Deep Learning for Data Science
DS 542

Midterm Review

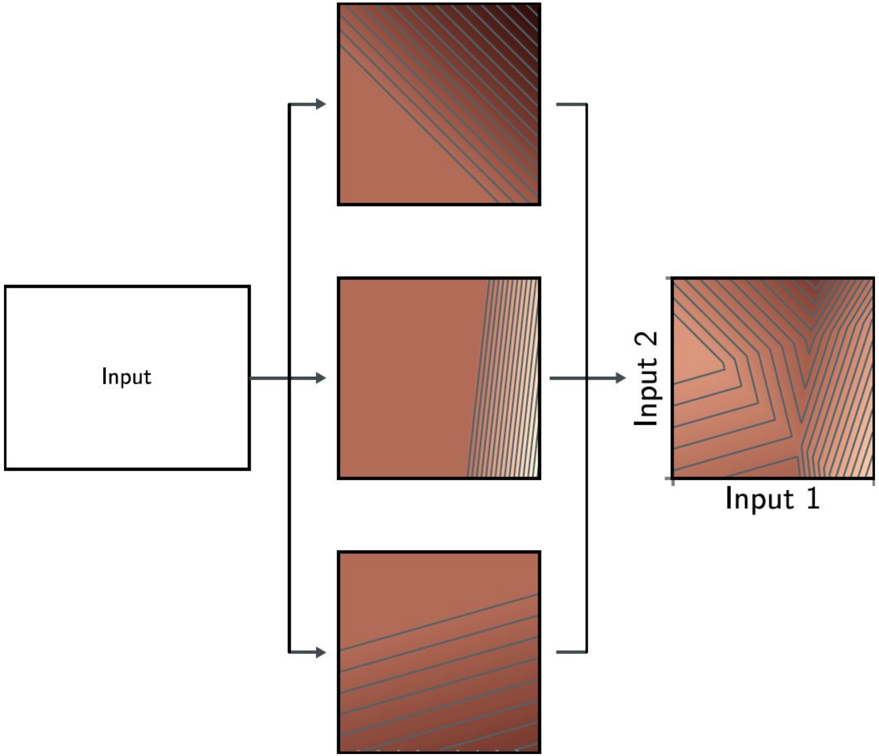
Slides originally by Thomas Gardos.

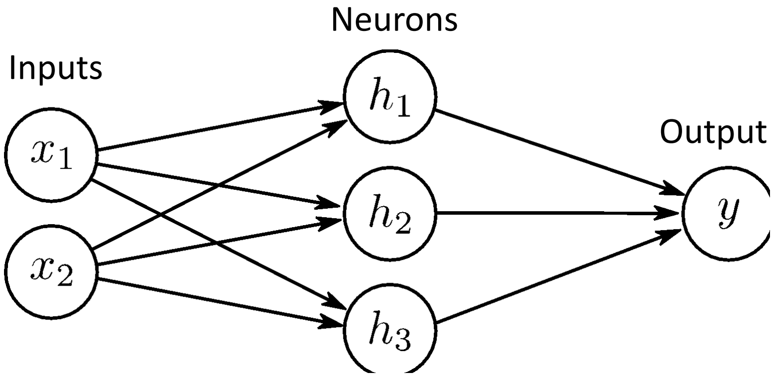
Images from [Understanding Deep Learning] unless otherwise cited.

Administrivia

- No discussion today
- Midterm tomorrow
  - Bring your laptops to get started in class!
  - Due Friday night

How would you draw and write this neural network?





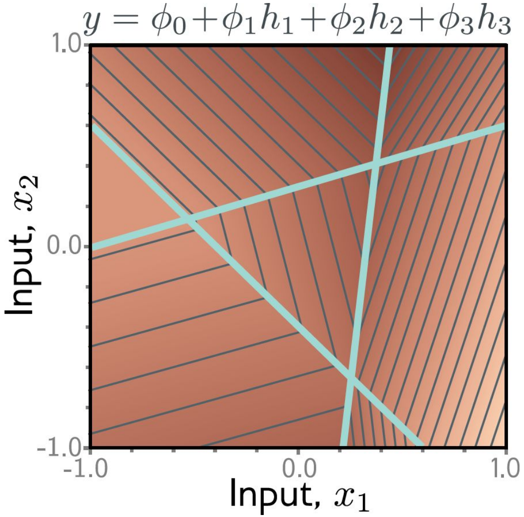
"neural network"

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11} x_1 + \theta_{12} x_2]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21} x_1 + \theta_{22} x_2]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31} x_1 + \theta_{32} x_2]$$

Number of output regions

- In general, each output consists of multi-dimensional convex polytopes
- With two inputs, and three hidden units, we saw there were seven polygons for each output:

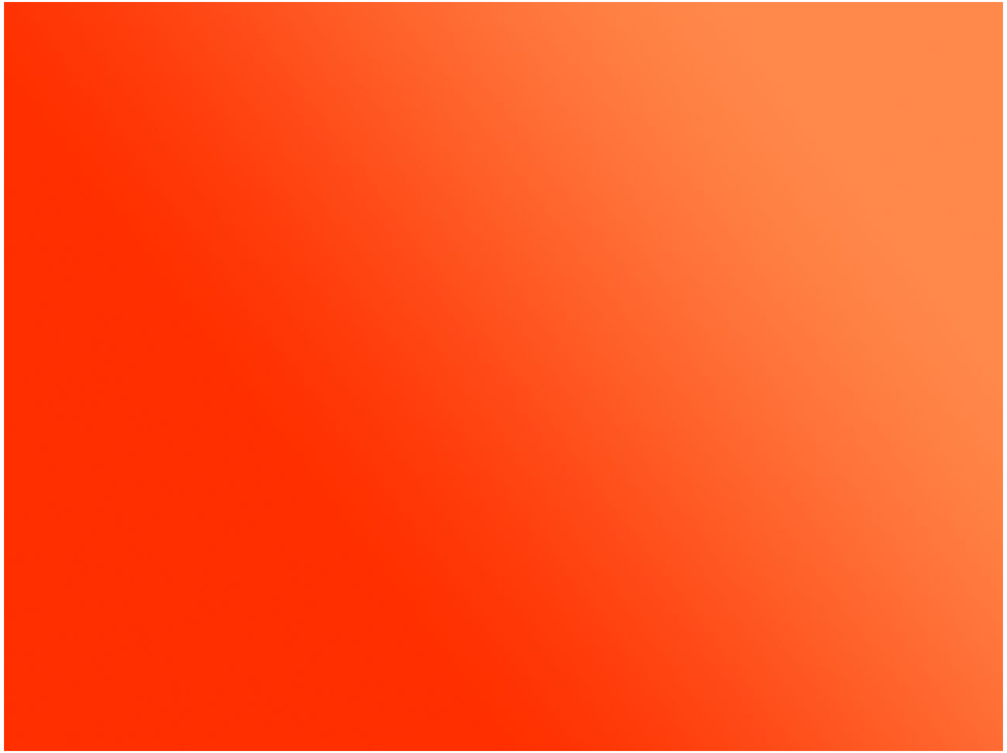


Polytope -- Wikipedia
In elementary geometry, a polytope is a geometric object with flat sides (faces). Polytopes are the generalization of three-dimensional polyhedra to any number of dimensions. Polytopes may exist in any general number of dimensions n as an n-dimensional polytope or n-polytope.

Why does flat matter?

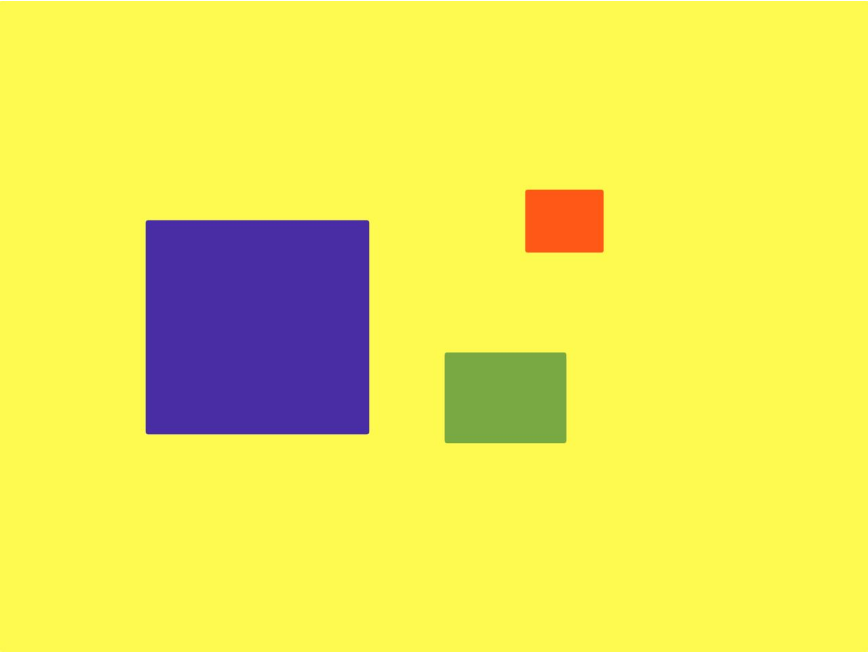
- Neural networks with only ReLU activation functions are piecewise linear.
- Each output region is a linear function.

What does a linear function look like?

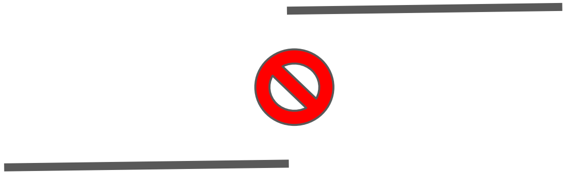


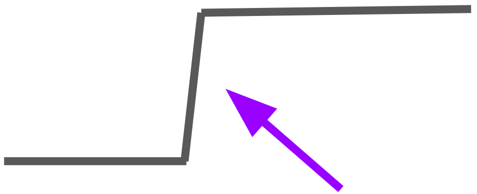
Neural networks have continuous output

This is not particularly easy for a neural network.



Neural networks have continuous output





Not a vertical line!

Also, making that line steeper requires matching changes to keep next line flat.

Smoothing does not help.

My Obsession with Neural Fields

Neural field = neural network taking in coordinates as input and outputting some quantity related to that position.

- One of the homework notebooks used them to recreate images.
- Very visual way to see the biases of neural networks.
  - Many blurry images
  - Also some networks that got stuck and effectively just trained constants.

I also have some research related to them, but that's another story.

Recap

- So far, we talked about *linear regression*, *shallow neural networks* and *deep neural networks*
- Each have parameters, ϕ, that we want to choose for a *best possible mapping between input and output* training data
- A *loss function* or *cost function*, $L[\phi]$, returns a single number that describes a mismatch between $f[x_i, \phi]$ and the ground truth outputs, y_i.

Gradient descent algorithm

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}.$$

Also notated as $\nabla_w L$

Step 2. Update the parameters according to the rule:

$$\phi \longleftarrow \phi - \alpha \frac{\partial L}{\partial \phi},$$

where the positive scalar α determines the magnitude of the change.

Deep Learning depends on Gradient Descent

The majority of making deep learning work is making gradient descent behave!

- He initialization
  - Avoid exploding or vanishing values and gradients at first step.
  - Does not guarantee that values and gradients stay well behaved after many steps.
- ~~Batch~~ layer normalization
  - Keep values and gradients well behaved as parameters change.
  - Messes up our neat pictures before, but stability is worth it.
- Residual networks
  - Make output computation more incremental
  - Add short gradient paths from intermediate layers to output
  - Requires functional form change, limits output shape.
  - Still needs some kind of normalization with many layers.

Backpropagation with Matrix Operations

If

$$\mathbf{f}_0 = \beta_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i$$

$$\mathbf{h}_k = \mathrm{a}[\mathbf{f}_{k-1}]$$

$$\mathbf{f}_k = \beta_k + \boldsymbol{\Omega}_k \mathbf{h}_k$$

Then,

(k-1) more of these when fully unwound

$$\frac{\partial l_i}{\partial \mathbf{f}_{k-1}} = \mathbf{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial l_i}{\partial \mathbf{f}_k} \right)$$

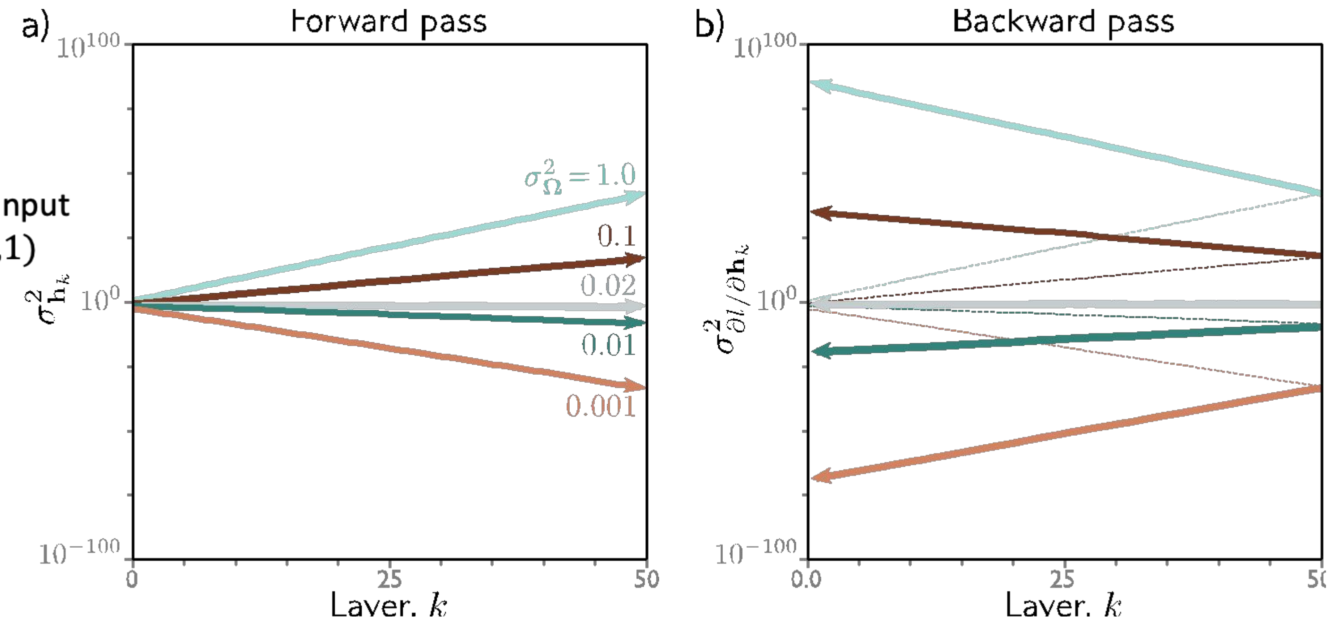
Initialization

Perhaps an obvious point -

- Initializing all parameters to zero is degenerate.
  - All units within a layer will see the same gradients.
  - All units within a layer will get the same updates.
  - All units within a layer will represent the same function.
  - All layers effectively become one wide.
- Generally do not want to start with any symmetries within layers
  - Different initializations are opportunities to learn different useful things.
  - Motivates random initializations.

Initialize weights to different variances

100D Input
~ $N(0,1)$



Exploding gradients

Vanishing gradients

Figure 7.4 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100 dimensional input $\mathbf{x}$ initialized with values from a standard normal distribution, a single output fixed at $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero and the weight matrices $\boldsymbol{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_{\boldsymbol{\Omega}} \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a)

He initialization (assumes ReLU)

- Forward pass: want the variance of hidden unit activations in layer k+1 to be the same as variance of activations in layer k:

$$\sigma_\Omega^2 = \frac{2}{D_h}$$

Number of units at layer k

- Backward pass: want the variance of gradients at layer k to be the same as variance of gradient in layer k+1:

$$\sigma_\Omega^2 = \frac{2}{D_{h'}}$$

Number of units at layer k+1

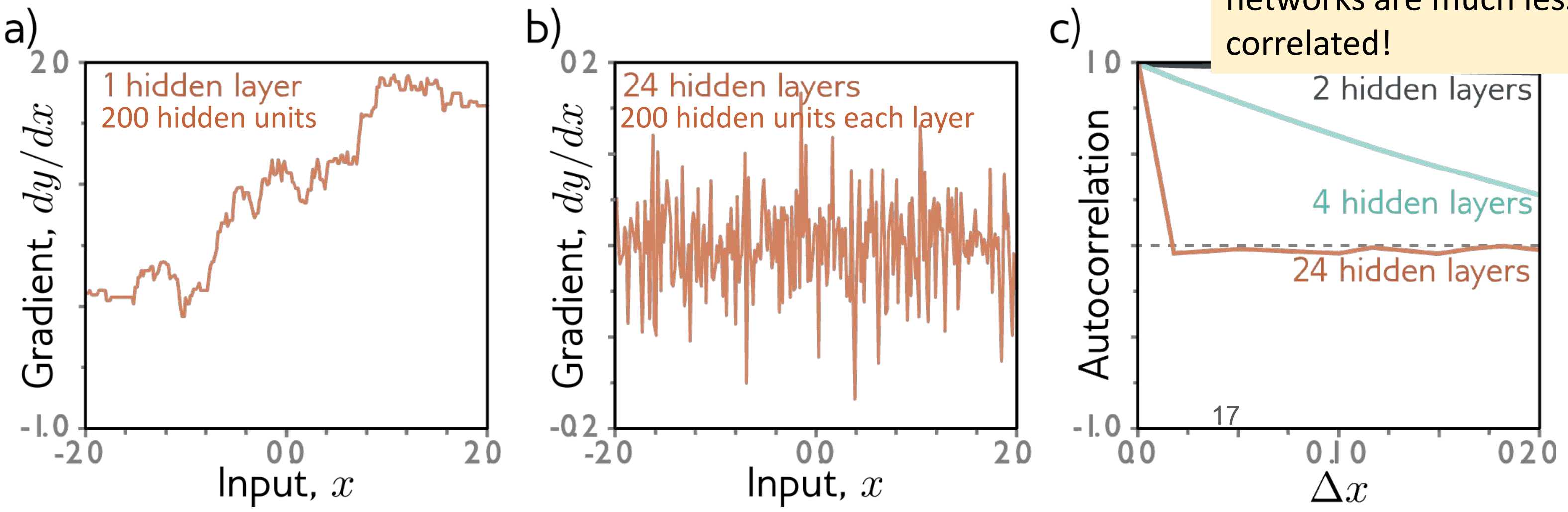
What’s going on?

The Shattered Gradient Phenomenon

Not completely understood, but…

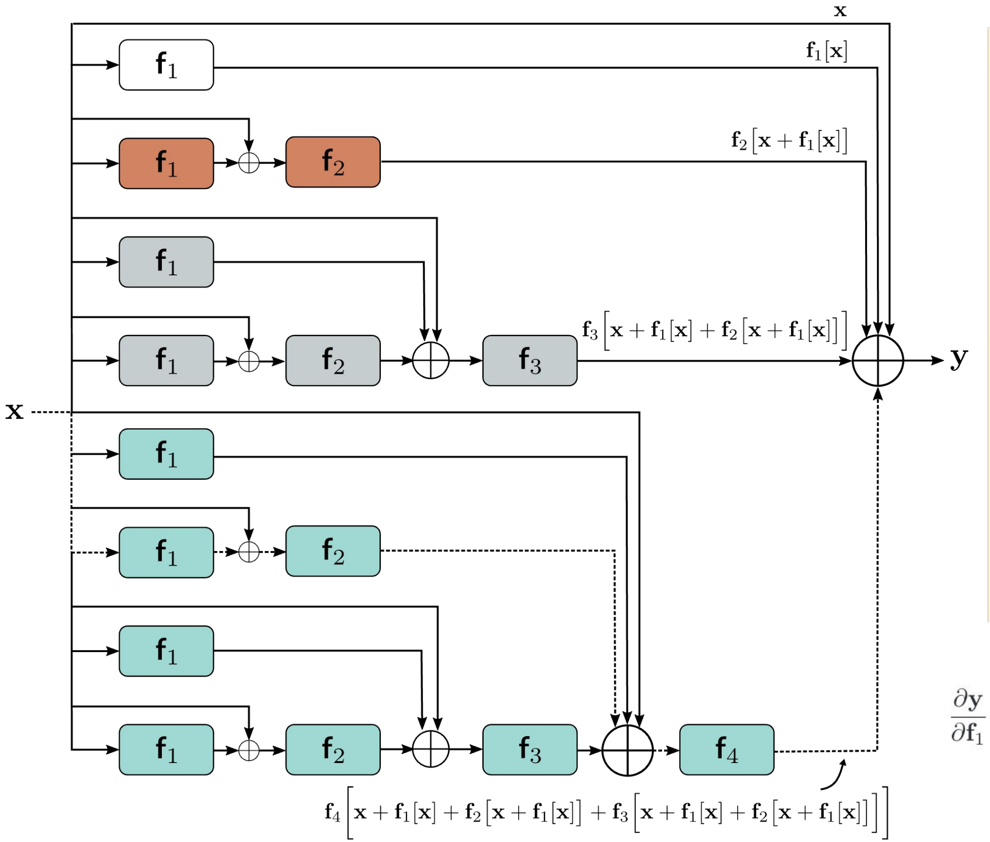
Take a look at $\partial y/\partial x$ for shallow and deep networks.

Gradients of deeper networks are much less correlated!



A small step in gradient descent may jump to wildly different valued gradient!

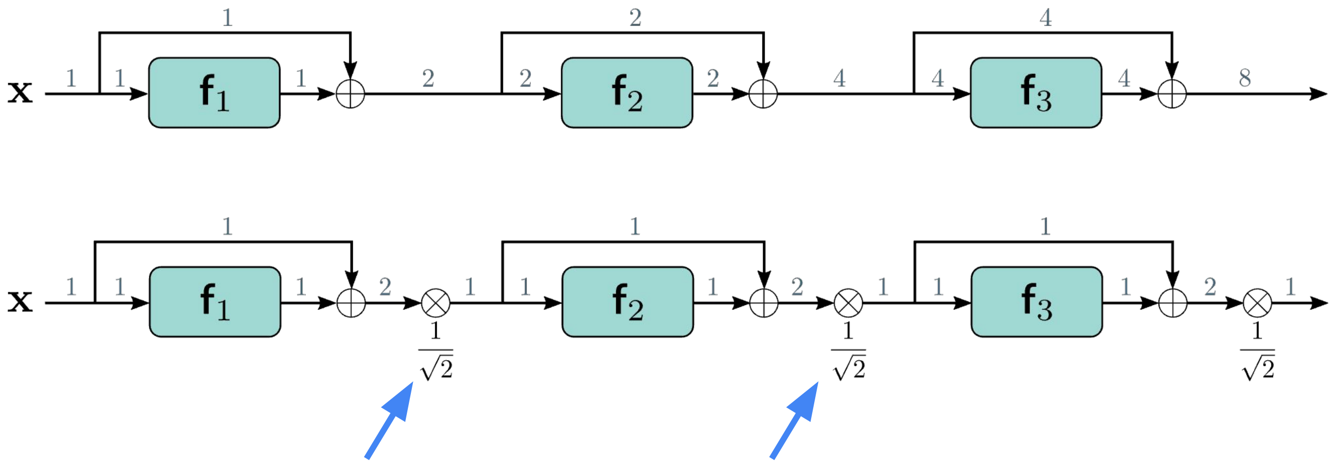
Residual Network as Ensemble of Networks



- 16 possible paths through the network!
- 8 paths include f_1
- The influence of f_1 on $\partial y/\partial f_1$ takes 8 different forms
- Gradients on shorter paths generally better behaved.

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \mathbf{I} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \left(\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\right) + \left(\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\right)$$

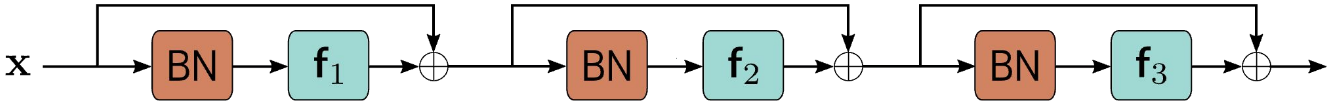
Exploding Gradients in Residual Networks



Could stabilize by renormalizing after adding each residual.

More common to apply *batch normalization*.

Batch Normalization (a.k.a. *BatchNorm*)



- Shifts and rescales each activation so that its mean and variance across the batch become values that are learned during training

Calculate the sample *mean* and *standard deviation* for each hidden unit across samples of the batch.

$$m_h = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} h_i$$

$$s_h = \sqrt{\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}(h_i - m_h)^2},$$

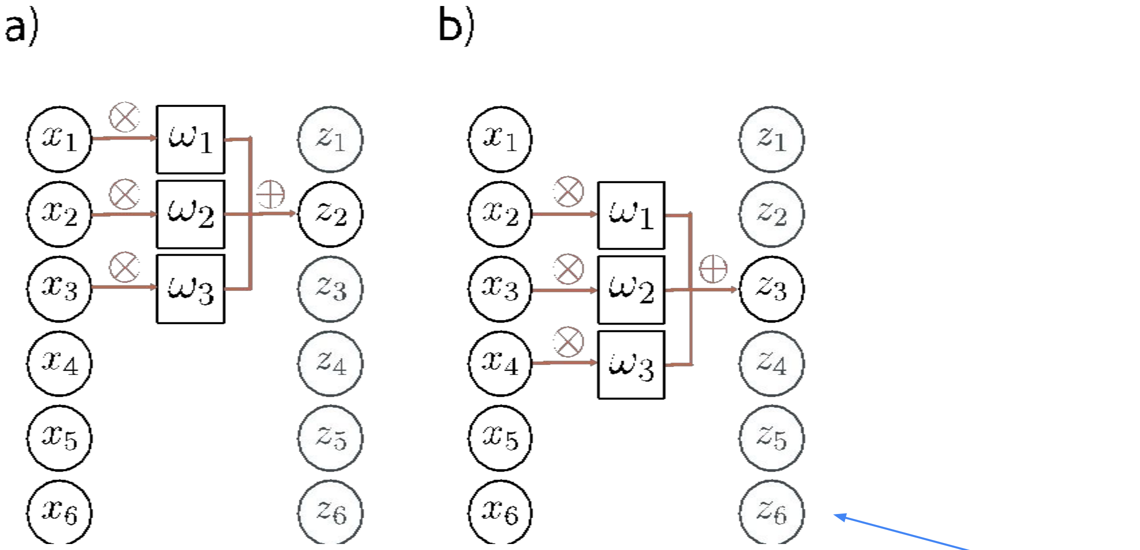
Standardize (*normalize*) to zero-mean and unit standard deviation.

$$\hat{h}_i \leftarrow \frac{h_i - m_h}{s_h + \epsilon} \qquad \forall i \in \mathcal{B},$$

Scale by γ and shift by δ, which are *learned* parameters.

$$h_i \leftarrow \gamma\hat{h}_i + \delta \qquad \forall i \in \mathcal{B}.$$

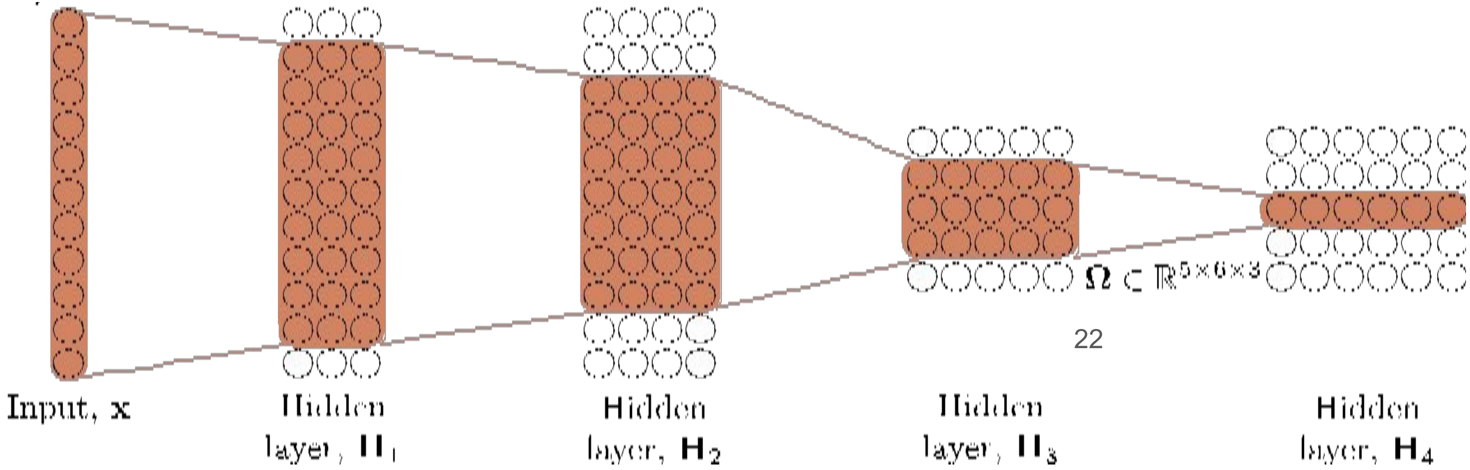
Convolution with kernel size 3



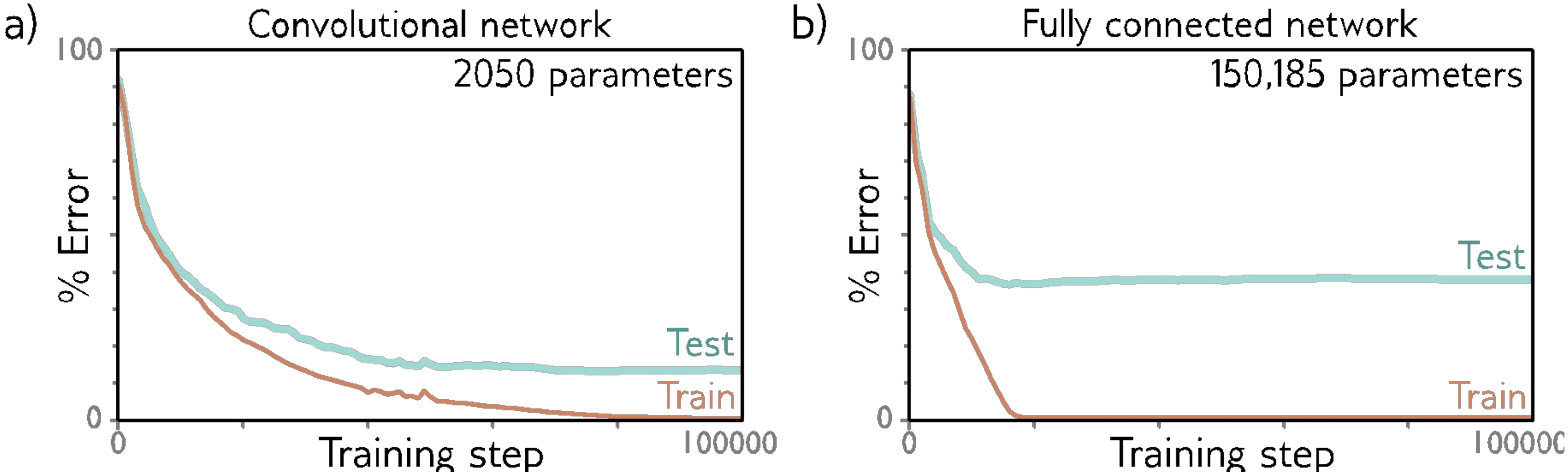
Equivariant to translation of input $\mathbf{f}[\mathbf{t}[\mathbf{x}]] = \mathbf{t}[\mathbf{f}[\mathbf{x}]]$

Receptive fields

$$\mathbb{R}^{C_i \times C_o \times K}$$



Performance



Why?

- Better inductive bias
- Forced the network to process each location similarly
- Shares information across locations
- Search through a smaller family of input/output mappings, all of which are plausible

Practical Tips

Plot your losses frequently.

- Most examples plot every 10-50 epochs. I plot every one if suspicious.
- If jagged, learning rate is too high.
- If flat, look at gradients.

Practical Tips

Consider plotting some measure of gradient values.

- I often just use sum of absolute values over all parameters…
- If the gradients go to zero, your network is done training whether you like it or not.

Practical Tips

Stochastic gradient descent is your friend.

- It is easier to write full batch gradient descent.
- But mini batches tend to be way faster and almost as good loss improvements.

Practical Tips

Bigger networks probably fit better, after you get smaller networks working.

- Test your setup on smaller networks first.
- If your loss improves for a while and then flattens out, maybe a bigger network?
- If you cannot get your loss to improve at all on a small network, just going bigger is not likely to help.

Practice Today

Repeat last (current) homework with FashionMNIST.

- https://github.com/zalandoresearch/fashion-mnist
- https://github.com/DL4DS/fa2024/blob/main/static_files/assignments/10_notebook.ipynb
- https://pytorch.org/vision/0.19/generated/torchvision.datasets.FashionMNIST.html

Feedback?