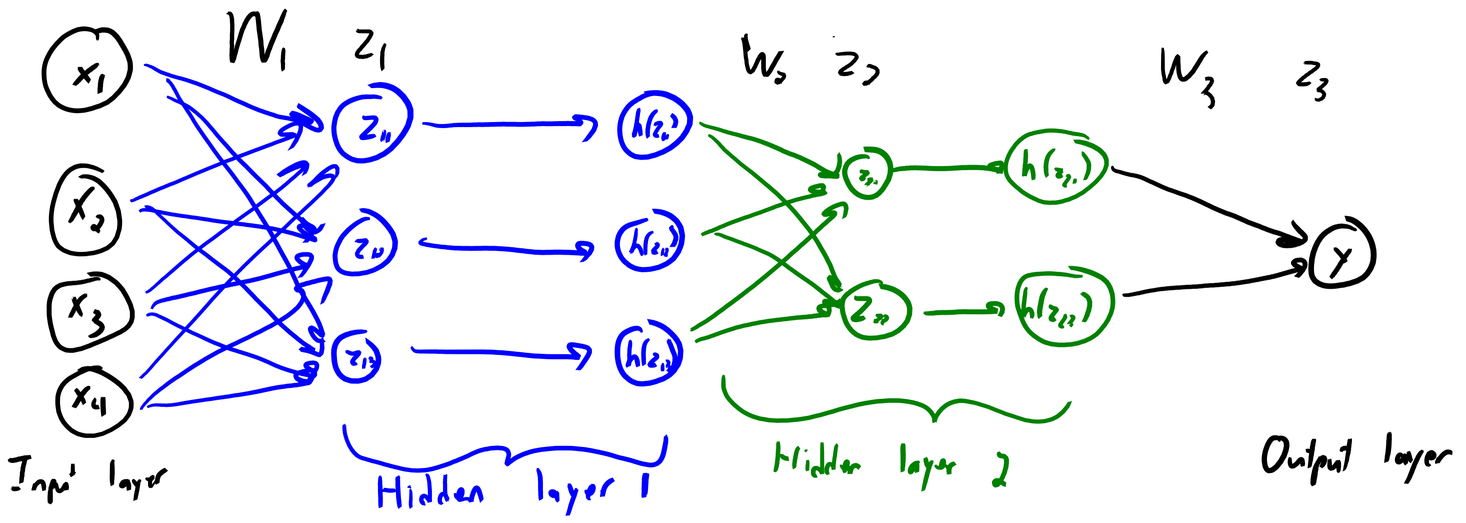
CPSC 440: Machine Learning

Automatic Differentiation

Winter 2022

Last Time: Deep Neural Networks

- We discussed deep neural networks (more than 1 hidden layer):



- Typically alternate between linear and non-linear transformations:

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

  - Prediction cost is dominated by multiplications by the W^l matrices.
- We discussed the vanishing gradient problem.
  - As the network gets deeper, the gradients can become arbitrarily small.
  - Can be reduced by using ReLU instead of sigmoid, or using skip connections.

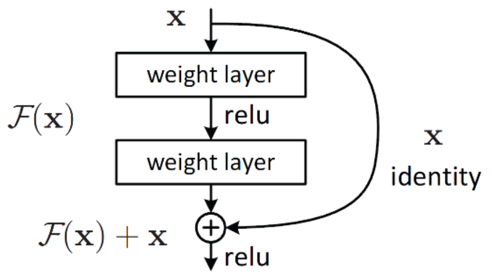
ResNet "Blocks"

- 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:

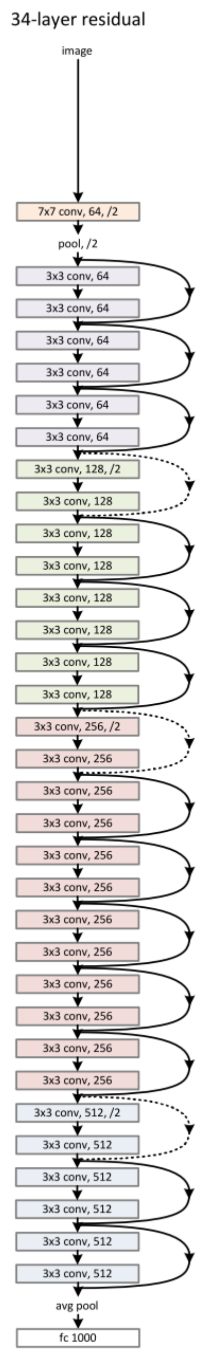
$$a^{l+2} = h(W^{l+1} h(W^l a^l))$$

↑ "activation at layer 'l'

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



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



https://en.wikipedia.org/wiki/Residual_neural_network
https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035

DenseNet

- More recent 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.

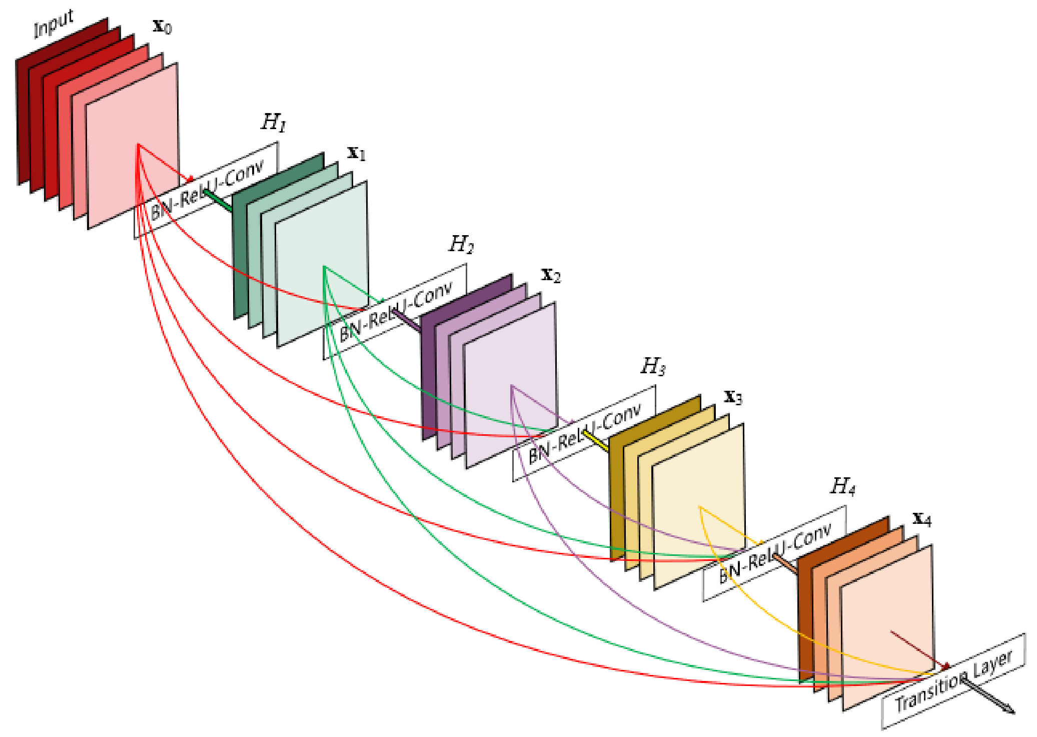


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

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!

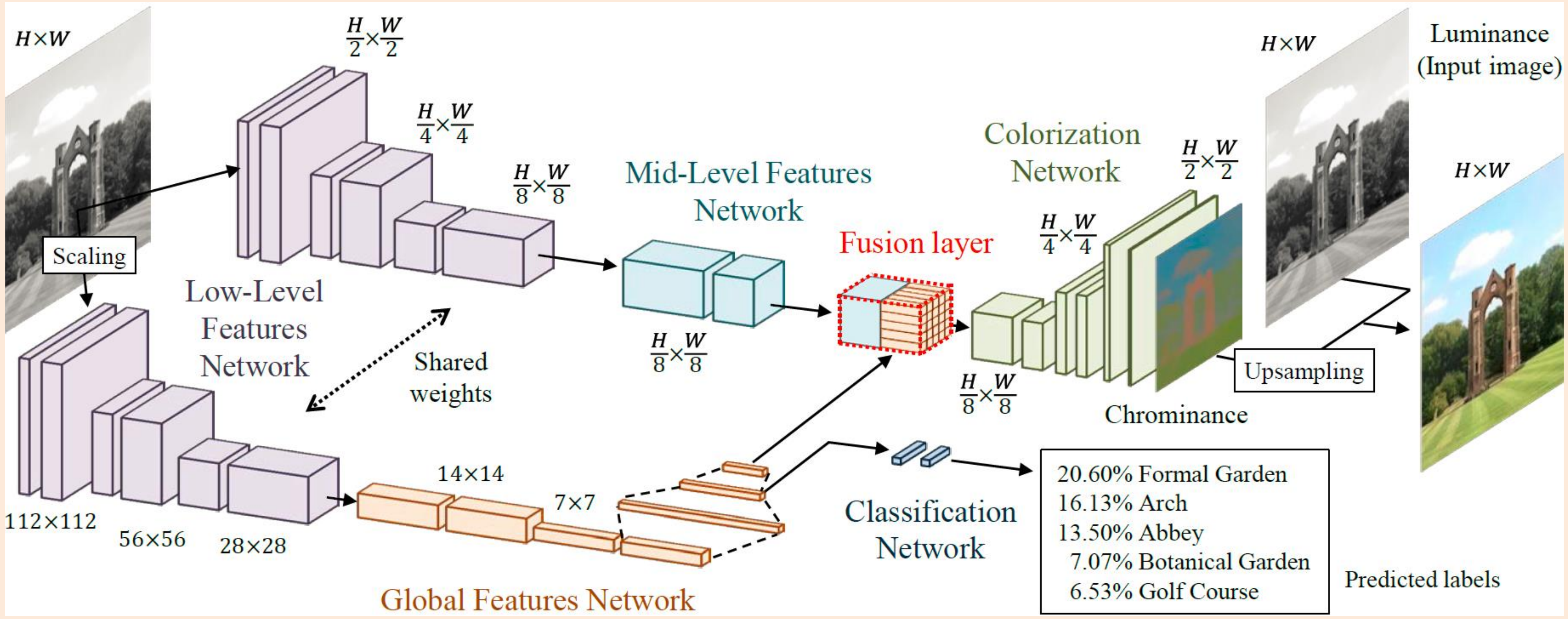
Missing Theory Behind Training Deep Networks

- 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 defined in original paper, 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.

Next Topic: Automatic Differentiation

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.

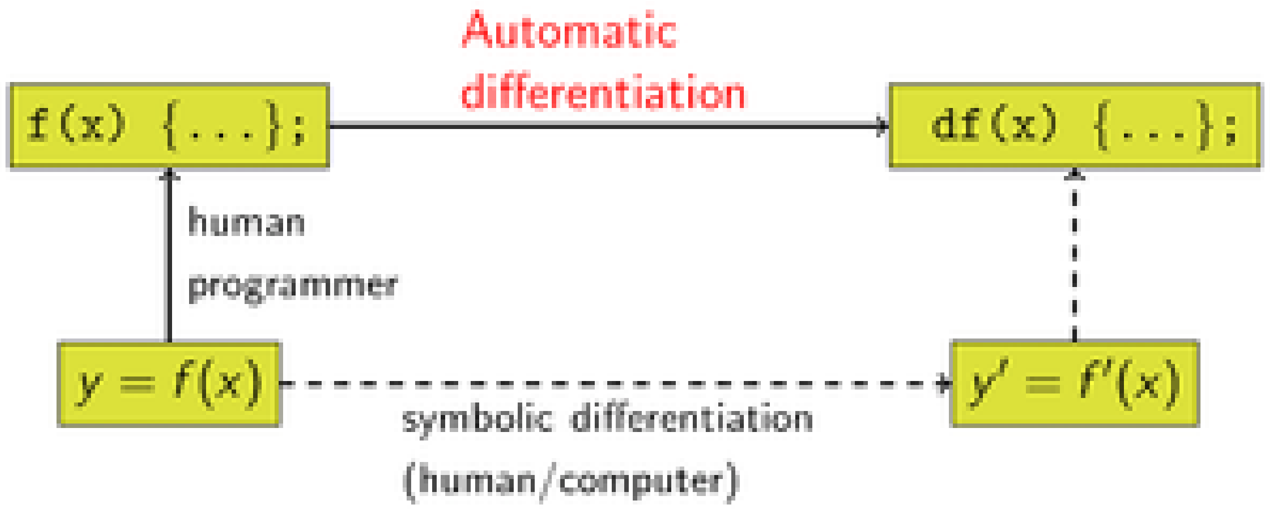


- Developing networks like this is made easier using automatic differentiation.

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

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: $f_5(f_4(f_3(f_2(f_1(x)))))$.
  - 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:
  - $f'(x) = f_5'(f_4(f_3(f_2(f_1(x)))))*f_4'(f_3(f_2(f_1(x))))*f_3'(f_2(f_1(x)))*f_2'(f_1(x))f_1'(x)$.

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

Automatic Differentiation – Single Input+Output

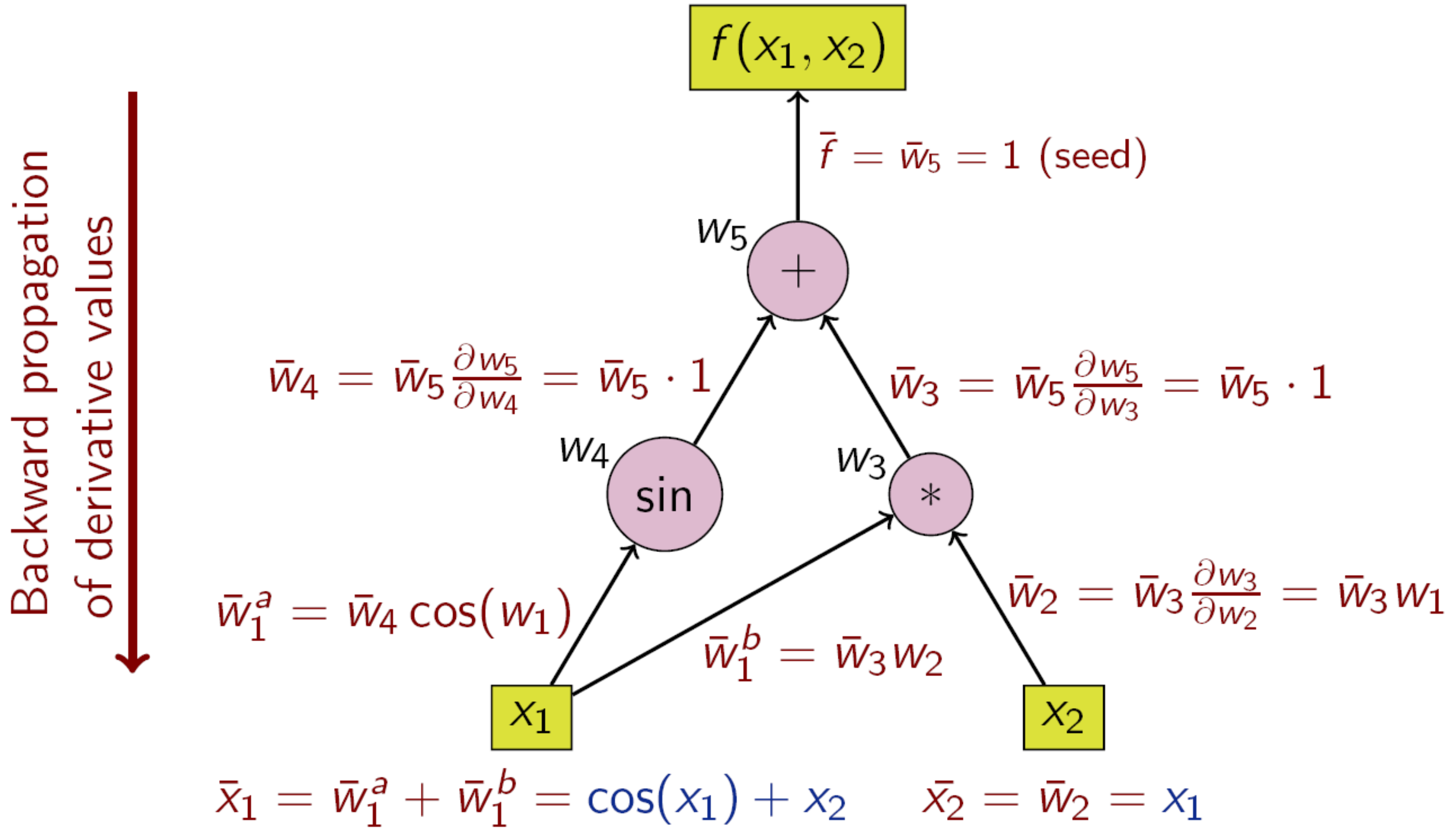
- Our function written as a set of compositions:
  - $f_5(f_4(f_3(f_2(f_1(x)))))$.
- The derivative written using the chain rule::
  - $f'(x) = f_5'(f_4(f_3(f_2(f_1(x)))))*f_4'(f_3(f_2(f_1(x))))*f_3'(f_2(f_1(x)))*f_2'(f_1(x))f_1'(x)$.
- Notice that this leads to repeated calculations.
  - For example,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



$f(x_1, x_2)$
$\bar{f} = \bar{w}_5 = 1$ (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
$\bar{x}_1 = \bar{w}_1^a + \bar{w}_1^b = \cos(x_1) + x_2$
$\bar{x}_2 = \bar{w}_2 = x_1$
Backward propagation of derivative values

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.

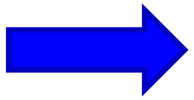
- But reverse-mode AD has some drawbacks:
  - Need to store all intermediate calculations, so requires a lot of storage.
    - For basic deep neural networks, hand-written code would only need to store the activations.
      - Modern codes have some of these space savings built in.
    - For other functions, 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 can be computed at a lower cost than the function value.
  - May miss opportunities for parallelism, or miss tricks to avoid numerically 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 do things like “take the derivative of a function of a sample from the distribution”.

Next Topic: Convolutional Neural Networks

Motivation: X-Ray Abnormality Detection

- Want to build a system that recognizes abnormalities in x-rays:



"Abnormality detected"
(binary classification)

- Applications:
  - Fast detection of tuberculosis, pneumonia, lung cancer, and so on.
- Deep learning has led to incredible progress on computer vision tasks.
  - Much of this progress has been driven by convolutional neural networks (CNNs).

Convolutional Neural Network (CNN) Motivation

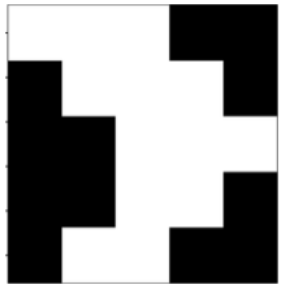
- Consider training neural networks on 500 pixel by 500 pixel images.
  - So the number of inputs 'd' to first layer is 250,000 inputs.
- If first layer has k=10,000, then 'W' has 2.5 billion parameters.
  - We want to avoid this huge number (due to storage and overfitting).

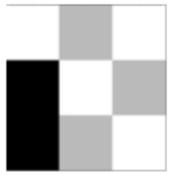
- CNNs drastically reduce the number of parameters by:
  - Having activations only depend on a small number of inputs.
  - Using the same parameters on the connections of many activations.
- Done using layers that look like "convolutions" in signal processing.

Illustration of 2D Convolution

- 2D convolution:
  - Inputs: an “input” image ‘x’ and a “filter” image ‘w’.
  - Output: new image ‘z’ (pixels are dot products of filter and image region).

Input image **Filter image** **Output image**





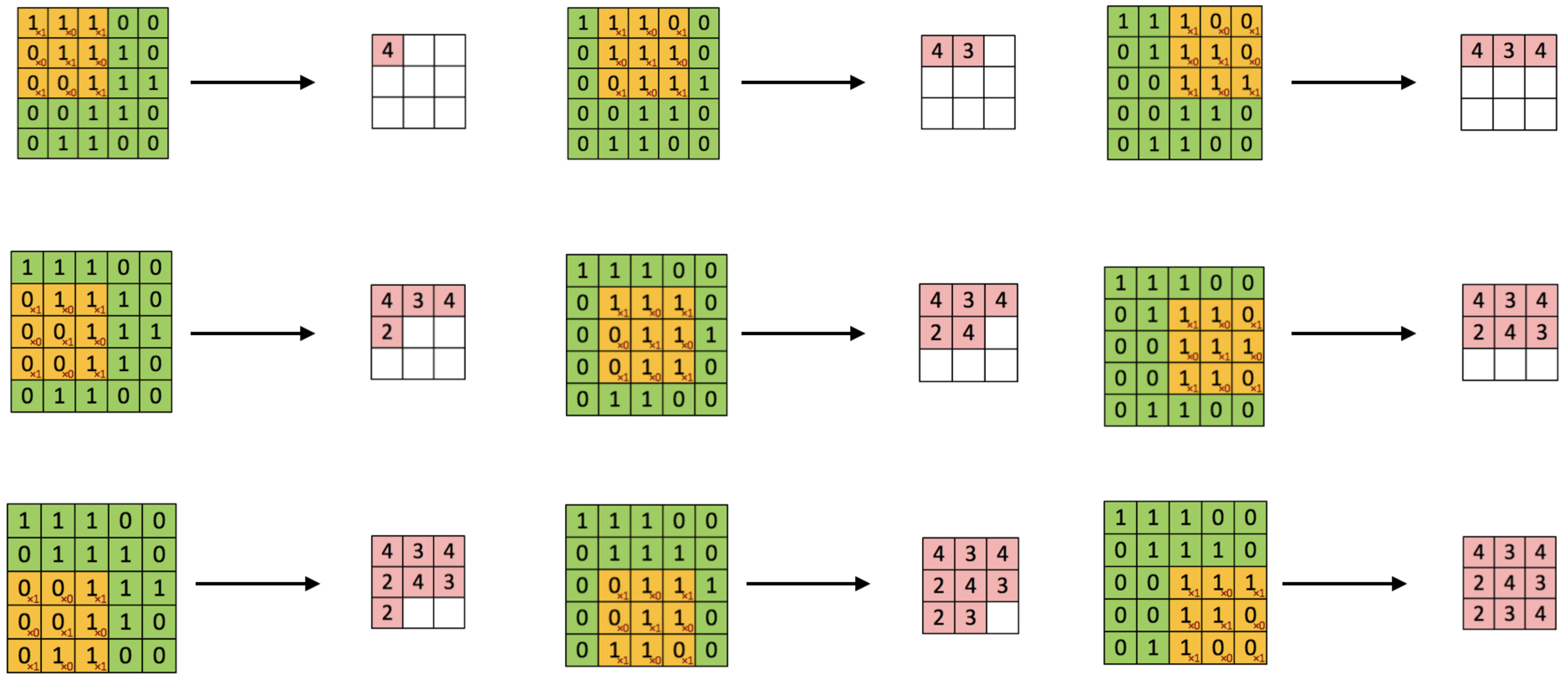
$$\underset{x}{\begin{array}{|c|c|c|c|c|}\hline 1&1&1&0&0\\\hline 0&1&1&1&0\\\hline 0&0&1&1&1\\\hline 0&0&1&1&0\\\hline 0&1&1&0&0\\\hline\end{array}} * \underset{w}{\begin{array}{|c|c|c|}\hline 1&0&1\\\hline 0&1&0\\\hline 1&0&1\\\hline\end{array}} = \underset{z}{\begin{array}{|c|c|c|}\hline 4&3&4\\\hline 2&4&3\\\hline 2&3&4\\\hline\end{array}}$$

Illustration of 2D Convolution

- 2D convolution:
  - Inputs: an “input” image ‘x’ and a “filter” image ‘w’.
  - Output: new image ‘z’ (pixels are dot products of filter and image region).

Filter image

1	0	1
0	1	0
1	0	1

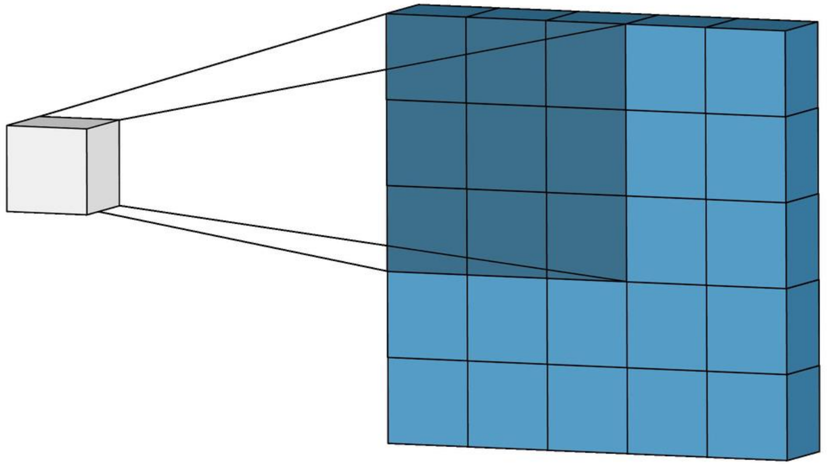


https://scientistcafe.com/ids/convolutional-neural-network.html

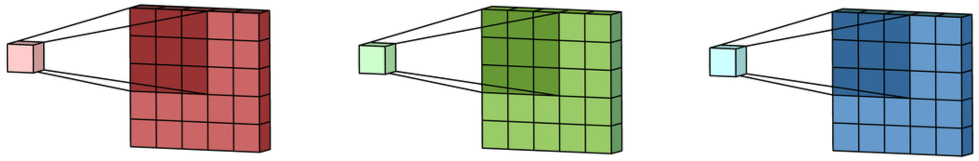
Illustration of 2D Convolution

- 2D convolution:
  - Inputs: an "input" image 'x' and a "filter" image 'w'.
  - Output: new image 'z' (pixel is dot product of filter and image region).
- As a formula:

$$z[i_1, i_2] = \sum_{j_1=-m}^{m} \sum_{j_2=-m}^{m} w[j_1, j_2]\, x[i_1 + j_1, i_2 + j_2]$$



  - Final Image 'z' can be written as usual z=Wx.
    - 'W' will be sparse, and have filter values repeated.
- 3D convolution (for colour images):
  - Weighted dot product across all three dimensions.



Summary

- 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.
  - But has some disadvantages compared to human-written code.
- Convolutions are flexible class of signal/image transformations.

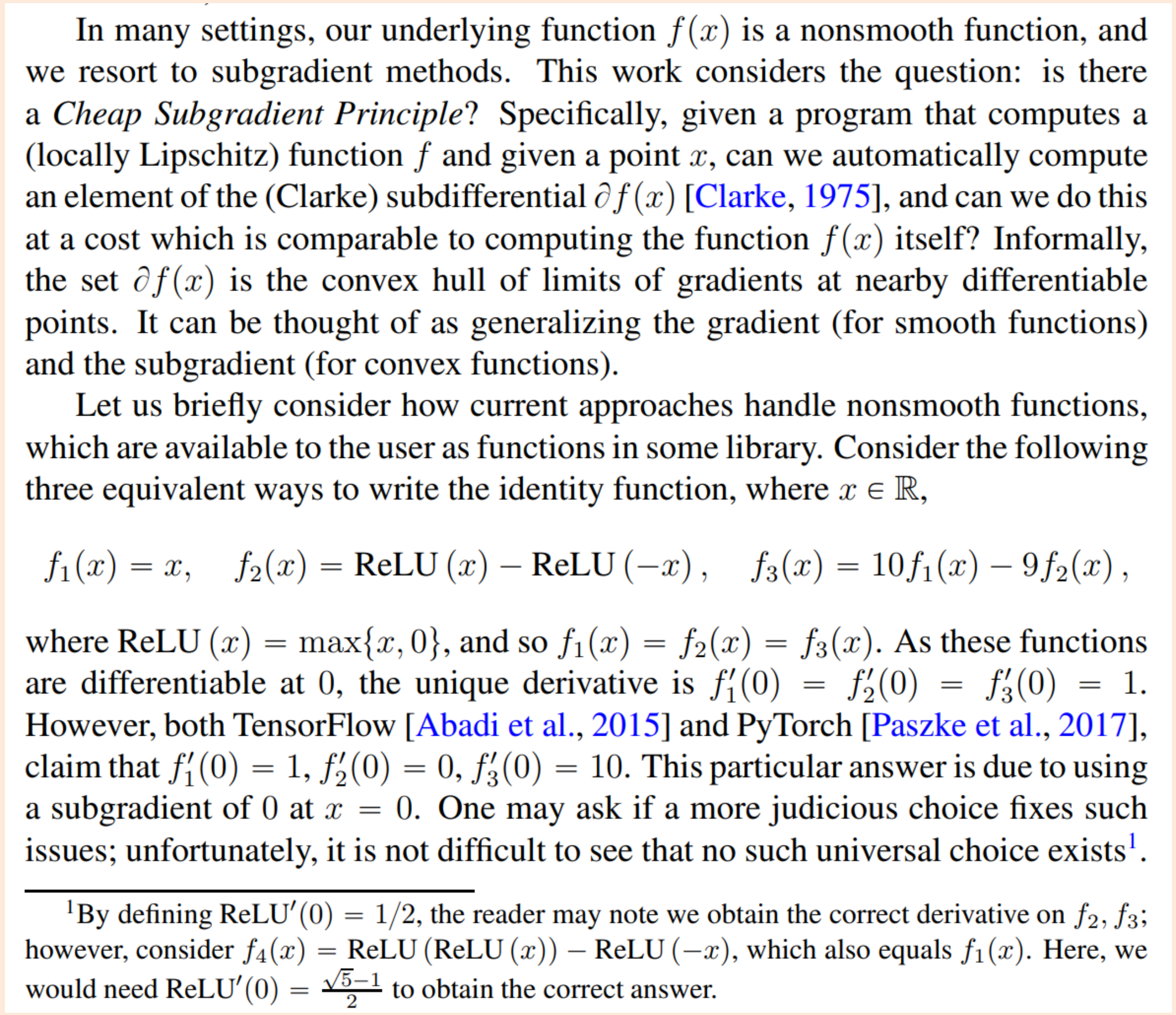
- Next time: convolutions for unprecedented vision performance.

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.

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

Formal Convolution Definition

- We have defined the convolution as:

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

- In other classes you may see it defined as:

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

(reverses 'w')

$$z_i = \int_{-\infty}^{\infty} w_j x_{i-j} \, dj$$

(assumes signal + filter are continuous)

- For simplicity we're skipping the "reverse" step, and assuming 'w' and 'x' are sampled at discrete points (not functions).
- But keep this mind if you read about convolutions elsewhere.