CS 188-2
Discussion-Week 10

Ali Hatamizadeh

11/06/2019

Final Exam Review
Based on 3 most Popular votes
(as time permitting)

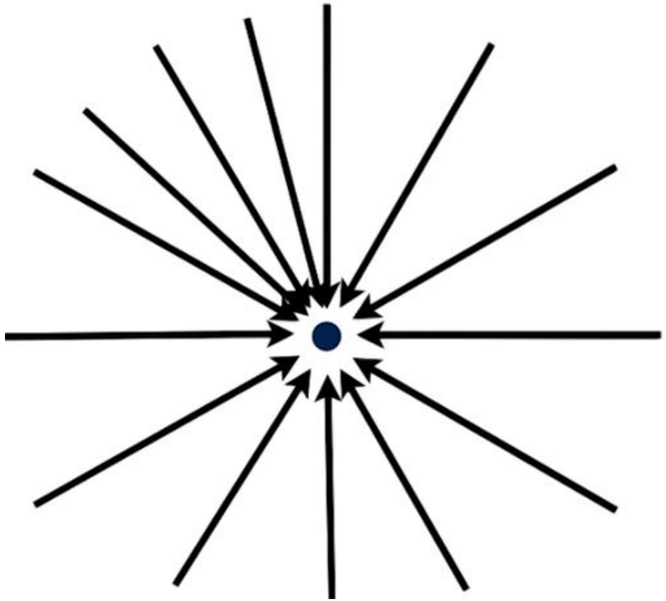
Homography & Panorama

- Let's assume that our goal is to create a panorama

For this, we want to take a number of pictures and stitch them together

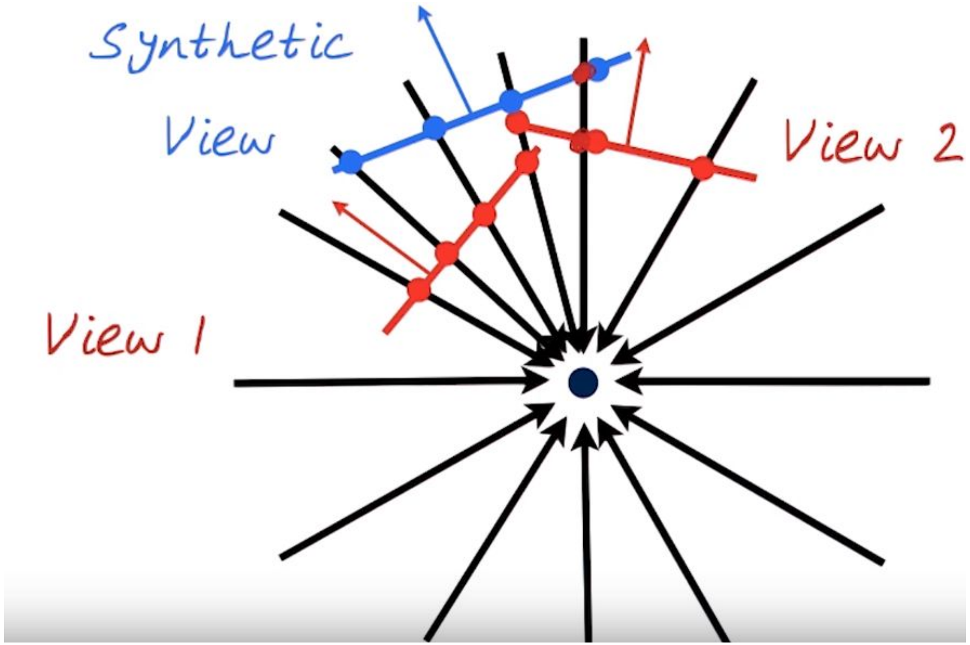
Bundle of Rays

- Let's say we have a camera which we rotate around a fixed point, and we capture a concentric set of light as shown below



Bundle of Rays

- If we have 2 views, we can create this synthetic view which happens to be somewhere in between the two.



Bundle of Rays

- It is possible to create any synthetic view as long as it has the same center of projection
- That's why when we create panoramas, we would want to rotate the camera around a single point

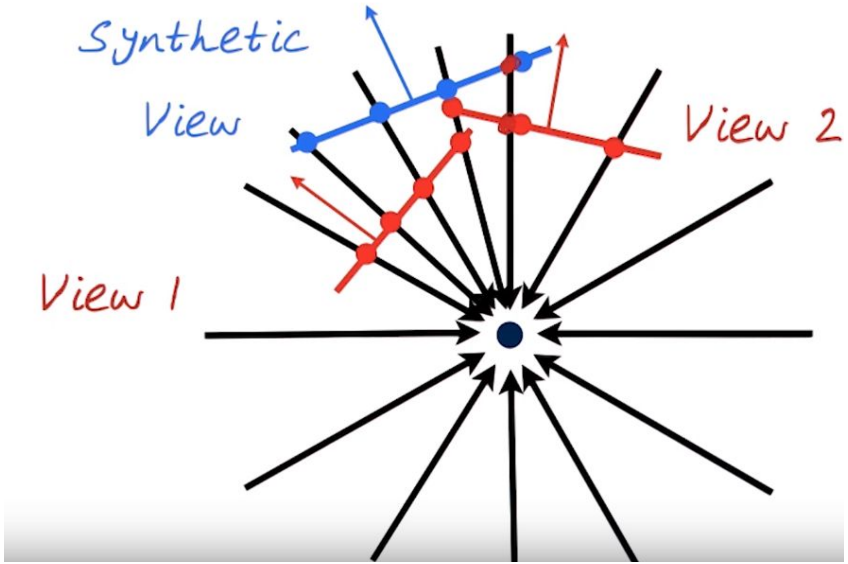
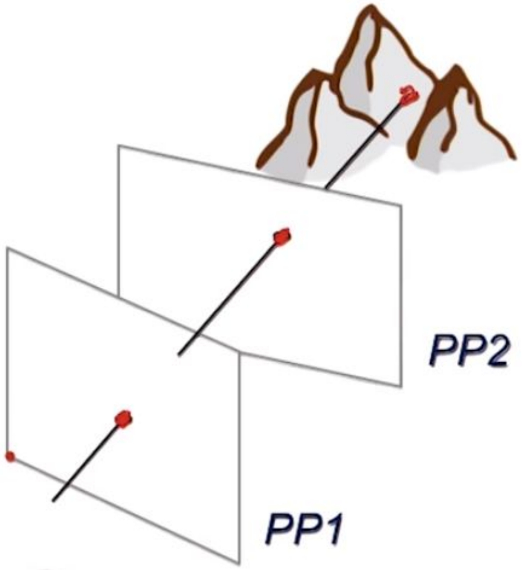


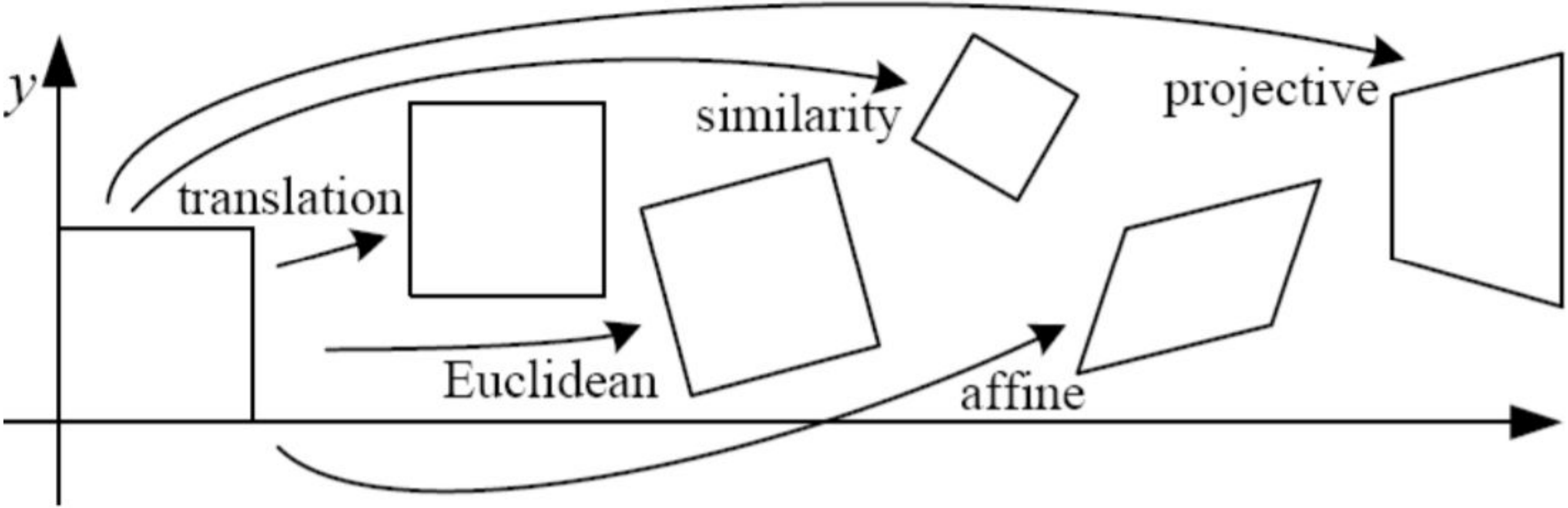
Image Re-Projection

- We would like to relate two images that have been taken from the same camera center
  - Technically, we would like to map corresponding pixels
- What's the procedure ?
  - Cast a ray from all pixels in PP1
  - Find where each ray is intersecting PP2 (feature detection)
- Let's treat this problem of finding the correspondence

  as a 2D image warping !



2D Image Transformation



2D Image Transformation

- Now for each of these:
  - Translation: 2 unknowns
  - Rotation: 1 unknown
  - Euclidean: 3 unknowns
  - Affine: 6 unknowns
  - Projective: 8 unknowns

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- What is projective transformation ?

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

2D Image Transformation

- What is projective transformation ?
  - Combination of affine + projective warps
- What are its properties?
  - Origin does not necessarily map to origin
  - Parallel lines may not remain parallel
  - Lines remain lines
  - Ratios are not preserved
  - 8 parameters !

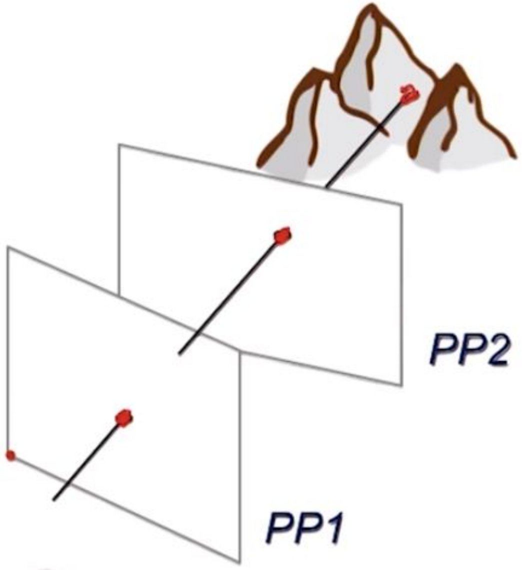
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Homography

- Relate two images with the same camera center
  - Rectangle should map to (arbitrary) quadratal (lines remain straight)

$$p' = Hp$$

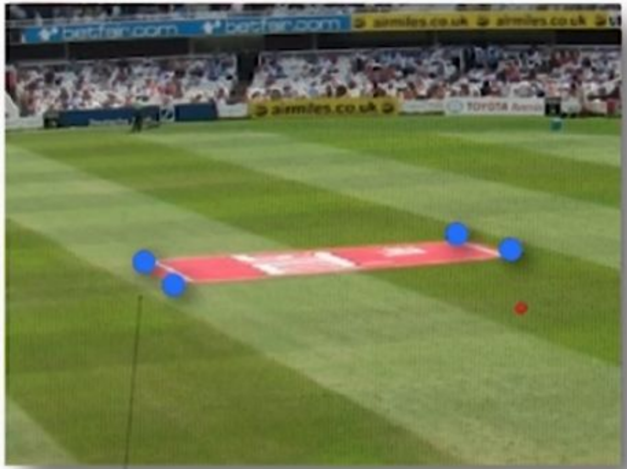
$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Homography

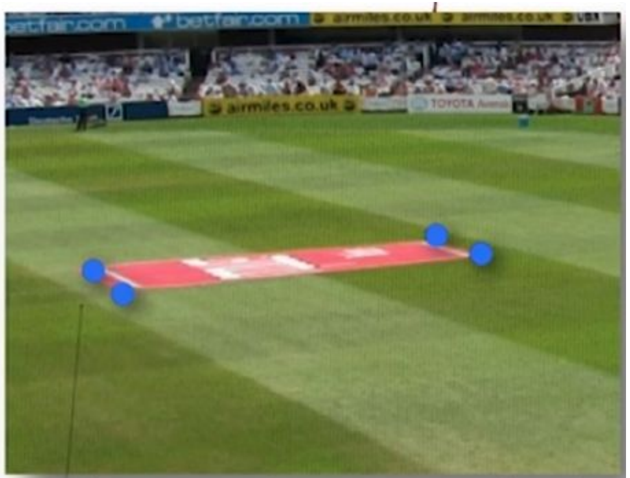
- Computing homography

Homography



(x, y)

$p_1, p_2, \ldots, p_n$



$(wx'/w, wy'/w) = (x', y')$

$p'_1, p'_2, \ldots, p'_n$

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad p' = Hp$$

Computing Homography

- We need a system of linear equations

  Ax=B

  x=[a,b,c,d,e,f,g,h]^T

- We need at least 8 eqs.
- Or 4 points.
- But, we can have it over-constrained
  - Sample more points!
  - Min of least squared solution ||ax-B||^2

$$p' = Hp$$

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Computing Homography

- We need a system of linear equations

  Ax=B

  x=[a,b,c,d,e,f,g,h]^T

- We need at least 8 eqs.
- Or 4 points.
- But, we can have it over-constrained
  - Sample more points!
  - Min of least squared solution ||ax-B||^2

$$p' = Hp$$

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Computing Homography

- For n points:

$$\mathbf{A}\boldsymbol{h} = \mathbf{0}$$

$$\begin{bmatrix} \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \\ \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \\ \vdots \\ \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Computing Homography

- Computing homography using DLT:

Given $\{\boldsymbol{x}_i, \boldsymbol{x}'_i\}$ solve for H such that $\boldsymbol{x}' = \mathbf{H}\boldsymbol{x}$

1. For each correspondence, create 2x9 matrix $\mathbf{A}_i$

2. Concatenate into single 2n x 9 matrix $\mathbf{A}$

3. Compute SVD of $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$

4. Store singular vector of the smallest singular value $\boldsymbol{h} = \boldsymbol{v}_{\hat{i}}$

5. Reshape to get $\mathbf{H}$

Random Sample Consensus (RANSAC)

1. Sample (randomly) the number of points required to fit the model
2. Solve for model parameters using samples
3. Score by the fraction of inliers within a preset threshold of the model

Repeat 1-3 until the best model is found with high confidence

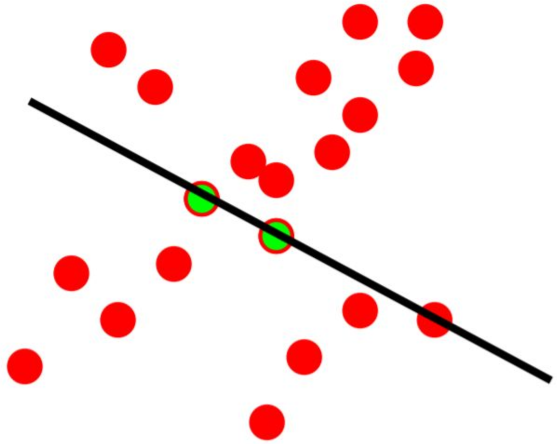
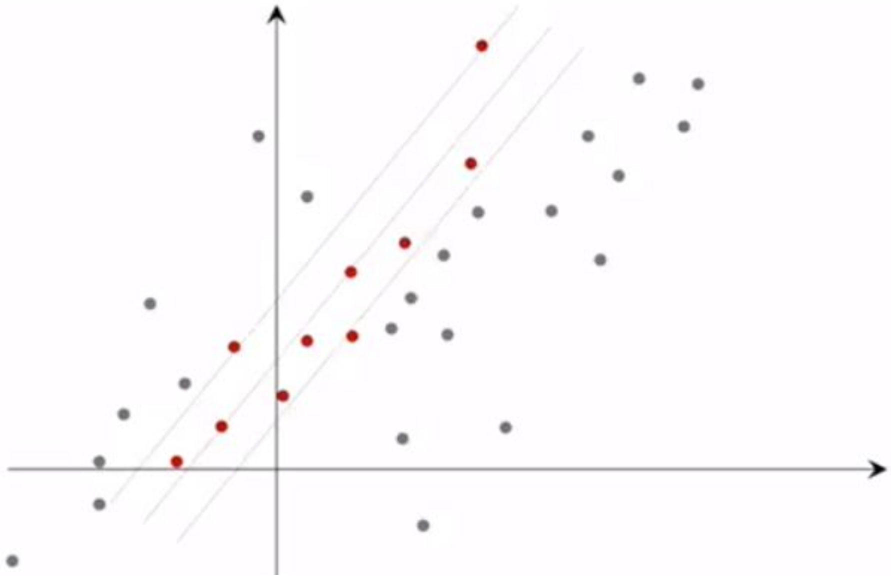


Image Correspondence Pipeline

1. Feature point detection (e.g. cornet harris detection for corner detection)
2. Feature Point Description (e.g. multi-scale oriented patch descriptor)
3. Feature Matching (we have an issue with outliers !)

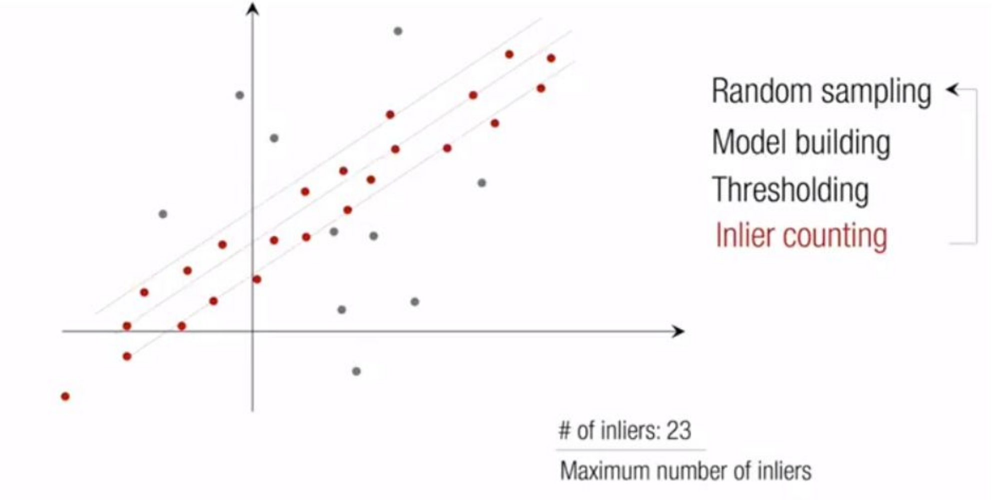
UCLA Random Sample Consensus (RANSAC)

Given 2 points , let’s fit a line and then check how good we are doing with respect to other points. We define a threshold and if the distance of other points to the fitted line is less than that threshold, then they are considered as inliers.



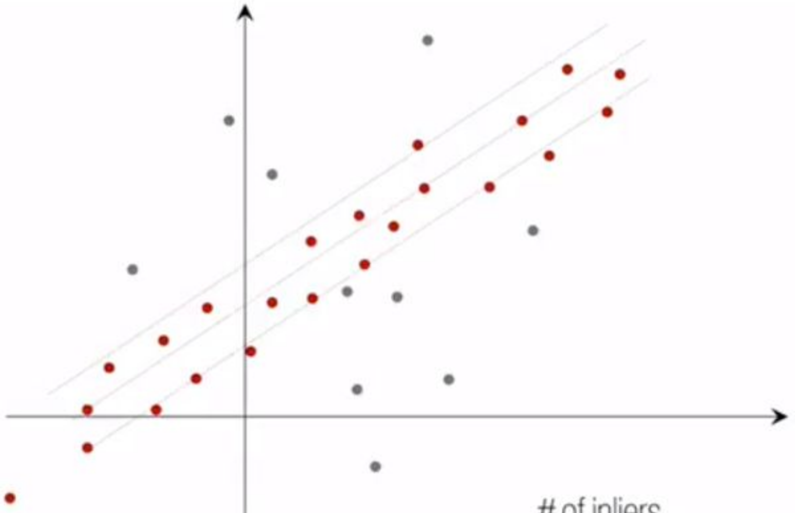
Random Sample Consensus (RANSAC)

We repeat this process many times, hoping that the two chosen points are positioned well (well within the center of inliners). If we repeat this process sufficiently enough, eventually we will converge (get lucky !). We stop where we found the best two lines that everyone agrees.



UCLA Random Sample Consensus (RANSAC)

How many times do we need to do the sampling ?



Random sampling

Model building

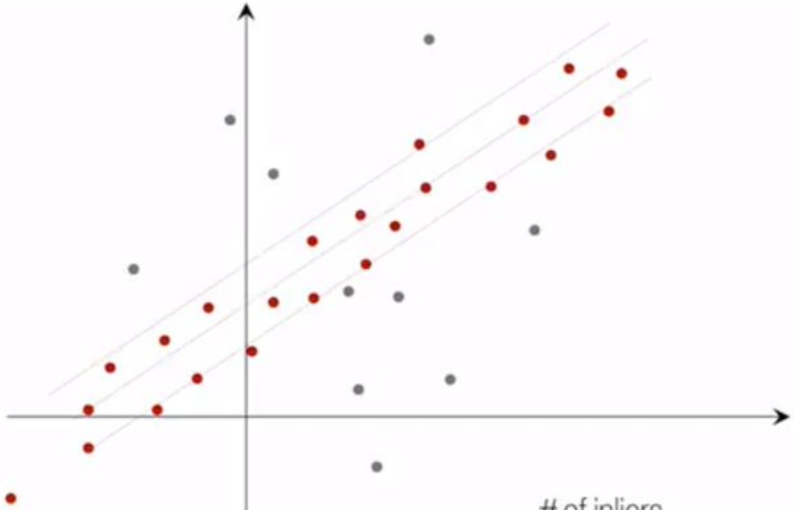
Thresholding

Inlier counting

Probability of choosing an inlier: $w = \dfrac{\#\text{ of inliers}}{\#\text{ of samples}}$

Probability of building a correct model: w^n where n is the number of samples to build a model.

Random Sample Consensus (RANSAC)



- Random sampling
- Model building
- Thresholding
- Inlier counting

Probability of choosing an inlier: $w = \dfrac{\#\text{ of inliers}}{\#\text{ of samples}}$

Probability of building a correct model: w^n where n is the number of samples to build a model.

Probability of not building a correct model during k iterations: $\left(1-w^n\right)^k$

$\left(1-w^n\right)^k = 1-p$ where p is desired RANSAC success rate.

$$k = \frac{\log\left(1-p\right)}{\log\left(1-w^n\right)}$$

UCLA Random Sample Consensus (RANSAC)

How to choose parameters?

- Number of samples N
  - Choose N so that, with probability p, at least one random sample is free from outliers (e.g. p=0.99) (outlier ratio: e)
- Number of sampled points s
  - Minimum number needed to fit the model
- Distance threshold δ
  - Choose δ so that a good point with noise is likely (e.g., prob=0.95) within threshold
  - Zero-mean Gaussian noise with std. dev. σ: $t^2=3.84\sigma^2$

$$N = \frac{\log(1-p)}{\log\left(1-(1-e)^s\right)}$$

	proportion of outliers *e*						
s	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

Homography using RANSAC

- RANSAC loop
  1. Get four point correspondences (randomly)
  2. Compute H using DLT
  3. Count inliers
  4. Keep H if largest number of inliers
- Recompute H using all inliers

Image Correspondence Pipeline

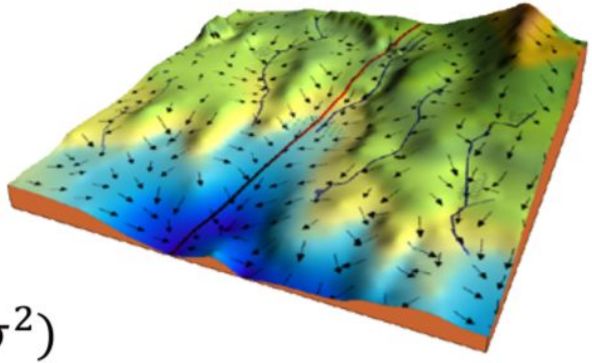
1. Feature point detection
   - Detect corners using the Harris corner detector.

2. Feature point description
   - Describe features using the Multi-scale oriented patch descriptor.

3. Feature matching *and* homography estimation
   - Do both simultaneously using RANSAC.

Gradient Descent

Gradient Descent



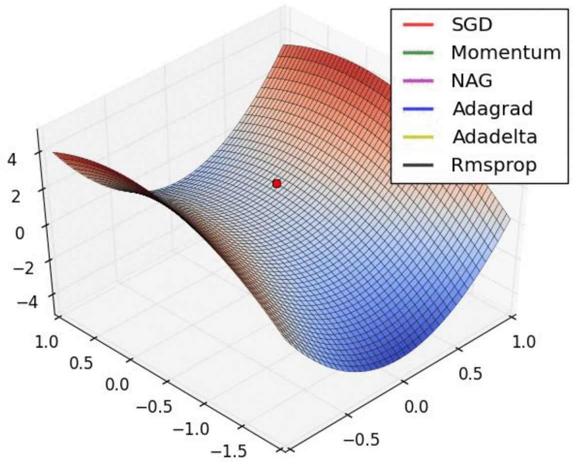
Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Gradient Descent

- Basic idea: minimize an objective function parameterized by model's parameters by updating the parameters in the opposite direction of the gradient of the objective function w.r.t. to the parameters

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$



Batch Gradient Descent

- Batch gradient descent: compute the gradient of the cost function w.r.t. to the parameters for the entire training dataset:
  - Need to calculate the gradients for the whole dataset to perform just **one** update.
  - Batch gradient descent may not be feasible to use in many cases. **Why** ?

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

- A Pythonic overview

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Batch Gradient Descent

- Batch gradient descent: compute the gradient of the cost function w.r.t. to the parameters for the entire training dataset:
    - Need to calculate the gradients for the whole dataset to perform just **one** update.
    - Batch gradient descent may not be feasible to use in many cases. **Why** ?

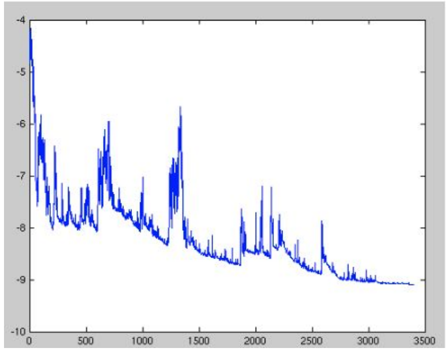
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

- Batch gradient descent is **guaranteed** to converge to the **global minimum** for **convex error surfaces** and to a **local minimum** for **non-convex surfaces**.

Stochastic Gradient Descent

- Stochastic Gradient Descent: performs a parameter update for each training example:
    - Performs redundant computations in large datasets
        - We draw random examples with replacement thus results in independent sampling
        - The loss is approximated from one training example leading to noisy gradient
        - Noisy gradient can be useful for non-convex loss functions
- (it needs to recompute the gradient for very similar examples before making an update)

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$



UCLA

Stochastic Gradient Descent

- SGD would probably show similar convergence behavior as batch gradient descent if we properly decrease the learning rate

- Pythonic overview

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Mini-Batch Gradient Descent

- Mini-batch gradient descent performs an update for every mini-batch of n training examples (a.k.a batch size):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

- It generally results in more stability in convergence
- It may not guarantee a proper convergence
- Initial learning rate and its schedule needs to be properly chosen
- Minimizing highly non-convex error functions can be challenging due to being trapped in their many suboptimal local minima
- For sparse data-sets, with data samples having different frequencies, having the same learning rate applied to all of them is not optimal.

Mini-Batch Gradient Descent

- Mini-batch gradient descent performs an update for every mini-batch of n training examples (a.k.a batch size):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

- Pythonic overview

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Momentum

- Around local optima, surface curves may be much more steep in one direction than the other.
- SGD has trouble navigating around these areas
  - SGD occilitates across the slopes of ravine making hesitant progress along the bottom towards the local optimum

Image 2: SGD without momentum

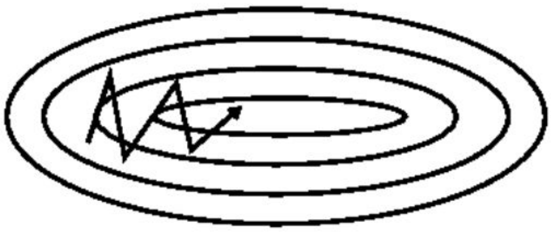
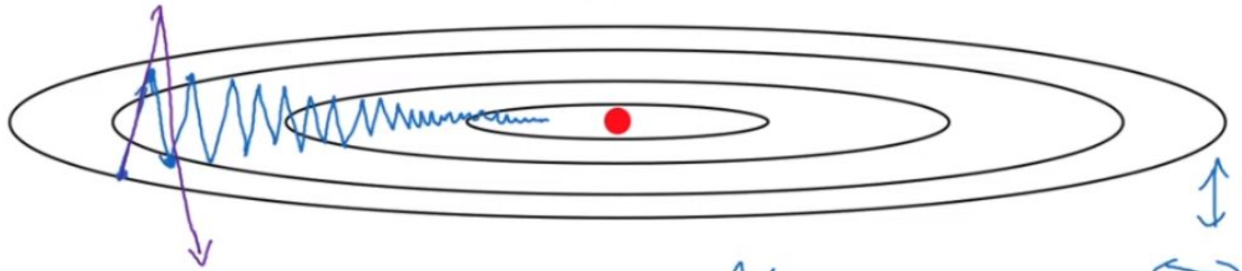


Image 3: SGD with momentum

UCLA

Momentum



↕ slower learning

↔ faster learning

Momentum:

On iteration t:

Compute dW, db on current mini-batch.

$$V_{dW} = \beta V_{dW} + (1-\beta) dW$$

$$V_{db} = \beta V_{db} + (1-\beta) db$$

"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

$$W := W - \alpha V_{dW}, \quad b := b - \alpha V_{db}$$

Momentum

- Momentum accelerates SGD in the relevant direction and dampens oscillations

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Image 2: SGD without momentum

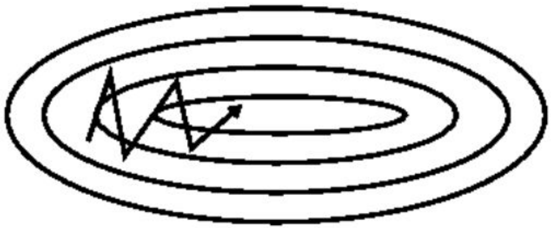


Image 3: SGD with momentum

Back Propagation

- Different way for computing gradient
  - Numerically
    - Slow, and approximate but easy to calculate !
  - Analytically
    - Fast and exact but easy to make mistakes !

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

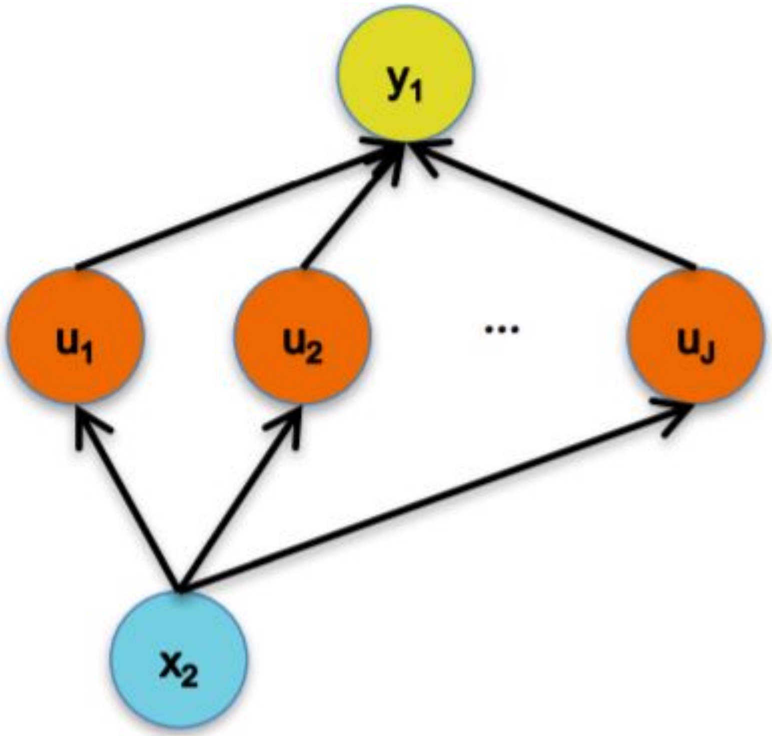
Back Propagation

- Chain Rule

Given: $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

Chain Rule:

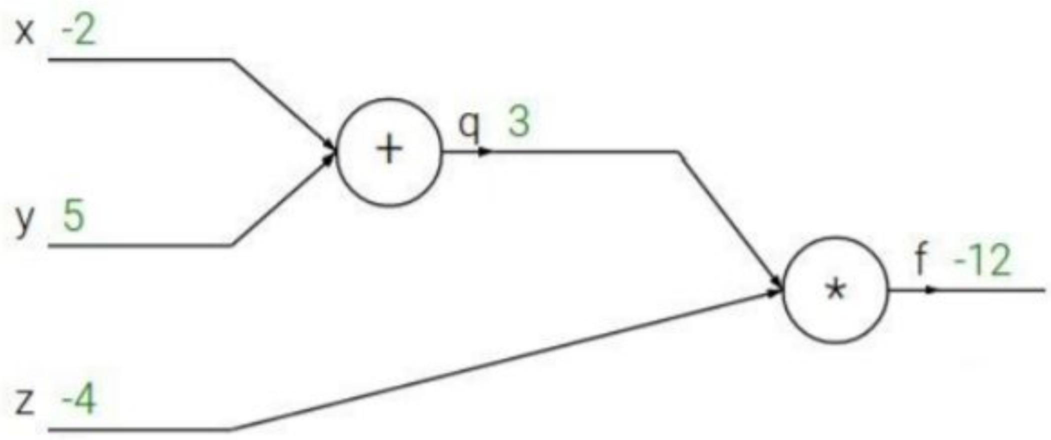
$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j}\frac{du_j}{dx_k}, \quad \forall i, k$$



UCLA

Back Propagation

- Let's do one example ...
  - Compute the forward pass in this computational graph



$$f(x, y, z) = (x + y)z$$

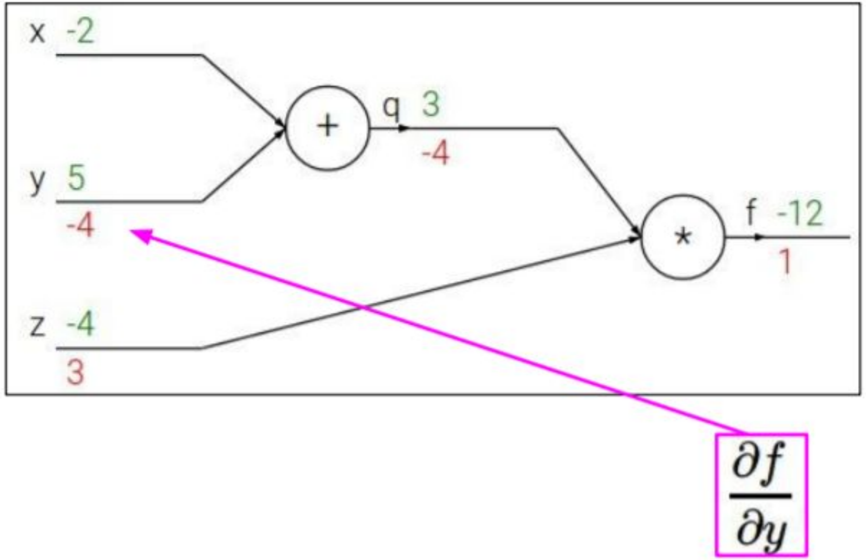
e.g. x = -2, y = 5, z = -4

Back Propagation

- Let's do one example ...
  - Compute $\frac{\partial f}{\partial y}$.

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



UCLA Back Propagation

- Let's do one example ...
  - Compute $\frac{\partial f}{\partial y}$.

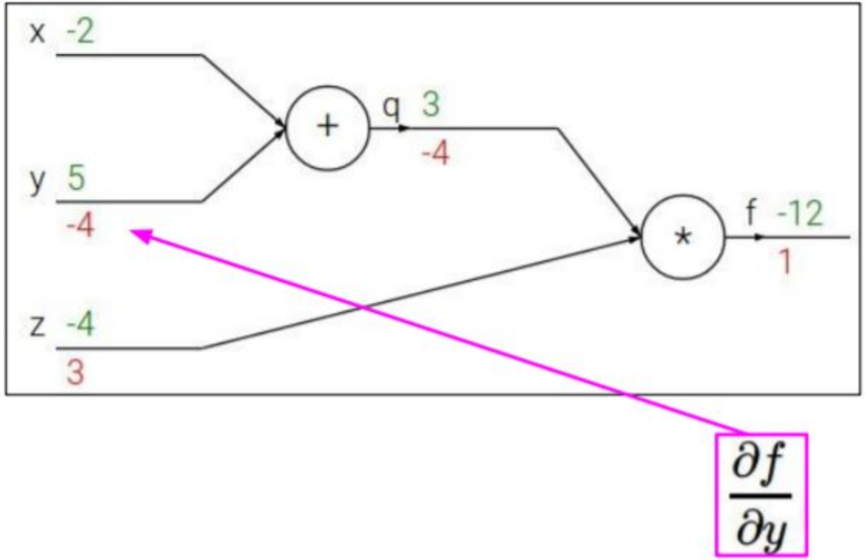
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

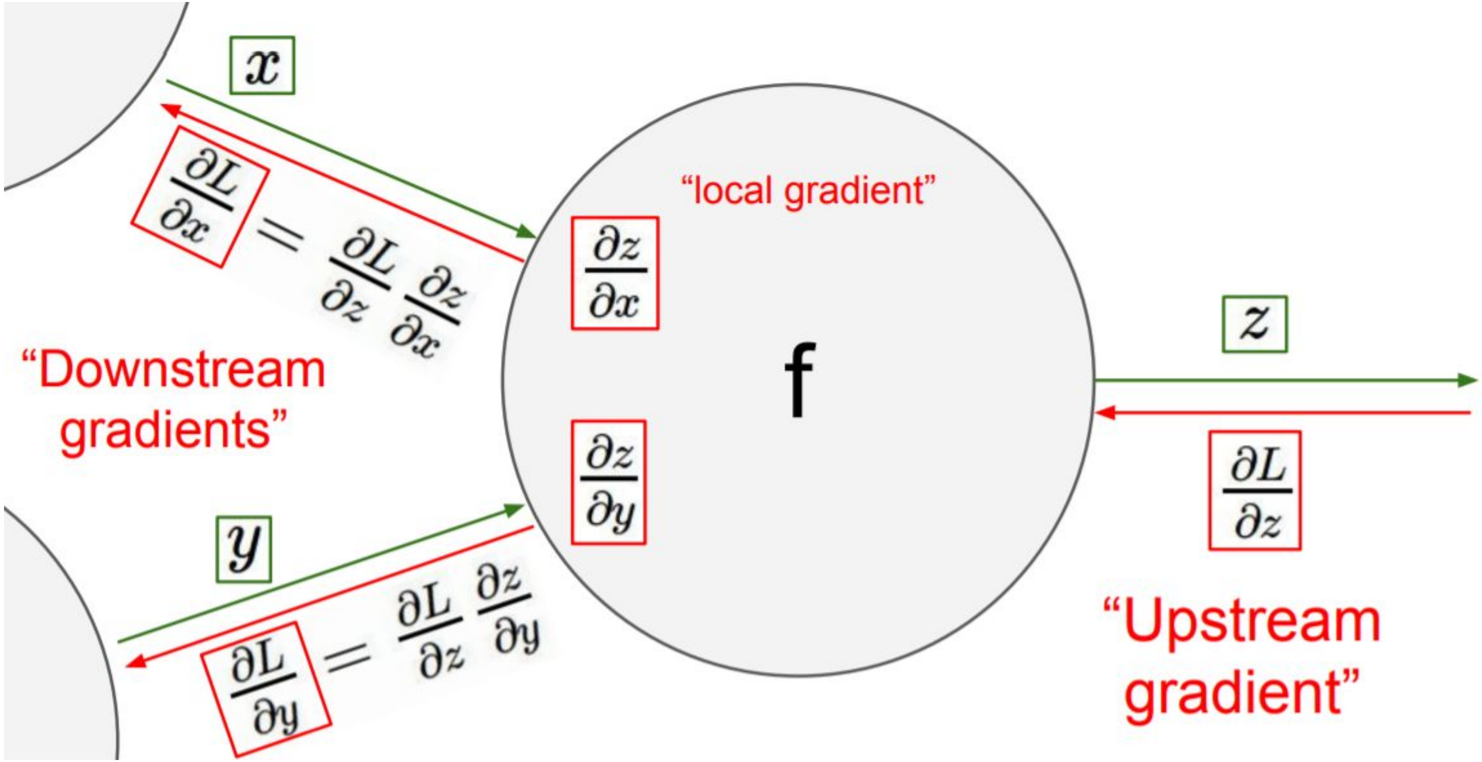
$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}}$$

Upstream gradient   Local gradient



UCLA

Back Propagation



Back Propagation

- Automatic Differentiation
  - Forward Pass
    - For each node in the computational graph (in topological order), assuming variable u_i and inputs v_1,...v_N:
      - Compute the u_i=g_i(v_1,...V_N) and cache the results.
  - Backward Pass
    - Calculate all local gradients.
    - For each node in the computational graph (in reverse topological order), for variable u_i=g_i(v_1,...V_N):
      - Use chain rule to calculate dy/dv_j=(dy/du_i)(du_i/dv_j)

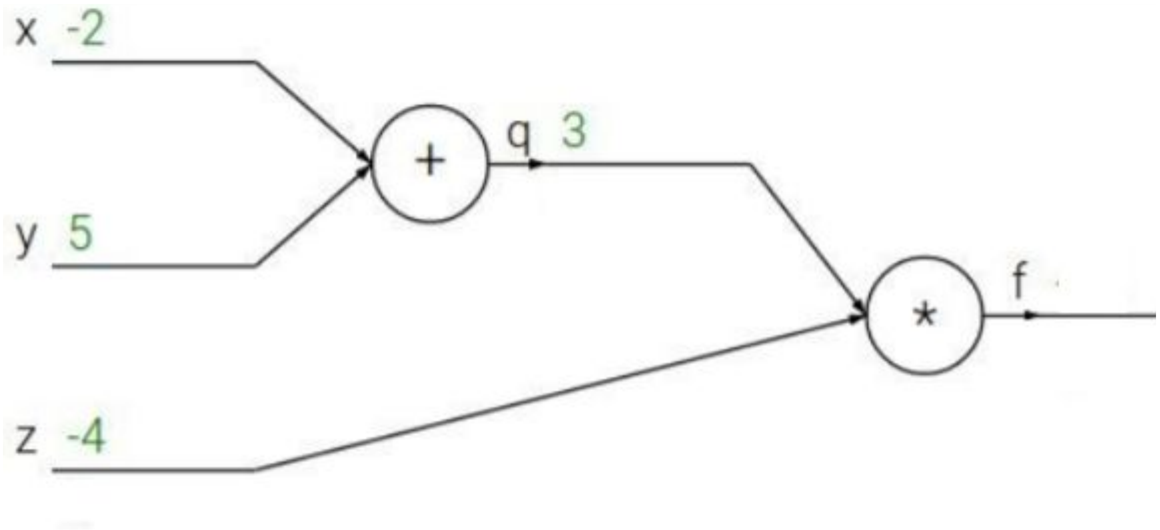
Back Propagation

- Use a computational graph to calculate the gradient of f with respect to x_i and w_i

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

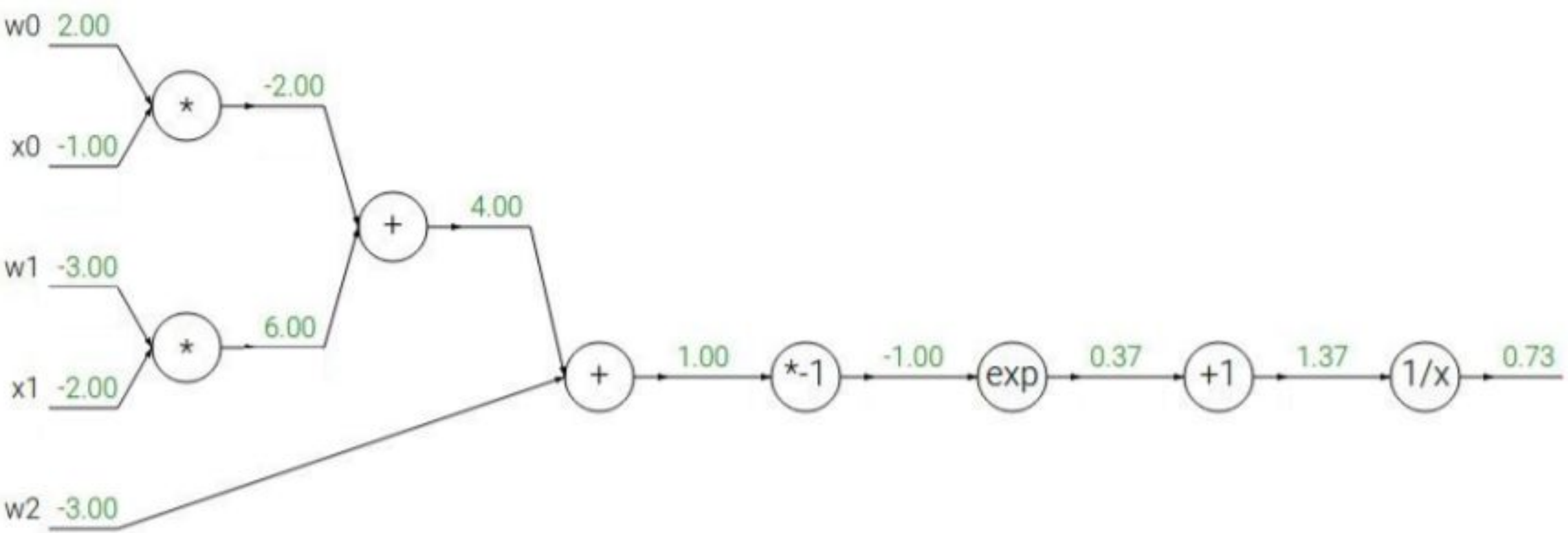
Back Propagation

- Let's do one example ...
  - Compute the forward pass in this computational graph



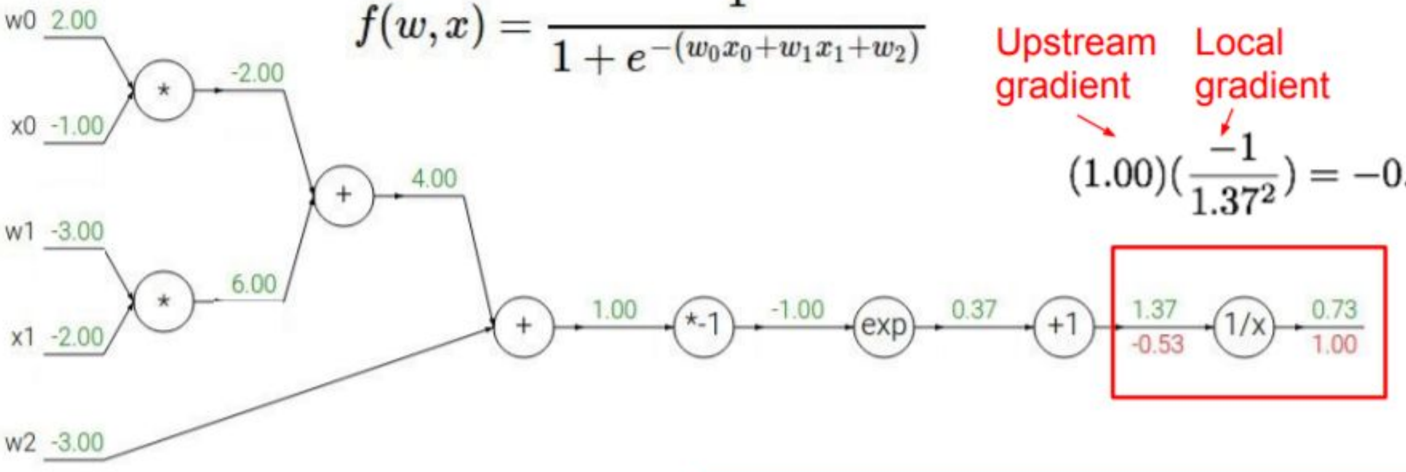
Back Propagation

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Back Propagation

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Upstream gradient

Local gradient

$$(1.00)\left(\frac{-1}{1.37^2}\right) = -0.53$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

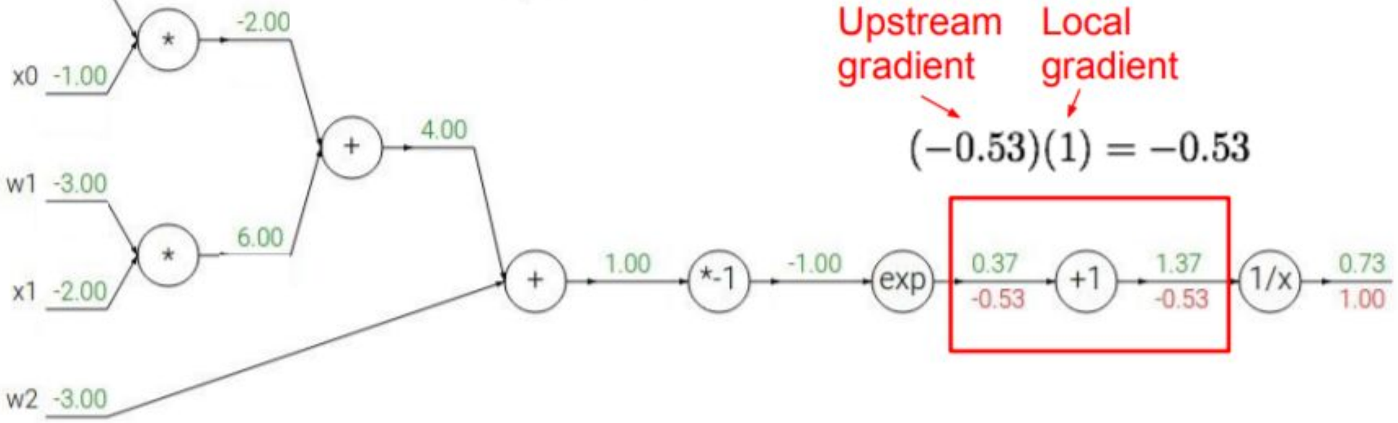
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$\boxed{f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2}$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

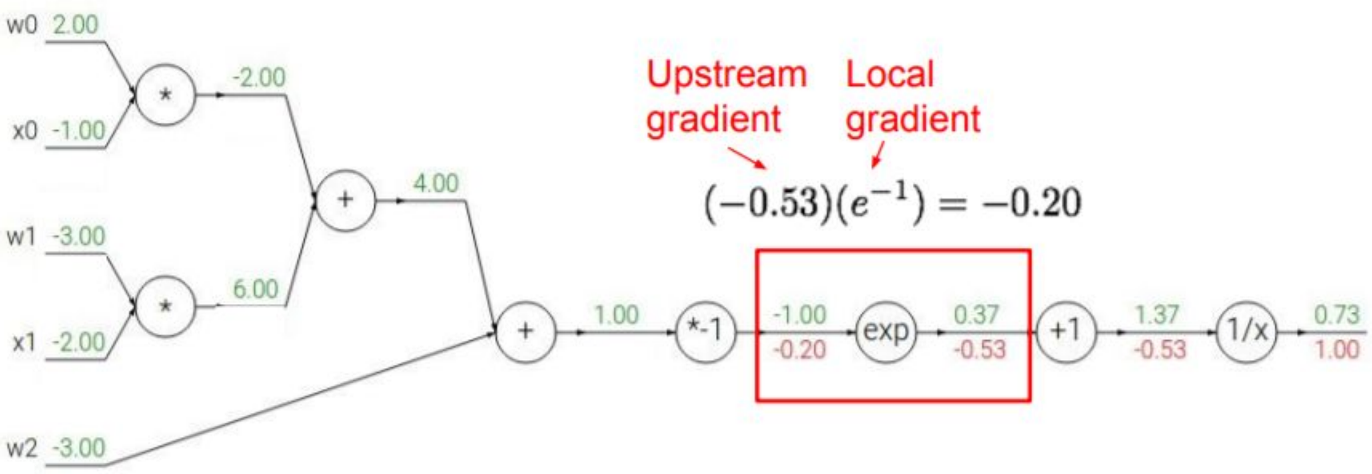
Back Propagation

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Back Propagation



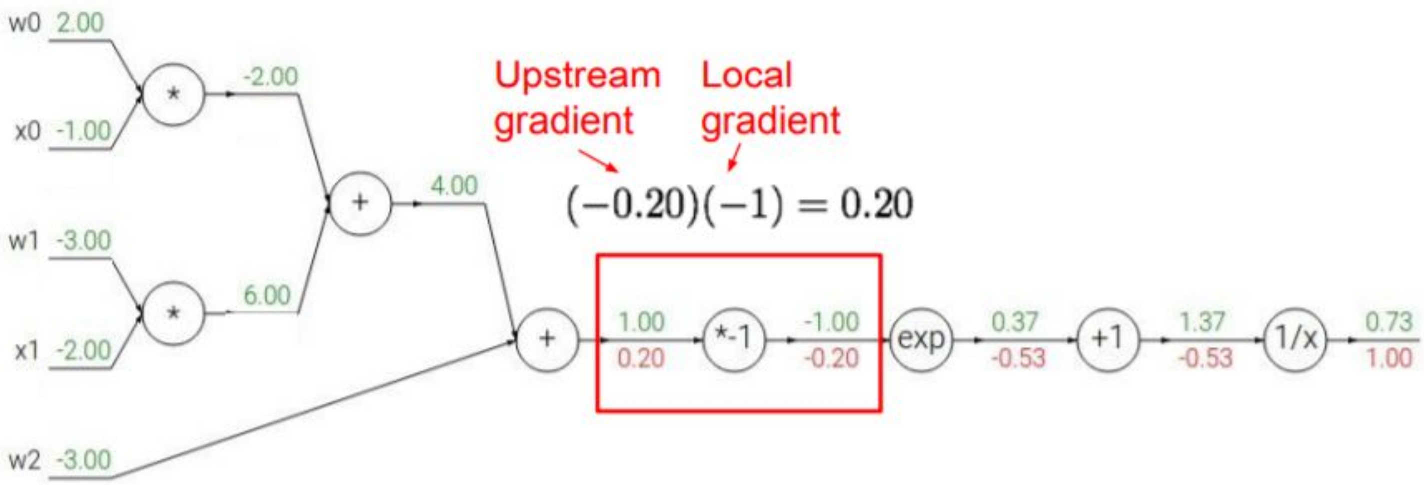
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Back Propagation



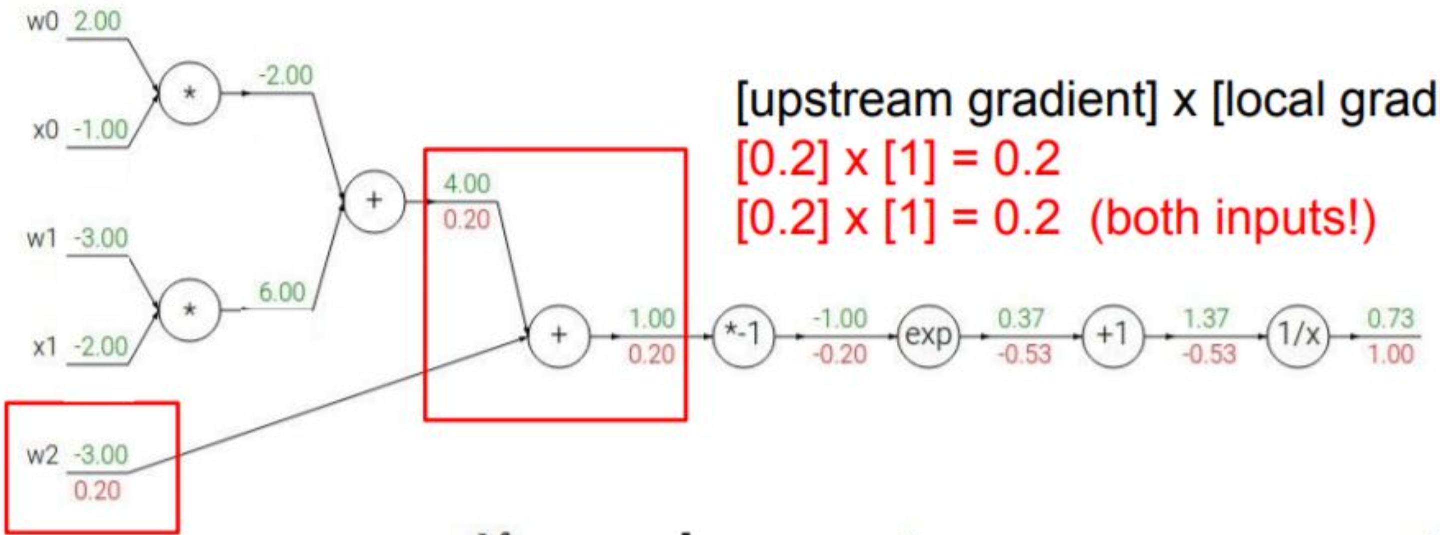
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Back Propagation



[upstream gradient] x [local gradient]

[0.2] x [1] = 0.2

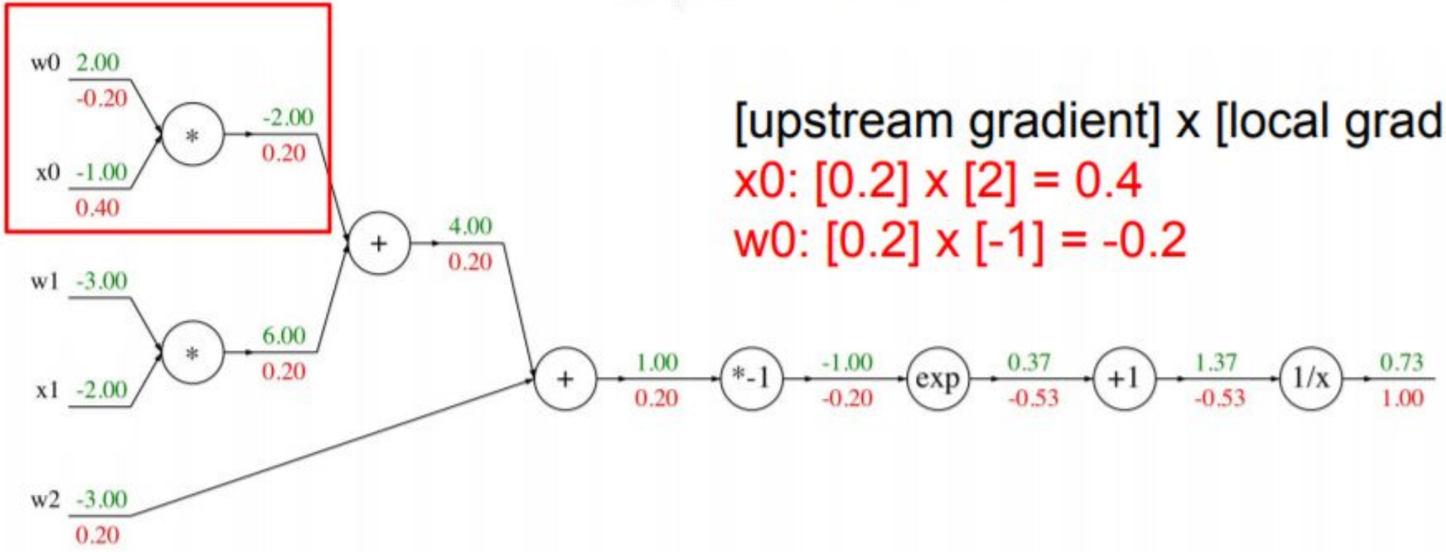
[0.2] x [1] = 0.2 (both inputs!)

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \qquad\qquad f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \qquad\qquad f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

UCLA

Back Propagation



[upstream gradient] x [local gradient]

x0: [0.2] x [2] = 0.4

w0: [0.2] x [-1] = -0.2

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Batch, Epoch, and Iteration

epoch: equivalent to the forward and backward processing of the entire training set

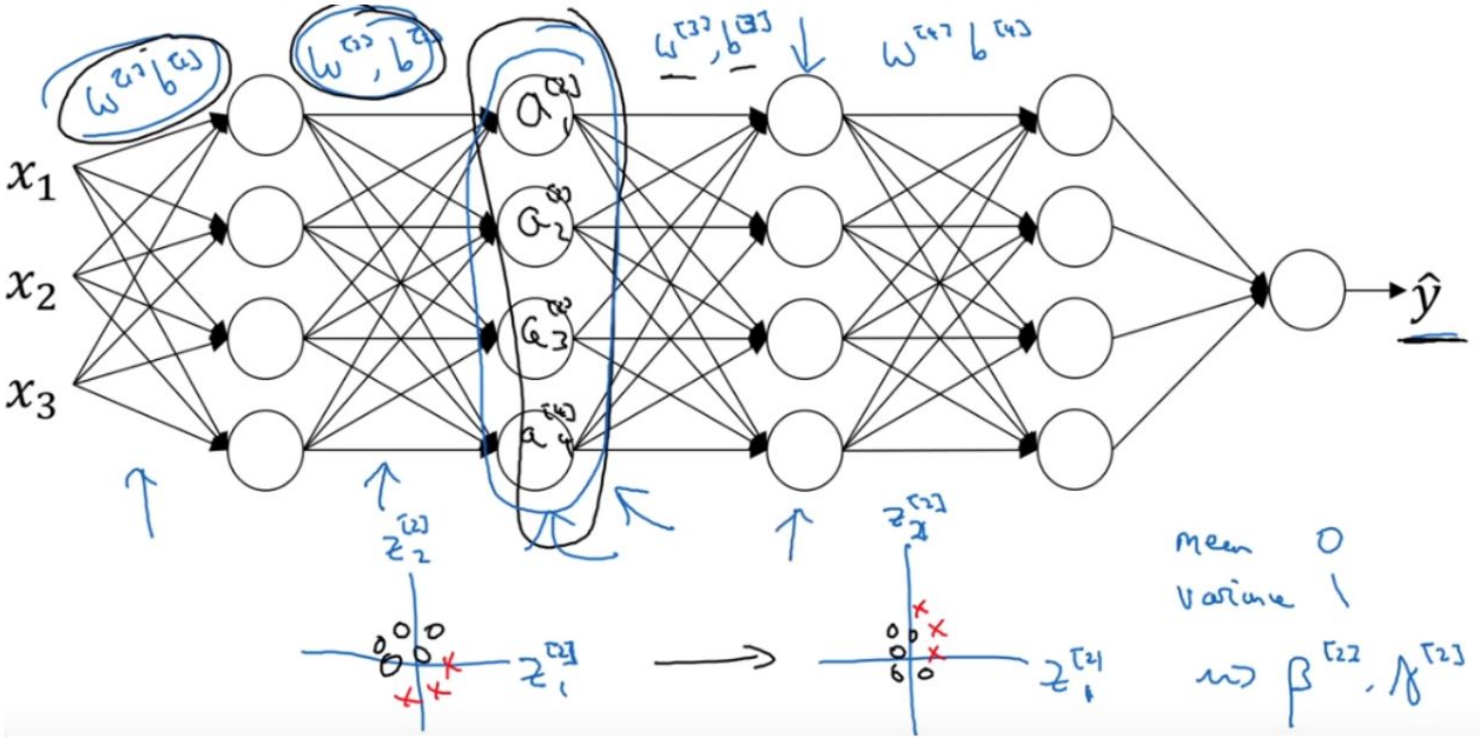
Batch: subset of training samples
Batch size: # of training samples in 1 batch

Iteration: # of batches to complete 1 epoch
(or # of passes to complete 1 epoch.
1 pass = 1 forward + 1 backward pass)

Batch Normalization

- Normalizing features to have mean of 0 and variance of 1 can speed up the learning process since input features now take on a similar range of values.
- Batch norm disseminates this idea by making the weights in later stages of a neural network to be less susceptible to changes.
- In simpler words, by making sure that the input to all layers have been normalized centered around zero, subsequent layers in a neural networks can be more effectively trained. As a result, it speeds up training.

Batch Normalization



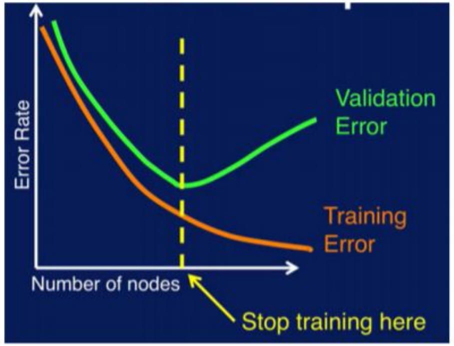
Splitting the Dataset

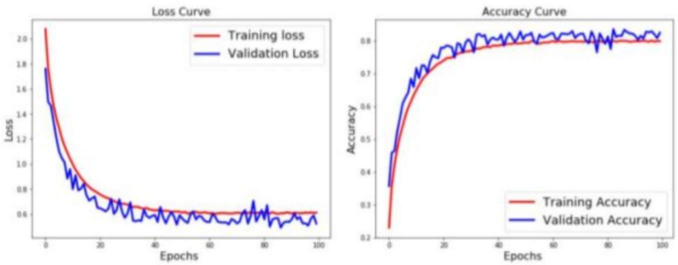
Training | Testing

(holdout sample)

Training | Validation | Testing

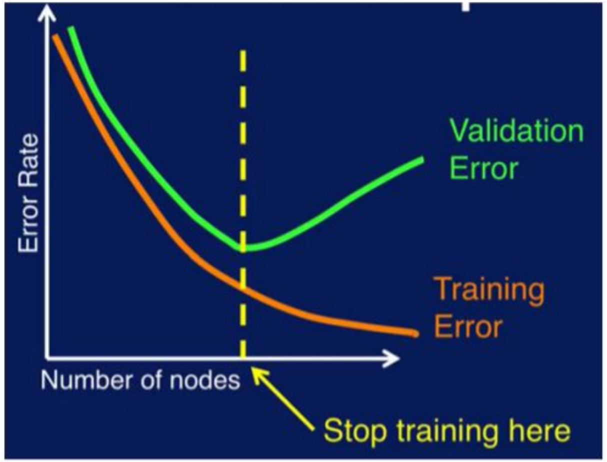
(validation holdout sample) | (testing holdout sample)





Overfit vs Underfit

- Underfitting happens when the model does not fit the data well enough
  - The model is often too simple such that it cannot capture the trend of the data
- Overfitting happens when the model fits the data too well
  - The model essentially has also learned the noise of the data
  - The model suffers from lack of generalizability



UCLA

K-fold Cross Validation

- Algorithm:

Iteration 1	Test	Train	Train	Train	Train
Iteration 2	Train	Test	Train	Train	Train
Iteration 3	Train	Train	Test	Train	Train
Iteration 4	Train	Train	Train	Test	Train
Iteration 5	Train	Train	Train	Train	Test

1. Take the group as a holdout or test data set
2. Take the remaining groups as a training data set
3. Fit a model on the training set and evaluate it on the test set
4. Retain the evaluation score and discard the model

Vanishing Gradient

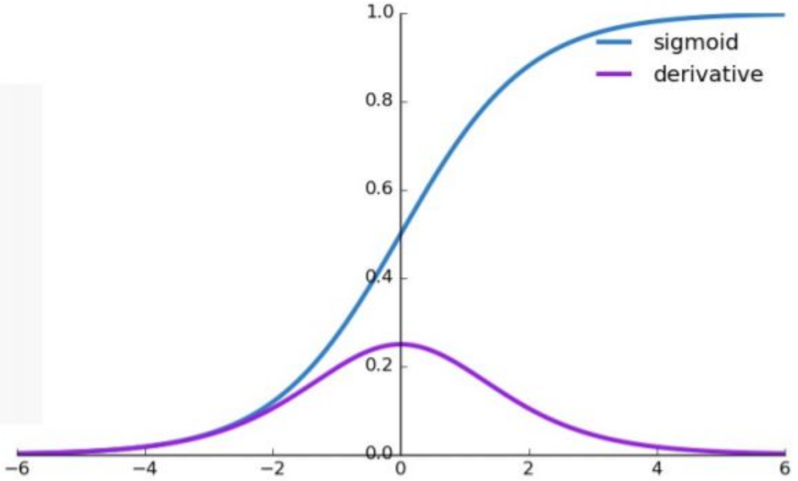
- Activation functions such as sigmoid map their input into a small region(e.g. 0 and 1). This means that large changes in the input would cause a small change in the output and you can expect that the gradient would be small accordingly.

The sigmoid function is defined as follows

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

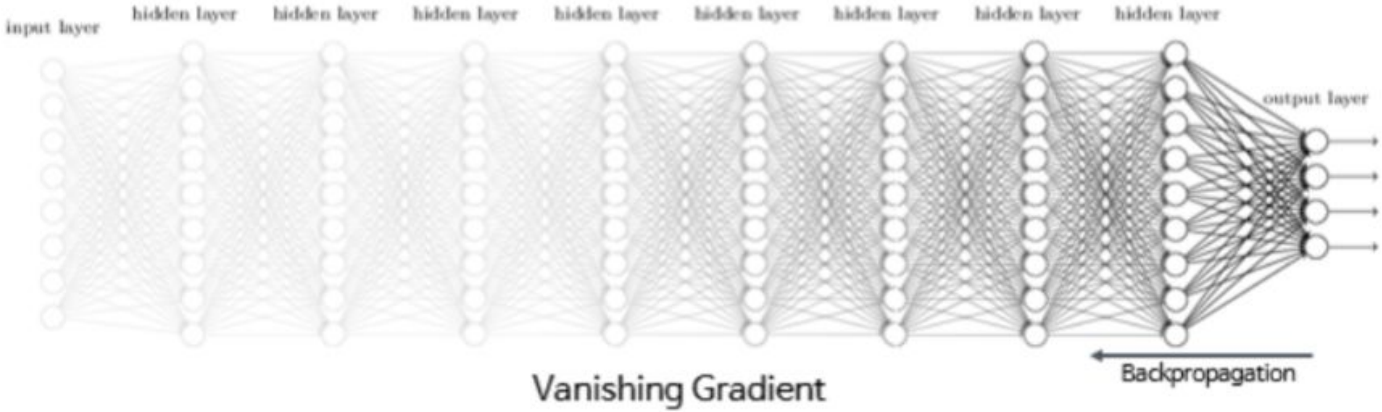
This function is easy to differentiate because

$$\frac{d\sigma(x)}{d(x)} = \sigma(x) \cdot (1 - \sigma(x)).$$



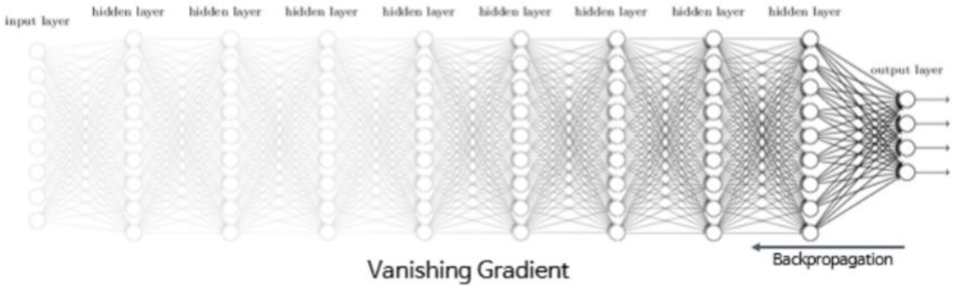
Vanishing Gradient

- When you stack multiple of such activations on top of each other, the problem intensifies more, because the large input space is now mapped into a much smaller output space and this results in even more smaller gradients.



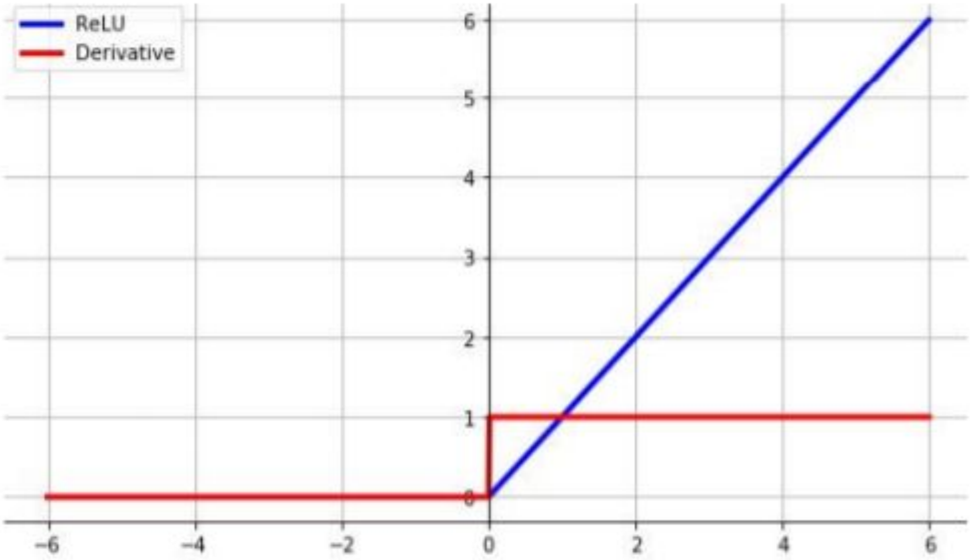
Vanishing Gradient

- Backpropagating the error in this case results in the values of gradients to become very small (vanishing)
- When vanishing gradient occurs, the gradient reaches a very small value when by the time it reaches the first layers
- Thus we can't effectively change the weights according to our learning rule.



Solutions to Vanishing Gradient

- Relu activation function
- Batch normalization
- Residual Connections



Residual Learning

- Simple Identity block

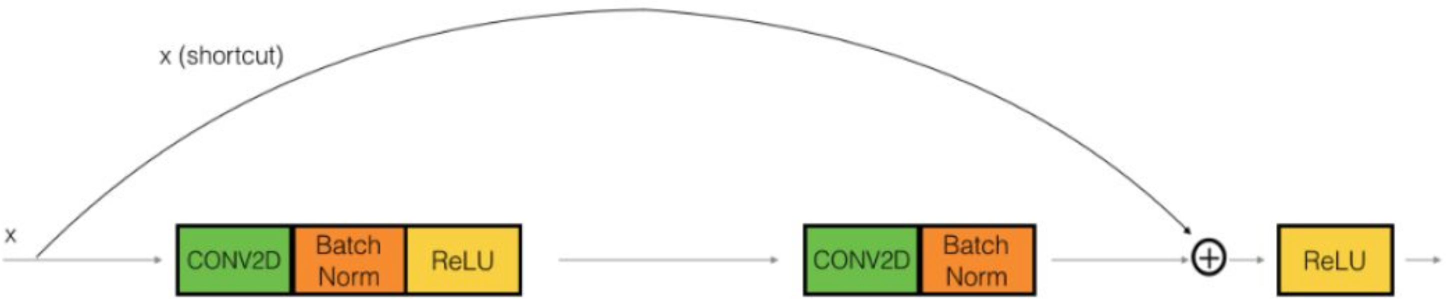
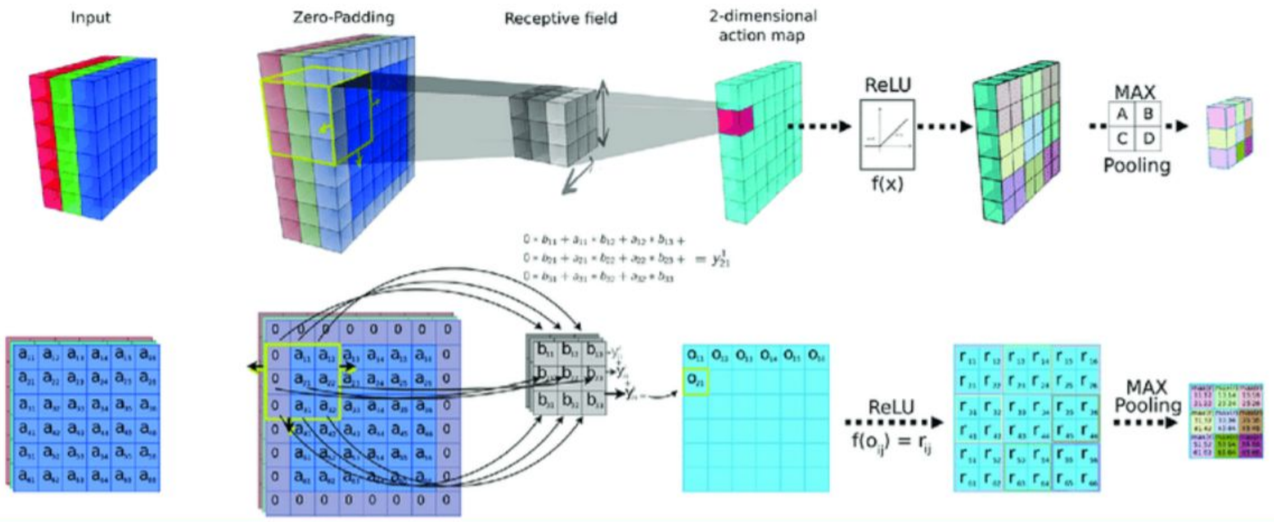


Figure 3: Simple identity block.

Convolutional Layer

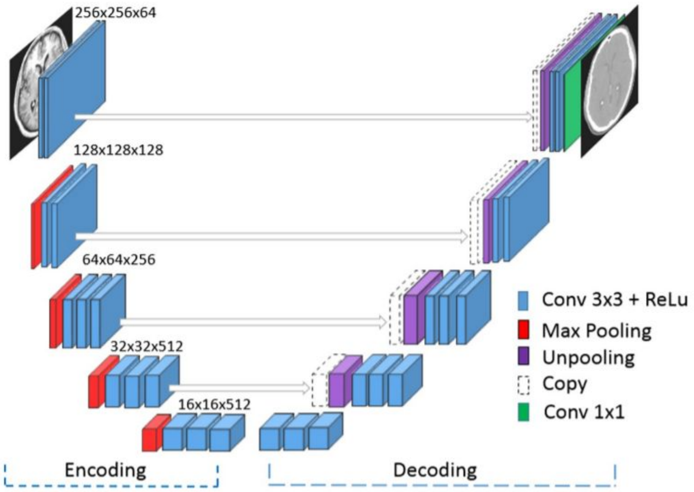
Convolution layer

- Parametrized by: height, width, depth, stride, padding, number of filters, type of activation function



Fully Convolutional NNs

- In a fully convolutional neural networks, we only leverage convolutional layers in the architecture (e.g. no dense layer)
- Fully convolutional neural networks are commonly used for applications such as image segmentation.



Medical Physics, Volume: 44, Issue: 4, Pages: 1408-1419, First published: 13 February 2017, DOI: (10.1002/mp.12155)

Thank you!