A whirlwind review/overview of deep learning

CPSC 440/550: Advanced Machine Learning

`cs.ubc.ca/~dsuth/440/23w2`

University of British Columbia, on unceded Musqueam land

2023-24 Winter Term 2 (Jan–Apr 2024)

admin

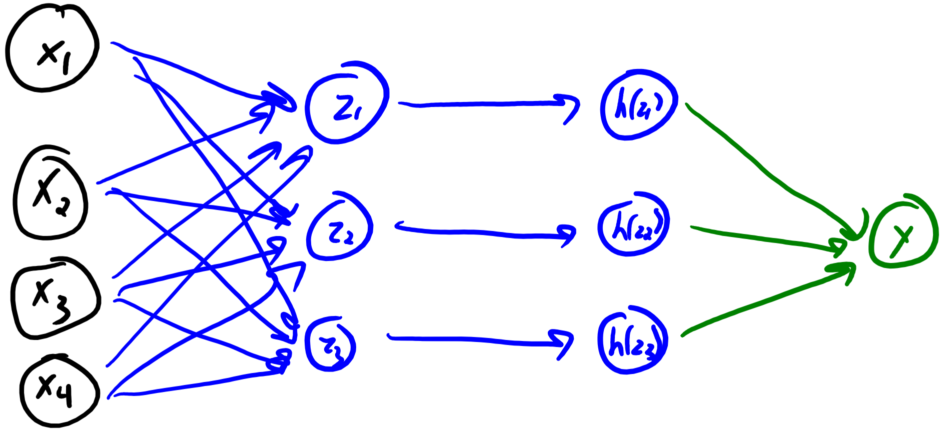
Admin

- Quiz 1 delayed to next week
- You’ll be able to book a session starting tomorrow
- Tentative plan is Quiz 2 and on stick to original timing
  - Quiz 2 week after next, future ones every two weeks of class from then

- Ten of you didn’t hand in assignment 1!
  - Count doesn’t include auditors or small number of people I’ve spoken to
  - Some probably are about to drop and it just didn’t process yet
  - Others: if something came up/etc, **please talk to me**

Last Time: Neural Networks

- We started with neural networks with one hidden layer:



$$\hat{y} = \underbrace{\underbrace{v}_{k\times 1}^T h(\underbrace{W}_{k\times d} \underbrace{x}_{d\times 1}}_{\text{Cost: } O(kd)})$$

$h: \mathbb{R}^k \mapsto \mathbb{R}^k$ (must be non-linear)

  - "Simultaneously learn the features and the linear model."
  - Often perform better with bias variables and/or residual/skip connections.
  - Probabilistic framing: a different parameterization of $\Pr(Y = y \mid x) = \theta_x$
    - We still usually assume $Y \mid X = x \sim Bern(\theta_x)$ and do MLE/MAP
    - Leads to non-convex training objective; train with SGD

Universal approximation

- For most activation functions, wide networks are "universal approximators"
  - Any continuous function can be approximated arbitrarily well on a bounded domain
  - Even with only one hidden layer

- But this result is for a non-parametric setting of the parameters:
  - The width of the hidden layer needs to grow with *n*
  - A fixed-size network is not a universal approximator

- Other universal approximators (always non-parametric):
  - K-nearest neighbours – if *k* grows with *n* (but fixed *k* would be non-parametric anyway)
  - Linear models on polynomial feature transformations – if degree grows with *n*
  - Linear models with Gaussian RBFs as non-linear features (with one basis per training *x*)
  - Linear models with a Gaussian kernel
  - Deep networks with fixed width (at least *d+2*), growing depth

Is Training Neural Networks Scary?

- Learning:
  - For binary classification, the NLL under the sigmoid loss is:

$$f(W,v) = \sum_{i=1}^{n} \underbrace{\log(1+\exp(-y^i v^T h(Wx^i)))}_{f_i}$$

loss function on example i

    - With *W* fixed this is convex, but with *W* and *v* as variables it is non-convex.
    - Finding the global optimum of non-convex functions is NP-hard in general.
  - Nearly always trained with variations on stochastic gradient descent (SGD).

$$W^{k+1} = W^k - \alpha^k \nabla_W f_{i_k}(W^k, v^k)$$

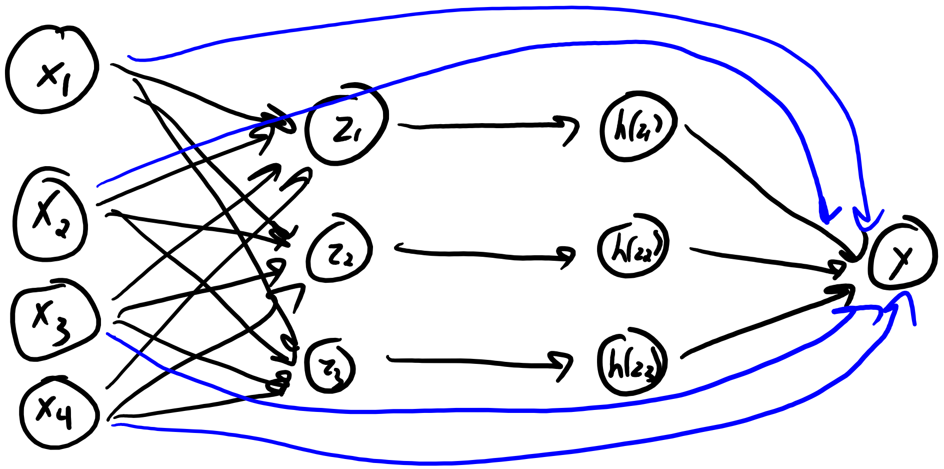
$$v^{k+1} = v^k - \alpha^k \nabla_v f_{i_k}(W^k, v^k)$$

i_k is a training example chosen uniformly at random

    - Many variations exist (adding “momentum”, AdaGrad, Adam, and so on).
    - SGD is not guaranteed to reach a global minimum for non-convex problems.
- Is non-convexity a big drawback compared to logistic regression?
  - And if *k* is large, is this likely to overfit?

Neural Networks ≥ Logistic Regression

- Consider a neural network with one hidden layer and connections from input to output layer.
  - The extra connections are called “skip” connections.



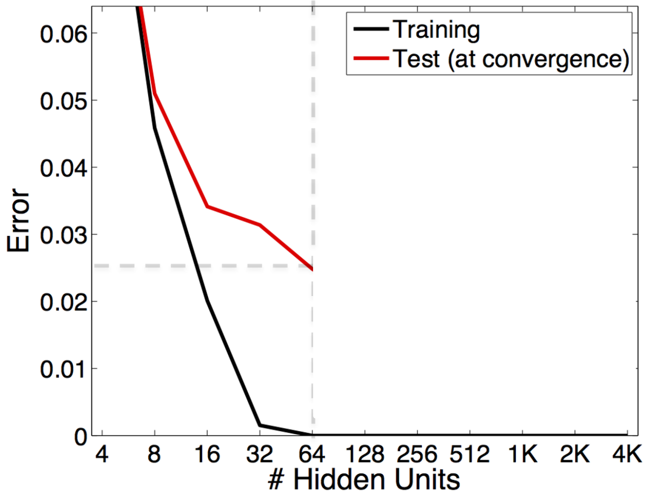
$$\hat{y} = \underbrace{w^T x}_{\text{linear model}} + \underbrace{v^T h(Wx)}_{\text{neural network}}$$

- You could first set *v*=0, then optimize *w* using logistic regression.
  - This is a convex optimization problem that gives you the logistic regression model.
- You could then set *W* and *v* to small random values, and start SGD from the logistic regression model.
  - Even though this is non-convex, the neural network can only improve on logistic regression (improves “residual” error).
- And if you are worried about overfitting, you could stop SGD by checking performance on validation set.
  - This is called regularization by “early stopping”.
- In practice, we typically optimize everything at once (which usually works better than the above).

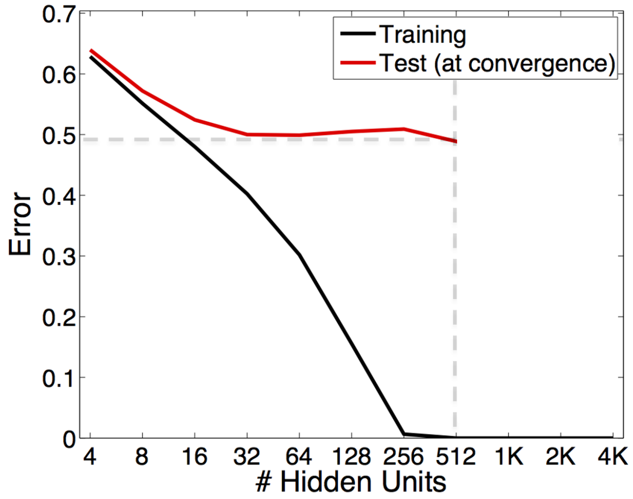
“Hidden” Regularization in Neural Networks

- Fitting single-layer neural network with SGD and no regularization:

MNIST



CIFAR-10



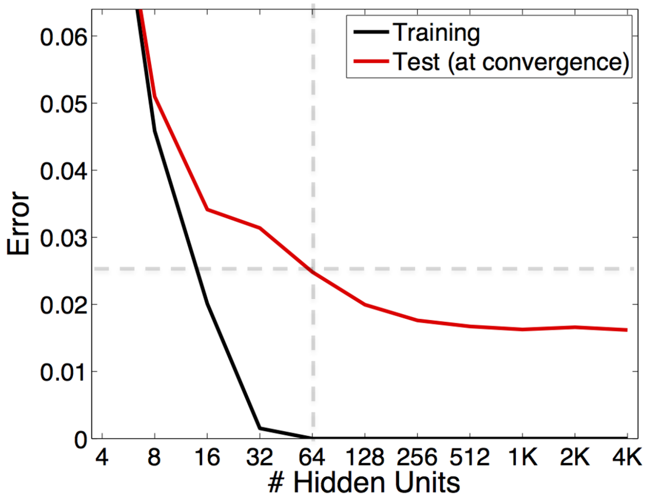
- On each step of the x-axis, the network is re-trained from scratch.
- Training goes to 0 with enough units: we’re finding a global min.
- What should happen to training and test error for larger #hidden?

https://www.neyshabur.net/papers/inductive_bias_poster.pdf

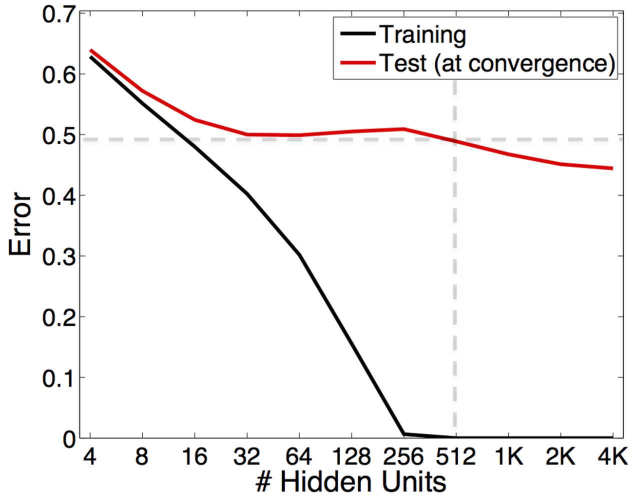
“Hidden” Regularization in Neural Networks

- Fitting single-layer neural network with SGD and no regularization:

MNIST



CIFAR-10

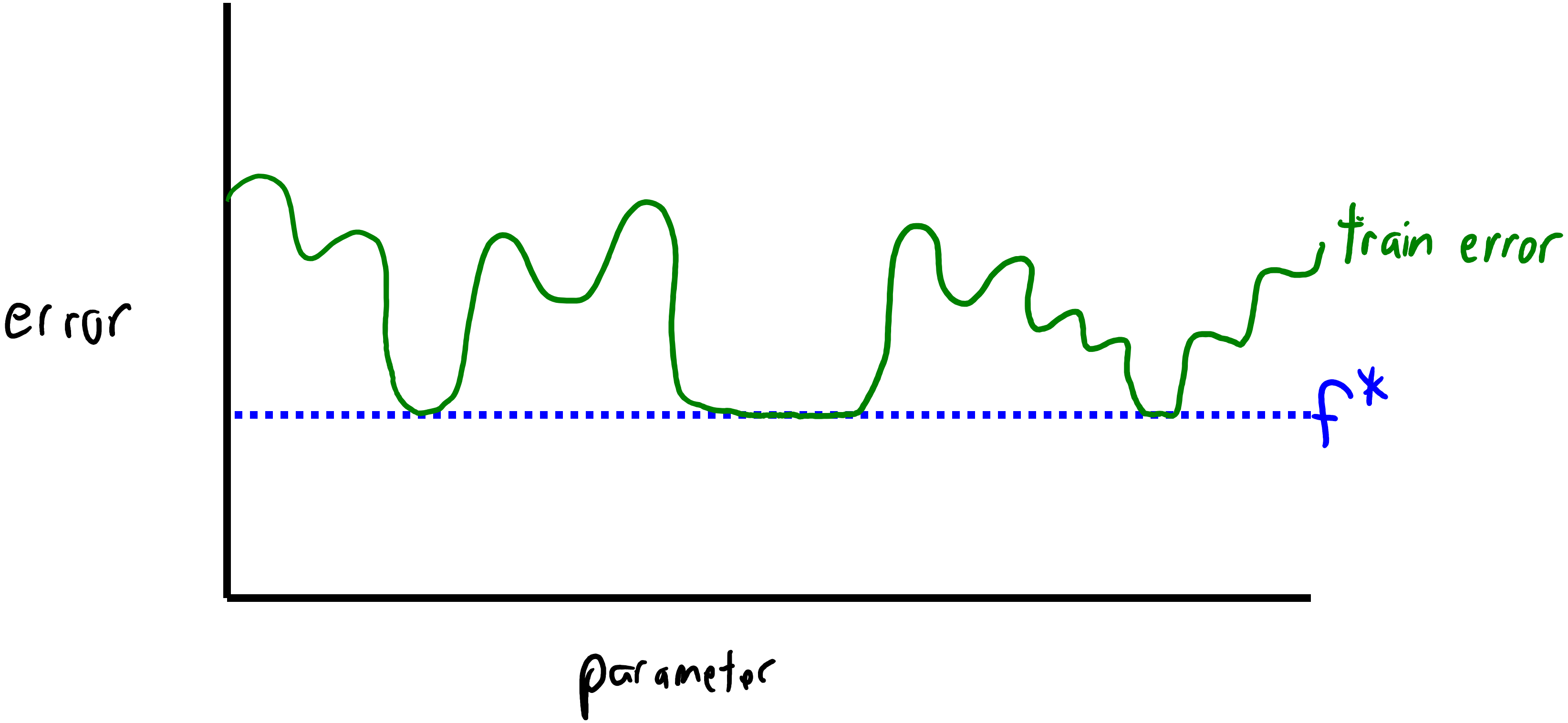


- Test error continues to go down!?!
  - What happened to “more complex models overfit more”???
- There do exist global mins with large #hidden units have test error = 1.
  - But among the global minima, SGD is somehow converging to “good” ones.

https://www.neyshabur.net/papers/inductive_bias_poster.pdf

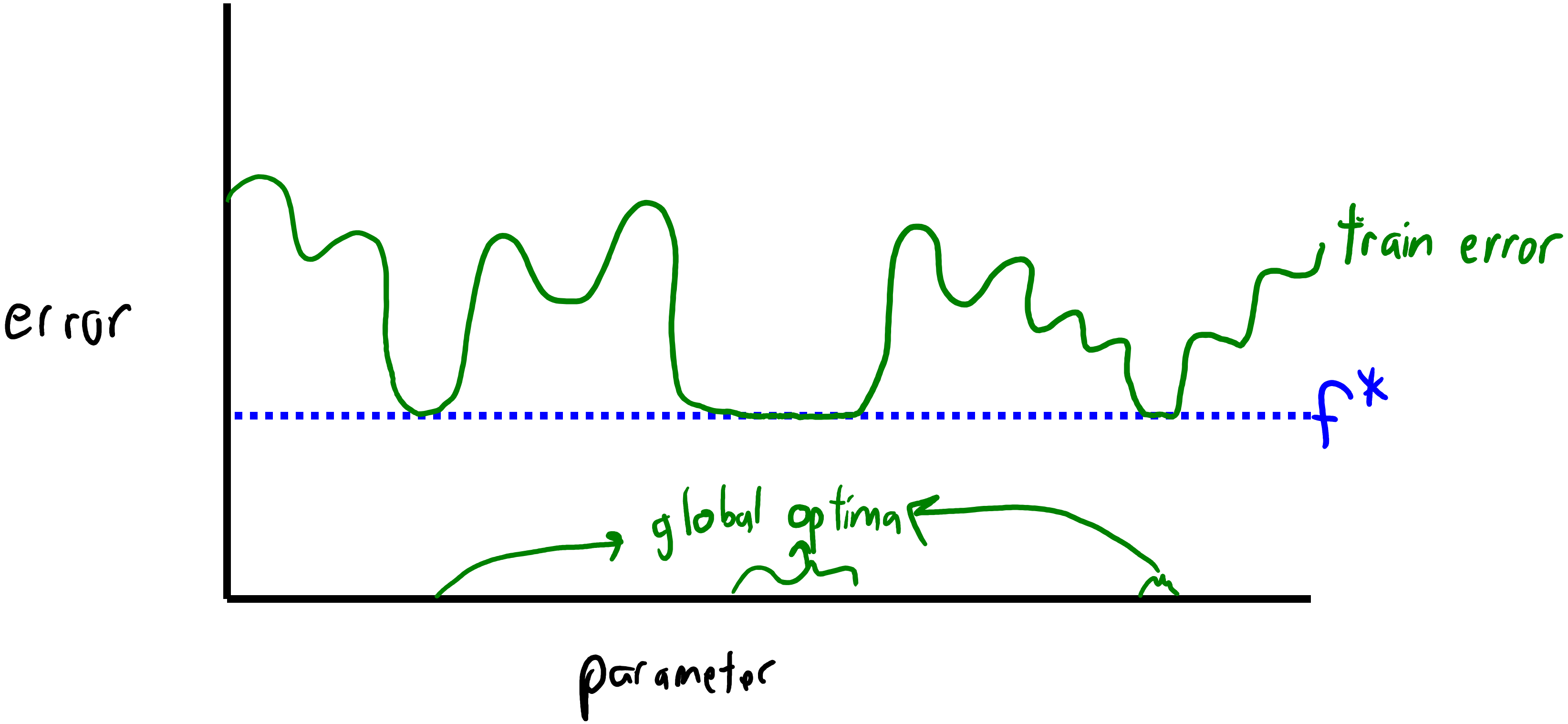
Multiple Global Minima?

- For standard objectives, there is a global min function value f*:



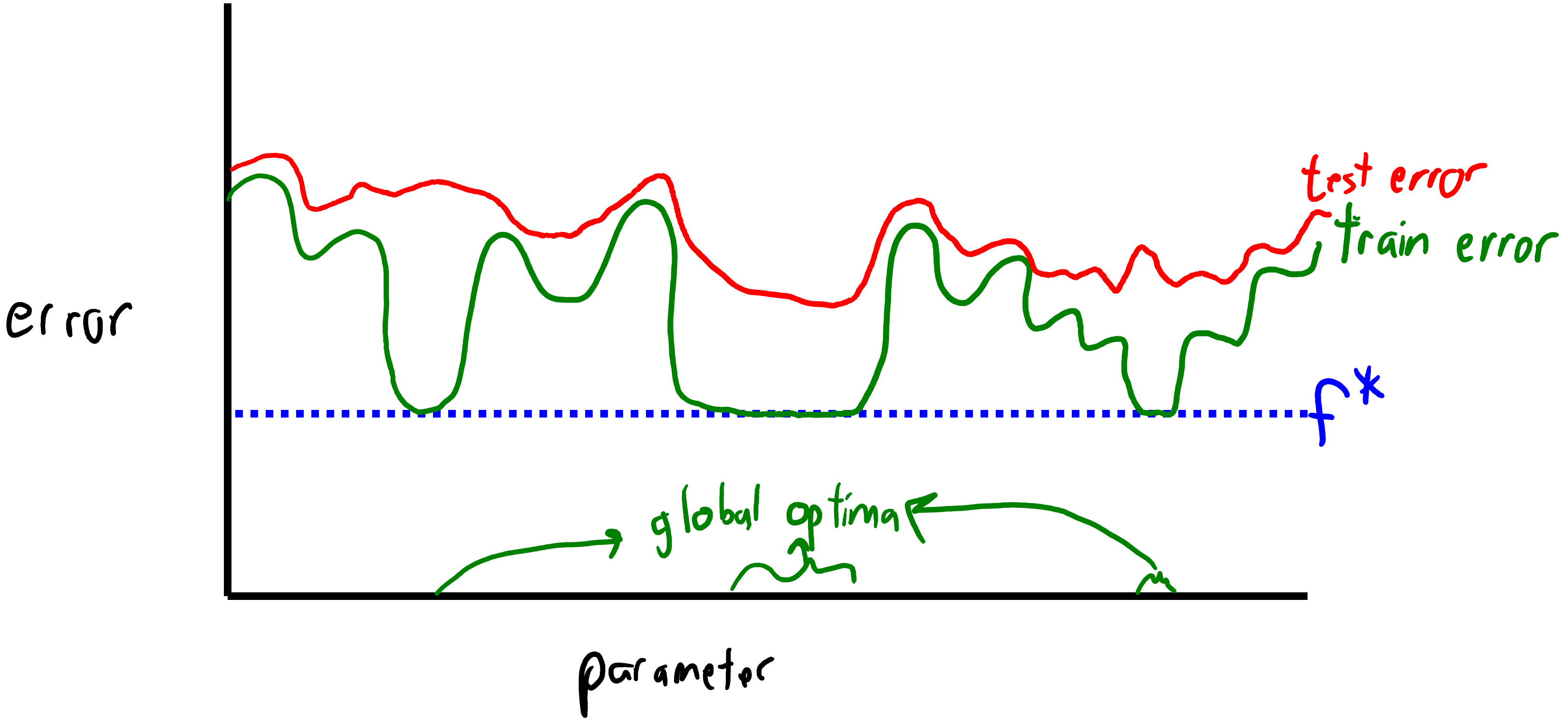
Multiple Global Minima?

- For standard objectives, there is a global min function value f*:



- But this may be achieved by many different parameter values.

Multiple Global Minima?



- These training-error global minima may have very different test errors.
- Some of these global minima may be “more regularized” than others.

bonus!

Implicit Regularization of (S)GD

- There is empirical evidence that using SGD regularizes parameters.
  - We call this the “implicit regularization” of the optimization algorithm.
- Beyond empirical evidence, we know this happens in simpler cases.
- Example of implicit regularization:
  - Consider a least squares problem where there exists a w where $\mathbf{X}w=y$.
    - Residuals are all zero, we fit the data exactly.
    - If $d > n$, there are infinitely many exact solutions.
  - You run [stochastic] gradient descent starting from $w=0$, small learning rate
  - Converges to the solution $\mathbf{X}w=y$ that has the minimum L2-norm.
    - Using (S)GD is equivalent to (infinitesimal) L2-regularization here; regularization is “implicit”.
    - Using `w = np.linalg.solve(X, y)` gives you this same solution.

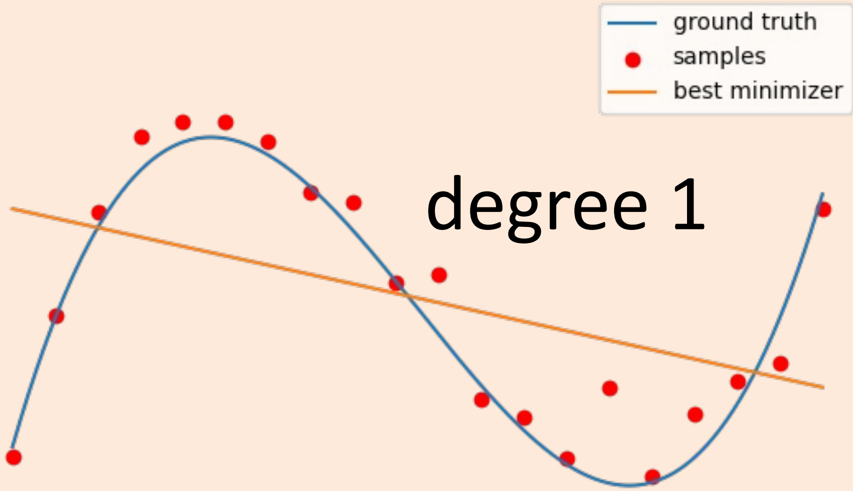
bonus!

Implicit Reg. of (S)GD for linear regression: Proof sketch

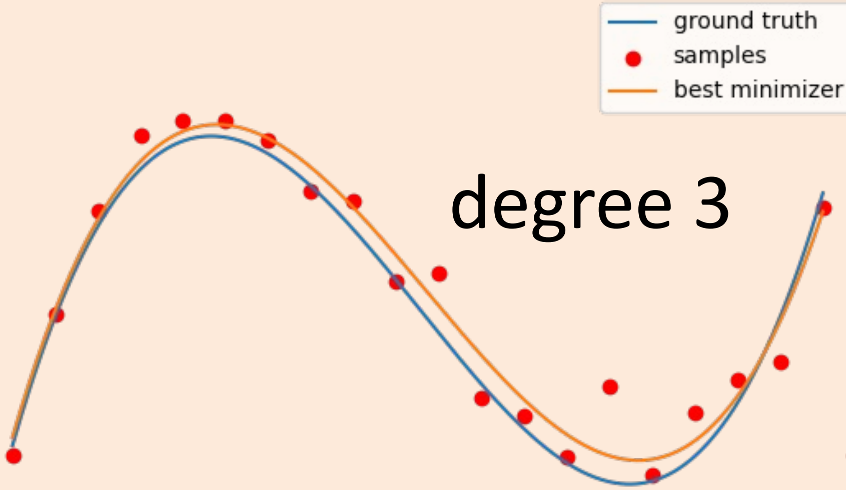
- Assume that (stochastic) gradient descent converges to a local min
  - It's a smooth, convex optimization problem
- When "interpolation" is possible (e.g. *d* > *n*), implies that $\boldsymbol{X} w_\infty = \boldsymbol{y}$
- During training, gradient looks like $\nabla f = \sum_i 2\left(w^T x^{(i)} - y^{(i)}\right) x^{(i)}$
  - (S)GD only ever moves in the span of the data
- Must have $w_\infty = w_0 + \boldsymbol{X}^T \alpha$ for some α
- Combining: $\boldsymbol{X}\left(w_0 + \boldsymbol{X}^T \alpha\right) = \boldsymbol{y}$; $\alpha = \left(\boldsymbol{X}\boldsymbol{X}^T\right)^{-1} (\boldsymbol{y} - \boldsymbol{X} w_0)$
  - $w_\infty = \boldsymbol{X}^T\left(\boldsymbol{X}\boldsymbol{X}^T\right)^{-1} (\boldsymbol{y} - \boldsymbol{X} w_0) + w_0 = \boldsymbol{X}^T\left(\boldsymbol{X}\boldsymbol{X}^T\right)^{-1} \boldsymbol{y} + \left(I - \boldsymbol{X}^T\left(\boldsymbol{X}\boldsymbol{X}^T\right)^{-1} \boldsymbol{X}\right) w_0$
  - "Closest interpolator to w_0"; when $w_0 = 0$, the minimum-norm interpolator

“Classical” regime: one global min

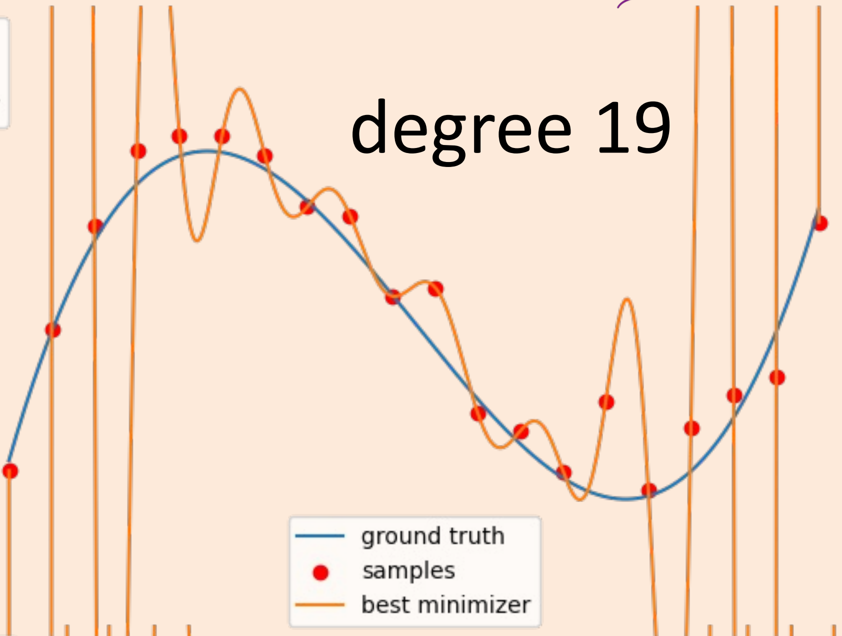
bonus!



degree 1

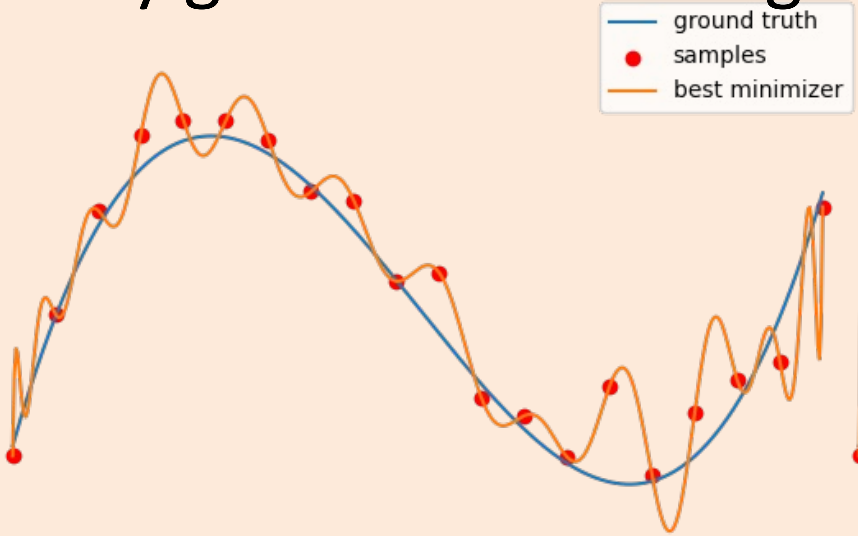


degree 3

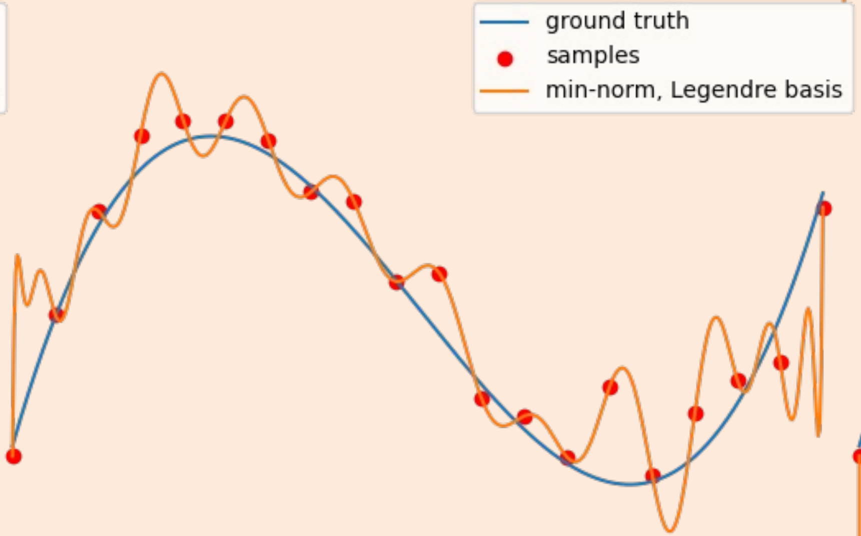


degree 19

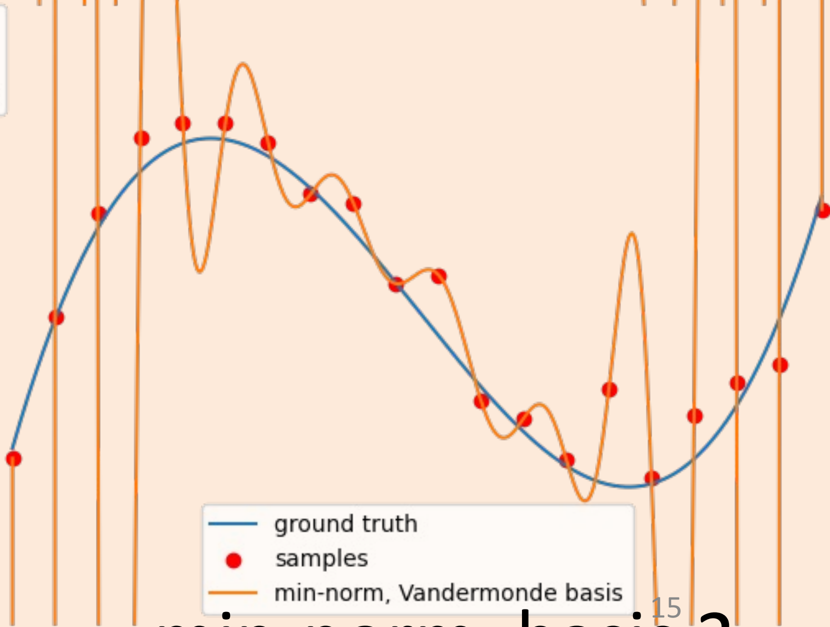
Many global minima: degree 25



“best”



min-norm, basis 1

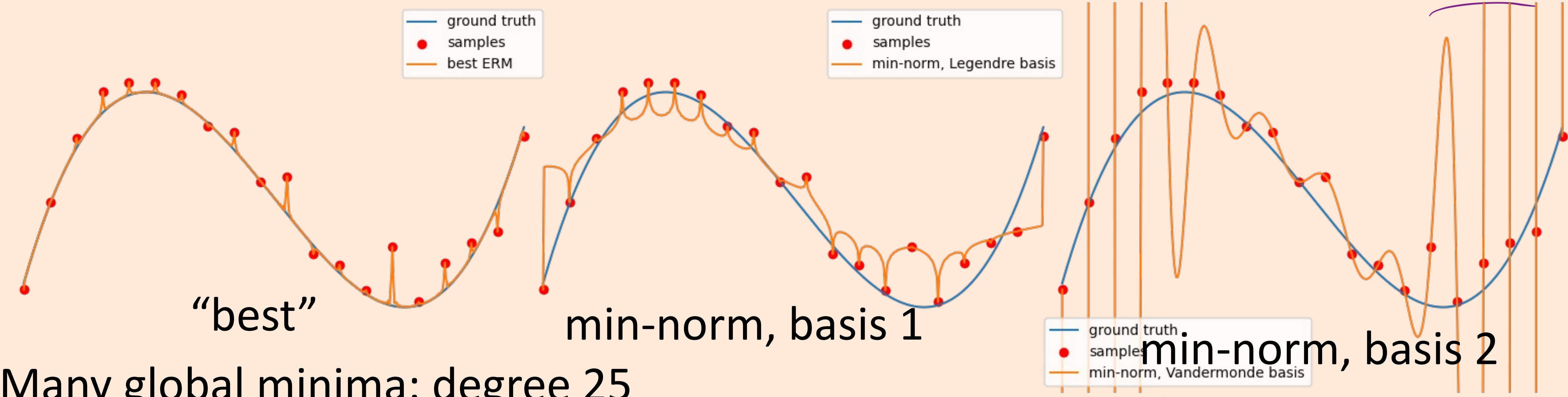


min-norm, basis 2

Nakkiran et al. blog post's companion notebook

Extreme overparameterization: degree 1000

bonus!

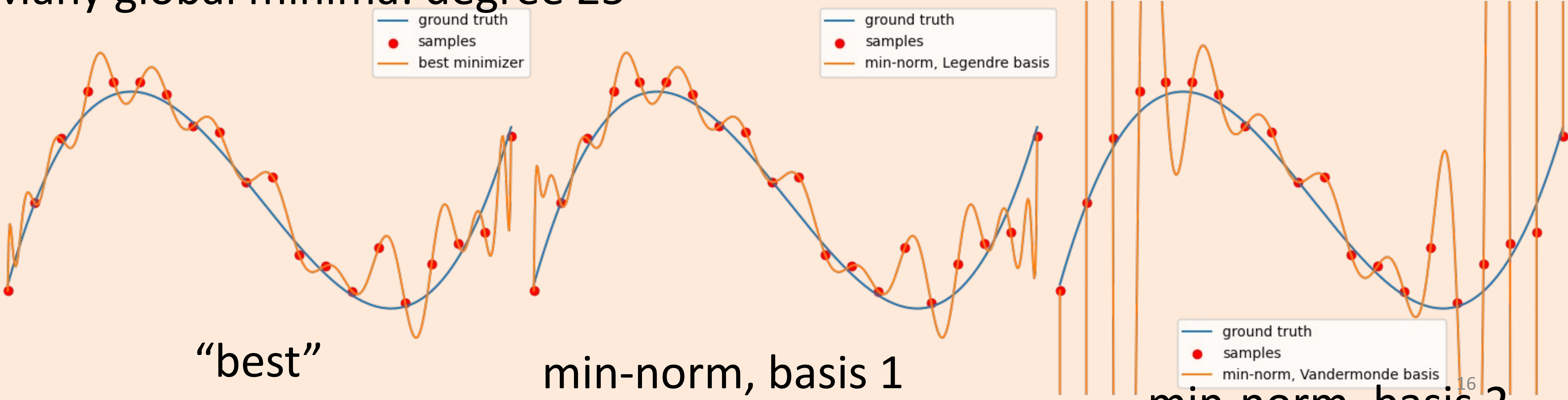


“best”

min-norm, basis 1

min-norm, basis 2

Many global minima: degree 25



“best”

min-norm, basis 1

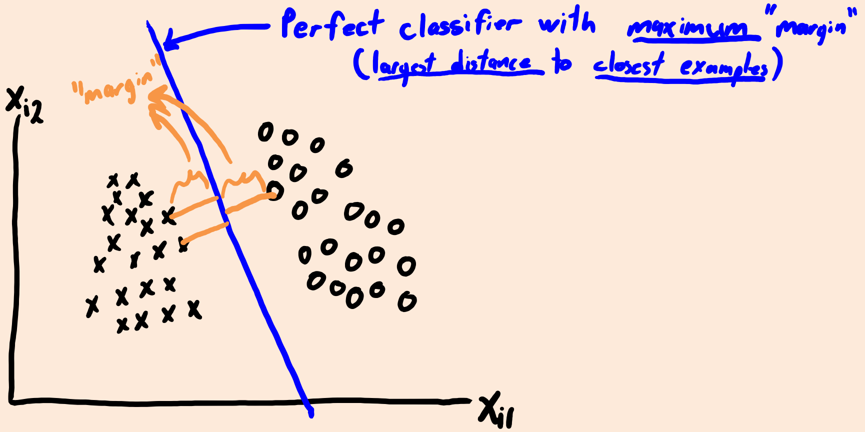
min-norm, basis 2

Nakkiran et al. blog post's companion notebook

bonus!

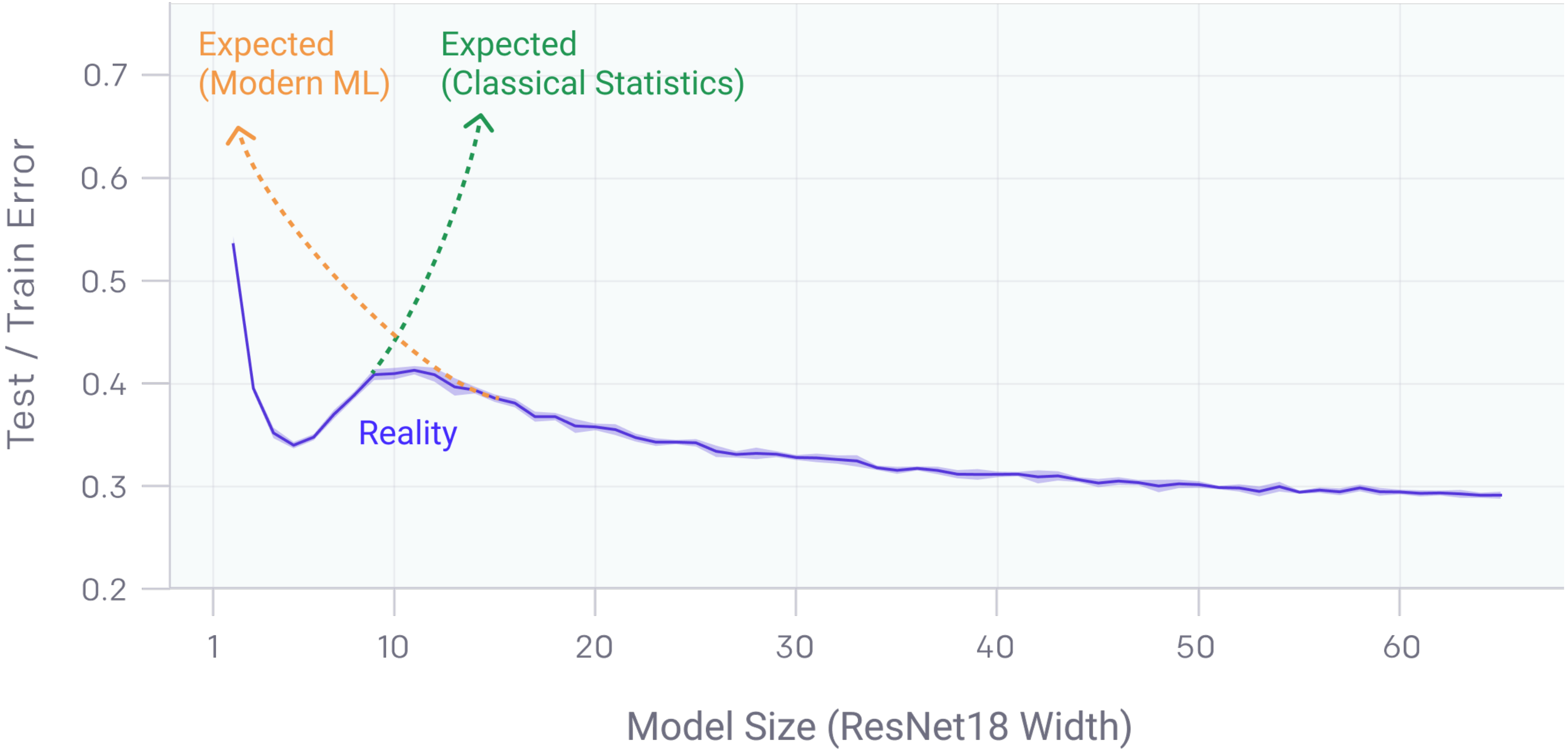
Implicit Regularization of (S)GD

- Example of implicit regularization:
  - Consider a logistic regression problem where data is linearly separable.
    - A linear model can perfectly separate the data.
  - You run gradient descent from any starting point.
  - Converges to max-margin solution of the problem (minimum L2-norm solution; SVM).
    - So using gradient descent is equivalent to encouraging large margin.


Perfect classifier with maximum "margin"
(largest distance to closest examples)
"margin"
x_{i2}
x_{i1}

- Related results are known for (some cases of) non-separable logistic regression, non-separable SVMs, boosting, matrix factorization, deep homogeneous nets…

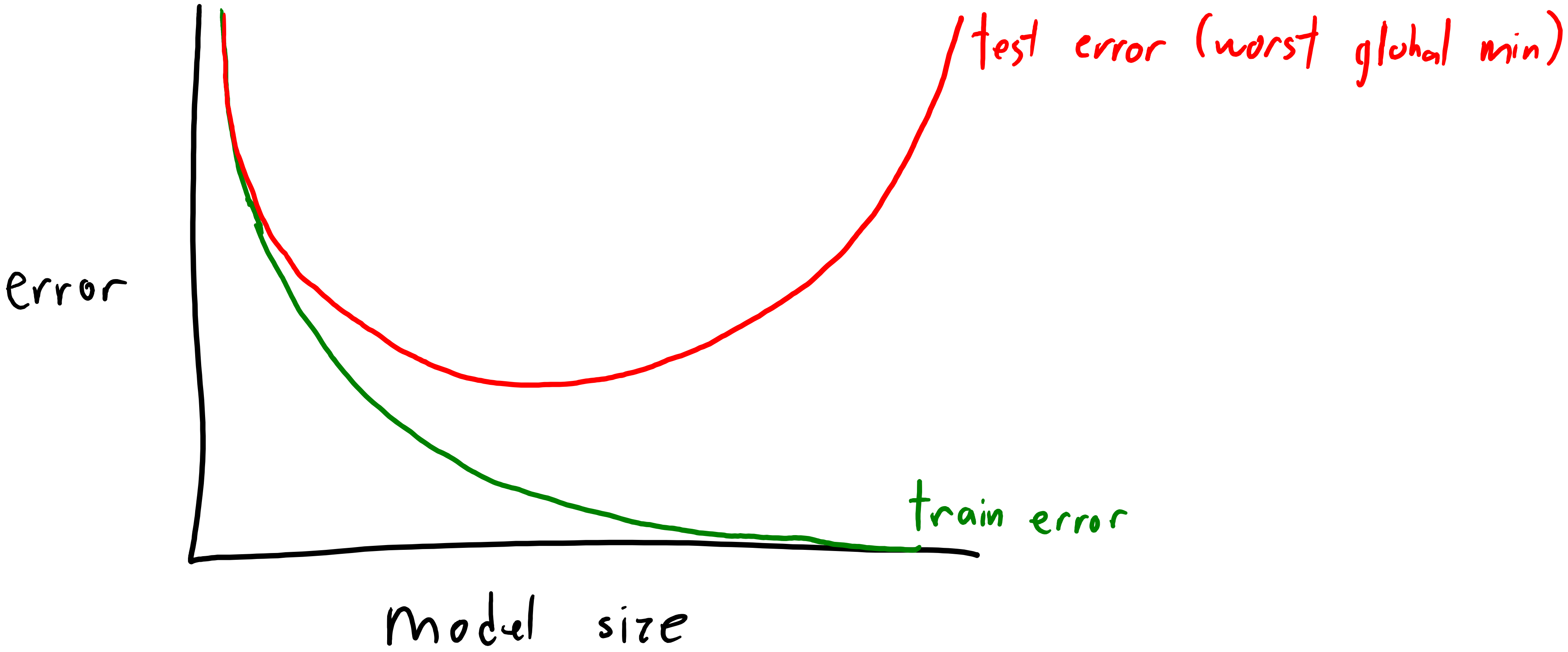
Double Descent Curves



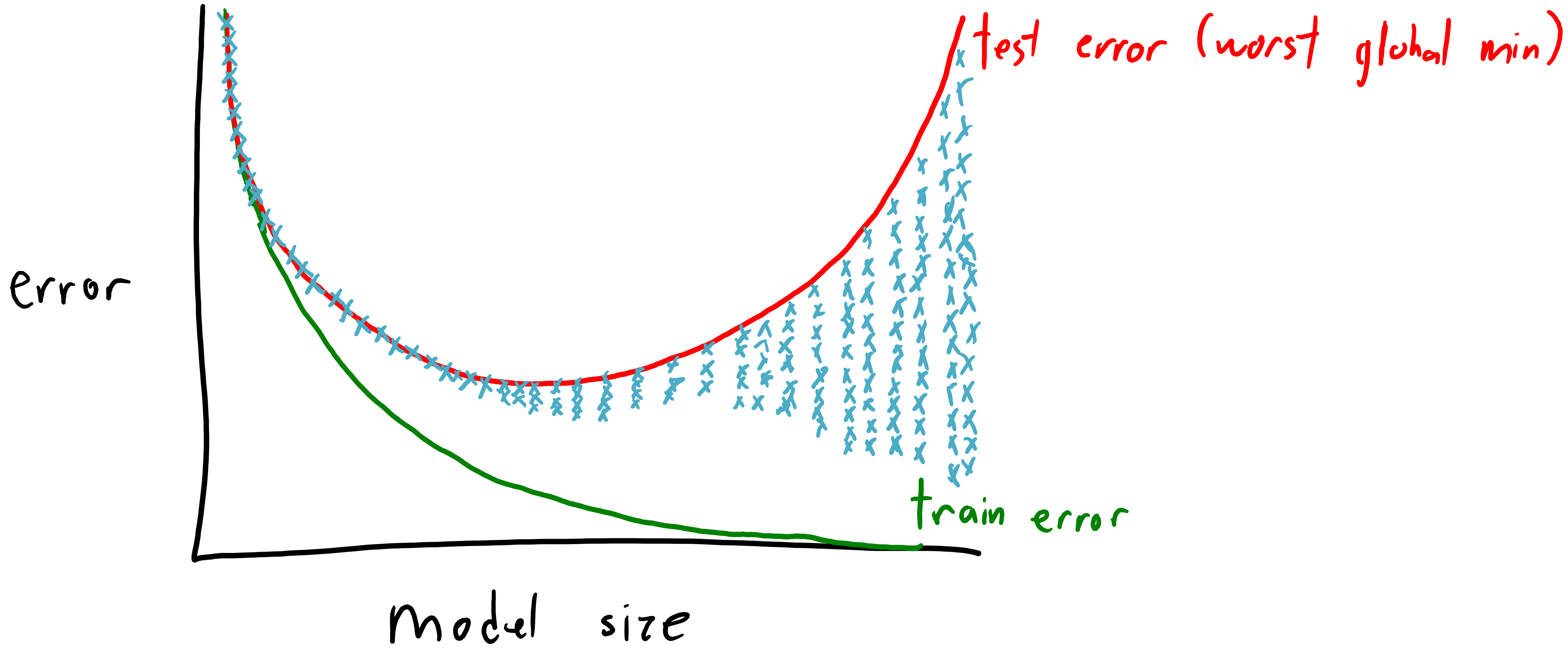
- What is going on???

https://openai.com/blog/deep-double-descent/

Worst vs. Best "Global Minimum"

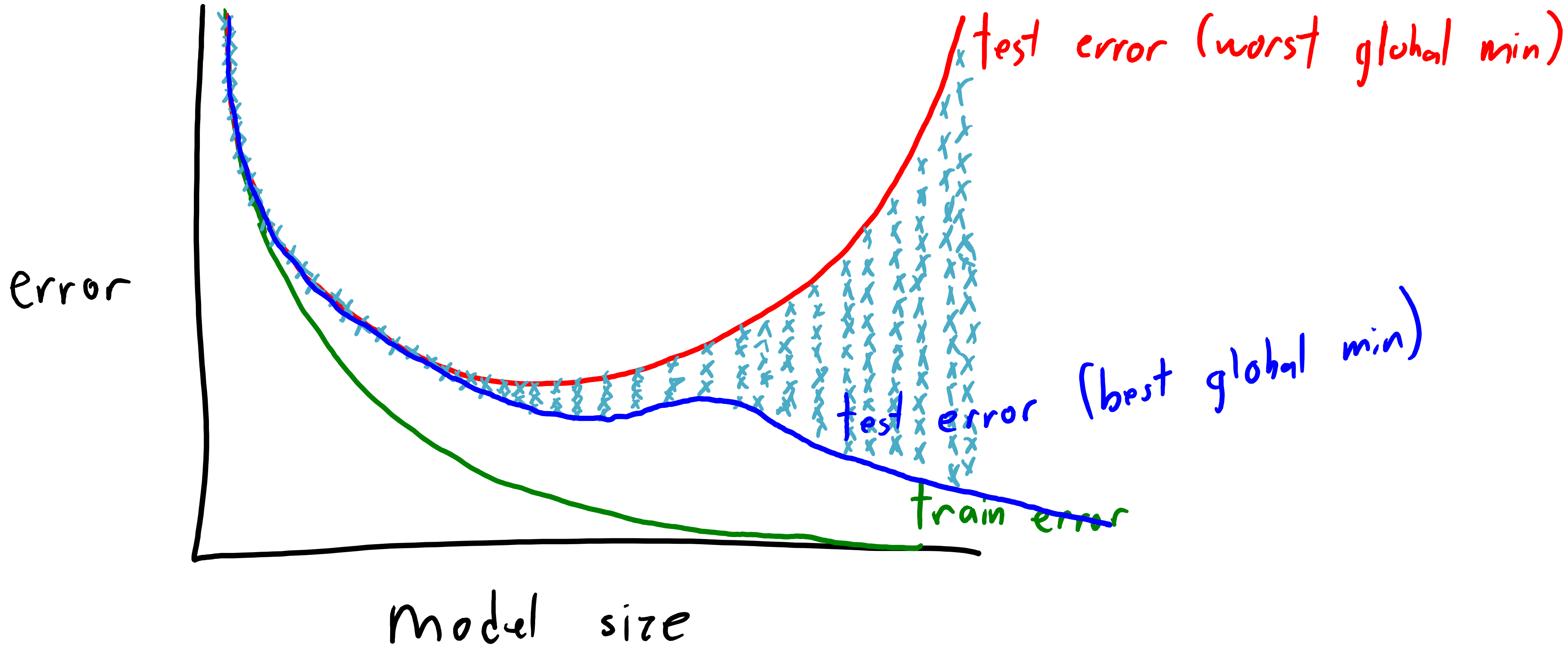


Worst vs. Best “Global Minimum”



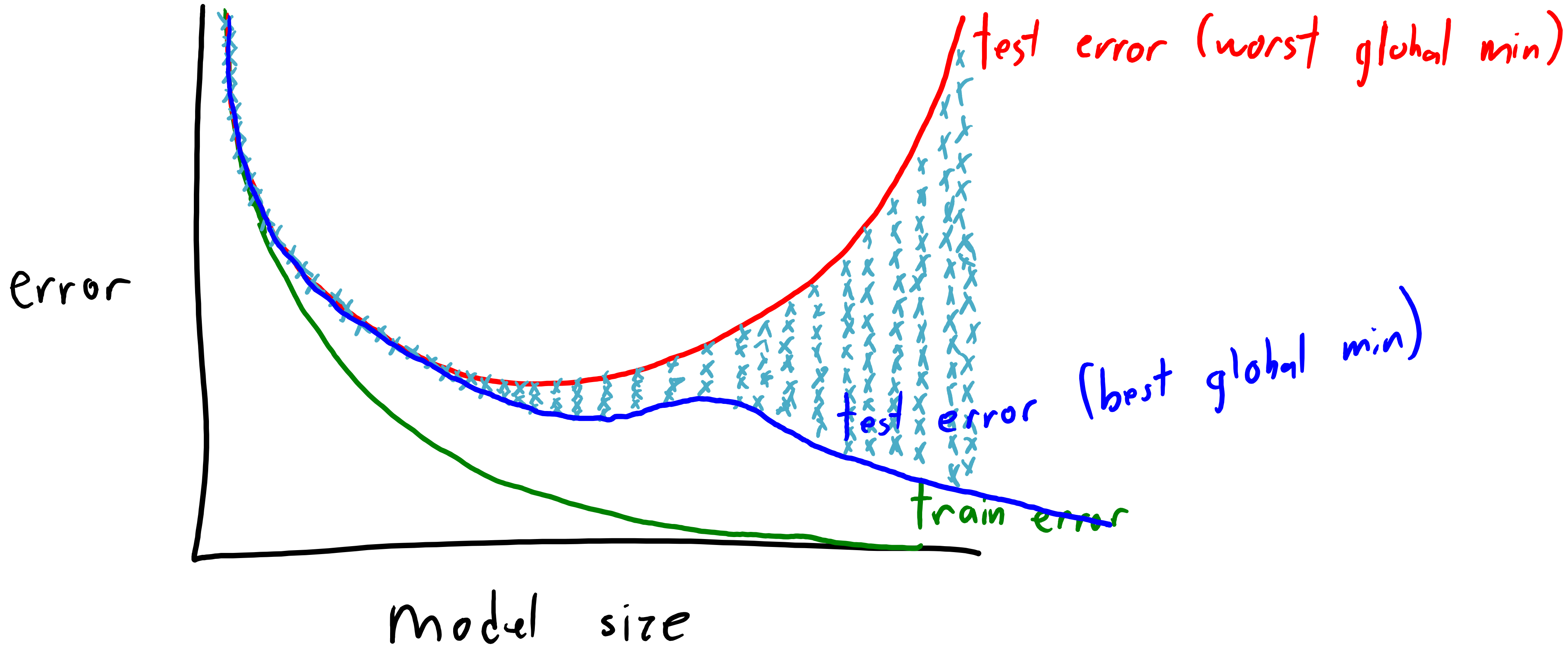
- Learning theorists usually analyze the *worst* global min (in test error)
  - Actual test error for many global minima may be better than worst case bound
  - Theory is correct, but maybe “worst overfitting possible” is too pessimistic?

Worst vs. Best "Global Minimum"



- Think about instead the global min with best test error
  - With small models, "minimize training error" leads to unique (or similar) global mins
  - With larger models, there is a lot of flexibility in the space of global mins (gap between best/worst)
- Gap between "worst" and "best" global min can grow with model complexity

Worst vs. Best “Global Minimum”

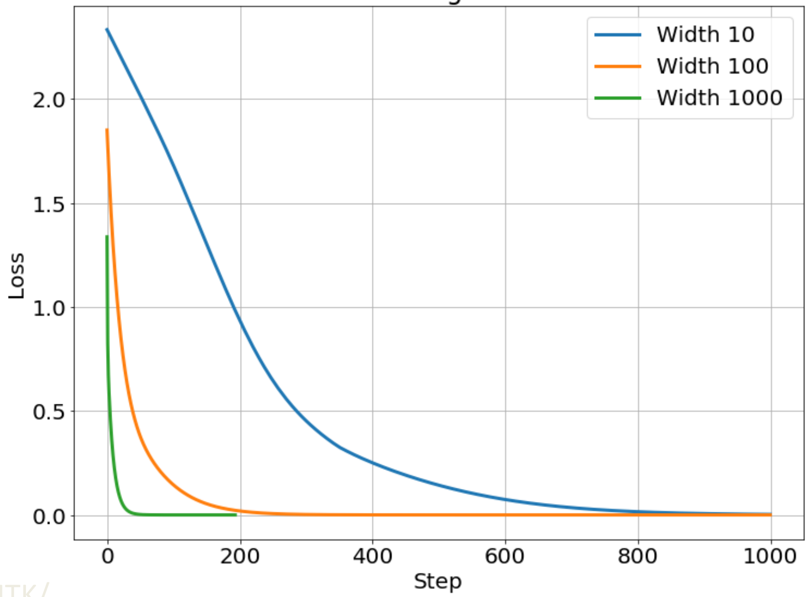


- Can get “double descent” curve in practice if parameters roughly track “best” global min shape
  - One way (ish) to do this: increase regularization as you increase model size
- Maybe “neural network trained with SGD” has “more implicit regularization for bigger models”?
  - But this behavior is not specific to implicit regularization of SGD and not specific to neural networks.

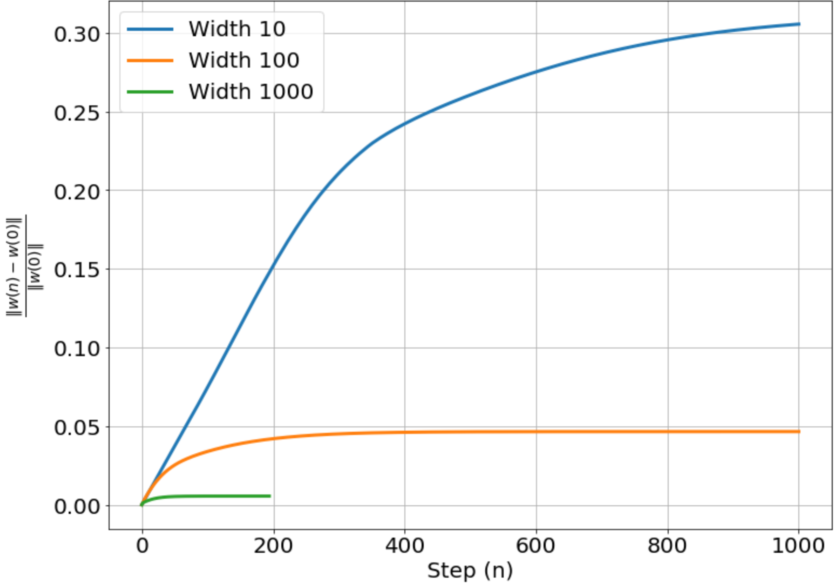
Implicit Regularization of SGD (as function of size)

- Why would implicit regularization of SGD increase with dimension?
  - Maybe SGD finds low-norm solutions?
    - In higher dimensions, might be more flexibility in global mins to have a low norm
  - Maybe SGD stays closer to starting point as we increase dimension?
    - This would be more like a regularizer of the form $||w - w^0||$.

Training loss



Relative change in norm of weights from initialization

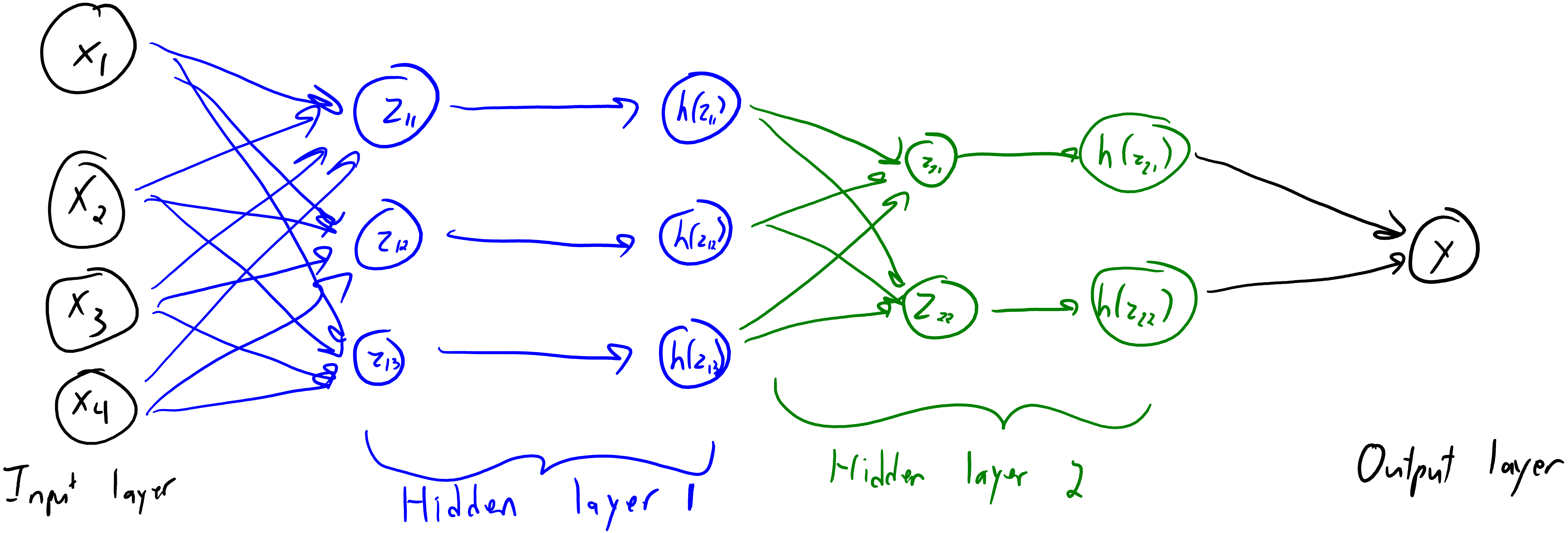


https://rajatvd.github.io/NTK/

Next Topic: Deep Learning

Deep Learning

- Deep learning models have more than one hidden layer:

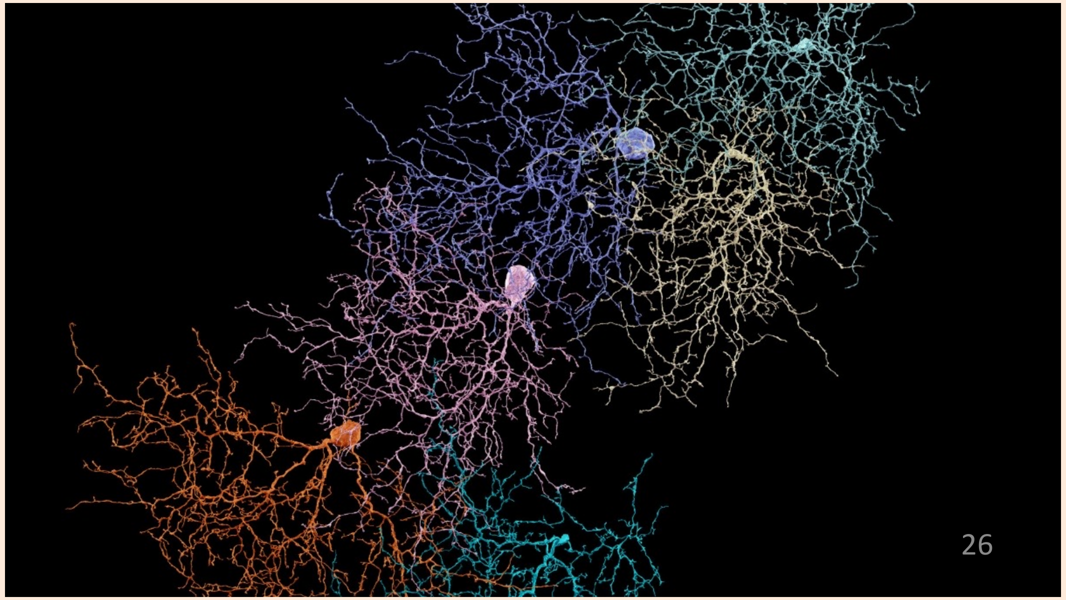


- We transform our activations one or more times.

bonus!

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.

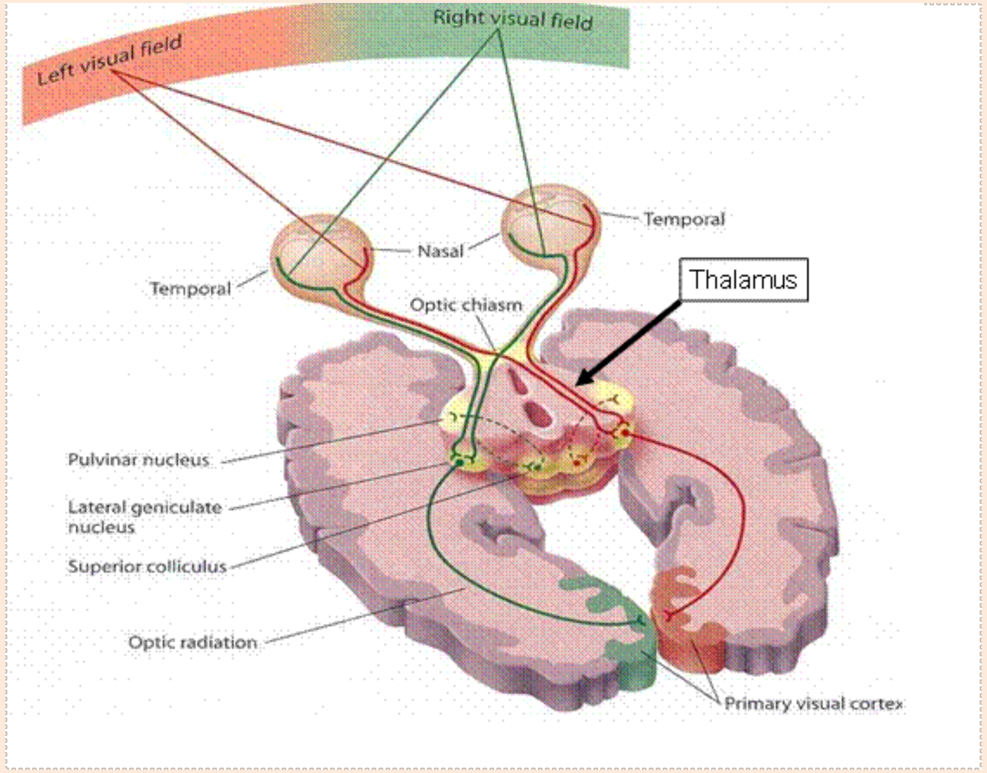


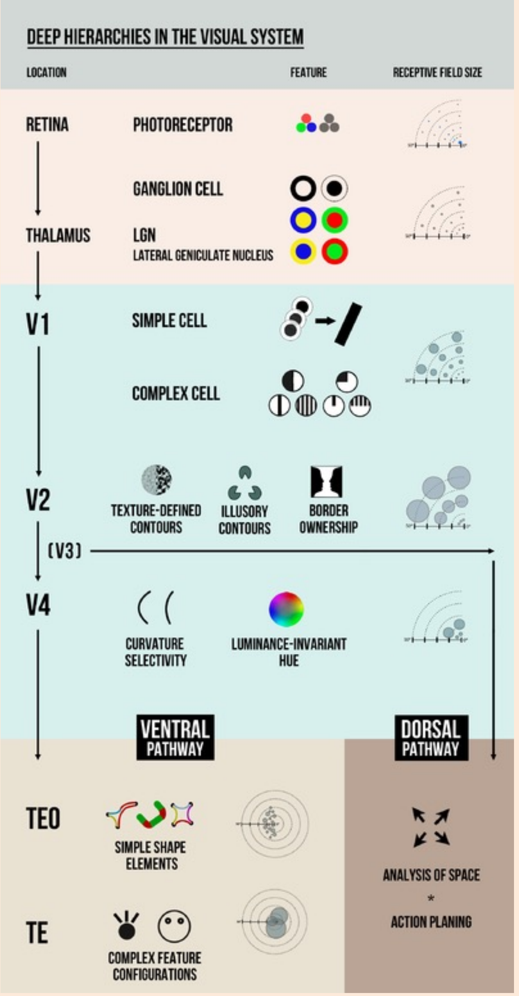
https://www.nytimes.com/2015/01/11/magazine/sebastian-seungs-quest-to-map-the-human-brain.html

bonus!

Why Multiple Layers?

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





DEEP HIERARCHIES IN THE VISUAL SYSTEM

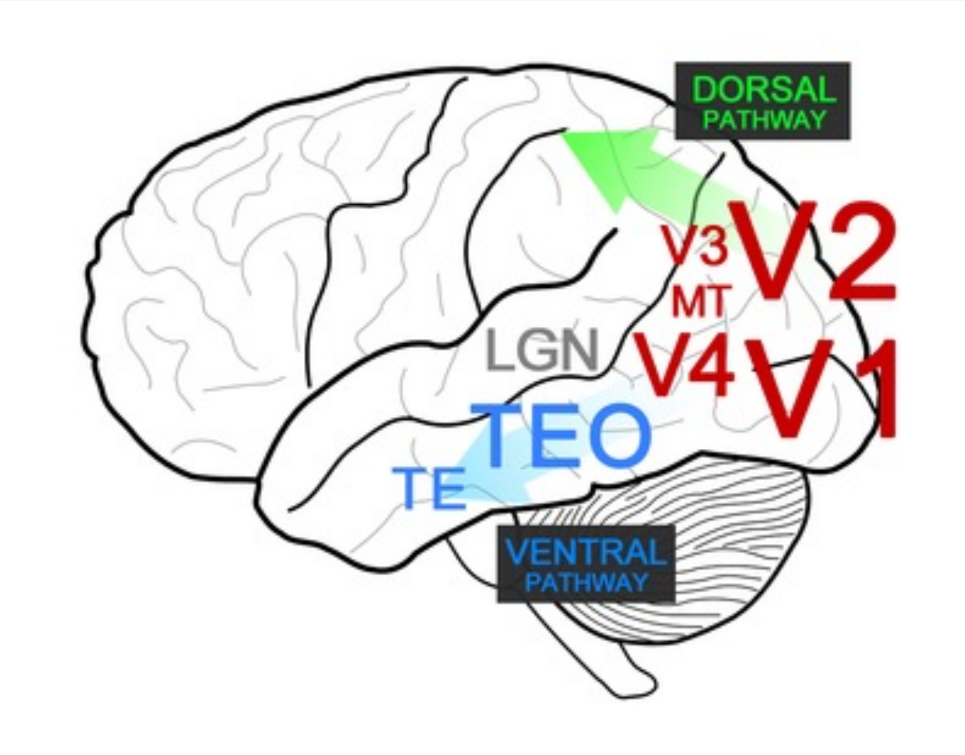
LOCATION	FEATURE	RECEPTIVE FIELD SIZE
RETINA	PHOTORECEPTOR	
	GANGLION CELL	
THALAMUS	LGN LATERAL GENICULATE NUCLEUS	
V1	SIMPLE CELL	
	COMPLEX CELL	
V2	TEXTURE-DEFINED CONTOURS ILLUSORY CONTOURS BORDER OWNERSHIP	
(V3)		
V4	CURVATURE SELECTIVITY LUMINANCE-INVARIANT HUE	

VENTRAL PATHWAY

TEO	SIMPLE SHAPE ELEMENTS	
TE	COMPLEX FEATURE CONFIGURATIONS	

DORSAL PATHWAY

ANALYSIS OF SPACE
*
ACTION PLANING

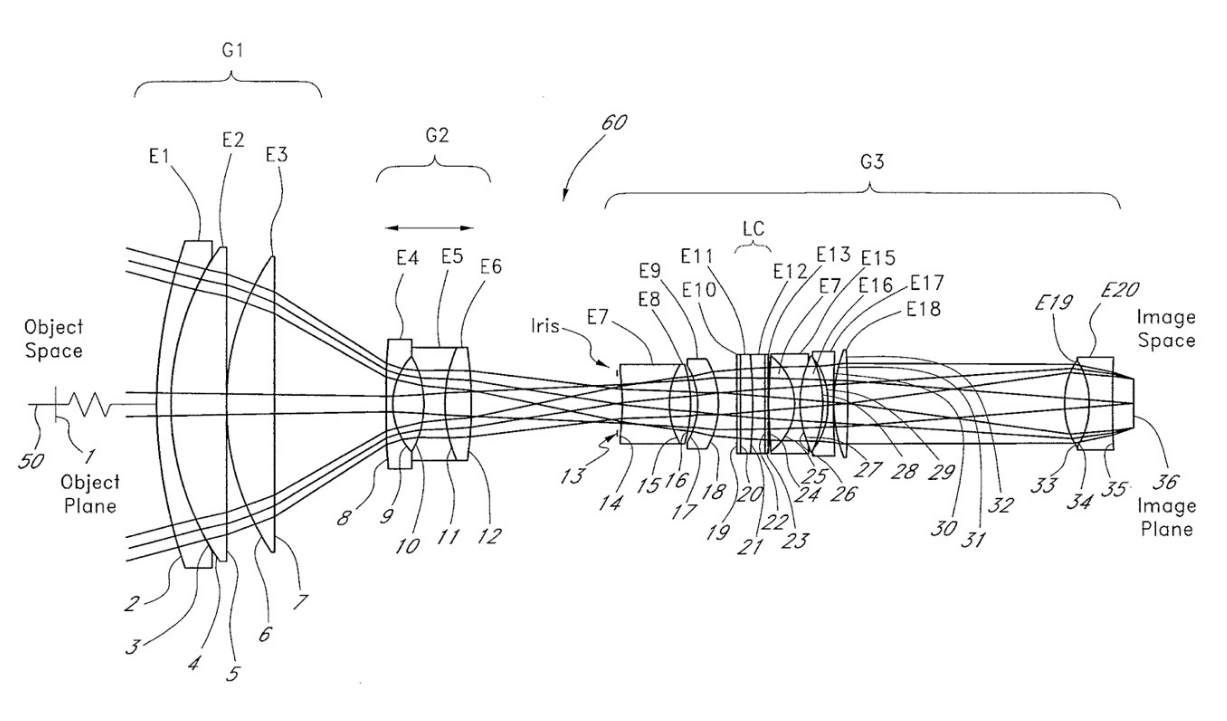


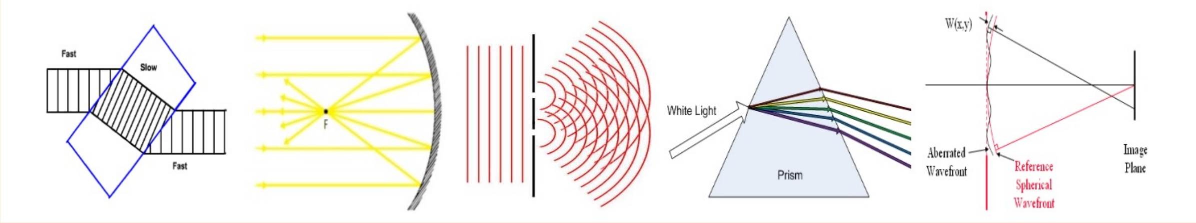
http://www.strokenetwork.org/newsletter/articles/vision.htm
https://en.wikibooks.org/wiki/Sensory_Systems/Visual_Signal_Processing

bonus!

Why Multiple Layers?

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



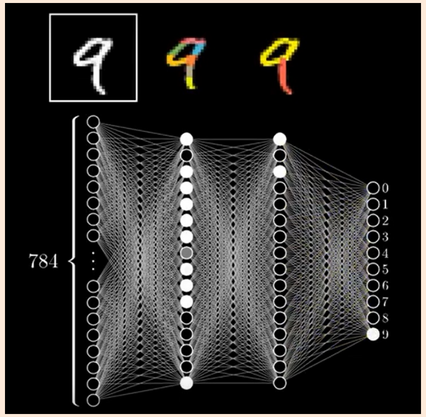


http://www.argmin.net/2018/01/25/optics/

bonus!

Why Multiple Layers?

- There are also mathematical motivations for using multiple layers:
  - 1 layer gives us a universal approximator of any (reasonable) function.
    - 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.
- Relevant video:
  - https://www.youtube.com/watch?v=aircAruvnKk

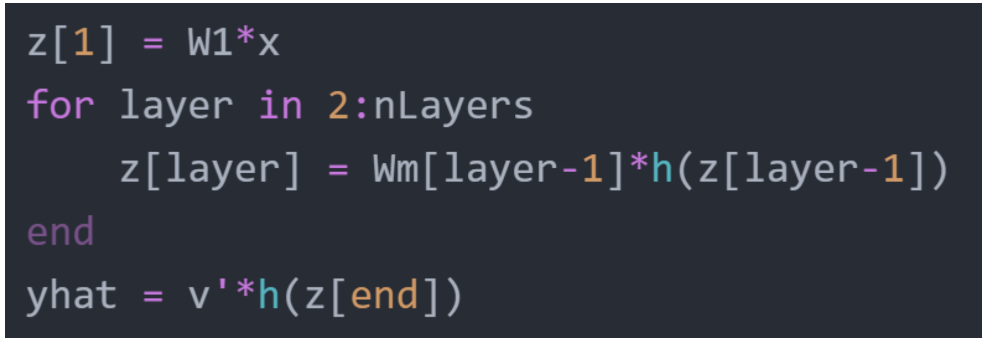


Inference In Deep Neural Networks

- The “textbook” choice for deep neural networks:
  - Alternate between doing linear transformations and non-linear transforms.

$$\hat{y} = v^T h(W^4 h(W^3 h(W^2 h(W^1 x))))$$

  - Each “layer” might have a different size.
    - W^1 is k^1 x d.
    - W^2 is k^2 x k^1.
    - W^3 is k^3 x k^2.
    - W^4 is k^4 x k^3.
    - v is k^4 x 1.

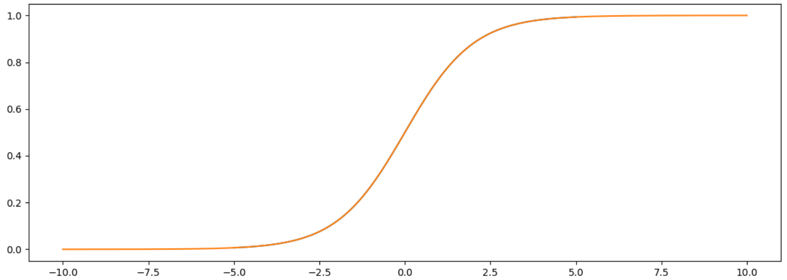


```
z[1] = W1*x
for layer in 2:nLayers
    z[layer] = Wm[layer-1]*h(z[layer-1])
end
yhat = v'*h(z[end])
```

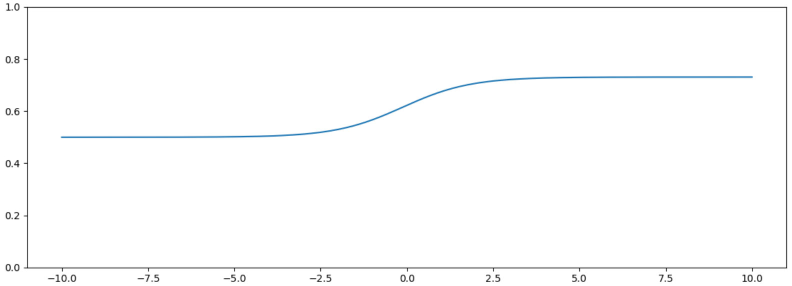
  - We use the same non-linear transform, such as sigmoid, at each layer.
  - Cost for prediction, which is called “forward propagation”:
    - Cost of the matrix multiplies: $O(k^1 d + k^2 k^1 + k^3 k^2 + k^4 k^3)$
    - Cost of the non-linear transforms is $O(k^1 + k^2 + k^3 + k^4)$, so does not change cost.
  - Once you have $\hat{y}$, inference works as it does for Bernoulli with $\theta = 1/(1+\exp(-\hat{y}))$.

New Issue: Vanishing Gradients

- Consider the sigmoid function:



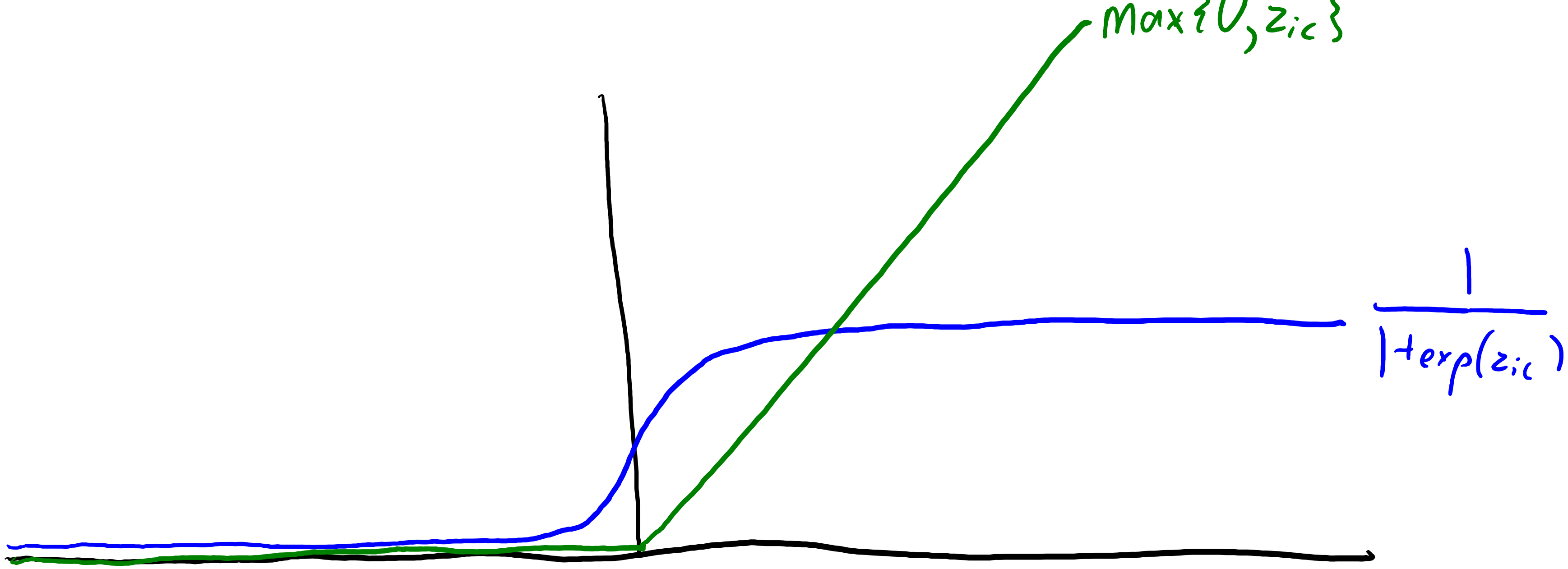
- Away from the origin, the gradient is nearly zero.
- The problem gets worse when you take the sigmoid of a sigmoid:



- In deep networks, many gradients can be nearly zero everywhere.
  - And numerically they will be set to 0.

Rectified Linear Units (ReLU)

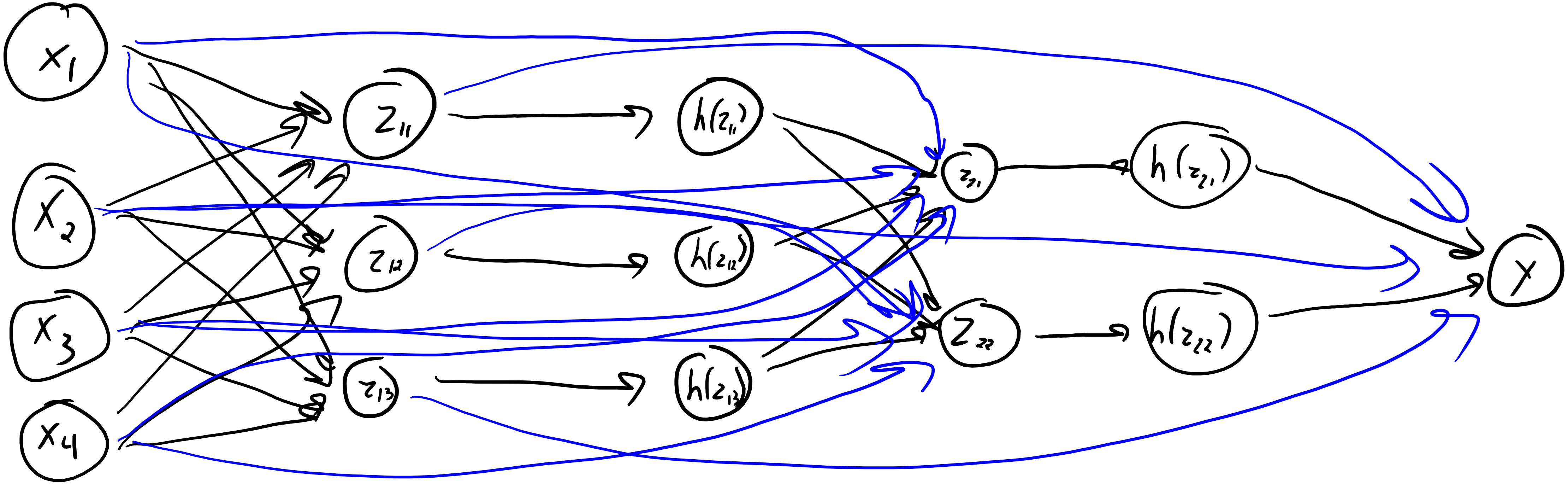
- Modern networks almost always replace sigmoid with ReLUs:



- 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 in Deep Learning

- Skip connections can also reduce vanishing gradient problem:



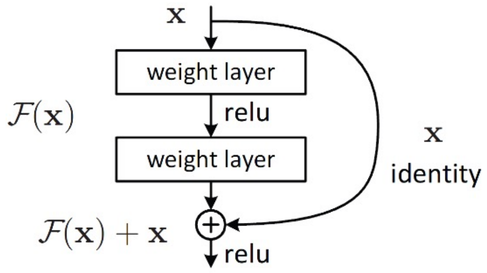
- Makes "shortcuts" from input to output with fewer transformations.
  - Many variations exist on skip connections locations and how they are used.

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:

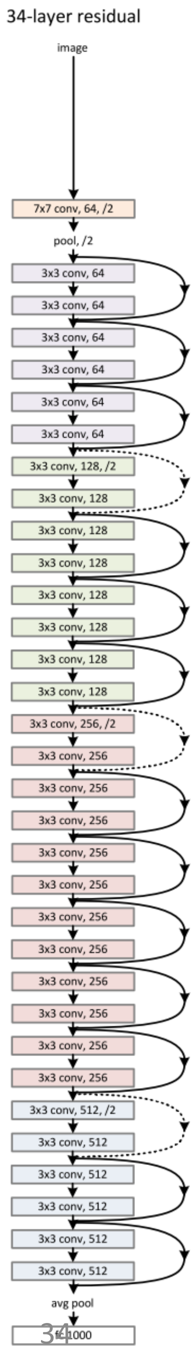
$$a^{l+2} = h(W^{l+1} h(W^l \underset{\uparrow \text{"activation at layer 'l'"}}{a^l}))$$

- ResNet “block”: $a^{l+2} = h(a^l + W^{l+1} h(W^l a^l))$
  - Adds activations from “2 layers ago”.



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

- 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

bonus!

DenseNet

- Another variation 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.

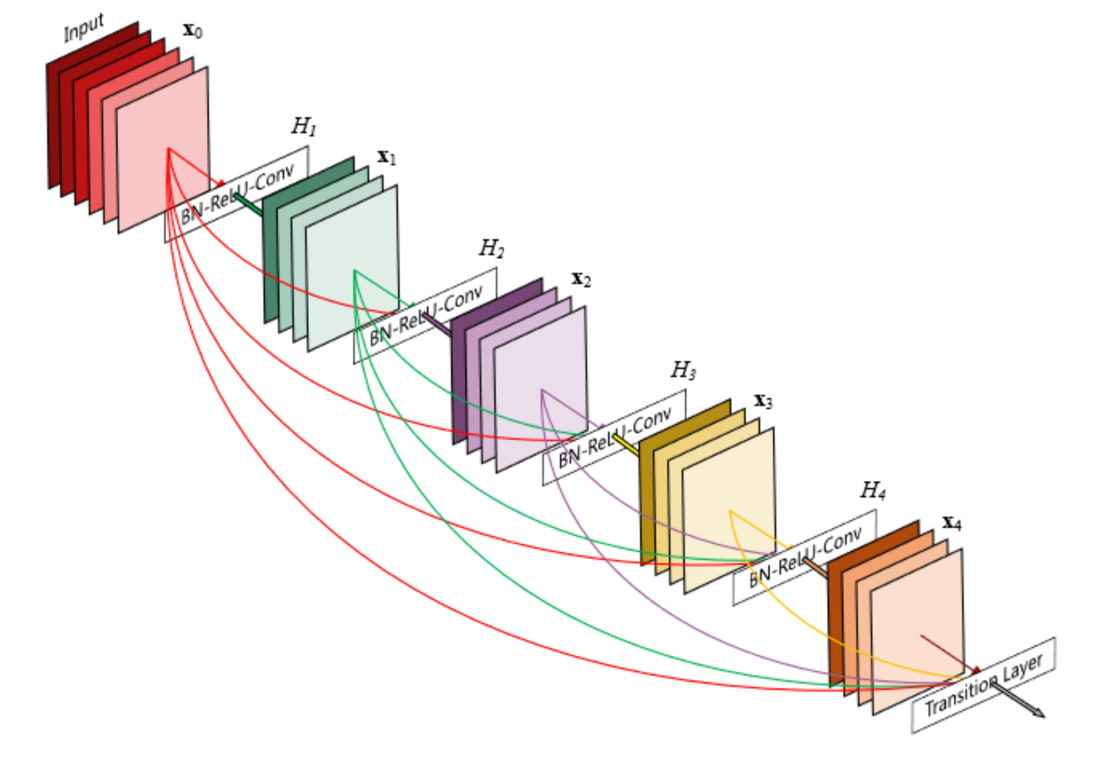


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

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

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.
  - But we are discovering sets of tricks that often make things easier to tune.
    - Data standardization (“centering” and “whitening”).
    - Adding bias variables.
    - Parameter initialization: “small but different", standardizing within layers.
    - Step-size selection: “babysitting", Bottou trick.
    - Momentum: heavy-ball and Nesterov-style modfications.
    - Step size for each coordinate: AdaGrad, RMSprop, Adam.
    - Rectified linear units (ReLU): replace sigmoid with max{0,h} to avoid gradients close to 0.
      - Makes objective non-differentiable, but we now know SGD still converges in this setting.
    - Batch normalization: adaptive standardizing within layers.
      - Often allows sigmoid activations in deep networks.
    - 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 through the space of hyper-parameters.
      - This gets expensive!

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.
- Popular Examples:
  - Batch normalization originally proposed to fix “internal covariate shift”.
    - Internal covariate shift not really defined in original paper, batch norm does seem to reduce it.
      - Famously singled out as an example of “alchemy” in ML research.
    - 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 combination of ideas from several existing algorithms.
    - Such as “momentum” and “AdaGrad”, both of which are well-understood theoretically.
      - Theory in the original paper was incorrect; Adam fails at solving some very-simple optimization problems.
    - But is Adam is often used because it is amazing at training some networks.
      - It’s been hypothesized that we “converged” towards networks that are easier for current SGD methods like Adam.

Regularization in Deep Neural Networks

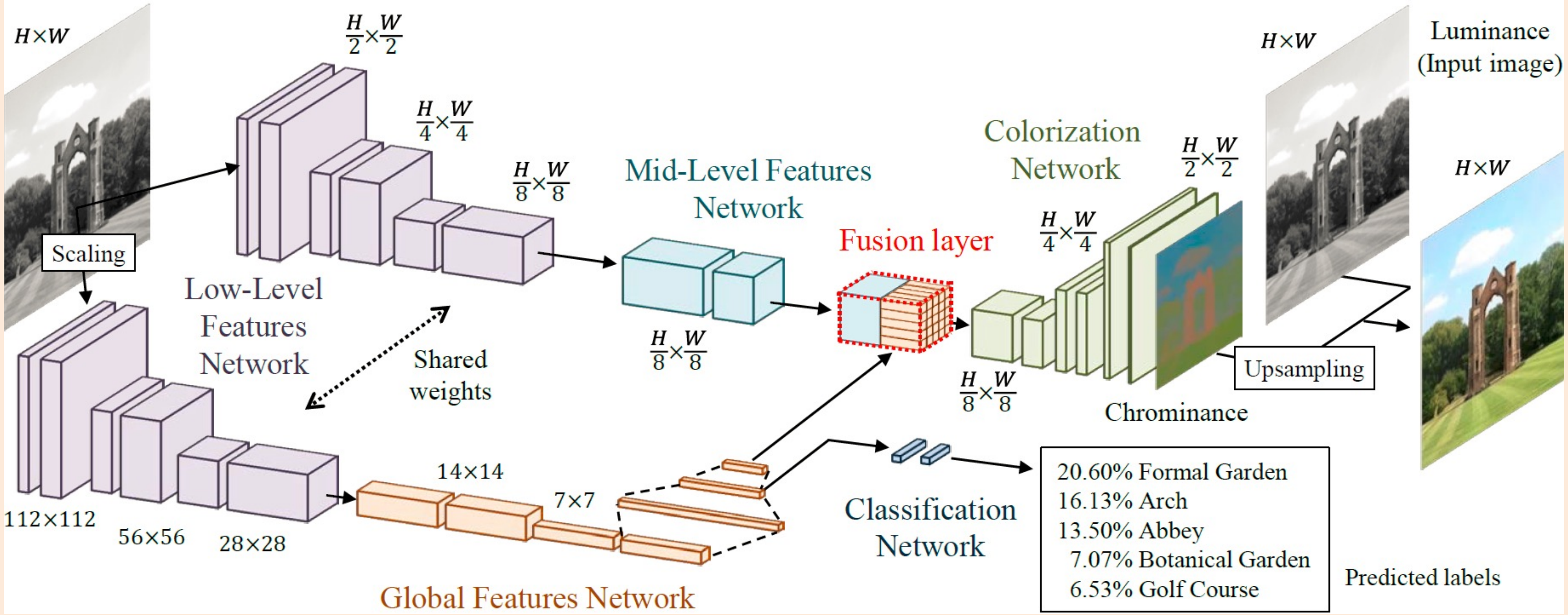
- Some common tricks to reduce overfitting:
  - Standard L2-regularization or L1-regularization ("weight decay").
    - Sometimes with different λ for each layer.
    - Recent work shows this can introduce bad local optima.
  - Early stopping of the optimization based on validation accuracy.
  - Dropout: randomly zeroes activations *z* values to discourage dependence.
  - Implicit regularization from using SGD.
  - Special architectures like convolutional neural networks.

Next Topic: Automatic Differentiation

bonus!

More-Complicated Layers

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

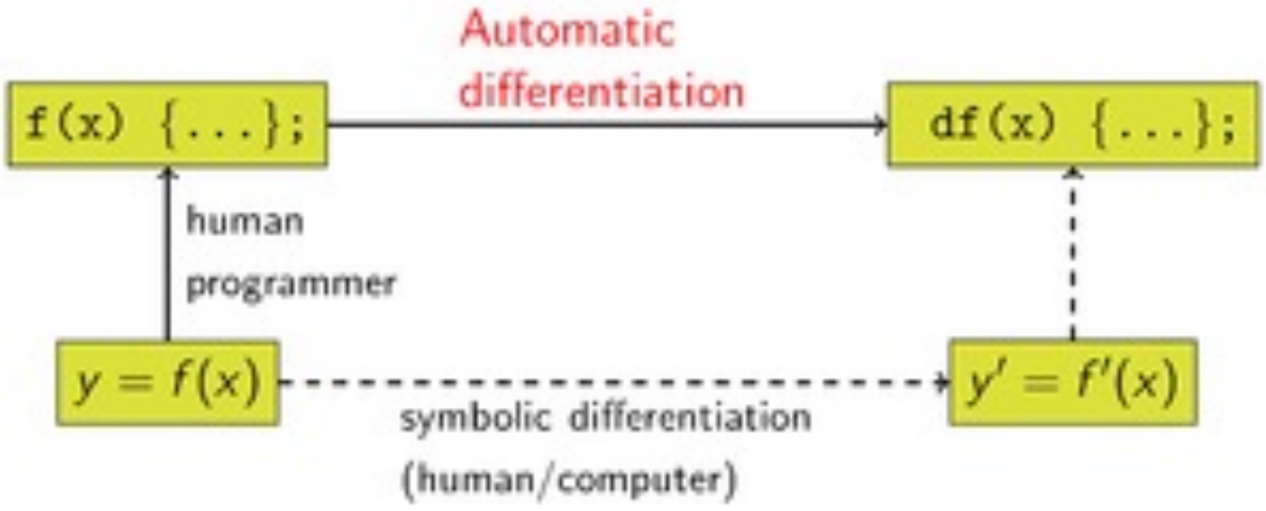


- Developing networks like this is made easier using automatic differentiation.

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.


Automatic differentiation
f(x) {...};
df(x) {...};
human programmer
y = f(x)
y′ = f′(x)
symbolic differentiation (human/computer)

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.
  - AD can compute gradient of any differentiable layer you can implement.
    - 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 the redundant calculations”.
  - When you implement backpropagation,
    it uses the same sequence of operations as AD.
  - AD basically just writes every operation as instance of the chain rule.

If $f(x) = g(h(x))$

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

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: $f_5(f_4(f_3(f_2(f_1(x)))))$.
  - $f_1(x) = -2*x$, $f_2(z) = \exp(z)$, $f_3(z) = 1+z$, $f_4(z) = \log(z)$, $f_5(z) = 10*x$.
    - 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:
  - $f'(x) = f_5'(f_4(f_3(f_2(f_1(x)))))*f_4'(f_3(f_2(f_1(x))))*f_3'(f_2(f_1(x)))*f_2'(f_1(x))f_1'(x)$.

$$\underbrace{f_5'(f_4(f_3(f_2(f_1(x)))))}_{10} * \underbrace{f_4'(f_3(f_2(f_1(x))))}_{1/f_3(f_2(f_1(x))) \,=\, \frac{1}{1+exp(-2x)}} * \underbrace{f_3'(f_2(f_1(x)))}_{1} * \underbrace{f_2'(f_1(x))}_{exp(f_1(x)) \,=\, exp(-2x)} \underbrace{f_1'(x)}_{-2} \Rightarrow -\frac{20exp(-2x)}{1+exp(-2x)}$$

Automatic Differentiation – Single Input+Output

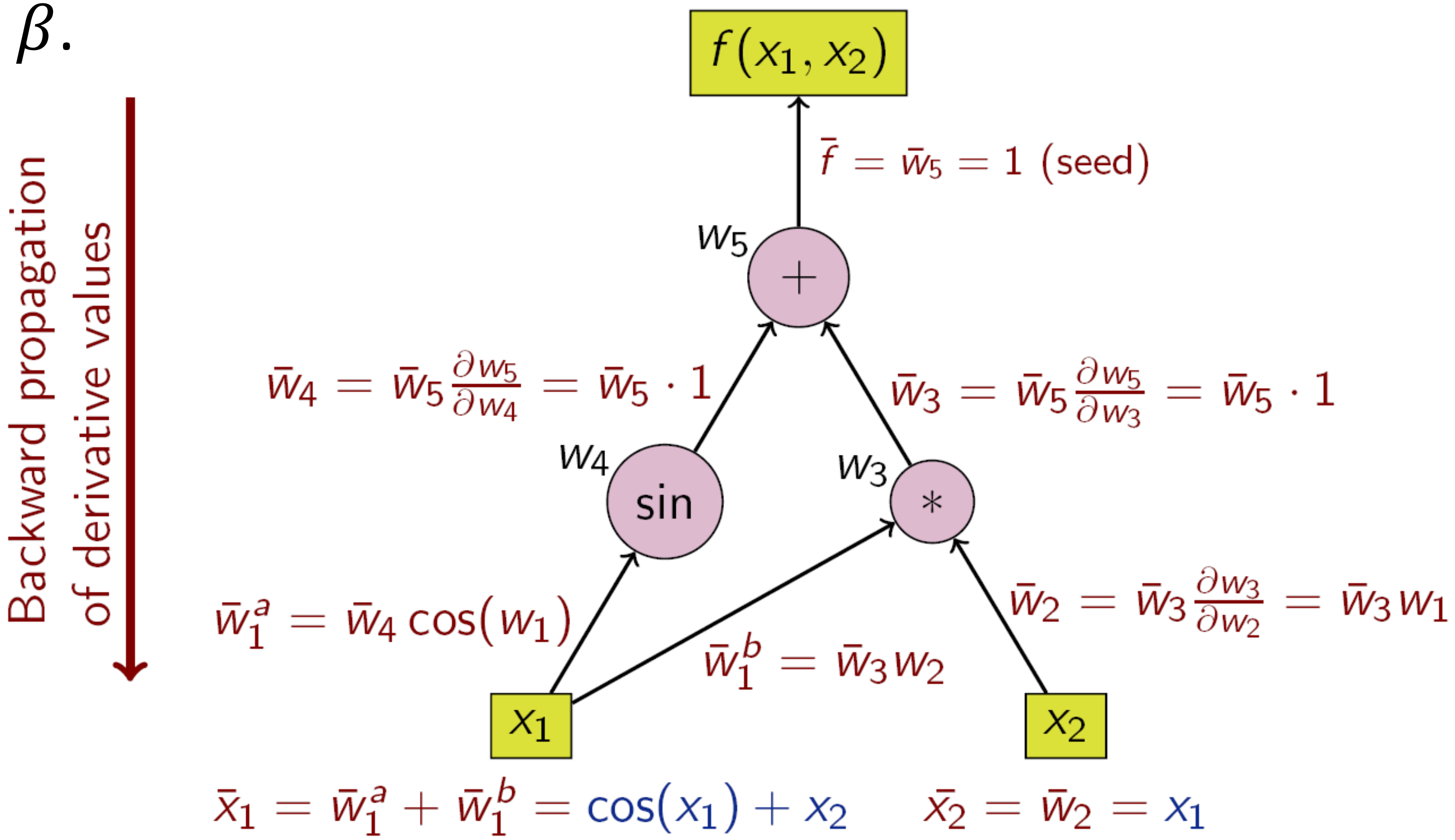
- Our function written as a set of compositions:
  - $f_5(f_4(f_3(f_2(f_1(x)))))$.
- The derivative written using the chain rule::
  - $f'(x) = f_5'(f_4(f_3(f_2(f_1(x)))))*f_4'(f_3(f_2(f_1(x))))*f_3'(f_2(f_1(x)))*f_2'(f_1(x))f_1'(x)$.
- Notice that this leads to repeated calculations.
  - For example, 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).

Automatic Differentiation – Multiple Parameters

- In ML problems, we often have more than 1 parameter.
  - And we want to compute the gradient for the same cost as the function.
- To generalize AD to this case, we define a computation graph:
  - A directed acyclic graph (DAG).
  - Root nodes are the parameters (and inputs).
  - Intermediate nodes are computed values (α values).
  - Leaf node is the function value.
- Computing the gradient with AD:
  - The forward pass evaluates the function and stores intermediate values.
    - Going from the roots through the intermediate nodes to the leaf.
  - The backward pass applies the f_i' functions to the α values.
    - Accumulating the needed pieces of the chain rule until each root has its partial derivative.

Automatic Differentiation – Multiple Parameters

- Wikipedia's example of a computation graph:
  - For computing the gradient of $f(x_1,x_2) = \sin(x_1) + x_1x_2$.
  - Using w for α, $\bar{w}$ for β.



Backward propagation of derivative values
$f(x_1, x_2)$
$\bar{f} = \bar{w}_5 = 1$ (seed)
w_5
$+$
$\bar{w}_4 = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = \bar{w}_5 \cdot 1$
$\bar{w}_3 = \bar{w}_5 \frac{\partial w_5}{\partial w_3} = \bar{w}_5 \cdot 1$
w_4
sin
w_3
$*$
$\bar{w}_1^a = \bar{w}_4 \cos(w_1)$
$\bar{w}_2 = \bar{w}_3 \frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1$
$\bar{w}_1^b = \bar{w}_3 w_2$
x_1
x_2
$\bar{x}_1 = \bar{w}_1^a + \bar{w}_1^b = \cos(x_1) + x_2$
$\bar{x}_2 = \bar{w}_2 = x_1$

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

Automatic Differentiation - Discussion

- AD is amazing – get gradient for the same cost as the function.
  - You can try out lots of stuff, and enjoy thoroughly overfitting validation set!
  - Modern AD codes have lots of features, like built-in derivatives of matrix operations.

- But reverse-mode AD has some drawbacks:
  - Need to store all intermediate calculations, so requires a lot of storage.
    - For basic deep neural networks, hand-written code would only need to store the activations.
      - Modern code has some of these space savings built in.
    - For other functions, the storage cost of AD is much higher than handwritten derivative code.
      - “Checkpointing” exists to reduce storage, but increases computational cost.
  - Has the same cost as computing the function, which is a pro and a con.
    - For basic deep neural networks, these have the same cost so this is what we want.
    - For other functions, the gradient might be possible to compute at a lower cost than the function value.
  - May miss opportunities for parallelism, or miss tricks to avoid numerical problems.

- AD only makes sense at points where the function is differentiable.
  - TensorFlow and PyTorch can give incorrect “subderivatives” at non-differentiable ReLU points.
  - AD cannot (directly) do things like “take the derivative of a function of a sample from the distribution”.

Summary

- Implicit regularization and double descent curves.
- Deep learning:
  - Can allow learning with smaller models and less data than “wide” networks.
- Vanishing gradient in deep networks (gradient may be close to 0).
  - Can reduce with rectified linear units (ReLU) and skip connections.
- Overview of neural network training heuristics.
- Automatic differentiation:
  - Decomposing code using the chain rule, to make derivative code.
  - Can compute gradient for same cost as objective function.
  - Some disadvantages compared to manual.
- Convolutions are flexible class of signal/image transformations.
  - Can approximate derivatives and integrals at different scales/orientations.
- Convolutional neural networks:
  - Include layers that apply several (learned) convolutions.
  - Significantly decreases number of parameters.
  - Achieves a degree of translation invariance.
  - Often combined with pooling operations like max pooling.
- Next time: non-binary likelihoods! 😮

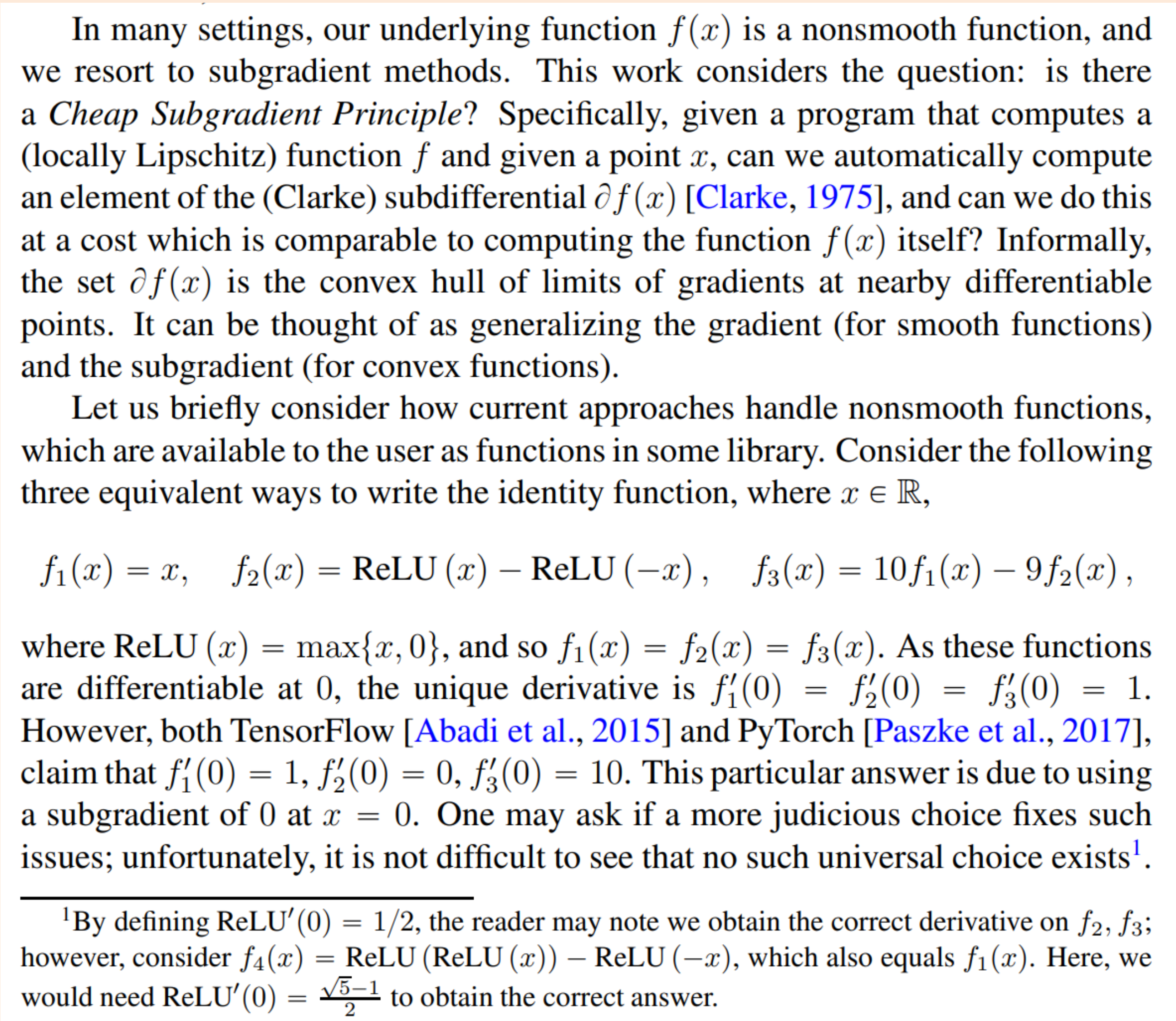
bonus!

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 per partial derivative.
- Forward-mode can be better if output dim > input dim
- “Mixed mode” also possible

bonus!

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 $x \in \mathbb{R}$,

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

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

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