



CS545 – Machine Learning for Signal Processing

Deep Learning and Stochastic Neural Models

1 November 2023

Today's lecture

- Shallow vs. Deep learning
- Stochastic neural models
 - Deep learning structures
- Tricks that enable deep learning

“Shallow” models

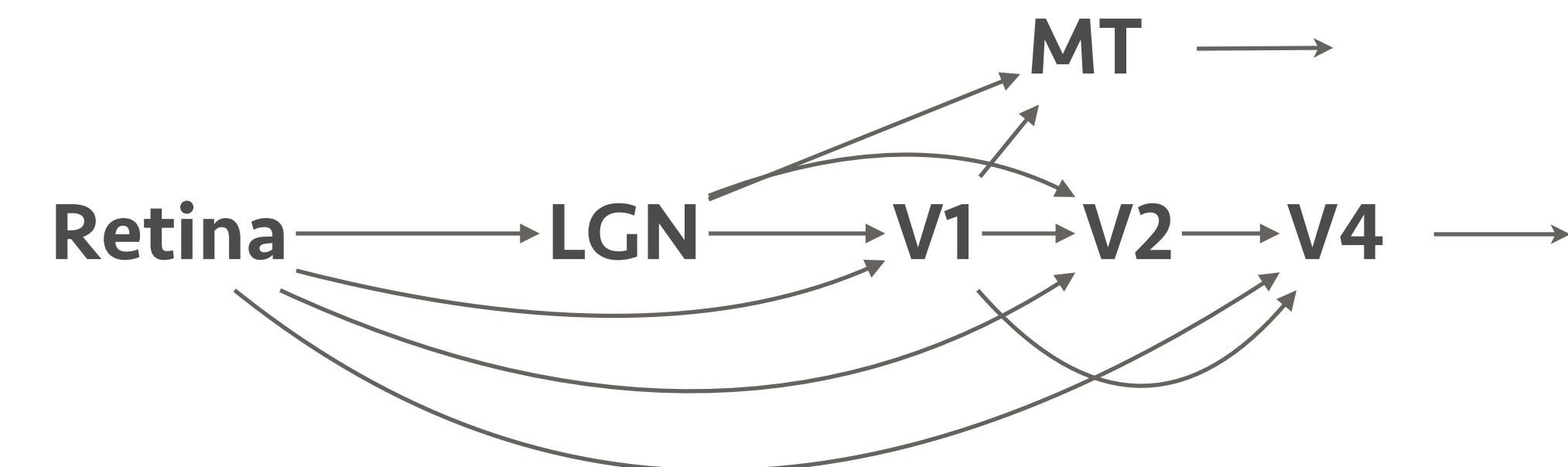
- Most models we used so far were “shallow”

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x}$$

- Single level of processing (PCA, linear classifier, ...)
 - Allows us to use simple representations of the input
 - e.g. linear features, likelihoods, etc.

Looking for depth

- The way we think is inherently hierarchical (i.e. “deep”)
 - e.g. remember the perceptual pathways?
 - Features of features of features, ...



- So far we haven’t looked into algorithms that work like that
 - But it sounds like a good idea!

Trying to get some depth

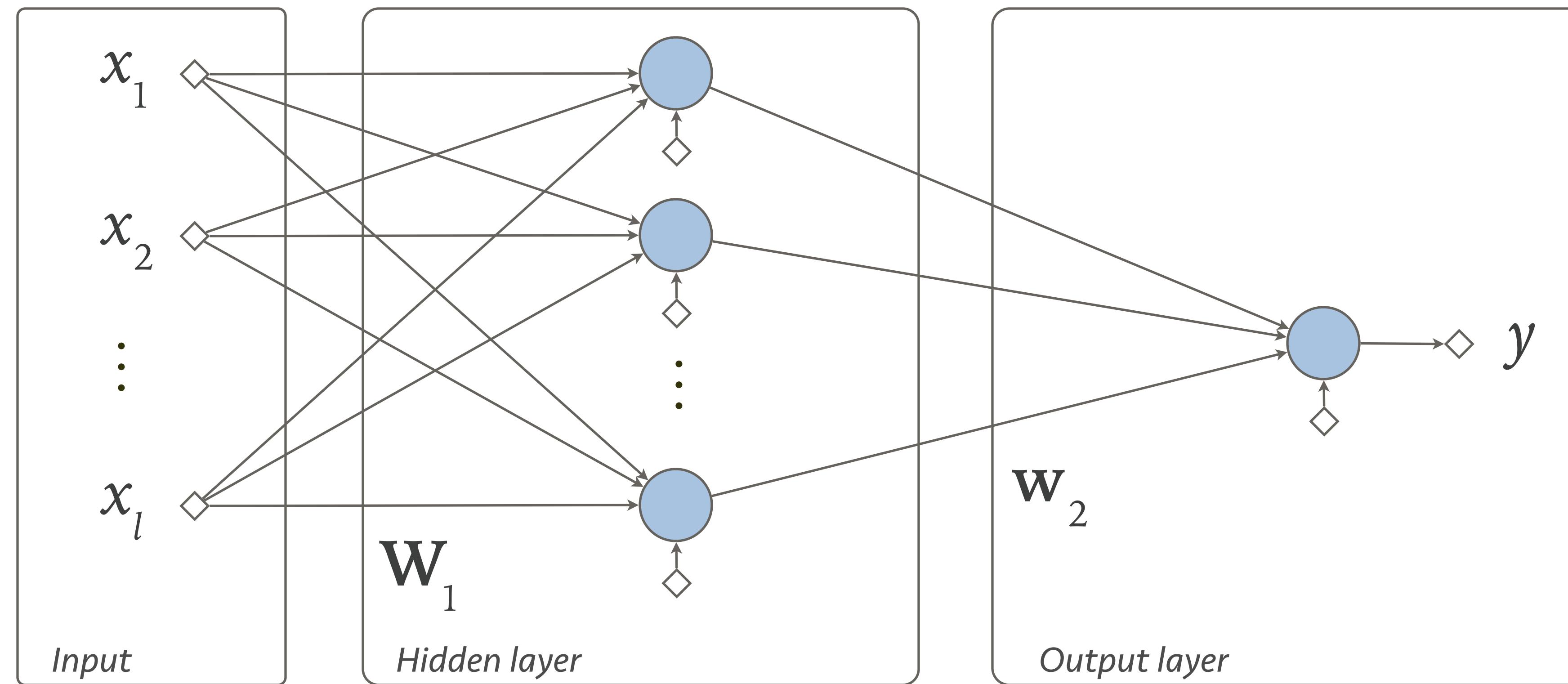
- How about we try multilayer PCA?

$$\mathbf{y} = \mathbf{W}_1 \cdot \mathbf{W}_2 \cdot \mathbf{x}$$

- \mathbf{W}_1 can contain eigenvectors of eigenvectors!
 - Is that a good idea?
- Not really, $\mathbf{W}_1 = \mathbf{I}$ since $\mathbf{W}_2 \cdot \mathbf{x}$ is whitened already
 - And, in general, additional linear transforms make no sense
 - They all collapse to a single linear transform

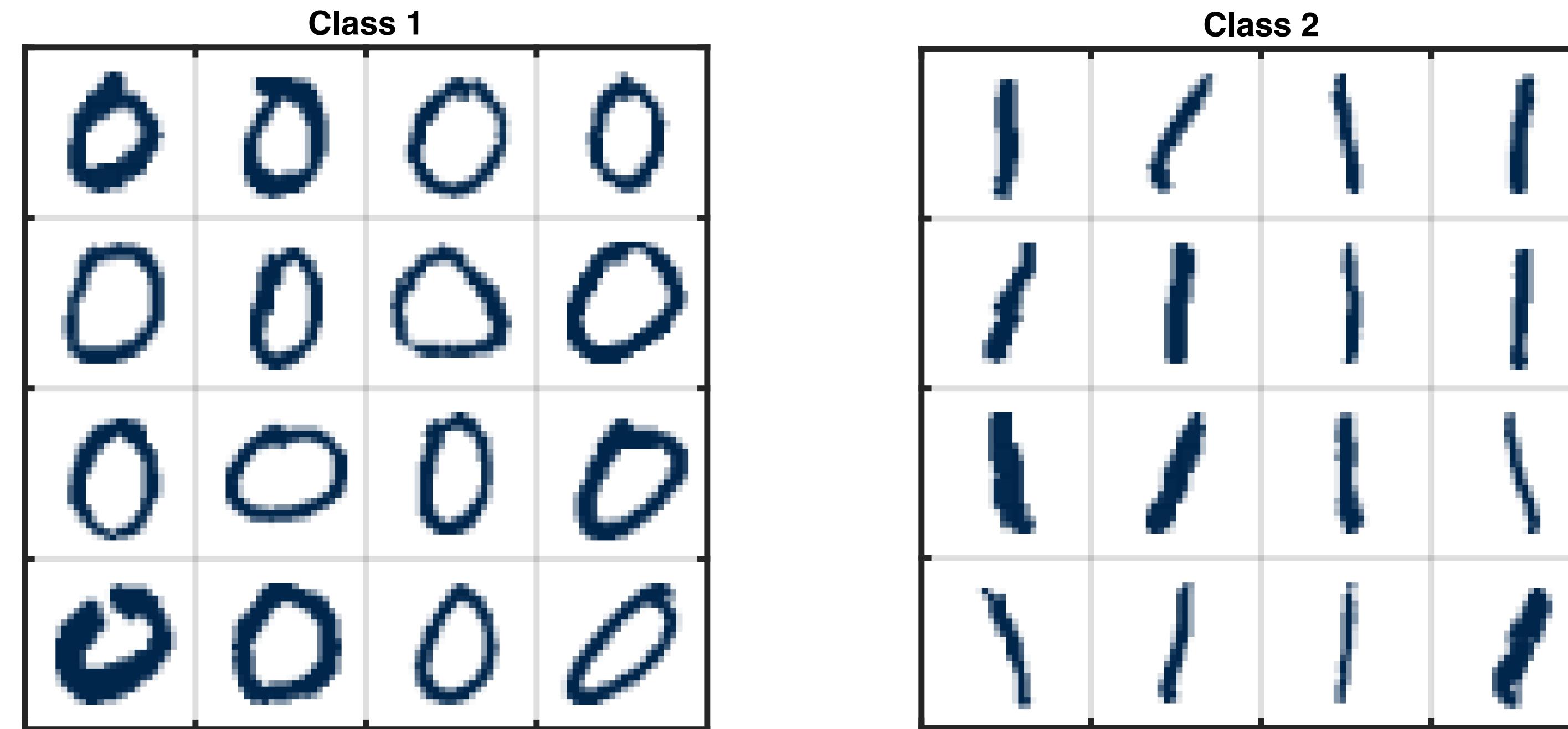
Revisiting the neural net

- An example of a “deep” architecture



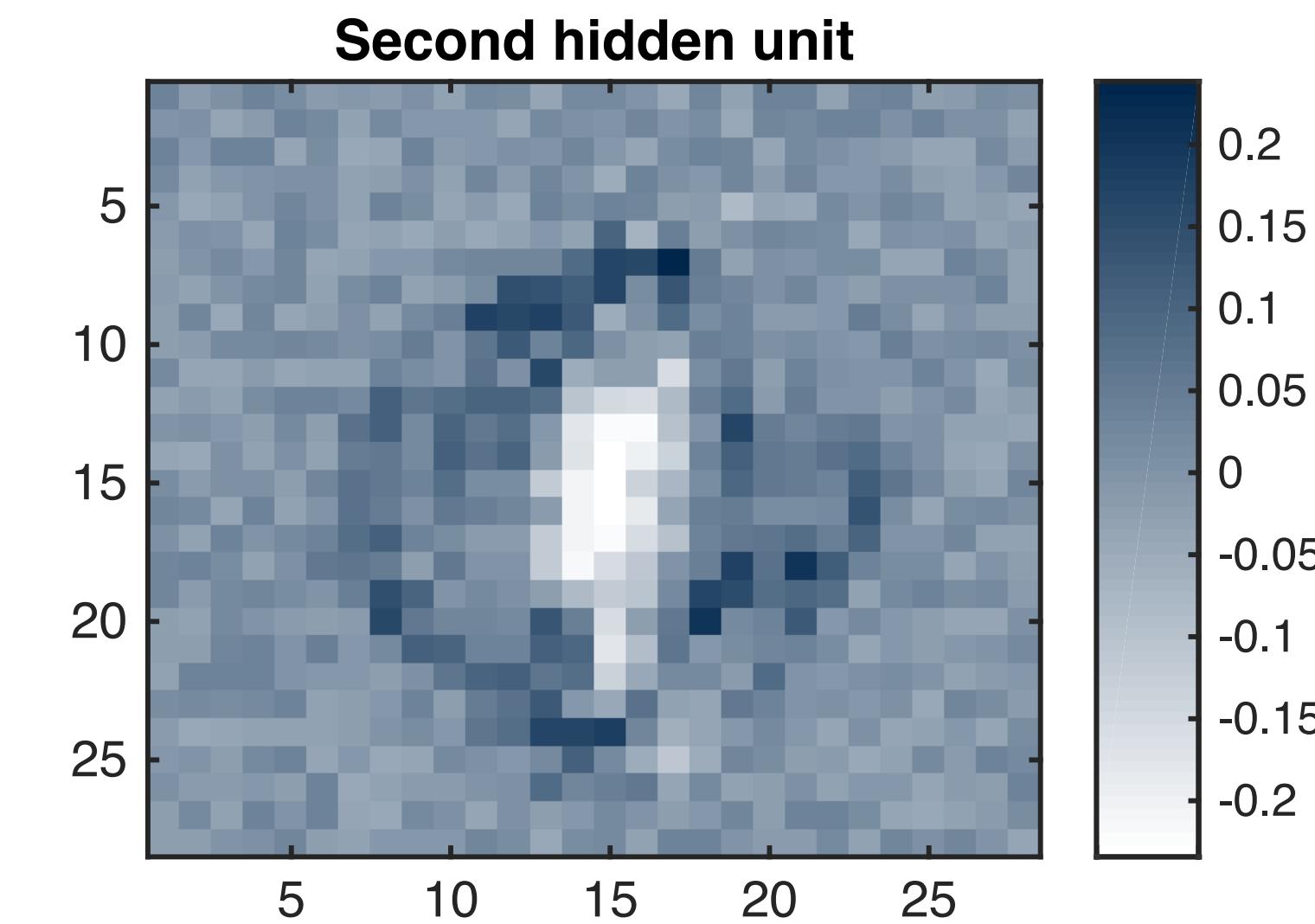
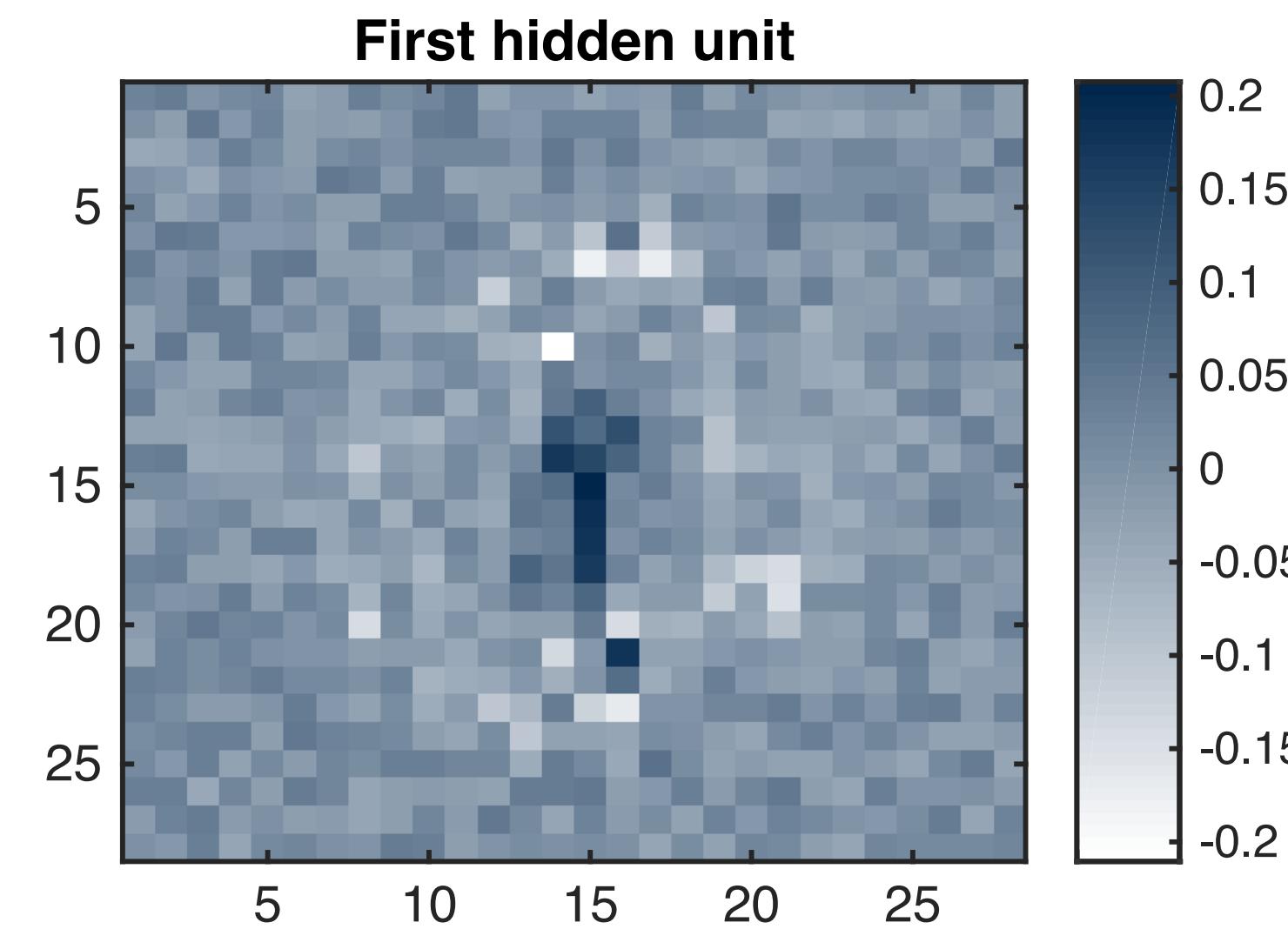
Example classifier

- A “0” vs. “1” digit classifier
 - 2-layer neural net with 2 hidden units and one output



Learned weights

- First layer is a “feature” transform picking 0 or 1
- Second layer is a simple threshold classifier



The magic ingredient is the non-linearity

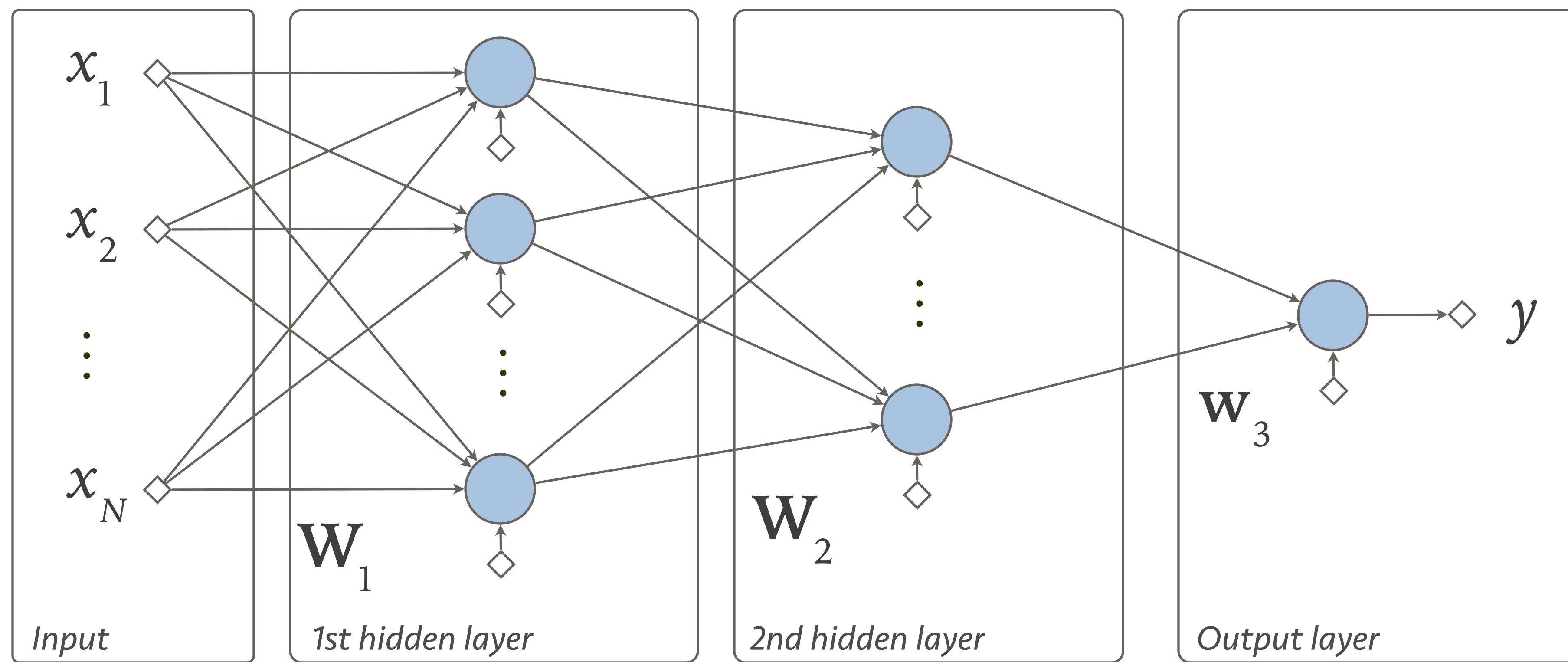
- At every layer we apply a sigmoid:

$$y = g\left(W_1 \cdot g\left(W_2 \cdot x\right)\right)$$

- This allows us to meaningfully *stack* transforms
 - And hence obtain a deep architecture

Multiple levels of features

- First layer contains low-level features, second layer contains mid-level features, ..., final layer is a classifier



Arbitrarily deep architectures

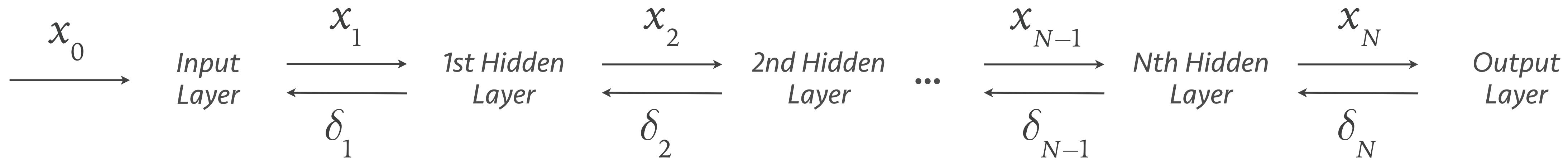
- We can stack as many transforms as we like
 - The goal is to find rich representations
 - Thus we make a "deep" model
- The goal is to get features of features of features of ...
 - "Like the brain does"

Whoa, wait a minute ...

- You told us that three layers are enough for anything!
 - Why should we use more layers than that?
 - Aren't shallow models good enough?
- Yes, a shallow model is fine, but
 - There is no guarantee that you'll easily find the parameters!
 - Nor that you won't need a bazillion units

But there is a problem with depth

- Deep architectures have lots of parameters
 - In some cases in the billions!
- Typical neural net optimization becomes a problem



Vanishing gradient

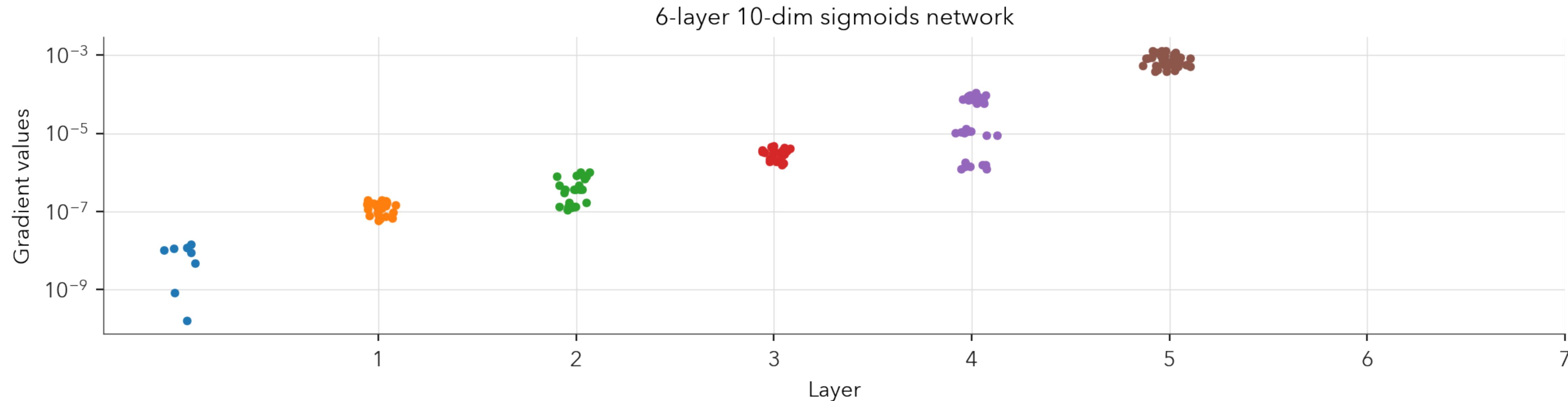
- As we calculate the gradient, we get smaller and smaller values propagating through each layer

$$\mathbf{x}_i = g\left(\mathbf{W}_i \cdot \mathbf{x}_{i-1}\right)$$

Usually between {-1, 1}

$$\delta_i = \left(\mathbf{W}_{i+1}^\top \cdot \delta_{i+1}\right) \cdot g'\left(\mathbf{W}_i \cdot \mathbf{x}_{i-1}\right)$$

Usually much smaller than 1



Problems with backpropagation

- Through many layers gradients become too small
 - We can't propagate errors through too many layers
- Potentially lots of local minima (due to lots of parameters)
 - Even with shallow networks this can be a problem
- Biologically it's a stretch
 - Does the visual cortex influence the retina?
 - Do we really have target values?

Desiderata

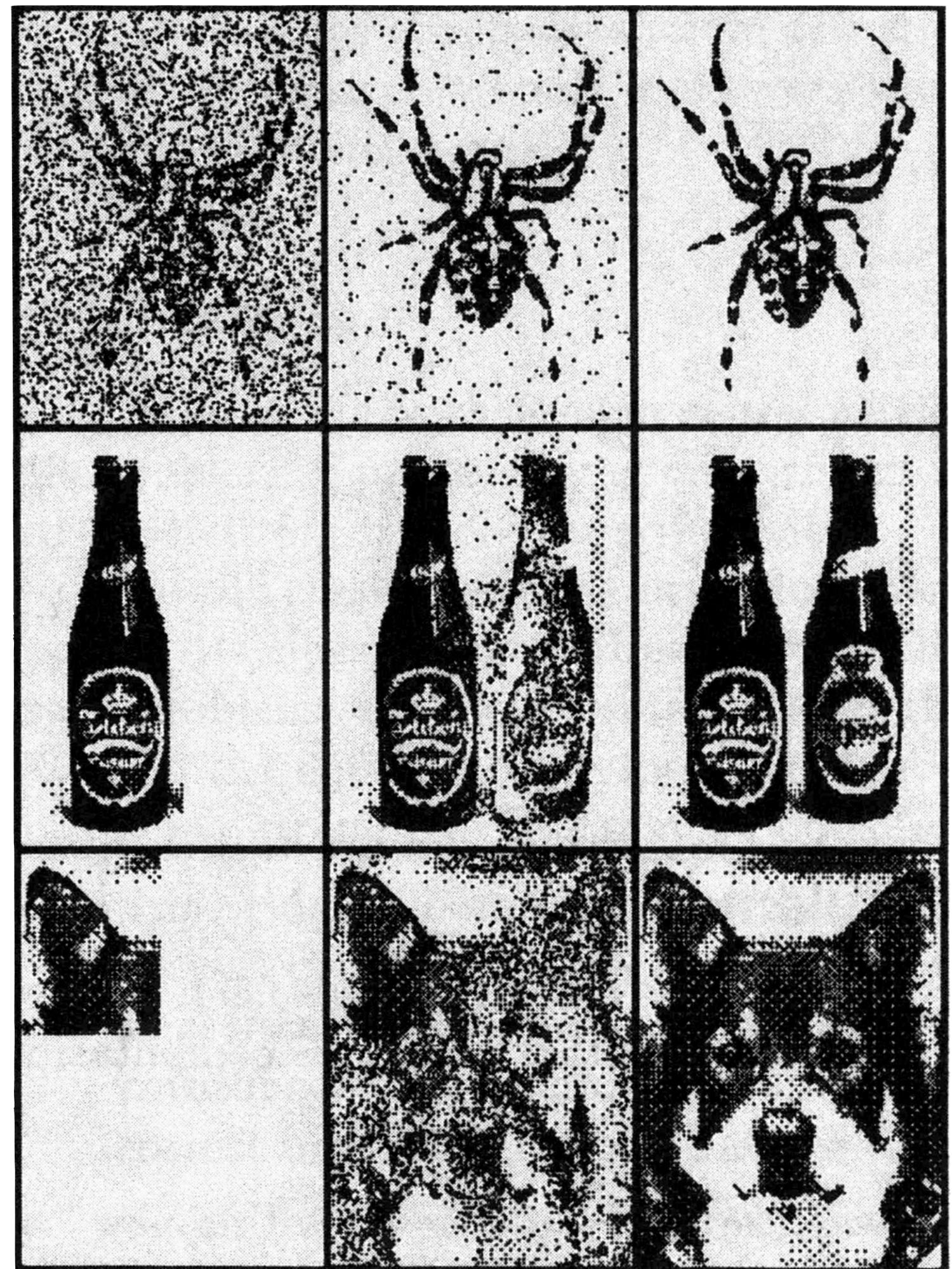
- We need a numerically stable learning procedure
- We like biological plausibility
 - Computations should be mostly local
 - We like distributed systems
- We thus need to move away from what we know

A digression

- A bit on stochastic neural networks
 - A different structure from what we've seen so far
 - Hopfield and Boltzmann models
- Strong influences from physics and neuroscience
 - One of the factors in the resurgence of neural nets

The Hopfield/Boltzmann networks

- Auto-associative memory
 - Learns patterns by finding stable equilibria
- Fully connected and recurrent
 - All nodes have a binary state
- Model can learn to recall patterns

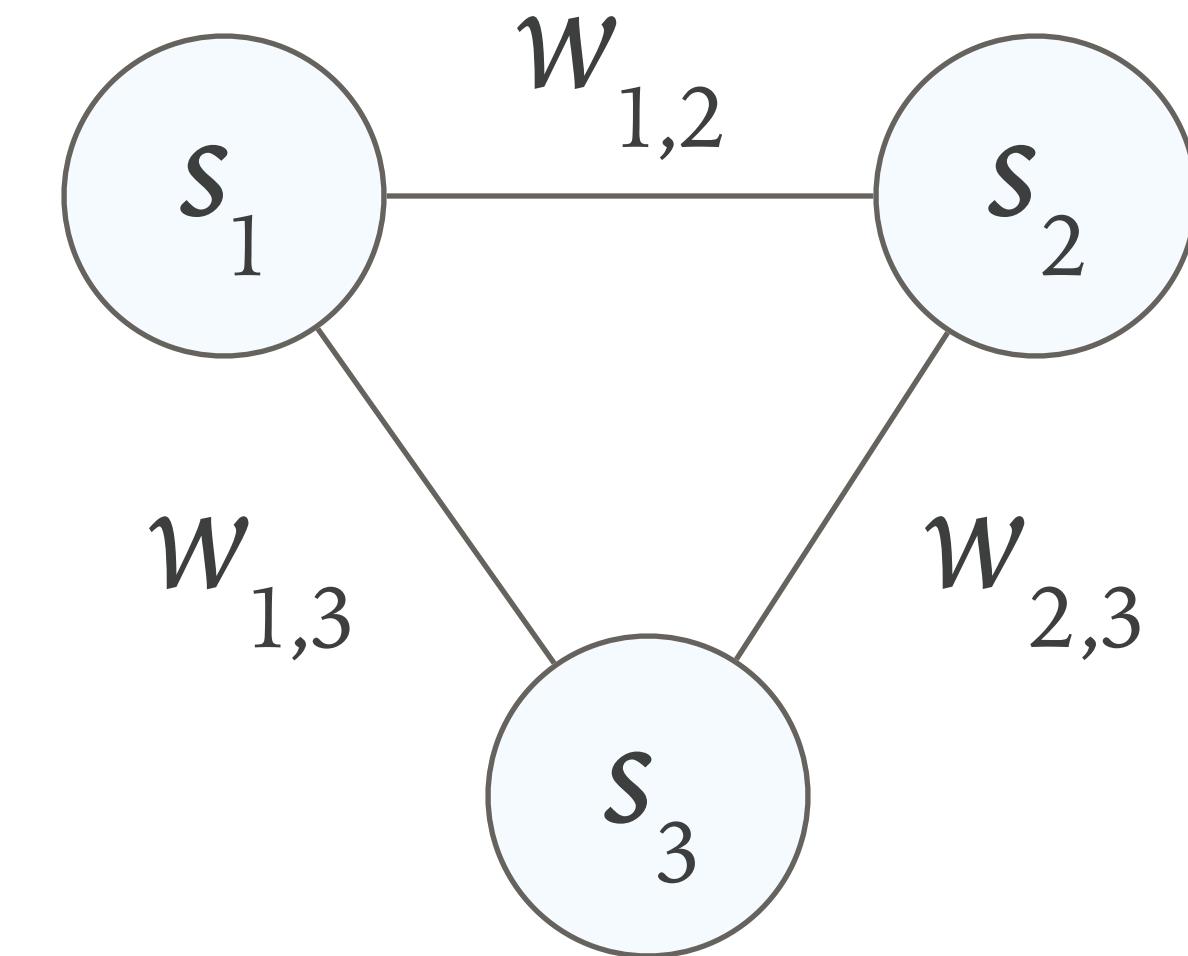


Energy minimization

- Parameters: State s_i , Weight w_{ij} , Threshold θ_i
- After assigning patterns we minimize model “energy”

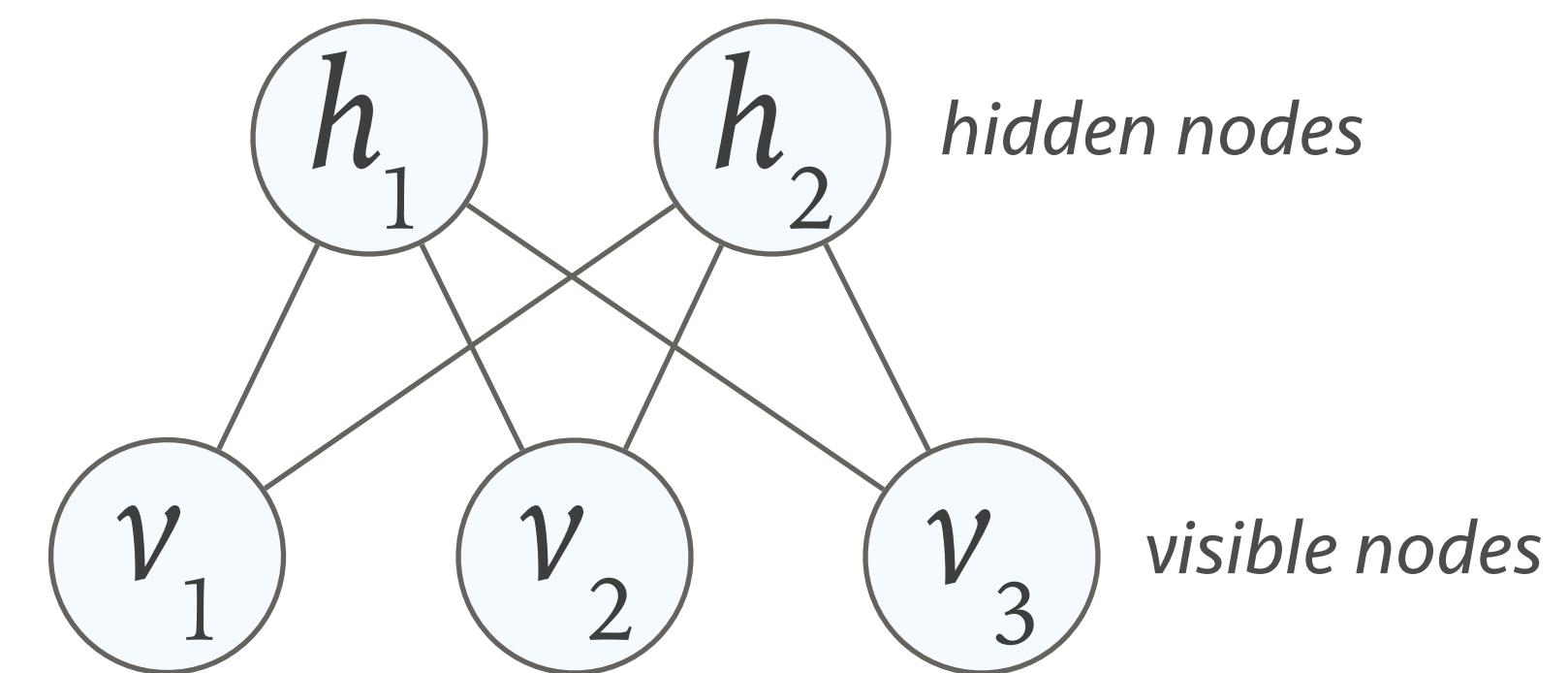
$$E = -\sum_{i < j} w_{i,j} s_i s_j + \sum_i \theta_i s_i$$

- Painful optimization problem!
 - Gradient solution for Hopfield
 - Stochastic solution for Boltzmann



The Restricted Boltzmann Machine (RBM)

- A more manageable form of Boltzmann machines
 - Two-layer, no intra-layer connections
- Visible and hidden nodes
 - Visible nodes represent known data
 - Hidden nodes are used for internal representation
- Easier to train and very useful for many tasks



Getting a handle on RBMs

- Define a *probability* of the network energy:

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{a}^\top \cdot \mathbf{v} - \mathbf{b}^\top \cdot \mathbf{h} - \mathbf{h}^\top \cdot \mathbf{W} \cdot \mathbf{v}$$

$$P(v, h) = \frac{1}{Z} e^{-E(v, h)}$$

- Z is a *partition function* and \mathbf{a}, \mathbf{b} are node biases
- We also define the node probabilities

$$P(h_i | v) = g\left(b_i + \sum_j w_{i,j} v_j\right), \quad P(v_i | h) = g\left(a_i + \sum_j w_{i,j} h_j\right)$$

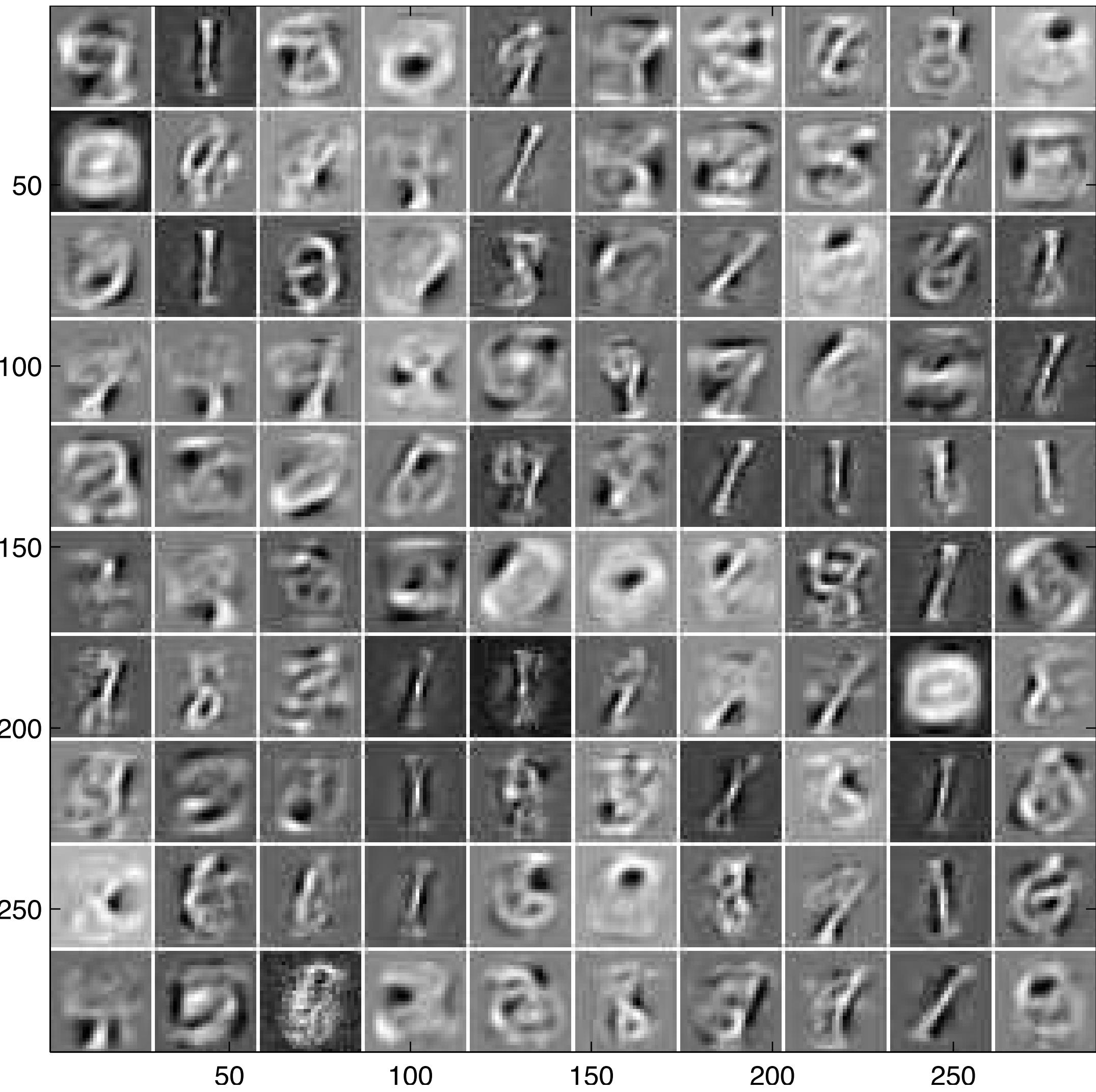
Contrastive Divergence learning

- Maximize product of all $P(\mathbf{v})$
 - 1) For a training sample v_1 compute hidden node probabilities
 - 2) Generate a hidden layer activation vector h_1 from above
 - 3) Go back and generate a new input vector v_2 based on h_1
 - 4) Go forth and generate a new activation vector h_2 from v_2
 - 5) Update corresponding w using: $\Delta w \propto v_1 h_1 - v_2 h_2$

"Positive gradient" 
"Negative gradient" 

So what does this do?

- Example on digit data
- 28×28 visible nodes
 - Set each pattern to a digit
- 100 hidden nodes
- W is 784×100
 - i.e. 100 “basis” functions

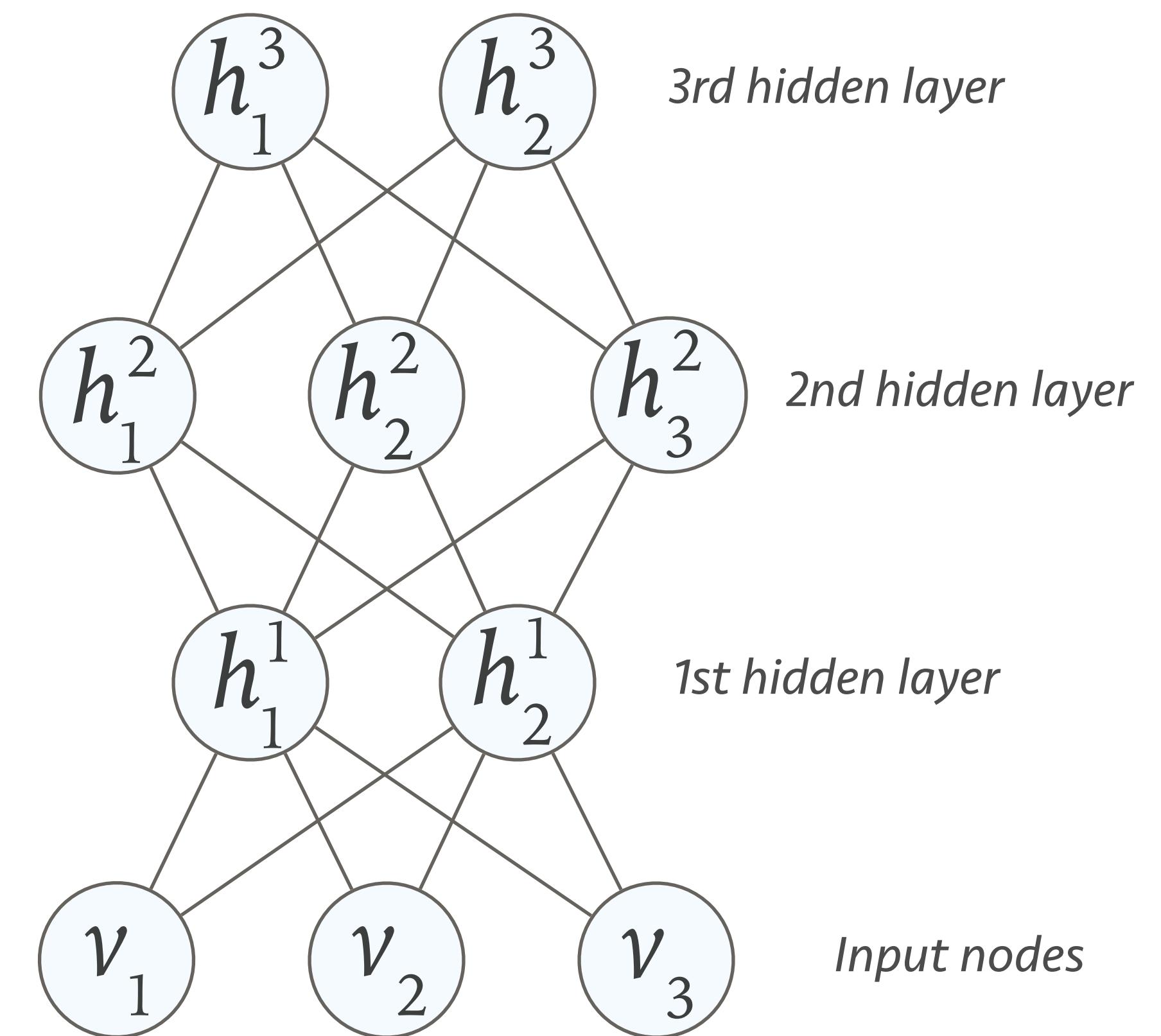


Back to deep learning

- The RBM is a shallow learner
- But we can use it to connect multiple layers
 - Treat each layer in a multi-layer input as an RBM
- The big idea: Train locally, group globally
 - This helps computational complexity and is biologically plausible

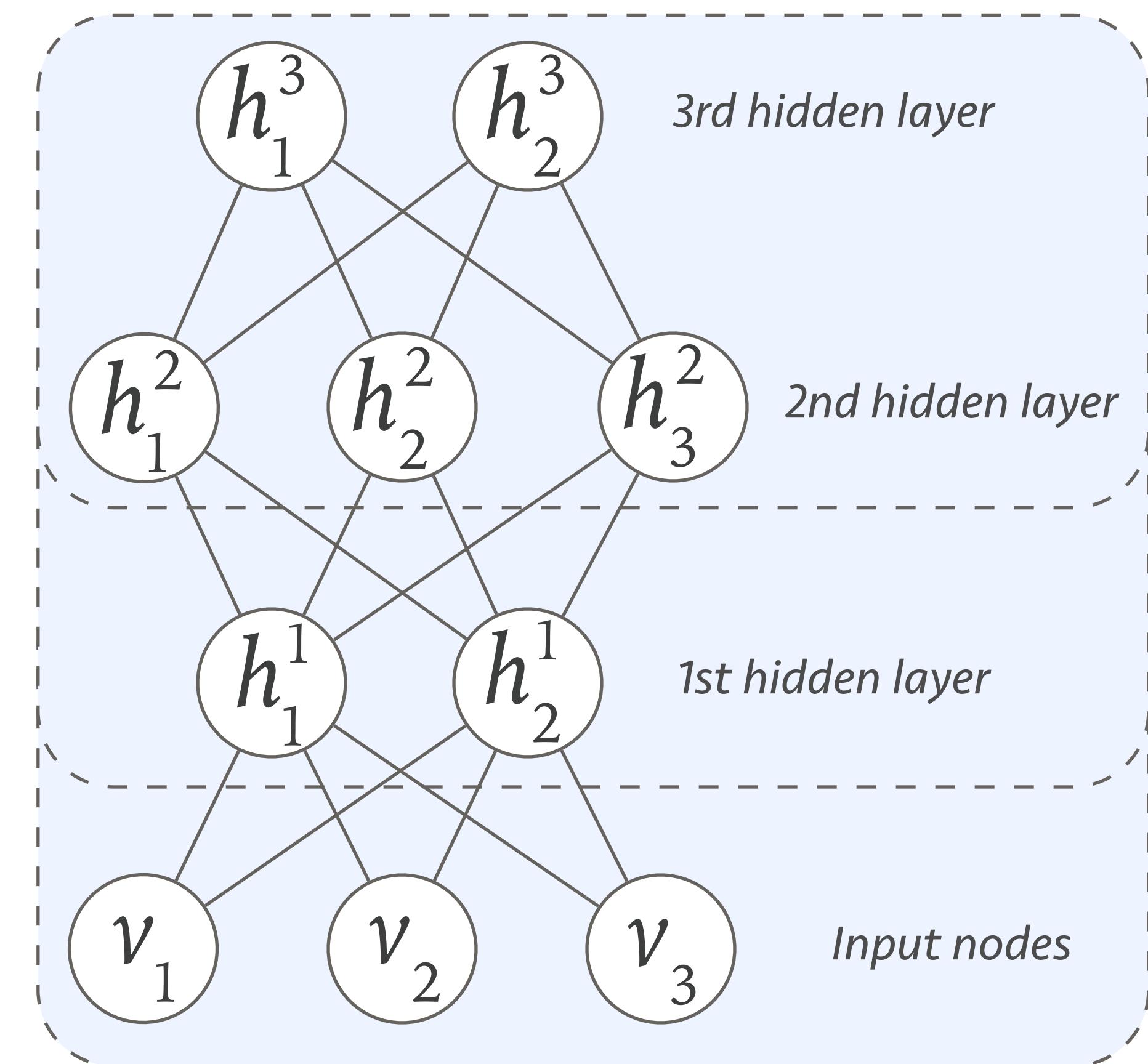
Deep Belief Networks (DBN)

- A stack of multiple RBMs
 - A deep generative model
- Initial weights are set using RBM training
- Further refinement using backpropagation



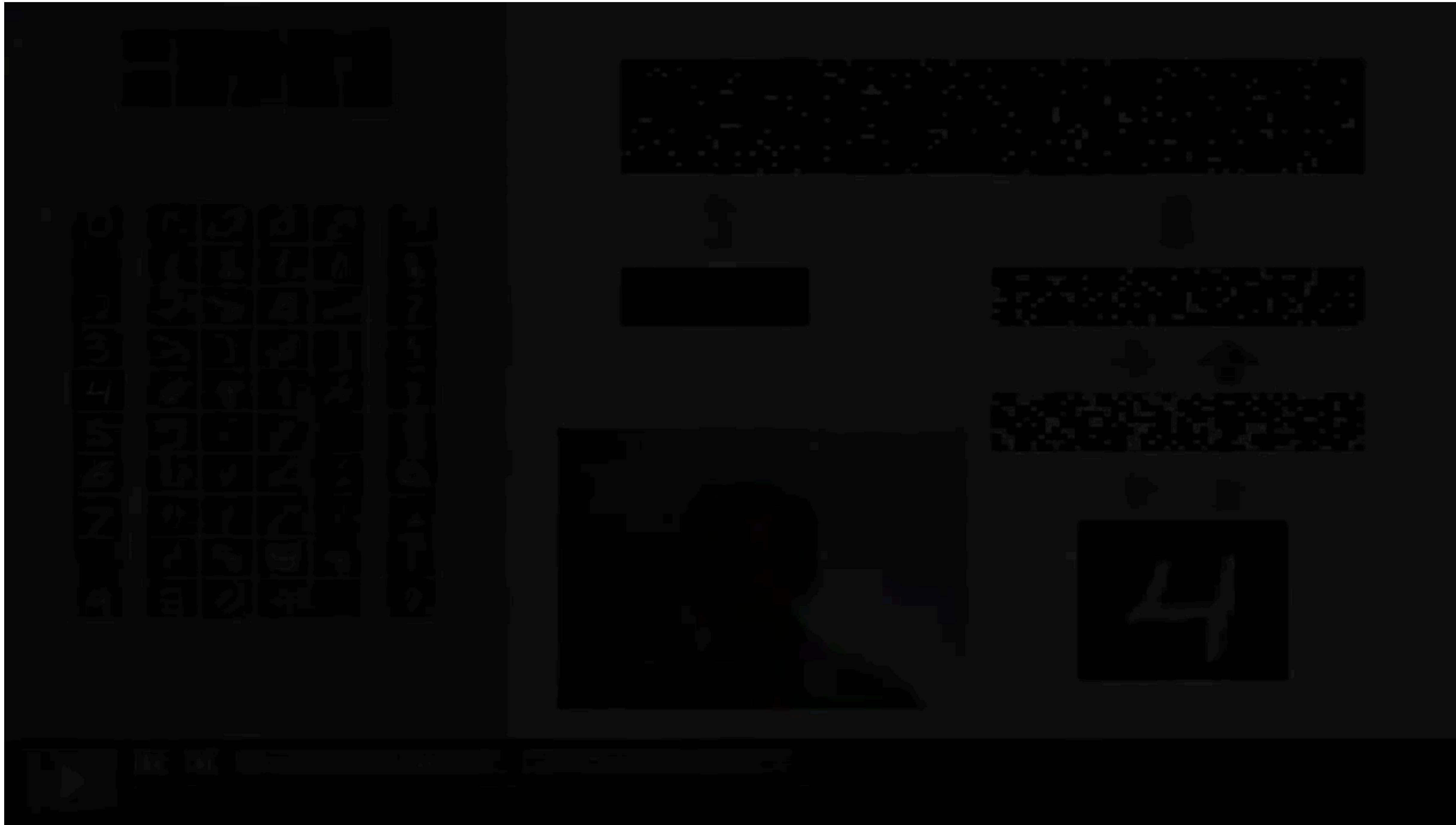
Greedy learning

- Step 1: Train an RBM for first layer
 - Visible nodes are the inputs
- Step 2: Fix hidden layer
 - Pretend it's visible and train next layer
- Step 3: Keep going



So what is it good for?

- We can learn complex representations of data
 - Learn a deep model with multiple feature levels
- We can learn to classify
 - Use visible nodes to represent classes
- Example simulations on digit data:
 - Hinton's Neural Network Simulation (Generative)
 - Demo: <https://www.youtube.com/watch?v=KuPaiOogiHk#t=47s>
 - 10 labels / 2000 l3 units / 500 l2 units / l1 500 units / 784 pixels



Some well-known press exposure

- A billion weights network trained on 10M YouTube frames
 - 1,000 machines for 3 days!
- Conclusion: YouTube has lots of cats! :)
 - and that we can get some great features that way

The cat neuron



The human body neuron

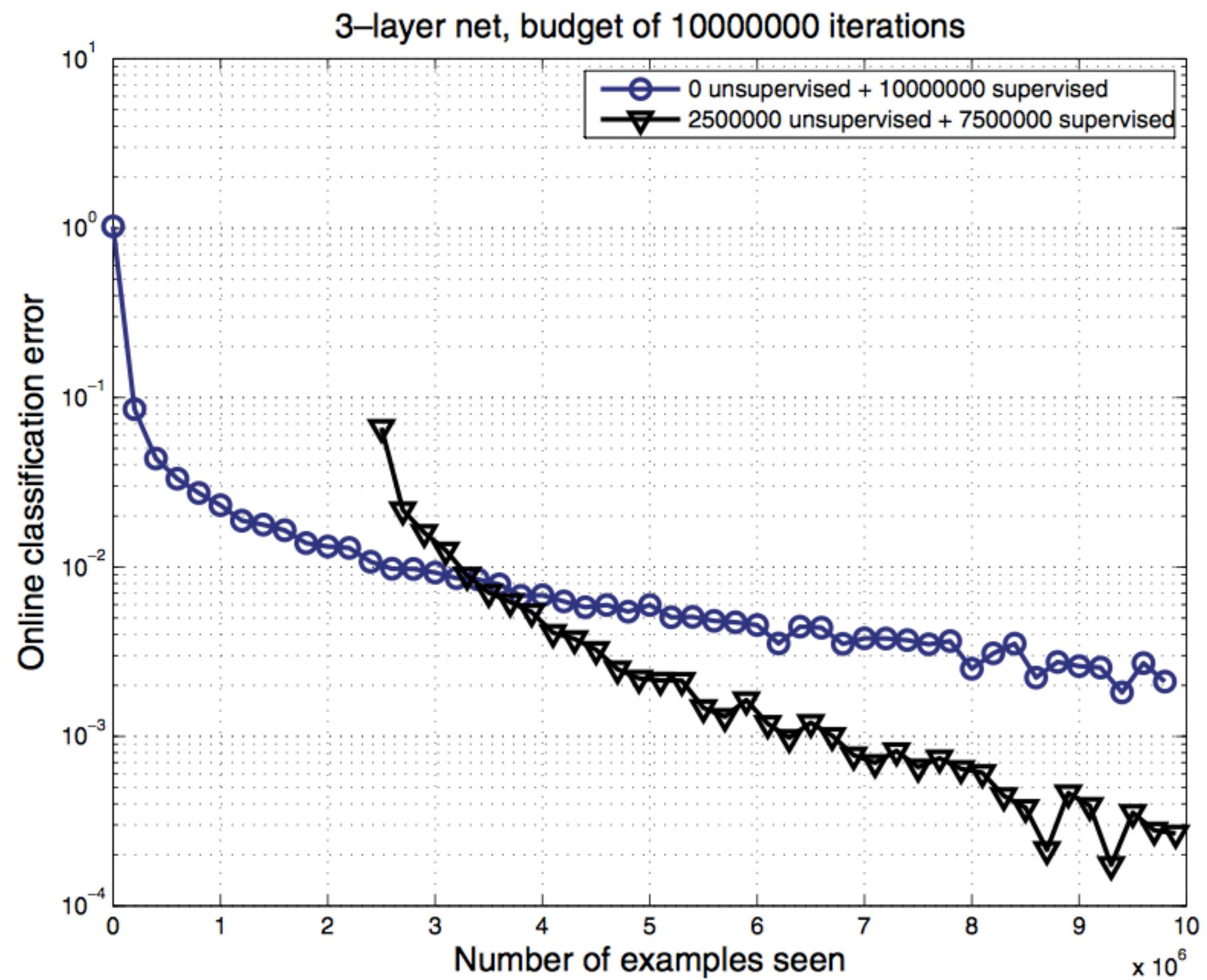


Helping out backpropagation

- We can also use this approach to learn large networks
 - e.g. a multiple layer classification network
- Use greedy learning to find initial values for weights
 - Treat each layer set as an RBM
- Once trained use as initial values for backprop

The importance of a good start

- Find “sensible” weight values
 - Don’t start from irrelevant points
- Starts from a space that is well-tuned to the data at hand
- Reduces the amount of required computations

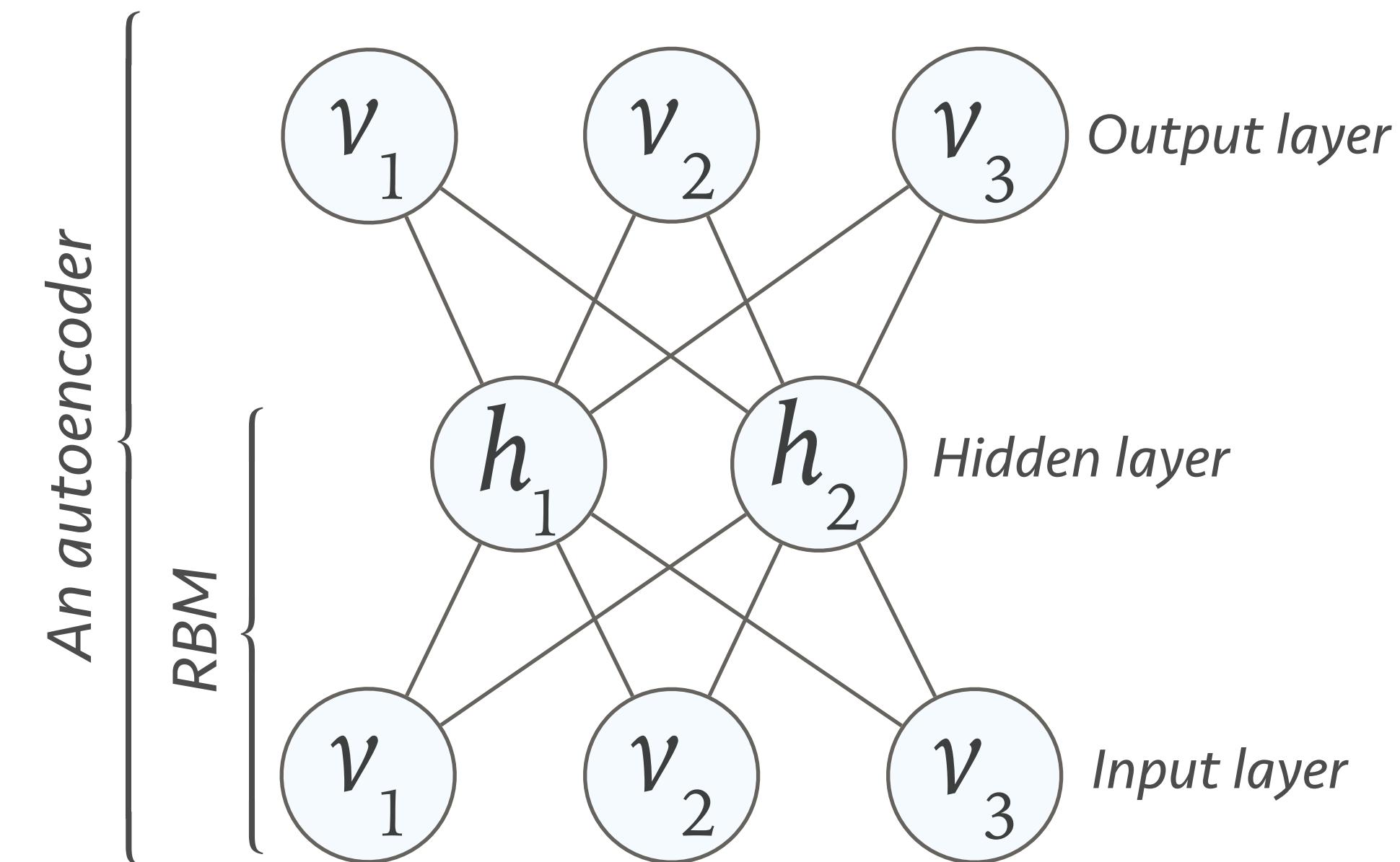


Ugh, that RBM business is difficult ...

- There's no reason to stick to RBMs
 - The math is a little tricky and the optimization costly
- Instead we could use another type of “shallow” learner
 - But it has to have a structure conducive to what we want
 - E.g. overcomplete ICA so that we can have more output nodes
- Or we can use an *autoencoder*

Autoencoders

- A very simple approach to designing a shallow non-linear feature extractor
- Try to learn an identity mapping
 - but we won't make it that easy!
 - We won't give it enough resources



More formally

- Neural net autoencoders
 - Series of layers that lead to a lower dimensional latent representation

$$\mathbf{h} = g(\mathbf{W}_e \cdot \mathbf{x} + \mathbf{b}_e) \in \mathbb{R}^K$$

$$\hat{\mathbf{x}} = g(\mathbf{W}_d \cdot \mathbf{h} + \mathbf{b}_d) \in \mathbb{R}^{M>K}$$

or

$$\begin{aligned}\mathbf{h} &= g\left(\mathbf{W}_{e_2} \cdot g\left(\mathbf{W}_{e_1} \cdot \mathbf{x} + \mathbf{b}_{e_1}\right) + \mathbf{b}_{e_2}\right) \\ \hat{\mathbf{x}} &= g\left(\mathbf{W}_{d_2} \cdot g\left(\mathbf{W}_{d_1} \cdot \mathbf{h} + \mathbf{b}_{d_1}\right) + \mathbf{b}_{d_2}\right)\end{aligned}$$

- And we can just learn this with plain backpropagation

Ways to constraint an autoencoder

- Restrict the number of hidden nodes
 - Creates an information bottleneck
 - Resulting in an informative low-rank representation
- Go to higher dimensions but use sparsity
 - Creates informative “bases” and projects to high-D space
- Add some structure to the form of the layers
 - e.g. orthogonality, independence, non-negativity, etc.

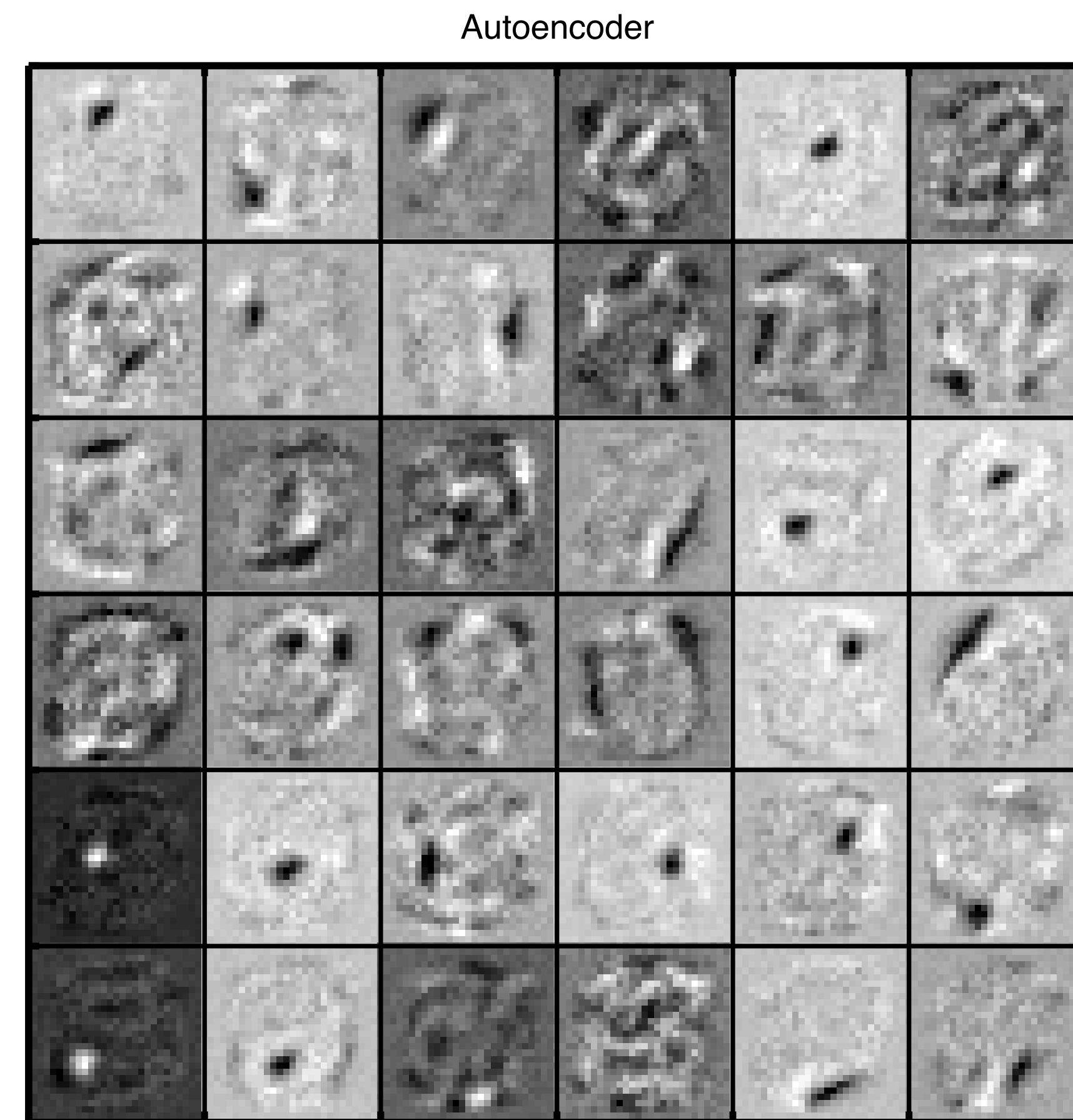
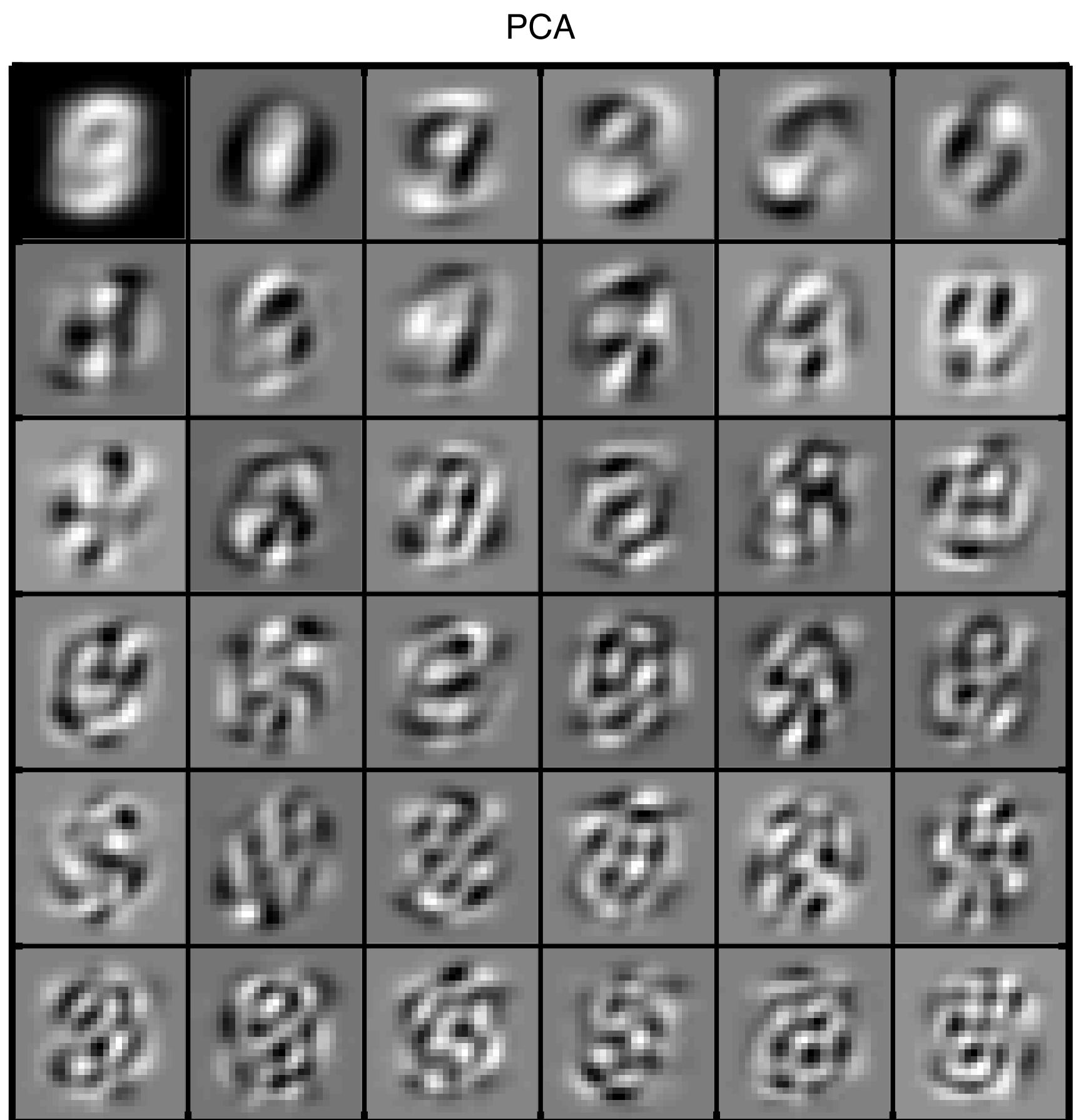
Neural net PCA

Neural net ICA

Neural net NMF

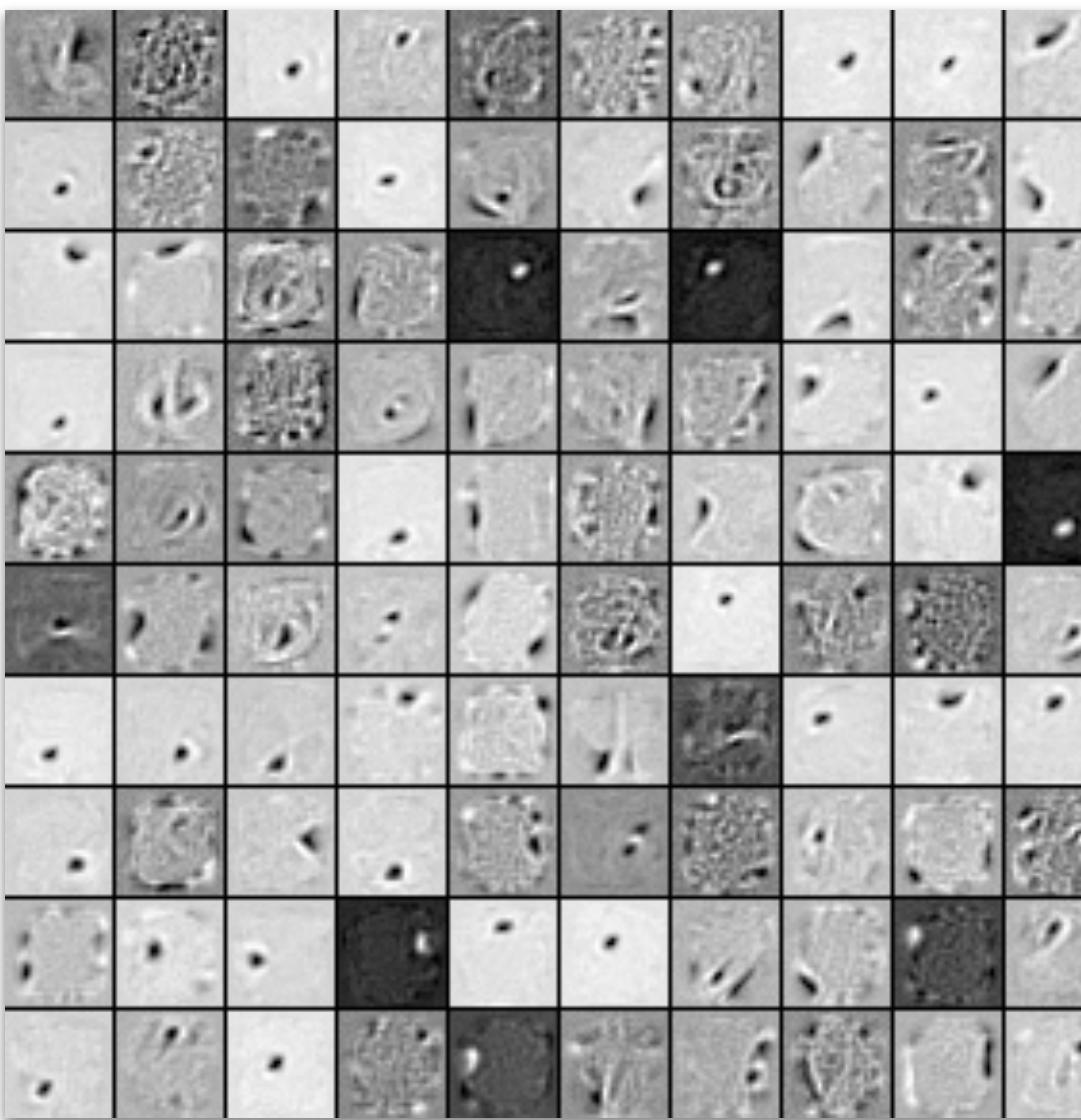
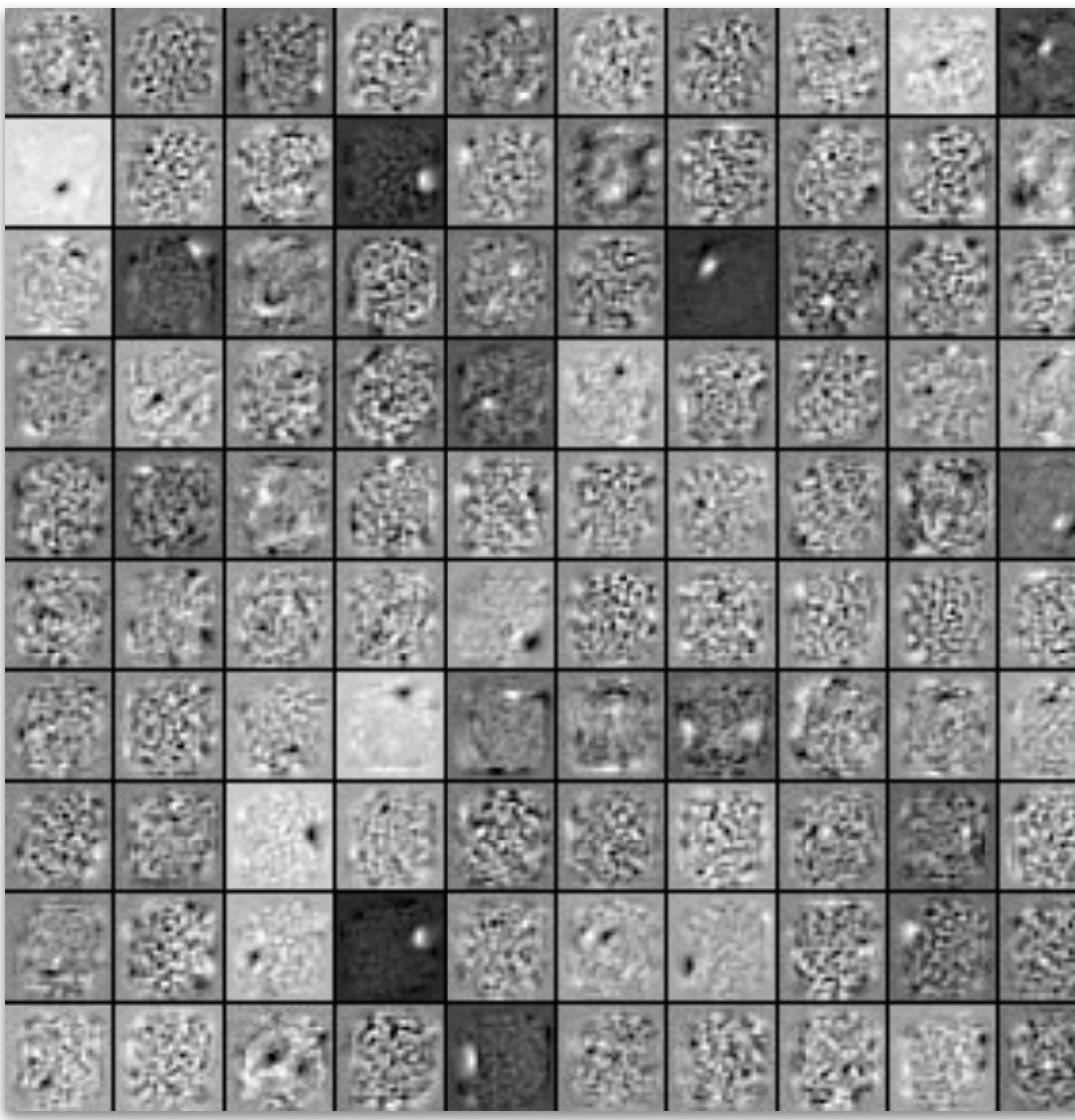
Example case

- Unconstrained AE on digit data



Noisy autoencoders

- Find a “robust” representation
 - But stochastically corrupt the input
 - e.g. adding noise, removing random bits, transform it in non-linear ways, etc.
- Now the input is not always the same as the output
 - We need robust features that map all the noisy inputs to the proper output

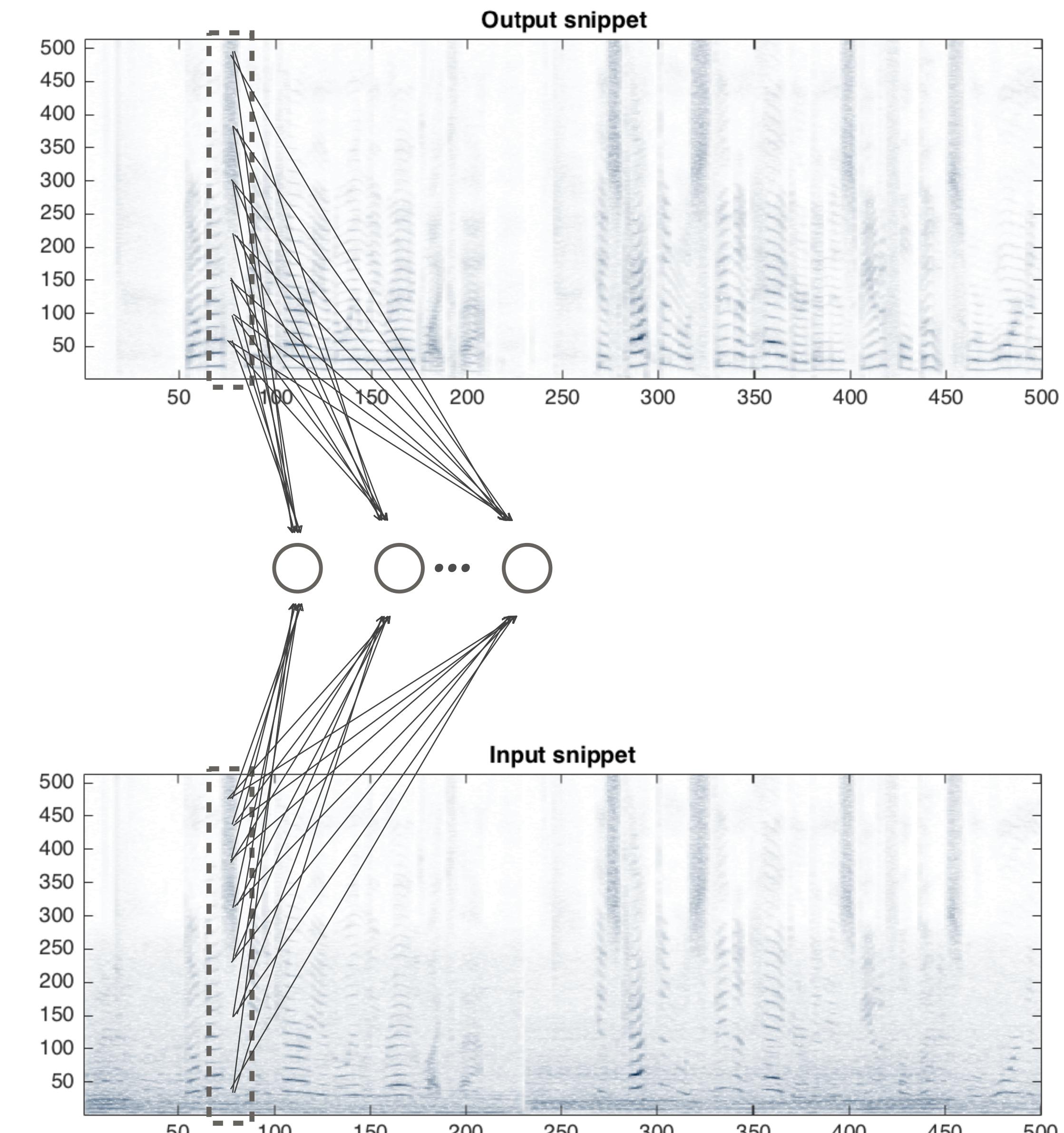


Noisy autoencoders for enhancement

- We can also use noisy autoencoders to clean signals
 - Learn to predict a desirable output for a noisy input
- Can be used for multiple enhancement tasks
 - Removing noise, recovering higher resolution, ...
- Generate noisy/clean training data and learn a network

Toy example: Speech denoising

- Trained on 30sec inputs
 - Speech + street noise
 - Known speaker
 - Takes 30sec to train
 - on a laptop (2-3sec with GPU)
- Parameters
 - 1024pt spectra
 - 1 hidden layer, 100 nodes
 - Leaky ReLU activations



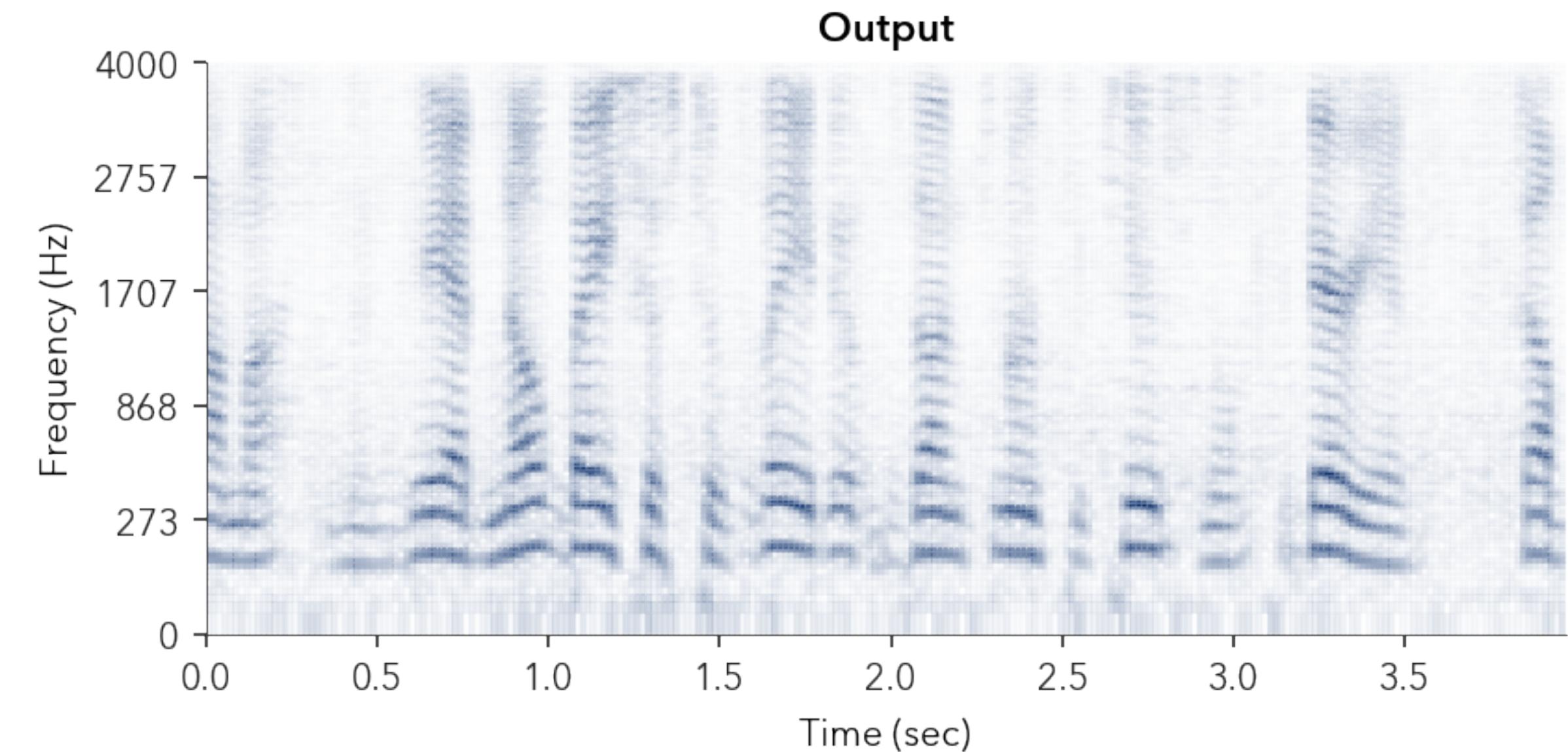
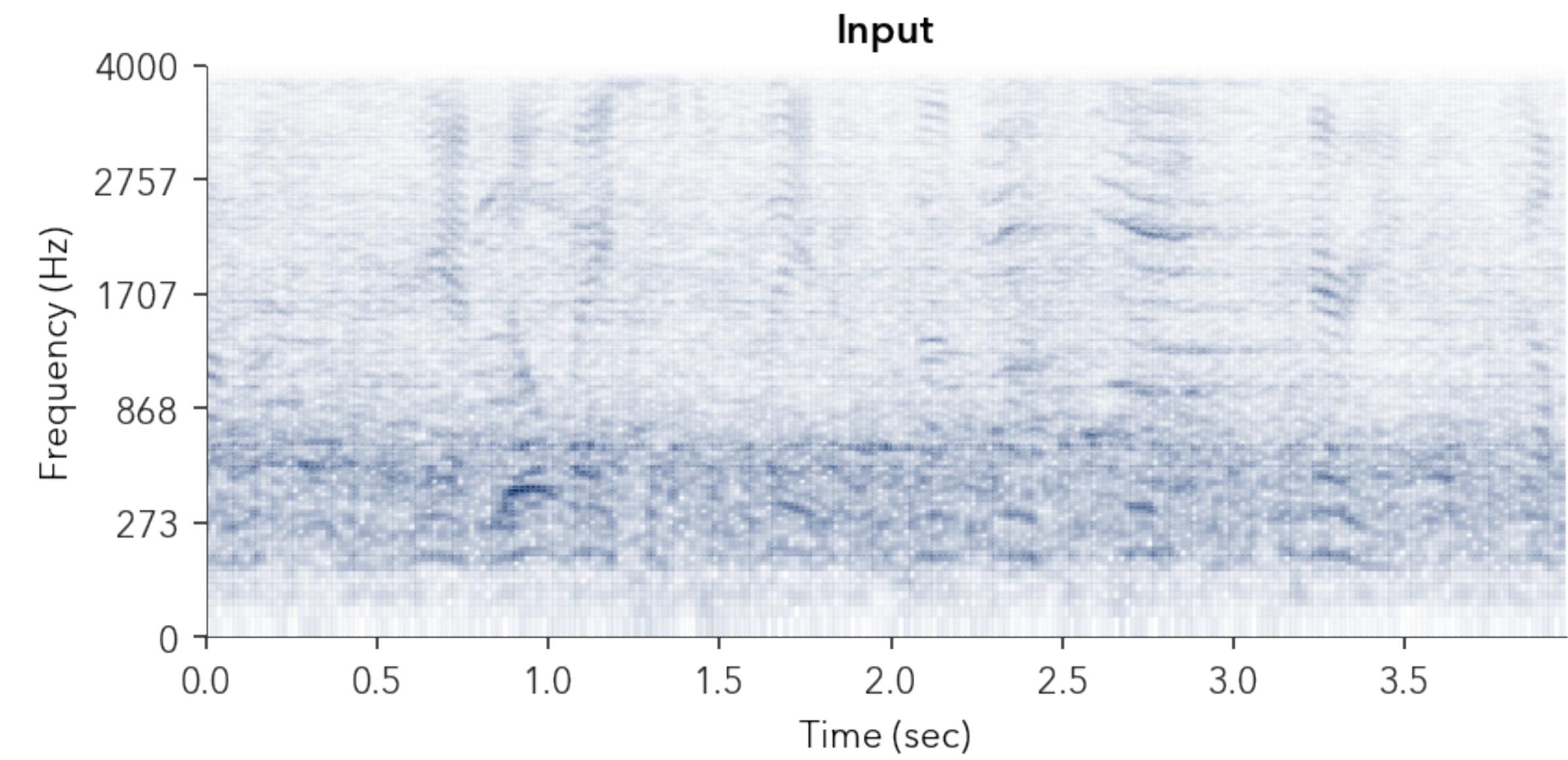
Runtime denoising

- Very lightweight process
 - ~300x real-time
 - 0.01sec in this case
- Works better than NMF
 - But cannot generalize to new noise types!

Input mixture

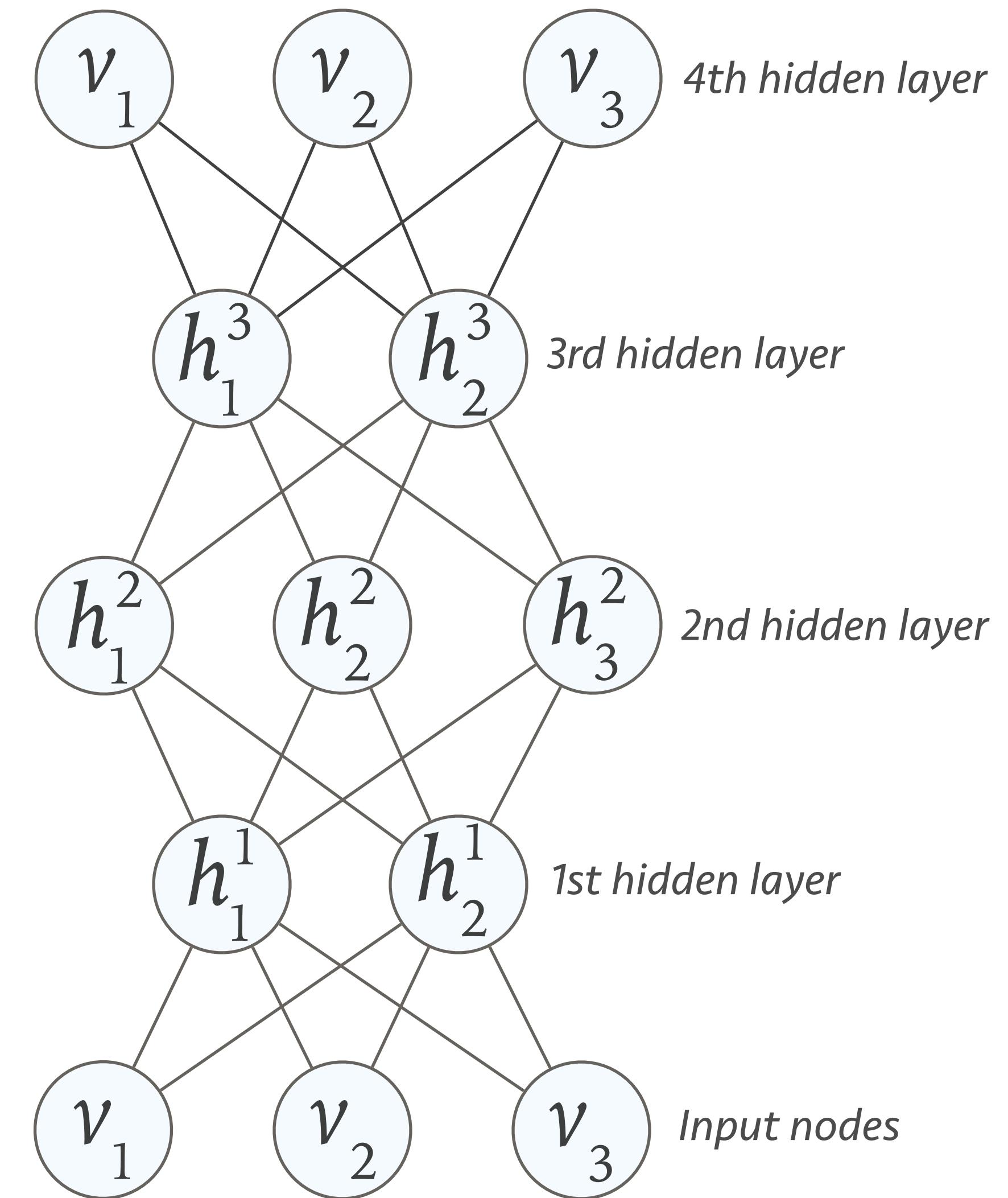


Predicted output



Stacked autoencoders

- Similar to DBNs
 - Multiple-layer architecture
 - Train each layer separately
 - Stack them all in the end
- Can also be used to classify
 - Once trained, add a classification layer and refine with backprop

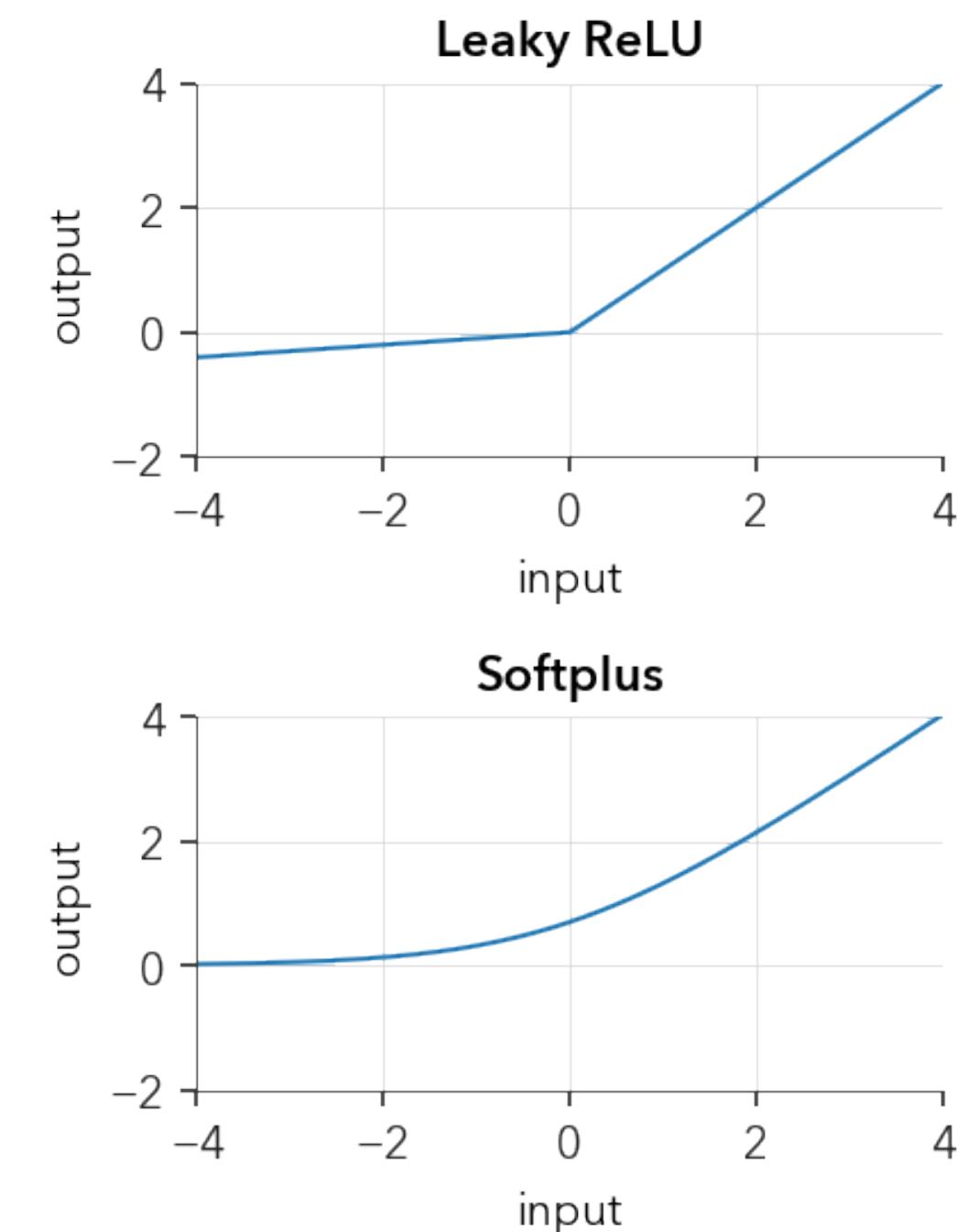
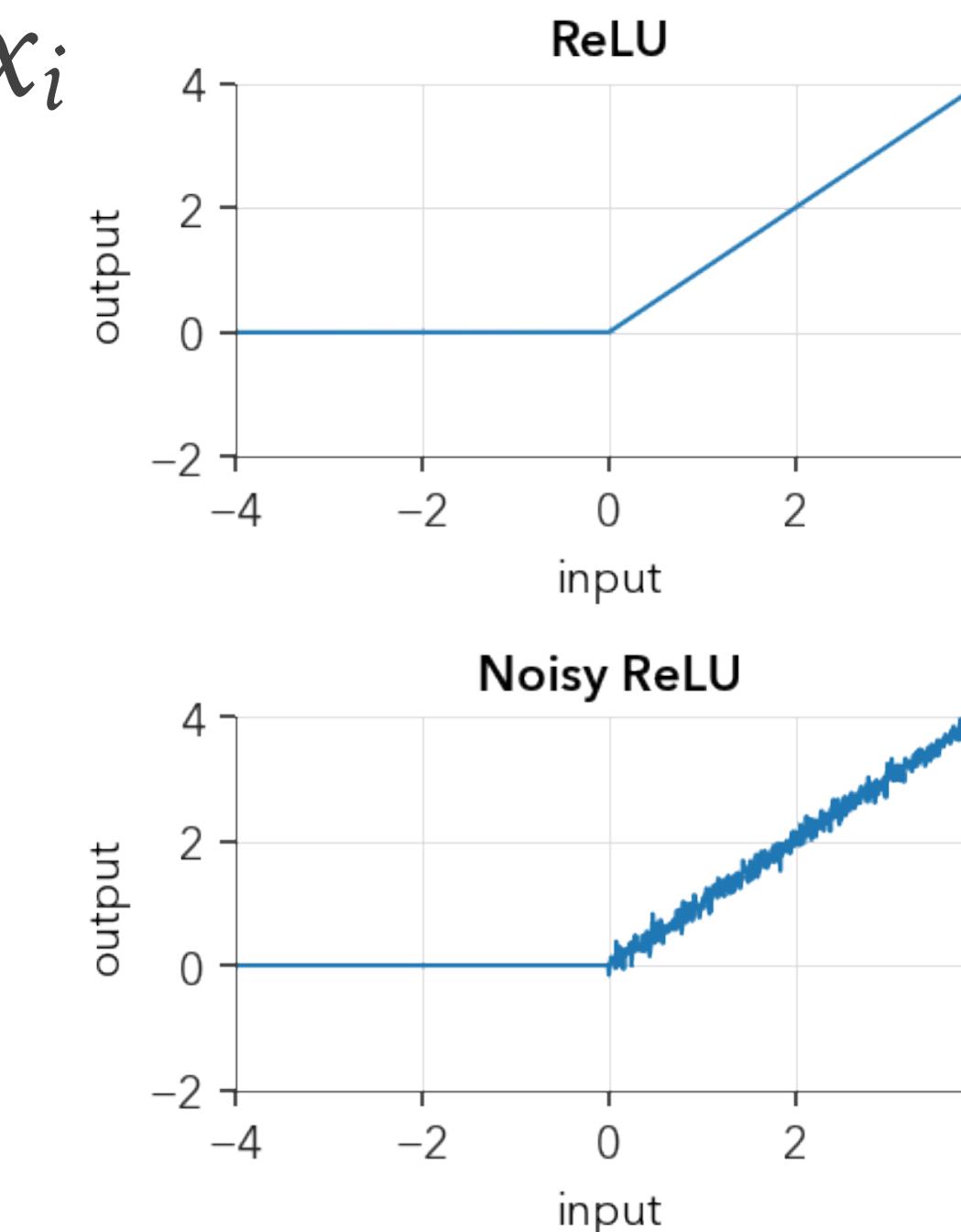


So what's new from the 90's?

- The cynical view: Not much
 - We have more data and faster processors
 - So we can train deeper models with more data
- But, we now have many more tricks of the trade
 - Better activations, smarter training strategies, many little tricks to assist better convergence, more elaborate models than before
 - Many techniques to avoid vanishing gradients

Modern activation functions

- Rectified Linear Units (ReLU's)
 - Instead of a sigmoid use: $y_i = \max(0, x_i)$
 - Much faster since there is minimal computation
 - Leaky versions: $y_i = x_i$ if $x_i > 0$ else $a x_i$
 - Noisy versions: $y_i = \max(0, x_i + n_i), \dots$
 - Doesn't saturate as much
- Softplus: $y_i = \log(1 + e^{x_i})$
 - “Softer” version of ReLU



Dropout for better training

- Drop-out training
 - Randomly “turn off” units at every training iteration
 - Usually 50%
 - Puts pressure on units to be useful
- Results in a more robust networks
 - Equivalent to training multiple nets with shared weights, but slightly different connectivity patterns

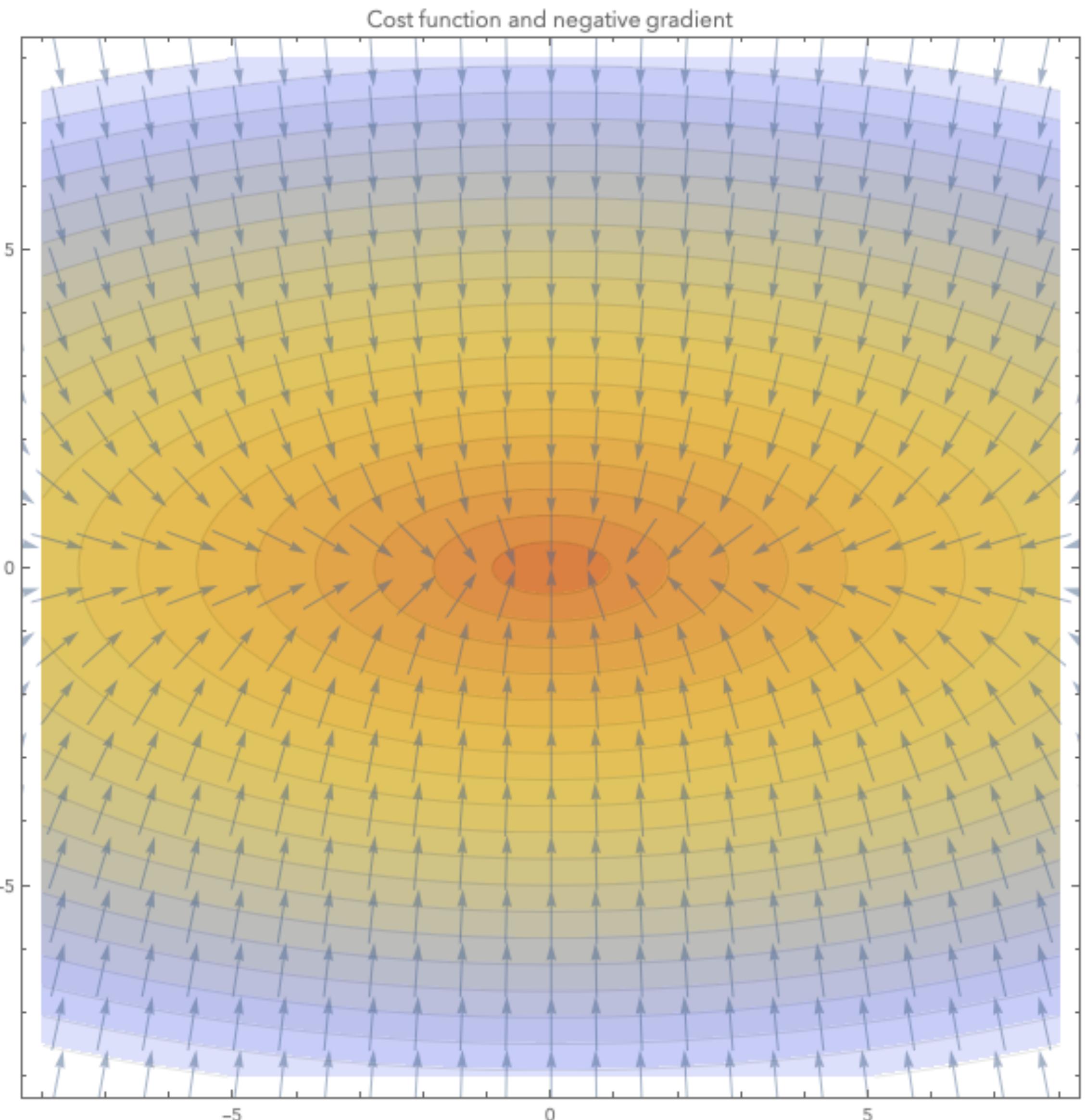
Updating the weights

- Gradient descent

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla f(\mathbf{w}_t)$$

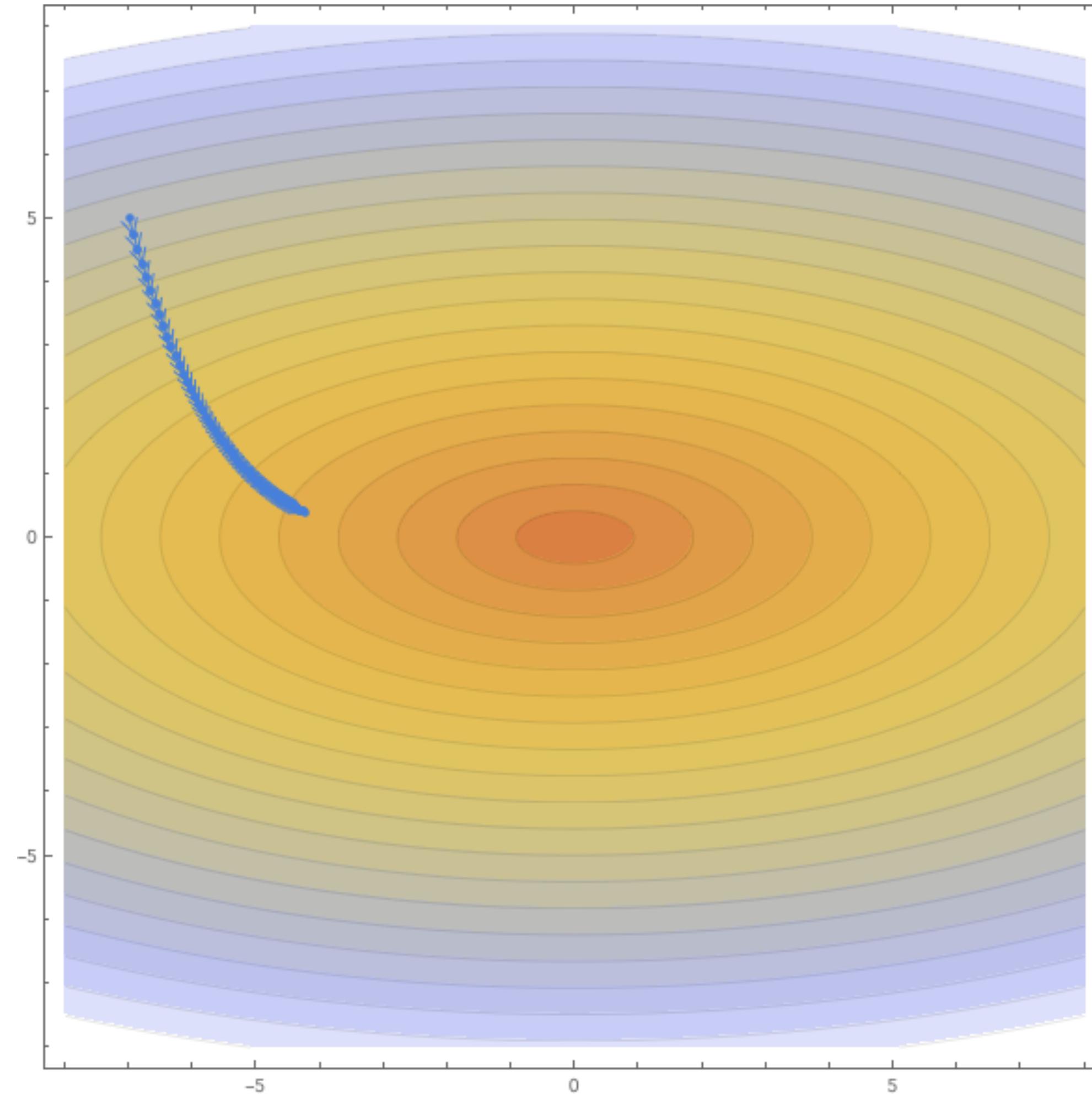
Learning rate ↓
 ↗
Parameters *Neural network*

- η is learning rate
 - How much to move along
 - Usually $\eta = 0.01$ -ish
- We can also average over multiple passes (batch)

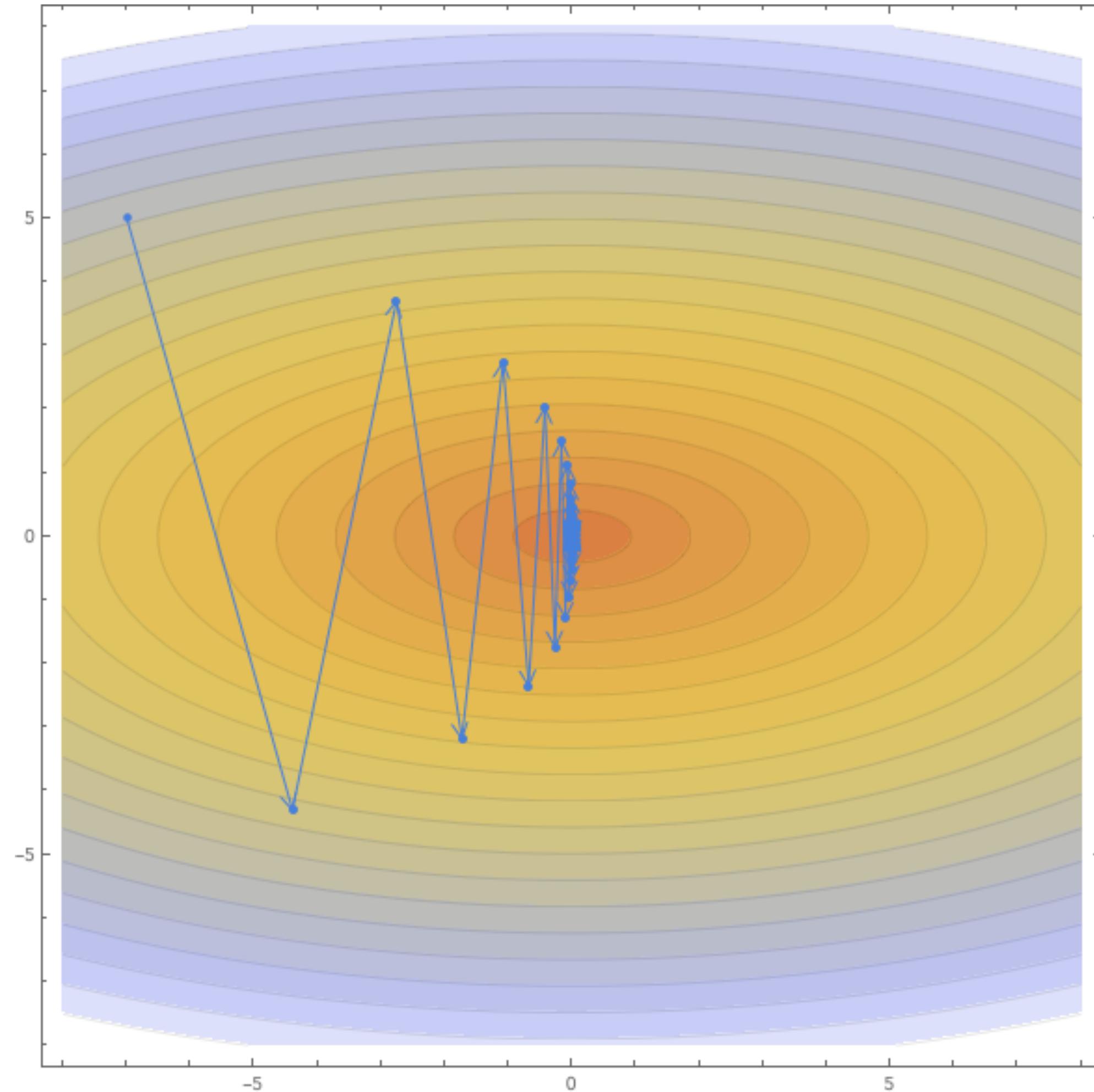


Some problems

Small learning rate → slow convergence



Large learning rate → erratic movements



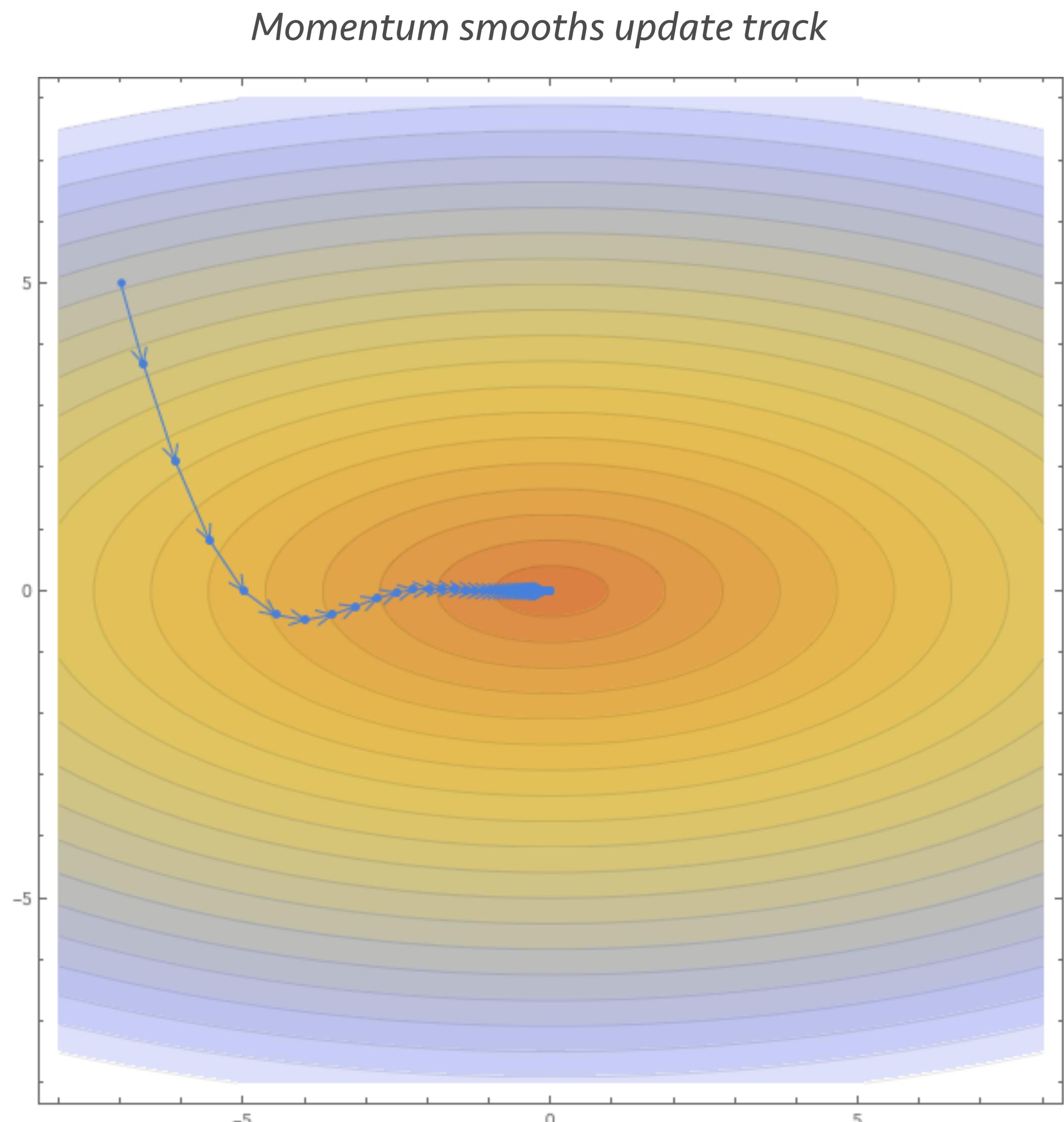
Momentum

- Lowpass filter the gradient
 - Removes jerkiness
 - Reinforces consistent movements

$$\mathbf{u}_t = \eta \nabla f(\mathbf{w}_t) + \mu \mathbf{u}_{t-1}$$

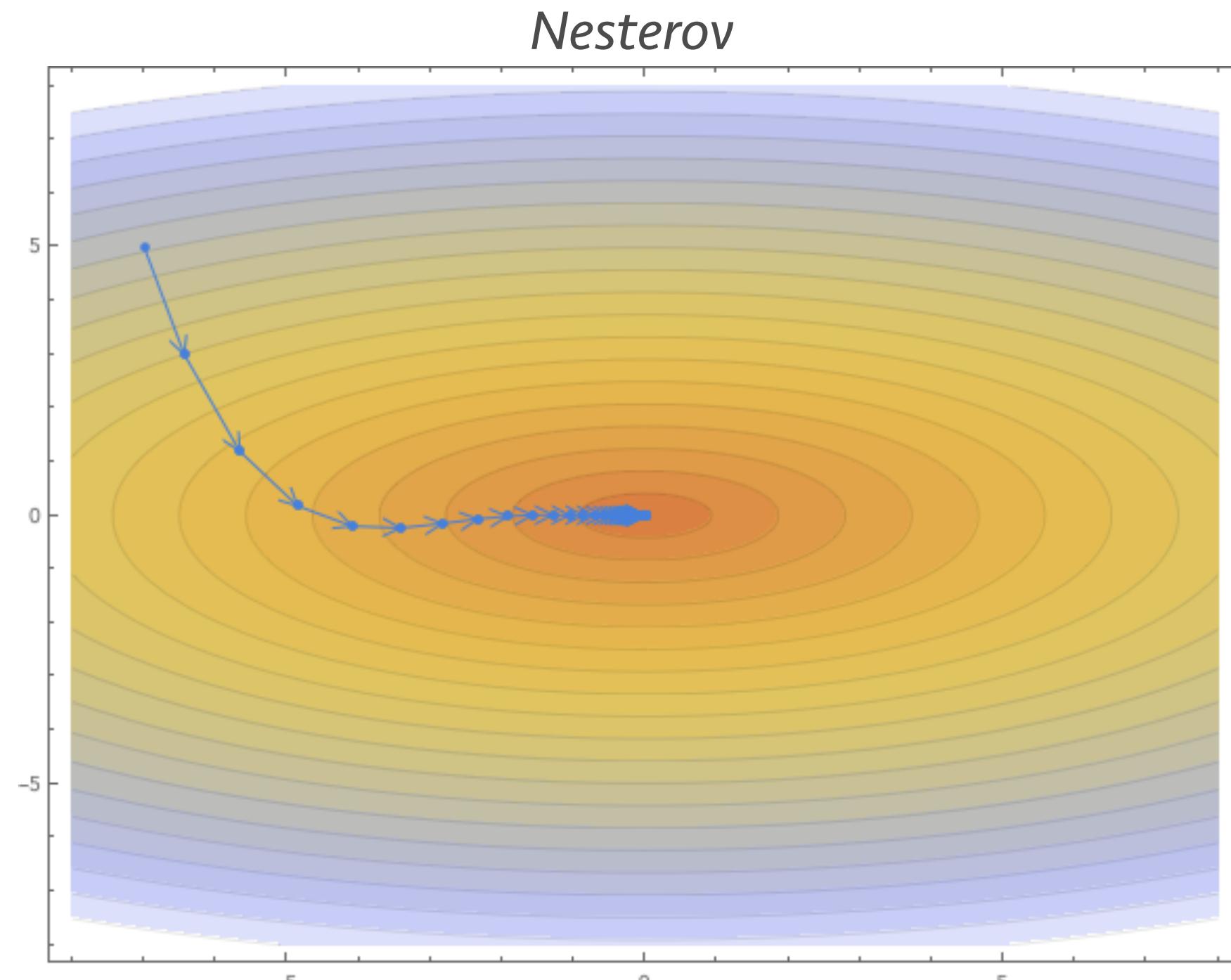
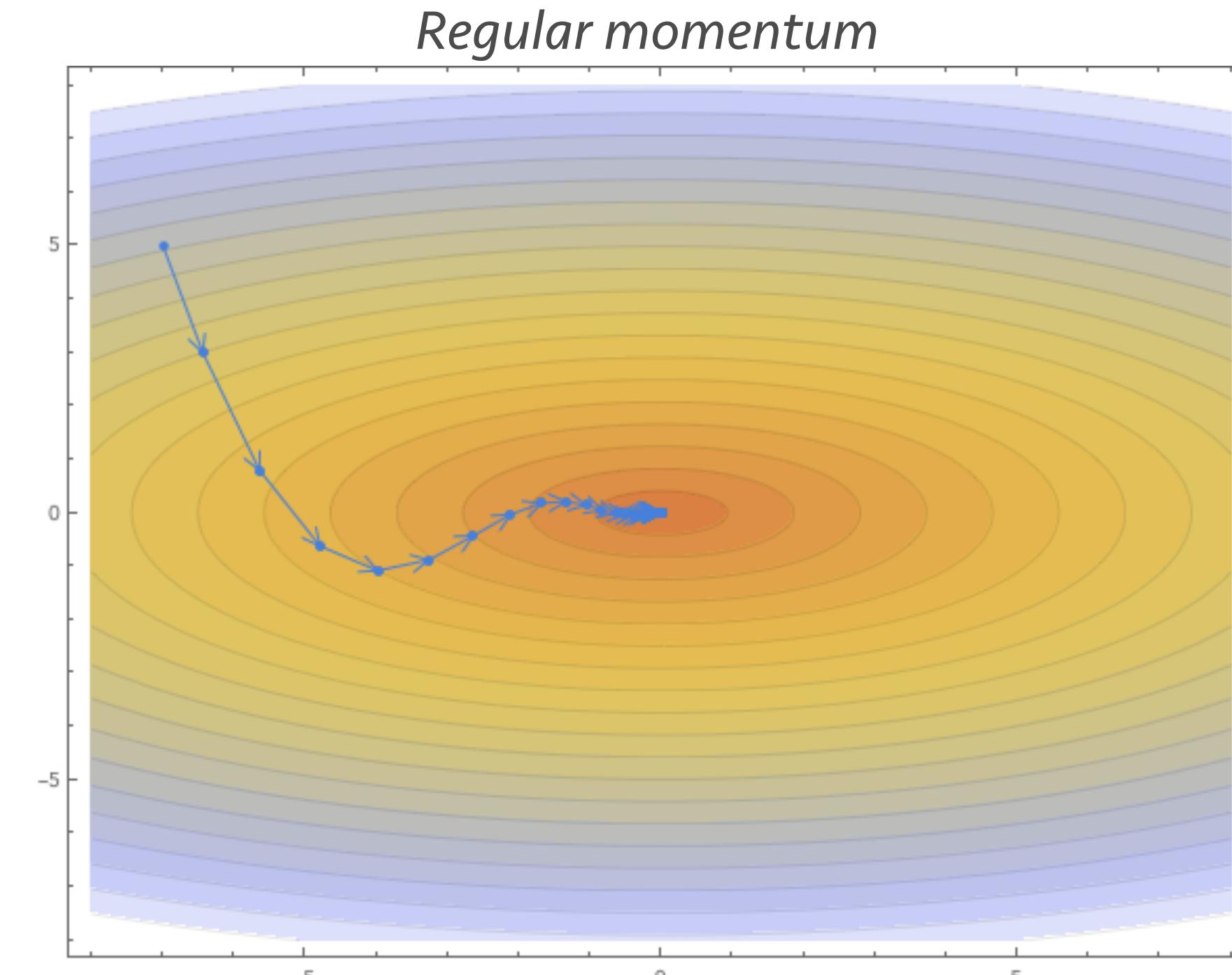
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{u}_t$$

- Usually $\mu = 0.9$ -ish



Nesterov accelerated gradient

- Momentum is not quite right
 - We use the gradient without accounting for momentum offset
 - Instead do: $\mathbf{u}_t = \eta \nabla f(\mathbf{w}_t - \mu \mathbf{u}_{t-1}) + \mu \mathbf{u}_{t-1}$



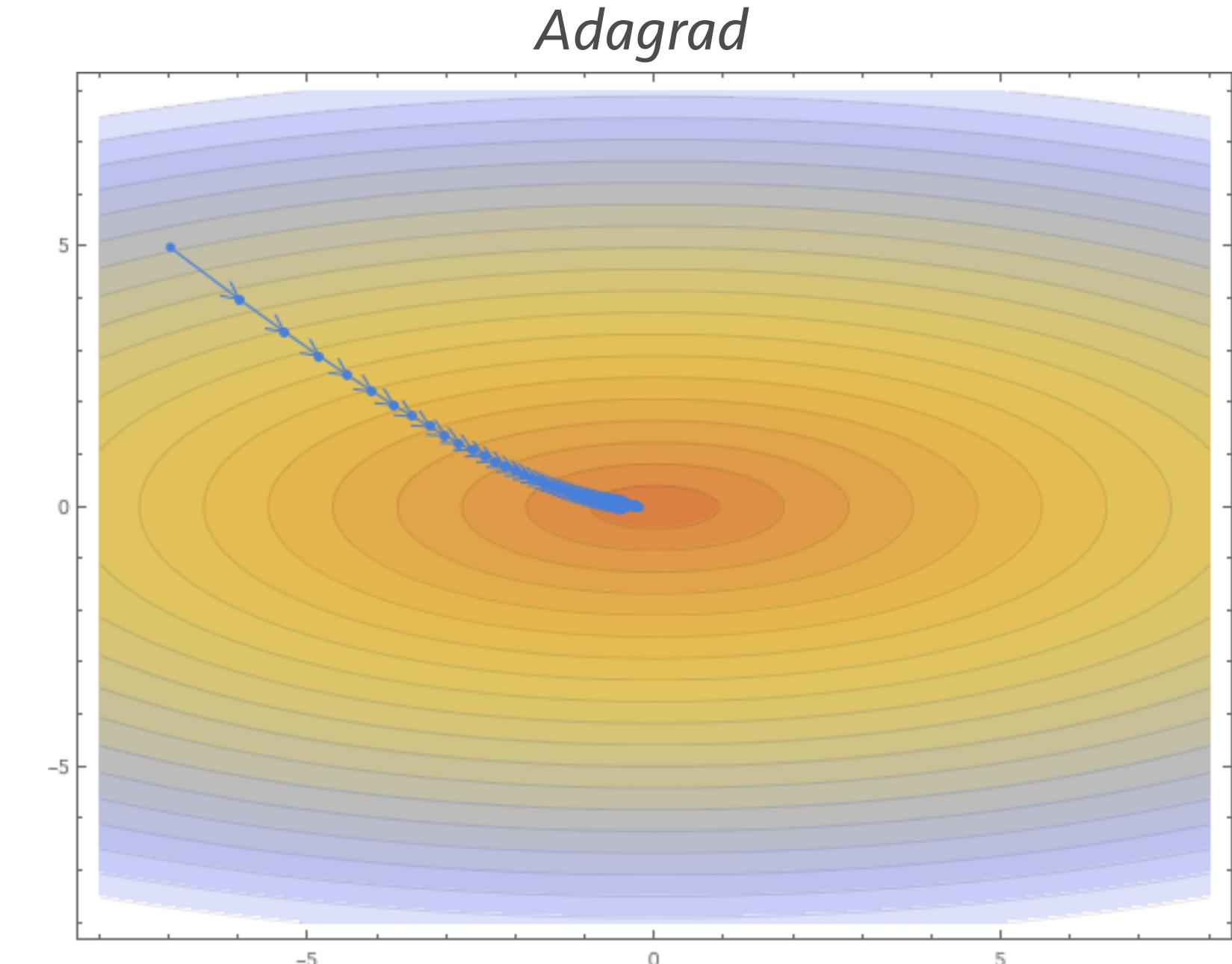
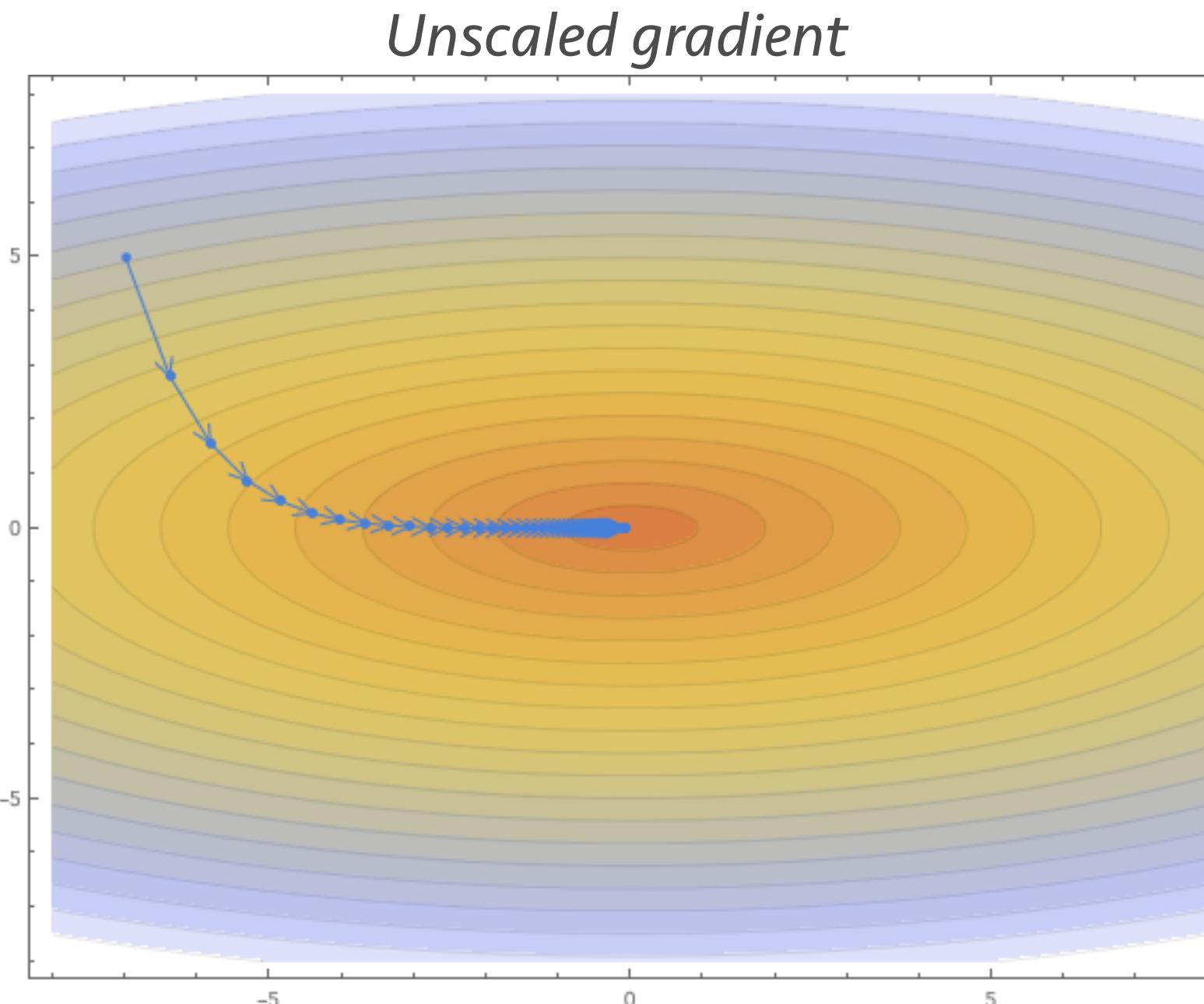
Per-parameter learning rate

- Use a different rate for each parameter's update
 - *Adagrad*: penalize consistently large updates
 - Move faster on flatter dimensions, cautiously on bumpy ones

Adagrad

$$\mathbf{g}_{t+1} = \mathbf{g}_t + \nabla f(\mathbf{w}_t)^2$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\nabla f(\mathbf{w}_t)}{\sqrt{\sum_{k=1}^{t+1} \mathbf{g}_k^2}}$$

Divide each parameter's update by accumulated amount of past updates



Embellishments

- Adagrad slows down to a halt after a while
 - Scaling becomes increasingly larger, updates tend to zero
 - *RMSprop* resolves that with an online estimate
 - *Adam* additionally smooths gradient (with a momentum)

RMSprop

$$\mathbf{g}_{t+1} = \gamma \mathbf{g}_t + (1 - \gamma) \nabla f(\mathbf{w}_t)^2$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\nabla f(\mathbf{w}_t)}{\sqrt{\mathbf{g}_{t+1}}}$$

*Same as Adagrad, but
get online estimate of \mathbf{g}*

Adam

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \nabla f(\mathbf{w}_t)$$

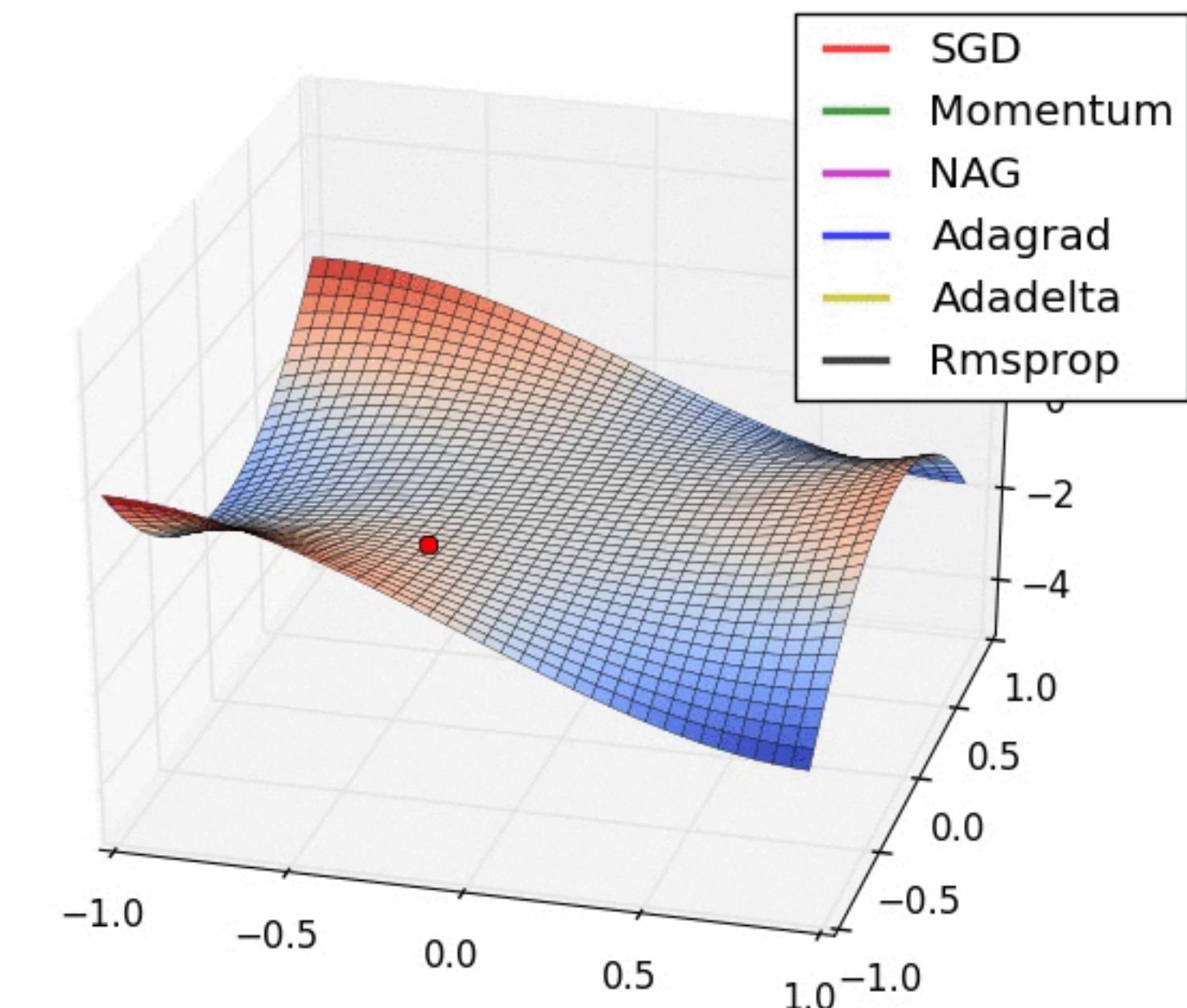
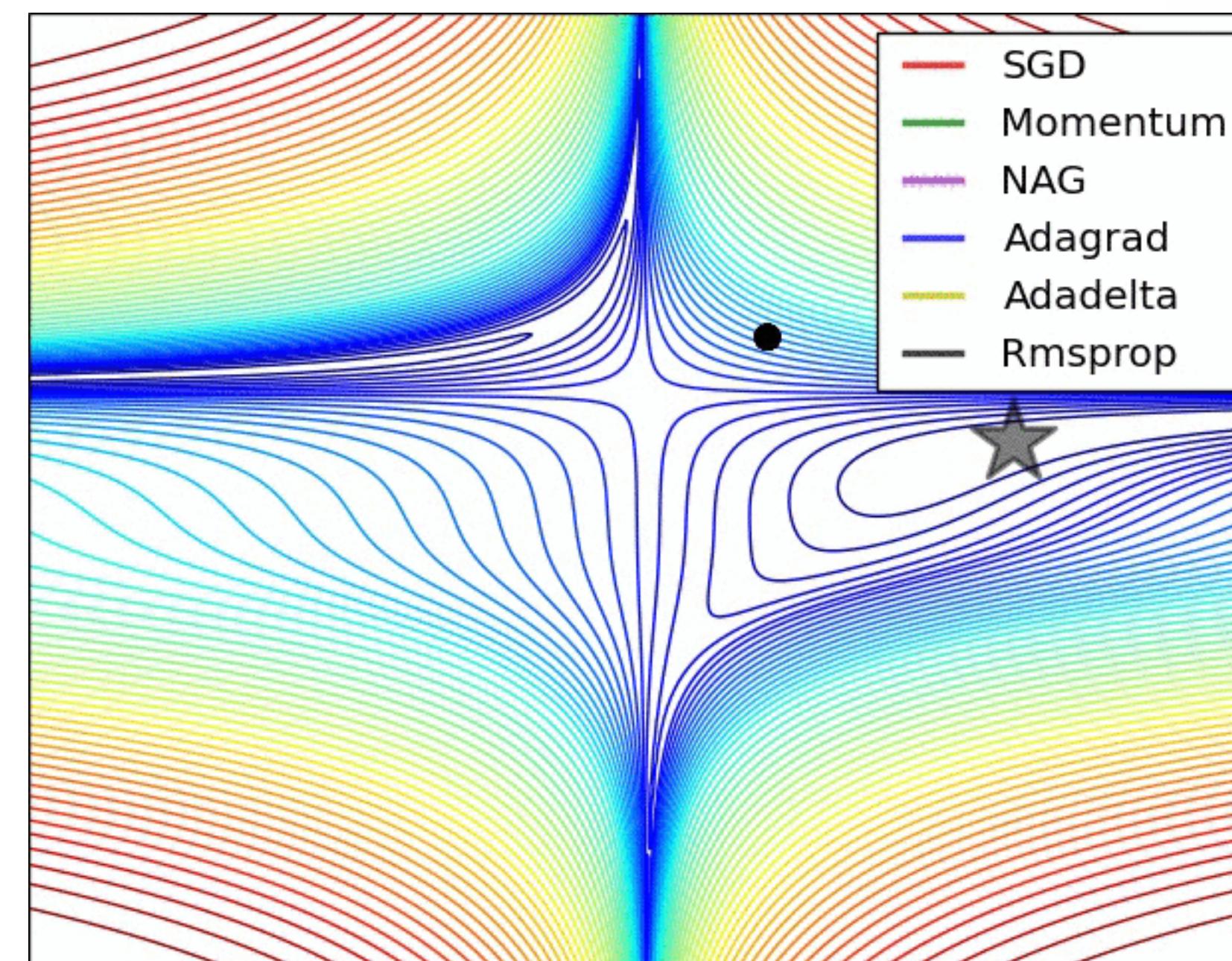
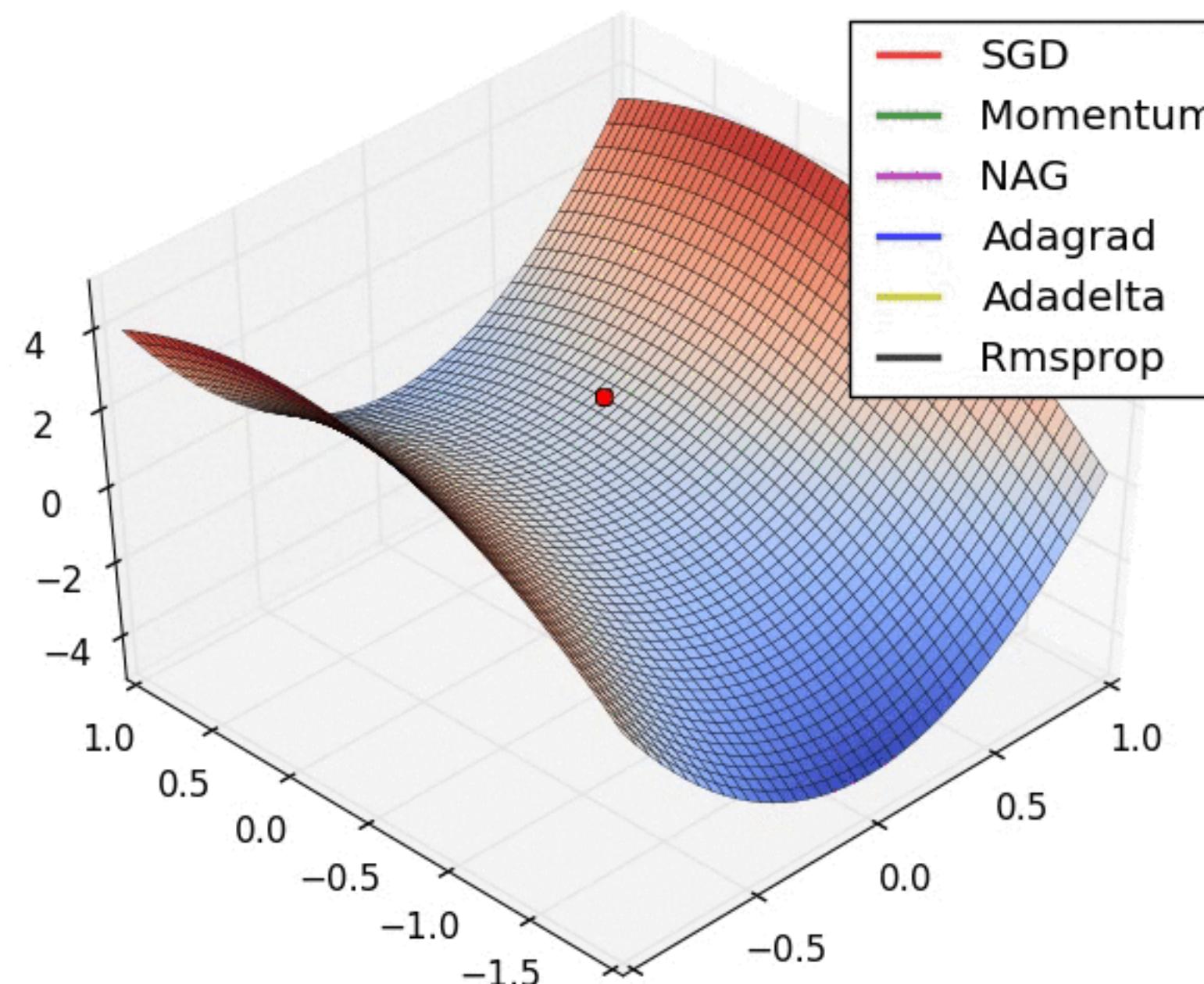
$$\mathbf{g}_{t+1} = \beta_2 \mathbf{g}_t + (1 - \beta_2) \nabla f(\mathbf{w}_t)^2$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{g}_{t+1}}}$$

*Same as RMSprop, but use
smoothed estimate of gradient*

What difference does it make?

- Alec Radford's excellent optimization animations:



<http://imgur.com/a/Hqolp>

Second-order updates

- We can also update using 2nd order methods:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla^2 f(\mathbf{w}_t)^{-1} \cdot \nabla f(\mathbf{w}_t)$$

Hessian matrix *Gradient vector*

- Problems
 - Hessian is of size # of parameters squared (i.e. a large matrix)
 - Computing its inverse is flaky and slow
- Alternatives (e.g. L-BFGS) exist, but can be inefficient
 - In real-world, gradient methods are a more practical choice

Normalization

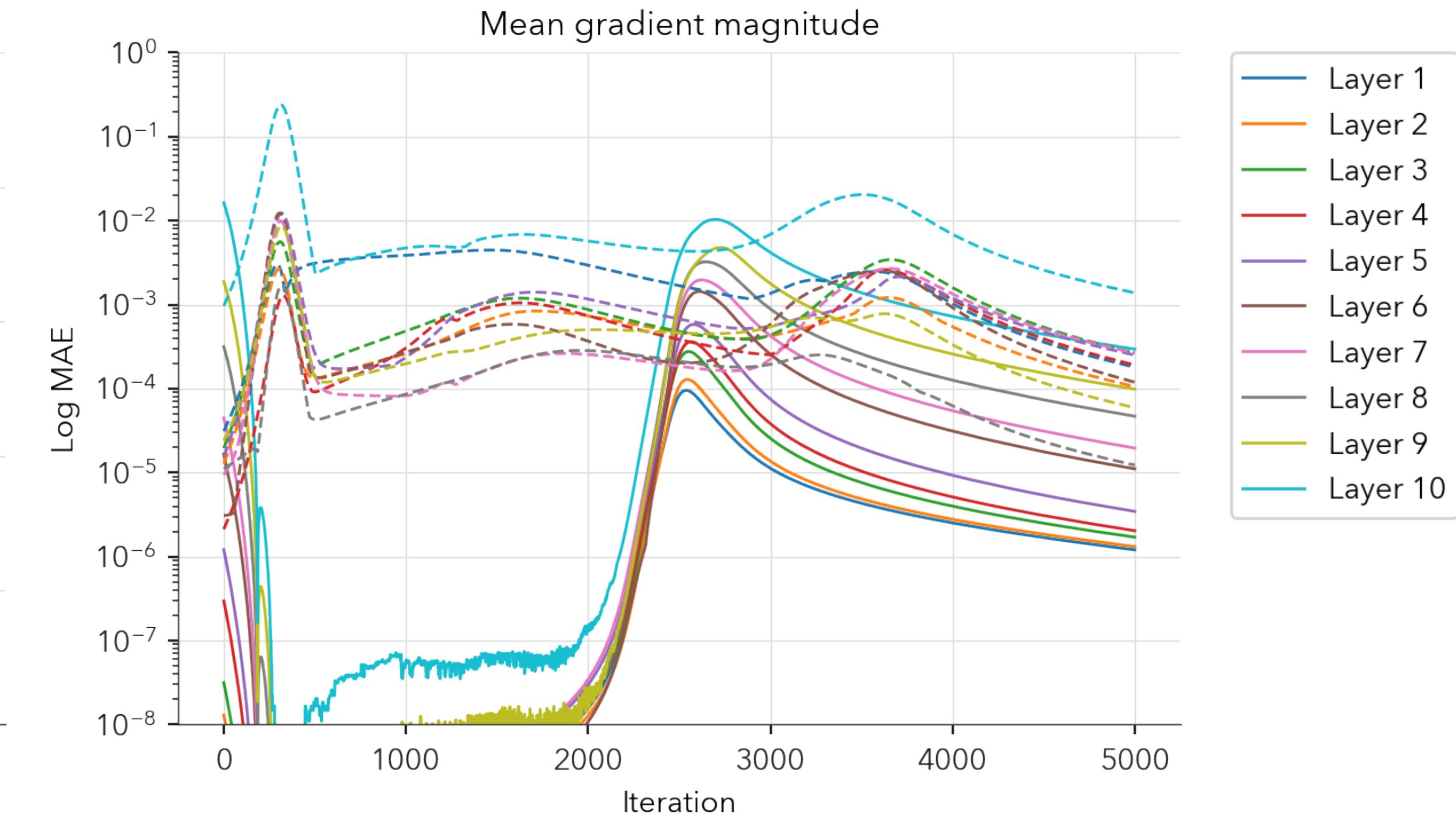
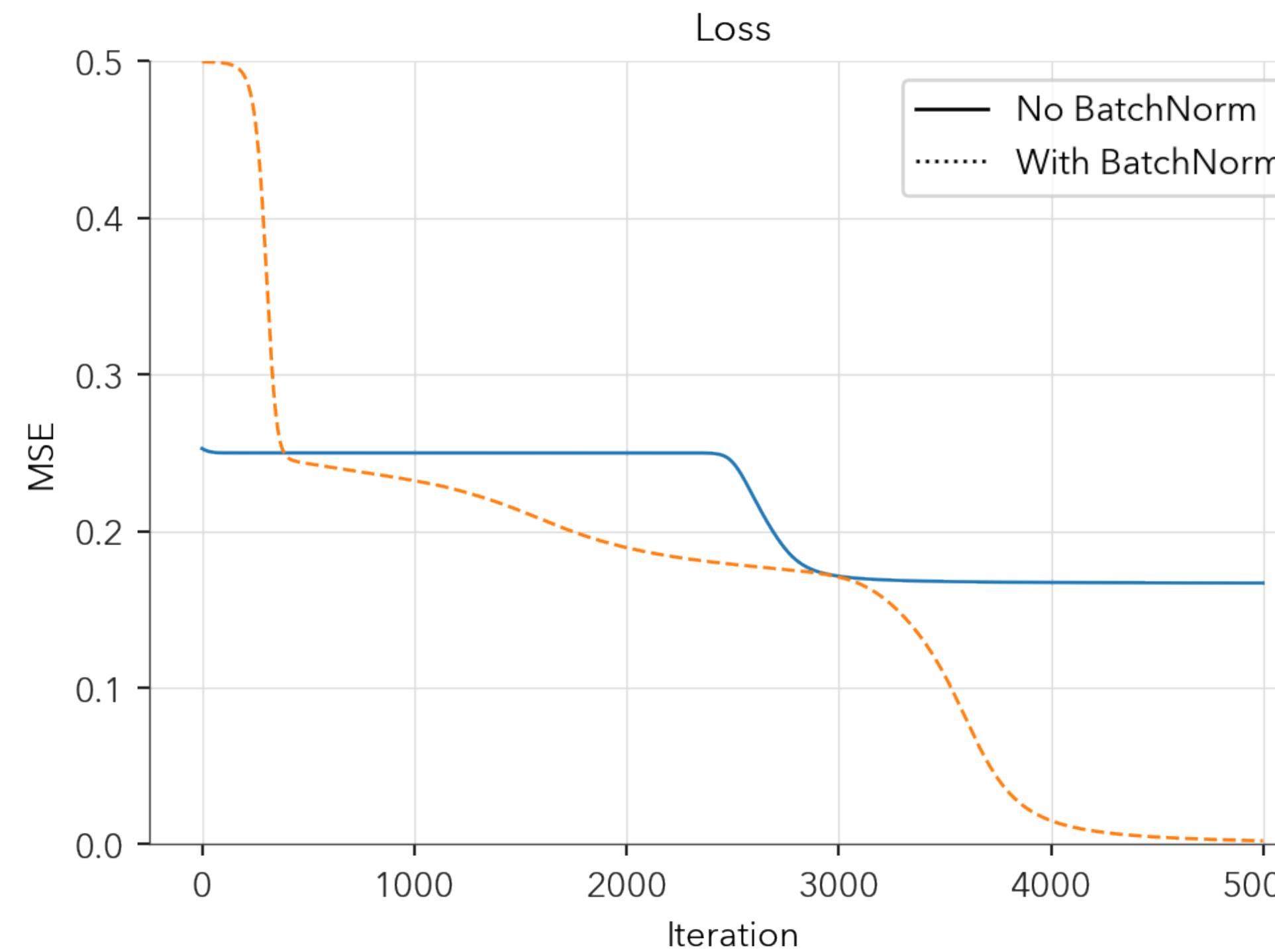
- We can also help the network by normalizing outputs
 - Scale and translate over all batches/channels/layer outputs/...:

$$\mathbf{x}_i = g(\mathbf{W}_i \cdot \mathbf{x}_{i-1}) \quad \hat{\mathbf{x}}_i = \gamma \frac{\mathbf{x}_i - \mu_{\mathbf{x}_i}}{\sigma_{\mathbf{x}_i}} + \beta$$

- Learn optimal parameters γ and β
- Estimate μ and σ online from training data
- Ensures that layer's inputs do not saturate the layer
 - Saturating a layer results in small gradients

Effect of BatchNorm in training

- Same network trained without and with BatchNorm
 - Using BatchNorm helps speed up convergence
 - Gradient magnitudes are more reasonable (in IEEE float format)

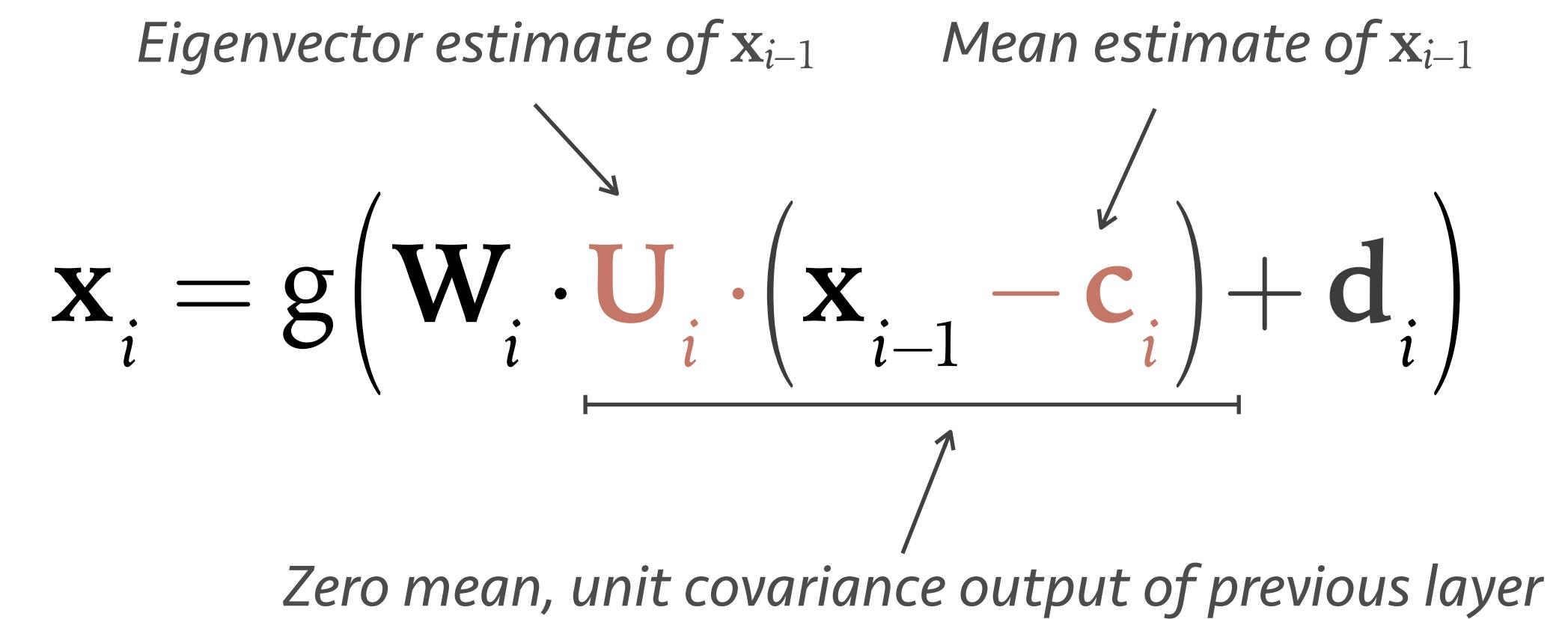


PRONG

- Instead of BatchNorm, we can *whiten* the outputs!
 - Don't scale the variance, scale the covariance (like PCA)

$$\mathbf{x}_i = g\left(\mathbf{W}_i \cdot \underbrace{\mathbf{U}_i \cdot \left(\mathbf{x}_{i-1} - \mathbf{c}_i\right)}_{\text{Zero mean, unit covariance output of previous layer}} + \mathbf{d}_i\right)$$

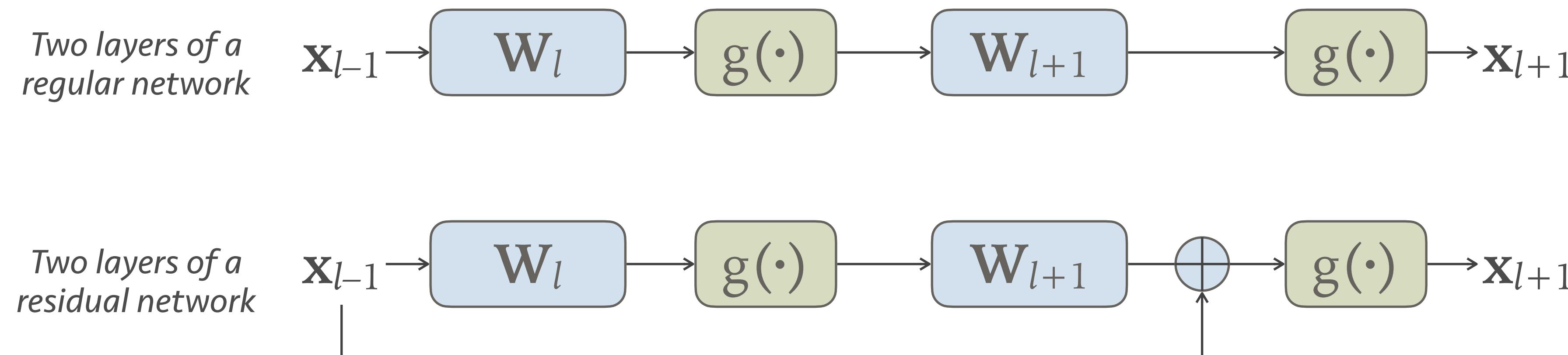
Eigenvector estimate of \mathbf{x}_{i-1} *Mean estimate of \mathbf{x}_{i-1}*



- Simplifies optimization significantly (at a cost of course!)

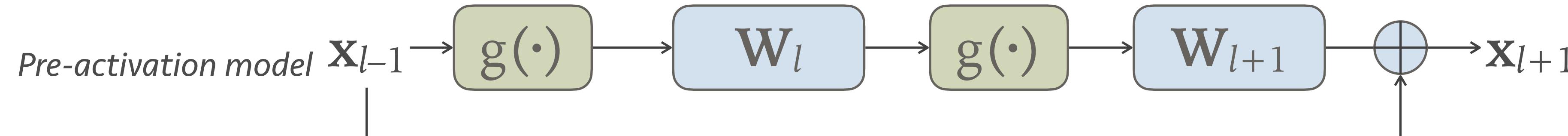
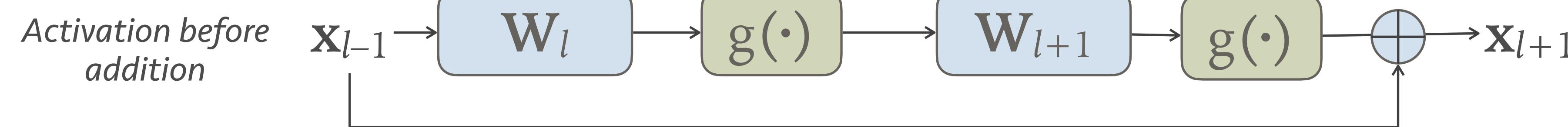
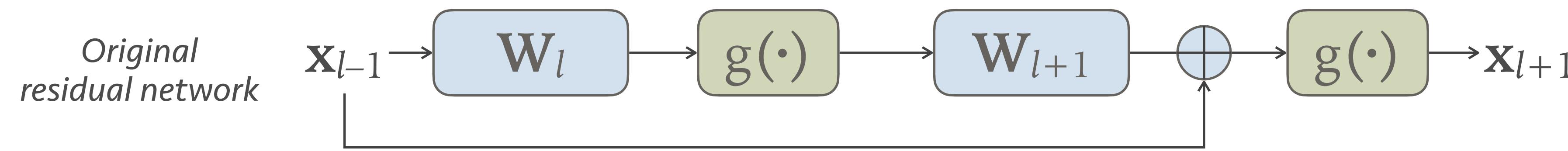
Residual nets

- Really deep networks do not train as well
 - Gradients vanish after backpropagating too many layers
- *Residual* nets bypass this problem
 - Add preceding layer's inputs to future layer's outputs
 - Effectively “shallows” network and facilitates convergence



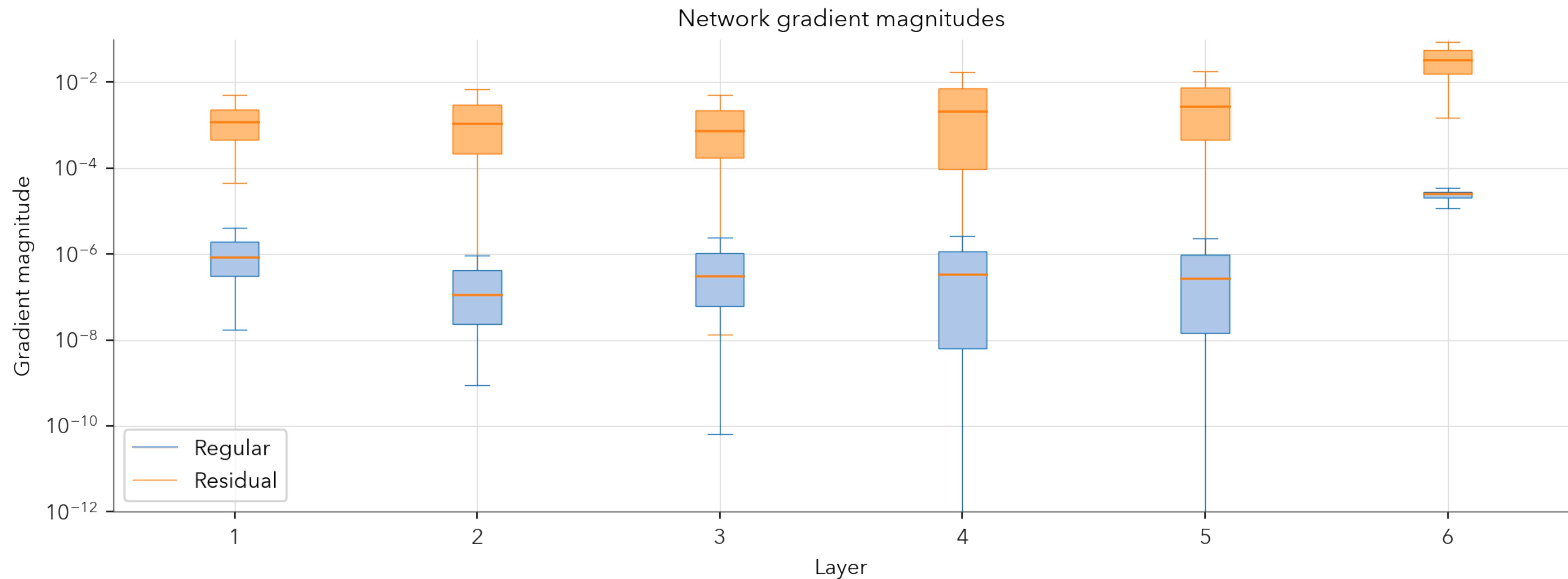
Residual net variations

- Subsequent tweaking of the original idea
 - Can also use various BatchNorms in between



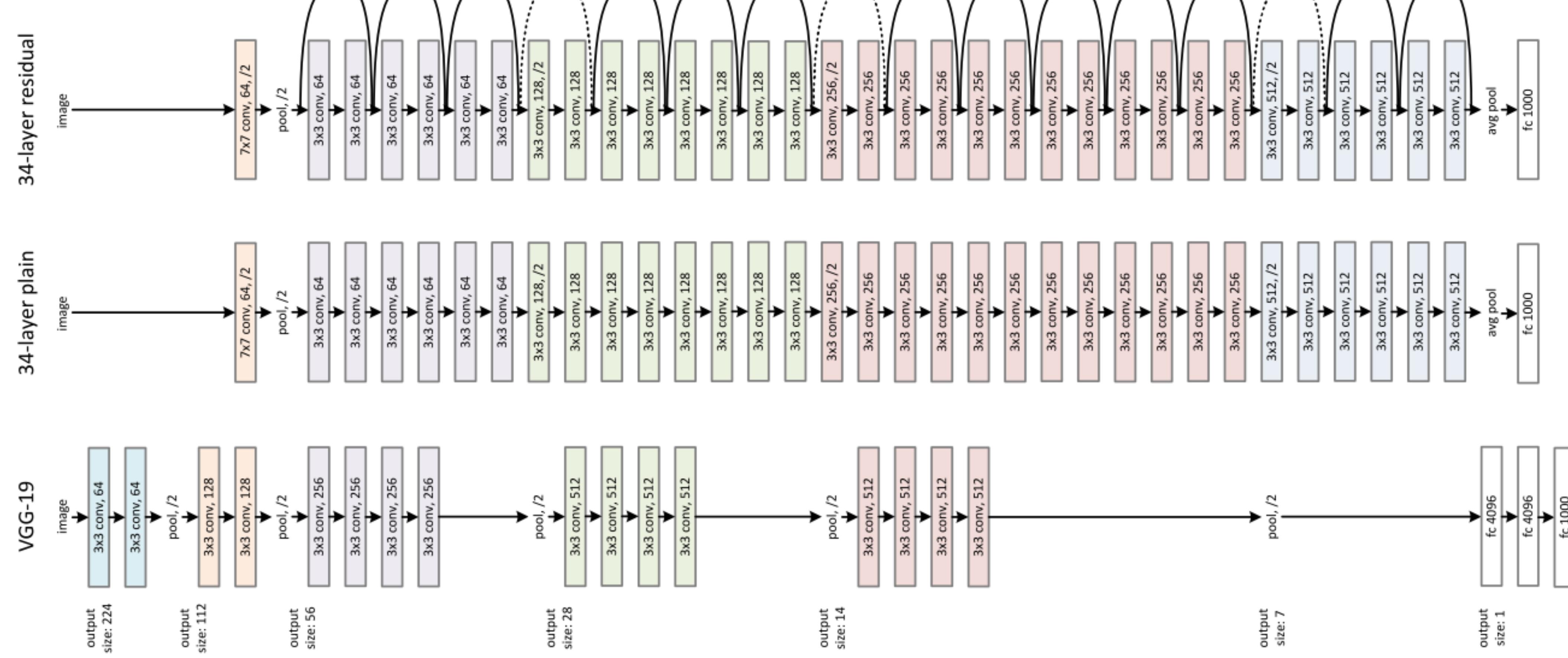
Effect of residual nets

- Results in more usable gradient values
 - Allows for the design of networks with hundreds of layers



ResNets in practice

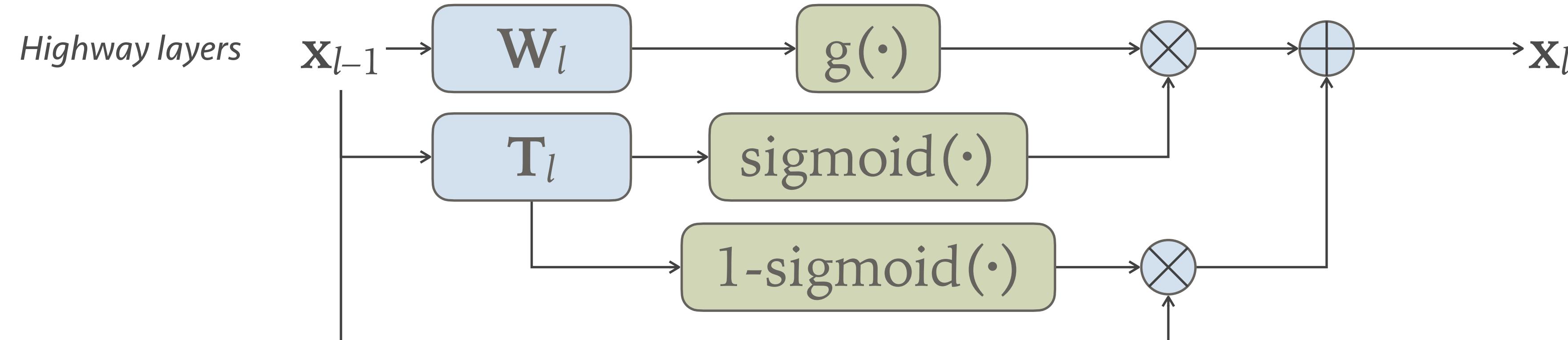
- This is the small one



Highway networks

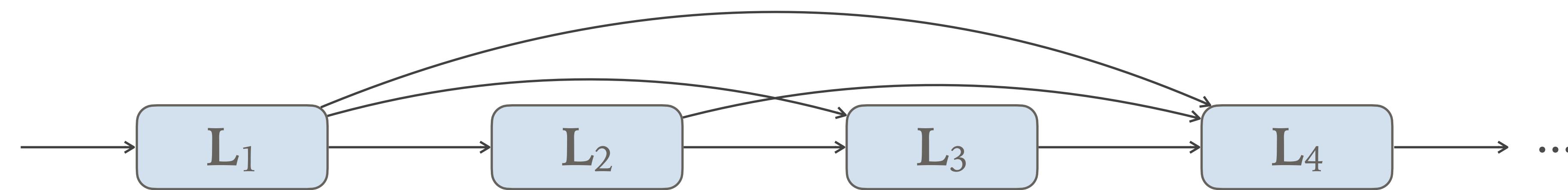
- Generalizing the residual network idea
 - Add a data-driven gate to the skip connection

$$\mathbf{x}_l = g(\mathbf{W}_l \cdot \mathbf{x}_{l-1}) \text{sigmoid}(\mathbf{T}_l \cdot \mathbf{x}_{l-1}) + \mathbf{x}_{l-1} (1 - \text{sigmoid}(\mathbf{T}_l \cdot \mathbf{x}_{l-1}))$$



DenseNets

- And why not go all out and connect all layers?
 - Unlike previous approaches, we use past outputs as extra input dimensions to the subsequent layers
 - A drawback: we need a lot more memory!



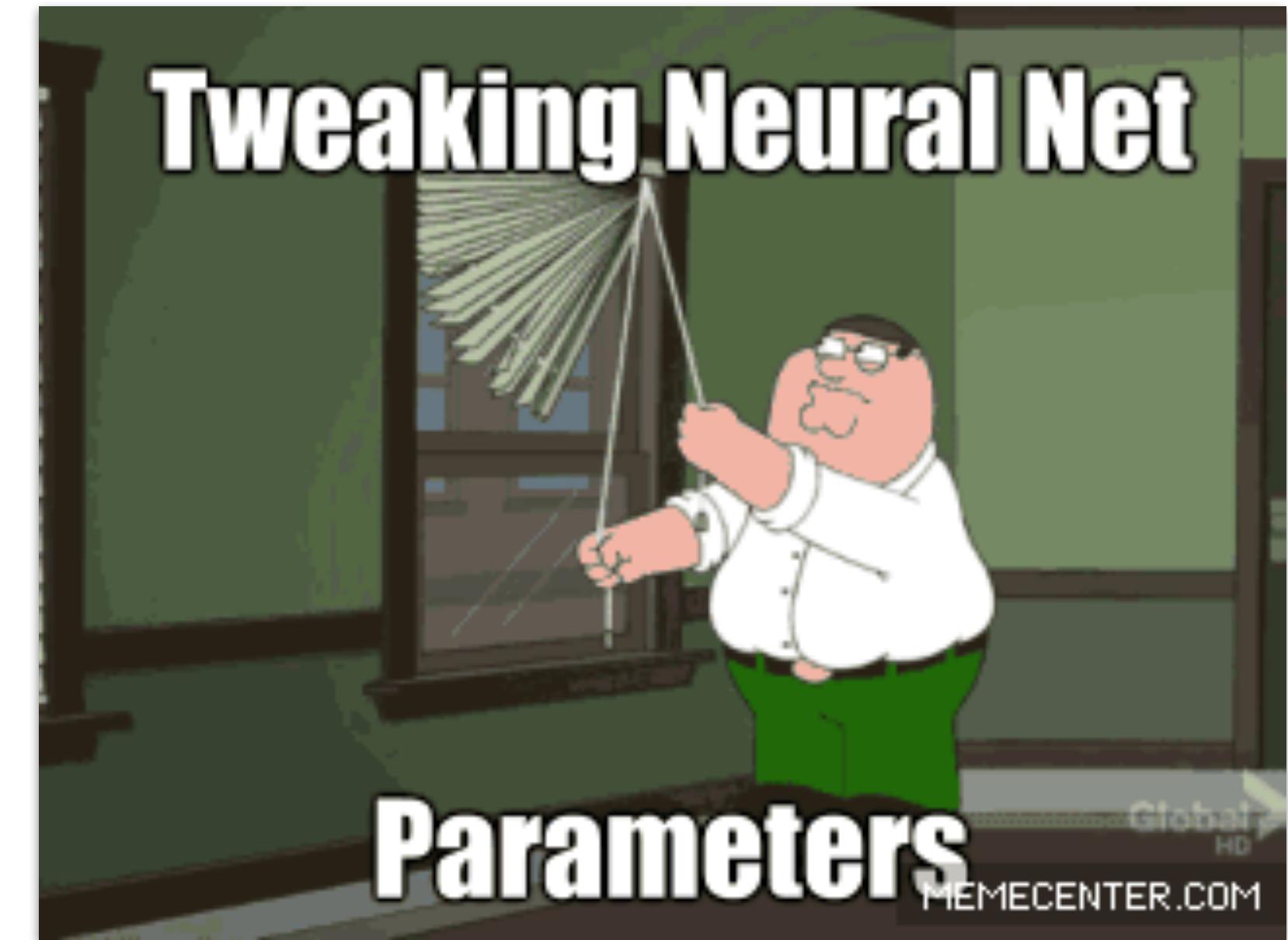
- Advantage: Gradients propagate nicely through network

Model Compression

- Is being shallow necessary?
 - Remember that three layers should be all we need
- Using a deep structure tends to facilitate training
 - Despite the introduction of many more parameters
- Train shallow networks to mimic trained deep networks
 - Train a deep network, generate a lot of outputs for random inputs and use them as training data for a shallow network

So what's the verdict?

- Good news:
 - Very powerful and flexible approaches
 - Potential biological(ish) plausibility
 - And computability implications
- Bad news:
 - Too flexible, picking right structure is very much an art
 - Cumbersome and potentially finicky training procedures



Progression to remember

- Try a linear model: $\mathbf{x}_1 = \mathbf{W} \cdot \mathbf{x}_0 + \mathbf{b}$
 - PCA, linear classifiers, etc
- Try a non-linear model: $\mathbf{x}_1 = g(\mathbf{W} \cdot \mathbf{x}_0 + \mathbf{b})$
 - Quadratic classifiers, logistic regression, non-linear models
- Try an (increasingly) deeper model: $\mathbf{x}_i = g(\mathbf{W}_i \cdot \mathbf{x}_{i-1} + \mathbf{b}_i)$
 - Try this lecture's material to get convergence
 - Then try upcoming lectures' models ...

Recap

- Stochastic shallow networks
 - Boltzmann machine
- Deep learning concepts
- Common deep architectures and tricks
 - Optimization schemes, residuals, highways,

More material

- “Deep Learning”, the book:
 - <http://www.deeplearningbook.org>
 - (no PDF but you can read online)
- A Fast Learning Algorithm for Deep Belief Nets
 - <http://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>

Next lecture

- Deep Learning on Time Series
 - Convolutional and Recurrent architectures
 - Sequence learning
 - Attention and Transformers