Image recognition

General recipe

Logistic Regression!

- Fix hypothesis class

$$h(x; \mathbf{w}, b) = \sigma(\mathbf{w}^T \phi(x) + b)$$

- Define loss function

$$L(h(x; \mathbf{w}, b), y) = -y \log h(x; \mathbf{w}, b) + (1 - y) \log(1 - h(x; \mathbf{w}, b))$$

- Minimize average loss on the training set using SGD

$$\min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^{n} L(h(x_i; \mathbf{w}, b), y_i)$$

Optimization using SGD

- Need to minimize average training loss
- Initialize parameters
- Repeat
  - Sample *minibatch* of k training examples
  - Compute average gradient of loss on minibatch
  - Take step along negative ofaverage gradient

$$\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} f(x_i, y_i, \boldsymbol{\theta})$$

$$\boldsymbol{\theta}^{(0)} \leftarrow \text{random}$$

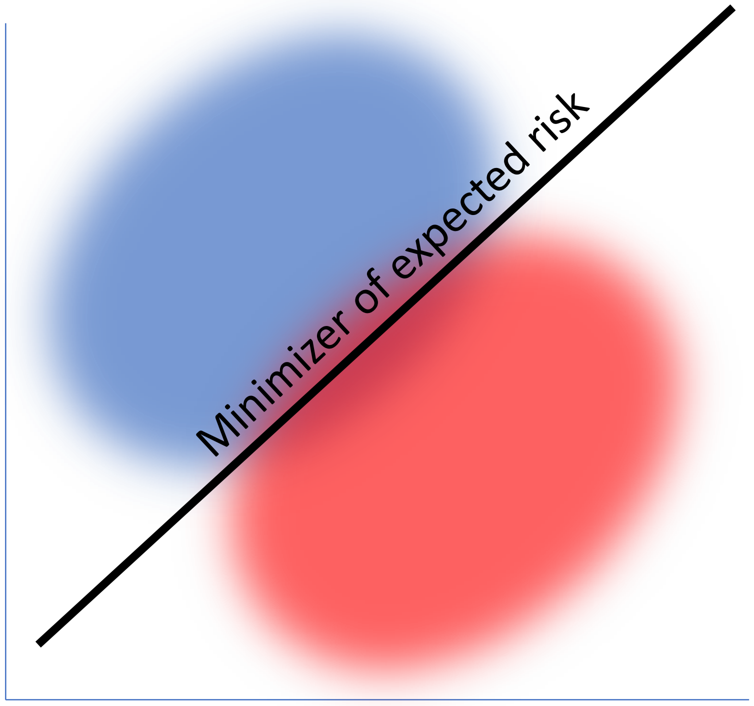
$$\text{for } t = 1, \ldots, T$$

$$i_1, \ldots, i_k \sim \text{Uniform}(n)$$

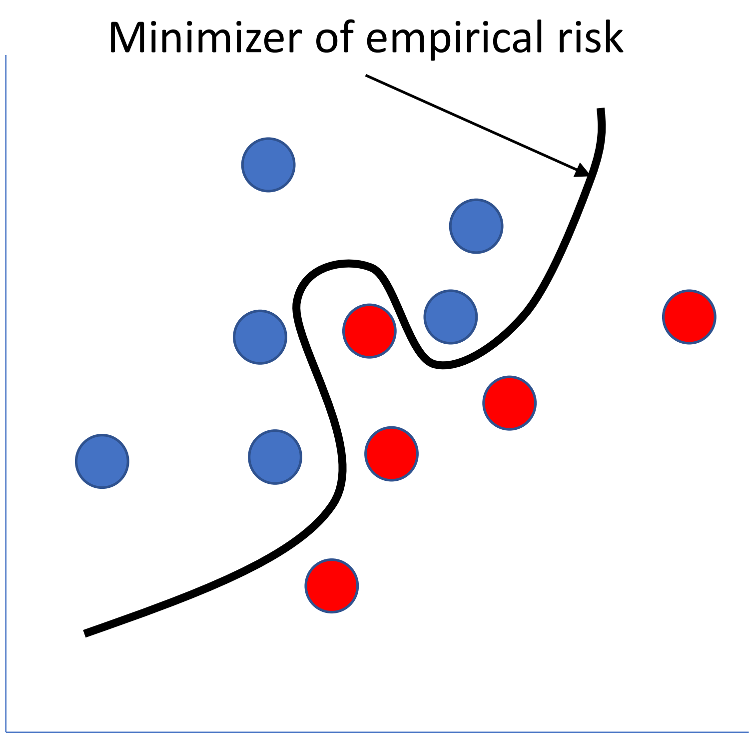
$$\mathbf{g}^{(t)} \leftarrow \frac{1}{k} \sum_{j=1}^{k} \nabla f(x_{i_j}, y_{i_j}, \boldsymbol{\theta}^{(t-1)})$$

$$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \lambda \mathbf{g}^{(t)}$$

Overfitting = fitting the noise



True distribution



Sampled training set

Generalization

$$R(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}} L(h(x), y) \qquad \hat{R}(S,h) = \frac{1}{|S|} \sum_{(x,y)\in S} L(h(x), y)$$

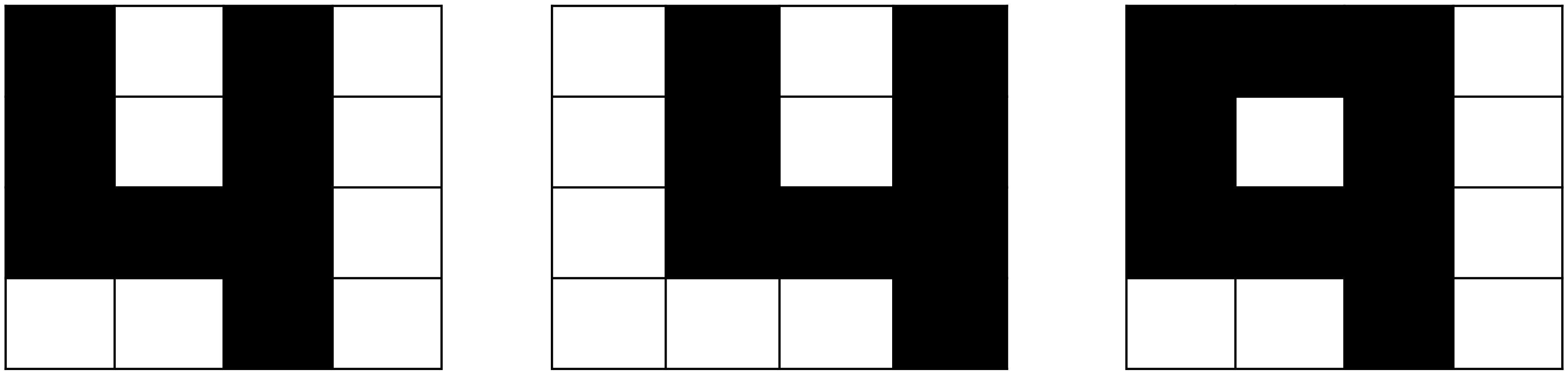
$$R(h) = \underbrace{\hat{R}(S,h)}_{\text{Training error}} + \underbrace{(R(h) - \hat{R}(S,h))}_{\text{Generalization error}}$$

Controlling generalization error

- Variance of empirical risk inversely proportional to size of S (central limit theorem)
  - Choose very large S!
- *Larger* the hypothesis class H, *Higher* the chance of hitting bad hypotheses with low training error and high generalization error
  - Choose small H!
- For many models, can *bound* generalization error using some property of parameters
  - "Regularization"

Back to images

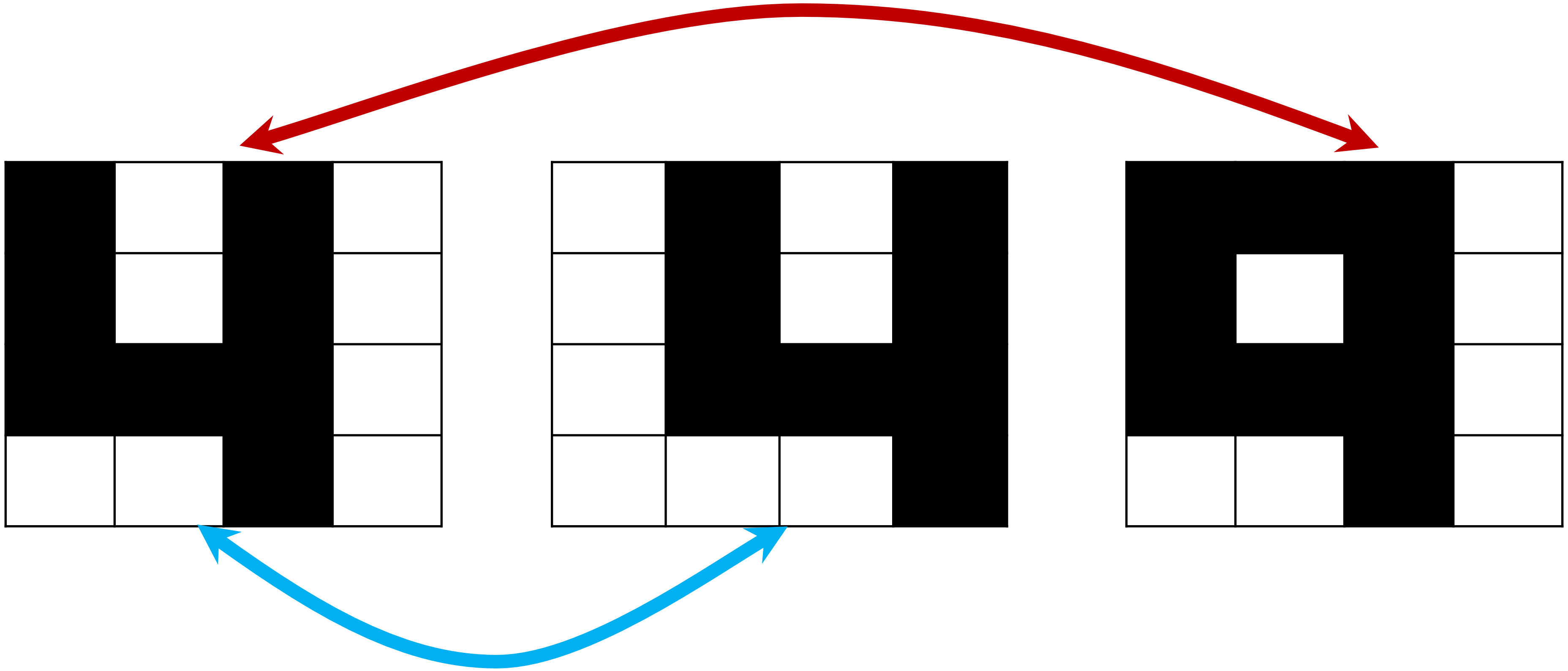
Linear classifiers on pixels are bad



- Solution 1: Better feature vectors
- Solution 2: Non-linear classifiers

Better feature vectors

These must have different feature vectors: *discriminability*

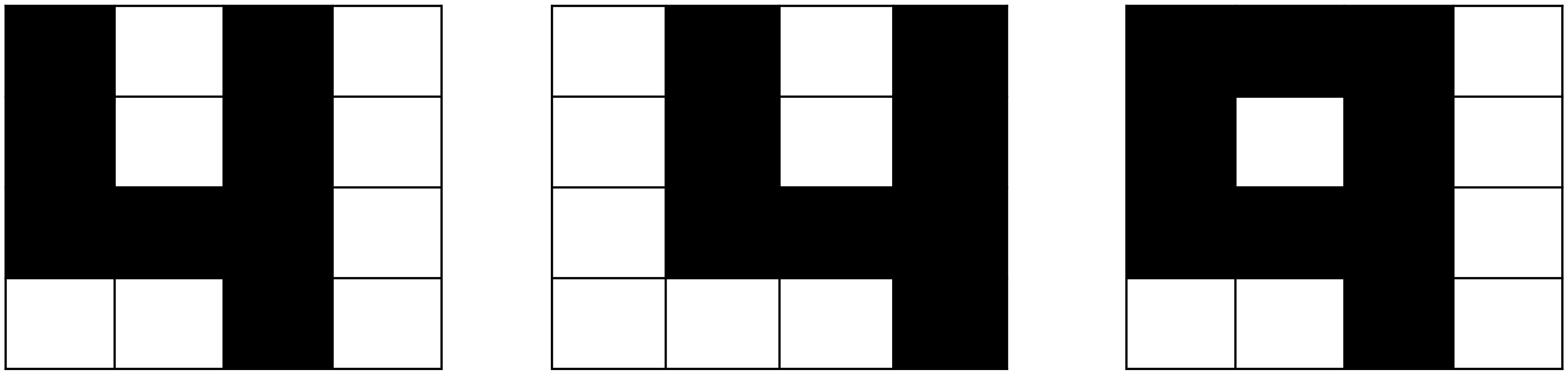


These must have similar feature vectors: *invariance*

SIFT

- Match *pattern of edges*
  - Edge orientation – clue to shape
- Be resilient to *small deformations*
  - Deformations might move pixels around, but slightly
  - Deformations might change edge orientations, but slightly
- *Not* resilient to large deformations: important for recognition
- Other feature representations exist

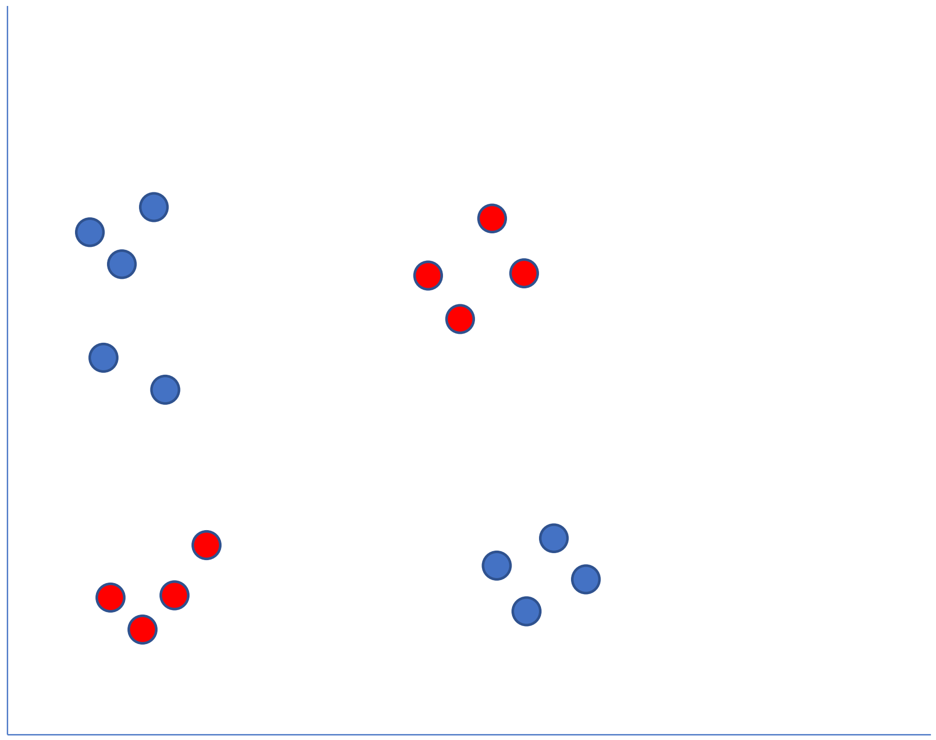
Linear classifiers on pixels are bad



- Solution 1: Better feature vectors
- Solution 2: Non-linear classifiers

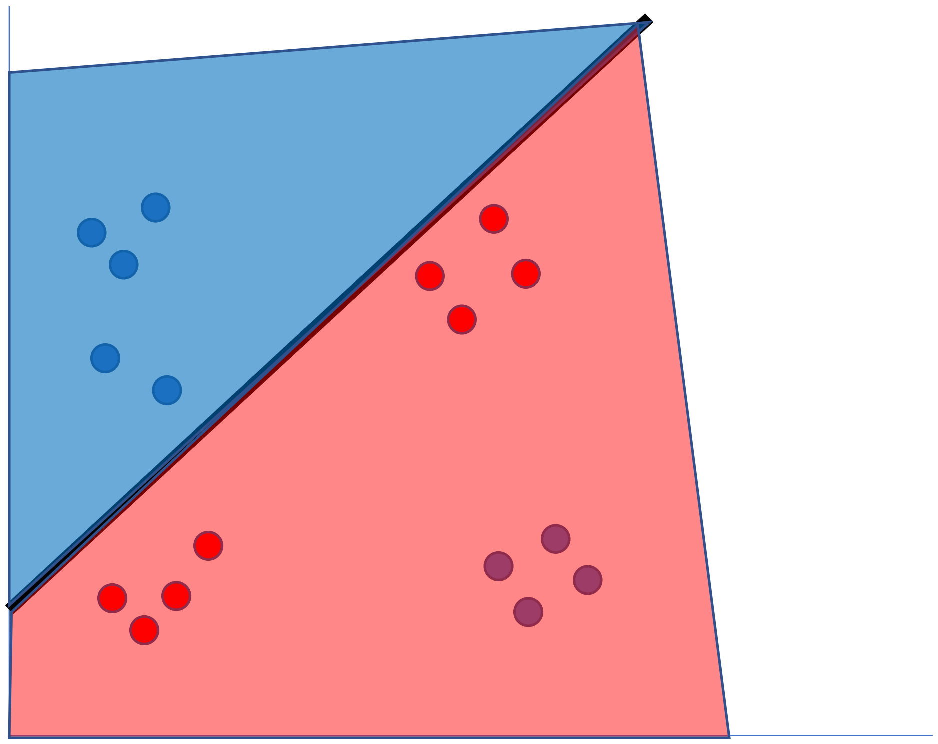
Non-linear classifiers

- Suppose we have a feature vector for every image



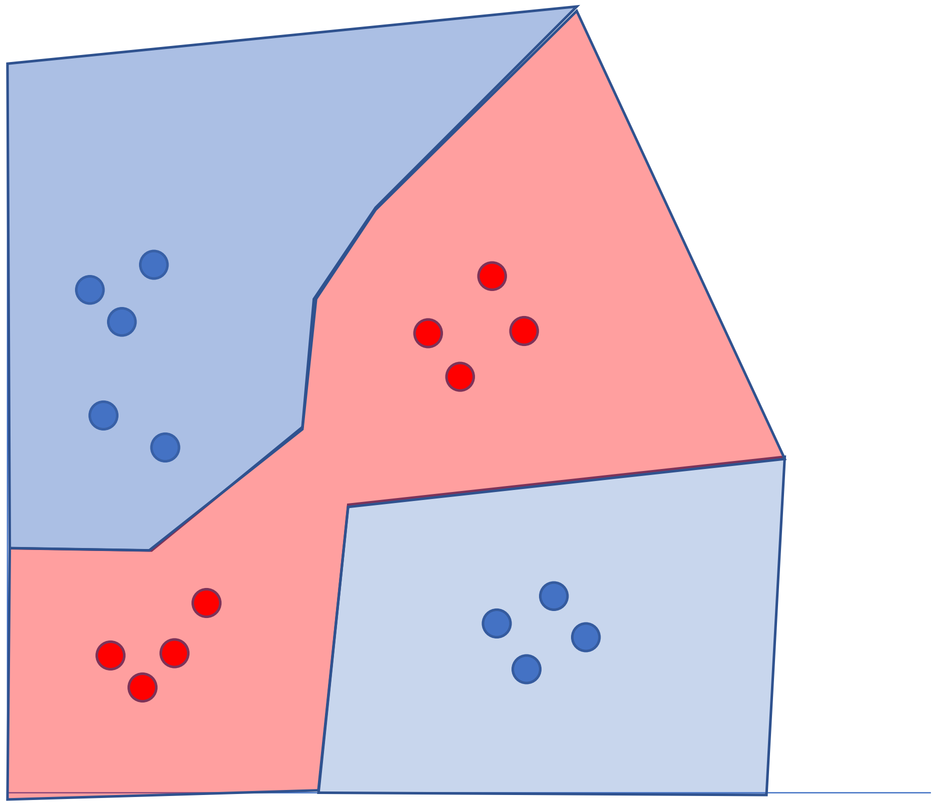
Non-linear classifiers

- Suppose we have a feature vector for every image
  - Linear classifier



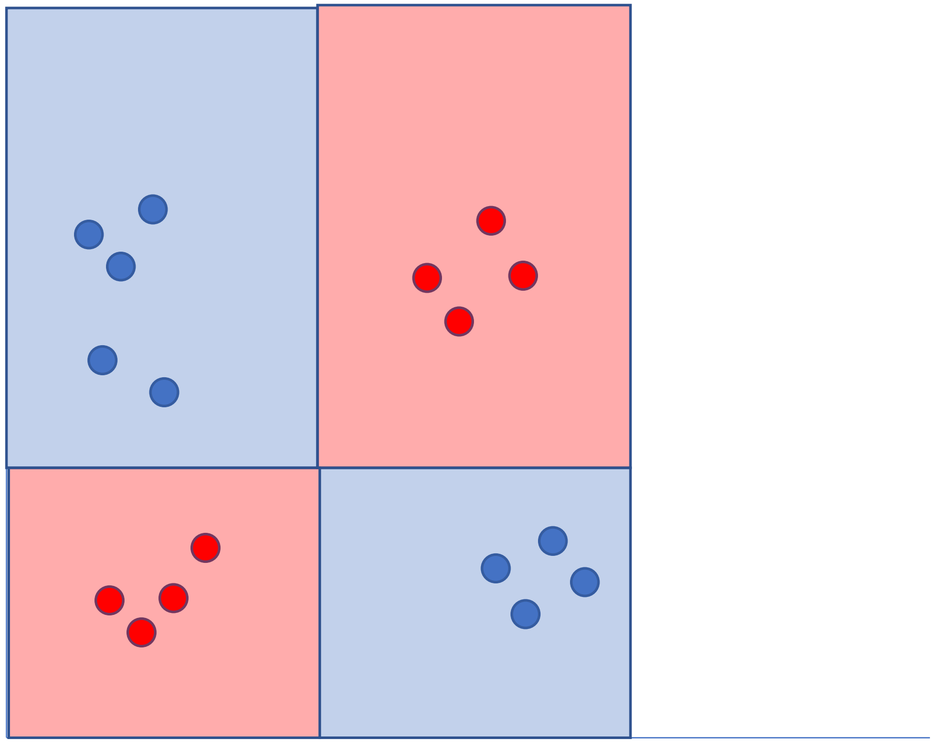
Non-linear classifiers

- Suppose we have a feature vector for every image
  - Linear classifier
  - Nearest neighbor: assign each point the label of the nearest neighbor



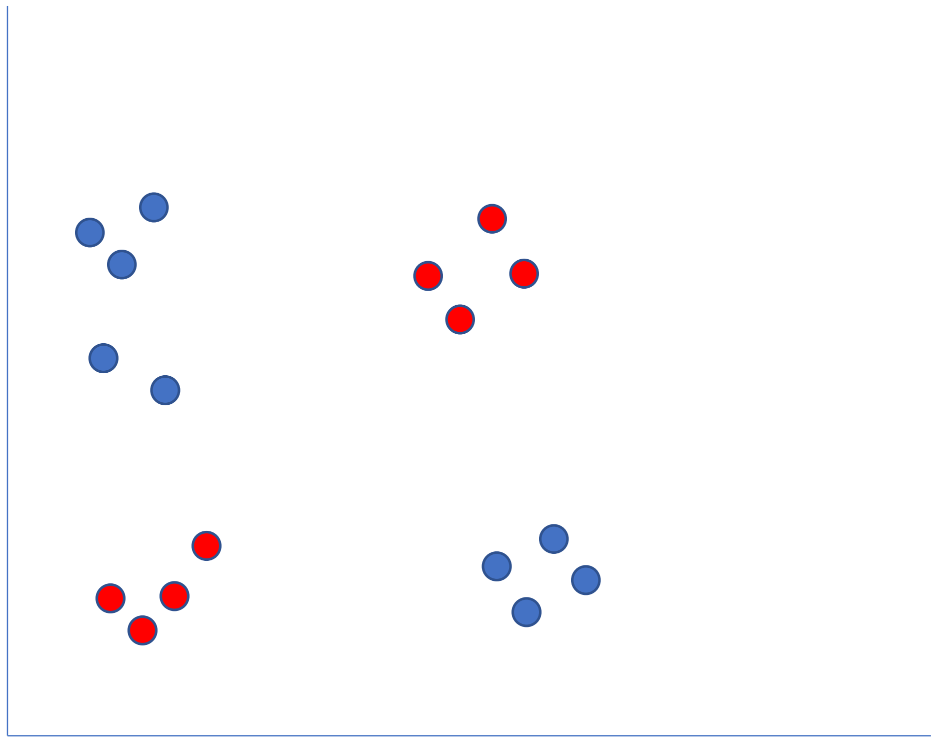
Non-linear classifiers

- Suppose we have a feature vector for every image
  - Linear classifier
  - Nearest neighbor: assign each point the label of the nearest neighbor
  - Decision tree: series of if-then-else statements on different features



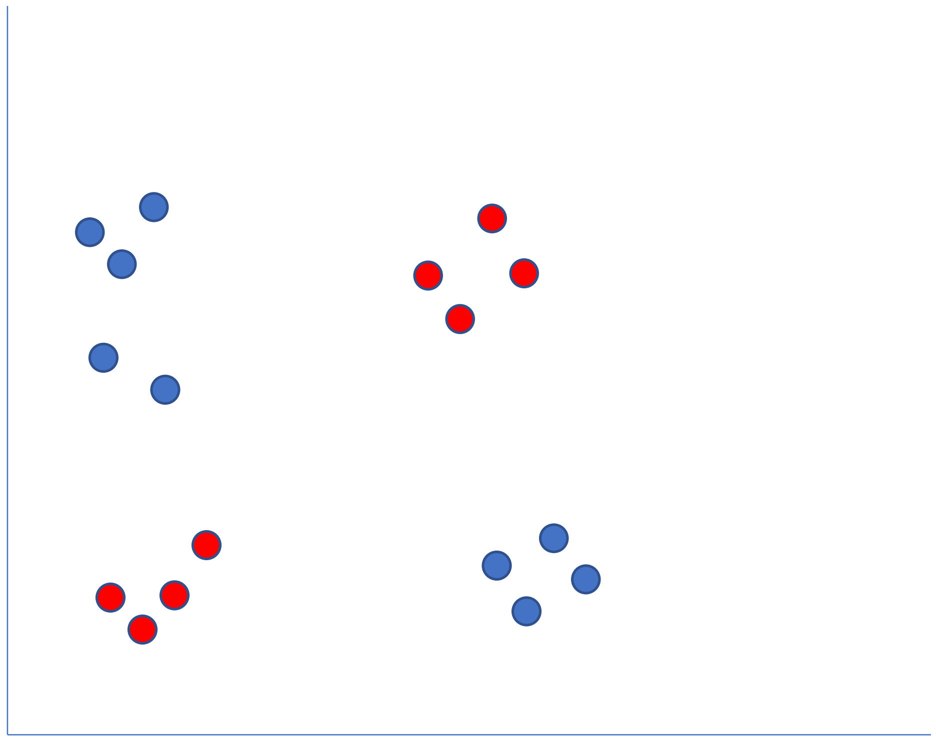
Non-linear classifiers

- Suppose we have a feature vector for every image
  - Linear classifier
  - Nearest neighbor: assign each point the label of the nearest neighbor
  - Decision tree: series of if-then-else statements on different features
  - Neural networks



Non-linear classifiers

- Suppose we have a feature vector for every image
  - Linear classifier
  - Nearest neighbor: assign each point the label of the nearest neighbor
  - Decision tree: series of if-then-else statements on different features
  - Neural networks / multi-layer perceptrons



Multilayer perceptrons

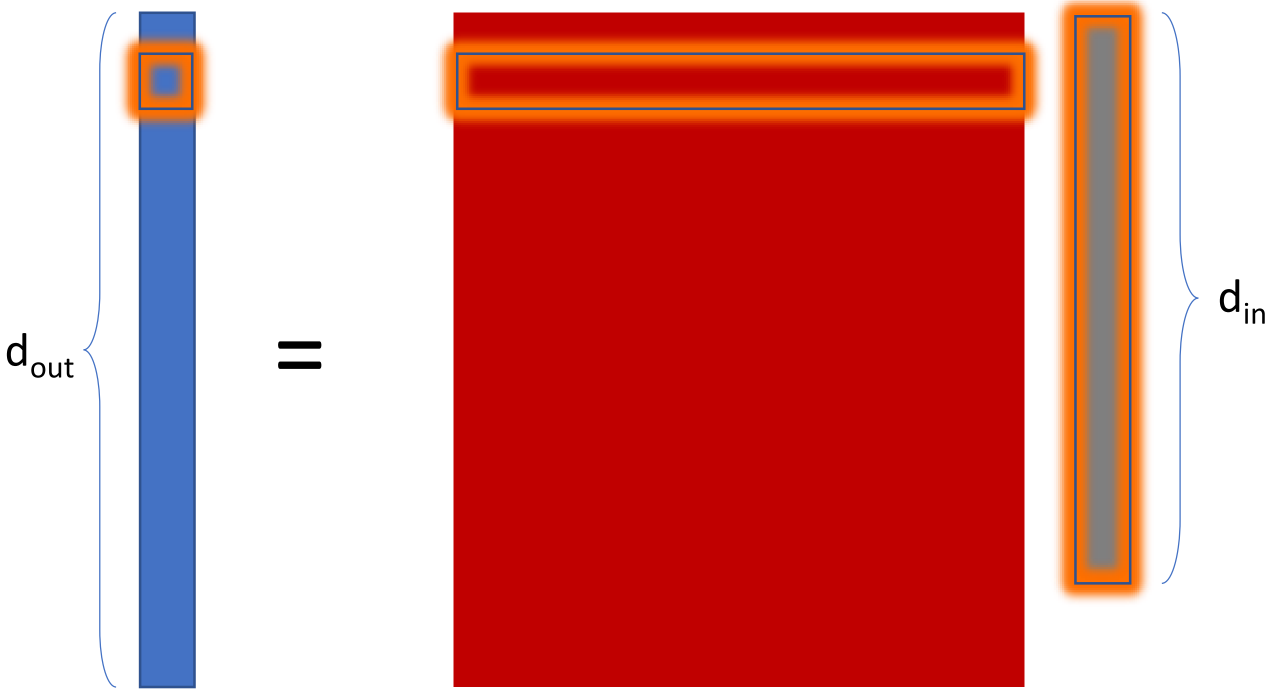
- Key idea: build complex functions by composing simple functions
- Caveat: simple functions must include non-linearities
- W(U(Vx)) = (WUV)x
- Let us start with only two ingredients:
  - *Linear: y = Wx + b*
  - *Rectified linear unit (ReLU, also called half-wave rectification): y = max(x,0)*

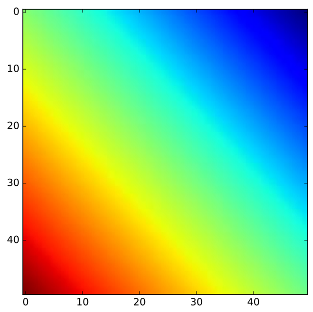
The linear function

- $y = Wx + b$
- Parameters: W, b
- Input: x (column vector, or 1 data point per column)
- Output: y (column vector or 1 data point per column)
- Hyperparameters:
  - Input dimension = # of rows in x
  - Output dimension = # of rows in y
  - W : outdim x indim
  - b : outdim x 1

The linear function

- y = Wx + b
- Every row of y corresponds to a hyperplane in x space

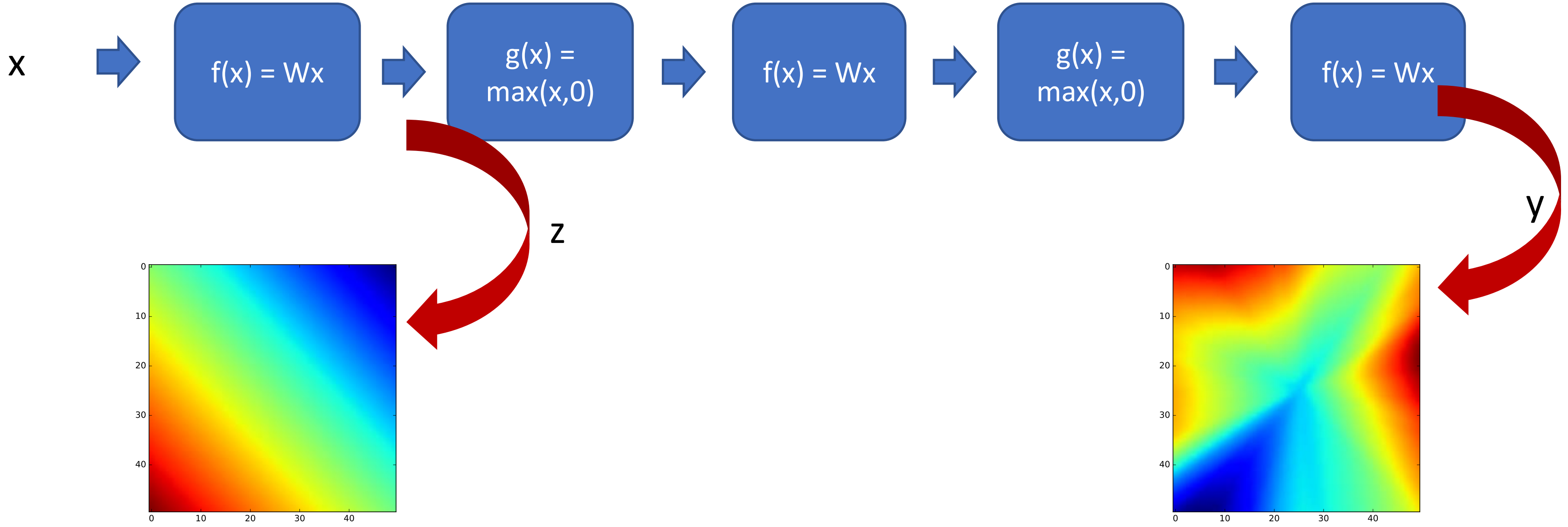




The case when d_{in} = 2. A single row in y plotted for every possible value of x

Multilayer perceptrons

- Key idea: build complex functions by composing simple functions

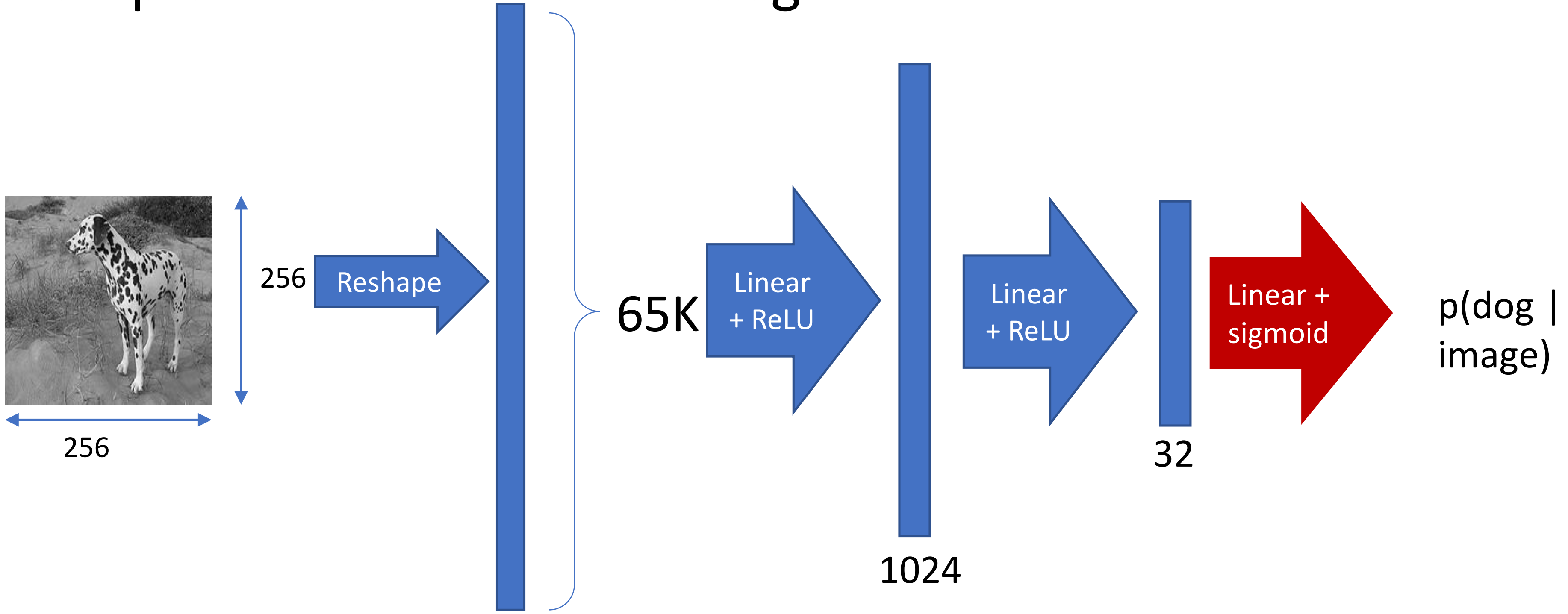


1 row of z
plotted for
every value of x

1 row of y
plotted for
every value of x

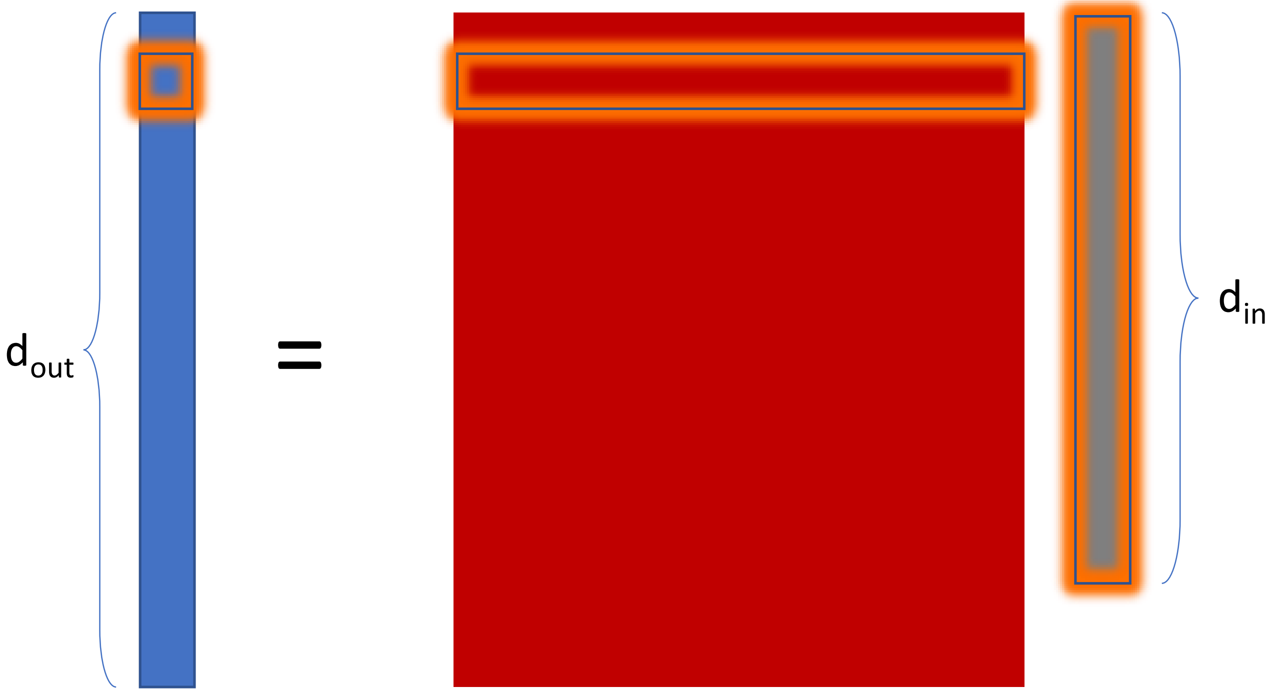
Multilayer perceptron on images

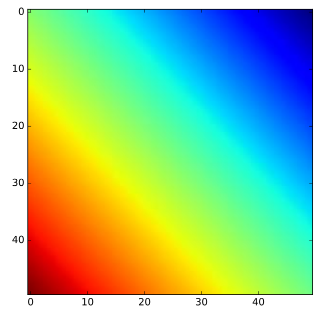
- An example network for cat vs dog



The linear function

- $y = Wx + b$
- How many parameters does a linear function have?



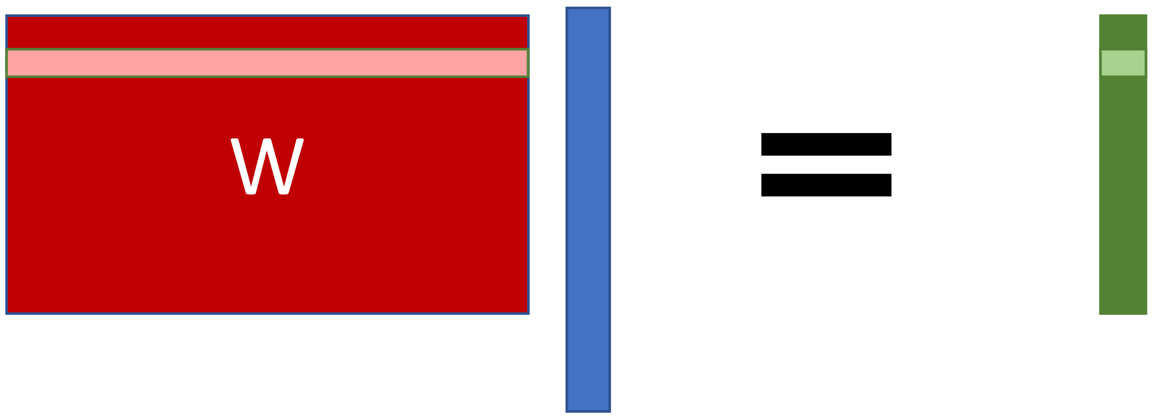


The case when $d_{in} = 2$. A single row in y plotted for every possible value of x

The linear function for images

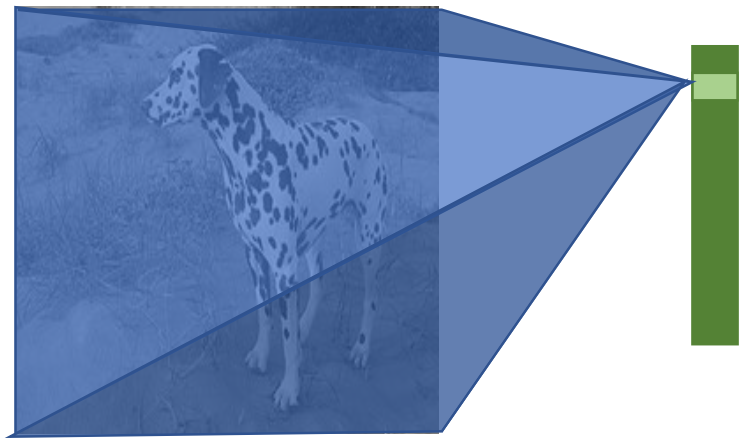
Reducing parameter count

- A single "pixel" in the output is a weighted combination of *all* input pixels



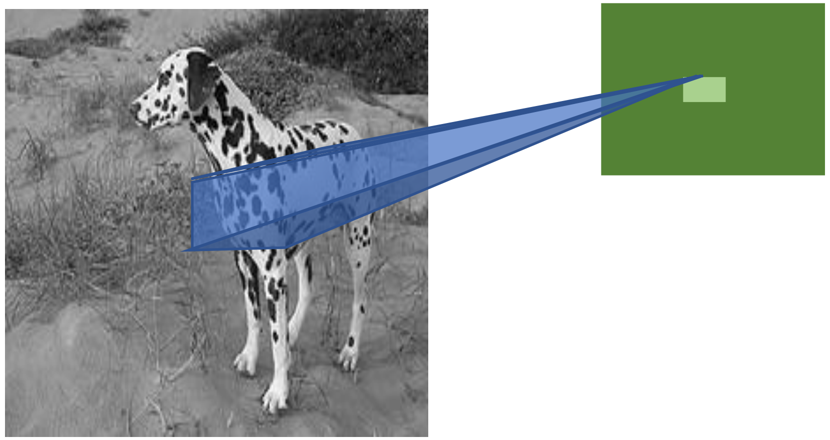
Reducing parameter count

- A single “pixel” in the output is a weighted combination of *all* input pixels



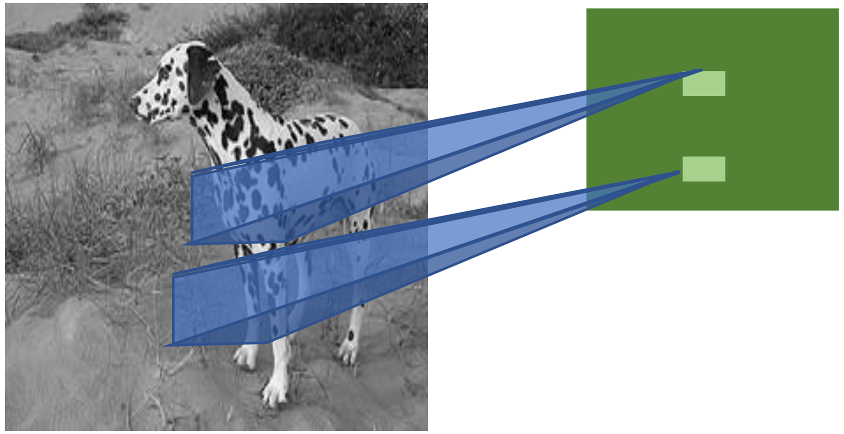
Idea 1: local connectivity

- Instead of inputs and outputs being general vectors suppose we keep both as 2D arrays.
- Reasonable assumption: output pixels only produced by nearby input pixels



Idea 2: Translation invariance

- Output pixels weighted combination of nearby pixels
- Weights should not depend on the location of the neighborhood



Linear function + translation invariance = *convolution*

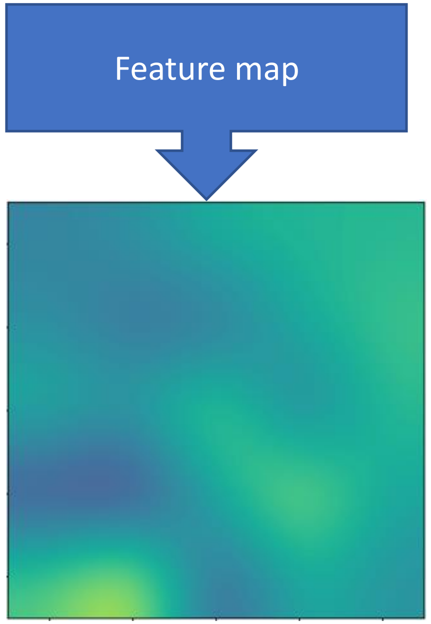
- Local connectivity determines kernel size

5.4	0.1	3.6
1.8	2.3	4.5
1.1	3.4	7.2

Linear function + translation invariance = *convolution*

- Local connectivity determines kernel size
- Running a filter on a single image gives a single *feature map*

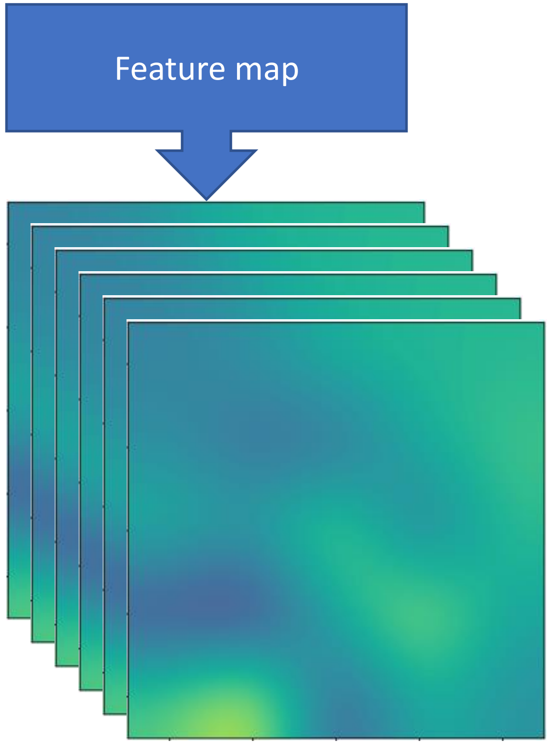
5.4	0.1	3.6
1.8	2.3	4.5
1.1	3.4	7.2



Convolution with multiple filters

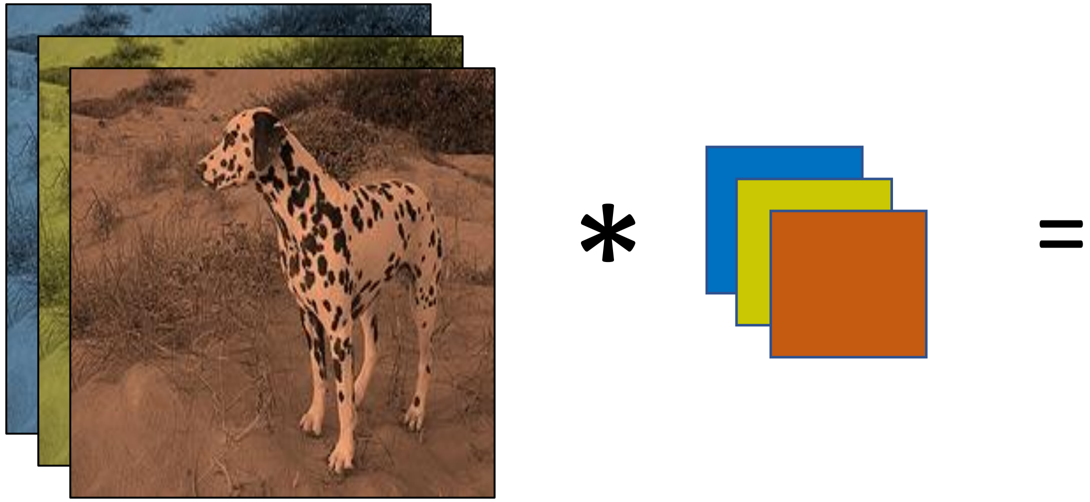
- Running multiple filters gives *multiple feature maps*
- Each feature map is a *channel* of the output

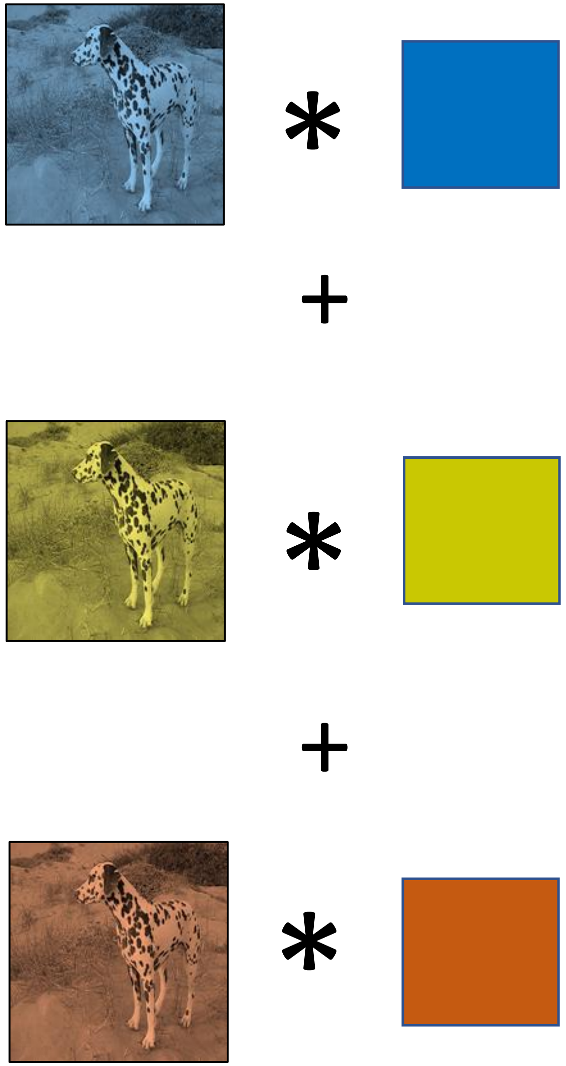
5.4	0.1	3.6
1.8	2.3	4.5
1.1	3.4	7.2



Convolution over multiple channels

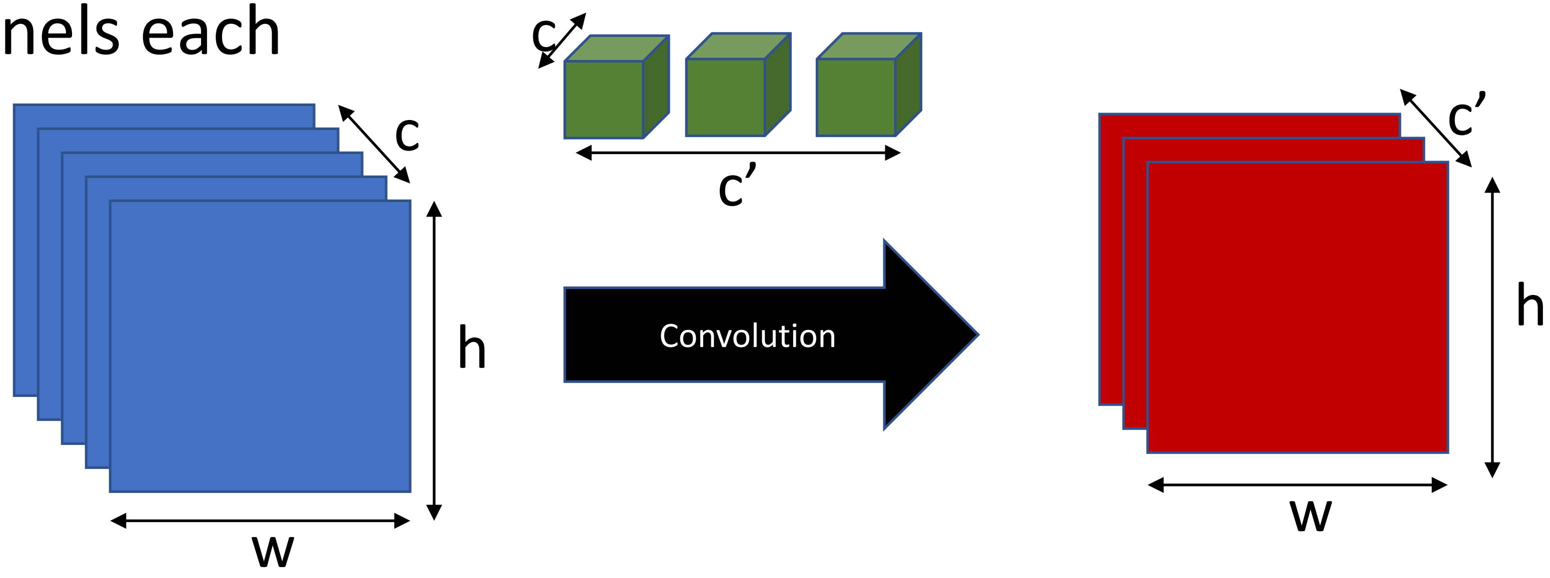
- If the input also has multiple channels, each filter also has multiple channels, and output of a filter = sum of responses across channels





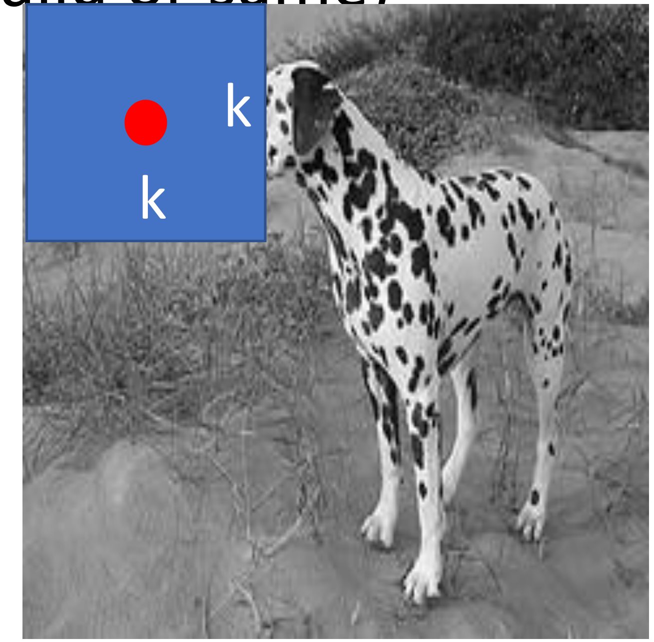
Convolution as a primitive

- To get c’ output channels out of c input channels, we need c’ filters of c channels each



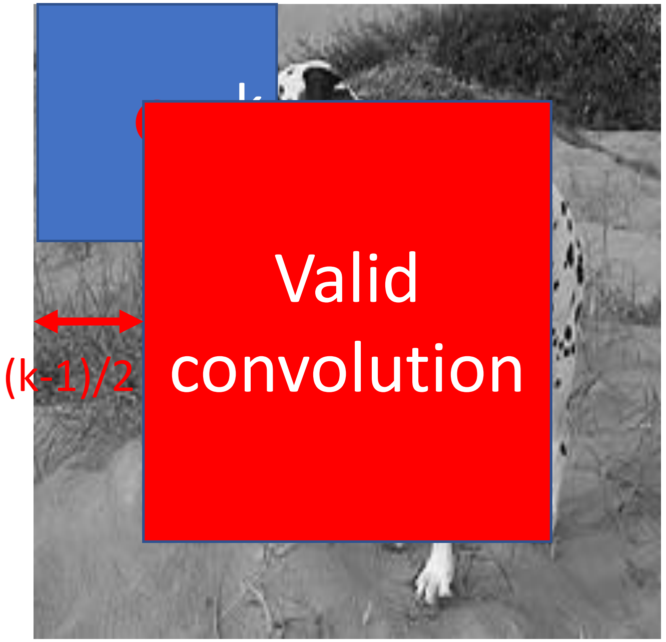
Kernel sizes and padding

- As with standard convolution, we can have ”valid”, “same” or “full” convolution (typically valid or same)



Kernel sizes and padding

- Valid convolution decreases size by (k-1)/2 on each side
- Pad by (k-1)/2!

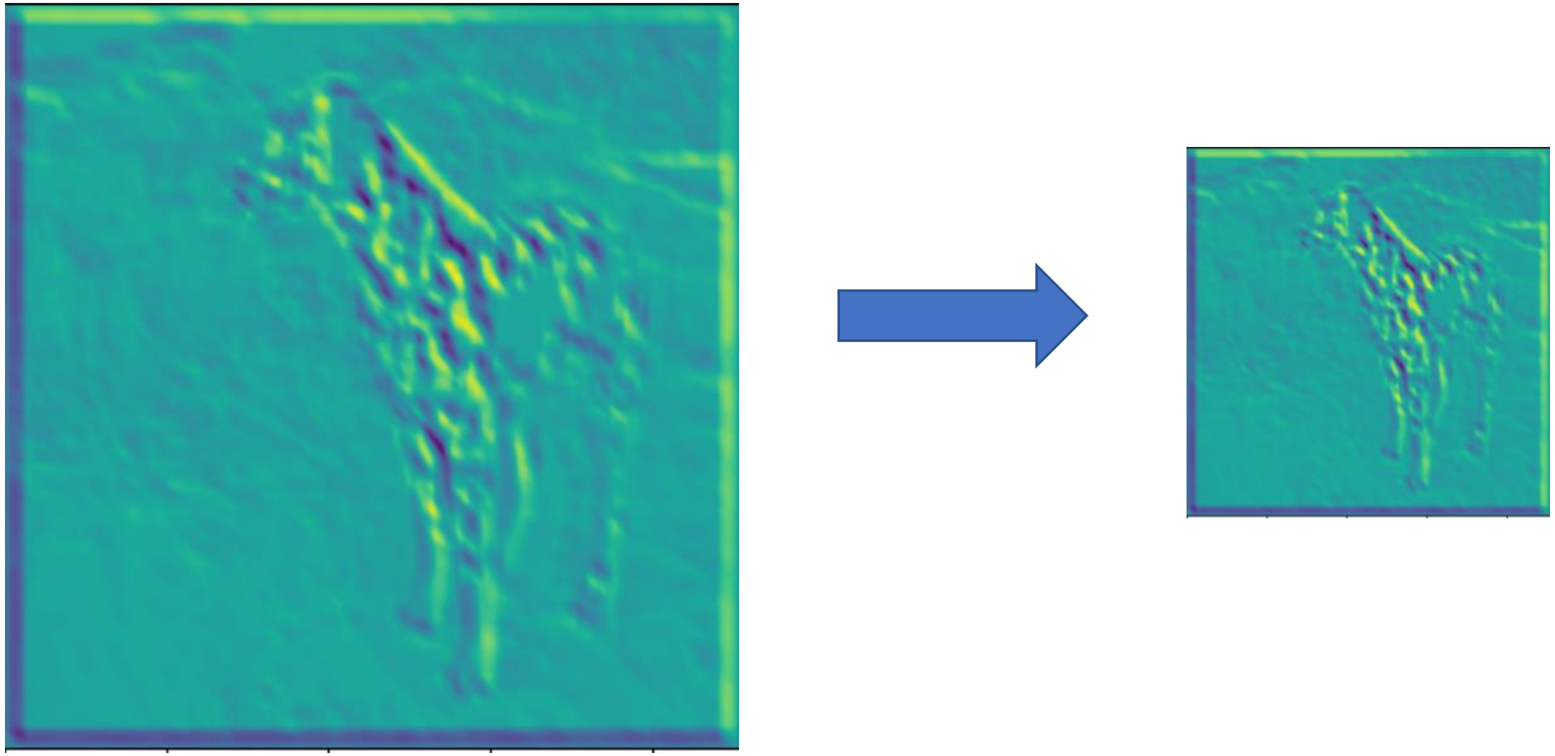


The convolution unit

- Each convolutional unit takes *a collection of feature maps* as input, and produces *a collection of feature maps* as output
- Parameters: Filters (+bias)
- If c_{in} input feature maps and c_{out} output feature maps
  - Each filter is k x k x c_{in}
  - There are c_{out} such filters
- Other hyperparameters: padding

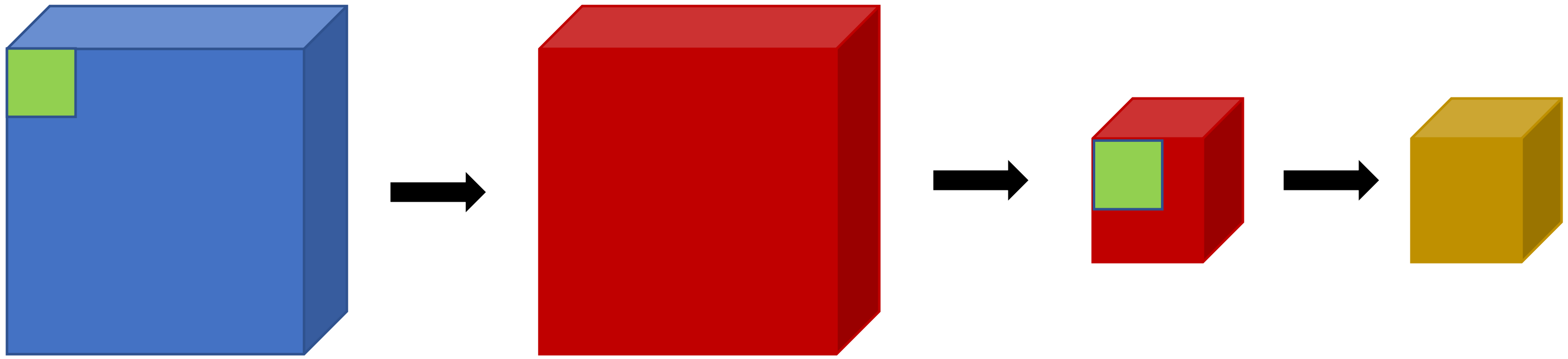
Invariance to distortions: Subsampling

- Convolution by itself doesn’t grant invariance
- But by subsampling, large distortions become smaller, so more invariance



Convolution-subsampling-convolution

- Interleaving convolutions and subsamplings causes later convolutions to capture a *larger fraction of the image* with the same kernel size
- Set of image pixels that an intermediate output pixel depends on = *receptive field*
- Convolutions after subsamplings increase the receptive feild



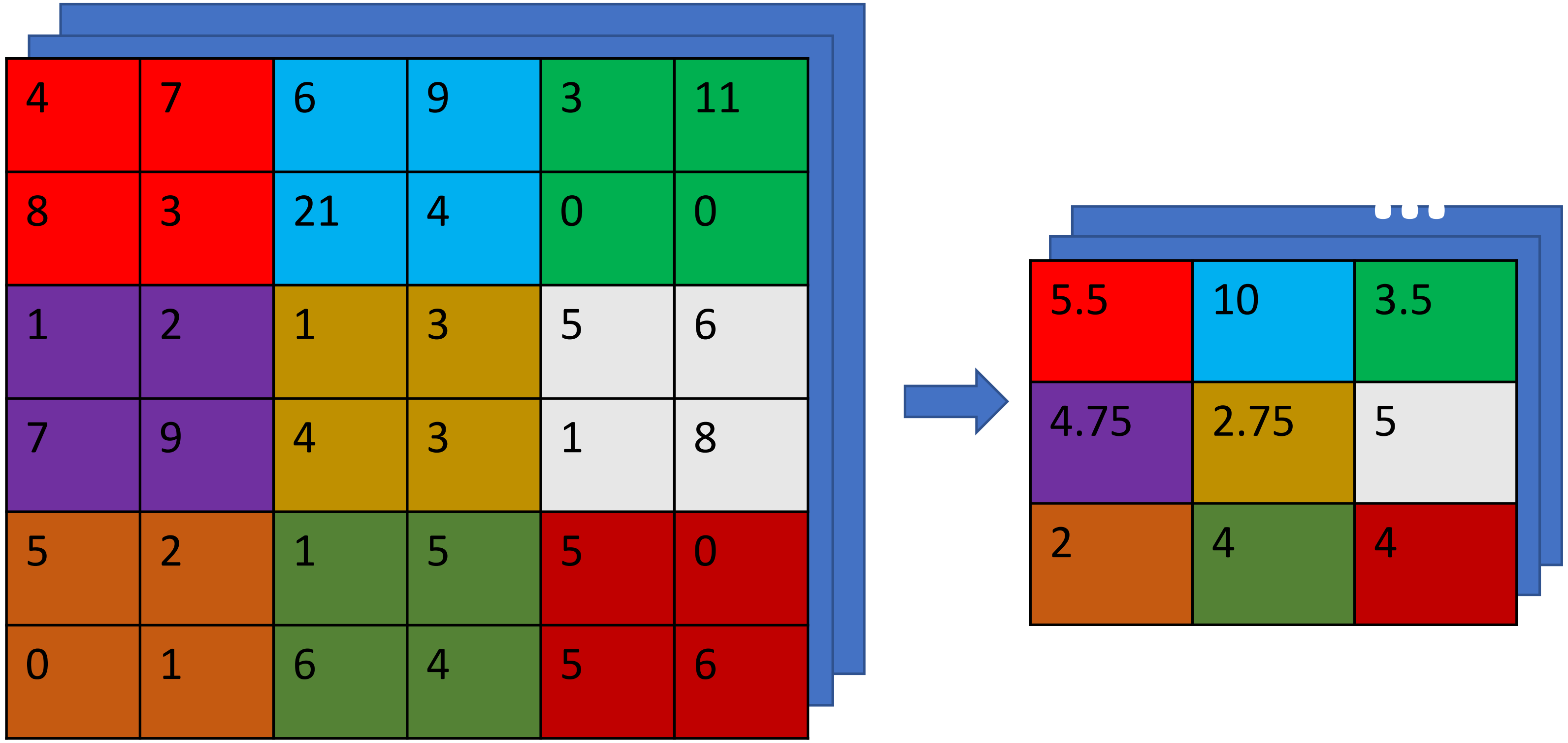
Convolution subsampling convolution

- Convolution in earlier steps detects *more local* patterns *less resilient* to distortion
- Convolution in later steps detects *more global* patterns *more resilient* to distortion
- Subsampling allows capture of *larger, more invariant* patterns

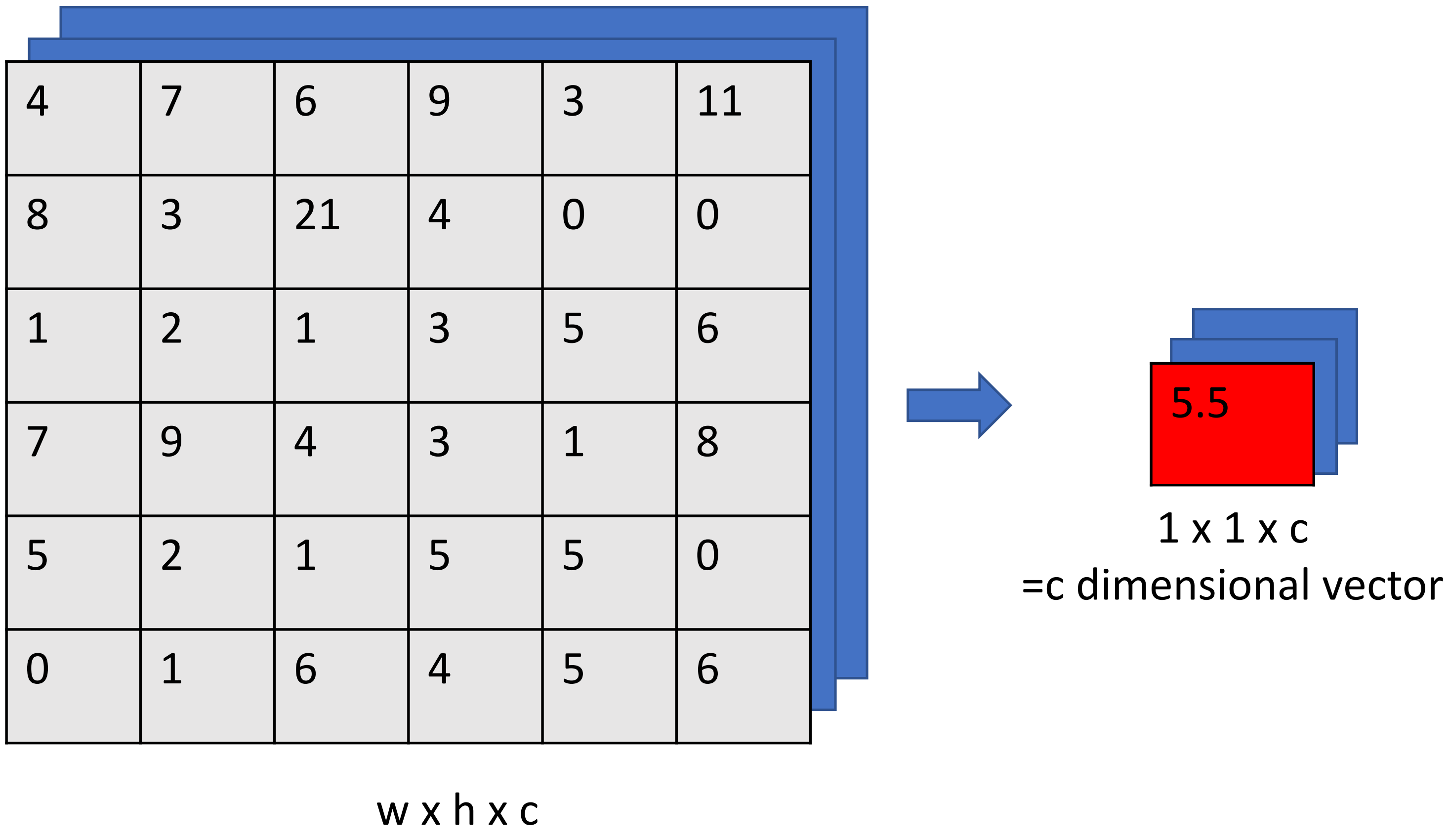
Strided convolution

- Convolution with stride s = standard convolution + subsampling by picking 1 value every s values
- Example: convolution with stride 2 = standard convolution + subsampling by a factor of 2

Invariance to distortions: Average Pooling



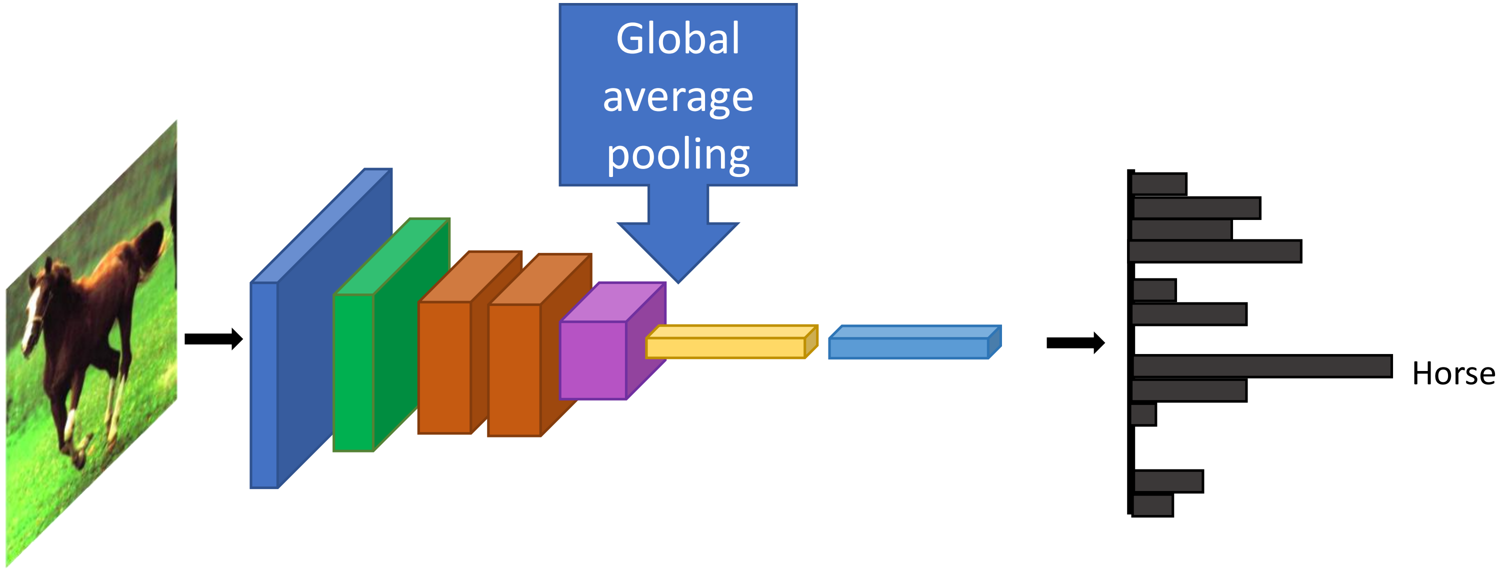
Global average pooling



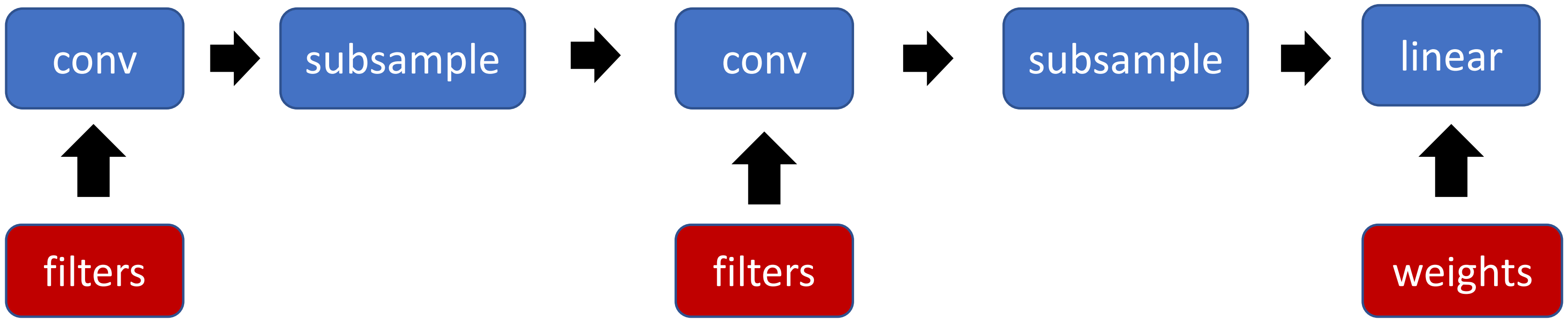
The pooling unit

- Each pooling unit takes *a collection of feature maps* as input and produces *a collection of feature maps* as output
- Output feature maps are usually smaller in height / width
- Parameters: None

Convolutional networks



Convolutional networks



Empirical Risk Minimization

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} L(h(x_i; \boldsymbol{\theta}), y_i)$$

Convolutional network

$$\underbrace{\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \lambda \frac{1}{N} \sum_{i=1}^{N} \nabla L(h(x_i; \boldsymbol{\theta}), y_i)}_{\text{Gradient descent update}}$$

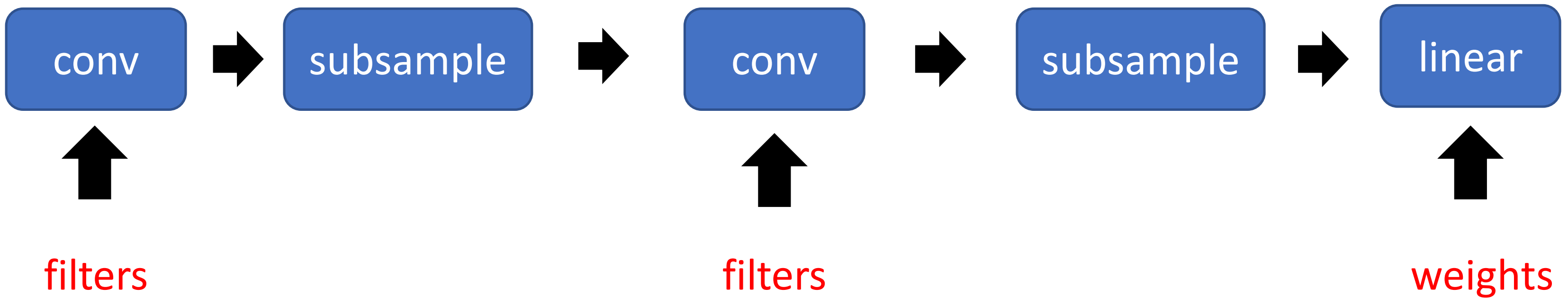
Computing the gradient of the loss

$$\nabla L(h(x;\boldsymbol{\theta}), y)$$

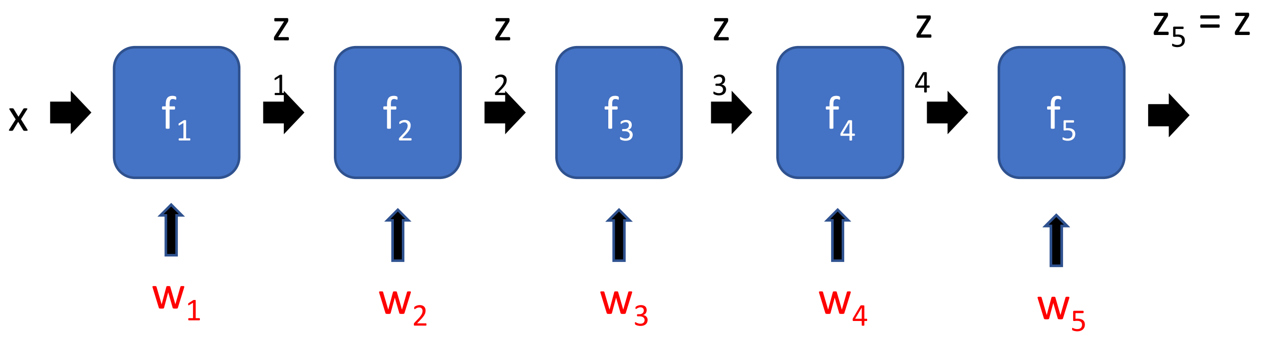
$$z = h(x;\boldsymbol{\theta})$$

$$\nabla_{\boldsymbol{\theta}} L(z, y) = \frac{\partial L(z, y)}{\partial z} \frac{\partial z}{\partial \boldsymbol{\theta}}$$

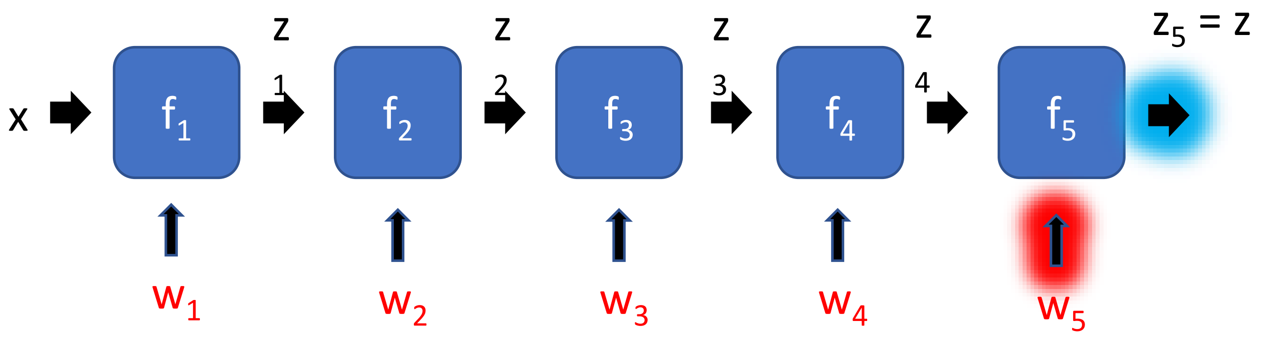
Convolutional networks



The gradient of convnets

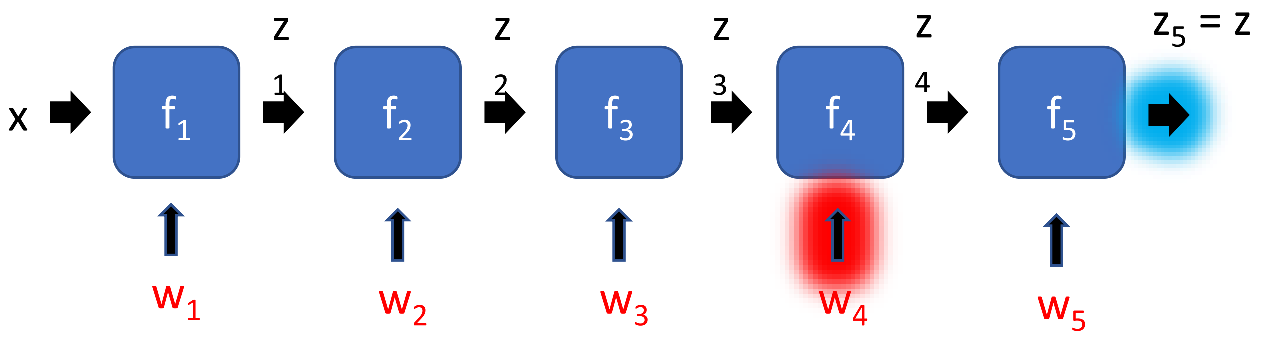


The gradient of convnets



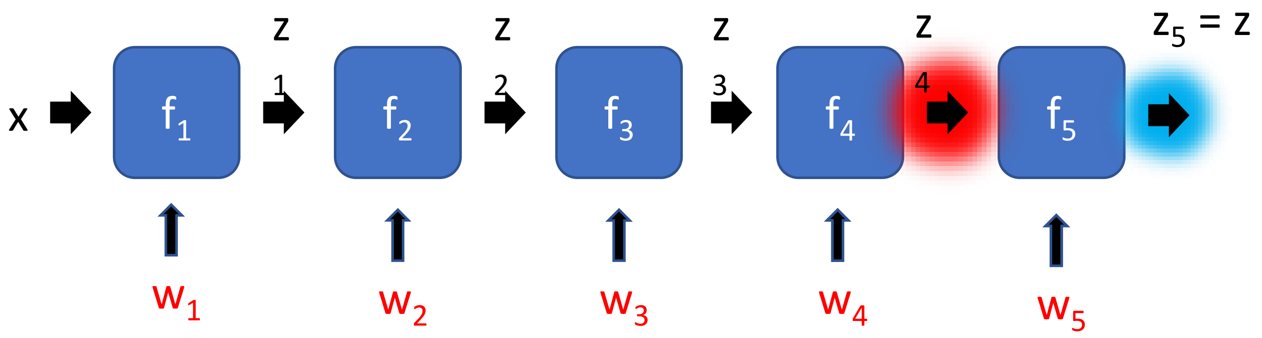
$$\frac{\partial z}{\partial w_5}$$

The gradient of convnets



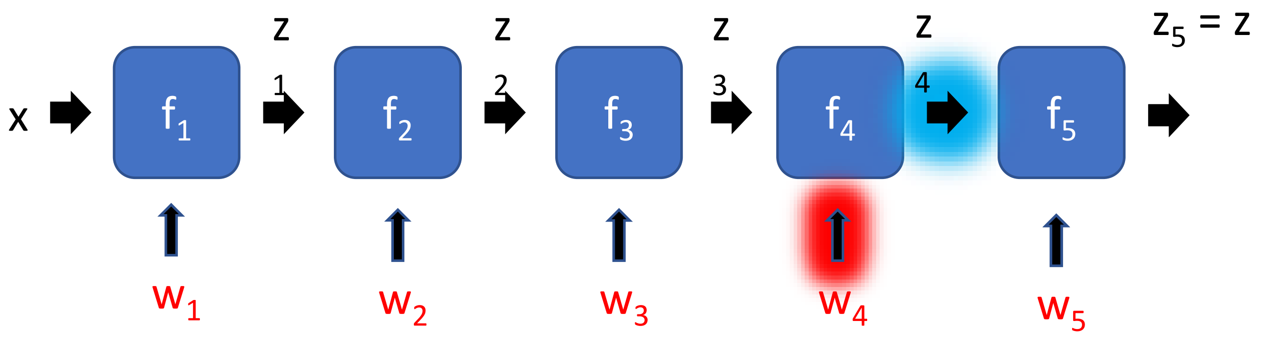
$$\frac{\partial z}{\partial w_4}$$

The gradient of convnets



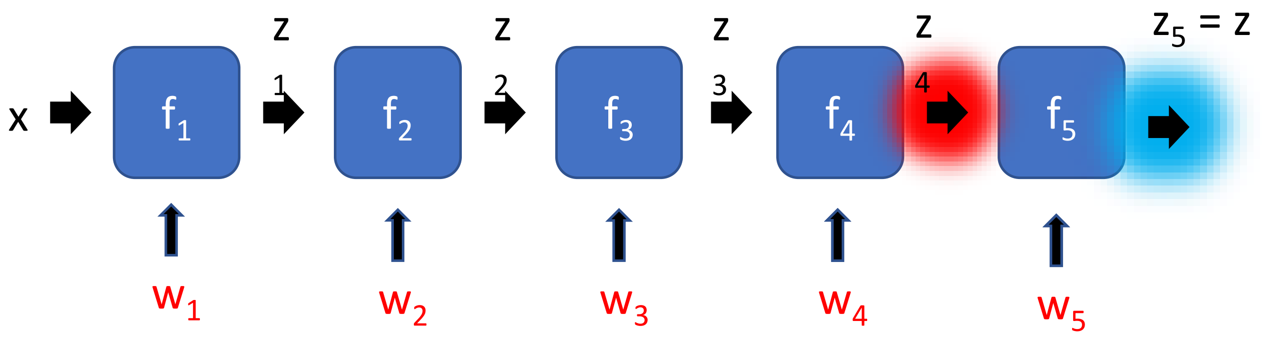
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4} \frac{\partial z_4}{\partial w_4}$$

The gradient of convnets



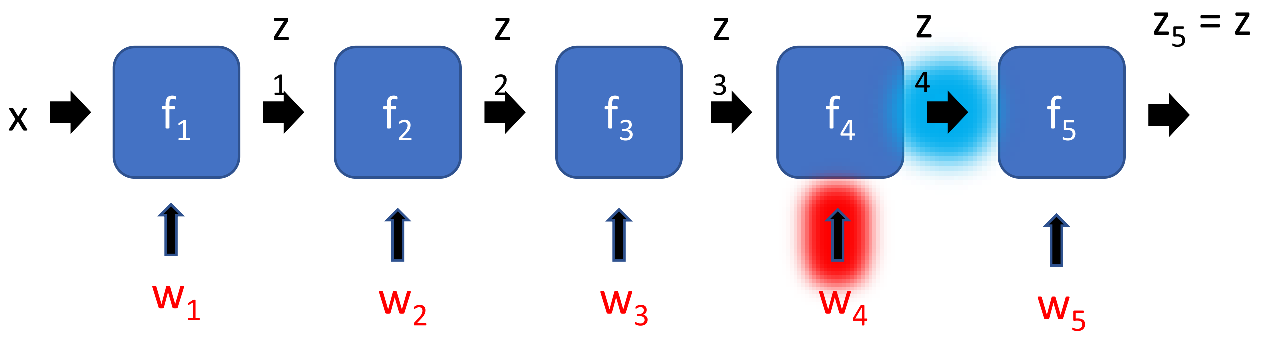
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4}$$

The gradient of convnets



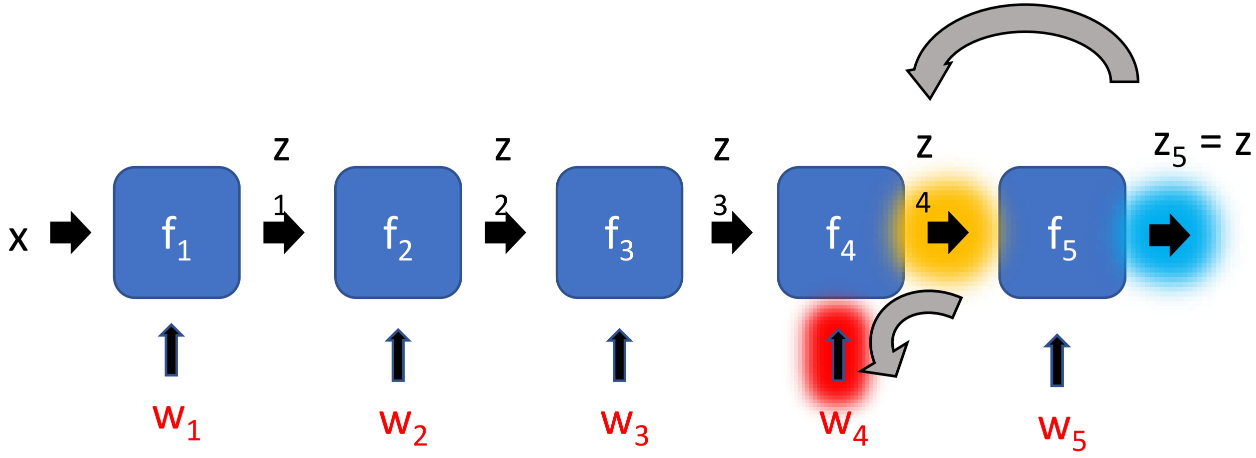
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4} = \frac{\partial f_5(z_4, w_5)}{\partial z_4}\frac{\partial f_4(z_3, w_4)}{\partial w_4}$$

The gradient of convnets



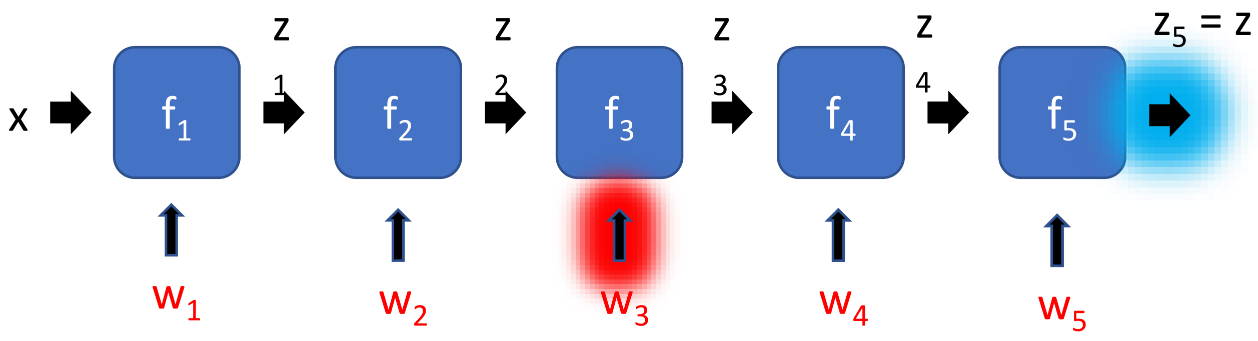
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4} = \frac{\partial f_5(z_4, w_5)}{\partial z_4}\frac{\partial f_4(z_3, w_4)}{\partial w_4}$$

The gradient of convnets



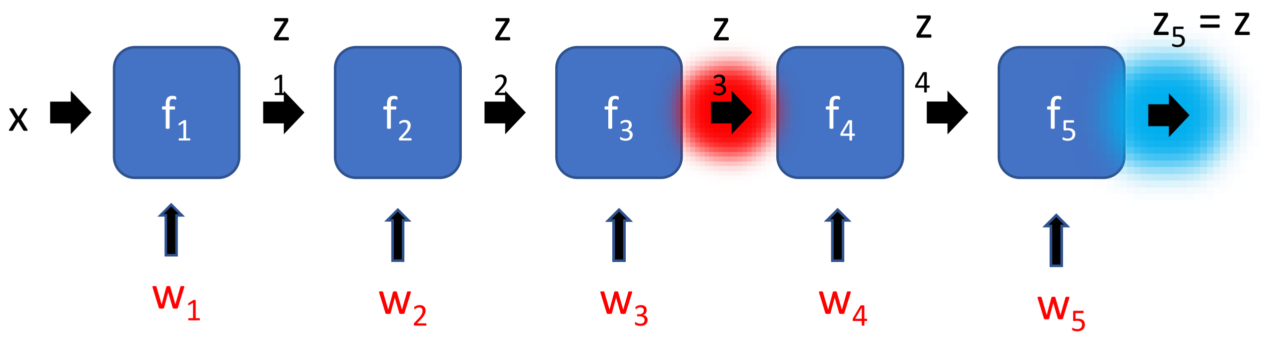
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4} = \frac{\partial f_5(z_4, w_5)}{\partial z_4}\frac{\partial f_4(z_3, w_4)}{\partial w_4}$$

The gradient of convnets



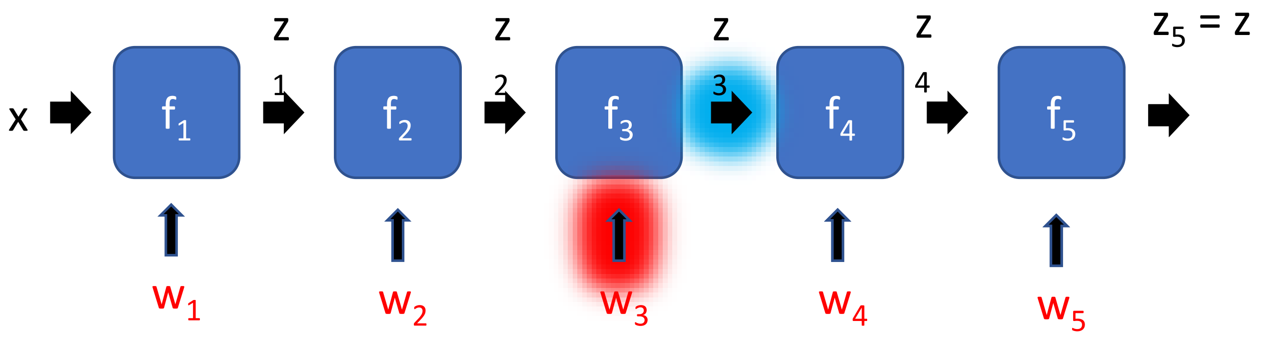
$$\frac{\partial z}{\partial w_3}$$

The gradient of convnets



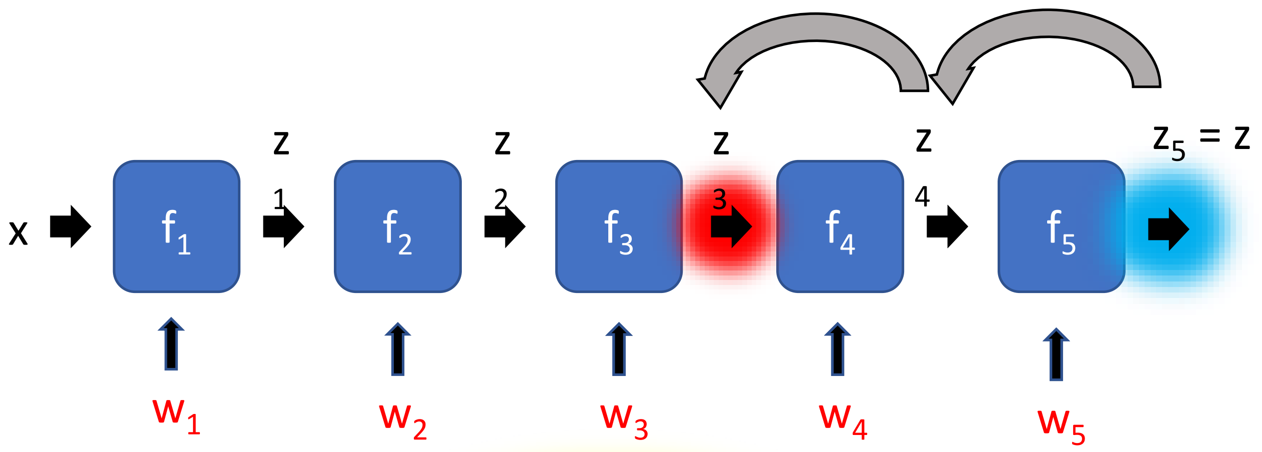
$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

The gradient of convnets



$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

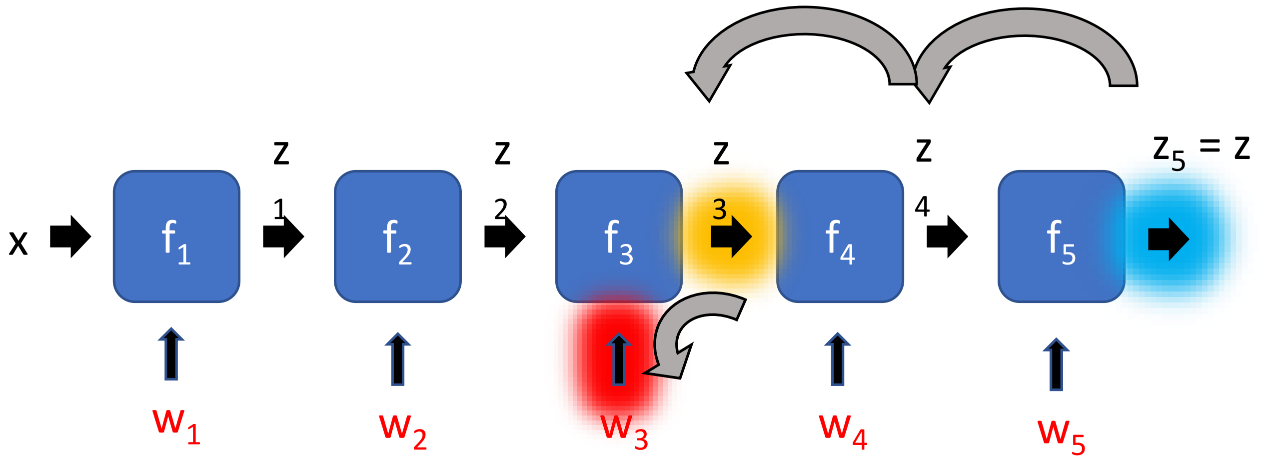
The gradient of convnets



$$\frac{\partial z}{\partial z_3} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial z_3}$$

$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

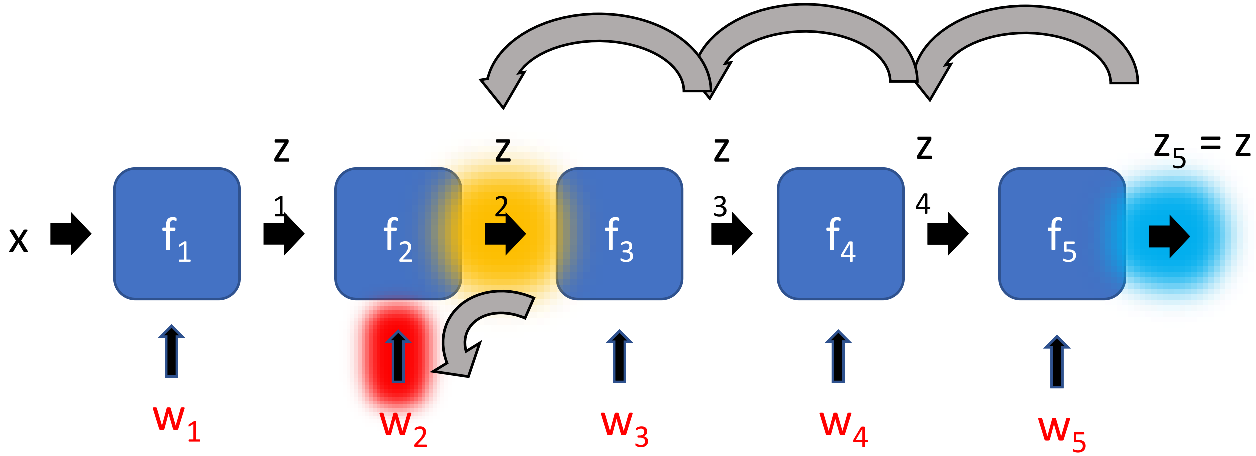
The gradient of convnets



$$\frac{\partial z}{\partial z_3} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial z_3}$$

$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

The gradient of convnets

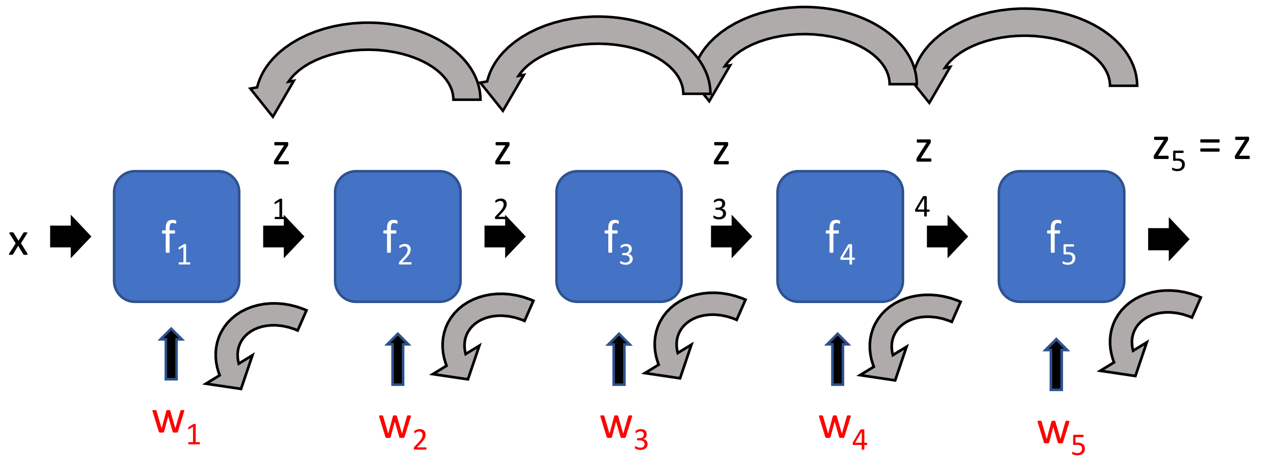


$$\frac{\partial z}{\partial z_2} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial z_2}$$

$$\frac{\partial z}{\partial w_2} = \frac{\partial z}{\partial z_2}\frac{\partial z_2}{\partial w_2}$$

Recurrence going backward!!

The gradient of convnets



Backpropagation

Backpropagation for a sequence of functions

$$z_i = f_i(z_{i-1}, w_i)$$
$$z_0 = x$$
$$z = z_n$$

Previous term

$$\frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial z_i}$$

Function derivative

$$\frac{\partial z}{\partial w_i} = \frac{\partial z}{\partial z_i} \frac{\partial z_i}{\partial w_i}$$

Backpropagation for a sequence of functions

$$z_i = f_i(z_{i-1}, w_i) \qquad z_0 = x \qquad z = z_n$$

- Assume we can compute partial derivatives of each function

$$\frac{\partial z_i}{\partial z_{i-1}} = \frac{\partial f_i(z_{i-1}, w_i)}{\partial z_{i-1}} \qquad \frac{\partial z_i}{\partial w_i} = \frac{\partial f_i(z_{i-1}, w_i)}{\partial w_i}$$

- Use $g(z_i)$ to store gradient of z w.r.t z_i, $g(w_i)$ for w_i
- Calculate g_i by iterating backwards

$$g(z_n) = \frac{\partial z}{\partial z_n} = 1 \qquad g(z_{i-1}) = \frac{\partial z}{\partial z_i}\frac{\partial z_i}{\partial z_{i-1}} = g(z_i)\frac{\partial z_i}{\partial z_{i-1}}$$

- Use gi to compute gradient of parameters

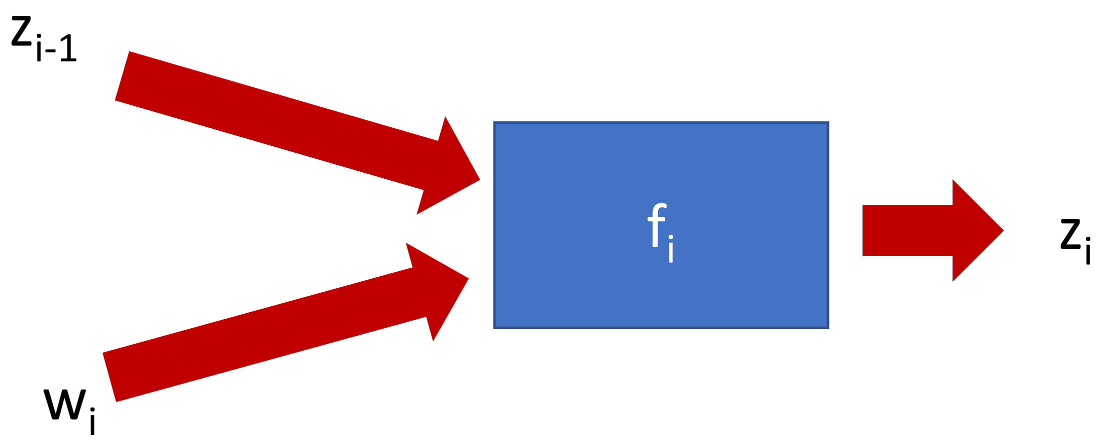
$$g(w_i) = \frac{\partial z}{\partial z_i}\frac{\partial z_i}{\partial w_i} = g(z_i)\frac{\partial z_i}{\partial w_i}$$

Backpropagation for a sequence of functions

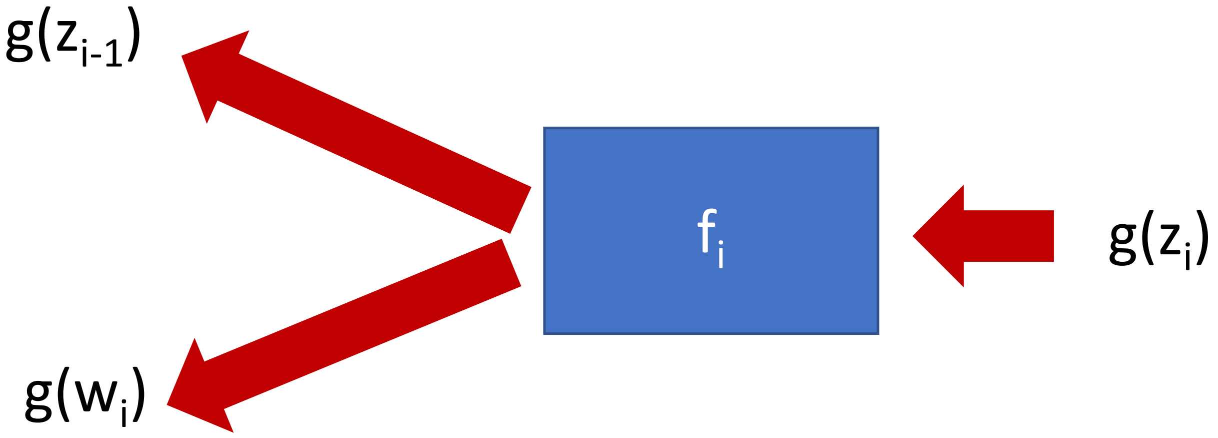
- Each “function” has a “forward” and “backward” module
- Forward module for f_i
  - takes z_{i-1} and weight w_i as input
  - produces z_i as output
- Backward module for f_i
  - takes $g(z_i)$ as input
  - produces $g(z_{i-1})$ and $g(w_i)$ as output

$$g(z_{i-1}) = g(z_i)\frac{\partial z_i}{\partial z_{i-1}} \qquad g(w_i) = g(z_i)\frac{\partial z_i}{\partial w_i}$$

Backpropagation for a sequence of functions



Backpropagation for a sequence of functions



Chain rule for vectors

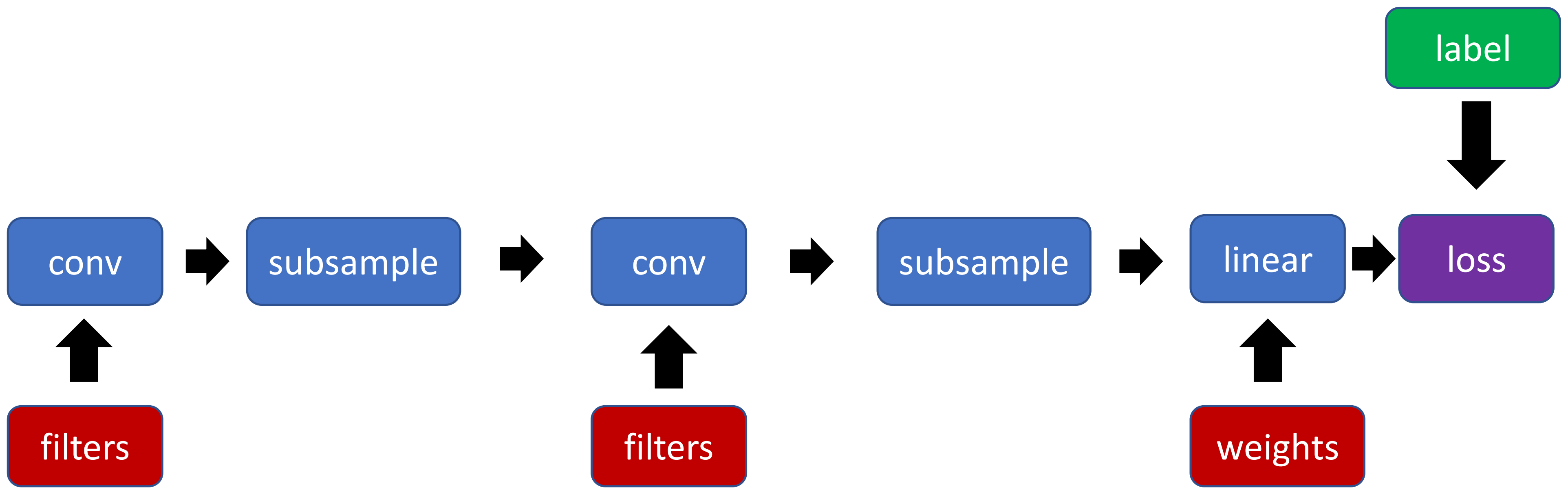
$$\frac{\partial a}{\partial b} = \frac{\partial a}{\partial c}\frac{\partial c}{\partial b} \qquad \frac{\partial a_i}{\partial b_j} = \sum_k \frac{\partial a_i}{\partial c_k}\frac{\partial c_k}{\partial b_j}$$

$$\frac{\partial \mathbf{a}}{\partial \mathbf{b}}(i,j) = \frac{\partial a_i}{\partial b_j}$$

Jacobian

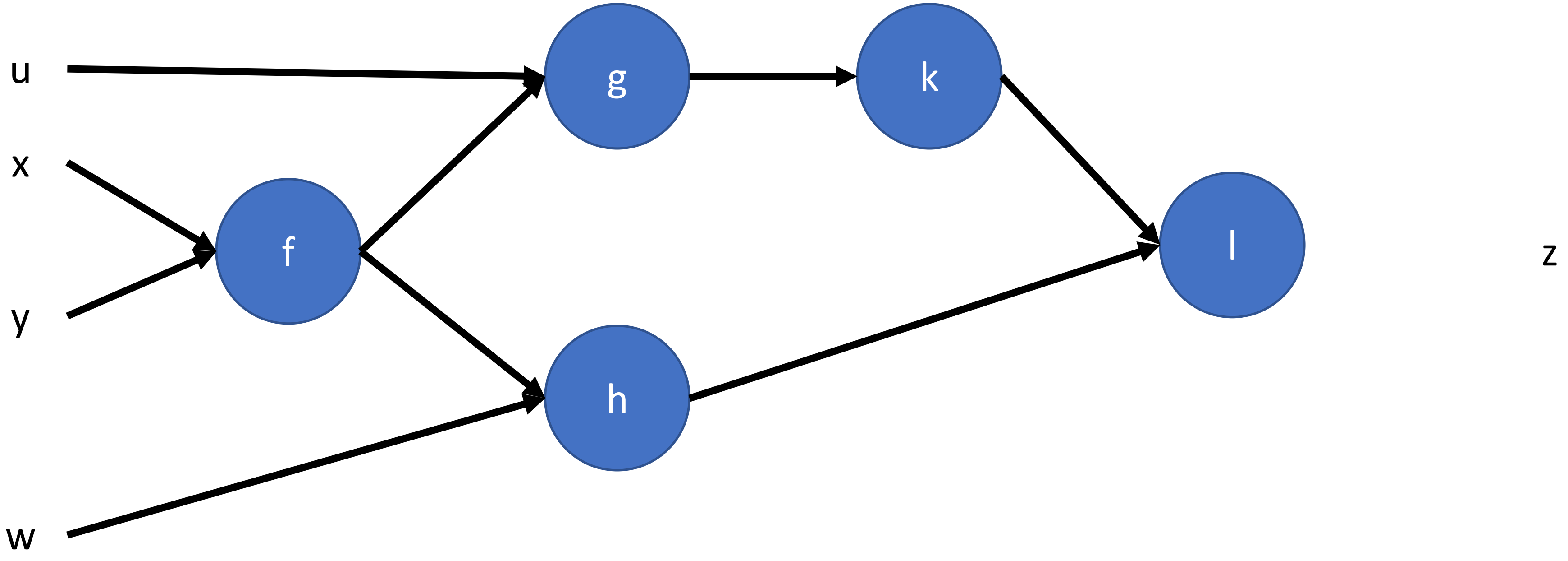
$$\frac{\partial \mathbf{a}}{\partial \mathbf{b}} = \frac{\partial \mathbf{a}}{\partial \mathbf{c}}\frac{\partial \mathbf{c}}{\partial \mathbf{b}}$$

Loss as a function



Beyond sequences: computation graphs

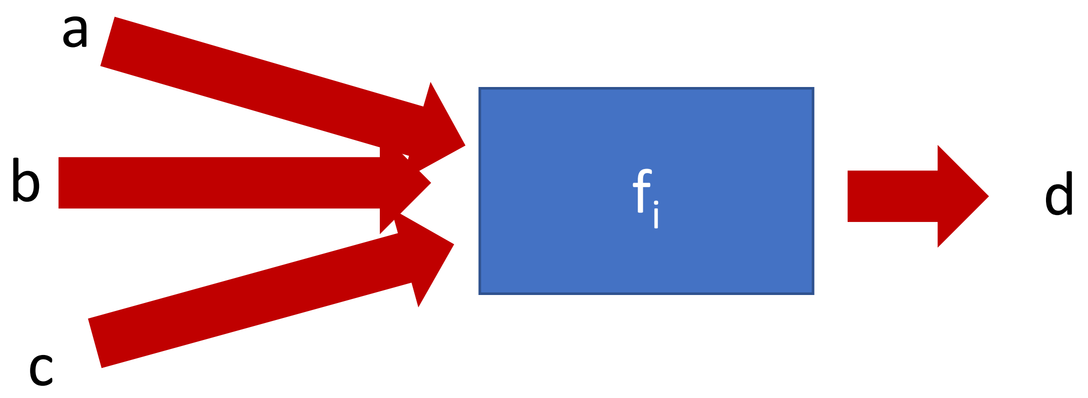
- Arbitrary *graphs* of functions
- No distinction between intermediate outputs and parameters



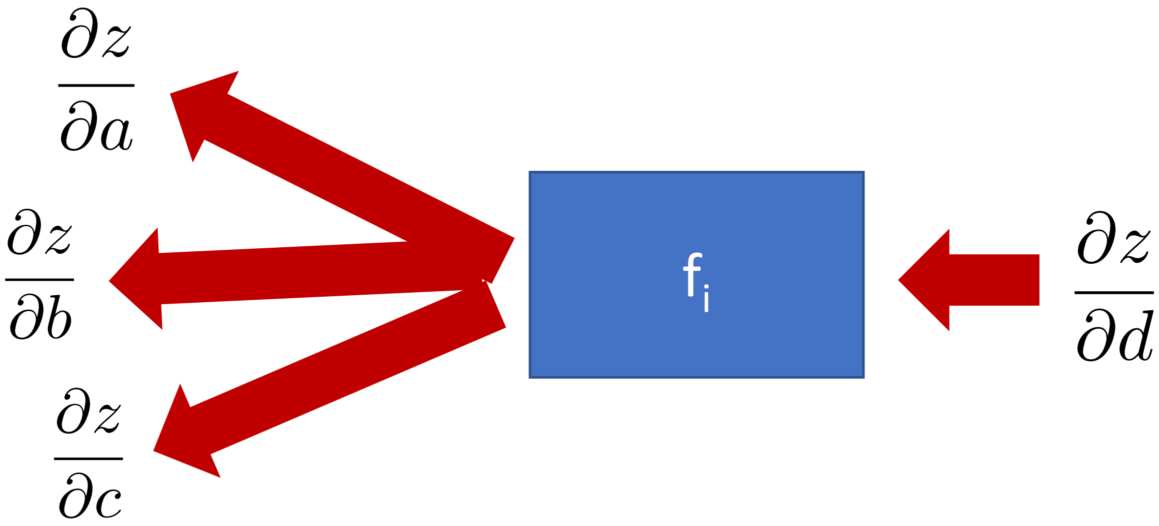
Computation graph - Functions

- Each node implements two functions
  - A “forward”
    - Computes output given input
  - A “backward”
    - Computes derivative of z w.r.t input, given derivative of z w.r.t output

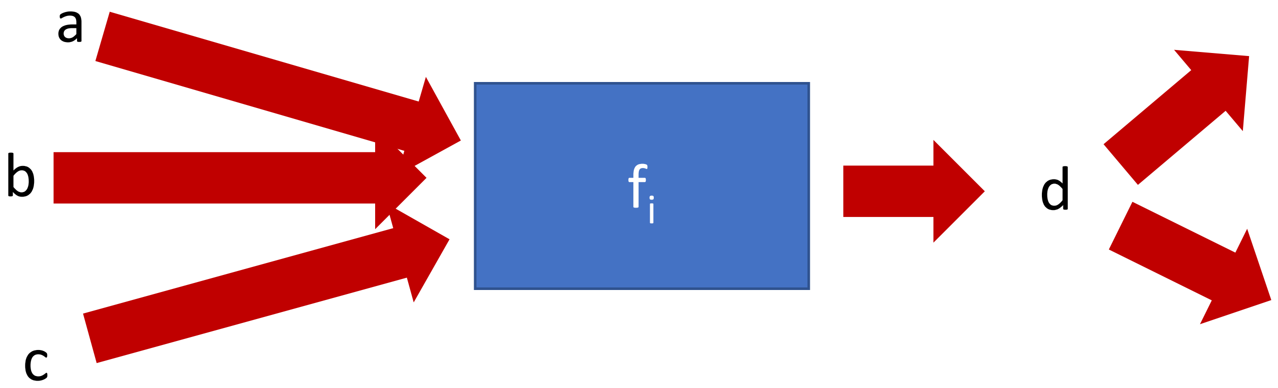
Computation graphs



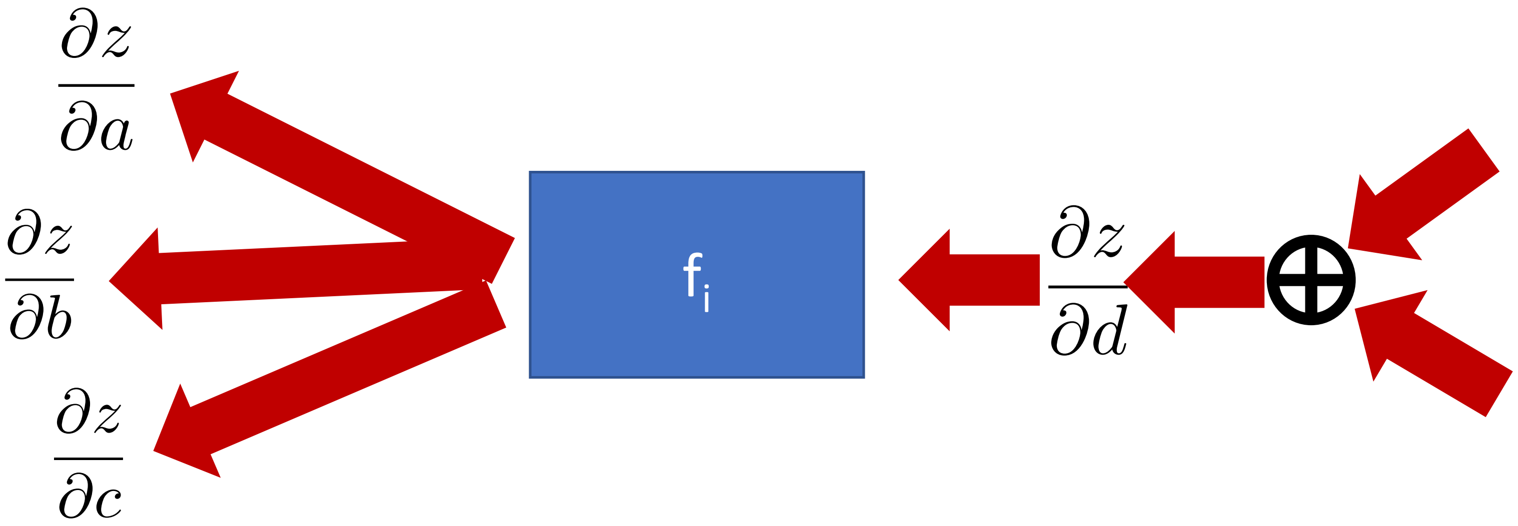
Computation graphs



Computation graphs



Computation graphs



Neural network frameworks

PYTORCH

Caffe2

mxnet