CPSC 340:
Machine Learning and Data Mining

Neural networks: training

and

Convolutions

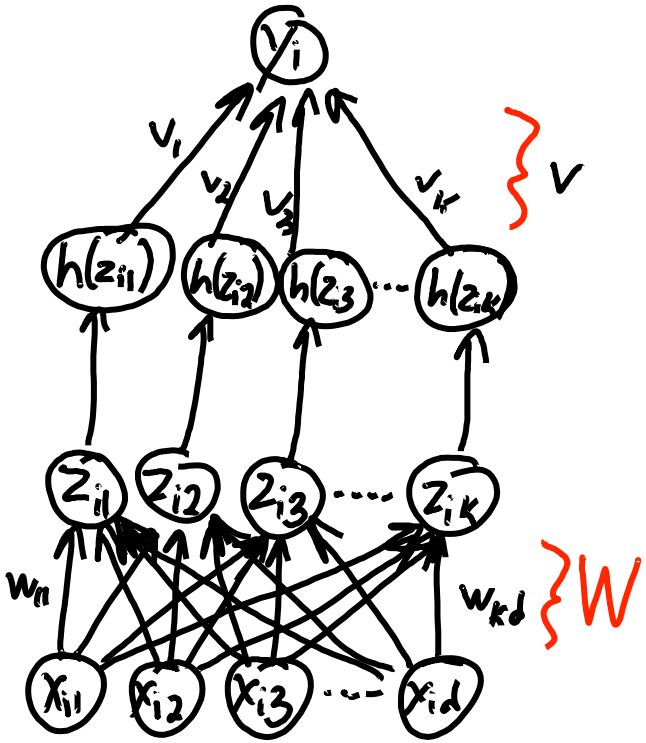
Original version of these slides by Mark Schmidt, with modifications by Mike Gelbart.

Admin

- Assignment 6:
  - Due next Thursday (April 5)

Last Time: Deep Learning

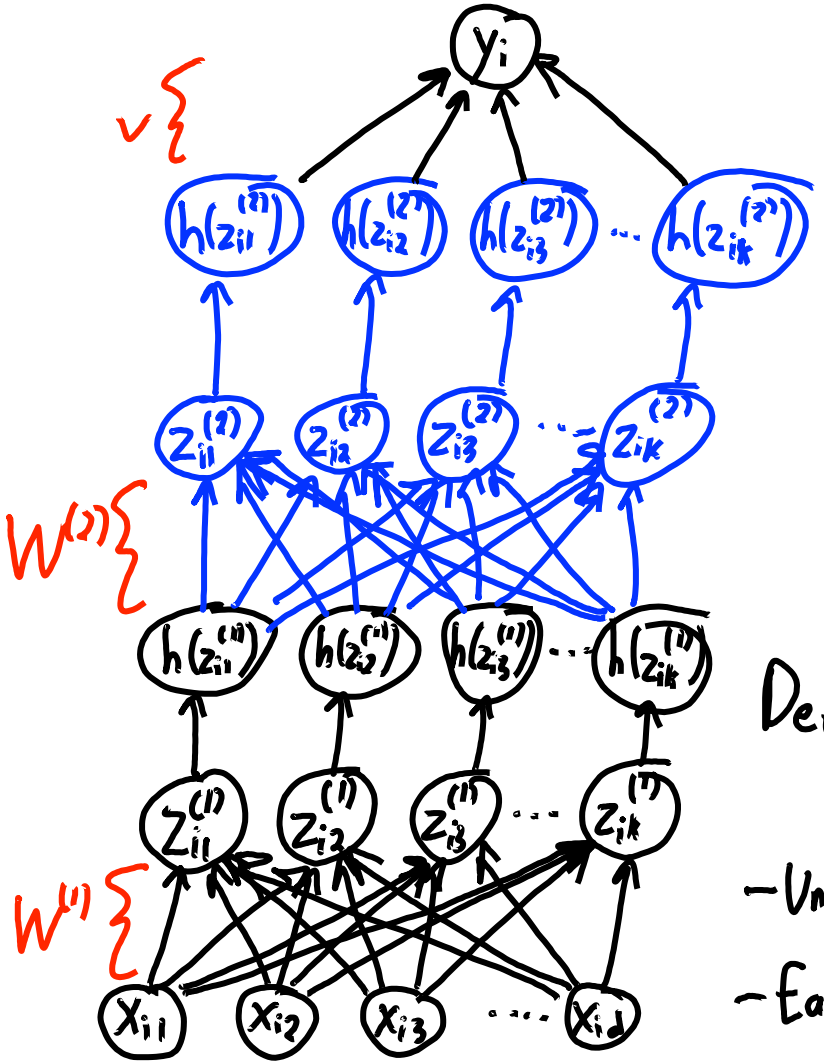
Neural network:



$$y_i = v^T h(Wx_i)$$

Learn 'W' and 'v' together.

- Learn features for supervised learning.
- Non-linear 'h' makes it a universal approximator for large 'K'



Deep neural networks:

$$y_i = v^T h(W^{(2)} h(W^{(1)} x_i))$$

- Unprecedented performance on difficult problems.
- Each layer combines "parts" from previous layer.

Two things I forgot to say last time

- Check out the 3Blue1Brown video on the course website
- Biological motivation: L29 bonus slides

Artificial Neural Networks

- With squared loss, our objective function for one hidden layer is:

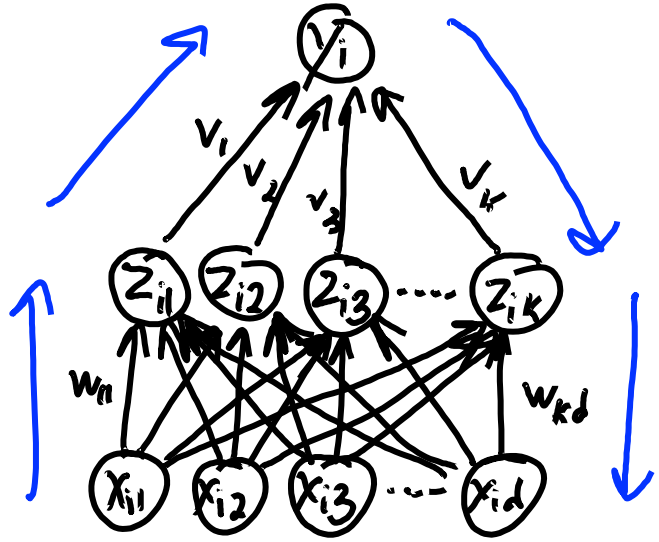
$$f(v, W) = \frac{1}{2} \sum_{i=1}^{n} (v^T h(Wx_i) - y_i)^2$$

- Usual training procedure: stochastic gradient.
  - Compute gradient of random example 'i', update both 'v' and 'W'.
  - Highly non-convex and can be difficult to tune.

- Computing the gradient is known as "backpropagation".
  - Video giving motivation on course webpage.

Backpropagation

- Overview of how we compute neural network gradient:
  - Forward propagation:
    - Compute $z_i^{(1)}$ from x_i.
    - Compute $z_i^{(2)}$ from $z_i^{(1)}$.
    - …
    - Compute $yhat_i$ from $z_i^{(m)}$, and use this to compute error.
  - Backpropagation:
    - Compute gradient with respect to regression weights ‘v’.
    - Compute gradient with respect to $z_i^{(m)}$ and weights $W^{(m)}$.
    - Compute gradient with respect to $z_i^{(m-1)}$ and weights $W^{(m-1)}$.
    - …
    - Compute gradient with respect to $z_i^{(1)}$ and weights $W^{(1)}$.



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

Backpropagation

- I've put the backprop math in the bonus slides.
  - Usually handled for you with neural network / automatic differentiation software
- Do you need to know how to do this?
  - Exact details are probably not vital (there are many implementations), but understanding basic idea helps you know what can go wrong.
  - See discussion here by a neural network expert (and UBC grad) Andrej Karpathy.
  - But right now CPSC 340 is serving a very broad audience, and time is limited
- What I want you to know:
  - The intuition of why, if you naively computed all derivatives, it would be wasteful
  - Cost dominated by matrix multiplications by $W^{(1)}$, $W^{(2)}$, $W^{(3)}$, and 'v'.
    - If have 'm' layers and all z_i have 'k' units, cost would be $O(dk + mk^2)$.

Neural networks for classification

- We've been thinking of NNs as "crazy features + linear regression"
  - For classification, we can do the same but with logistic regression

- For multi-class with 'k' classes, our last layer has size 'k'
  - So we replace 'v' by a matrix
  - Softmax activation at last layer, to produce probabilities
  - Softmax loss is often called "cross entropy" in neural network papers.
  - Typically prepare the labels with a one-hot encoding into a matrix 'Y'.
  - Similar approaches work for multi-label classification

ImageNet Challenge

- ImageNet challenge:
  - Use millions of images to recognize 1000 objects.
- ImageNet organizer visited UBC summer 2015.
- "Besides huge dataset/model/cluster, what is the most important?"
  1. Image transformations (translation, rotation, scaling, lighting, etc.).
  2. Optimization.
- Why would optimization be so important?
  - Neural network objectives are highly non-convex (and worse with depth).
  - Optimization has huge influence on quality of model.

Stochastic Gradient Training

- Challenging to make SG work:
  - Often doesn't work as a "black box" learning algorithm.
  - But people have developed a lot of tricks/modifications to make it work.
- Highly non-convex, so are local mimina hurting us?
  - Some empirical/theoretical evidence that local minima are not the problem.
  - If the network is "deep" and "wide" enough, we think all local minima are good.
  - But it can be hard to get SG to even find a local minimum.

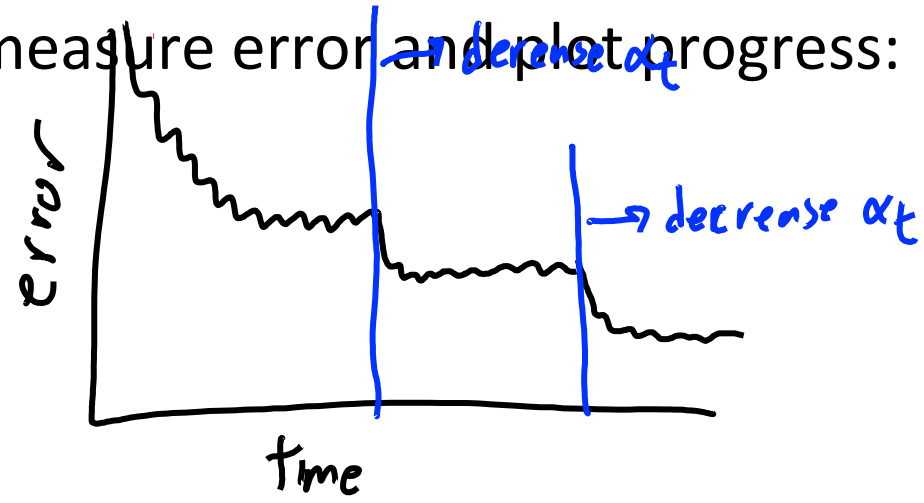
Parameter Initialization

- Parameter initialization is crucial:
  - Can't initialize weights in same layer to zero, 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 Gaussian with small std dev (e.g., 0.00001).
  - Performing multiple initializations does not seem to be important.

- Popular approach from 10 years ago:
  - Initialize with deep unsupervised model (like "autoencoders" – see bonus).

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.
- Bias step-size multiplier: use bigger step-size for the bias variables.
- Momentum:
  - 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 $\beta^t = 0.9$.

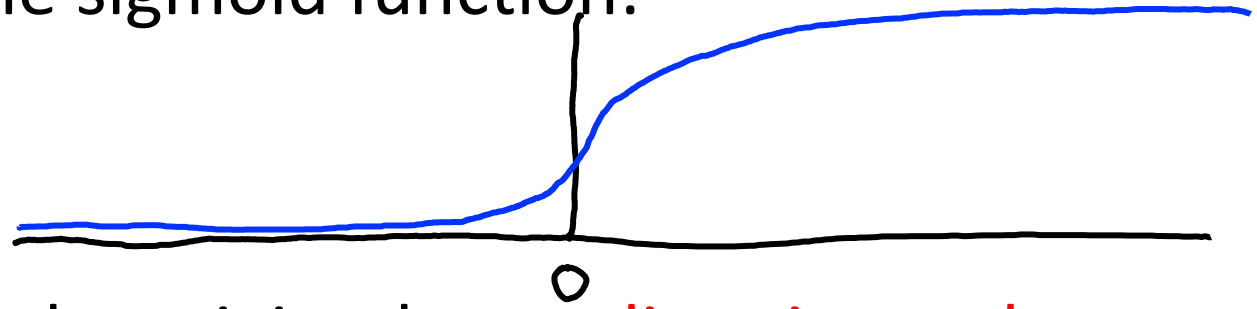
Setting the Step-Size (bonus)

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

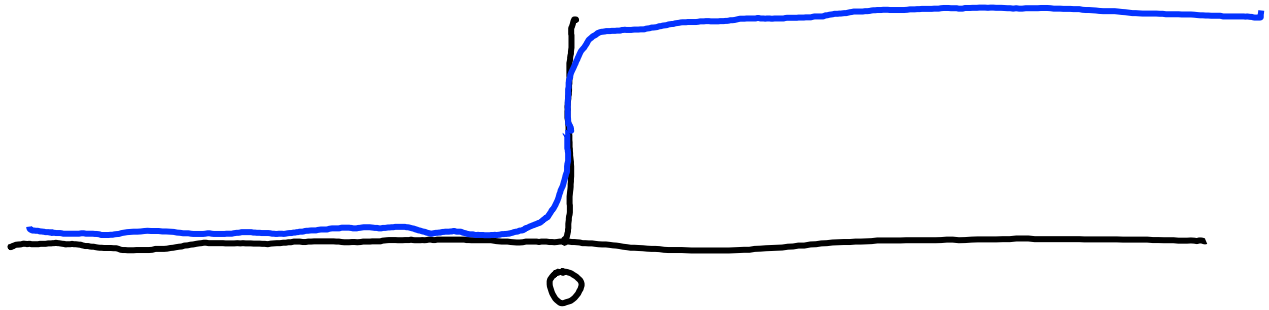
- Several recent methods using a step size for each variable:
  - AdaGrad, RMSprop, Adam (often work better "out of the box").
  - Seem to be losing popularity to stochastic gradient (often with momentum).
    - Often yields lower test error but this requires more tuning of step-size.
- Batch size (number of random examples) also influences results.
  - Bigger batch sizes often give faster convergence but to worse solutions.
- Another recent trick is batch normalization:
  - Try to "standardize" the hidden units within the random samples as we go.

Vanishing Gradient Problem

- Consider the sigmoid function:



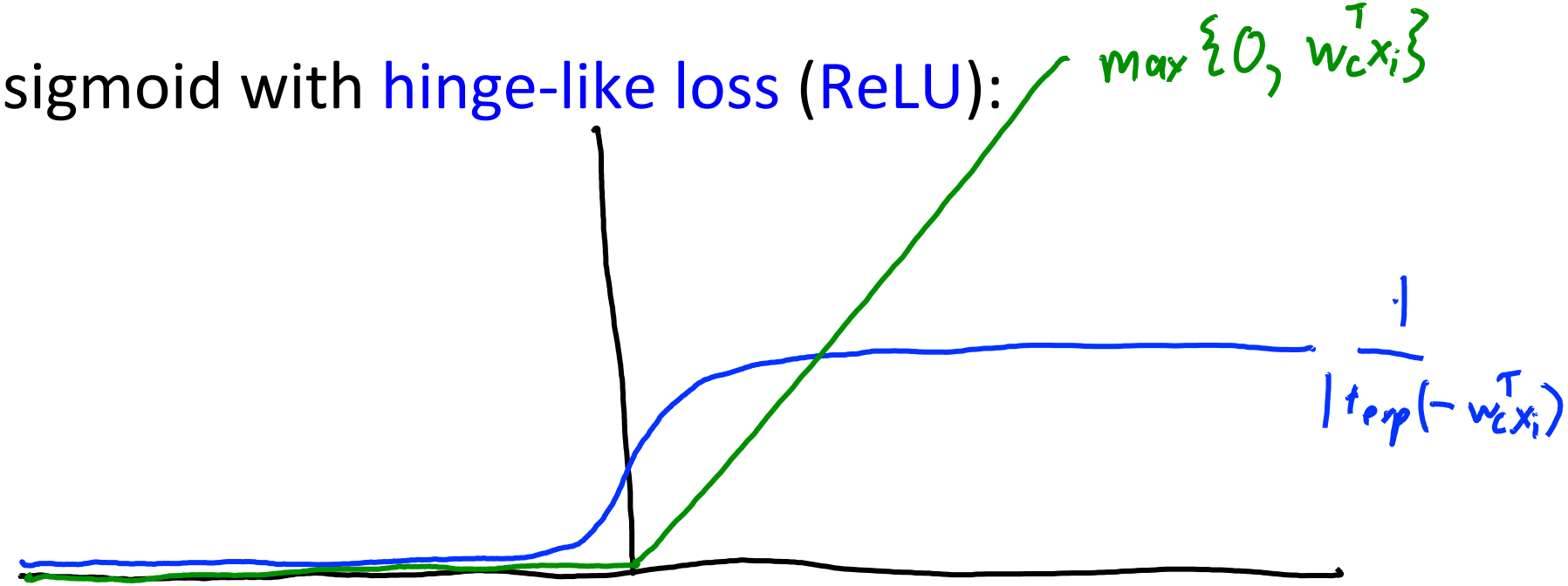
- Away from the origin, the gradient is nearly zero.
- The problem gets worse when you take the sigmoid of a sigmoid:



- In deep networks, many gradients can be nearly zero everywhere.

Rectified Linear Units (ReLU)

- Replace sigmoid with hinge-like loss (ReLU):



- Just sets negative values z_{ic} to zero.
  - Fixes vanishing gradient problem.
  - Works well in practice.

Deep Learning and the Fundamental Trade-Off

- Neural networks are subject to the fundamental trade-off:
  - As we increase the depth, training error decreases.
  - As we increase the depth, training error no longer approximates test error.

- We want deep networks to model highly non-linear data.
  - But increasing the depth leads to overfitting.

- How could GoogLeNet (L29 bonus slides) use 22 layers?
  - Many forms of regularization and keeping model complexity under control.

Standard Regularization

- We typically add 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 is called "weight decay" in neural network papers.
  - Could also use L1-regularization.
- Hyperparameter optimization:
  - Try to optimize validation error in terms of λ_1, λ_2, λ_3, λ_4, …

- Unlike linear models, typically use multiple types of regularization.

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.
- Training accuracy should continue going up.

Dropout

- Dropout is a more recent form of regularization:
  - On each iteration, randomly set some x_i and z_i to zero (often use 50%).
    - Prevents "co-adaptation"
  - After a lot of success, dropout may already be going out of fashion.
  - See bonus slides for more info

Vocabulary

- One-hot encoding
- Dropout
- Weight decay
- Momentum
- Batch normalization
- Vanishing gradient

(pause)

Convolutions

- Next class we'll talk about convolutional neural networks
  - These dominate computer vision
- For the rest of today we'll talk about convolutions

1D Convolution (notation is specific to this lecture)

- 1D convolution input:
  - Signal 'x' which is a vector length 'n'.
    - Indexed by i=1,2,…,n.
  - Filter 'w' which is a vector of length '2m+1':
    - Indexed by i=-m,-m+1,…-2,0,1,2,…,m-1,m

$x = [0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13]$

$w = [0 \ -1 \ 2 \ -1 \ 0]$

$w_{-2} \ w_{-1} \ w_0 \ w_1 \ w_2$

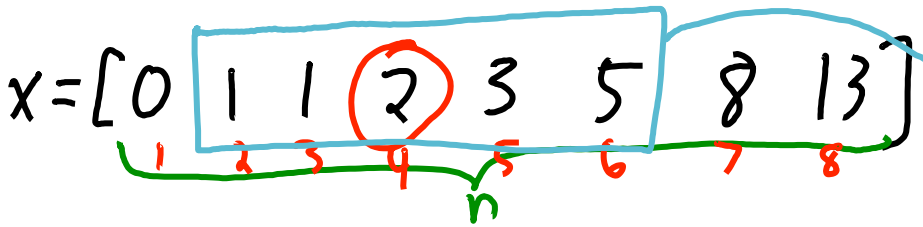
- Output is a vector of length 'n' with elements:

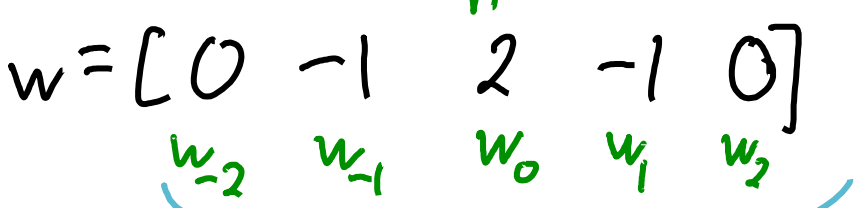
$$z_i = \sum_{j=-m}^{m} w_j x_{i+j}$$

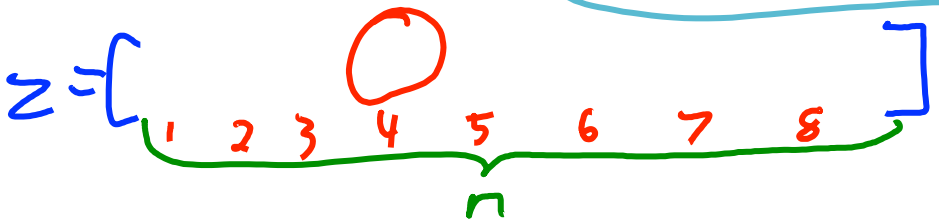
  - You can think of this as centering w at z_i and taking a dot product.

1D Convolution

- 1D convolution example:
  - Signal:
  - Filter:
  - Convolution:







Let's compute z_4:

$x_{4-2:4+2} = [1\ 1\ 2\ 3\ 5]$

Multiply element-wise

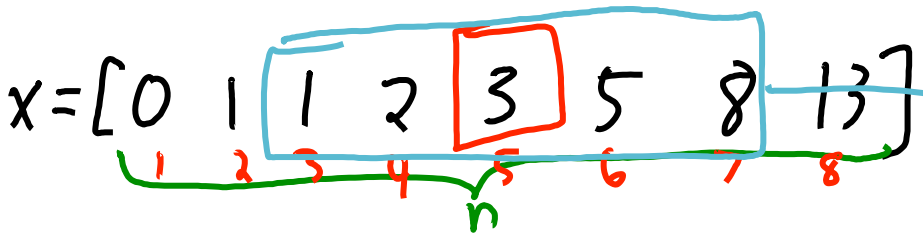
$[0\ -1\ 4\ -3\ 0]$

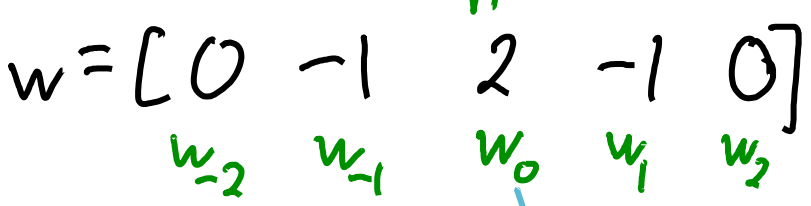
↓ Add

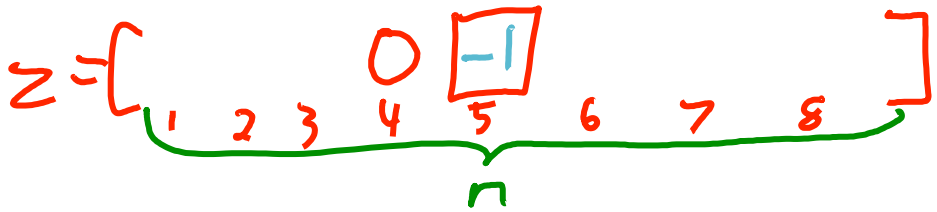
$z_4 = 0 - 1 + 4 - 3 + 0 = 0$

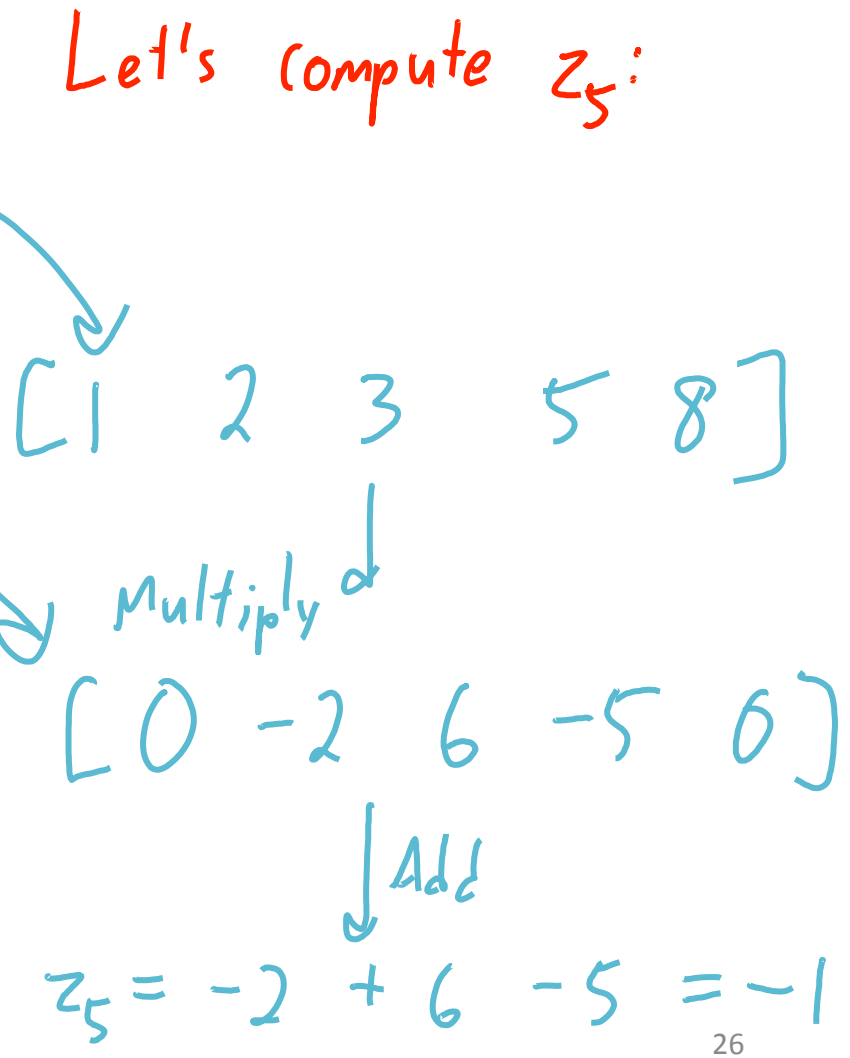
1D Convolution

- 1D convolution example:
  - Signal:
  - Filter:
  - Convolution:









1D Convolution Examples

- Examples:
  - "Identity"

$w = [0 \;\; 1 \;\; 0]$

Let $x = [0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13]$

$z = [0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13]$

$0 \cdot x_0 + 1 \cdot x_1 + 0 \cdot x_2$ $\quad$ $0 \cdot x_1 + 1 \cdot x_2 + 0 \cdot x_3$

  - "Translation"

$w = [0 \;\; 0 \;\; 1]$

$z = [1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13 \;\; ?]$

$0 \cdot x_0 + 0 \cdot x_1 + 1 \cdot x_2$

1D Convolution Examples

- Examples:
  - "Identity"

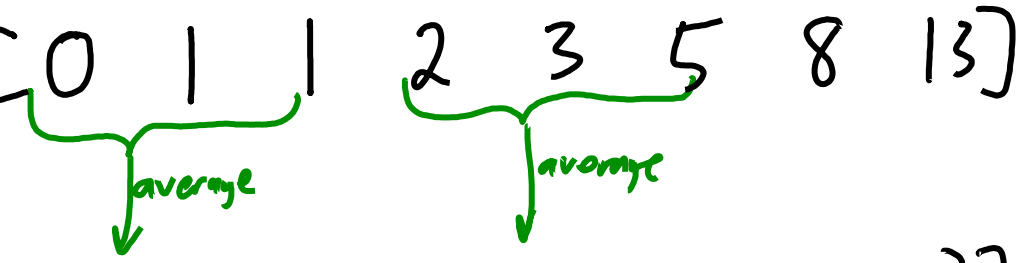
↳ $w = [0 \;\; 1 \;\; 0]$

Let $x = [0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13]$

$z = [0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13]$

  - "Local Average"

↳ $w = [\frac{1}{3} \;\; \frac{1}{3} \;\; \frac{1}{3}]$



$z = [? \;\; \frac{2}{3} \;\; 1\frac{1}{3} \;\; 2 \;\; 3\frac{1}{3} \;\; 5\frac{1}{3} \;\; 8\frac{2}{3} \;\; ?]$

Boundary Issue

- What can we about the "?" at the edges?

If x=[0 1 1 2 3 5 8 13] and w=[⅓ ⅓ ⅓] then z=[? 2⅓ 1⅓ 2 3⅓ 5⅓ 8⅔ ?]

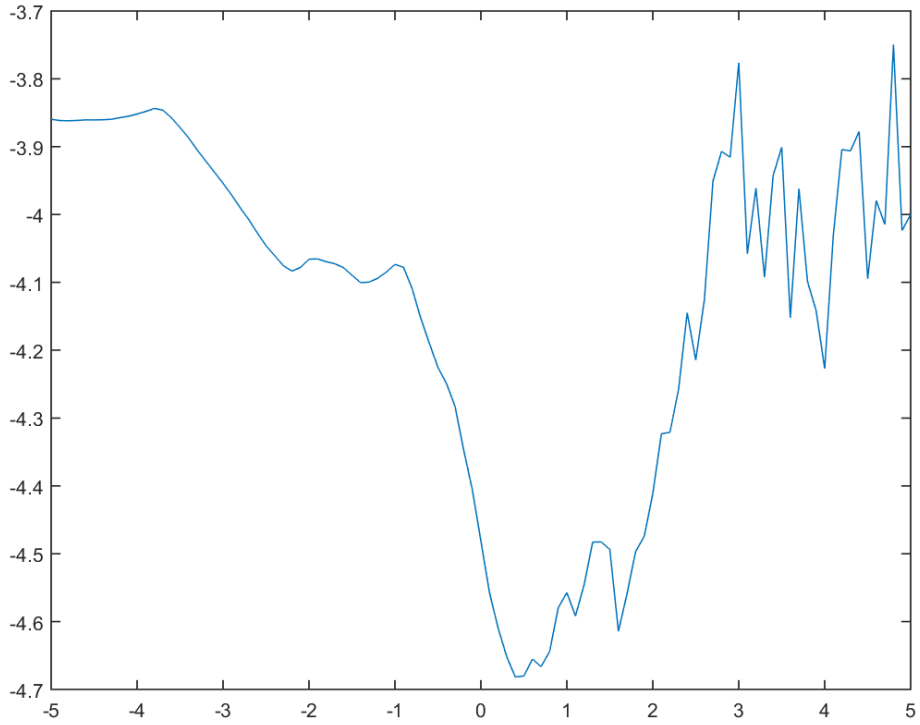
- Can assign values past the boundaries:
  - "Zero": x=[0 0 0 0 1 1 2 3 5 8 13 0 0 0]
  - "Replicate": x=[0 0 0 0 1 1 2 3 5 8 13 13 13 13]
  - "Mirror": x=[2 1 1 0 1 1 2 3 5 8 13 8 5 3]
- Or just ignore the "?" values and return a shorter vector:

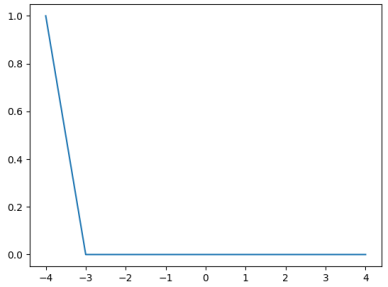
z=[2⅓ 1⅓ 2 3⅓ 5⅓ 8⅔]

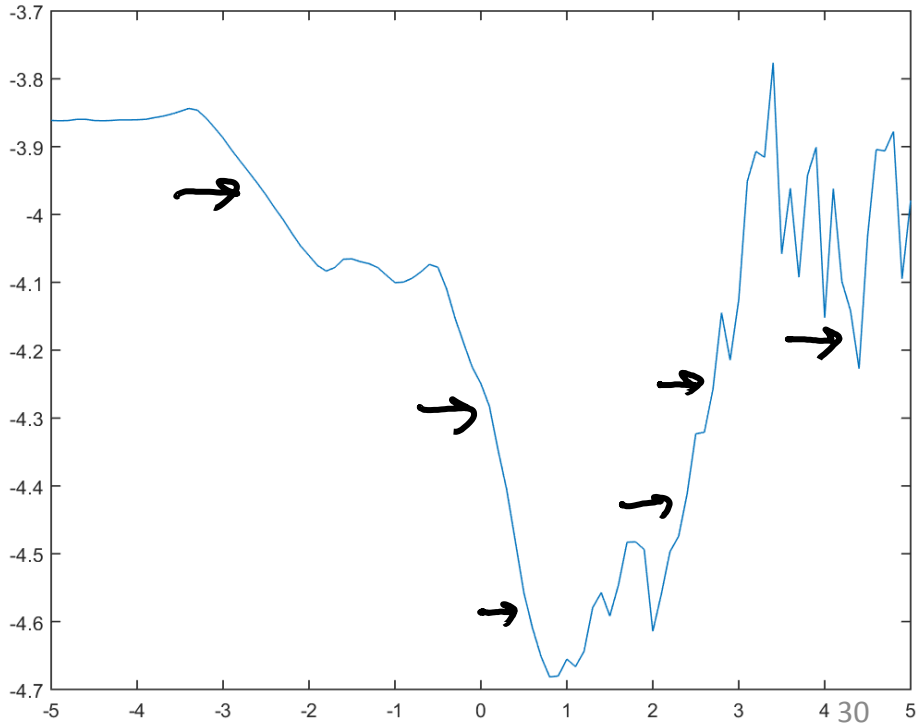
1D Convolution Examples

- Translation convolution shift signal:

$$w = [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$



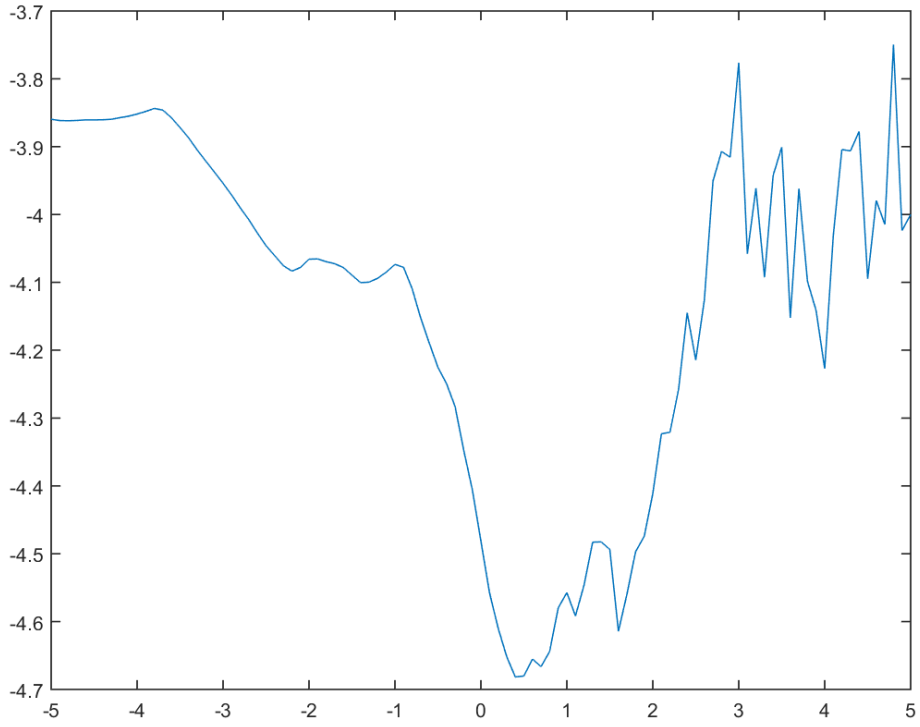


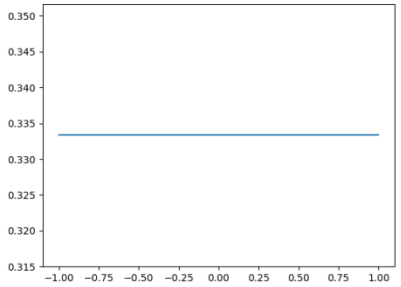


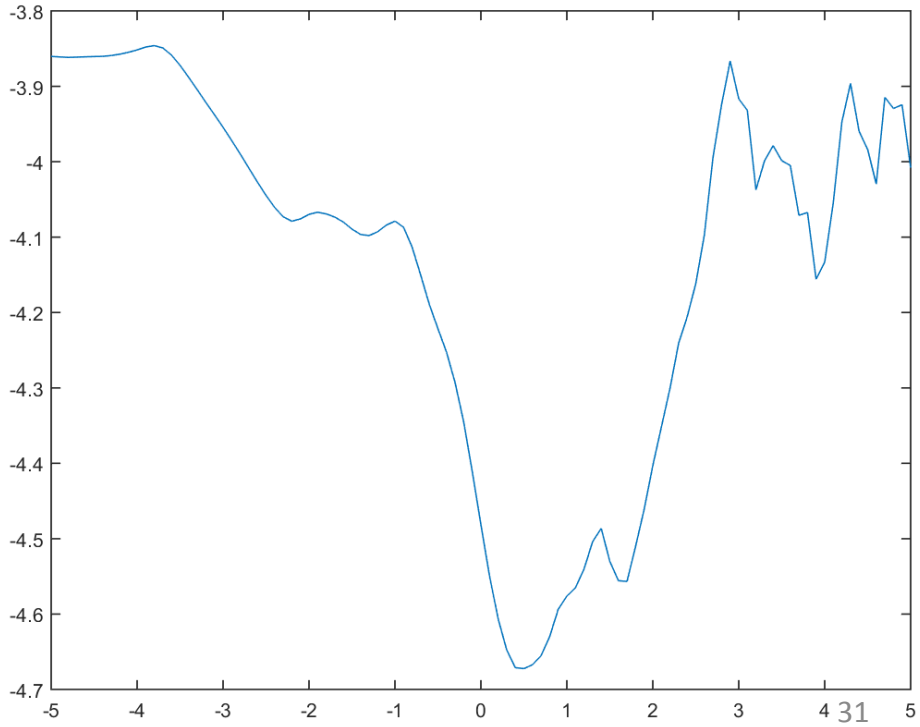
1D Convolution Examples

- Averaging convolution computes local mean:

$$w = [\,1/3 \quad 1/3 \quad 1/3\,]$$



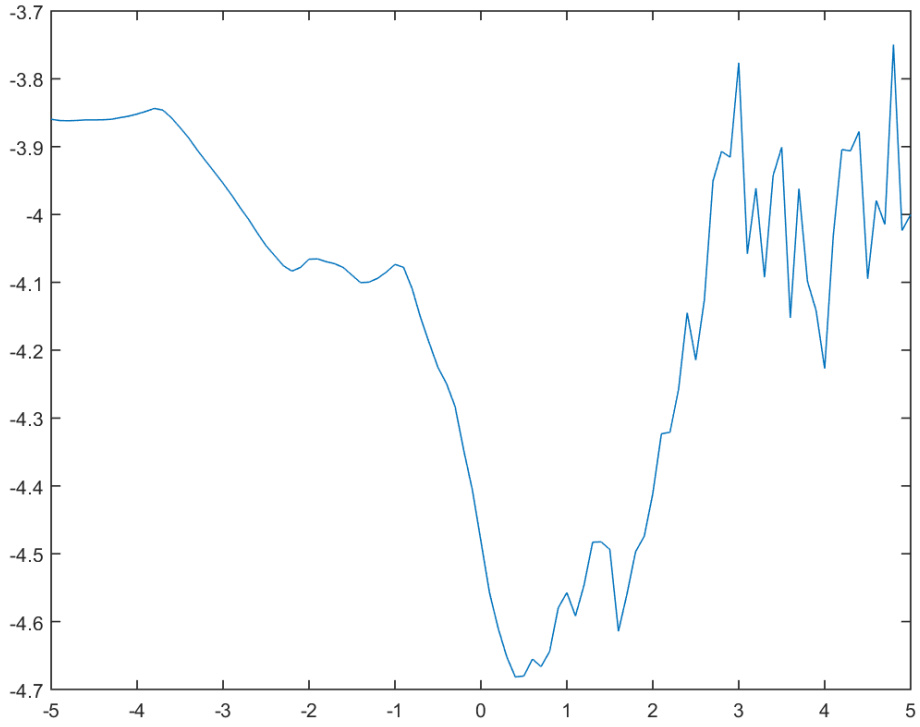


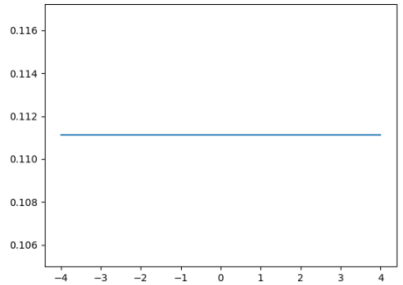


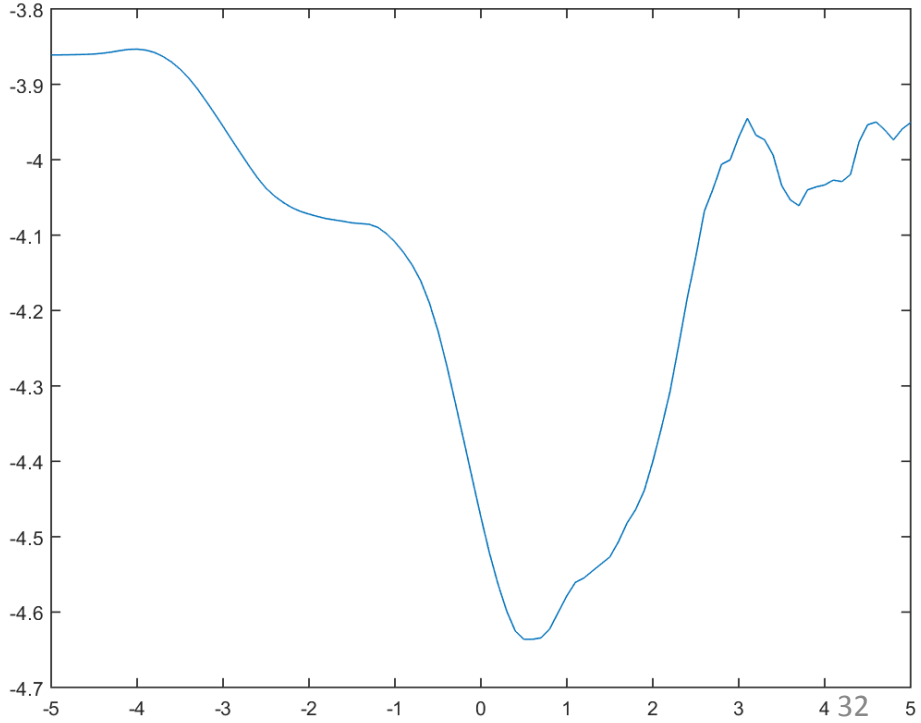
1D Convolution Examples

- Averaging over bigger window gives coarser view of signal:

$$w = [1/9 \quad 1/9 \quad 1/9 \quad 1/9 \quad 1/9 \quad 1/9 \quad 1/9 \quad 1/9 \quad 1/9]$$





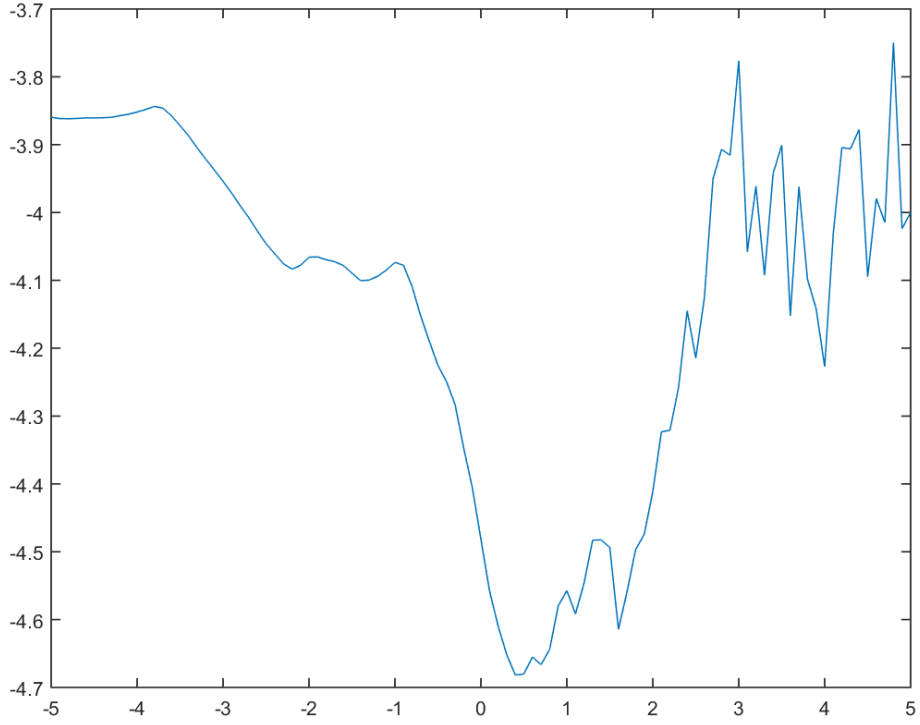


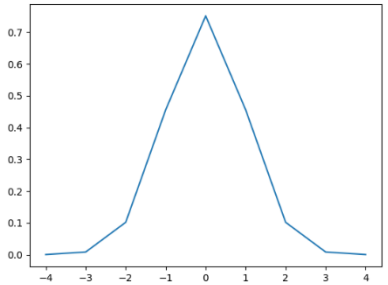
1D Convolution Examples

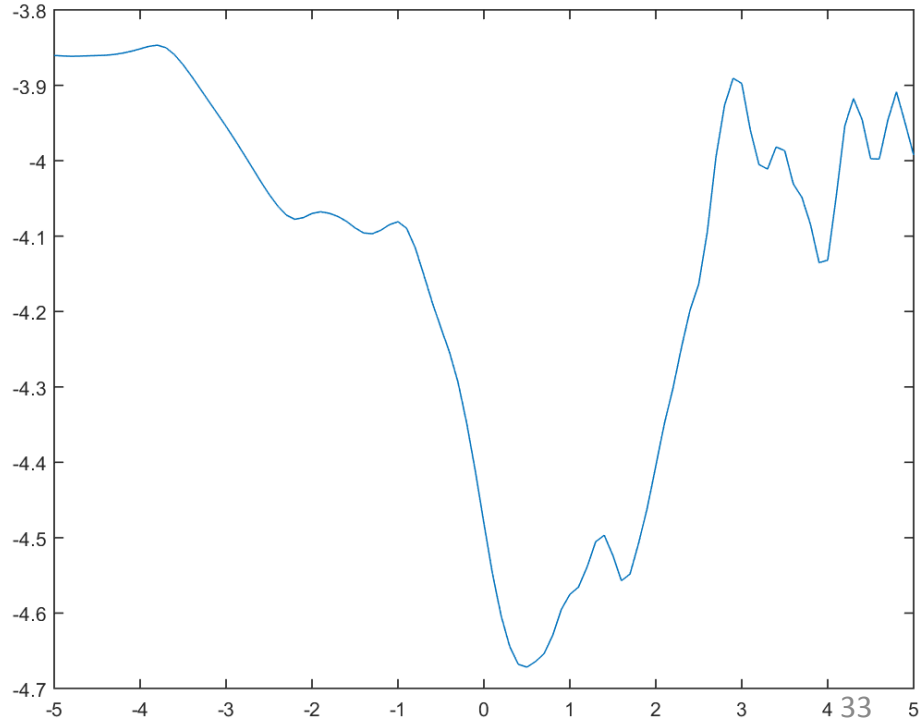
- Gaussian convolution blurs signal: $w_i \propto \exp\left(-\frac{i^2}{2\sigma^2}\right)$
  - Compared to averaging it's more smooth and maintains peaks better.

$W = [0.0001 \;\; 0.0044 \;\; 0.0540 \;\; 0.2420 \;\; 0.3989 \;\; 0.2420 \;\; 0.0540 \;\; 0.0044 \;\; 0.0001]$

$(\sigma = 1, m = 4)$



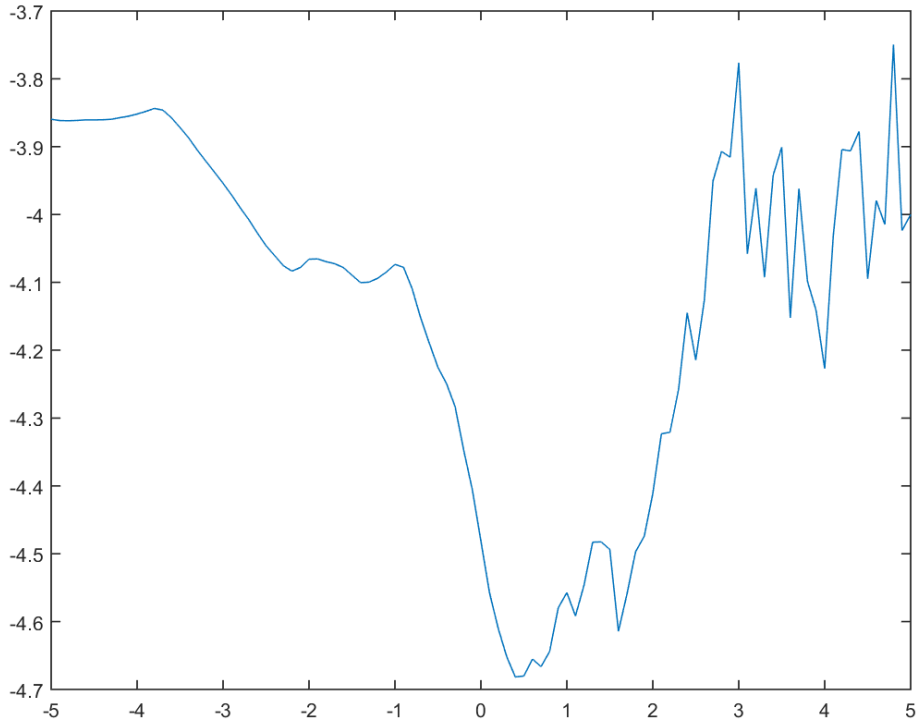


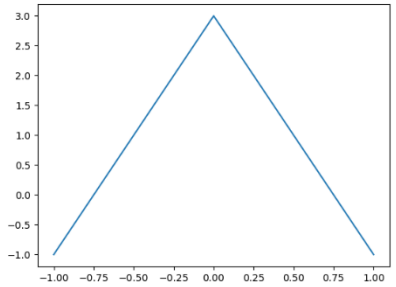


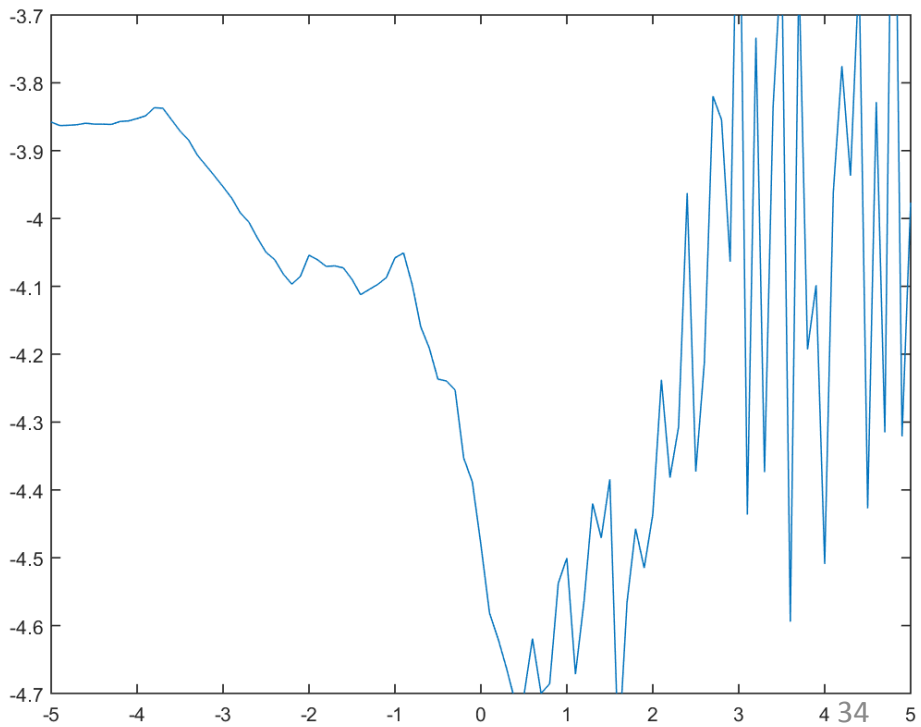
1D Convolution Examples

- Sharpen convolution enhances peaks.
  - An "average" that places negative weights on the surrounding pixels.

$$w = [-1 \quad 3 \quad -1]$$



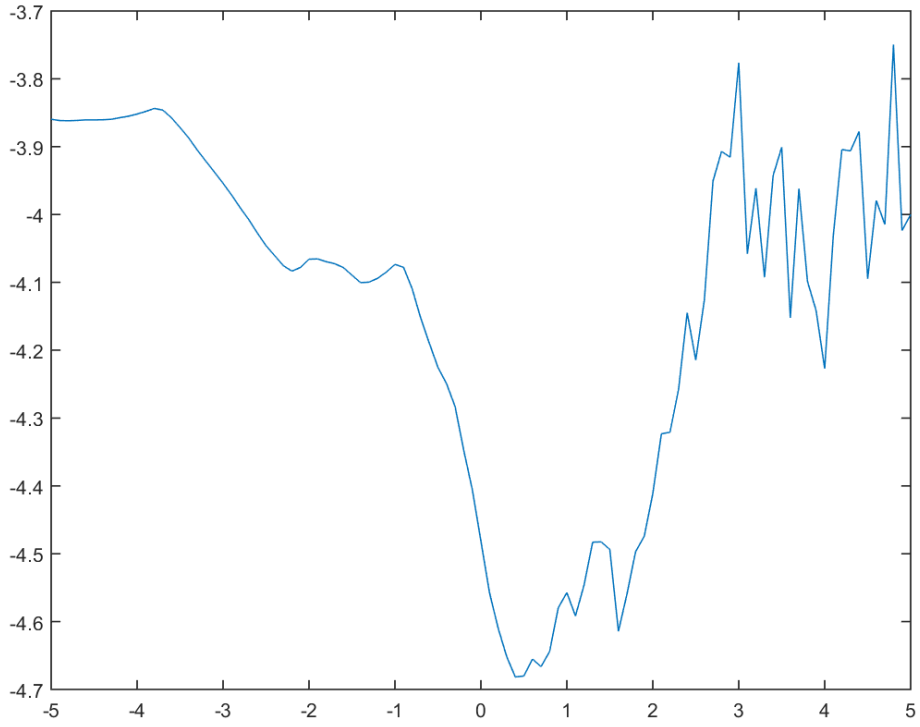


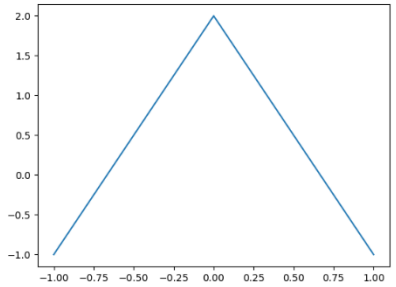


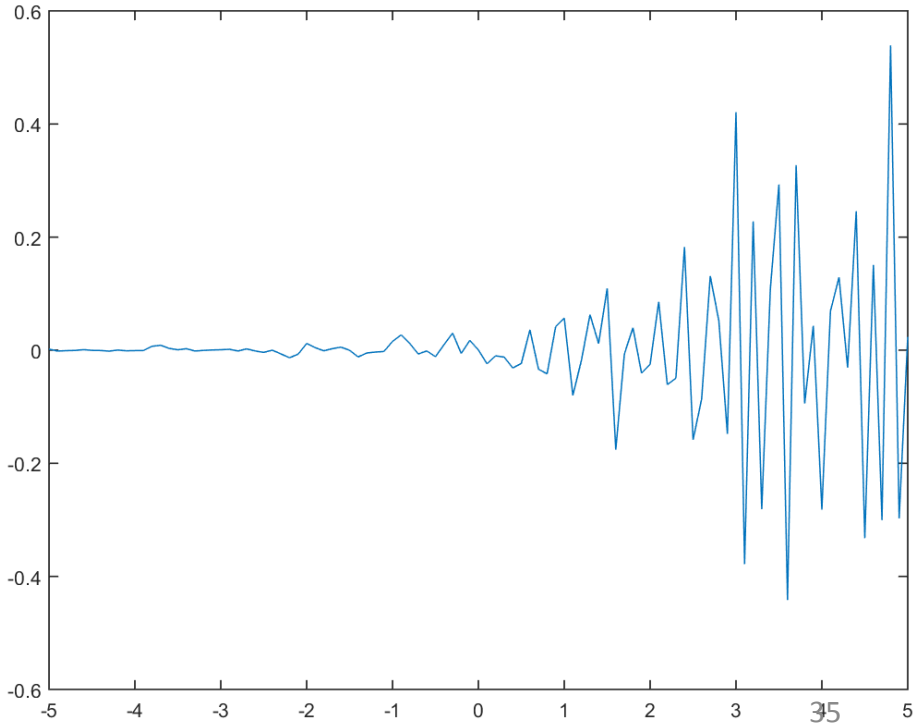
1D Convolution Examples

- Laplacian convolution approximates second derivative:
  - "Sum to zero" filters "respond" if input vector looks like the filter

$$w = [-1 \quad 2 \quad -1]$$

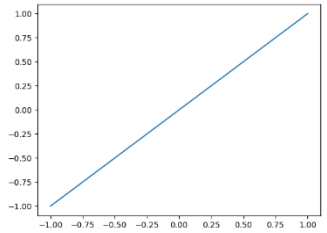




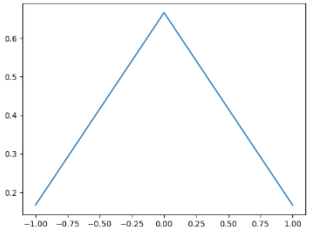


Digression: Derivatives and Integrals

- Numerical derivative approximations can be viewed as filters:
  - Centered difference: [-1, 0, 1] .
  - Gradient checkers often use forward difference: [-1, 1]



- Numerical integration approximations can be viewed as filters:
  - "Simpson's" rule: [1/6, 4/6, 1/6] (a bit like Gaussian filter).



- Derivative filters add to 0, integration filters add to 1,
  - For constant function, derivative should be 0 and average = constant.

1D Convolution Examples

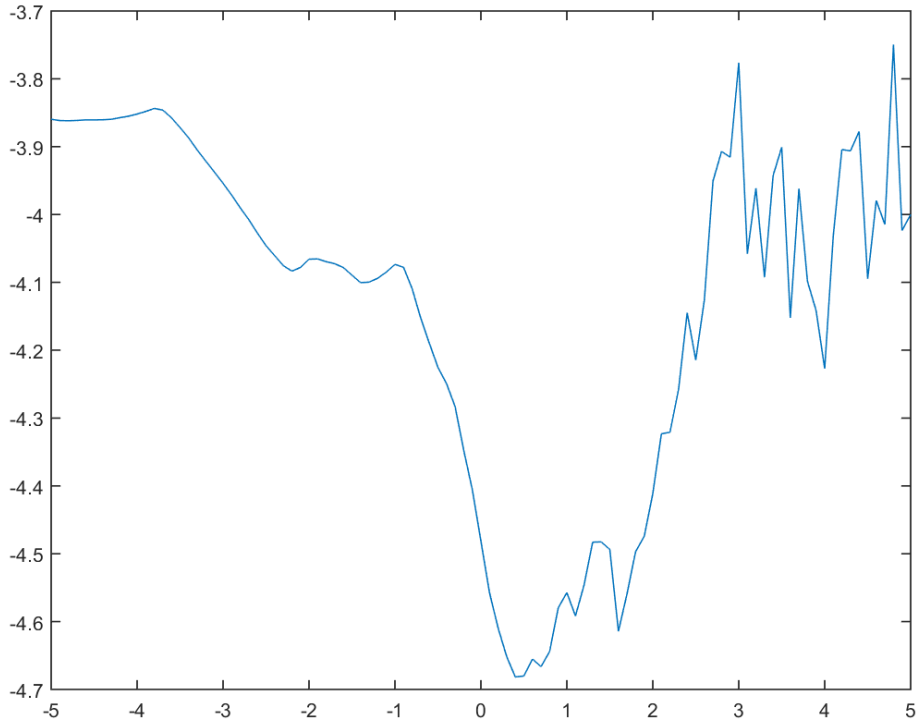
- Laplacian of Gaussian is a smoothed 2nd-derivative approximation:

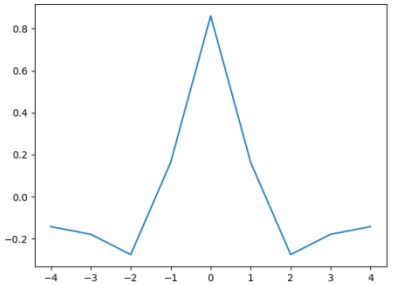
$$w_i = \left(1 - \frac{i^2}{2\sigma^2}\right) \exp\left(-\frac{i^2}{2\sigma^2}\right)$$

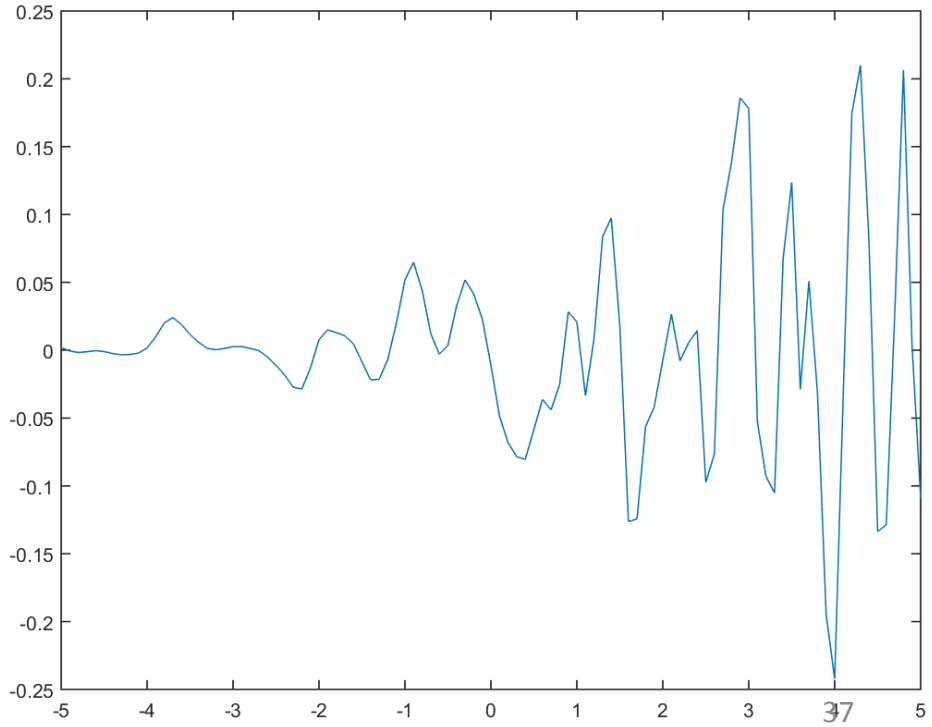
(then subtract mean)

$w = [-0.1416 \ -0.1781 \ -0.2746 \ 0.1640 \ 0.8607 \ 0.1640 \ -0.2746 \ -0.1781 \ -0.1416]$

$(\sigma^2 = 1, \ m = 4)$







Summary

- Backpropagation computes neural network gradient via chain rule.
- Parameter initialization is crucial to neural net performance.
- Optimization and step size are crucial to neural net performance.
- Regularization is crucial to neural net performance:
  - L2-regularizaiton, early stopping, dropout.
- Convolutions are linear operators that capture local information

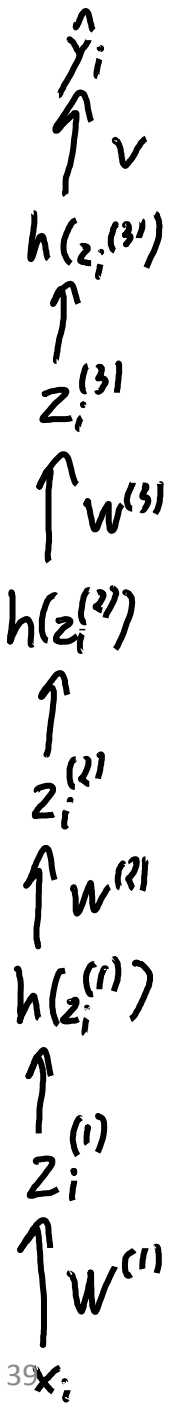
Backpropagation

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

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

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

$$\frac{\partial f}{\partial w^{(3)}} = (\hat{y}_i - y_i)\, v\, h'(w^{(3)} h(w^{(2)} h(w^{(1)} x_i)))\, h(w^{(2)} h(w^{(1)} x_i)) = (\hat{y}_i - y_i)\, v\, h'(z_i^{(3)})\, h(z_i^{(2)})$$



Backpropagation

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

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

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

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

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

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

$r^{(2)}$ → "backprop"

Backpropagation

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

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

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

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

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

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

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

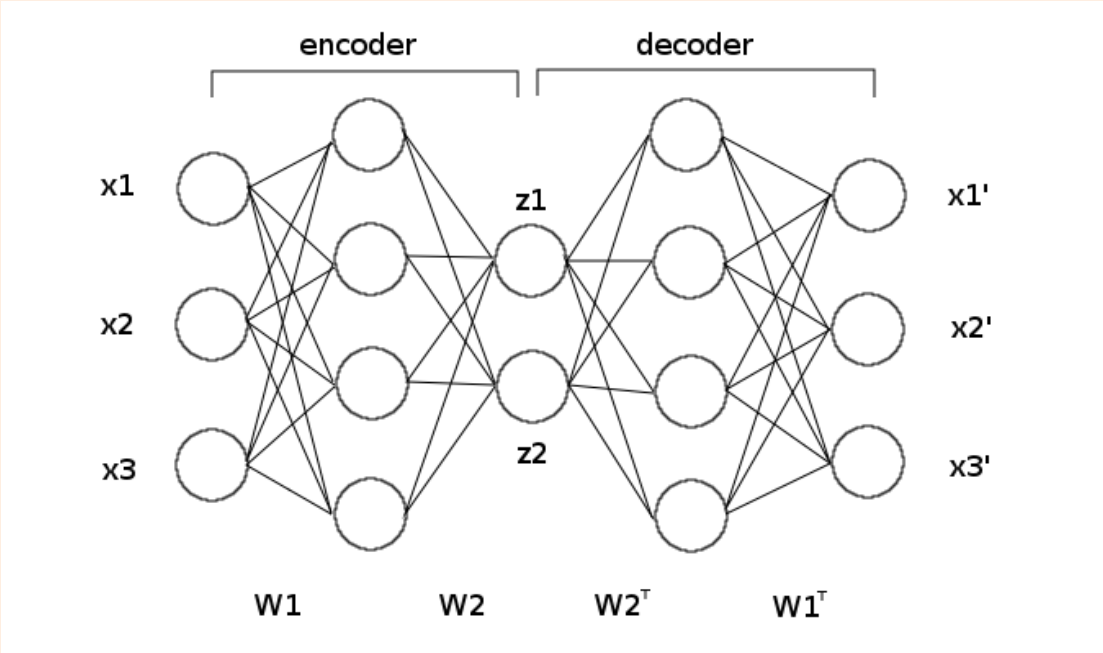
$$\frac{\partial f}{\partial w_{c'c}^{(2)}} = \left[\sum_{c''=1}^{k} r_{c''}^{(3)} W_{c''c'}^{(3)}\right] h'(z_{ic'}^{(2)})\, h(z_{ic}^{(1)})$$

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

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

Autoencoders

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

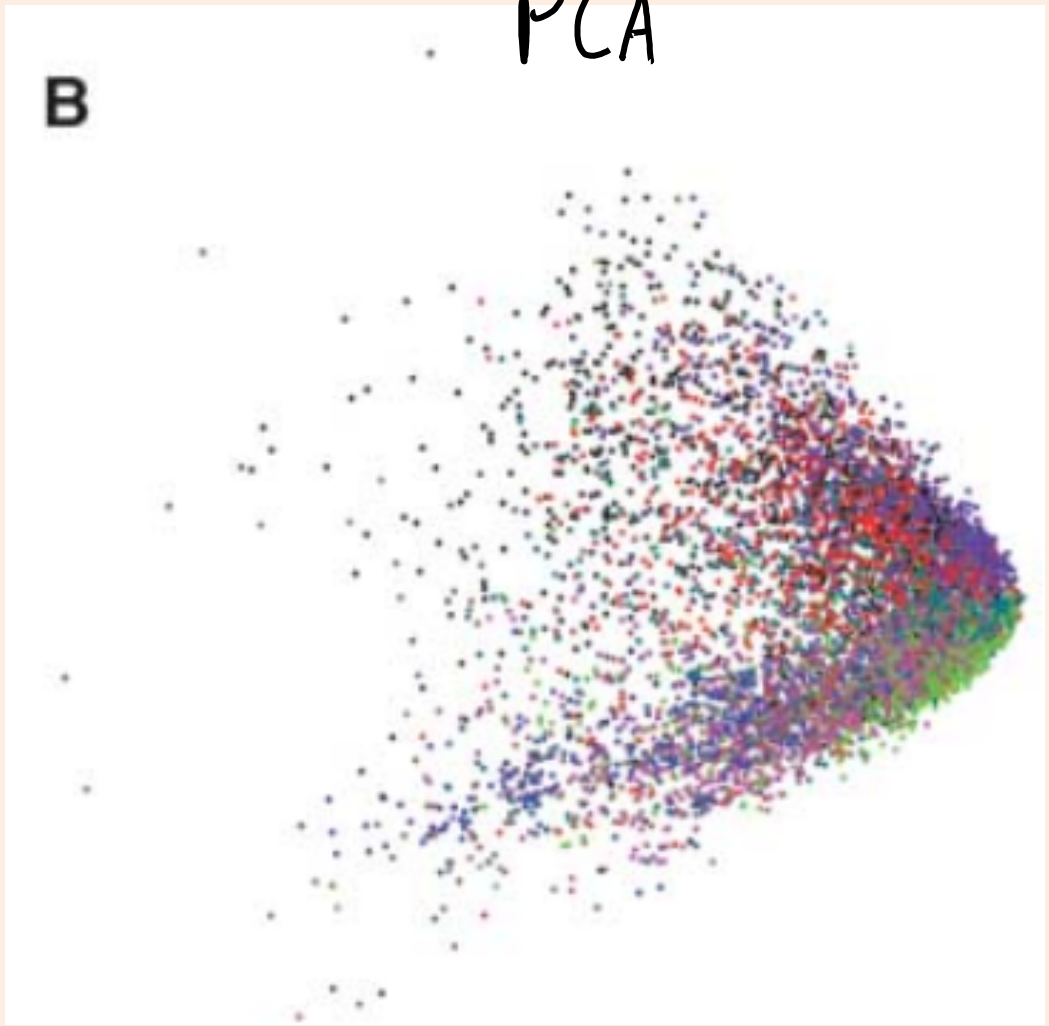


  - Middle layer could be latent features in non-linear latent-factor model.
    - Can do outlier detection, data compression, visualization, etc.
  - A non-linear generalization of PCA.

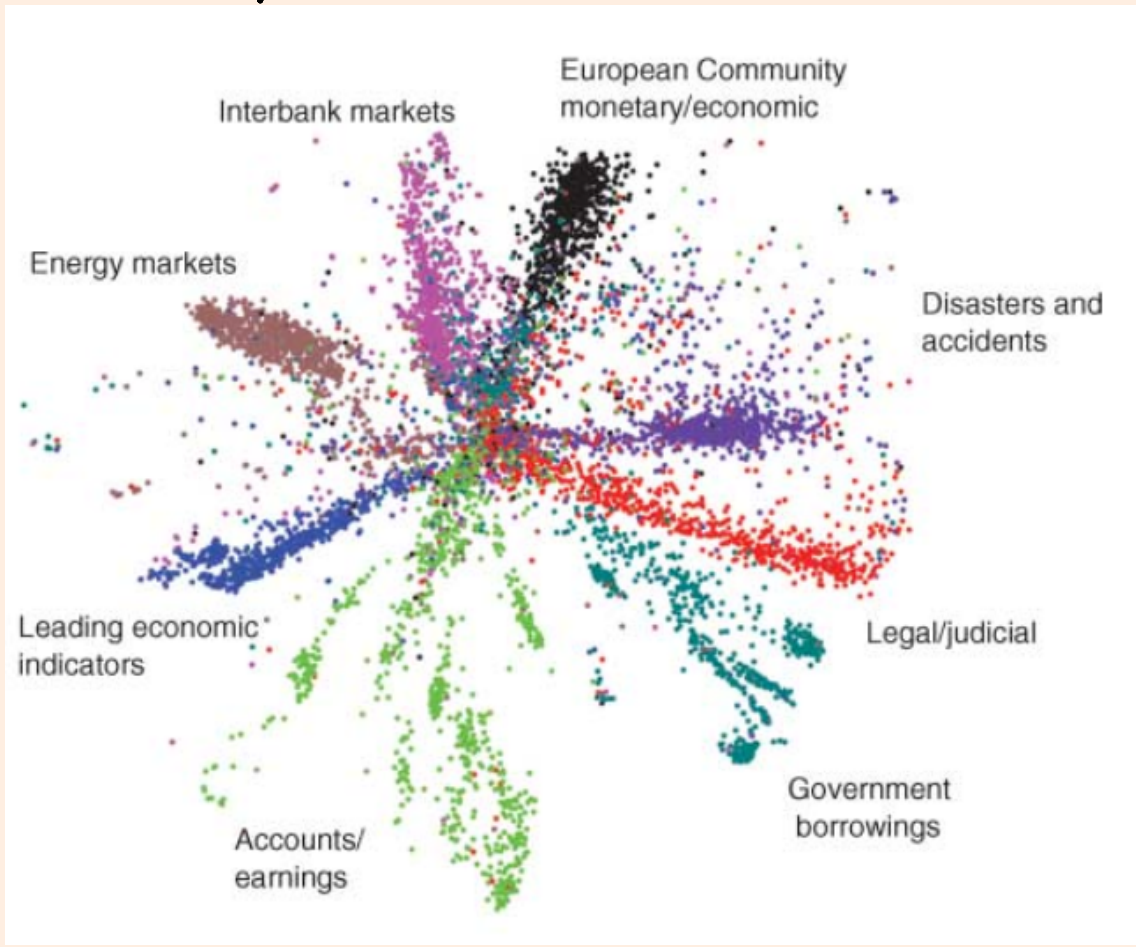
http://inspirehep.net/record/1252540/plots

Autoencoders

PCA



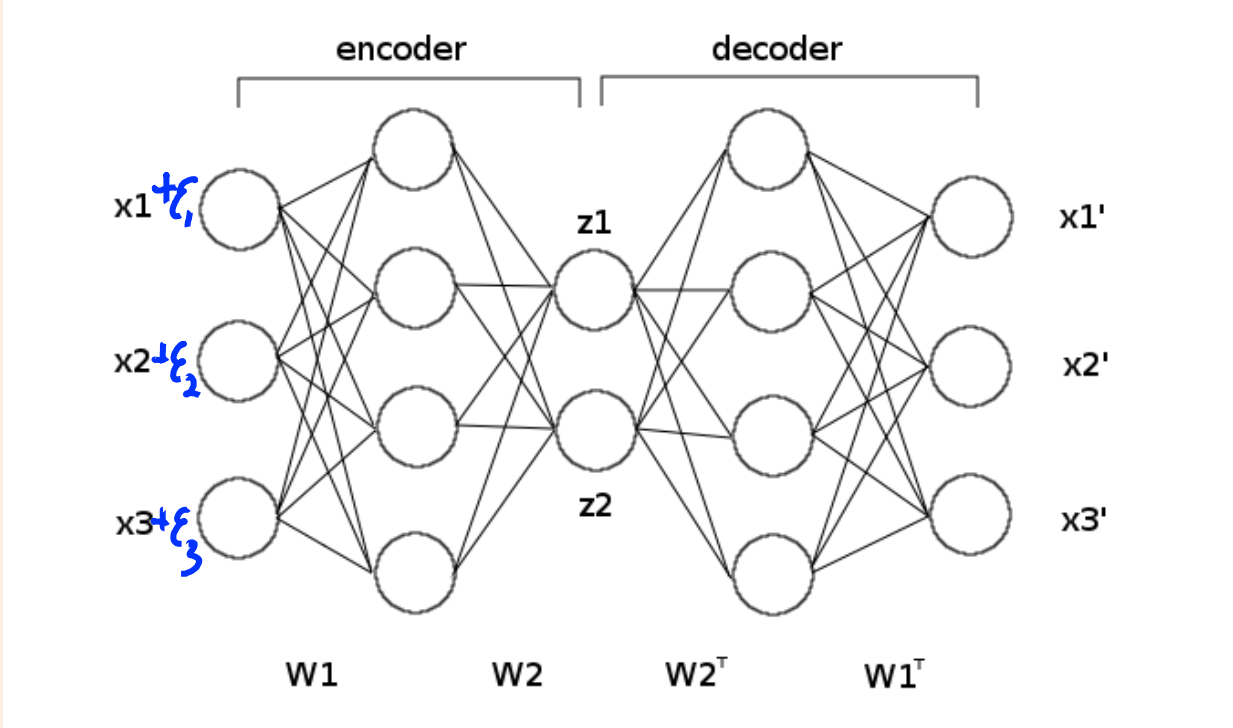
Autoencoder



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

Denoising Autoencoder

- Denoising autoencoders add noise to the input:



  - Learns a model that can remove the noise.

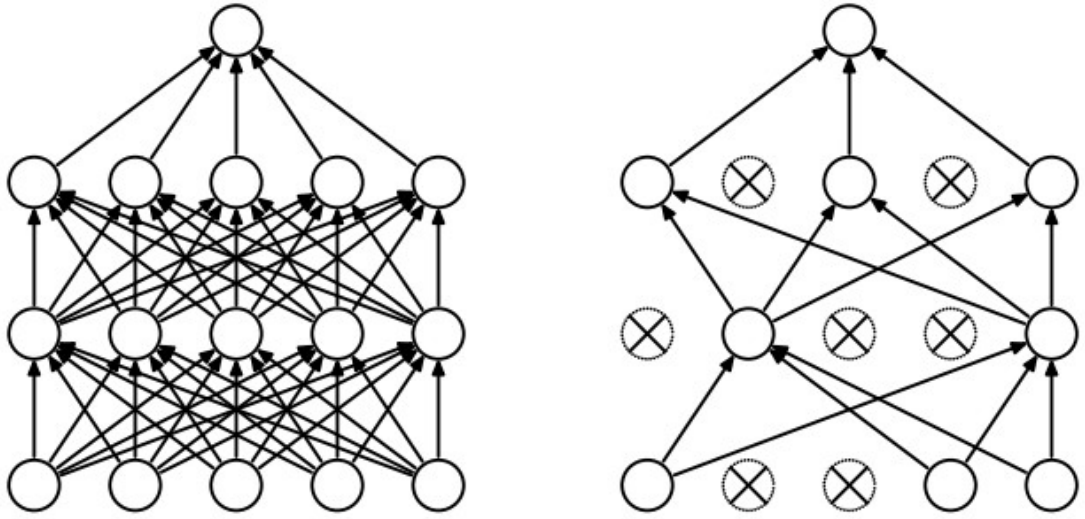
http://inspirehep.net/record/1252540/plots

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 standardize data:
  - Subtract mean, divide by standard deviation, "whiten", standardize y_i.
- More recent initializations try to standardize initial z_i:
  - Use different initialization in each layer.
  - Try to make variance of z_i the same across layers.
  - Use samples from standard normal distribution, divide by sqrt(2*nInputs).
  - Use samples from uniform distribution on [-b,b], where $b = \frac{\sqrt{6}}{\sqrt{k^{(m)} + k^{(m-1)}}}$

Dropout

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



(a) Standard Neural Net (b) After applying dropout.

  - Encourages distributed representation rather than using specific z_i.
  - Like ensembling a lot of models but without the high computational cost.
  - After a lot of success, dropout may already be going out of fashion.

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