Deep Learning

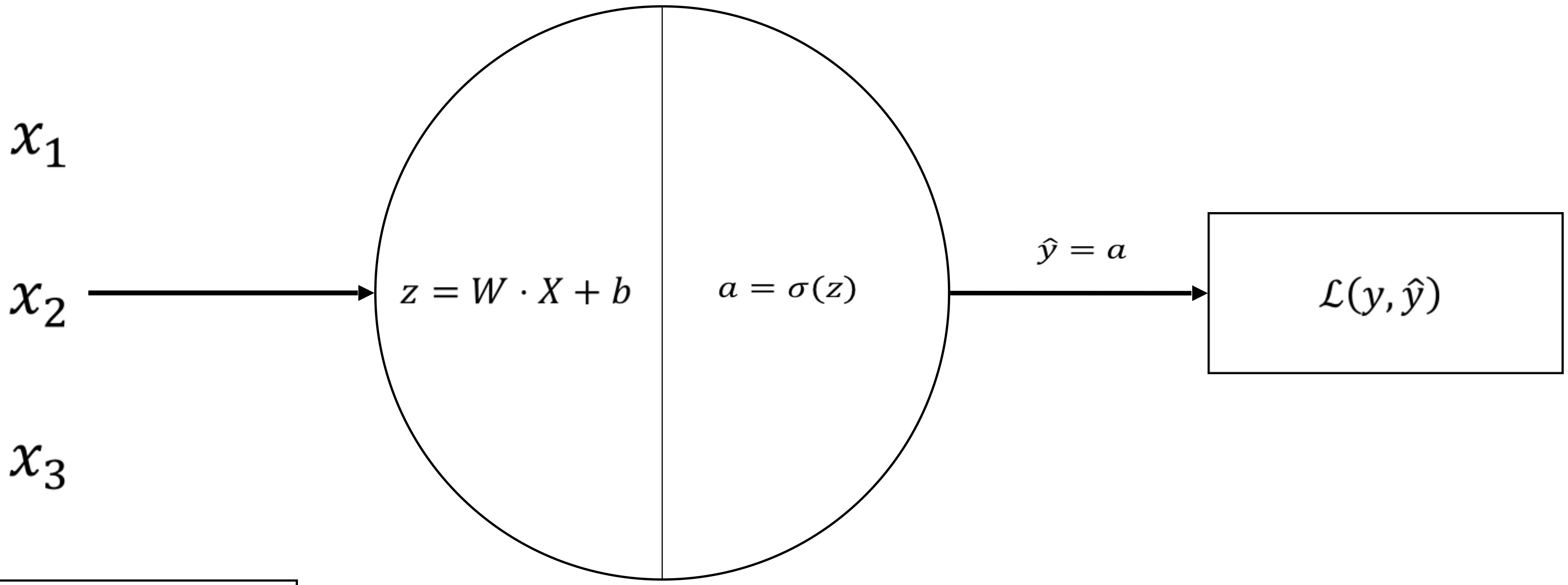
Notation

- The input data X is a matrix with n rows and m columns;
- m is the number of examples in the dataset;
- n is the number of features;
- The labels y is an array with m columns;
- The labels can be 0 or 1 (binary classification);
- $x_1^{(2)}$ denotes the value of the first feature of the second example.

Logistic Regression

- Similar to the perceptron but has a probabilistic interpretation of the output;
- Smooth activation function makes it a good building block for neural networks;
- Easy take-off point to learn about deep learning.

$W = [w1 \quad w2 \quad w3]$ shape = (1, 3)
b is a real number shape = (1, 1)

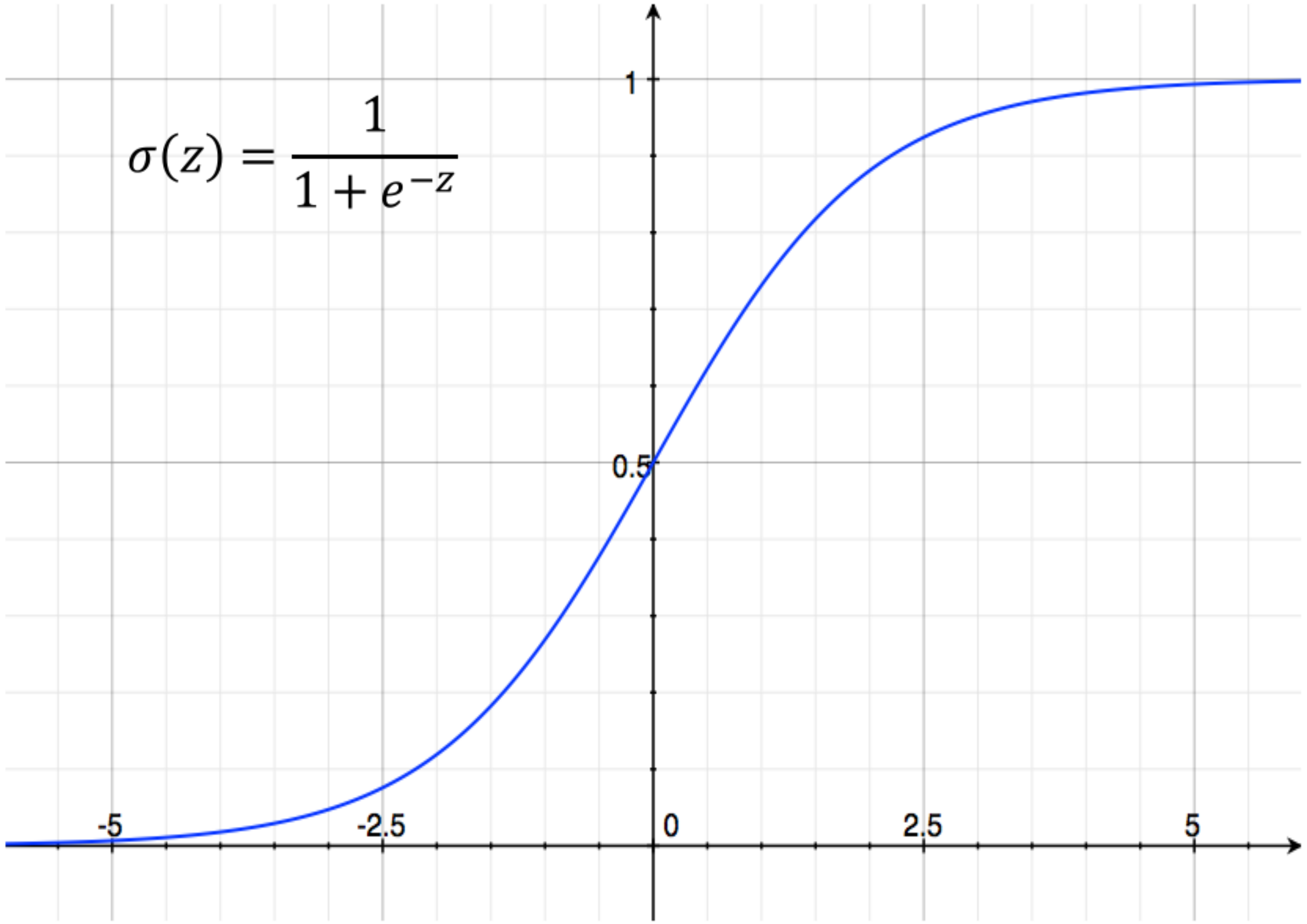


$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$
$$W = [w_1 \quad w_2 \quad w_3]$$
$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

Loss function for one example (cross entropy function):

$$\boldsymbol{\mathcal{L}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))}$$

Sigmoid activation function



Cross-entropy loss function

- $$\boldsymbol{\mathcal{L}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))}$$

  - If y is 0 and $\hat{y}$ is close to 0 then the loss 0;
  - If y is 0 and $\hat{y}$ is close to 1 then the loss tends to infinity;
  - If y is 1 and $\hat{y}$ is close to 1 then the loss 0;
  - If y is 1 and $\hat{y}$ is close to 0 then the loss tends to infinity;

The loss is smaller when y and $\hat{y}$ are close to each other.

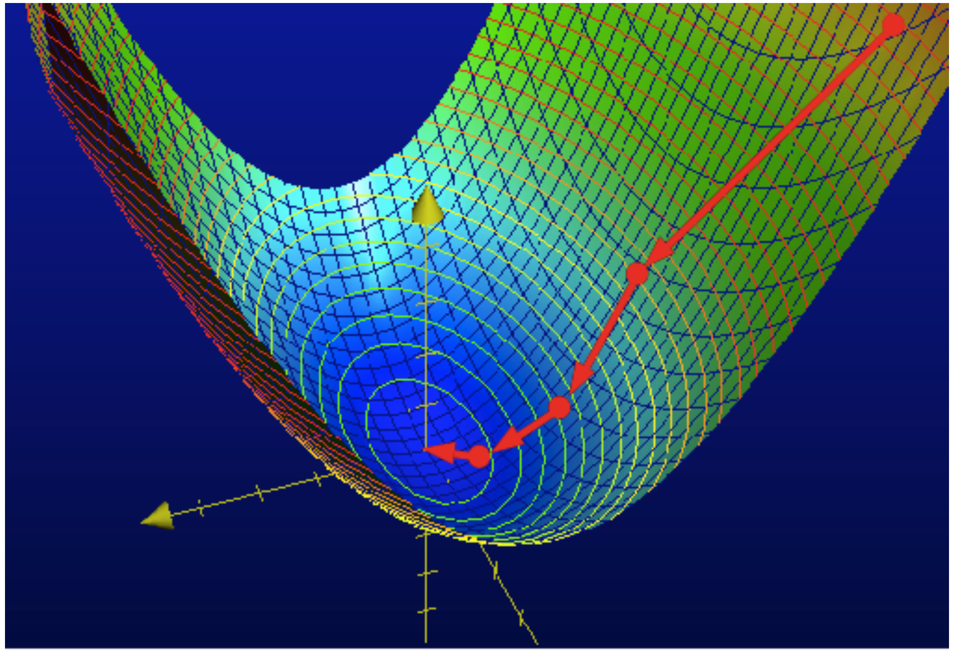
What if there are more than 1 example?

- • $\hat{y}$ becomes an array with the same shape of y;
- W maintains the same shape;
- b maintains the shape but must be broadcasted to all examples;
- The **cost** function is the average of the loss function over all examples:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right)$$

- The cost function can be minimized via gradient descent to determine the optimal values of W and b.

Gradient Descent



- Initialize W and b;
- for every step in range(max_steps):
  - Calculate dW and db:
    - $dW = \frac{\partial J(W,b)}{\partial W}$
    - $db = \frac{\partial J(W,b)}{\partial b}$
  - Update W and b:
    - $W := W - \alpha\, dW$
    - $b := b - \alpha\, db$
- The learning rate α and number of steps for which to train for max_steps are hyper-parameters.

Calculating derivatives on 1 example

- To calculate the derivatives dW and db we use the chain rule:

$$\frac{\partial \mathcal{L}(y,\ a)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{\partial a}{\partial z} = a(1-a)$$

$$dz = \frac{\partial \mathcal{L}(y,a)}{\partial z} = \frac{\partial \mathcal{L}(y,a)}{\partial a}\frac{\partial a}{\partial z} = a - y$$

dz is a real number

-

$$dw_1 = \frac{\partial \mathcal{L}(y,\ a)}{\partial w_1} = \frac{\partial \mathcal{L}(y,a)}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_1} = dz\, x_1$$

$$dW = [dz\, x_1 \quad dz\, x_2 \quad \cdots \quad dz\, x_n]$$

$$db = \frac{\partial \mathcal{L}(y,\ a)}{\partial b} = \frac{\partial \mathcal{L}(y,a)}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b} = dz \times 1$$

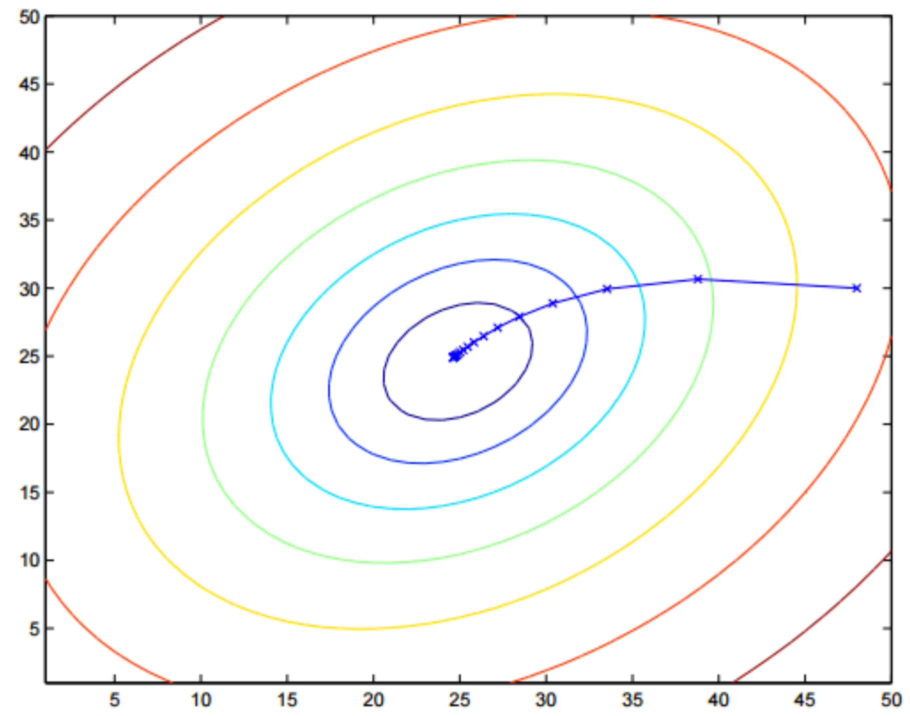
Calculating derivatives on m examples

- Since the cost function is a linear combination of the loss functions of each example, so are its derivatives:

$$dW = \frac{\partial J(W,b)}{\partial W} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}\left(y^{(i)},a^{(i)}\right)}{\partial W}$$

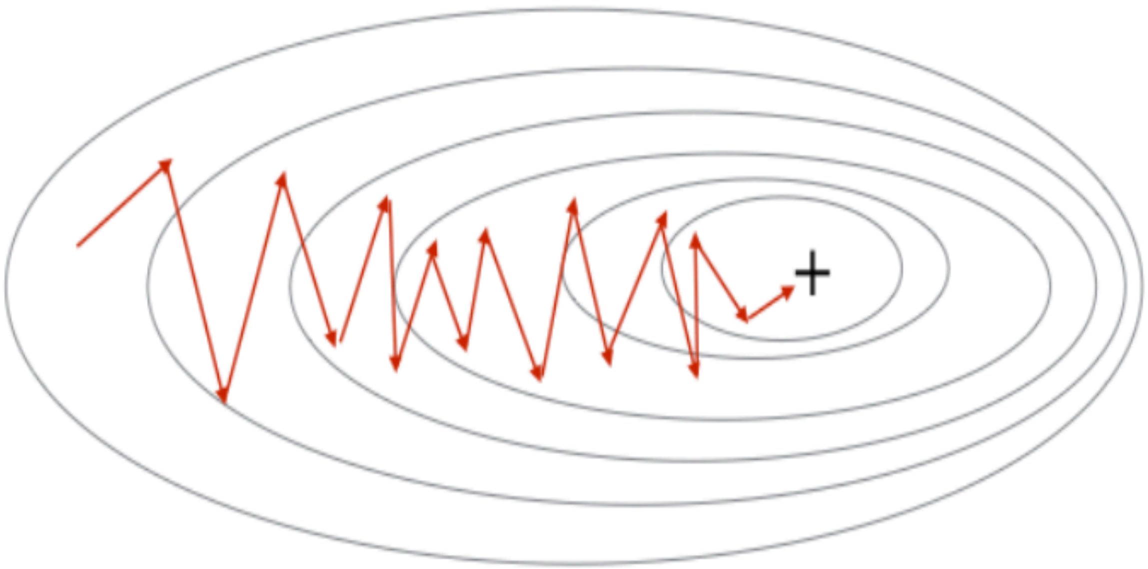
$$db = \frac{\partial J(W,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}\left(y^{(i)},a^{(i)}\right)}{\partial b}$$

Types of Gradient Descer

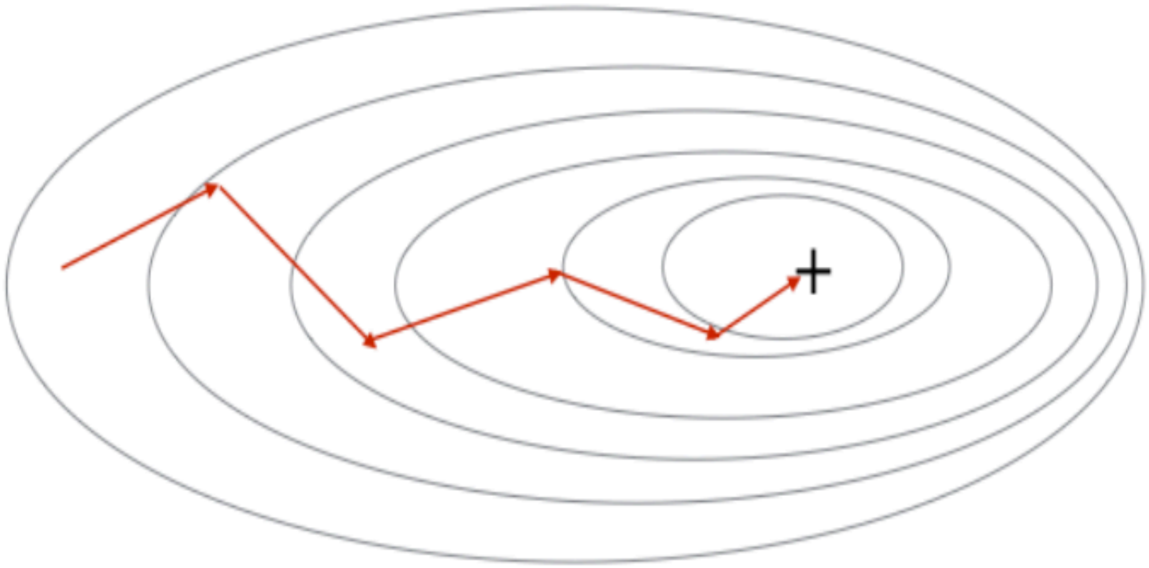


- Batch Gradient Descent:
  - Each iteration uses all m examples;
  - Slow computation of each iteration;
  - Decreases cost function at each iteration.
- Stochastic Gradient Descent (SGD):
  - Each iteration uses only 1 example;
  - Fast computation of each iteration;
  - Does not always decrease cost at each iteration;
  - Does not take advantage of vectorization (slower speed per example).
- Mini-batch gradient descent: (Use this one)
  - Each iteration uses only a subset of the total examples, a mini-batch;
  - The mini-batch size is a hyper-parameter (e.g. 10, 100);
  - Decreases cost function at most iterations;
  - Takes advantage of vectorization.

Stochastic Gradient Descent

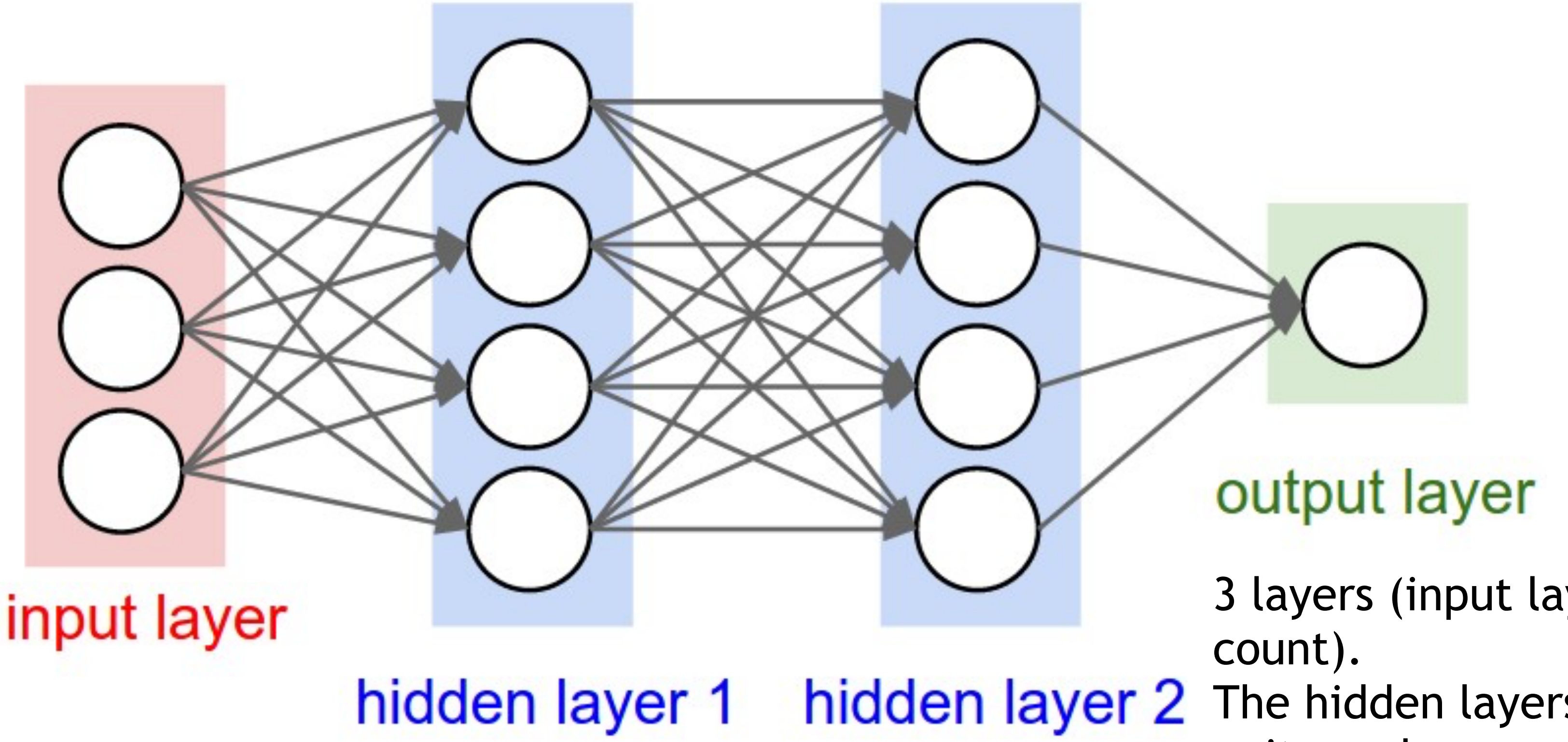


Mini-Batch Gradient Descent



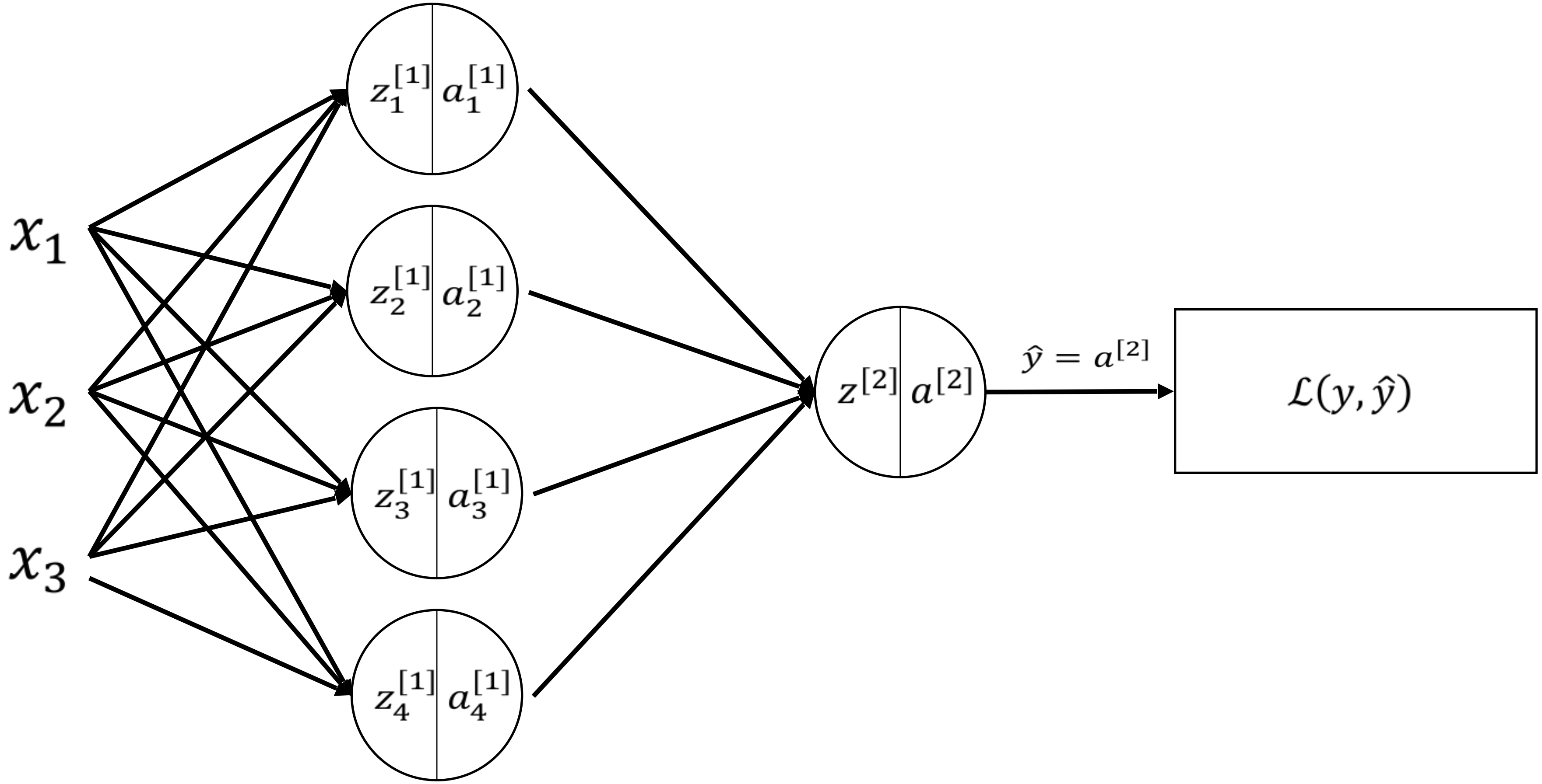
(Fully Connected) Neural Networks

- $w_{ij}^{[l]}$ denotes the weight for input i and output j relative to layer l.



3 layers (input layer does not count).
The hidden layers have 4 hidden units each.

2-layer Neural Network



Forward pass

- $$W^{[1]} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \\ w_{31}^{[1]} & w_{32}^{[1]} & w_{33}^{[1]} \\ w_{41}^{[1]} & w_{42}^{[1]} & w_{43}^{[1]} \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

$$W^{[2]} = \begin{bmatrix} w_{11}^{[2]} & w_{12}^{[2]} & w_{13}^{[2]} & w_{14}^{[2]} \end{bmatrix} \qquad b^{[2]} = \begin{bmatrix} b_1^{[2]} \end{bmatrix}$$

$Z^{[1]} = W^{[1]}X + b^{[1]}$ $\quad A^{[1]} = g^{[1]}\left(Z^{[1]}\right)$ $\quad$ both have shape $(4, m)$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$ $\quad A^{[2]} = g^{[2]}\left(Z^{[2]}\right) = \sigma(Z^{[2]})$ $\quad$ both have shape $(1, m)$

Backpropagation

- $dZ^{[2]} = A^{[2]} - Y$

$dW^{[2]} = \frac{1}{m} dZ^{[2]} {A^{[1]}}^T$ $\quad db_i^{[2]} = \frac{1}{m} \sum_{j=1}^{m} dZ_{i,j}^{[2]}$ same as sum of columns `np.sum(dZ2, axis=1, keepdims=true)`

$dZ^{[1]} = {W^{[2]}}^T dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$ $*$ is element wise product

$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$ $\quad db_i^{[1]} = \frac{1}{m} \sum_{j=1}^{m} dZ_{i,j}^{[1]}$ same as sum of columns `np.sum(dZ1, axis=1, keepdims=true)`

General case L-layer network

- $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \quad A^{[l]} = g^{[l]}\left(Z^{[l]}\right)$

$dZ^{[l]} = {W^{[l+1]}}^T dZ^{[l+1]} * g^{[l]\prime}\,(Z^{[l]})$ $\quad *$ is element wise product

$dW^{[l]} = \frac{1}{m} dZ^{[l]} {A^{[l-1]}}^T \qquad db_i^{[l]} = \frac{1}{m} \sum_{j=1}^{m} dZ_{i,j}^{[l]}$

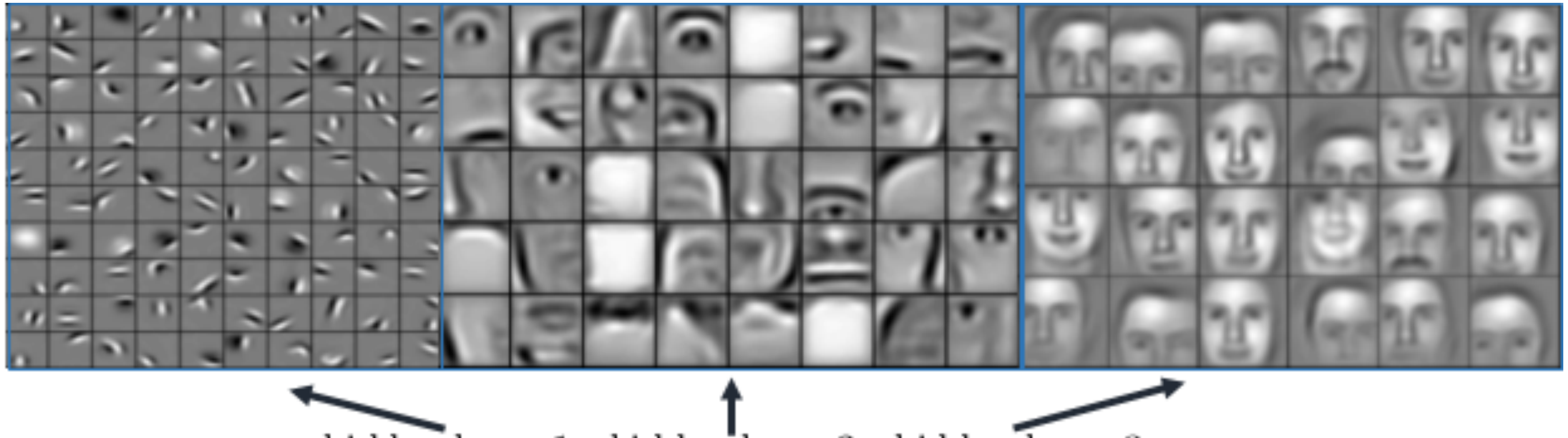
Except:

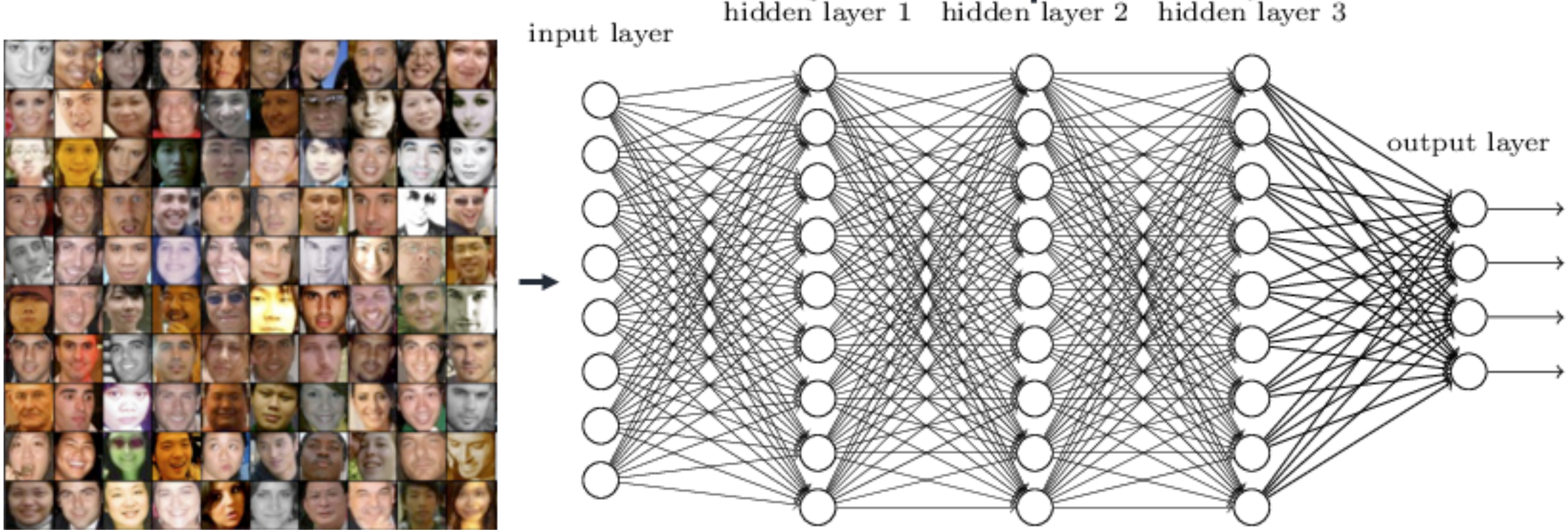
1. In the last layer $\quad dZ^{[L]} = A^{[L]} - Y$
2. In the first layer $\quad A^{[0]} = X$

Shapes

- $W^{[l]}$ and d$W^{[l]}$ have the shape: $(\#outputs\_units, \#inputs\_units)$;
- $b^{[l]}$ and $db^{[l]}$ have the shape $(\#outputs\_units, 1)$:
  - $b^{[l]}$ must be broadcasted to all examples in $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$;
- $Z^{[l]}$ and $A^{[l]}$ has the shape $(\#outputs\_units, m)$.

Deep neural networks learn hierarchical feature representations





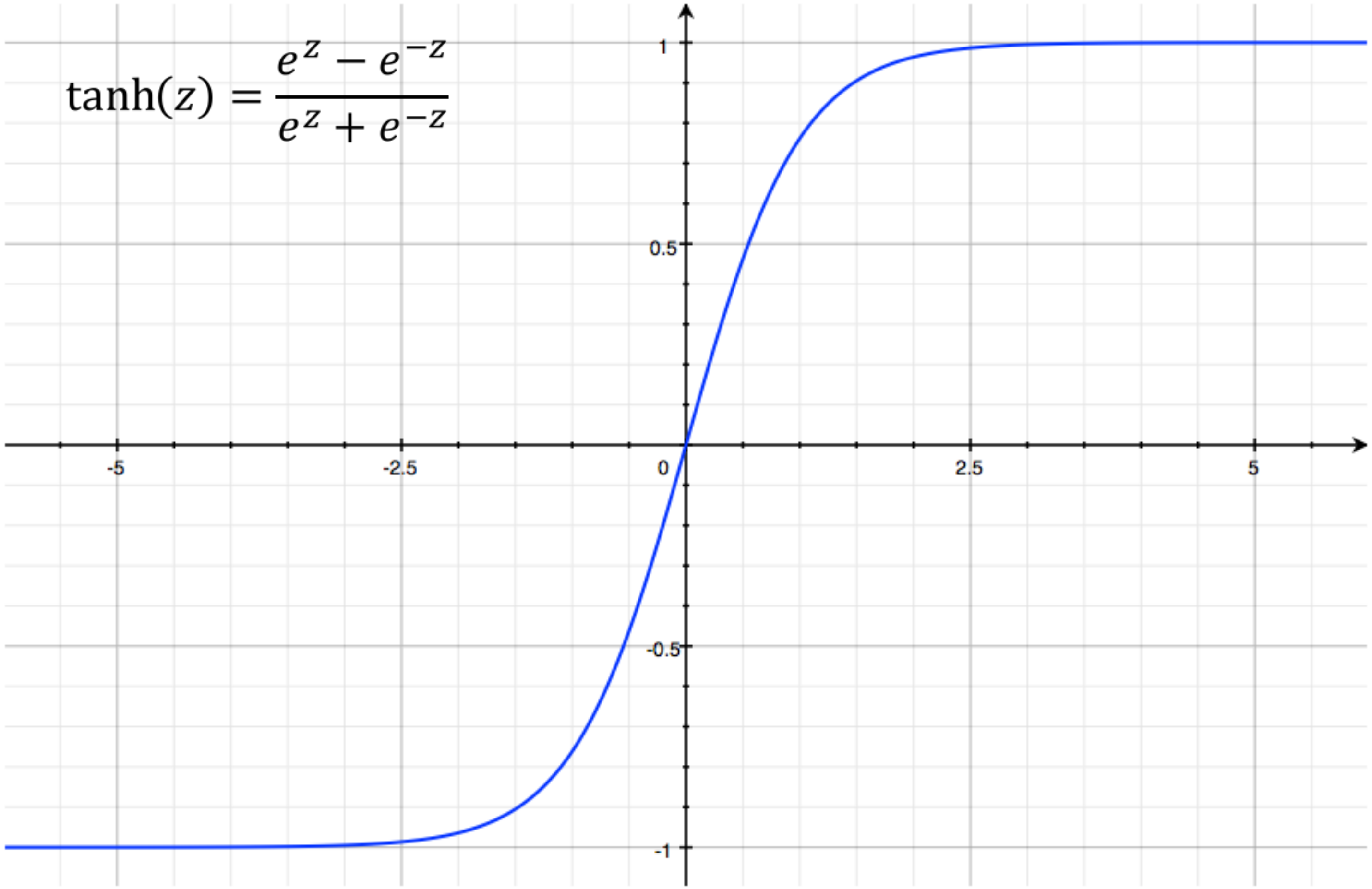
Gradient Descent (2)

- W must be initialized to random values close to zero to break symmetry:
  - For example drawn from a normal distribution with mean 0.0 and standard deviation 1.0;
- b can be initialized to an array of zeros;
- A good choice of the learning rate α is crucial for learning:
  - Typical values 0.1, 0.01, 0001;
  - Choice depends on the problem at hand.
- Always scale your features to have zero mean and unit variance.

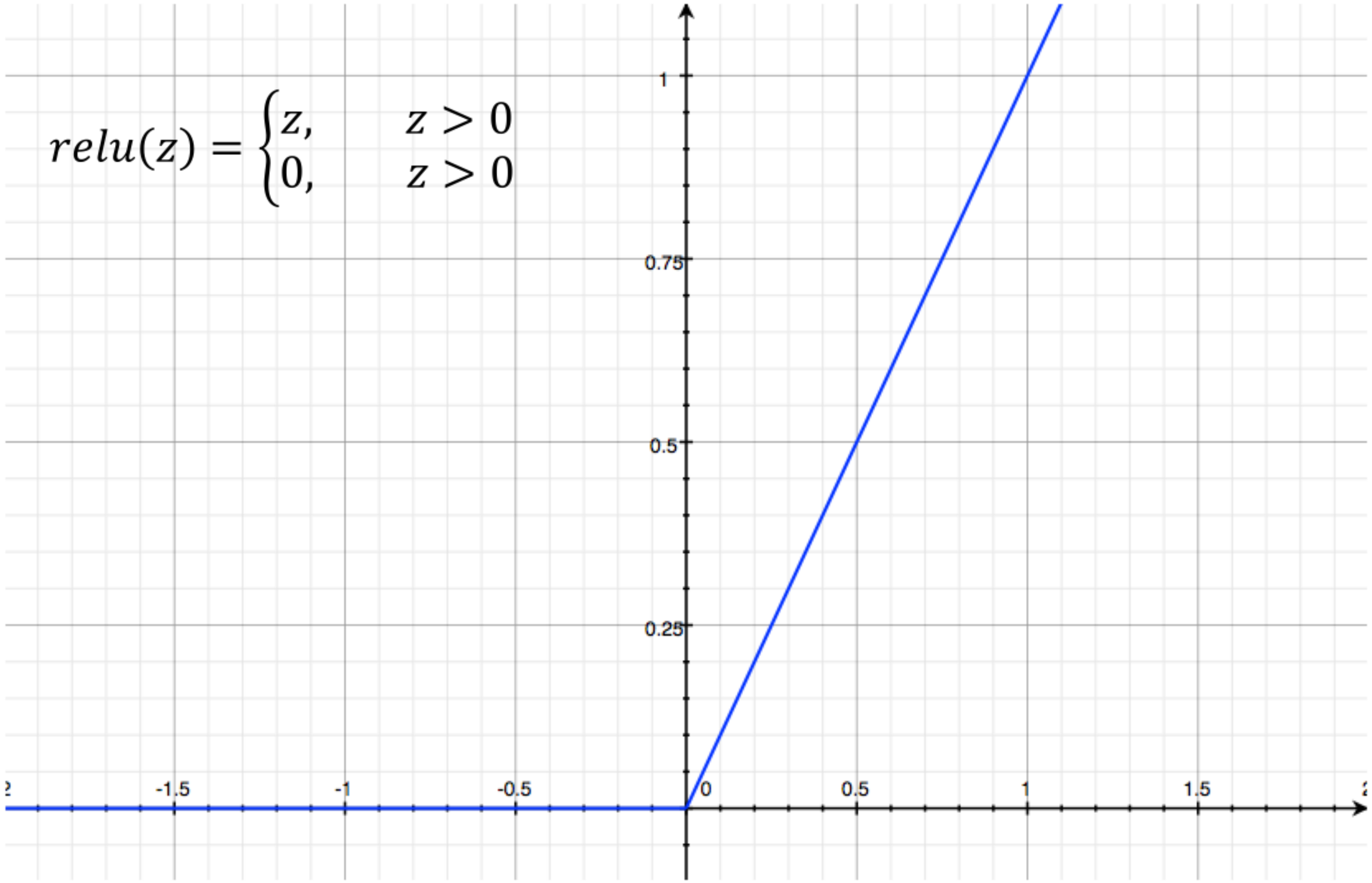
Activation Functions

- The gradient of the sigmoid function is close to zero when the absolute value of the activations are large. This slows learning;
- The activation function of the hidden layers does not need to be the sigmoid function;
- Other functions:
  - Hyperbolic Tangent (Tanh);
  - Rectified Linear Unit (Relu);
  - Leaky Rectified Linear Unit (LeakyRelu);
  - Parametric Rectified Linear Unit (PRelu);
  - More.

Hyperbolic Tangent

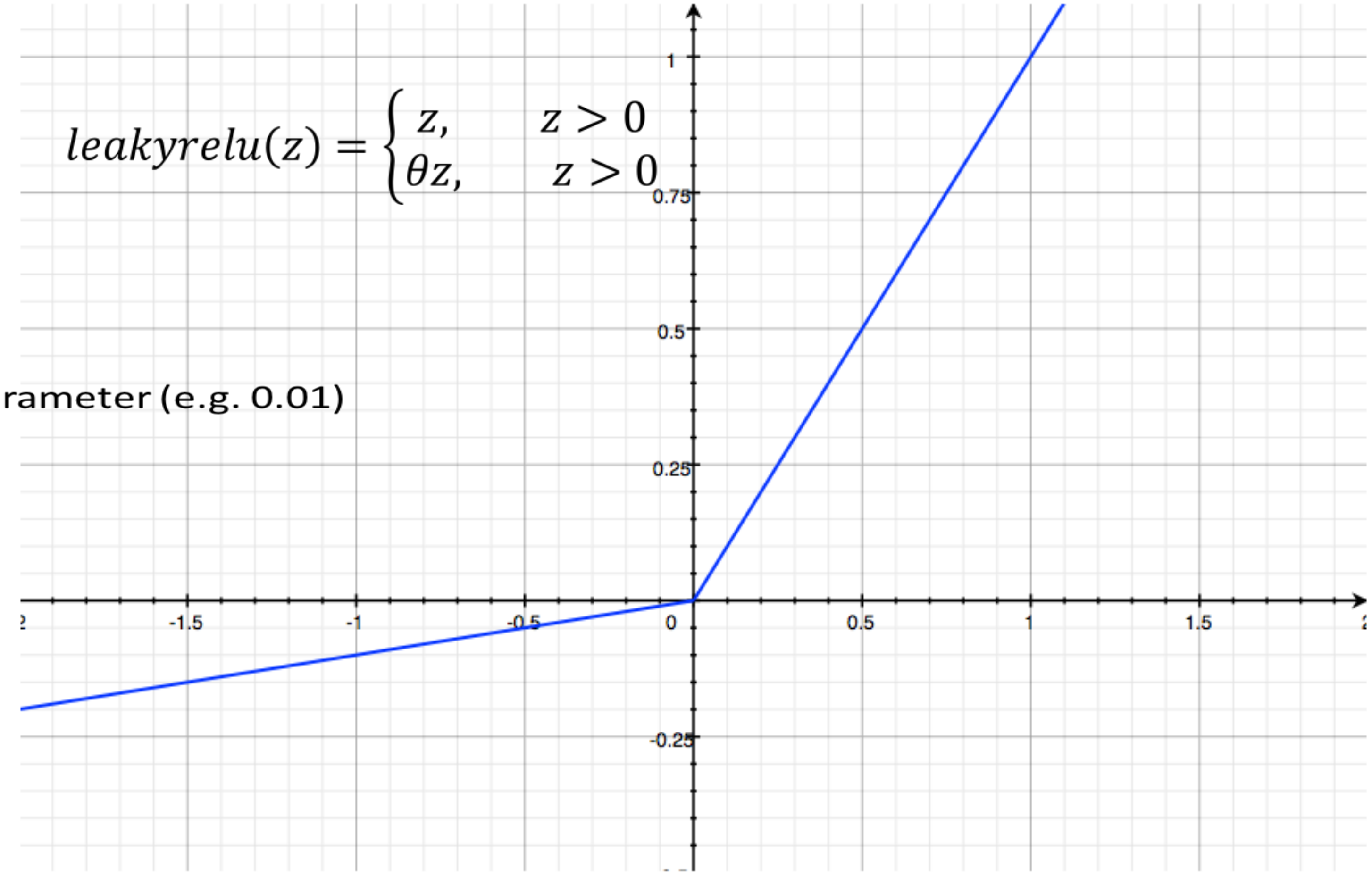


Rectified Linear Unit (Relu)



Leaky Rectified Linear Unit (LeakyRelu)

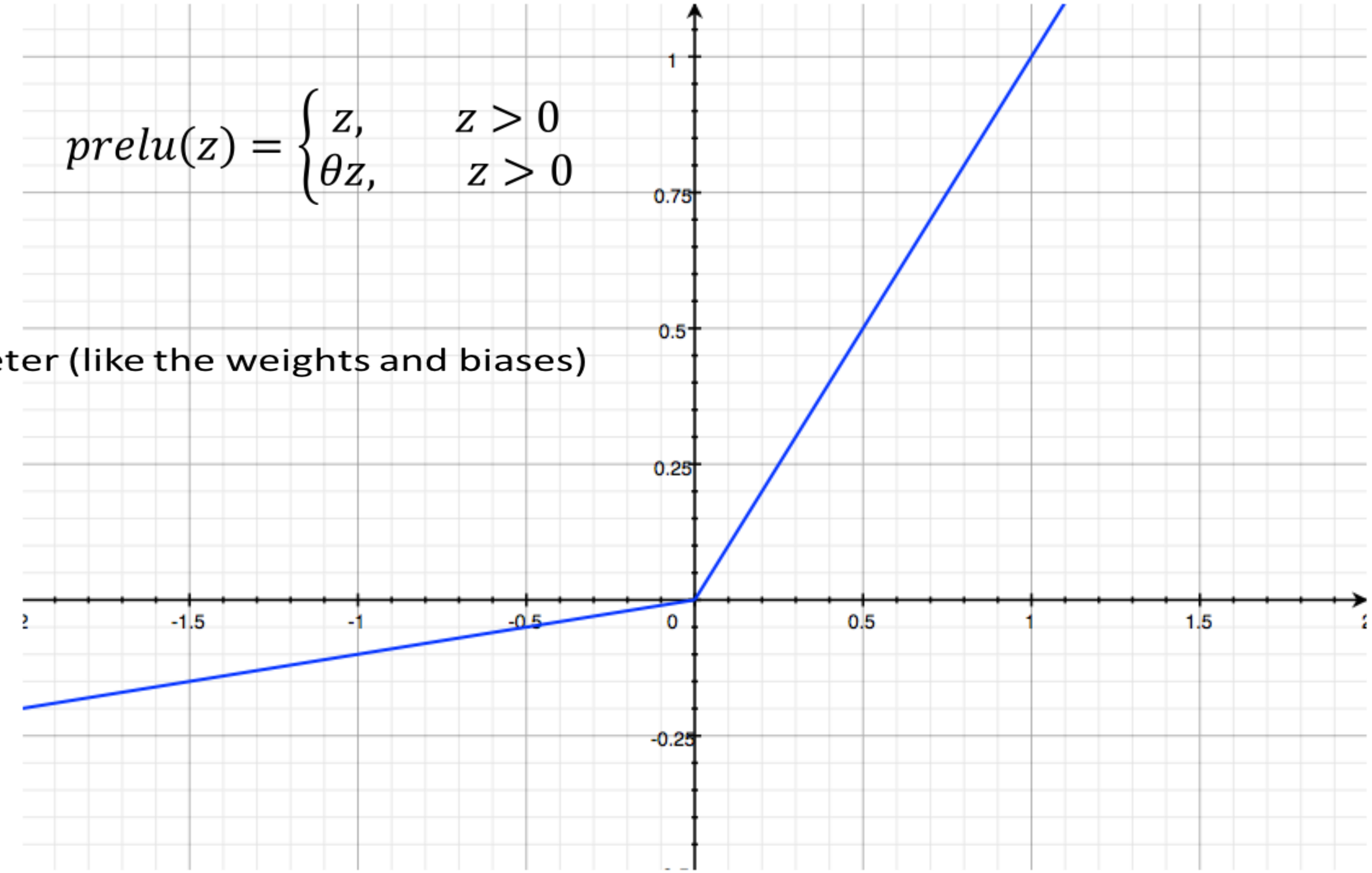
$$leakyrelu(z) = \begin{cases} z, & z > 0 \\ \theta z, & z > 0 \end{cases}$$



θ is a fixed hyper-parameter (e.g. 0.01)

Parametric Rectified Linear Unit (PRelu)

$$prelu(z) = \begin{cases} z, & z > 0 \\ \theta z, & z > 0 \end{cases}$$



θ is a trainable parameter (like the weights and biases)

Programming Frameworks

- You only have to calculate the forward pass, the frameworks compute backpropagation and update the parameters automatically;
- List of programming frameworks:
  - Tensorflow;
  - Theano;
  - Keras;
  - Caffe;
  - Torch;
  - And many more.

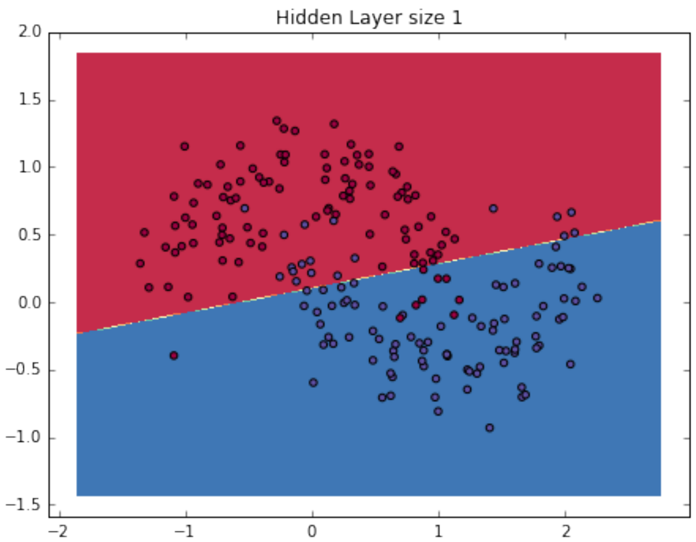
Training, validation and test sets

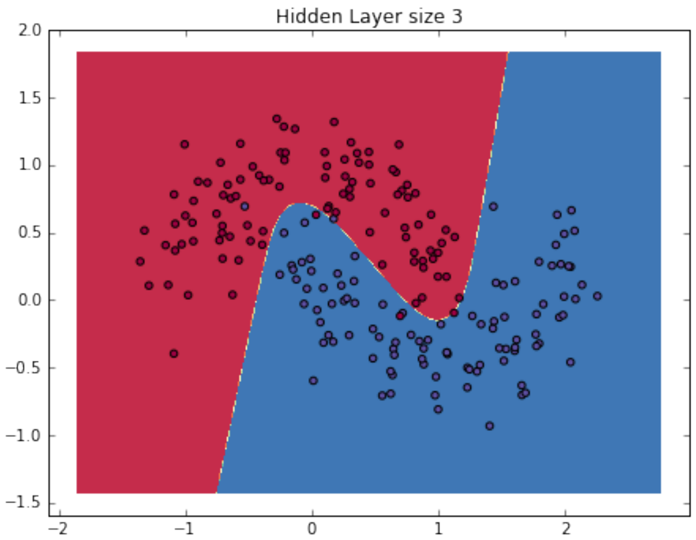
- Training set:
  - Used to train the model;
- Validation set (aka development set):
  - Used to tune the model's parameters and architecture;
  - Ensures you are not overfitting to the training set;
- Test set:
  - Used to get a sense of the model's real world performance;
  - Ensures that you are not overfitting to the validation set;

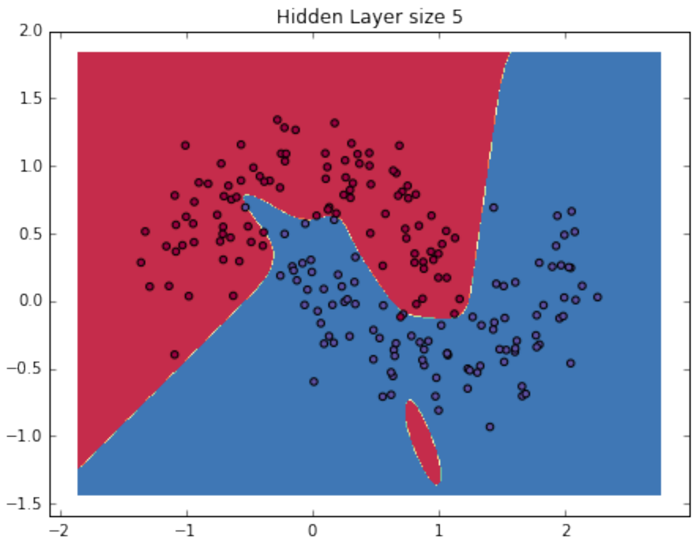
How to split the data?

- Before deep learning (few examples ~1000):
  - Train/Dev/Test: 60%/20%/20%
  - Train/Dev: 70%/30%
  - Use k-fold cross-validation
- In the deep learning era (a lot of examples ~1e6):
  - Train/Dev/Test: 98%/1%/1%
  - Train/Dev: 99%/1%

Bias and Variance







High bias - underfitting

Good fit

High variance - overfitting

Bayes Error and Human Level Performance

- Bayes error:
  - The theoretical lower limit for the error in any machine learning task;
  - Hard or impossible to know most of the time;
- Human level performance:
  - Best performance achieved by humans;
  - In tasks where humans are very good (e.g. image recognition) it can be used as an approximation of Bayes error.

Bias and variance are not opposites, you can have both high bias and high variance.

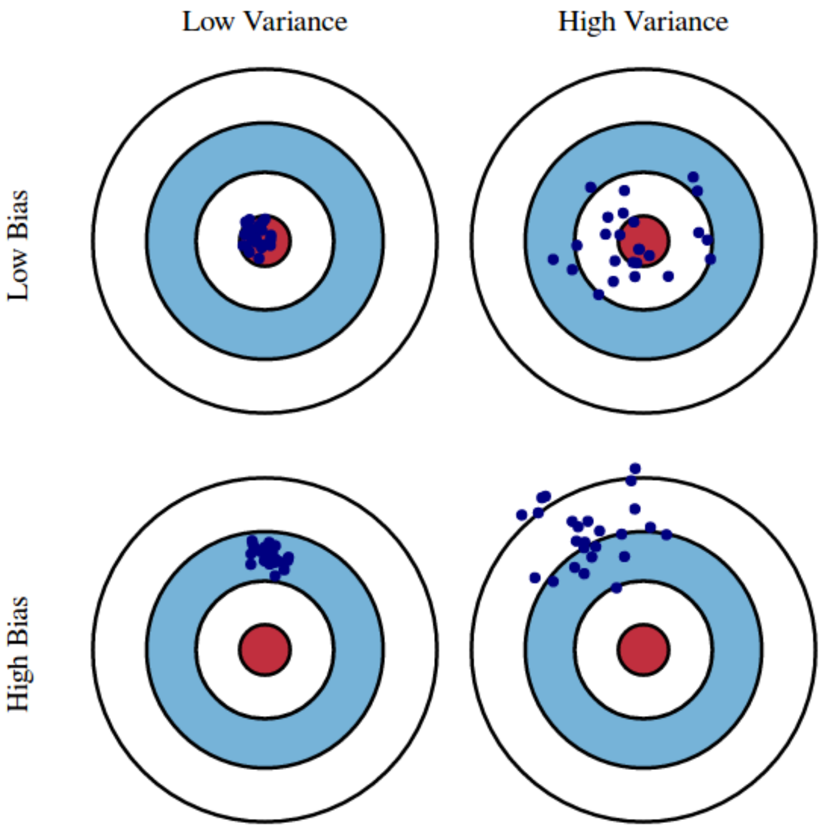


Fig. 1 Graphical illustration of bias and variance.

Human error	Train error	Dev error	Test error	Problems
1%	5%	5.2%	5.3%	High bias
1%	1%	1.1%	6.4%	High variance
1%	5%	5.2%	10%	High bias and High variance
1%	1%	1.1%	1.2%	-

Avoiding underfitting (high bias)

- Make sure you have chosen a good learning rate;
- Train a bigger (more complex model):
  - Try adding more layers to the network;
  - Try adding more hidden units to the layer;
- Different network architectures perform better for certain applications:
  - E.g. Convolutional Neural Networks for Computer Vision (see later slides).

Avoiding overfitting in neural networks.

- Get more examples:
  - More examples in the training set reduces overfitting;
  - More examples in the development set makes it harder to overfit to the validation (i.e. to tune your parameters with a few examples in mind);
- L1 or L2 regularization;
- Dropout.
- More (e.g. data augmentation).

L2 and L1 regularization (1)

- L2 regularization for L-layer network:
  - $J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|_2^2;$
  - The L2-norm of the weights is: $\|W\|_2^2 = \sum_{i=1}^{n} w_i^2;$
  - λ is the regularization constant, it is tuneable hyper-parameter that determines the importance of the regularization term in the cost function.
- The optimization reduces the norm of the weights, some of the weights are close to zero, which simplifies the network;
- L1 regularization for L-layer network :
  - $J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) + \frac{\lambda}{m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|_1;$
  - $\|W\|_1 = \sum_{i=1}^{n}|w_i|;$
  - Same as L2 regularization but also induces sparsity by forcing some of the weights to zero.

L1 and L2 regularization (2)

- Backpropagation:
  - L2: $dW^{[l]} = (same\ as\ before) + \frac{\lambda}{m} W^{[l]}$
  - L1: $dW^{[l]} = (same\ as\ before) + \frac{\lambda}{m} sign(W^{[l]})$
- Nothing stops you from adding regularization on a layer by layer basis, each one with its unique type and regularization constant. In this case you must calculate the derivatives accordingly, or use a programming framework.

Dropout

- During training randomly eliminate nodes from the network with a given probability;
- Most common type “Inverted Dropout”:
  - d = np.random.rand(a.shape[0], a.shape[1]) < keep_prob
  - a = np.multiply(a, d)
  - a= a / keep_prob
- Only apply during training!
- Backpropagation and prediction don’t change;
- Works because at each iteration of gradient descent you are training a smaller network. The network can never rely on one single node.

Improvements on Gradient Descent

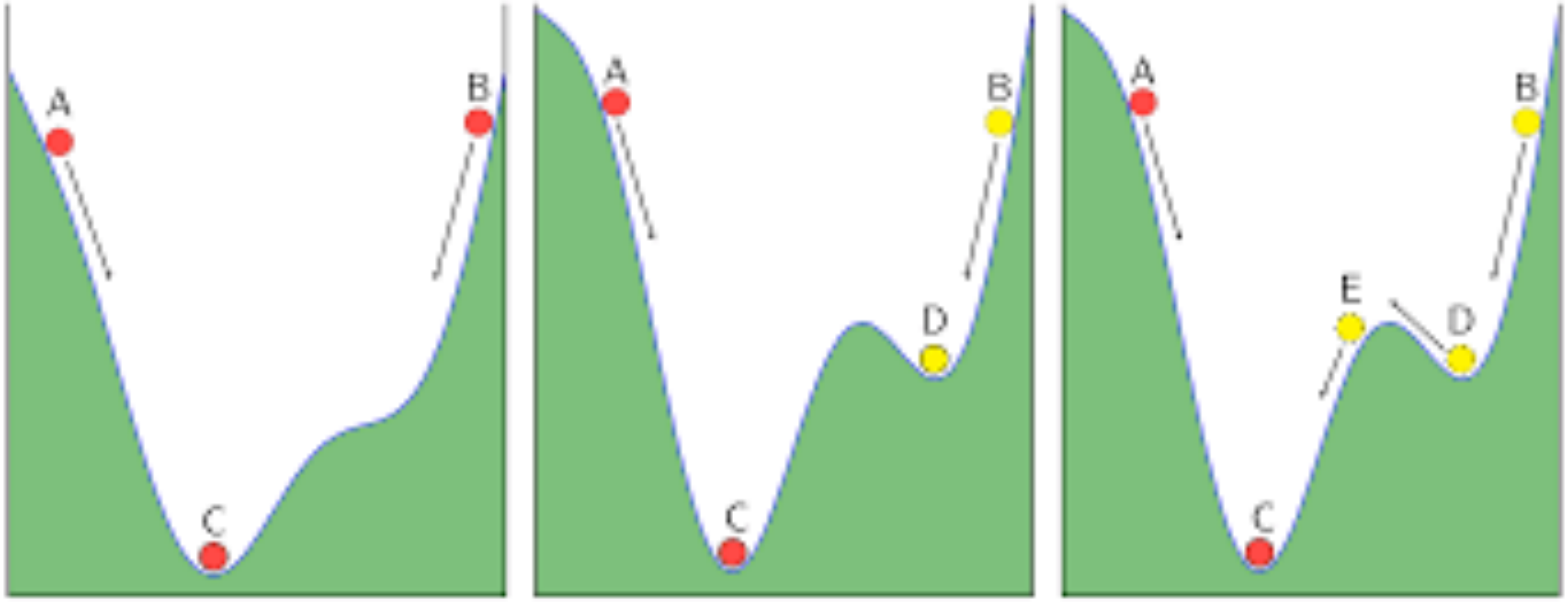
- Gradient Descent with Momentum;
- RMSprop;
- Adam optimizer.

Gradient descent with momentum (1)

- Initialize the moving averages v_{dW} and v_{db} to matrices/arrays of zeros;
- At each iteration of gradient descent:
  - $v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) dW$
  - $v_{db} = \beta_1 v_{db} + (1 - \beta_1) db$
- β_1 is an hyper-parameter (usually 0.9 no need to tune it);
- Bias correction (optional step not really necessary but more correct):
  - $v_{dW} = \frac{v_{dW}}{1-{\beta_1}^t}$
  - $v_{db} = \frac{v_{db}}{1-{\beta_1}^t}$
- The update step used the moving averages of the gradients instead of the gradients:
  - $W = W - \alpha\, v_{dw}$
  - $b = b - \alpha\, v_{db}$

Gradient descent with momentum (2)

- Makes gradient descent converge faster;
- Makes gradient descent more robust to local minima and saddle points;
- Makes gradient descent more robust to hyper-parameter choices.

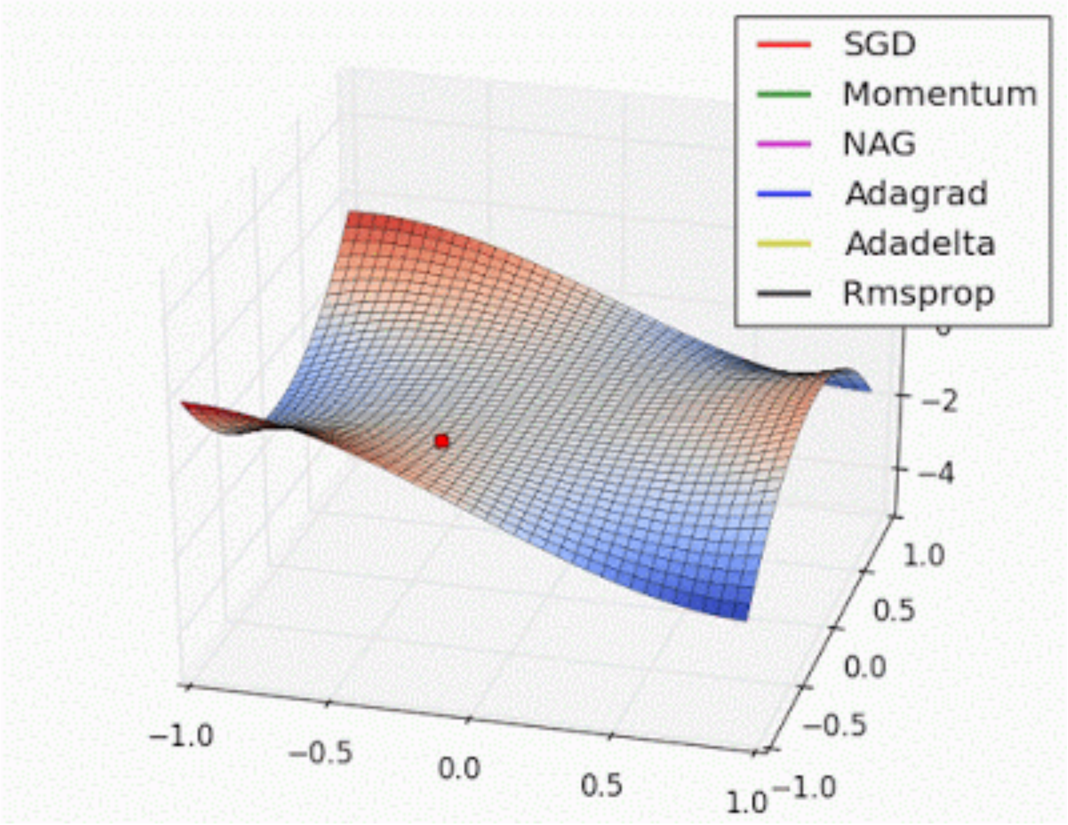


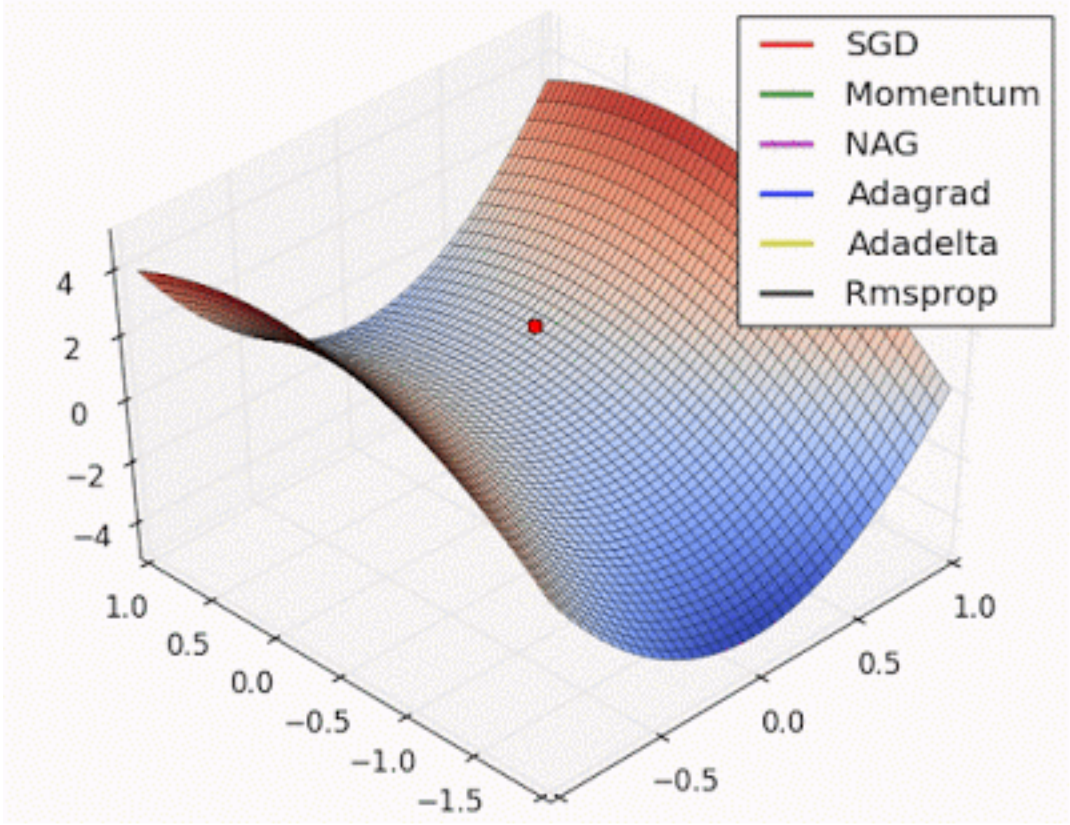
a) GD b) GD c) GD with momentum

RMSprop (1)

- Initialize the moving averages s_{dW} and s_{db} to matrices/arrays of zeros;
- At each iteration of gradient descent:
  - $s_{dW} = \beta_2 s_{dW} + (1 - \beta_2) dW^2$
  - $s_{db} = \beta_2 s_{db} + (1 - \beta_2) db^2$
- β_2 is an hyper-parameter (usually 0.999 no need to tune it);
- Bias correction (optional step)
- The update step used the moving averages of the gradients instead of the gradients:
  - $W = W - \alpha \frac{dW}{\sqrt{s_{dw}}}$
  - $b = b - \alpha \frac{db}{\sqrt{s_{db}}}$
- **Dampens oscillations.**

Ignore Adagrad, Adadelta and NAG;





Adam optimizer

- Combines momentum with RMSprop:
  - $W = W - \alpha \frac{v_W}{\sqrt{s_{dw}}}$;
  - $b = b - \alpha \frac{v_b}{\sqrt{s_{db}}}$;
- β_1 and β_2 are 0.9 and 0.999 respectively (no need to tune);
- Has become the deep learning standard.

Convolutional Neural Networks

What We See

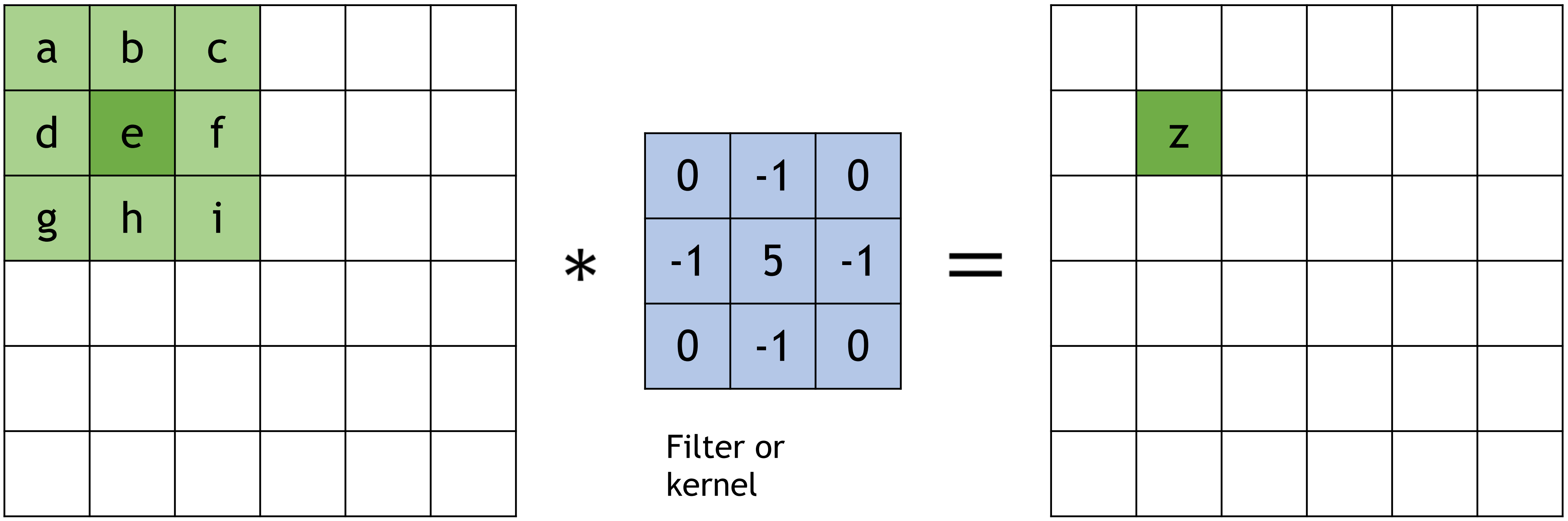
08	02	22	97	38	15	00	40	00	75	04	05	07	78	52	12	50	77	91	08
49	49	99	40	17	81	18	57	60	87	17	40	98	43	69	48	04	56	62	00
81	49	31	73	55	79	14	29	93	71	40	67	53	88	30	03	49	13	36	65
52	70	95	23	04	60	11	42	69	24	68	56	01	32	56	71	37	02	36	91
22	31	16	71	51	67	63	89	41	92	36	54	22	40	40	28	66	33	13	80
24	47	32	60	99	03	45	02	44	75	33	53	78	36	84	20	35	17	12	50
32	98	81	28	64	23	67	10	26	38	40	67	59	54	70	66	18	38	64	70
67	26	20	68	02	62	12	20	95	63	94	39	63	08	40	91	66	49	94	21
24	55	58	05	66	73	99	26	97	17	78	78	96	83	14	88	34	89	63	72
21	36	23	09	75	00	76	44	20	45	35	14	00	61	33	97	34	31	33	95
78	17	53	28	22	75	31	67	15	94	03	80	04	62	16	14	09	53	56	92
16	39	05	42	96	35	31	47	55	58	88	24	00	17	54	24	36	29	85	57
86	56	00	48	35	71	89	07	05	44	44	37	44	60	21	58	51	54	17	58
19	80	81	68	05	94	47	69	28	73	92	13	86	52	17	77	04	89	55	40
04	52	08	83	97	35	99	16	07	97	57	32	16	26	26	79	33	27	98	66
88	36	68	87	57	62	20	72	03	46	33	67	46	55	12	32	63	93	53	69
04	42	16	73	38	25	39	11	24	94	72	18	08	46	29	32	40	62	76	36
20	69	36	41	72	30	23	88	34	62	99	69	82	67	59	85	74	04	36	16
20	73	35	29	78	31	90	01	74	31	49	71	48	86	81	16	23	57	05	54
01	70	54	71	83	51	54	69	16	92	33	48	61	43	52	01	89	19	67	48

What Computers See

Motivation

- Suppose you want to build a cat classifier:
  - It outputs 1 if the picture is a cat and 0 otherwise.
- If you have low resolution RGB images a fully connected neural network might work fine, e.g. 64x64x3 = 12288 weights;
- If you have high resolution RGB images the number of parameters increases exponentially, e.g. 1000x1000x3 = 3 million weights:
  - This can very easily lead to overfitting, and very slow training;
- In addition, different pixels of the image are not completely different features;

Image Convolution (1)



$$z = 0\times a + (-1)\times b + 0\times c + (-1)\times d + 5\times e + (-1)\times f + 0\times g + (-1)\times h + 0\times i$$

Image Convolution (2)

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

$*$

0	-1	0
-1	5	-1
0	-1	0

Filter or kernel

$=$

	3				

$$3 = 1\times0 + 2\times(-1) + 4\times0 + 0\times(-1) + 1\times5 + 0\times(-1) + 1\times0 + 0\times(-1) + 3\times0$$

Image Convolution (3)

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

*

0	-1	0
-1	5	-1
0	-1	0

Filter or kernel

=

	3	-8			

Image Convolution (4)

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

*

0	-1	0
-1	5	-1
0	-1	0

Filter or kernel

=

	3	-8	-4		

Image Convolution (5)

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

*

0	-1	0
-1	5	-1
0	-1	0

Filter or kernel

=

	3	-8	-4	1	

Image Convolution (6)

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

*

0	-1	0
-1	5	-1
0	-1	0

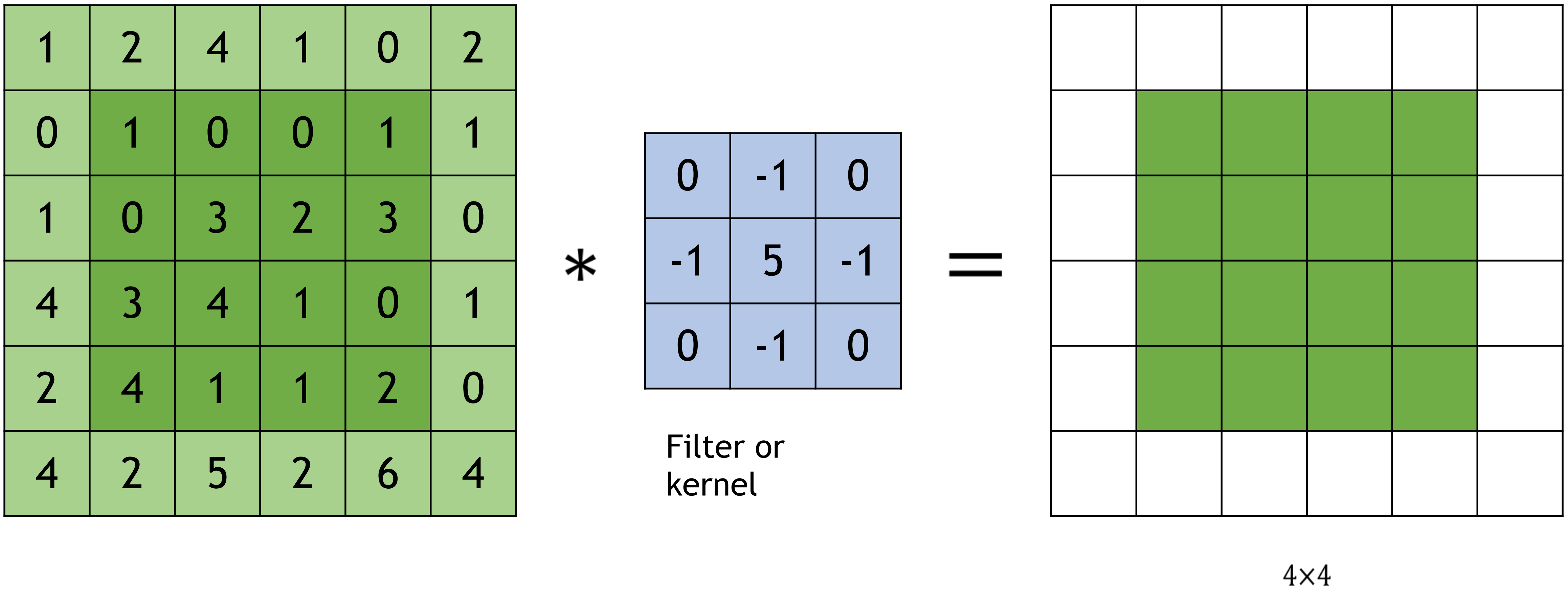
Filter or kernel

=

	3	-8	-4	1	
	-8				

And so on...

Valid Padding (no padding)



Same Padding (1)

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 4 & 1 & 0 & 2 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 3 & 2 & 3 & 0 & 0 \\
0 & 4 & 3 & 4 & 1 & 0 & 1 & 0 \\
0 & 2 & 4 & 1 & 1 & 2 & 0 & 0 \\
0 & 4 & 2 & 5 & 2 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
*
\underbrace{\begin{bmatrix}
0 & -1 & 0 \\
-1 & 5 & -1 \\
0 & -1 & 0
\end{bmatrix}}_{\text{Filter or kernel}}
=
\underbrace{\begin{bmatrix}
\; & \; & \; & \; & \; & \; \\
\; & \; & \; & \; & \; & \; \\
\; & \; & \; & \; & \; & \; \\
\; & \; & \; & \; & \; & \; \\
\; & \; & \; & \; & \; & \; \\
\; & \; & \; & \; & \; & \;
\end{bmatrix}}_{6\times6}
$$

Same Padding (2)

- In Same padding **size output = size input;**
- Input image has size $n \times n$;
- The convolution filter has size $f \times f$;
- p is number of pixels to pad for in each direction;
- Then: $n + 2p - f + 1 = n \Leftrightarrow p = \frac{f-1}{2}$
- f is usually odd, if f is even you need asymmetric padding.

Image Convolution

- Types of filters (kernels):
  - Sharpen
  - Blur
  - Emboss
  - Outline
  - Bottom Sobel
  - Top Sobel
  - Right Sobel
  - Left Sobel
  - Etc..

http://setosa.io/ev/image-kernels/

Strided Convolutions (1)

Strided convolution with stride 2 in both directions
Patch size 3x3

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 4 & 1 & 2 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 3 & 2 & 0 & 0 \\
0 & 2 & 4 & 1 & 1 & 0 & 0 \\
0 & 4 & 2 & 5 & 2 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
*
\underbrace{\begin{bmatrix}
0 & -1 & 0 \\
-1 & 5 & -1 \\
0 & -1 & 0
\end{bmatrix}}_{\text{Filter or kernel}}
=
\begin{bmatrix}
3 & & \\
 & & \\
 & &
\end{bmatrix}
$$

Strided Convolutions (2)

$$\begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 2 & 4 & 1 & 2 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 3 & 2 & 0 & 0 \\ \hline 0 & 2 & 4 & 1 & 1 & 0 & 0 \\ \hline 0 & 4 & 2 & 5 & 2 & 4 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} * \underbrace{\begin{array}{|c|c|c|}\hline 0 & -1 & 0 \\ \hline -1 & 5 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array}}_{\text{Filter or kernel}} = \begin{array}{|c|c|c|}\hline 3 & 17 & \\ \hline & & \\ \hline & & \\ \hline \end{array}$$

Strided Convolutions (3)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 4 & 1 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 3 & 2 & 0 & 0 \\ 0 & 2 & 4 & 1 & 1 & 0 & 0 \\ 0 & 4 & 2 & 5 & 2 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \underbrace{\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}}_{\text{Filter or kernel}} = \begin{bmatrix} 3 & 17 & 8 \\ & & \\ & & \end{bmatrix}$$

Strided Convolutions (4)

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

$*$

0	-1	0
-1	5	-1
0	-1	0

Filter or kernel

$=$

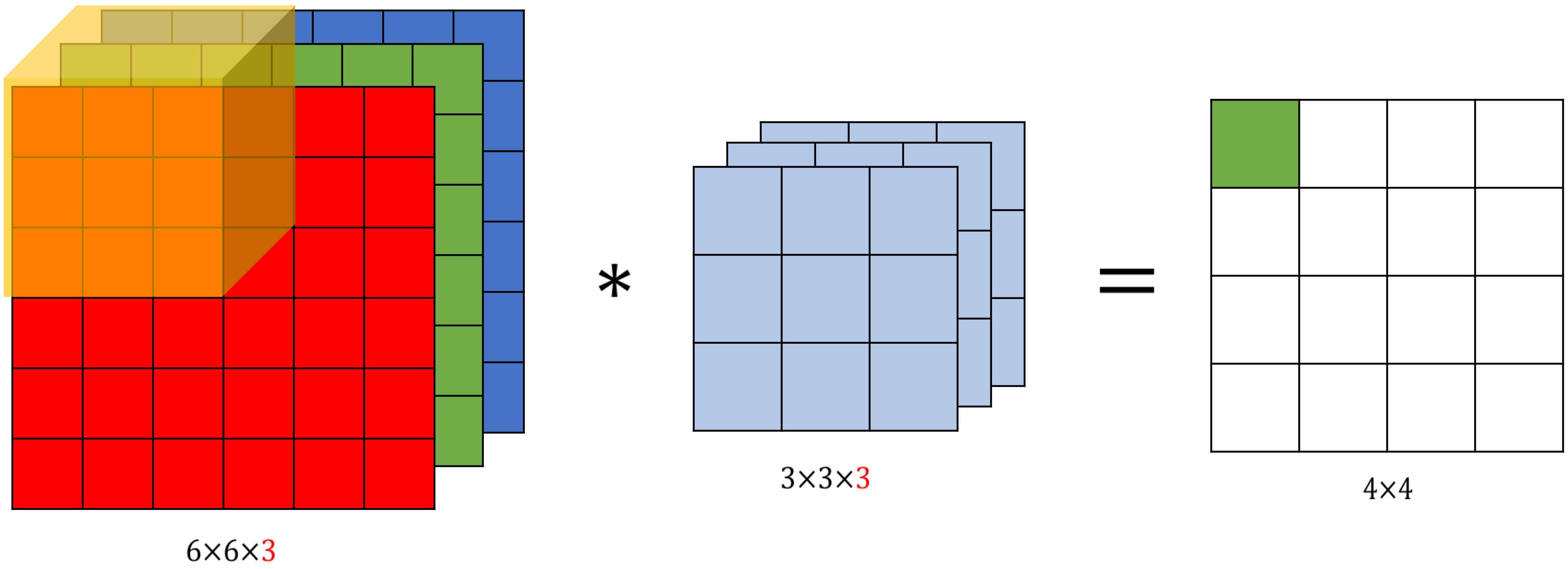
3	17	8
3		

And so on...

Strided convolutions (5)

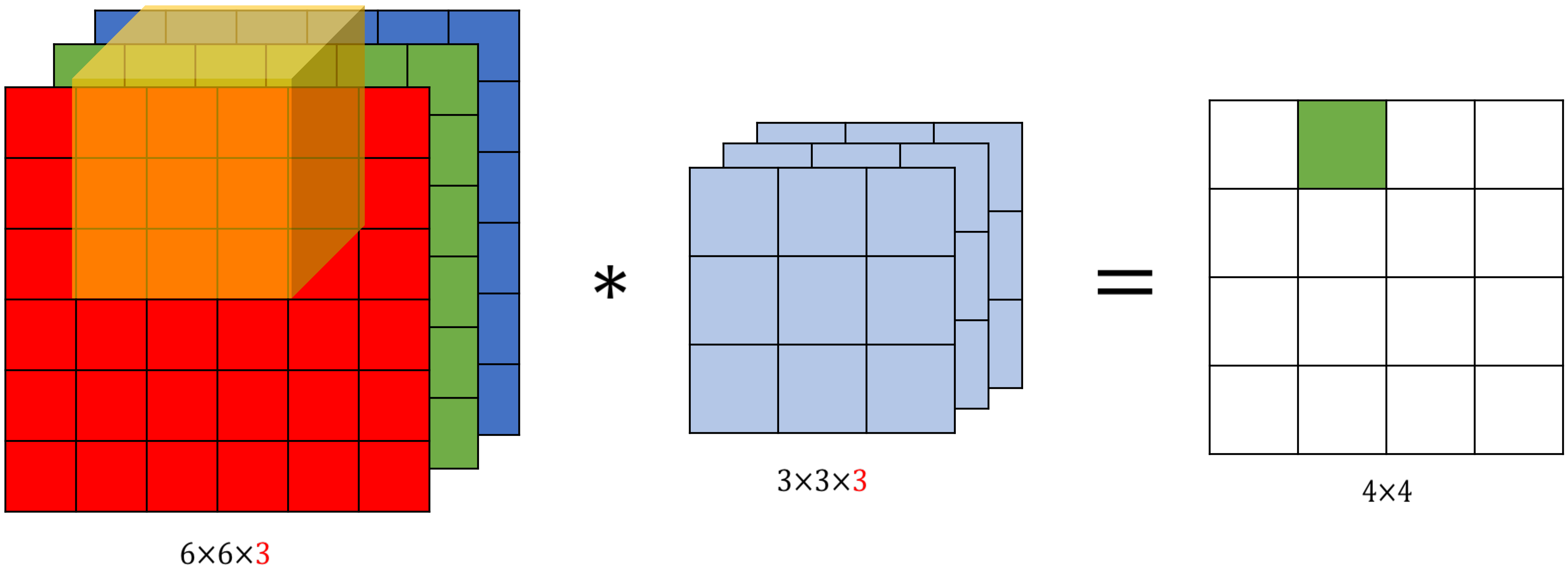
- $ouput\ size = \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$, where s is the stride;
- Strided convolutions run faster than regular ones since they require less operations;
- Useful for very high resolution images;

Convolutions with multiple input channels (RGB images) (1)



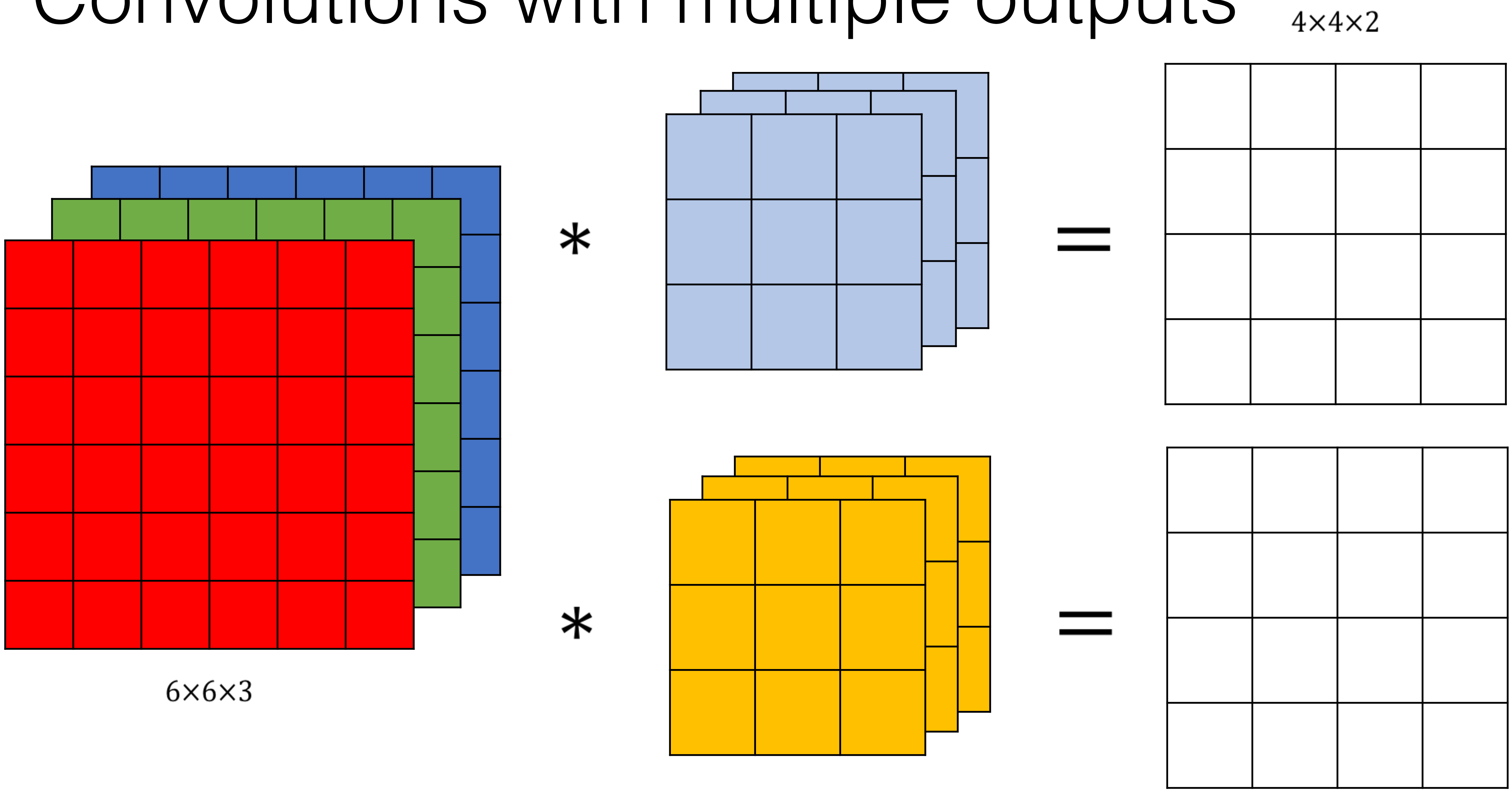
The red 3 is the number of input channels

Convolutions with multiple input channels (RGB images) (2)



The red 3 is the number of input channels

Convolutions with multiple outputs



http://cs231n.github.io/assets/conv-demo/index.html

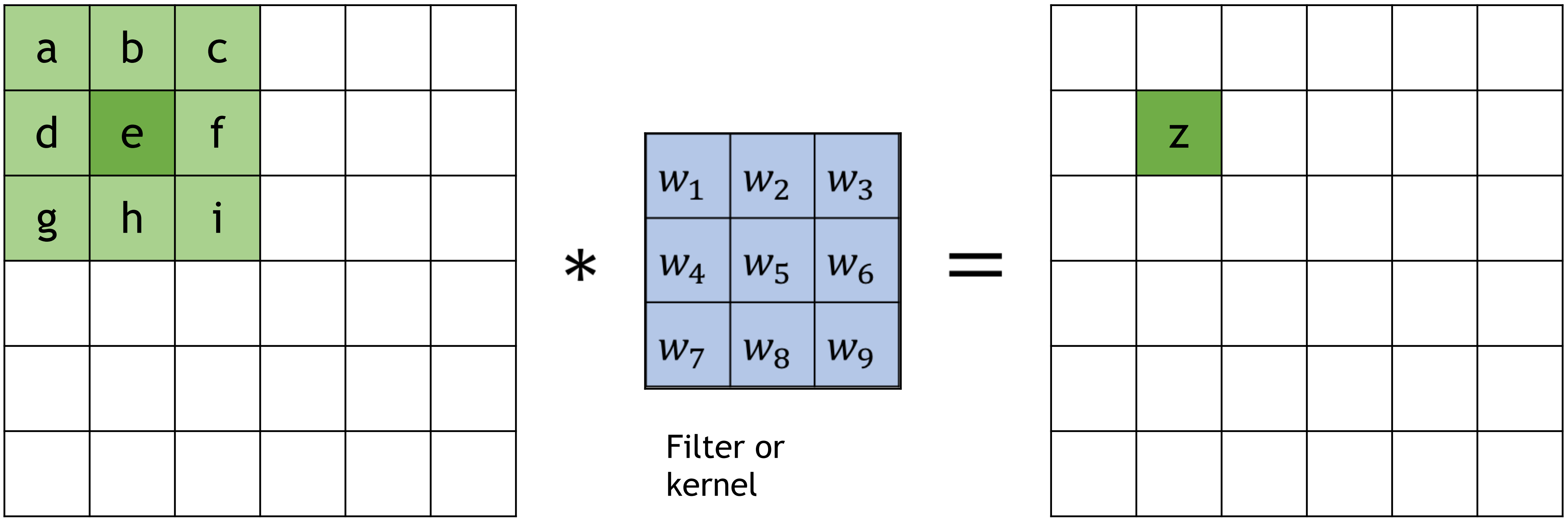
Example

- Same padding;
- Input: 6x6 RGB image (6x6x3);
- Filters: twelve 3x3x3 filters;
- Output: twelve 6x6 images;
- The number of filters in each convolution layer is an hyper-parameter you choose;

Convolution Layer (1)

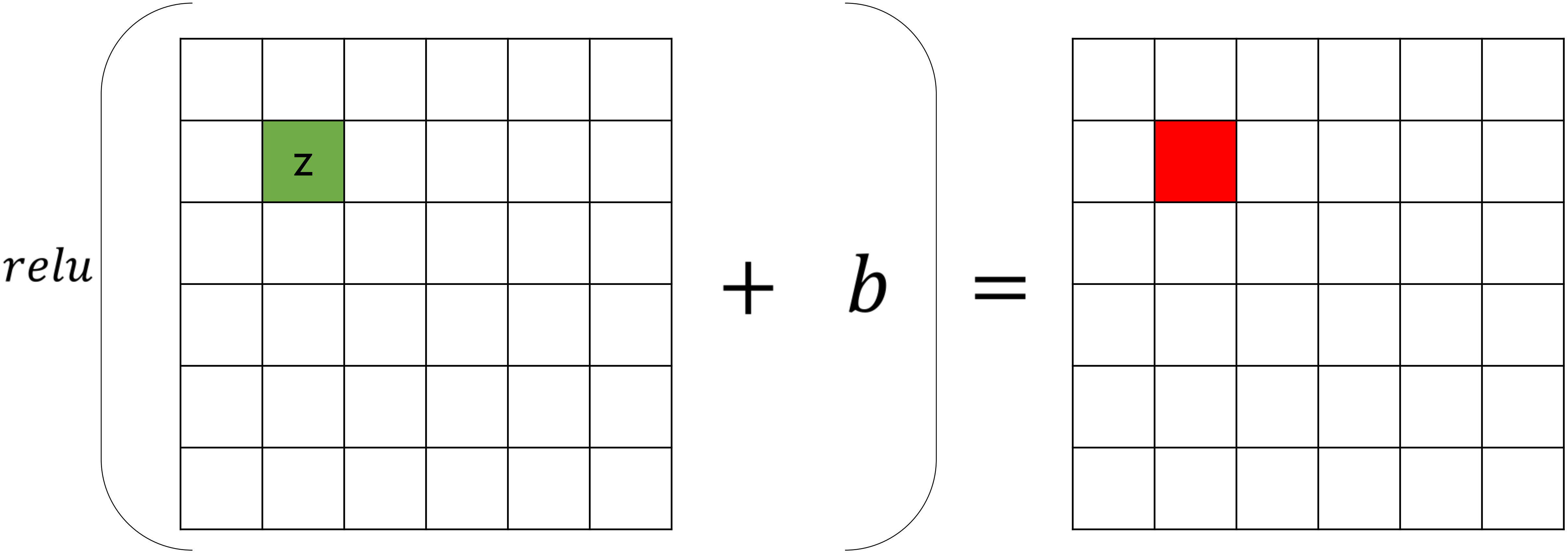
- The are filters are **learnt** from data, not fixed hyper-parameters;
- The filter (kernel) of the convolution is a patch made up of weights that are **trainable**;
  - The patch size is an hyper-parameter (e.g. 3x3, 5x5, 7x7);
- You must add **biases** and apply an **activation function** like in a normal neural network;

Convolution Layer (2)



$$z = relu(w_1 a + w_2 b + w_3 c + w_4 d + w_5 e + w_6 f + w_7 g + w_8 h + w_9 i + b)$$

Convolution Layer (3)



$$activation = relu(w_1 a + w_2 b + w_3 c + w_4 d + w_5 e + w_6 f + w_7 g + w_8 h + w_9 i + b)$$

Convolution Layer (4)

- Each layer of the CNN can use more than one filter;
- Each filter has its own weights and biases;
- The convolution operation and the backpropagation are already implemented in Tensorflow and other programming frameworks, so we will skip the math of backpropagation;
- If you have 10 3x3x3 filters you have 280 parameters (10x(3x3x3+1)), regardless of the size of the image;

Notation of convolution layer l

- $f^{[l]}$ = filter size
  $p^{[l]}$ = padding
  $s^{[l]}$ = stride
  $n_c{}^{[l]}$ = number of filters

Each filter is: $f^{[l]} \times f^{[l]} \times n_c{}^{[l-1]}$

Activation: $a^{[l]} \rightarrow n_H{}^{[l]} \times n_W{}^{[l]} \times n_c{}^{[l]}$

Weights: $W^{[l]} \rightarrow f^{[l]} \times f^{[l]} \times n_c{}^{[l-1]} \times n_c{}^{[l]}$

Biases: $b^{[l]} \rightarrow 1 \times 1 \times 1 \times n_c{}^{[l]}$

Input: $n_H{}^{[l-1]} \times n_W{}^{[l-1]} \times n_c{}^{[l-1]}$

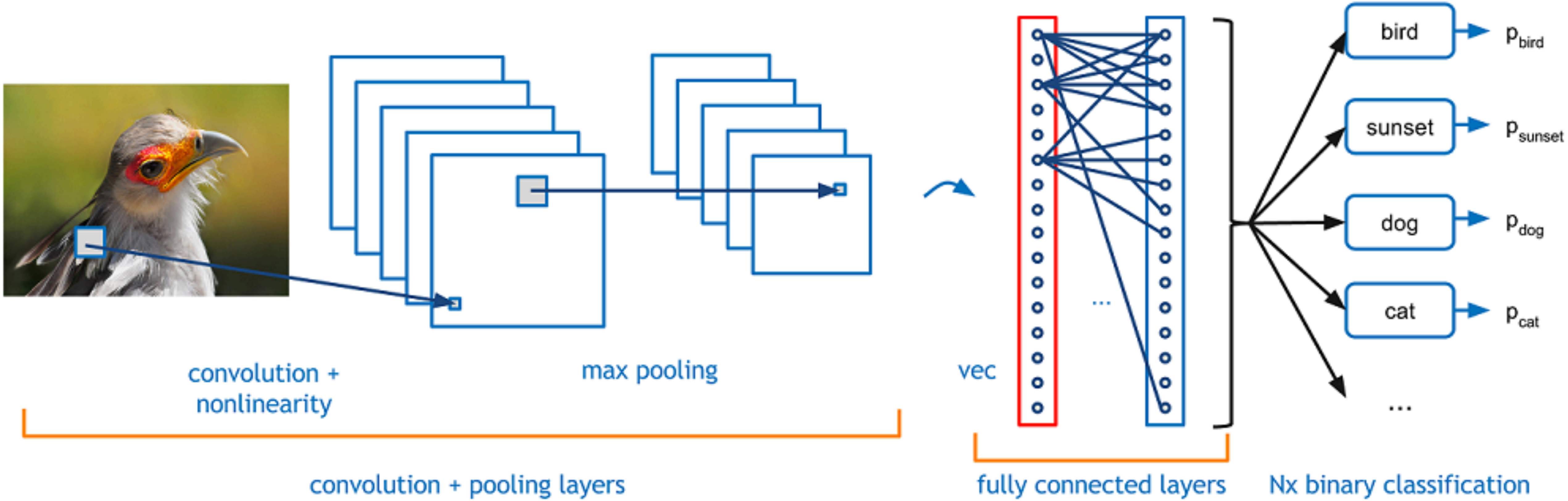
Output: $n_H{}^{[l]} \times n_W{}^{[l]} \times n_c{}^{[l]}$

$$n_H{}^{[l]} = \left\lfloor \frac{n_H{}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

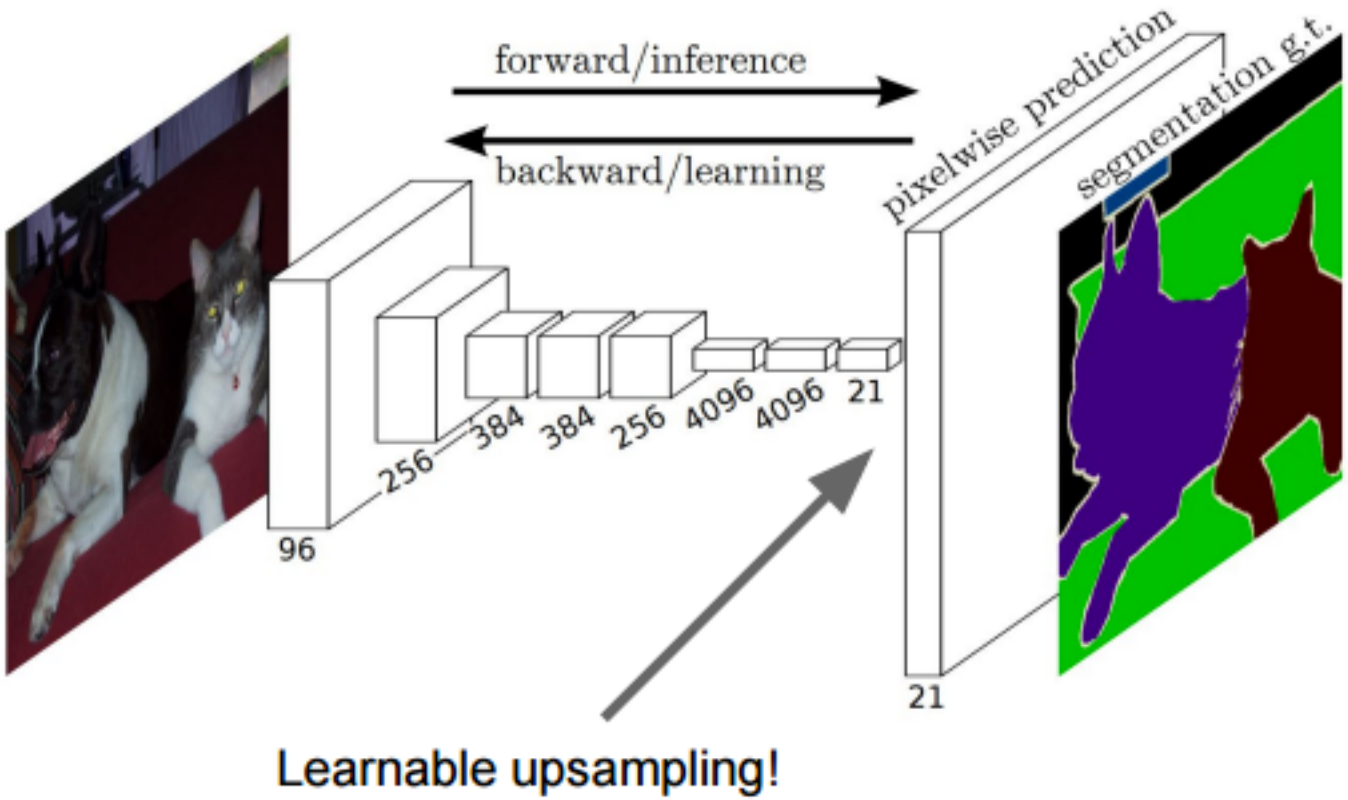
$$n_W{}^{[l]} = \left\lfloor \frac{n_W{}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

$$A^{[l]} \rightarrow m \times n_H{}^{[l]} \times n_W{}^{[l]} \times n_c{}^{[l]}$$

Convolution Neural Network (CNN)



Fully Convolutional Neural Network (FCNN)



Useful Links

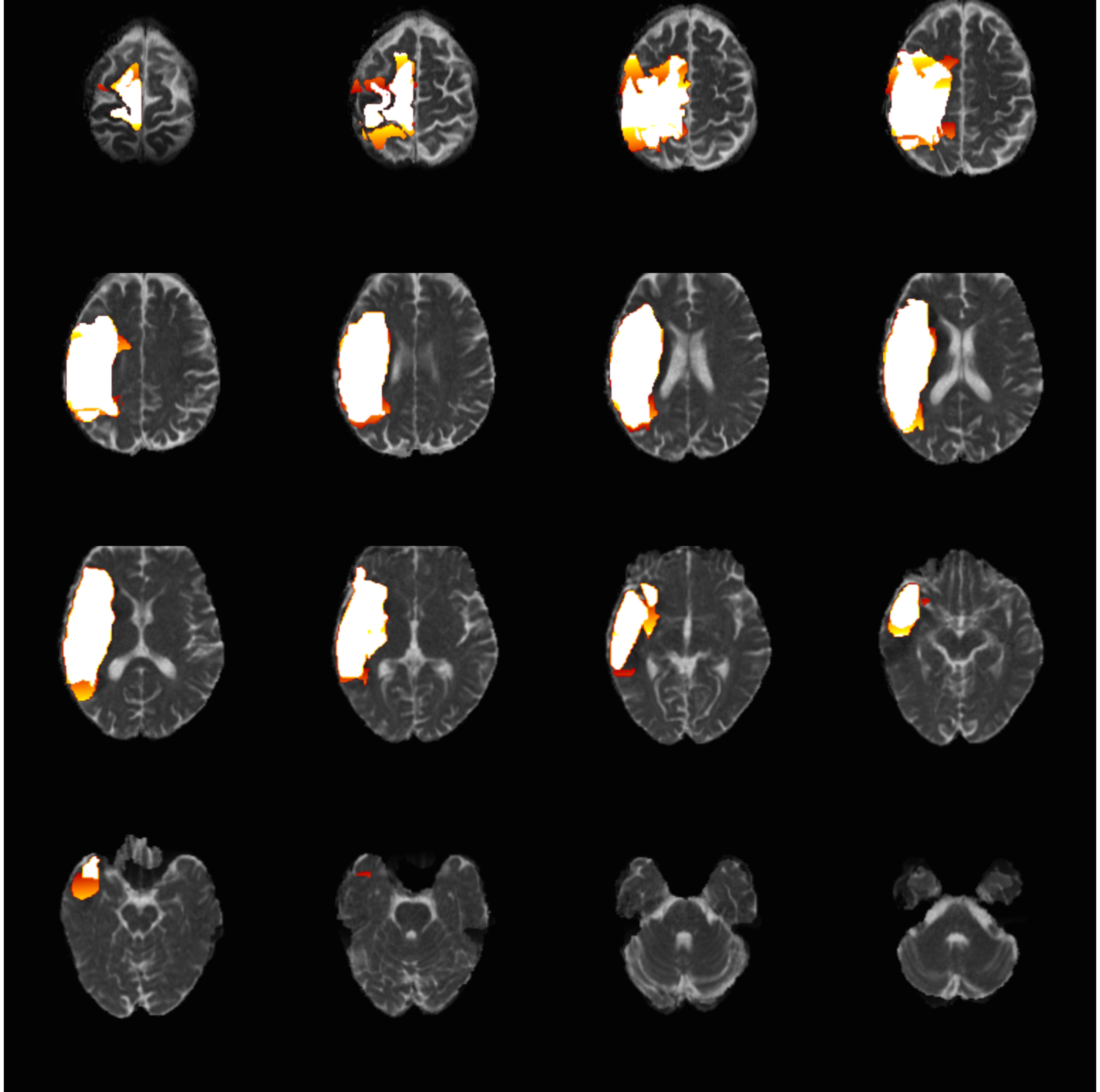
- http://cs231n.github.io/convolutional-networks/
- https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

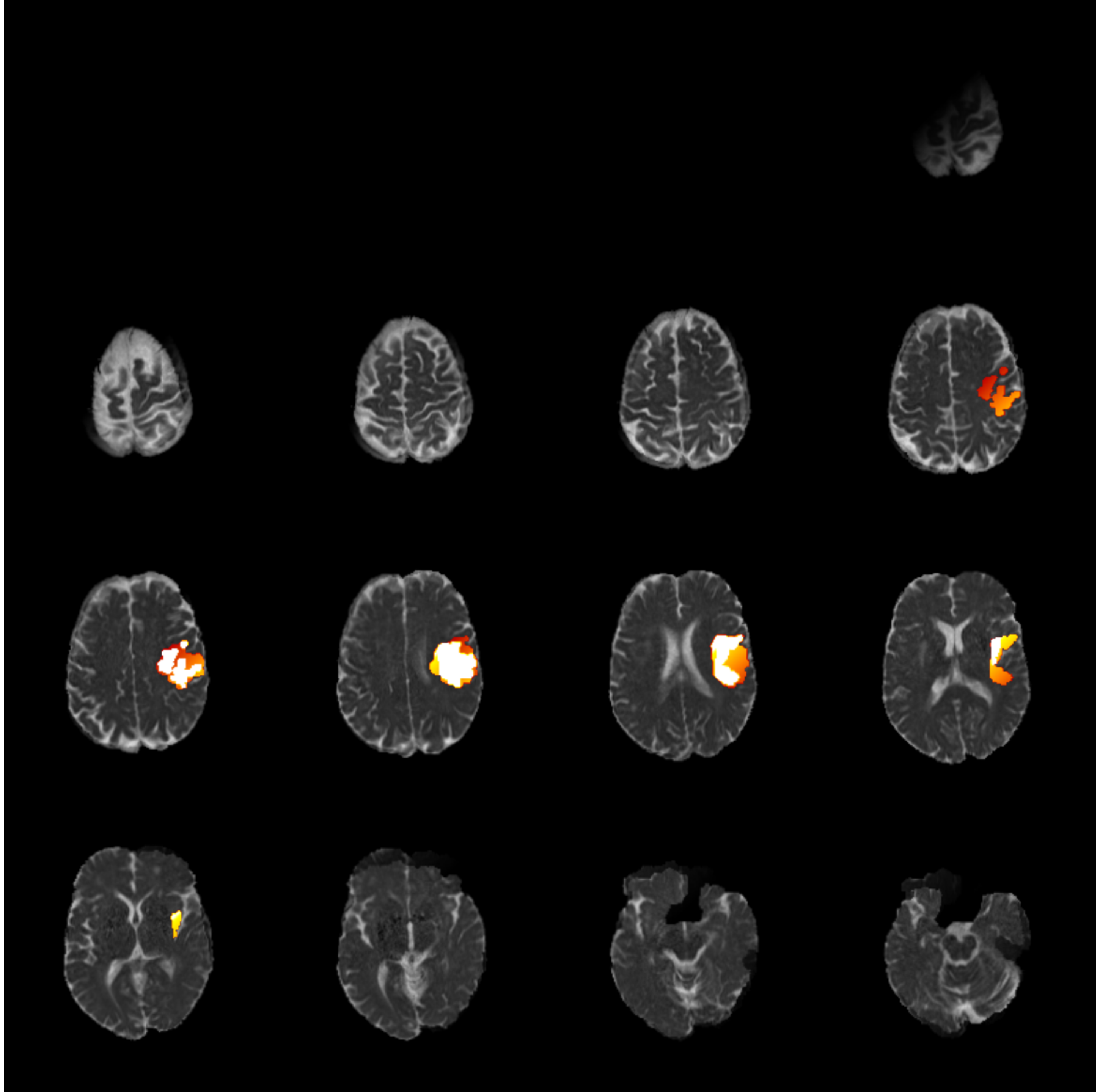
Stroke Lesion Segmentation Prediction using Fully Convolutional Neural Networks

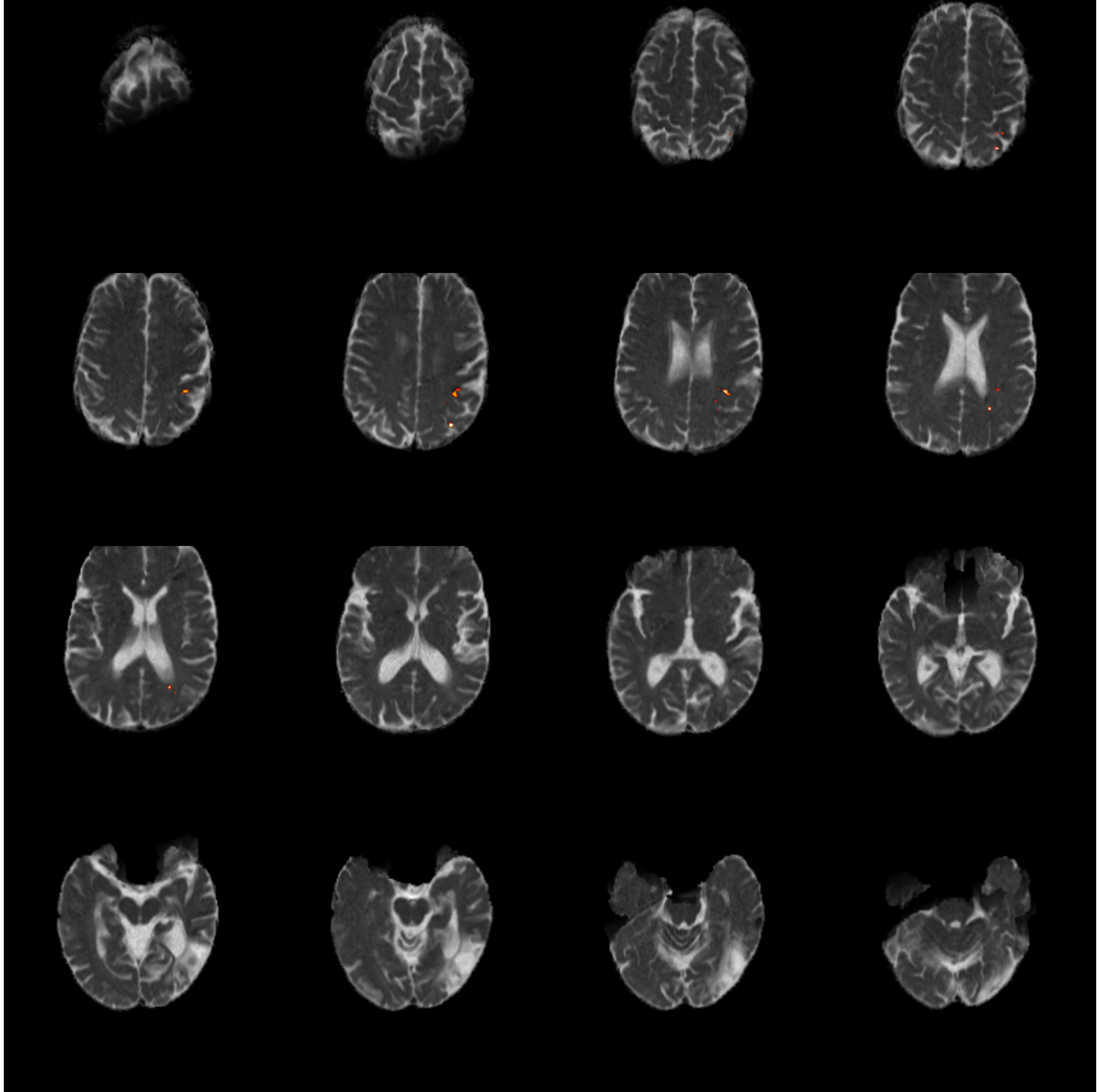
Setup

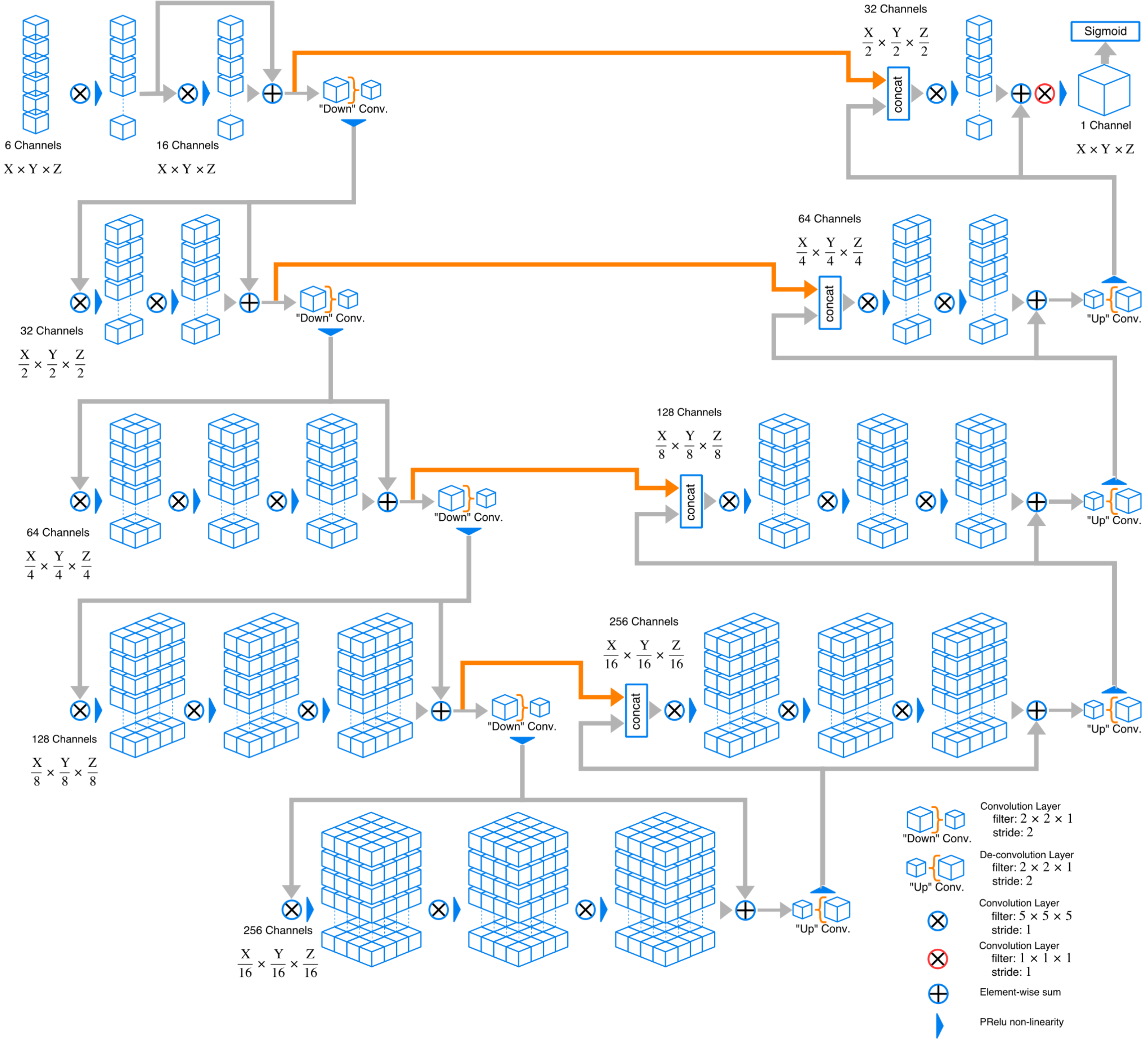
- Goal: To predict the outcome segmentation of a stroke lesion 90 days after the stroke had occurred using only MRI data collected the day of the stroke;
- Motivation: doctors can see the lesion just fine immediately after the stroke, what is harder is to know how the lesion will evolve over time;
- Data: 43 patients: 3D images with 6 channels each, but with varying spatial dimensions $(width \times height \times depth \times 6)$
- The ground-truth segmentation used for training was done by experts;
- Architecture: "V-Net" a FCNN used for 3D medical image.

Shape (x,y,z)	Count
(128, 128, 25)	10
(192, 192, 19)	17
(192, 192, 24)	1
(192, 192, 30)	6
(256, 256, 24)	9
Total	43









6 Channels
X × Y × Z
16 Channels
X × Y × Z
"Down" Conv.
32 Channels
X/2 × Y/2 × Z/2
concat
Sigmoid
1 Channel
X × Y × Z
64 Channels
X/4 × Y/4 × Z/4
"Up" Conv.
128 Channels
X/8 × Y/8 × Z/8
256 Channels
X/16 × Y/16 × Z/16
Convolution Layer
filter: 2 × 2 × 1
stride: 2
De-convolution Layer
filter: 2 × 2 × 1
stride: 2
Convolution Layer
filter: 5 × 5 × 5
stride: 1
Convolution Layer
filter: 1 × 1 × 1
stride: 1
Element-wise sum
PRelu non-linearity

Training

- Using TensorFlow and Google Cloud ML Engine to take advantage of parallelizing training across multiple GPUs;
- The spatial dimensions of the images were not scaled to the same size in order to avoid distortion, however, this means SGD had to be used (train with one example per gradient descent step);
- 5-fold cross validation due to only having 43 examples;

Evaluation Metrics

- Dice Coefficient (DC):
  - The fraction of overlap between the ground-truth segmentation and the prediction;
  - A number between 0 and 1, being that 1 corresponds to a perfect segmentation;
- Hausdorff Distance (HD):
  - Measures the presence of outliers in the segmentation;
- Average Symmetric Surface Distance (ASSD):
  - Measures the overall surface deformity between the ground-truth and prediction.

Loss function (only for one example)

- n denotes the voxel number and not example number (voxel = 3D pixel);
- Cross entropy **loss** function:

$$cross\ entorpy = -\frac{1}{\#voxels}\sum_{n=1}^{\#voxels} -(y_n \log(\widehat{y_n}) + (1-y_n)\log(1-\widehat{y_n}))$$

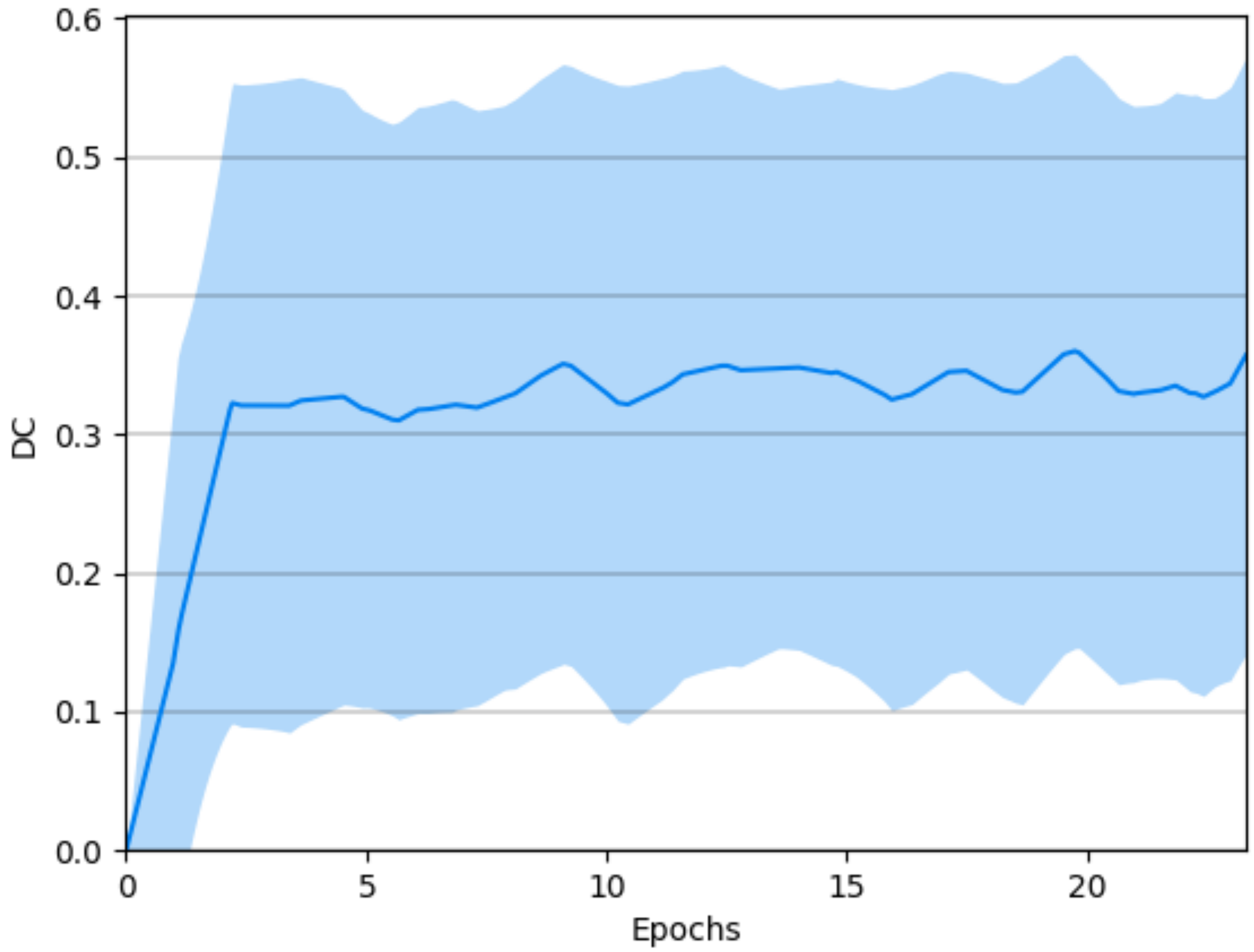
- Dice Loss:

$$dice\ loss = -\frac{2\sum_{n=1}^{\#voxels} y_n\widehat{y_n}}{\sum_{n=1}^{\#voxels} y_n + \sum_{n=1}^{\#voxels}\widehat{y_n}}$$

- Our loss function:

$$loss = cross\ entropy + \ dice\ loss$$

Evolution of DC during training

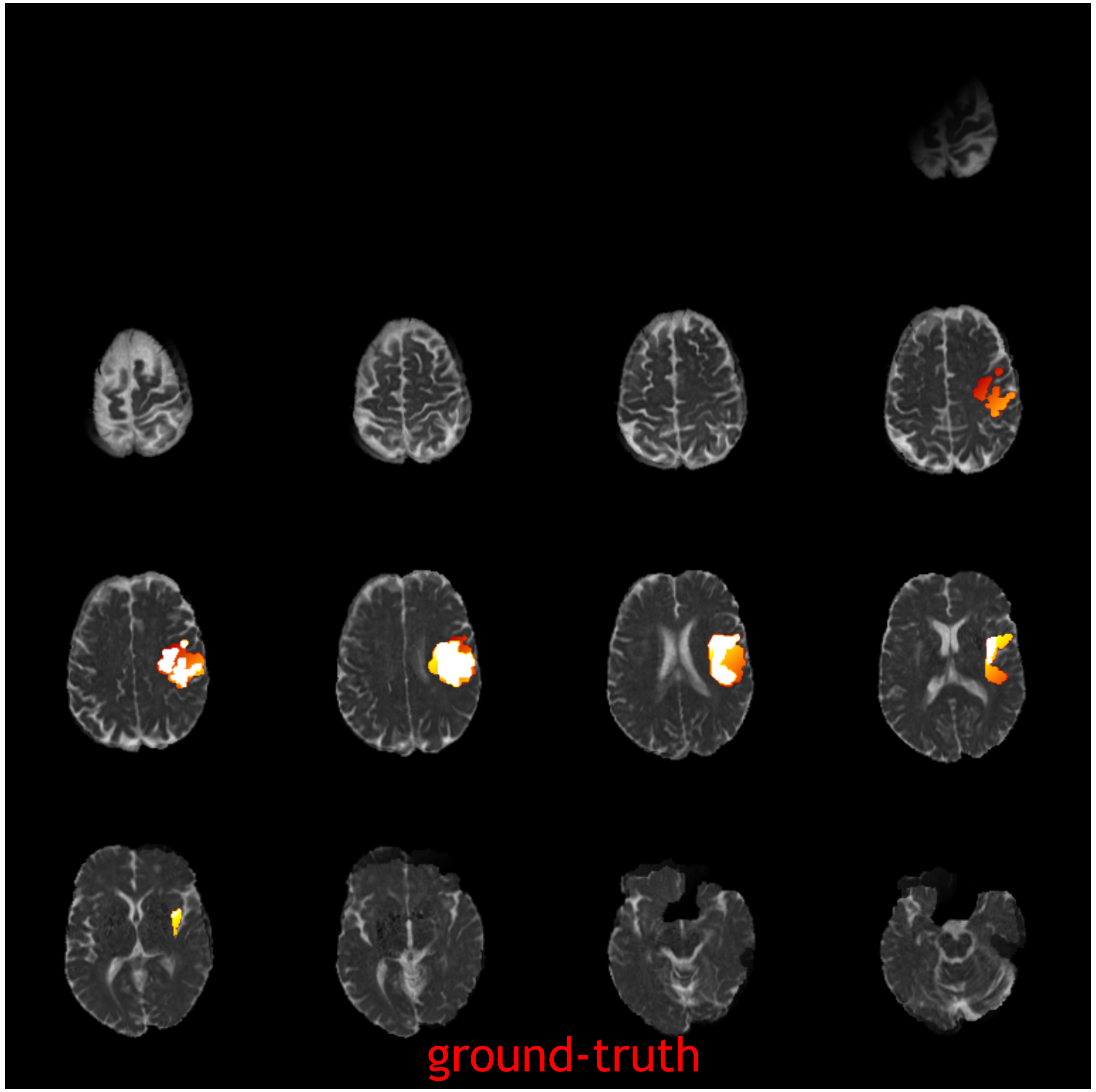


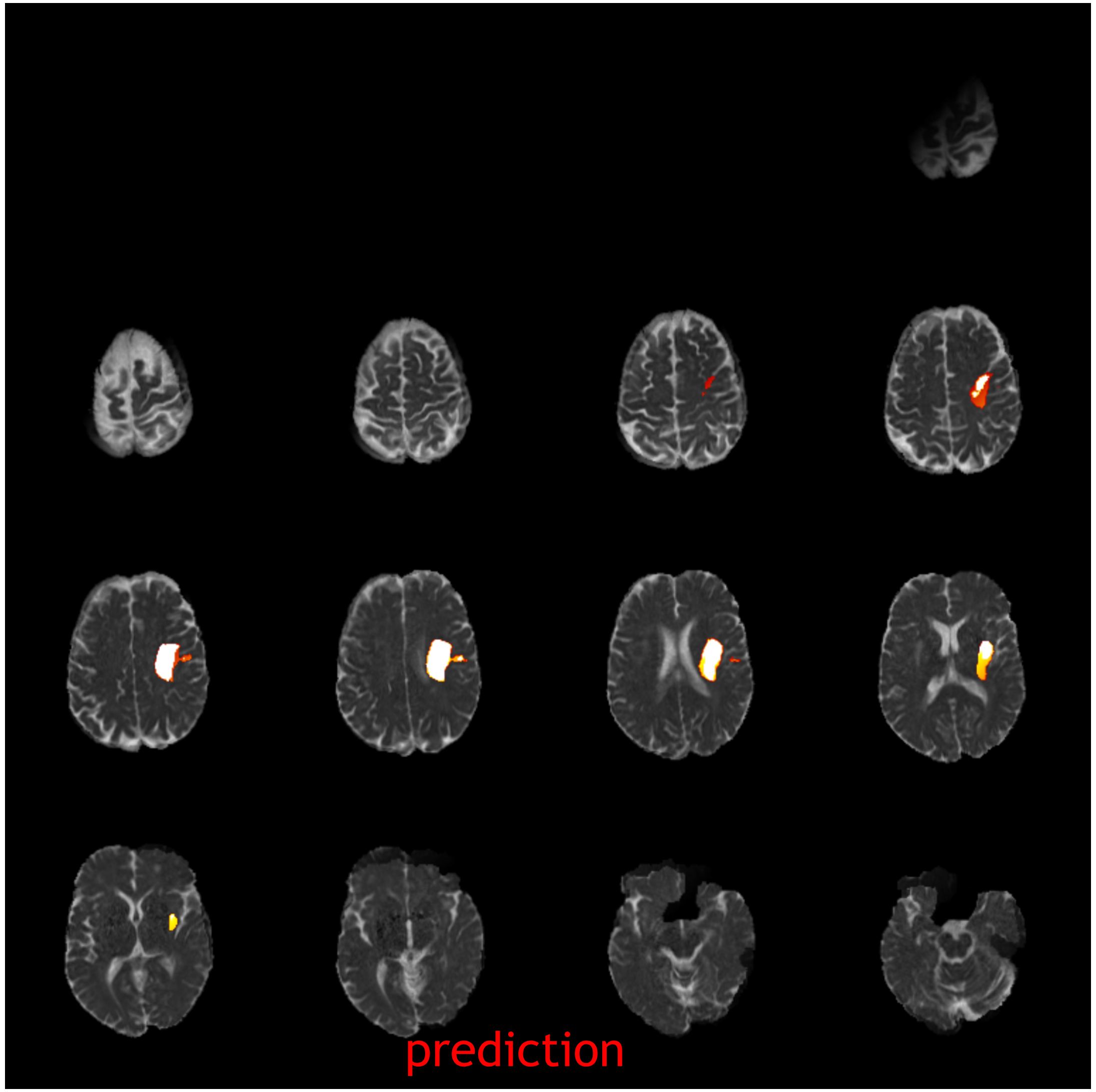
Results

	Raw Prediction	Post-Processed	Mean Gain
DC	0.357 ± 0.216	$\mathbf{0.370 \pm 0.215}$	3.662%
HD	30.823 ± 18.512	$\mathbf{23.398 \pm 18.753}$	24.091%
ASSD	4.426 ± 3.546	$\mathbf{3.722 \pm 3.389}$	15.895%

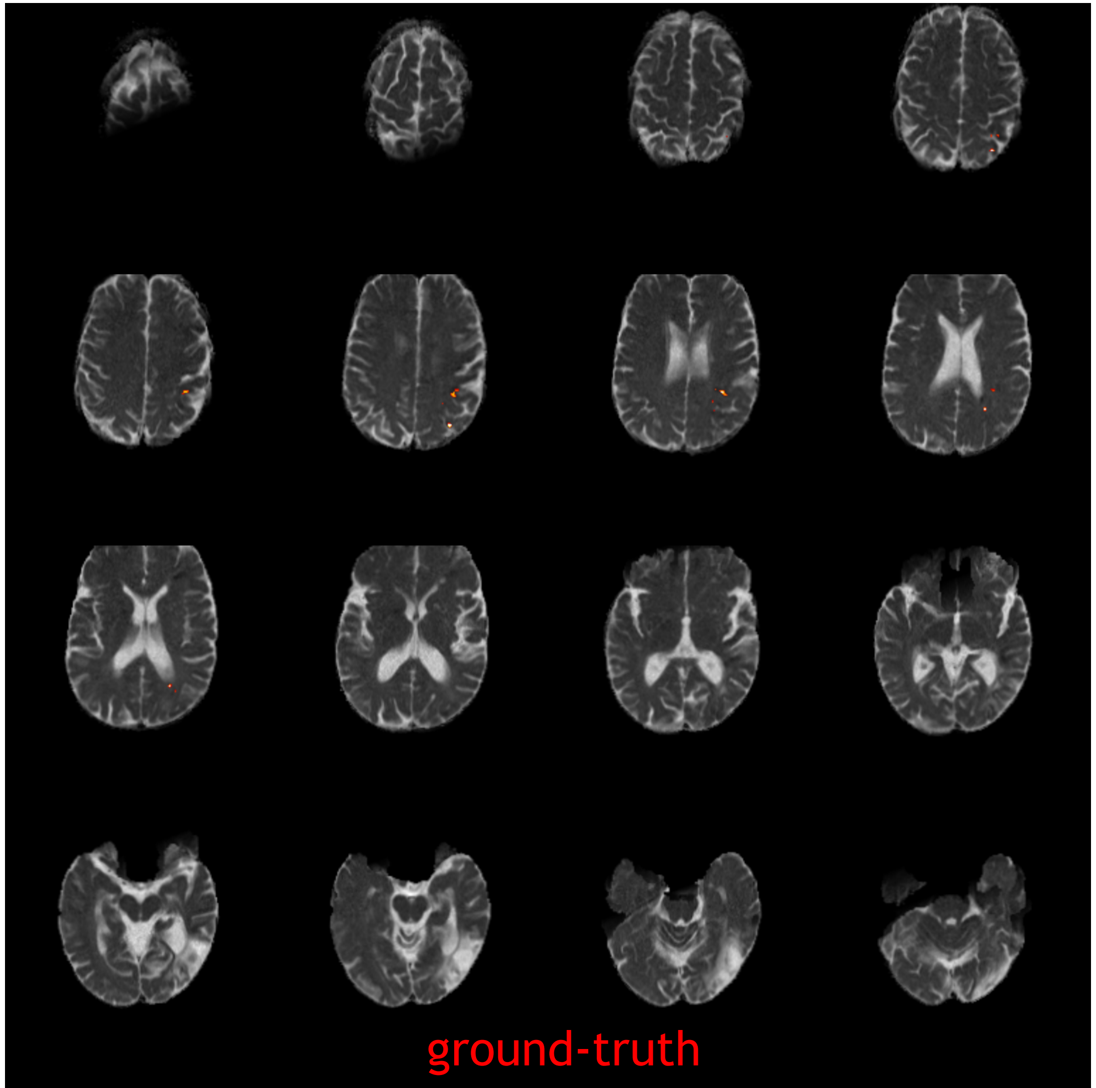
- Post Processing: remove any unconnected regions that had a volume smaller than 50% of the largest volume.
- This is because stroke lesions have one core and surrounding penumbra, usually there are not multiple unconnected affected regions;

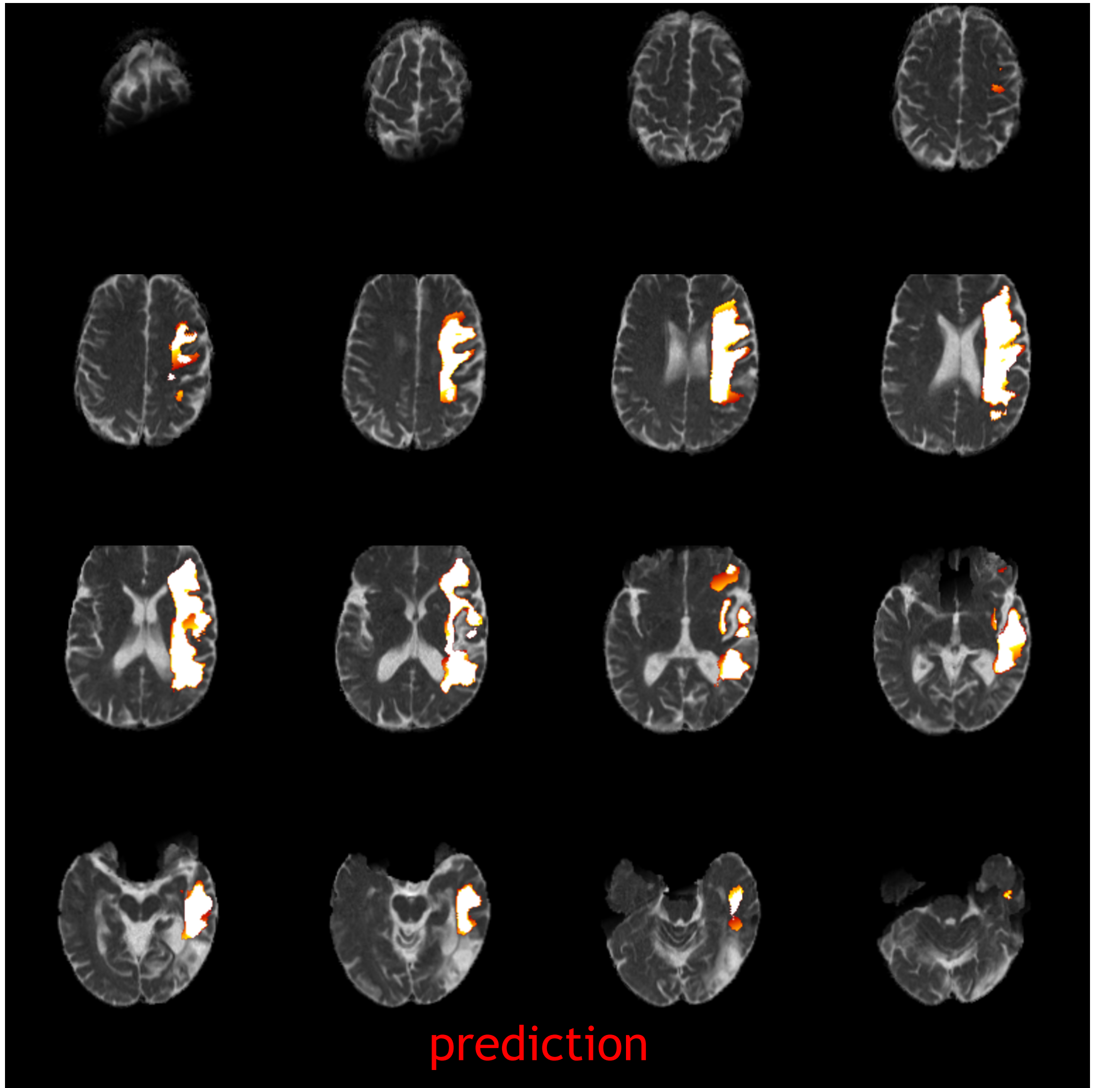
Median Case (DC = 43%)





Worst Case (DC = 0%)





Best Case (DC = 73%)