Sandia National Laboratories

Exceptional service in the national interest

SCIENTIFIC MACHINE LEARNING AND TENSORFLOW TUTORIAL

Neural Networks

Ravi G. Patel

Scientific Machine Learning Department

February 1 – 2, 2024

Numerical PDEs: Analysis, Algorithms, and Data Challenges

ICERM

Brown University

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

WHAT ARE NEURAL NETWORKS?

- Neural networks are a type of “brain inspired” function approximation scheme

- A single “hidden layer” network is the composition of an elementwise nonlinearity sandwiched between two affine transformations

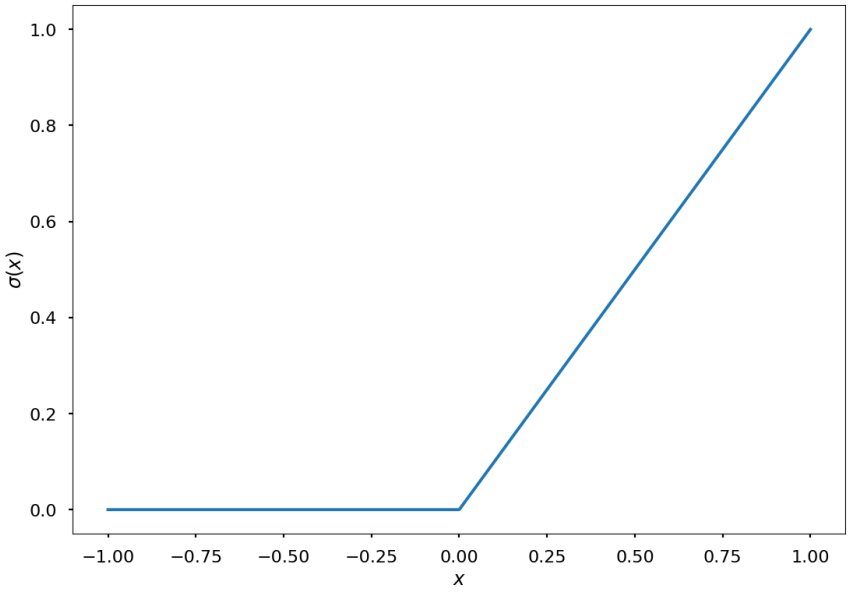
$$x_1 = L_1(x_0) = \sigma(W_1 x_0 + b_1)$$
$$x_2 = L_2(x_1) = W_2 x_1 + b_2$$
$$x_0 \in \mathbb{R}^{n_0}, \quad x_1 \in \mathbb{R}^{n_1}, \quad x_2 \in \mathbb{R}^{n_2}$$

W_i : weights

b_i : biases

σ : activiation function,

e.g., $\mathrm{ReLU}(x) = \max(0, x)$



- By choosing the right the weights and biases, neural networks can approximate any function

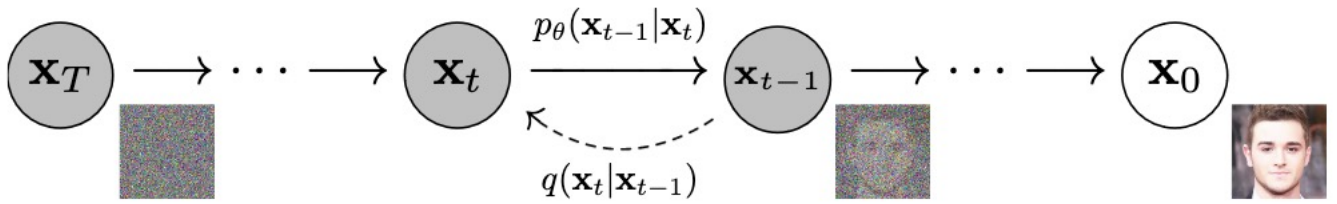
WHAT ARE DEEP NEURAL NETWORKS (DNN'S)?

- Deep neural networks compose these layers together,

$$x_n = L_n \sigma L_{n-1} \sigma L_{n-2} \ldots \sigma L_1 x_0$$
$$x_n = N(x_0, \xi)$$
$$\xi = \{W_1, \ldots, W_N, b_1, \ldots, b_N\}$$

WHY USE NEURAL NETWORKS?

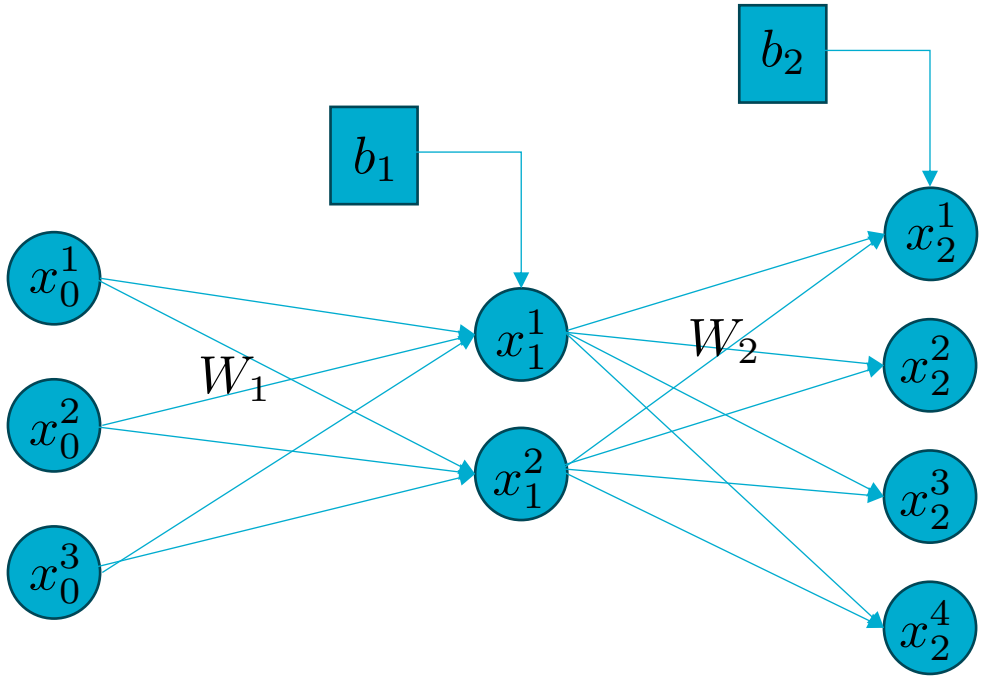
- They tend to be excellent high dimensional approximators
  - Mostly empirical evidence
  - Some theoretical results
    - A. Choromanska et al., *AISTATS* (2015)
- Simple to implement
- Efficiently runs on GPU's
- Specialized hardware



Diffusion models use neural networks to learn denoising maps for images
J. Ho et al., *NeurIPS* (2020)

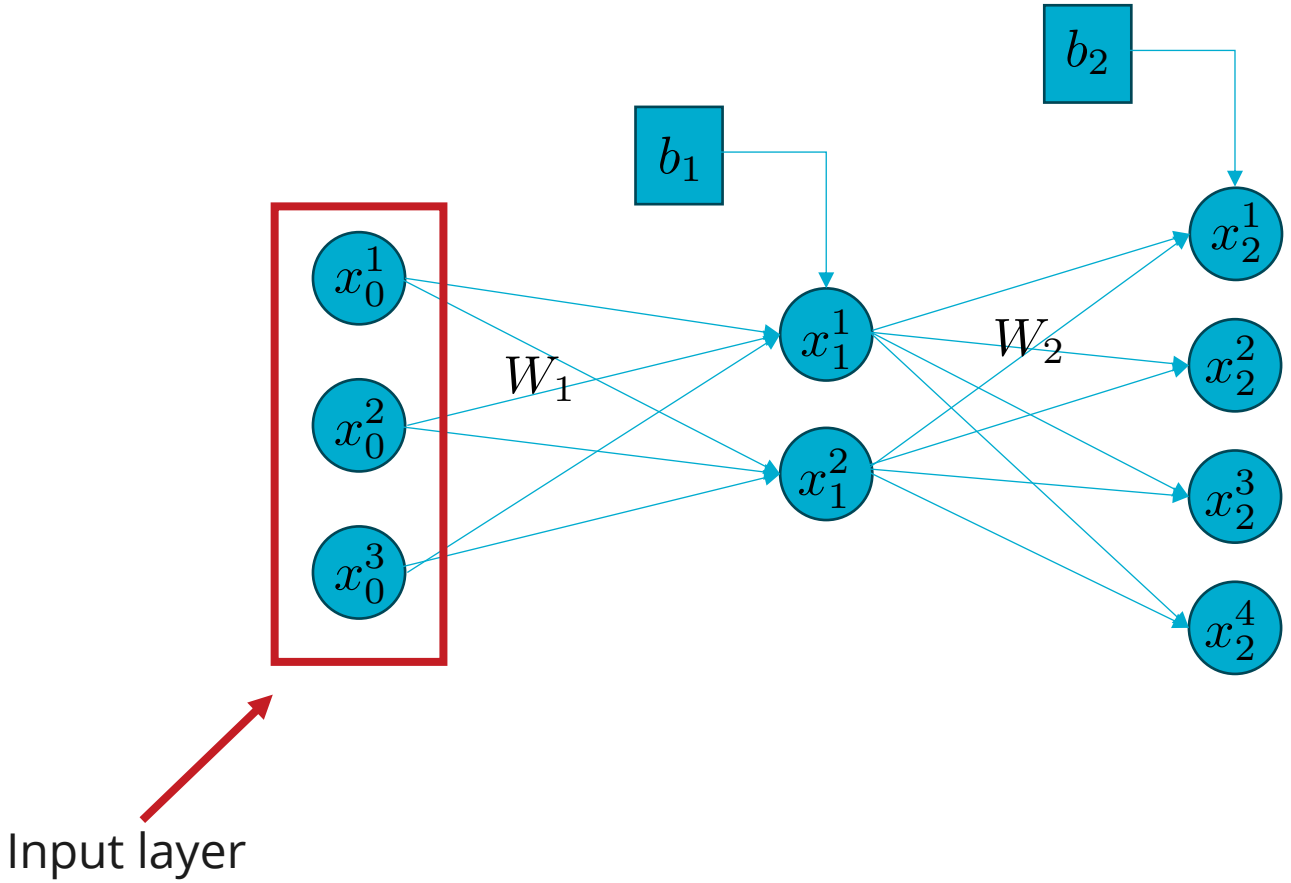
GRAPHICAL REPRESENTATIONS OF NEURAL NETWORKS

- Neural networks are often represented graphically,



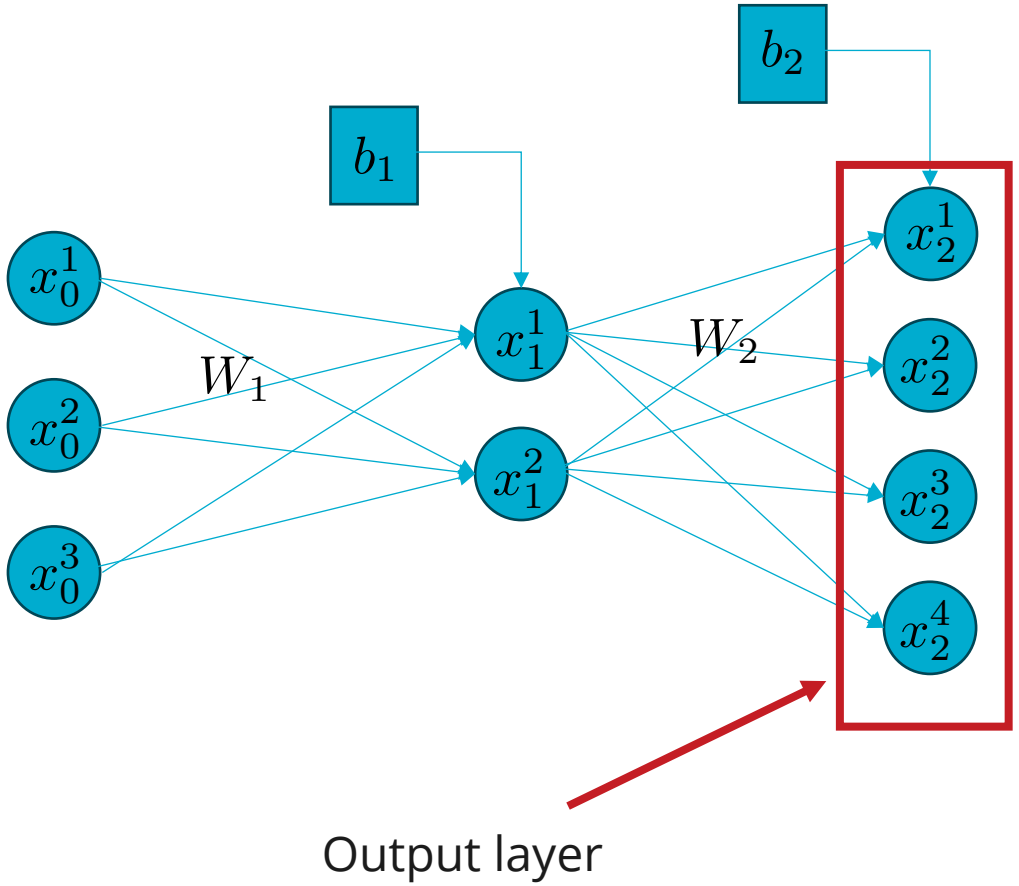
GRAPHICAL REPRESENTATIONS OF NEURAL NETWORKS

- Neural networks are often represented graphically,



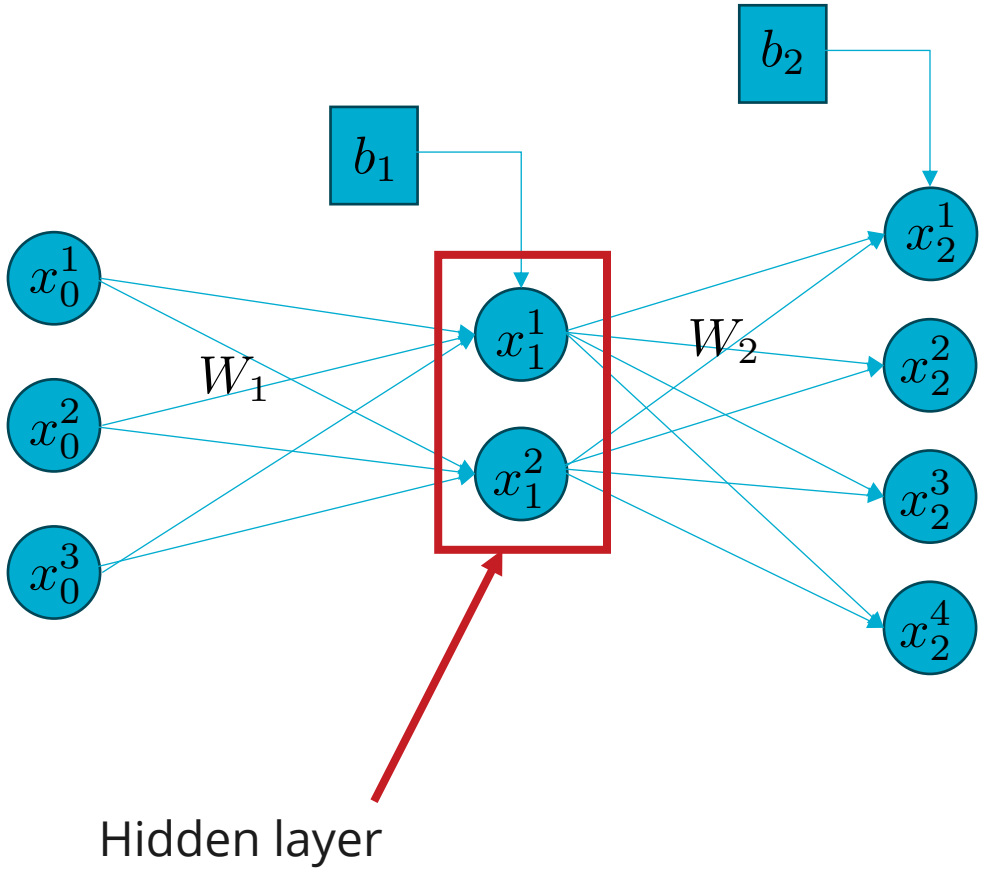
GRAPHICAL REPRESENTATIONS OF NEURAL NETWORKS

- Neural networks are often represented graphically,



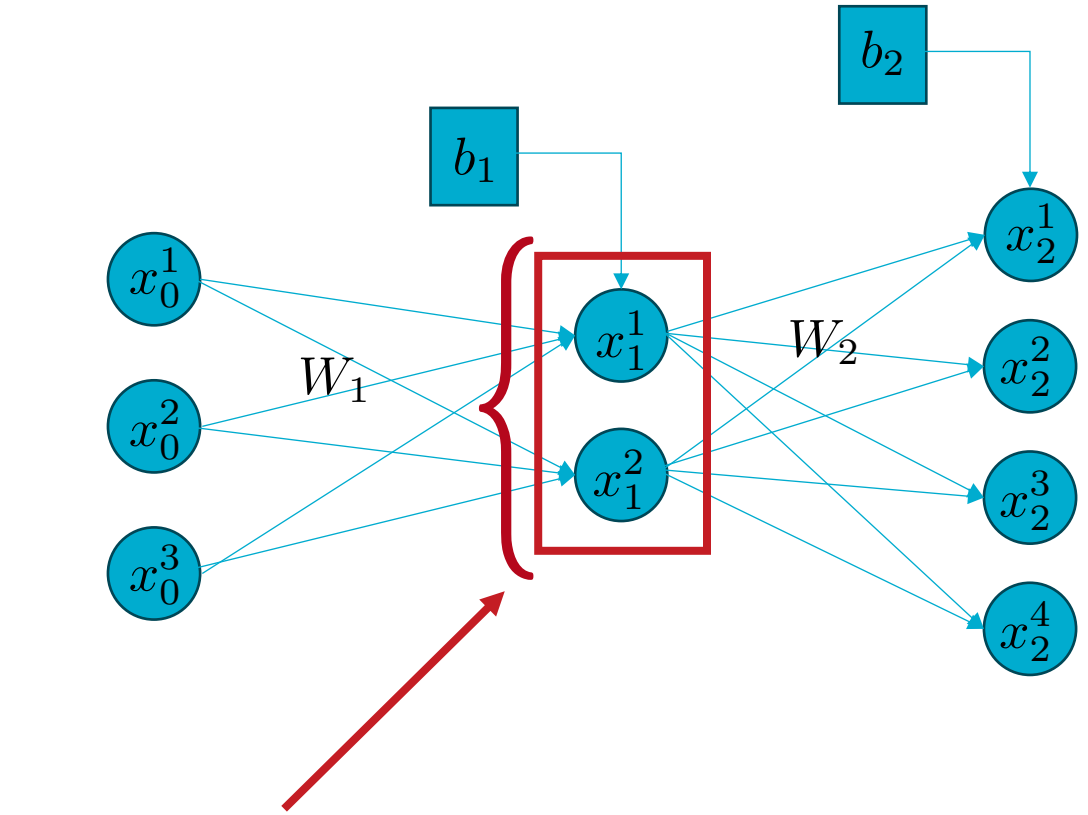
GRAPHICAL REPRESENTATIONS OF NEURAL NETWORKS

- Neural networks are often represented graphically,



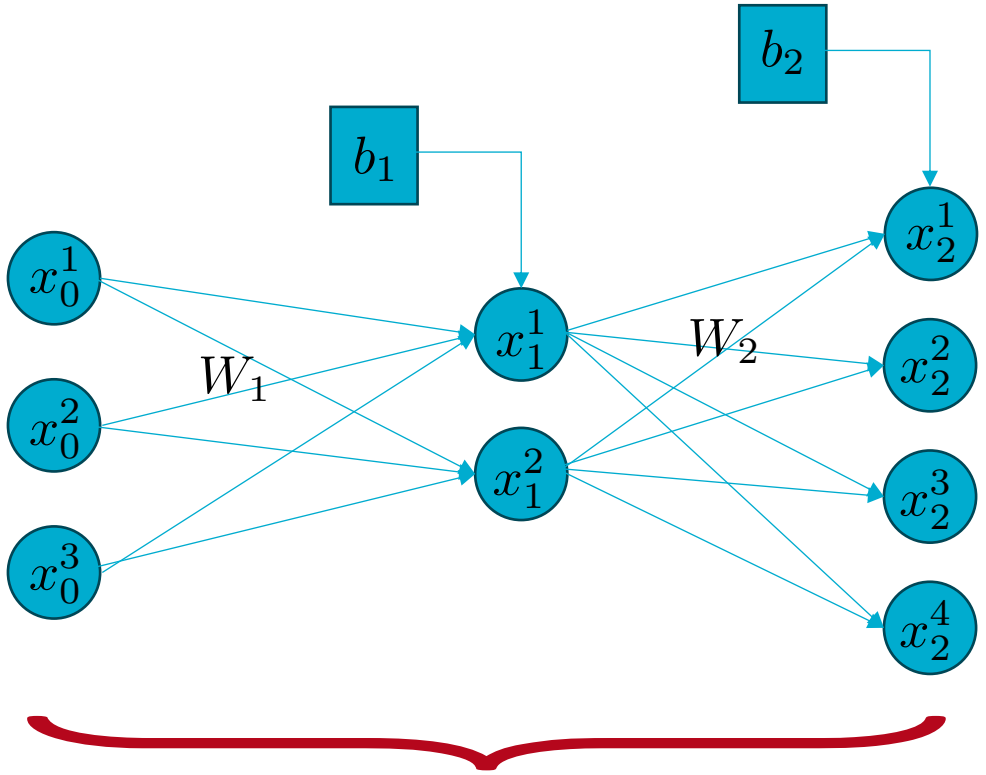
GRAPHICAL REPRESENTATIONS OF NEURAL NETWORKS

- Neural networks are often represented graphically,



GRAPHICAL REPRESENTATIONS OF NEURAL NETWORKS

- Neural networks are often represented graphically,



NEURAL NETWORK TRAINING

- Neural networks are universal approximators
  - For any function, $f : \mathbb{R}^n \supseteq K \to \mathbb{R}^m$, there's a neural network with error[1], $||f - N||_\infty < \epsilon$
  - Constructive proofs are not generally available

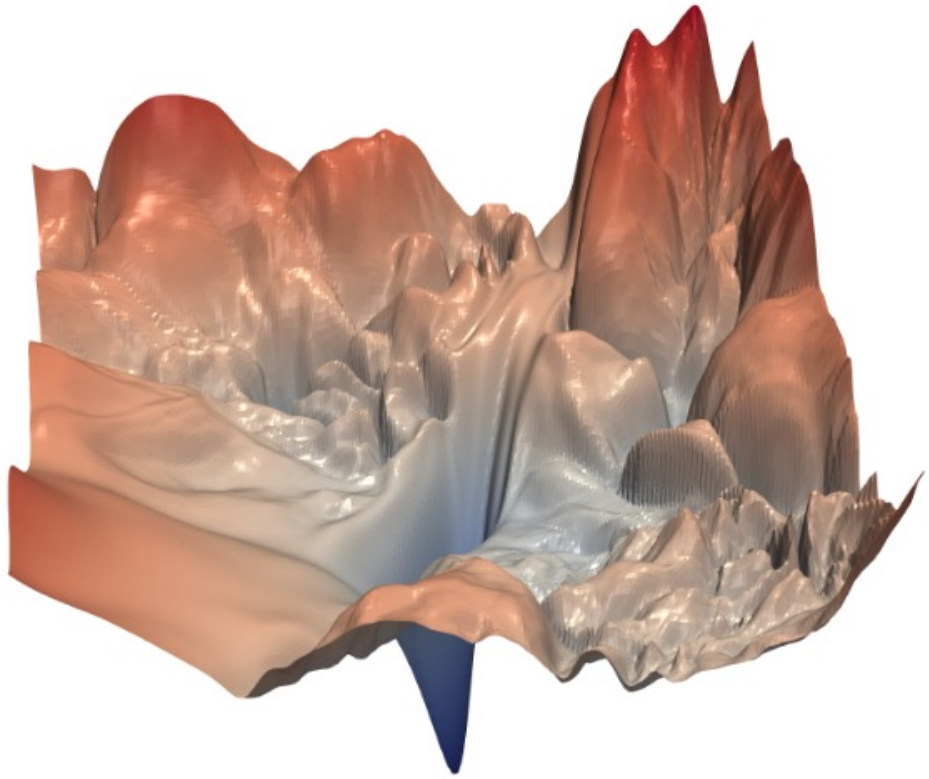
- We use optimization to train the neural network, i.e. regression
  - Say we have pairs of data $\{x_i, y_i\}^{i=1,\ldots,N}$
  - We find the weights and biases of the network,

$$\xi = \underset{\xi^*}{\operatorname{argmin}} \sum_i ||N(x_i, \xi^*) - y_i||_2^2$$

[1]G. Cybenko, *MCSS* (1989)

OPTIMIZATION FOR NEURAL NETWORKS

- The loss surface is not convex and high dimensional
  - Can’t find the global minimizer
- Empirical and theoretical results suggest that all local minima are roughly equivalent



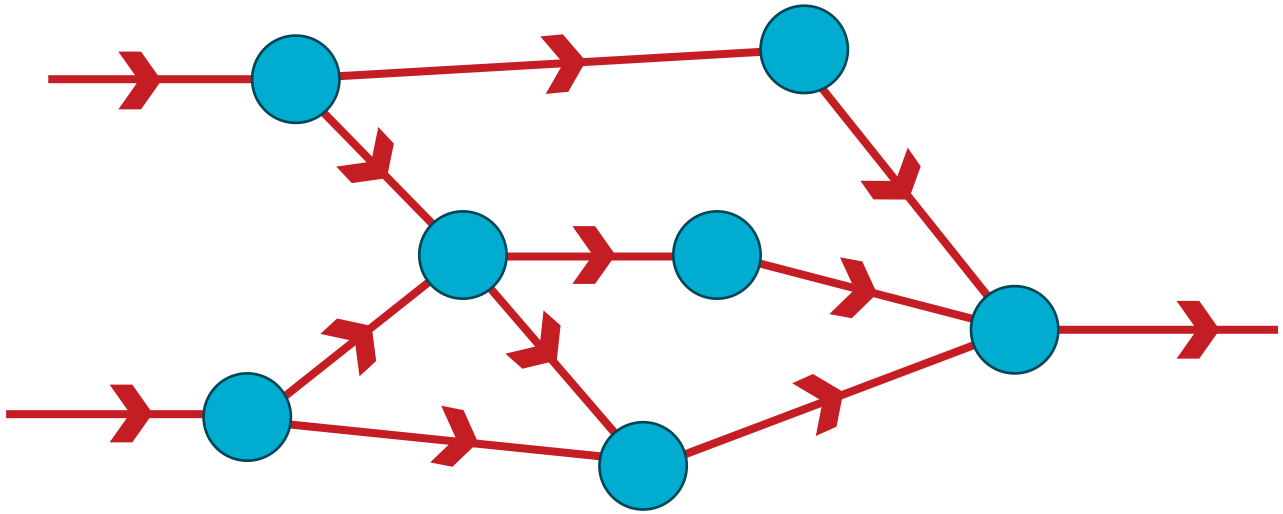
A 2d slice of the loss surface of ResNet-56
H. Li et al., *NeurIPS* (2018)

REVERSE MODE AUTOMATIC DIFFERENTIATION (BACKPROP)

- We can find optimal weights for a neural network using constrained optimization[1]

$$\xi = \underset{\xi^*}{\operatorname{argmin}} \sum_i ||N(x_i, \xi^*) - y_i||_2^2$$

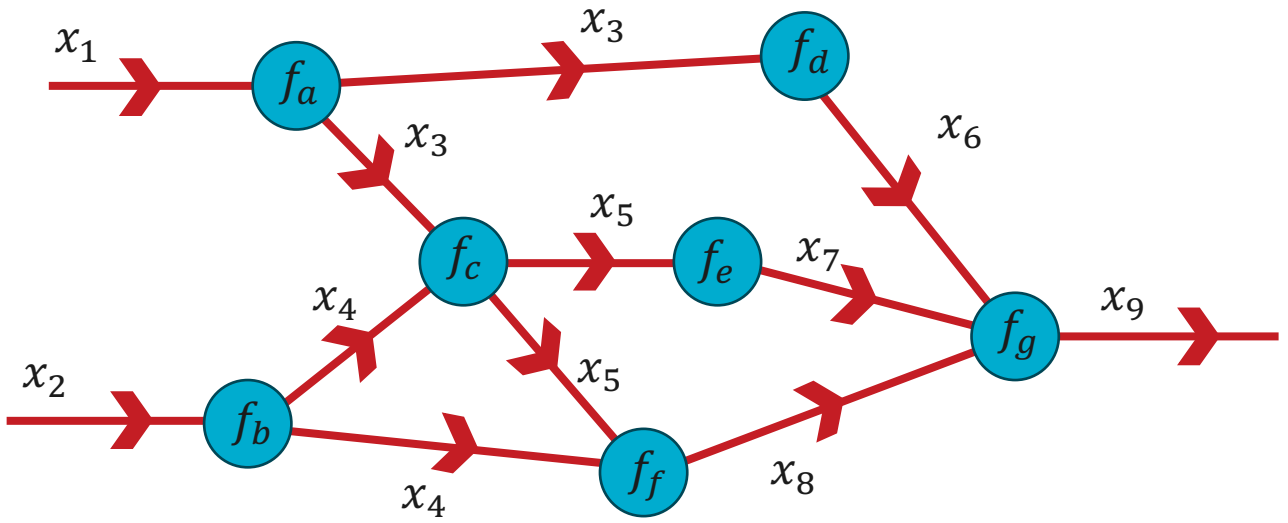
- Many objective functions can be represented by a directed acyclic graph (DAG)



- Functions at nodes, $f_a(x_i, \ldots, \xi_a)$, and data at edges, $x_1, \ldots$

[1]T. Vieira, https://timvieira.github.io/blog/post/2017/08/18/backprop-is-not-just-the-chain-rule/

REVERSE MODE AUTOMATIC DIFFERENTIATION (BACKPROP)

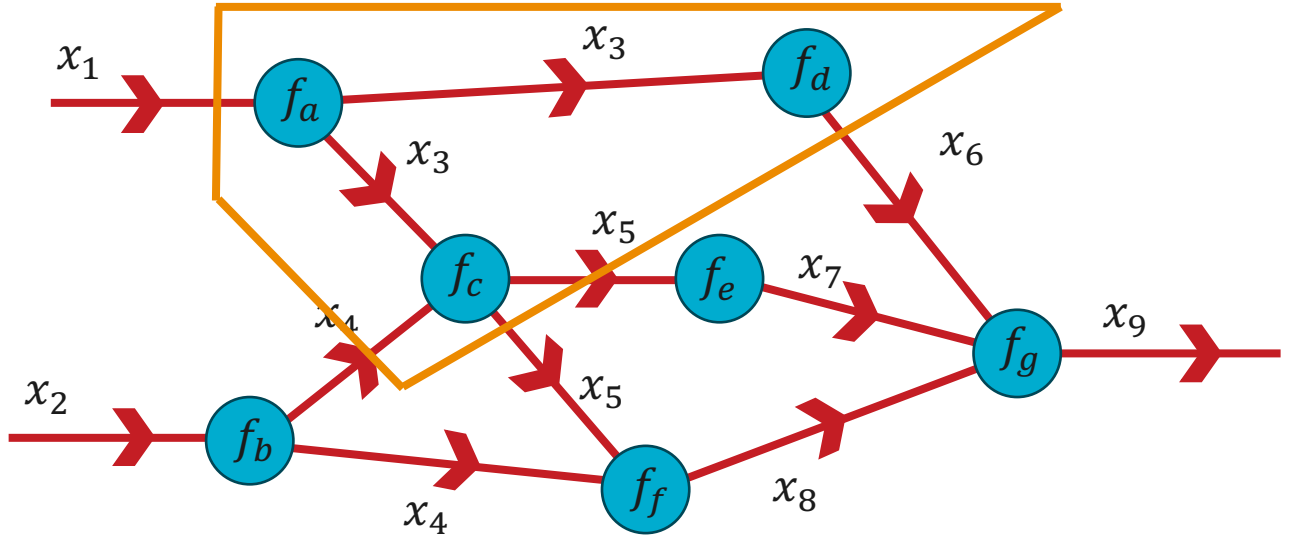


- The optimal weights are found at a critical point of the Lagrangian,

$$\mathcal{L} = \mathcal{J}(x_9) + \sum_{i=a}^{g} \lambda_i \left(f_i(\{x_j\}^{j \in I[a]}, \xi_i) - x_{k \in O[a]} \right)$$

$$\nabla_{\lambda_n} \mathcal{L} = 0, \quad \nabla_{x_m} \mathcal{L} = 0, \quad \nabla_{\xi_n} \mathcal{L} = 0$$

REVERSE MODE AUTOMATIC DIFFERENTIATION (BACKPROP)



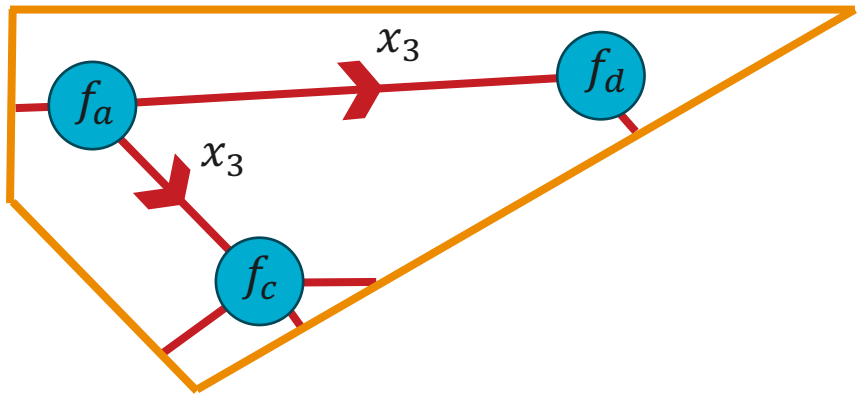
- The optimal weights are found at a critical point of the Lagrangian,

$$\mathcal{L} = \mathcal{J}(x_9) + \sum_{i=a}^{g} \lambda_i \left(f_i(\{x_j\}^{j \in I[a]}, \xi_i) - x_{k \in O[a]} \right)$$

$$\nabla_{\lambda_n} \mathcal{L} = 0, \quad \nabla_{x_m} \mathcal{L} = 0, \quad \nabla_{\xi_n} \mathcal{L} = 0$$

- To update ξ_a, let's look at the nodes connecting x_3

REVERSE MODE AUTOMATIC DIFFERENTIATION (BACKPROP)



- The critical points are found when,
  - Forward pass,

    $$\partial_{\lambda_a}\mathcal{L} = 0 \rightarrow x_3 = f_a(x_1)$$

  - Reverse pass,

    $$\partial_{x_3}\mathcal{L} = 0 \rightarrow \lambda_a = \lambda_c \partial_{x_3} f_c|_{x_3,\xi_c} + \lambda_d \partial_{x_3} f_d|_{x_3,\xi_d}$$

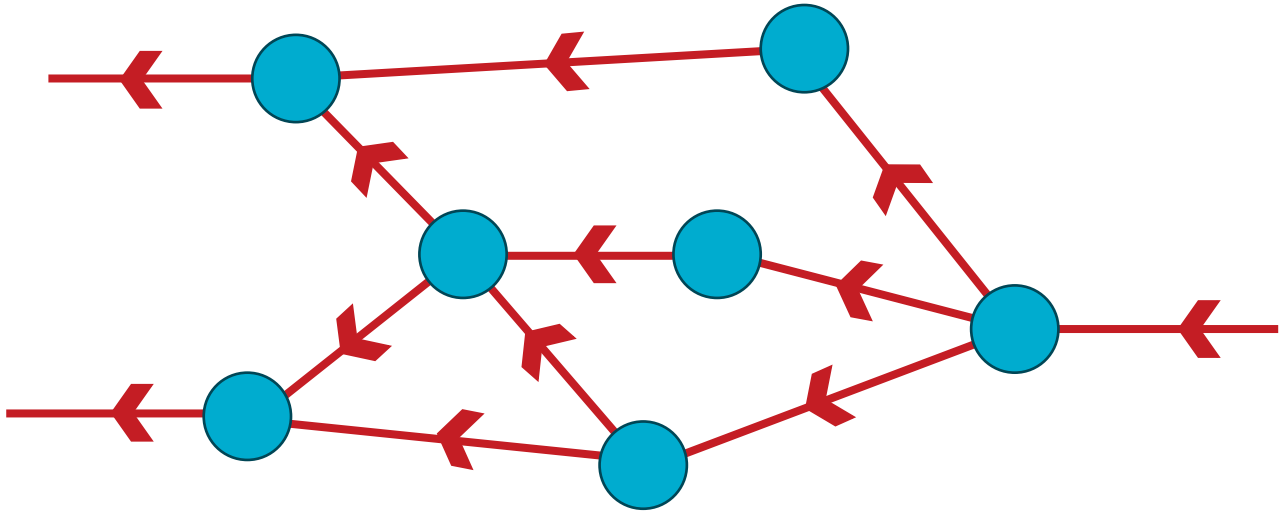
  - Update,

    $$\partial_{\xi_a}\mathcal{L} = 0 \rightarrow \lambda_a \partial_{\xi_a} f_a|_{x_3,\xi_a} = 0$$

    - Can't find exactly, so we update iteratively. E.g., using gradient descent,
      $\xi_a^* = \xi_a - \alpha\lambda_a \partial_{\xi_a} f_a|_{x_3,\xi_a}$
    - λ_a is a function of the λ_n after it, which are all functions of the λ_n before them, and so on.
    - If you write out $\lambda_a \partial_{\xi_a} f_a|_{x_3,\xi_a}$ explicitly, it's the chain rule for $\partial_{\xi_a}\mathcal{J}$, working backwards from $\mathcal{J}(x_9)$
    - Let's write this as, $g_a = \lambda_a \partial_{\xi_a} f_a|_{x_3,\xi_a}$

REVERSE MODE AUTOMATIC DIFFERENTIATION (BACKPROP)

- Constrained optimization reverses the flow of computation,



- The reverse pass has the same computational complexity as the forward pass
- Easy to implement and optimize -> TensorFlow

BACKPROP GENERALIZATIONS

- The data can live in more general spaces as long as there's a differentiable inner product,

$$x_i \in \mathcal{X}_i, \quad (\cdot,\cdot)_{\mathcal{X}_i} : \mathcal{X}_i \times \mathcal{X}_i \to \mathbb{R}$$

  - Replace product between the Lagrange multiplier and function output with the inner product,

$$\left(\lambda_i, \left(f_i(\{x_j\}^{j\in I[a]}, \xi_i) - x_{k\in O[a]}\right)\right)_{\mathcal{X}_i}$$

- E.g., the data can be finite dimensional vectors,

$$\mathcal{X} = \mathbb{R}^n, \quad (\lambda, x) = \sum_i \lambda_i x_i$$

- E.g., the data comes from an ODE (see the Neural ODE[1] and adjoint sensitivity[2]),

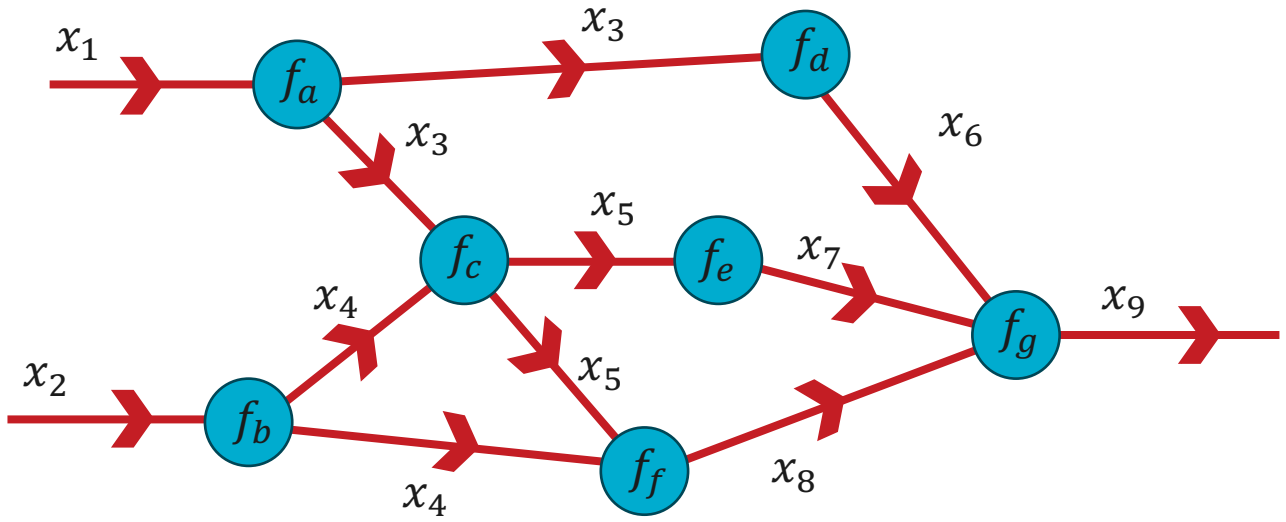
$$\mathcal{X} = L_2[0,T], \quad \int_0^T \sum_i \lambda_i \left(\frac{du}{dt} - f(u,\xi)\right)_i dt$$

[1]R. Chen et al., *NeurIPS* (2018)

[2]A. Bradley, PDE-constrained optimization and the adjoint method (2019)

ALTERNATIVE TO BACKPROP - FORWARD MODE AUTOMATIC DIFFERENTIATION

- Instead of backprop, forward mode automatic differentiation follows the same data flow as computing the objective function



- But we also compute the Jacobian along the way using the chain rule, e.g.,

$$\partial_{\xi_a} f_d = \partial_{x_3} f_d \partial_{\xi_a} f_a$$

- Tends to be more expensive when the output is lower dimensional than the input

UPDATING THE WEIGHTS AND BIASES

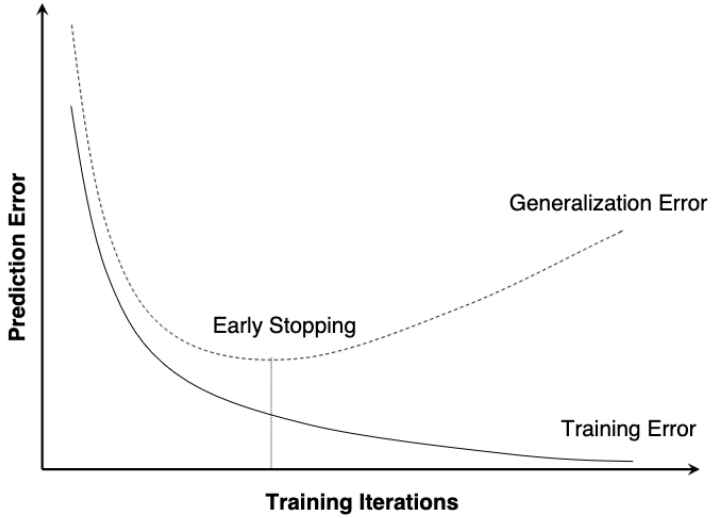
- Typically we use first order optimizers to update the weights and biases, e.g. gradient descent,
  $\xi_a^* = \xi_a - \alpha g_a$
- More sophisticated optimizers will include
  - Momentum

    $$\begin{bmatrix} \xi_a^* \\ v_a^* \end{bmatrix} = \begin{bmatrix} \xi_a \\ v_a \end{bmatrix} - \alpha f(g_a, v_a)$$

  - Decay
- ADAM is probably the most popular, (Kingma and Ba, arXiv:1412.6980, (2014))

BEST PRACTICES FOR TRAINING

- Usually at every iteration, estimates of the gradient, g_a, are computed from a subset of the data, i.e., *minibatching*
  - Faster updates – don't have to process all the data at once
  - Extra noise helps with saddle points
  - An epoch is a set of updates using all of the minibatches
- Separate data into three sets,
  - 60% training data – use this to compute the gradients for your optimizer
  - 20% validation data – use this to evaluate your objective function while training, validation loss
  - 20% test data – report this as your model error



Learning curve for a neural network
C. Perlich, IBM Research Report (2009)

REGULARIZATION

- There's several techniques to prevent overfitting
  - Weight decay, ℓ_2 regularization of the neural network parameters,

$$\xi = \underset{\xi^*}{\operatorname{argmin}} \sum_i ||N(x_i, \xi^*) - y_i||_2^2 + \lambda ||\xi^*||_2^2$$

  - Lasso regularization, ℓ_1 regularization of the neural network parameters,

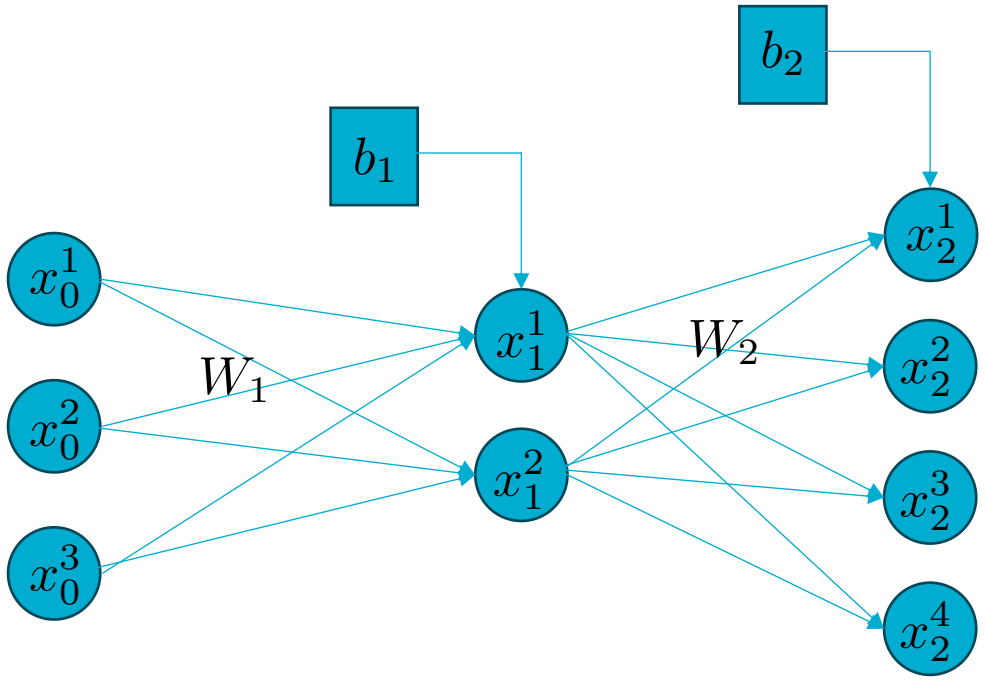
$$\xi = \underset{\xi^*}{\operatorname{argmin}} \sum_i ||N(x_i, \xi^*) - y_i||_2^2 + \lambda ||\xi^*||_1$$

  - Dropout, stochastically set a percentage of the weights to zero during training

NEURAL NETWORK VARIANTS

Dense Network

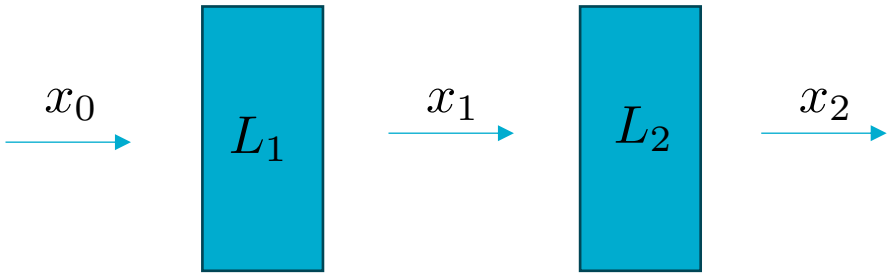
- We've been looking at a dense neural network,



NEURAL NETWORK VARIANTS

Dense Network

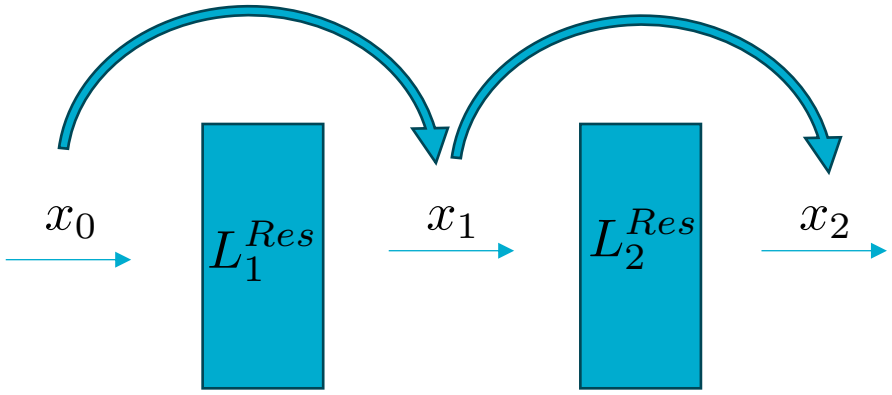
- Let’s use a simplified representation,



NEURAL NETWORK VARIANTS

Residual Neural Network (ResNet)

$$x_1 = L_1^{Res} x_0 = W_1(x_0) + b_1 + x_0$$



- For deep ResNets there is a connection to ODE's via the Forward Euler discretization,

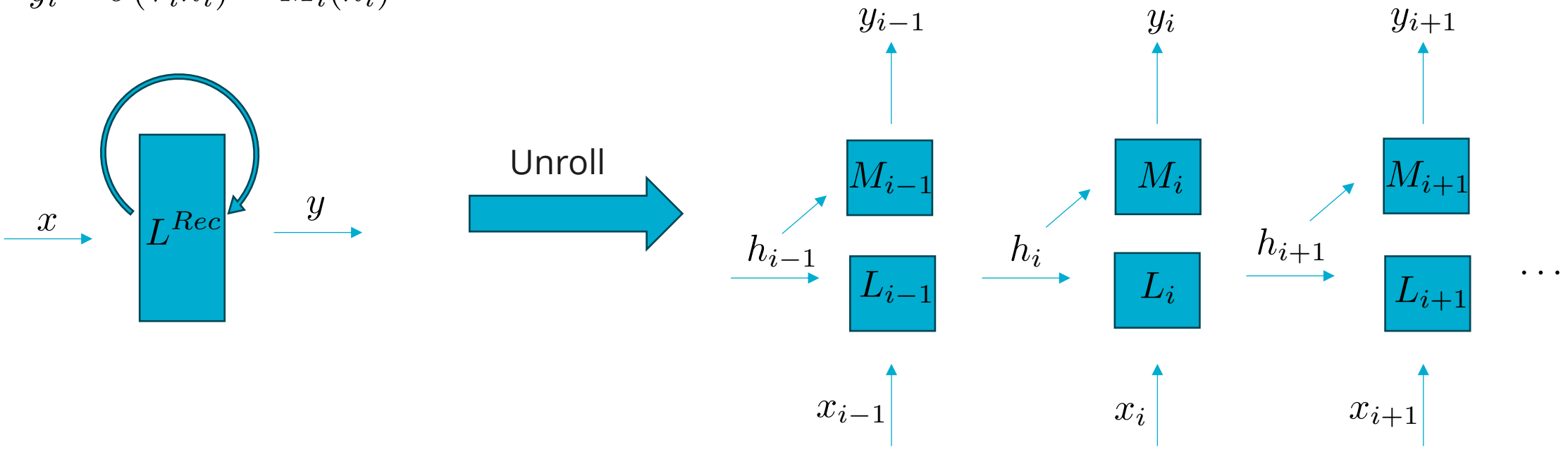
$$u^{n+1} = u^n + \Delta t L(t^n, u^n)$$

NEURAL NETWORK VARIANTS

Recurrent Neural Network (RNN)

- Reuse the same weights between layers

$$h_i = L_i(x_{i-1}, h_{i-1}) = \sigma(W_i x_{i-1} + U_i h_{i-1} + b_i)$$
$$y_i = \sigma(V_i h_i) = M_i(h_i)$$



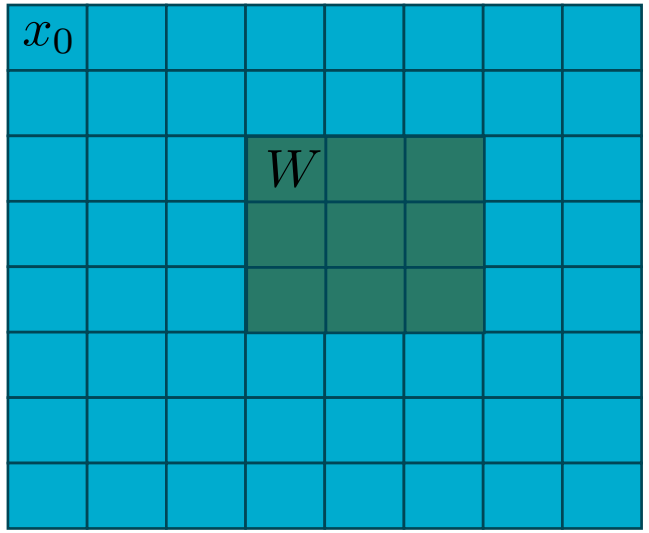
- Useful for time series

NEURAL NETWORK VARIANTS

Convolutional Neural Network (ConvNet)

- For spatial data on a grid, ConvNets use a banded structure for the weights. For 2D,

$$x_1^{ij} = \sigma\left(\sum_{p=-k}^{k}\sum_{q=-l}^{l} W_1^{pq} x_0^{i+p,j+q} + b_1^{ij}\right)$$



- Typically, we'll also map between multiple channels, e.g., RGB in an image

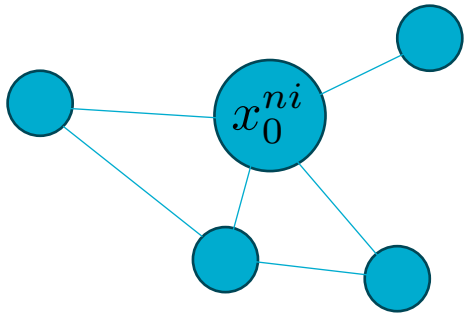
$$x_1^{mij} = \sigma\left(\sum_{n}\sum_{p=-k}^{k}\sum_{q=-l}^{l} W_1^{mnpq} x_0^{n,i+p,j+q} + b_1^{mij}\right)$$

NEURAL NETWORK VARIANTS

Graph Neural Network (GNN)

- Convolutions can be generalized to work on graphs,

$$x_1^{mi} = \sigma\left(\sum_n \sum_{p \in N(i)} W_1^{mnp} x_0^{np} + b_1^{mi}\right)$$



NEURAL NETWORKS IN TENSORFLOW

- There are two ways to build and train neural networks in TensorFlow
  - Manually
  - Keras Library
    - Neural networks “layers”
    - Models composed of layers
    - Optimizers
    - Regularizers
    - Metrics
    - Callbacks
- Let’s look at some code examples

EXERCISE

- Use a neural network to fit the 2D function in the unit box,

  $f(x) = x^T x$

- Try both the low level interface and the Keras interface
- Experiment with different architectures and optimizers