ENGR-E 533 “Deep Learning Systems” Lecture 05: Vanishing Gradients and Pretraining

Minje Kim
Department of Intelligent Systems Engineering
Email: minje@indiana.edu
Website: http://minjekim.com
Research Group: http://saige.sice.indiana.edu
Meeting Request: http://doodle.com/minje

INDIANA UNIVERSITY
SCHOOL OF INFORMATICS, COMPUTING, AND ENGINEERING

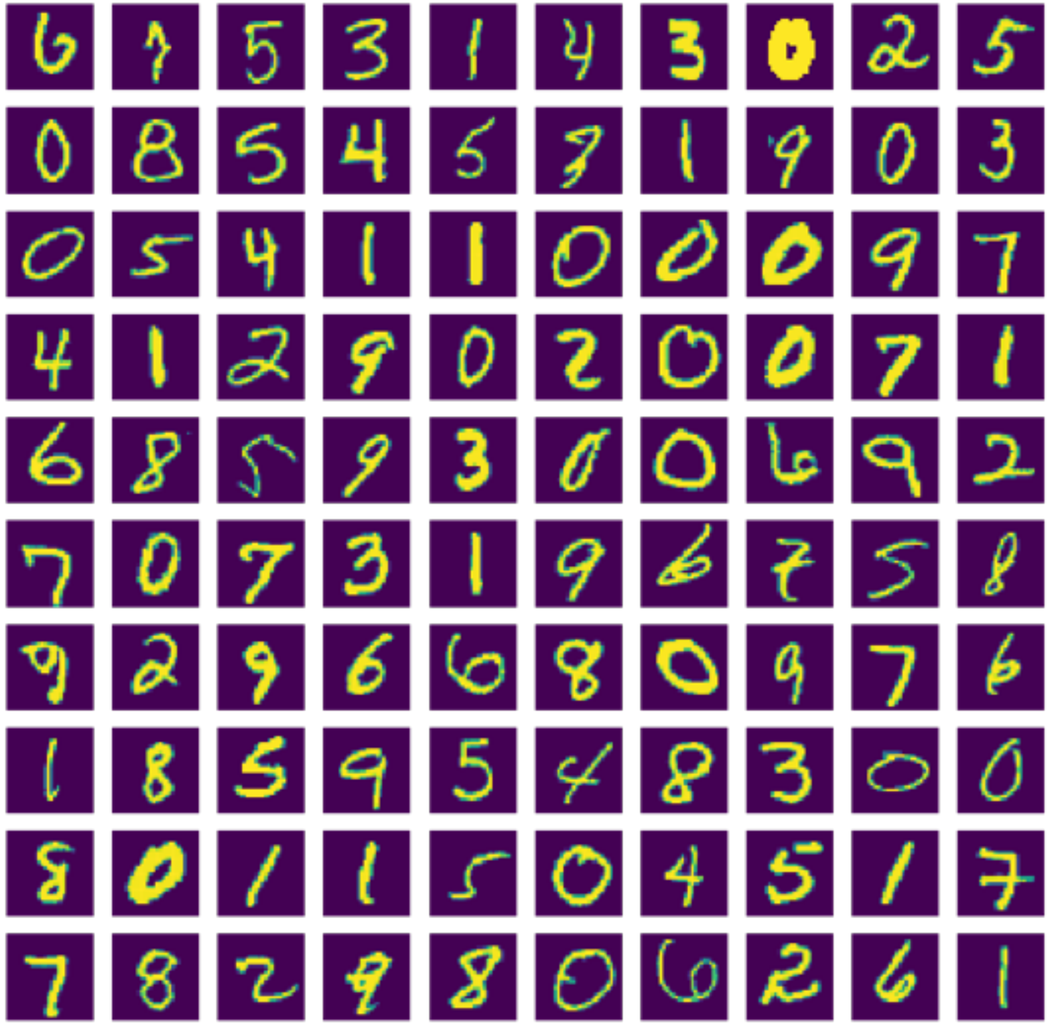
What We've Learned So Far

- Feature extraction is important
  - But difficult
  - Usually involves unsupervised learning
- Supervised learning
  - Is easy if you start from a good set of features
  - Otherwise you'll need some nonlinearity
- A shallow neural network takes care of both tasks at the same time
  - The first layer learns features
  - The last layer learns the regressor
- Universal Approximation Theorem
  - If there are many many hidden units, a shallow neural network can approximate any (nonlinear) function
  - e.g. Sinusoidal functions
  - But how many?

A Little More Difficult Problem

-MNIST handwritten digit recognition

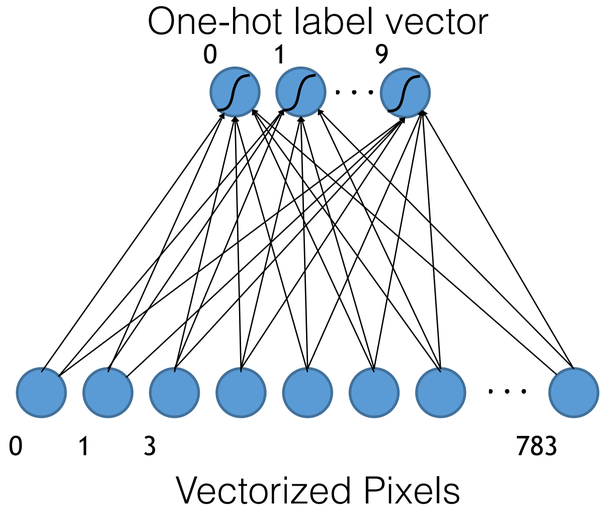
- Considered as the “Hello, World!” problem in deep learning
- 28X28=784 pixels per image
- 55,000 training images
- 10,000 testing images

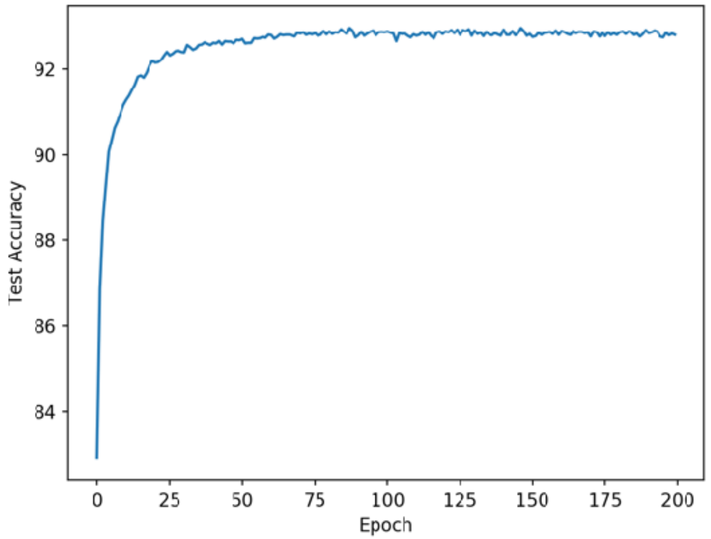


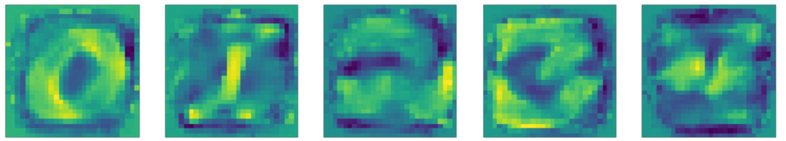
A Little More Difficult Problem

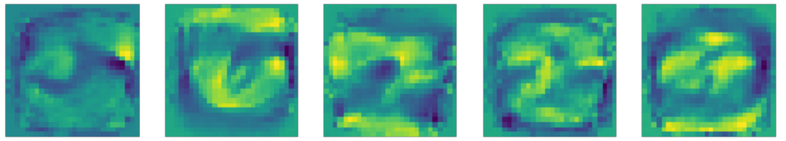
-MNIST handwritten digit recognition

- A softmax classification with no feature extraction
  - i.e. Raw pixel intensities are the features

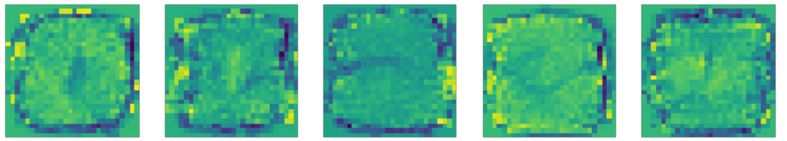


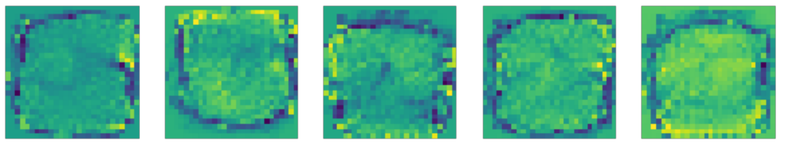






Weights at 20th epoch





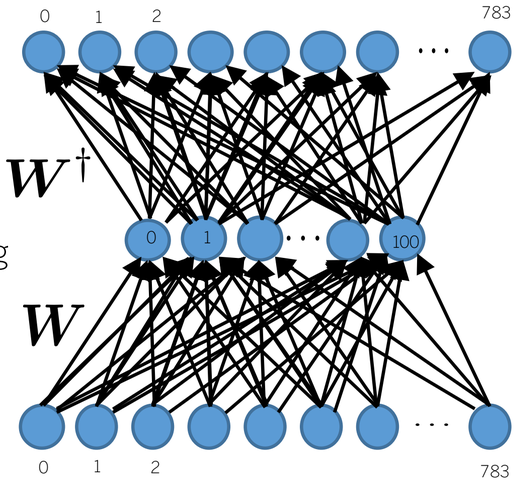
Weights at 200th epoch

- It's not bad, but how about some feature extraction?
  - YALT last time

Feature Extraction

-Sparse AE

- Let's extract some features, and then do the softmax
  - Two cases: naïve AE versus sparse AE
  - The AE objective function $\arg\min_{\boldsymbol{W},\boldsymbol{H}} \mathcal{D}(\boldsymbol{X}||\boldsymbol{W}^{\dagger}\boldsymbol{H}) + \lambda g(\boldsymbol{H}), \quad \text{where} \boldsymbol{H} = \sigma(\boldsymbol{W}\boldsymbol{X})$



Unsupervised Feature Learning

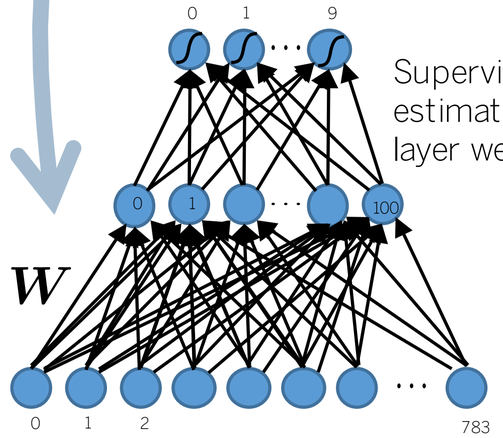
- With the sparsity constraint

$$\hat{\alpha} = \frac{1}{T}\sum_{j=0}^{T-1} \boldsymbol{H}_{i,j}$$ ← Average activation (logistic)

$$g(\boldsymbol{H}_{i,:}) = KL(\alpha||\hat{\alpha}) = -\alpha \log\left(\frac{\alpha}{\hat{\alpha}}\right) - (1-\alpha)\log\left(\frac{1-\alpha}{1-\hat{\alpha}}\right)$$ ← Desired probability of activations

$$g(\boldsymbol{H}) = \sum_i g(\boldsymbol{H}_{i,:})$$

Initialize the weight matrix with the learned one and don't update



Supervised learning estimates the softmax layer weights

- For example, $x = [1,0,1,0,0,0,0,0,0,0]$ $\hat{\alpha} = 0.2$ If $\alpha = 0.2$ $g(x) = 0$

  $x = [0.8,0,0.9,0,0.2,0,0.1,0,0,0]$ $\hat{\alpha} = 0.2$

- A less sparse one

  $x = [0.8,1,0.9,1,0.2,0,0.1,0,0,0]$ $g(x) = 0.092$

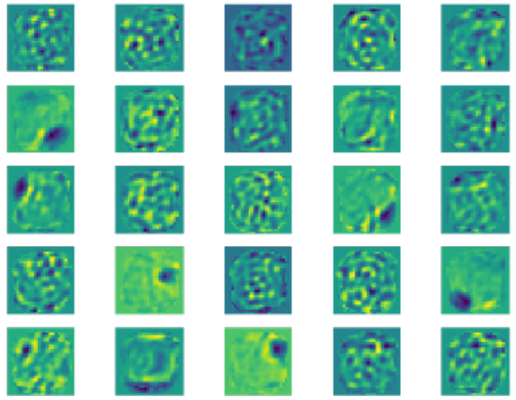
  $\hat{\alpha} = 0.4$

Feature Extraction

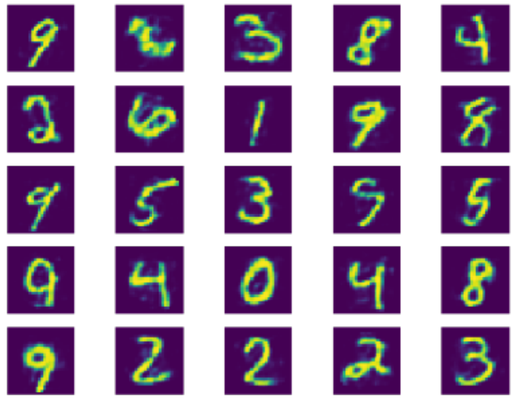
-Sparse AE

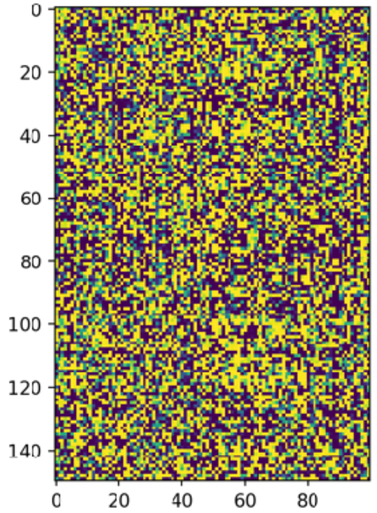
- Case A: Naïve AE
  - No sparsity constraint

Weights



Recons.

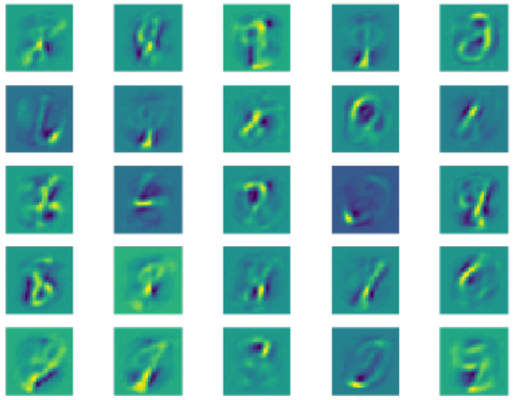




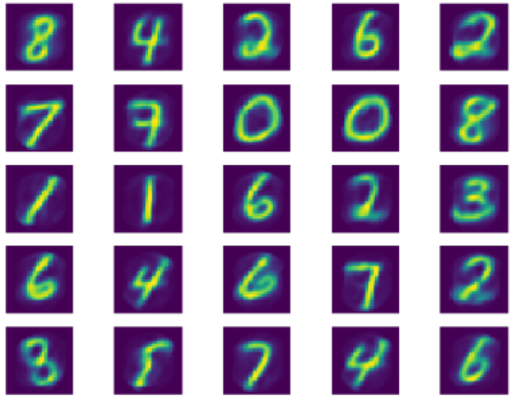
Hidden unit activations

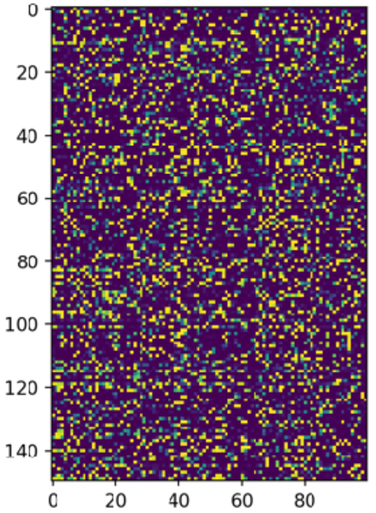
- Case B: Sparse AE
  - λ=0.1, α=0.2

Weights



Recons.



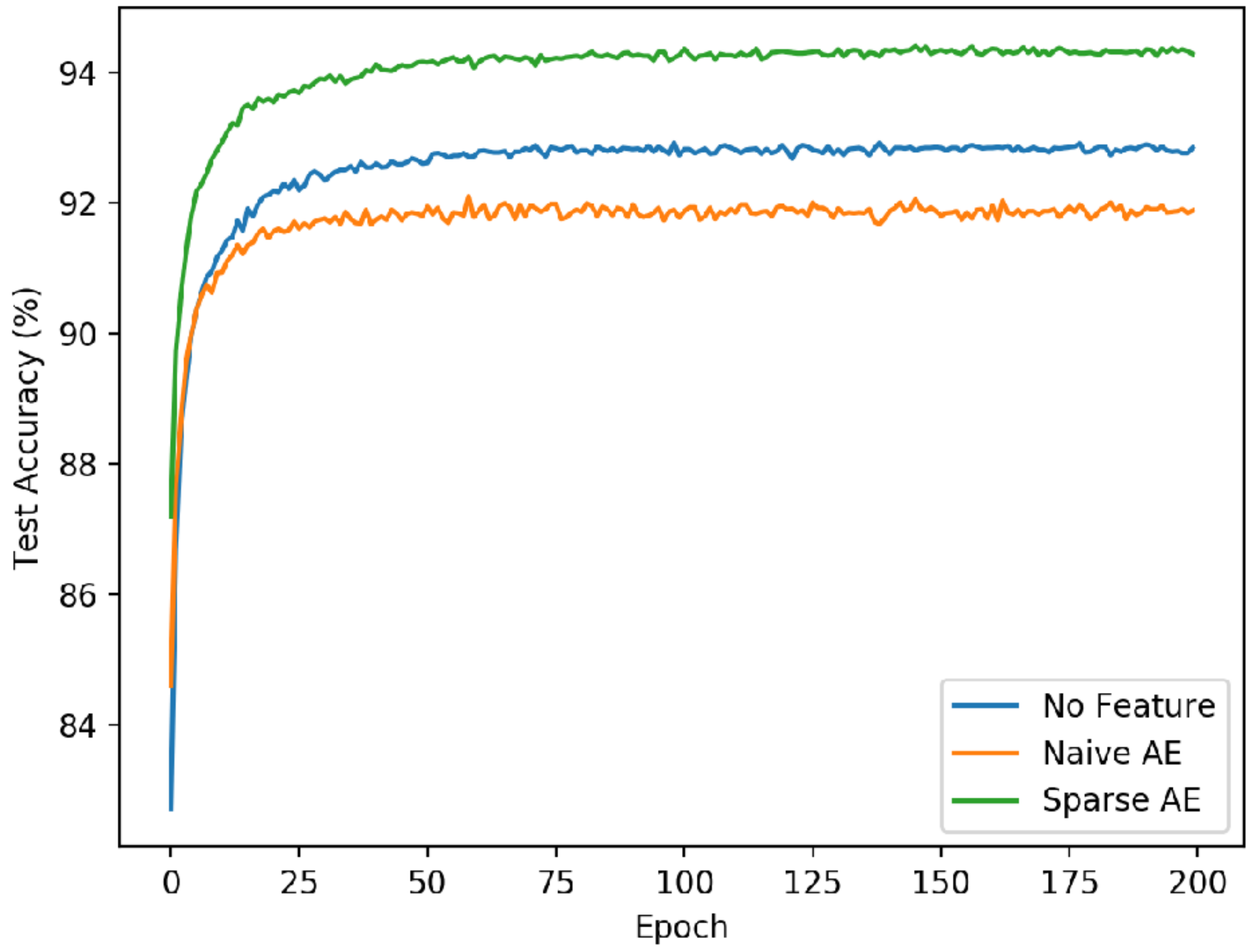


Hidden unit activations

Feature Extraction + Softmax

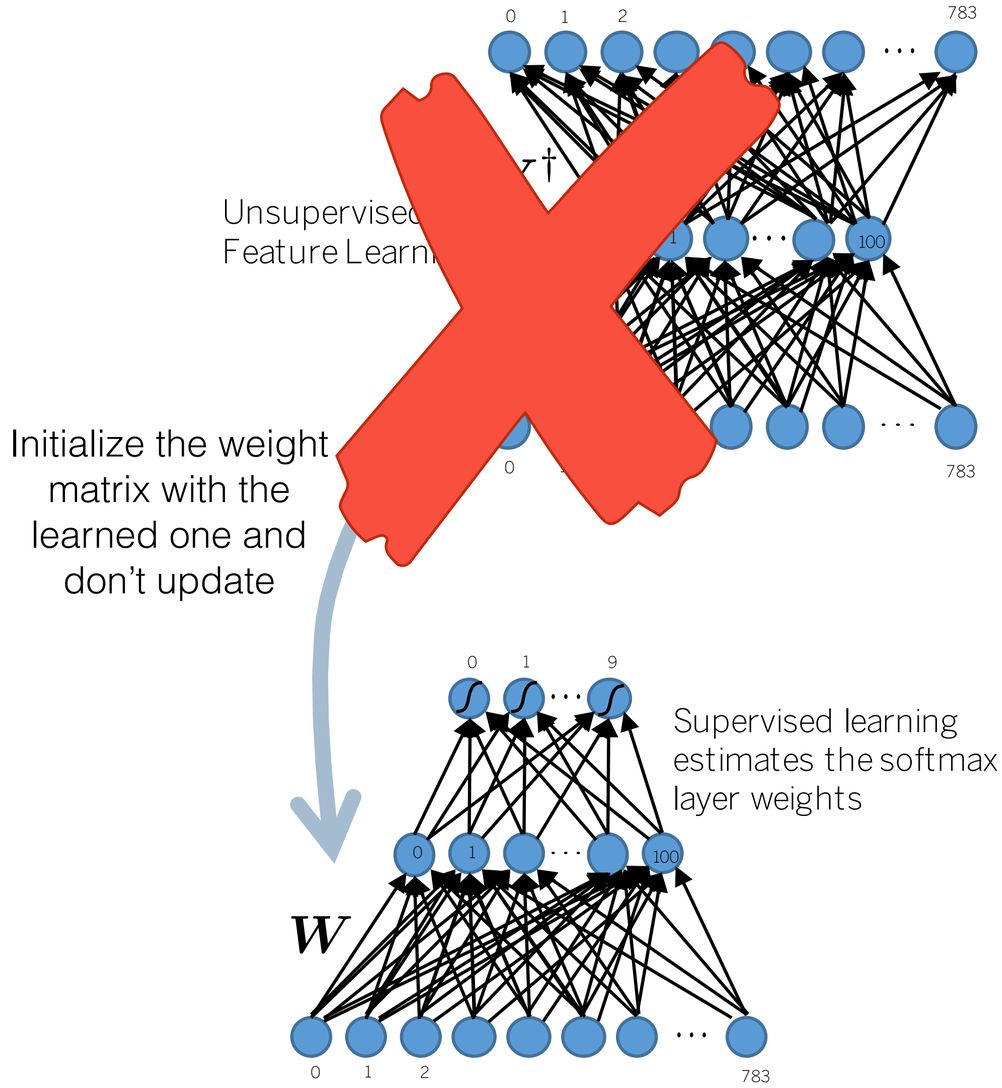
- Any better than the raw pixels?

- A direct softmax classification on the raw pixel intensities (blue line)
  - Works pretty well
  - No features
- If your features are NOT good enough (naïve AE)
  - They degrade the performance
- If your features are good enough (sparse AE)
  - They improve the performance



Feature Extraction + Softmax?

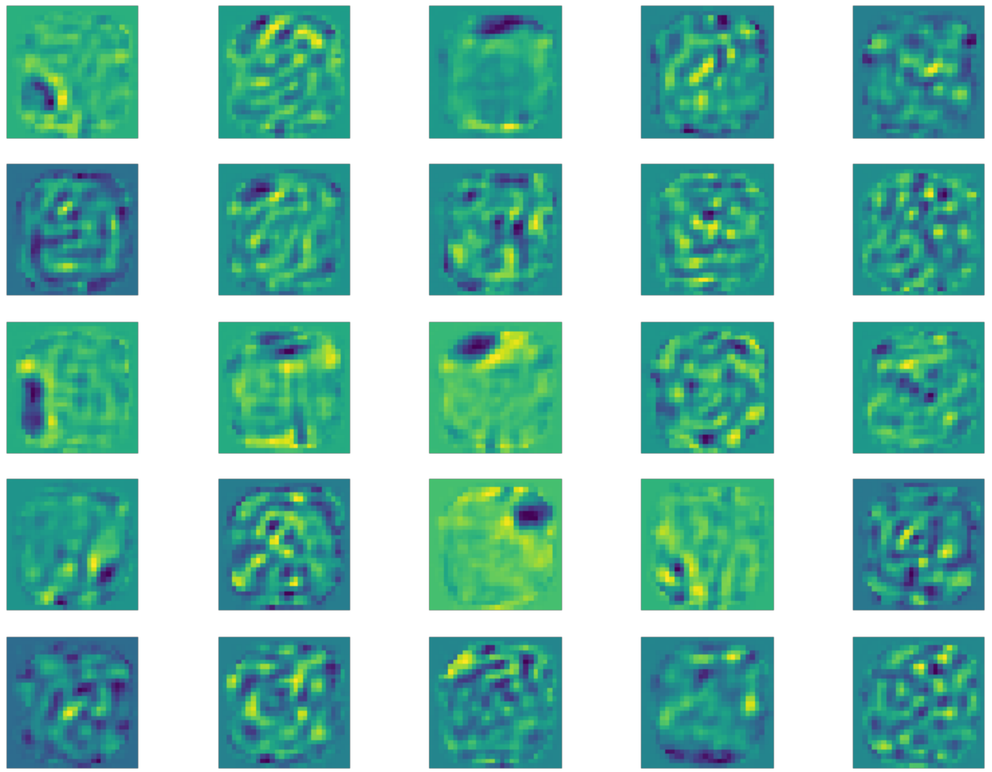
- Big question: why bother learning the features?
  - Feature learning suffers from too many different decisions to make
    - e.g. regularization, constraints, ranks, error functions, activation functions, etc.
    - Could be unpredictable until you go ahead and test them out
  - Supervised learning has a lot of options, too
    - e.g. choice of the kernel functions, generative models, hyperparameters, etc.
- We can learn a neural network that has one hidden layer
  - Randomly initialize both weight matrices in the two layers
  - Estimate them using backpropagation
  - Does it work better than the previous methods with feature engineering?
    - See the graph in the next page



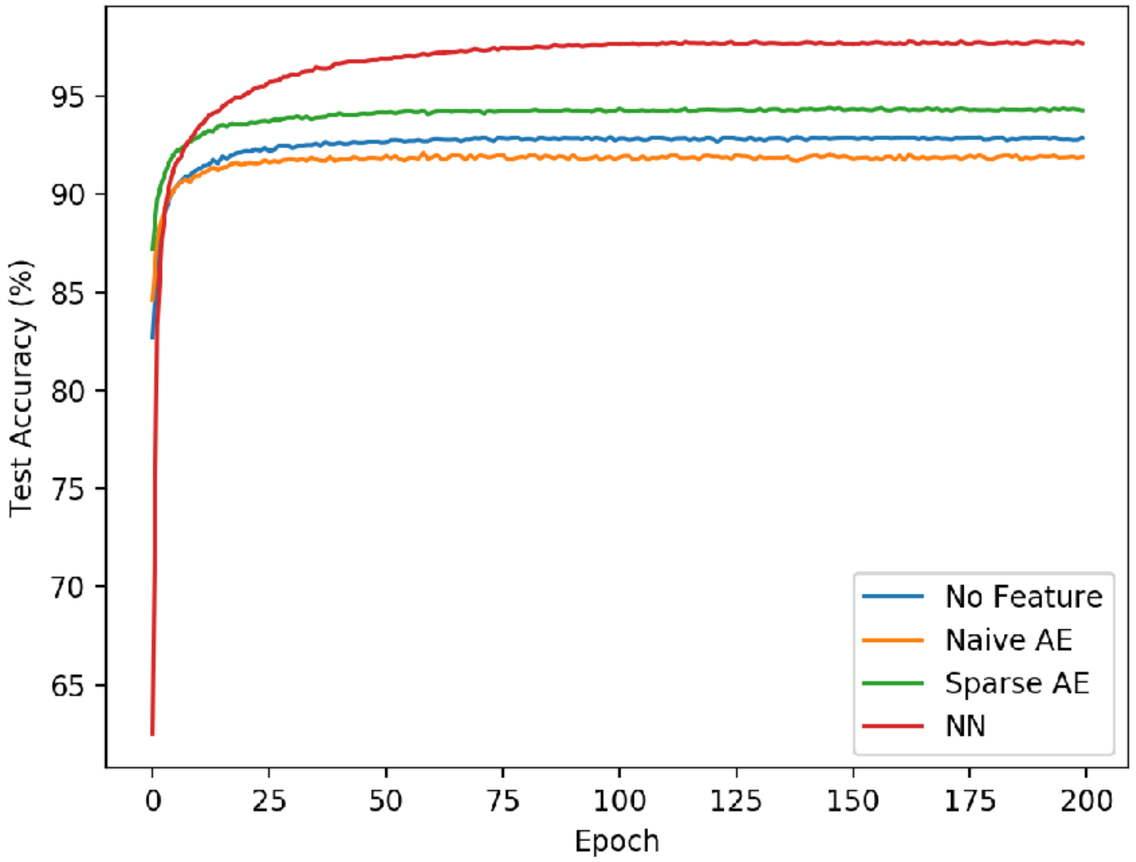
Shallow Neural Networks

-No feature learning, but a hidden layer

- Feature learning is absorbed in the NN learning process



First Layer Weights



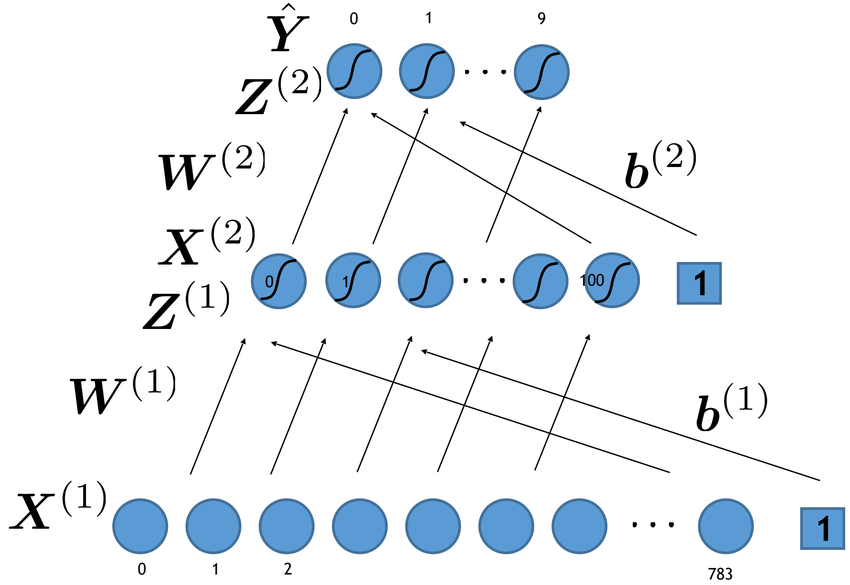
Backpropagation (Revisited)

-Now that we know the first layer weights should be updated, too

- i=0
  - Initialize parameters with small random numbers
  - Calculate the prediction using all training samples (i.e. input and target pairs)

$$\boldsymbol{Z}^{(1)} \leftarrow \boldsymbol{W}^{(1)}\boldsymbol{X}^{(1)} + \boldsymbol{b}^{(1)} \qquad \boldsymbol{X}^{(2)} \leftarrow \sigma(\boldsymbol{Z}^{(1)})$$

$$\boldsymbol{Z}^{(2)} \leftarrow \boldsymbol{W}^{(2)}\boldsymbol{X}^{(2)} + \boldsymbol{b}^{(2)} \qquad \hat{\boldsymbol{Y}} \leftarrow g(\boldsymbol{Z}^{(2)})$$



  - Calculate the error (cost)

$$\mathcal{E} = \sum_t \mathcal{D}(\boldsymbol{Y}_{:,t} || \hat{\boldsymbol{Y}}_{:,t})$$

  - Update the parameters

$$\boldsymbol{W}^{(2)} \leftarrow \boldsymbol{W}^{(2)} - \rho\frac{\partial\mathcal{E}}{\partial\boldsymbol{W}^{(2)}} \qquad \boldsymbol{W}^{(1)} \leftarrow \boldsymbol{W}^{(1)} - \rho\frac{\partial\mathcal{E}}{\partial\boldsymbol{W}^{(1)}}$$

$$\boldsymbol{b}^{(2)} \leftarrow \boldsymbol{b}^{(2)} - \rho\frac{\partial\mathcal{E}}{\partial\boldsymbol{b}^{(2)}} \qquad \boldsymbol{b}^{(1)} \leftarrow \boldsymbol{b}^{(1)} - \rho\frac{\partial\mathcal{E}}{\partial\boldsymbol{b}^{(1)}}$$

- i>0
  - Repeat 2-4

Backpropagation (Revisited)

-Now that we know the first layer weights should be updated, too

$$\boldsymbol{Z}^{(1)} \leftarrow \boldsymbol{W}^{(1)}\boldsymbol{X}^{(1)} + \boldsymbol{b}^{(1)}$$
$$\boldsymbol{X}^{(2)} \leftarrow \sigma(\boldsymbol{Z}^{(1)})$$
$$\boldsymbol{Z}^{(2)} \leftarrow \boldsymbol{W}^{(2)}\boldsymbol{X}^{(2)} + \boldsymbol{b}^{(2)}$$
$$\hat{\boldsymbol{Y}} \leftarrow g(\boldsymbol{Z}^{(2)})$$

- For the last layer

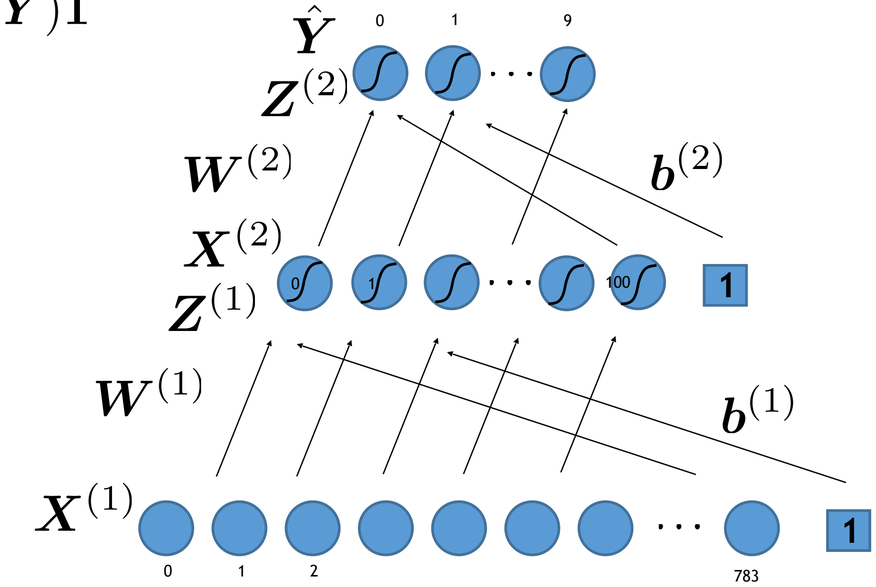
$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(2)}} = \underbrace{\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{Y}}}\frac{\partial \hat{\boldsymbol{Y}}}{\partial \boldsymbol{Z}^{(2)}}}_{\text{BP Error } \boldsymbol{\Delta}^{(2)}}\frac{\partial \boldsymbol{Z}^{(2)}}{\partial \boldsymbol{W}^{(2)}} = \underbrace{(\hat{\boldsymbol{Y}} - \boldsymbol{Y})}_{\text{BP Error } \boldsymbol{\Delta}^{(2)}}\boldsymbol{X}^{(2)\top} \qquad \frac{\partial \mathcal{E}}{\partial \boldsymbol{b}^{(2)}} = (\hat{\boldsymbol{Y}} - \boldsymbol{Y})\mathbf{1}$$

  - Backpropagation Error $\boldsymbol{\Delta}^{(l)}$:
    the derivative of the cost function w.r.t. the input to the (l)-th hidden layer unit

- For the first layer

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(1)}} = \underbrace{\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{Y}}}\frac{\partial \hat{\boldsymbol{Y}}}{\partial \boldsymbol{Z}^{(2)}}}_{\text{BP Error } \boldsymbol{\Delta}^{(2)}}\underbrace{\frac{\partial \boldsymbol{Z}^{(2)}}{\partial \boldsymbol{X}^{(2)}}\frac{\partial \boldsymbol{X}^{(2)}}{\partial \boldsymbol{Z}^{(1)}}}_{\text{BP Error } \boldsymbol{\Delta}^{(1)}}\frac{\partial \boldsymbol{Z}^{(1)}}{\partial \boldsymbol{W}^{(1)}}$$

$$= \underbrace{\left(\left(\boldsymbol{W}^{(2)\top}(\hat{\boldsymbol{Y}} - \boldsymbol{Y})\right) \odot \sigma'(\boldsymbol{Z}^{(1)})\right)}_{\text{BP Error } \boldsymbol{\Delta}^{(1)}}\boldsymbol{X}^{(1)\top} \qquad \frac{\partial \mathcal{E}}{\partial \boldsymbol{b}^{(1)}} = \boldsymbol{\Delta}^{(1)}\mathbf{1}$$



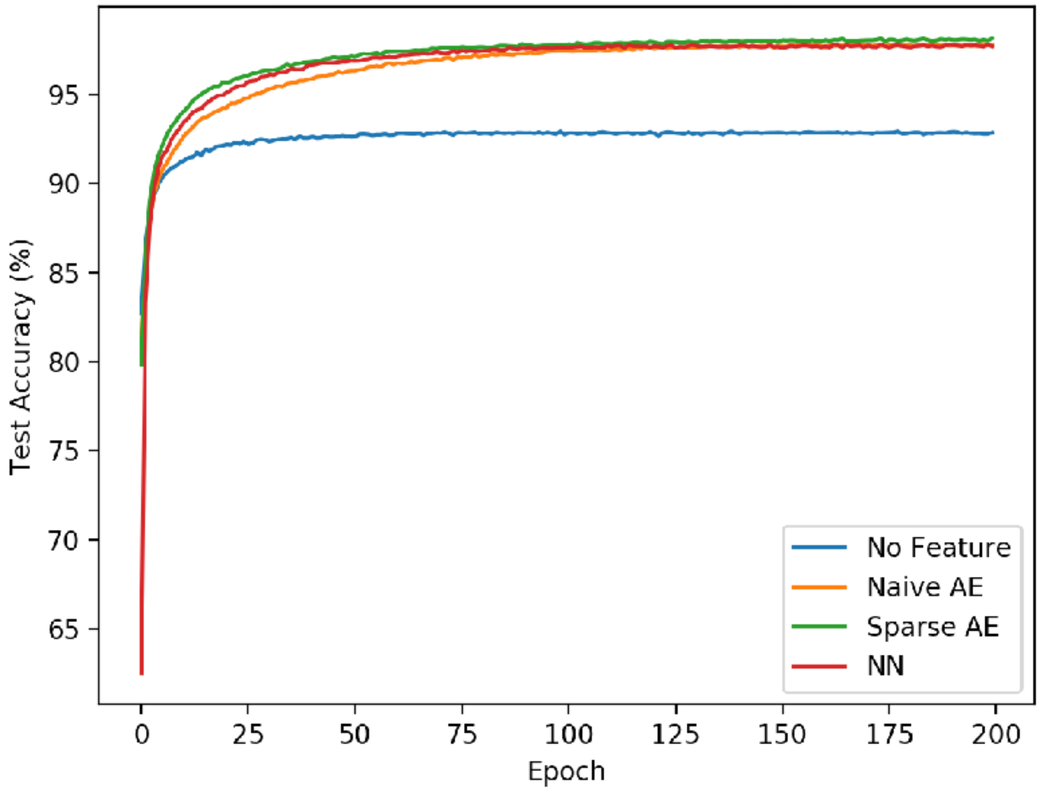
- The gradient for $\boldsymbol{W}^{(l)}$ is [BP Error at (l)] X [Input from the previous layer]
- $\boldsymbol{\Delta}^{(l)}$: [BP Error at (l+1)] X [Weights at (l+1)] X [Derivative of the activation at (l)]

A Step Towards Deep Neural Networks?

-BP on DNNs?

- We noticed that a separate feature learning procedure is NOT super helpful for softmax classification
  - A direct BP down to the first layer is doable and works better
    - Why is it better?
    - The feature learning part (i.e. the first layer weights) is aware of the discrimination objective
  - It might NOT be true if the last layer is more complicated (e.g. SVM)
    - Maybe the features don't really have to be that great
- Another option
  - What if we initialize the first layer with the trained AE weights, and then update weights of both layers?
    - See the graph
  - A better initialization doesn't seem to contribute to the convergence (why?)
    - BP just gets down to the first layer
- Conclusion (so far): we don't need pretraining for shallow NNs
- What if we make the network deeper?

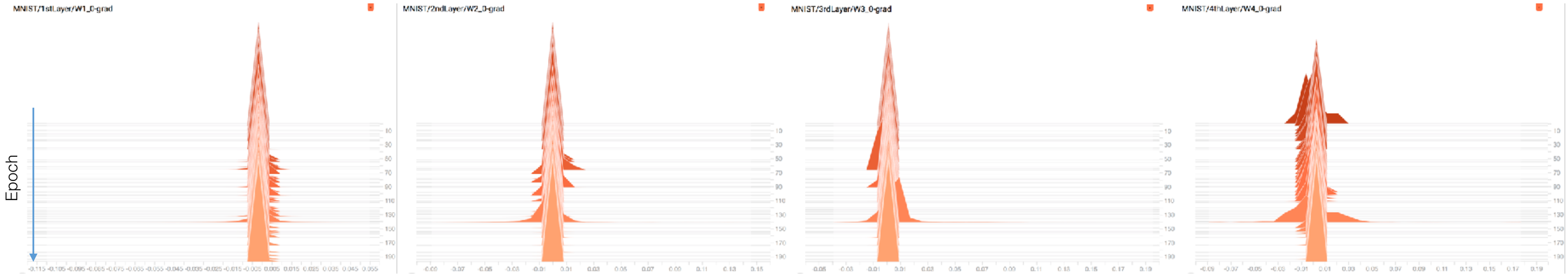
Full BP training of shallow net with one hidden layer (different initialization schemes)

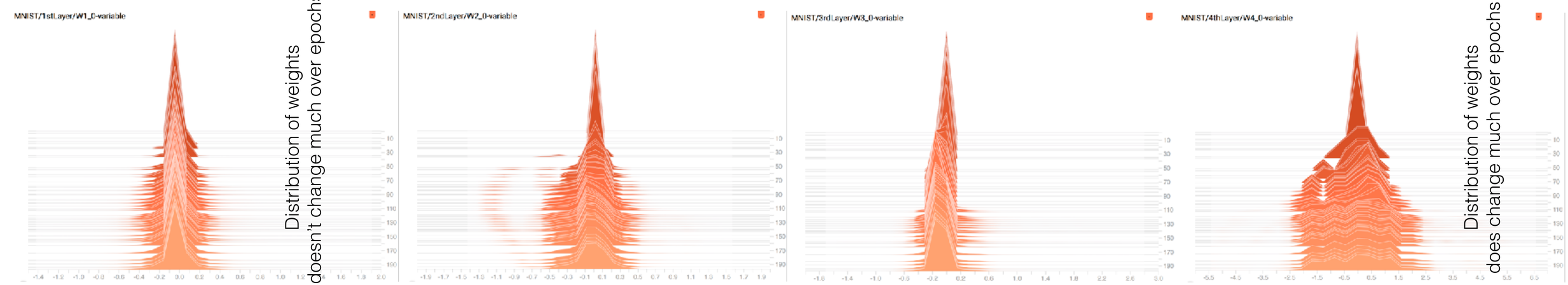


Vanishing Gradients

-BP doesn’t get down to the first layer

- The vanishing gradient problem

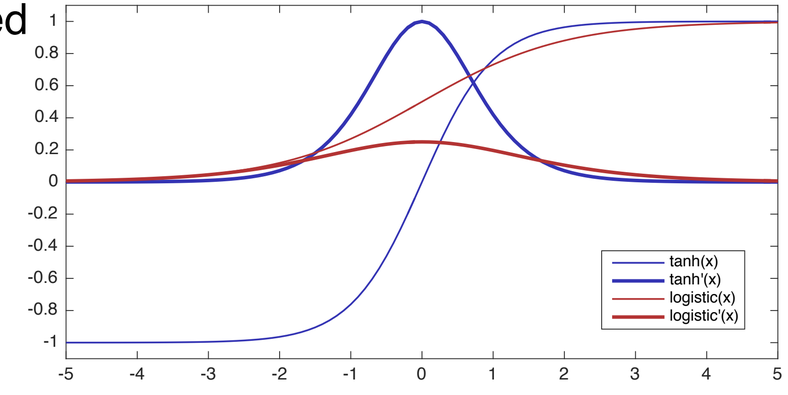




Vanishing Gradients

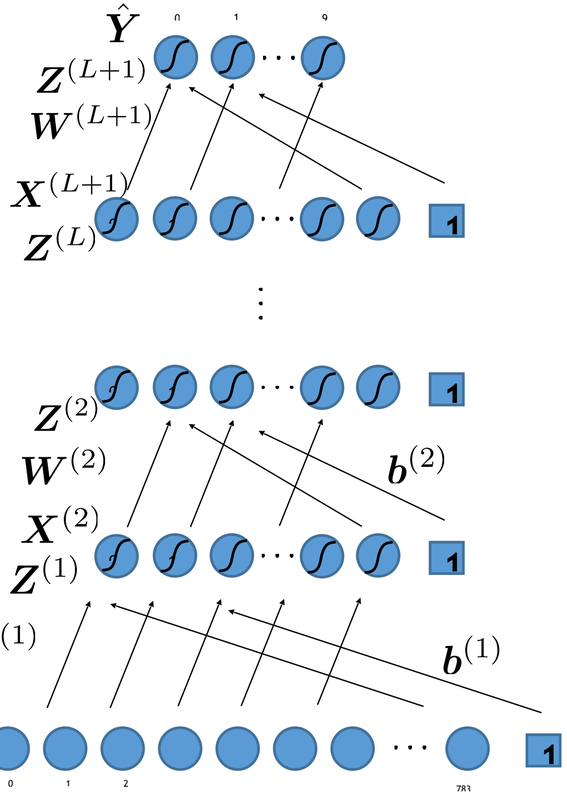
-Why does it happen?

- It's due to the sigmoid function. What's wrong with them?
  - Their gradients are small when the function is saturated



- Suppose a DNN with L hidden layers

$$\begin{aligned}
\boldsymbol{Z}^{(1)} &\leftarrow \boldsymbol{W}^{(1)}\boldsymbol{X}^{(1)} + \boldsymbol{b}^{(1)} & \boldsymbol{X}^{(2)} &\leftarrow \sigma(\boldsymbol{Z}^{(1)}) \\
\boldsymbol{Z}^{(2)} &\leftarrow \boldsymbol{W}^{(2)}\boldsymbol{X}^{(2)} + \boldsymbol{b}^{(2)} & \boldsymbol{X}^{(3)} &\leftarrow \sigma(\boldsymbol{Z}^{(2)}) \\
&\cdots & & \\
\boldsymbol{Z}^{(L)} &\leftarrow \boldsymbol{W}^{(L)}\boldsymbol{X}^{(L)} + \boldsymbol{b}^{(L)} & \boldsymbol{X}^{(L+1)} &\leftarrow \sigma(\boldsymbol{Z}^{(L)}) \\
\boldsymbol{Z}^{(L+1)} &\leftarrow \boldsymbol{W}^{(L+1)}\boldsymbol{X}^{(L+1)} + \boldsymbol{b}^{(L+1)} & \hat{\boldsymbol{Y}} &\leftarrow \mathrm{softmax}(\boldsymbol{Z}^{(L+1)})
\end{aligned}$$



- Gradients per layer

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(L+1)}} = \underbrace{\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{Y}}}\frac{\partial \hat{\boldsymbol{Y}}}{\partial \boldsymbol{Z}^{(L+1)}}}_{\text{BP error } \boldsymbol{\Delta}^{(L+1)}}\frac{\partial \boldsymbol{Z}^{(L+1)}}{\partial \boldsymbol{W}^{(L+1)}} = \underbrace{(\hat{\boldsymbol{Y}} - \boldsymbol{Y})}_{\text{BP error } \boldsymbol{\Delta}^{(L+1)}}\boldsymbol{X}^{(L+1)\top}$$

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(L)}} = \underbrace{\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{Y}}}\frac{\partial \hat{\boldsymbol{Y}}}{\partial \boldsymbol{Z}^{(L+1)}}\frac{\partial \boldsymbol{Z}^{(L+1)}}{\partial \boldsymbol{X}^{(L+1)}}\frac{\partial \boldsymbol{X}^{(L+1)}}{\partial \boldsymbol{Z}^{(L)}}}_{\boldsymbol{\Delta}^{(L)}}\frac{\partial \boldsymbol{Z}^{(L)}}{\partial \boldsymbol{W}^{(L)}} = \underbrace{\left((\boldsymbol{W}^{(L+1)\top}(\hat{\boldsymbol{Y}} - \boldsymbol{Y})) \odot \sigma'(\boldsymbol{Z}^{(L)})\right)}_{\boldsymbol{\Delta}^{(L)}}\boldsymbol{X}^{(L)\top}$$

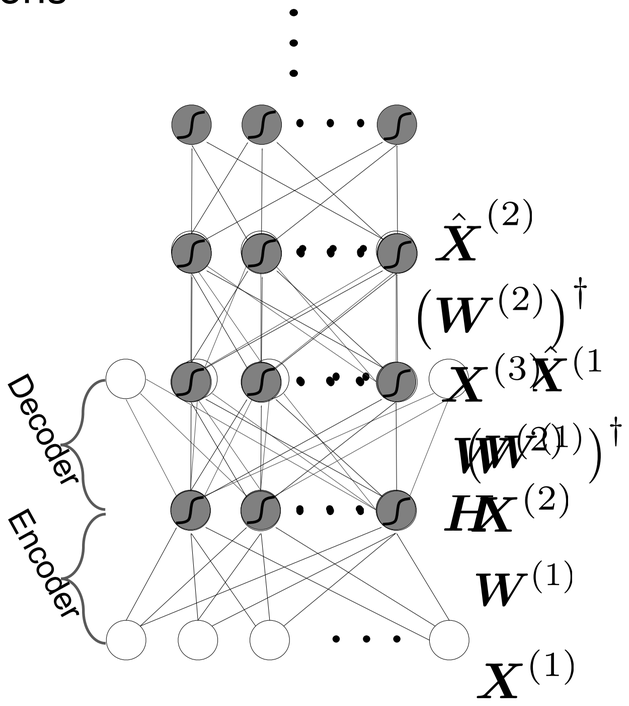
$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(1)}} = \underbrace{\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{Y}}}\frac{\partial \hat{\boldsymbol{Y}}}{\partial \boldsymbol{Z}^{(l+1)}}\prod_{l=1}^{L}\left[\frac{\partial \boldsymbol{Z}^{(l+1)}}{\partial \boldsymbol{X}^{(l+1)}}\frac{\partial \boldsymbol{X}^{(l+1)}}{\partial \boldsymbol{Z}^{(l)}}\right]}_{\boldsymbol{\Delta}^{(1)}}\frac{\partial \boldsymbol{Z}^{(1)}}{\partial \boldsymbol{W}^{(1)}} = \underbrace{\left(\left(\boldsymbol{W}^{(2)\top}\left(\cdots\left(\boldsymbol{W}^{(L+1)\top}(\hat{\boldsymbol{Y}} - \boldsymbol{Y})\right) \odot \sigma'(\boldsymbol{Z}^{(L)})\cdots\right)\right) \odot \sigma'(\boldsymbol{Z}^{(1)})\right)}_{\boldsymbol{\Delta}^{(1)}}\boldsymbol{X}^{(1)\top}$$

L speed bumps

Pretraining

-Greedy layer-wise unsupervised learning

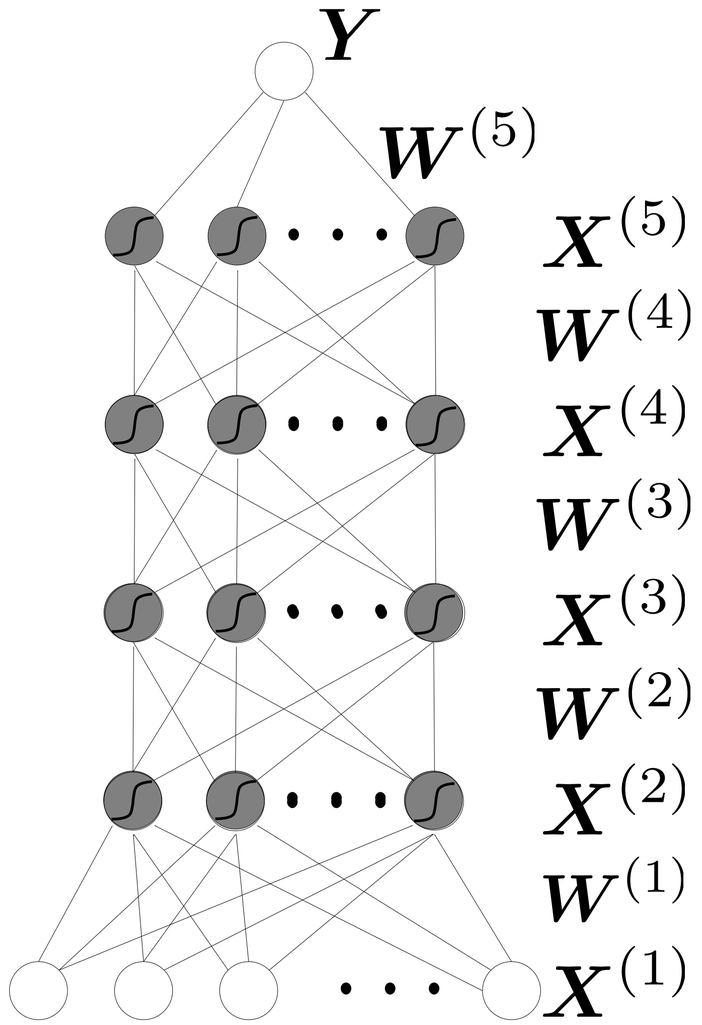
- A primitive method to train a DNN
  - The BP error vanishes in the lower layer due to the use of sigmoid functions
  - Pretraining learns the lower layer weights in a different way
- Stacked AE
  - For the 1st layer
    $$\hat{\boldsymbol{X}}^{(1)} \approx \boldsymbol{W}^{(1)\dagger}\sigma(\boldsymbol{W}^{(1)}\boldsymbol{X}^{(1)})$$
  - For the 2nd layer
    $$\boldsymbol{X}^{(2)} \leftarrow \sigma(\boldsymbol{W}^{(1)}\boldsymbol{X}^{(1)})$$
    $$\hat{\boldsymbol{X}}^{(2)} \approx \boldsymbol{W}^{(2)\dagger}\sigma(\boldsymbol{W}^{(2)}\boldsymbol{X}^{(2)})$$
  - For the (l)-th layer
    $$\boldsymbol{X}^{(l)} \leftarrow \sigma(\boldsymbol{W}^{(l-1)}\boldsymbol{X}^{(l-1)})$$
    $$\hat{\boldsymbol{X}}^{(l)} \approx \boldsymbol{W}^{(l)\dagger}\sigma(\boldsymbol{W}^{(l)}\boldsymbol{X}^{(l)})$$
- What kinds of autoencoders are there?
  - Hinton's first pretraining method used Restricted Boltzman Machine
  - Andrew Ng's group showed sparse AEs work well
  - Bengio's group suggested denoising AEs



Pretraining

-Greedy layer-wise unsupervised learning

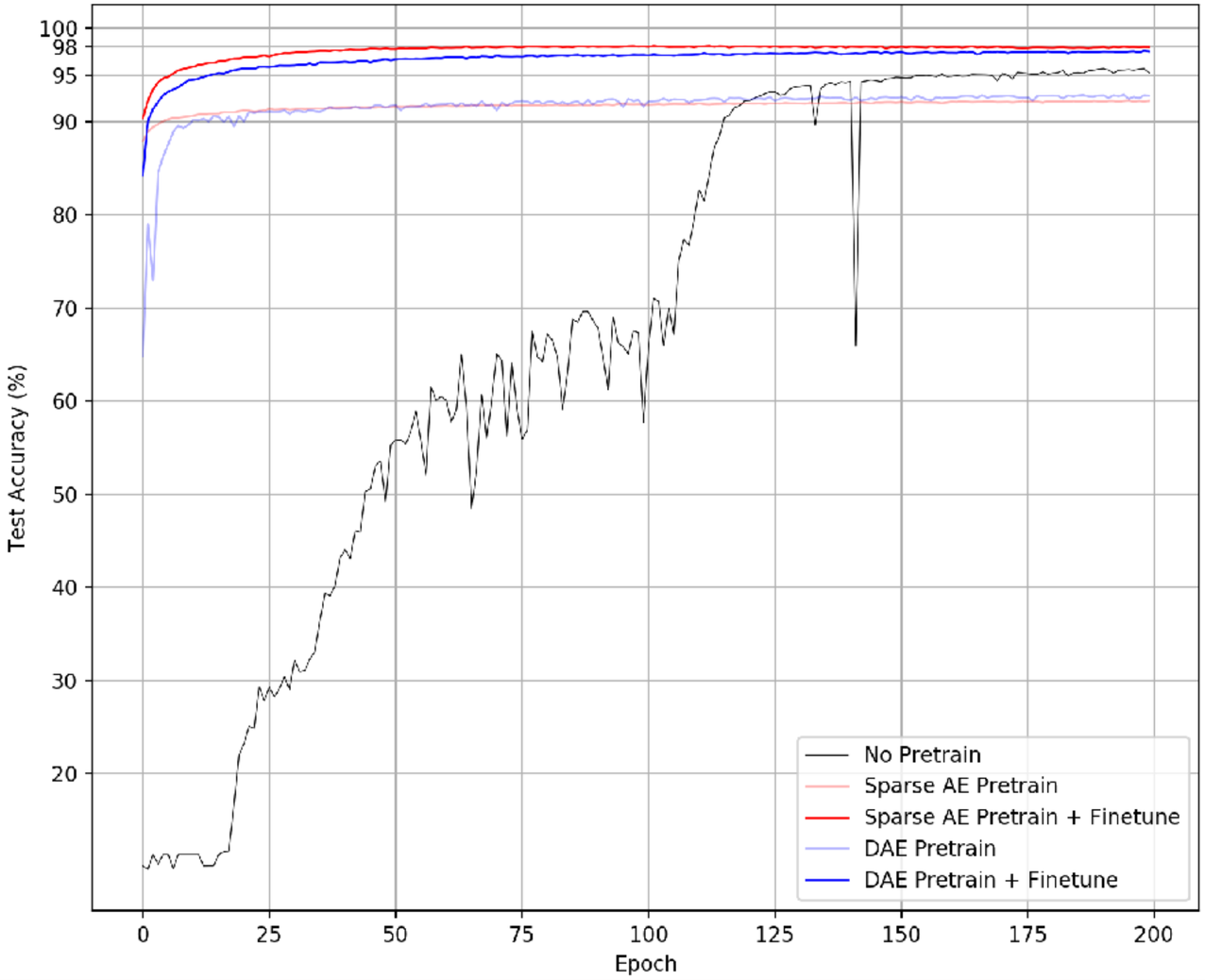
- Now that you know some better weights, the DNN can be initialized with the pretrained weights
  - Any problems with this?
- Each layer-wise AE creates approximation error
  - which propagates up to the last layer
  - Last layer weights in the stacked AE are not in good quality
  - What can we do?
- Finetuning
  - This procedure is not different from a regular BP
  - It might not be very effective to update the lower layers
  - But can fix the last layer weights
  - A good solution for the stacked AE
- The pretraining-and-finetuning pipeline was the first deep learning method
- But pretraining is known to be difficult to train



Pretraining and Finetuning

-Does this work well?

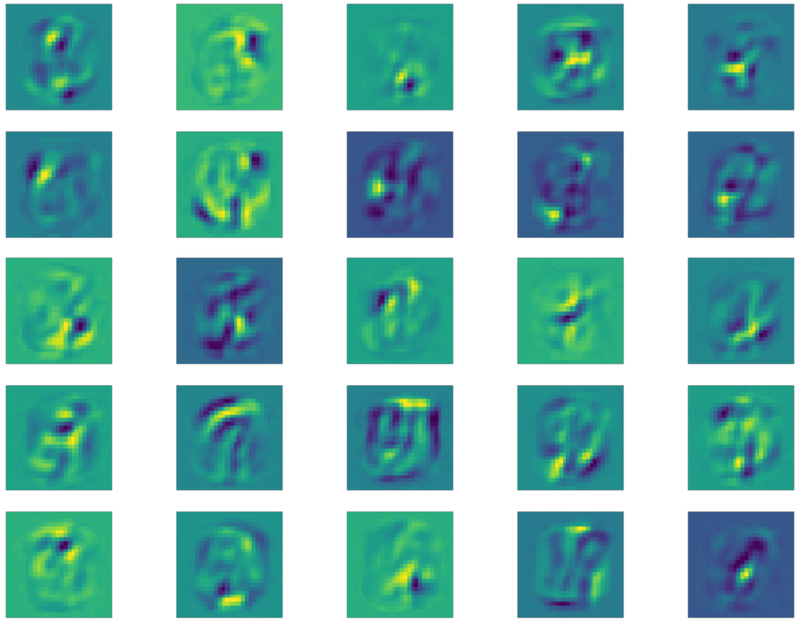
- 100X3 Logistic Hidden Units
- Stochastic Gradient Descent



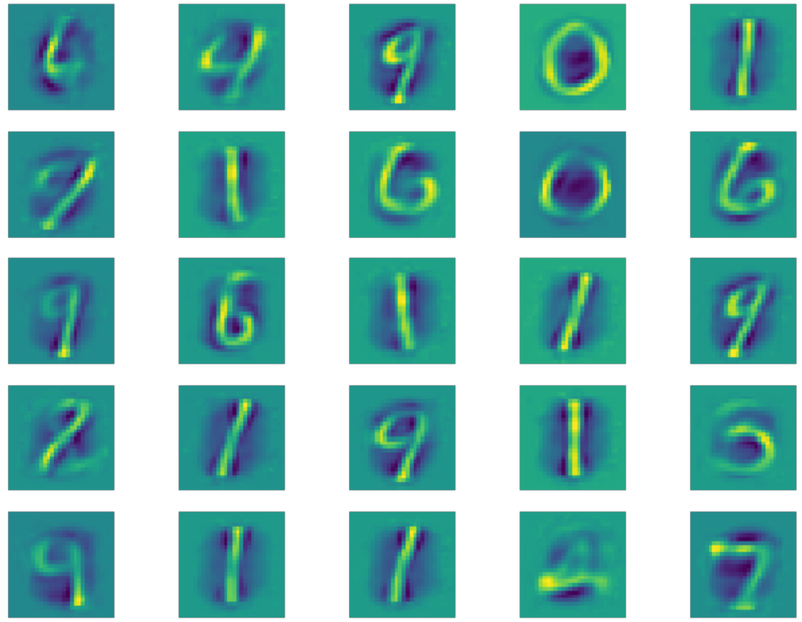
Pretraining and Finetuning

-Does this work well?

- The thing is, training such autoencoders is tricky in a lot of senses
  - There are some criteria known to work (e.g. sparsity), but it's difficult to know until we try them out
  - It's unsupervised learning, so there's a chance of overfitting
  - Some hyperparameters to tune as usual
    - e.g. What would be the right amount of sparsity?



$\alpha = 0.5$



$\alpha = 0.05$

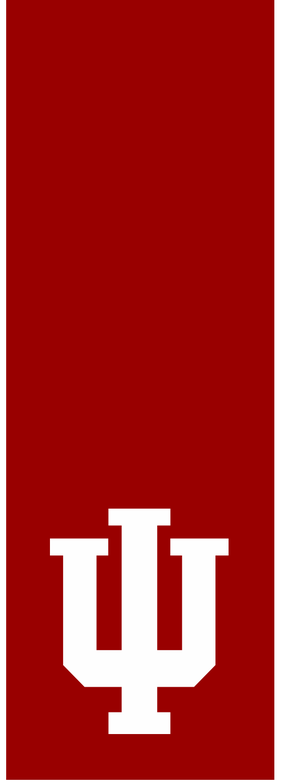
Pretraining and Finetuning

-Does this work well?

- More importantly, the higher layers are hard to visualize to figure out the best way to train
- It turned out I'm not the only one who doesn't like this approach
- What would be the other ways to train DNNs?
  - Choosing the activation function wisely
  - Securing the pathway of BP error
  - Better optimization methods help, too

Reading

- Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets." Neural computation 18.7 (2006): 1527-1554.
- Lee, Honglak, Chaitanya Ekanadham, and Andrew Y. Ng. "Sparse deep belief net model for visual area V2." Advances in neural information processing systems. 2008.
- Vincent, Pascal, et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." Journal of Machine Learning Research 11.Dec (2010): 3371-3408.



Thank You!