CPSC 340:
Machine Learning and Data Mining

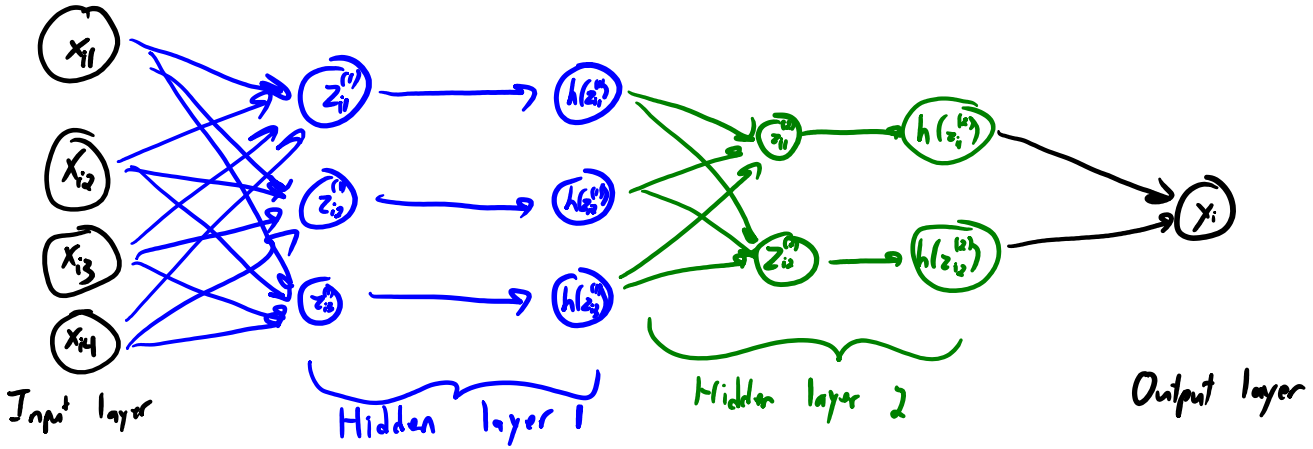
Deep Learning & Automatic Differentiation

Andreas Lehrmann and Mark Schmidt
University of British Columbia, Fall 2022

https://www.students.cs.ubc.ca/~cs-340

Last Time: Deep Learning

- Neural networks with multiple hidden layers:



$$\hat{y}_i = v^T h(W^{(2)} h(W^{(1)} x_i))$$

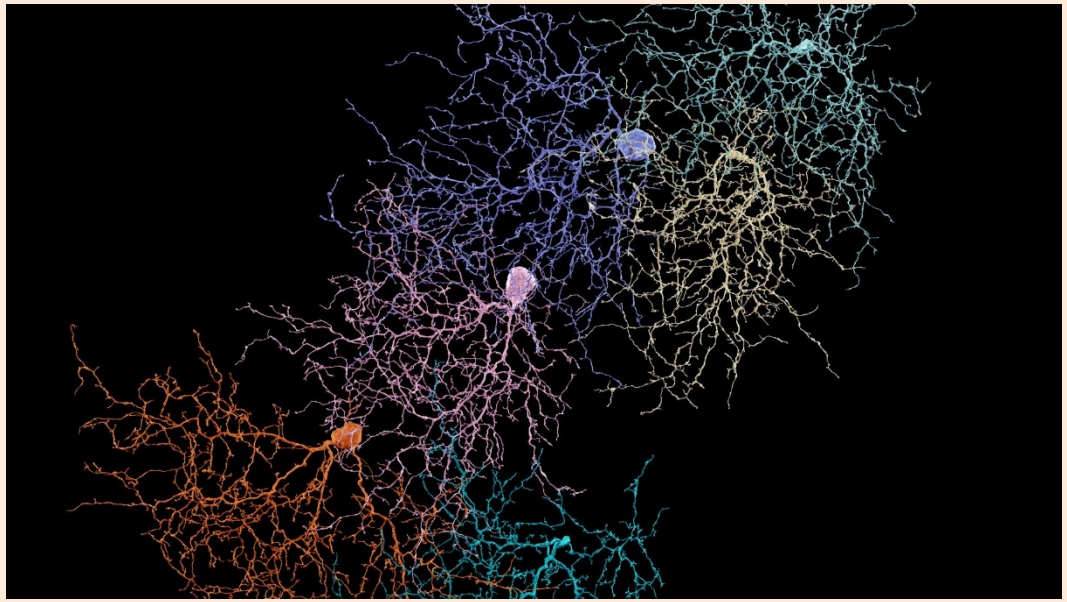
$z_i^{(1)}$

$z_i^{(2)}$

- Linear transformation followed by non-linear activation at each layer.
- Hidden layers learn latent features $h(z_i^{(l)})$; last activations serve as input to linear output layer.
  - Different objective functions lead to different tasks (e.g., binary classification, multi-class classification, regression).

Why Multiple Layers?

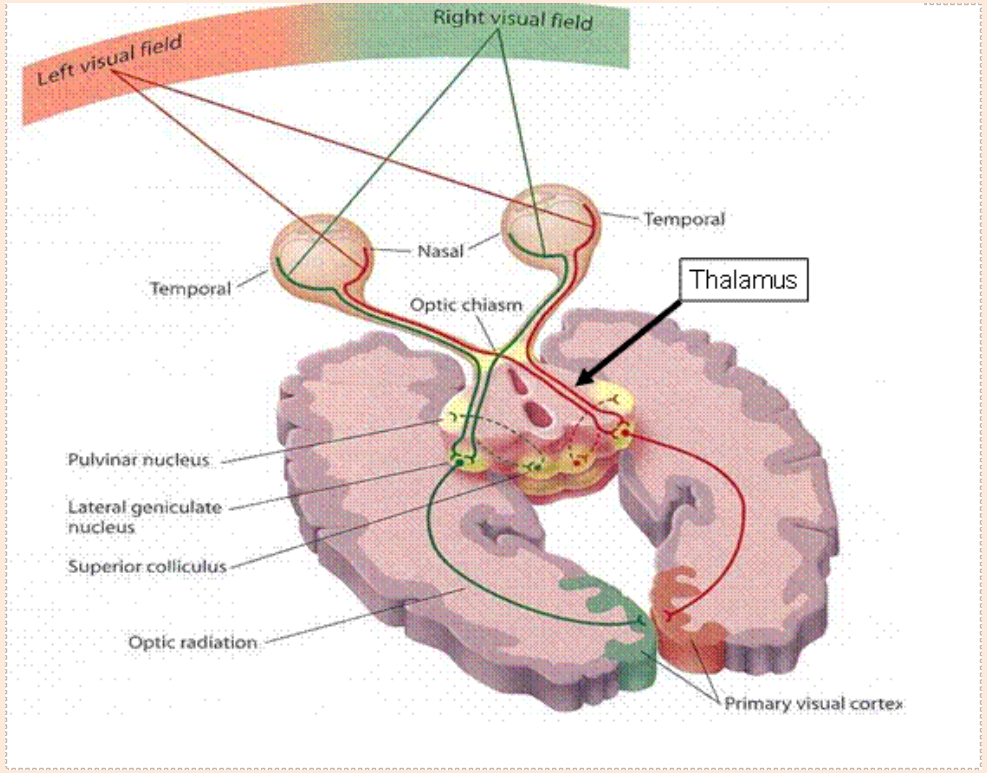
- Historically, deep learning was motivated by "connectionist" ideas:
  - Brain consists of network of highly-connected simple units.
    - Same units repeated in various places.
    - Computations are done in parallel.
    - Information is stored in distributed way.
    - Learning comes from updating of connection strengths.
    - One learning algorithm used everywhere.

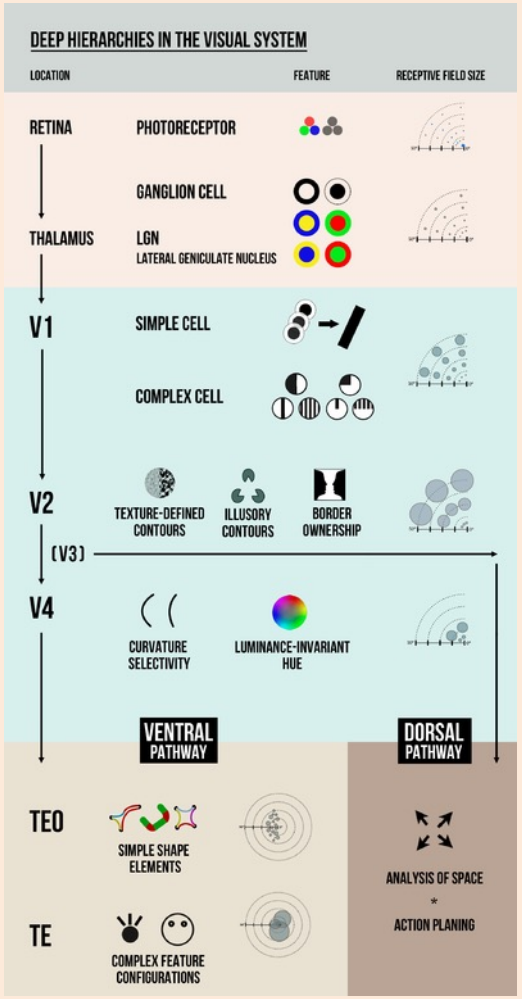


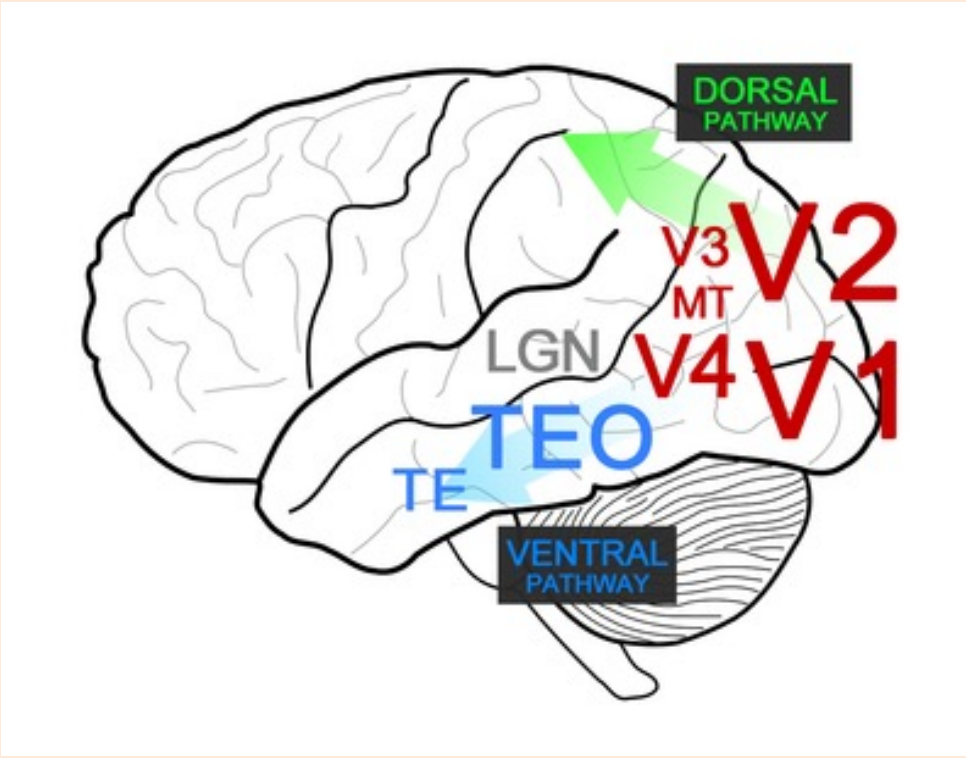
https://www.nytimes.com/2015/01/11/magazine/sebastian-seungs-quest-to-map-the-human-brain.html

Why Multiple Layers?

- And theories on the hierarchical organization of the visual system:



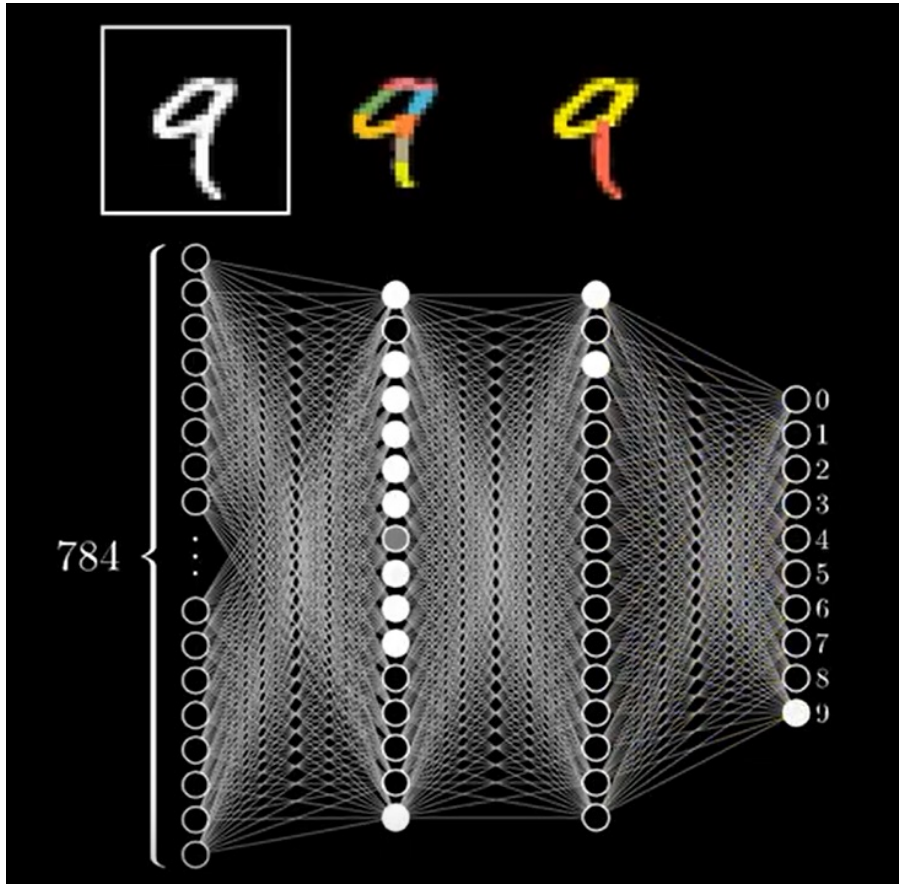




http://www.strokenetwork.org/newsletter/articles/vision.htm
https://en.wikibooks.org/wiki/Sensory_Systems/Visual_Signal_Processing

“Hierarchies of Parts” Intuition for Deep Learning

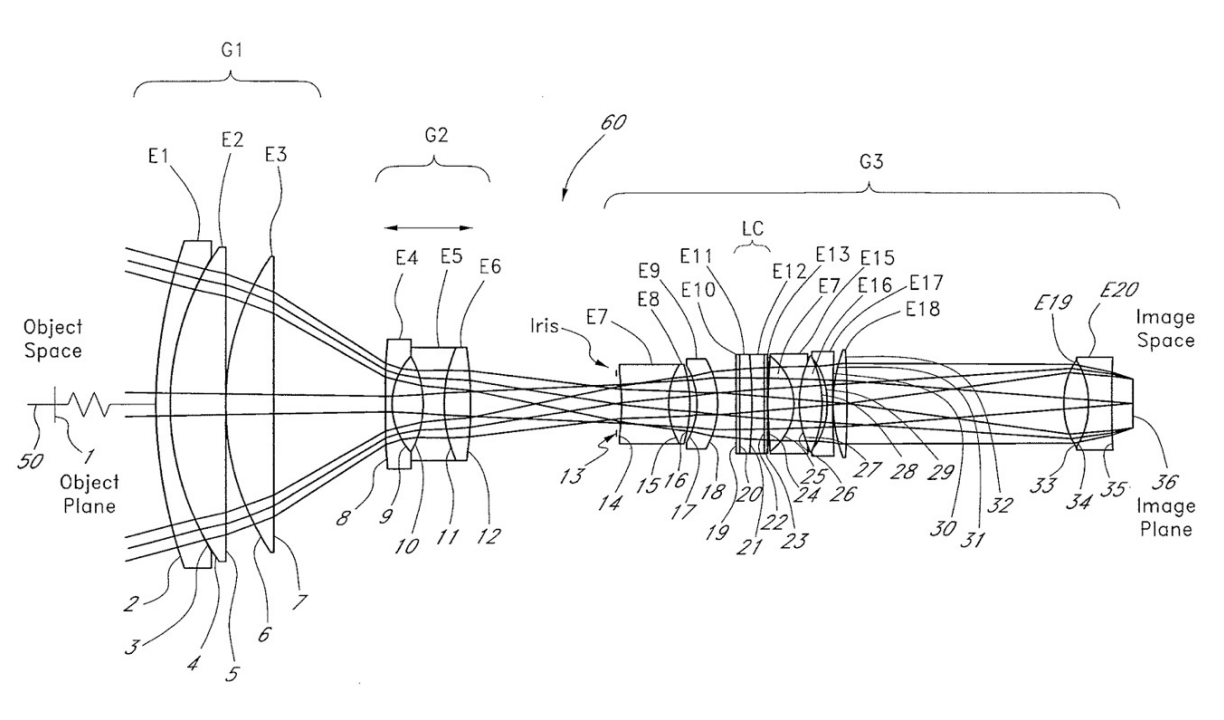
- Each “neuron” might recognize a “part” of a digit.
  - “Deeper” neurons might recognize combinations of parts.
  - Represent complex objects as combinations of simpler parts.

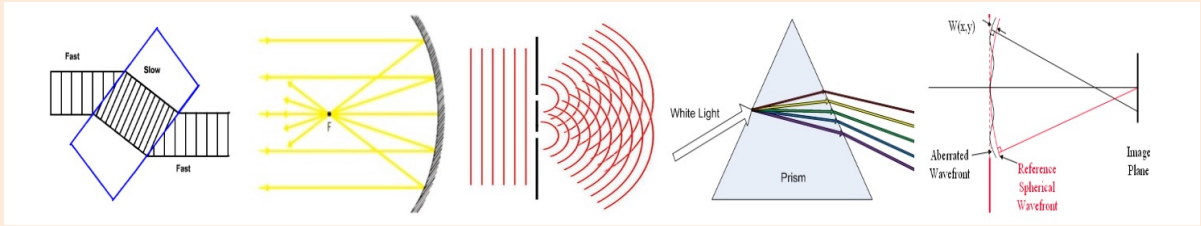


- Watch the full video here:
  - https://www.youtube.com/watch?v=aircAruvnKk

Why Multiple Layers?

- The idea of multi-layer designs appears in engineering too:
  - Deep hierarchies in camera design:





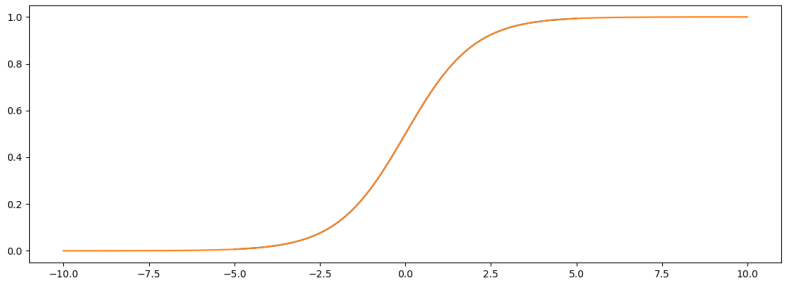
http://www.argmin.net/2018/01/25/optics/

Why Multiple Layers?

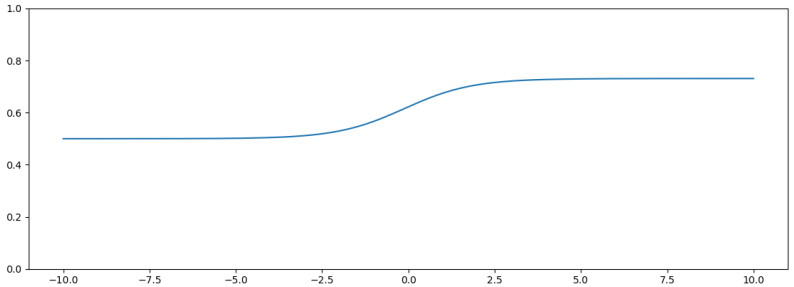
- There are also mathematical motivations for using multiple layers:
  - 1 layer gives us a universal approximator.
    - But this layer might need to be huge.
  - With deep networks:
    - Some functions can be approximated with exponentially-fewer parameters.
      - Compared to a network with 1 hidden layer.
    - So deep networks may need fewer parameters than “shallow but wide” networks.
      - And hence may need less data to train.
- Empirical motivation for using multiple layers:
  - In many domains deep networks have led to unprecedented performance.

New Issue: Vanishing Gradients

- Consider the sigmoid function:



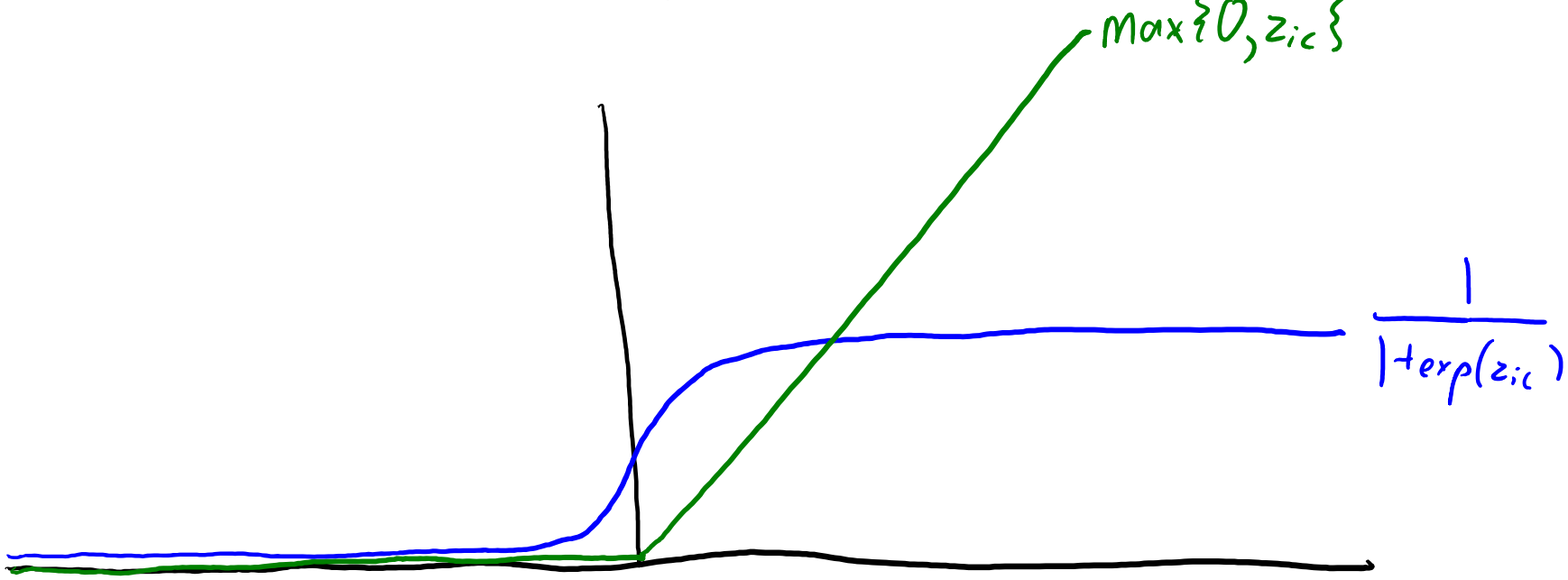
- Away from the origin, the gradient is nearly zero.
- The problem gets worse when you take the sigmoid of a sigmoid:



- 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)

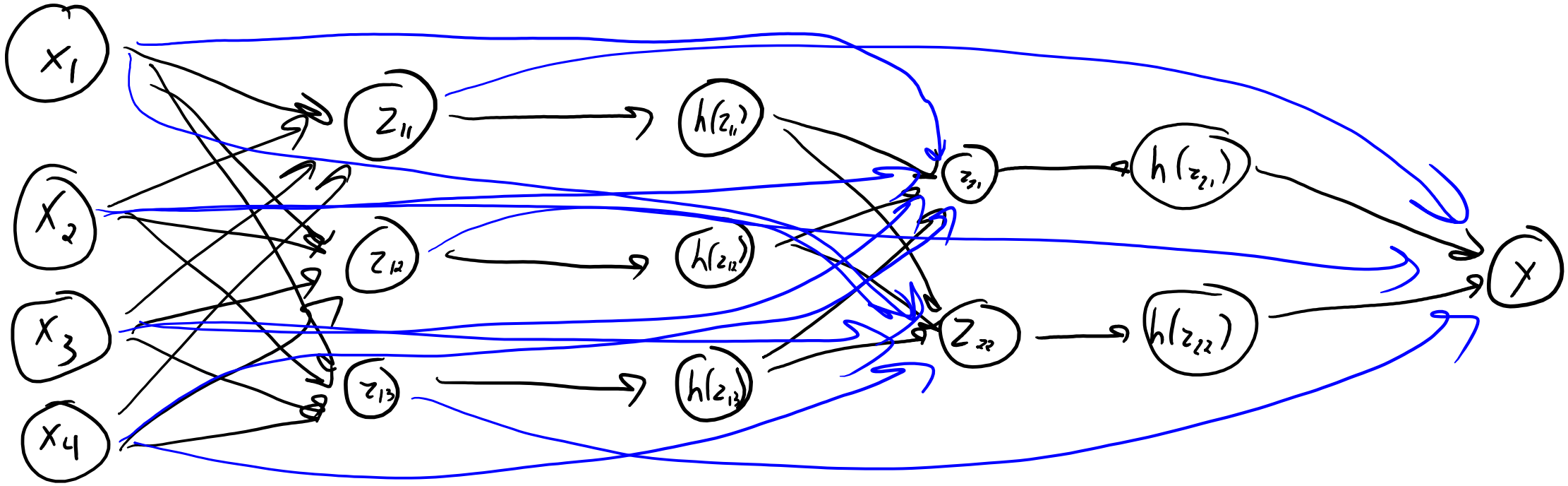
- Modern networks often replace sigmoid with perceptron loss (ReLU):



- 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'.

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.

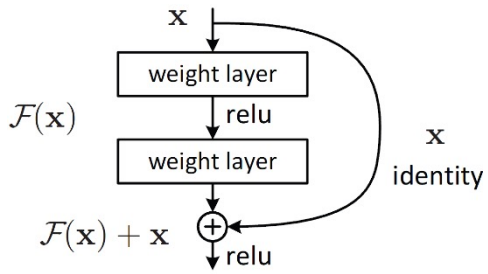
ResNet "Blocks"

- 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:

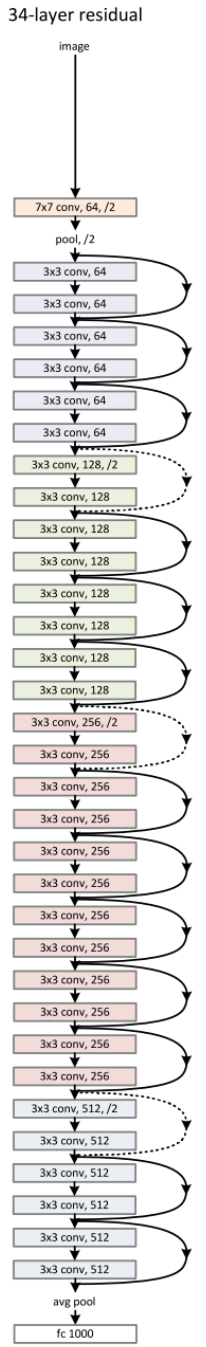
$$a^{l+2} = h(W^{l+1} h(W^l a^l))$$

↑ "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

Learning in Deep Neural Networks

- Usual training procedure is again stochastic gradient descent (SGD).
  - Deep networks are highly non-convex and notoriously difficult to tune.

- Highly non-convex, so are the problem local mimina?
  - Empirical/theoretical evidence that local minima are not the problem.
    - We think all local optima are good in typical over-parameterized cases.
  - But can be hard to get SG to close to a local minimum in reasonable time.

- We are discovering tricks that often make things easier to tune.
  - And other tricks for reducing overfitting.

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 λ 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

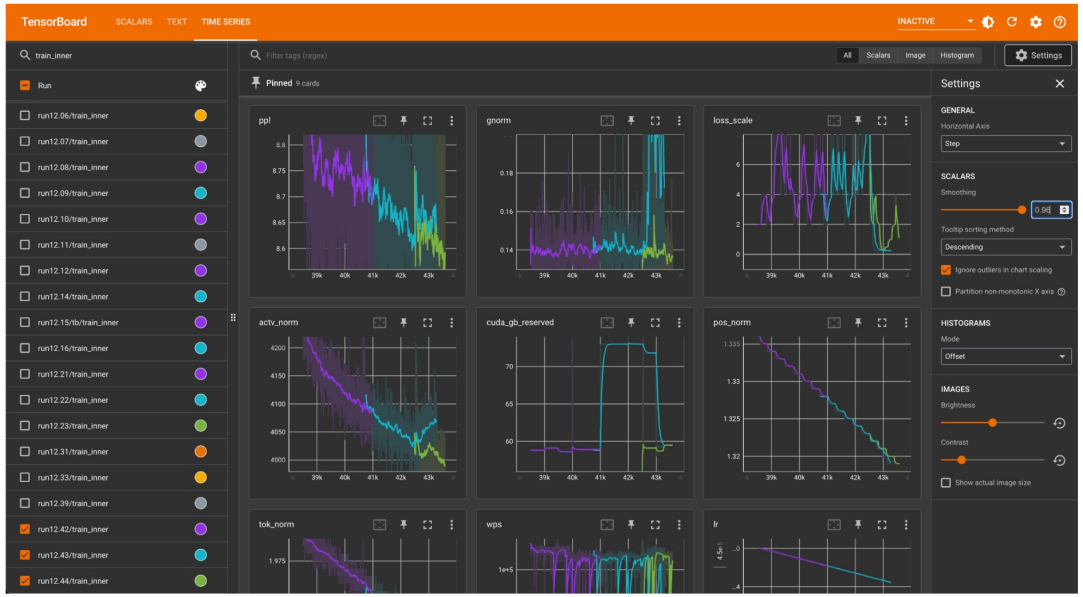
- Unfortunately, we do not understand many of these tricks very well.
  - Large portion of theory is on degenerate case of linear neural networks.
    - Or other weird cases like "1 hidden unit per layer".
  - A lot of research is performed using "grad student descent".
    - Several variations are tried, 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.

Modern Babysitting

- From 114-page babysitting log of training 175B parameter model:

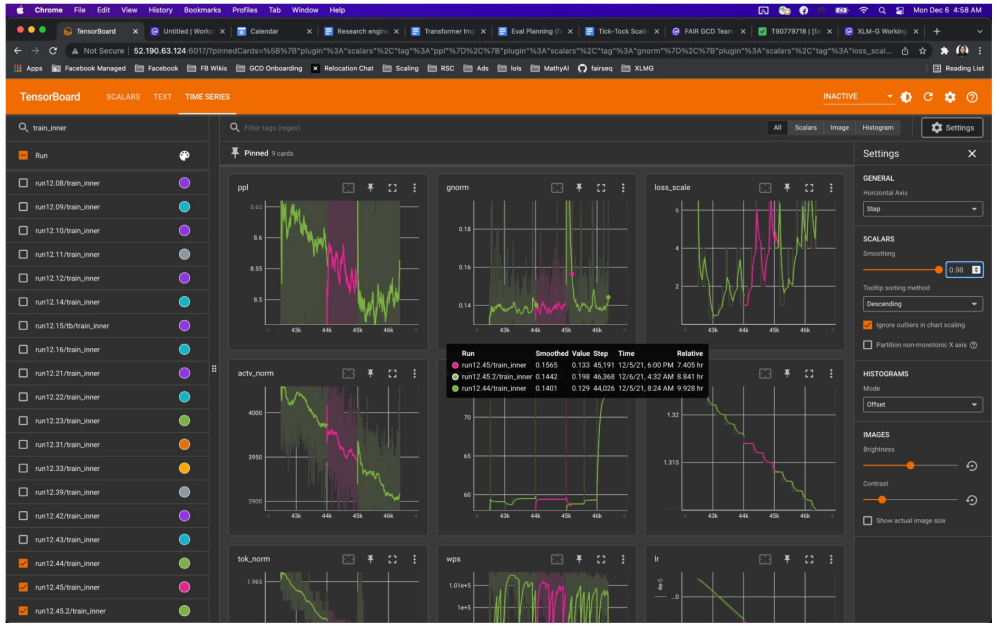
2021-12-05 05:35 PT: Checking on 12.44

[From Susan: looks like lowering LR helps keep us on track wrt ppl. Loss scale that stays below 1 for too long could be a leading indicator of instability going forward. Set smoothing to ~0.95 to see these trends.]



2021-12-06 05:00 PT: Grad norm spiking, ppl trending up

[From Susan: reading more tea leaves here, but seems like we've had a couple of grad norm spikes and our ppl is now slowly diverging. Recommending restarting with half the LR]



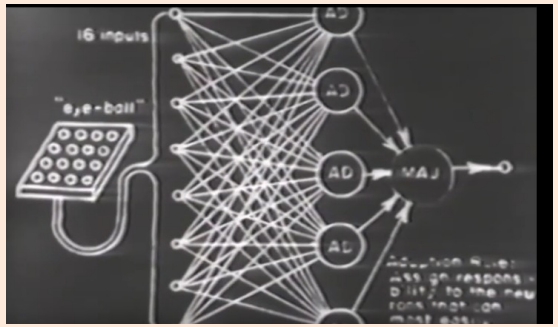
https://github.com/facebookresearch/metaseq/blob/d46662cef7a21149b55b44f47442d4aefd6c3c23/projects/OPT/chronicles/OPT175B_Logbook.pdf

Digression: Deep Learning Vocabulary

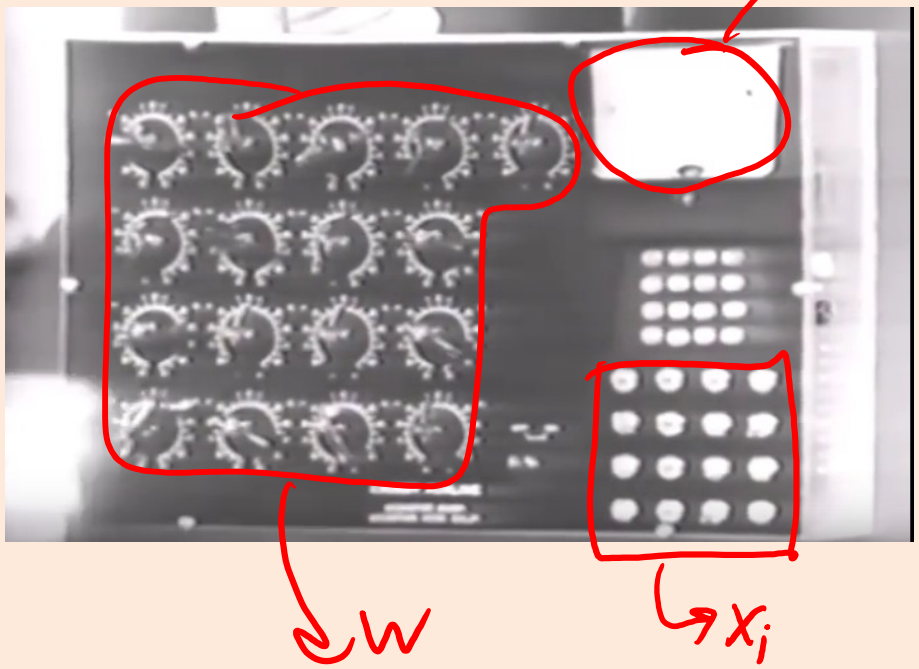
- "Deep learning": Models with many hidden layers.
  - Usually neural networks.
- "Neuron": node in the neural network graph.
  - "Visible unit": feature.
  - "Hidden unit": latent factor z_{ic} or $h(z_{ic})$.
- "Activation function": non-linear transform.
- "Activation": $h(z_i)$.
- "Backpropagation": 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 gradient": underflow/overflow during gradient calculation.

ML and Deep Learning History

- 1950 and 1960s: Initial excitement.
  - Perceptron: linear classifier and stochastic gradient (roughly).
  - "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." New York Times (1958).
    - https://www.youtube.com/watch?v=IEFRtz68m-8
  - Object recognition assigned to students as a summer project



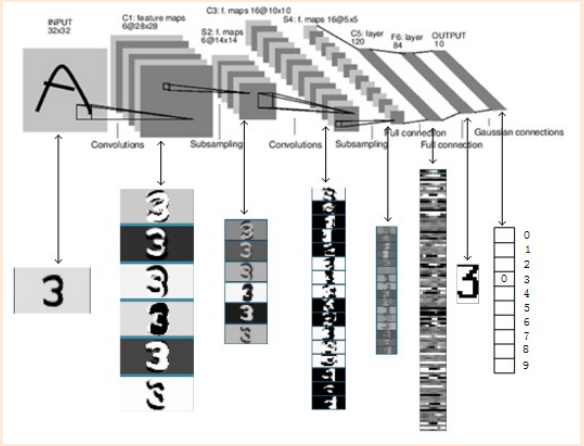
- Then drop in popularity:
  - Quickly realized limitations of linear models.

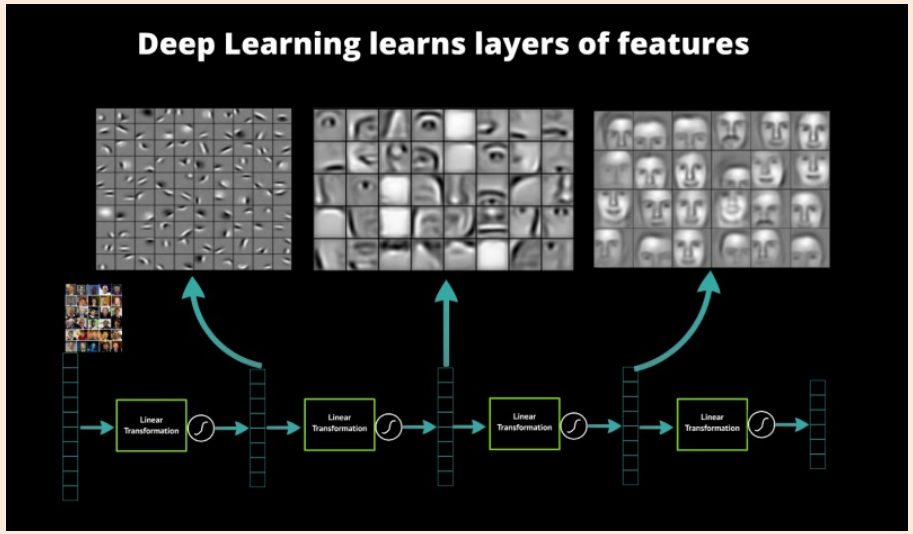


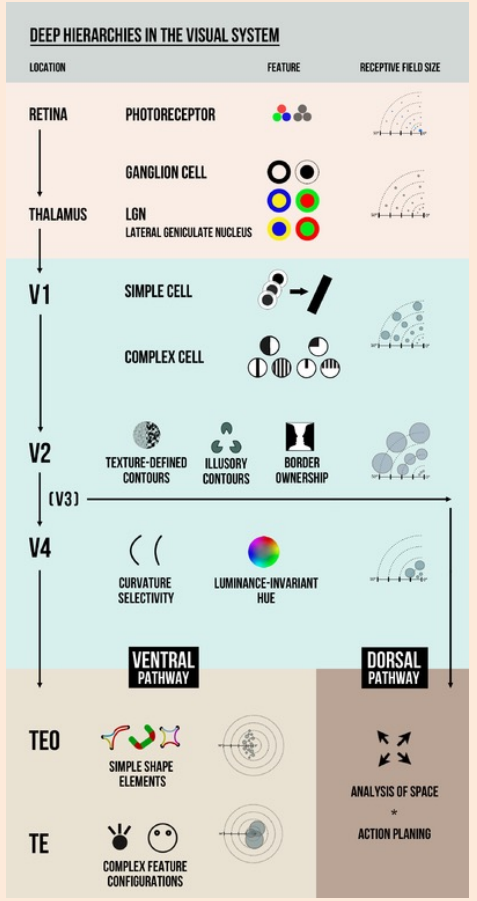
https://mitpress.mit.edu/books/perceptrons/

ML and Deep Learning History

- 1970 and 1980s: Connectionism (brain-inspired ML)
  - Want "connected networks of simple units".
    - Use parallel computation and distributed representations.
  - Adding hidden layers z_i increases expressive power.
    - With 1 layer and enough sigmoid units, a universal approximator.
  - Success in optical character recognition.







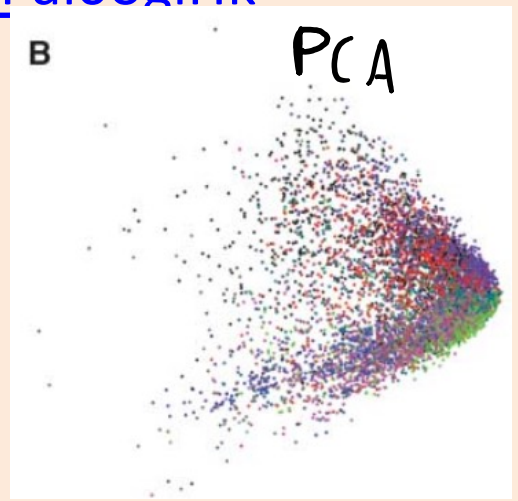
https://en.wikibooks.org/wiki/Sensory_Systems/Visual_Signal_Processing
http://www.datarobot.com/blog/a-primer-on-deep-learning/
http://blog.csdn.net/strint/article/details/44163869

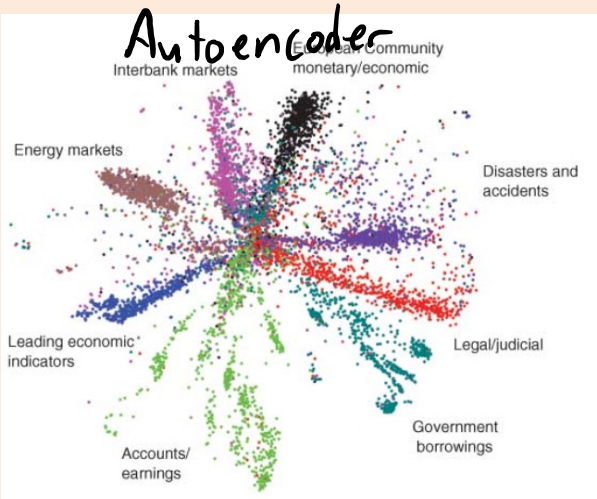
ML and Deep Learning History

- 1990s and early-2000s: drop in popularity.
  - It proved really difficult to get multi-layer models working robustly.
  - We obtained similar performance with simpler models:
    - Rise in popularity of logistic regression and SVMs with regularization and kernels.
  - Lots of internet successes (spam filtering, web search, recommendation).
  - ML moved closer to other fields like numerical optimization and statistics.

ML and Deep Learning History

- Late 2000s: push to revive connectionism as "deep learning".
  - Canadian Institute For Advanced Research (CIFAR) NCAP program:
    - "Neural Computation and Adaptive Perception".
    - Led by Geoff Hinton, Yann LeCun, and Yoshua Bengio ("Canadian mafia").
  - Unsupervised successes: "deep belief networks" and "autoencoders".
    - Could be used to initialize deep neural networks.
    - https://www.youtube.com/watch?v=KuPai0ogiHk

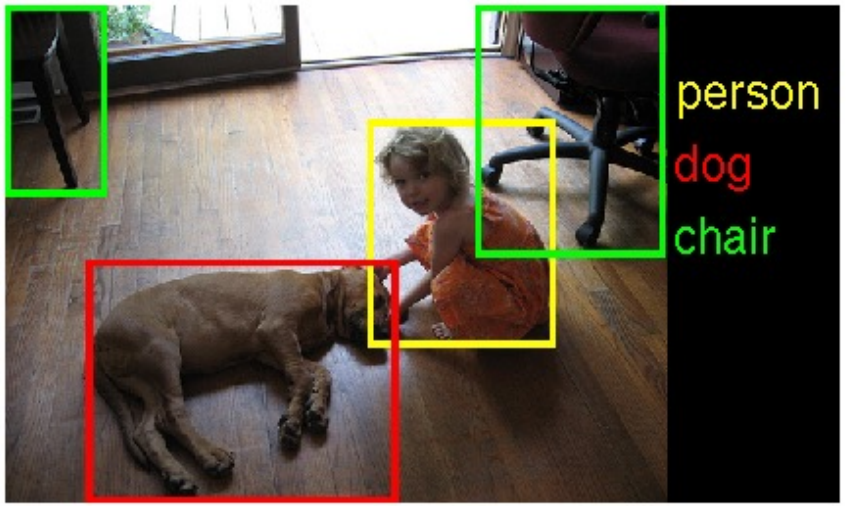




https://www.cs.toronto.edu/~hinton/science.pdf

2010s: DEEP LEARNING!!!

- Bigger datasets, bigger models, parallel computing (GPUs/clusters).
  - And some tweaks to the models from the 1980s.
- Huge improvements in automatic speech recognition (2009).
  - All phones now have deep learning.
- Huge improvements in computer vision (2012).
  - Changed computer vision field almost instantly.
  - This is now finding its way into products.



http://www.image-net.org/challenges/LSVRC/2014/

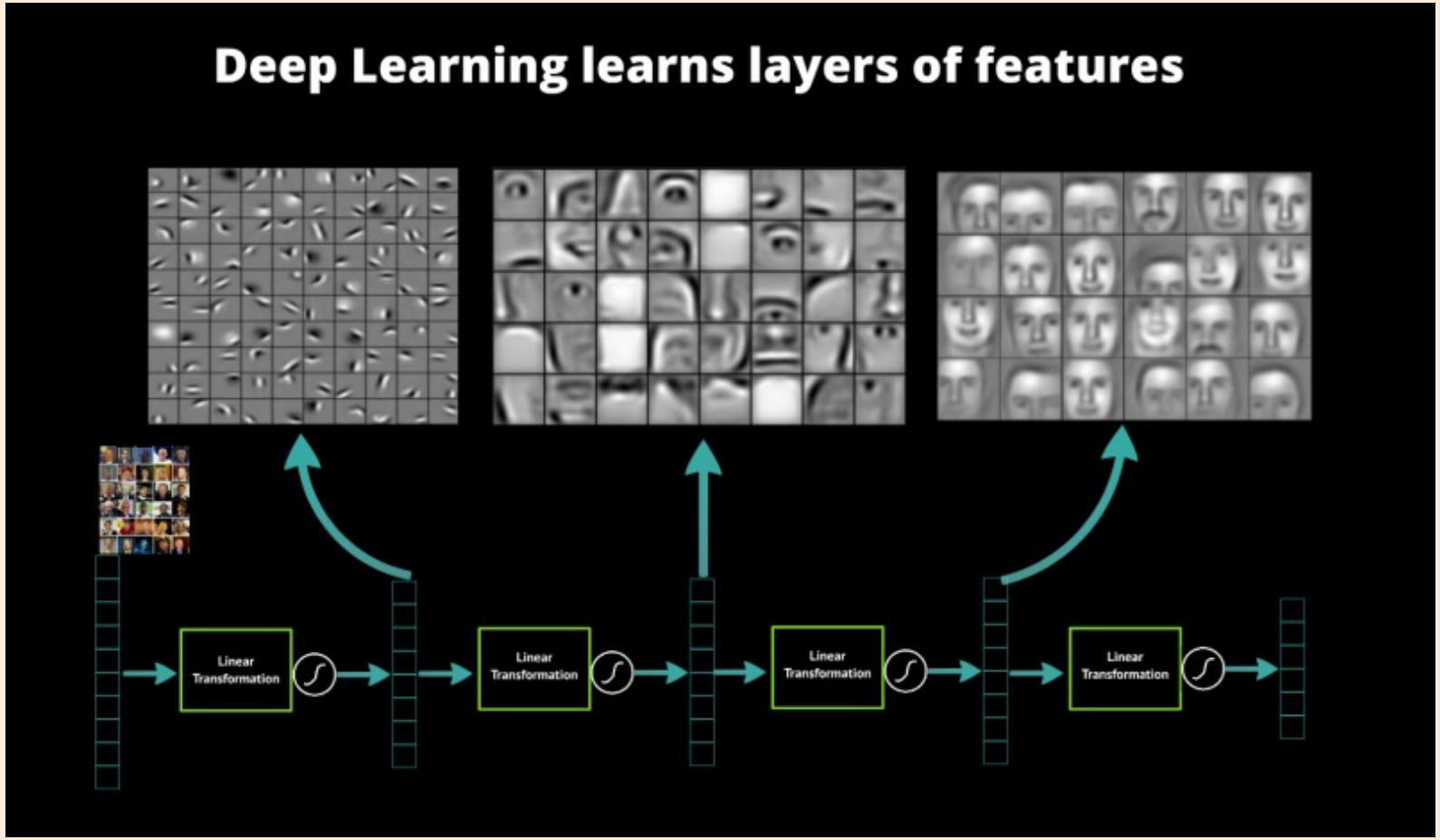
2010s: DEEP LEARNING!!!

- Media hype:
  - "How many computers to identify a cat? 16,000"
    New York Times (2012).
  - "Why Facebook is teaching its machines to think like humans"
    Wired (2013).
  - "What is 'deep learning' and why should businesses care?"
    Forbes (2013).
  - "Computer eyesight gets a lot more accurate"
    New York Times (2014).

- 2015: huge improvement in language understanding.

Cool Picture Motivation for Deep Learning

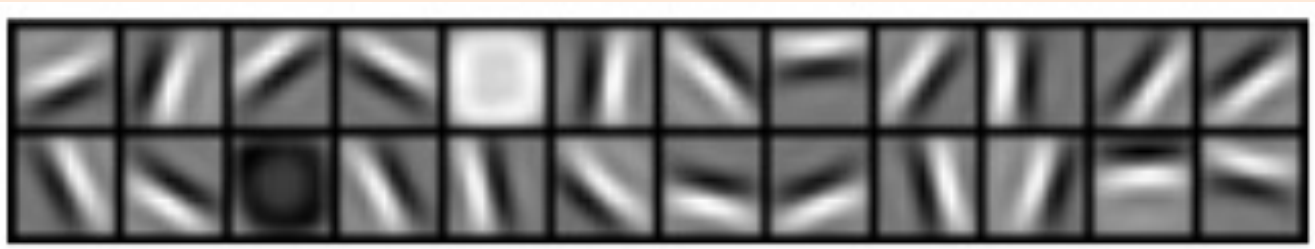
- Faces might be composed of different “parts”:



http://www.datarobot.com/blog/a-primer-on-deep-learning/

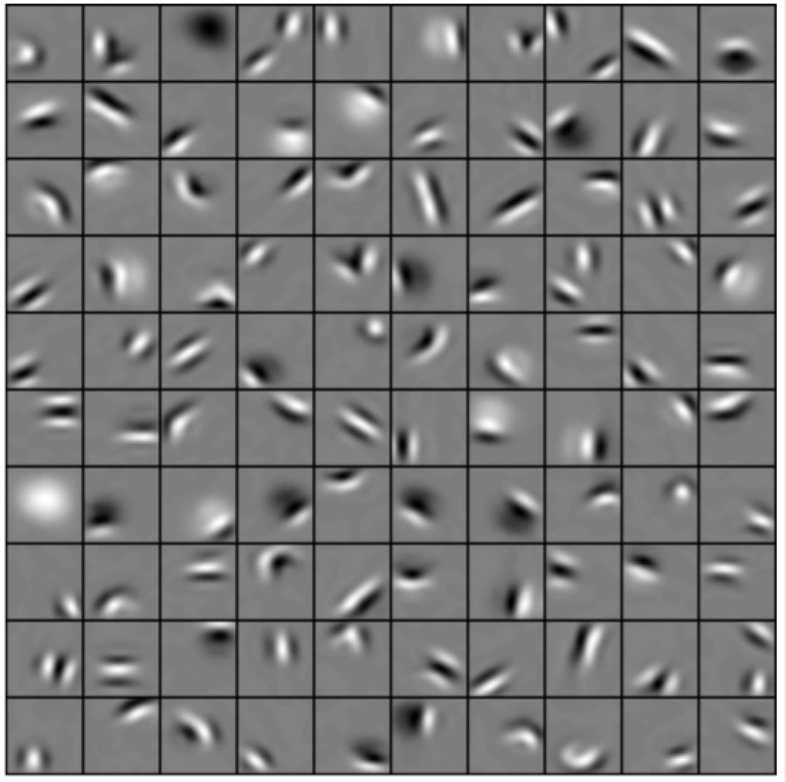
Cool Picture Motivation for Deep Learning

- First layer of z_i trained on 10 by 10 image patches:



"Gabor filters"

- Attempt to visualize second layer:
  - Corners, angles, surface boundaries?

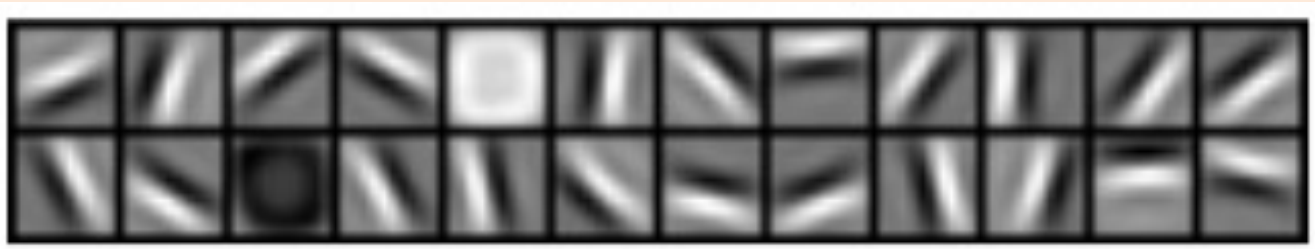


- Models require many tricks to work.
  - We'll discuss these next time.

http://www.cs.toronto.edu/~rgrosse/icml09-cdbn.pdf

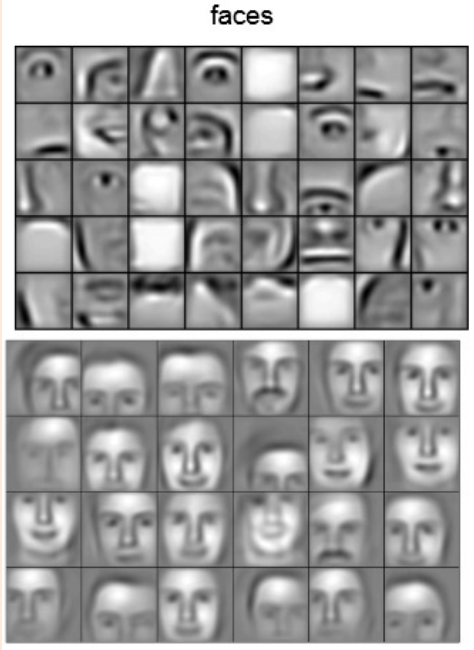
Cool Picture Motivation for Deep Learning

- First layer of z_i trained on 10 by 10 image patches:



"Gabor filters"

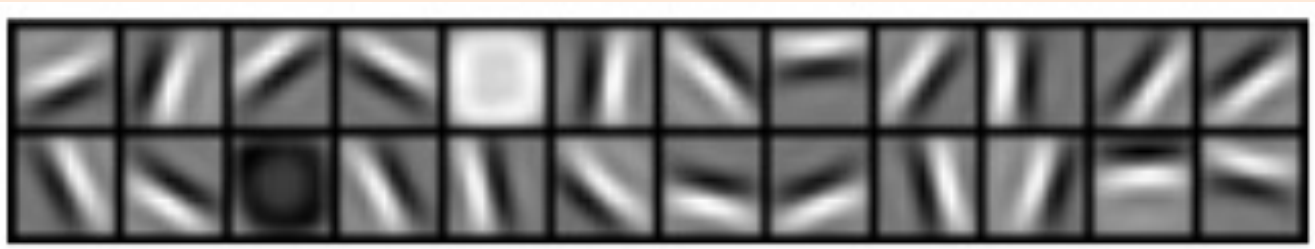
- Visualization of second and third layers trained on specific objects:



http://www.cs.toronto.edu/~rgrosse

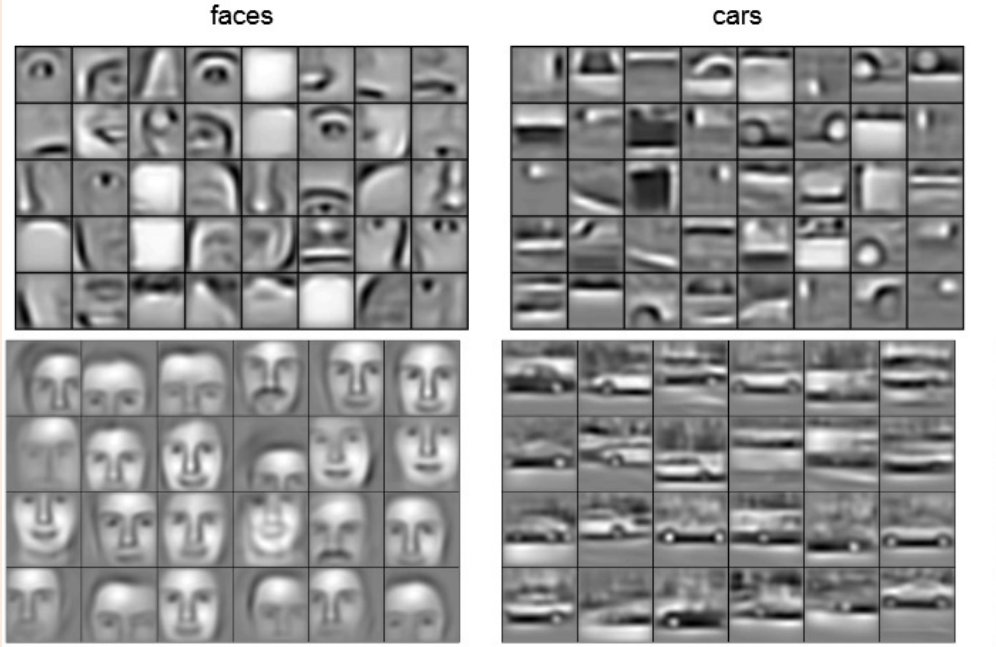
Cool Picture Motivation for Deep Learning

- First layer of z_i trained on 10 by 10 image patches:



"Gabor filters"

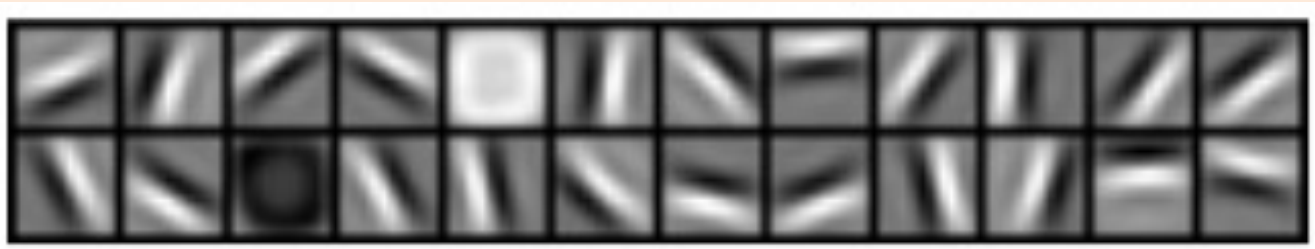
- Visualization of second and third layers trained on specific objects:



http://www.cs.toronto.edu/~rgrosse/icml09-cdbn.pdf

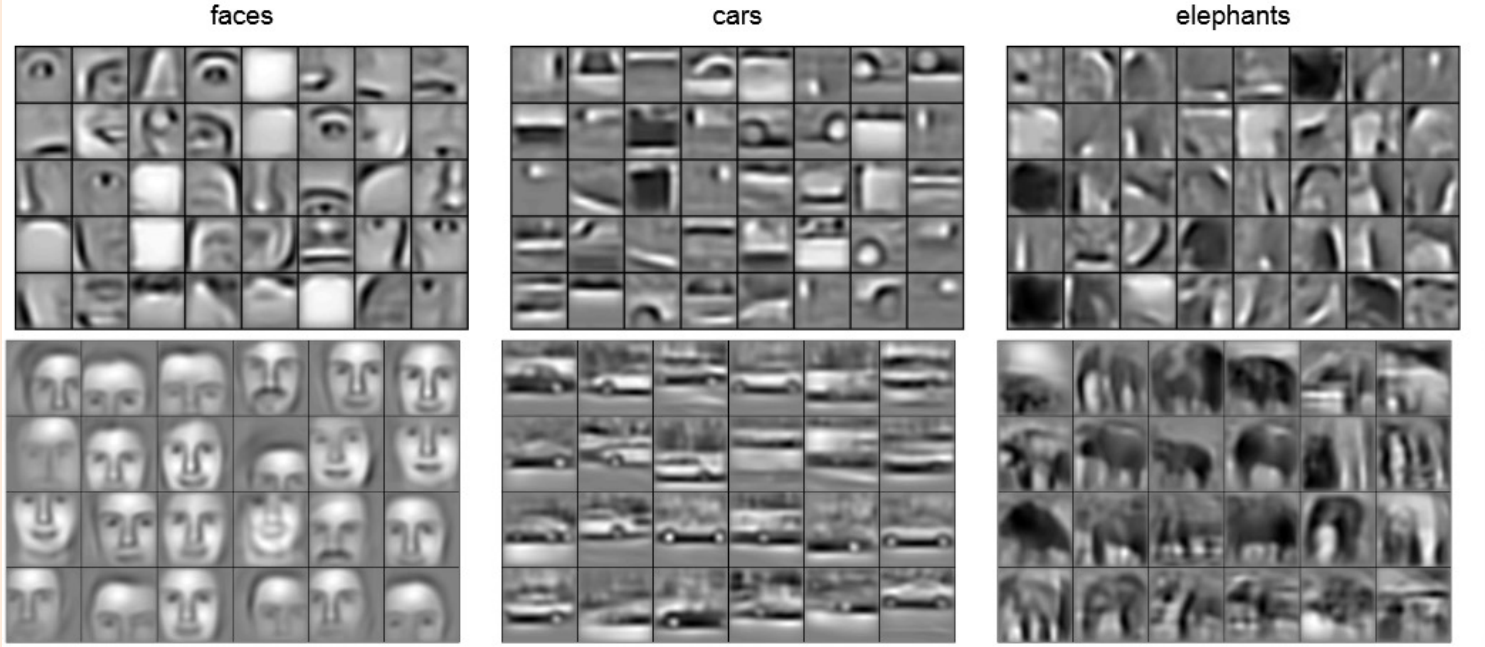
Cool Picture Motivation for Deep Learning

- First layer of z_i trained on 10 by 10 image patches:



"Gabor filters"

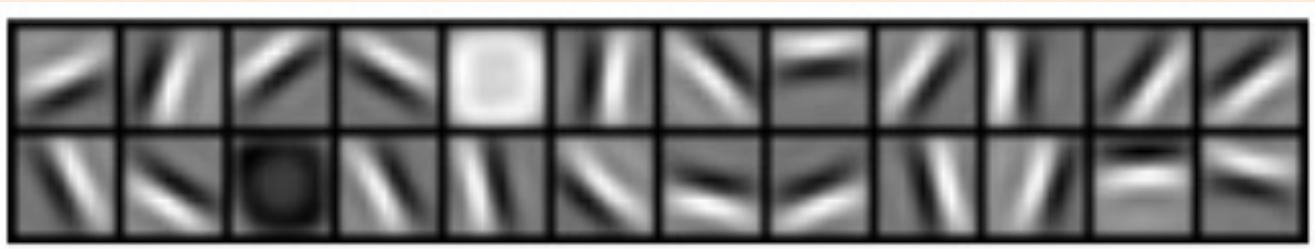
- Visualization of second and third layers trained on specific objects:



http://www.cs.toronto.edu/~rgrosse/icml09-cdbn.pdf

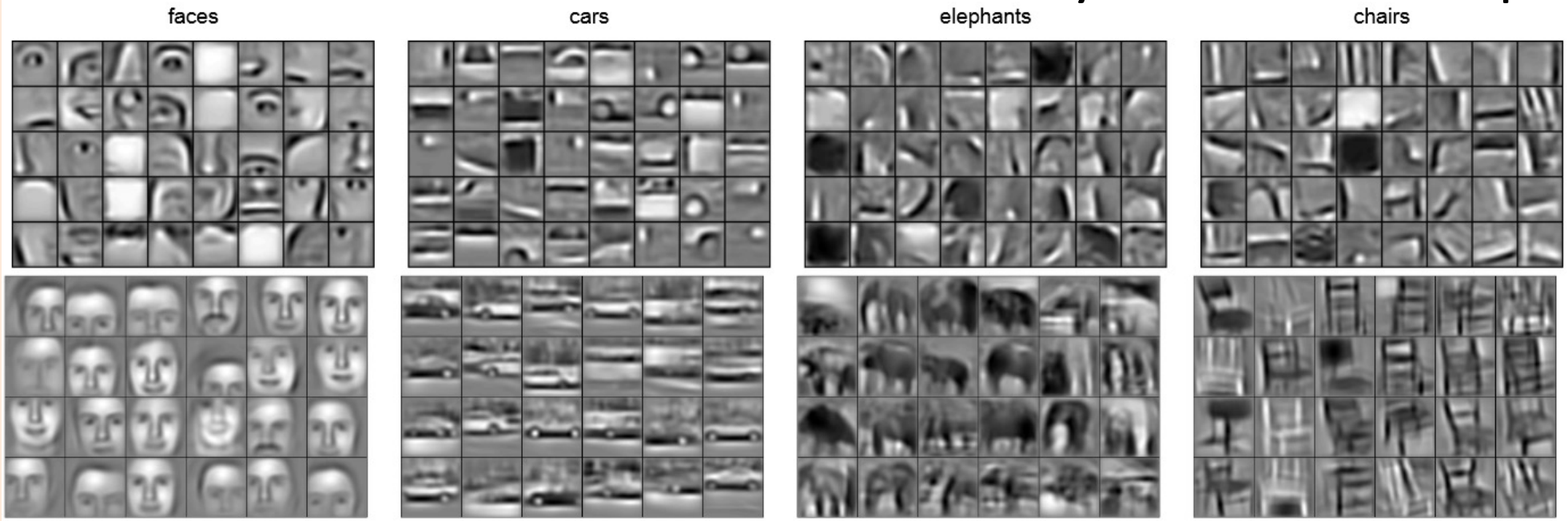
Cool Picture Motivation for Deep Learning

- First layer of z_i trained on 10 by 10 image patches:



"Gabor filters"

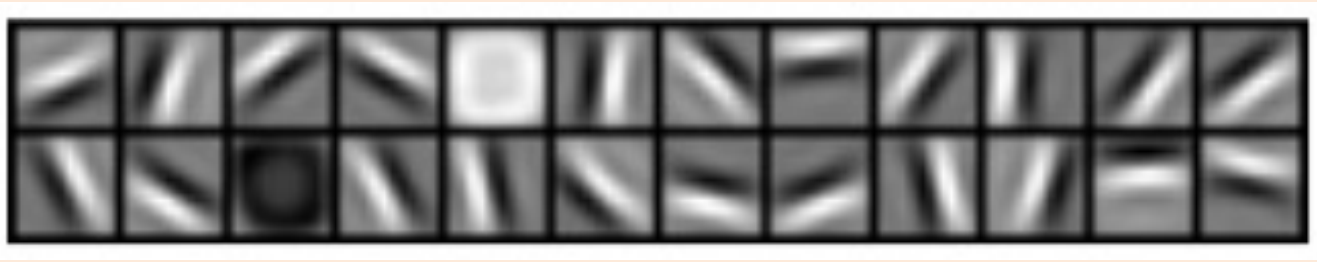
- Visualization of second and third layers trained on specific objects:



http://www.cs.toronto.edu/~rgrosse/icml09-cdbn.pdf

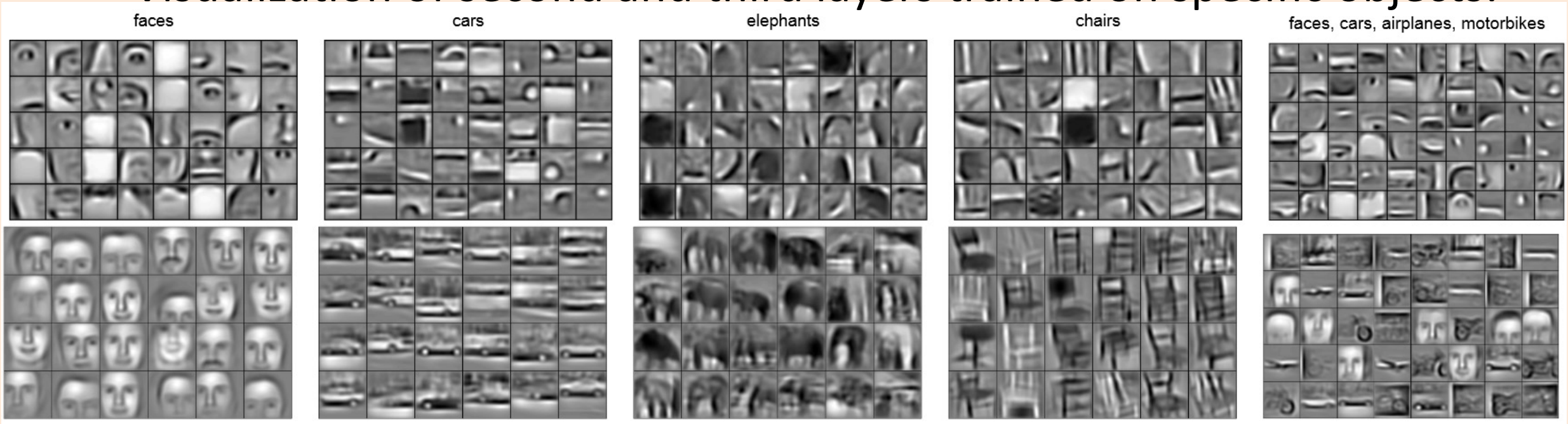
Cool Picture Motivation for Deep Learning

- First layer of z_i trained on 10 by 10 image patches:



"Gabor filters"

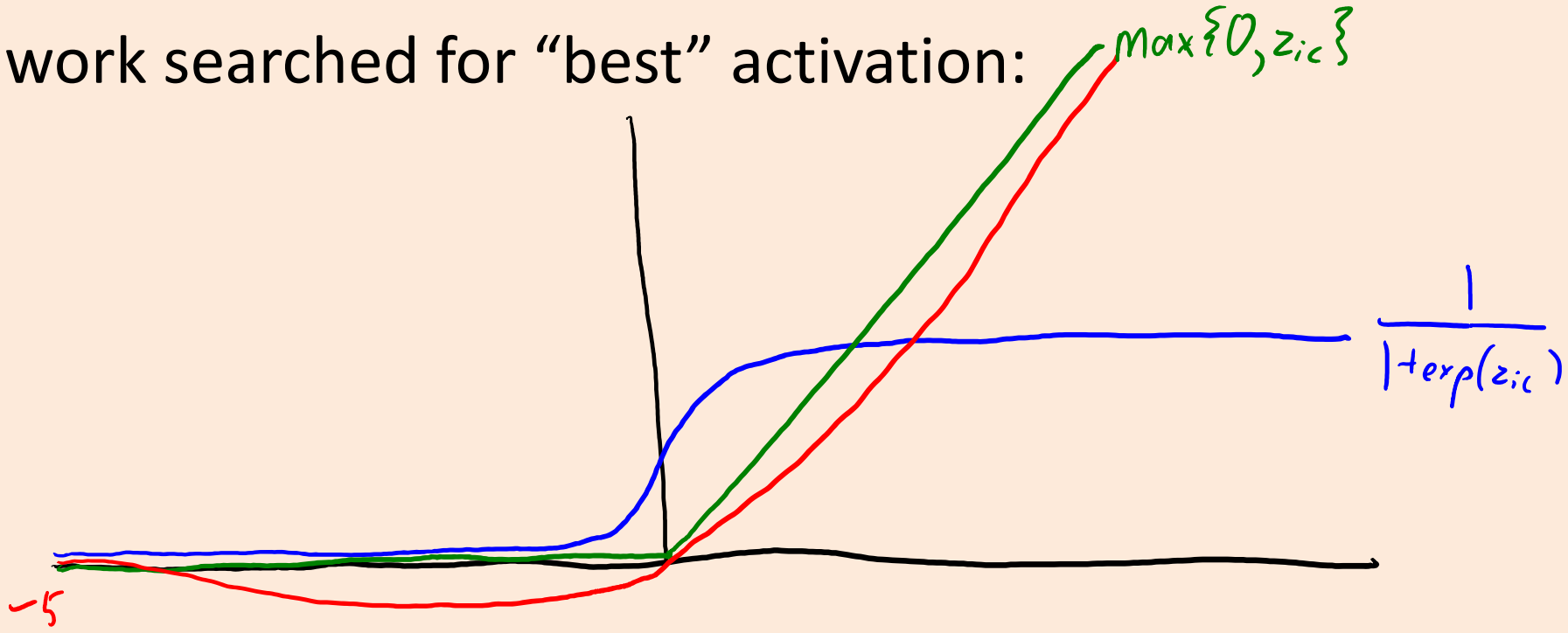
- Visualization of second and third layers trained on specific objects:



http://www.cs.toronto.edu/~rgrosse/icml09-cdbn.pdf

“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.

DenseNet

- Variation on ResNets is “DenseNets”:
  - Each layer can see all the values from many previous layers.
  - Significantly reduces vanishing gradients.
  - May get same performance with fewer parameters/layers.

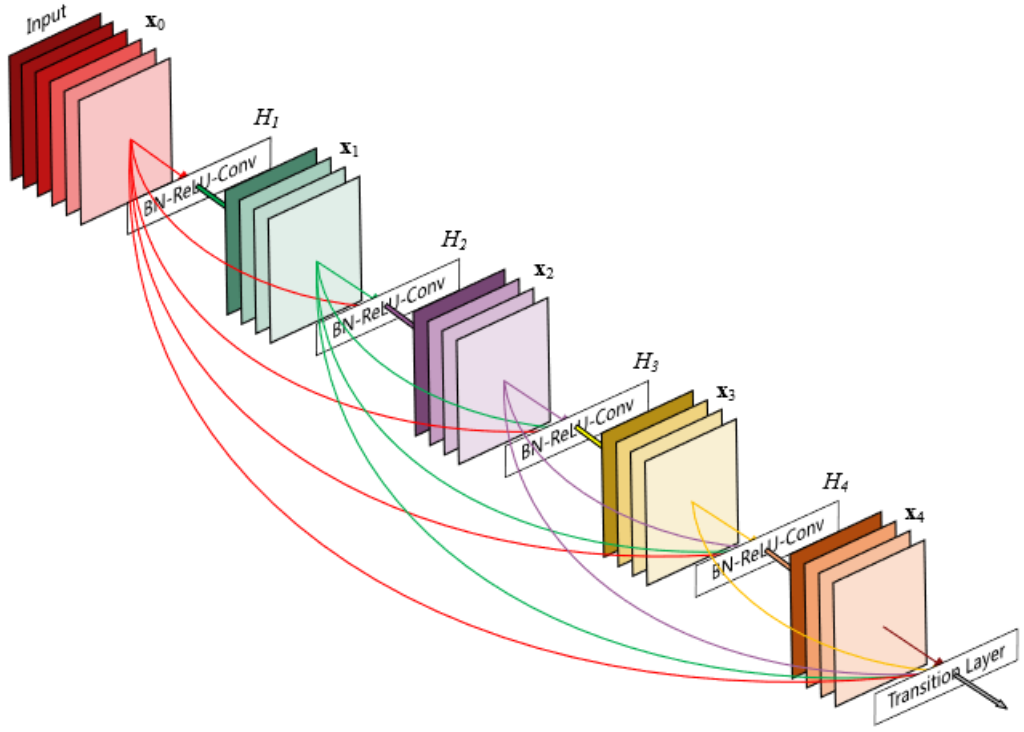


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

https://arxiv.org/pdf/1512.03385v1.pdf

Parameter Initialization

- Parameter initialization is crucial:
  - Can't initialize weights in same layer to same value, or they will stay same.
  - 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.

Parameter Initialization

- Parameter initialization is crucial:
  - Can't initialize weights in same layer to same value, or they will stay same.
  - Can't initialize weights too large, it will take too long to learn.
- Also common to transform data in various ways:
  - Subtract mean, divide by standard deviation, "whitten", 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

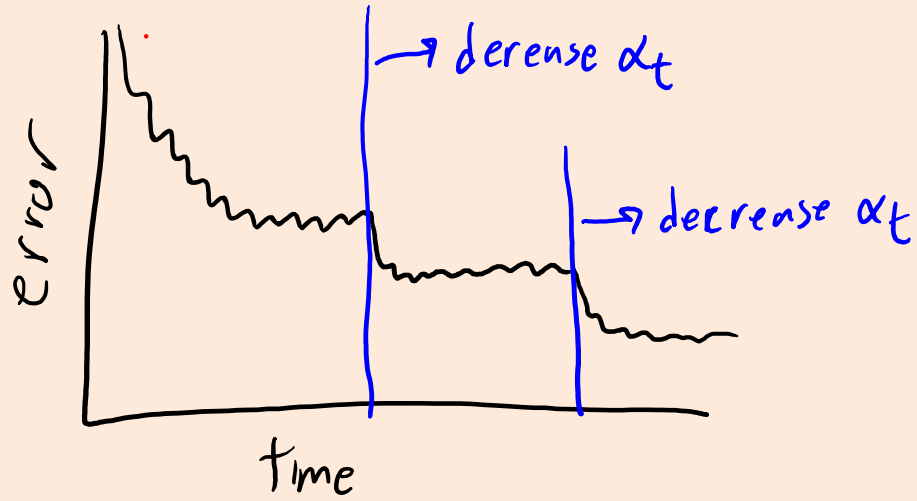
$$b = \frac{\sqrt{6}}{\sqrt{k^{(m)} + k^{(m-1)}}}$$

Setting the Step-Size: Bottou Trick

- 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).

Setting the Step-Size

- Stochastic gradient is very sensitive to the step size in deep models.
- Common 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

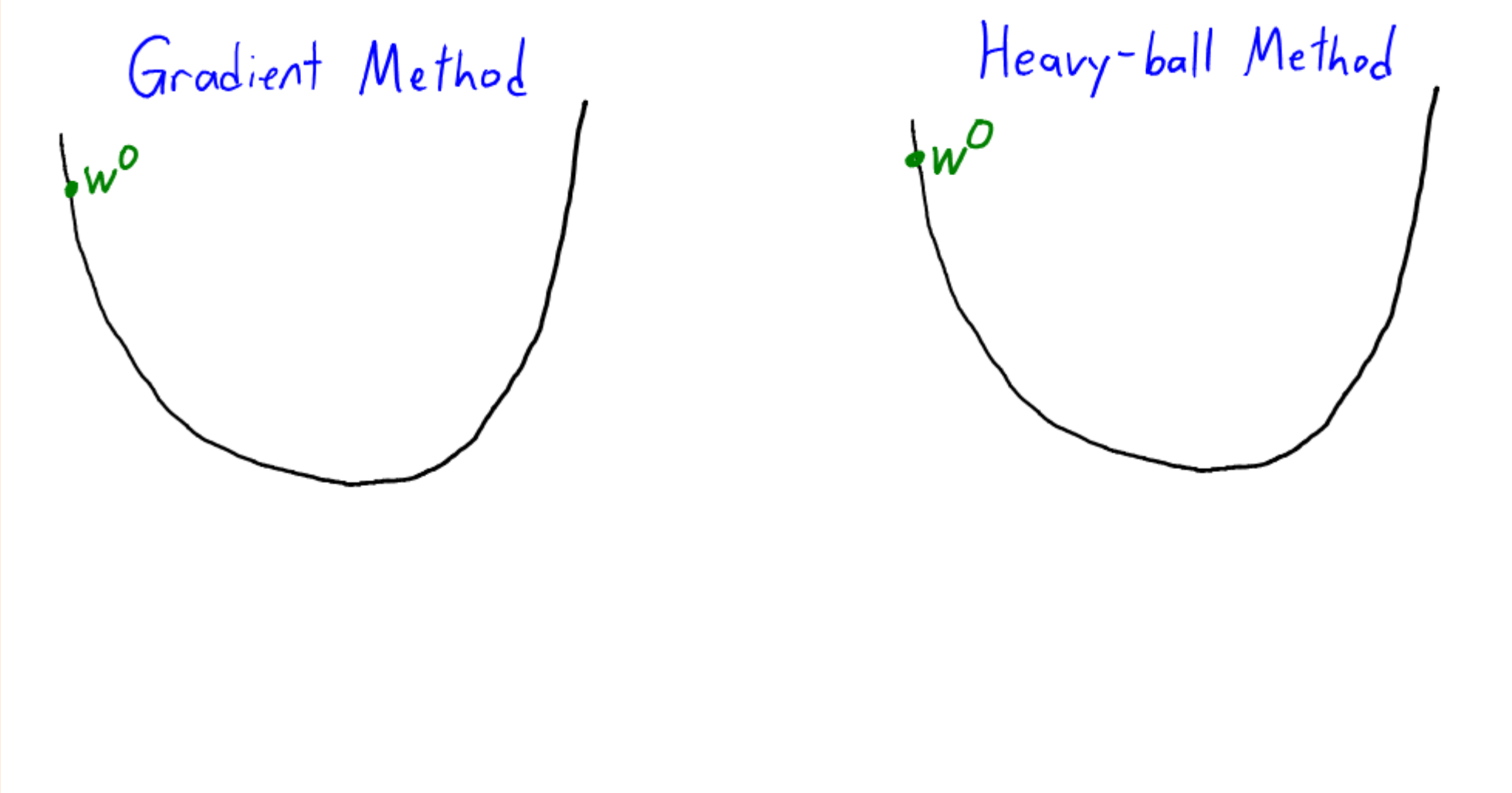
- Stochastic gradient is very sensitive to the step size in deep models.
- Momentum (stochastic version of "heavy-ball" algorithm):
  - Add term that moves in previous direction:

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

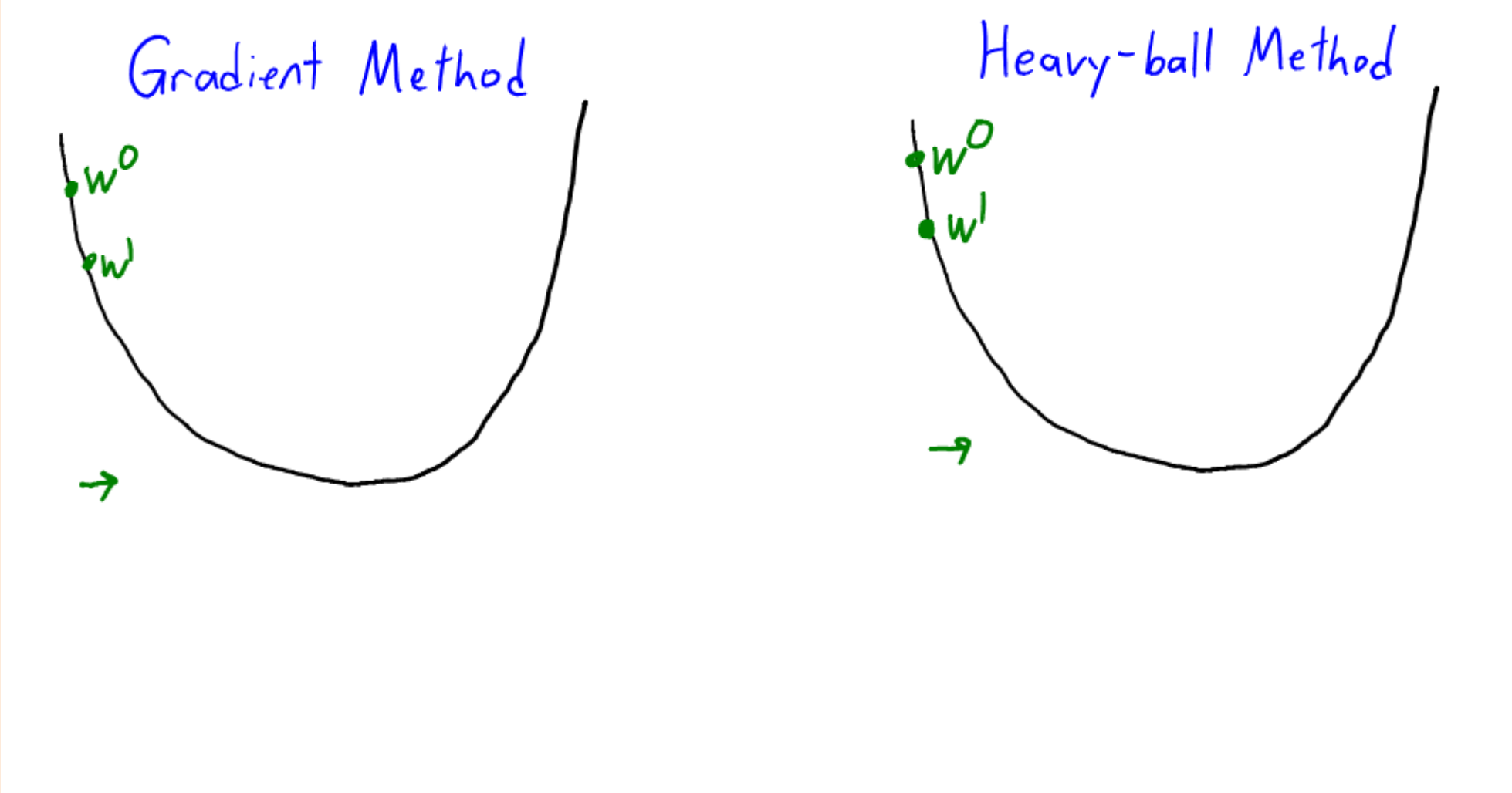
keep going in the old direction

  - Usually $\beta^t = 0.9$.

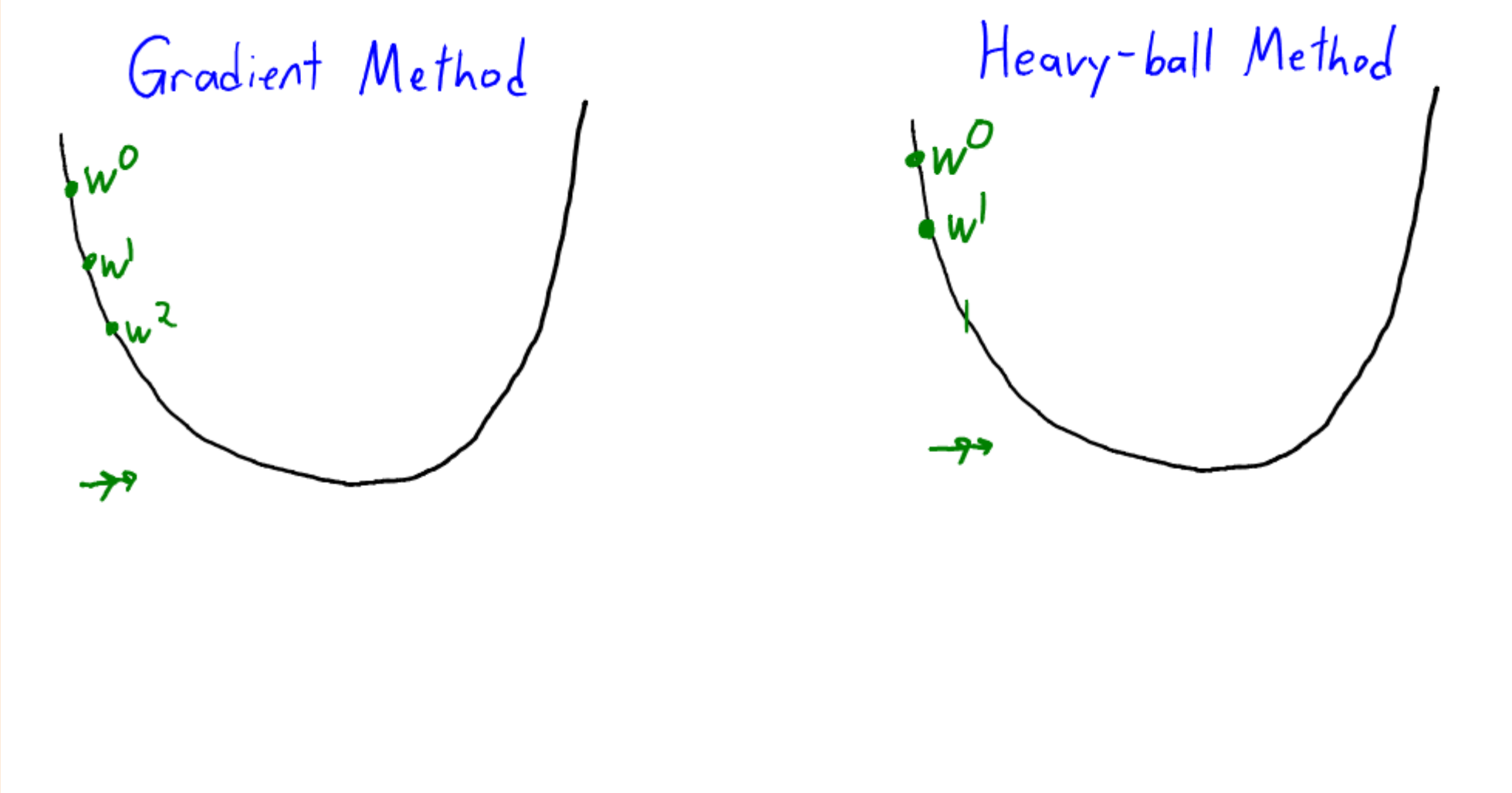
Gradient Descent vs. Heavy-Ball Method



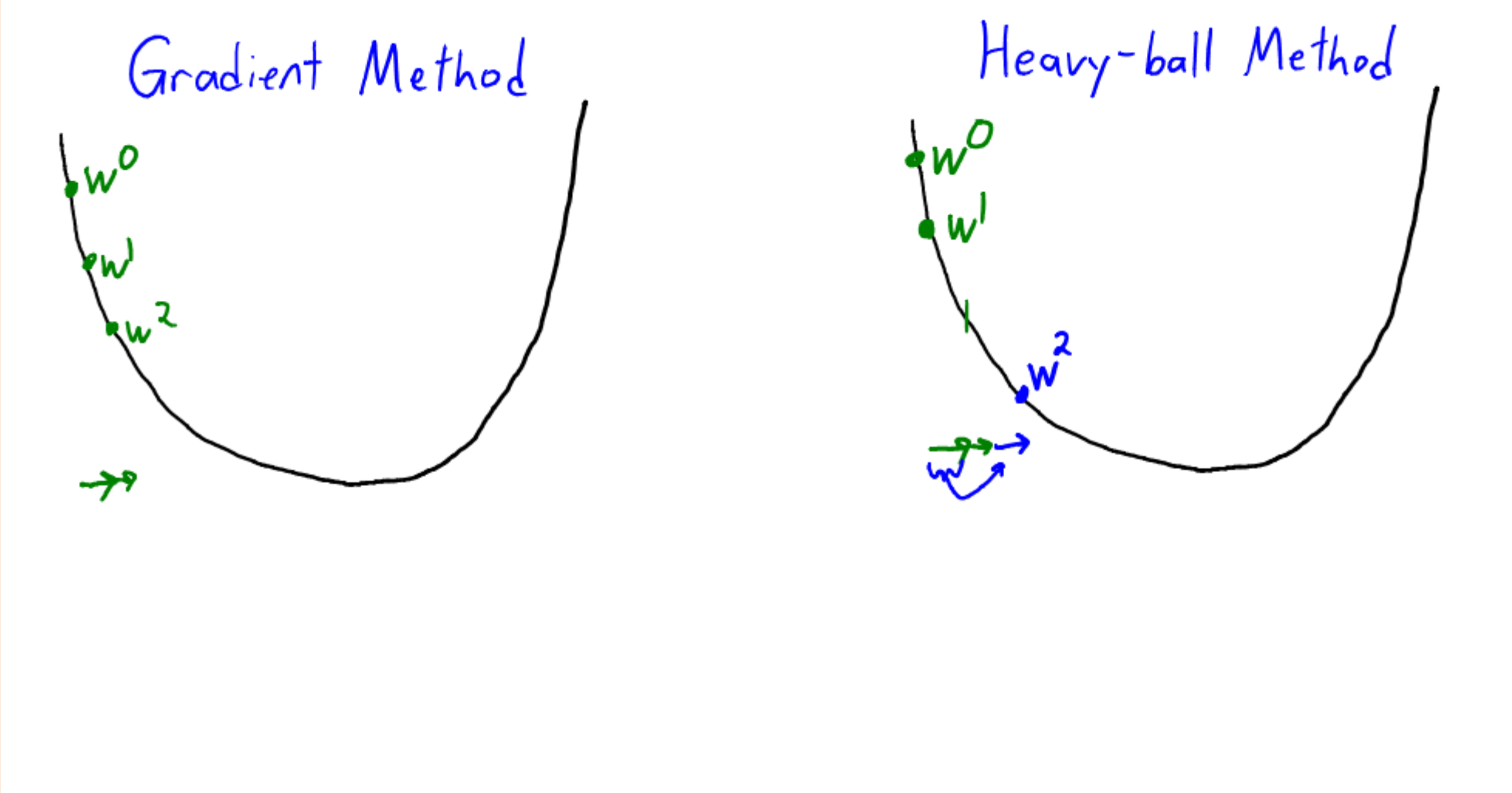
Gradient Descent vs. Heavy-Ball Method



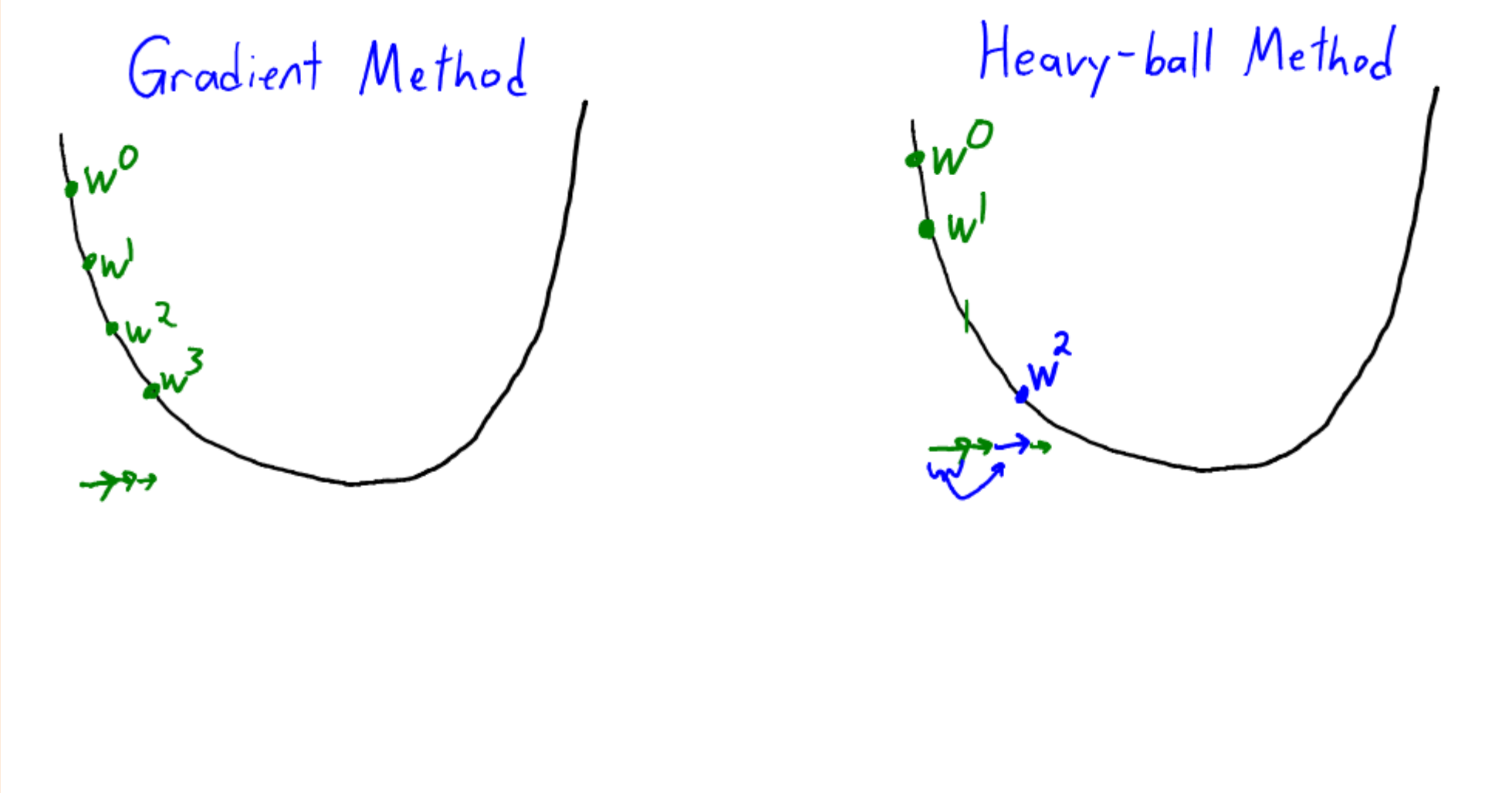
Gradient Descent vs. Heavy-Ball Method



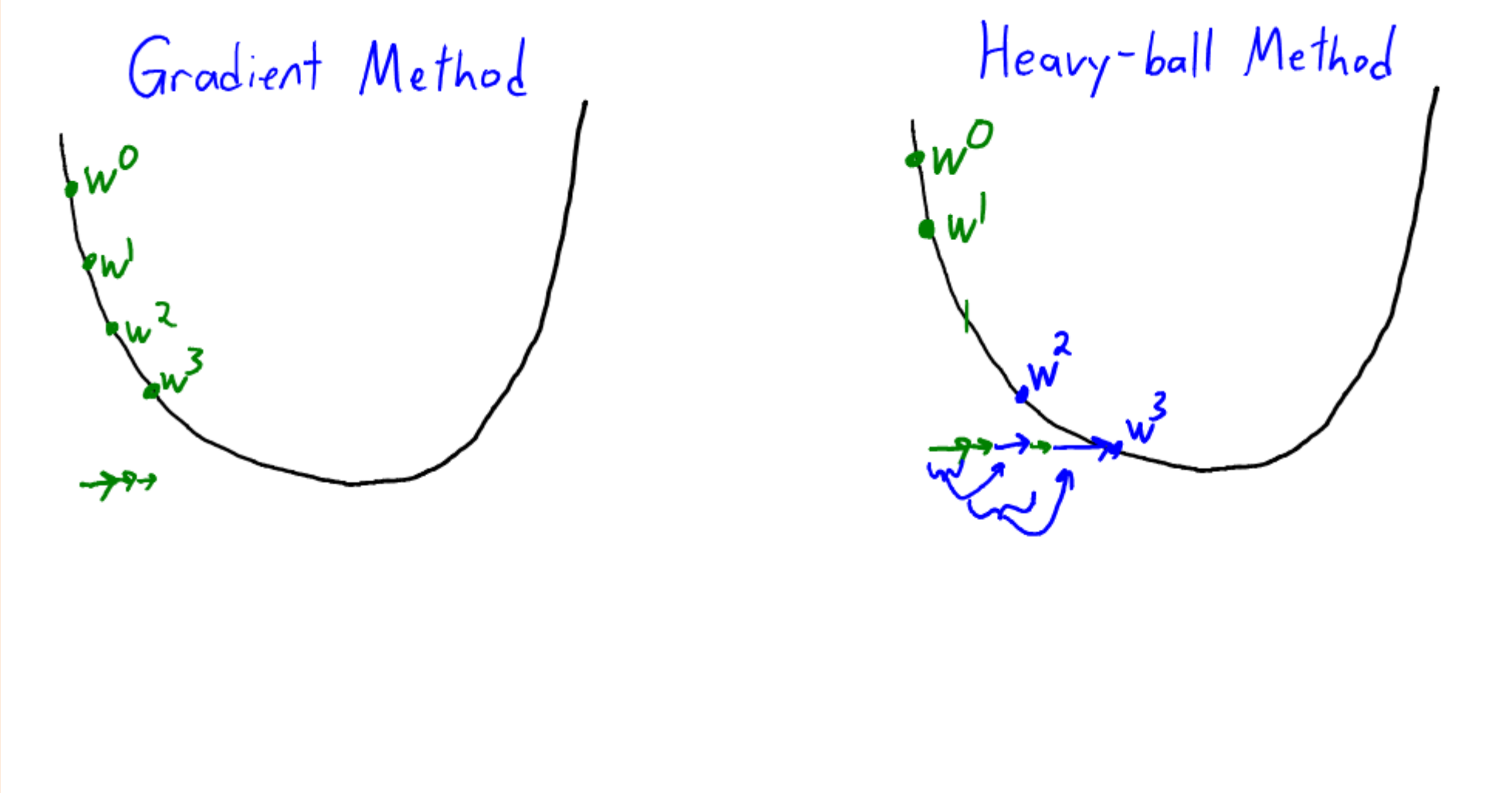
Gradient Descent vs. Heavy-Ball Method



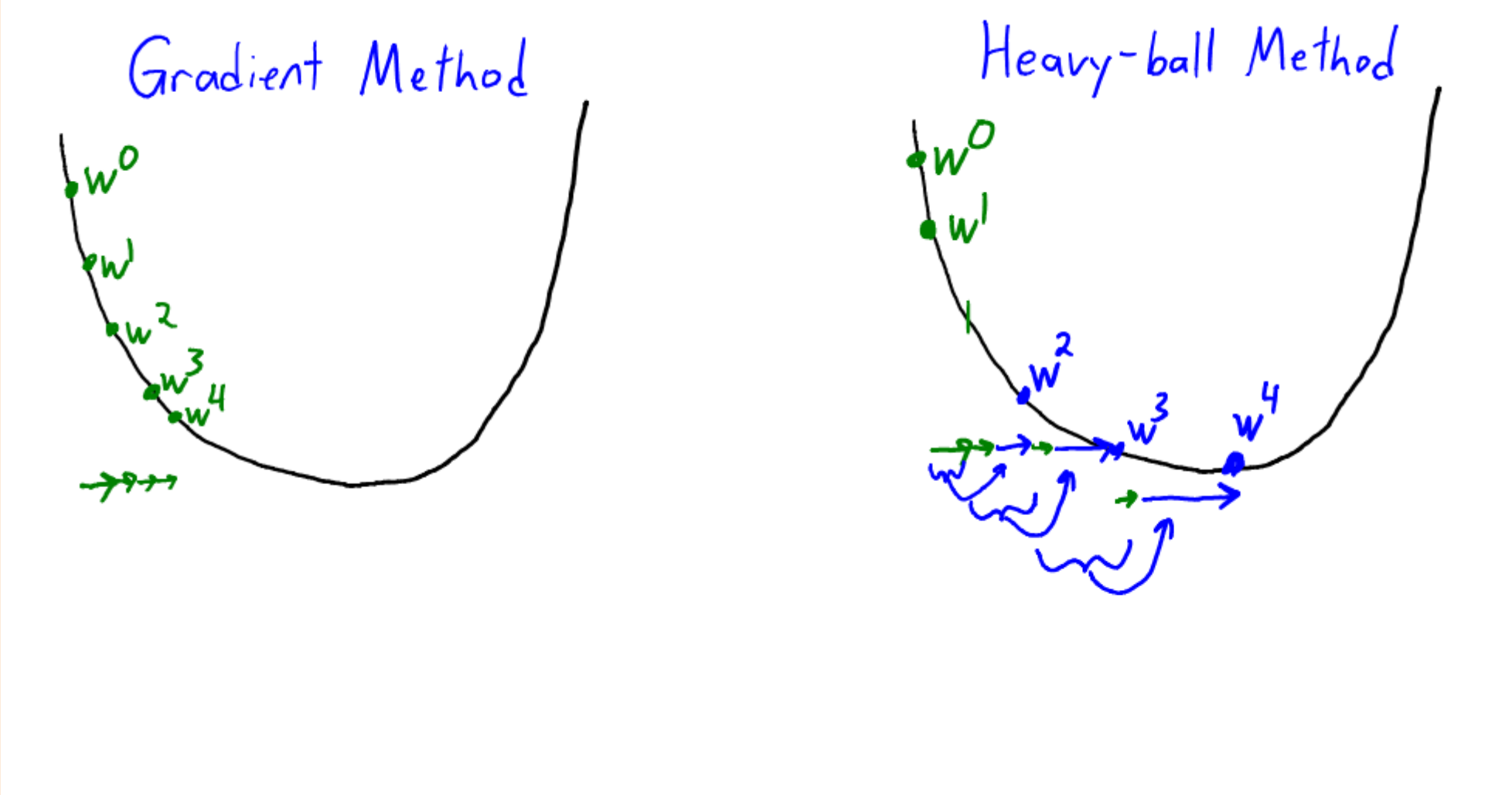
Gradient Descent vs. Heavy-Ball Method



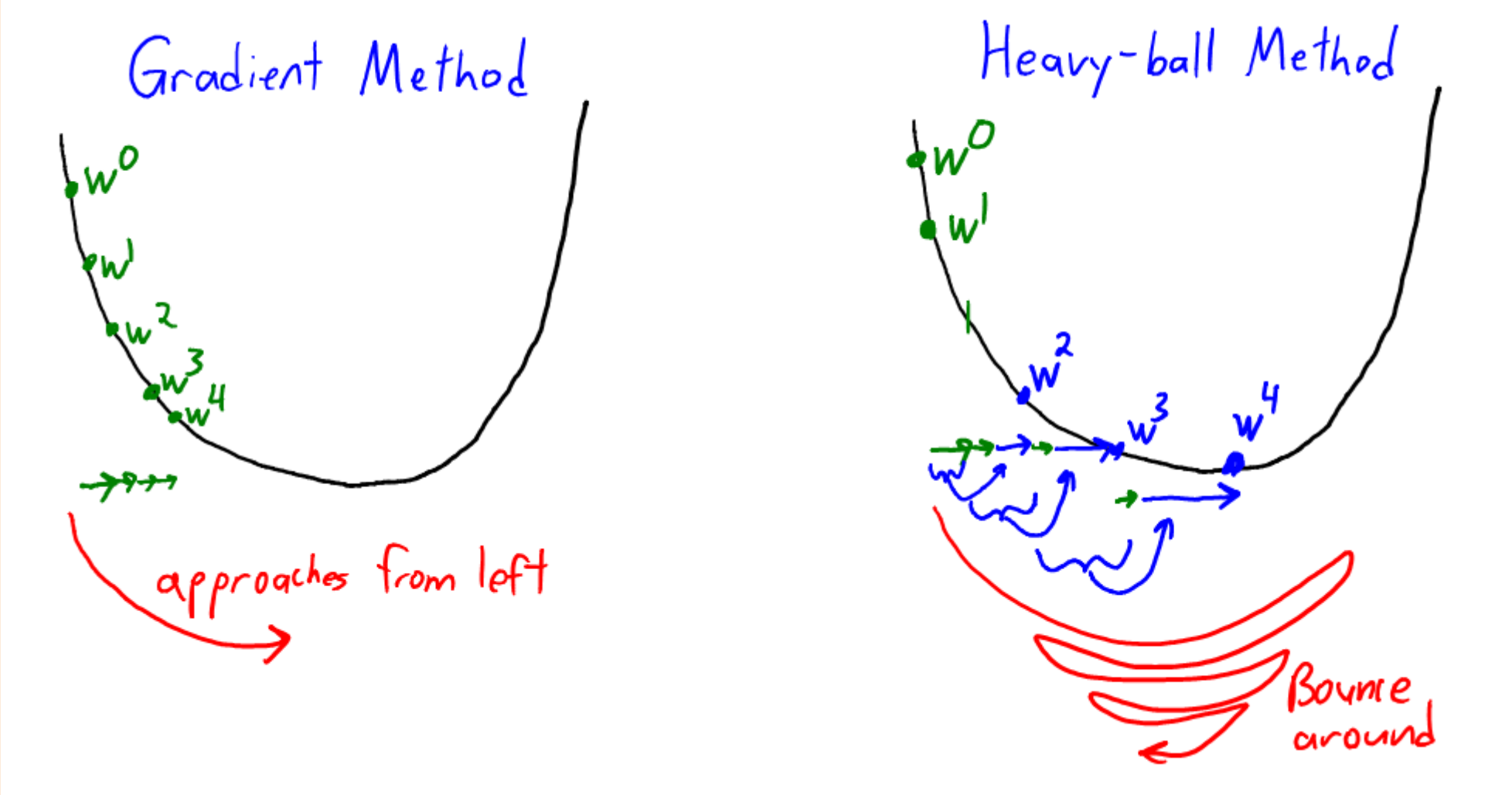
Gradient Descent vs. Heavy-Ball Method



Gradient Descent vs. Heavy-Ball Method



Gradient Descent vs. Heavy-Ball Method

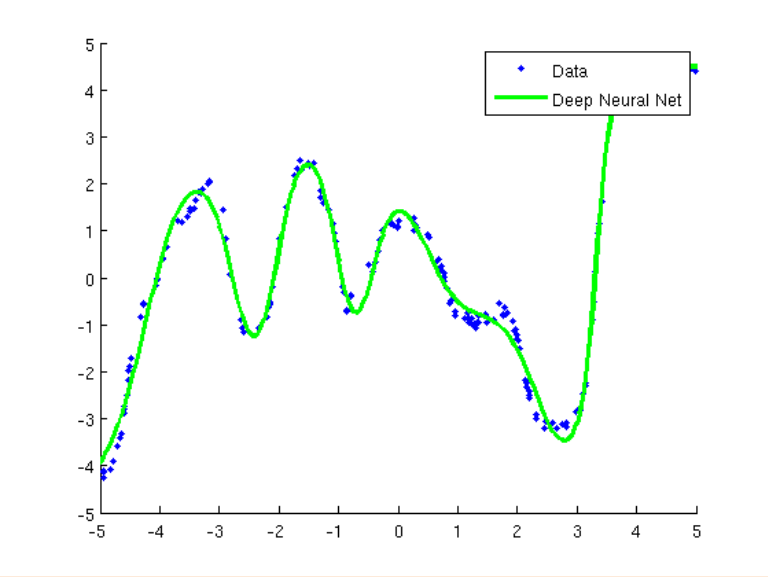


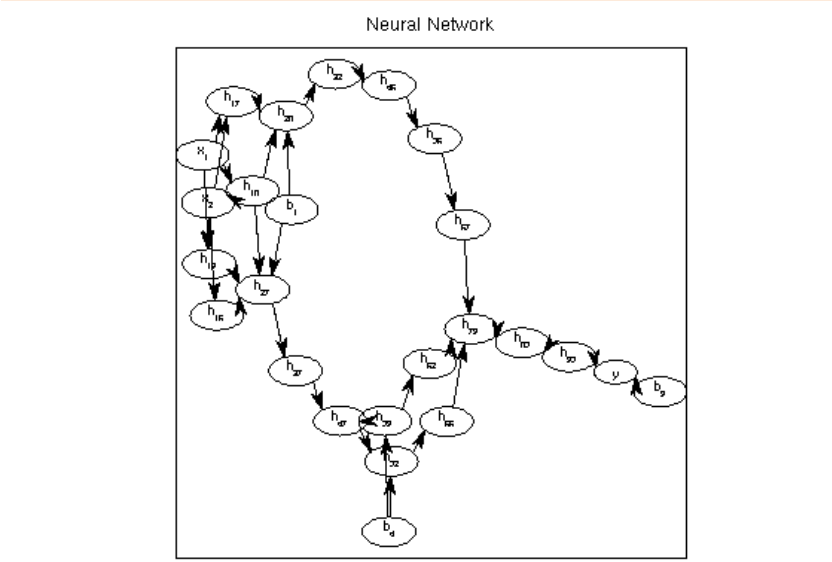
Standard Regularization

- Traditionally, we've added our usual L2-regularizers:

$$f(v, W^{(3)}, W^{(2)}, W^{(1)}) = \frac{1}{2}\sum_{i=1}^{n}(v^T h(W^{(3)} h(W^{(2)} h(W^{(1)} x_i))) - y_i)^2 + \frac{\lambda_4}{2}\|v\|^2 + \frac{\lambda_3}{2}\|W^{(3)}\|_F^2 + \frac{\lambda_2}{2}\|W^{(2)}\|_F^2 + \frac{\lambda_1}{2}\|W^{(1)}\|_F^2$$

- L2-regularization often called "weight decay" in this context.
  - Could also use L1-regularization: gives sparse network.





Standard Regularization

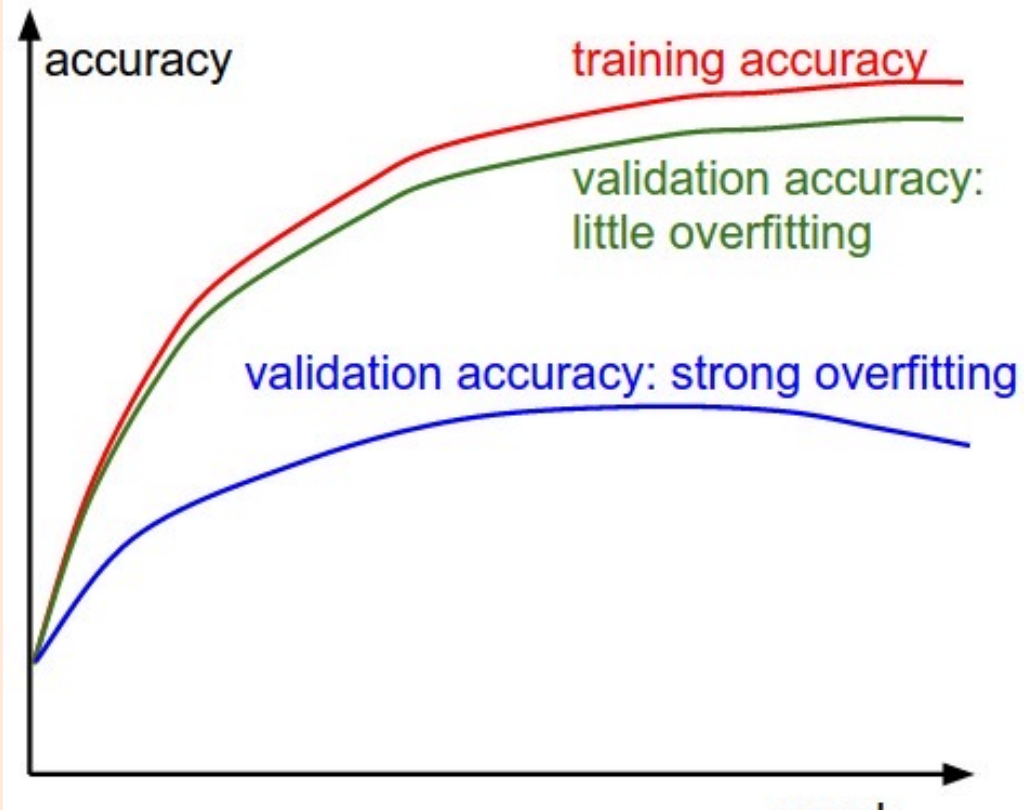
- Traditionally, we've added our usual L2-regularizers:

$$f(v, W^{(3)}, W^{(2)}, W^{(1)}) = \frac{1}{2}\sum_{i=1}^{n}(v^T h(W^{(3)} h(W^{(2)} h(W^{(1)} x_i))) - y_i)^2 + \frac{\lambda_4}{2}\|v\|^2 + \frac{\lambda_3}{2}\|W^{(3)}\|_F^2 + \frac{\lambda_2}{2}\|W^{(2)}\|_F^2 + \frac{\lambda_1}{2}\|W^{(1)}\|_F^2$$

- L2-regularization often called "weight decay" in this context.
  - Could also use L1-regularization: gives sparse network.
- Hyper-parameter optimization gets expensive:
  - Try to optimize validation error in terms of λ_1, λ_2, λ_3, λ_4.
  - In addition to step-size, number of layers, size of layers, initialization.
- Recent result:
  - Adding a regularizer in this way creates bad local optima.

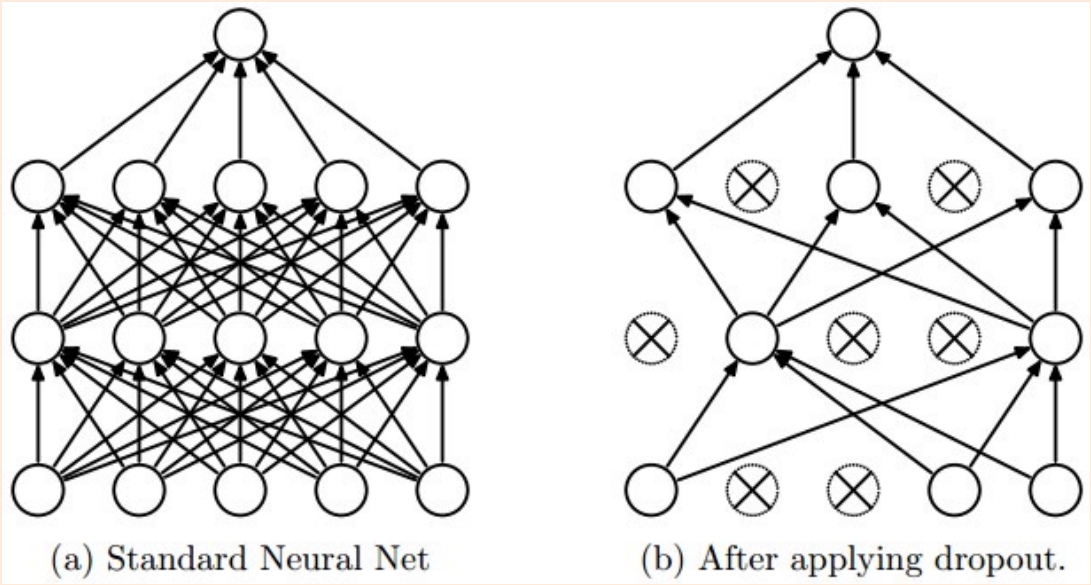
Early Stopping

- Another common type of regularization is "early stopping":
  - Monitor the validation error as we run stochastic gradient.
  - Stop the algorithm if validation error starts increasing.



Dropout

- Dropout is a more recent form of explicit regularization:
  - On each iteration, randomly set some x_i and z_i to zero (often use 50%).



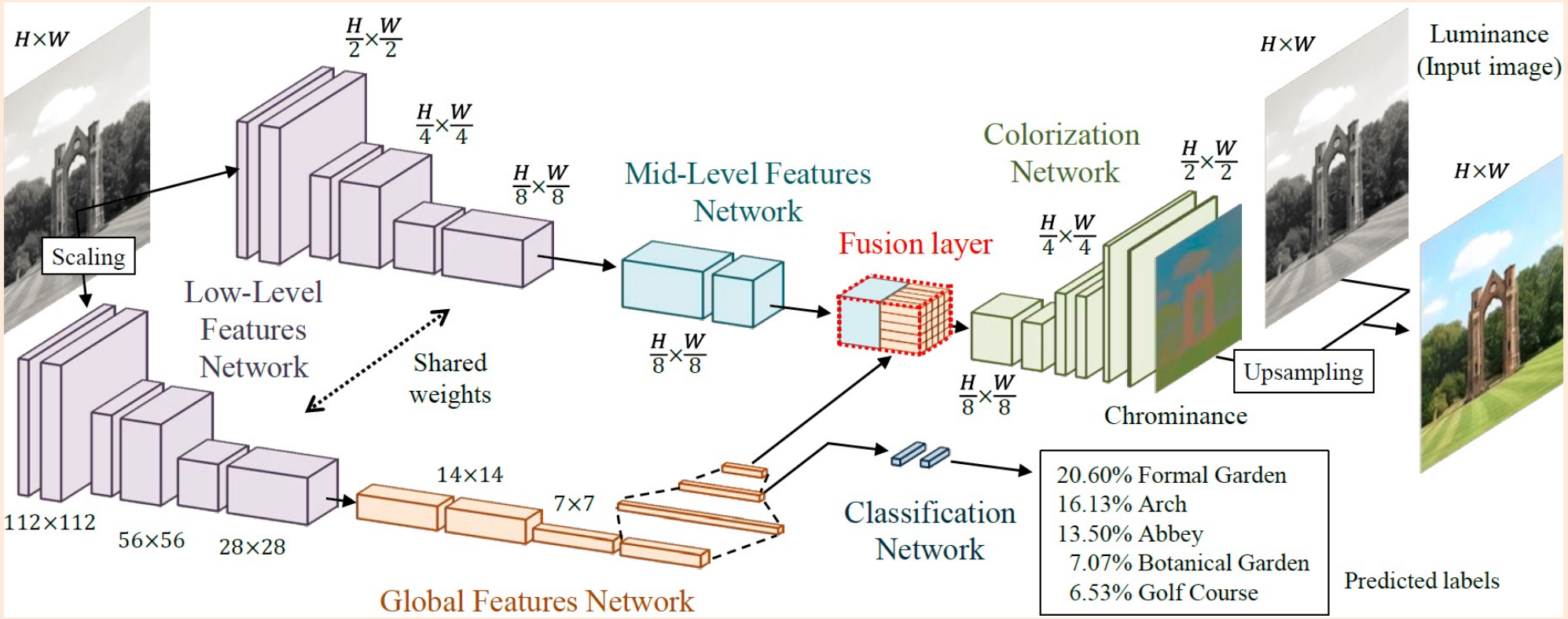
  - Adds invariance to missing inputs or latent factors
    - Encourages distributed representation rather than relying on specific z_i.
  - Can be interpreted as an ensemble over networks with different parts missing.
  - After a lot of success, dropout may already be going out of fashion.

http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

Next Topic: Automatic Differentiation

More-Complicated Layers

- Modern networks often have complicated structures:
  - Each step might be doing a different operation.
  - This makes coding up the gradient both time-consuming and prone to errors.

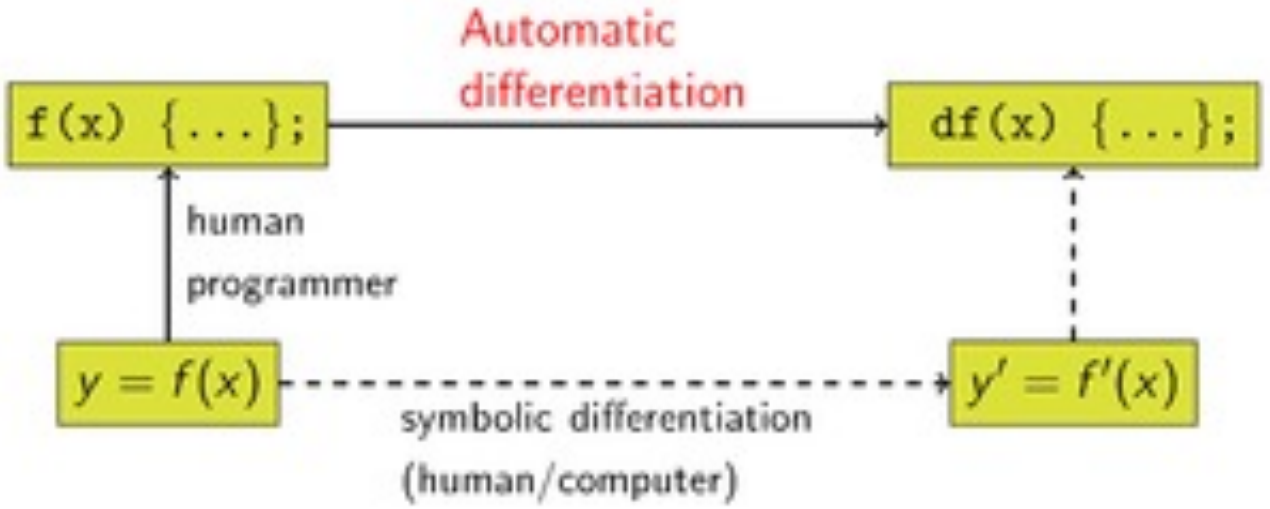


- Developing networks like this is made easier using automatic differentiation.

http://iizuka.cs.tsukuba.ac.jp/projects/colorization/en/

Automatic Differentiation (AD)

- Automatic differentiation (AD):
  - Input: code computing a function.
  - Output: code to compute one or more derivatives of the function.
    - No loss in accuracy, unlike finite-difference approximations.
    - The output code has the same asymptotic runtime as the input code.
    - Does not give you a "formula" for the derivative, just code that computes it.



https://en.wikipedia.org/wiki/Automatic_differentiation

“Reverse Mode” Automatic Differentiation (AD)

- In machine learning, we typically use “reverse mode” AD.
  - Gives code for computing the gradient of a differentiable function.
    - The slides will exclusively talk about “reverse mode”. For “forward mode”, see bonus.
    - You use this gradient to train the via SGD.

- Has a close connection to backpropagation.
  - Classic algorithm to compute the gradient of neural network parameters.
    - “Apply the chain rule, store redundant calculations”.
  - Backpropagation can be viewed as special case of AD.

If $f(x) = g(h(x))$
then $f'(y) = g'(h(x))h'(x)$

- AD basically writes every operation as instance of the chain rule.

Automatic Differentiation – Single Input+Output

- Consider the function f(x) = 10*log(1+exp(-2*x)).
- We write the function as a series of compositions: $f_5(f_4(f_3(f_2(f_1(x)))))$.
  - Where $f_1(x) = -2*x$, $f_2(z) = \exp(z)$, $f_3(z) = 1+z$, $f_4(z) = \log(z)$, $f_5(z) = 10*x$.
    - So we have $f_1'(x) = -2$, $f_2'(z) = \exp(z)$, $f_3'(z) = 1$, $f_4'(z) = 1/z$, $f_5'(z) = 10$.
      - These all cost O(1).
- Recursively applying the chain rule we get:
  - $f'(x) = f_5'(f_4(f_3(f_2(f_1(x)))))*f_4'(f_3(f_2(f_1(x))))*f_3'(f_2(f_1(x)))*f_2'(f_1(x))f_1'(x)$.

$10 * \frac{1}{f_3(f_2(f_1(x)))} * 1 * \exp(f_1(x)) * -2 \Rightarrow -\frac{20\exp(-2x)}{1+\exp(-2x)}$

$\frac{1}{1+\exp(-2x)}$ $\exp(-2x)$

Automatic Differentiation – Single Input+Output

- Our function written as a set of compositions:
  - $f_5(f_4(f_3(f_2(f_1(x)))))$.
- The derivative written using the chain rule::
  - $f'(x) = f_5'(f_4(f_3(f_2(f_1(x)))))*f_4'(f_3(f_2(f_1(x))))*f_3'(f_2(f_1(x)))*f_2'(f_1(x))f_1'(x)$.
- Notice that this leads to repeated calculations.
  - For example, we use $f_1(x)$ four different times.
  - We can use dynamic programming to avoid redundant calculations.
- First, the "forward pass" will compute and store the expressions:
  - $\alpha_1 = f_1(x)$, $\alpha_2 = f_2(\alpha_1)$, $\alpha_3 = f_3(\alpha_2)$, $\alpha_4 = f_4(\alpha_3)$, $\alpha_5 = f_5(\alpha_4) = f(x)$.
- Next, the "backward pass" uses stored α_k values and f_i' functions:
  - $\beta_5 = 1*f_5'(\alpha_4)$, $\beta_4 = \beta_5*f_4'(\alpha_3)$, $\beta_3 = \beta_4*f_3'(\alpha_2)$, $\beta_2 = \beta_3*f_2'(\alpha_1)$, $\beta_1 = \beta_2*f_1'(x) = f'(x)$.
- A generic method to make code computing f'(x) for same cost as f(x).

Summary

- Vanishing gradients in deep networks (gradient may be close to 0).
  - Can be reduced using rectified linear units (ReLU) as non-linear transforms.
  - Can be reduced using various forms of skip connections.
- ResNets include untransformed previous layers.
  - Network focuses non-linearity on residual, allows huge number of layers.

- Automatic differentiation (AD):
  - Decomposing code using the chain rule, to make derivative code.
  - Can compute gradient for same cost as objective function.
- Backpropagation is a form of AD and computes neural network gradients via chain rule.

- Next time: The most important idea in computer vision?

Forward-Mode Automatic Differentiation

- We discussed "reverse-mode" automatic differentiation.
  - Given a function, writes code to compute its gradient.
  - Has same cost as original function.
  - But has high memory requirements.
    - Since you need to store all the intermediate calculations.

- There is also "forward-mode" automatic differentiation.
  - Given a function, writes code to compute a directional derivative.
    - Scalar value measuring how much the function changes in one direction.
  - Has same memory requirements as original function.
  - But has high cost if you want the gradient.
    - Need to use it once to get each partial derivative.

Failure of AD on ReLUs

In many settings, our underlying function $f(x)$ is a nonsmooth function, and we resort to subgradient methods. This work considers the question: is there a *Cheap Subgradient Principle*? Specifically, given a program that computes a (locally Lipschitz) function f and given a point x, can we automatically compute an element of the (Clarke) subdifferential $\partial f(x)$ [Clarke, 1975], and can we do this at a cost which is comparable to computing the function $f(x)$ itself? Informally, the set $\partial f(x)$ is the convex hull of limits of gradients at nearby differentiable points. It can be thought of as generalizing the gradient (for smooth functions) and the subgradient (for convex functions).

Let us briefly consider how current approaches handle nonsmooth functions, which are available to the user as functions in some library. Consider the following three equivalent ways to write the identity function, where $x \in \mathbb{R}$,

$$f_1(x) = x, \quad f_2(x) = \mathrm{ReLU}\,(x) - \mathrm{ReLU}\,(-x)\,, \quad f_3(x) = 10 f_1(x) - 9 f_2(x)\,,$$

where $\mathrm{ReLU}\,(x) = \max\{x, 0\}$, and so $f_1(x) = f_2(x) = f_3(x)$. As these functions are differentiable at 0, the unique derivative is $f_1'(0) = f_2'(0) = f_3'(0) = 1$. However, both TensorFlow [Abadi et al., 2015] and PyTorch [Paszke et al., 2017], claim that $f_1'(0) = 1$, $f_2'(0) = 0$, $f_3'(0) = 10$. This particular answer is due to using a subgradient of 0 at $x = 0$. One may ask if a more judicious choice fixes such issues; unfortunately, it is not difficult to see that no such universal choice exists[1].

[1] By defining $\mathrm{ReLU}'(0) = 1/2$, the reader may note we obtain the correct derivative on f_2, f_3; however, consider $f_4(x) = \mathrm{ReLU}\,(\mathrm{ReLU}\,(x)) - \mathrm{ReLU}\,(-x)$, which also equals $f_1(x)$. Here, we would need $\mathrm{ReLU}'(0) = \frac{\sqrt{5}-1}{2}$ to obtain the correct answer.

https://arxiv.org/pdf/1809.08530.pdf