What is Learning?

What is Learning?

- There are lots of answers to this question, and explanations often become philosophical
- A more practical question might be:

What can we teach/train a person, animal, or machine to do?

Example: Addition “+”

- How is addition taught in schools?
  - Memorize rules for pairs of numbers from the set {0,1,2,3,4,5,6,7,8,9}
  - Memorize redundant rules collectively, for efficiency, e.g. $0+x=x$
  - Learn to treat powers of 10 implicitly, e.g. $12+34=46$ since $1+3=4$ and $2+4=6$
  - Learn to carry when the sum of two numbers is larger than 9
  - Learn to add larger sets of numbers by considering them one pair at a time
  - Learn how to treat negative numbers
  - Learn how to treat decimals and fractions
  - Learn how to treat irrational numbers

Knowledge Based Systems (KBS)

Contains two parts:

1) Knowledge Base
    - Explicit knowledge or facts
    - Often populated by an expert (expert systems)
2) Inference Engine
    - Way of reasoning about the facts in order to generate new facts
    - Typically follows the rules of Mathematical Logic

KBS Approach to Addition

- Rule: x and y commute
- Start with x and y as single digits, and record all $x + y$ outcomes as facts
- Add rules to deal with muti-digit numbers by pulling out powers of 10
- Add rules for negative numbers, decimals, fractions, irrationals, etc.

- Mimics human learning (or at least human teaching)
- This is a discrete approach, and it has no inherent error

Machine Learning (ML)

Contains two parts:

1) Training Data
   - Data Points - typically as domain/range pairs
   - Hand labeled by a user, measured from the environment, generated procedurally, etc.
2) Model
   - Derived from the Training Data in order to estimate new data points with (hopefully) minimal error
   - Uses Algorithms, Statistical Reasoning, Rules, Networks, Etc.

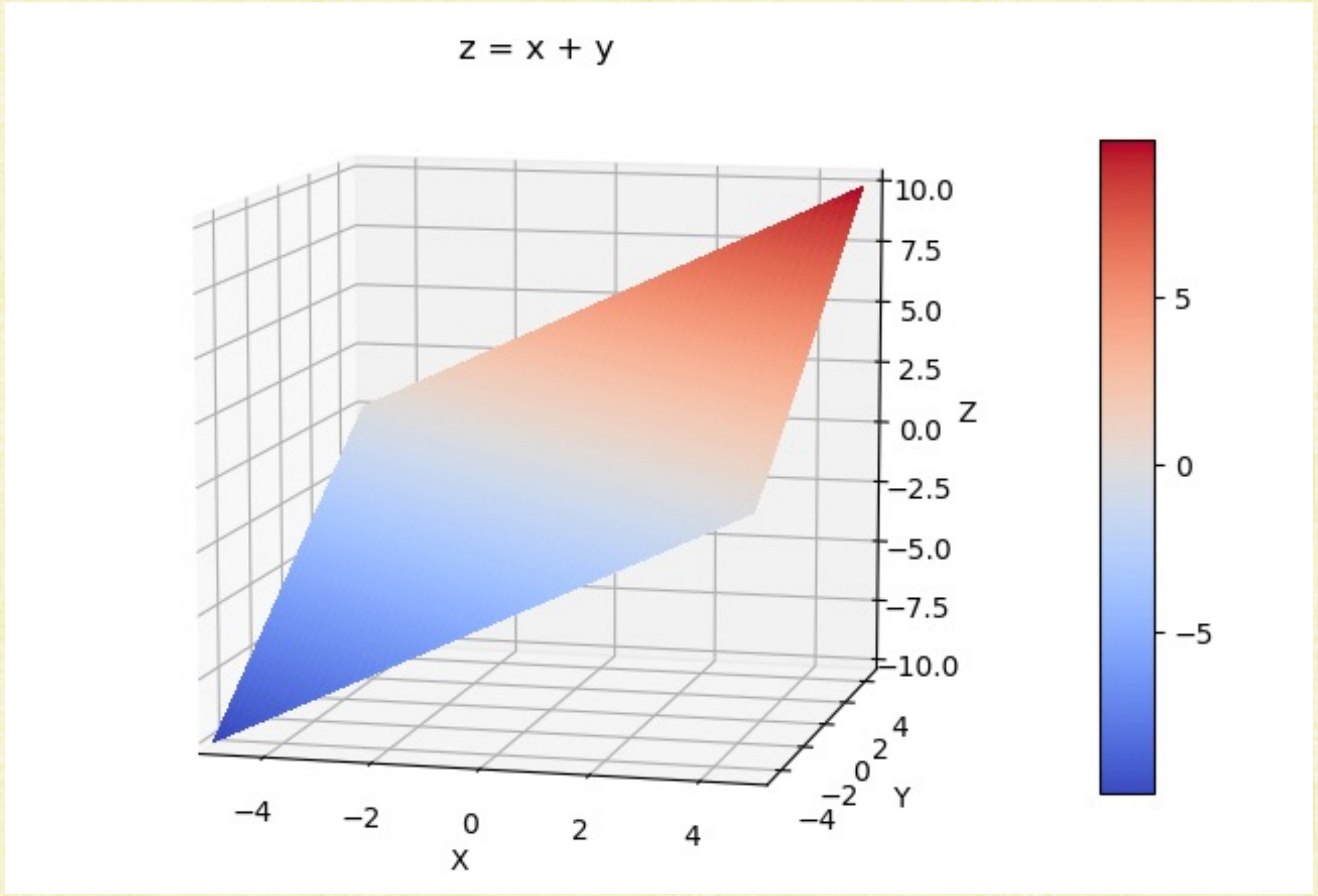
KBS vs. ML

- KBS and ML can be seen as the <u>discrete math</u> and <u>continuous math</u> approaches (respectively) to the same problem
- KBS’s Knowledge Base serves the same role as ML’s Training Data
- Logic is the algorithm used to discover new discrete facts for KBS, whereas many numerical algorithms/methods are used to approximate continuous facts/data for ML
    - Logic (in particular) happens to be especially useful for discrete facts
    - Numercial algorithms are especially usefully for continuous approximations
- <u>ML</u>, derived from continuous math, <u>will tend to have inherent approximation errors</u>

ML Approach to Addition

- Make a $2D$ domain in R^2, and a $1D$ range in R^1 for the addition function
- As training data, choose a number of input points (x_i , y_i) with output $x_i + y_i$
- Plot the 3D points $(x_i , y_i , x_i + y_i)$ and determine a <u>model function</u> $z = f(x, y)$ that best approximates the training data
- Turns out that the plane $z = x + y$ exactly fits the training data
    - Only need 3 training points to determine this plane
- Don’t need special rules for negative numbers, decimals, fractions, irrationals such as $\sqrt{2}$ and π, etc.
- However, small errors in the training data lead to a slightly incorrect plane, which has quite large errors far away from the training data
- This can be alleviated to some degree by adding training data where one wants smaller errors (and computing the best fitting plane to all the training data)

ML Approach to Addition

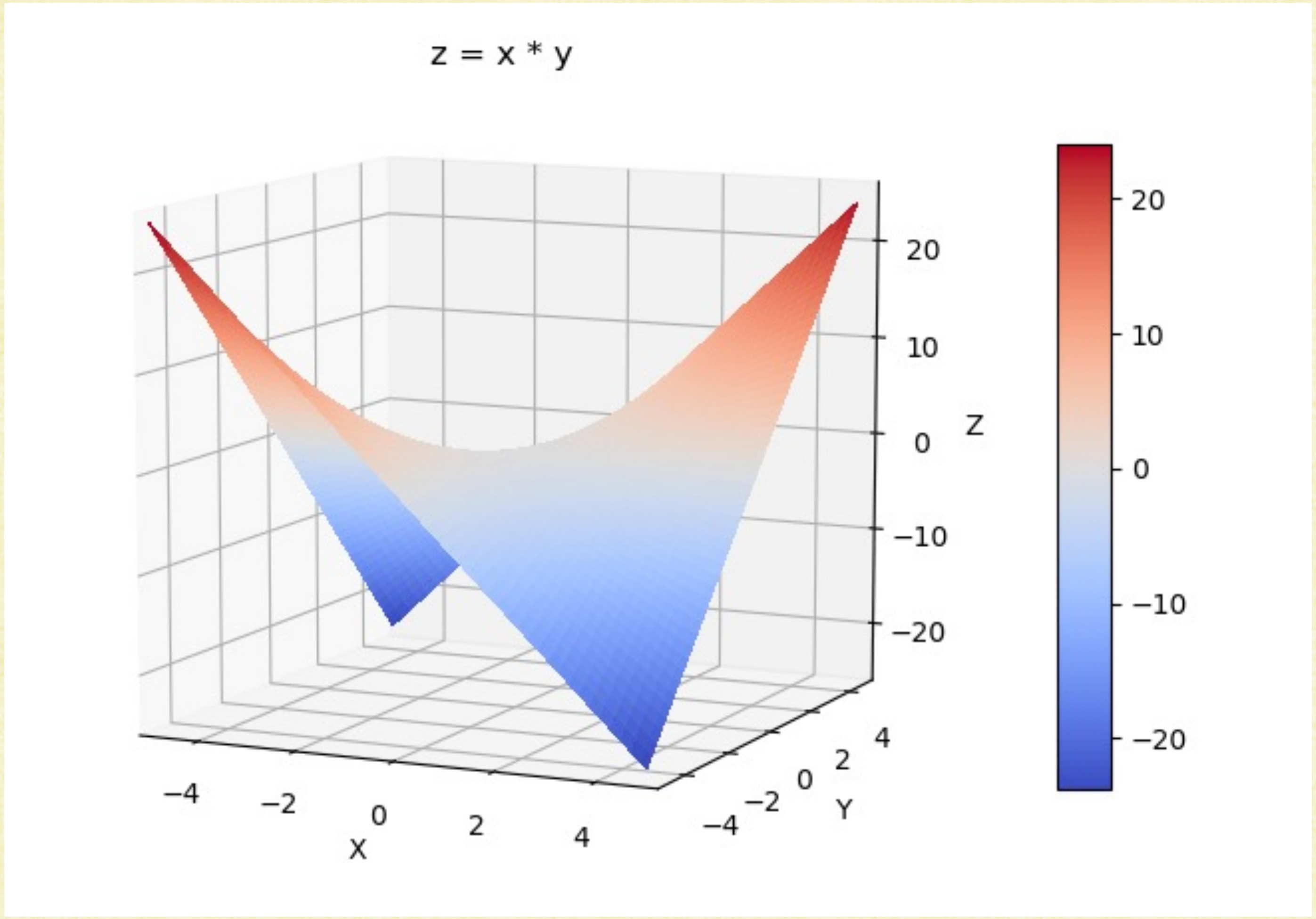


Example: Multiplication “$*$”

- KBS creates new rules for $x * y$, utilizing the rules from addition too

- ML utilizes a set of 3D points $(x_i\ , y_i\ , x_i * y_i)$ as training data, and the model function $z = x * y$ can be found to exactly fit the training data
    - However, one may claim that it is “cheating” to use an inherently represented floating point operation (i.e., multiplication) as the model

ML Approach to Multiplication



Example: Unknown Operation “#”

- KBS fails!
- How can KBS create rules for $x\#y$ when we don’t even know what # means?
- This is the case for many real-world phenomena that are not fully understood
- However, sometimes it is possible to get some examples of $x\#y$
- That is, through experimentation or expert knowledge, one might be able to discover $z_i = x_i\#y_i$ for some number of pairs $(x_i\ , y_i)$
- Subsequently, these known (or estimated) $3D$ points $(x_i\ , y_i\ , z_i)$ can be used as training data to determine a model function $z = f(x, y)$ that approximately fits the data

Determining the Model Function

- How does one determine $z = f(x, y)$ near the training data, so that it robustly predicts/infers $\hat{z}$ for new inputs $(\hat{x}, \hat{y})$ not contained in the training data?
- How does one minimize the effect of inaccuracies or noise in the training data?

- Caution: away from the training data, the model function f is likely to be highly inaccurate (extrapolation is ill-posed)

Nearest Neighbor

- If asked to multiply 51.023 times 298.5, one might quickly estimate that 50 times 300 is $15{,}000$
- This is a nearest neighbor algorithm, relying on nearby data where the answer is known, better known, or more easy to come by
- Given $(\hat{x}\,, \hat{y})$, find the closest (Euclidean distance) training data (x_i, y_i) and return its associated z_i (with error $\|z_i - \hat{z}\|$)
- This represents $z = f(x, y)$ as a piecewise constant function with discontinuities on the boundaries of Voronoi regions around the training data
- This is the <u>simplest possible Machine Learning algorithm</u> (a piecewise constant function), and it works in an arbitrary number of dimensions

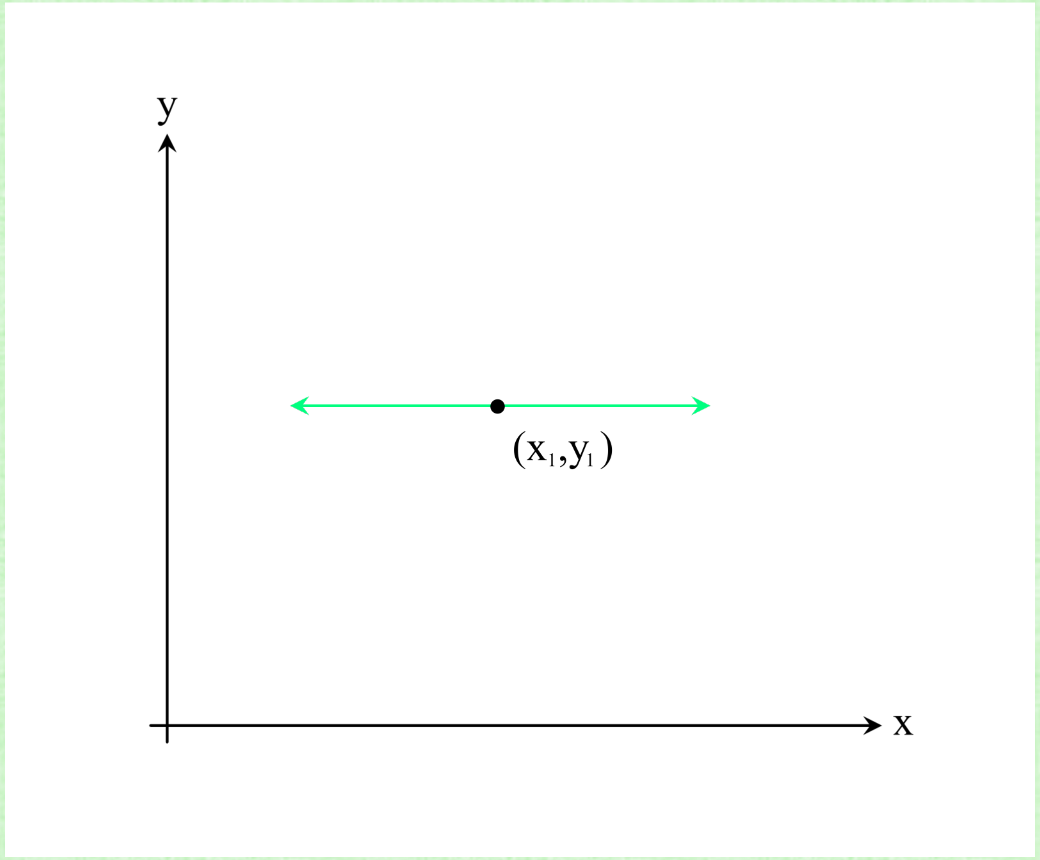
Data Interpolation

- In order to elucidate various concepts, let's consider data interpolation in more detail

- Let's begin with a very simple case with $1D$ inputs and $1D$ outputs, i.e. $y = f(x)$

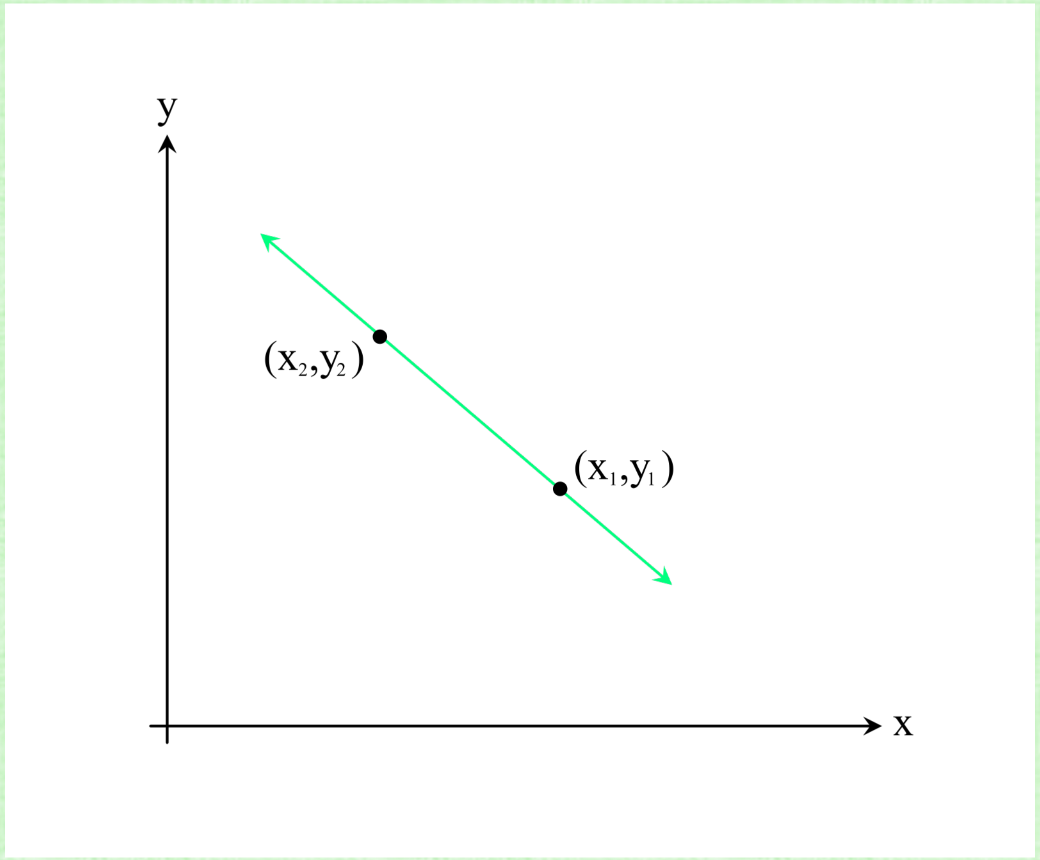
Polynomial Interpolation

- Given 1 data point, one can (at best) draw a constant function



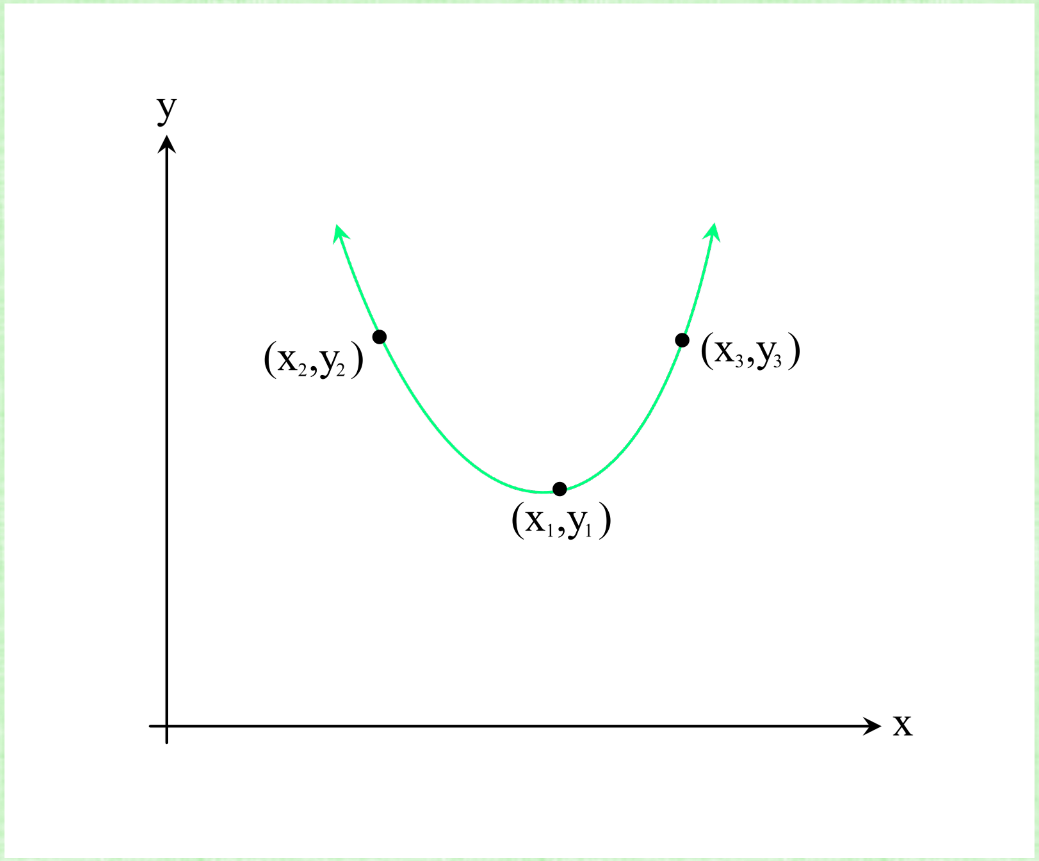
Polynomial Interpolation

- Given 2 data points, one can (at best) draw a linear function



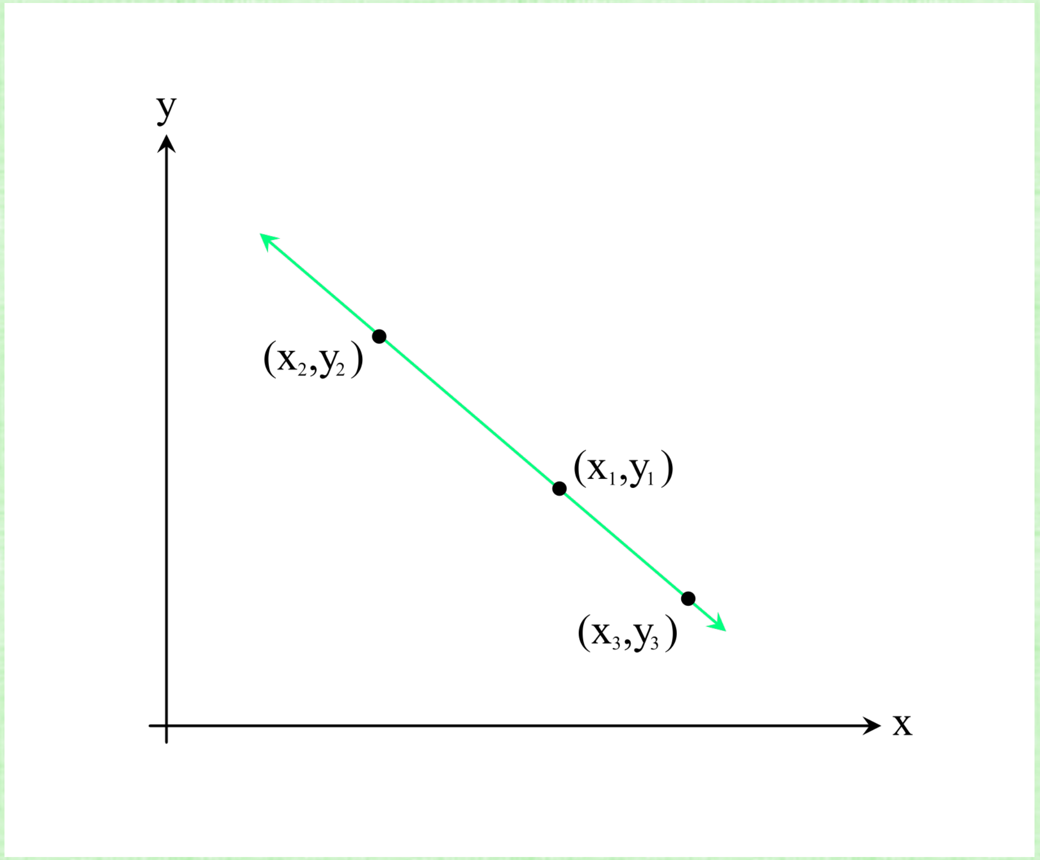
Polynomial Interpolation

- Given 3 data points, one can (at best) draw a quadratic function



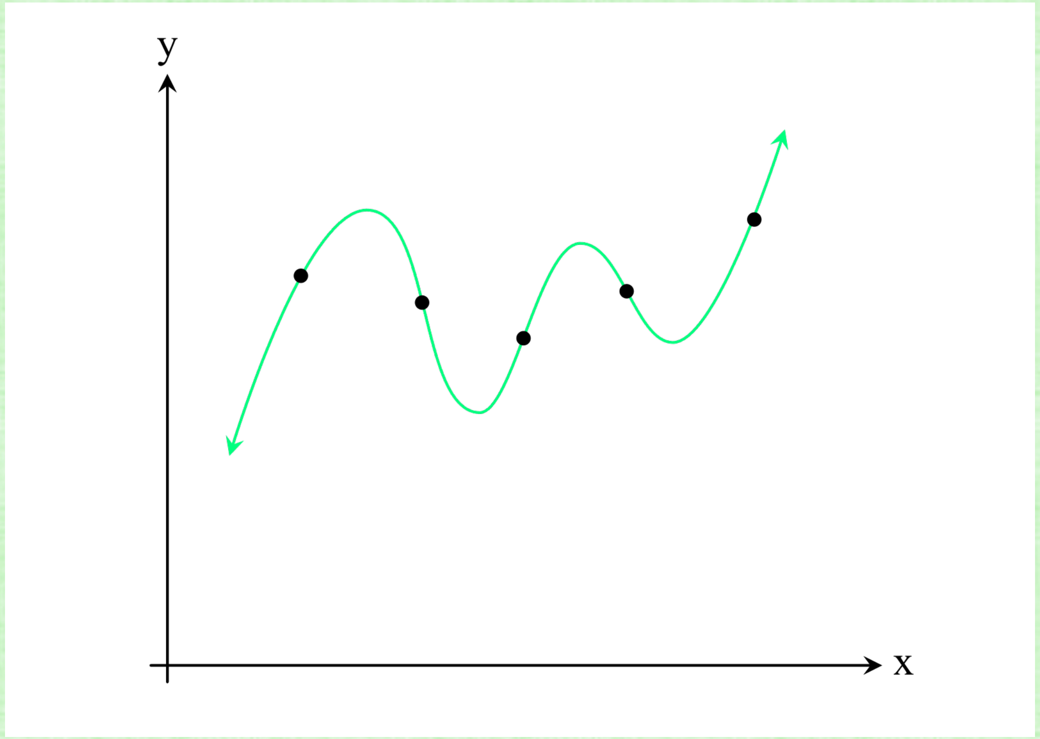
Polynomial Interpolation

- Unless all 3 points are on the same line, in which case one can only draw a linear function



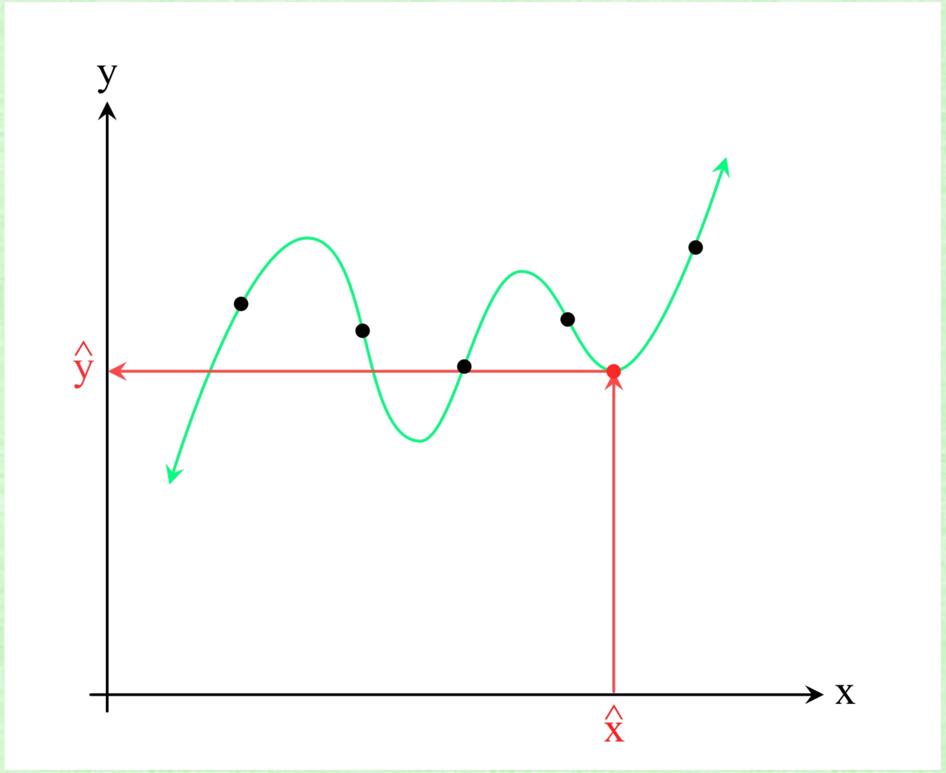
Polynomial Interpolation

- Given m data points, one can (at best) draw a unique $m - 1$ degree polynomial that goes through all of them
    - As long as they are not degenerate, like 3 points on a line



Overfitting

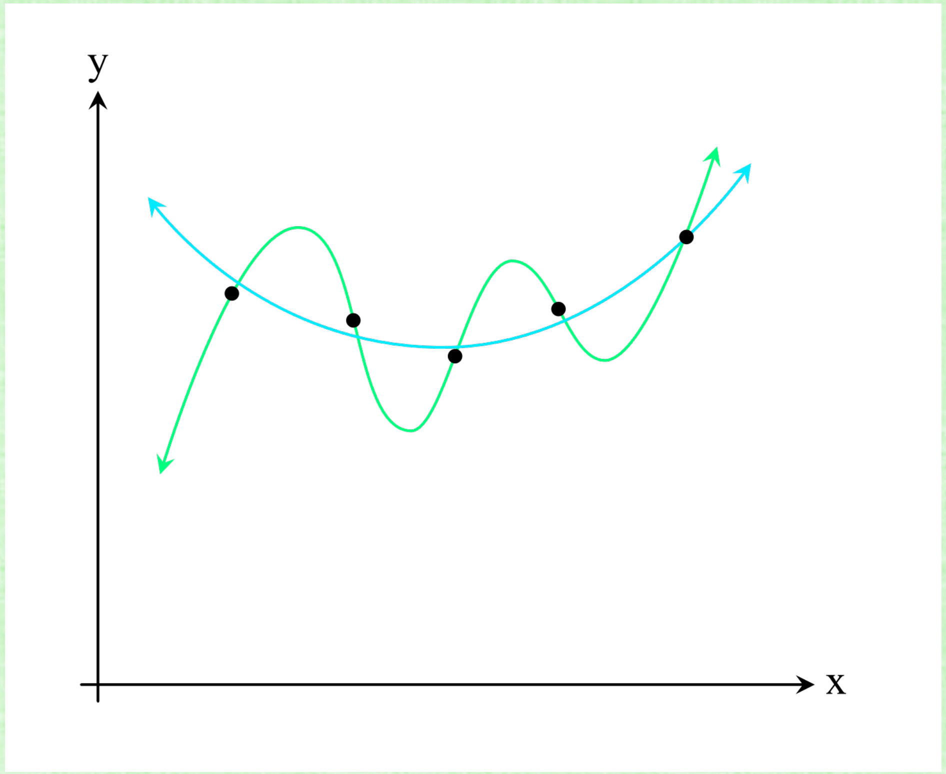
- Given a new input $\hat{x}$, the interpolating polynomial infers/predicts an output $\hat{y}$ that may be far from what one may expect



- Interpolating polynomials are smooth (continuous function and derivatives)
- Thus, they wiggle/overshoot in between data points (so that they can smoothly turn back and hit the next point)
- Overly forcing polynomials to exactly hit every data point is called overfitting (overly fitting to the data)
- It results in inference/predictions that can vary wildly from the training data

Regularization

- Using a lower order polynomial that doesn’t (can’t) exactly fit the data points provides some degree of regularization

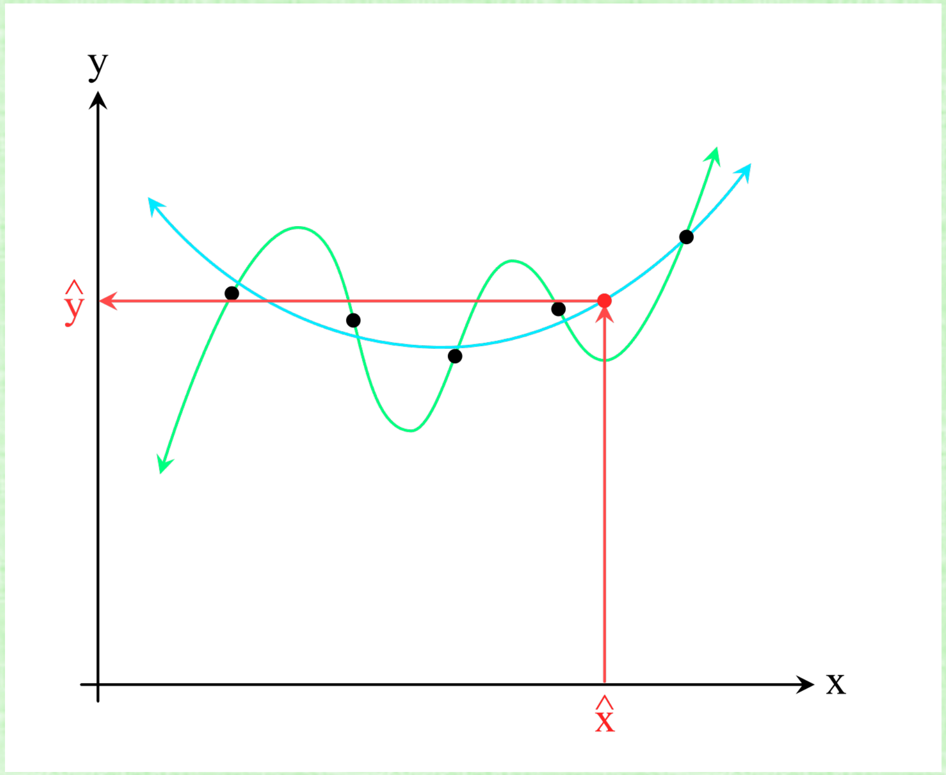


- A regularized interpolant contains <u>intentional errors</u> in the interpolation, missing some/all of the data points
- However, this hopefully makes the function <u>more predictable/smooth</u> in between the data points

- The data points themselves may contain noise/error, so it is not clear whether they should be interpolated exactly anyways

Regularization

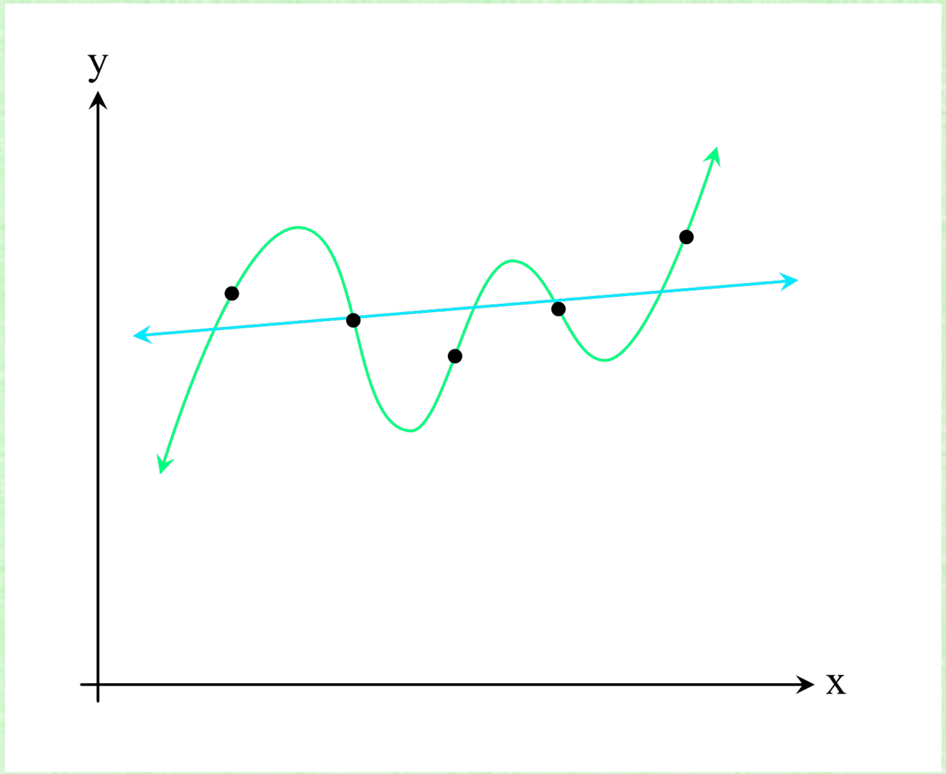
- Given $\hat{x}$, the regularized interpolant infers/predicts a more reasonable $\hat{y}$



- There is a trade-off between sacrificing accuracy on fitting the original input data, and obtaining better accuracy on inference/prediction for new inputs

Underfitting

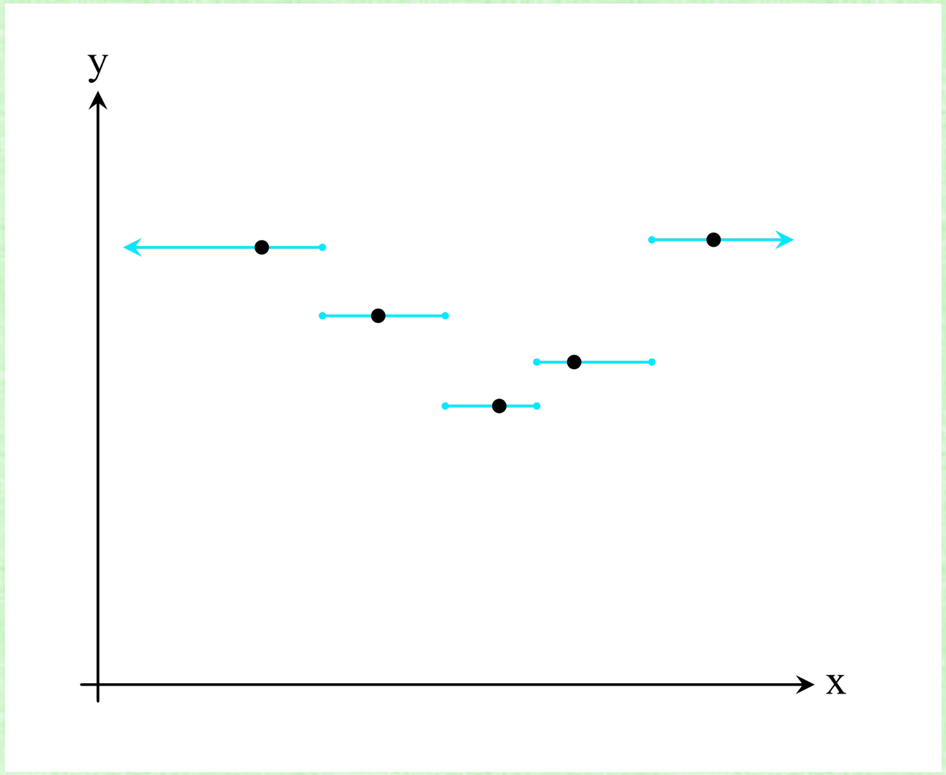
- Using <u>too low</u> of an order polynomial causes it to miss the data by too much



- A linear function doesn't capture the essence of this data as well as a quadratic function does
- Choosing too simple of a model function or regularizing too much prevents one from properly representing the data

Nearest Neighbor

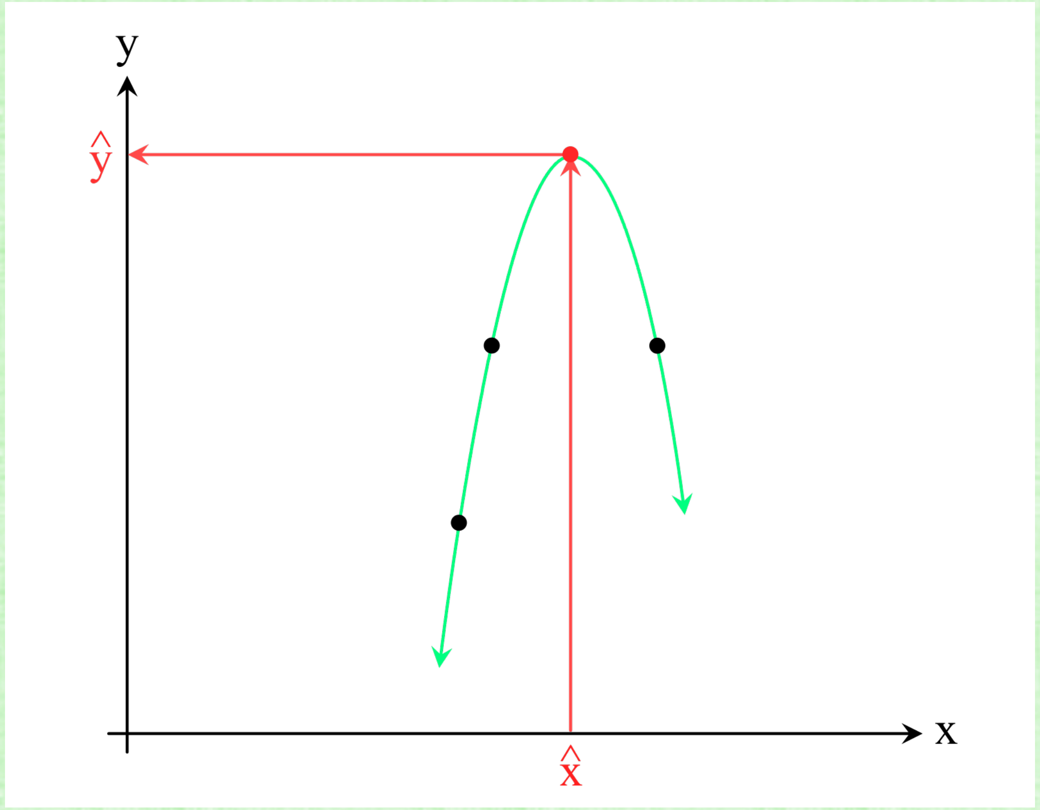
- Piecewise-constant interpolation on this data (equivalent to nearest neighbor)



- The reasonable behavior of the piecewise constant (nearest neighbor) function stresses the importance of approximating data locally
- We address Local Approximations in Unit 6

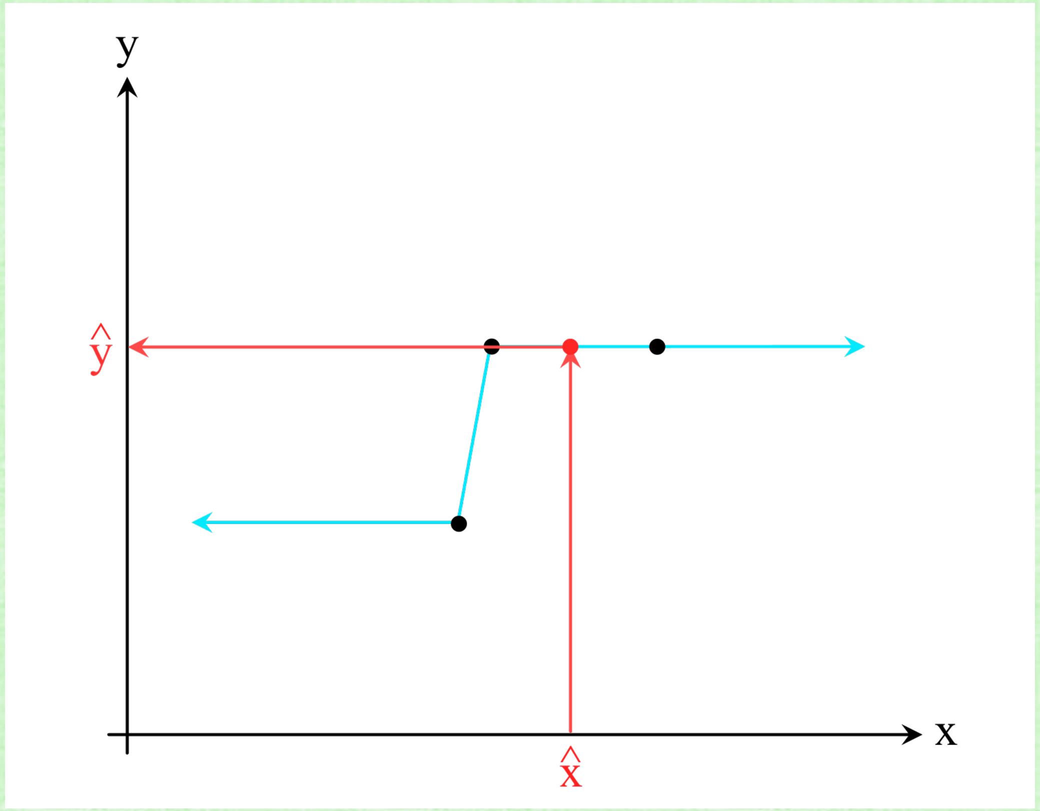
Caution: Overfitting

- Higher order polynomials tend to oscillate wildly, but even a simple quadratic polynomial can overfit by quite a bit



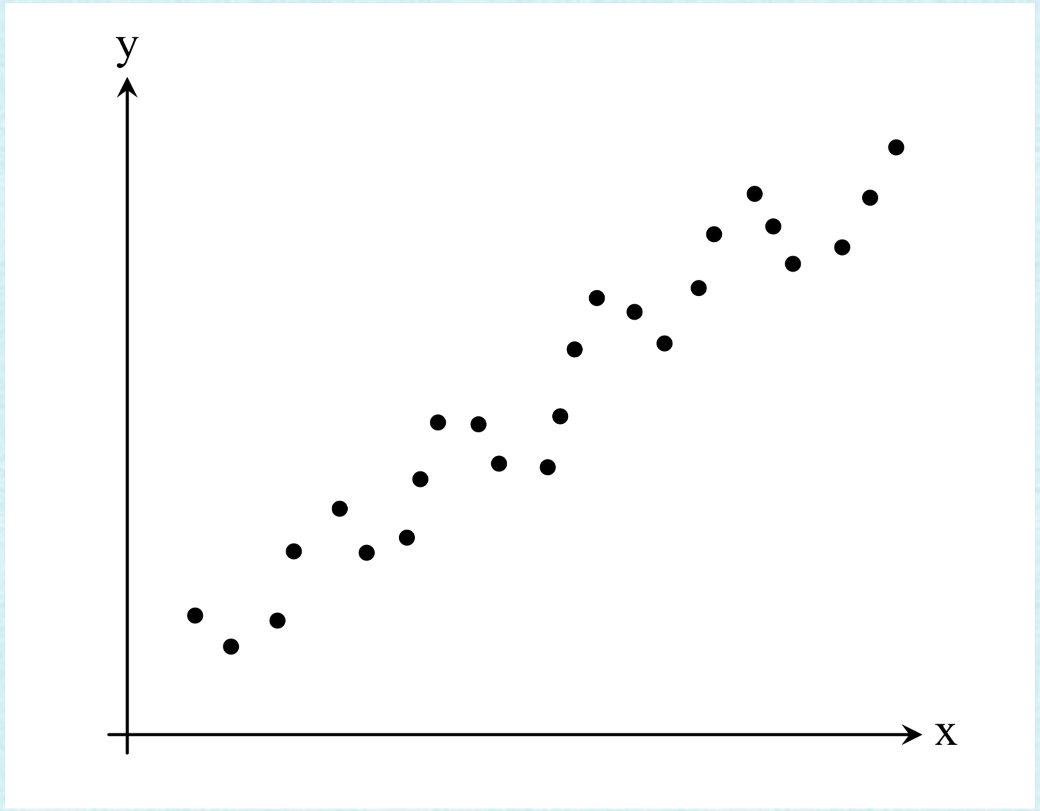
Caution: Overfitting

- A <u>piecewise</u> linear approach works much better on this data



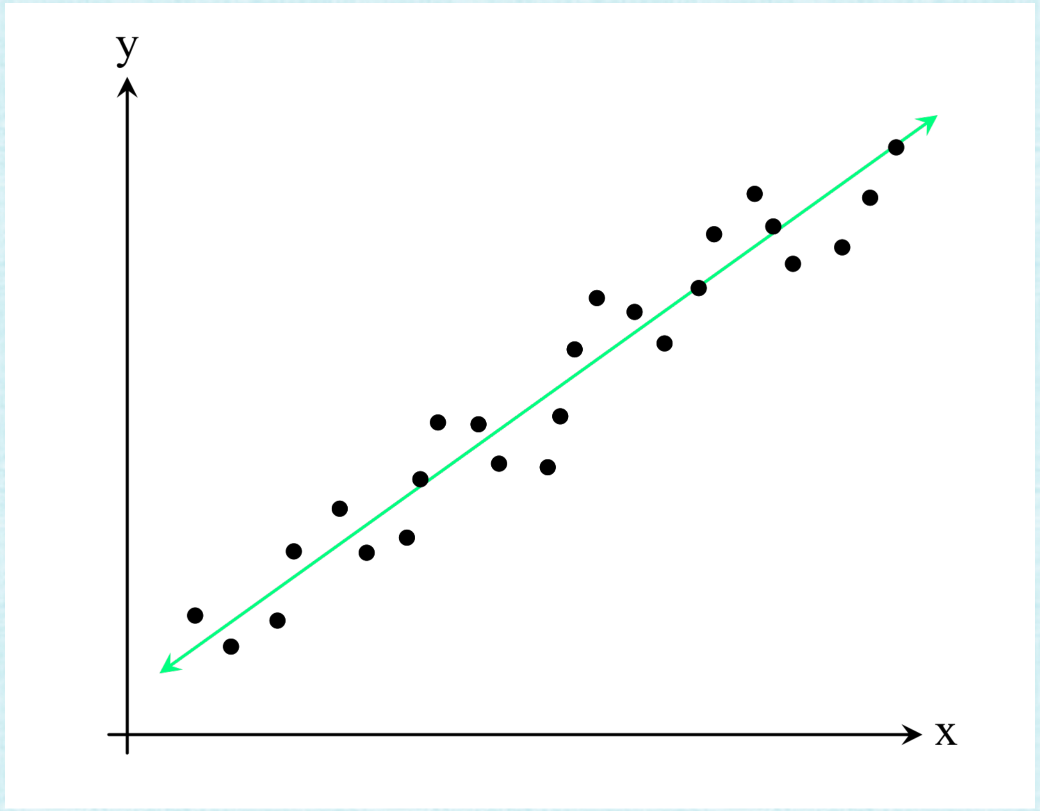
Noisy Data

- There may be many sources of error in data, so it can be unwise to attempt to fit data too closely



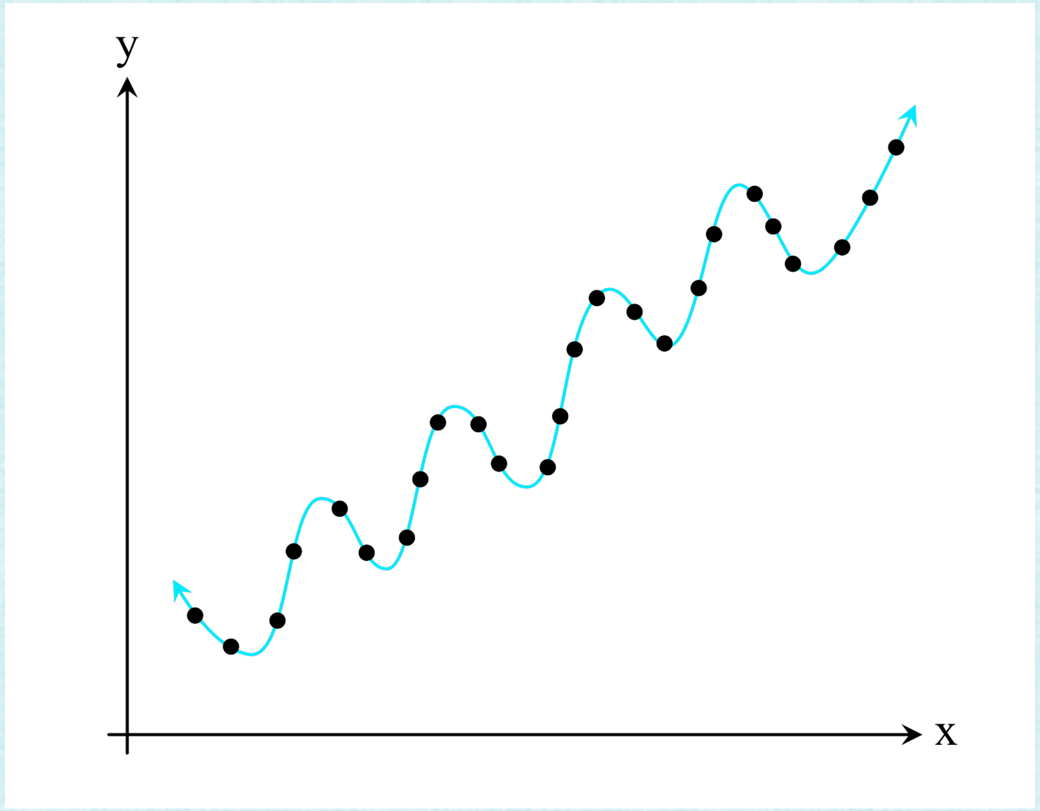
Linear Regression

- One commonly fits a low order model to such data, while minimizing some metric of mis-interpolating or mis-representing the data



Noise vs. Features

- But how does one differentiate between noise and features?



Noise vs. Features

- When training a neural network, split the available data into 3 sets
- E.g., 80% training data, 10% model validation data, and 10% test data
- Training data is used to train the neural network
  - An interpolating function is fit to the training data (potentially overfitting it)
- When considering features vs. noise, overfitting, etc., model validation data is used to select the best model function or the best fitting strategy
  - Compare inference/prediction on model validation data to the known answers
- Finally, when disseminating results advocating the “best” validated model, inferencing on the test data gives some idea as to how well that validated model might generalize to unseen data
  - Competitions on unseen data have become a good way to stop “cheating” on test data

Errors in Equations

- **Modeling errors** – Parts of a problem under consideration might be ignored. E.g., when simulating solids/fluids, sometimes frictional/viscous effects are not included.
- **Empirical constants** – Some numbers are unknown, and measured with limited precision. Others may be known more accurately, but limited precision hinders the ability to express them. E.g. Avogadro's number, the speed of light in a vacuum, the charge on an electron, Planck's constant, Boltzmann's constant, pi, etc. (Note that the speed of light is 299792458 m/s exactly, so ok for double precision but not for single precision.)

Errors in Numerical Methods

- **<u>Rounding errors</u>:** Even integer calculations lead to floating point numbers, e.g. 5/2=2.5, and floating point calculations frequently admit rounding errors, e.g. 1./3.=.3333333… cannot be expressed on the computer. Machine precision is 10^{-7} for single precision and 10^{-16} for double precision.
- **<u>Truncation errors</u>** – Also called discretization errors. These occur in the mathematical approximation of an equation as opposed to an approximation of the physics (modeling errors). E.g. one (often) cannot take a derivative/integral exactly on a computer, and instead approximates them (recall Simpson’s rule from Calculus).

Errors in Inputs

- **Inaccurate inputs** – Often, one is only concerned with part of a calculation, where a given set of inputs is used to produce outputs. Those inputs may have previously been subjected to any of the errors listed above, and thus may already have limited accuracy. This has implications for various algorithms. E.g., if inputs are only accurate to 4 decimal places, it probably doesn’t make sense to carry out an algorithm to an accuracy of 8 decimal places.
- **Inaccurate Measurements** – It can be difficult to accurately measure real-world phenomena, generating another source of inaccurate inputs.

A Robust Computational Approach

- **Well Posedness:** A problem is ill-posed if small changes in the inputs lead to large changes in the outputs. Any source of error dominates the result.
- **Condition Number:** An algorithm is ill-conditioned if small changes in the inputs lead to large changes in the outputs. Large condition numbers are bad (sensitive), and small condition numbers are good (insensitive). If the relative changes in the inputs and outputs are identical, the condition number is 1.
- **Stability:** An algorithm is stable if it can complete itself in any meaningful way. Unstable algorithms give wild (explosive) results, usually leading to NaN's.
- **Accuracy:** Accuracy refers to the size of the error, or how close the answer is to the correct solution.

A Robust Computational Approach

- A problem should be <u>well-posed</u> before even considering it computationally
- Computational Approach:
  - 1) <u>Conditioning</u> - formulate a well-conditioned approach, or as well-conditioned as is possible
  - 2) <u>Stability</u> - devise a stable algorithm; otherwise, the result is typically NaNs
  - 3) <u>Accuracy</u> – even a well-conditioned and stable approach can result in large errors; so, make the algorithm as accurate as is warranted or practical

Being Careful: Vector Norms

- Consider the norm of a vector: $\|x\|_2 = \sqrt{x_1^2 + \cdots + x_m^2}$
- Straightforward algorithm:

```
for (i=1,m) sum+=x(i)*x(i); return sqrt(sum);
```

- This can overflow MAX_FLOAT/MAX_DOUBLE for large m
- Safer algorithm:

```
find z=max(abs(x(i)))
for (i=1,m) sum+=sqr(x(i)/z); return z*sqrt(sum);
```

Being Careful: Quadratic Formula

- Consider $.0501x^2 - 98.78x + 5.015 = 0$
  - To 10 digits of accuracy: $x \approx 1971.605916$ and $x \approx .05077069387$
- Using 4 digits of accuracy in the quadratic formula gives:

$$\frac{98.78+98.77}{.1002} = 1972 \quad \text{and} \quad \frac{98.78-98.77}{.1002} = .0998$$

  - The second root is completely wrong (in the leading significant digit!)
- De-rationalize: $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ to $\frac{2c}{-b\mp\sqrt{b^2-4ac}}$
- Using 4 digits of accuracy in this de-rationalized quadratic formula gives:

$$\frac{10.03}{98.78-98.77} = 1003 \quad \text{and} \quad \frac{10.03}{98.78+98.77} = .05077$$

  - Now the second root is fine, but the first root is wrong!
- Conclusion: use one formula for each root

Being Careful: L’Hopitals Rule

- Consider $\frac{x^2-4}{x-2}$ near $x = 2$ where it becomes $\frac{0}{0}$
- Adding a small number to the denominator $\frac{x^2-4}{x-2+\epsilon}$ incorrectly gives 0 near $x = 2$
- Noting that $\frac{x^2-4}{x-2} = x + 2$ leads to correct values near 4 when x is near 2
- Similar issues occur for $\frac{\sin x}{x}$ near $x = 0$
- Similar issues occur for 0 times ∞ and other cases where L’Hopitals rule is needed to address removable singularities

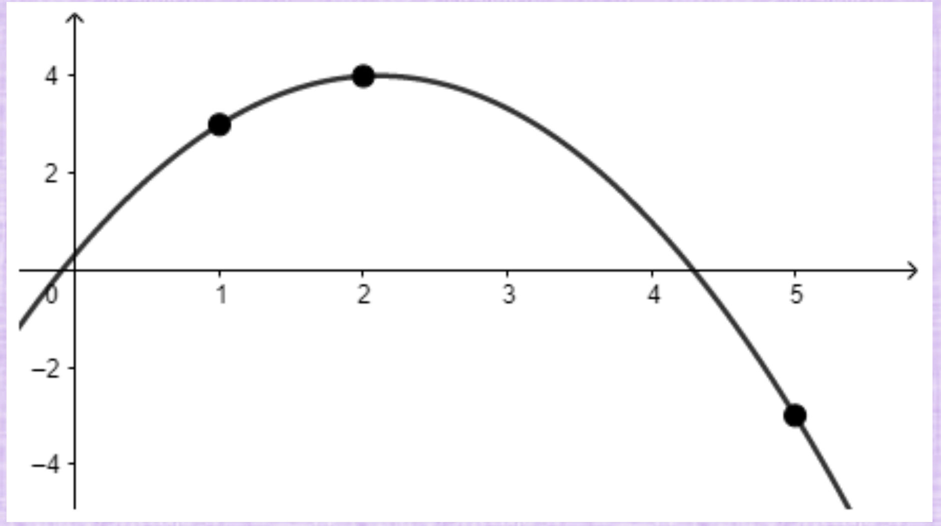
Did you know about these issues?

- Imagine debugging code with the correct quadratic formula implementation and getting zero digits of accuracy on a test case!

Polynomial Interpolation

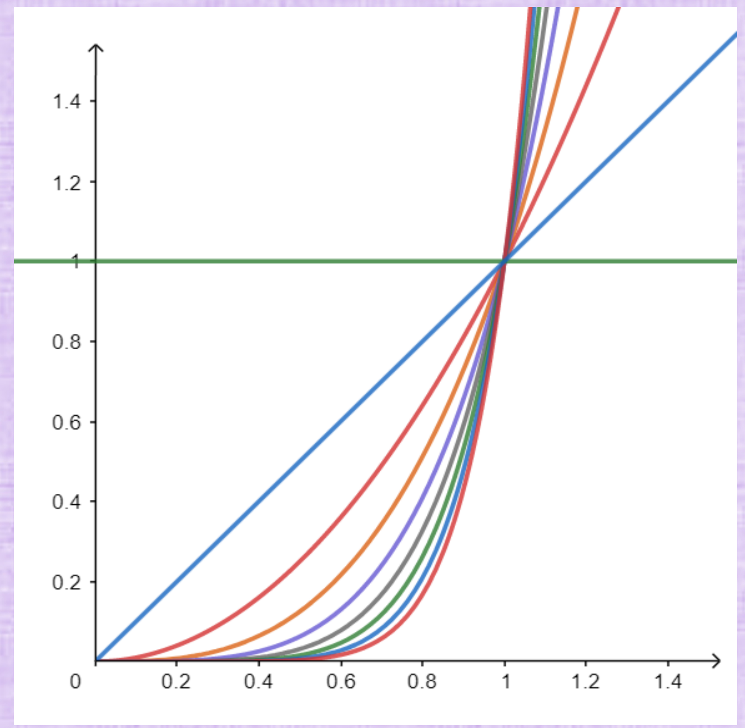
- Given m data points (x_i, y_i), find the unique polynomial that passes through them: $y = c_1 + c_2 x + c_3 x^2 + \cdots + c_m x^{m-1}$
- Write an equation for each data point, note that the equations are linear, and put into matrix form
- For example, consider $(1,3)$, $(2,4)$, $(5,-3)$ and a quadratic polynomial
- Then, $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 5 & 25 \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ -3 \end{pmatrix}$ gives

$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 7/2 \\ -5/6 \end{pmatrix}$ and $f(x) = \frac{1}{3} + \frac{7}{2}x - \frac{5}{6}x^2$



Polynomial Interpolation

- In general, solve $Ac = y$ where A (the Vandermonde matrix) has a row for each data point of the form $(1 \quad x_i \quad x_i^2 \quad \cdots \quad x_i^{m-1})$

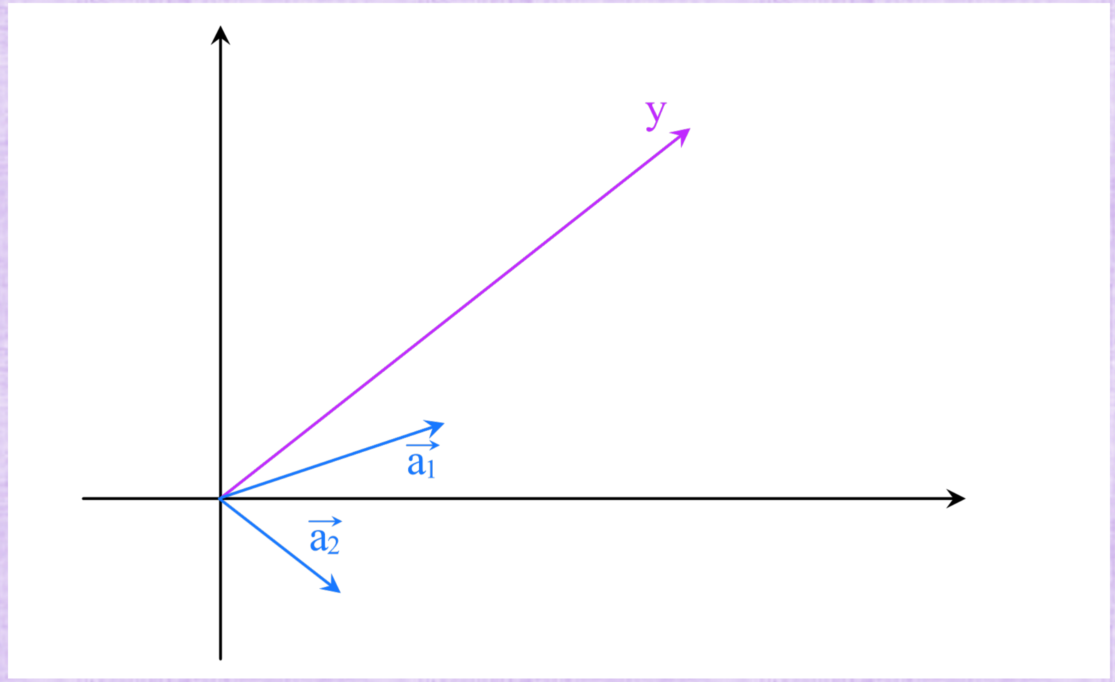


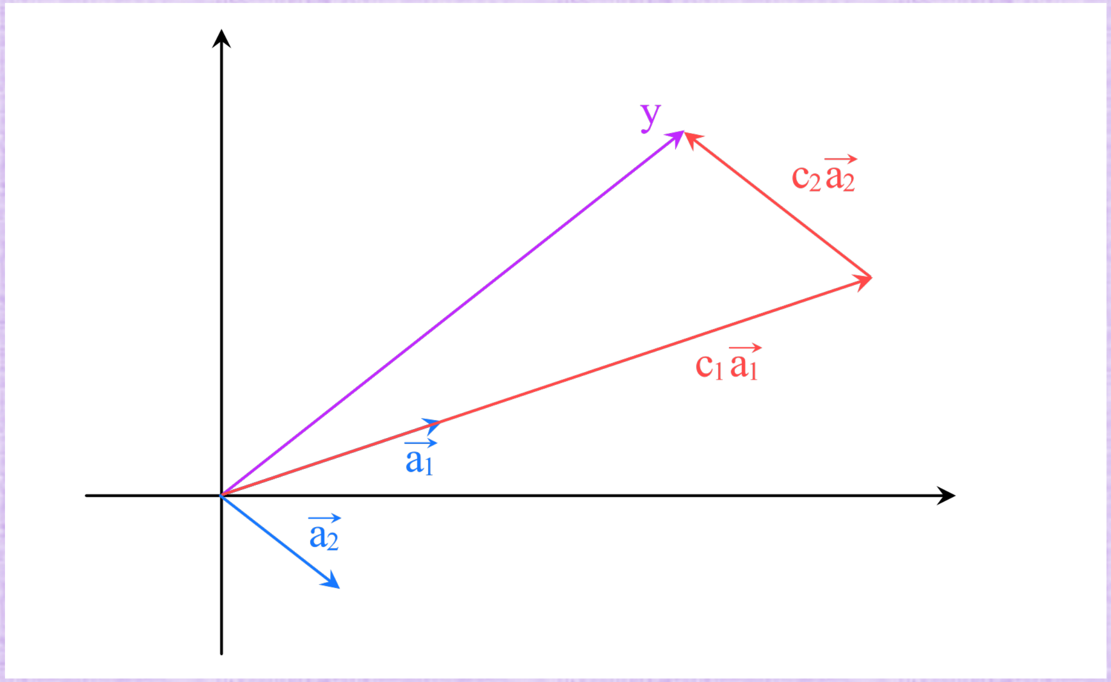
$f(x) = 1, x, x^2, x^3, x^4, x^5, x^6, \mathrm{x}^7, \mathrm{x}^8$

- Monomials look more similar at higher powers
- This makes the rightmost columns of a Vandermonde matrix tend to become more parallel
  - Round-off errors and other numerical approximations exacerbate this
- More parallel columns make the matrix less invertible, and thus it becomes more difficult to solve for the parameters c_k
- Too nearly parallel columns make the matrix ill-conditioned to invert (and thus difficult/impossible to invert with a computer)

Matrix Columns as Vectors

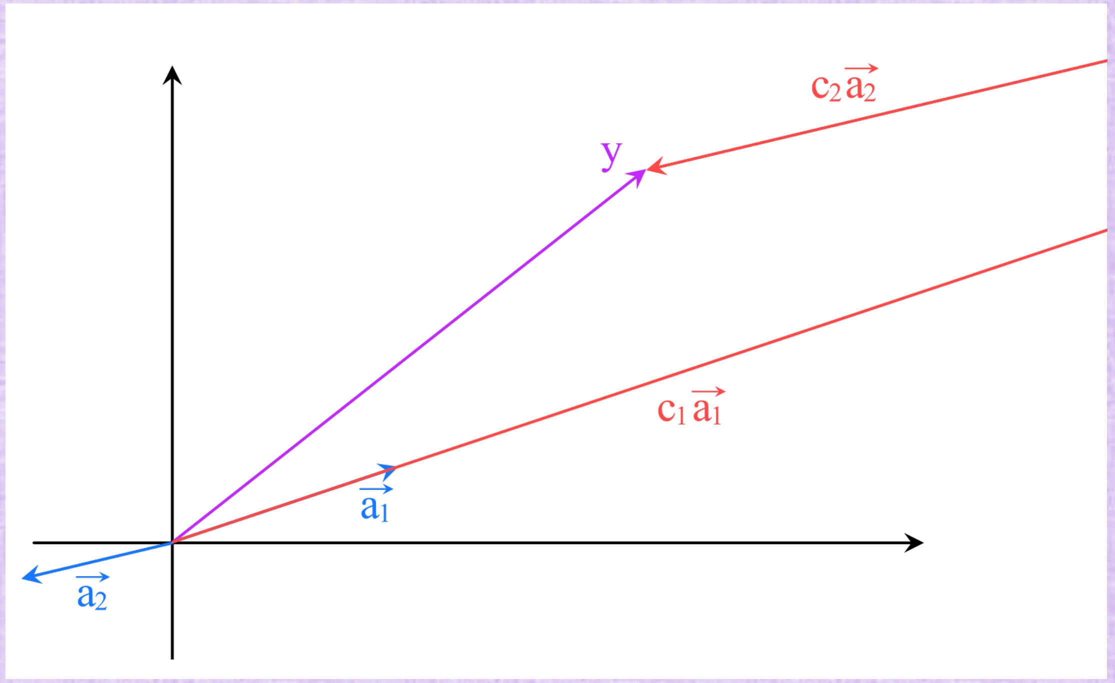
- Let the k-th column of A be vector a_k, so $Ac = y$ is equivalent to $\sum_k c_k a_k = y$
- Find a linear combination of the columns of A that gives the right hand side vector y





Matrix Columns as Vectors

- As columns become more parallel, the values of c become arbitrarily large, ill-conditioned, and prone to error



- In this example, the red vectors go too far to the right and back in order to (fully) illustrate

Singular Matrices

- If two columns of a matrix are parallel, they may be combined in an infinite number of ways while still obtaining the same result
  - Thus, the problem does not have a <u>unique solution</u>
- In addition, the n columns of A span at most an $n-1$ dimensional subspace
  - So, the <u>range</u> of A is at most $n-1$ dimensional
- If the right hand side vector is not contained in this $n-1$ dimensional subspace, the problem has <u>no solution</u>
  - otherwise, there are <u>infinite solutions</u>

Singular Matrices

- If any column of a matrix is a linear combination of other columns, they may be combined in an infinite number of ways while still obtaining the same result
  - Thus, the problem does not have a unique solution
- In addition, the n columns of A span at most an $n-1$ dimensional subspace
  - So, the range of A is at most $n-1$ dimensional
- If the right hand side vector is not contained in this $n-1$ dimensional subspace, the problem has no solution
  - otherwise, there are infinite solutions

Near Singular Matrices

- With limited numerical precision, one struggles when columns (or linear combinations of columns) are too close to being parallel to each other
- Analytically invertible matrices may not be computationally invertible
- A condition number can be used to describe how close a matrix is to being non-invertible on a computer
- The condition number is ∞ for a singular matrix and 1 for the identity matrix

Being Careful: Polynomial Interpolation

- Given basis functions ϕ and unknowns c:

$$y = c_1\phi_1 + c_2\phi_2 + \cdots + c_n\phi_n$$

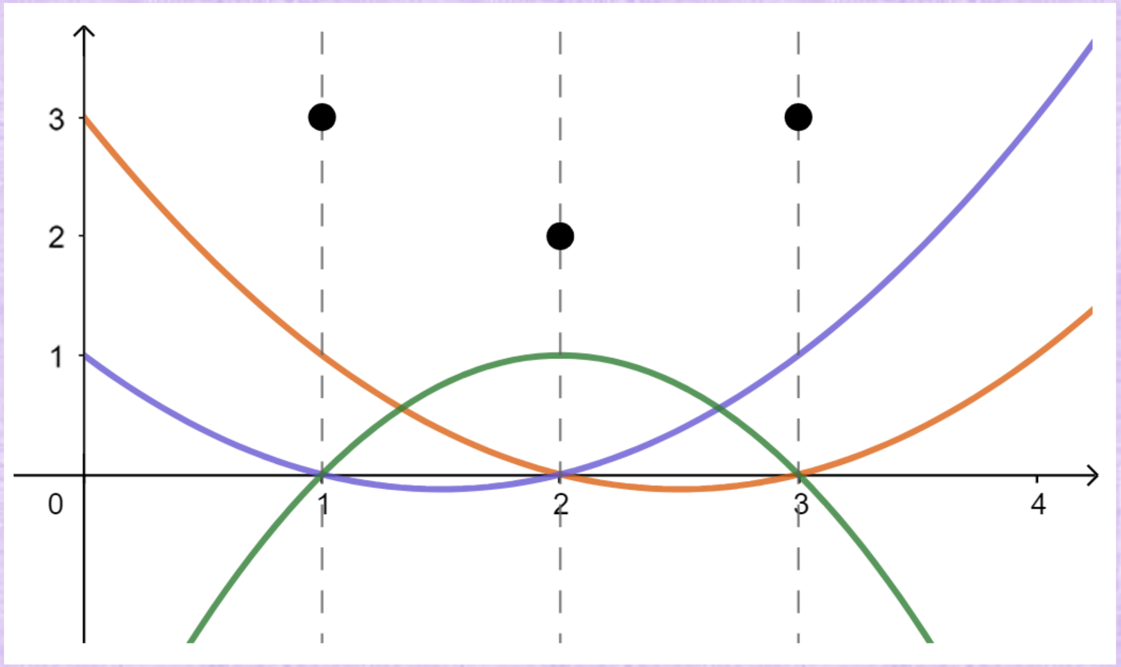
- Monomial basis: $\phi_k(x) = x^{k-1}$
- As we have seen, the Vandermonde matrix may become near-singular and difficult to invert!

Lagrange Basis

- Basis functions: $\phi_k(x) = \frac{\prod_{i \neq k} x - x_i}{\prod_{i \neq k} x_k - x_i}$
- Thus, $\phi_k(x_k) = 1$
- Thus, $\phi_k(x_i) = 0$ for $i \neq k$
- As usual: write an equation for each point, note that the equations are linear, and put into matrix form
- Obtain $Ac = y$ where A is the identity matrix (i.e. $Ic = y$), so $c = y$ trivially
- Easy to solve for c, but evaluation of the polynomial (with lots of terms) is expensive
  - i.e. inference is expensive

Lagrange Basis

- Consider data (1,3), (2,2), (3,3) with quadratic basis functions that are 1 at their corresponding data point and 0 at the other data points



- $\phi_1(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)} = \frac{1}{2}(x-2)(x-3)$
- $\phi_1(1) = 1, \phi_1(2) = 0, \phi_1(3) = 0$
- $\phi_2(x) = \frac{(x-1)(x-3)}{(2-1)(2-3)} = -(x-1)(x-3)$
- $\phi_2(1) = 0, \phi_2(2) = 1, \phi_2(3) = 0$
- $\phi_3(x) = \frac{(x-1)(x-2)}{(3-1)(3-2)} = \frac{1}{2}(x-1)(x-2)$
- $\phi_3(1) = 0, \phi_3(2) = 0, \phi_3(3) = 1$

Newton Basis

- <u>Basis functions</u>: $\phi_k(x) = \prod_{i=1}^{k-1} x - x_i$
- $Ac = y$ has a lower triangular A (as opposed to being dense or diagonal)
- Columns don't overlap, and it's not too expensive to evaluate/inference
- Can solve via a divided difference table:
    - Initially: $f[x_i] = y_i$
    - Then, at each level, recursively: $f[x_1, x_2, \cdots, x_k] = \frac{f[x_2,x_3,\cdots,x_k] - f[x_1,x_2,\cdots,x_{k-1}]}{x_k - x_1}$
    - Finally: $c_k = f[x_1, x_2, \cdots, x_k]$
- As usual, high order polynomials still tend to be oscillatory
    - Using unequally spaced data points can help, e.g. Chebyshev points

Summary

- Monomial/Lagrange/Newton basis all give the same exact unique polynomial
    - as one can see by multiplying out and collecting like terms

- But the representation used makes it easier/harder to find the polynomial as well as to subsequently evaluate the polynomial

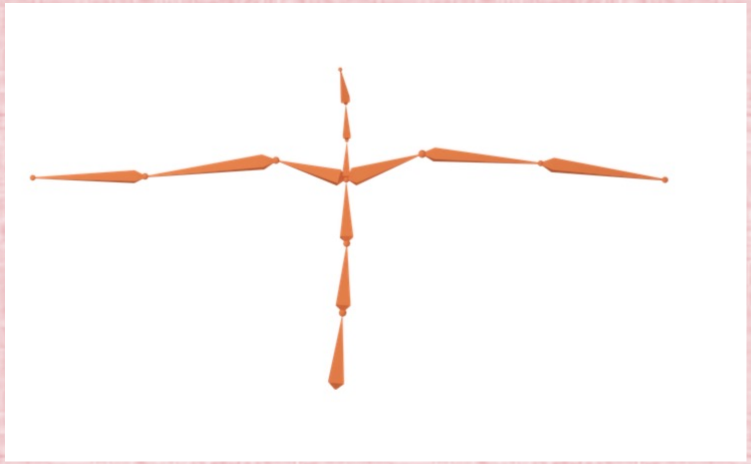
Representation Matters

- Consider: Divide CCX by VI
- As compared to: Divide 210 by 6

- See Chapter 15 on Representation Learning in the Deep Learning book

Predict 3D Cloth Shape from Body Pose

- Input: pose parameters θ are joint rotation matrices
  - 10 upper body joints with a $3x3$ rotation matrix for each gives a $90D$ pose vector ($30D$ when using quaternions)
  - Ignore global translation/rotation of the root frame
- Output: $3D$ cloth shape φ
  - $3{,}000$ vertices in a cloth triangle mesh gives a $9{,}000D$ shape vector
- Function f: $\mathbf{R}^{90} \rightarrow \mathbf{R}^{9000}$



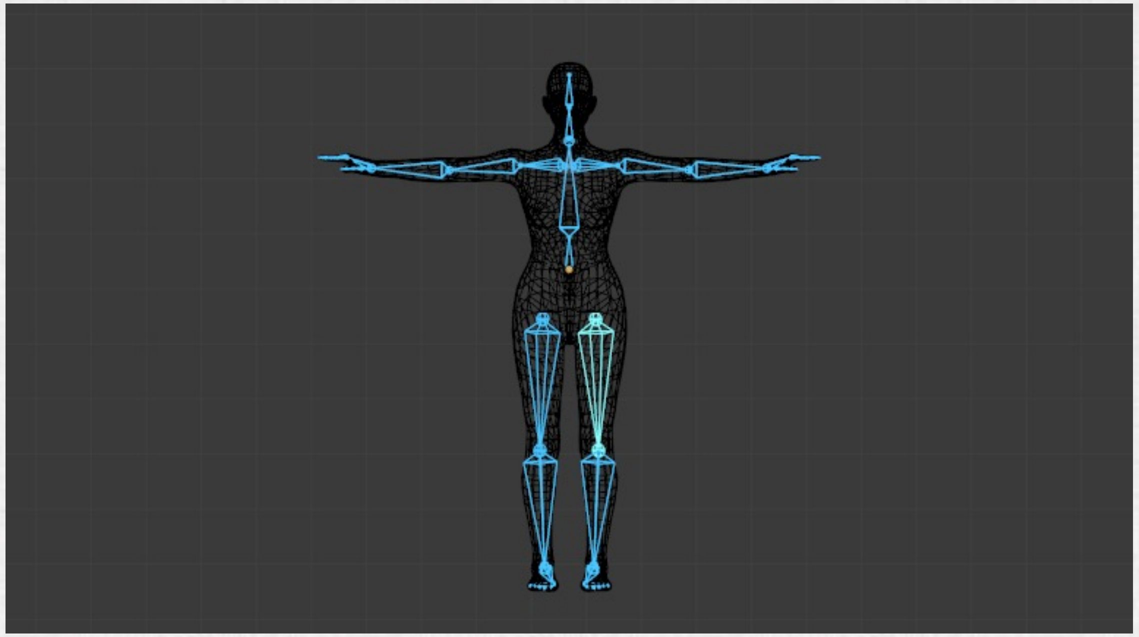
$\longrightarrow$

Approach

- Given: m training data points (θ_i, φ_i) generated from the true/approximated function $\varphi_i = f(\theta_i)$
  - E.g. using physical simulation or computer vision techniques
- Goal: learn an $\hat{f}$ that approximates f
  - i.e. $\hat{f}(\theta) = \hat{\varphi} \approx \varphi = f(\theta)$
- Issue: As joints rotate (rotation is highly nonlinear), cloth vertices move in complex nonlinear ways that are difficult to capture with a neural network
  - i.e. it is difficult to ascertain a suitable $\hat{f}$
- How should the nonlinear rotations be handled?

Skinning

- Deforms a body surface mesh to match a skeletal pose
    - well studied and widely used in graphics
- Each vertex of the body surface mesh is associated with one or more nearby bones
- A weight (for each bone/vertex pair) dictates how much a change in a bone's position/orientation impacts a vertex's position
- As the pose changes, bone changes dictate new positions for body surface mesh vertices



Credit: Blender website

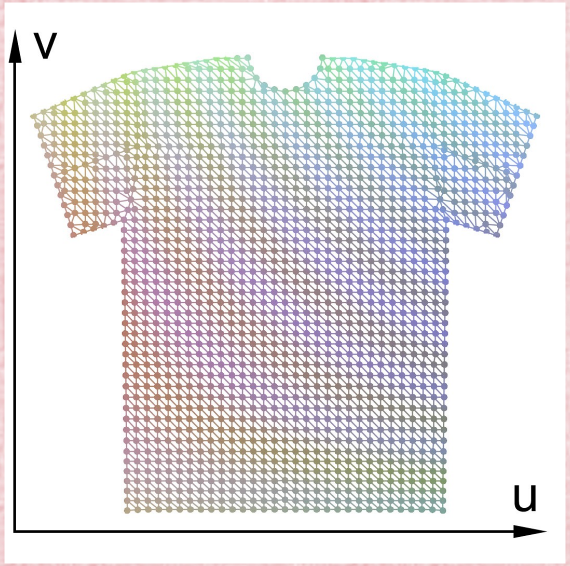
Leveraging Skinning (to be careful)

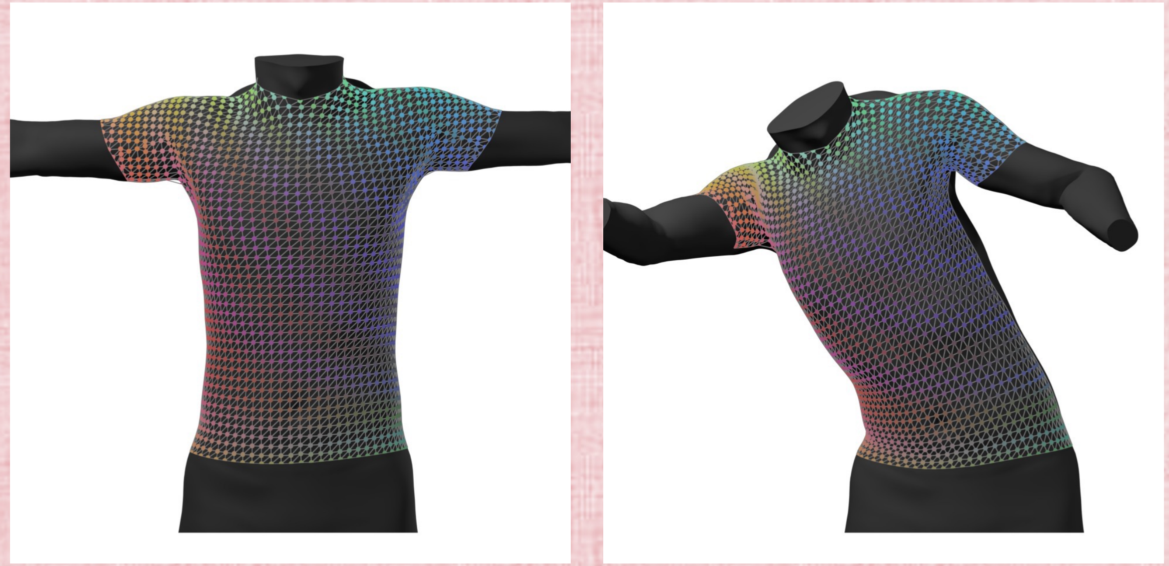
- Leverage the plethora of prior work on procedural skinning to estimate the body surface mesh S based on pose parameters θ
- Then, represent the cloth mesh as offsets $D(\theta)$ from the skinned mesh $S(\theta)$
- Overall, $\varphi = f(\theta) = S(\theta) + D(\theta)$, where only $D(\theta)$ needs to be learned

- The skinning prior $S(\theta)$ captures much of the nonlinearities, so that the remaining $D(\theta)$ is a smoother function and thus easier to approximate/learn

Shrink Wrap the Cloth Mesh

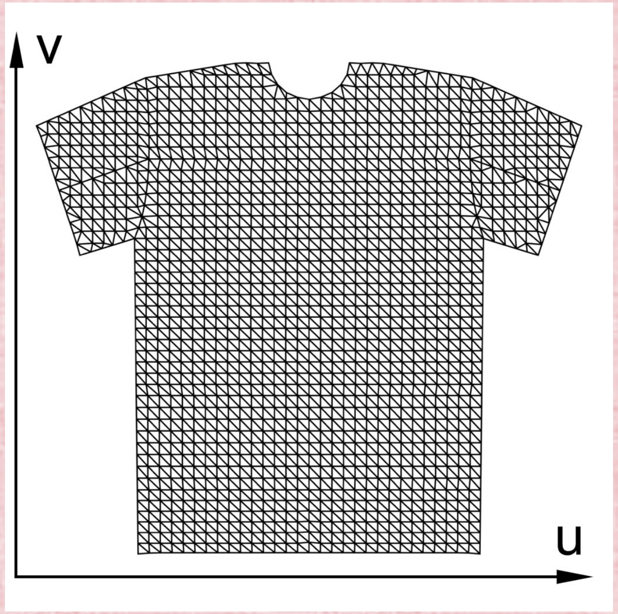
- Shrink-wrap the cloth vertices to the body triangle mesh
- Barycentrically embed the cloth vertices to follow body mesh triangles
- As the body deforms, cloth vertices move with their parent triangles

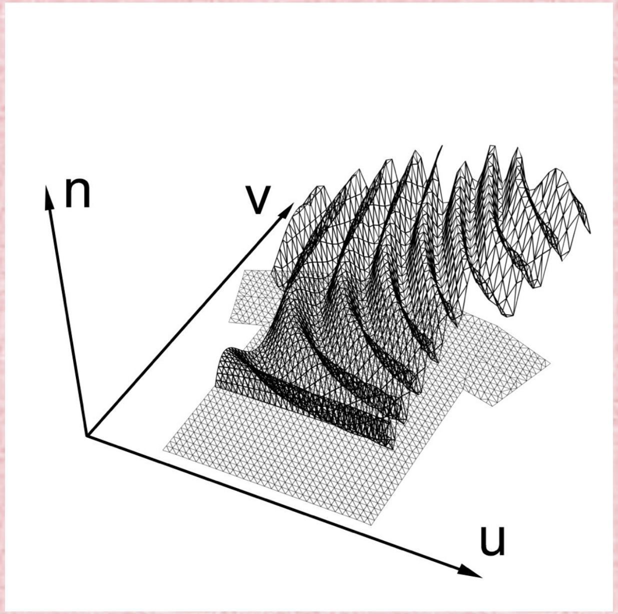


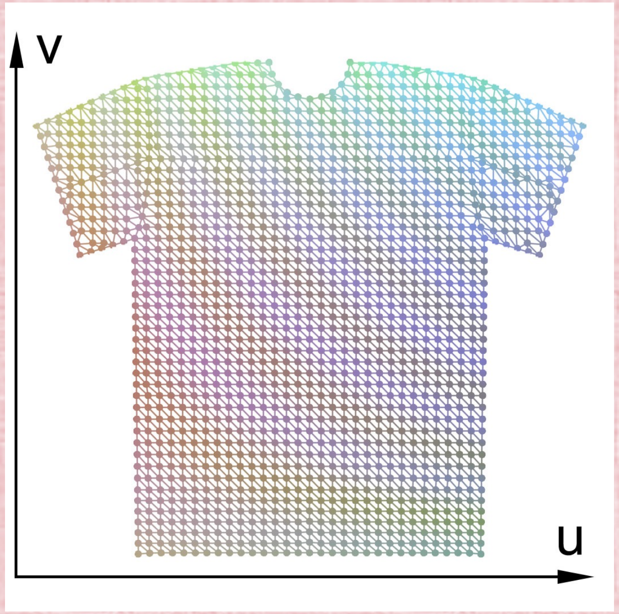


Displacement Map

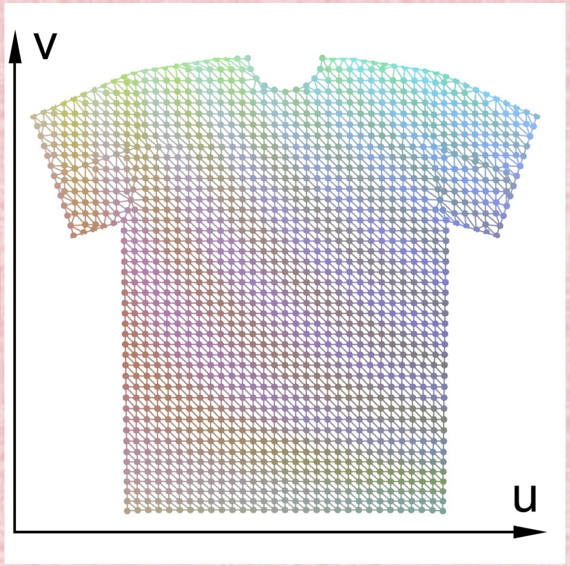
- Assign (u, v) texture coordinates to the cloth vertices and transfer the mesh into texture space
- Store (u, v, n) offsets in texture space
- Convert (u, v, n) offsets to RGB-triple color values

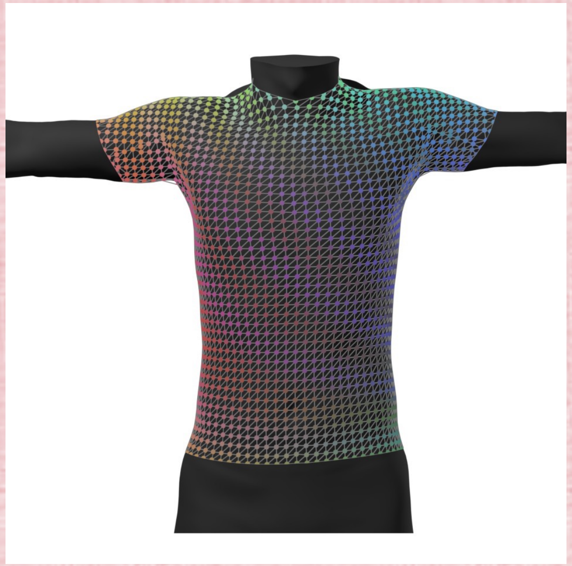


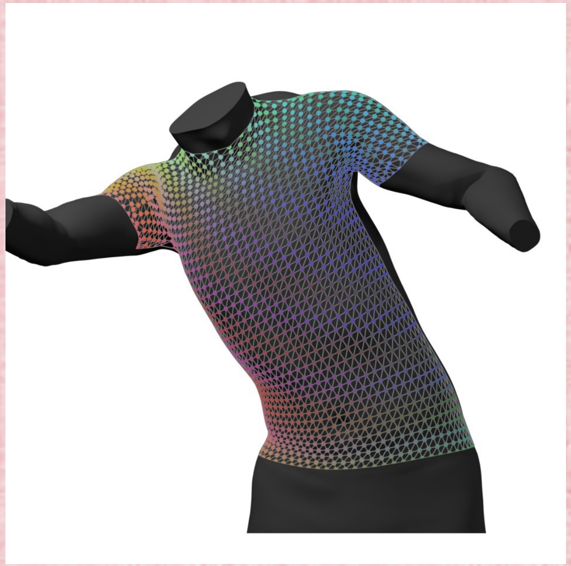


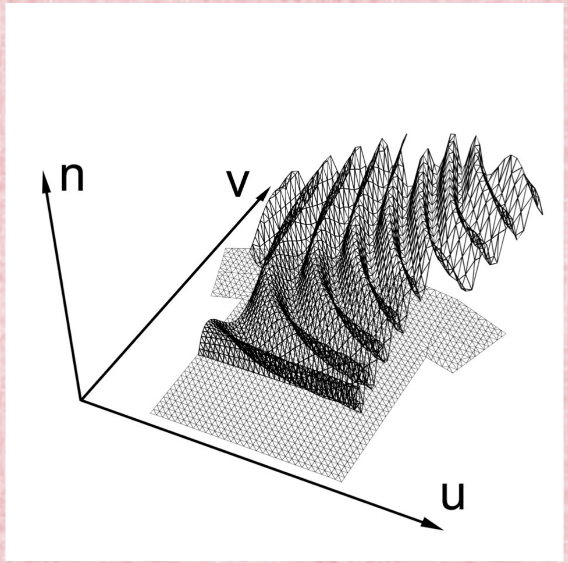


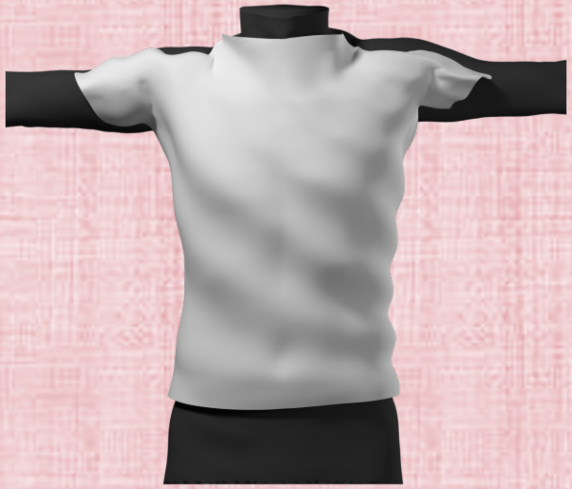
Displacement Map











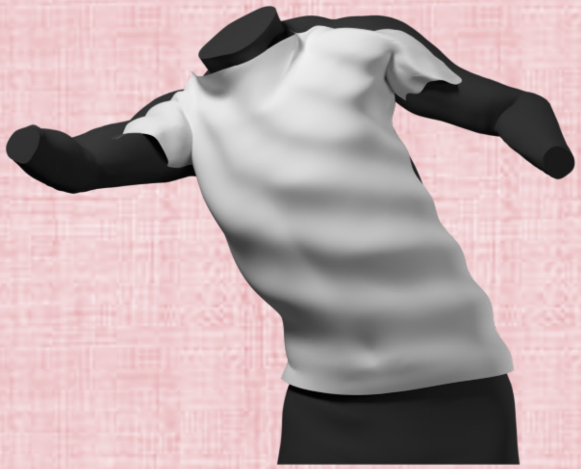
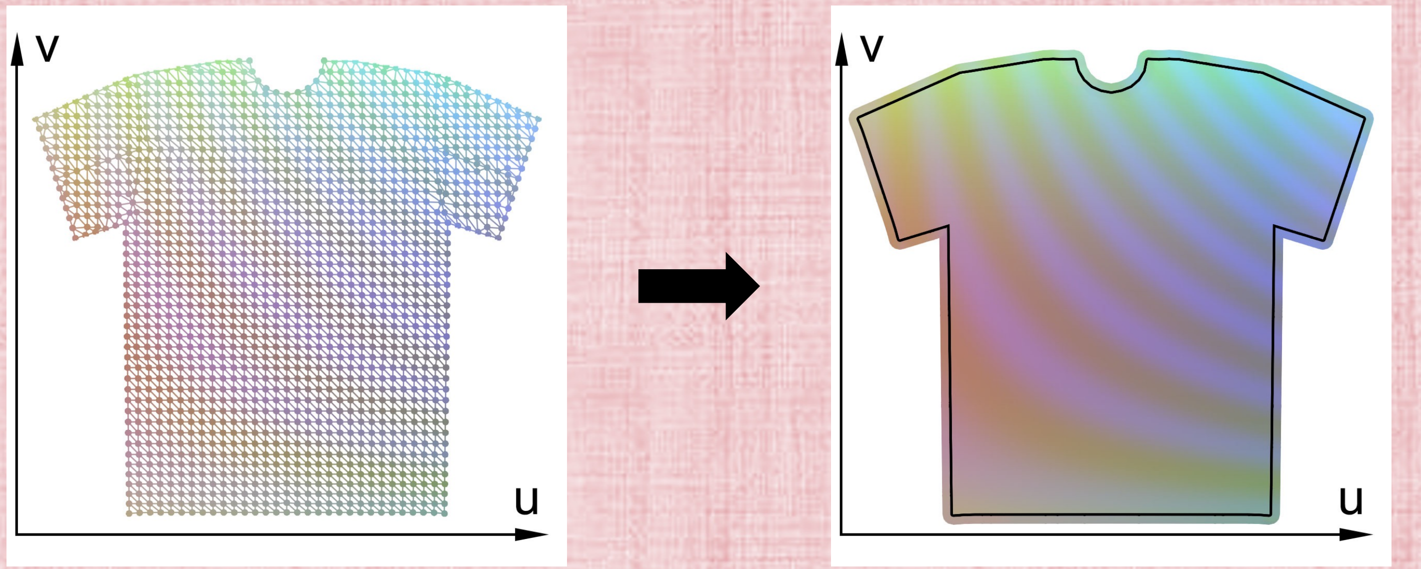


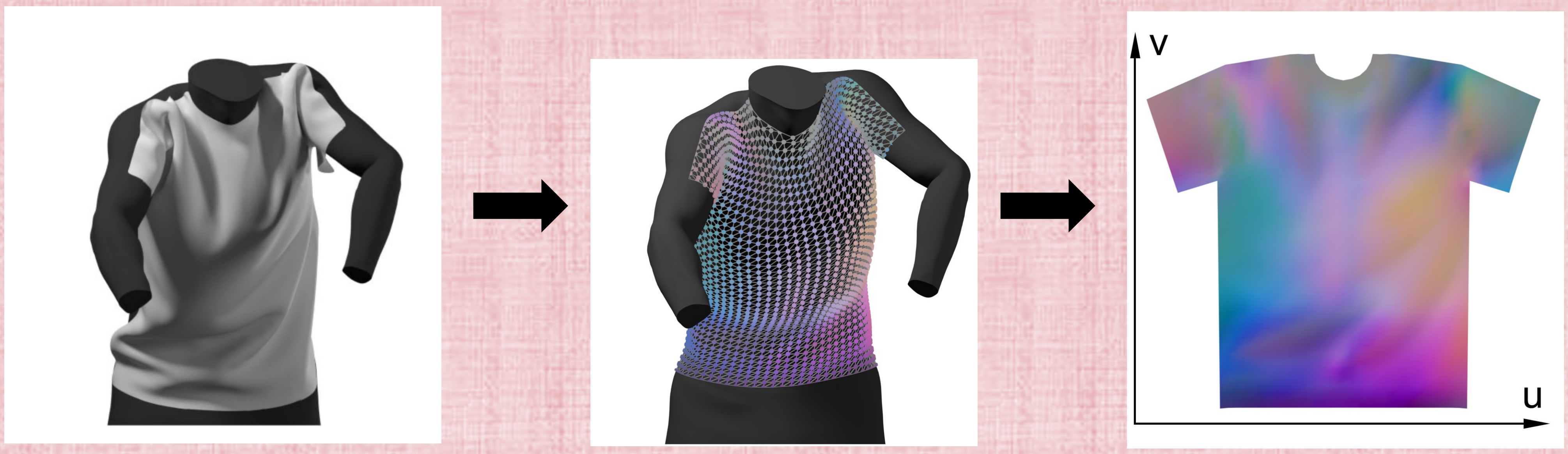
Image Based Cloth

- Rasterize triangle vertex colors to 2D image pixels (in texture space)
- Function output becomes a 2D RGB image, instead of displacements
- The images are more continuous than cloth vertices (which have discrete mesh/graph topology)
- Learn to predict images as a function of pose θ, using Convolutional Neural Networks (CNNs)



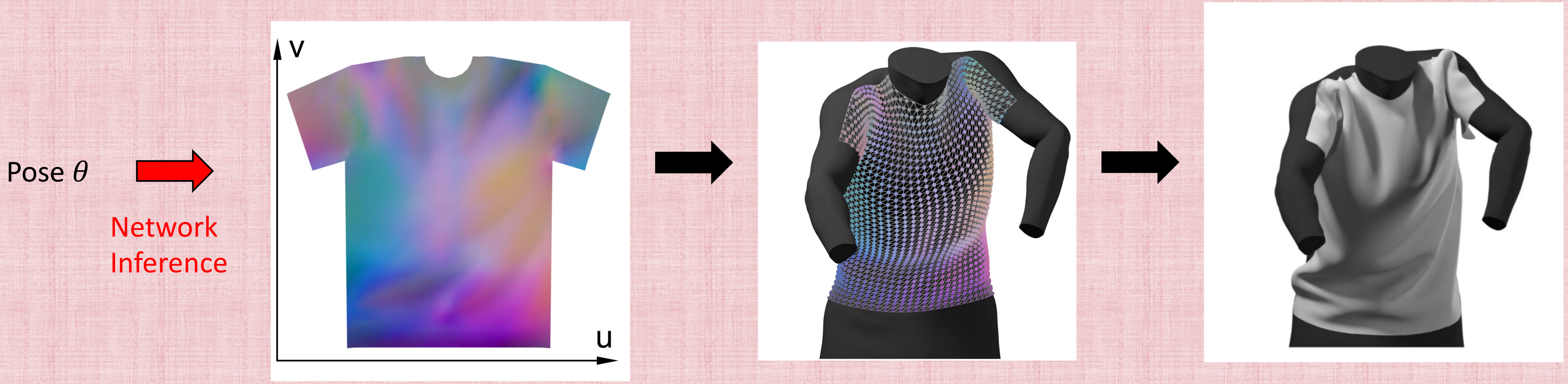
Training Data

- For each pose in the training data, calculate per-vertex offsets and rasterize them into an image in texture space



Inference

- Learn to predict an image from pose parameters, i.e. learn $\hat{I}(\theta) \approx I(\theta)$
- Given an inferenced $\hat{I}(\theta)$, interpolate to cloth vertices and convert RGB values to offsets added to the skinned vertex positions: $\hat{\varphi}(\theta) = S(\theta) + \psi(\hat{I}(\theta))$


Pose θ
Network Inference
v
u