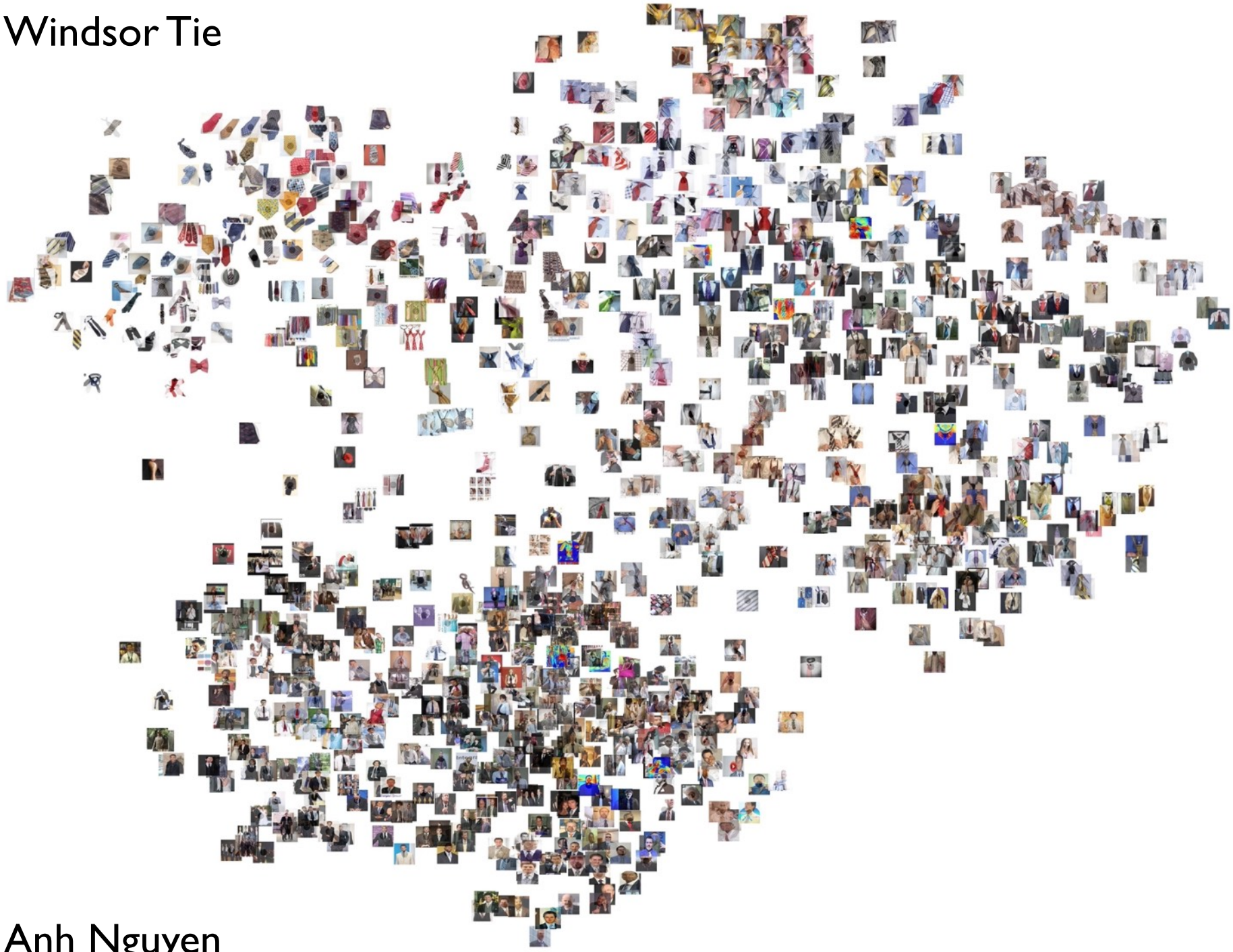
# CPSC 340:
# Machine Learning and Data Mining

Deep Learning

# Admin

- Extra lecture (no guest lecture) to spend more time on deep learning + convolutions
- Please bring a laptop (or similar) Monday for evaluations
    - The Faculty of Science requests we allocate class time to filling them out

Windsor Tie



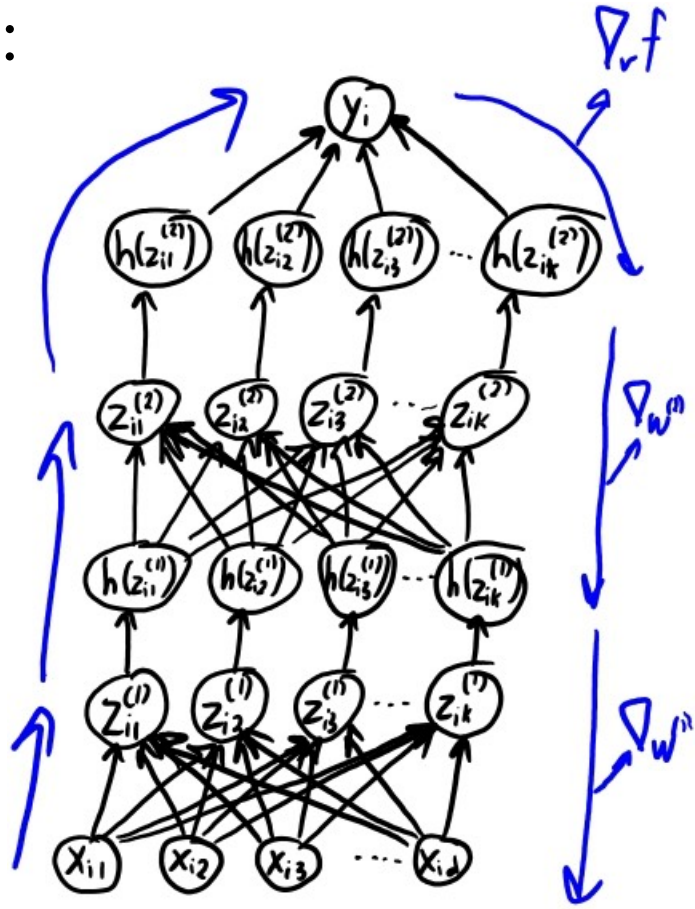
Anh Nguyen

# Today

- One more dose of intuition for DNNs
- Finish discussion of how to train deep neural networks
  - algorithms, tips, and tricks, and miscellaneous key info

# Backpropagation

- Overview of how we compute neural network gradient:
  - Forward propagation :
    - Compute $z_{i(1)}$ from $x_i$ .
    - Compute $z_{i(2)}$ from $z_{i(1)}$ .
    - …
    - Compute $\hat{Y}_i$ from $z_{i(m)}$ , and use this to compute error.
  - Backpropagation :
    - Compute gradient with respect to regression weights ‘v’.
    - Compute gradient with respect to $z_{i(m)}$ weights $W_{(m)}$ .
    - Compute gradient with respect to $z_{i(m-1)}$ weights $W_{(m-1)}$ .
    - …
    - Compute gradient with respect to $z_{i(1)}$ weights $W_{(1)}$ .




- “Backpropagation” is the chain rule plus some bookkeeping for speed.

bonus!

# Backpropagation

- Instead of the next few bonus slides, I HIGHLY recommend watching this video from former UBC master's student Andrej Karpathy (of OpenAI, former director of AI and Autopilot Vision at Tesla)
  - https://www.youtube.com/watch?v =i94OvYb6noo

# Backpropagation

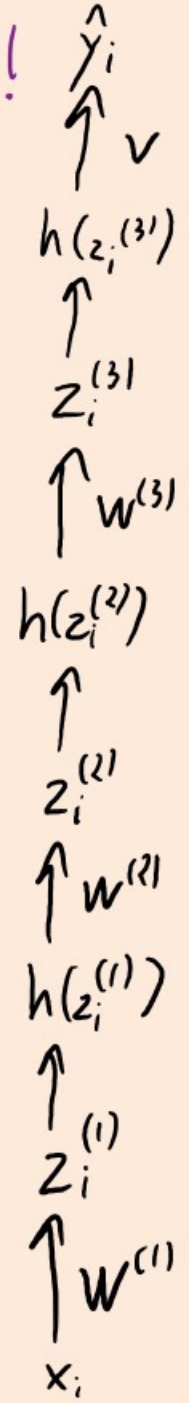
- Let's illustrate backpropagation in a simple setting:
  - 1 training example, 3 hidden layers, 1 hidden "unit" in layer.

$$f(w^{(1)}, w^{(2)}, w^{(3)}, v) = \frac{1}{2}(\hat{y}_i - y_i)^2 \quad \text{where} \quad \hat{y}_i = v\,h(w^{(3)}\,h(w^{(2)}\,h(w^{(1)} x_i)))$$

$$\frac{\partial f}{\partial v} = r\,h(w^{(3)}\,h(w^{(2)}\,h(w^{(1)} x_i))) = r\,h(z_i^{(3)})$$

$$\frac{\partial f}{\partial w^{(3)}} = r\,v\,h'(w^{(3)}\,h(w^{(2)}\,h(w^{(1)} x_i)))\,h(w^{(2)}\,h(w^{(1)} x_i)) = r\,v\,h'(z_i^{(3)})\,h(z_i^{(2)})$$

bonus!



bonus!

# Backpropagation

- Let's illustrate backpropagation in a simple setting:
  - 1 training example, 3 hidden layers, 1 hidden "unit" in layer.

$$f(w^{(1)}, w^{(2)}, w^{(3)}, v) = \frac{1}{2}(\hat{y}_i - y_i)^2 \quad \text{where} \quad \hat{y}_i = v h(w^{(3)} h(w^{(2)} h(w^{(1)} x_i)))$$

$$\frac{\partial f}{\partial v} = r\, h(w^{(3)} h(w^{(2)} h(w^{(1)} x_i))) = r\, h(z_i^{(3)})$$

$$\frac{\partial f}{\partial w^{(3)}} = r\, v h'(w^{(3)} h(w^{(2)} h(w^{(1)} x_i))) h(w^{(2)} h(w^{(1)} x_i)) = r\, v h'(z_i^{(3)}) h(z_i^{(2)})$$

$$\frac{\partial f}{\partial w^{(2)}} = r\, v h'(w^{(3)} h(w^{(2)} h(w^{(1)} x_i))) w^{(3)} h'(w^{(2)} h(w^{(1)} x_i)) h(w^{(1)} x_i) = r^{(3)} w^{(3)} h'(z_i^{(2)}) h(z_i^{(1)})$$

$$\frac{\partial f}{\partial w^{(1)}} = r\, v h'(w^{(3)} h(w^{(2)} h(w^{(1)} x_i))) w^{(3)} h'(w^{(2)} (w^{(1)} x_i)) w^{(2)} h'(w^{(1)} x_i) x_i = r^{(2)} w^{(2)} h'(z_i^{(1)}) x_i$$

bonus!

# Backpropagation

- Let's illustrate backpropagation in a simple setting:
  - 1 training example, 3 hidden layers, 1 hidden "unit" in layer.

$$\frac{\partial f}{\partial v} = r\, h(z_i^{(3)})$$

$$\frac{\partial f}{\partial w^{(3)}} = r\, v\, h'(z_i^{(3)})\, h(z_i^{(2)})$$

$$\frac{\partial f}{\partial w^{(2)}} = r^{(3)} W^{(3)} h'(z_i^{(2)})\, h(z_i^{(1)})$$

$$\frac{\partial f}{\partial w^{(1)}} = r^{(2)} W^{(2)} h'(z_i^{(1)})\, x_i$$

$$\frac{\partial f}{\partial v_c} = r\, h(z_{ic}^{(3)})$$

$$\frac{\partial f}{\partial w^{(3)}_{c'c}} = r\, v_c\, h'(z_{ic'}^{(3)})\, h(z_{ic}^{(2)})$$

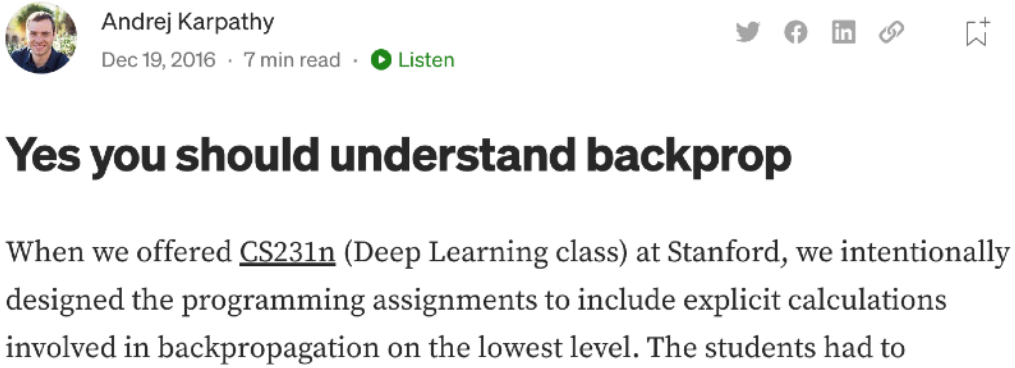
$$\frac{\partial f}{\partial w^{(2)}_{c'c}} = \left[\sum_{c''=1}^{k} r_{c''}^{(3)} W^{(3)}_{c''c'}\right] h'(z_{ic'}^{(2)})\, h(z_{ic}^{(1)})$$

$$\frac{\partial f}{\partial w^{(1)}_{cj}} = \left[\sum_{c''=1}^{k} r_{c''}^{(2)} W^{(2)}_{c''c}\right] h'(z_{ic}^{(1)})\, x_j$$

  - Only the first 'r' changes if you use a different loss.
  - With multiple hidden units, you get extra sums.
    - Efficient if you store the sums rather than computing from scratch.

# Backpropagation

- We've made backprop details bonus material
- Do you need to know how to do this?
  - Exact details are probably not vital (there are many implementations).
  - "Automatic differentiation " is now standard and has same cost.
  - But understanding basic idea helps you know what can go wrong.
    - Or give hints about what to do when you run out of memory.
  - See discussion by a neural network expert (Andrej!)
    - https://karpathy.medium.com/yes-you-should-understand-backprop-e2f06eab496b



Andrej Karpathy

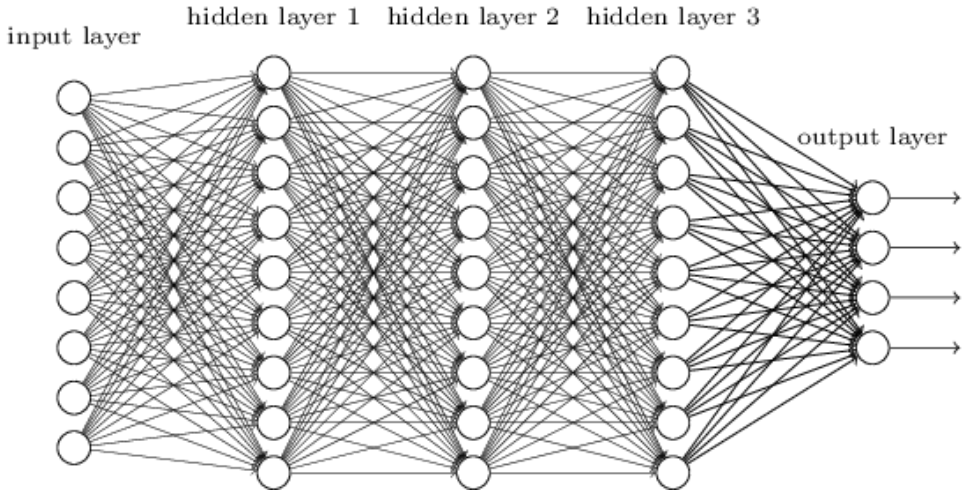
Dec 19, 2016 · 7 min read · Listen

**Yes you should understand backprop**

When we offered CS231n (Deep Learning class) at Stanford, we intentionally designed the programming assignments to include explicit calculations involved in backpropagation on the lowest level. The students had to

# Backpropagation

- You should know cost of backpropagation:
  - Forward pass dominated by matrix multiplications by $W^{(1)}$, $W^{(2)}$, $W^{(3)}$, and v
    - If have 'm' weight layers and all $z_i$ have 'k' elements, cost would be $O(dk + mk^2)$
  - Backward pass has same cost as forward pass

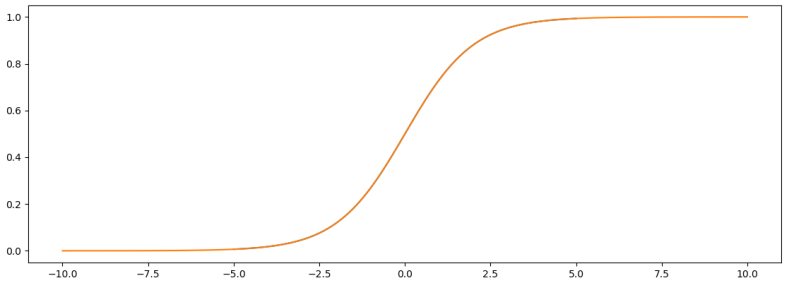


# Stochastic Gradient Training

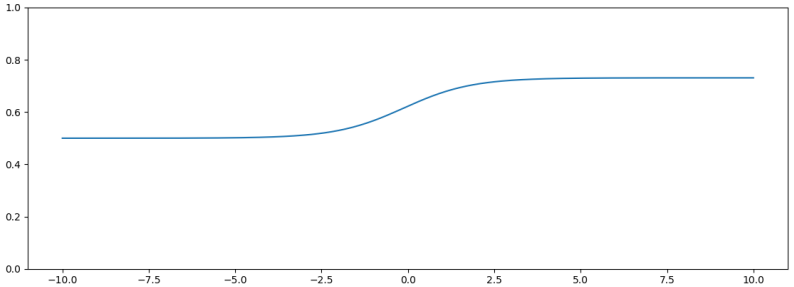
- Standard training method is stochastic gradient (SG):
  - Choose a random example ‘i ’ (more common: mini-batch of samples)
  - Use backpropagation to get gradient with respect to all parameters.
  - Take a small step in the negative gradient direction.
- Challenging to make SG work:
  - Often doesn’t work as a “black box” learning algorithm.
  - But people have developed a lot of tricks/modifications to make it work.
- Highly non- convex , so are the problem local minima?
  - Some empirical/theoretical evidence that local minima are not the problem.
  - If the network is “deep” and “wide” enough, we think all local minima are good.
  - But it can be hard to get SG to close to a local minimum in reasonable time.

# New Issue: Vanishing Gradients

- Consider the sigmoid function:



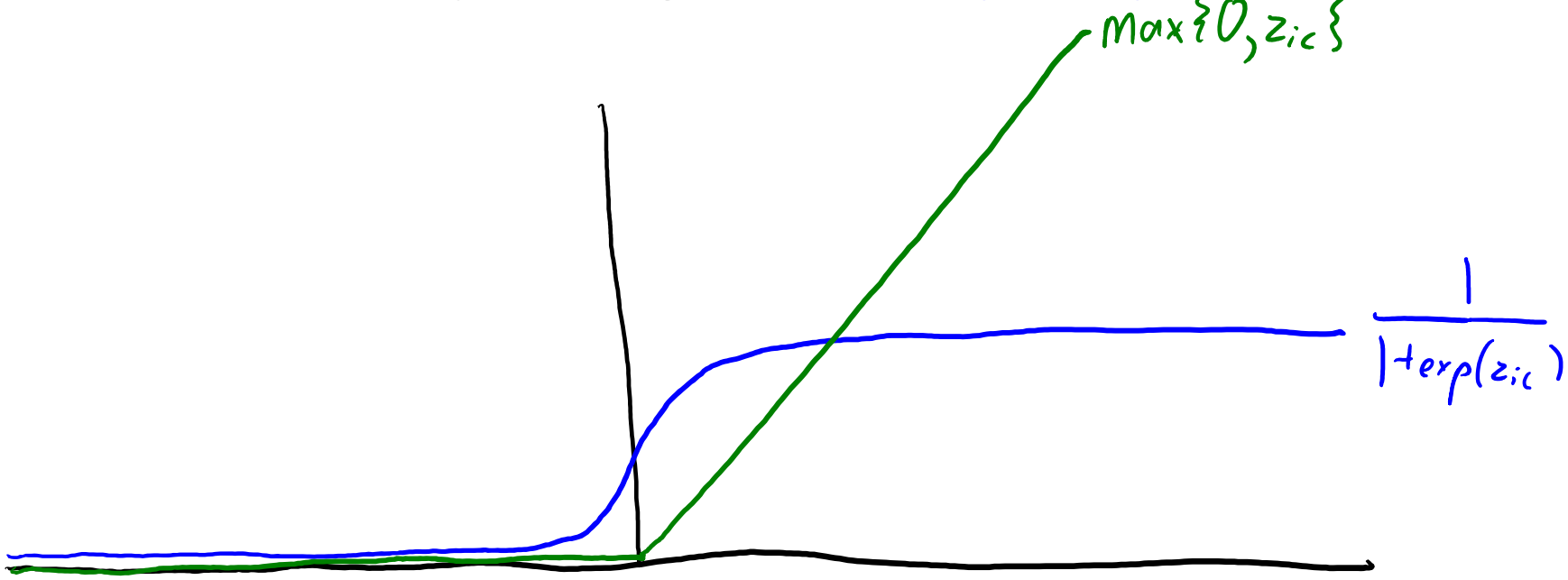

- Away from the origin, the gradient is nearly zero.
- The problem gets worse when you take the sigmoid of a sigmoid:




- In deep networks, many gradients can be nearly zero everywhere.
  - And numerically they will be set to 0, so SGD does not move.

# Rectified Linear Units (ReLU)

- Modern networks often replace sigmoid with perceptron loss (ReLU):

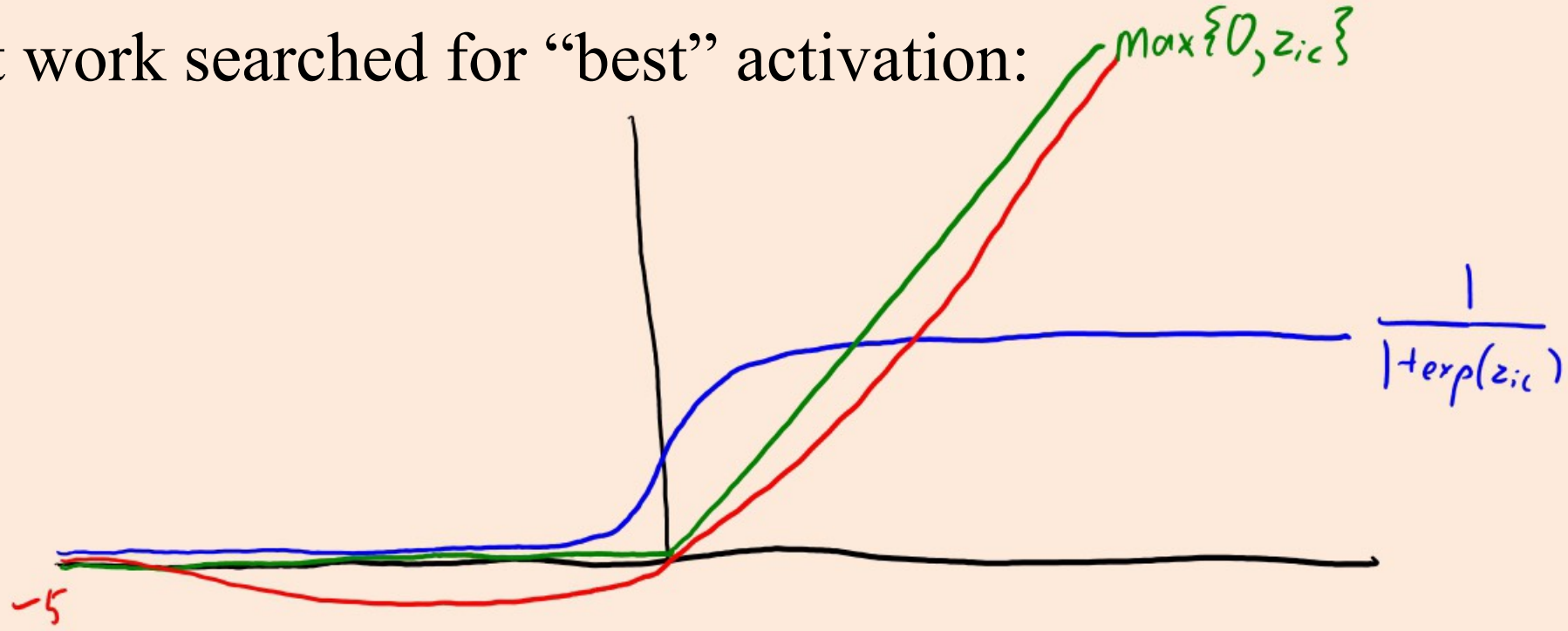



- Just sets negative values $z_{ic}$ to zero.
  - Reduces vanishing gradient problem (positive region is never flat).
  - Gives sparser activations.
  - Still gives a universal approximator if size of hidden layers grows with 'n'.

bonus!

# "Swish" Activiation

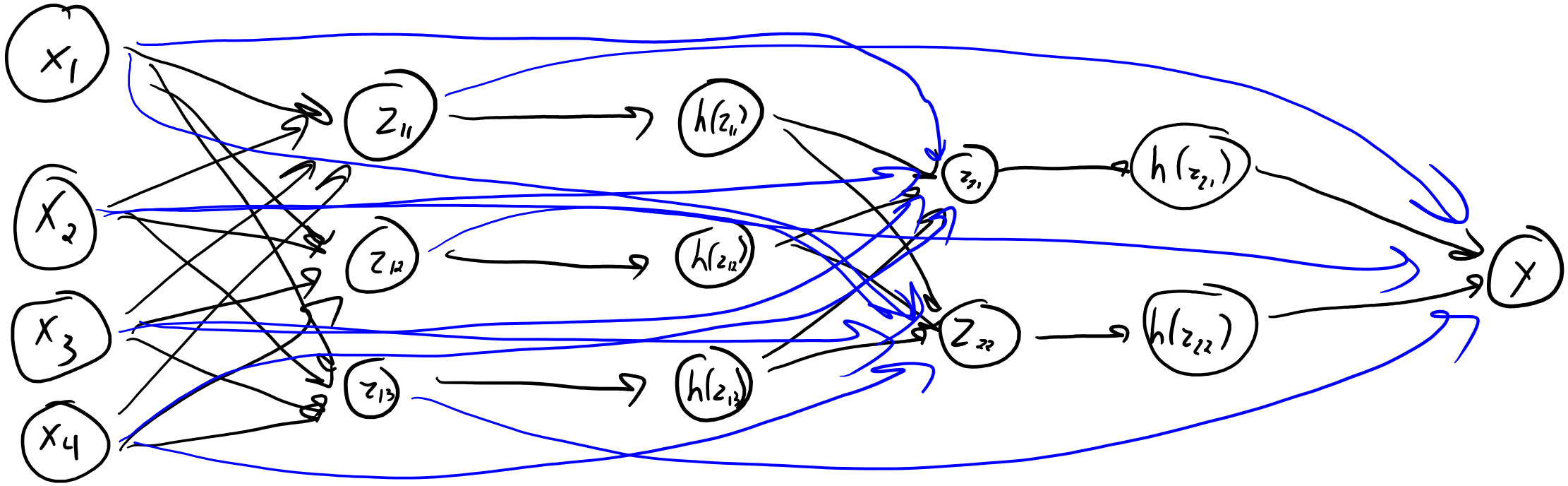
- Recent work searched for "best" activation:




- Found that $z_{ic}/(1+\exp(-z_{ic}))$ worked best ("swish" function).
  - A bit weird because it allows negative values and is non- monotonic.
  - But basically the same as ReLU when not close to 0.

# Skip Connections Deep Learning

- Skip connections can also reduce vanishing gradient problem:




- Makes "shortcuts" between layers (so fewer transformations).
  - Many variations exist on skip connections exist.

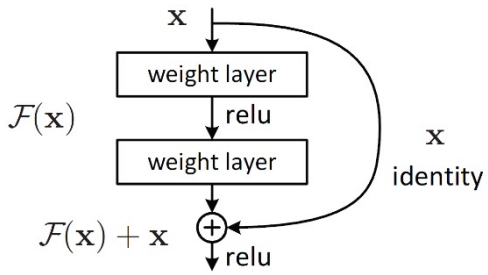
# ResNet "Blocks"

- Residual networks (ResNets) are a variant on skip connections.
  - Consist of repeated "blocks", first methods that successfully used 100+ layers.
- Usual computation of activation based on previous 2 layers:

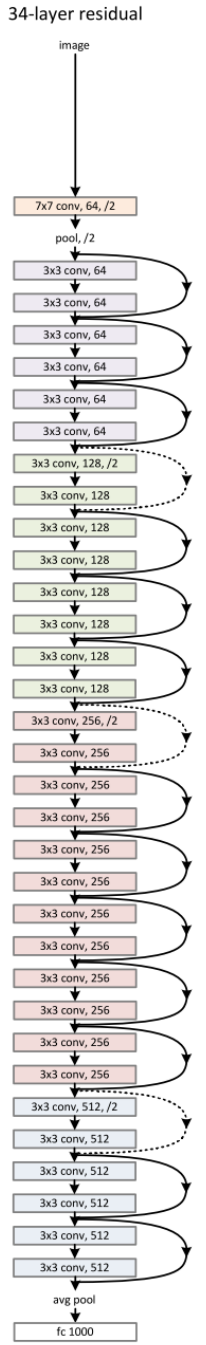
$$a^{l+2} = h(W^{l+1} h(W^l a^l))$$

↑ "activation at layer 'l'"

- ResNet "block": $a^{l+2} = h(a^l + W^{l+1} h(W^l a^l))$




- Differences from usual skip connections:
  - Activations vectors $a^l$ and $a^{l+2}$ must have the same size.
  - No weights on $a^l$, so $W^l$ and $W^{l+1}$ must focus on "updating" $a^l$ (fit "residual").
    - If you use ReLU, then $W^l=0$ implies $a^{l+2}=a^l$.




https://en.wikipedia.org/wiki/Residual_neural_network
https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035

# Parameter Initialization

- Parameter initialization is crucial:
  - Can't initialize weights in same layer to same value, or units will stay the same.
    - Architecture is symmetric, so gradient would be the same for every hidden unit in the layer, so they'd all just always stay doing the exact same thing.
  - Can't initialize weights too large, it will take too long to learn.
- A traditional random initialization:
  - Initialize bias variables to 0.
  - Sample from standard normal, divided by $10^5$ (0.00001* randn).
    - w = .00001* randn(k,1)
  - Performing multiple initializations does not seem to be important (except maybe with very small networks)

bonus!

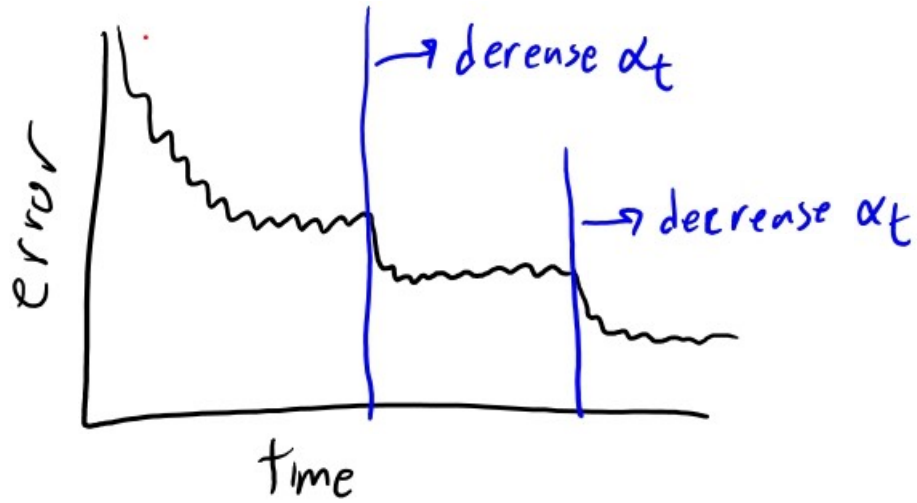
# Parameter Initialization

- Also common to transform data in various ways:
  - Subtract mean, divide by standard deviation, "whiten", standardize $y_i$.
- More recent initializations try to standardize initial $z_i$:
  - Use different initialization in each layer.
  - Try to make variance of $z_i$ the same across layers.
    - Popular approach is to sample from standard normal, divide by sqrt(2* nInputs ).
  - Use samples from uniform distribution on [ - b,b], where

$$b = \frac{\sqrt{6}}{\sqrt{k^{(m)} + k^{(m-1)}}}$$

# Setting the Step - Size

- Stochastic gradient is very sensitive to the step size in deep models.
- One approach: manual "babysitting" of the step-size.
  - Run SG for a while with a fixed step- size.
  - Occasionally measure error and plot progress:




  - If error is not decreasing, decrease step- size.

# Setting the Step - Size

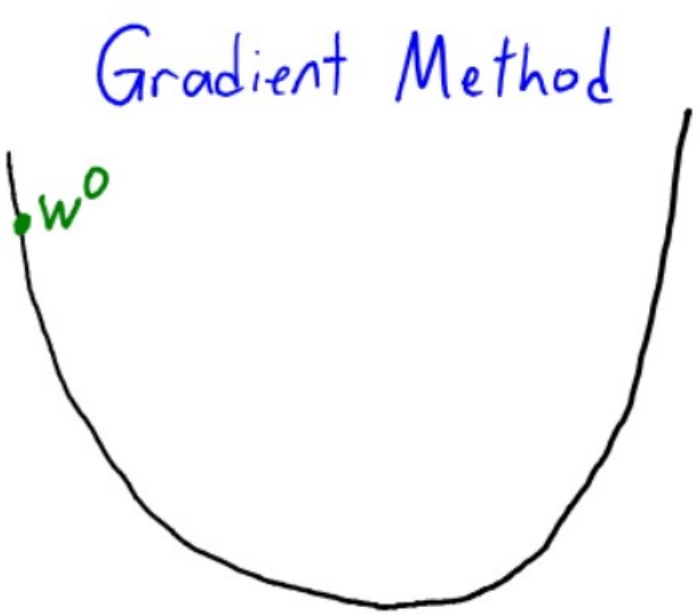
- Stochastic gradient is very sensitive to the step size in deep models.
- Bias step -size multiplier : use bigger step-size for the bias variables.
- Momentum (stochastic version of "heavy-ball" algorithm):
  - Add term that moves in previous direction:

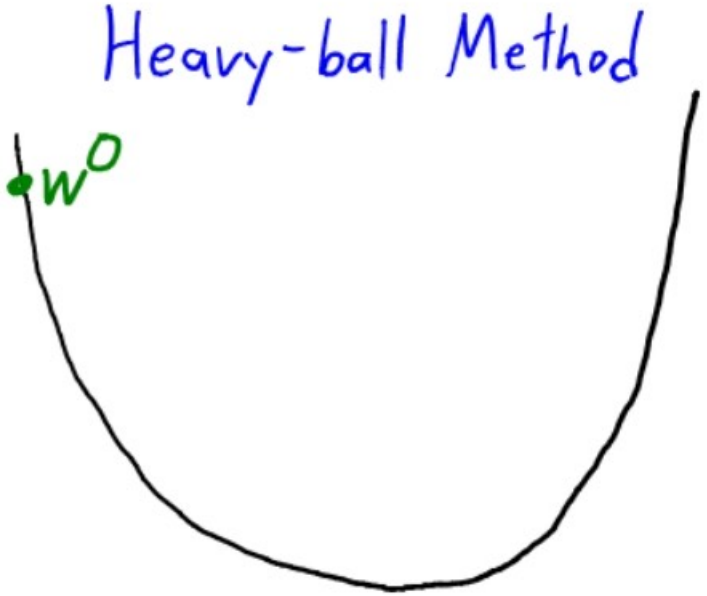
$$w^{t+1} = w^t - \alpha^t \nabla f_i(w^t) + \beta^t (w^t - w^{t-1})$$

keep going in the old direction

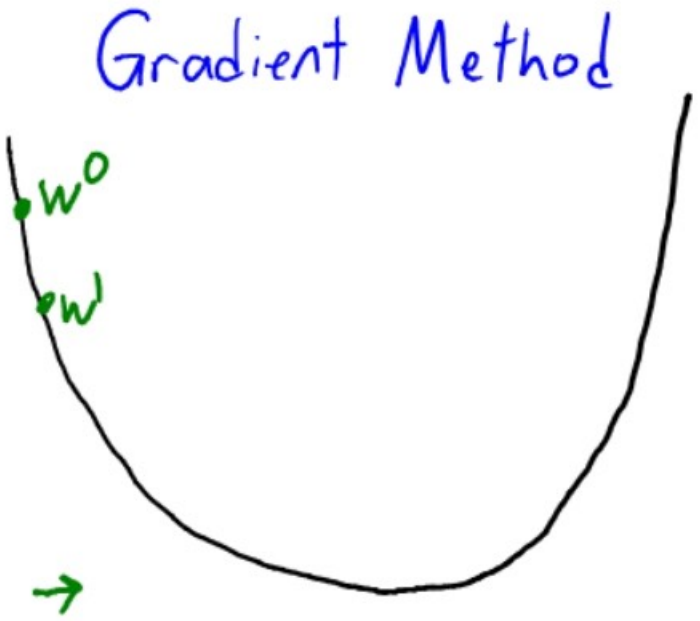
  - Usually $\beta_t = 0.9$.

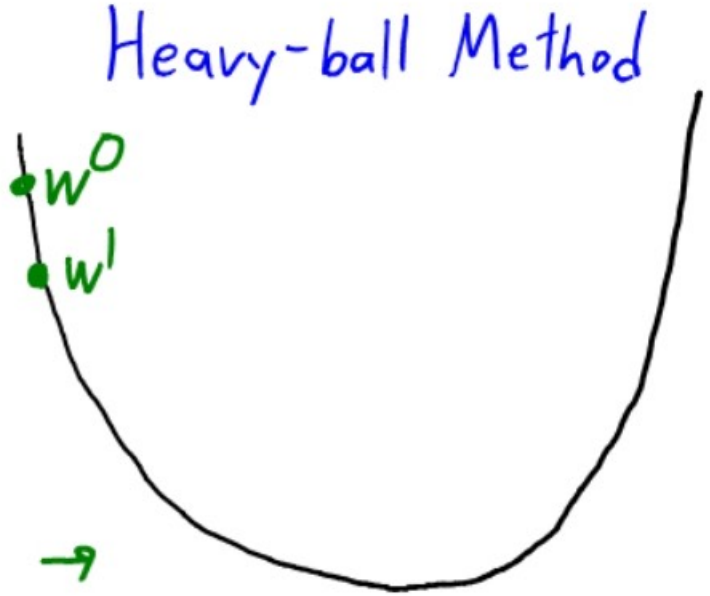
# Gradient Descent vs. Heavy- Ball Method



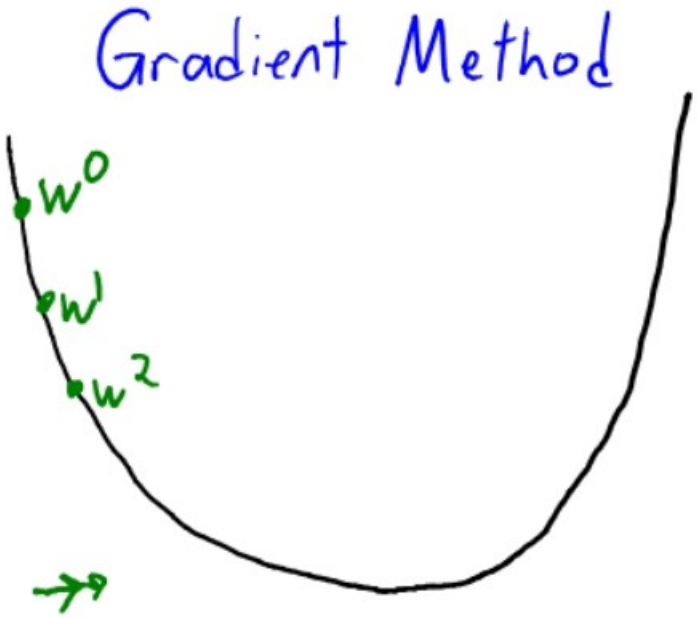



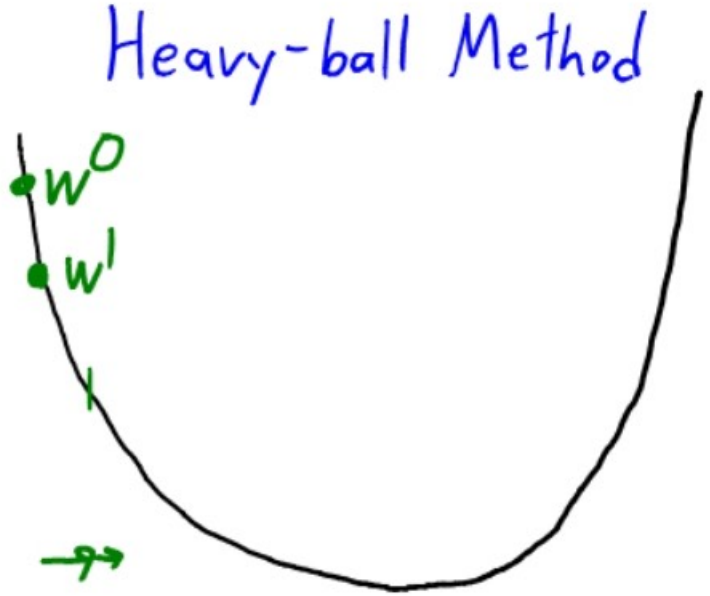
# Gradient Descent vs. Heavy- Ball Method



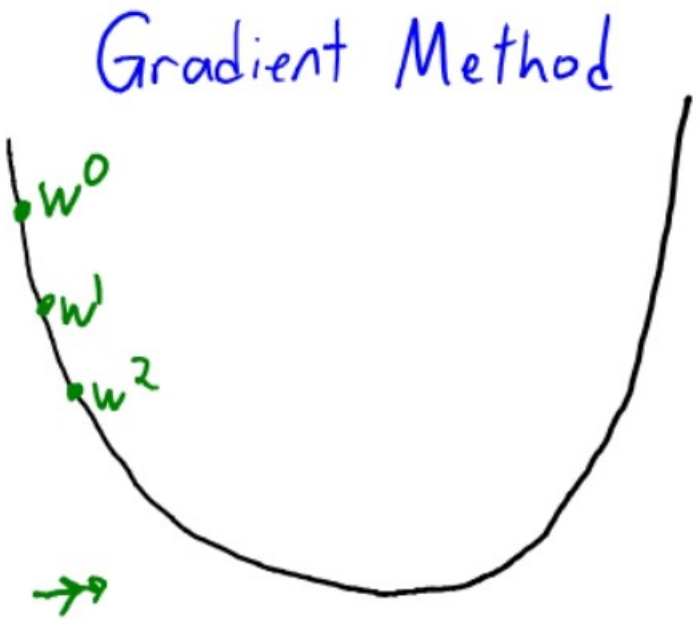



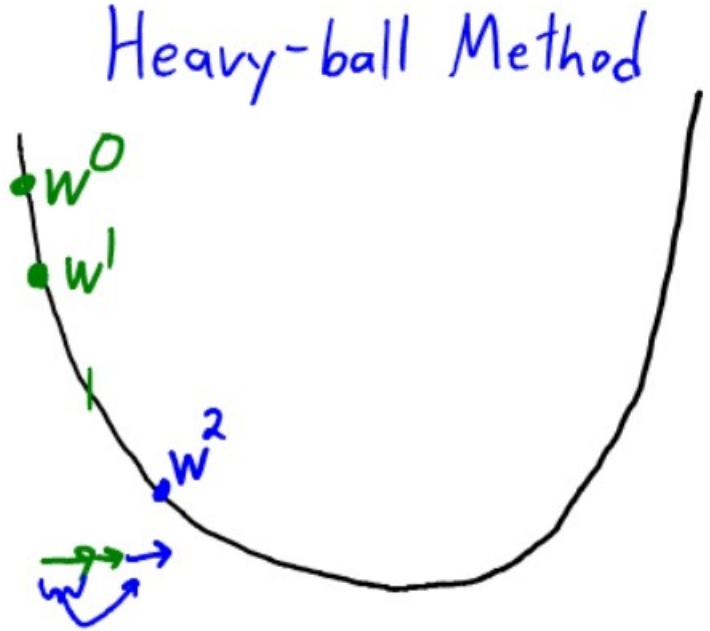
# Gradient Descent vs. Heavy- Ball Method



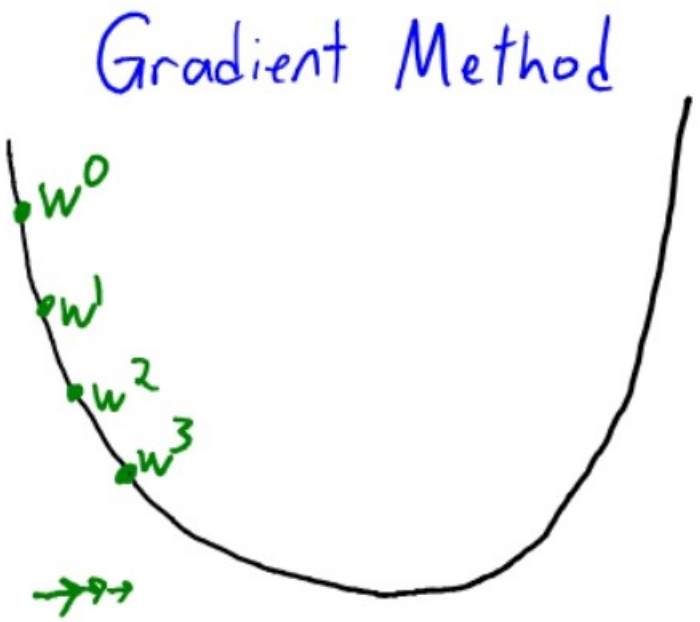



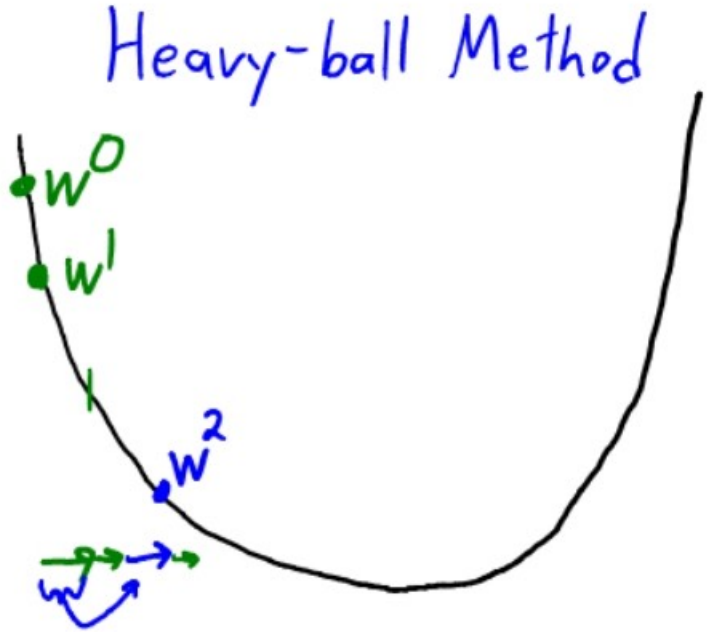
# Gradient Descent vs. Heavy- Ball Method



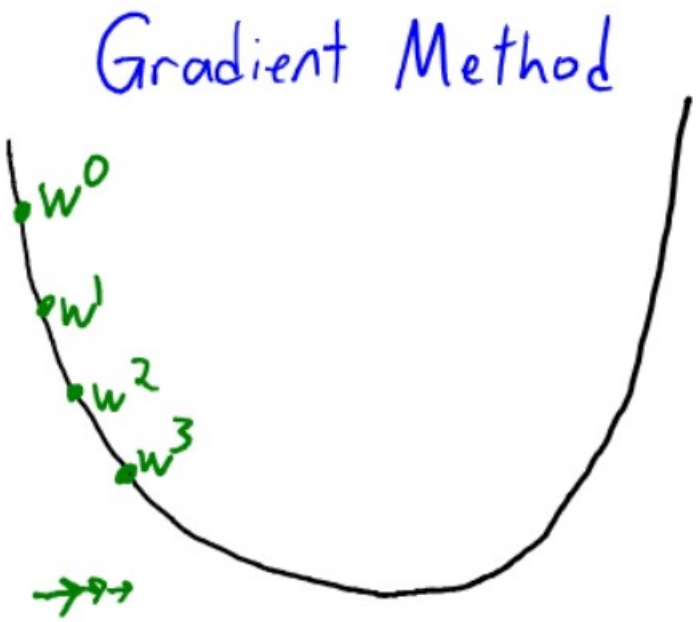



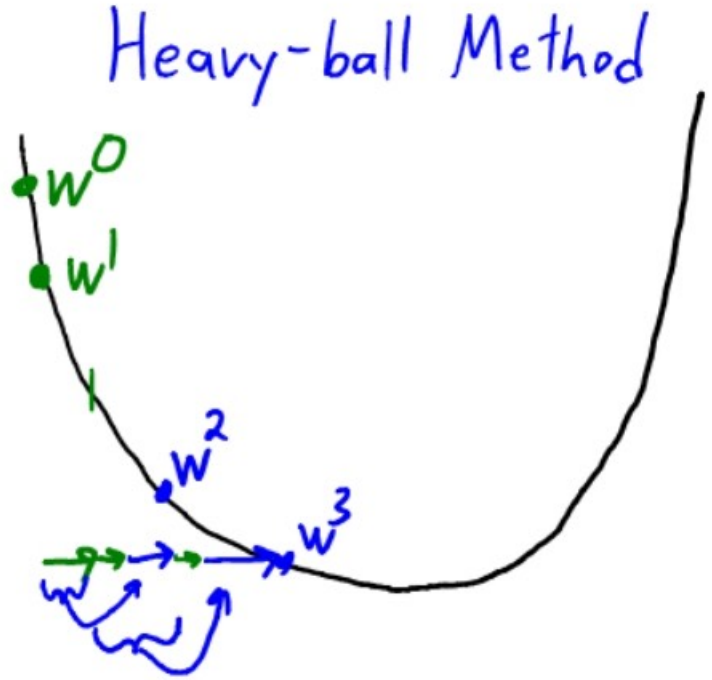
# Gradient Descent vs. Heavy- Ball Method



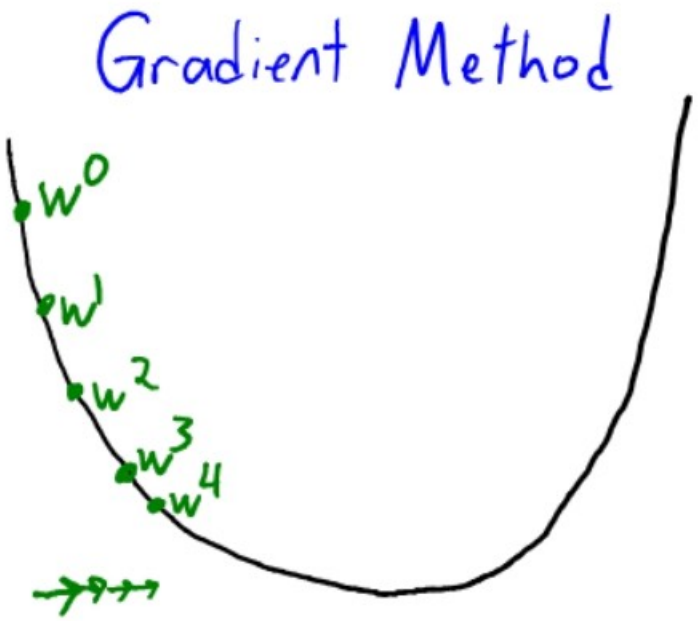



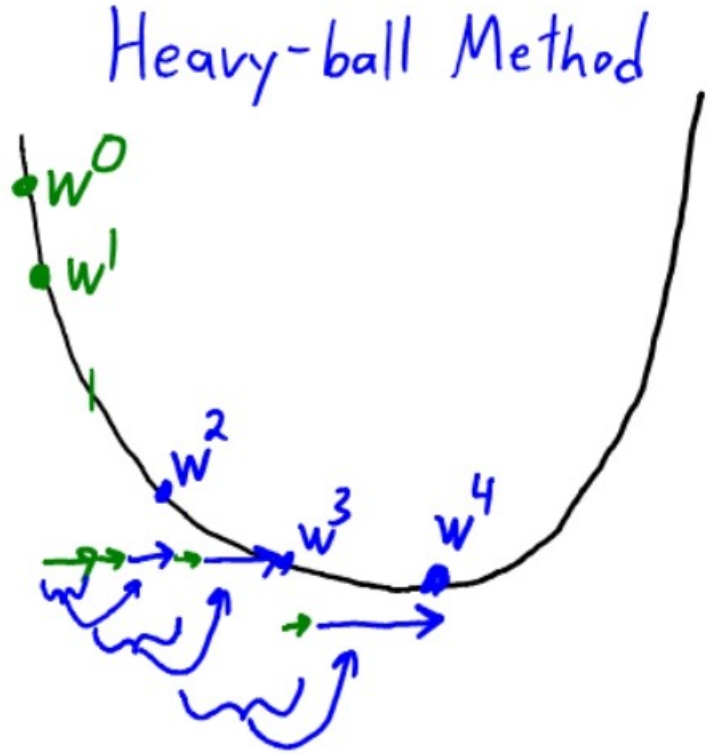
# Gradient Descent vs. Heavy- Ball Method



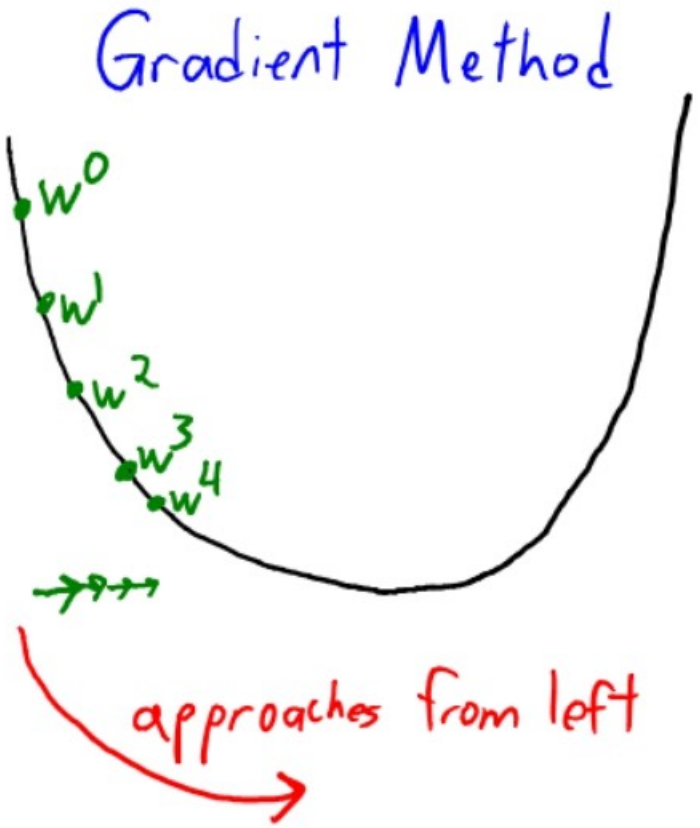



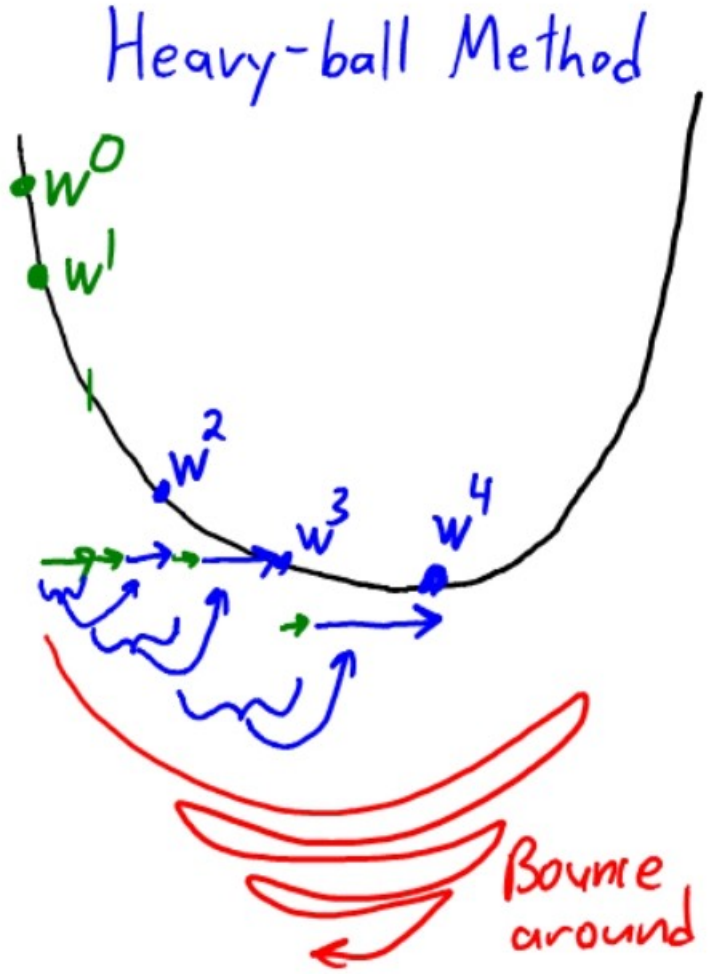
# Gradient Descent vs. Heavy- Ball Method



# Gradient Descent vs. Heavy- Ball Method




Good demo to check out: https://distill.pub /2017/momentum/

bonus!

# Setting the Step - Size

- Automatic method to set step size is Bottou trick :
  1. Grab a small set of training examples (maybe 5% of total).
  2. Do a binary search for a step size that works well on them.
  3. Use this step size for a long time (or slowly decrease it from there).

- Several recent methods using a step size for each variable :
  - AdaGrad, RMSprop, Adam (often work better "out of the box").
  - Some controversy versus plain stochastic gradient (often with momentum).
    - SGD can often get lower test error, even though it takes longer and requires more tuning of step- size.
- Batch size (number of random examples) also influences results.
  - Bigger batch sizes often give faster convergence but maybe to worse solutions?

- Another recent trick is batch normalization:
  - Try to "standardize" the hidden units within the random samples as we go.
  - Held as example of deep learning "alchemy " (blog post here about deep learning claims).
    - Sounds science - ey and often works, but little theoretical understanding.

# Common Deep Learning Tricks

- Data standardization ("centering" and "whitening").
- Parameter initialization: "small but different".
  - If we initialize all parameters in the layer to same value, they stay the same.
  - Also common to use initializations that are standardized within layers.
- Step-size selection: "babysitting".
  - Use bigger step-size for the bias variables, or different for each layer.
  - Methods that use a step size for each coordinate (AdaGrad, RMSprop, Adam).
- Early stopping of the optimization based on validation accuracy.
- Momentum: adds weighted sum of previous SGD directions.
- Batch normalization: adaptive standardizing within layers.
  - Often allows sigmoid activations in deep networks.

# Common Deep Learning Tricks

- L2-regularization or L1-regularization ("weight decay").
  - Sometimes with different $\lambda$ for each layer.
  - Recent work shows this can introduce bad local optima.
- Dropout: randomly zeroes activations 'z' values to discourage dependence.
- Rectified linear units (ReLU) as non-linear transformation.
  - Makes objective non-differentiable, but we now know SGD still converges in this setting.
- Residual/skip connections: connect layers to multiple previous layers.
  - We now know that such connections make it more likely to converge to good minima.
- Neural architecture search: try to cleverly search space of hyper-parameters.
  - This gets expensive!
- Some of these tricks are explored in bonus slides.

# Missing Theory Behind Training Deep Networks

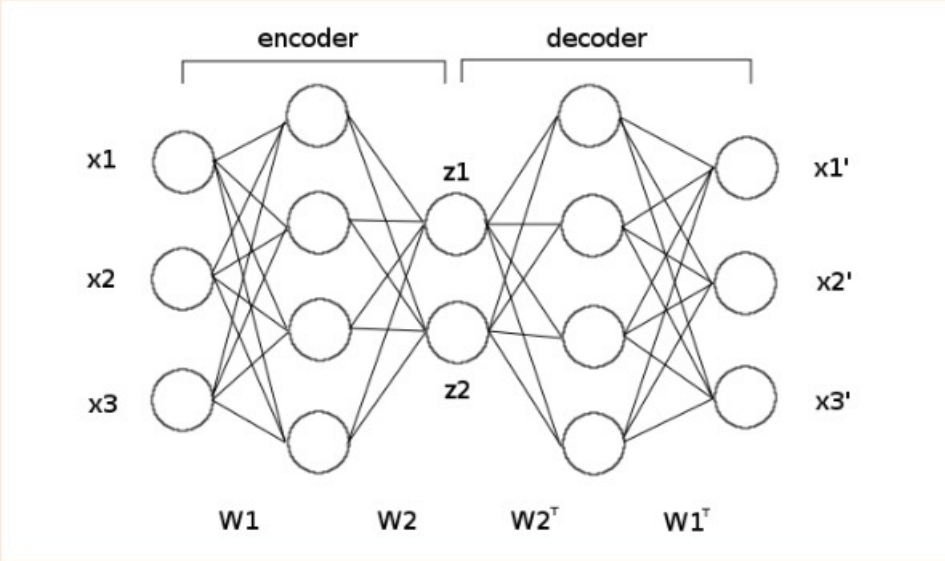
- Unfortunately, we do not understand many of these tricks very well.
  - Large portion of theory is on degenerate case of linear neural networks.
    - Or other weird cases like "1 hidden unit per layer".
  - A lot of research is performed using "grad student descent".
    - Several variations are tried, ones that perform well empirically are kept (possibly overfitting).
- Popular Examples:
  - Batch normalization originally proposed to fix "internal covariate shift".
    - Internal covariate shift not defined in original paper, and batch norm does seem to reduce it.
      - Often singled out as an example of problems with machine learning scholarship.
    - Like many heuristics, people use batch norm because they found that it often helps.
      - Many people have worked on better explanations.
  - Adam optimizer is a nice combinations of ideas from several existing algorithms.
    - Such as "momentum" and "AdaGrad", both of which are well-understood theoretically.
      - But theory in the original paper was incorrect, and Adam fails at solving some very-simple optimization problems.
    - But is Adam is often used because it is amazing at training some networks.
      - It has been hypothesized that we "converged" towards networks that are easier for current SGD methods like Adam.

# Autoencoders

bonus!



- Autoencoders are an unsupervised deep learning model:
  - Use the inputs as the output of the neural network.

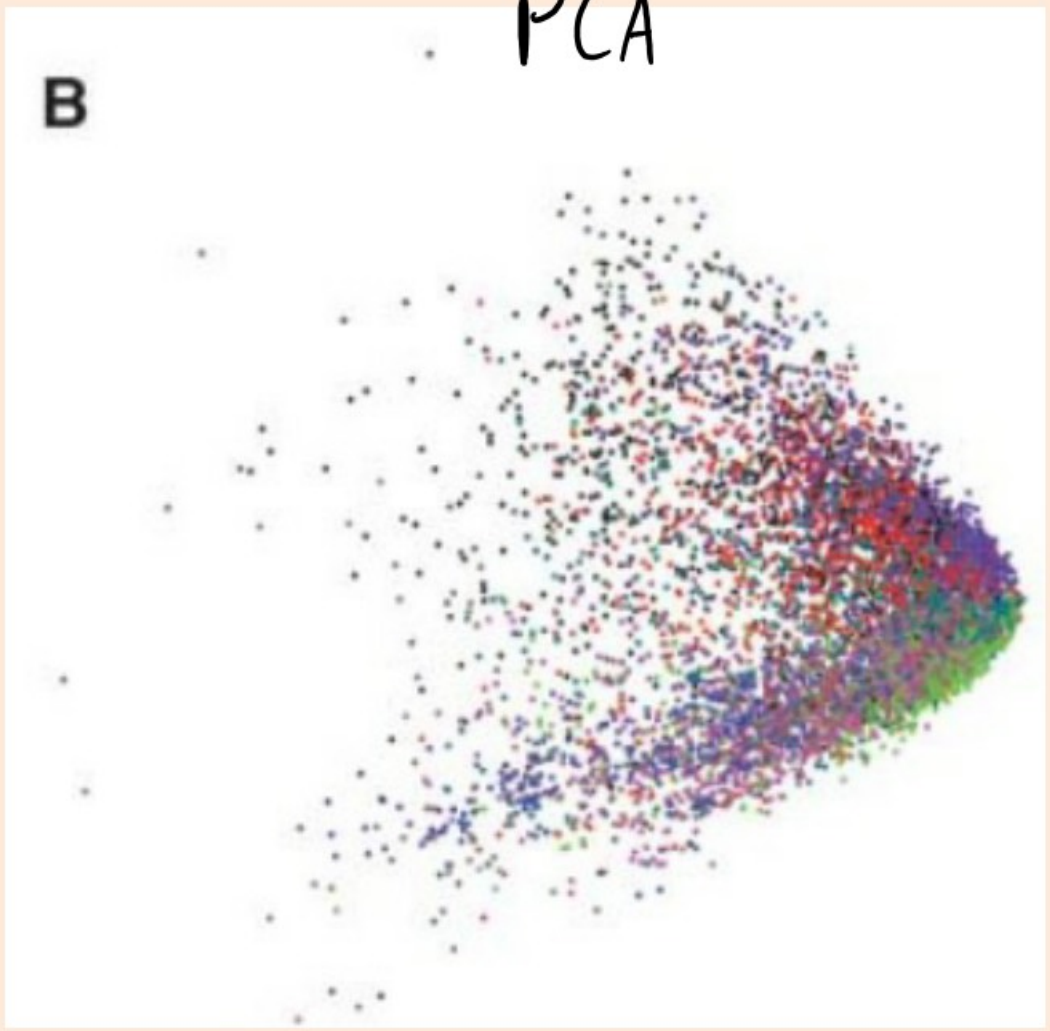


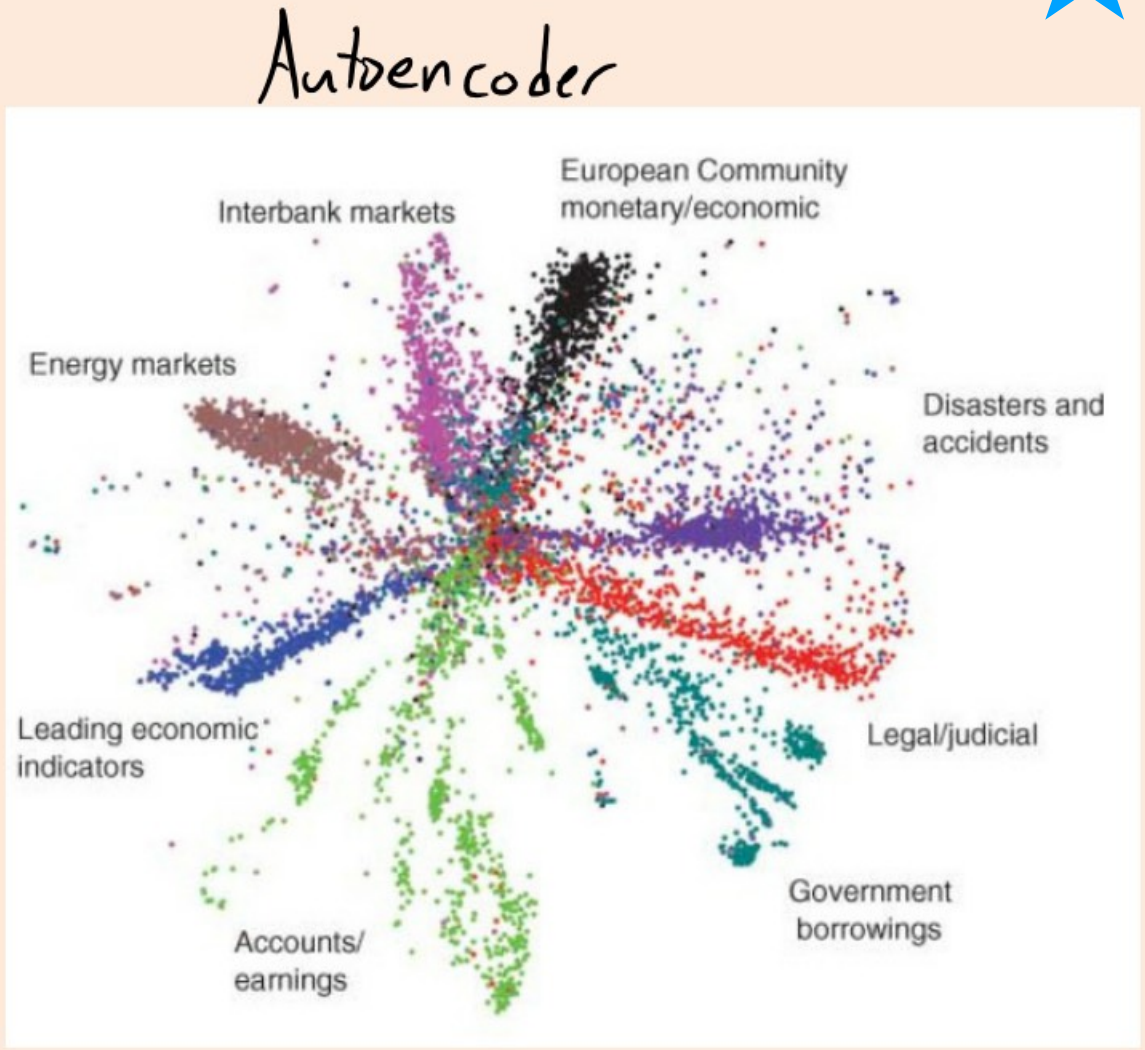

  - Middle layer could be latent features in non - linear latent-factor model.
    - Can do outlier detection, data compression, visualization, etc.
  - A non - linear generalization of PCA.
    - Equivalent to PCA if you don't have non- linearities .

http://inspirehep.net/record/1252540/plots

# Autoencoders

bonus!

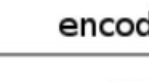



https://www.cs.toronto.edu/~hinton/science.pdf

# Denoising Autoencoder

bonus!



- Denoising autoencoders add noise to the input:




  - Learns a model that can remove the noise.

http://inspirehep.net/record/1252540/plots

# Deep Learning Vocabulary

- "Deep learning": Models with many hidden layers.
  - Usually neural networks.
- "Neuron": node in the neural network graph.
  - " Visible unit ": feature.
  - " Hidden unit ": latent factor $z_{ic}$ or $h(z_{ic})$.
- "Activation function": non- linear transform.
- "Activation": $h(z_i)$.
- "Backpropagation": compute gradient of neural network.
  - Sometimes "backpropagation" means " training with SGD ".
- " Weight decay ": L2- regularization.
- "Cross entropy ": softmax loss.
- "Learning rate": SGD step- size.
- "Learning rate decay ": using decreasing step- sizes.
- " Vanishing/Exploding gradient ": gradient becoming real small/big for deep net

# Summary

- Backpropagation computes neural network gradient via chain rule.
- Parameter initialization is crucial to neural net performance.
- Optimization and step size are crucial to neural net performance.
  - "Babysitting", schedules, momentum.
- ReLU avoid "vanishing gradients".

- Next: The most important idea in computer vision?