BST 261: Data Science II
Lecture 2
Perceptrons, Backpropagation and MLPs

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2019

Neural Nets

What is a neural net?

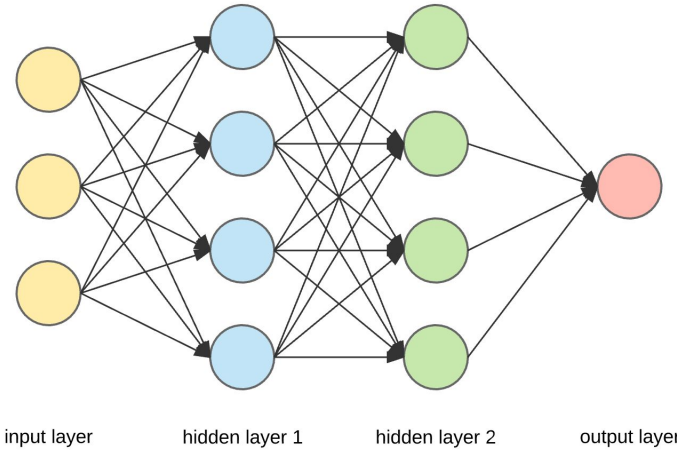
A neural net is composed of 3 things:

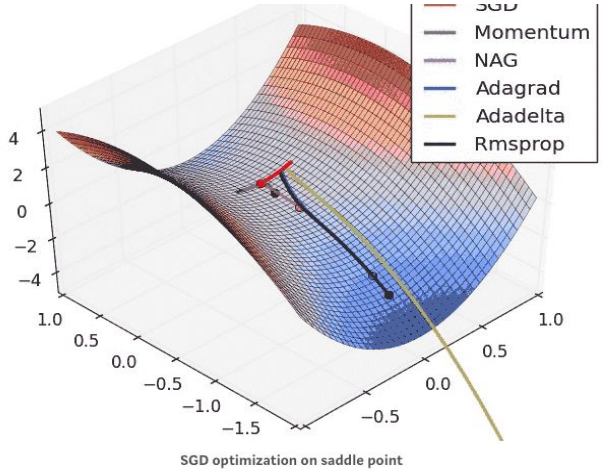
The network structure

The loss function

The optimizer

$$-y_i * \log(p_i) - (1 - y_i) * \log(1 - p_i)$$



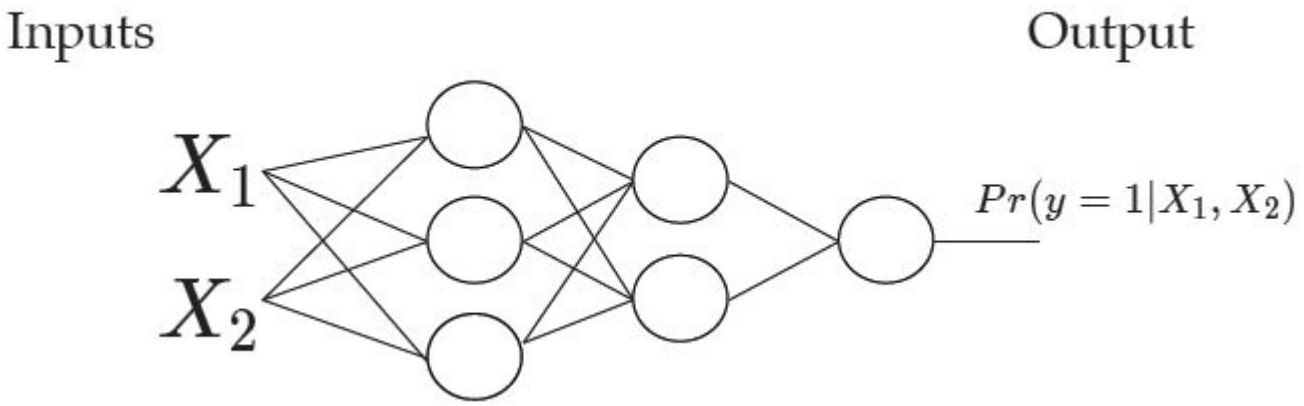


https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

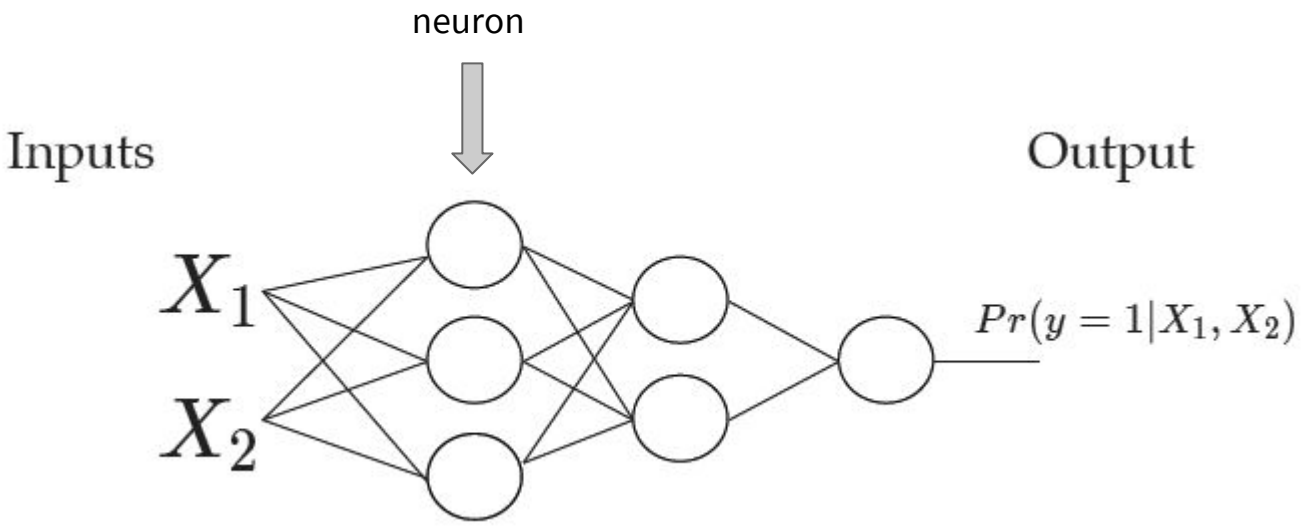
Neural Net Structure

- A neural net is a modular way to build a classifier



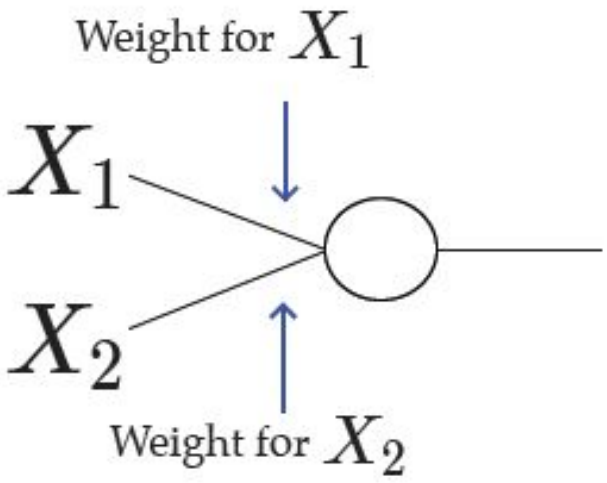
Neural Net Structure

- A neural net is a modular way to build a classifier
- The neuron is the basic functional unit in a neural network



Neurons

- A neuron does 2 things:



1) Weighted sum of inputs

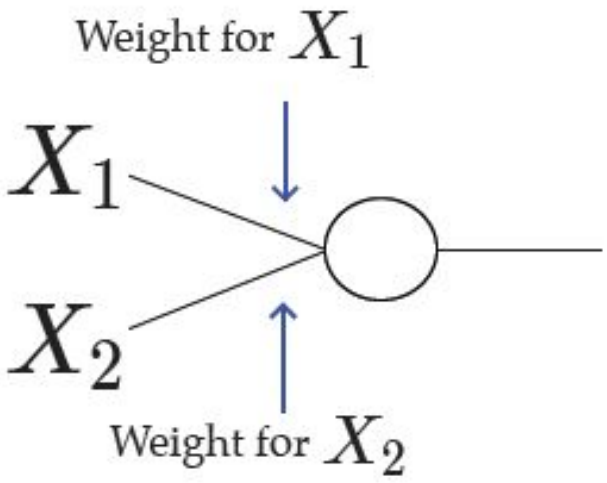
$$w_1 * X_1 + w_2 * X_2$$

2) Nonlinear transformation

$$\phi(w_1 * X_1 + w_2 * X_2)$$

Neurons

- A neuron does 2 things:



1) Weighted sum of inputs

$$w_1 * X_1 + w_2 * X_2$$

2) Nonlinear transformation

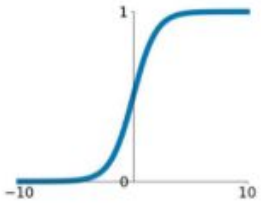
$$\phi(w_1 * X_1 + w_2 * X_2)$$

Activation function - a non-linear transformation

Activation Functions

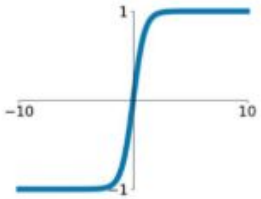
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



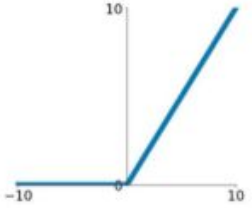
tanh

$$\tanh(x)$$



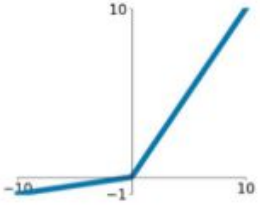
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

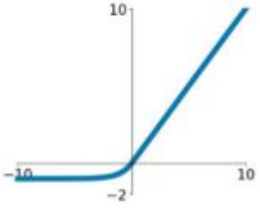


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

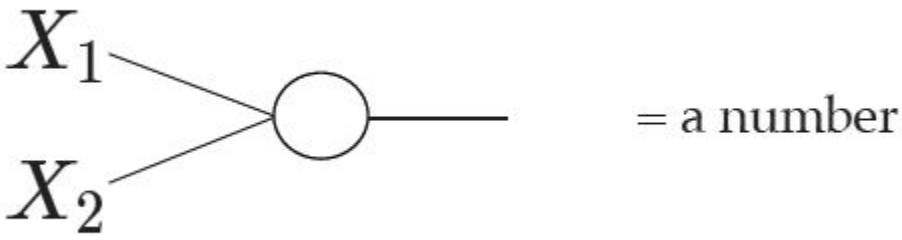
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c

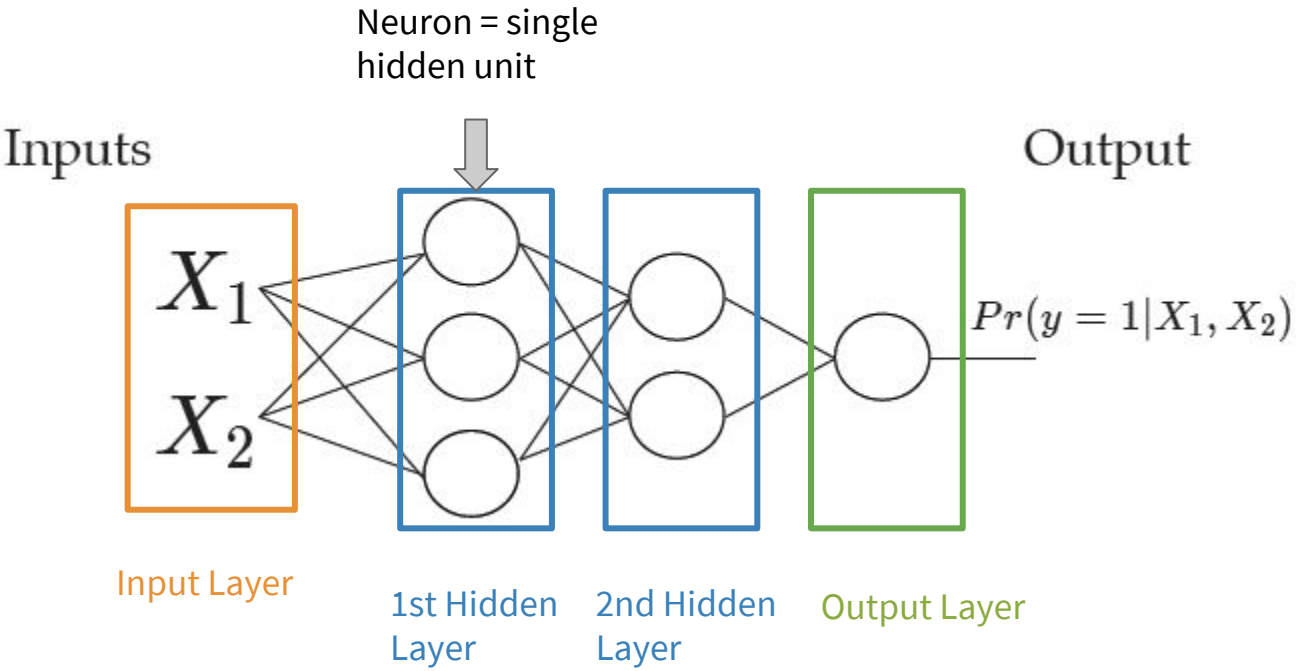
Neuron

- A neuron produces a single number that is a nonlinear transformation of its input connections



Neural Net Structure

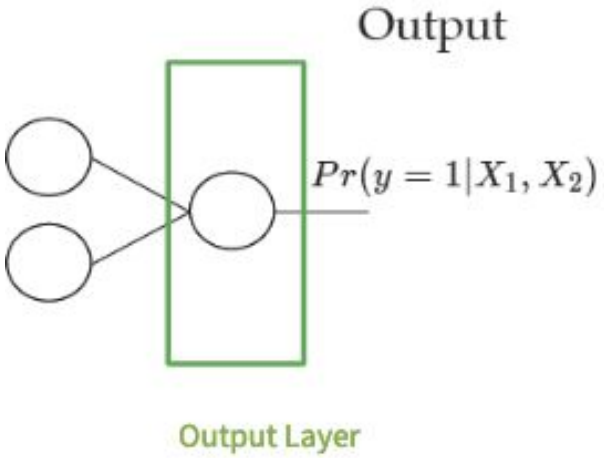
- Neural nets are organized into **layers**



Loss Functions

- We need a way to measure how well the network is performing
  - Is it making good predictions?

- A **loss function** is a function that returns a single number which indicates how closely a prediction matches the ground truth, i.e. the actual label
  - We want to minimize the loss to achieve more accurate predictions
  - Also known as the objective function, cost function, loss, etc.



Loss Functions

One of the simplest loss functions is binary cross-entropy which is used for binary classification

y_i is the true label

$$p_i = \mathrm{P}(y_i = 1 | X_1, X_2)$$

$$l(y_i, p_i) = -y_i * log(p_i) - (1 - y_i) * log(1 - p_i)$$

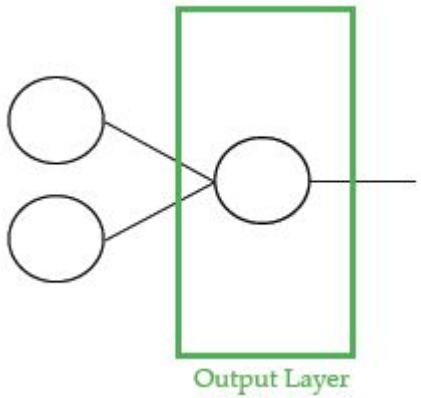
y	p	Loss
0	0.1	0.1
0	0.9	2.3
1	0.1	2.3
1	0.9	0.1

Output Layer and Loss

- The output layer needs to "match" the loss function
- Correct shape
- Correct scale
- For binary cross-entropy, the network needs to produce a single probability:

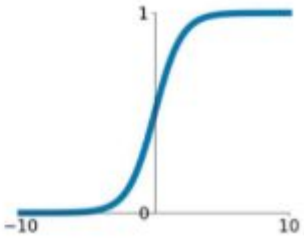
$$p_i = \mathrm{P}(y_i = 1 | X_1, X_2)$$

- One unit in the output layer represents this probability
- Activation function must "squash" the output to be between 0 and 1



Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



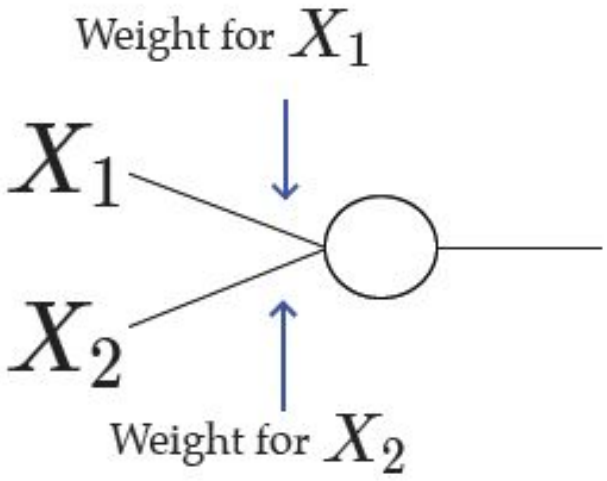
Output Layer and Loss

- We can change the output layer and loss to model many different kinds of data

Task	Last-layer activation	Loss function
Binary classification	sigmoid	Binary cross-entropy
Multiclass, single-label classification	softmax	Categorical cross-entropy
Multiclass, multilabel classification	sigmoid	Binary cross-entropy
Regression to arbitrary values	None	Mean square error (MSE)
Regression to values between 0 and 1	sigmoid	MSE or binary cross-entropy

The Optimizer

Remember this?



1) Weighted sum of inputs

$$w_1 * X_1 + w_2 * X_2$$

2) Nonlinear transformation

$$\phi(w_1 * X_1 + w_2 * X_2)$$

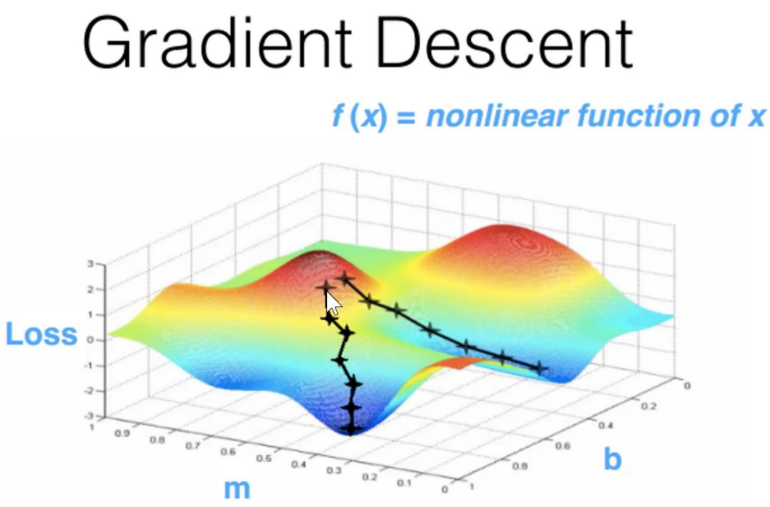
We have specified the network and the loss function, but what about values for the weights? We want weights that minimize the loss function - how do we calculate them?

The Optimizer

Q: How do we minimize the loss function?

A: Stochastic Gradient Descent (SGD)

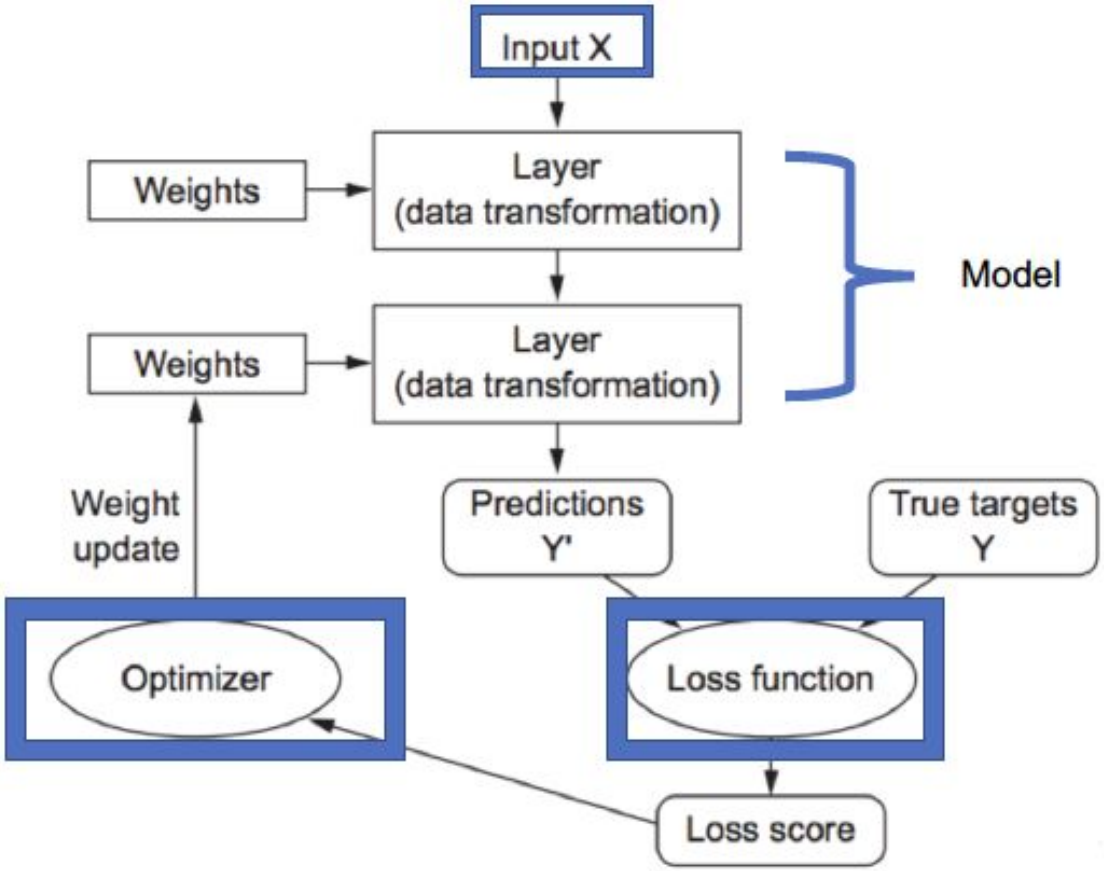
1. Give weight random initial values
2. Evaluate the partial derivative of each weight with respect to the negative log-likelihood at the current weight value on a mini-batch
3. Take a step in the direction opposite the gradient
4. Repeat



The Optimizer

- Many variations on the basic idea of SGD are available
  - Rmsprop
  - Adagrad
  - Adadelta
  - Momentum
  - NAG
  - etc.

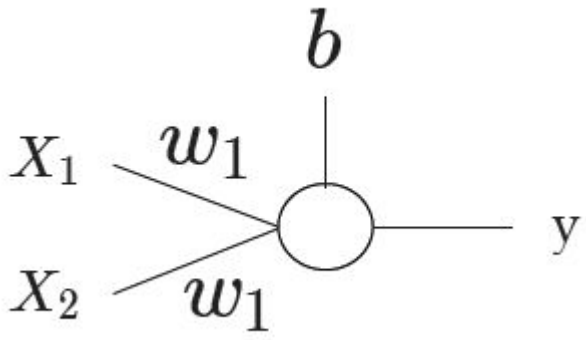
Workflow


Input X
Weights
Layer
(data transformation)
Model
Weights
Layer
(data transformation)
Weight
update
Predictions
Y'
True targets
Y
Optimizer
Loss function
Loss score

Perceptrons

Perceptrons

- Let's put this all together
- Out first network will be a single neuron that will learn a simple function

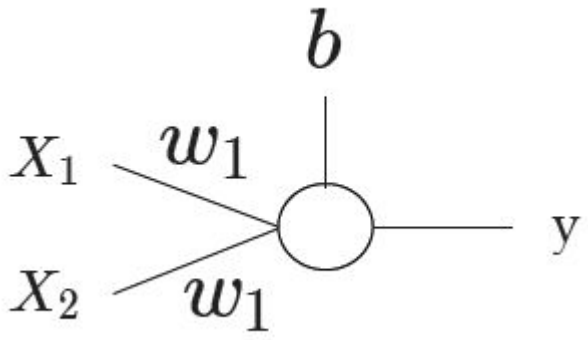


Observations

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Perceptrons

- How do we make a prediction for each observation?



Assume the following values:

w1	w2	b
1	-1	-0.5

Observations

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Predictions

- For the first observation, $X_1 = 0, X_2 = 0, y = 0$
- First compute the weighted sum:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$h = 1 * 0 + -1 * 0 + (-0.5)$$

$$h = -0.5$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Observations

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Predictions

- For the first observation, $X_1 = 0, X_2 = 0, y = 0$
- First compute the weighted sum:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$h = 1 * 0 + -1 * 0 + (-0.5)$$

$$h = -0.5$$

Transform to a probability:

$$p = \frac{1}{1+\exp(-h)}$$

$$p = \frac{1}{1+\exp(-0.5)}$$

$$p = 0.38$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Predictions

- For the first observation, $X_1 = 0, X_2 = 0, y = 0$
- First compute the weighted sum:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$h = 1 * 0 + -1 * 0 + (-0.5)$$

$$h = -0.5$$

Transform to a probability:

$$p = \frac{1}{1+\exp(-h)}$$

$$p = \frac{1}{1+\exp(-0.5)}$$

$$p = 0.38$$

Round to get prediction:

$$\hat{y} = \text{round}(p)$$

$$\hat{y} = 0$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Predictions

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

$$\hat{y} = \text{round}(p)$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Complete the table:

X1	X2	y	h	p	$\hat{y}$
0	0	0	-0.5	0.38	0
0	1	1			
1	0	1			
1	1	1			

Predictions

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

$$\hat{y} = \text{round}(p)$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Complete the table:

X1	X2	y	h	p	$\hat{y}$
0	0	0	-0.5	0.38	0
0	1	1	-1.5	0.18	0
1	0	1	0.5	0.38	0
1	1	1	-0.5	0.38	0

Performance

- Our network isn't so great
- How do we make it better?
- What does *better* mean?
  - Need to define a measure of performance
  - There are many ways

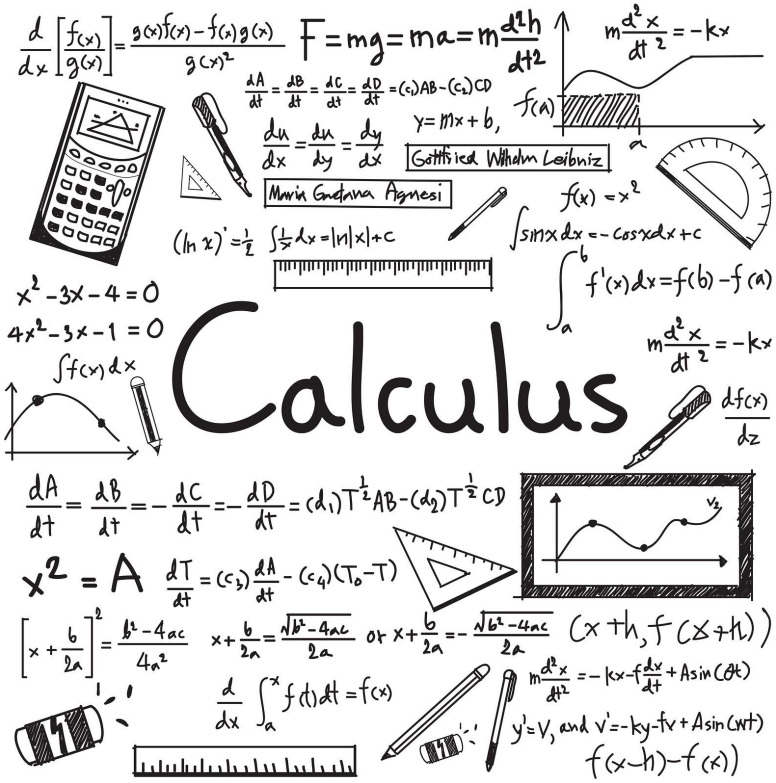
- Let's begin with squared error: $(y - p)^2$
- We need to find values for w_1, w_2, b that make this error as small as possible.
- We need to **learn** values for w_1, w_2, b such that the difference between the predicted and actual values is as small as possible.

Learning From Data

How do we find the best values for w_1, w_2, b ?

Learning From Data

How do we find the best values for w_1, w_2, b ?

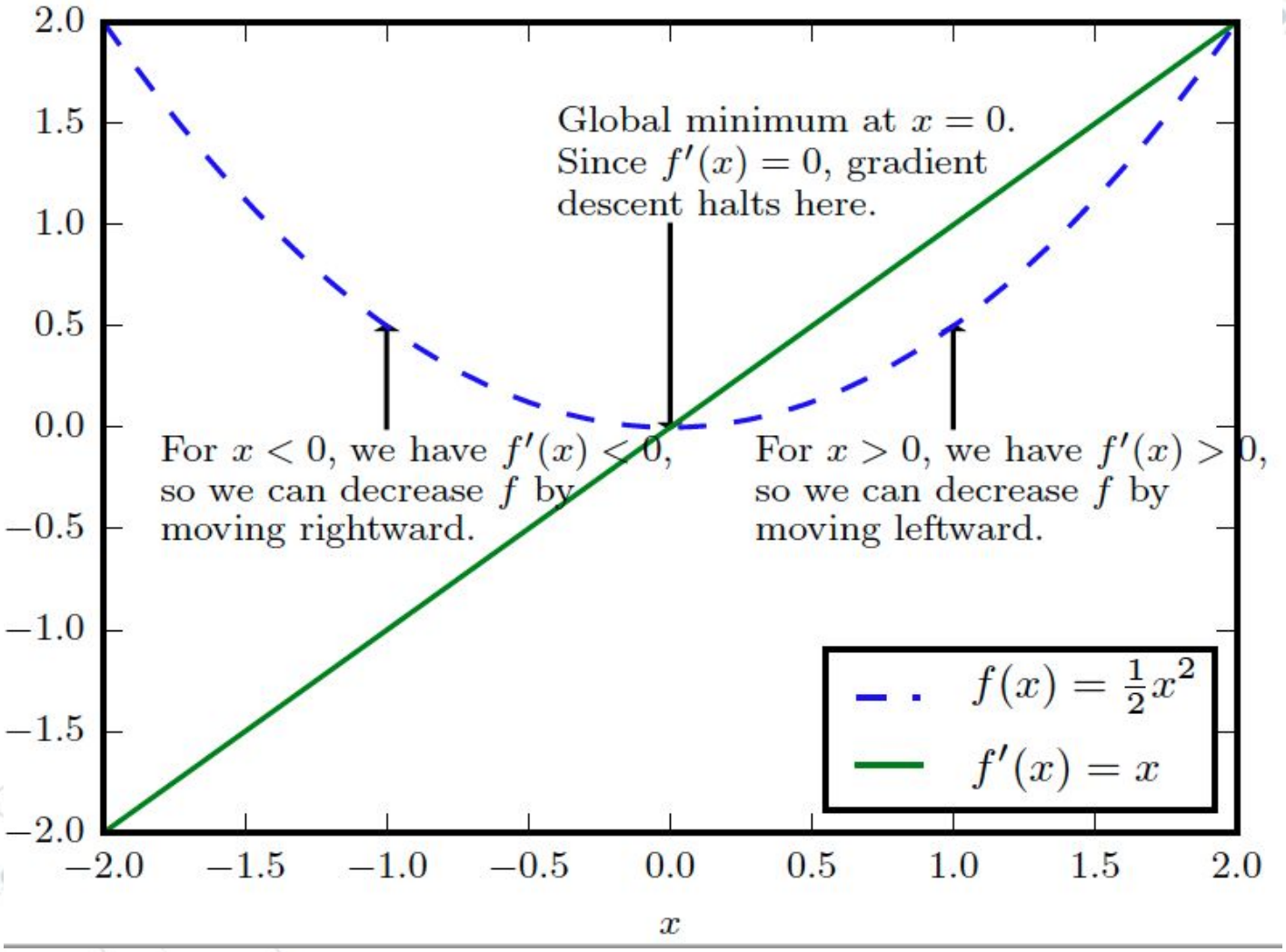


Learning From Data

- Recall that the derivative of a function tells you how it is changing at any given location.
  - If the derivative is positive, it means it's going up
  - If the derivative is negative, it means it's going down

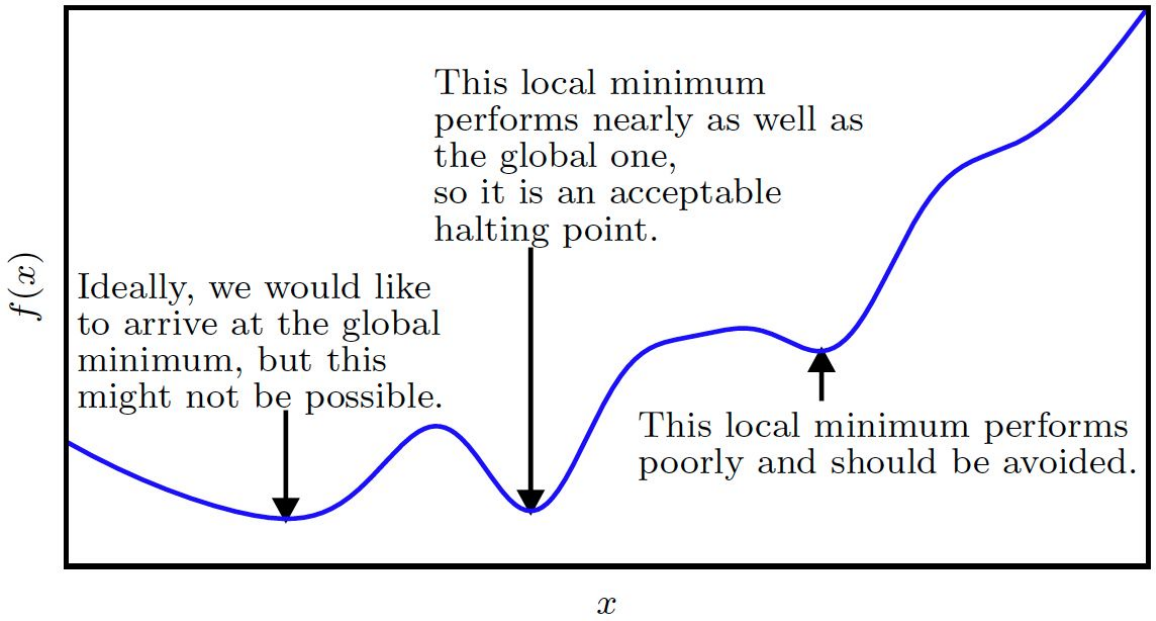
- Strategy:
  - Start with initial values for w_1, w_2, b
  - Take partial derivatives of the loss function with respect to w_1, w_2, b
  - Subtract the derivative (also called the **gradient**) from each
  - This is known as **gradient descent**

Gradient-Based Optimization



Gradient-Based Optimization

- A point that obtains the absolute lowest value of *f(x)* is a global minimum
- There may be one global minimum or multiple global minima
- It is also possible for there to be local minima that are not globally optimal
- It is common in many settings to settle for a value *f* that is very low but not necessarily minimal



Gradient-Based Optimization

- To minimize *f*, we would like to find the direction in which *f* decreases the fastest
- It can be shown that the gradient points directly uphill and the negative gradient directly downhill
- We can therefore decrease *f* by moving in the direction of the negative gradient
- For example, for a weight w_i

$$w_i^{\text{new}} = w_i^{\text{old}} - \eta g$$

where η is the **learning rate** (how fast you want to move down the gradient), and g is the gradient

The Backpropagation Algorithm

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

The Backpropagation Algorithm

- Our perceptron performs the following computations:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

- We want to minimize this quantity:

$$l = (y - p)^2$$

- We'll compute the gradients for each parameter by "backpropagating" errors through each component of the network

The Backpropagation Algorithm

For w_1 we need to compute

$$\frac{\partial l}{\partial w_1}$$

To get there we will use the chain rule

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

This is “backprop”

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$l = (y - p)^2$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} =$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

$$\boxed{\frac{\partial p}{\partial h}} =$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$\boxed{p = \frac{1}{1+\exp(-h)}}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\frac{\partial l}{\partial p} = 2 * (p - y)$$

$$\frac{\partial p}{\partial h} = p * (1 - p)$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$l = (y - p)^2$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\frac{\partial l}{\partial p} = 2 * (p - y)$$

$$\frac{\partial p}{\partial h} = p * (1 - p)$$

$$\frac{\partial h}{\partial w_1} =$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$l = (y - p)^2$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\frac{\partial l}{\partial p} = 2 * (p - y)$$

$$\frac{\partial p}{\partial h} = p * (1 - p)$$

$$\frac{\partial h}{\partial w_1} = X_1$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$l = (y - p)^2$$

Putting it all together:

$$\frac{\partial l}{\partial w_1} = 2 * (p - y) * p * (1 - p) * X_1$$

Gradient Descent with Backprop

For some number of iterations:

1. Compute the gradient for w_1
2. Update w_1

$$w_i^{\text{new}} = w_i^{\text{old}} - \eta g$$

3. Repeat until “convergence”

Do this for each weight and bias term.

Multilayer Perceptrons

Perceptron ⇨ MLP

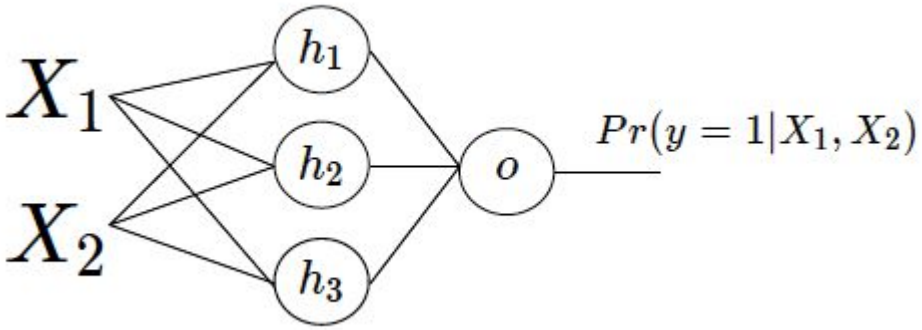
We can turn our perceptron model into a multilayer perceptron

- Instead of just one linear combination, we are going to take several, each with a different set of weights
- Each linear combination will be followed by a nonlinear activation
- Each of these nonlinear features will be fed into the logistic regression classifier (binary classifier)
- All of the weights are learned end-to-end via SGD

MLPs learn a set of nonlinear features directly from data - “feature engineering” is the hallmark of deep learning approaches

Multilayer Perceptrons (MLPs)

Suppose we have the following MLP with 1 hidden layer that has 3 hidden units:



Each neuron in the hidden layer is going to do exactly the same thing as before.

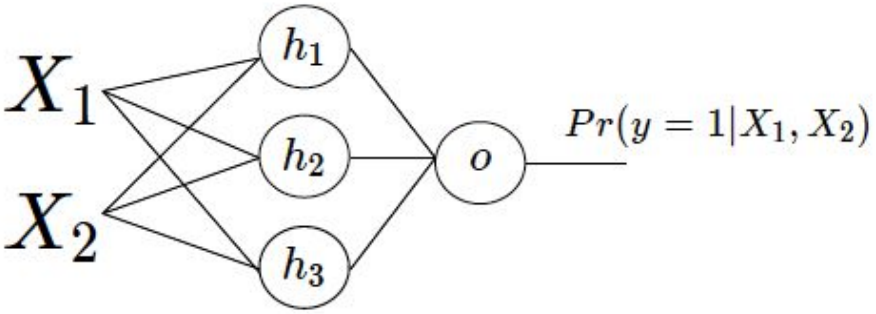
Multilayer Perceptrons (MLPs)

Computations:

$$h_j = \phi(w_{1j} * X_1 + w_{2j} * X_2 + b_j)$$

$$o = b_o + \sum_{j=1}^{3} w_{oj} * h_j$$

$$p = \frac{1}{1+\exp(-o)}$$



*If we use a sigmoid activation function

Output layer weight derivatives

$$\frac{\partial l}{\partial w_{oj}} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial o} * \frac{\partial o}{\partial w_{oj}}$$

$$= (p - y) * p * (1 - p) * h_j$$

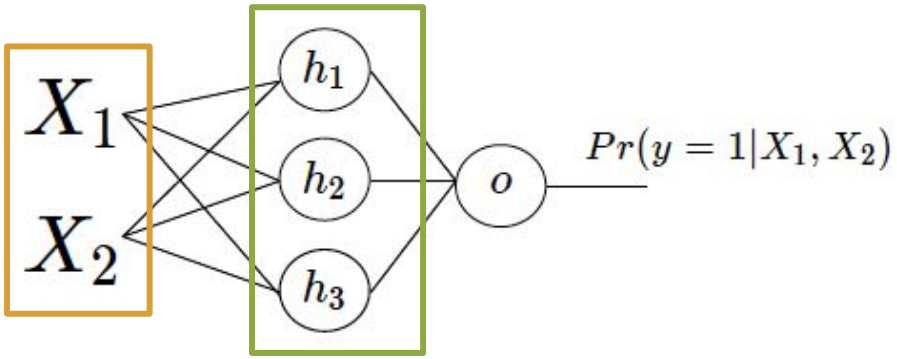
Hidden layer weight derivatives

$$\frac{\partial l}{\partial w_{1j}} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial o} * \frac{\partial o}{\partial h} * \frac{\partial h}{\partial w_{1j}}$$

$$= (p - y) * p * (1 - p) * h_j * (1 - h_j) * X_1$$

Matrix Notation

Sum notation starts to get unwieldy quickly. We can use matrix notation to represent each calculation in a more concise way.



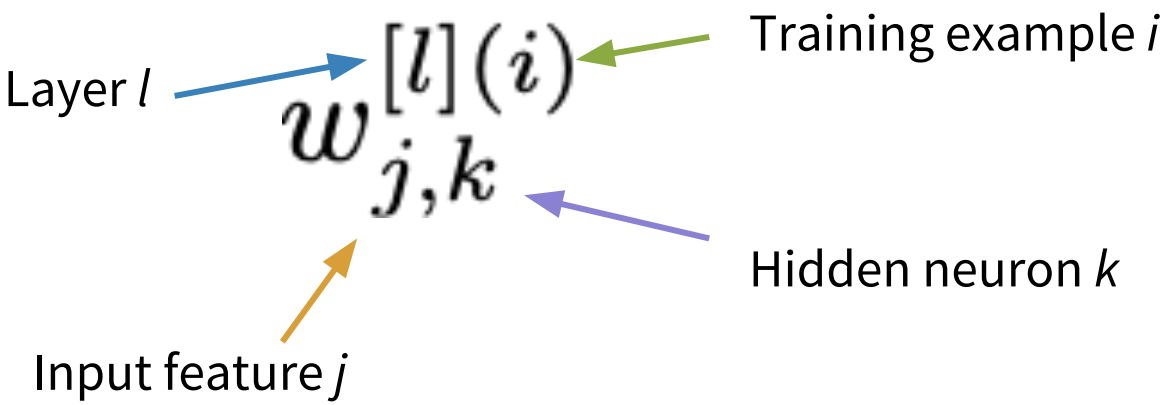
$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \quad W = \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{3,2} \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$Z = W^T X + B \qquad H = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \phi(Z)$$

Notation

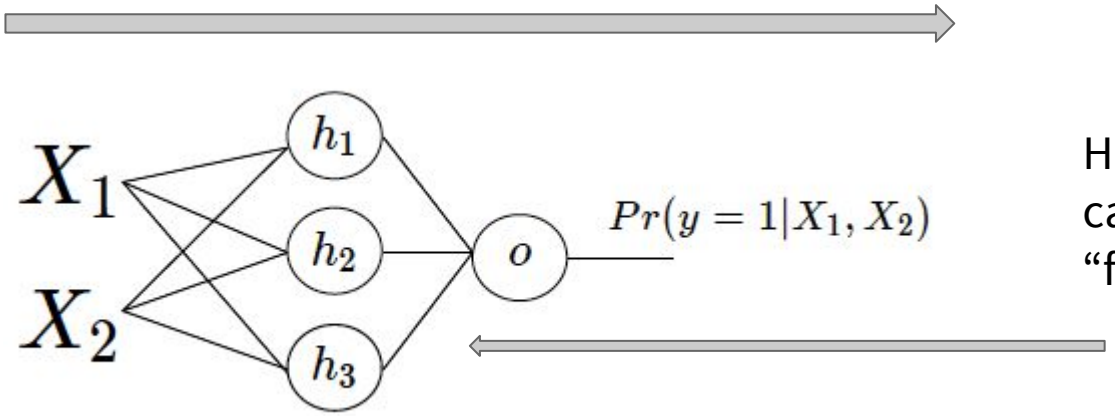
As the number of layers grows, the number of matrices grows and we have to add a superscript to denote the layer. We also have to add a superscript to denote which training example we are referencing.

Example notation for 1 weight in 1 hidden layer for 1 training example:



MLP Terminology

Forward pass = computing probability from input

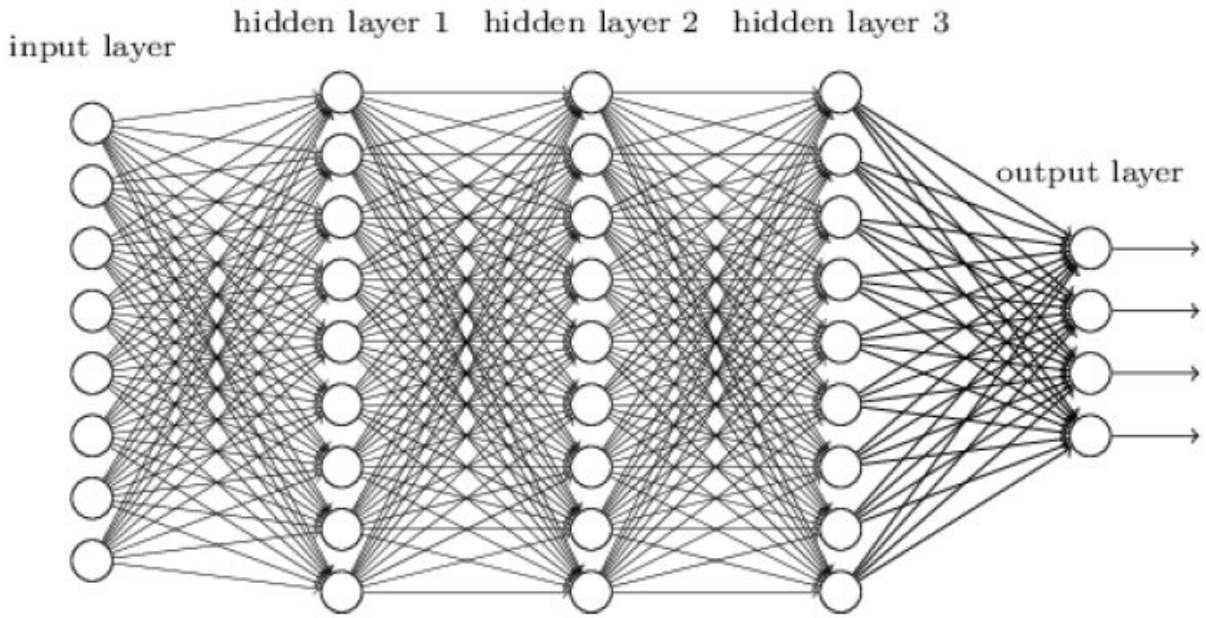


Hidden layers are also called “dense” layers or “fully connected” layers

Backward pass = computing derivatives from the output

MPLs

Increasing the number of layers increases the flexibility of the model - but run the risk of overfitting



Regularization

Regularization

- One of the biggest problems with neural networks is overfitting.
- Regularization schemes combat overfitting in a variety of different ways
- A perceptron represents the following optimization problem:

$$\text{argmin}_W l(y, f(X))$$

where

$$f(X) = \frac{1}{1+\exp(-\phi(XW))}$$

Regularization

One way to regularize is to introduce penalties and change

$$\text{argmin}_W l(y, f(X))$$

To

$$\text{argmin}_W l(y, f(X)) + \lambda R(W)$$

where R(W) is often the L1 or L2 norm of W. These are the well-known ridge and LASSO penalties, and referred to as **weight decay** by the neural net community.

L2 Regularization

We can limit the size of the L2 norm of the weight vector:

$$\text{argmin}_W l(y, f(X)) + \lambda \|W\|_2$$

Where

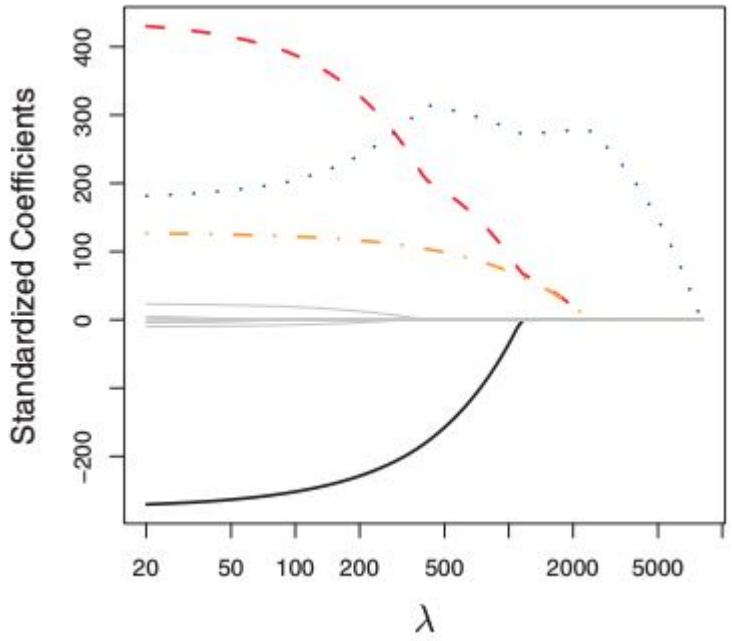
$$\|W\|_2 = \sum_{j=1}^{p} w_j^2$$

We can do the same for the L1 norm. What do the penalties do?

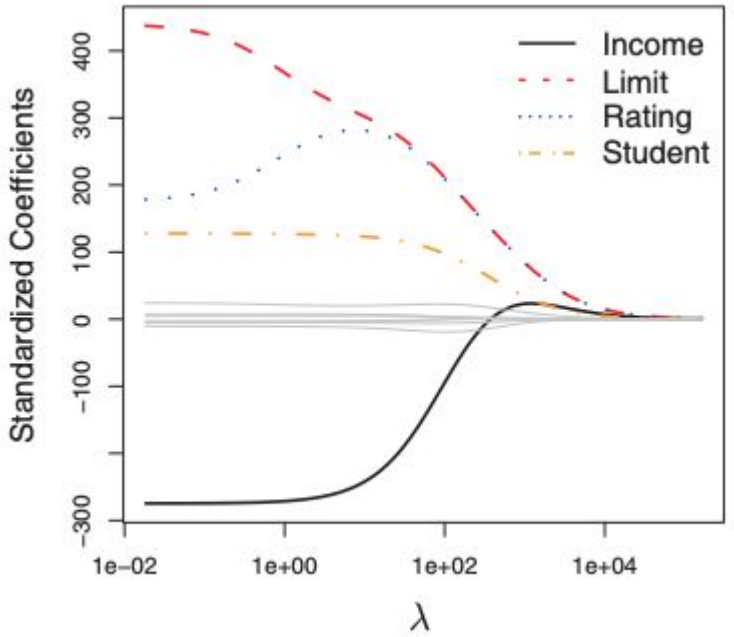
Shrinkage

The L1 and L2 penalties shrink the weights towards 0.

L2 Penalty

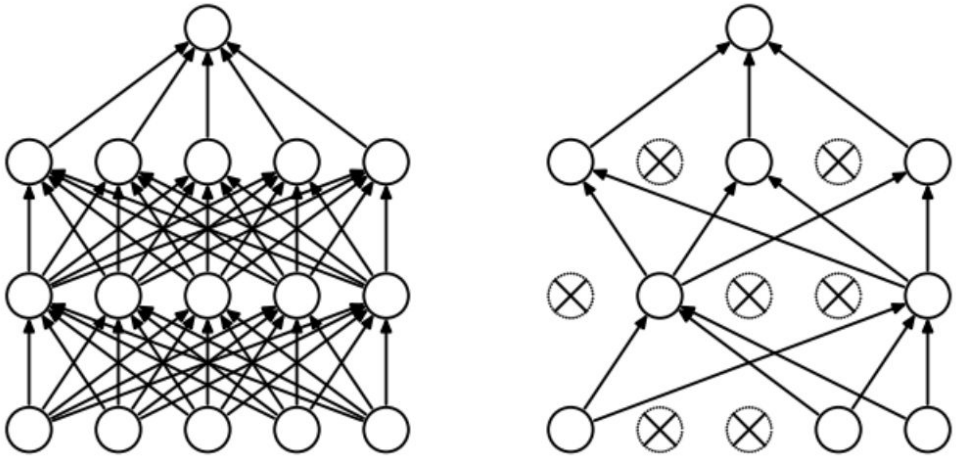


L1 Penalty



Stochastic Regularization

- Why is this a good idea?
- Often, we will inject noise into the neural network during training
  - The most popular way to do this is **dropout**
- Given a hidden layer, we are going to set each element of the hidden layer to 0 with probability *p* each SGD update.



(a) Standard Neural Net

(b) After applying dropout.

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Stochastic Regularization

- One way to think of this is the network is trained by bagged versions of the network.
- **Bagging** reduces variance.
- Others have argued this is an approximate Bayesian model

Dropout as a Bayesian Approximation:
Representing Model Uncertainty in Deep Learning

Yarin Gal YG279@CAM.AC.UK
Zoubin Ghahramani ZG201@CAM.AC.UK
University of Cambridge

Abstract

Deep learning tools have gained tremendous attention in applied machine learning. However such tools for regression and classification do not capture model uncertainty. In comparison, Bayesian models offer a mathematically grounded framework to reason about model uncertainty, but usually come with a prohibitive computational cost. In this paper we develop a new theoretical framework casting dropout training in deep neural networks (NNs) as approximate Bayesian inference in deep Gaussian processes. A direct result of this theory gives us tools to model uncertainty with dropout NNs – extracting information from existing models that has been thrown away so far. This mitigates the problem of representing uncertainty in deep

With the recent shift in many of these fields towards the use of Bayesian uncertainty (Herzog & Ostwald, 2013; Trafimow & Marks, 2015; Nuzzo, 2014), new needs arise from deep learning tools.

Standard deep learning tools for regression and classification do not capture model uncertainty. In classification, predictive probabilities obtained at the end of the pipeline (the softmax output) are often erroneously interpreted as model confidence. A model can be uncertain in its predictions even with a high softmax output (fig. 1). Passing a point estimate of a function (solid line 1a) through a softmax (solid line 1b) results in extrapolations with unjustified high confidence for points far from the training data. x^* for example would be classified as class 1 with probability 1. However, passing the distribution (shaded area 1a) through a softmax (shaded area 1b) better reflects classification uncertainty far from the training data.

Stochastic Regularization

- Many have argued that SGD itself provides regularization

Journal of Machine Learning Research 18 (2017) 1-35 Submitted 4/17; Revised 10/17; Published 12/17

Stochastic Gradient Descent as Approximate Bayesian Inference

Stephan Mandt STEPHAN.MANDT@GMAIL.COM
Data Science Institute
Department of Computer Science
Columbia University
New York, NY 10025, USA

Matthew D. Hoffman MATHOFFM@ADOBE.COM
Adobe Research
Adobe Systems Incorporated
601 Townsend Street
San Francisco, CA 94103, USA

David M. Blei DAVID.BLEI@COLUMBIA.EDU
Department of Statistics
Department of Computer Science
Columbia University
New York, NY 10025, USA

Editor: Manfred Opper

Abstract

Stochastic Gradient Descent with a constant learning rate (constant SGD) simulates a Markov chain with a stationary distribution. With this perspective, we derive several new results. (1) We show that constant SGD can be used as an approximate Bayesian posterior inference algorithm. Specifically, we show how to adjust the tuning parameters of constant SGD to best match the stationary distribution to a posterior, minimizing the Kullback-Leibler divergence between these two distri-

Initialization Regularization

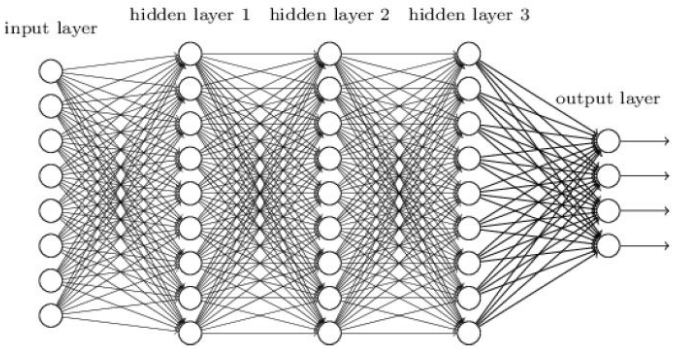
- The weights in a neural network are given random values initially.
- There is an entire literature on the best way to do this initialization
  - Normal
  - Truncated Normal
  - Uniform
  - Orthogonal
  - Scaled by number of connections
  - Etc.
- Try to “bias” the model into initial configurations that are easier to train

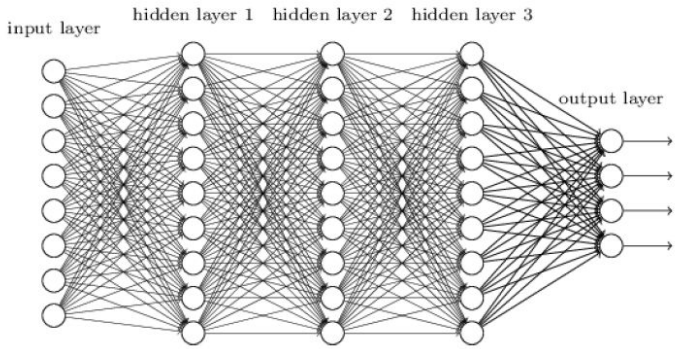
Initialization Regularization

- A popular way is to do **transfer learning**

Train the model on auxiliary task where lots of data is available

Use final weight values from previous task as initial values and “fine tune” on primary task





Structural Initialization

- The key advantage of neural nets is the ability to easily include properties of the data directly into the model through the network's structure
- Convolutional neural networks (CNNs) are a prime example of this

Conclusions

- Backprop, perceptrons, and MLPs are the building blocks of neural nets
- You'll get a chance to demonstrate your mastery in Homework 1
- We will use these concepts for the rest of the semester

Action Items

- Download and install Jupyter Notebook
- I personally prefer using jupyterlab or Jupyter Notebook in Anaconda Navigator
- You are free to use whichever platform you prefer, as long as you turn in a **.ipynb** or **.py** file for your homework assignments