Image representations

Idea:

Filter banks + ReLU yield scores for many different patterns

Idea:

You can compose this, so patterns of patterns of …

Idea:

if the filters are well chosen, you could use the representation to:

- denoise images
- find edges
- find interest points
- classify images…

Learned Image Representations

Idea:

- Filtering an image and applying a ReLU produces a simple pattern detector
- It is also, roughly, invertible
  - Reconstruct filter responses from ReLU outputs
  - Reconstruct image from responses (conv. Theorem)

Notice:

- Last step of reconstruction is filtering

Now apply many different pattern detectors

You have an overcomplete representation

redundant information

one local image patch is scored against many different patterns

downside:

representation is larger

upside:

you can recover image despite some errors in representation

Reconstruction for many different filters+ReLUs

Eg least squares: find the image that produces the representation closest to the one observed

Now imagine input image is noisy:

The least squares reconstruction might not be

(with some care and some luck)

Idea:

Build device that can accept noisy image, produce clean

Denoising autoencoder

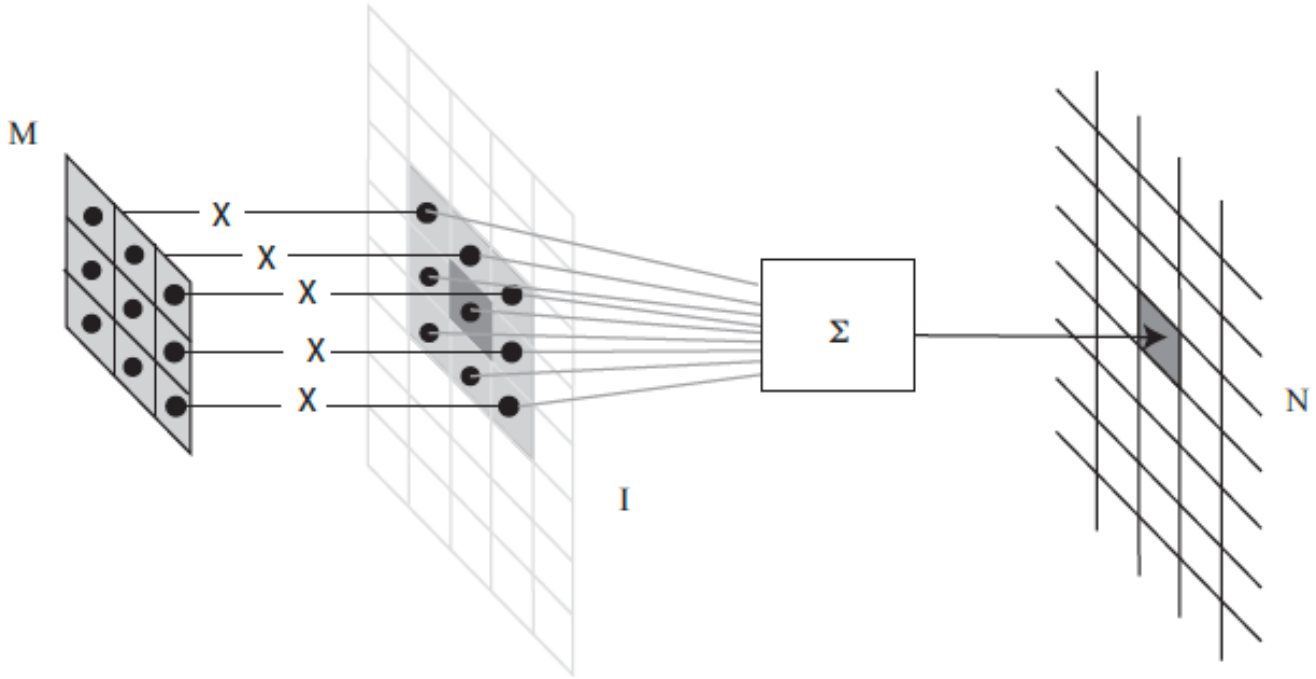
Learned Image Representations

Idea:

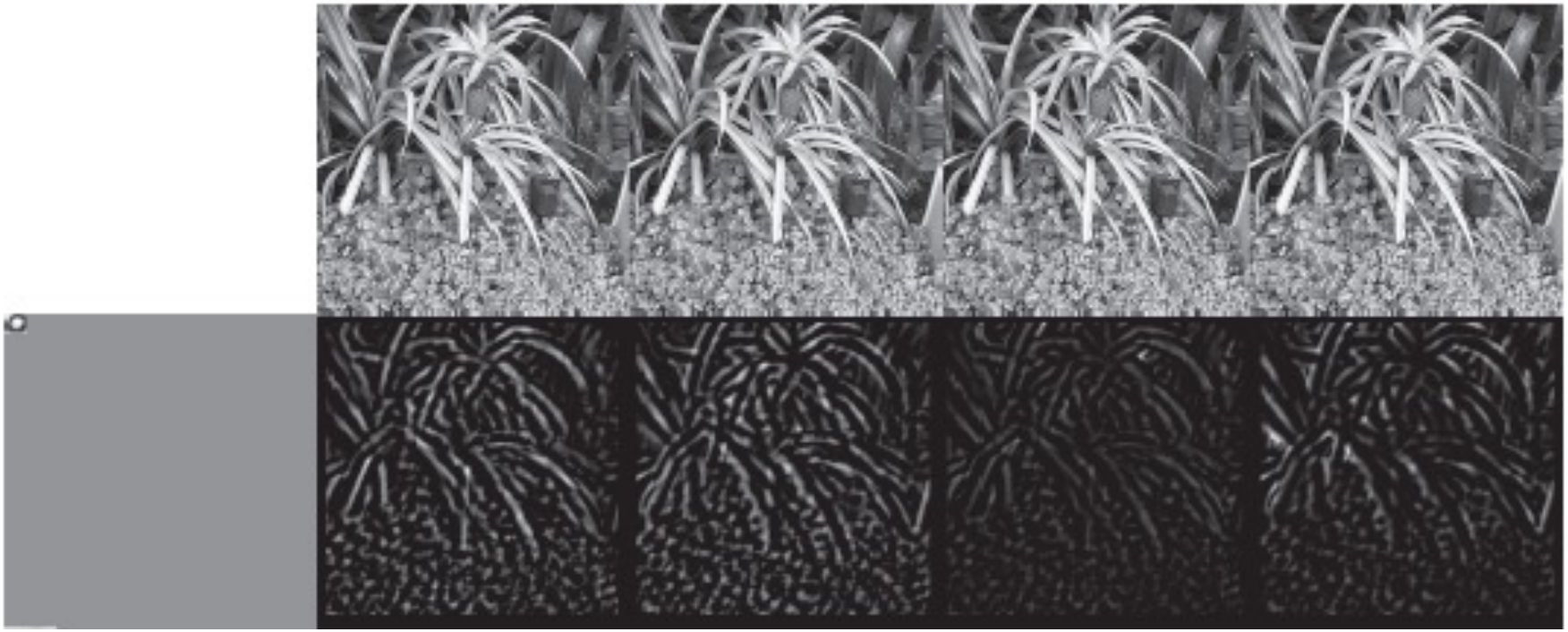
It might be possible to produce an image representation from a lot of filters

AND reconstruct the image from the representation using a lot more filters

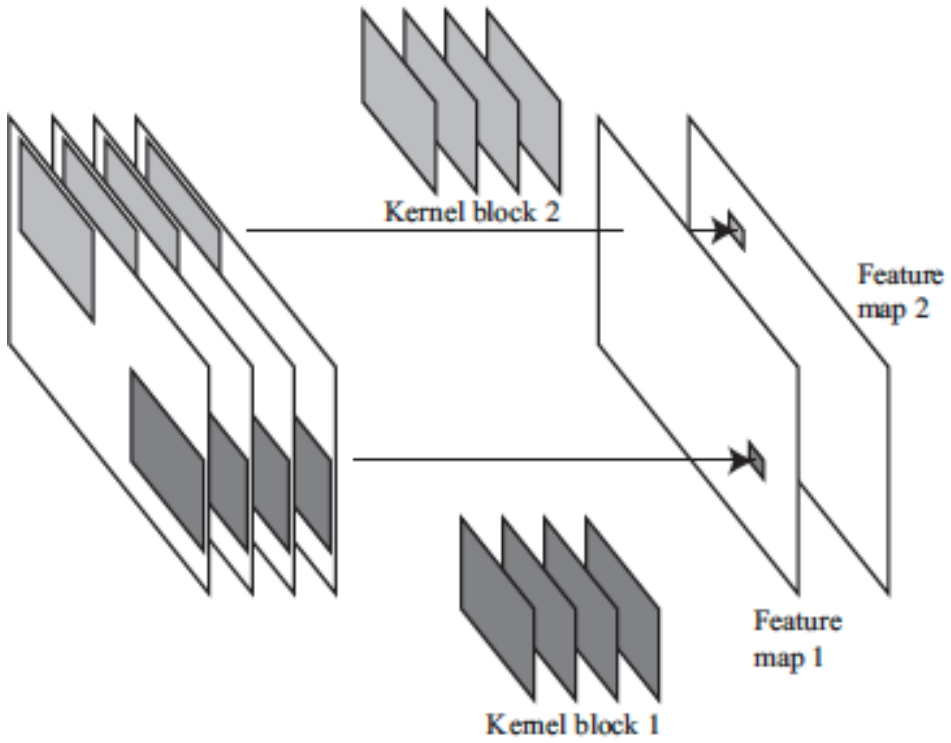
Recall convolution

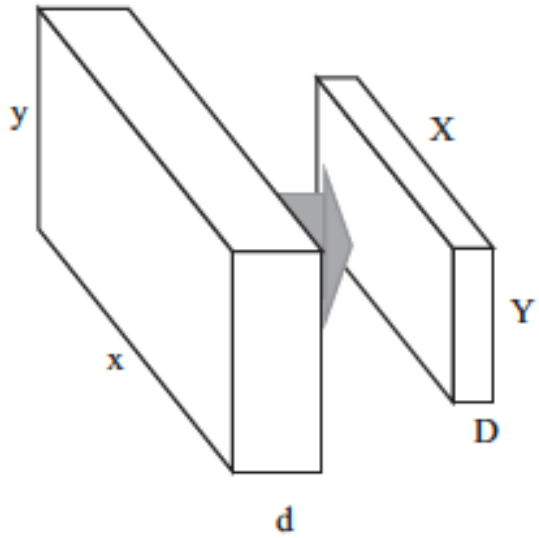


Recall Convolution + ReLU = pattern detector

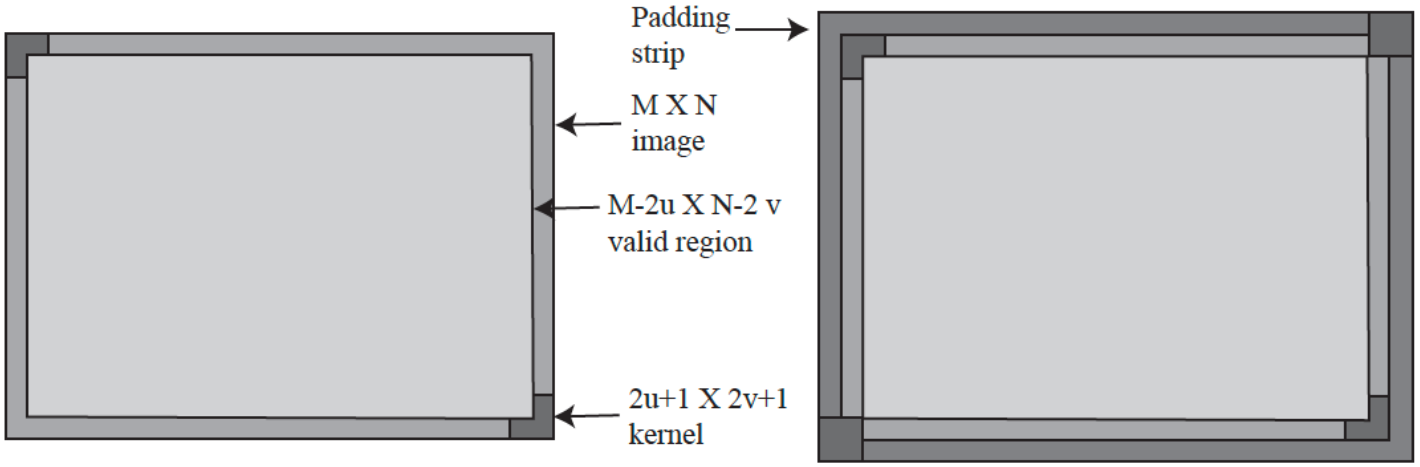


Recall Multi-channel Convolution





Recall padding



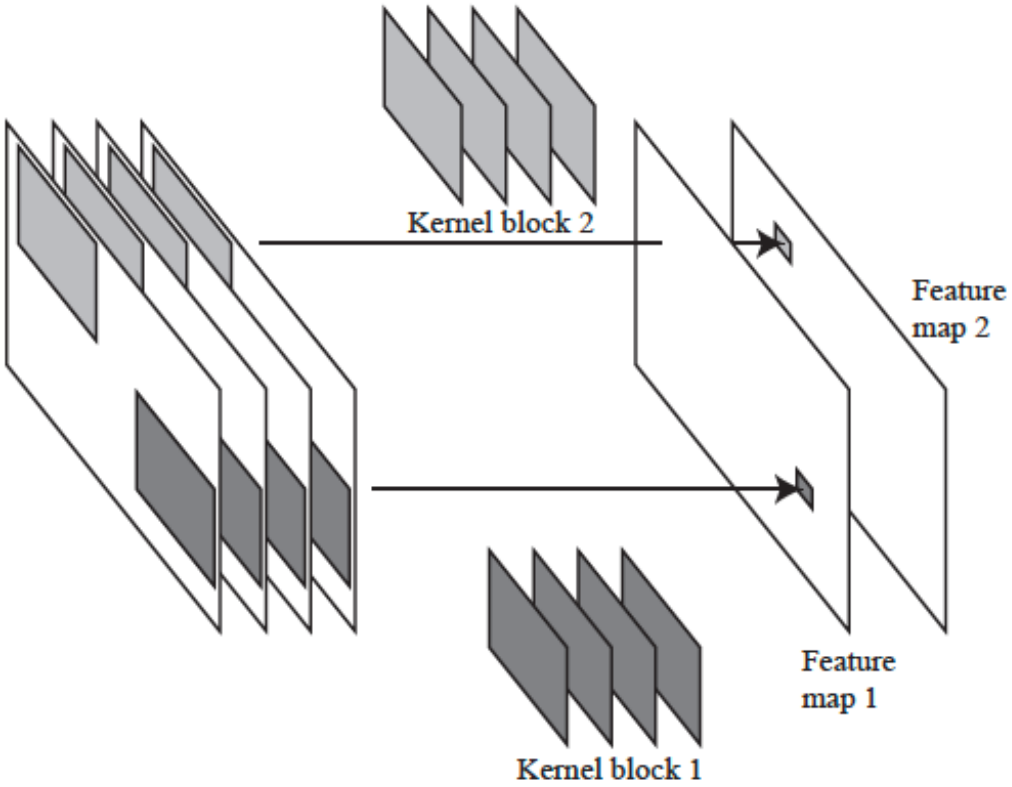
Stride

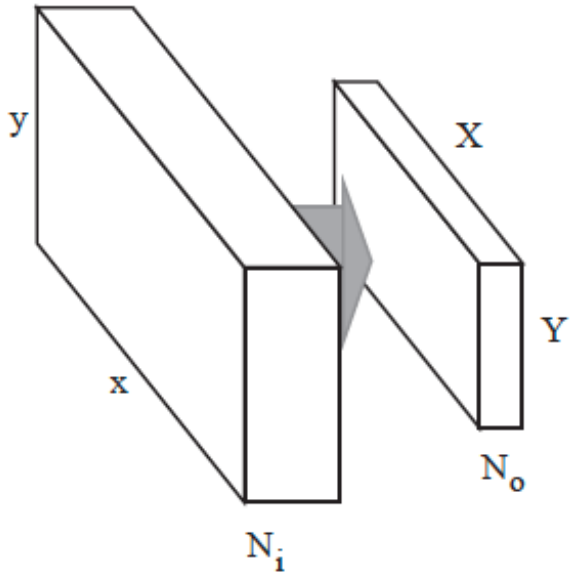
How far across/down to go to the next pixel?

Stride 1: what we're used to

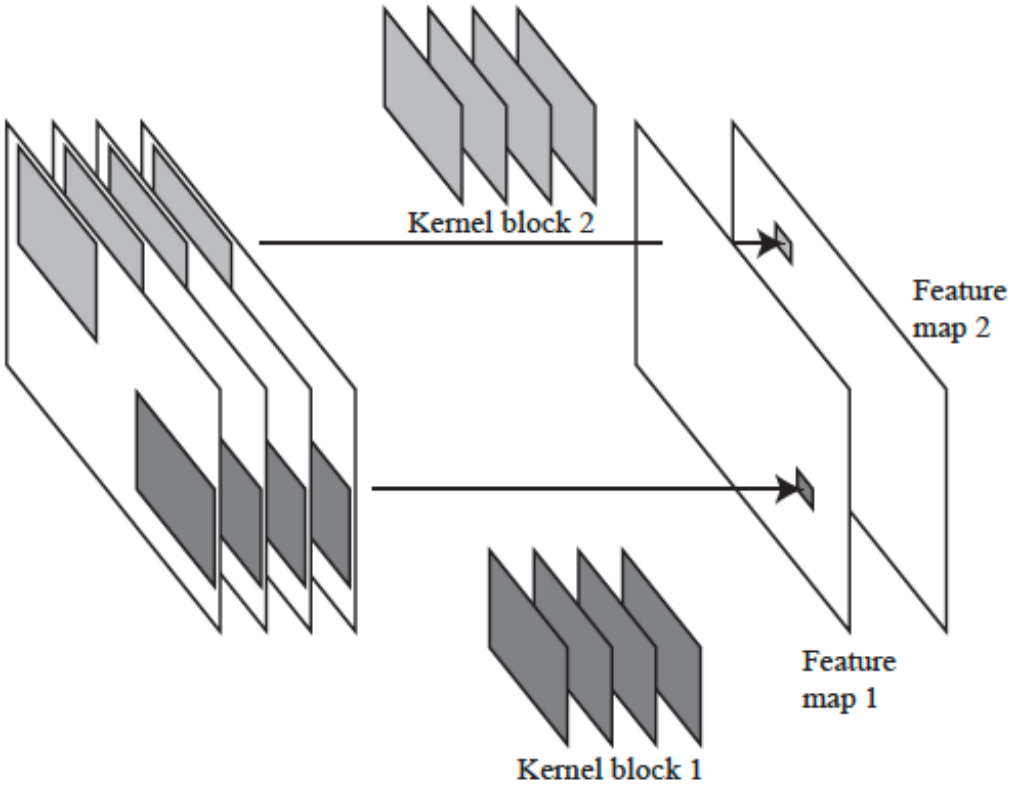
Stride 2: place the kernel on every second pixel

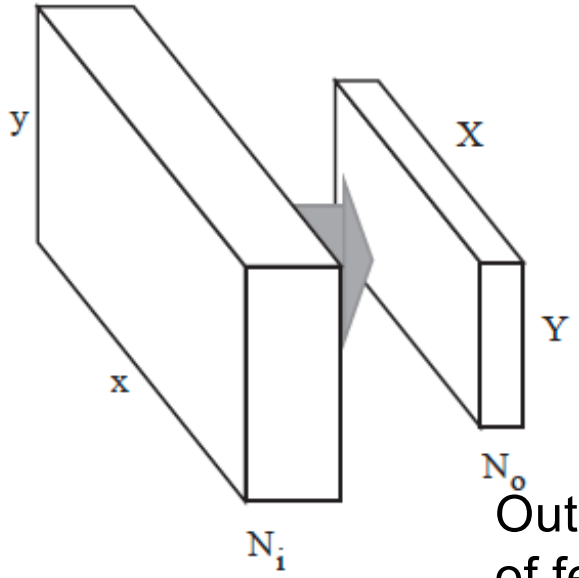
Convolutional Layers





Convolutional Layers

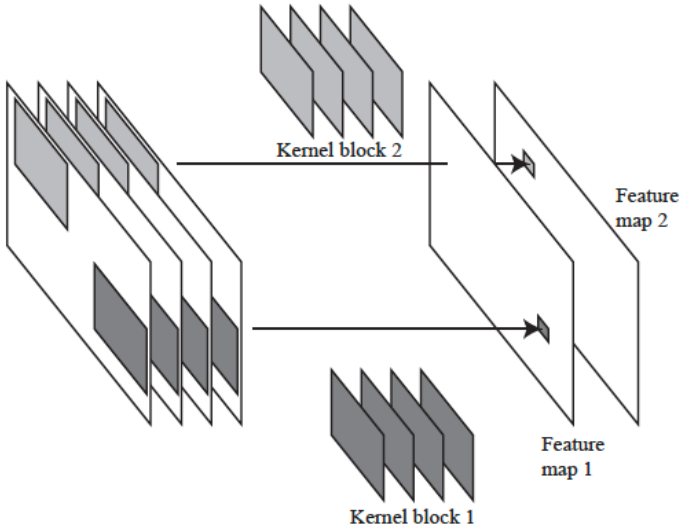


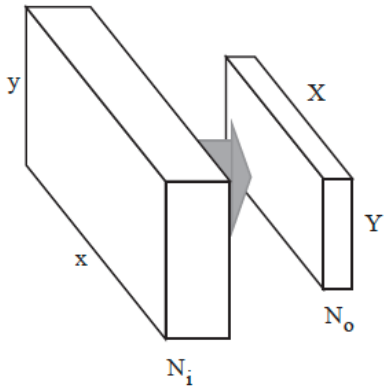


Input number of features

Output number of features

Convolutional Layers





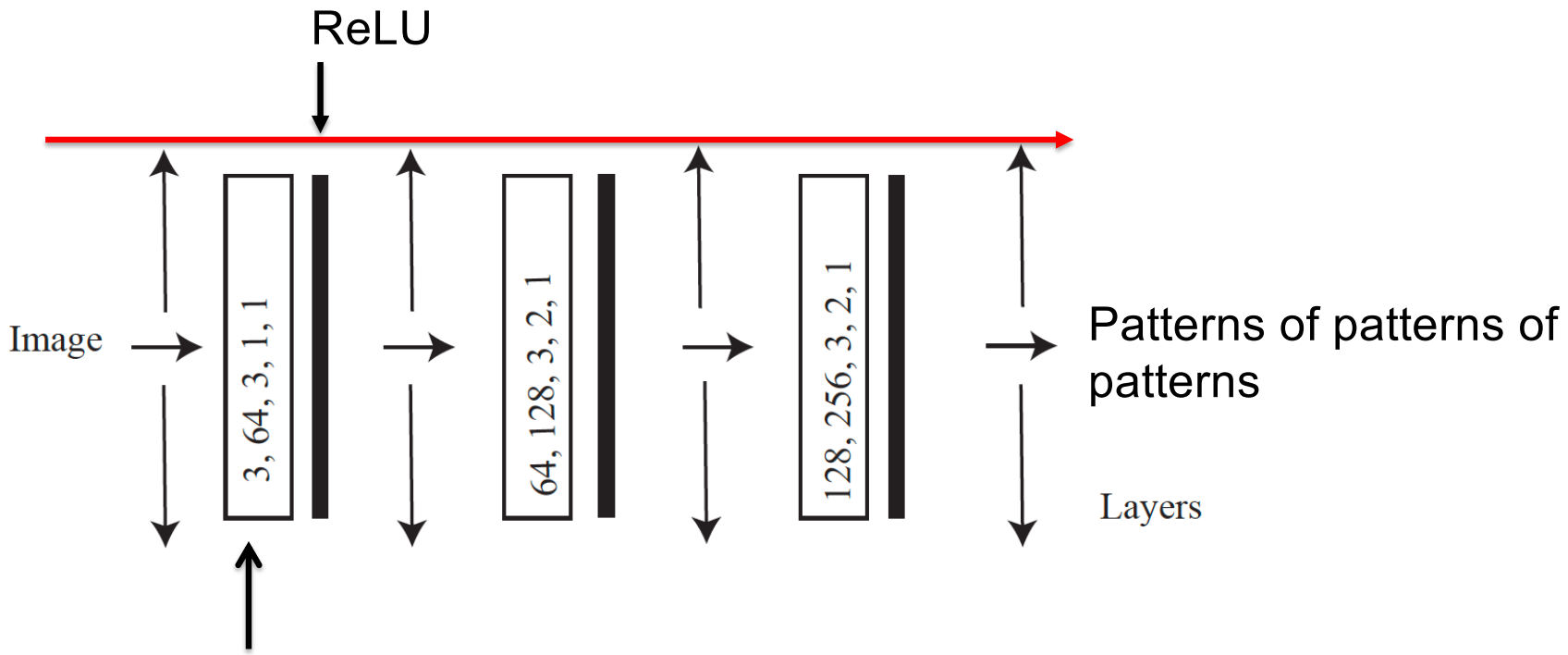
Output size will be determined by:

- input size,
- kernel size,
- padding,
- stride,

ReLU operates on data block

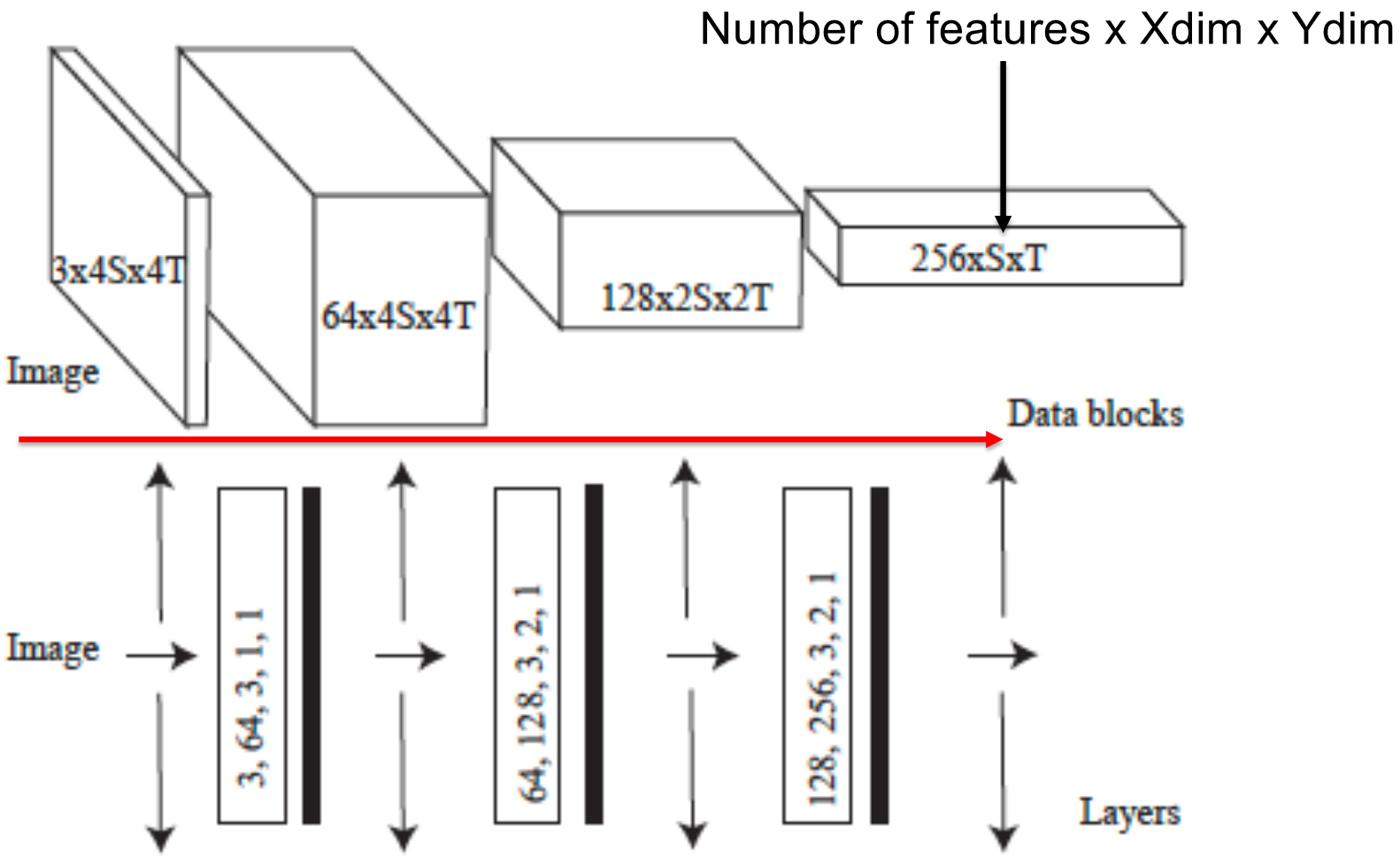
Trivially – just ReLU at each location

A very simple encoder



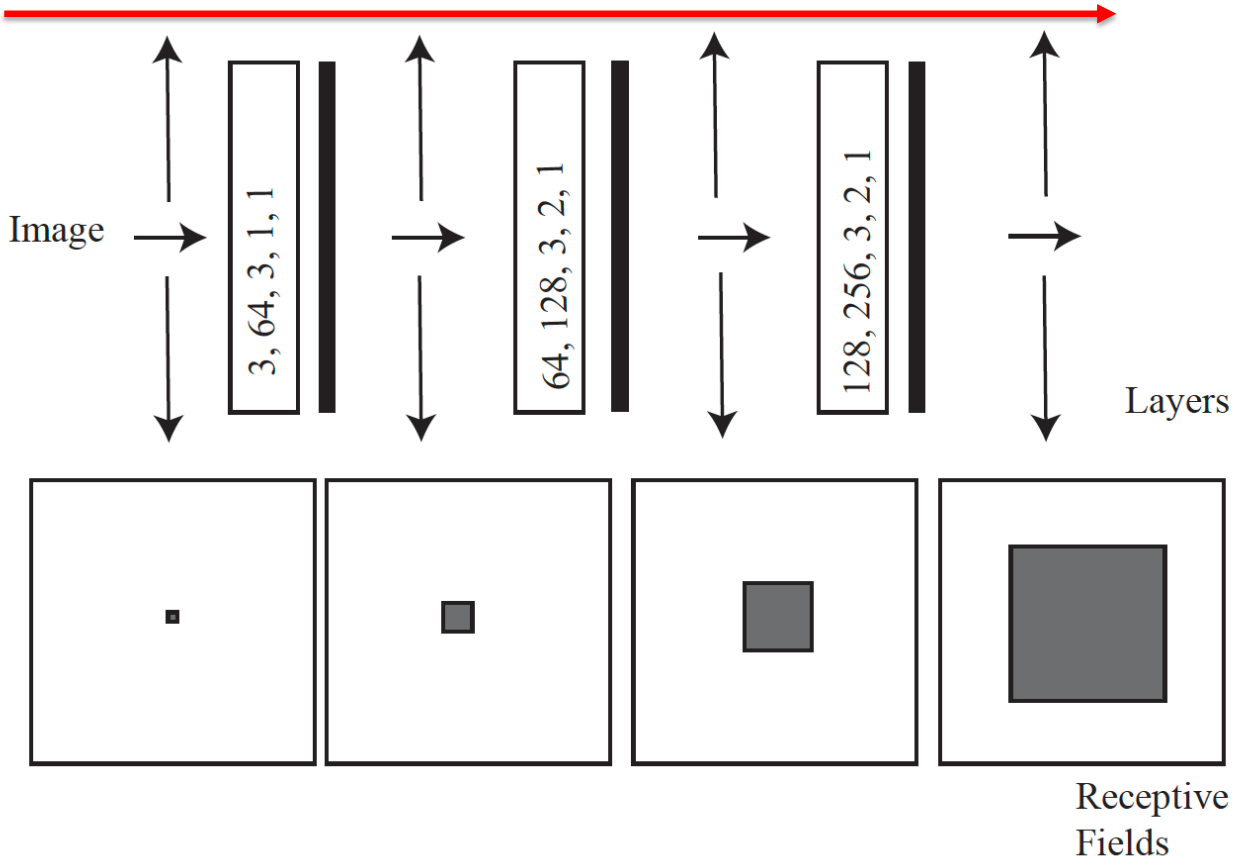
Input number of features, output number of features, kernel size, padding, stride

As data blocks



Receptive fields

Support for a value in the feature map



Decoding

Want:

map rep'n (patterns of patterns of patterns…) to image

Have:

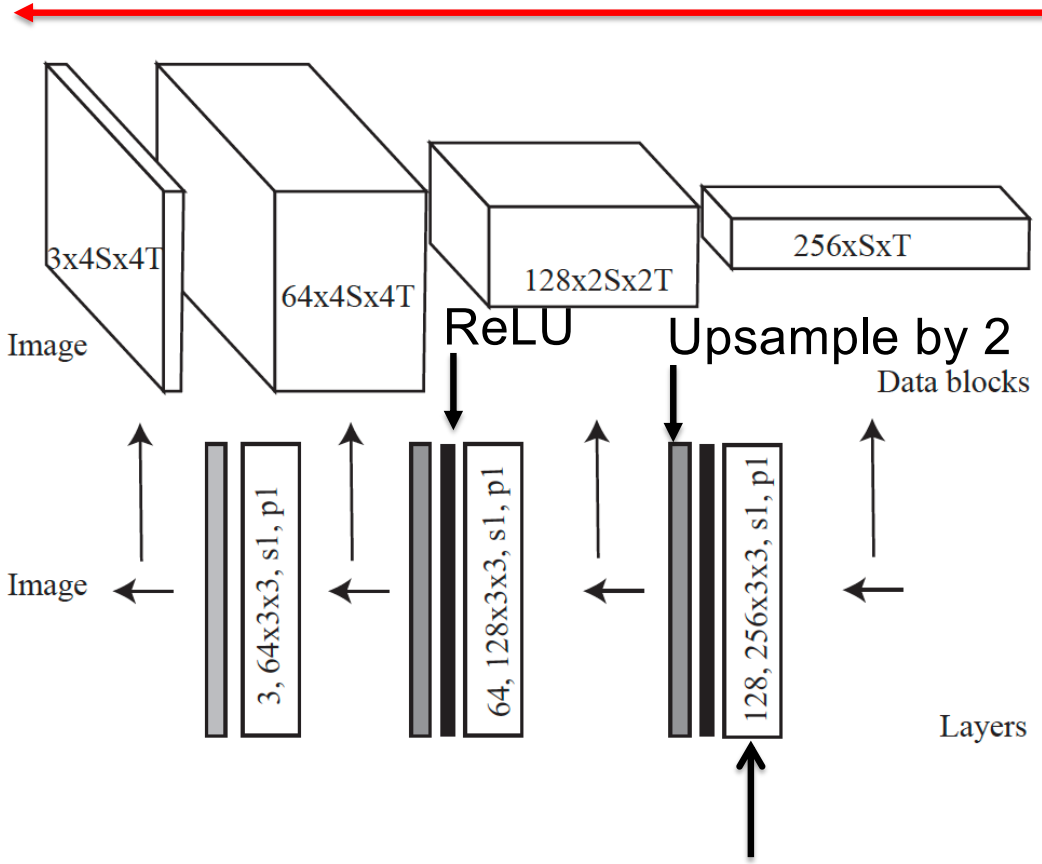
If rep'n is filter outputs, convolution is enough

Rep'n is spatially smaller than image

Idea:

Filter+ReLU+upsample on occasion might do it

A decoder



Patterns of patterns patterns

Input number of features, output number of features, kernel size, padding, stride

Big idea

With the right choice of filters in encoder and decoder
a decoder could reconstruct an image from an encoders rep'n

The rep'n is overcomplete, and "sees" the image at many scales
so the pair should be able to denoise

But what is the right choice of filters?

Learning the filters

Procedure:

find many training pairs (noisy image, clean image)

adjust filters so that

Decode(Encode(noisy image)) is close to clean image

on average, over pairs

hope that this generalizes to new images

Result:

Denoising autoencoder

Find many training pairs

No noise – system might ”cheat”

- Produce a representation that isn’t useful

What noise should you use?

- Options:
  - Gaussian (but a fairly simple filter will deal with this)
  - Poisson (median filter)
  - knock out blocks of pixels (more challenging, and helpful)
  - etc.

Adjusting the filters: notation

Write $\mathcal{E}(\cdot;\psi)$ for an encoder which accepts an image (in the $\cdot$ slot), produces an encoding, and has parameters ψ (the filter banks). Write $\mathcal{D}(\cdot;\phi)$ for a decoder that accepts an encoding ($\cdot$ slot again), produces an image, and has parameters ϕ (the filter banks). Stack the ψ and ϕ into one vector θ. Write $\mathcal{S}$ for a set of N training images. The i'th image is $\mathcal{I}_i$.

Adjusting the filters: loss

The autoencoder produces some image $\mathcal{O}(\mathcal{I}, \theta) = \mathcal{D}(\mathcal{E}(\mathcal{I}; \psi); \phi)$ when given $\mathcal{I}$. Construct a cost function $\mathcal{C}(\mathcal{O}(\mathcal{I}, \theta), \mathcal{I}_i)$ that compares the output of the autoencoder to $\mathcal{I}$. This cost function is typically a weighted combination of the L2 norm and the L1 norm (Section 9.2.2).

Now write

$$\mathcal{L}_{\mathcal{S}}(\theta) = \frac{1}{N} \sum_{i \in \mathcal{S}} \mathcal{C}(\mathcal{O}(\mathcal{I}_i, \theta), \mathcal{I}_i)$$

for the *loss* – an average over a set $\mathcal{S}$ of images of the cost per image. The problem is to find a θ that produce an acceptably small value of the loss. In an ideal world, $\mathcal{S}$ would be all possible images, but this isn't practical. Instead, train on some large set of images (the *training set*). If this set is large enough and representative enough, expect that the autoencoder will also have low loss on other images, a property called *generalization*.

Adjusting the filters: optimization problem, but weird

$$\mathcal{L}_{\mathcal{S}}(\theta) = \frac{1}{N} \sum_{i \in \mathcal{S}} \mathcal{C}(\mathcal{O}(\mathcal{I}_i, \theta), \mathcal{I}_i)$$

Issues:

- The cost function is very hard to evaluate (N is big)
- There are lots of parameters (millions-billions)
  - so no newton's method
- You don't actually want an optimum
  - you want a set of filters that works well on other images

Stochastic gradient descent

$$\mathcal{L}_{\mathcal{S}}(\theta) = \frac{1}{N} \sum_{i \in \mathcal{S}} \mathcal{C}(\mathcal{O}(\mathcal{I}_i, \theta), \mathcal{I}_i)$$

Loss is a population mean

you can estimate this quite well with a sample mean

draw a small batch, average over that

Stochastic gradient descent

In the case of the loss function, choose a sample size B – usually called a *batch size* – draw $\mathcal{B}$, a set of B images $\mathcal{I}_j$ drawn uniformly and at random, and form

$$\nabla_\theta \mathcal{L}_{\mathcal{B}}(\theta) = \frac{1}{B} \sum_{j \in \mathcal{B}} \nabla_\theta \mathcal{C}(\mathcal{I}_j; \theta)$$

and use this as an estimate of

$$\nabla_\theta \mathcal{L}_{\mathcal{S}}$$

to take a descent step. Write

$$\hat{\nabla}_\theta \mathcal{L}$$

for this estimate. Choose a stepsize η_n for the n'th step, and the descent method becomes

$$\theta_{n+1} = \theta_n - \eta_n \hat{\nabla}_\theta \mathcal{L}.$$

This is *stochastic gradient descent* or *SGD*. Calling η_n a stepsize is dubious (the gradient isn't a unit vector); an alternative is to call it the *learning rate* (which isn't much better because it isn't a rate).

Stochastic gradient descent

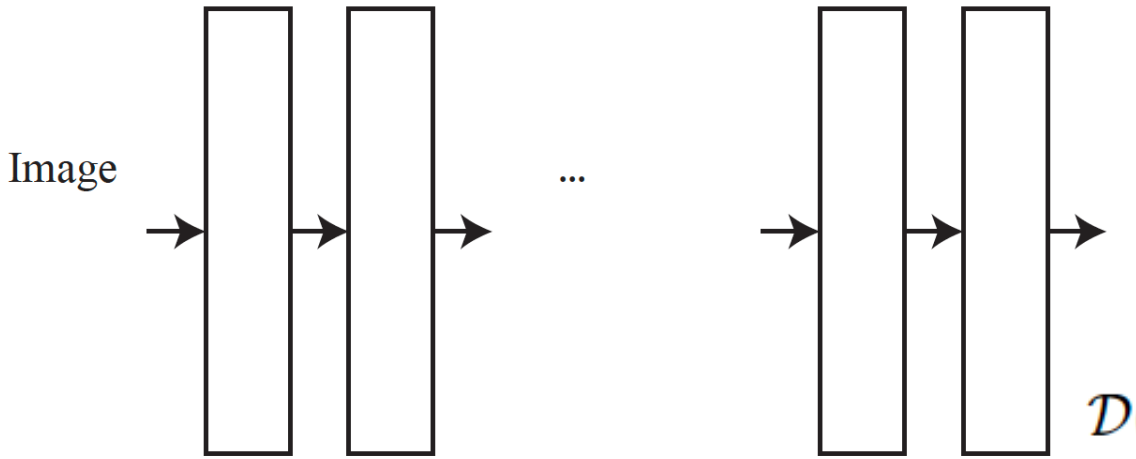
$$\theta_{n+1} = \theta_n - \eta_n \hat{\nabla}_\theta \mathcal{L}.$$

How big a step?

- Line search
  - you can't – N is too big
- Fixed length
  - too big (doesn't settle down)
  - too small (no progress)
- Learning rate schedule
  - start biggish, take steps, make smaller
  - how big is biggish? try

Evaluating the gradient

$$\hat{\nabla}_\theta \mathcal{L} = \frac{1}{B} \sum_{j \in \mathcal{B}} \nabla_\theta \mathcal{C}(\mathcal{O}(\theta), \mathcal{I}_j).$$



$$\begin{aligned}
\mathcal{D}(\mathcal{E}(\mathcal{I};\psi);\phi) &= B_{k+1} \\
&\text{where} \\
B_{k+1} &= L_k(B_k;\theta_k) \\
B_k &= L_{k-1}(B_{k-1};\theta_{k-1}) \\
&\cdots \\
B_1 &= \mathcal{I}
\end{aligned}$$

Recursion from chain rule (Backpropagation)

$$
\begin{aligned}
\mathbf{u}_0^T &= \nabla_{\mathcal{O}} \mathcal{C}^T \\
\nabla_{\theta_k} \mathcal{C} &= \mathbf{u}_0^T \mathcal{J}_{L_k;\theta_k} \\
\mathbf{u}_1^T &= \mathbf{u}_0^T \mathcal{J}_{L_k;B_k} \\
\nabla_{\theta_{k-1}} \mathcal{C} &= \mathbf{u}_1^T \mathcal{J}_{L_{k-1};\theta_{k-1}} \\
&\dots \\
\mathbf{u}_r &= \mathbf{u}_{r-1}^T \mathcal{J}_{L_{k-r+1};B_{k-r+1}} \\
\nabla_{\theta_{k-r}} \mathcal{C} &= \mathbf{u}_r \mathcal{J}_{L_{k-r};\theta_{k-r}} \\
&\dots \\
\nabla_{\theta_1} \mathcal{C} &= \mathbf{u}_{k-1} \mathcal{J}_{L_1;\theta_1}
\end{aligned}
$$

Derivatives of layer outputs with respect to parameters

Derivatives of layer outputs with respect to inputs

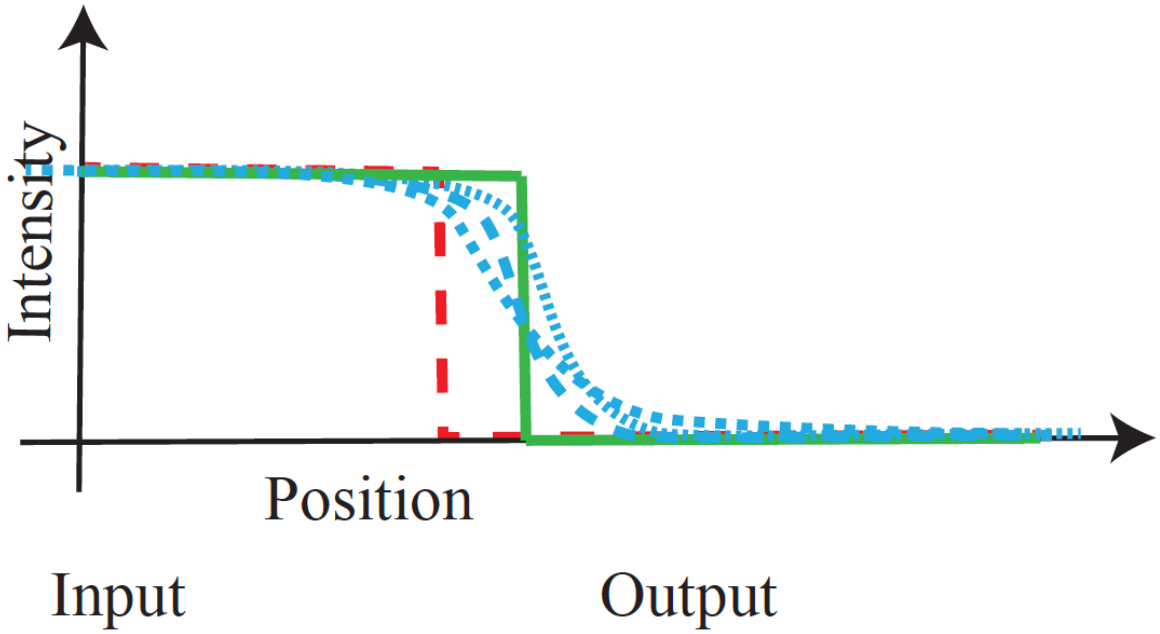
Losses – the L2 loss

Loss functions typically evaluate *residuals* – the difference between what the system provides and ground truth. The *SSD loss* compares a reconstructed training image $\mathcal{R}$ to the ground truth $\mathcal{G}$ by

$$\mathcal{C}_{L2}(\mathcal{R}, \mathcal{G}) = \sum_{ij} \Delta_{ij}^2,$$

where $\Delta_{ij} = \mathcal{R}_{ij} - \mathcal{G}_{ij}$ is the *residual*. This is the square of the L2 norm of Δ, and is sometimes (rather disreputably) referred to as an *L2 loss*. This might seem a

L2 loss creates blur



Sharp edge in the wrong place (red) is expensive

Compared to blurry edge in about the right place (blue)

Input

Output

L1 loss

the gradient to have zeros, assuming the optimization process can cope. Using an L1 term, written

$$\mathcal{C}_{L1}(\mathcal{R},\mathcal{G}) = \sum_{ij} |\Delta_{ij}|$$

will tend to encourage the residual to have zeros in it, and will tend to discourage blurring (Figure **??**).

Idea:

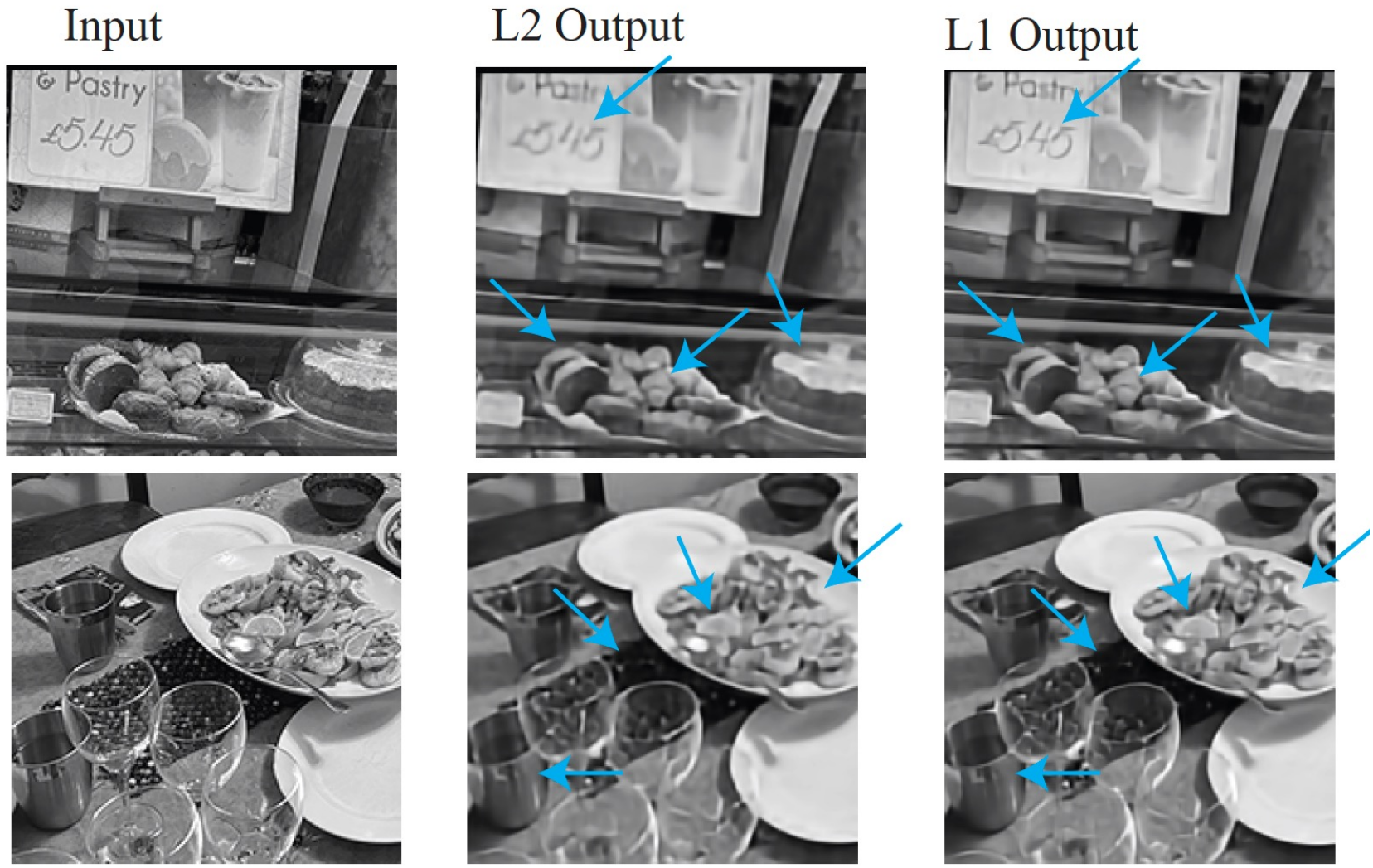
penalize absolute value of residual

We saw the L1 norm in denoising

Square of a small number is very small; absolute value of small number isn't

Tends to discourage blurring

L1 loss



General ideas: Losses and Gradients

Notice that L1 loss, L2 loss DON'T:

- Force values to be non-negative
- Force values to be less than 1

(VERY BAD) Idea:

Here is an example of a bad loss. The *indicator function* is a function that tests its argument against a condition, then reports 1 if the condition is true and zero otherwise. For example,

$$\mathbb{I}_{[x<0]}(x) = \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

is 1 when $x < 0$ and 0 otherwise. Note some redundancy here; the condition usually means it is obvious what the argument is, so it is quite usual to write $\mathbb{I}_{[x<0]}$ rather than $\mathbb{I}_{[x<0]}(x)$. The following (BAD) choice of loss could be intended to force an output to be non-negative:

$$\mathcal{C}_{\text{bad}}(\mathcal{I}) = \sum_{ij} \mathbb{I}_{[\mathcal{I}_{ij}<0]}$$

General ideas: Losses and Gradients

Bad, because it supplies no gradient

- Pixel is +ve: loss and gradient are zero
- Pixel is –ve: loss is 1, gradient is zero – no information about how to change filters

CF L1 loss:

- Pixel value too large: gradient pushes it down
- Pixel value too small: gradient pushes it up
- Pixel value just right: non-differentiability doesn't matter
  - never happens
  - choose -1<= gradient <=1: everything works fine

General ideas: Cheating

Stochastic gradient descent is a very effective search

- Astonishing, but true
- It might find a solution you don't expect and don't want

Example:

> Here is an example in the simple case of an autoencoder with two encoder layers and two decoder layers. For concreteness, the first layer in this example has 3x3 filters with stride 1. Make one filter that simply reports the image value at the center location (the other filters don't matter). The pixel value is non-negative, and so passes through the ReLU without alteration. The next layer has 3x3 filters and stride 2; this means that mild ingenuity with multiple filters is required to pick out the pixel values to pass on (exercises). Very little further ingenuity is required to ensure the decoder layer produces the original image. Experience shows that searching for parameters using stochastic gradient descent is extraordinarily powerful, and is perfectly capable of finding a set of parameters that cheats like this. This is cheating because the search has minimized the loss function, but the representation isn't actually of any use. Worse, adding layers, filters, and so on might simply increase the scope for cheating while making it more difficult to understand the detailed structure of any particular cheating strategy.

General ideas: Cheating

Cheating example above won’t denoise

It is important to train representation to denoise images

(Noisy, clean) pairs will produce something useful

(clean, clean) pairs won’t!

General ideas: Generalization

We have no particular interest in denoising training images

We want to denoise new images

Options:

- Data augmentation: make training dataset look bigger
- Regularization: make it hard to choose filters that are specialized

General ideas: Data Augmentation

Applies to many learned systems

For now:

- A left/right flipped image is still an image
- An up/down flipped image is still an image
- An image crop is still an image
- Etc

When forming a batch, randomly

- crop, flip, etc. images

General ideas: Regularization

Prefer filters with small coefficients to filters with large coeffs.

A filter with large coeffs that works well on training data might produce an unexpected large response on new data

Discourage filters with large coeffs by penalizing loss

(cost of error on batch) + scale*(penalty for large coeffs)

Known as weight decay

API will do this for you if you ask

General ideas: Regularization

How to choose scale?

Train for many different choices

Evaluate each on held out data to choose

Now re-train using the best value

Evaluate the result on new dataset