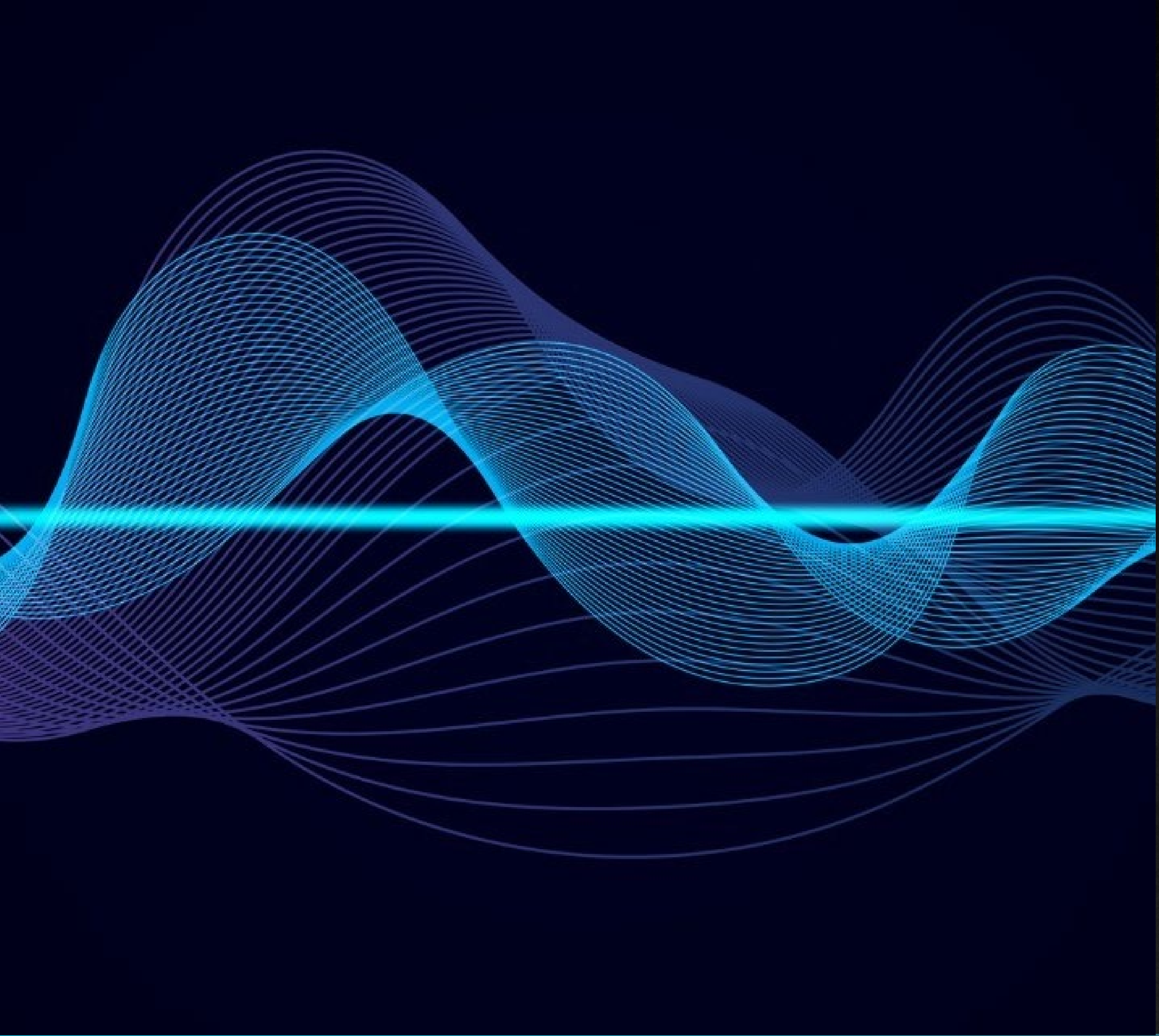


Computational Physics

PHYS 6260

Machine Learning: Neural Networks

Announcements:

- Spring Break next week!
- Project progress report: Due Friday 3/28

Lecture Outline

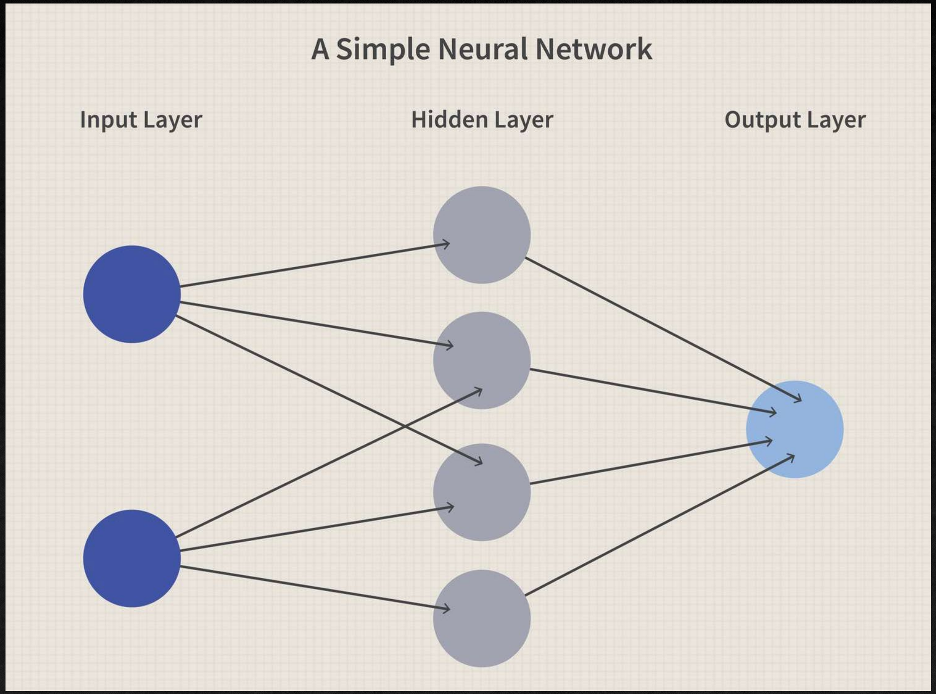
We will cover these topics

- Basics of neural networks (NN)
- Mathematical basis of NN minimization

Machine Learning

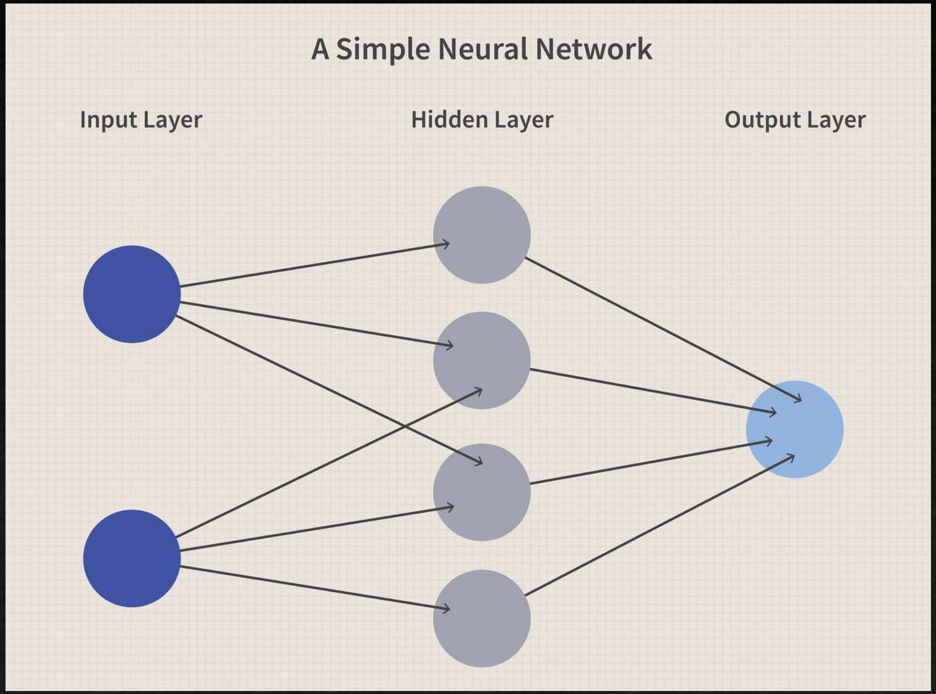
- Machine learning (ML) is a huge and dynamic topic
- For this class, we will focus on neural networks (NN)

- As discussed in the previous lecture, we will want to make some prediction (regression) or classification.
- Use known data: inputs with corresponding outputs
- Predict the output for any input



Neural Network Overview

- Neural networks attempt to mimic the action of neurons in a brain
- Applications generally involve predicting the output from some input or classification
  - Separate populations in some parameter space
- Some uses:
  - Character / image recognition
  - AI for games (the “Go” program that beat a human)
  - Classification of data (e.g. autonomous driving)
  - Quantitative Finance (e.g. stock market trends)



Neural networks

- Computers are good at arithmetic but not great at pattern recognition

- Neural networks attempt to model how neurons transmit information

(original source unknown)

Neural networks

- Basic strategy
  - Create a non-linear fitting routing with free parameters
  - Train a network on data with known input and output to set the parameters
  - Trained network can be used on new inputs to predict outcome

- A linear example
  - Inputs: $x \in \mathbb{R}^n$
  - Outputs: $z \in \mathbb{R}^m$
  - Neural network is a map ($\mathbb{R}^n \rightarrow \mathbb{R}^m$) that can be expressed as a matrix **A**
  - Z = Ax, where A is an m x n matrix
- Given enough input, we can estimate the matrix elements in A

Need for non-linear

- A linear map cannot capture all of these input / output pairs

- We need to find **A** such that

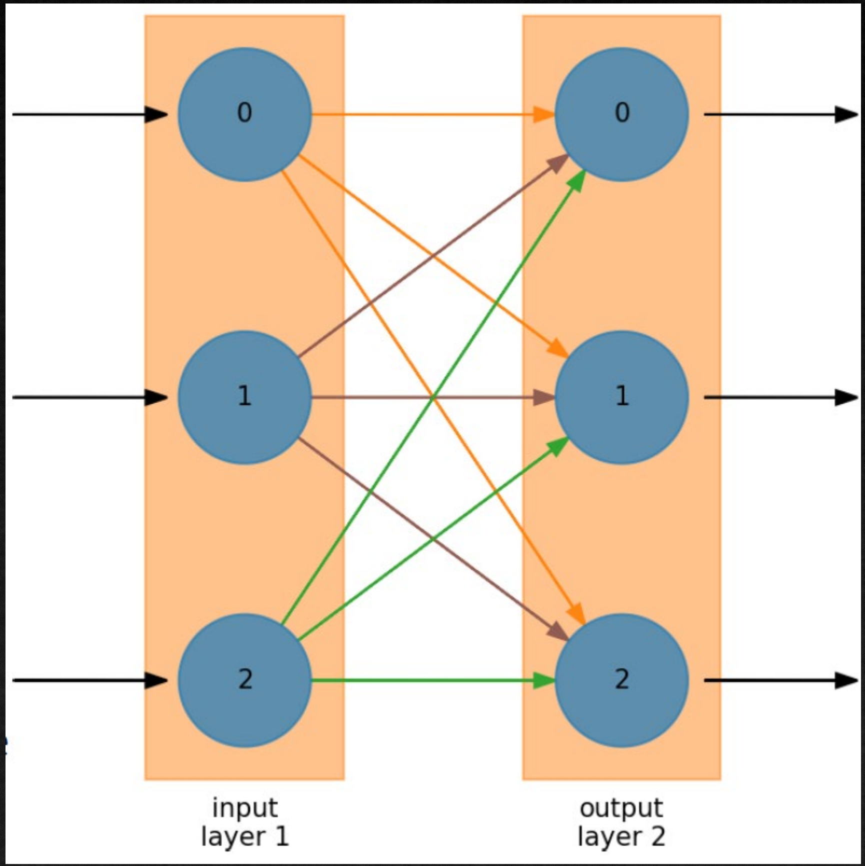
$$z^{(1)} = Ax^{(1)}$$
$$z^{(2)} = Ax^{(2)}$$
$$z^{(3)} = Ax^{(3)}$$

$$\mathbf{x}^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \mathbf{z}^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\mathbf{x}^{(2)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \mathbf{z}^{(2)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
$$\mathbf{x}^{(3)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \mathbf{z}^{(3)} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

- We cannot satisfy all three constraints with a linear model

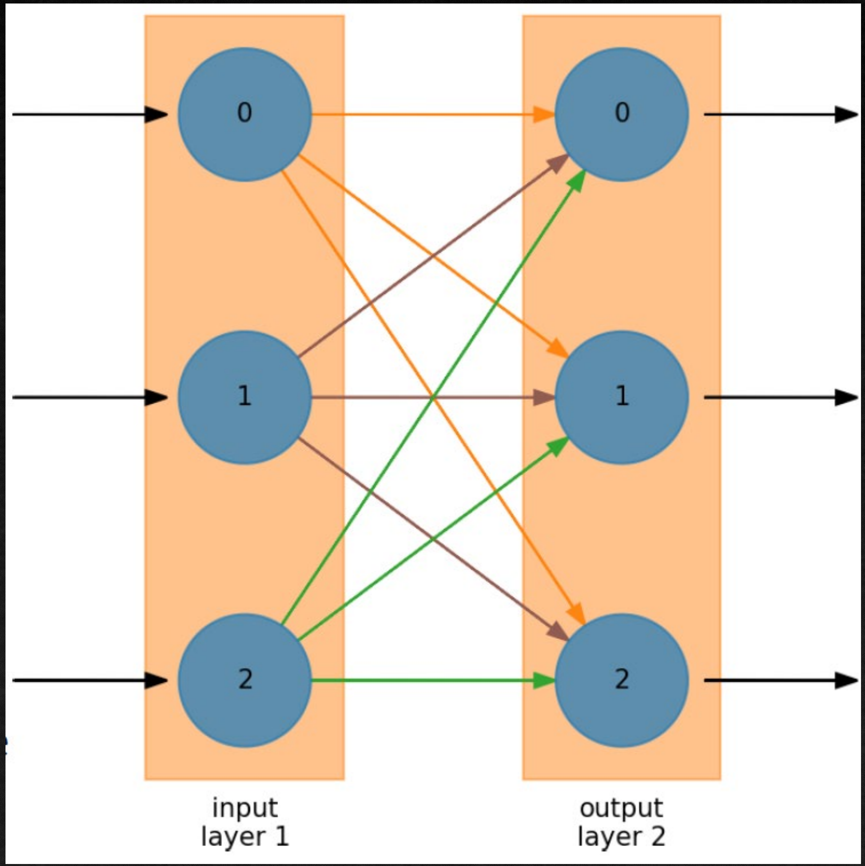
Neural network overview

- Neural networks are divided into layers
  - There's always an input layer – it doesn't do any processing – just accepts the input
  - There is always output layers
- Within a layer, there are neurons or nodes
  - For input, there will be one node for each input variable
- Every node in the first layer connects to every node in the next layer
  - The weight associated with the connection can vary – these are the matrix elements



Neural network overview

- In this example, the processing is done in layer 2
- When you train a neural network, you are adjusting the weights connecting the nodes
- Some connections may have zero weight
- This mimics nature – a single neurons can connect to several (or lots) of other neurons



Non-linear model

- We'll use a non-linear function g(p) that acts on a vector

$$g(\vec{x}) = \begin{pmatrix} g(x_0) \\ g(x_1) \\ \vdots \\ g(x_{n-1}) \end{pmatrix}$$

- Recall that z = g(Ax)
- For our previous example, g(p) = p^2 would fit all data
- New procedure: set the entries of **A** through training, using a simple non-linear function g(p) that fits our training data
- From the graphical representation, the non-linear function is applied on the output layer

Non-linear model

- Again, this mirrors the biology of neurons
- Neurons don’t act linearly
- There is a threshold that needs to be reached before a neuron “fires”

- A step function would work, but we want something differentiable

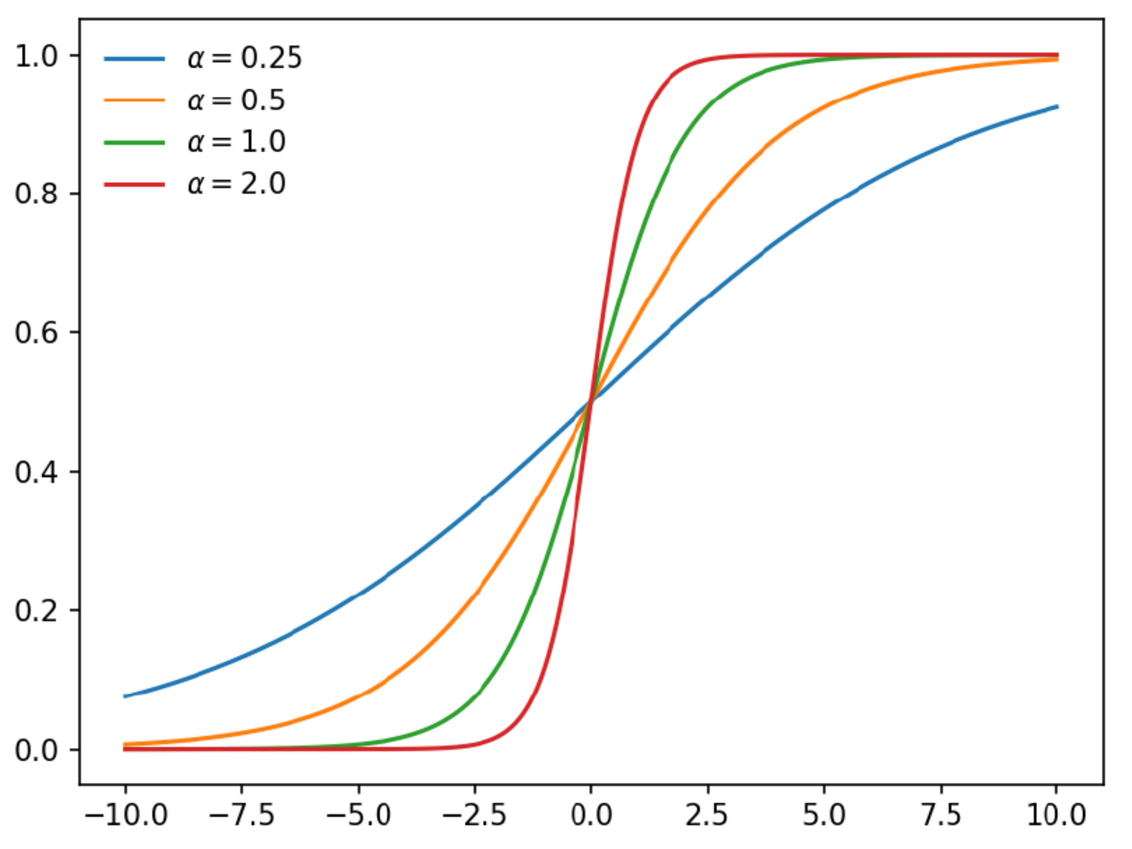
- There are a lot of different choices in the literature

Sigmoid function

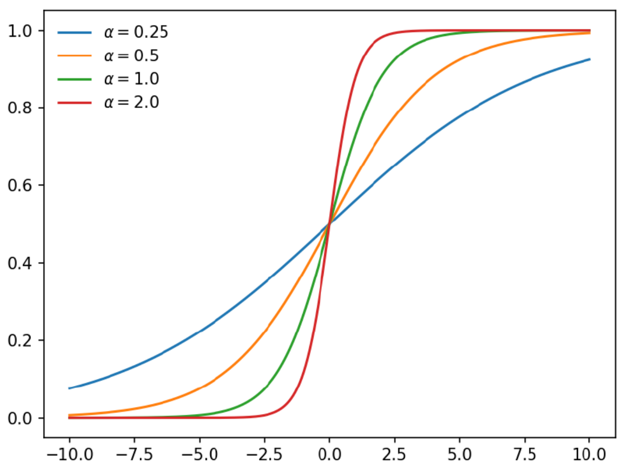
- Common choice: sigmoid function

$$g(p) = \frac{1}{1 + e^{-\alpha p}}$$

- Note: all outputs are scaled to be $z_j \in (0,1)$



Sigmoid function



- There are differing opinions on α
- Using $\alpha = 1$ seems to work well – this is what we'll do
  - Perhaps scale inputs to be in [0,1]
  - Note: inputs shouldn't be 0 because they cancel out weights
- There is a narrow range of non-linearity – pick α so that our inputs fall in that range
  - Elements of **A** are O(1)
  - p = Ax is O(n max{ |x| }
  - Choose:

$$\alpha = \frac{10}{n \max\{|x_i|\}}$$

Implementation

Basic operation

- Train the model with known input / output to get all A_{ij}
- Use z = g(Ax) to get output for a new input **x**

Training

- We have T pairs (x^k, y^k) for k = 1 … T
  - Important: remember that our y-values have to be scaled to (0,1), so they are in the same range that our function g(p) maps to
- We require that $g(Ax^k) = y^k$ for all k. Recall that g(p) is a scalar function

$$z_i = g([Ax]_i) = g\left(\sum_j A_{ij}x_j\right)$$

Implementation: Training

- We find the elements of **A**
- This can be expressed as a minimization problem, where we alter the matrix elements to achieve this agreement
- There may not be a unique set of A_{ij} so we will loop randomly over all training data multiple times to optimize **A**

$$f(A_{ij}) = \left|g(Ax^k) - y^k\right|^2$$

- Looks like a least-squares minimization
- The function we minimize is called the cost function
  - There are other choices than the square of the error

Implementation: Minimization

- A common technique for minimization is gradient descent (sometimes called steepest descent)
- This looks at the local derivative of the function f with respect to the parameters A_{ij} and moves a small distance *downhill* and iterates
- We can also utilize external libraries for minimization

Caveats

- When you minimize with one set of training data, there is no guarantee that you are still minimized with respect to the previous sets
- In practice, you feed the training data multiple times, in random order to the minimizer
- Each pass is called an epoch

Implementation: Minimization

- Gradient descent minimization
- Start at a point x_0 and evaluate the gradient
- Move *downhill* by following the gradient by some amount η
- Correct our initial guess and iterate

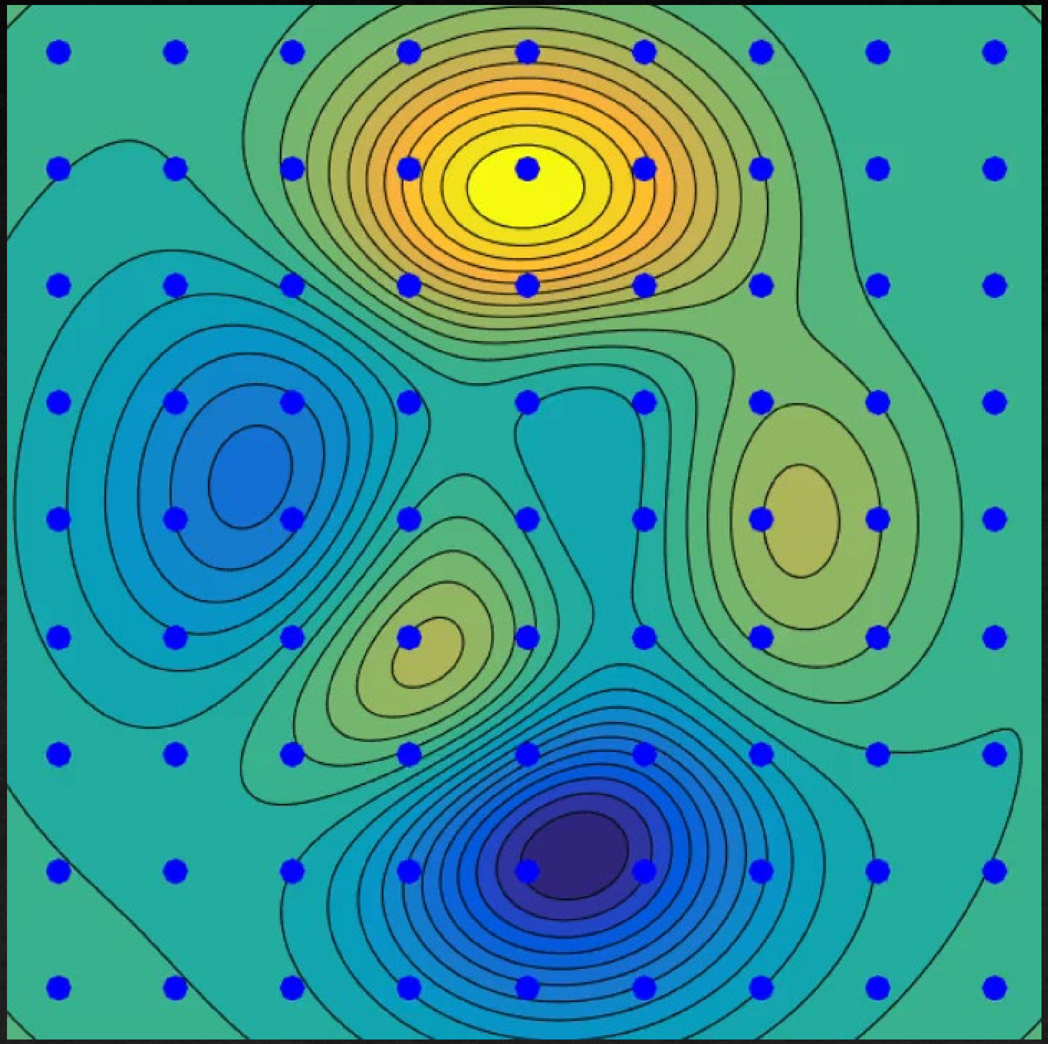
$$\vec{x} \leftarrow x - \eta \frac{\partial u}{\partial \vec{x}}$$

- Need to choose the amount to move each iteration
- Sometimes we instead define a unit vector in the direction of the local gradient, and then η represents the distance to travel in that direction
- Just think about a ball rolling on a surface. It rolls to a minimum, but this isn't guaranteed to be a global minimum

Implementation: Minimization

$$\vec{x} \leftarrow x - \eta \frac{\partial u}{\partial \vec{x}}$$

- Just think about a ball rolling on a surface. It rolls to a minimum, but this isn't guaranteed to be a global minimum

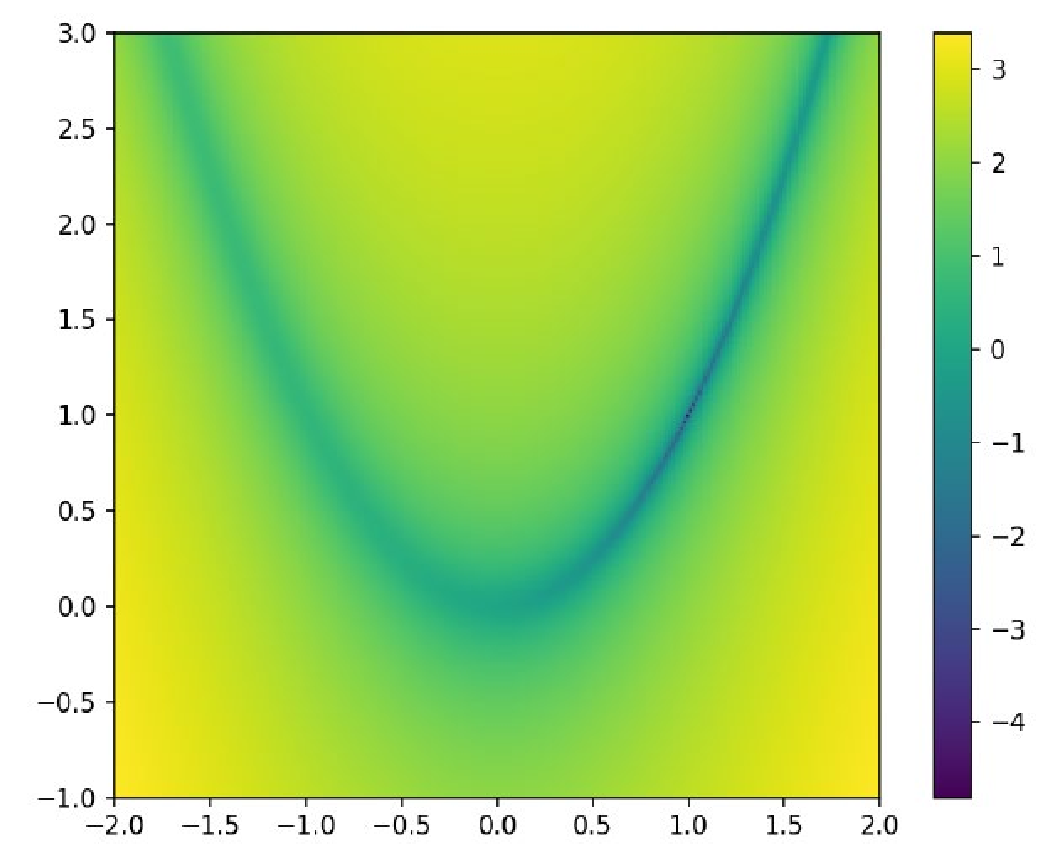


Implementation: Minimization

- Example: Rosenbrock function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

- To the right, $\log f$ is shown
- This is a hard problem for optimization
- The minimum exists at (a, a^2)

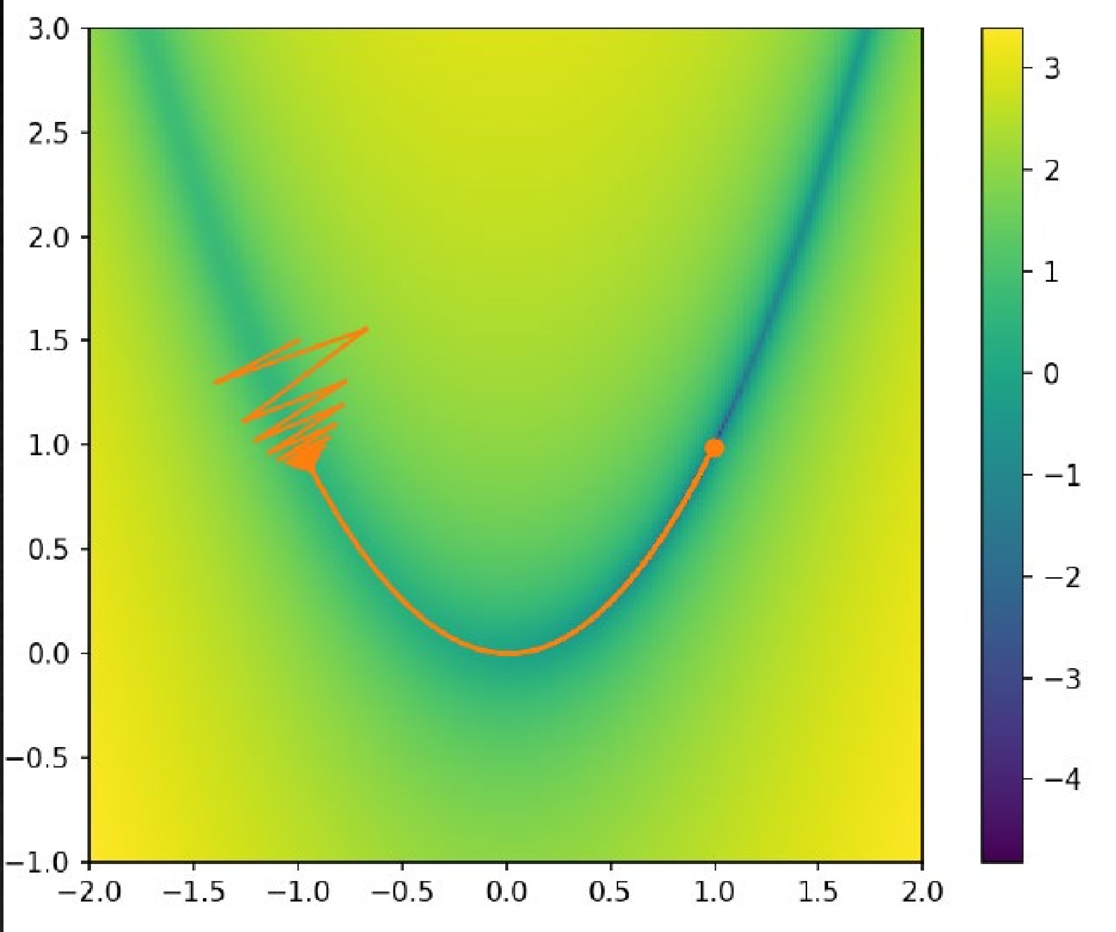


In-class problem: Minimization

- Minimimization with gradient descent is very sensitive to the choice of η
- Too large and you may shoot off far from the minimum
- Too small and you're subject to additional work
- See the code (18_gradient_descent0.py)

$$\vec{x} \leftarrow x - \eta \frac{\partial u}{\partial \vec{x}}$$

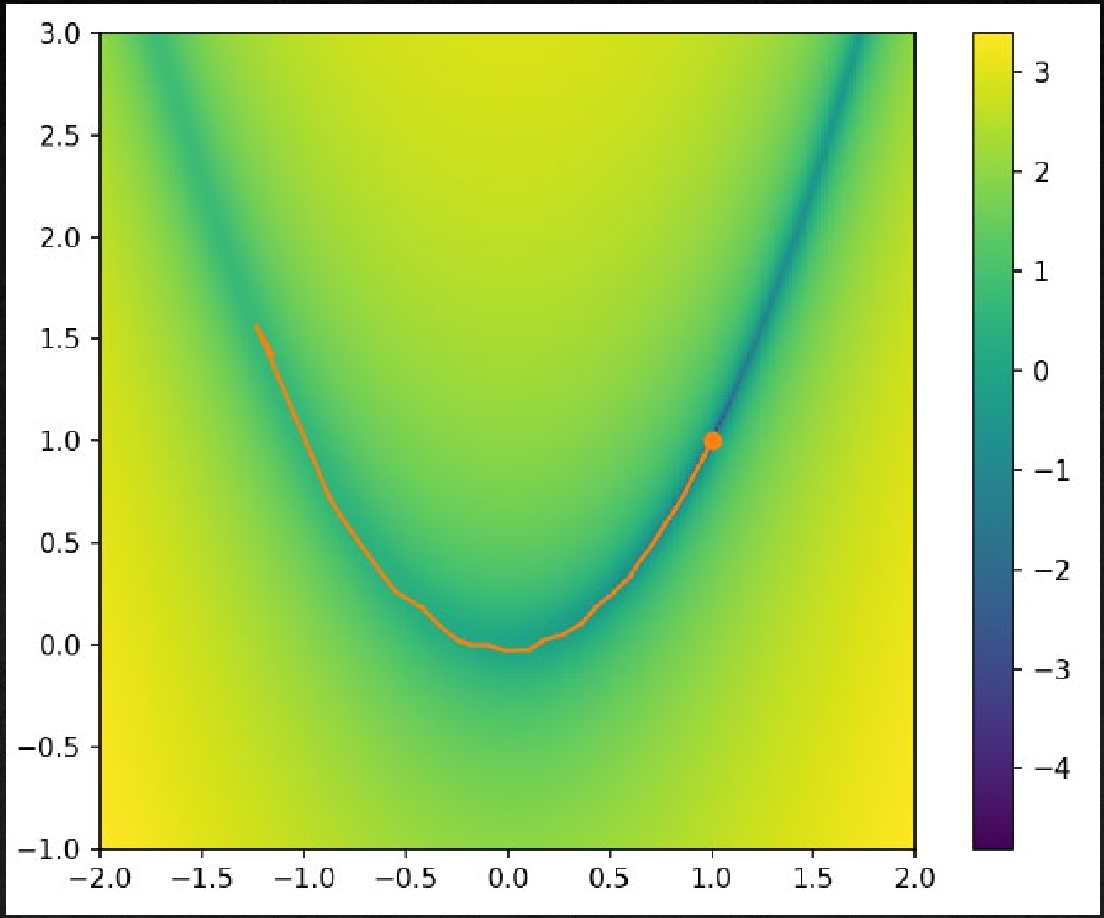
- Experiment with different starting points and learning rates (eta)



Implementation: Minimization

- Here is gradient descent using scipy's minimization function in the optimize module

- See the code (18_scipy_optimize.py)



Neural net minimization

- We are minimizing the function

$$f(A_{ij}) = \left|g(Ax^k) - y^k\right|^2$$

- This is the definition for a single training pair (x^k, y^k)

$$(x^k, y^k) = (\{x_1^k, x_2^k, \ldots, x_n^k\}, \{y_1^k, y_2^k, \ldots, y_n^k\})$$

- Our update would be

$$A_{pq} = A_{pq} - \eta \frac{\partial f}{\partial A_{pq}}$$

$$f(A_{ij}) = \sum_{i=1}^{m} \left[g\left(\sum_{j=1}^{n} A_{ij} x_j \right) - y_i \right]^2$$

$$A_{pq} = A_{pq} - \eta \frac{\partial f}{\partial A_{pq}}$$

$$f(A_{ij}) = \sum_{i=1}^{m} \left[g\left(\sum_{j=1}^{n} A_{ij} x_j \right) - y_i \right]^2$$

Neural net minimization

- We need its derivative

$$\frac{\partial f}{\partial A_{ij}} = 2(z_p - y_p)\alpha z_p(1 - z_p)x_q$$

- Recall *z = g(Ax)* is the output vector of the neural network and α comes from sigmoid
- We could use gradient descent, looping over the matrix elements and doing the minimization on them one-by-one, iterating until we converge
- Instead, we just do one push “downhill” following the gradient for a single training set and then move to the next
- η is often called the learning rate
- Gradient descent is often used for NNs because it only requires the 1st derivative
- Newton’s method requires the 2nd derivative (Hessian matrix)

Neural net minimization

- **A** is a (m x n) matrix
- **x** is a (n x 1) vector
- **y** (and hence **z**) is a (m x 1) vector
- We can write our derivative, using element-wise products (i.e. $a \circ b$)

$$\frac{\partial f}{\partial \vec{A}} = 2(\vec{z} - \vec{y}) \circ \alpha \vec{z} \circ (1 - \vec{z}) \cdot \vec{x}^T$$

- Then the correction to our matrix is

$$\Delta A = -2\eta(\vec{z} - \vec{y}) \circ \alpha \vec{z} \circ (1 - \vec{z}) \cdot \vec{x}^T$$

$$A \leftarrow A + \Delta A$$

Neural net minimization: initialization

- A common choice for initializing **A** is to set the elements to random numbers in [-1, 1]
- Some suggest that a better choice is a set of Gaussian random numbers with width $\mu = n^{-1/2}$ and $\alpha = 1$
- The initialization sets the starting point in the minimization, so different realizations can convent to different (local) minima