Last block:

Build an encoder and a decoder to:

Accept noisy image, produce clean version

By:

Constructing loss

Applying SGD to get minimal loss on training data

Housekeeping

There is a great deal of housekeeping:

If you build your own from scratch, you will suffer (but learn)
and it won't work all that well (missing tricks)

Use an API (list in notes)

Build from someone else's code (Github, Huggingface)

Missing tricks:

Scales and color

Blurred outputs and skip connections

Bad gradients and residual connections

Optimization tricks

Scale and color

Input scales:

- Helpful to scale, offset pixel values so that mean is zero and standard deviation is one

Output scales:

- Real pixels are in range [0, 1]
- How do we force output to be like this?
  - Options:
    - map value to range
    - penalize over/under
    - bit of both

Value to range

Terrible idea:

You might think that using

$$f(x) = \text{ReLU}(x) - \text{ReLU}(x - 1)$$

would be a good idea (because it maps any x to the range $[0, 1]$). In fact, it is a terrible idea, because once the output is outside that range, there is no gradient to push it back into the range (Section 16.3.1; **exercises**).

NO GRADIENT when you need it!

Value to range

Sigmoid:

$$\text{sigmoid}(x) = \frac{e^x}{1 + e^x}$$

but this creates gradient problems when output close to zero or one

Tanh:

x -> [-1, 1] and easy to get to [0, 1]

similar gradient problems

Penalties

A second strategy is to accept the output of the convolutional layer, but penalize values that appear outside the range you want with a loss term. For example, apply the loss

$$L_{ud}(x) = x^2 \mathbb{I}_{[x<0]} + (x-1)^2 \mathbb{I}_{[x>1]}.$$

Combination

It is often useful to mix these strategies. For example, a final layer that applies

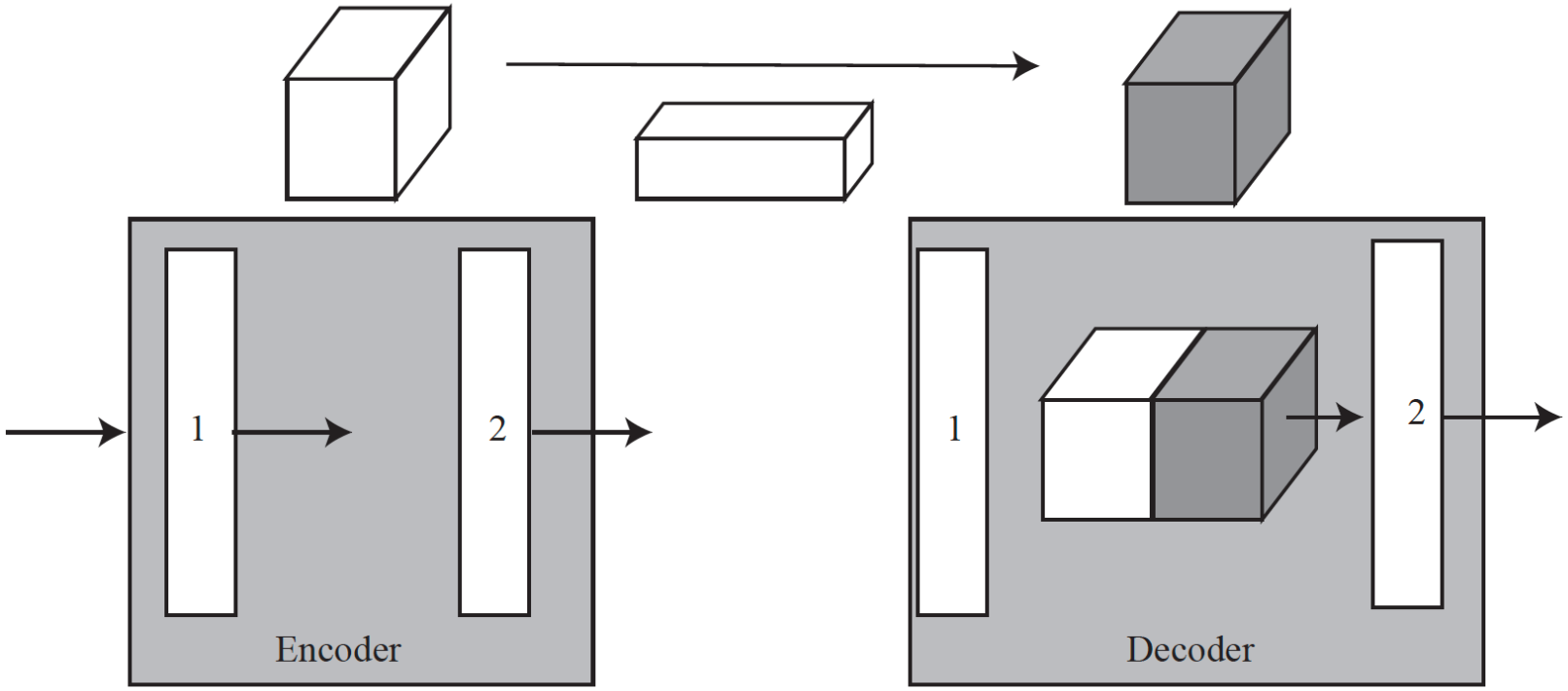
$$f(x) = a(\tanh(x) + b)$$

will map x to the range $(a(b-1), a(b+1))$. If you choose $b < 1$ and $a > 1/(b+1)$, then the output can be below zero (but not much) and above one (but not much). Further, the gradients won't be too small at zero or one. You can then push the outputs to be in that range with a penalty term.

Skip Connections:

An important difficulty presented by stacking many convolutional layers is that any feature produced by the encoder necessarily depends on a fairly large receptive field. This can make it difficult to produce reconstructions with sharp edges. A feature that depends on a very small neighborhood could provide enough information to place an edge accurately – for example, report the gradient of the image. If the receptive field is large, constructing a very local feature that isn't somewhat smoothed will require a set of weights that ignores many or most of the pixel values in the receptive field, which will be difficult to achieve. However, features with large receptive fields may be necessar to denoise, because they can observe long-range trends in the image.

Skip Connections:



Normalization:

Numbers with large magnitude in a neural network cause problems. Imagine some input to some unit is big and the weight applied to that input is small. Then a single gradient step could cause the weight to change sign, and the ReLU might cause the corresponding output to swing between strongly positive and zero. This can cause training problems, because the gradient will be a poor predictor of what will actually happen to the output. Ideally, relatively few values at the input of any layer will have large absolute values. A new layer, sometimes called a *batch normalization layer*, can be inserted between two existing layers to ensure this happens.

Batch normalization

Write $\mathcal{I}$ for the input of this layer, which is a $X \times Y \times F$ block of features, and $\mathcal{O}$ for its output, which is a block of features of the same dimension. The layer has two vectors of parameters, γ and β, each of dimension F. Write γ_i for the i'th component of γ, etc. Assume we know the mean (m_k) and standard deviation (s_k) of each feature in $\mathcal{I}$ computed over the whole dataset and over the spatial dimensions. Write ϵ for a small positive number chosen to avoid divide-by-zero. The data block $\mathcal{U}$, with ijk'th component

$$\mathcal{U}_{ijk} = \frac{(\mathcal{I}_{ijk} - m_k)}{(s_k + \epsilon)}$$

will tend to have small magnitude numbers in it, both positive and negative. The mean of each feature in this block should be about zero, because it is close to the mean over all blocks. The standard deviation of each feature in this block should be about one, because it is close to the standard deviation over all blocks. Now compute

$$\mathcal{O}_{ijk} = \gamma_k \mathcal{U}_{ijk} + \beta_k$$

and notice that $\mathcal{O}$ could be the same as $\mathcal{I}$ (set $\gamma_k = s_k$ and $\beta_k = m_k$). The output of this layer is a differentiable function of γ and β, which can be adjusted to achieve the best performance.

Batch normalization

But we don’t know the mean and standard deviation!

And it keeps changing during training anyhow!

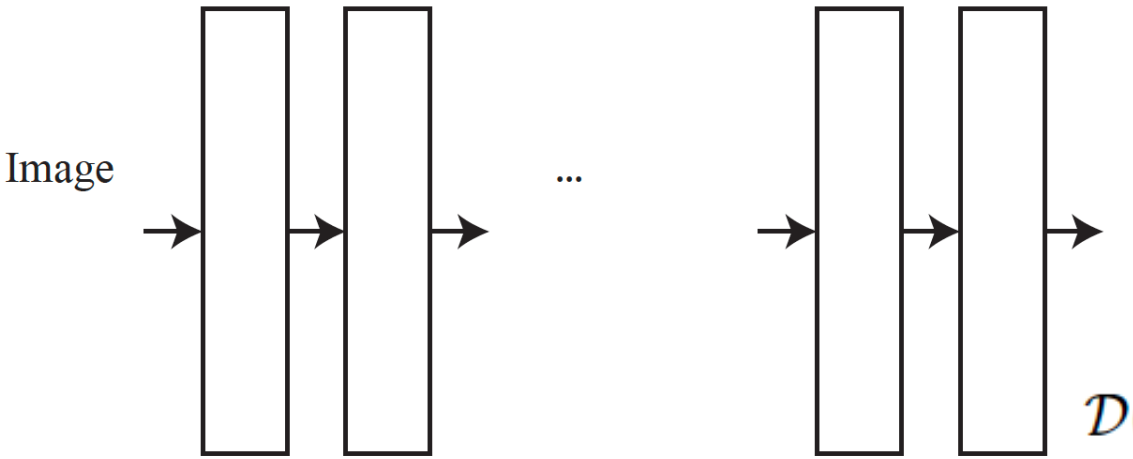
Strategy:

- Use the mean and standard deviation of the last batch at training time
- At test time, use constants (last seen mean/sd)
- API looks after this housekeeping

Landmine:

- You have to tell the networks when you switch from train to test
- If you don’t they work badly

Evaluating the gradient can present problems



Output

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

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

Residual connections

Idea:

Allow information to skip layers so there is always some reliable gradient

Procedure:

For some layers, add input to output

Issue: output might be a different size

Response: use a projection

Residual connections: Simple example

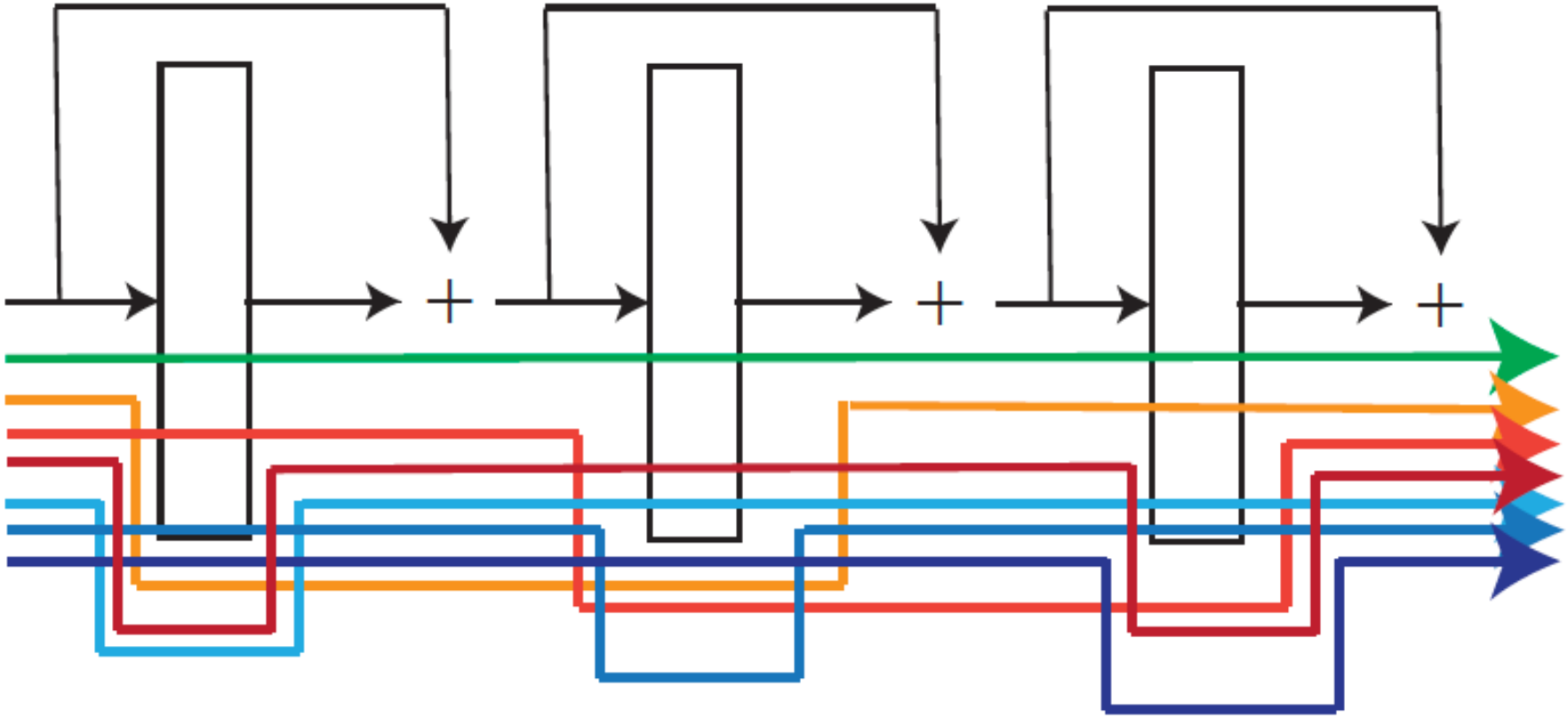
Now write $R_w(\cdot;\theta_w) = V_w + L_w(\cdot;\theta_w)$. Imagine stacking three such layers to get

$$\begin{aligned} B_4 &= R_3(B_3;\theta_3) \\ B_3 &= R_2(B_2;\theta_2) \\ B_2 &= R_1(B_1;\theta_1) \\ B_1 &= \mathcal{I}. \end{aligned}$$

In the simplest case, where V_w happens to be the identity, this is

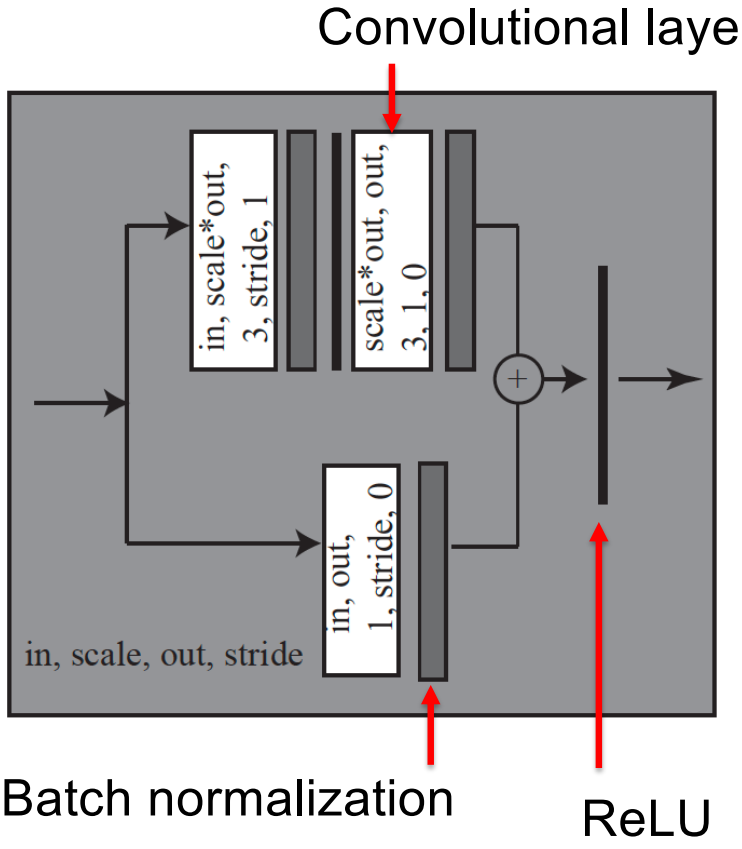
$$\begin{aligned} B_4 &= B_3 + L_3(B_3;\theta_3) \\ B_3 &= B_2 + L_2(B_2;\theta_2) \\ B_2 &= B_1 + L_1(B_1;\theta_1) \\ B_1 &= \mathcal{I} \end{aligned}$$

Residual connections: Simple example



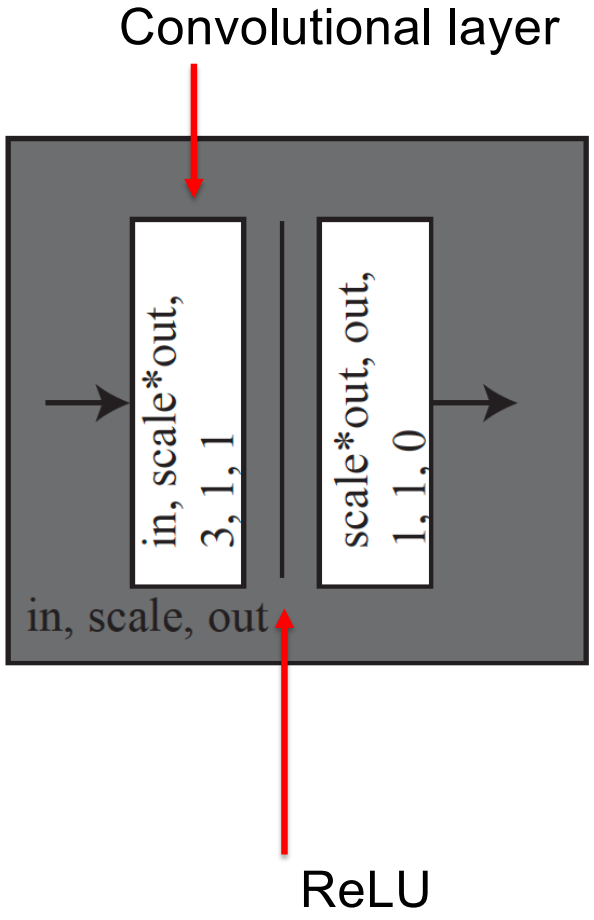
Residual blocks for encoder

in, scale, out, stride are parameters

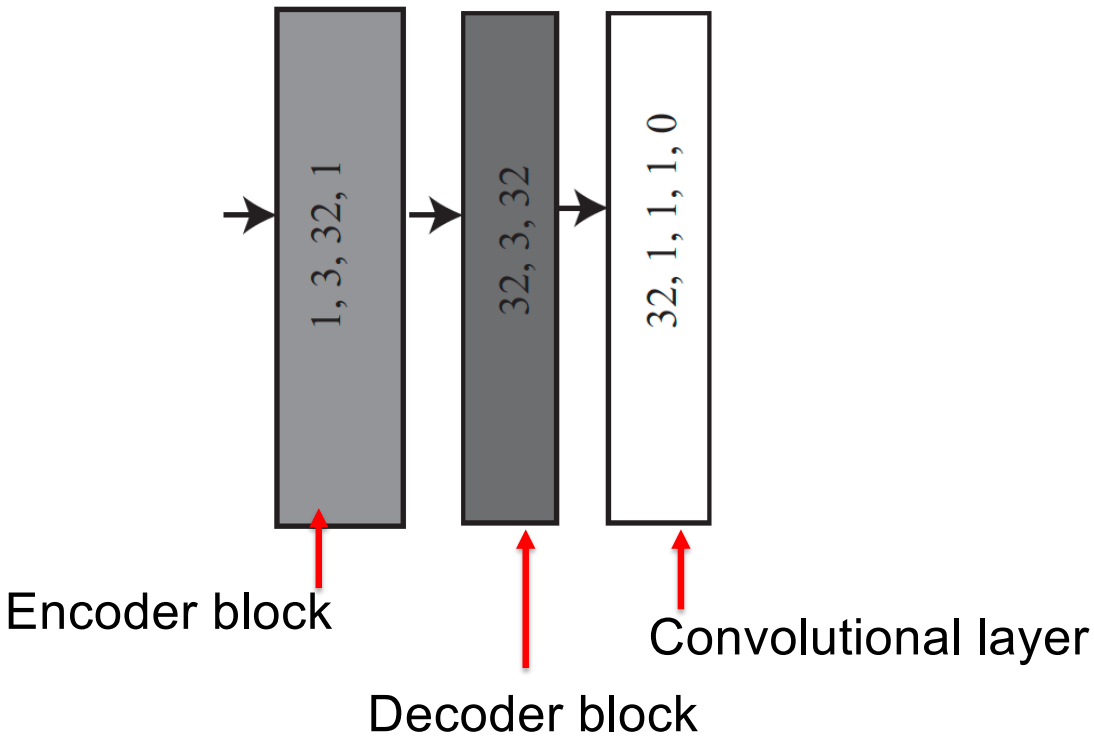


Residual blocks for decoder

in, scale, out are parameters

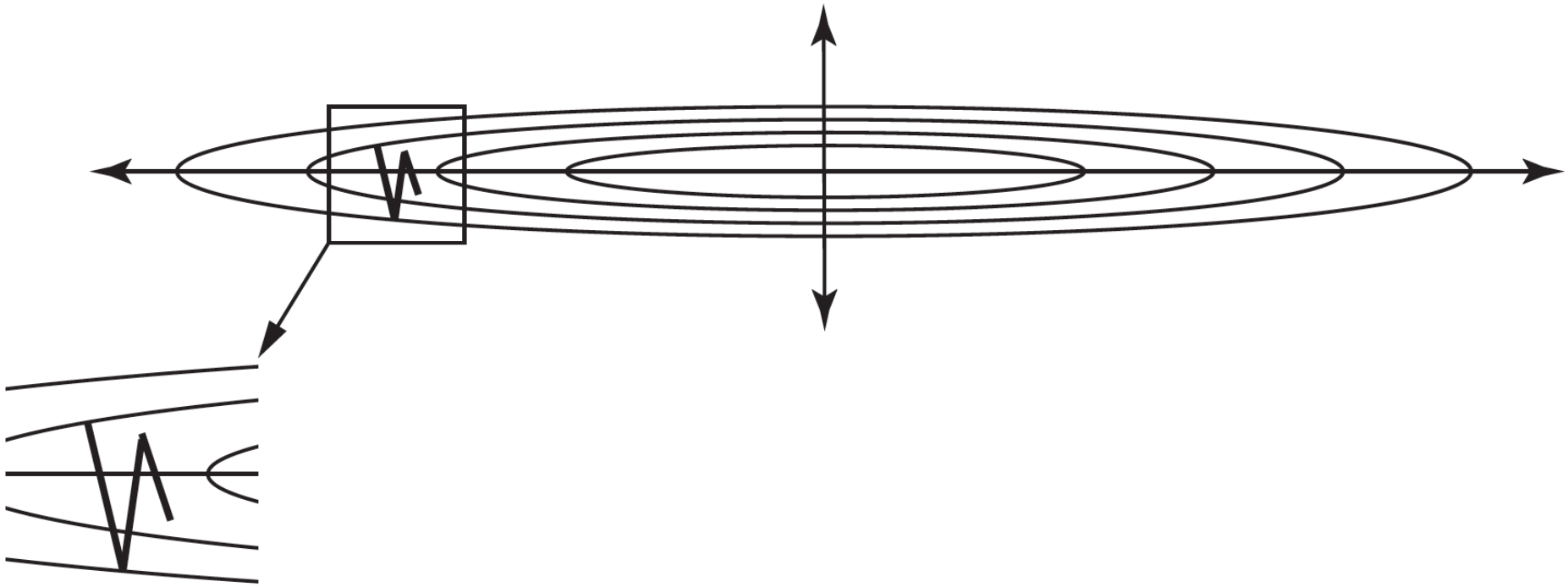


Composing these blocks...



Optimization

Actually, gradient descent is a bad idea....



Gradient is bad, algebra

Here is an example in algebra. Consider $f(x, y) = (1/2)(\epsilon x^2 + y^2)$, where ϵ is a small positive number. The gradient at (x, y) is $(\epsilon x, y)$. For simplicity, use a fixed learning rate η, so

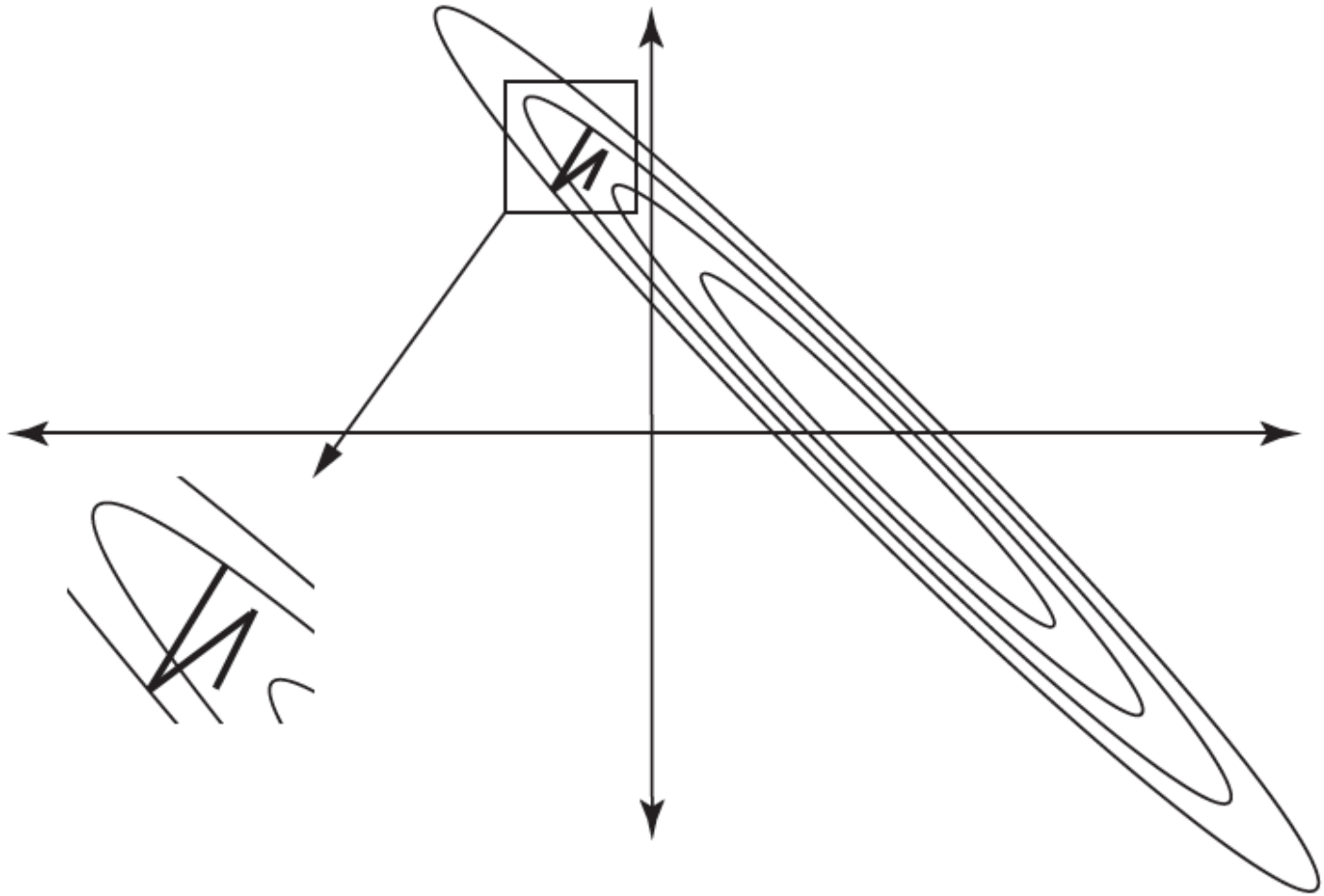
$$\begin{bmatrix} x^{(r)} \\ y^{(r)} \end{bmatrix} = \begin{bmatrix} (1 - \epsilon\eta)x^{(r-1)} \\ (1 - \eta)y^{(r-1)} \end{bmatrix}.$$

Start at, say, $(x^{(0)}, y^{(0)})$ and repeatedly go downhill along the gradient; you will travel very slowly to your destination. You can show that

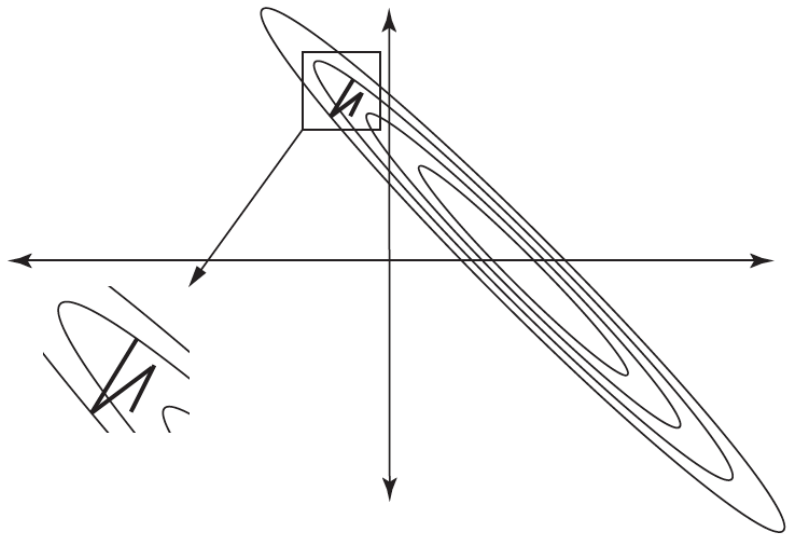
$$\begin{bmatrix} x^{(r)} \\ y^{(r)} \end{bmatrix} = \begin{bmatrix} (1 - \epsilon\eta)^r x^{(0)} \\ (1 - \eta)^r y^{(0)} \end{bmatrix}.$$

The problem is that the gradient in y is quite large (so y must change quickly) and the gradient in x is small (so x changes slowly). In turn, for steps in y to converge requires $|1 - \eta| < 1$; but for steps in x to converge requires only the much weaker constraint $|1 - \epsilon\eta| < 1$. Choose the largest η you dare for the y constraint. The y value will very quickly have small magnitude, though its sign will change with each step. But the x steps will move closer to the right spot only extremely slowly.

Rotating and translating function doesn't help



changes in sign of variables, gradients are a sign of trouble.



Momentum

Parameters need to be discouraged from "zig-zagging" as in the example above. In these examples, the problem is caused by components of the gradient changing sign from step to step. It is natural to try and smooth the gradient. Momentum forms a moving average of the gradient. Construct a vector $\mathbf{v}$, the same size as the gradient, and initialize this to zero. Choose a positive number $\mu < 1$. Then iterate

$$\begin{aligned} \mathbf{v}^{(r+1)} &= \mu \mathbf{v}^{(r)} + \eta \nabla_\theta E \\ \theta^{(r+1)} &= \theta^{(r)} - \mathbf{v}^{(r+1)} \end{aligned}$$

Notice that, in this case, the update is an average of all past gradients, each weighted by a power of μ. If μ is small, then only relatively recent gradients will participate in the average, and there will be less smoothing. Larger μ lead to more smoothing. A typical value is $\mu = 0.9$. It is reasonable to make the learning rate go down with epoch when you use momentum, but keep in mind that a very large μ will mean you need to take several steps before the effect of a change in learning rate shows. Correctly implementing weight decay requires care when momentum is present (**exercises**).

A variety of other optimization tricks available

RMSProp

AdaGrad

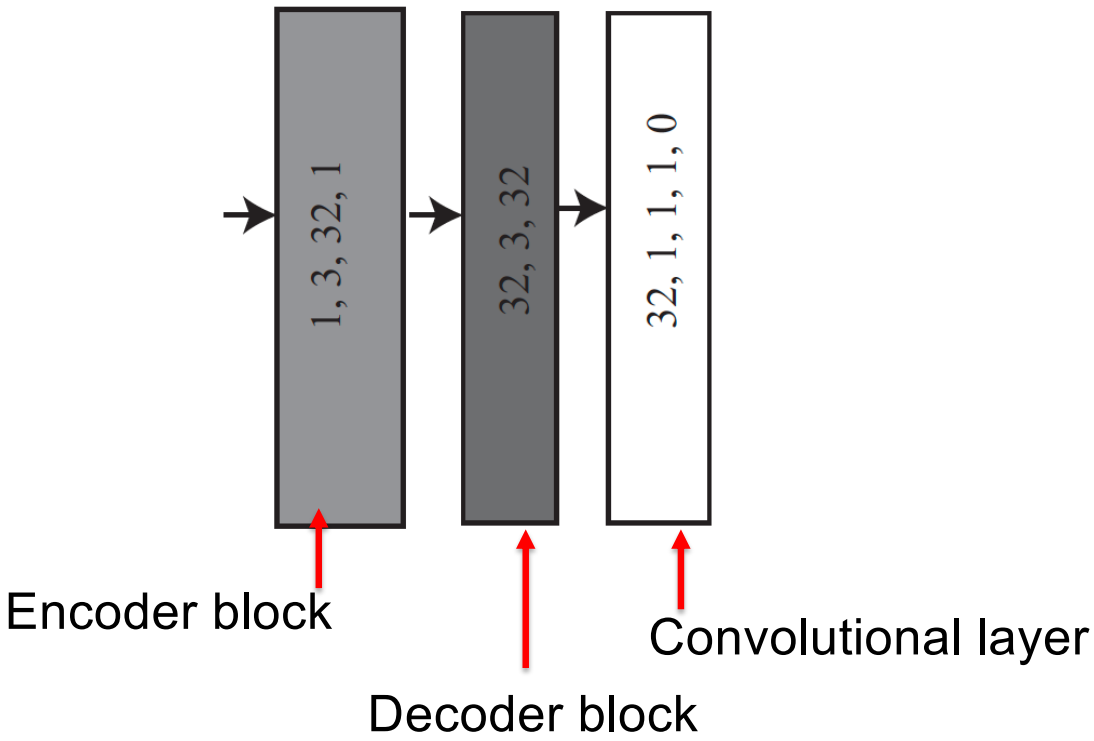
Adam

Adam is favored.

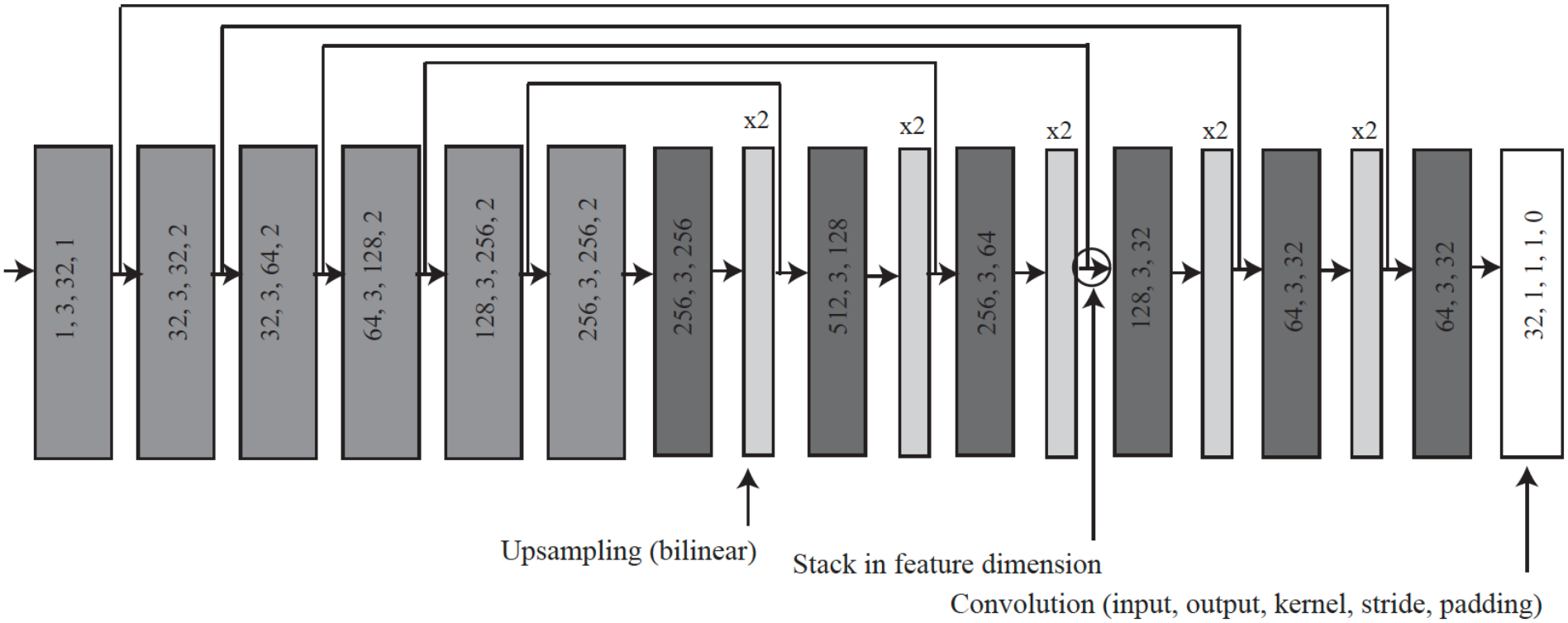
Experience:

- Adam is best for fast descent and quite good models
- SGD for robust models, but slow
  - (Adam for papers, SGD for production models)

Shallow autoencoder



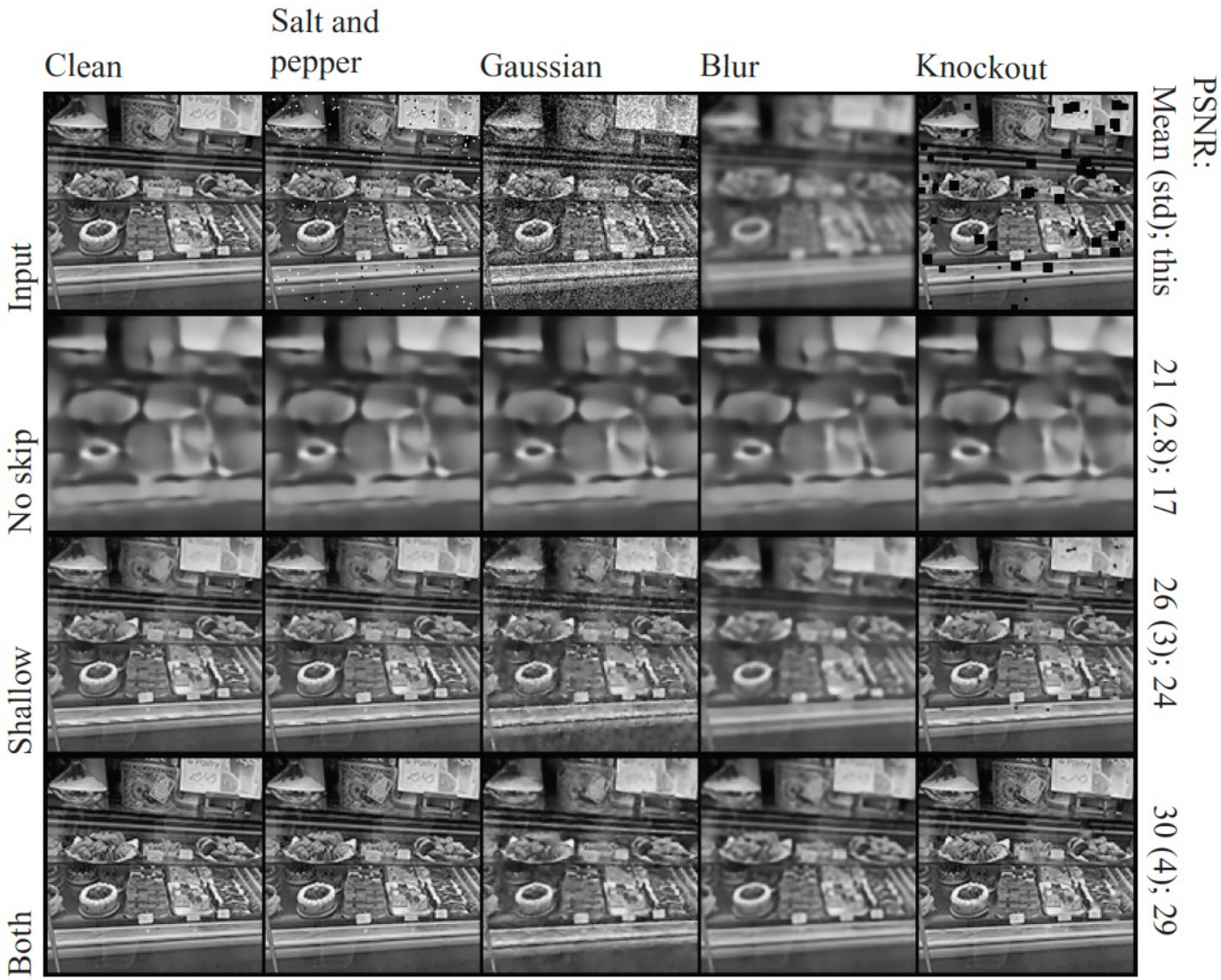
Deeper autoencoder



Example

Notice:

- skip helps, a lot
- deep helps a little



Detail windows

Notice:

- skip helps, a lot
- deep helps a little