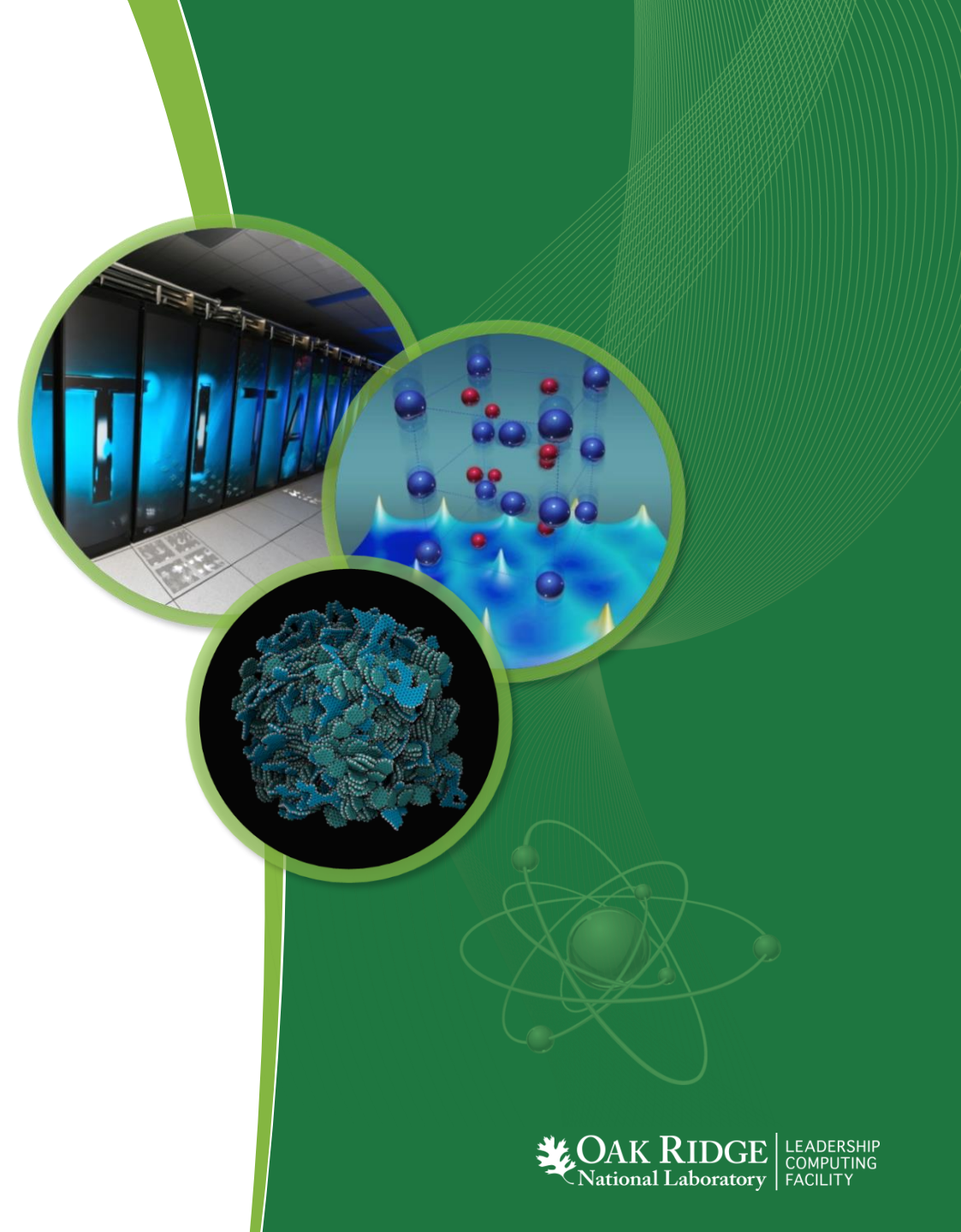


Introduction to Gradient Descent- Based Machine Learning

Matt Norman

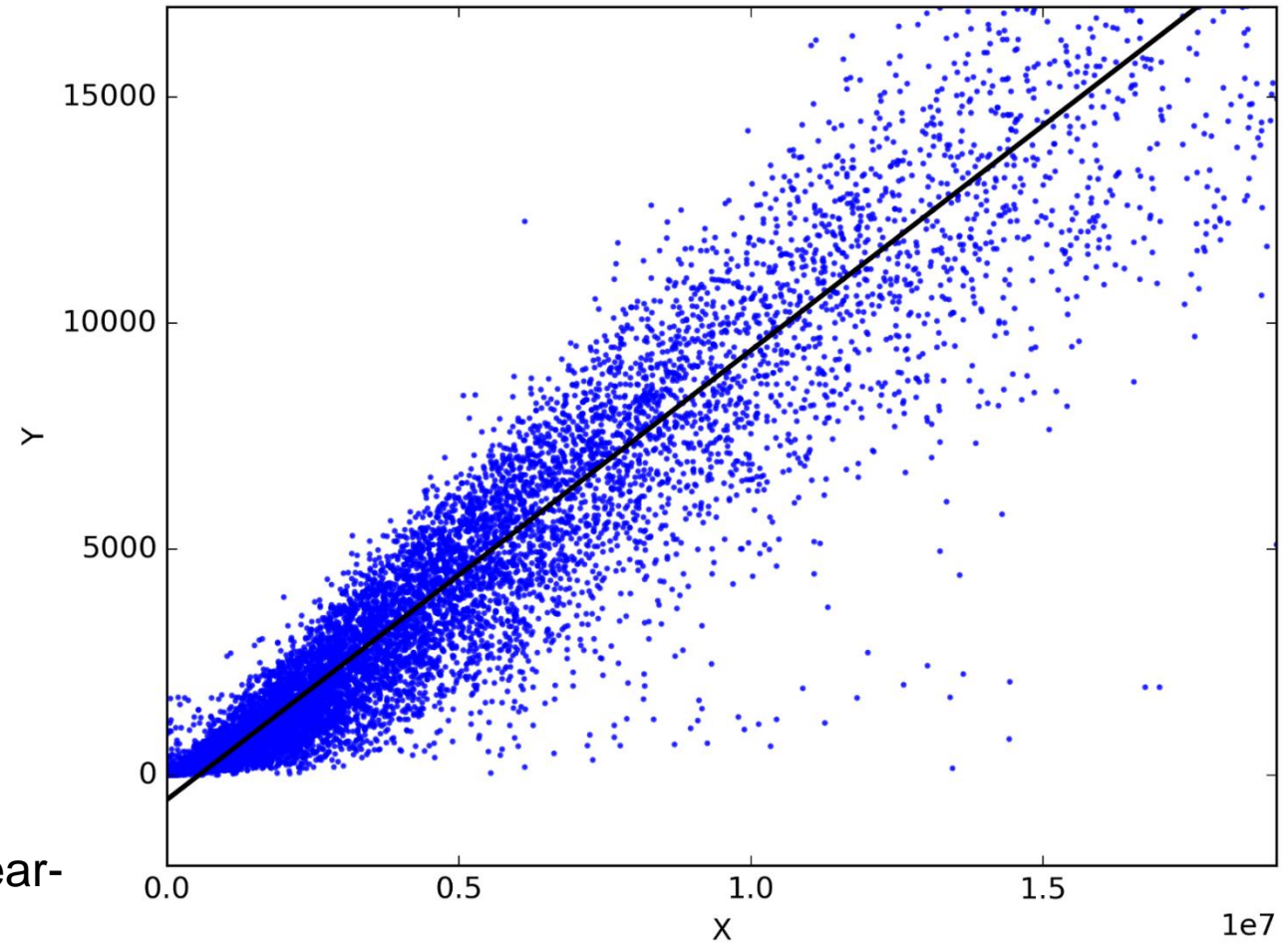
Scientific Computing Group

Oak Ridge National Laboratory



What Is Machine Learning?

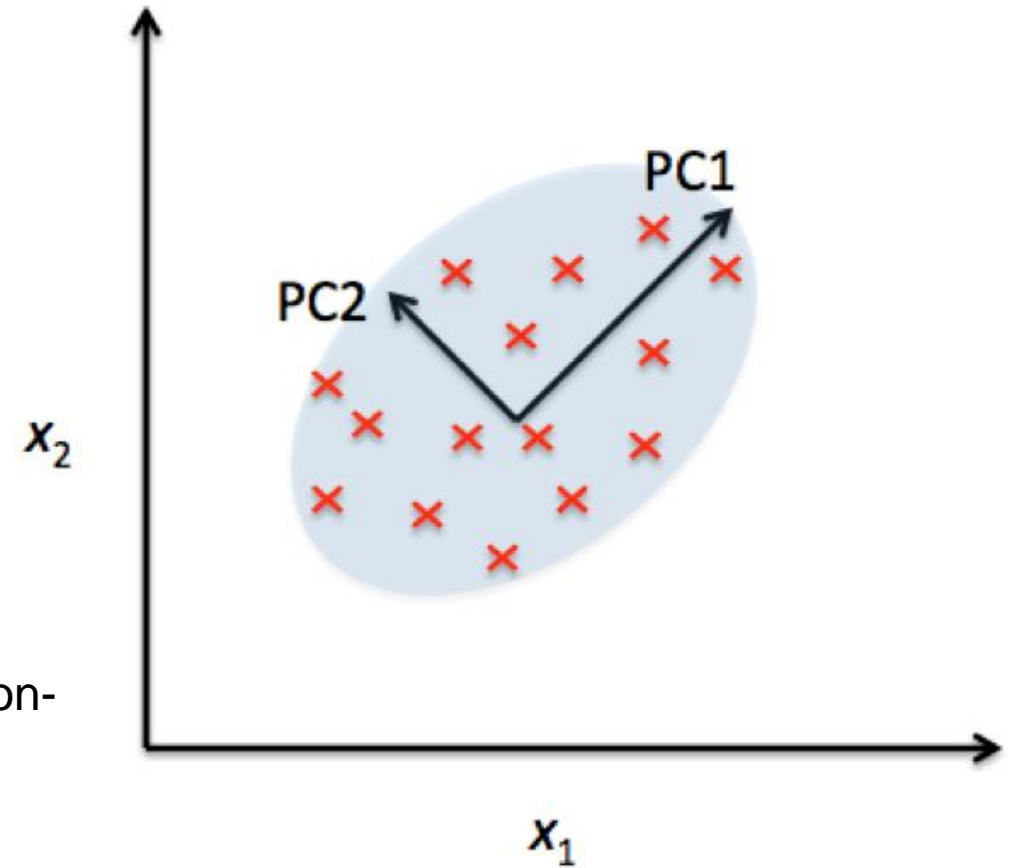
- Representing something on a (typically reduced) basis set
- Linear regression
- Lots of data $\rightarrow ax + b$



<https://medium.com/@amarbudhiraja/ml-101-linear-regression-tutorial-1e40e29f1934>

What Is Machine Learning?

- Representing something on a (typically reduced) basis set
- A set of principal components



<https://www.quora.com/Can-you-explain-the-comparison-between-principal-component-analysis-and-linear-discriminant-analysis-in-dimensionality-reduction-with-MATLAB-code-Which-one-is-more-efficient>

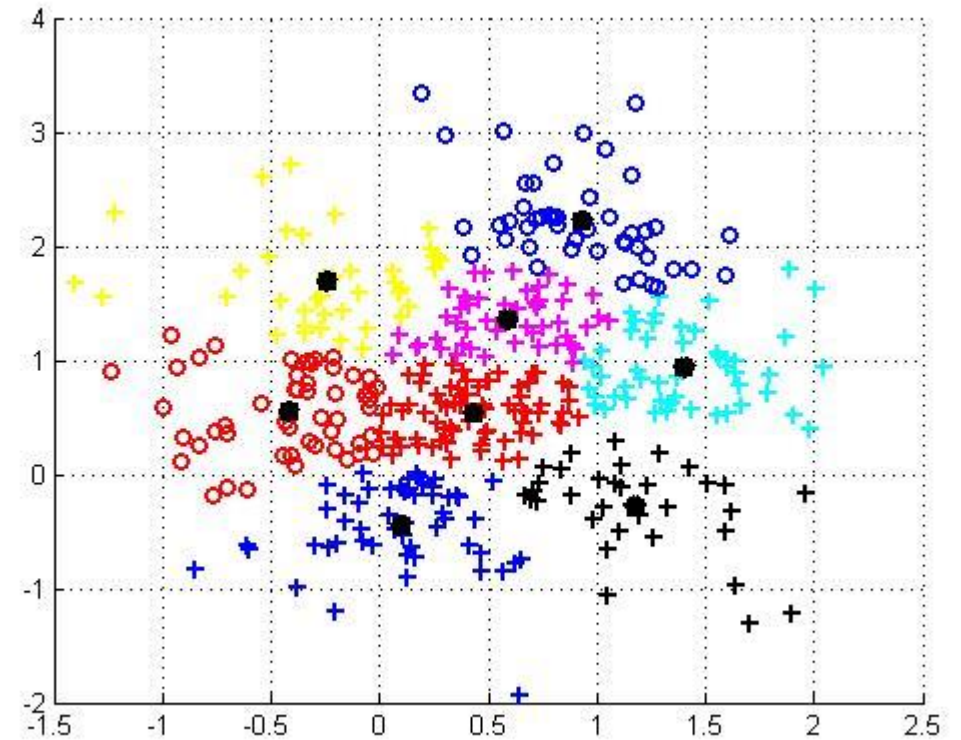
What Is Machine Learning?

- Representing something on a (typically reduced) basis set
- Deep Neural Networks
- Input \rightarrow Output:
 $\tanh(A_2 \tanh(A_1 x + b_1) + b_2)$

What Is Machine Learning?

- Representing something on a (typically reduced) basis set
- Clustering
- Lots of data → A set of categories

<https://fr.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/52579/versions/9/screenshot.jpg>

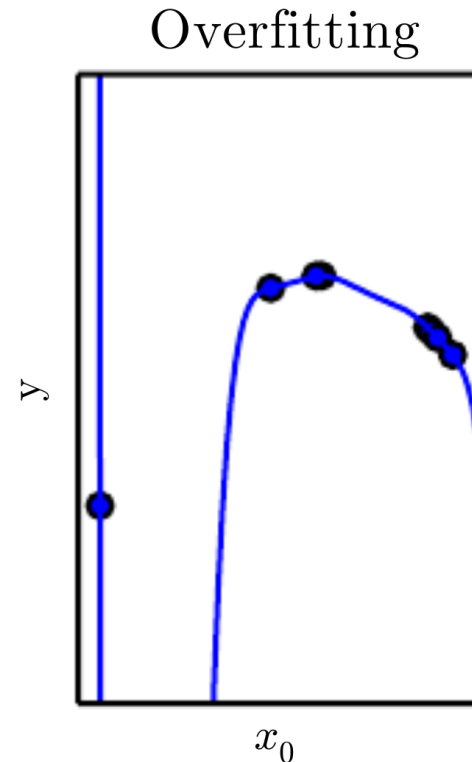
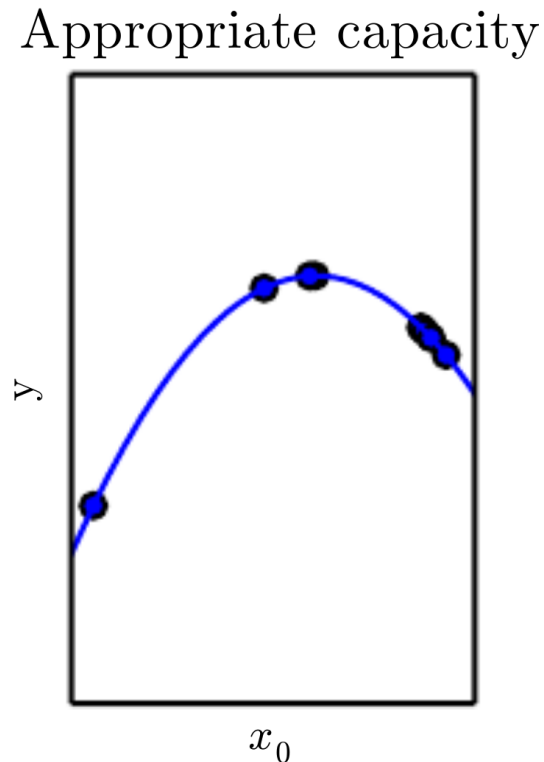
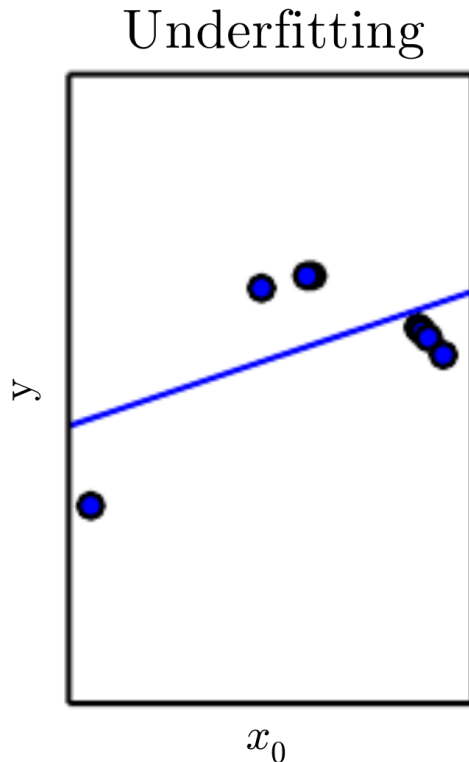


What Is “training”?

- Determining the parameters of a reduced basis set / model that “best” represents something of interest
 - It’s an inverse problem
- Could be cast as mathematical projection (e.g., least squares)
- Could be cast as an optimization problem (e.g., gradient descent)
 - Most modern machine learning casts training as an optimization problem

Capacity, overfitting, and generalization

- “Capacity” is in a sense, “how much” a model can learn.
 - E.g., an order 1 polynomial can’t learn an exponential curve well
- Generalization is the ability of a model to give reasonable answers to inputs it hasn’t been trained on



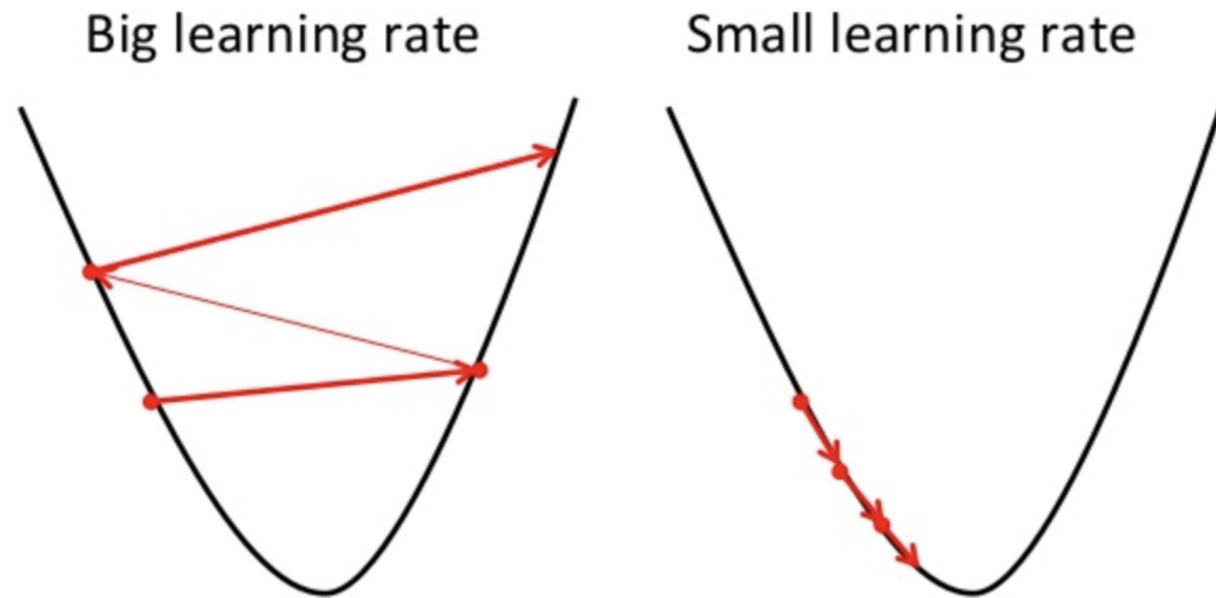
Deep Learning
by Ian
Goodfellow

“Loss function”, Gradient Descent, and Learning Rate

- Modern ML usually trains by minimizing a “loss” function. I think most other fields call this the “cost” function
- Goal is to find the model’s parameters that give the minimum “loss”
- Gradient Descent is the workhorse of most modern machine learning
 - Follow the function in the direction of the negative gradient, i.e., follow the function “downwards” (toward the minimum)
 - $x_{n+1} = x_n - \gamma \nabla_x L(x)$
 - x is the vector of model parameters
 - γ is the “learning rate”, the size of the step you take in the downward direction
 - $L(x)$ is the loss function (usually a scalar value)
 - $\nabla_x L(x)$ is the gradient of the loss function with respect to the model parameters
 - The heart of Gradient Descent is in calculating the gradient itself
- “If it’s down, go that way. How far? I don’t know, not too far...”

“Loss function”, Gradient Descent, and learning rate

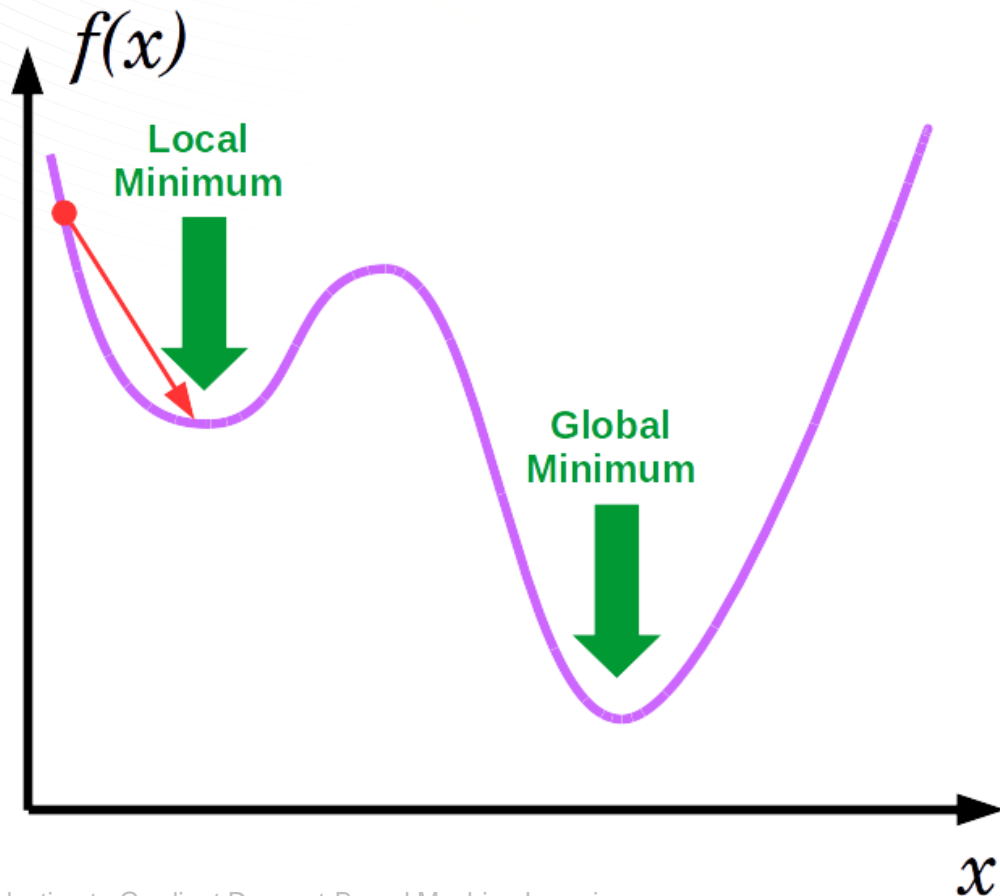
- What should my learning rate be? That’s a really good question...
- Most packages perform gradient descent with dynamic learning rates that are decreased when your loss doesn’t decrease and / or are increased when your loss is decreasing too slowly
 - Check out: <https://wiseodd.github.io/techblog/2016/06/22/nn-optimization/>



<https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0>

What About Non-Convex Functions?

- Everyone knows gradient descent doesn't work for non-convex functions...right?...right?
 - There's a risk that you'll get "stuck" in a local min and miss the global min



<https://www.neural-networks.io/en/single-layer/gradient-descent.php>

“Stochastic” Gradient Descent (SGD) and minibatches

- No gradient descent algorithm guarantees a global minimum, but in practice, if you have “enough data,” you usually find a “good” minimum with SGD
 - Why? I really have no idea
- Most ML involves a lot of samples to learn from
- Naïve gradient descent creates a “loss” function that sums over all samples
 - However, this requires passing through all of the data before taking a single step
- Stochastic Gradient Descent (SGD), instead, takes steps based on small subsets of the data, called “minibatches”, at a time
 - Minibatch sizes generally range from one to hundreds, sometimes thousands
 - By stepping toward the minimum as you go through the data rather than after, you get to the minimum much faster in practice
 - It’s “stochastic” because you shuffle the data each time you make a pass through it

Minibatches, epochs, training error

- Most ML packages inform you how you're doing as you train with SGD
- Each full pass through the data is called an “epoch”
 - You usually train over thousands of epochs, but it depends on your data
 - Each epoch is processed in a series of “minibatches”
- Each minibatch
 - The loss function is accumulated over the samples in the minibatch and added to the total epoch loss
 - The gradient of the accumulated loss with respect to model parameters is computed
 - The model parameters are updated by taking a step in the minibatch's negative gradient direction
- At the end of the epoch, you have the total summed “loss”, which is your “training error”. It's the error for the data you explicitly trained on

Test Error and Monitoring for Overfitting

- It's wise to split off a small proportion of data each epoch that you don't train on
 - Your model's "loss" over this set of data is called "test" error, though some packages (e.g., keras) label this as "validation" error
- Your "test" error tells you how well your model is generalizing to data it hasn't seen before after a single epoch
- **IMPORTANT: If training error goes down while test error goes up, you are overfitting your model, and it will not generalize well**
 - Many ML packages allow you to automatically stop training once this starts happening, and this is one means of preventing overfitting

Training Error, Test Error, and Validation Error

- Many people take another step and save data that the model is never trained on in any epoch and use that data for “validation”
- Training Error
 - The sum of your loss function over a minibatch and epoch. This is the error for data you explicitly just trained on
- Test Error
 - The sum of your loss function over a set of data that wasn't trained on for a given epoch. This is your first indication of how well your model generalizes
- Validation Error
 - The sum of your loss function over a set of data that was never trained on in any epoch. This is your final indication of how well your model generalizes

“Back Propagation”

- This is just one particular means of computing the gradient of the loss function with respect to the parameters of the learning model
- It's also known more broadly as reverse-mode “automatic differentiation” or “algorithmic differentiation”
 - This is an efficient reverse-engineering approach using the derivative chain rule
 - Most packages create a computational graph of operations, which each have an associated derivative
 - There are C++ (and even Fortran...) codes that overload operators to create an Directed Acyclic Graph (DAG) and save intermediate values
 - Then, you can compute the gradient efficiently when you want
 - For most ML models, though, full AD is overkill; you can code it more efficiently yourself
- You could replace this with something more familiar such as finite-differences
 - However, FD requires $N + 1$ model evaluations per gradient (N is # model params)
 - Back Propagation only requires 1 model evaluation per gradient
 - However, back-prop must traverse a Directed Acyclic Graph (in the general case)

Artificial Neural Networks and Activation Functions

- An Artificial Neural Network (ANN) is meant to mimic biological neural networks with dense linear algebra wrapped in non-linear functions
 - Matrices represent synapse connections between neurons
 - Spoiler alert: Our brains probably don't use linear algebra...
 - This is a common and popular model (basis set), but it's not the only one
 - Obviously it runs well on GPUs
- Example: Deep, “feed-forward”, “fully-connected” ANN (*terms defined later*)
 - $h_1 = \tanh(A_1 x_{In} + b_1)$; $h_2 = \tanh(A_2 h_1 + b_2)$; $x_{out} = \tanh(A_3 h_2 + b_3)$
 - A_1 , A_2 , and A_3 , are matrices; b_1 , b_2 , b_3 are “bias” vectors
 - h_1 and h_2 are called “hidden layers” because they're intermediary
 - The individual values of the hidden layers are called “neurons”
 - The action of a linear operator plus a non-linear “activation” (e.g., \tanh) is called a “layer”
 - “Deep” Neural Networks are ANNs with multiple layers (i.e., at least one hidden layer)
 - \tanh is an example of an “activation function”

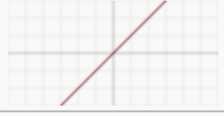


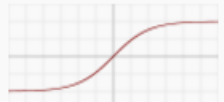
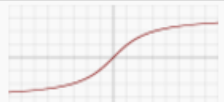


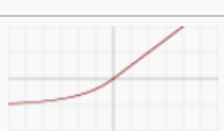
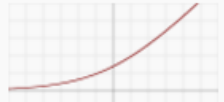
Activation Functions

- An activation function is a non-linear wrapper around a linear operator
- Without activations, the ANN couldn't emulate non-linear behavior
- Also, without them, deep neural nets would be meaningless because successive linear operators can be merged into a single operator

Activation Function Examples

- There are many to choose from, and you can make your own if you want
 - E.g., I have my own “leaky tanh” and “smooth leaky ReLU” functions made of piecewise low-order polynomials
- Finding the right one is often best done via trial and error

<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

“Dead” and “Saturated” Neurons

- If a neuron has a zero value, it's dead
- How does a neuron come to die? Often via certain activation functions
- ReLU (Rectified Linear Unit): $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
- A neuron to get stuck at zero via zero gradient
- Thus, “Leaky” ReLU is popular: $f(x) = \begin{cases} cx & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ where c is small
- Neurons can also be “saturated” (or frozen) in that they never change
- A neuron can get stuck in the “flat” region of tanh & have zero gradient
 - Remember the gradient is a chain rule, and any zero zeroes out the whole chain
 - Only the gradient of the loss function with respect to a given neuron can change that neuron's value in SGD

Activations on the last layer

- For regression
 - The role of the activation function for regression, in my opinion, is mainly:
 - To interrupt linear operators with non-linearity
 - To keep neurons in a well-behaved range
 - Thus, I don't think you need an activation on the last layer
 - Further, activations often “squash” the output, making the model harder to train
- For classification
 - Many people use a “softmax” activation on the last layer, and it's more useful for classification

Data Normalization and Optimization Constraints

- You **must** normalize your data before training
- Why normalize the input?
 - Because you don't want to end up with dead or frozen neurons
 - Your data should be in the range of your activation function's active region
- Why normalize the output?
 - Because the gradient descent algorithm responds only to the loss function accumulated over all data samples, which will respect larger values more
 - Normalizing your output is how you tell the optimizer what you care about most
- You can also add constraints to your optimization to keep neurons well-behaved
 - This is important especially for deep networks where the inner neurons vary more.
- How you normalize your data is of order one importance in how well your model is trained

Input, Output, and Samples

- “Input” is a vector you feed into your ML model
- “Output” is a scalar or vector you get out of your ML model
- A “sample” is a single input-output pair you use to train your ML model

What Loss Function Do I Use?

- Your loss function will depend on your application
- Remember that the gradient is everything
 - A non-smooth loss function is harder to converge toward a good minimum
- L2-norm type loss functions seem to behave well for regression
 - L1 and L-inf norms don't have globally defined gradients
- Classification problems usually use something called “cross-entropy”

How Do I Initialize the ANN Parameters?

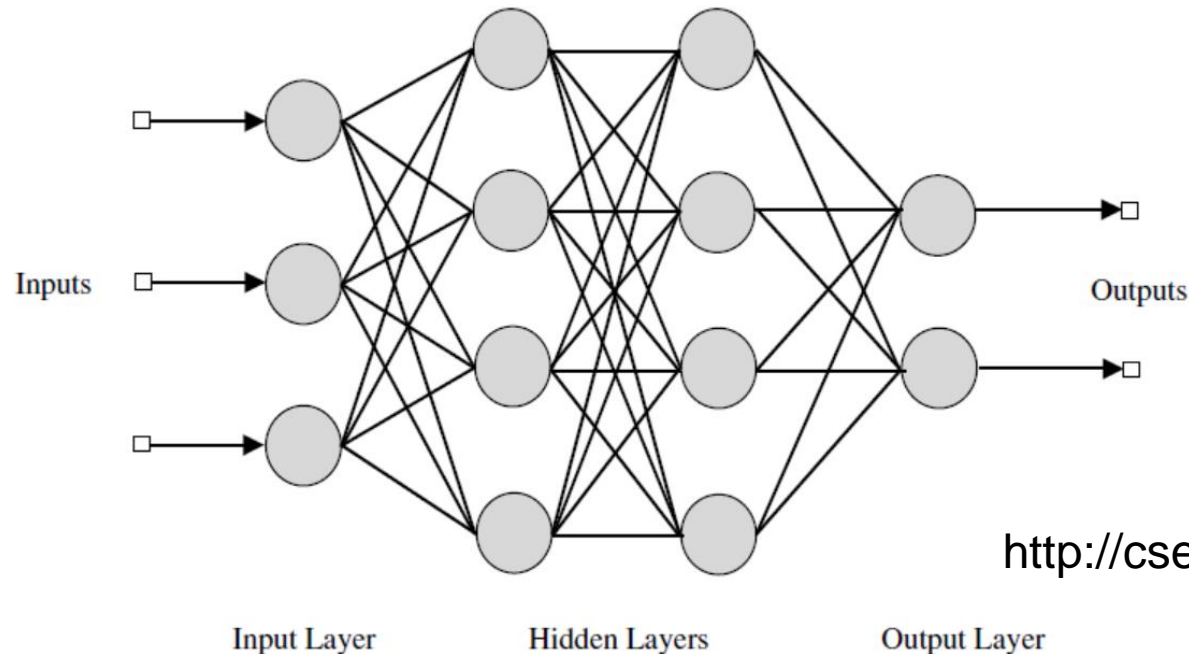
- How you initialize determines where you end up
- Ideally, with enough data, it won't matter that much how you initialize
- Usually, initializing from a random uniform or normal distribution is fine
- There is some literature on this, though
- A bad initialization of matrix weights combined with a “saturating” activation function can lead to dead or frozen neurons right from the start. Thus, it seems best to initialize them as small values:
 - Keras's default initialization is to sample from a random uniform distribution of the range $[-h, h]$, where $h = \left(\frac{6}{\# inputs + \# outputs} \right)^{\frac{1}{2}}$
 - (Glorot & Bengio, AISTATS 2010 - <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>)

Hyperparameters and Optimization

- Consider the previous DNN, the matrix and vector weights are the parameters of the model that are being optimized with SGD to fit data
- “Hyperparameters” are architectural parameters one step above this
 - What activation function should I use?
 - How many layers should I use?
 - How many neurons should each layer have?
 - What kind of “regularization” should I use?
- Hyperparameters are often optimized using genetic or stochastic algorithms, and there are many packages available

Feed-forward, fully-connected, sparsity, CNNs, and RNNs

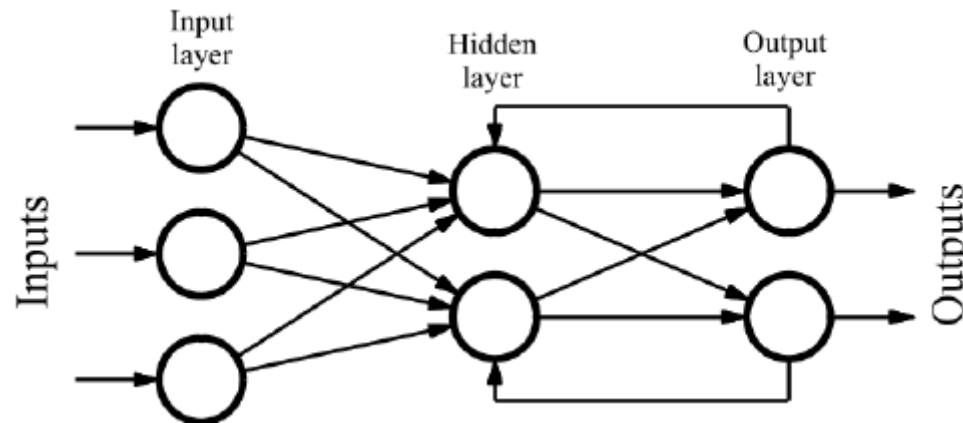
- Before, I mentioned “fully-connected” and “feed-forward”
- Fully-connected (or “dense”) means every neuron in one layer is connected to every neuron in the next layer (i.e., the model’s matrices are dense and not sparse)



<http://cse22-iiith.vlabs.ac.in/exp4/index.html>

Feed-forward, fully-connected, sparsity, CNNs, and RNNs

- Feed-forward simply means there is no recursion in the model
- Recurrent Neural Networks (RNNs) have recursion in the model
 - Meaning a layer's output is used as input in a previous layer, leading to recursion
 - RNNs are often used to learn sequences



https://www.researchgate.net/figure/Graph-of-a-recurrent-neural-network_fig3_234055140

Feed-forward, fully-connected, sparsity, CNNs, and RNNs

- Convolutional Neural Networks (CNNs) apply a kernel / window over the previous layer
 - Caveat, I don't know much about these
 - It's the same as splitting the input into windows of a certain size and training a small neural network on that
 - CNNs are often used to learn “features,” which are fed into other parts of a more complex neural network
 - After applying a convolution kernel and an activation function, usually a “pooling” layer is used to compute some sort of norm over a neighborhood (max, avg, etc)
 - Very frequently used in image processing
 - They are often used to exploit spatial structure

Regularization

- Overfitting to your training data means you won't generalize well
- Regularization is some means of penalizing overfitting
- Weight regularization penalizes sharp variation in matrix weights
 - How do I know when to use it? If test error is much larger than training error
 - How do I know how much to use? Start small and increase until test error = training error
 - If you use too much, your training error doesn't converge (underfitting)
 - If you use too little, your test error doesn't converge (overfitting)
- Dropout regularization
 - Randomly zero out a certain percentage of your outputs each epoch
 - Works surprisingly well (often people drop 50%, but it varies)
 - How much do I drop? Start small and increase until test error = training error

Supervised and Unsupervised Learning

- Supervised learning means you specify the inputs and outputs your model is trying to predict
 - You explicitly train on inputs and associated outputs
 - “Outputs” can be known category labels, known associated values, the next value in a sequence, etc.
 - Used to identify categories given an input, to replace a subroutine, to model a process, etc.
- Unsupervised learning means you’re letting the optimization structure your data according to some underlying model
 - You do not explicitly train on outputs but only on inputs
 - Used to explore the structure of a dataset (clustering, PCA, compression, etc.)

Auto-Encoder (Example of Unsupervised Learning)

- Objective: Find a compressed state for a given dataset
- Encoder: $h_1 = \phi_1(W_1 x_{In} + b_1)$; h_1 is much smaller than x_{In}
- Decoder: $x_{In} = \phi_2(W_2 h_1 + b_2)$;
- Full model: $x_{In} = \phi_2(W_2 \phi_1(W_1 x_{In} + b_1) + b_2)$
- You train the encoder and decoder simultaneously using the input as the output (i.e., you're trying to reconstitute the same vector from a compressed state)
- Want some basis functions (like PCA)? Set one neuron in the small compressed layer to 1 and the rest to zero, and you get basis functions

Regression and Classification

- Classification seeks to determine a discrete output from the inputs
 - Read handwriting and determine the letter written
 - Read an image and determine what's in it
 - Determine from data what method is most efficient to use
- Regression seeks to determine a continuous output from the inputs
 - Replace a subroutine with a ML model
 - Learn aspects about a process from data

Machine Learning Packages

- Most ML packages are in python
 - [Tensorflow](#)
 - [Keras](#)
 - [Caffe2](#)
 - [Scikitlearn](#)
 - [MXNet](#)
-
- I recommend Keras for ease of entry and Scikitlearn for breadth

Build Your Own SGD-Based Learning Model

- Let's face it, matvecs wrapped in tanh aren't always the best basis functions
- Sometimes, you need to optimize something more complex
 1. Create a set of training data
 2. Create a routine that computes a loss function based on a batch of inputs
 3. Create a routine that computes the gradient of the loss function wwith respect to parameters
 4. Initialize your parameters with small random values
 5. For each epoch
 1. For each minibatch
 1. Accumulate the loss over the minibatch by calling your function
 2. Compute the gradient of the accumulated loss function with respect to model parameters
 3. Update your parameters in negative gradient direction, given your learning rate
- Also, check out: <https://www.pyimagesearch.com/2016/10/17/stochastic-gradient-descent-sgd-with-python/>

Scattered Thoughts on ML

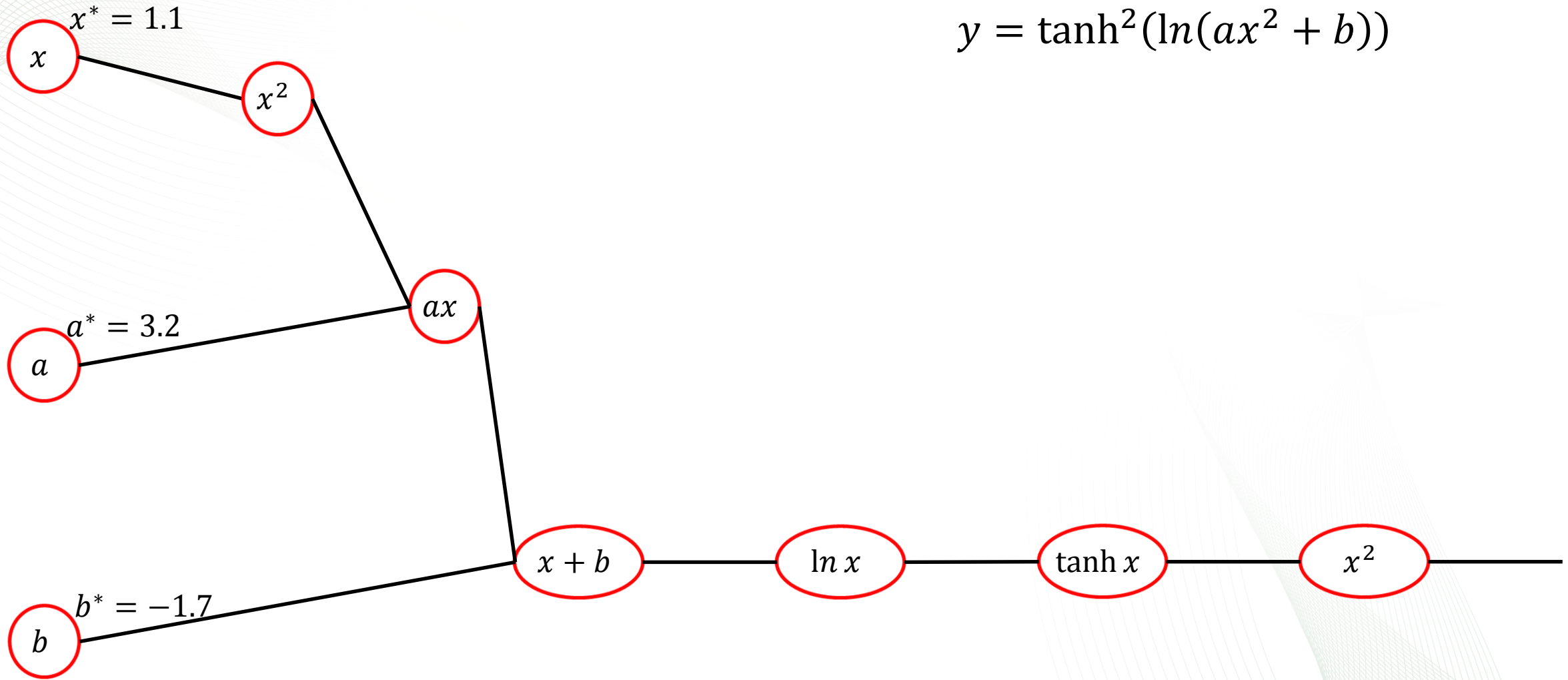
- The larger the model, the harder it is to train
 - Splitting a large model into a collection of smaller ones is easier to train and give better results
- The more discontinuous the process, the worse DNNs will do
 - DNNs seem best suited to problems already fairly well-described by linear operators
 - Identifying a large discontinuity and splitting into multiple DNNs is more accurate
- When an output has a large range over the samples, smaller-magnitude cases train poorly. However, often these matter less anyway.
 - I'm not sure how to fix this. SGD responds to error magnitude.
- Be careful blindly normalizing inputs and outputs. Amplifying small-magnitude, noisy values will hijack the training with useless noise
 - You need to pay attention to what the inputs and outputs are. Domain knowledge is necessary. These aren't black boxes.
- If an input or output doesn't vary across samples, remove it from the model
 - They simply bloat the model, making it noisier and harder to train

Back Propagation Example

- $y = \tanh^2(\ln(ax^2 + b))$
 - You must split the function into elemental pieces
 - $f_1(x) = x^2$; $f_2(x) = \tanh x$; $f_3(x) = \ln x$; $f_4(x) = x + b$; $f_5(x) = ax$; $f_6(x) = x^2$
 - Must be able to differentiate each function with respect to all parameters
 - $y = f_1 \left(f_2 \left(f_3 \left(f_4 \left(f_5 (f_6(x)) \right) \right) \right) \right)$
 - Solve with $x = 1.1$; $a = 3.2$; $b = -1.7$

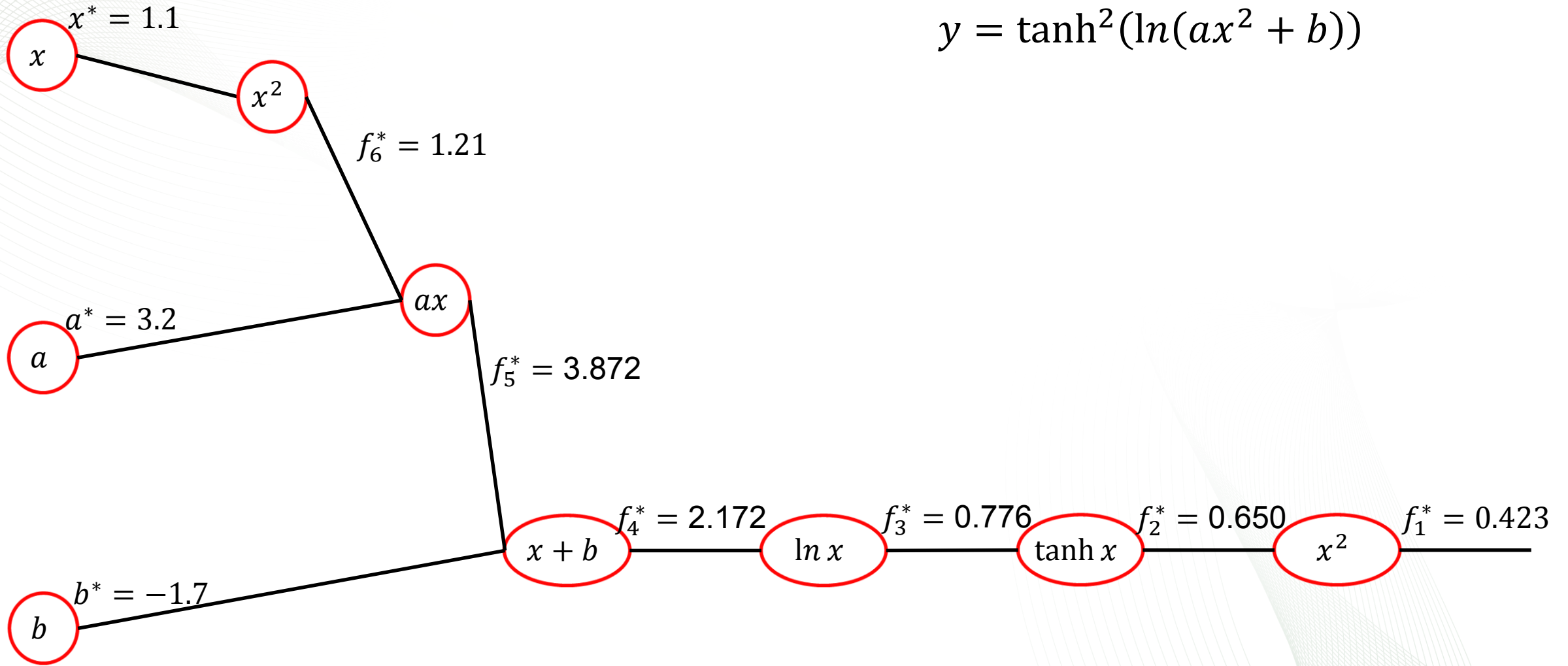
Back Propagation Example

$$y = \tanh^2(\ln(ax^2 + b))$$

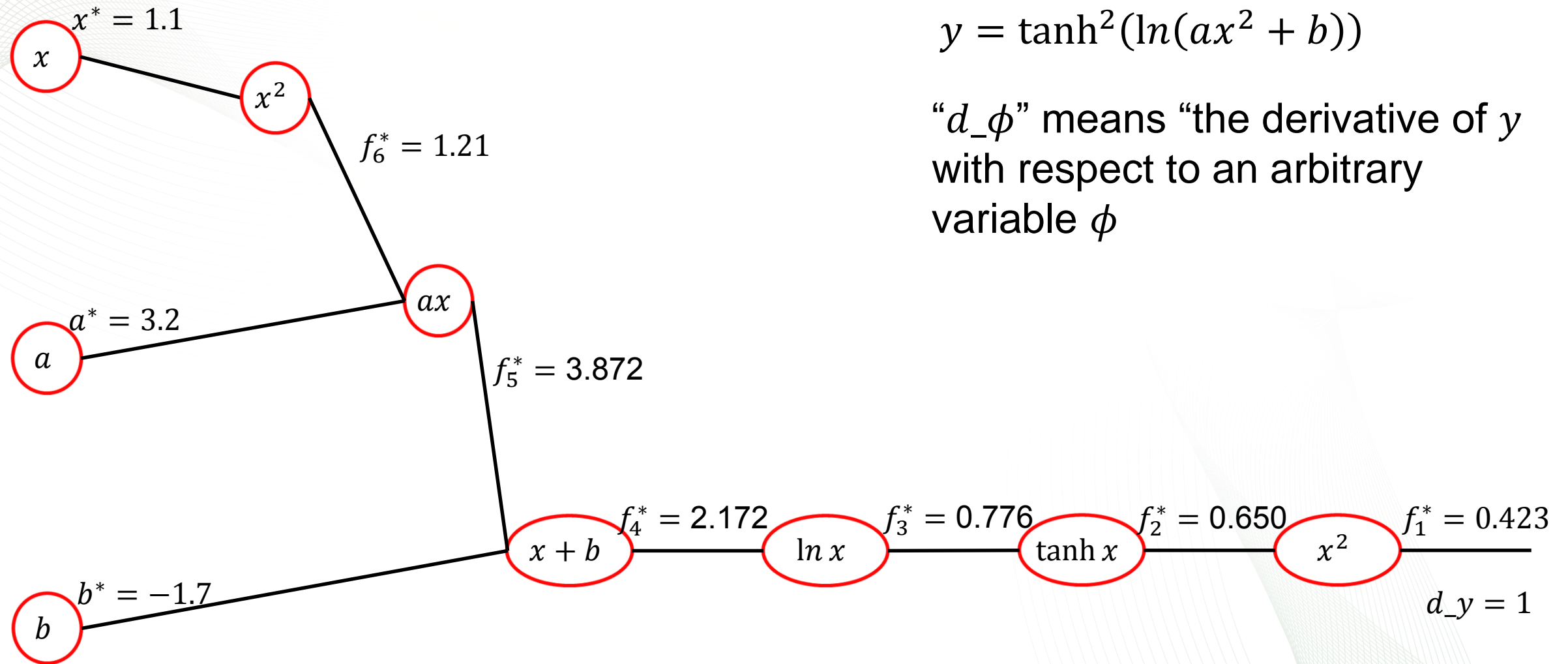


Back Propagation Example

$$y = \tanh^2(\ln(ax^2 + b))$$



Back Propagation Example

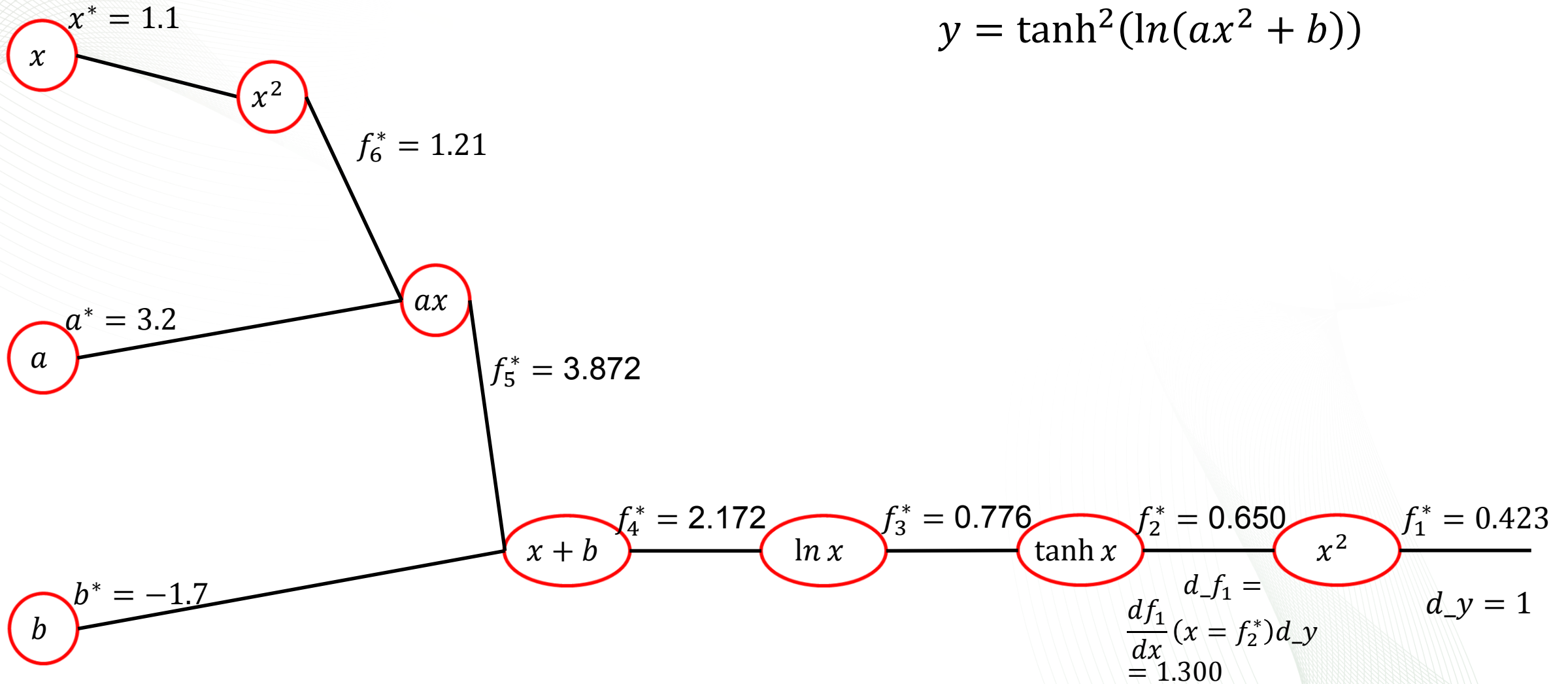


$$y = \tanh^2(\ln(ax^2 + b))$$

“ d_ϕ ” means “the derivative of y with respect to an arbitrary variable ϕ ”

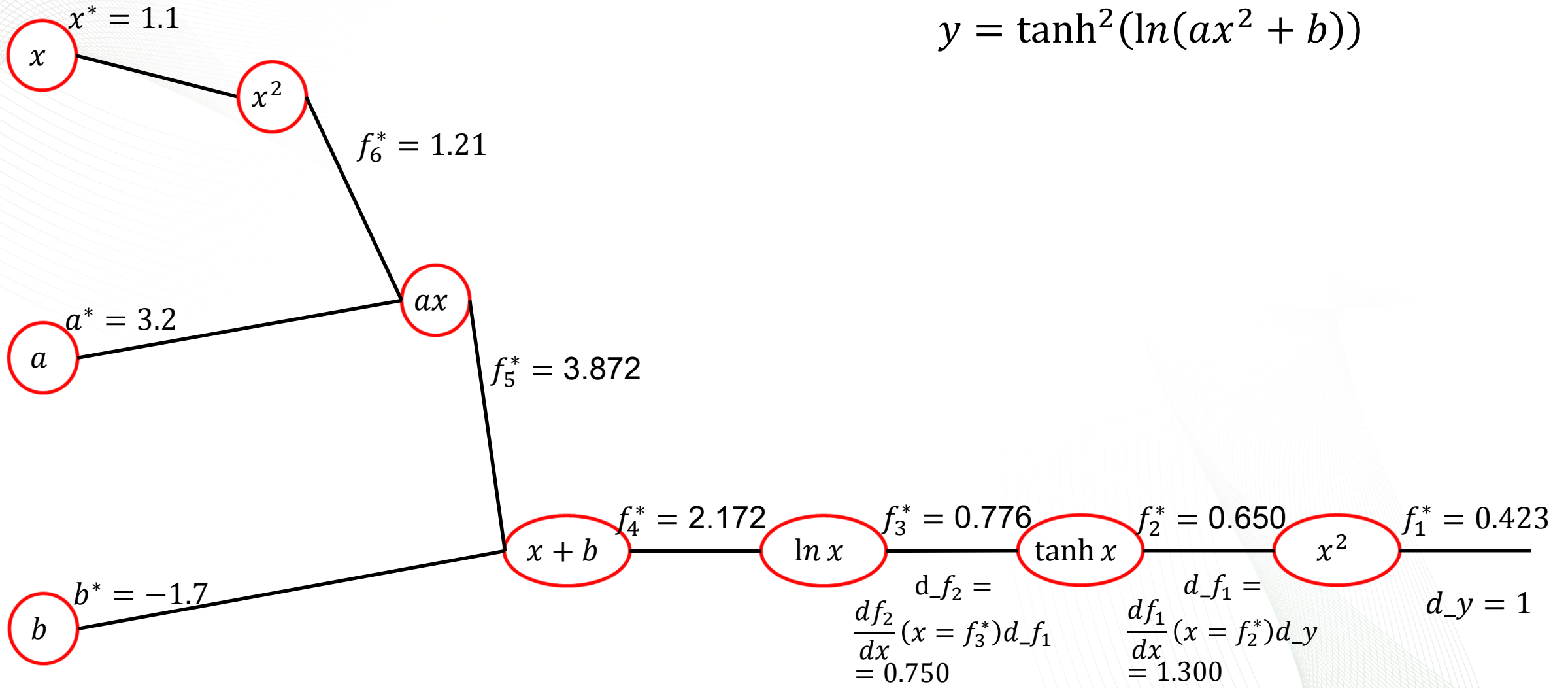
Back Propagation Example

$$y = \tanh^2(\ln(ax^2 + b))$$



Back Propagation Example

$$y = \tanh^2(\ln(ax^2 + b))$$



Back Propagation Example

$$y = \tanh^2(\ln(ax^2 + b))$$

