Week 7

CS 188.2 - Introduction to Computer Vision

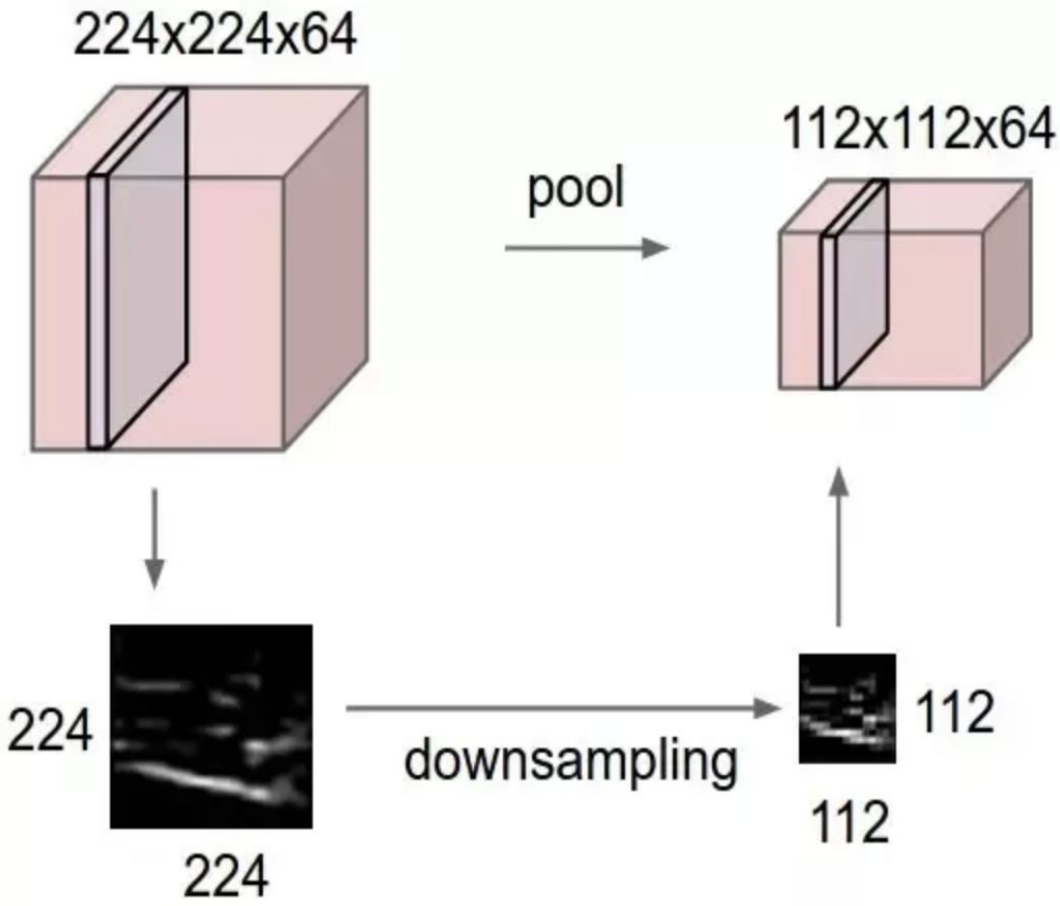
Topics

- Max Pooling
- Average Pooling
- Max Unpooling
- Transposed Convolution
- Dilated Convolution
- Commonly used terms in Deep Learning
- Training, Validation and Test Sets
- LOOCV, K-fold
- Backpropagation
- Batch Normalization
- Demo

Max Pooling

- Downsampling
- We pool because we want to reduce dimensions of the data and keep relevant information
- Make representations smaller and more manageable
  - Fewer parameters in the end
  - Invariance over transformations
- Takes input volume and spatially downsamples it
- Does not do anything in depth
  - Only pooling spatially
- Hyperparameters are filter size and stride
- Increasing the stride reduces the dimension of the output.

Real-life example:



Solve : Apply max pooling with different hyperparameters

An example Image Portion
for Max Pooling
Numbers represent
the pixel values

2	3	4	0
1	5	3	2
0	4	2	3
1	0	6	1

2*2, stride = 2 → ?

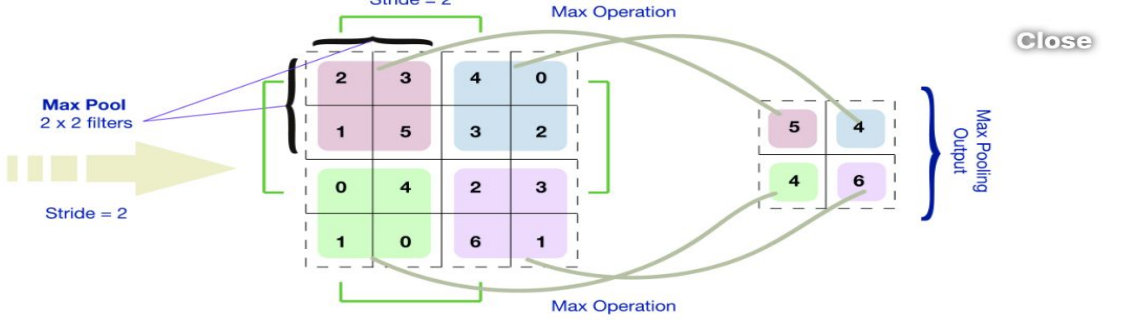
3*3, stride = 1 → ?

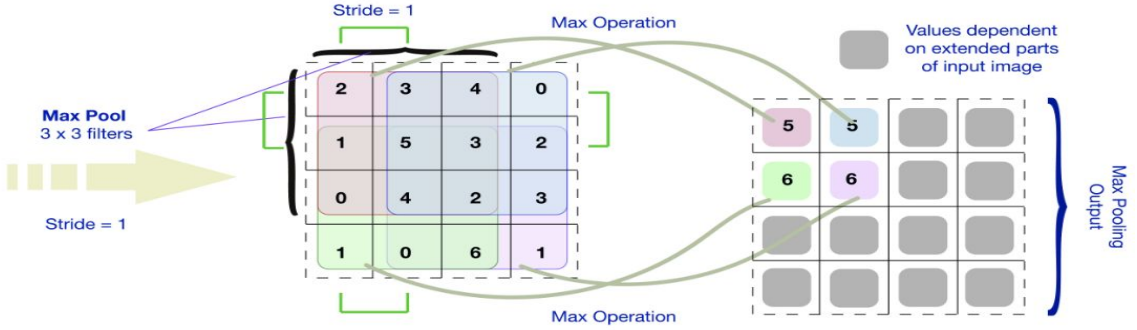
3*3, stride = 2 → ?

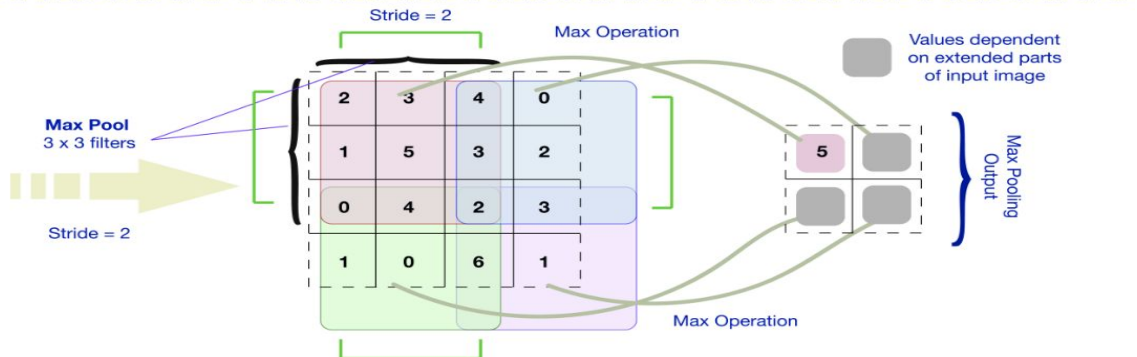
Solution

An example Image Portion
for Max Pooling
Numbers represent
the pixel values

2	3	4	0
1	5	3	2
0	4	2	3
1	0	6	1







Solve: Apply both max pooling and average pooling (2*2 filter, stride = 2) on this convolution output

Convolution Output

55	20	87	46	32	59	76	58
58	25	61	29	34	79	94	15
29	35	71	63	94	73	12	17
58	22	41	98	32	45	16	76
92	5	16	0	46	84	6	7
38	4	86	97	64	13	15	48
32	19	8	42	45	1	0	56
6	0	5	8	66	44	72	8

Solution

Max Pooling

58	87	79	94
58	98	94	76
92	97	84	48
32	42	66	72

Mean Pooling

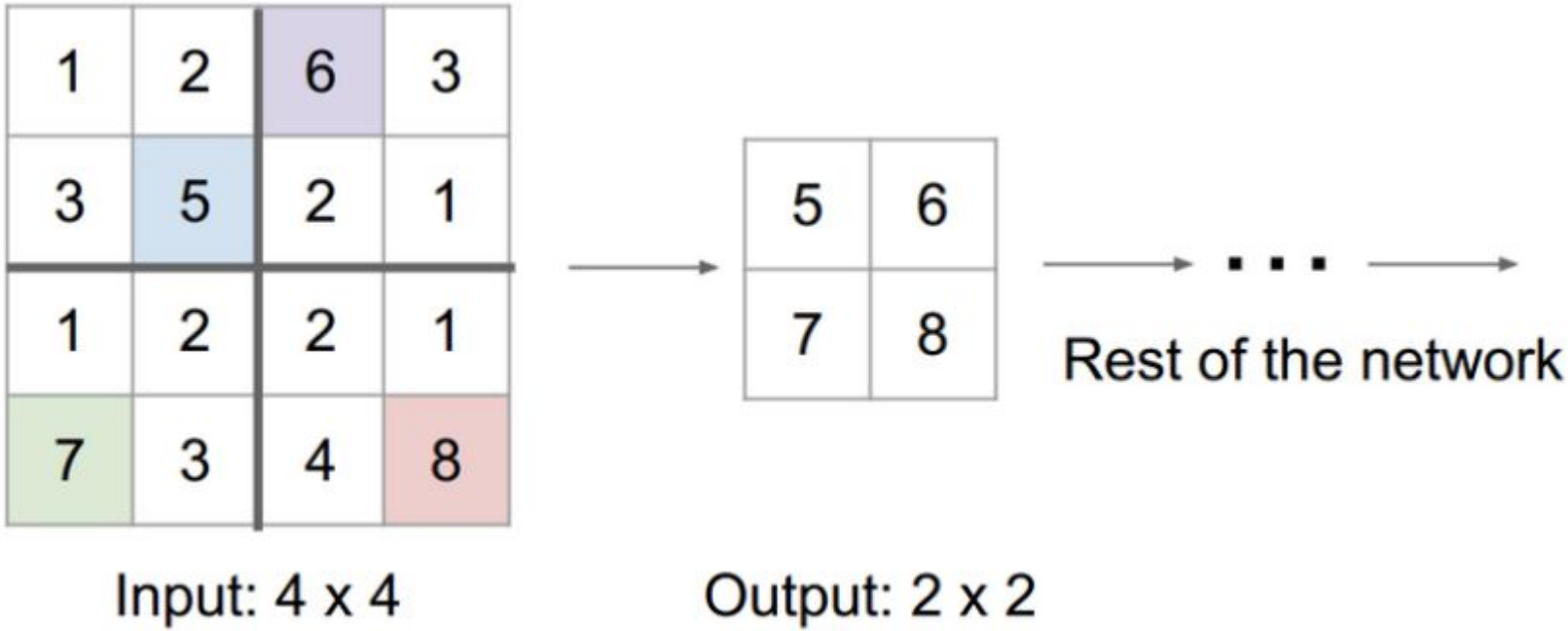
40	56	51	61
36	68	61	30
35	50	52	19
14	16	39	34

Max Unpooling

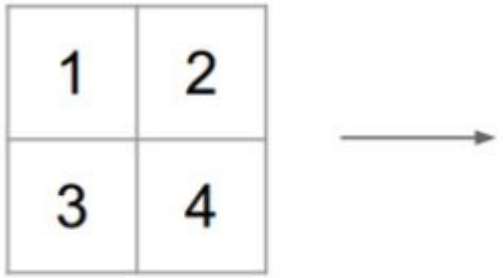
- We want to bring back the original resolution of the image
- We want to get rid of the unnecessary information (during pooling) such that, while reconstructing,we successfully rid ourselves of unnecessary information
- Loss of information since we ‘generalize’
- Can we incorporate this as a part of the learning?
    - Motivation for transposed convolution

Solve this question

A 4*4 input is max-pooled in the beginning layers like:



After passing through multiple layers, it looks like the 2*2 input shown below. This is then unpooled to get the output. What is the output?



Input: 2 x 2

Solution

0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

Output: 4 x 4

Transposed Convolution

- When we are computing convolution, we reduce information so we have a smaller result
- Here, in transposed convolution, we go from the result and want to recover what the input image was
- We use a filter to blow up the result to get the input image

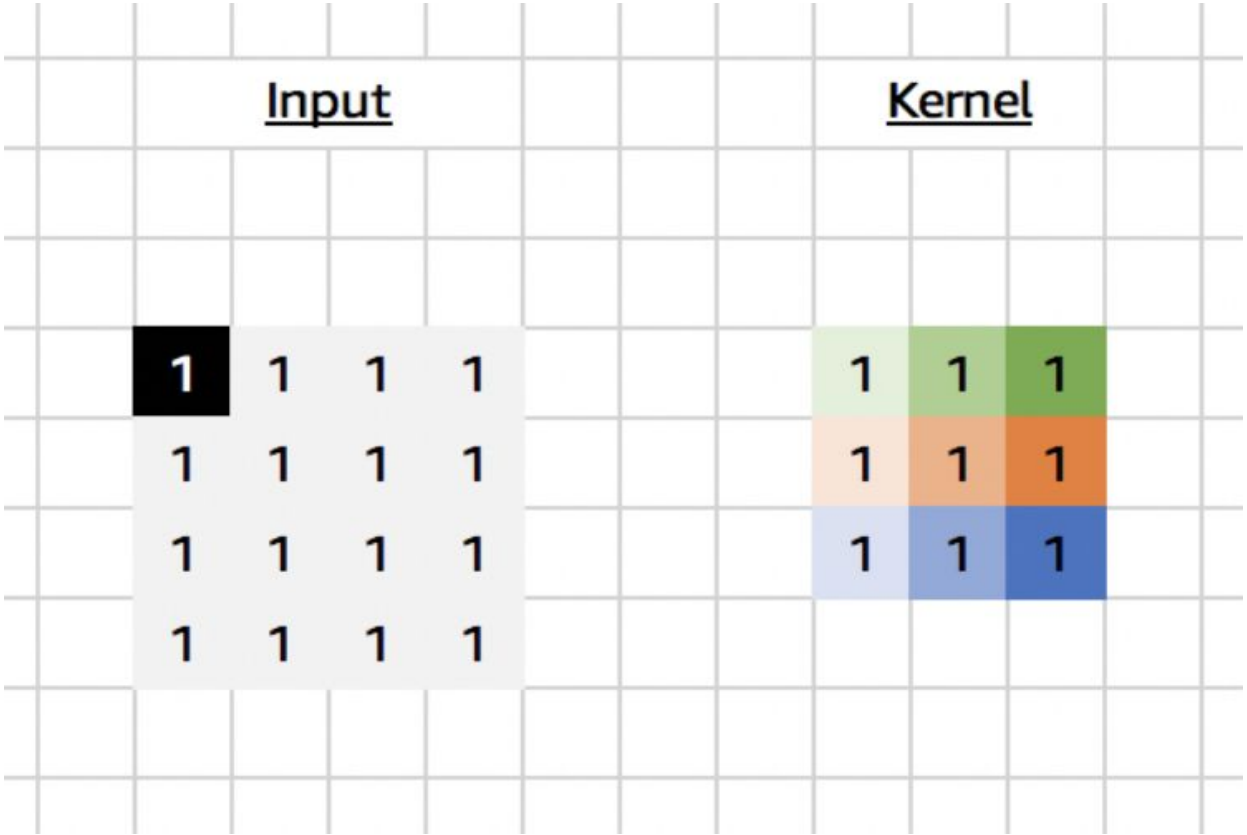
```
Image ---> convolution ---> Result
```

```
Result ---> transposed convolution ---> "originalish Image"
```

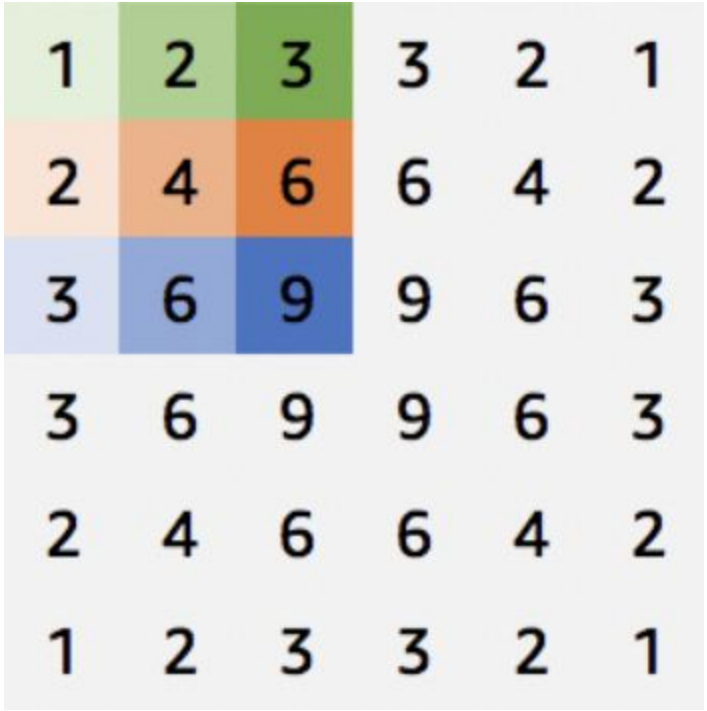
Transposed Convolution continued..

- Consider an image of 5x5 is fed into a convolutional layer. The stride is set to 2, the padding is deactivated and the kernel is 3x3. This results in a 2x2 image.
- If we wanted to reverse this process, we use transposed convolution
- The only thing it guarantees is that the output will be a 5x5 image (which is what we expect), while still performing a normal convolution operation.
- To achieve this, we need to perform some fancy padding on the input.

Perform a Conv2DTranspose with 3x3 kernel applied to a 4x4 input to give a 6x6 output



Solution


1 2 3 3 2 1
2 4 6 6 4 2
3 6 9 9 6 3
3 6 9 9 6 3
2 4 6 6 4 2
1 2 3 3 2 1

Perform a Conv2DTranspose with 3x3 kernel applied to a 4x4 input to give a 6x6 output

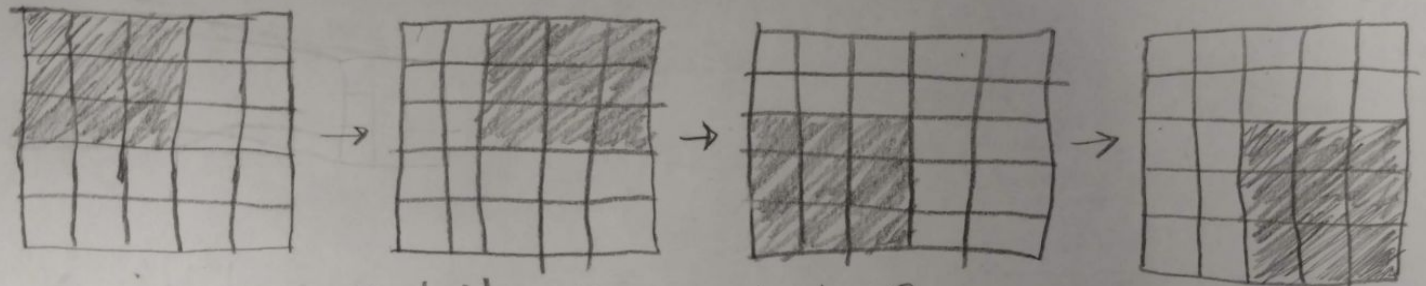
Solution

1	5	11	14	8	3
1	6	15	18	12	3
4	13	21	21	15	11
5	17	28	27	25	11
4	7	9	12	8	6
6	7	14	13	9	6

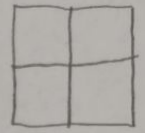
Dilated Convolution

- The more important point is that **the architecture is based on the fact that dilated convolutions support exponential expansion of the receptive field without loss of resolution or coverage**.
- Allows one to have **larger receptive field** with **efficient (same) computation and memory costs** while also **preserving resolution**.
- It **preserves the resolution/dimensions of data** at the output layer. This is because the layers are dilated instead of pooling, hence the name *dilated convolutions*.
- It **maintains the ordering of data**. For example, in 1D dilated causal convolutions when the prediction of output depends on previous inputs then the structure of convolution helps in maintaining the ordering of data.

Strided Convolution:

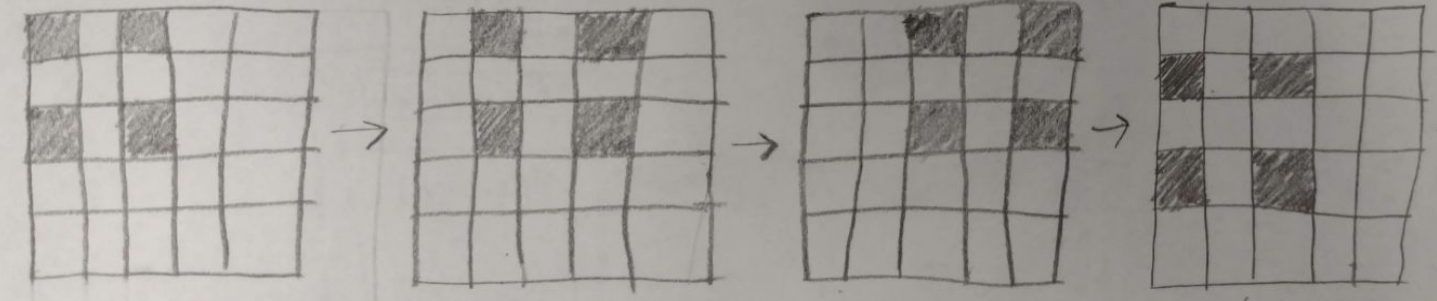


Output size:

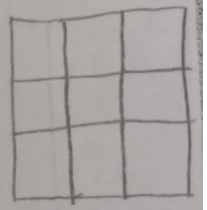


x_stride = 2, y_stride = 2, kernel_size = 3, dilated = False

Dilated Convolution (Atrous Convolution)



Output size:



x_stride = 1, y_stride = 1, kernel_size = 2 dilated = True

Solve: Given the image below and the 2*2 Kernel. Consider strides along x and y dimension to be 1, 1 and dilation to be set to True (D = 2). Find the dilated convolution's result. Assume Padding set to False.

$$Image = \begin{bmatrix} 1 & 4 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 5 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad \text{Kernel} = \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix}$$

Solution

- Using a dilated convolution increases the size of the receptive field relative to the kernel size. In my sketch, a 2x2 dilated convolution has the same receptive field as a 3x3 un-dilated convolution.

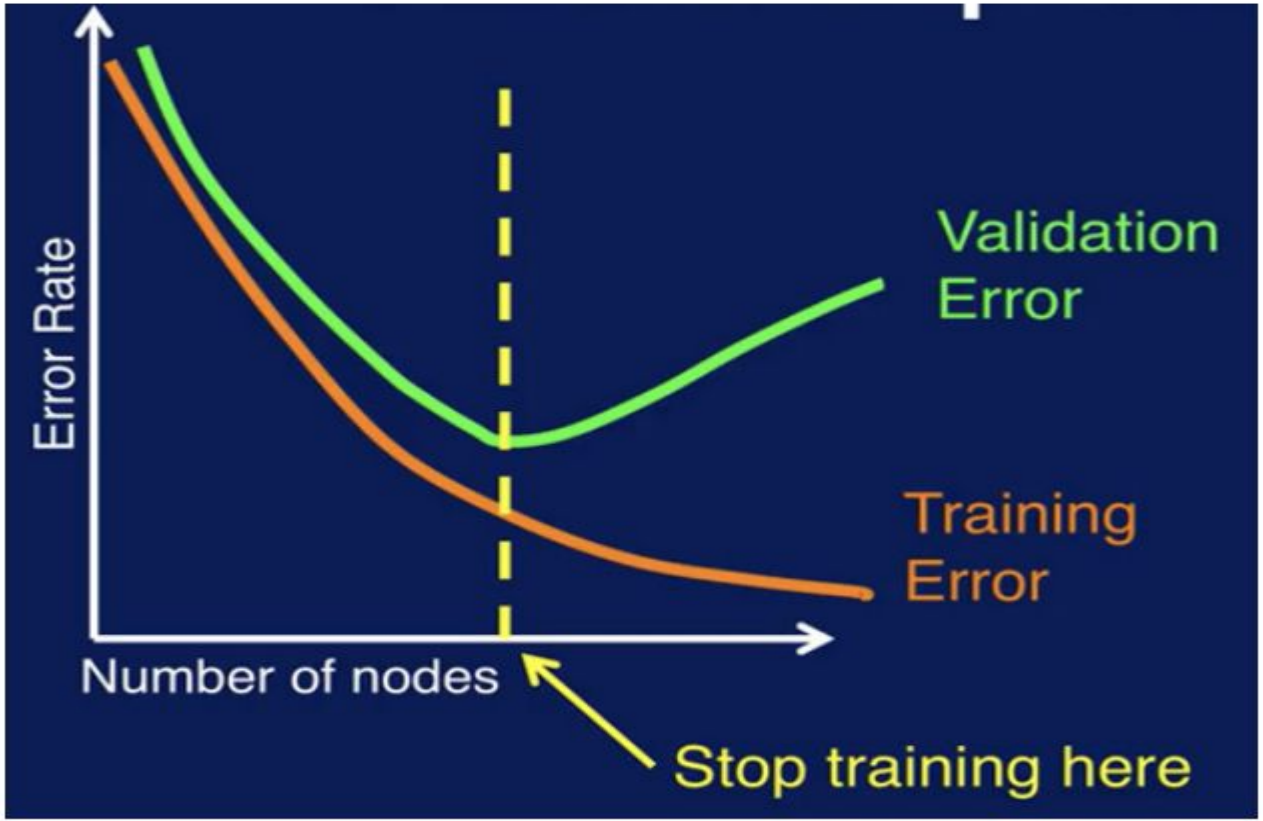
$$\text{Solution} = \begin{bmatrix} 25 & 35 & 33 \\ 17 & 17 & 17 \\ 37 & 17 & 41 \end{bmatrix}$$

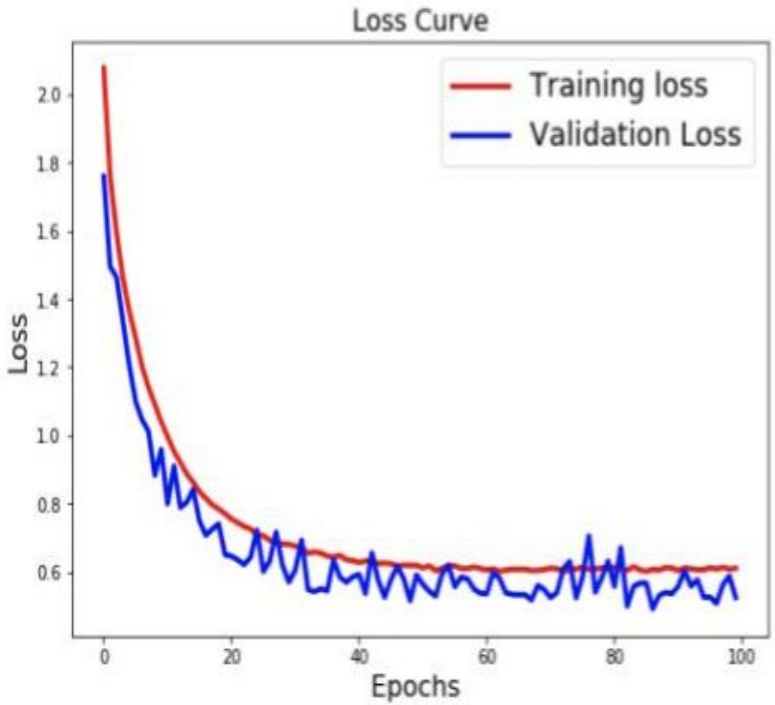
Terms in Deep Learning model

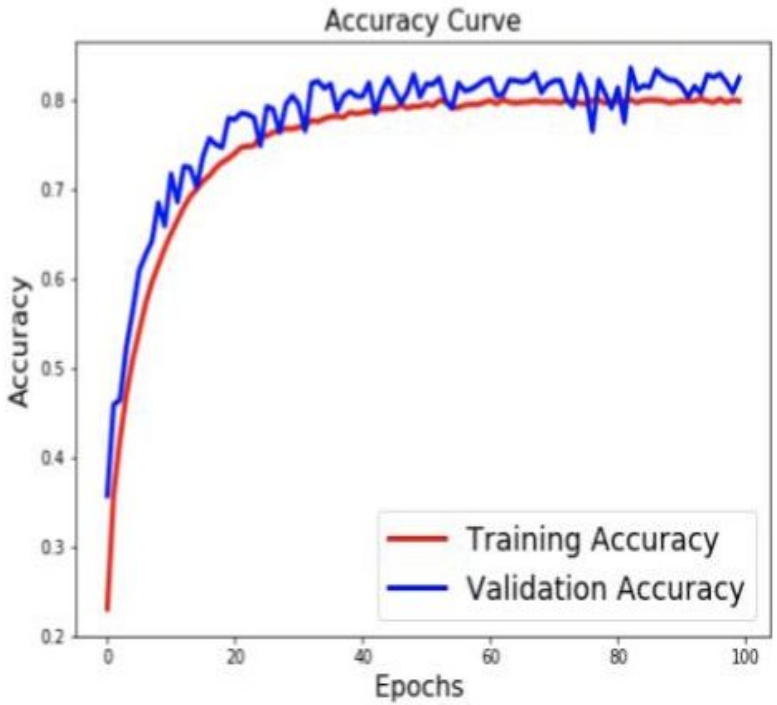
- Batch Size
  - # of batches to complete one epoch (fwd + back on many batches)
  - You need to decide how many samples you want to use
- Epoch
  - Applying forward and backward processing of the entire training set
  - Provides empirical risk at every epoch
- Iterations
  - For one iteration, we use one subset of the data
  - We call that subset - a batch which has a certain number of samples (input and output)
  - Why? - to use memory efficiently
    - Tough to compute forward and backward pass for the ENTIRE dataset
    - The values are replicated multiple times while forward
      - Memory intensive
- Example: If you have 2000 data points, with each batch has 500, we need to do 4 iterations to complete 1 epoch

Evaluate how well your dataset works

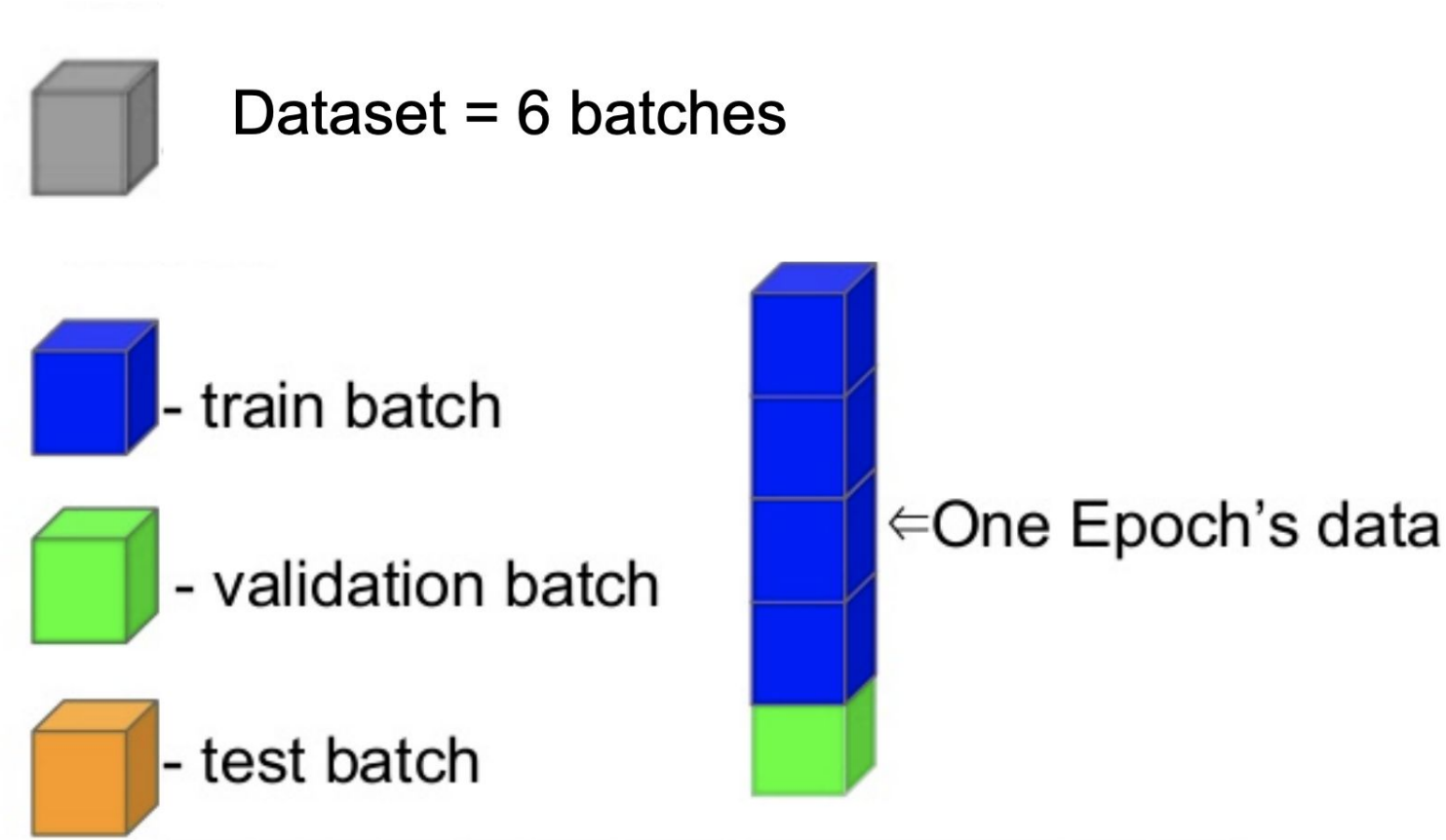
- Require a test set to evaluate how well your model works
  - Cannot use the test set to train your model - “cheating”
- Because it takes so long (large training set), we don’t want to wait to know if our model works
  - Training set (gradient descent) + Validation Set (Not used for training : while being trained [iterations happening], can see how model is performing on validation set; should be close to how it will perform on test set) + Test Set (Untouched until model trained)
  - Is the loss at each epoch reducing?
    - Can plot training loss (on training set, what is the loss for every data point)
      - If it is decreasing, it is good; learning something
    - Can plot validation loss


Error Rate
Validation Error
Training Error
Number of nodes
Stop training here


Loss Curve
Training loss
Validation Loss
Loss
Epochs
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0
20
40
60
80
100

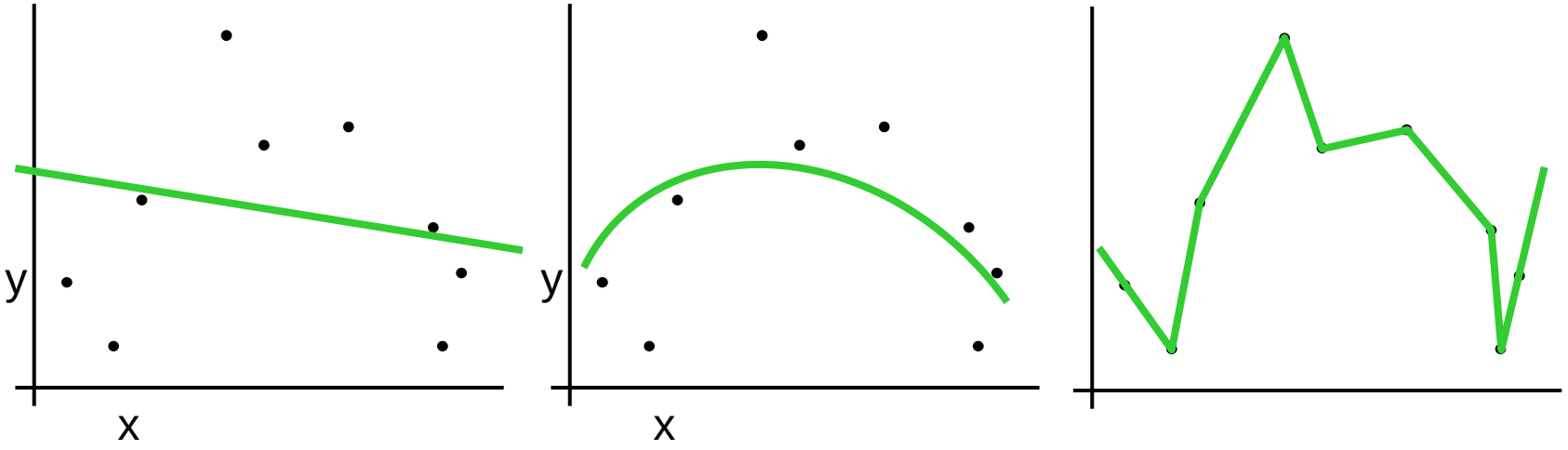

Accuracy Curve
Training Accuracy
Validation Accuracy
Accuracy
Epochs
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
100

Batch, Epoch, and Iteration



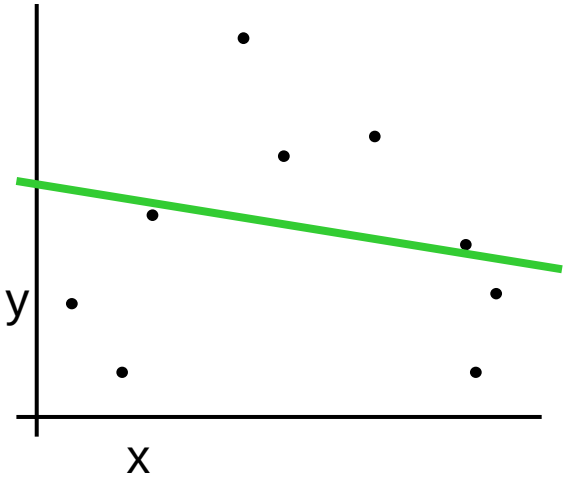
K fold Cross Validation

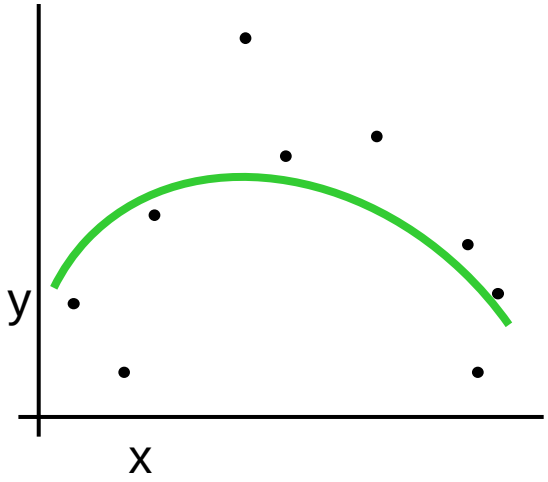
Which is best?

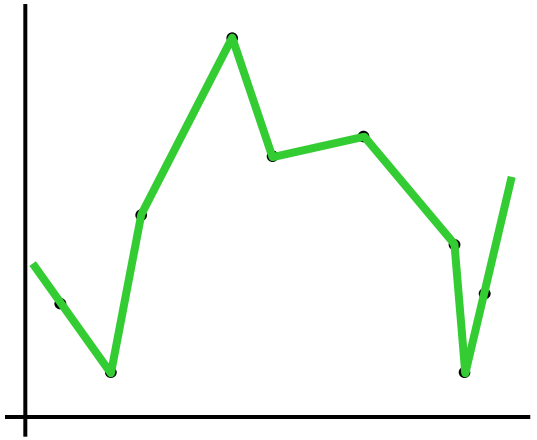


Why not choose the method with the best fit to the data?

What do we really want?



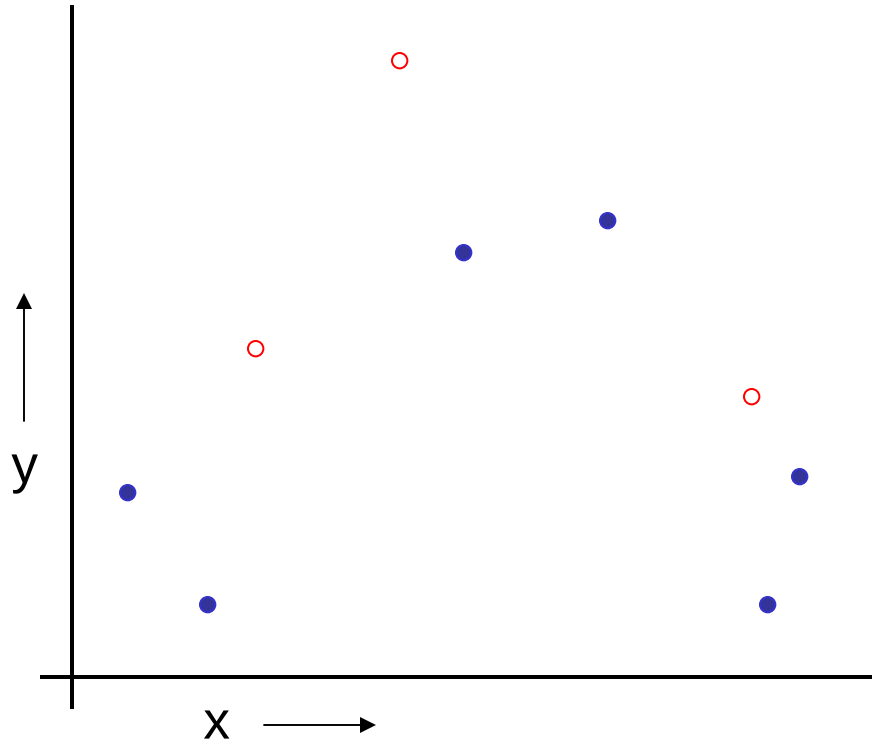




Why not choose the method with the best fit to the data?

"How well are you going to predict future data drawn from the same distribution?"

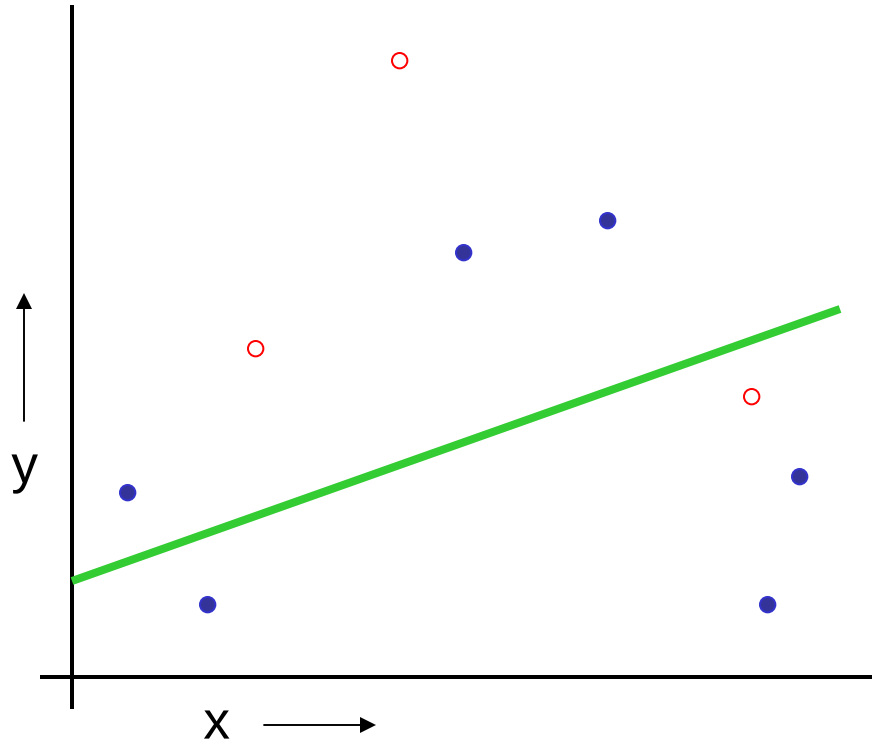
The test set method



1. Randomly choose 30% of the data to be in a test set

2. The remainder is a training set

The test set method



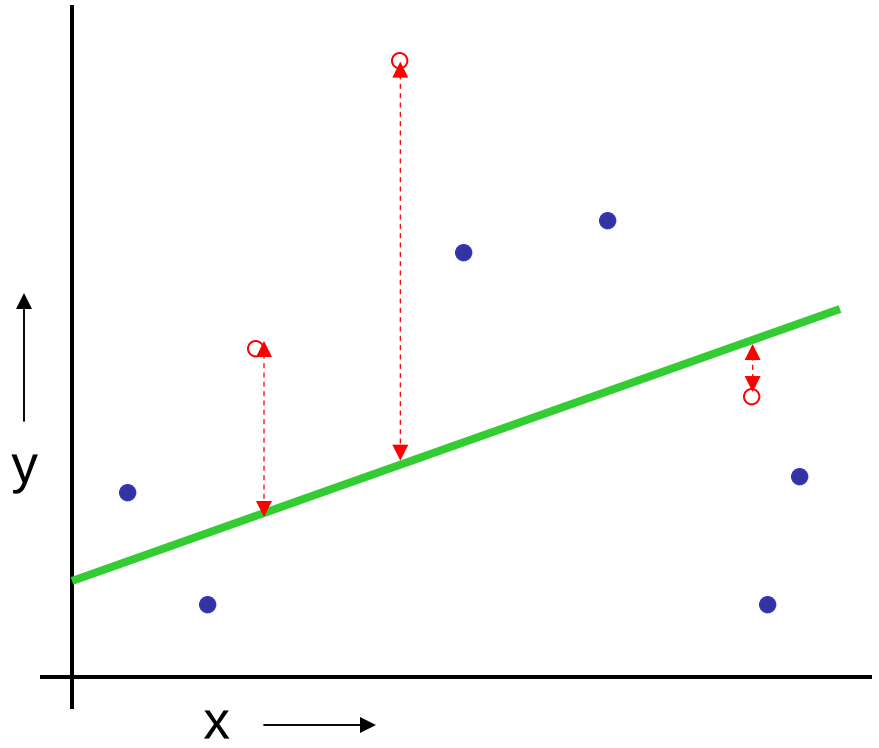
(Linear regression example)

1. Randomly choose 30% of the data to be in a test set

2. The remainder is a training set

3. Perform your regression on the training set

The test set method



(Linear regression example)

Mean Squared Error = 2.4

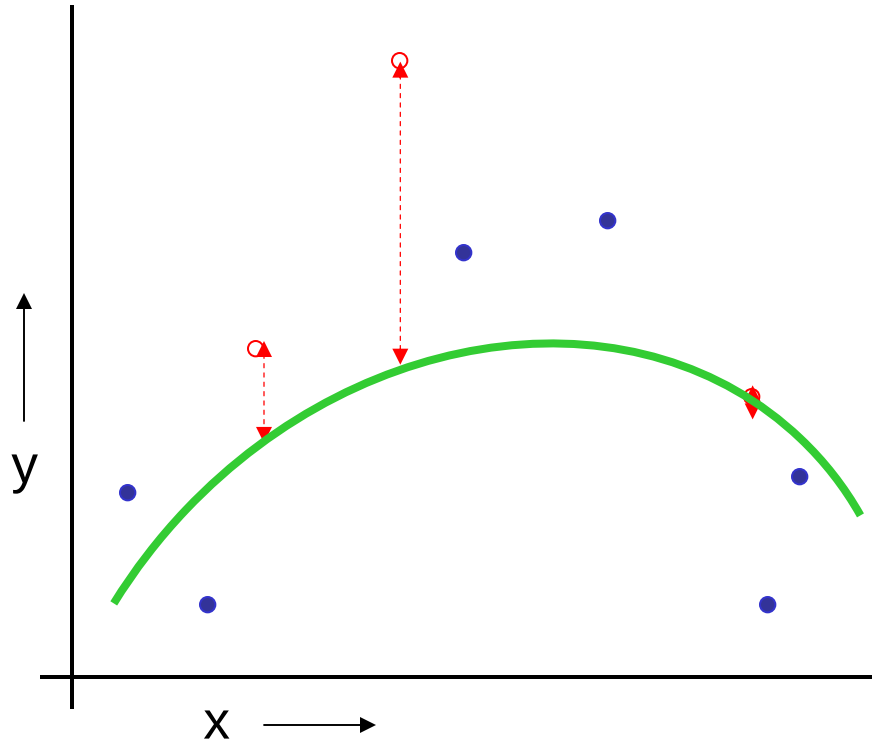
1. Randomly choose 30% of the data to be in a test set

2. The remainder is a training set

3. Perform your regression on the training set

4. Estimate your future performance with the test set

The test set method



(Quadratic regression example)

Mean Squared Error = 0.9

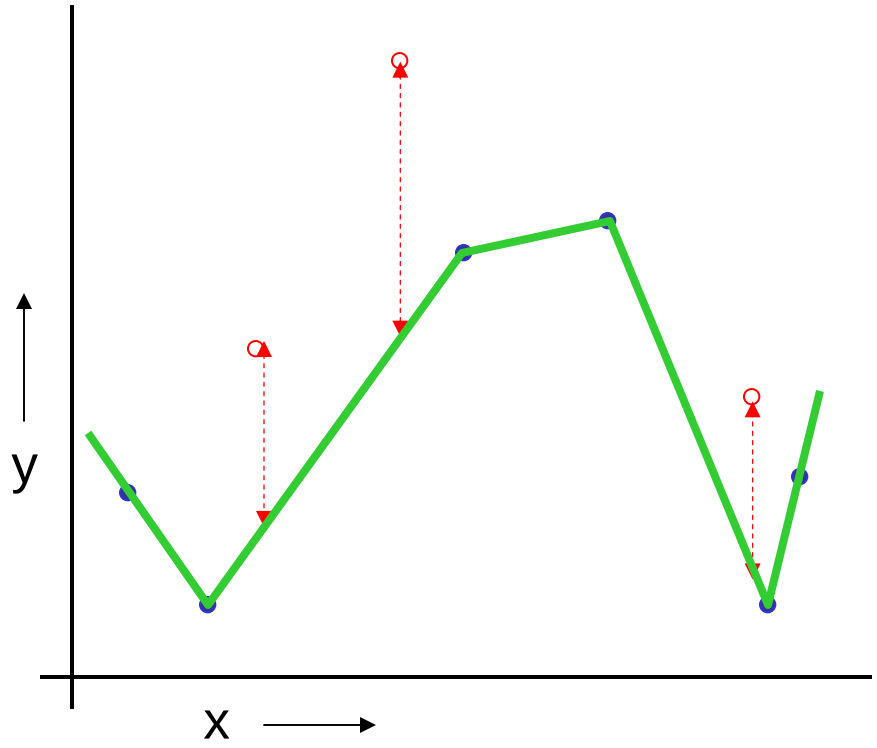
1. Randomly choose 30% of the data to be in a test set

2. The remainder is a training set

3. Perform your regression on the training set

4. Estimate your future performance with the test set

The test set method



(Join the dots example)

Mean Squared Error = 2.2

1. Randomly choose 30% of the data to be in a test set

2. The remainder is a training set

3. Perform your regression on the training set

4. Estimate your future performance with the test set

The test set method

Good news:

- Very very simple
- Can then simply choose the method with the best test-set score

Bad news:

- What's the downside?

The test set method

Good news:

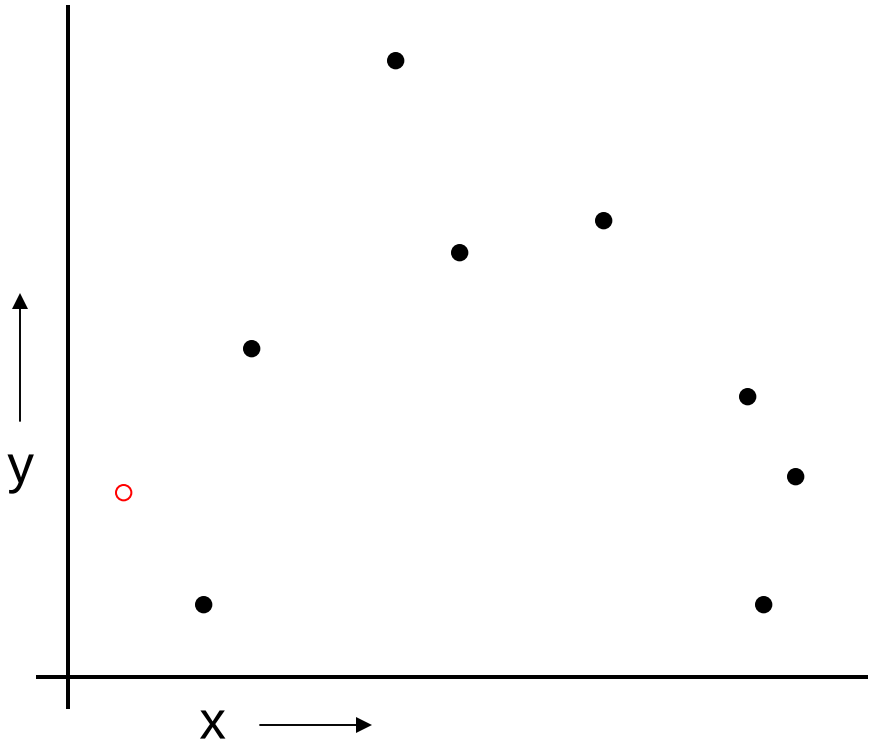
- Very very simple
- Can then simply choose the method with the best test-set score

Bad news:

- Wastes data: we get an estimate of the best method to apply to 30% less data
- If we don't have much data, our test-set might just be lucky or unlucky

We say the "test-set estimator of performance has high variance"

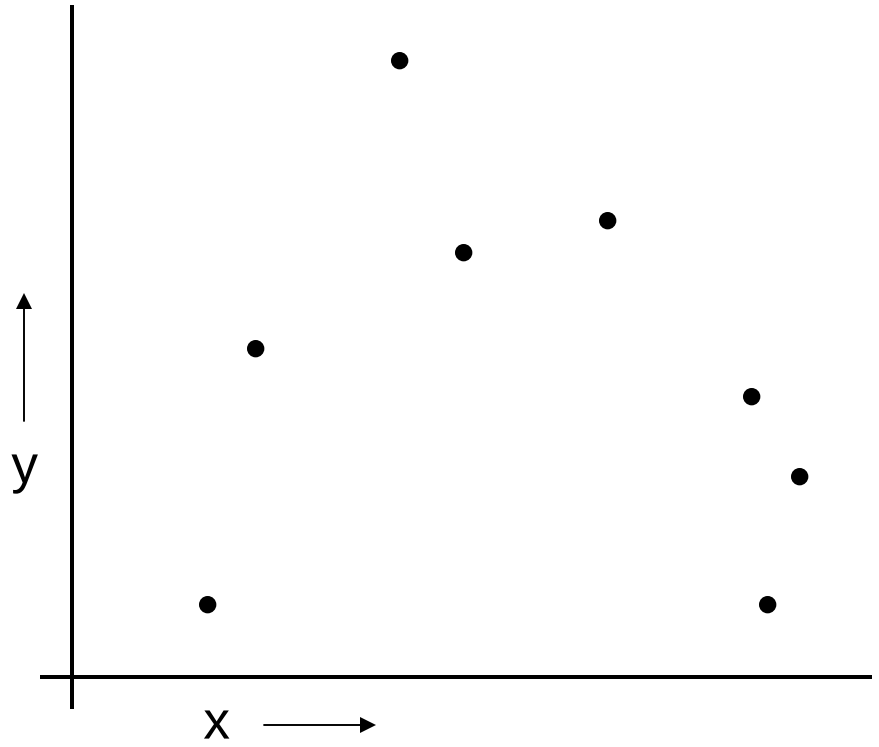
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. Let (x_k, y_k) be the k^{th} record

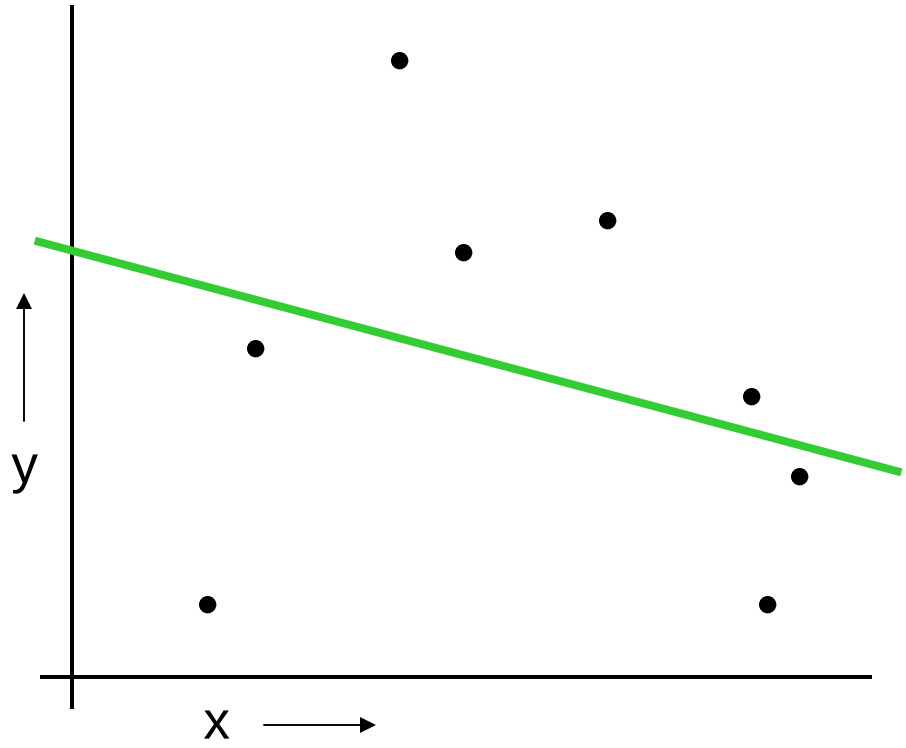
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset

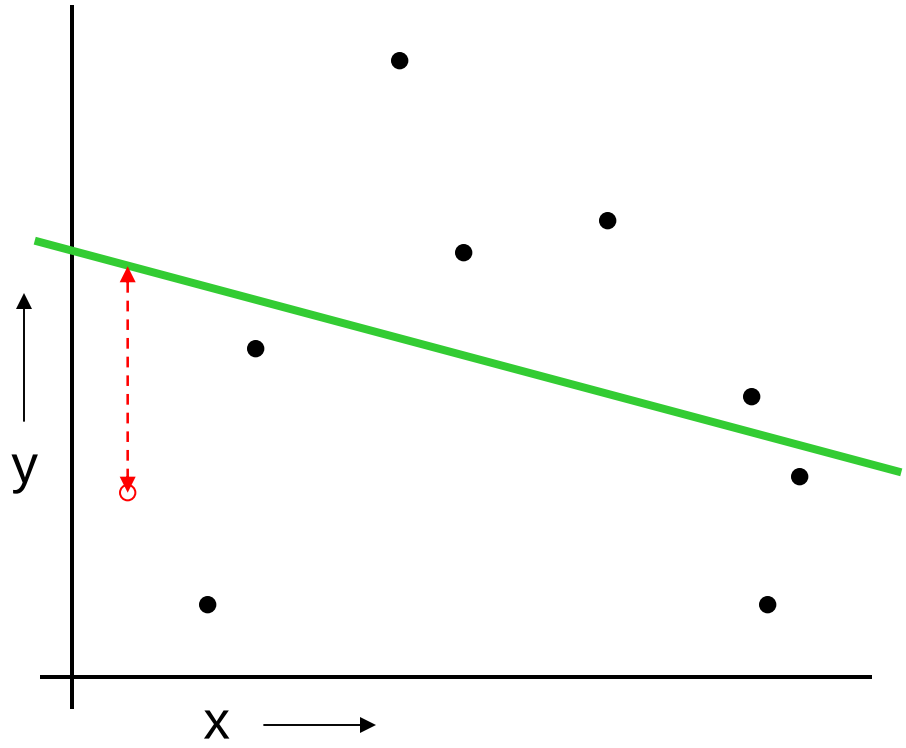
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints

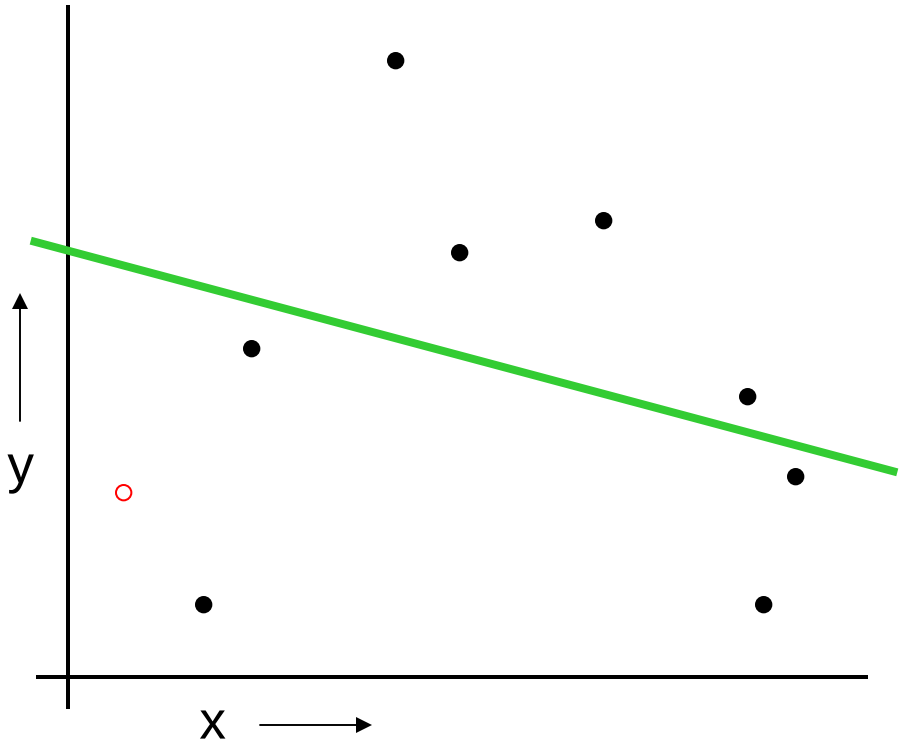
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

LOOCV (Leave-one-out Cross Validation)

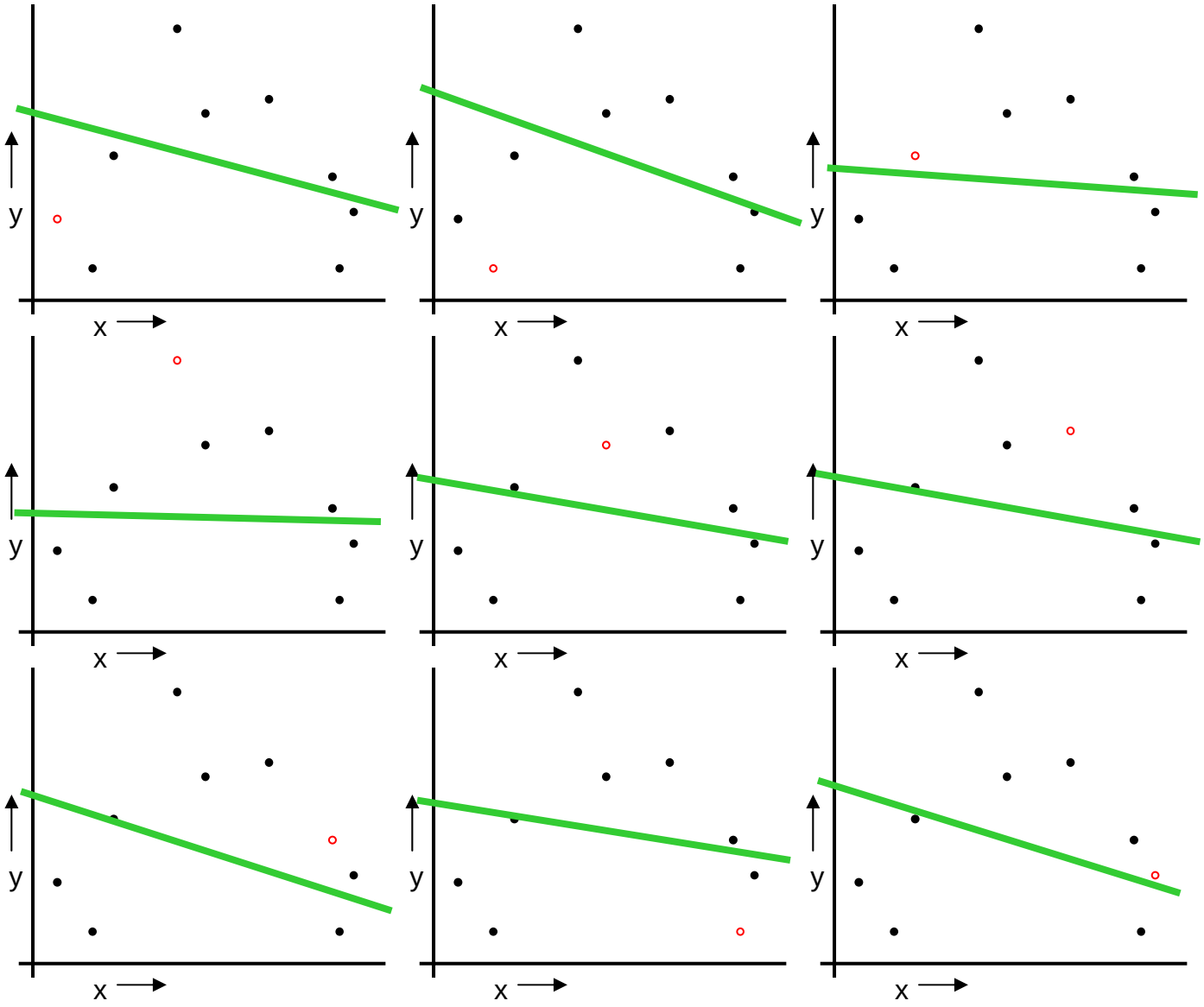


For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

When you've done all points, report the mean error.

LOOCV (Leave-one-out Cross Validation)



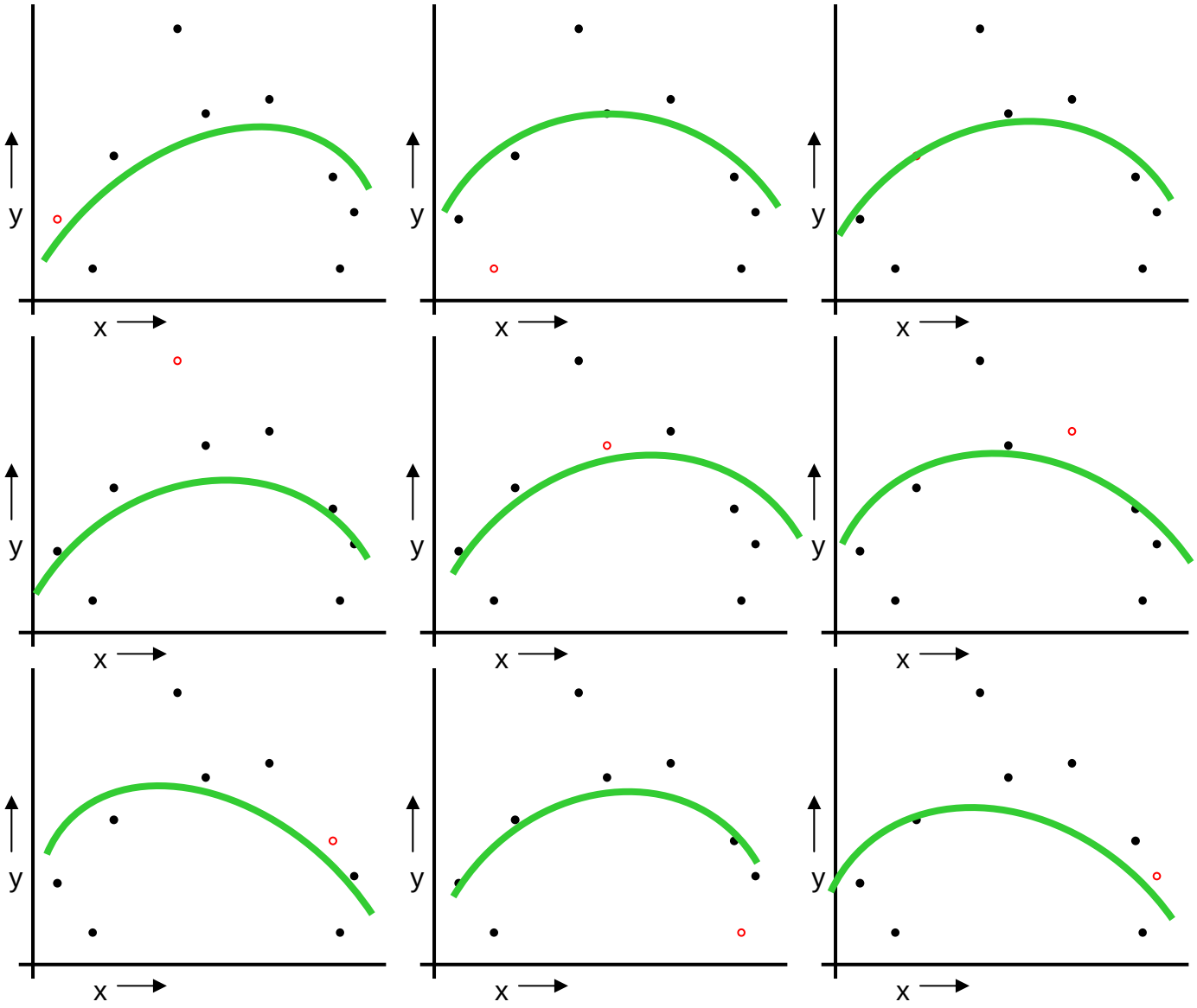
For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

When you've done all points, report the mean error.

$$MSE_{LOOCV} = 2.12$$

LOOCV for Quadratic Regression



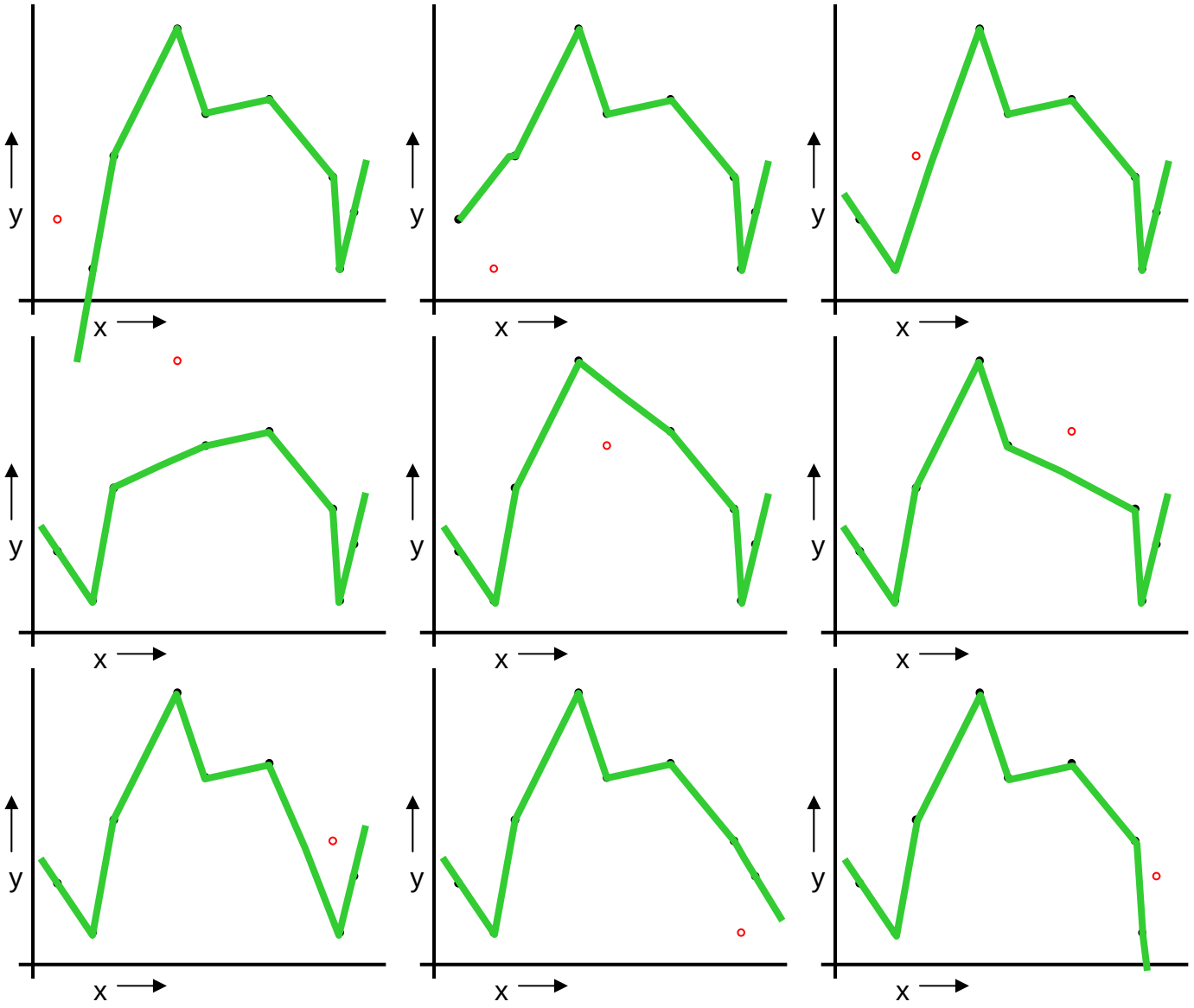
For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

When you've done all points, report the mean error.

$MSE_{LOOCV} = 0.962$

LOOCV for Join The Dots



For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

When you've done all points, report the mean error.

$MSE_{LOOCV} = 3.33$

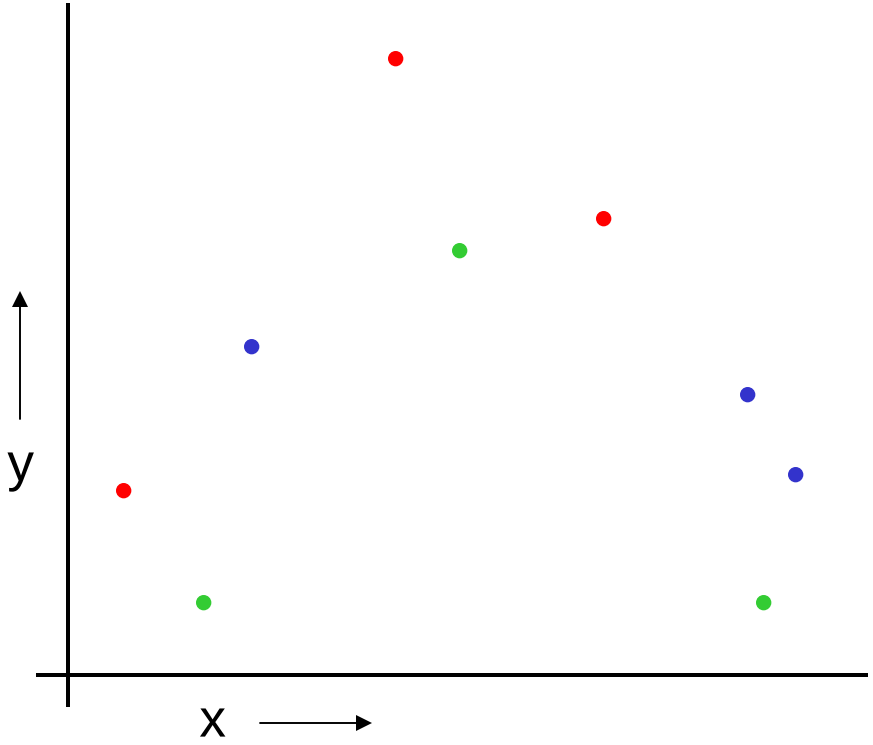
Which kind of Cross Validation?

	Downside	**Upside**
Test-set	Variance: unreliable estimate of future performance	Cheap
Leave-one-out	Expensive. Has some weird behavior	Doesn't waste data

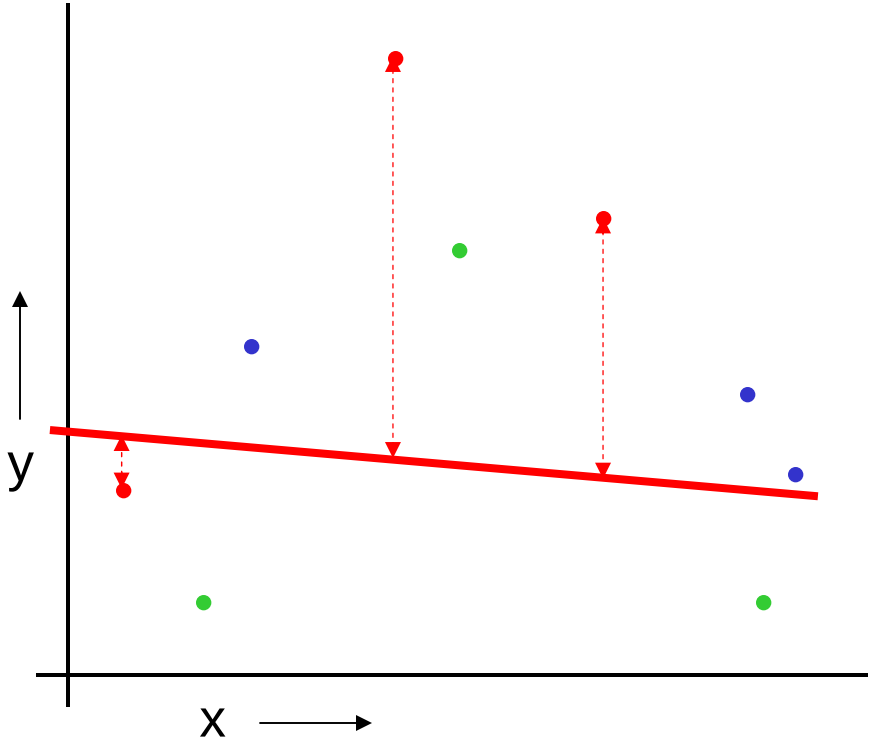
..can we get the best of both worlds?

k-fold Cross Validation

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)



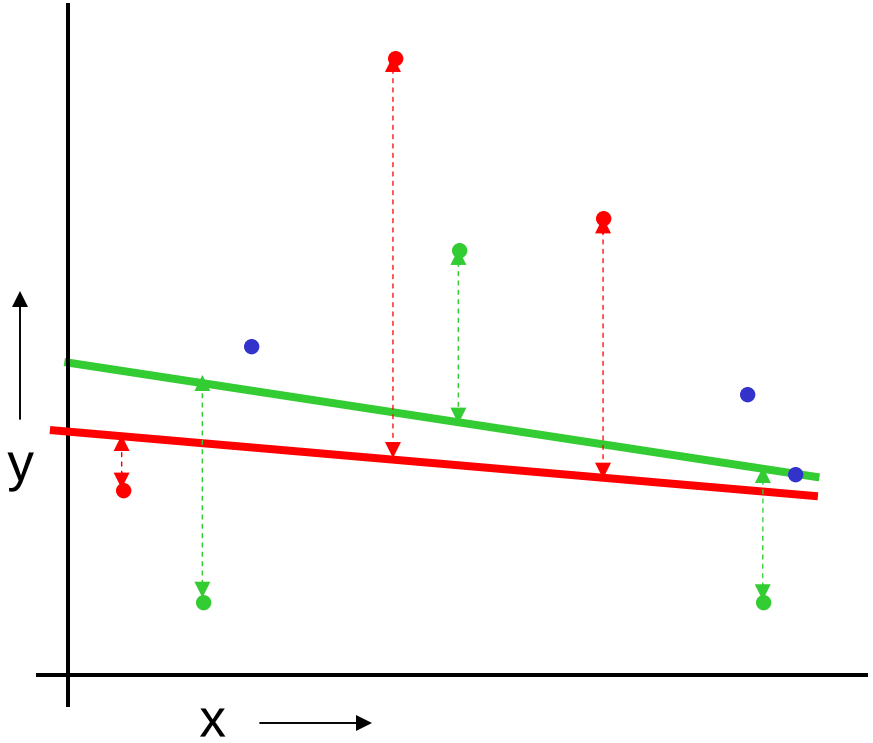
k-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

k-fold Cross Validation

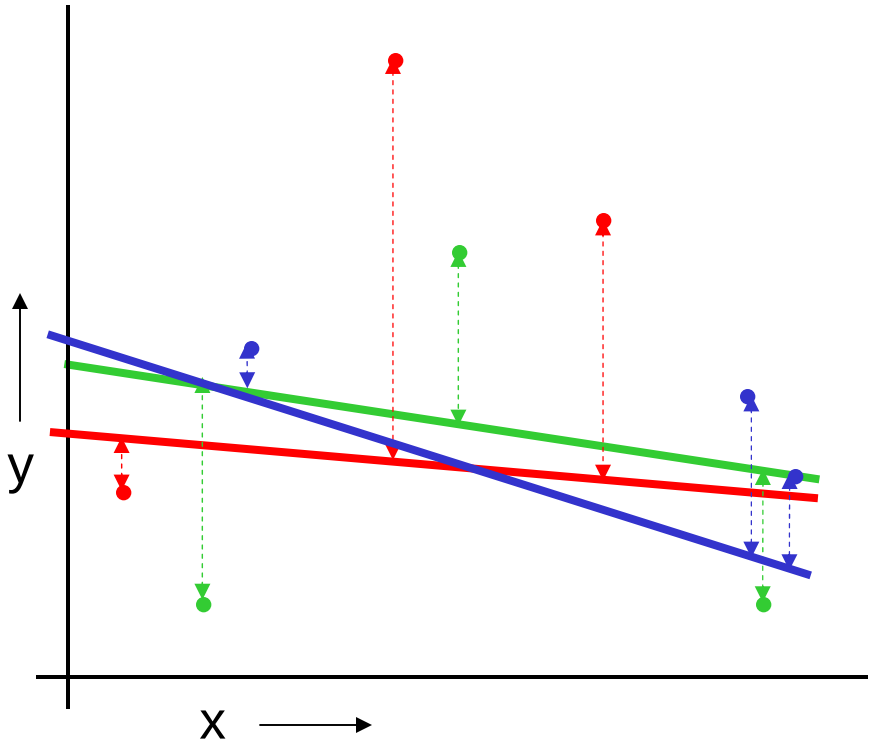


Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

k-fold Cross Validation



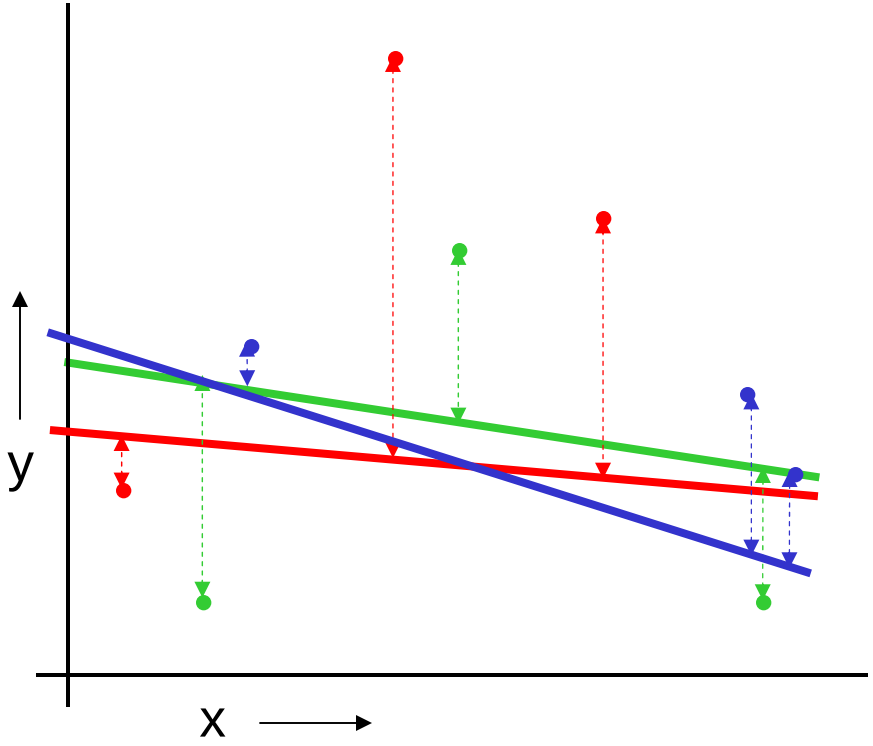
Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

k-fold Cross Validation



Linear Regression

$MSE_{3FOLD}=2.05$

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

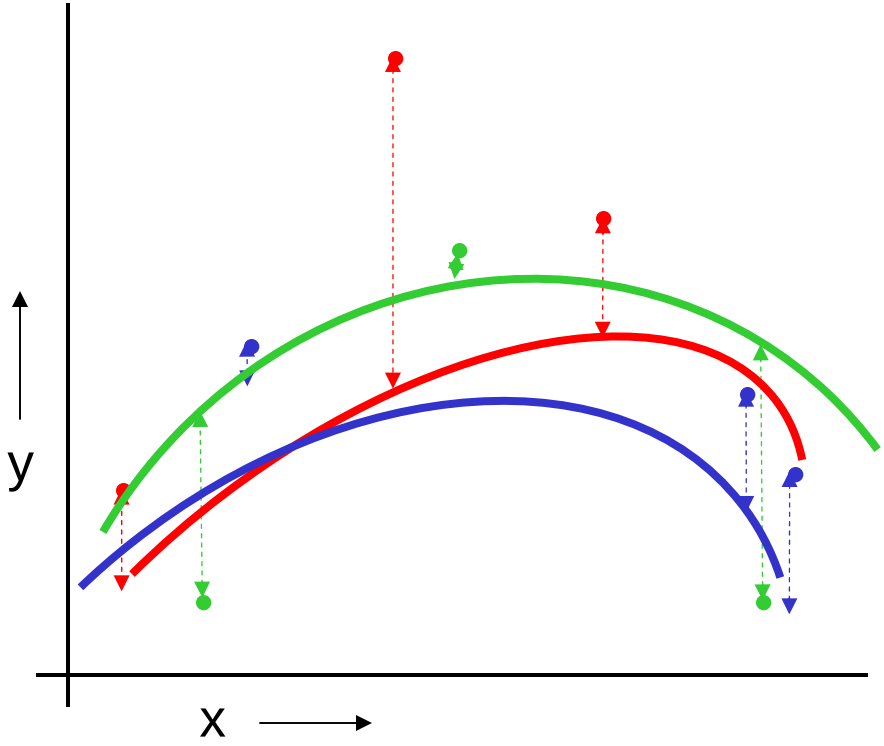
For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

k-fold Cross Validation



Quadratic Regression

$MSE_{3FOLD}=1.11$

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

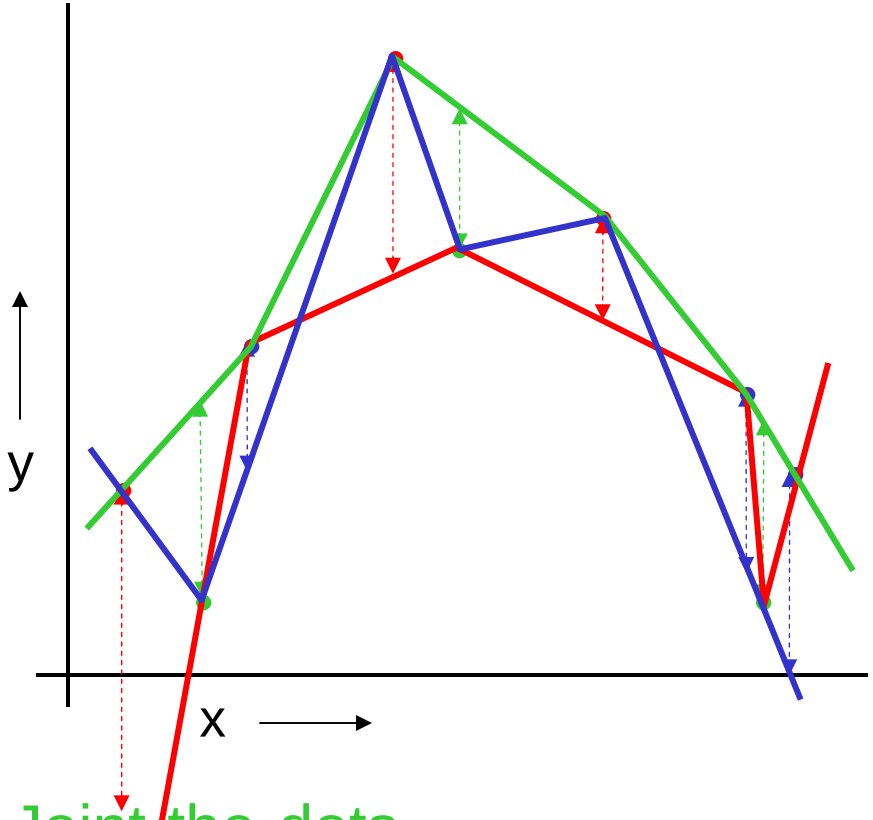
For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

k-fold Cross Validation



Joint-the-dots

$MSE_{3FOLD}=2.93$

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

Which kind of Cross Validation?

	Downside	**Upside**
Test-set	Variance: unreliable estimate of future performance	Cheap
Leave-one-out	Expensive. Has some weird behavior	Doesn't waste data
10-fold	Wastes 10% of the data. 10 times more expensive than test set	Only wastes 10%. Only 10 times more expensive instead of R times.
3-fold	Wastier than 10-fold. Expensivier than test set	Slightly better than test-set
R-fold	Identical to Leave-one-out	

Backpropagation

Chain rule

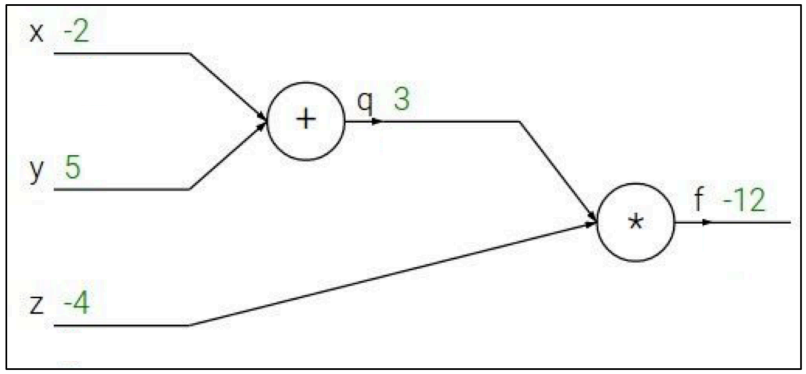
$$f(x,y,z) = (x+y)z$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

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

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

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

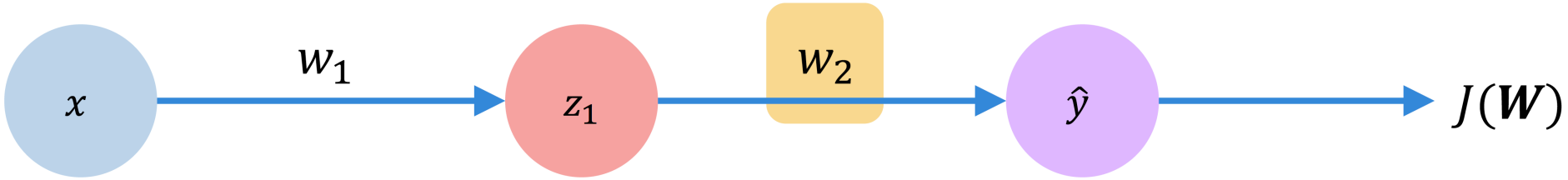


Computational graph

directed graph where the nodes correspond to **operations** or **variables**. Variables can feed their value into operations, and operations can feed their output into other operations.

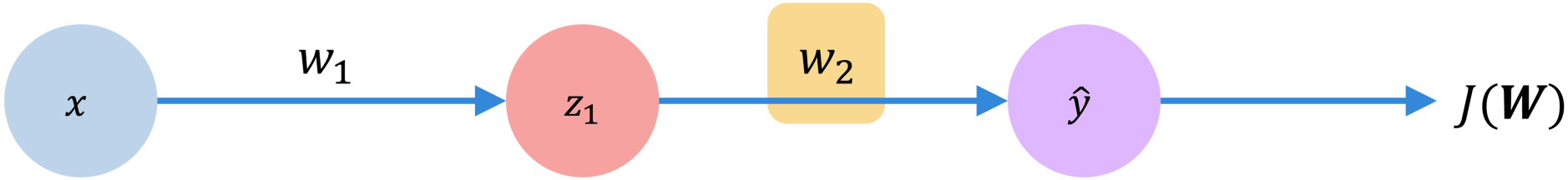
Backpropagation

How do we compute the gradient



How does a small change in one weight (ex. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?

Backpropagation

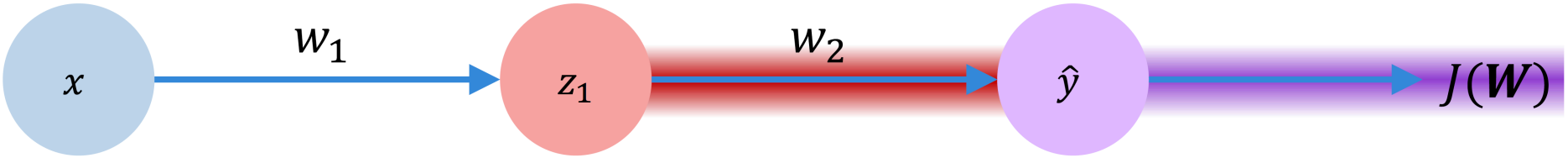


$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} =$$

Let's use the chain rule!

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

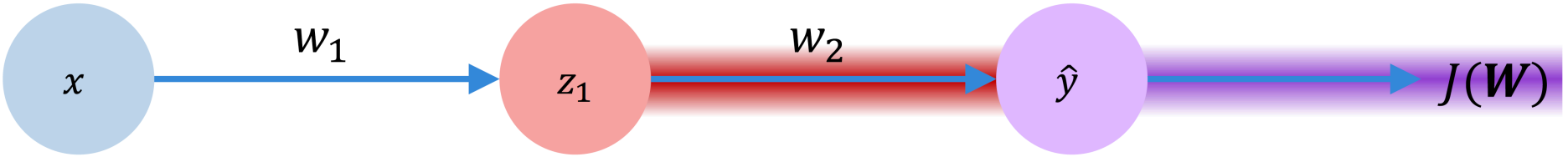
Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

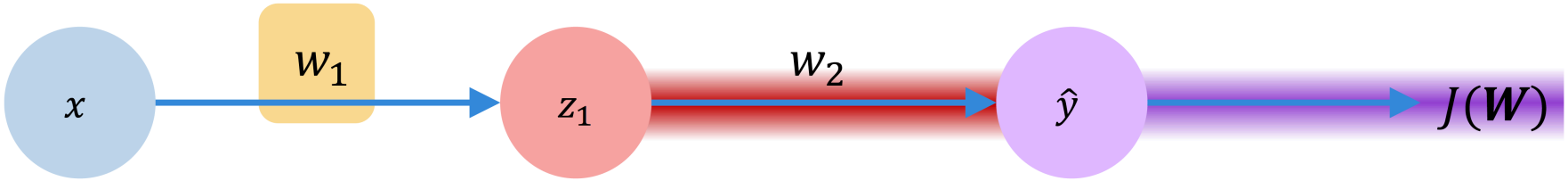
Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

Backpropagation



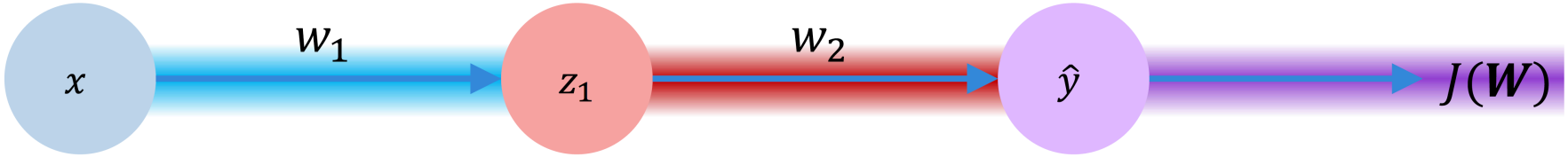
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

Backpropagation



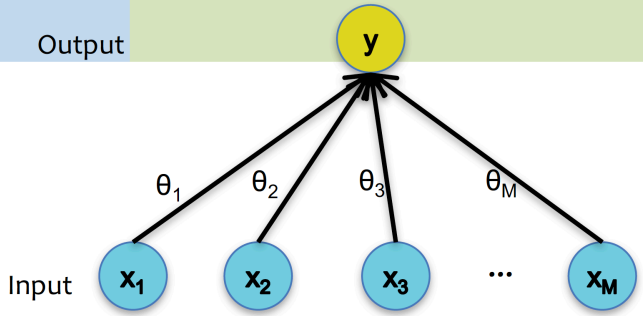
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

Training

Backpropagation

Case 1:
Logistic
Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

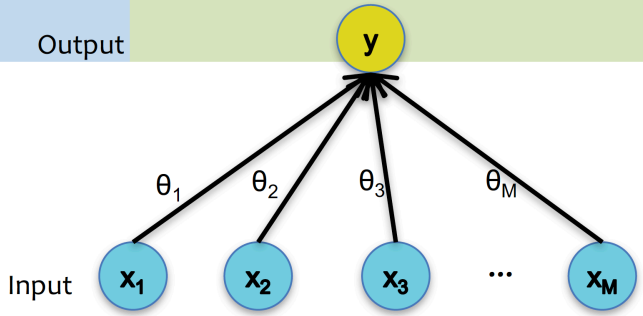
$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

Training

Backpropagation

Case 1:
Logistic
Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

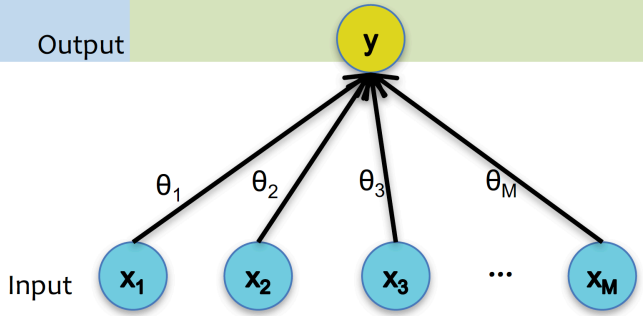
Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

Training

Backpropagation

Case 1:
Logistic
Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

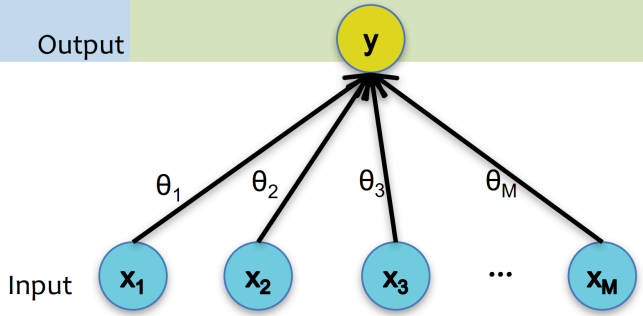
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

Training

Backpropagation

Case 1: Logistic Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

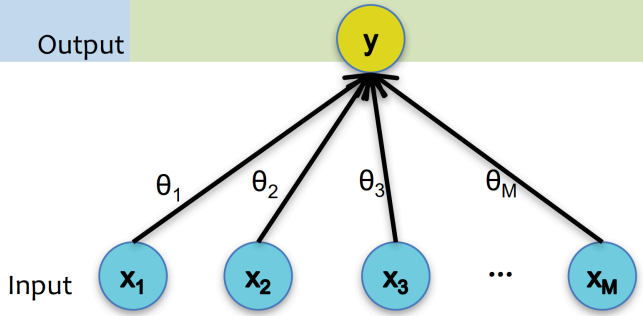
$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \frac{da}{d\theta_j} = x_j$$

Training

Backpropagation

Case 1: Logistic Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

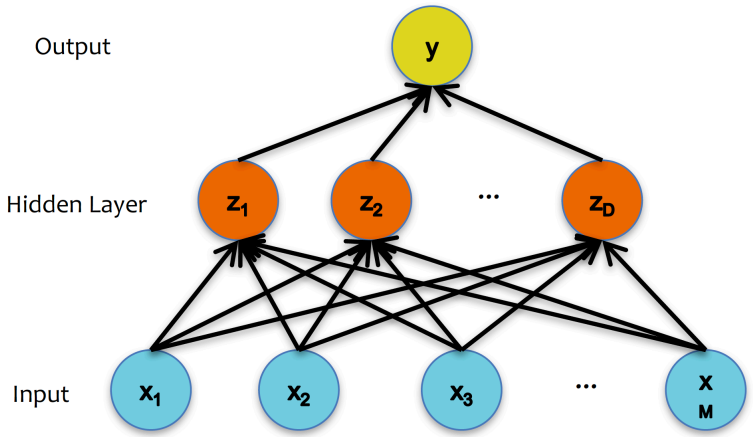
$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

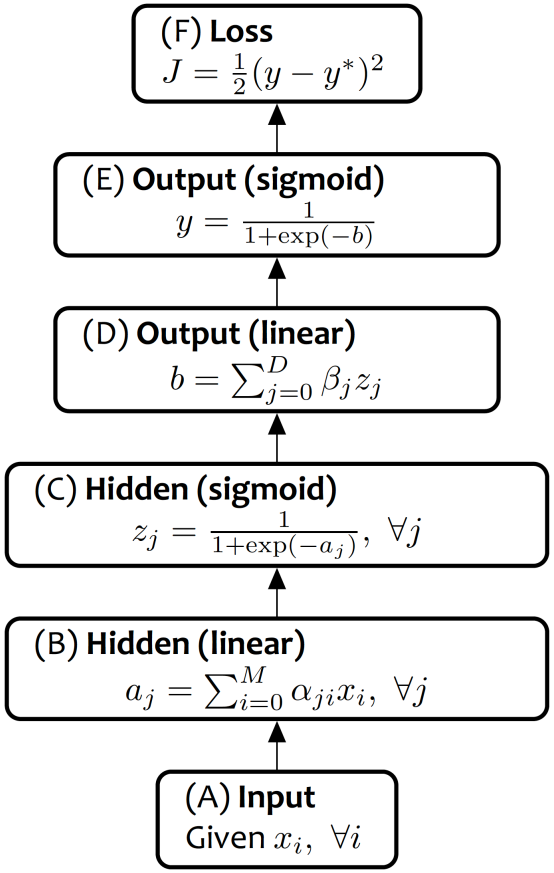
$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \frac{da}{d\theta_j} = x_j$$

$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \frac{da}{dx_j} = \theta_j$$

Training

Backpropagation

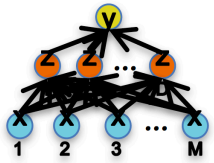




Training

Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

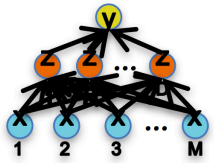
$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

Training

Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

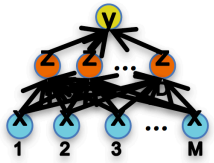
Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

Training

Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

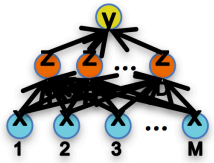
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

Training

Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

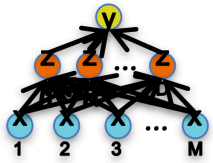
$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

Training

Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

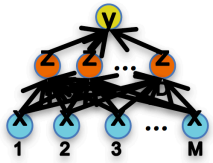
$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$$

Training

Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$$

Work out an example

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

How to avoid bias

- Batch Normalization
  - Can be applied anywhere in the network
    - After convolution/After pooling
  - The x values are fed to the network (5 filters/channels) and we get the responses for them
  - The data that you have will be made to have 0 mean and 1 variance (rescale)
  - We do BN with gamma and beta and train them
  - If some filters have to smooth and some have to detect edges
    - Normalizing them all together will lead to messy behaviour
    - Apply batch normalization on one channel at a time
      - One channel will have the response for one filter
    - Example : To BN your input, you'll have a batch for r, g, and b separately (three different Alpha, Beta and Gamma)

Why?

- The idea of batch normalization is to make the output of each layer have unit statistics.
- Learning then becomes simpler because parameters in the lower layers do not change the statistics of the input to a given layer. This makes learning more straightforward
- Importantly, the normalization and scale/shift operations are included in the computational graph of the neural network so that they participate not only in forward propagation but also in backpropagation

Batch normalization (BN)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

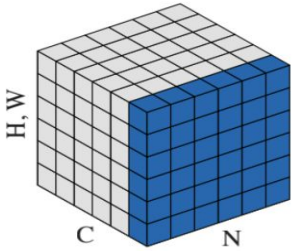
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

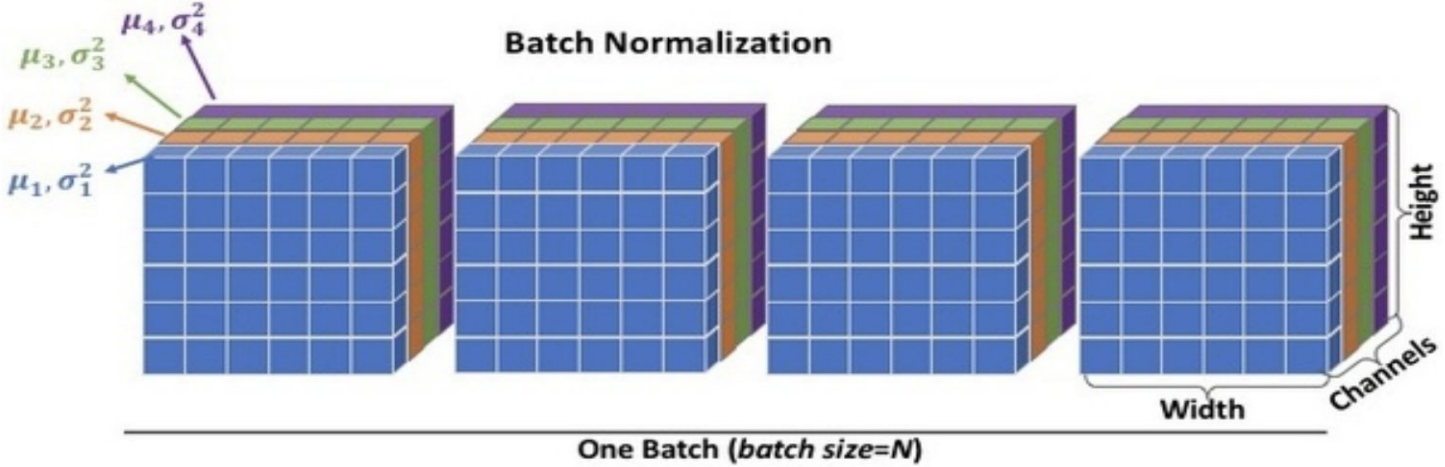
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch normalization (BN)



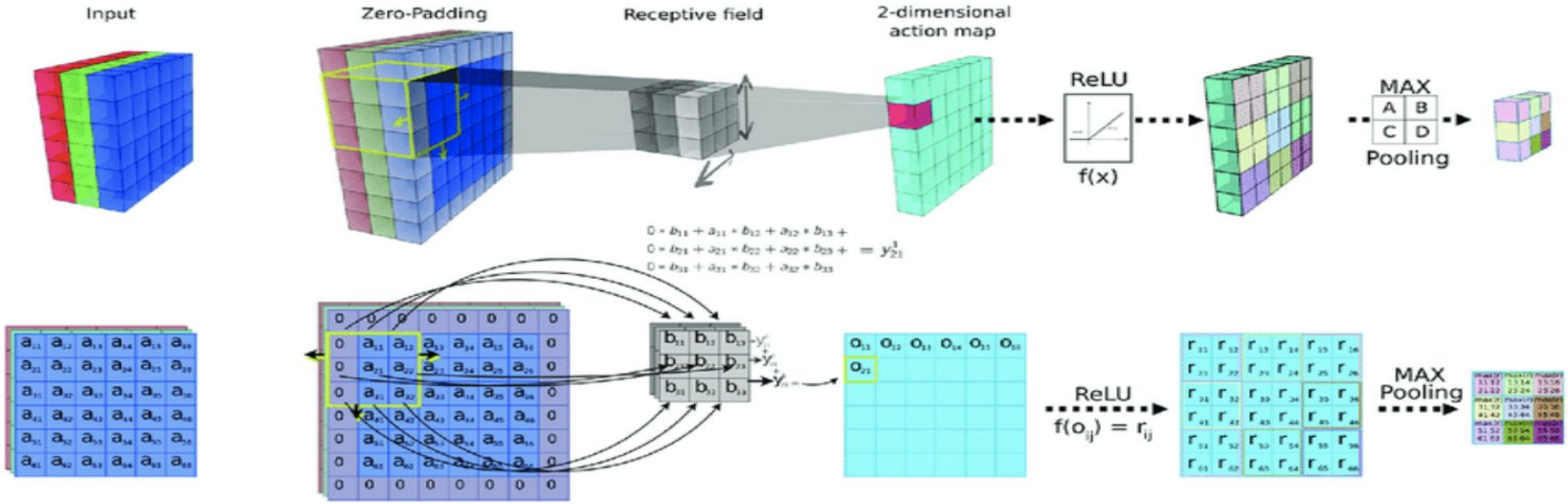


Each cube is one different data point which we have fed to the network

The big picture

Convolution layer

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



Deep Learning MNIST

- A set of handwritten digits (different people writing the same digits)
- Each of them belong to one of 10 classes
- Each image is 28*28 pixels
- 60,000 training samples
- What kind of deep learning can we learn on this?
  - We can define some:
    - Conv filters (padding, stride, kernel)
    - Activation function
    - Max Pooling

MNIST

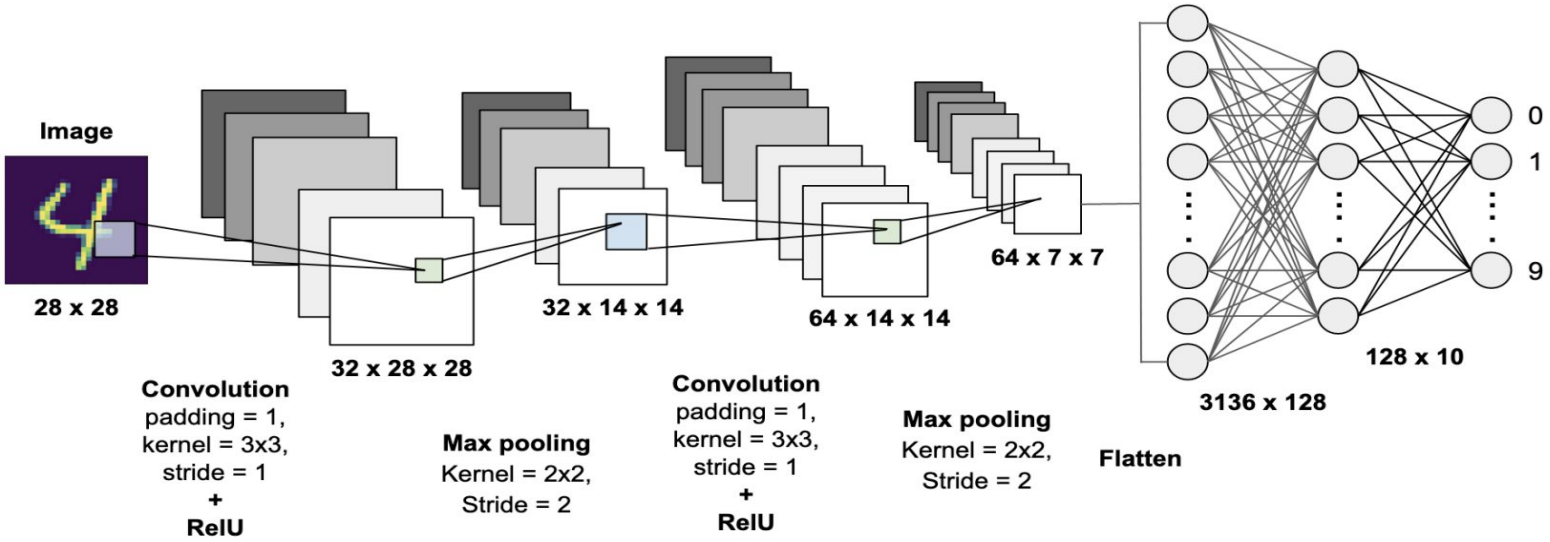


Image
28 x 28
32 x 28 x 28
Convolution
padding = 1,
kernel = 3x3,
stride = 1
+
RelU
32 x 14 x 14
Max pooling
Kernel = 2x2,
Stride = 2
64 x 14 x 14
Convolution
padding = 1,
kernel = 3x3,
stride = 1
+
RelU
64 x 7 x 7
Max pooling
Kernel = 2x2,
Stride = 2
Flatten
3136 x 128
128 x 10
0
1
9

Before our Live Demo of Code

- Import packages
  - Simplified version of tensorflow - has most of what you need
- Specify batch size, #classes and epochs
- We reshape our input, normalize it
- Convert our labels to binary (classification problem)
  - Each observation will be a binary vector with a 1 in the right position

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

- Main 10 lines of code:
  - Sequential
    - One layer after the other: start with the layer closest to the input
    - First Layer:
      - Conv2D : filters are going to be 2D with 32 filters and each filter will be size 3*3
      - Activation function is ReLu
    - Second Layer
      - Conv2D : filters are going to be 2D with 64 filters and each filter will be size 3*3
      - Activation function is ReLu
    - Third Layer:
      - Max Pooling : resize the 28*28 -> 14*14 (we will still have 32*64 channels)
    - Fourth Layer:
      - Dropout : Drop 1/4th of the inputs randomly (regularization to avoid overfitting)
        - Assigns 0 randomly
    - Fifth Layer:
      - Flatten
        - We take the matrix and make one huge vector from that
    - Sixth/Eighth Layer:
      - Dense Layer which makes fully connected network
      - In 8th, we keep the one that has the highest score to be the label

Deep dive into Keras

https://www.pyimagesearch.com/2018/09/10/keras-tutorial-how-to-get-started-with-keras-deep-learning-and-python/

Thank you