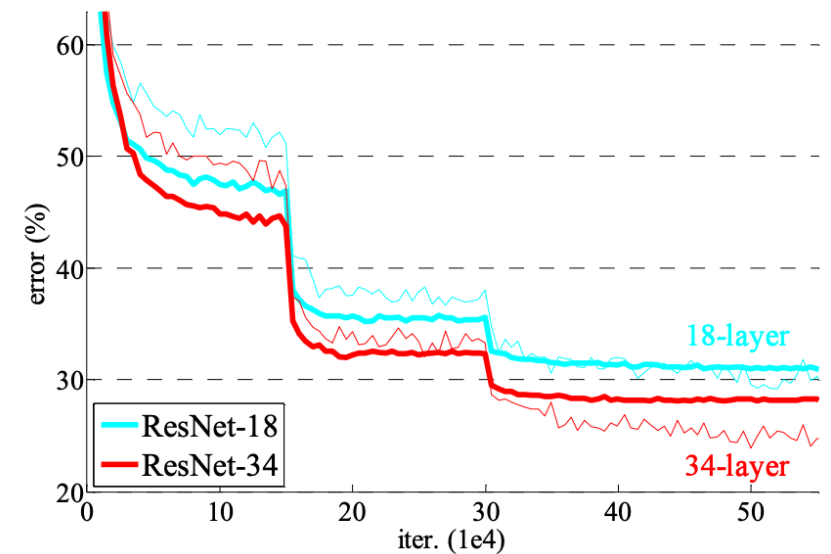
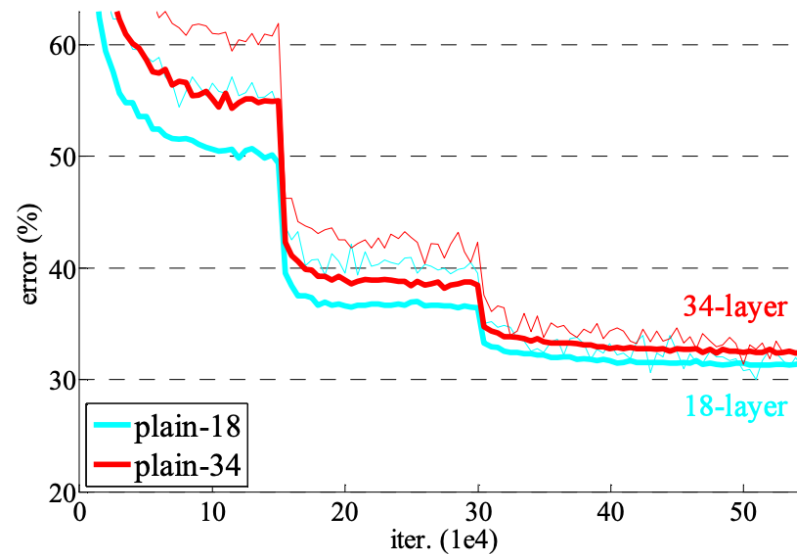
$$H' = H(x^{(i)}) + x^{(i)}$$

- Suppose the target function is f
 - Now instead of having to learn $f(x^{(i)})$, the hidden layer just needs to learn the residual $r = f(x^{(i)}) - x^{(i)}$
 - If f is the identity function, then the hidden layer just needs to learn $r = 0$, which is easy for a neural network!

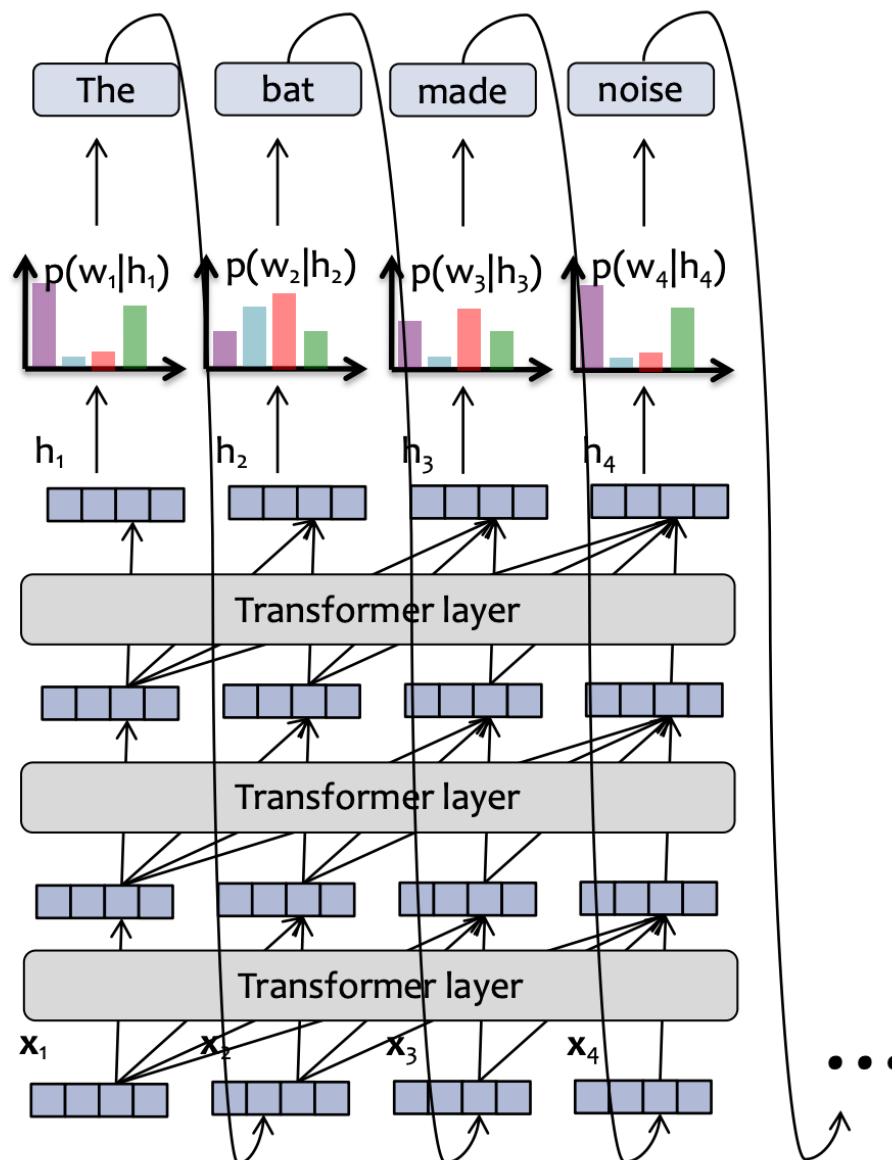
Residual Connections

- Observation: early deep neural networks suffered from the “degradation” problem where adding more layers actually made performance worse!
- Idea: add the input embedding back to the output of a layer

$$H' = H(x^{(i)}) + x^{(i)}$$



Transformers



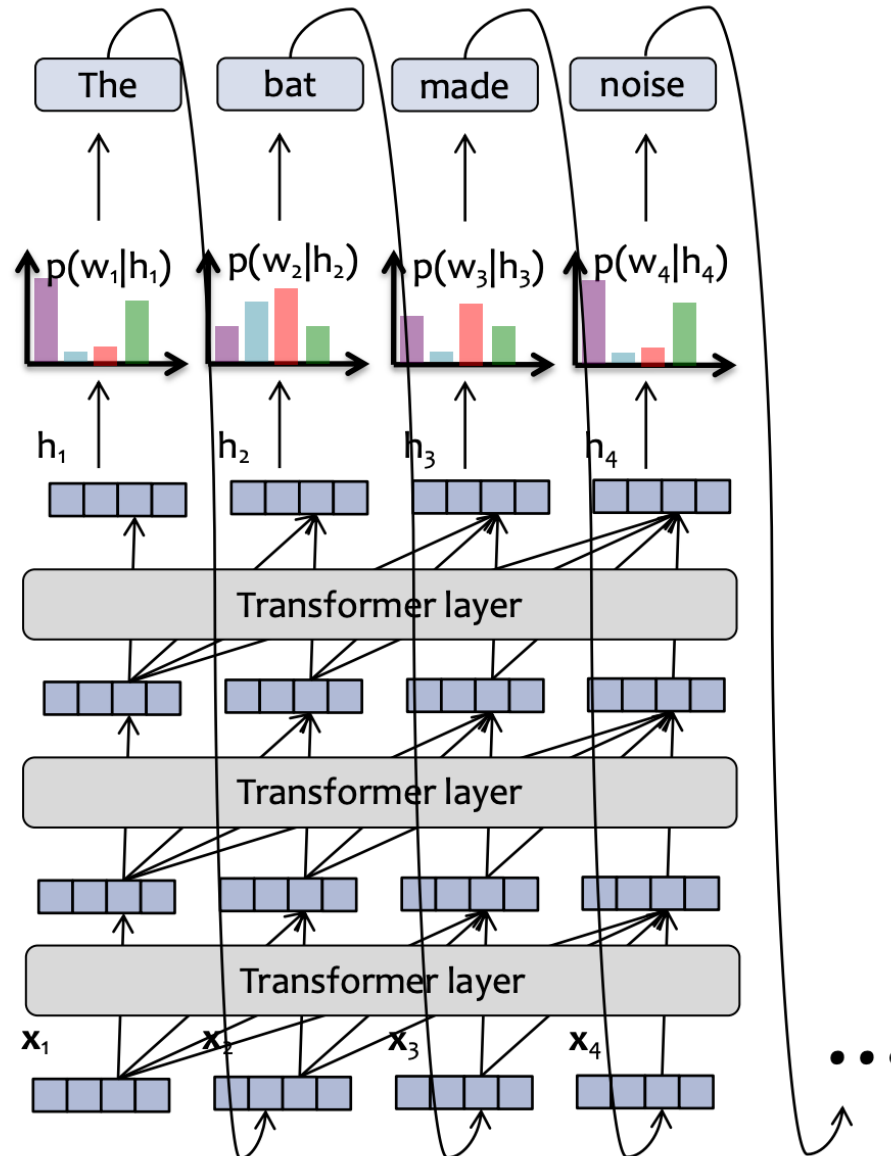
Each layer of a Transformer LM consists of several **sublayers**:

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer**.

The language model part is just like an RNN-LM.

Okay, but how on earth do we go about training these things?



Each layer of a Transformer LM consists of several **sublayers**:

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer**.

The language model part is just like an RNN-LM.

Backpropagation: Procedural Method

Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Params  $\alpha, \beta$ )
2:    $\mathbf{a} = \alpha \mathbf{x}$ 
3:    $\mathbf{z} = \sigma(\mathbf{a})$ 
4:    $\mathbf{b} = \beta \mathbf{z}$ 
5:    $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$ 
6:    $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

Algorithm 2 Backpropagation

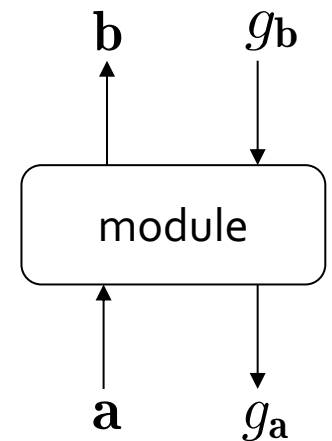
```
1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Params  $\alpha, \beta$ ,  
  Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$ 
4:    $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T (\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}} \hat{\mathbf{y}}^T)$ 
5:    $\mathbf{g}_{\beta} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$ 
6:    $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}$ 
7:    $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$ 
8:    $\mathbf{g}_{\alpha} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

Issues:

1. Hard to reuse / adapt for other models
2. Hard to optimize individual steps
3. Hard to debug using the finite-difference check

Module-based AutoDiff

- Key Idea:
 - componentize the computation of the neural-network into layers
 - each layer consolidates multiple **real-valued nodes** in the computation graph (a subset of them) into one **vector-valued node** (aka. a **module**)
- Each **module** is capable of two actions:
 - **Forward computation of the output given some input**
 - **Backward computation of the gradient with respect to the input given the gradient with respect to the output**



Module-based AutoDiff

Linear Module The linear layer has two inputs: a vector $\mathbf{a}$ and parameters $\omega \in \mathbb{R}^{B \times A}$. The output $\mathbf{b}$ is not used by LINEARBACKWARD, but we pass it in for consistency of form.

```
1: procedure LINEARFORWARD( $\mathbf{a}, \omega$ )
2:    $\mathbf{b} = \omega \mathbf{a}$ 
3:   return  $\mathbf{b}$ 
4: procedure LINEARBACKWARD( $\mathbf{a}, \omega, \mathbf{b}, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\omega = \mathbf{g}_\mathbf{b} \mathbf{a}^T$ 
6:    $\mathbf{g}_\mathbf{a} = \omega^T \mathbf{g}_\mathbf{b}$ 
7:   return  $\mathbf{g}_\omega, \mathbf{g}_\mathbf{a}$ 
```

Softmax Module The softmax layer has only one input vector $\mathbf{a}$. For any vector $\mathbf{v} \in \mathbb{R}^D$, we have that $\text{diag}(\mathbf{v})$ returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \dots, v_D$ and whose non-diagonal entries are zero.

```
1: procedure SOFTMAXFORWARD( $\mathbf{a}$ )
2:    $\mathbf{b} = \text{softmax}(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SOFTMAXBACKWARD( $\mathbf{a}, \mathbf{b}, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\mathbf{a} = \mathbf{g}_\mathbf{b}^T (\text{diag}(\mathbf{b}) - \mathbf{b} \mathbf{b}^T)$ 
6:   return  $\mathbf{g}_\mathbf{a}$ 
```

Sigmoid Module The sigmoid layer has only one input vector $\mathbf{a}$. Below σ is the sigmoid applied element-wise, and $\odot$ is element-wise multiplication s.t. $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \dots, u_M v_M]$.

```
1: procedure SIGMOIDFORWARD( $\mathbf{a}$ )
2:    $\mathbf{b} = \sigma(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SIGMOIDBACKWARD( $\mathbf{a}, \mathbf{b}, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\mathbf{a} = \mathbf{g}_\mathbf{b} \odot \mathbf{b} \odot (1 - \mathbf{b})$ 
6:   return  $\mathbf{g}_\mathbf{a}$ 
```

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $\mathbf{a}$ and a predicted probability distribution $\hat{\mathbf{a}}$. Its output $b \in \mathbb{R}$ is a scalar. Below $\div$ is element-wise division. The output b is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

```
1: procedure CROSSENTROPYFORWARD( $\mathbf{a}, \hat{\mathbf{a}}$ )
2:    $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$ 
3:   return  $b$ 
4: procedure CROSSENTROPYBACKWARD( $\mathbf{a}, \hat{\mathbf{a}}, b, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\mathbf{\hat{a}} = -\mathbf{g}_\mathbf{b} (\mathbf{a} \div \hat{\mathbf{a}})$ 
6:   return  $\mathbf{g}_\mathbf{\hat{a}}$ 
```

Module-based AutoDiff

Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ )
2:    $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \alpha)$ 
3:    $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$ 
4:    $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \beta)$ 
5:    $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$ 
6:    $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

Algorithm 2 Backpropagation

```
1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ , Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $g_J = \frac{dJ}{dJ} = 1$  ▷ Base case
4:    $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$ 
5:    $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$ 
6:    $\mathbf{g}_{\beta}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$ 
7:    $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$ 
8:    $\mathbf{g}_{\alpha}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$  ▷ We discard  $\mathbf{g}_{\mathbf{x}}$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

1. Easy to reuse / adapt for other models
2. Individual layers are easier to optimize
3. Simple to debug: just run a finite-difference check on each layer separately

Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:

- Let each module be an object and allow the **control flow** of the program to define the computation graph
- **No longer need to implement NNBackward($\cdot$)**, just follow the computation graph in reverse topological order

```
1 class Sigmoid(Module)
2     method forward(a)
3          $b = \sigma(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b \odot b \odot (1 - b)$ 
7         return ga
```

```
1 class Softmax(Module)
2     method forward(a)
3          $b = \text{softmax}(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b^T (\text{diag}(b) - bb^T)$ 
7         return ga
```

```
1 class Linear(Module)
2     method forward(a,  $\omega$ )
3          $b = \omega a$ 
4         return b
5     method backward(a,  $\omega$ , b, gb)
6          $g_\omega = g_b a^T$ 
7          $g_a = \omega^T g_b$ 
8         return g $\omega$ , ga
```

```
1 class CrossEntropy(Module)
2     method forward(a,  $\hat{a}$ )
3          $b = -a^T \log \hat{a}$ 
4         return b
5     method backward(a,  $\hat{a}$ , b, gb)
6          $g_{\hat{a}} = -g_b (a \div \hat{a})$ 
7         return ga
```

Module-based AutoDiff (OOP Version)

```
1  class NeuralNetwork(Module):
2
3      method init()
4          lin1_layer = Linear()
5          sig_layer = Sigmoid()
6          lin2_layer = Linear()
7          soft_layer = Softmax()
8          ce_layer = CrossEntropy()
9
10     method forward(Tensor  $\mathbf{x}$  , Tensor  $\mathbf{y}$  , Tensor  $\alpha$  , Tensor  $\beta$ )
11          $\mathbf{a}$  = lin1_layer.apply_fwd( $\mathbf{x}$ ,  $\alpha$ )
12          $\mathbf{z}$  = sig_layer.apply_fwd( $\mathbf{a}$ )
13          $\mathbf{b}$  = lin2_layer.apply_fwd( $\mathbf{z}$ ,  $\beta$ )
14          $\hat{\mathbf{y}}$  = soft_layer.apply_fwd( $\mathbf{b}$ )
15          $J$  = ce_layer.apply_fwd( $\mathbf{y}$ ,  $\hat{\mathbf{y}}$ )
16         return  $J$ .out_tensor
17
18     method backward(Tensor  $\mathbf{x}$  , Tensor  $\mathbf{y}$  , Tensor  $\alpha$  , Tensor  $\beta$ )
19         tape_bwd()
20         return lin1_layer.in_gradients[1] , lin2_layer.in_gradients[1]
```

Module-based AutoDiff (OOP Version)

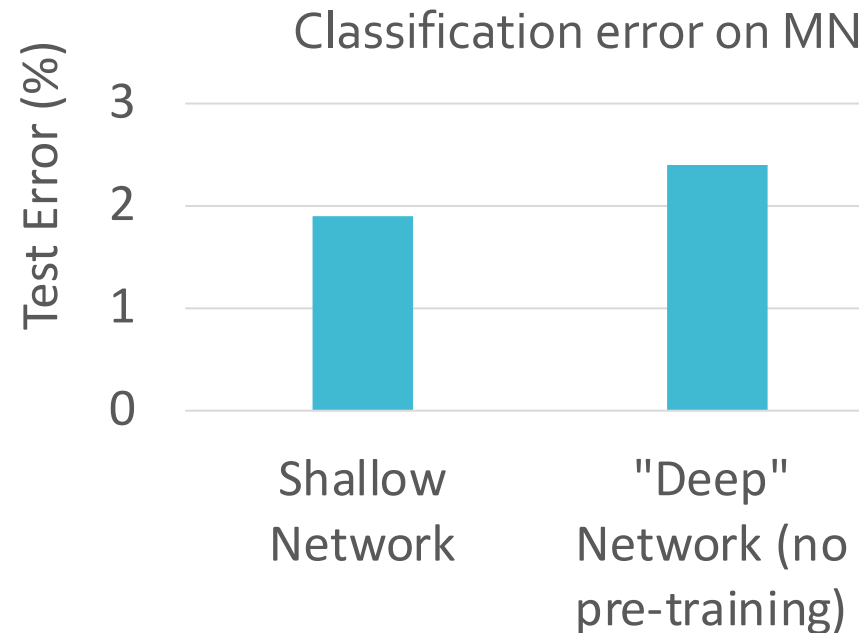
```
1 global tape = stack()
2
3 class Module:
4
5     method init()
6         out_tensor = null
7         out_gradient = 1
8
9     method apply_fwd(List in_modules)
10         in_tensors = [x.out_tensor for x in in_modules]
11         out_tensor = forward(in_tensors)
12         tape.push(self)
13         return self
14
15     method apply_bwd():
16         in_gradients = backward(in_tensors, out_tensor, out_gradient)
17         for i in 1, ..., len(in_modules):
18             in_modules[i].out_gradient += in_gradients[i]
19         return self
20
21 function tape_bwd():
22     while len(tape) > 0
23         m = tape.pop()
24         m.apply_bwd()
```

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

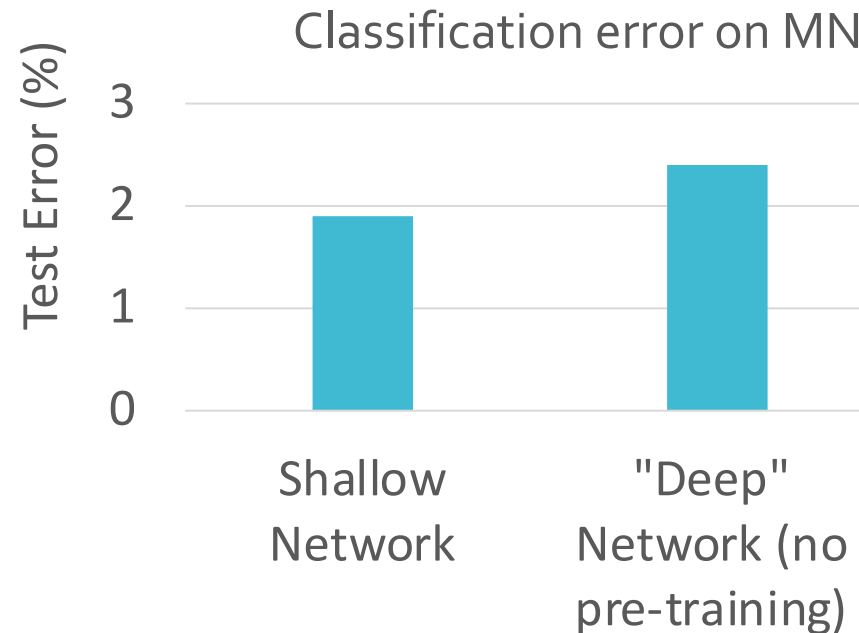
- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a tiny labelled dataset to train with
- You fit a massive deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high



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

Reality

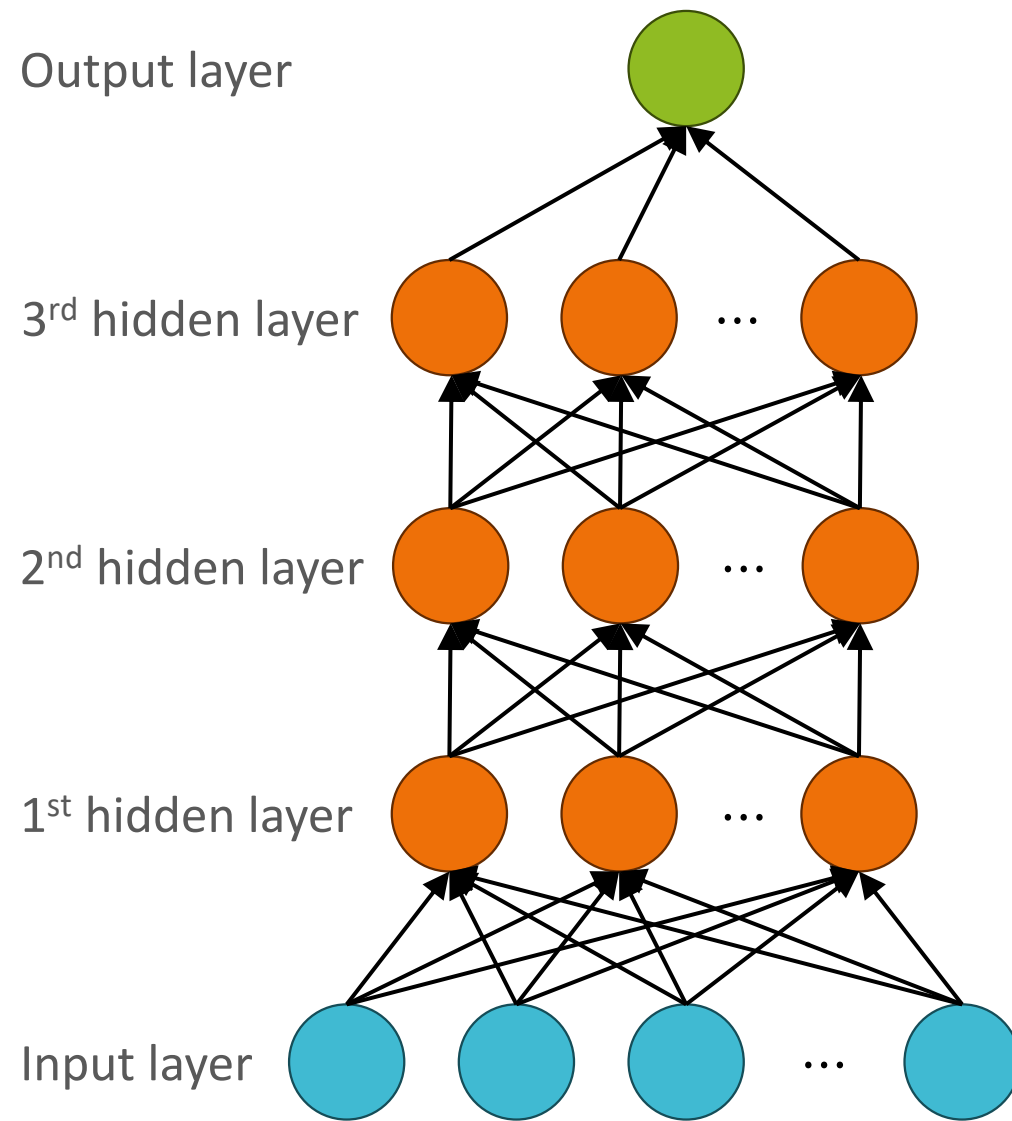
- You have some niche task that you want to apply machine learning to e.g., predicting the author of children's books
- You have a tiny labelled dataset to train with
- You fit a massive deep learning model to the dataset
- Surprise, surprise: it overfits and your test error is super high



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

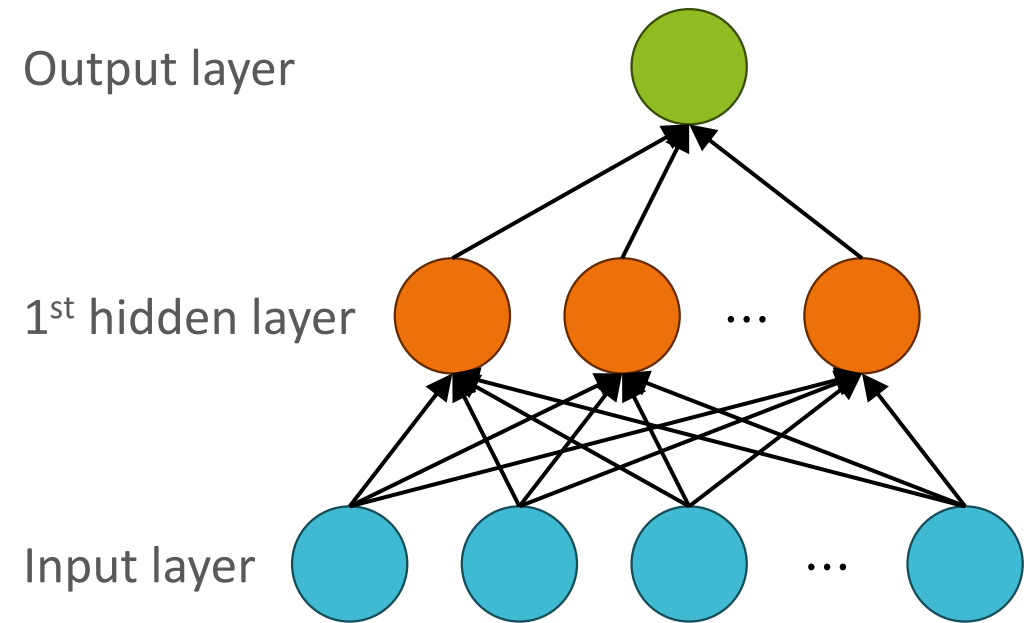
Pre-training (Bengio et al., 2006)

- 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



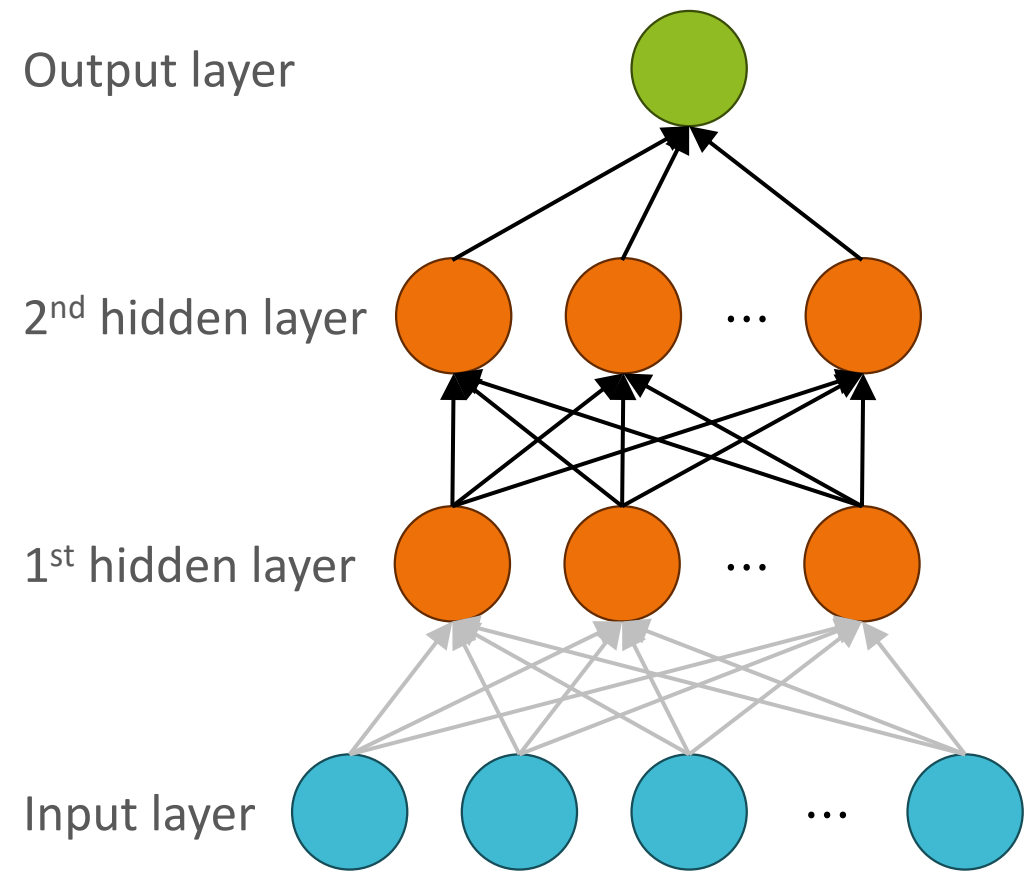
Pre-training (Bengio et al., 2006)

- 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



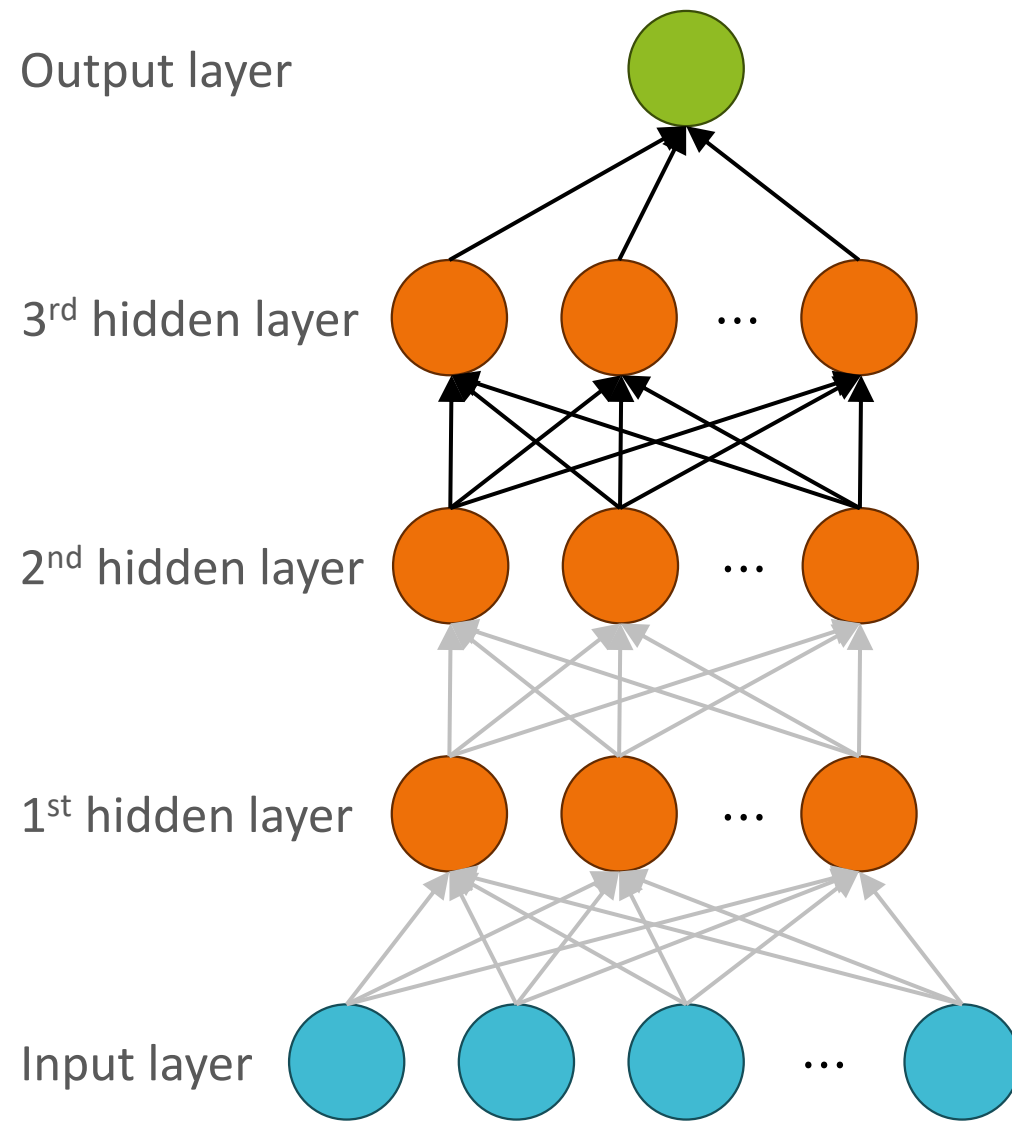
Pre-training (Bengio et al., 2006)

- 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



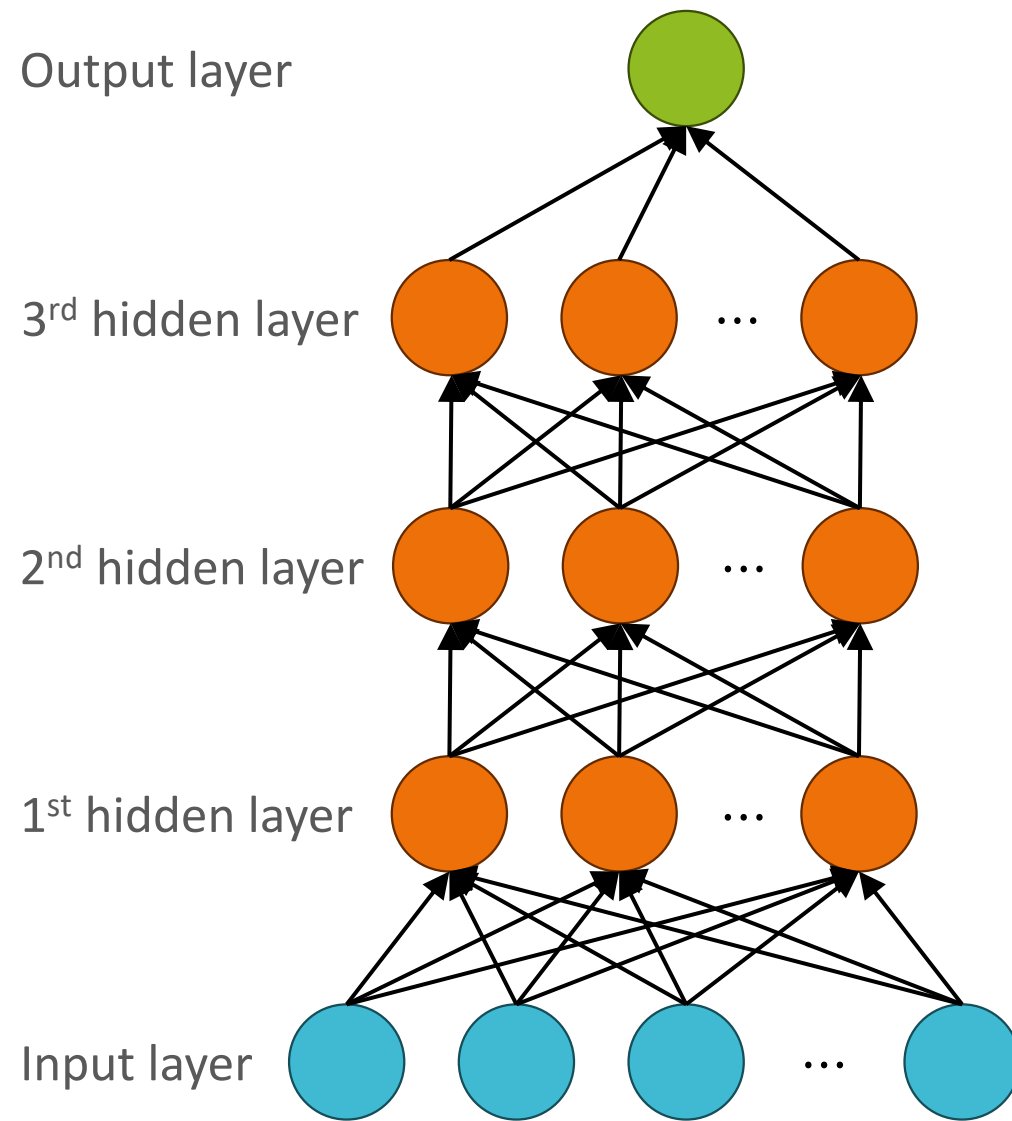
Pre-training (Bengio et al., 2006)

- 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



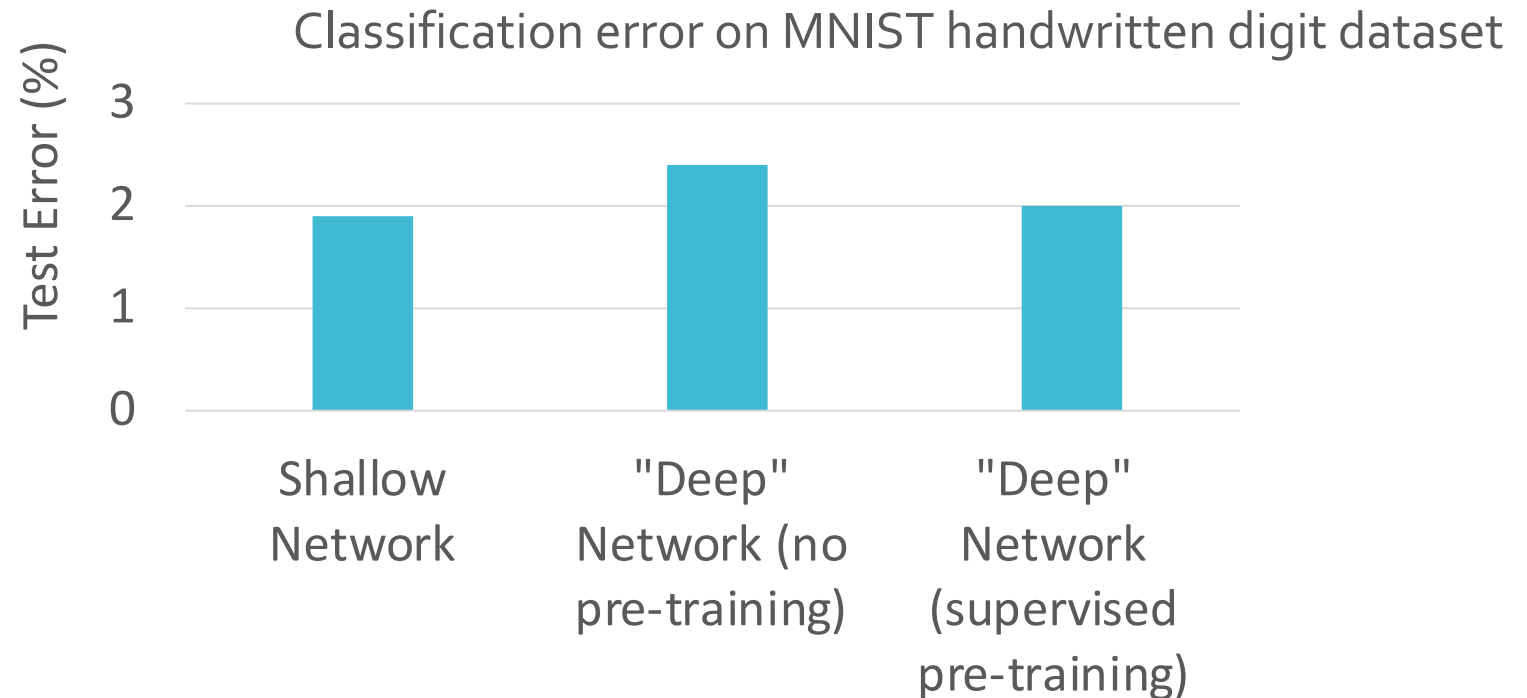
Fine-tuning (Bengio et al., 2006)

- 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



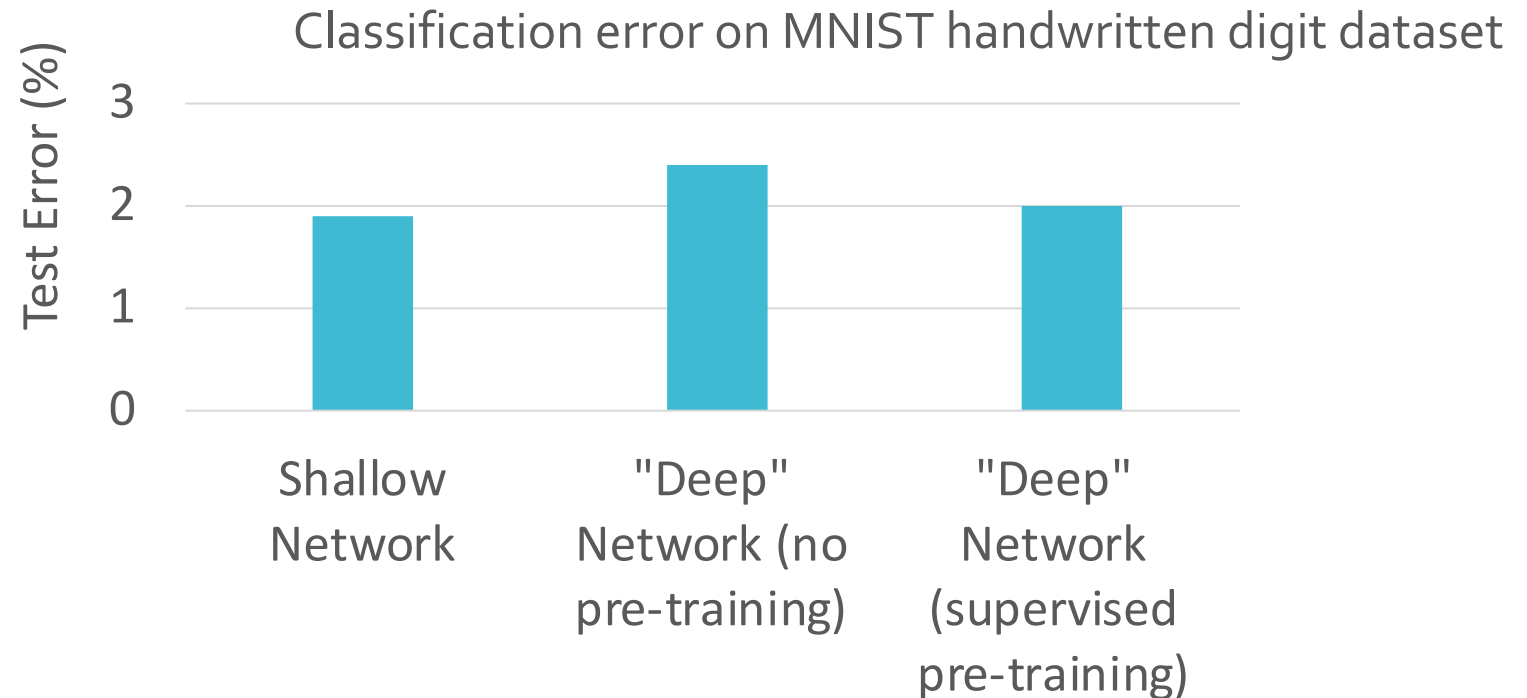
Supervised Pre-training (Bengio et al., 2006)

- 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



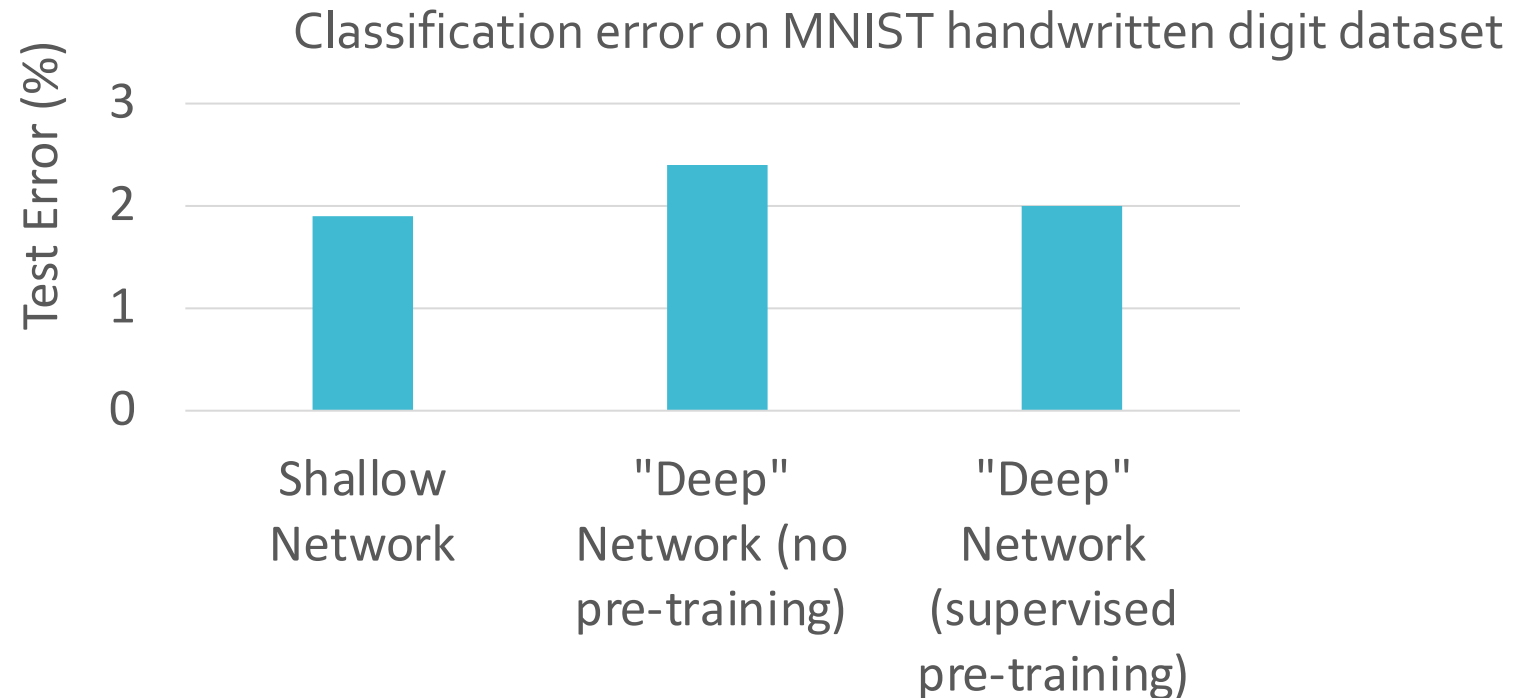
Supervised Pre-training (Bengio et al., 2006)

- 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



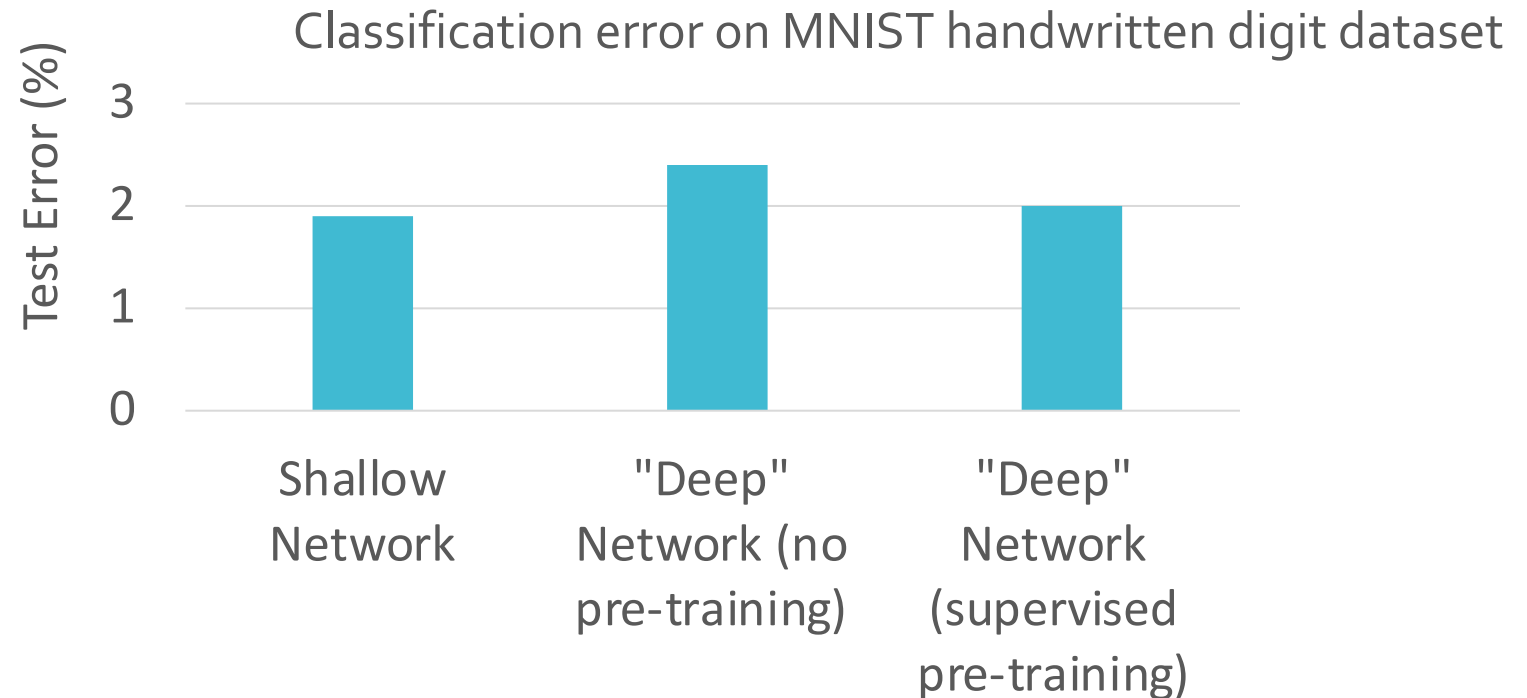
Is this the only thing we could do with the training data?

- 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)

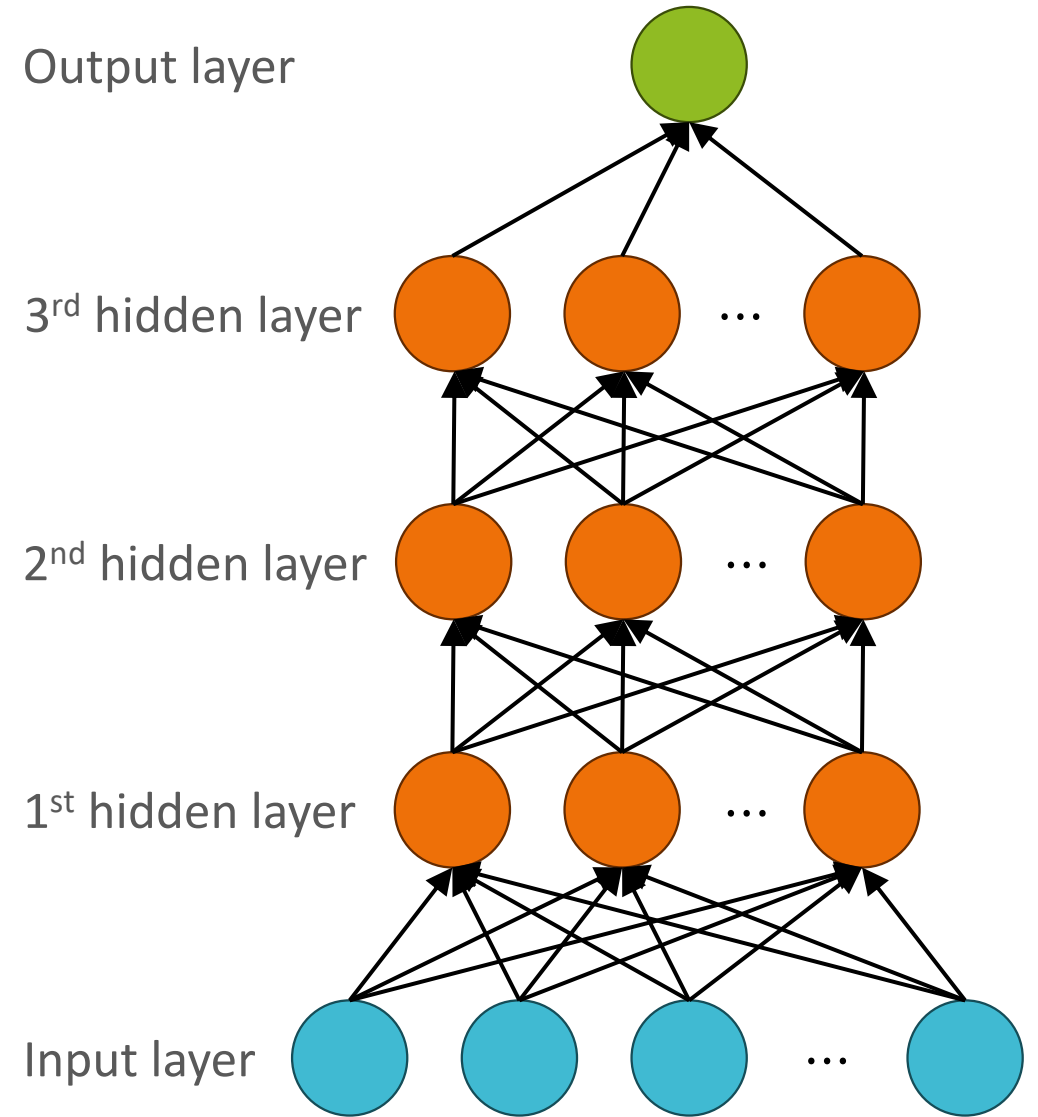
- Train each layer of the network iteratively using the training dataset *to learn useful representations*
- Idea: a good representation is one preserves a lot of information and could be used to recreate the inputs



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

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

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

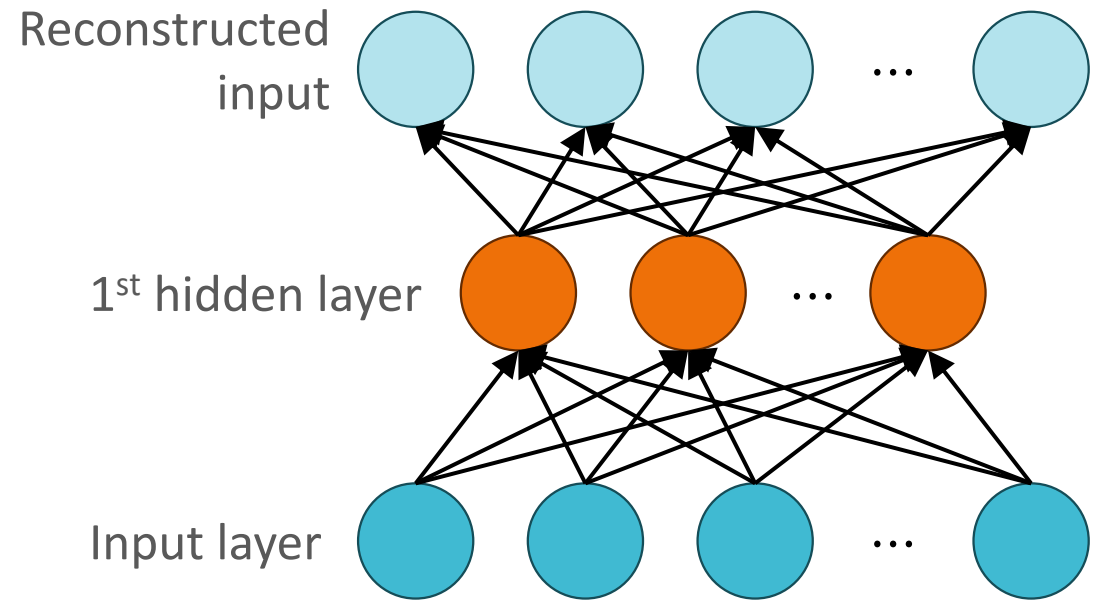


Unsupervised Pre-training (Bengio et al., 2006)

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

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

- This architecture/objective defines an *autoencoder*

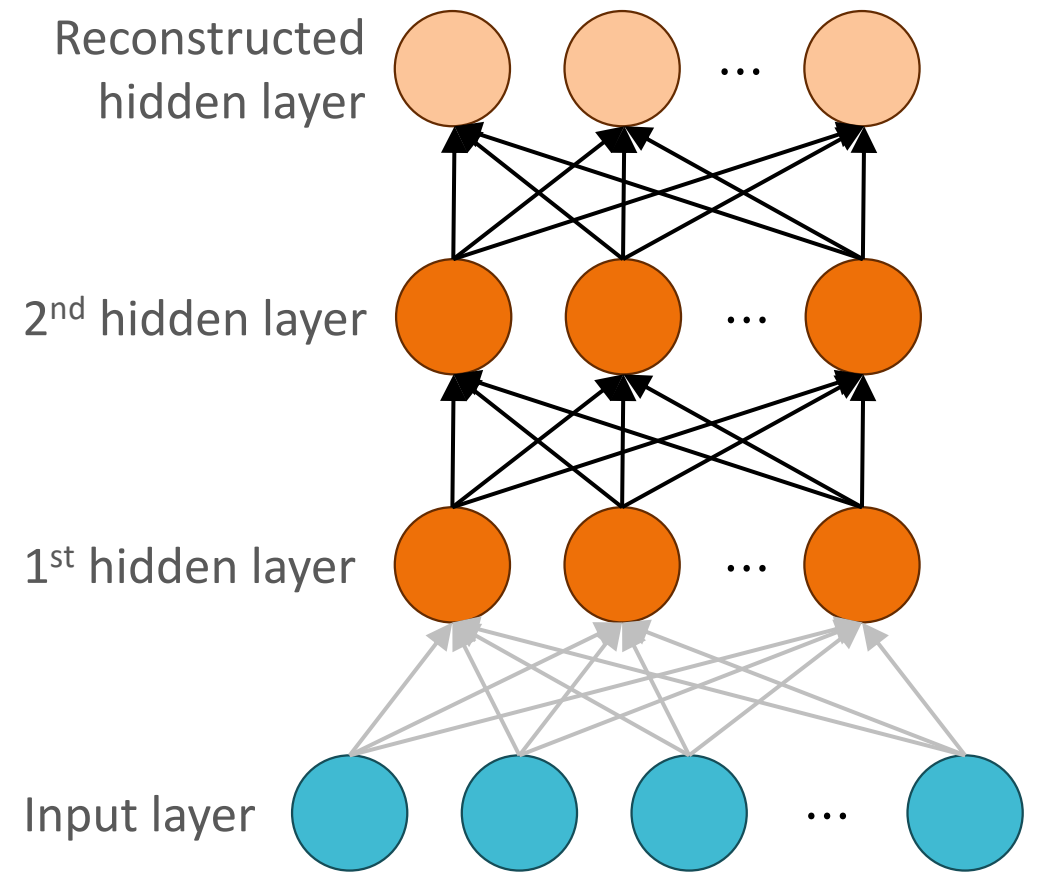


Unsupervised Pre-training (Bengio et al., 2006)

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

$$\|x - h(x)\|_2$$

- This architecture/objective defines an *autoencoder*

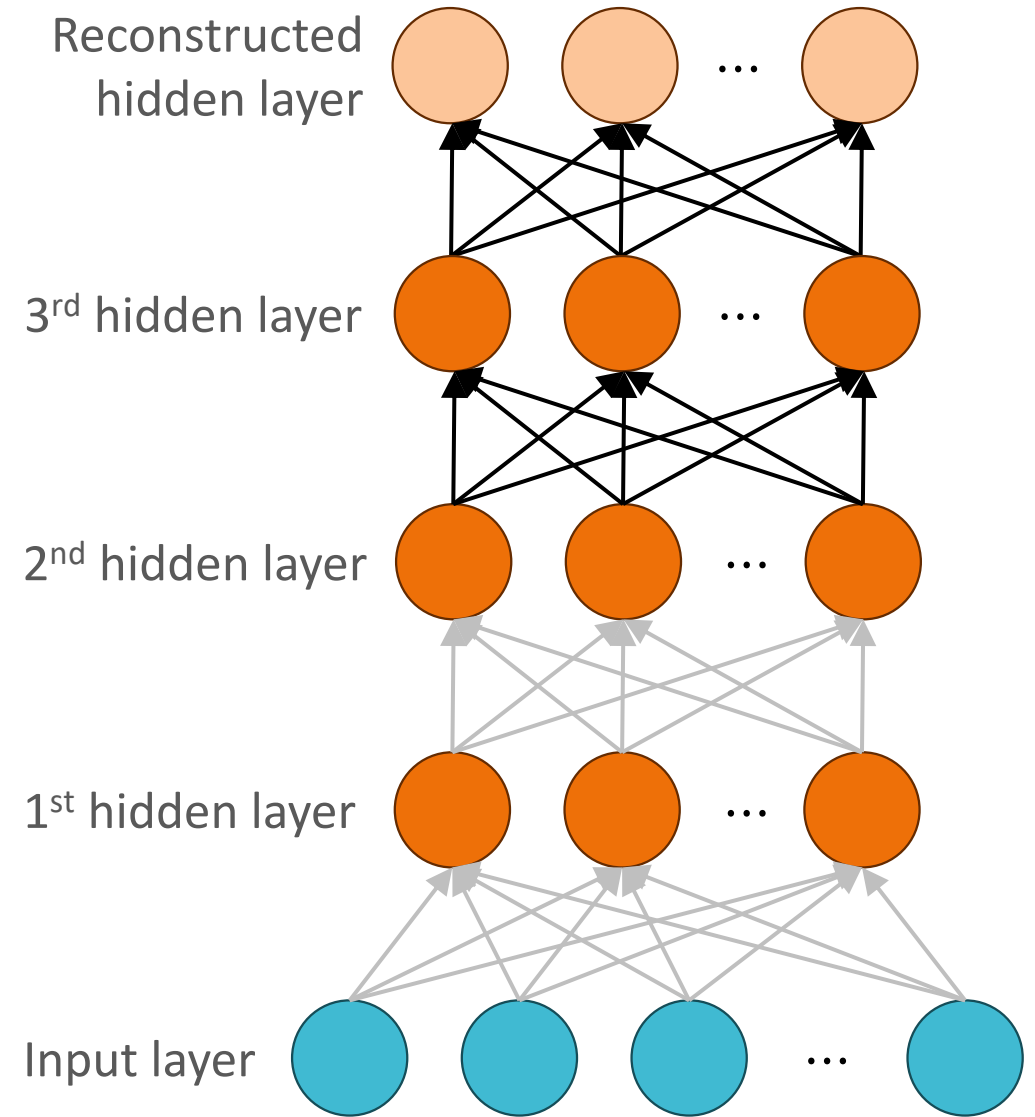


Unsupervised Pre-training (Bengio et al., 2006)

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

$$\|x - h(x)\|_2$$

- This architecture/objective defines an *autoencoder*

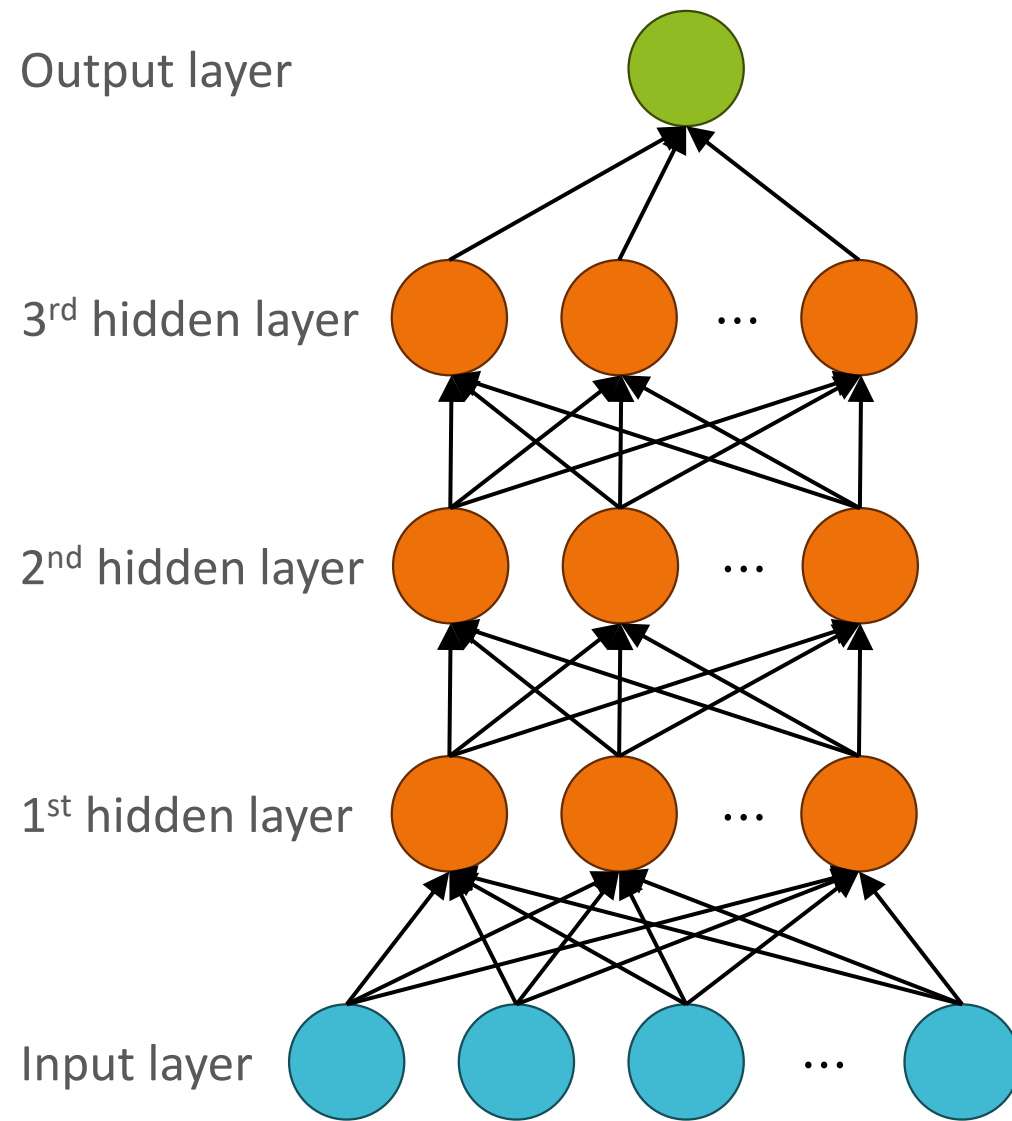


Fine-tuning (Bengio et al., 2006)

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

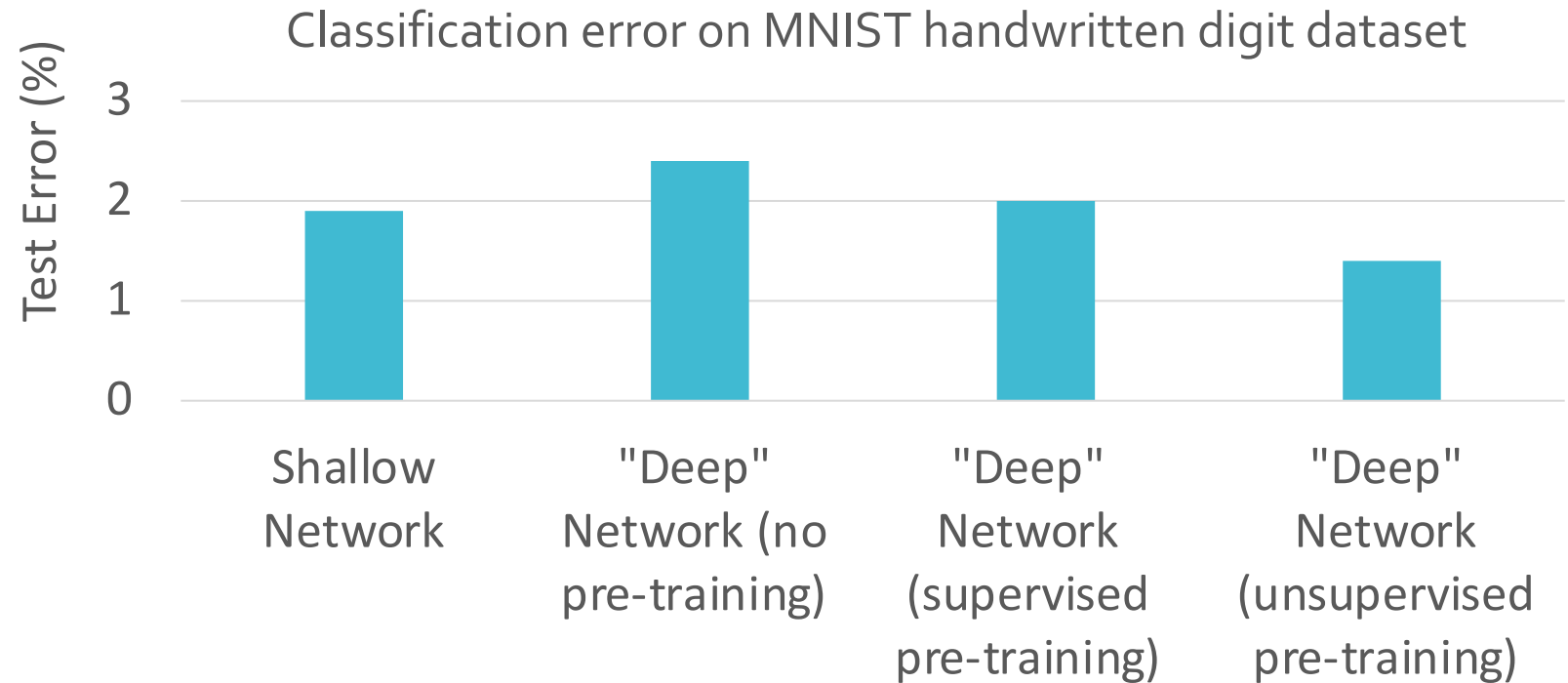
$$\|x - h(x)\|_2$$

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



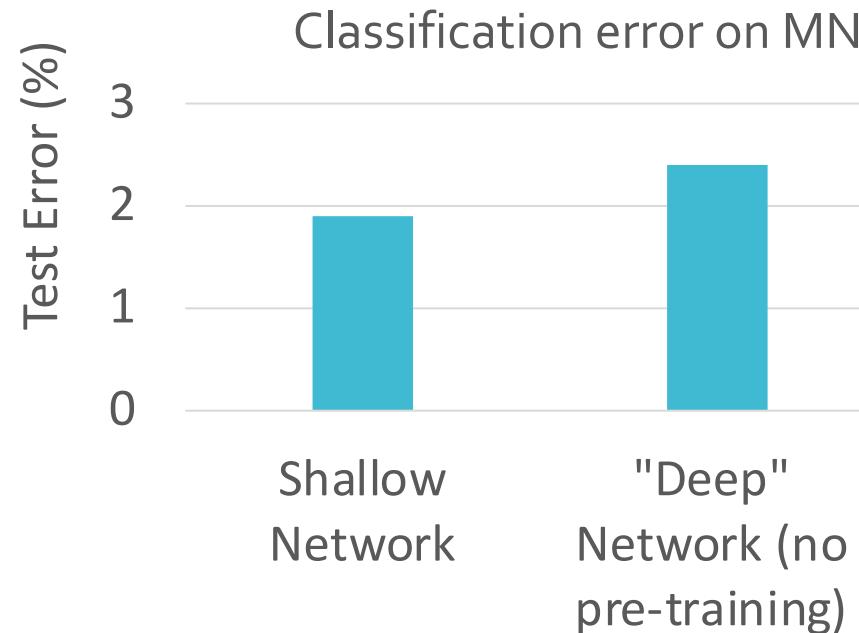
Unsupervised Pre-training (Bengio et al., 2006)

- 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. you don't have enough data to fine-tune your model?
 - B. the concept of labelled data doesn't apply to your task i.e., not every input has a "correct" label e.g., chatbots?

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)

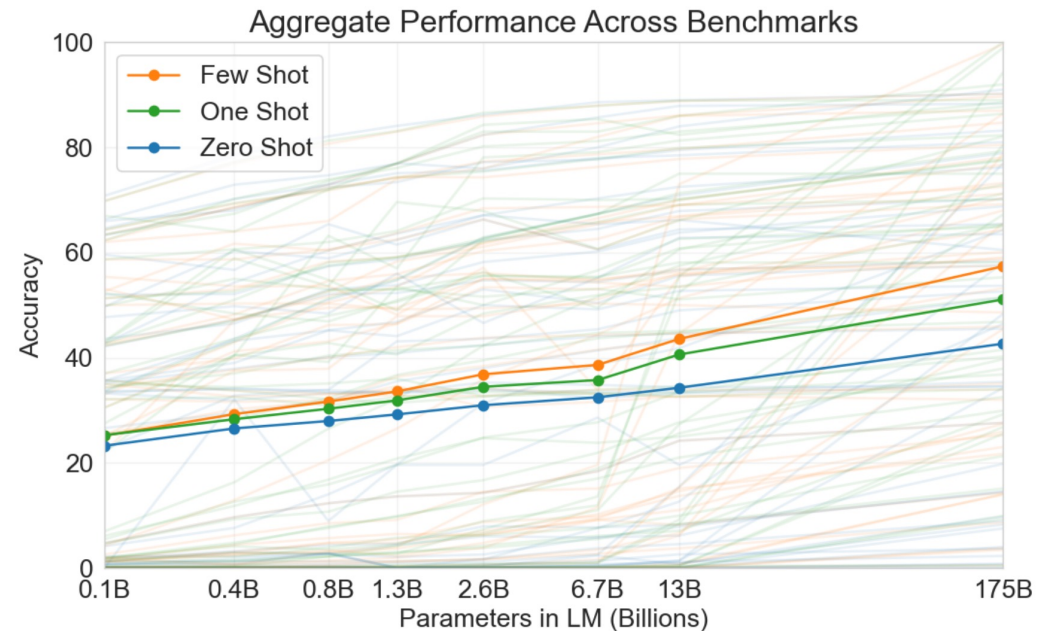
Fine-tuning

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!

Reinforcement Learning from Human Feedback (RLHF)

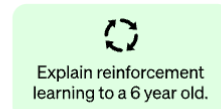
- Insight: for many machine learning tasks, there is no universal ground truth, e.g., there are lots of possible ways to respond to a question or prompt.
- Idea: use human feedback to determine how good or bad some prediction/response is!
- Issue: if the input space is huge (e.g., all possible chat prompts), 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.

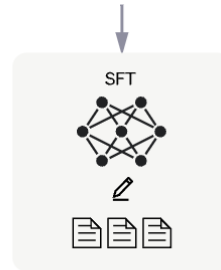
A prompt is sampled from our prompt dataset.



A labeler demonstrates the desired output behavior.



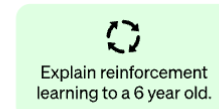
This data is used to fine-tune GPT-3.5 with supervised learning.



Step 2

Collect comparison data and train a reward model.

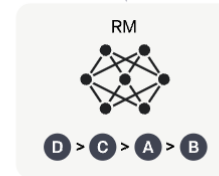
A prompt and several model outputs are sampled.



A labeler ranks the outputs from best to worst.



This data is used to train our reward model.



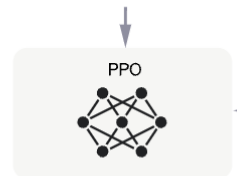
Step 3

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

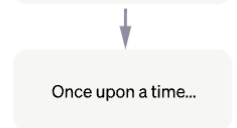
A new prompt is sampled from the dataset.



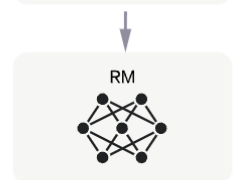
The PPO model is initialized from the supervised policy.



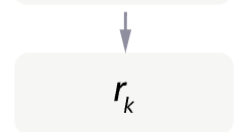
The policy generates an output.



The reward model calculates a reward for the output.



The reward is used to update the policy using PPO.



- RLHF is a form of fine-tuning that uses reinforcement learning where the reward function is learned from human preferences

What the heck is “Reinforcement Learning”?

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.

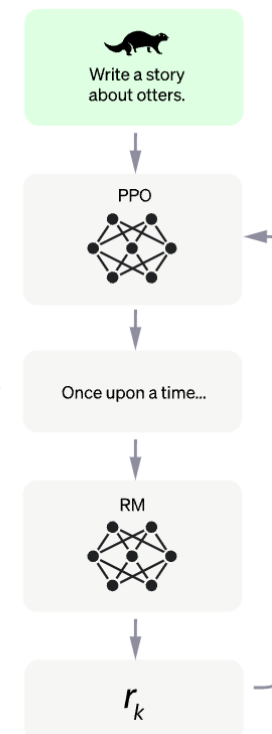
A new prompt is sampled from the dataset.

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 form of fine-tuning that uses *reinforcement learning* where the reward function is learned from human preferences