Lecture 6: More Neural Networks

Matthew Caldwell

COMP0088 Introduction to Machine Learning • UCL Computer Science

Contents

6.1 Inductive Bias

6.2 Convolutional Networks

6.3 Recurrent Networks

6.4 Practicalities

6.1: Inductive Bias

COMP0088 Introduction to Machine Learning · UCL Computer Science

- “I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.” — Abraham Maslow
- Model assumptions determine what solutions will be found
- Knowledge of problem structure can be used to align model assumptions with our expectations to improve learning
- Enforcement of structural priors biases the learning towards solutions we know are good, or at least to taking account of things we think are important
- At a cost of reduced generality, constrained results, inability to “think outside the box”

- Inductive biases are natural
  - → our brains are chock full of specialised equipment tailored to particular problems
    - ▸ indeed, CNNs are somewhat inspired by specialised neural structures in visual cortex
  - → even our “general purpose” approaches are saturated in knowledge about the world
- And to some extent inevitable
  - → No Free Lunch proves there is no one size fits all from-scratch learning approach, so we always have to tailor our approach to the task to some extent

- Fully connected neural networks are pretty general (as feedforward, fixed-problem models go), but that can also mean they are inefficient and prone to overfitting
- In week 1 we picked out two specific types of aggregate data with special structure: images (or digital signals) and sequences
  - → the former are always the latter, and the latter can often be posed as the former
  - → and indeed it is possible to use techniques for one with the other
- In both cases we can exploit the structure to get better performance
- Inductive biases are biases — we can view this as another bias-variance tradeoff — but can often be demonstrably beneficial

6.2: Convolutional Networks

COMP0088 Introduction to Machine Learning • UCL Computer Science

Think local

- Images* have local structure
- Nearby pixels are likely to be similar (or at least related), distant pixels may be quite different (and unrelated)
- This is a consequence of the image formation process: the pixels are sampled from a scene containing coherent physical objects
- Relatedness need not be smooth (objects can be textured) but is systematic
- Discontinuities — edges and corners — have a physical meaning that makes them “interesting”

* And digital signals generally. We’ll focus on images in this section, but much of the discussion applies to other signals too — eg, audio has local structure because it arises from vibrations of material objects that have to move continuously through space — they can’t teleport so they can’t generate perfect step responses

Convolution

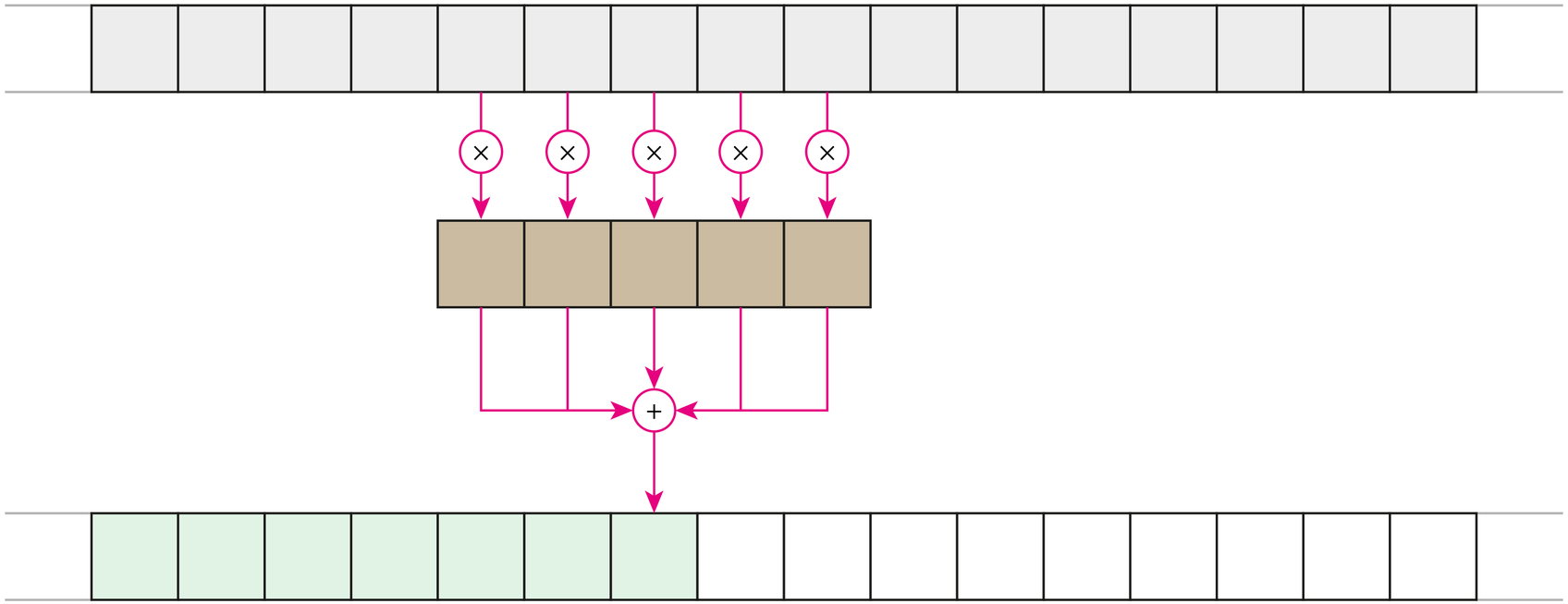
- Convolutions for continuous signals are a kind of integral transform

$$(f \otimes g)(t) = \int_{-\infty}^{\infty} f(t)g(t-s)\,\mathrm{d}s$$

- If you have a physics or engineering background you will probably have seen these in the context of Fourier analysis etc
- They occur when some spatial or temporal effect (eg reverberation or optical blurring) applies everywhere over another signal — the blur or echo at each point smears into the blur or echo from every other point
- Convolution Theorem: convolution in space/time domain is equivalent to pointwise multiplication in frequency domain, and vice versa

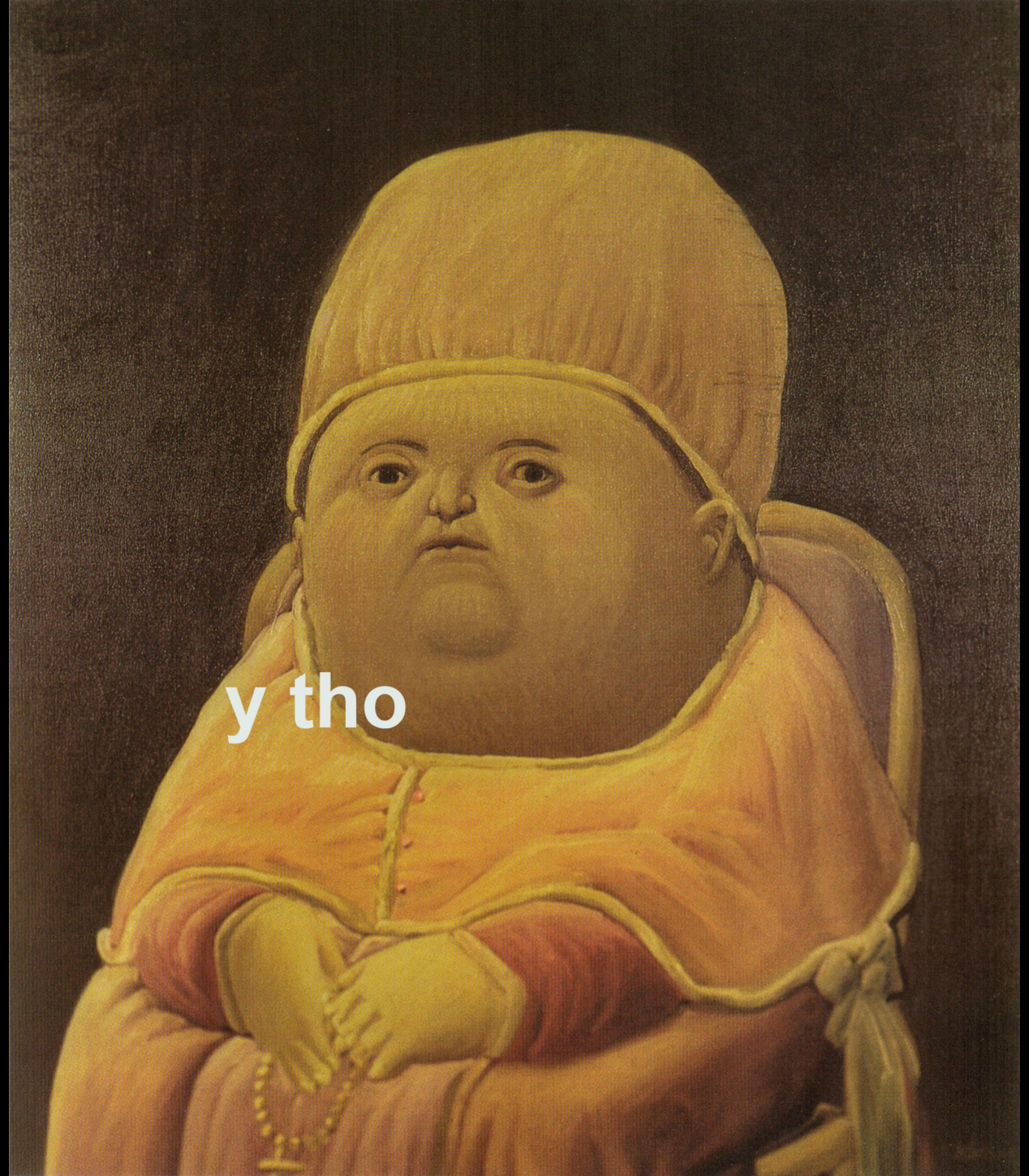
Convolution

- For discrete signals this becomes essentially a sliding window dot product
- A local signal fragment is dotted with the input signal at every offset



Convolution

- For discrete signals this becomes essentially a sliding window dot product
- A local signal fragment is dotted with the input signal at every offset
  - → NB: for images the fragment is a matrix or tensor, but the operation at each location is still a single weighted sum not a matrix or tensor product
- The fragment is known as a filter or kernel
  - → Unhelpful terminology alert: not to be confused with kernel functions used in SVMs
  - → Mathematically they are related, but for practical ML purposes they are not the same thing at all: convolution kernels participate in the inner product, they don't bypass it


y tho

Filters

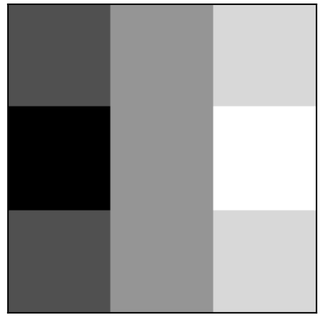
- The filter or kernel responds to patterns in the signal
  - → Recall that the inner product can be thought of as a similarity metric
  - → Output will be high when peaks in the filter align with peaks in the signal
  - → Output will be low when troughs in the filter align with peaks in the signal
- So filters can be thought of as detectors that find features in the image
- Because each filter is applied at every location, it should detect its particular feature wherever it occurs in the image
- This translational equivariance is one of our structural priors
  - → A cat detector should spot a cat anywhere in the image, not just in (eg) the top left

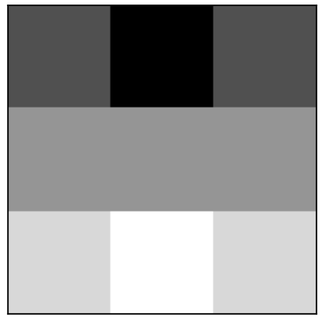
Engineered filters

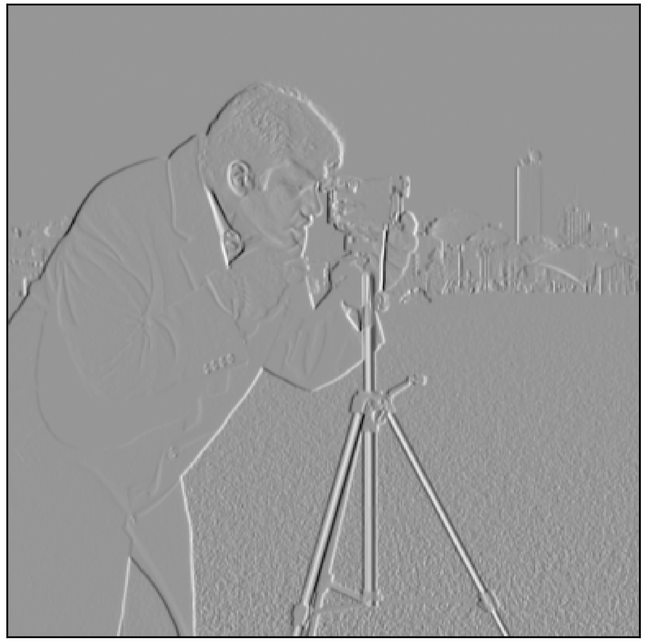
- Convolution kernels as feature detectors are nothing new and designed convolution filters have been used in digital signal processing for many years
  - → Edge detectors (eg Sobel)
  - → Gabor filters for texture detection
  - → Template matching

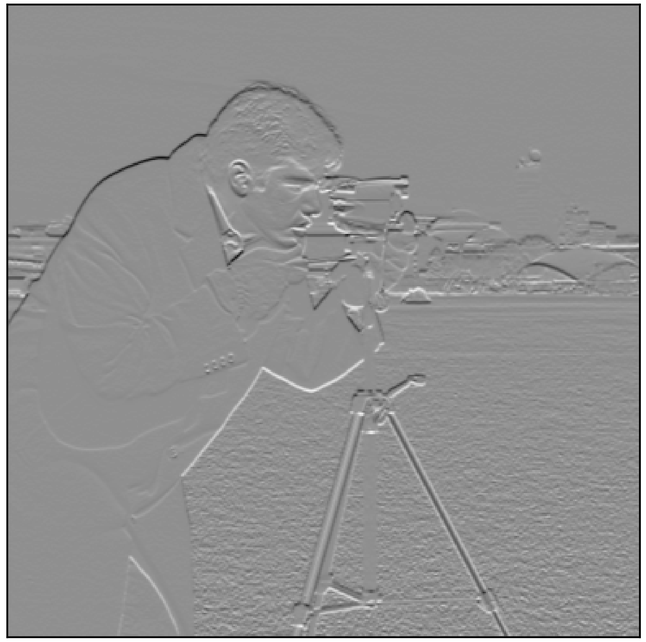
- Convolutions are one of the main workhorses of digital image processing, used for blurring, sharpening, texturing, emphasis
  - → If you've ever used Instagram or Photoshop, you've used engineered convolution filters

Sobel filters

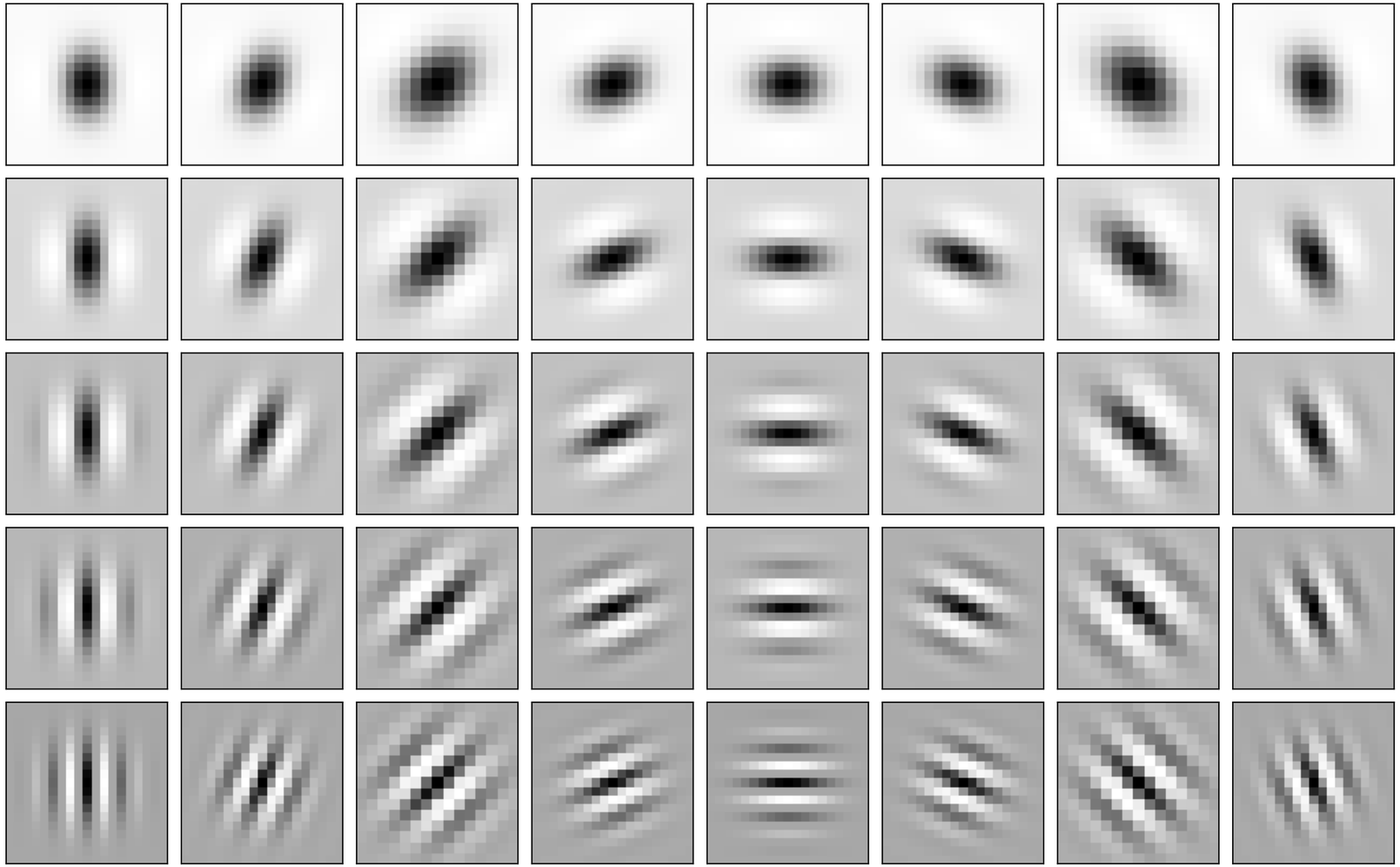




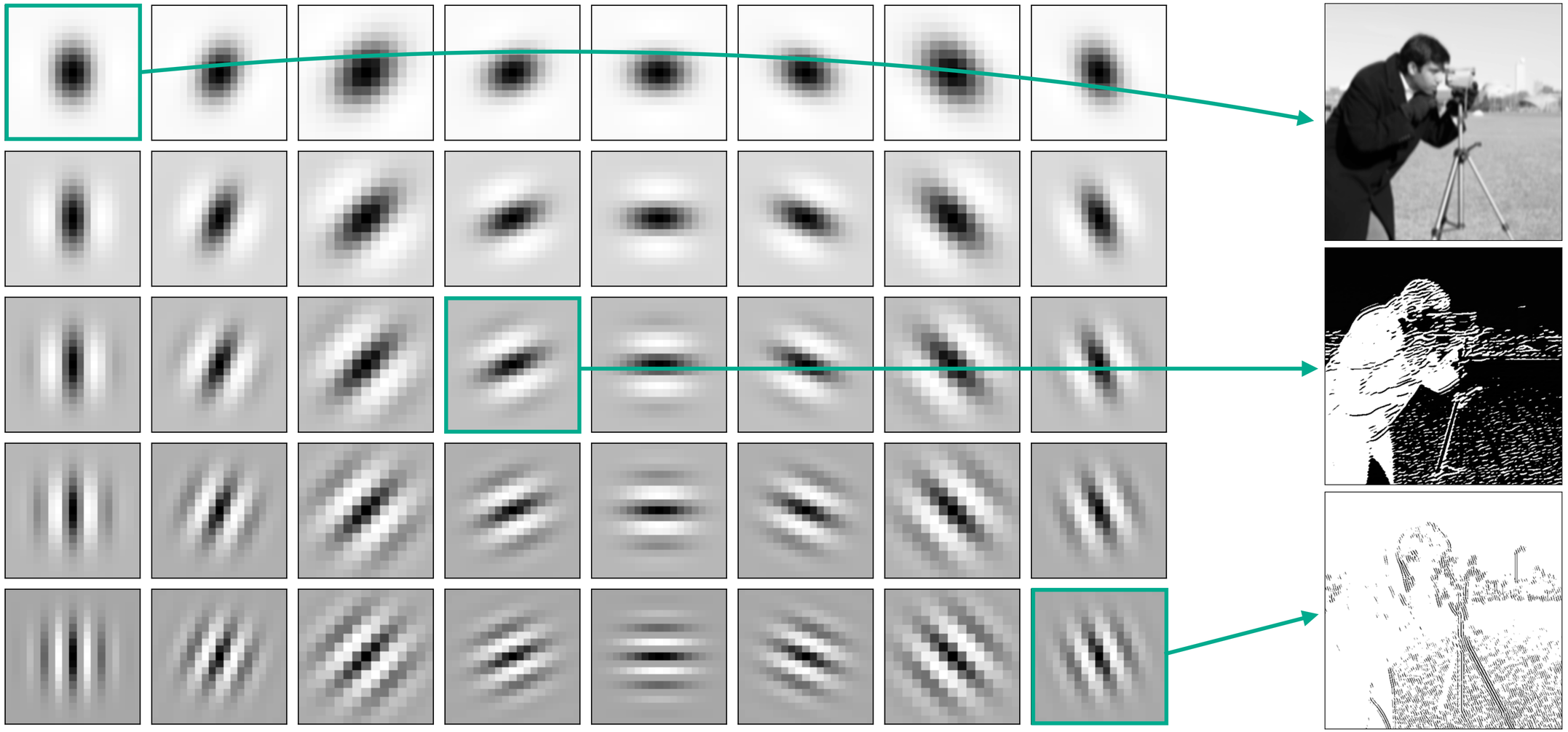




Gabor filters



Gabor filters

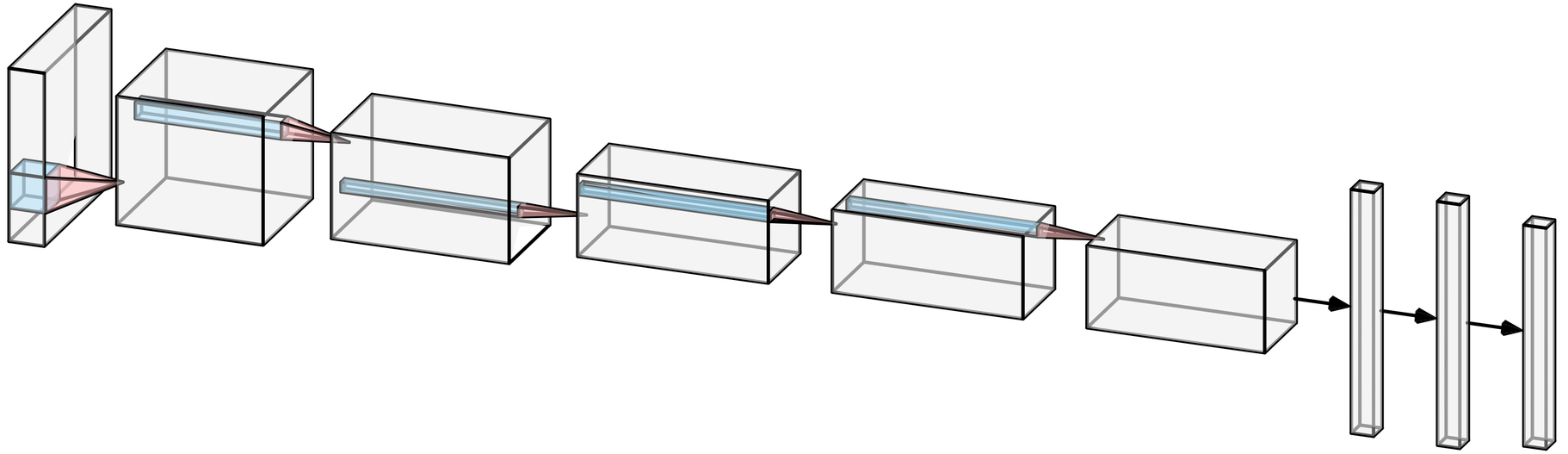


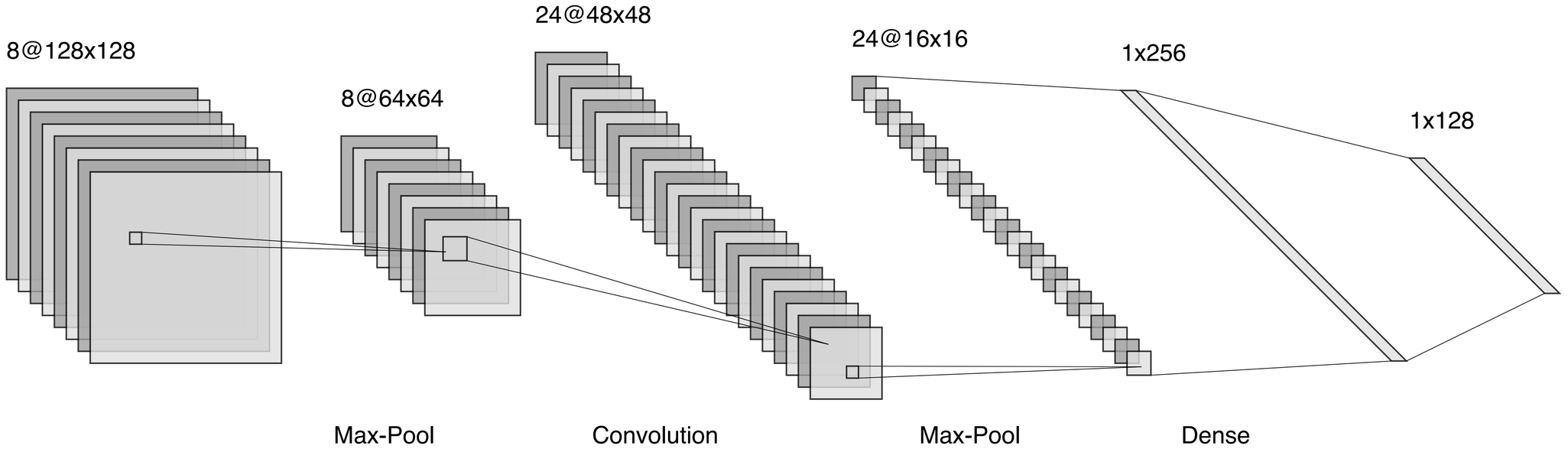
Convolutions in neural networks

- We don’t want to manually design our filters, but learn them
- Which means embedding them in a network graph such that we can calculate the filter weight gradients by backpropagation and update them using some form of gradient descent
- Convolutions are a bit different from the neural network layers we’ve seen already, but they’re ultimately going to boil down to the same kind of thing:
  - → Calculate weighted sums of the input data
  - → Apply a non-linear activation function to the results
- The difference is mostly in how the data and weights are organised

Convolutions in neural networks

- Images (and convolutional outputs) are semantically planar — they have a spatial extent in 2D, and within that are sampled on a uniform spatial grid
- They also have “depth” — so we are always dealing with a volume
  - → For input images, depth represents colour (usually 3 RGB or 1 greyscale)
  - → Convolutional outputs have one “channel” per filter — often called “feature maps”, these represent the spatial distribution of the filter responses — eg, a “cat” feature map would be high wherever there was a cat
  - → There will usually be many convolutional filters per layer, so many output channels
- Filters typically have a small spatial extent, but always encompass the full depth of the incoming feature map volume (with weights for each)
  - → E.g. a kernel might have 3×3×128 weights (plus 1 bias)



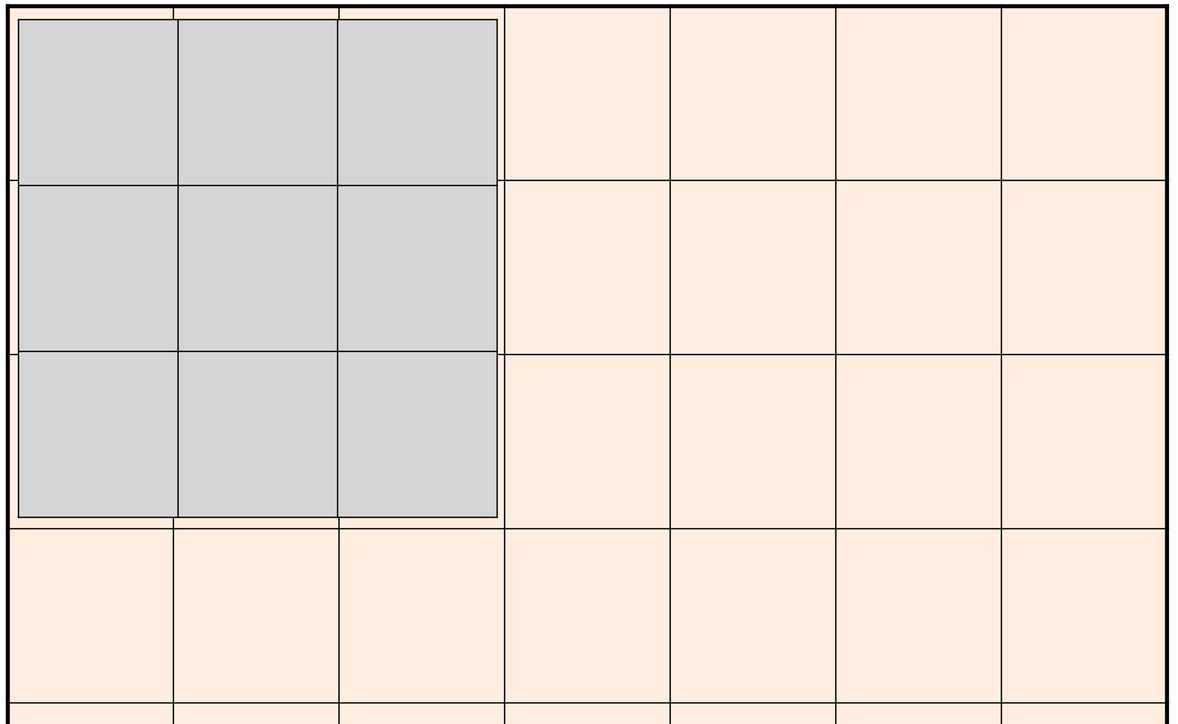

8@128x128
8@64x64
24@48x48
24@16x16
1x256
1x128
Max-Pool
Convolution
Max-Pool
Dense

Weight sharing

- Each filter has only the one set of weights, which it applies at all locations
- So the “output neuron” for each location shares the same weights
- Improves efficiency by having fewer parameters
- Has a regularising effect: encourages filters to be useful everywhere, rather than specialising for a single feature in a single location
  - → In a fully connected network, each neuron can be viewed as applying a convolutional filter, but it’s the size of the whole input, with nowhere to slide
- When backpropagating, gradient updates are summed over all locations

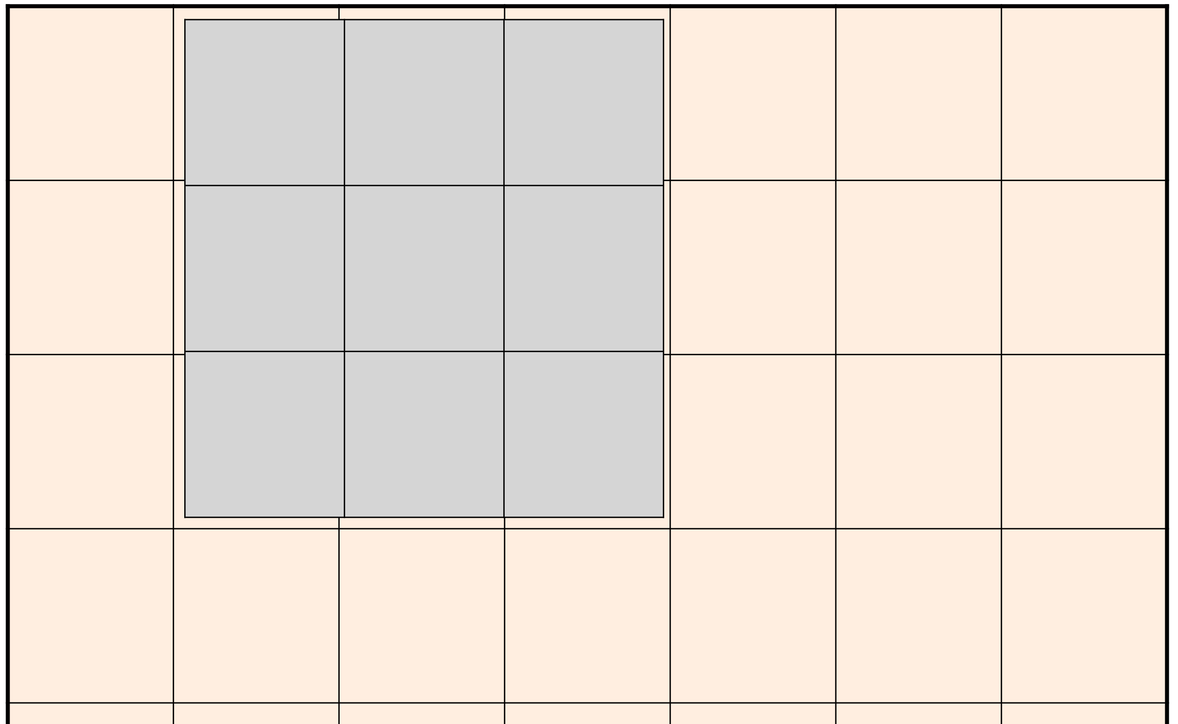
Kernel size vs input size

- There has to be an input value from the image for every weight in the kernel — it has to fit within the input array at all locations — so with a kernel size greater than 1 the output (spatial dimension) would be smaller than the input



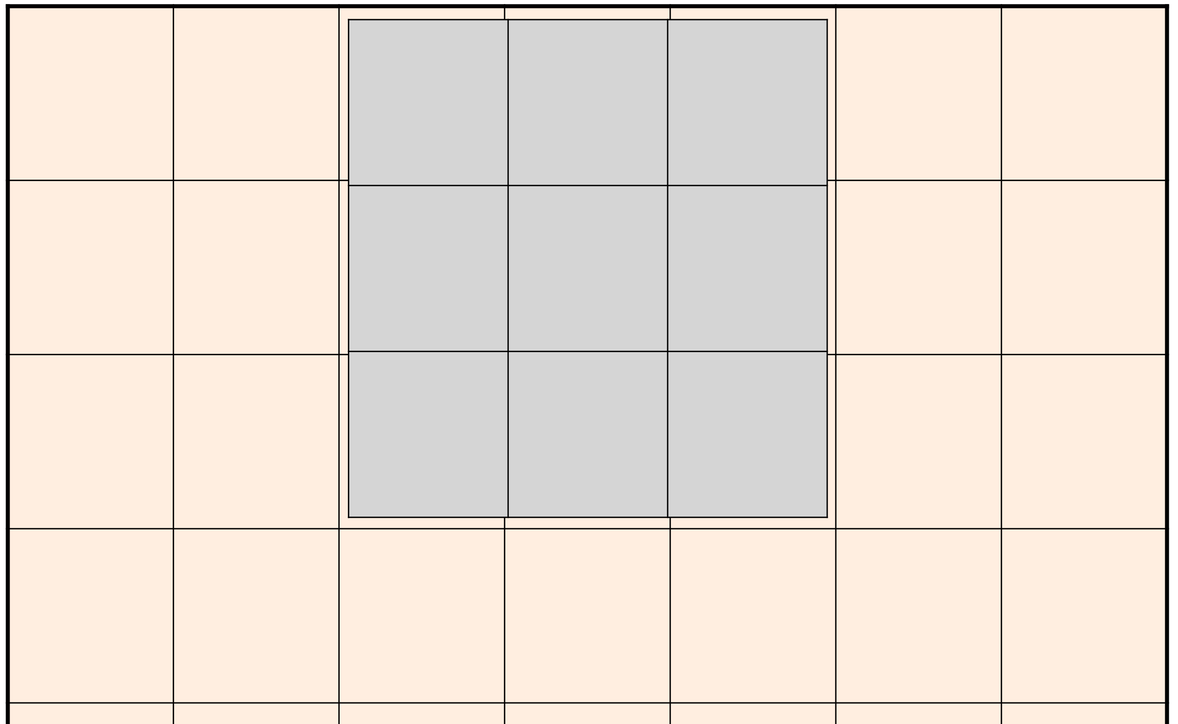
Kernel size vs input size

- There has to be an input value from the image for every weight in the kernel — it has to fit within the input array at all locations — so with a kernel size greater than 1 the output (spatial dimension) would be smaller than the input



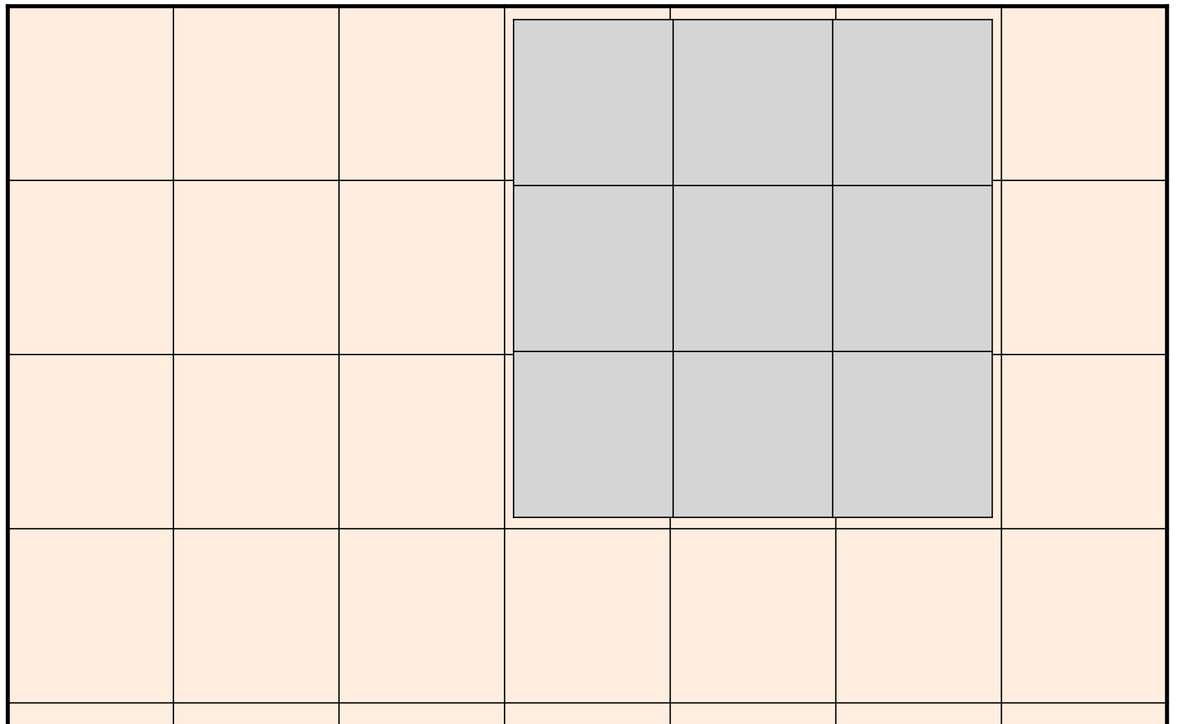
Kernel size vs input size

- There has to be an input value from the image for every weight in the kernel — it has to fit within the input array at all locations — so with a kernel size greater than 1 the output (spatial dimension) would be smaller than the input



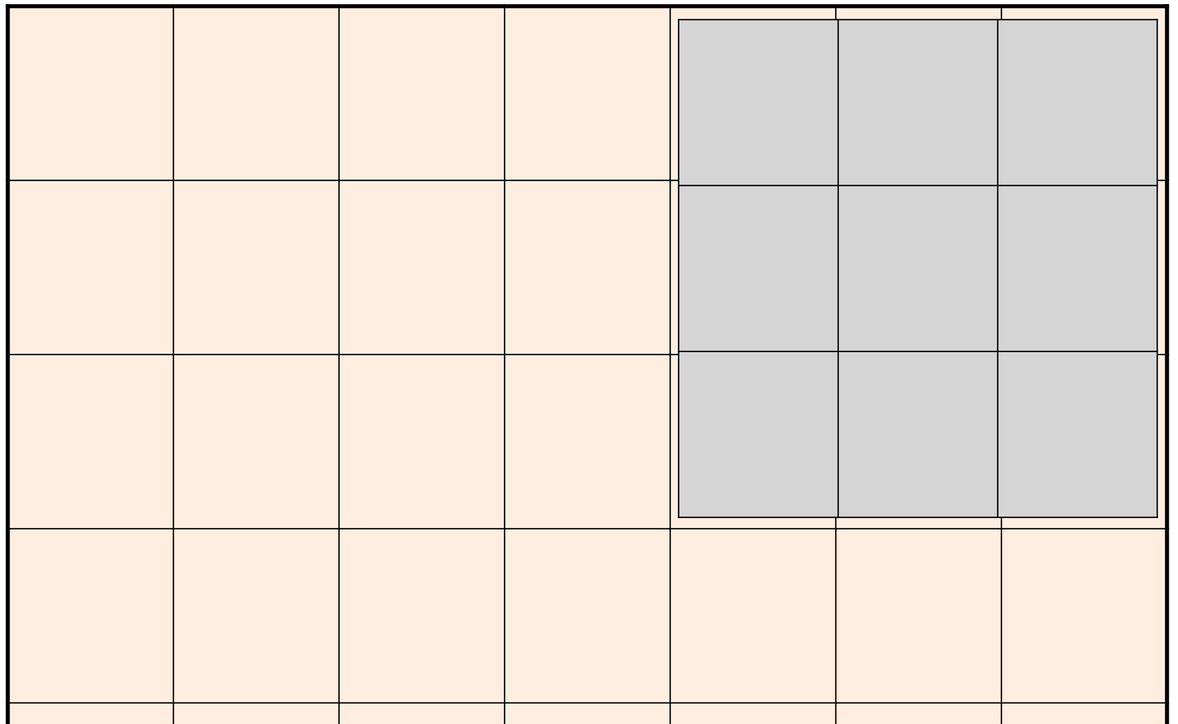
Kernel size vs input size

- There has to be an input value from the image for every weight in the kernel — it has to fit within the input array at all locations — so with a kernel size greater than 1 the output (spatial dimension) would be smaller than the input



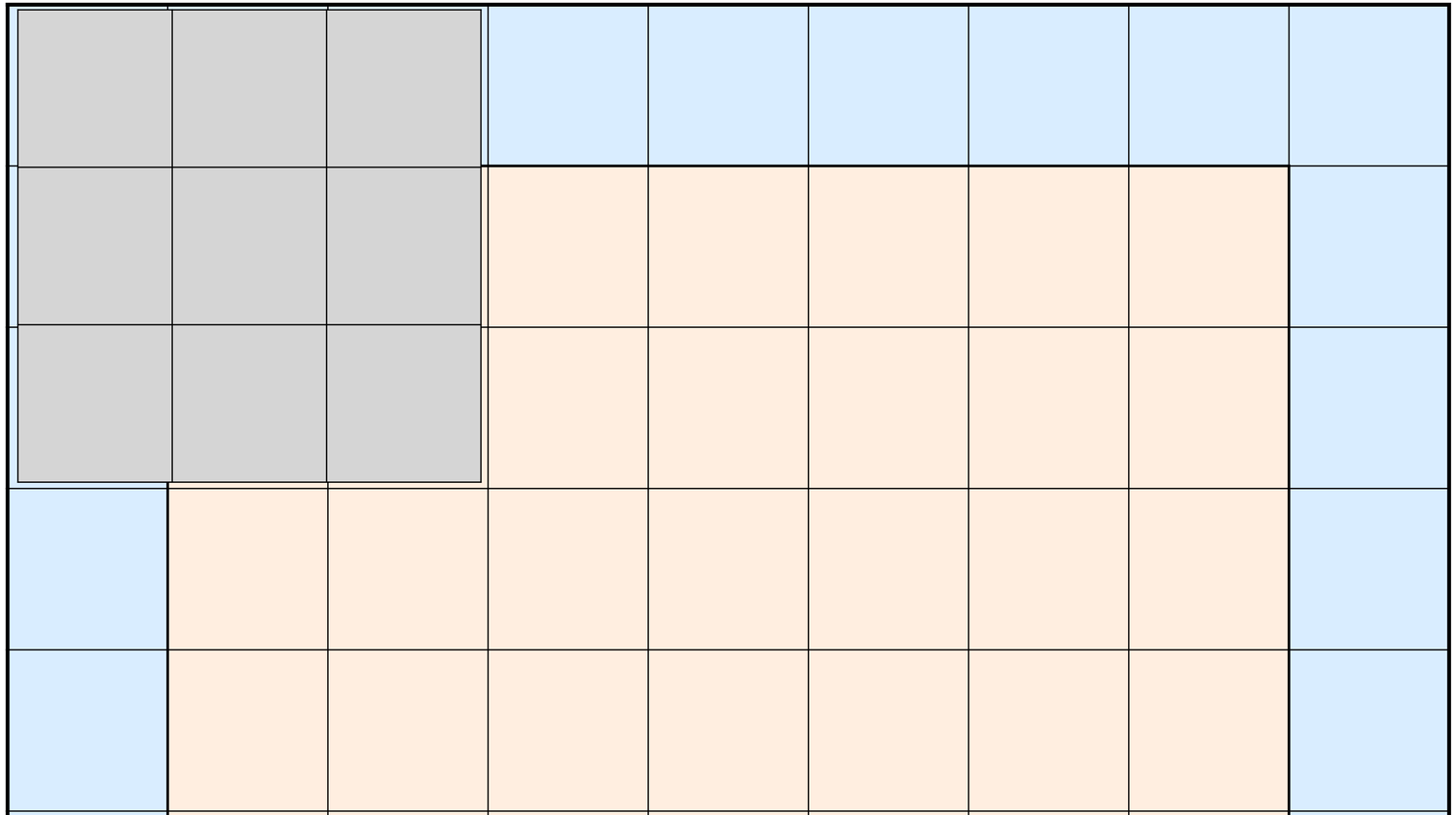
Kernel size vs input size

- There has to be an input value from the image for every weight in the kernel — it has to fit within the input array at all locations — so with a kernel size greater than 1 the output (spatial dimension) would be smaller than the input



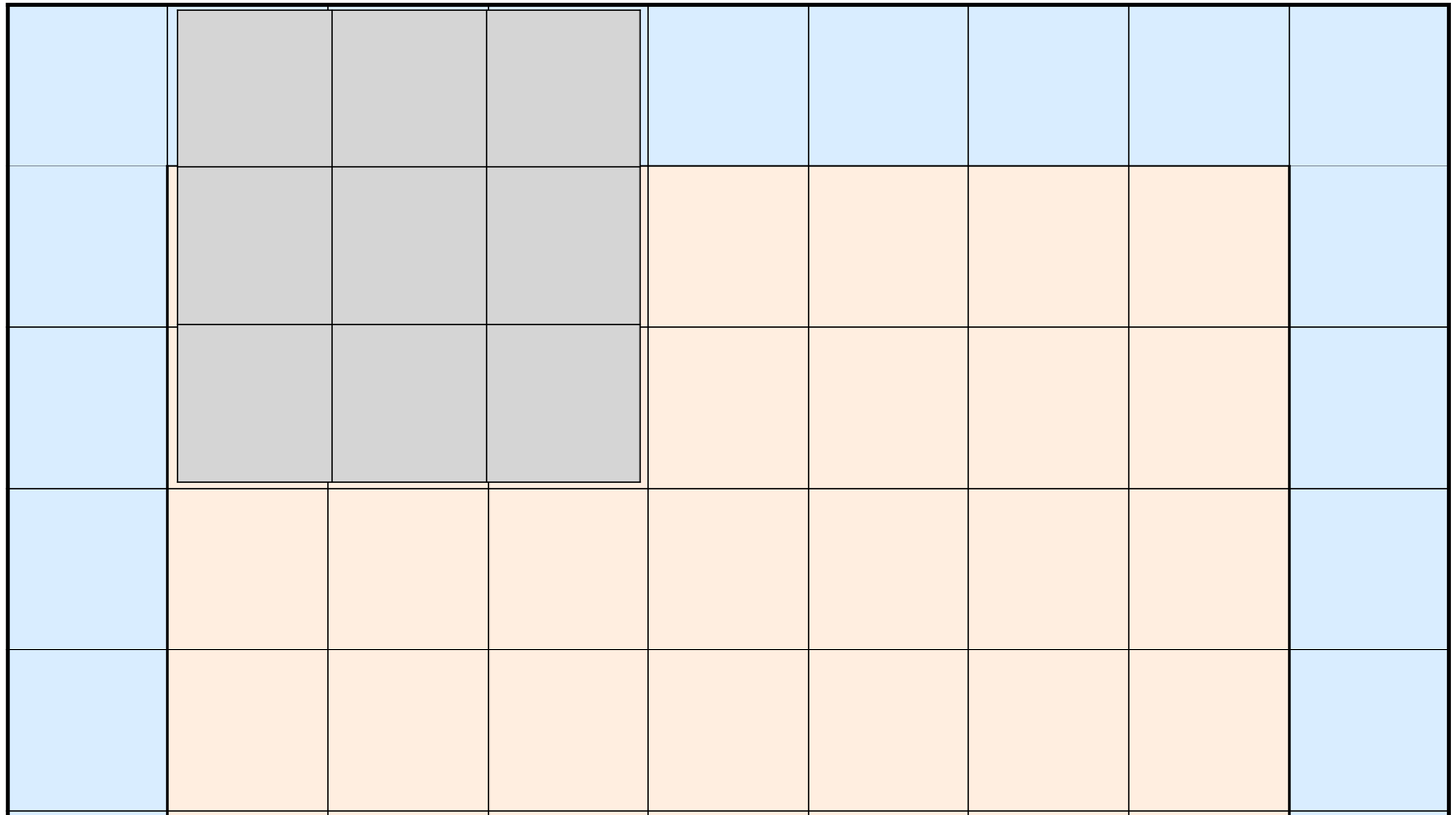
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



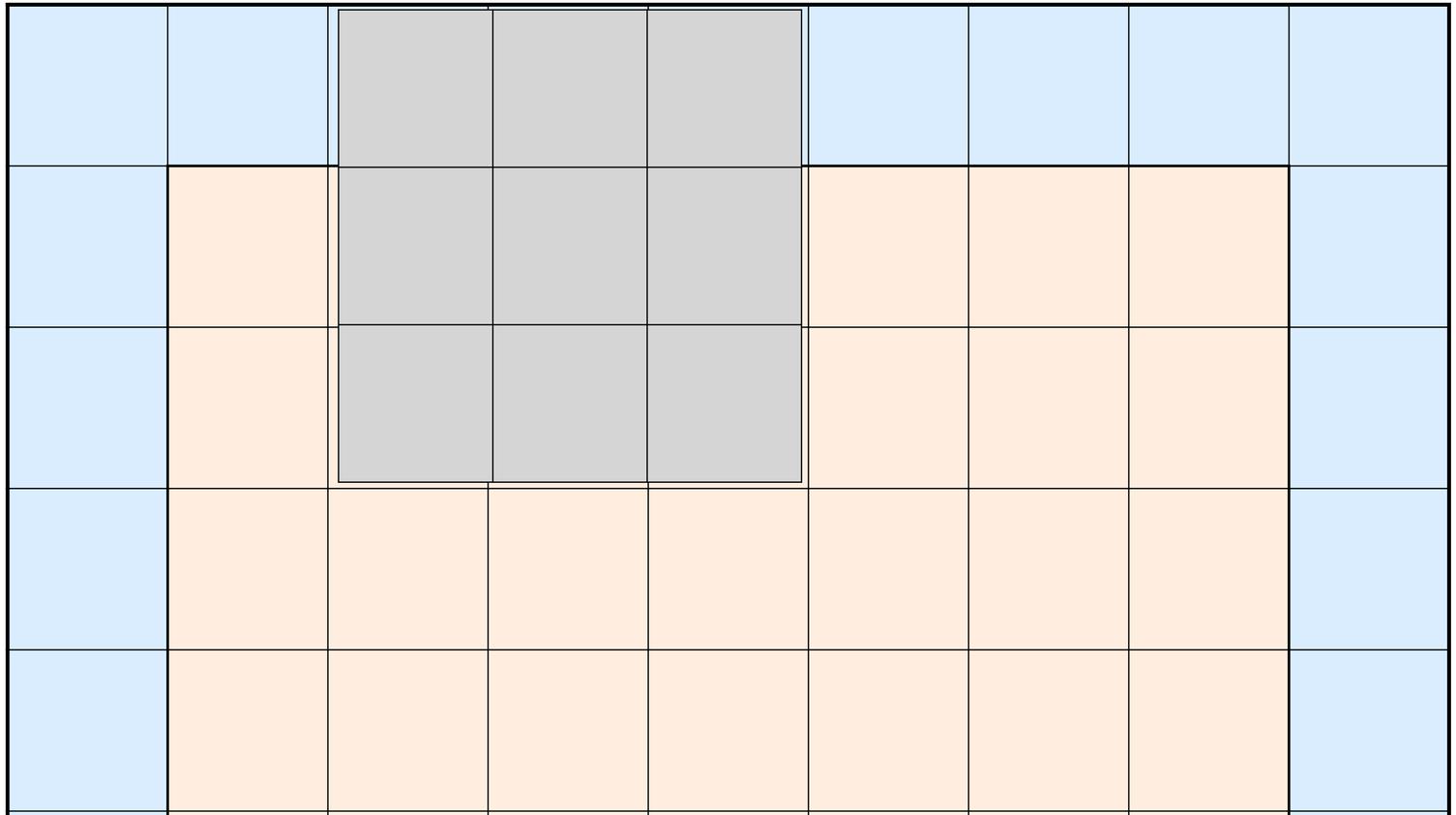
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



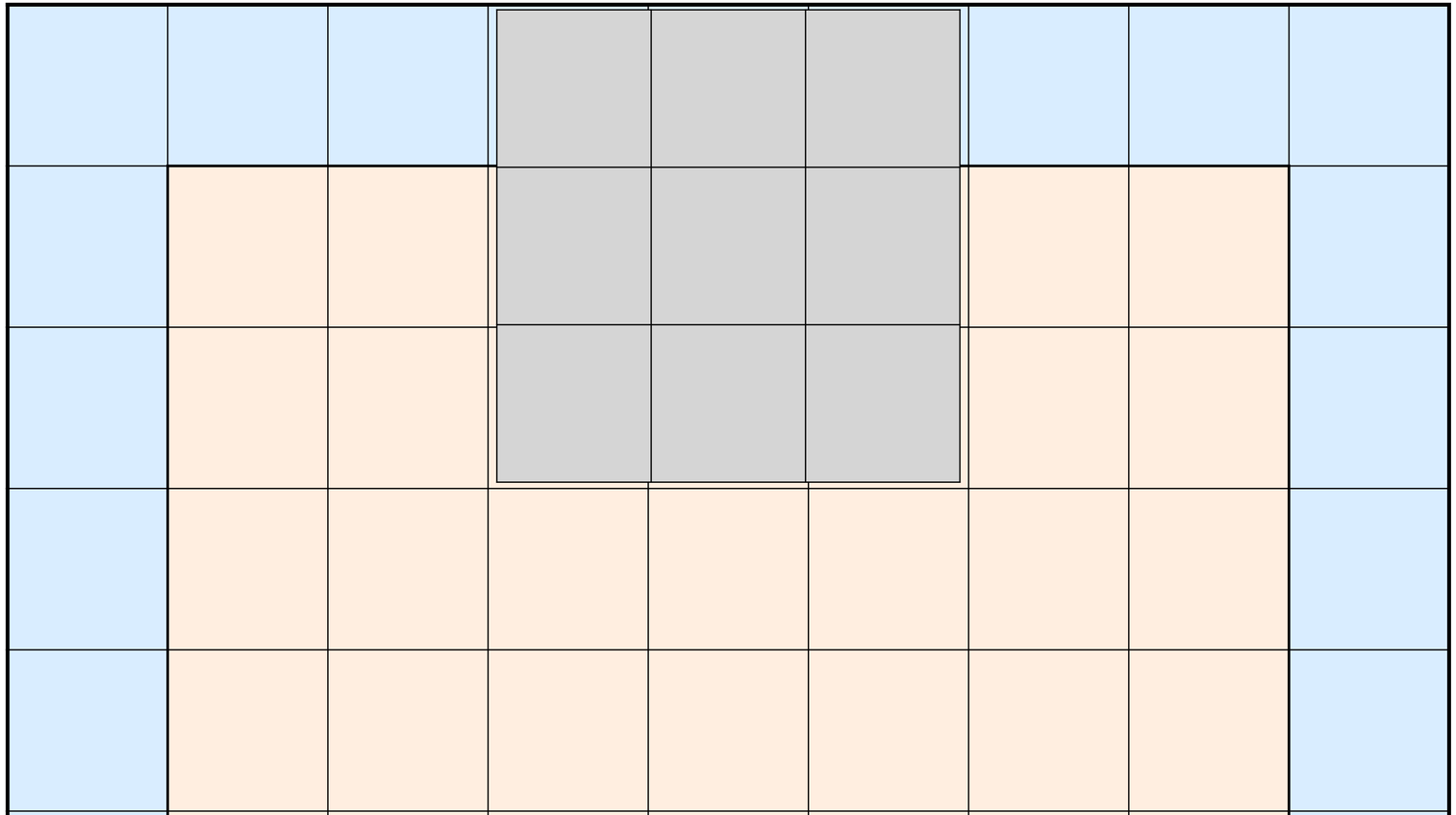
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



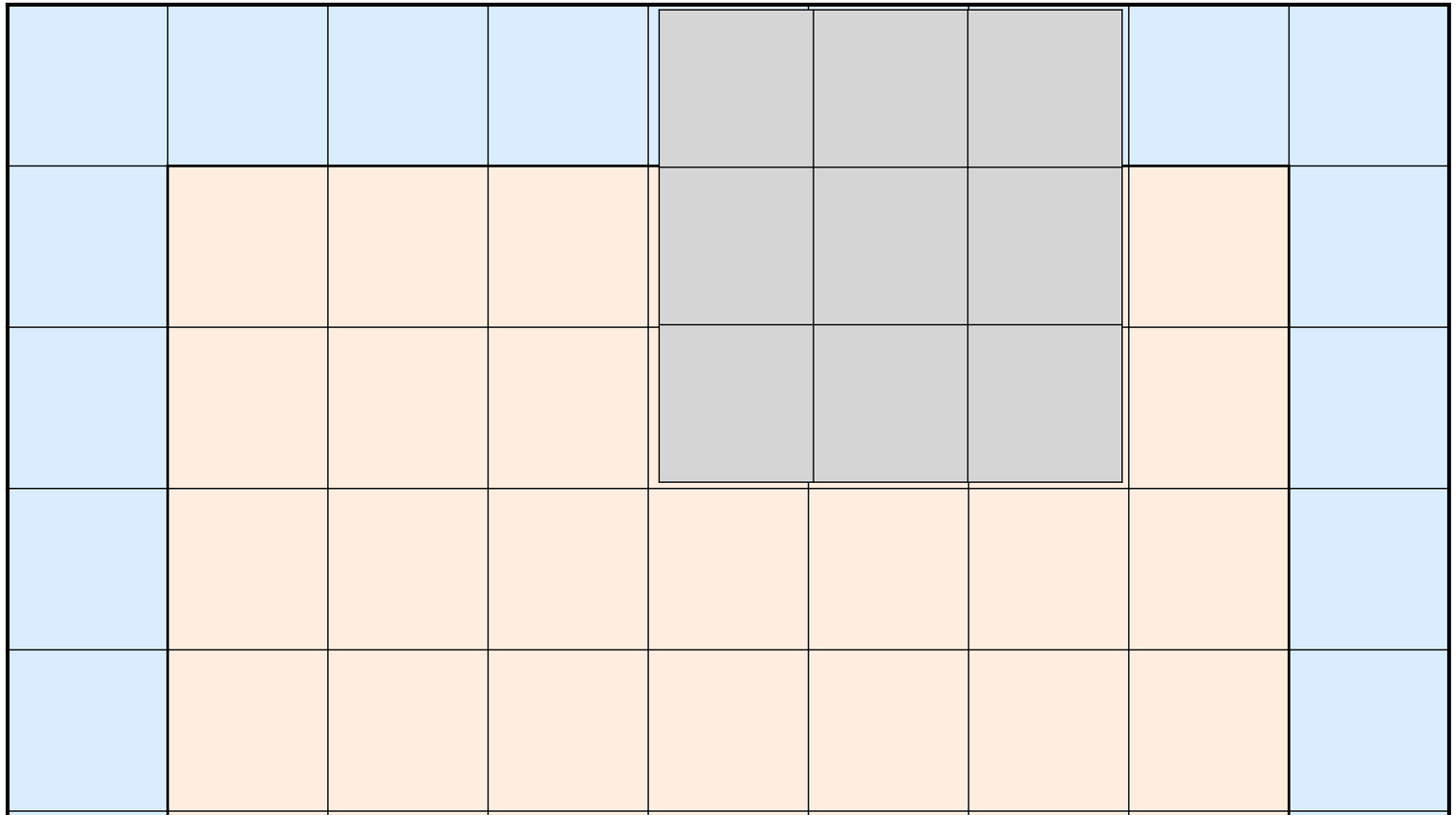
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



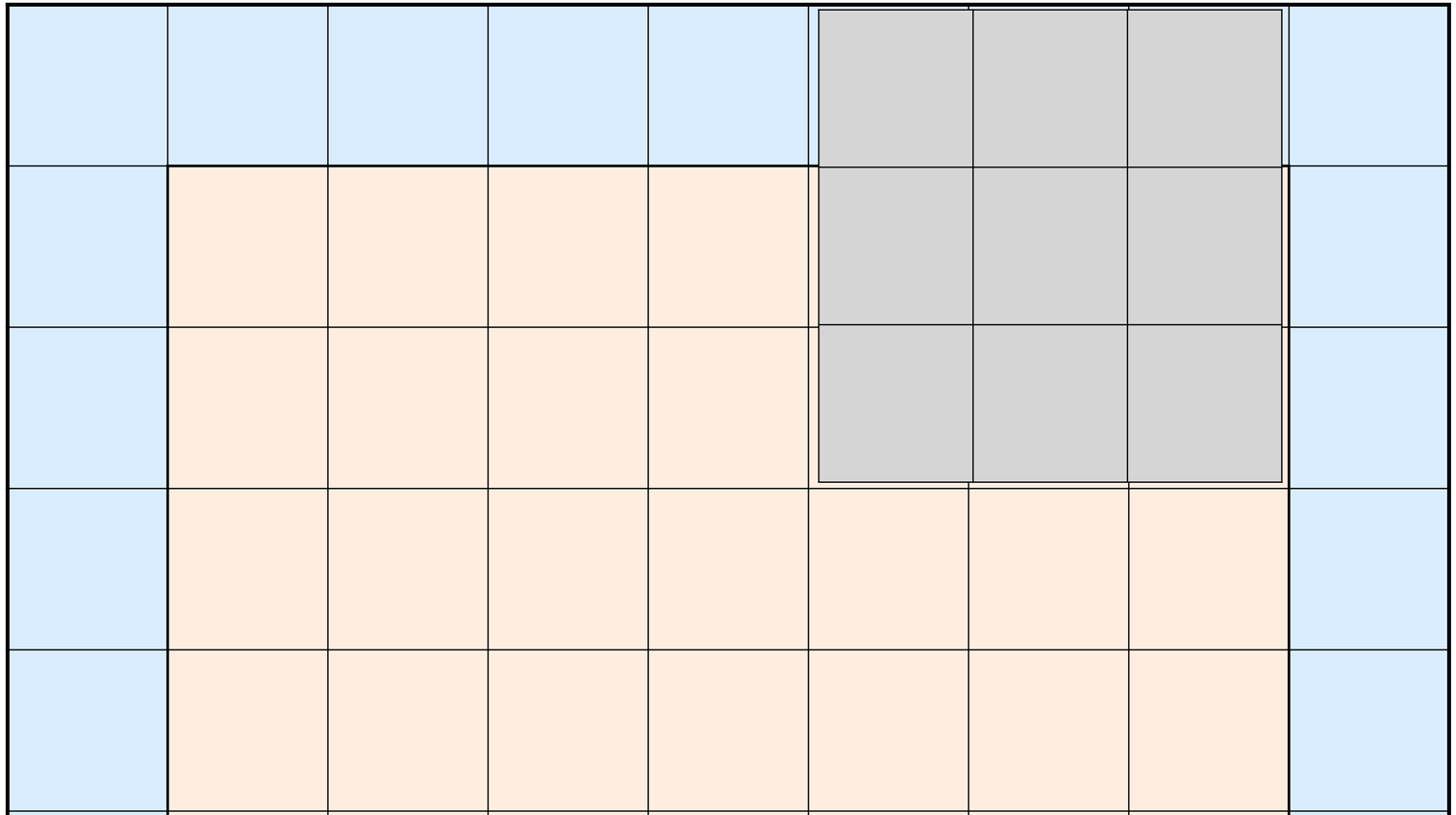
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



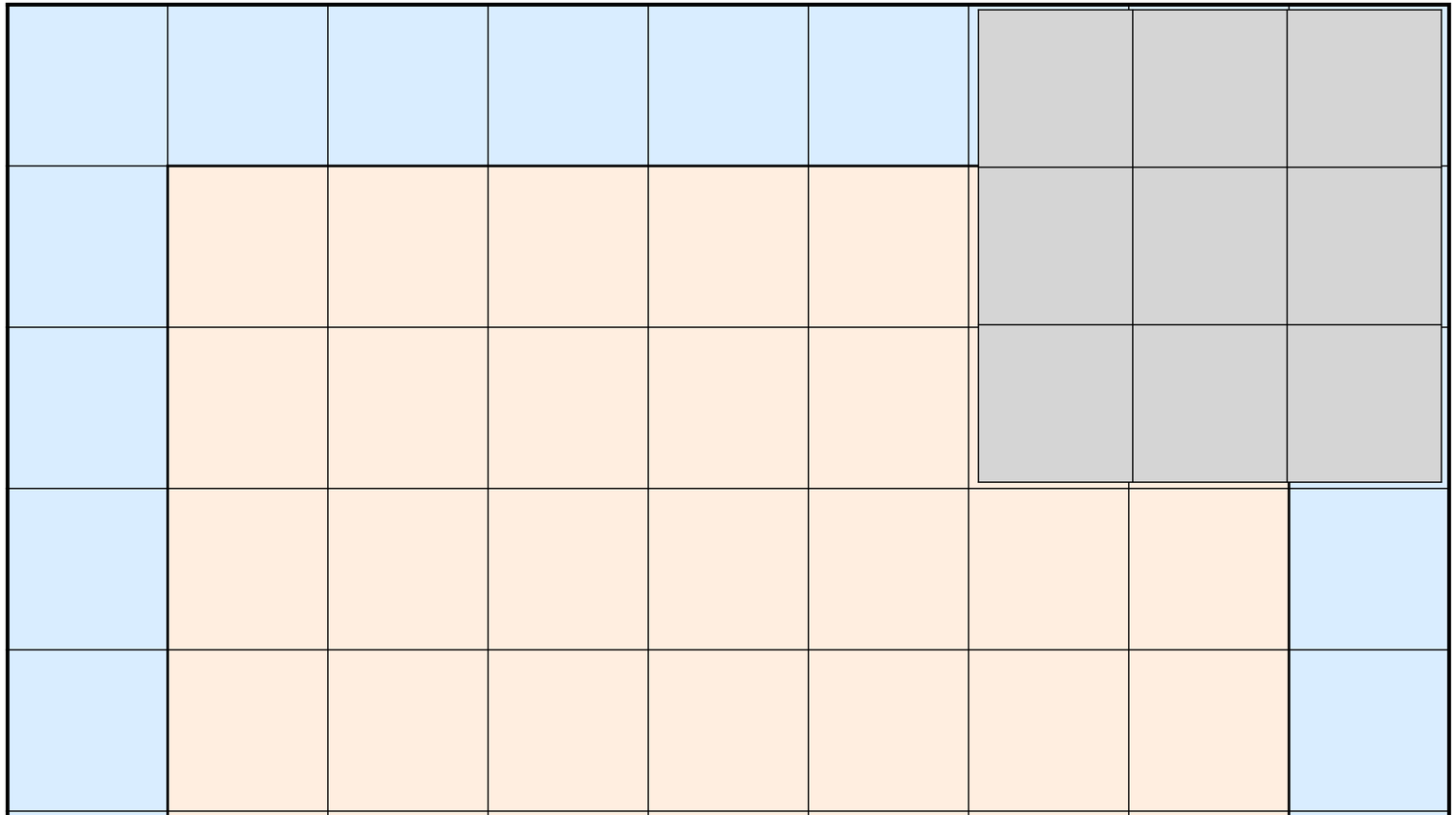
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



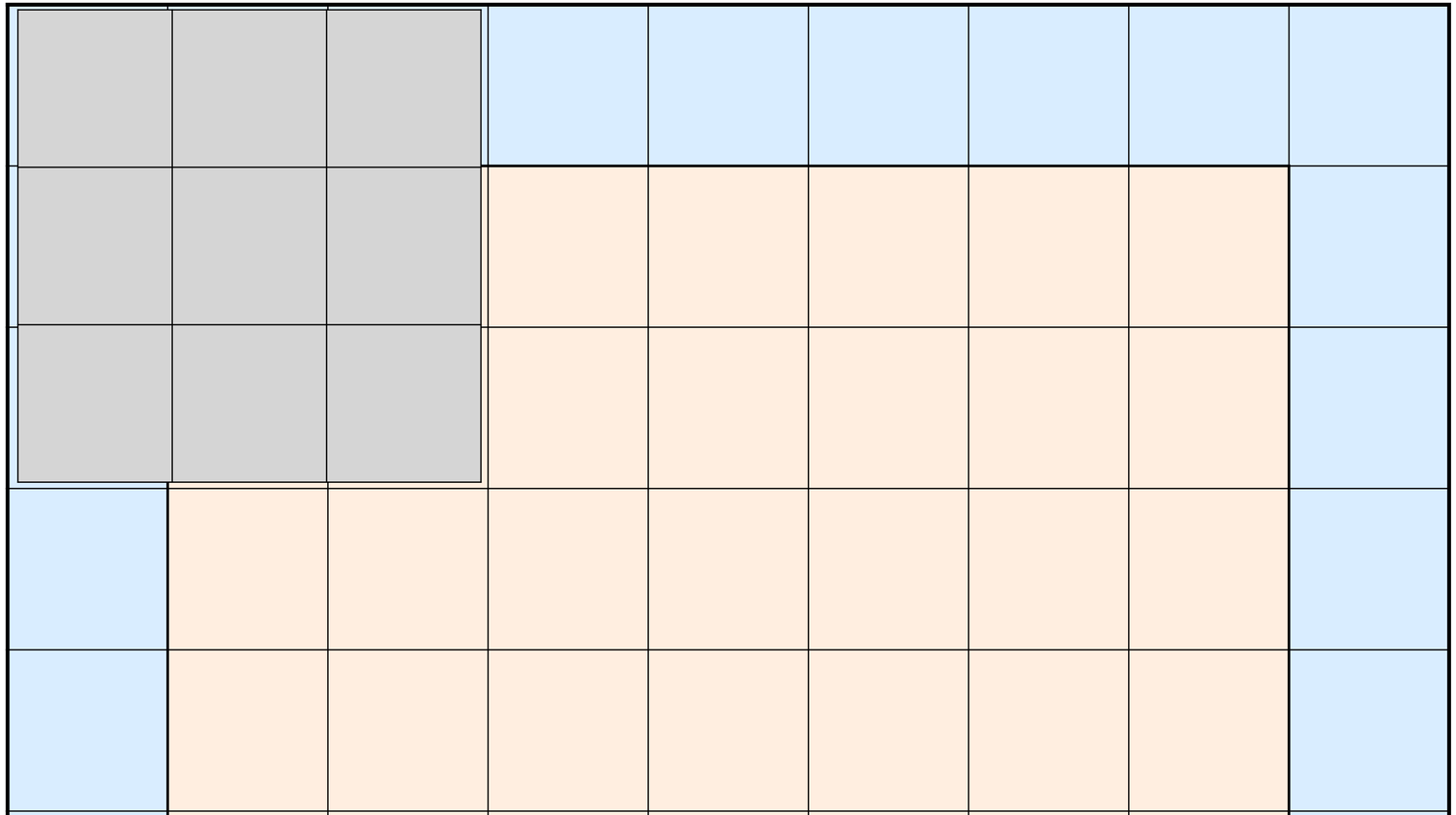
Padding

- We can get around this by adding padding (almost always with zeros) around the sides of the image allowing the kernel to fit at the edge locations



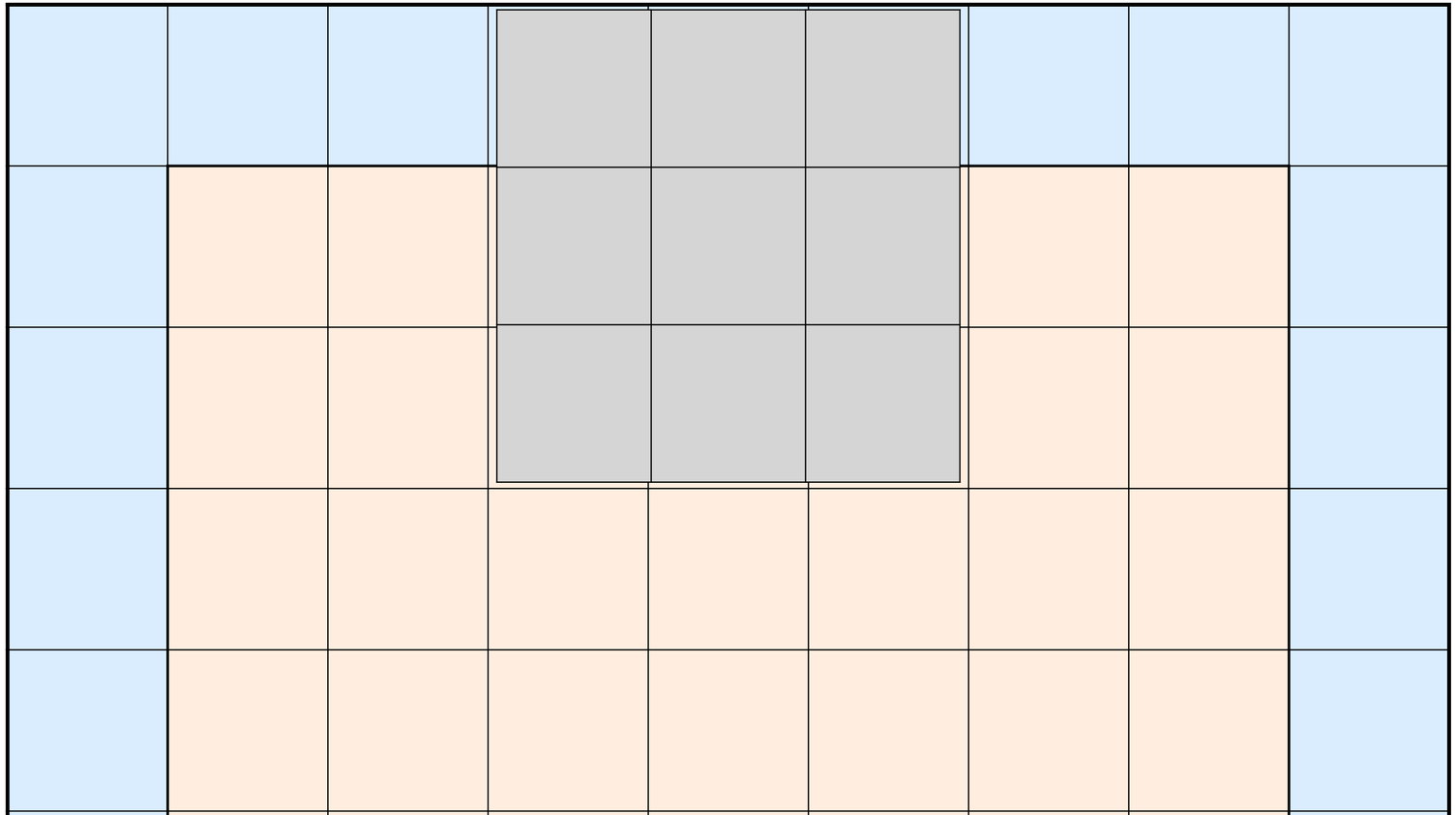
Stride

- I’ve said the convolution is applied everywhere, but it doesn’t have to be. The step by which the filter is scanned across the image is known as the stride, and it can be greater than 1.



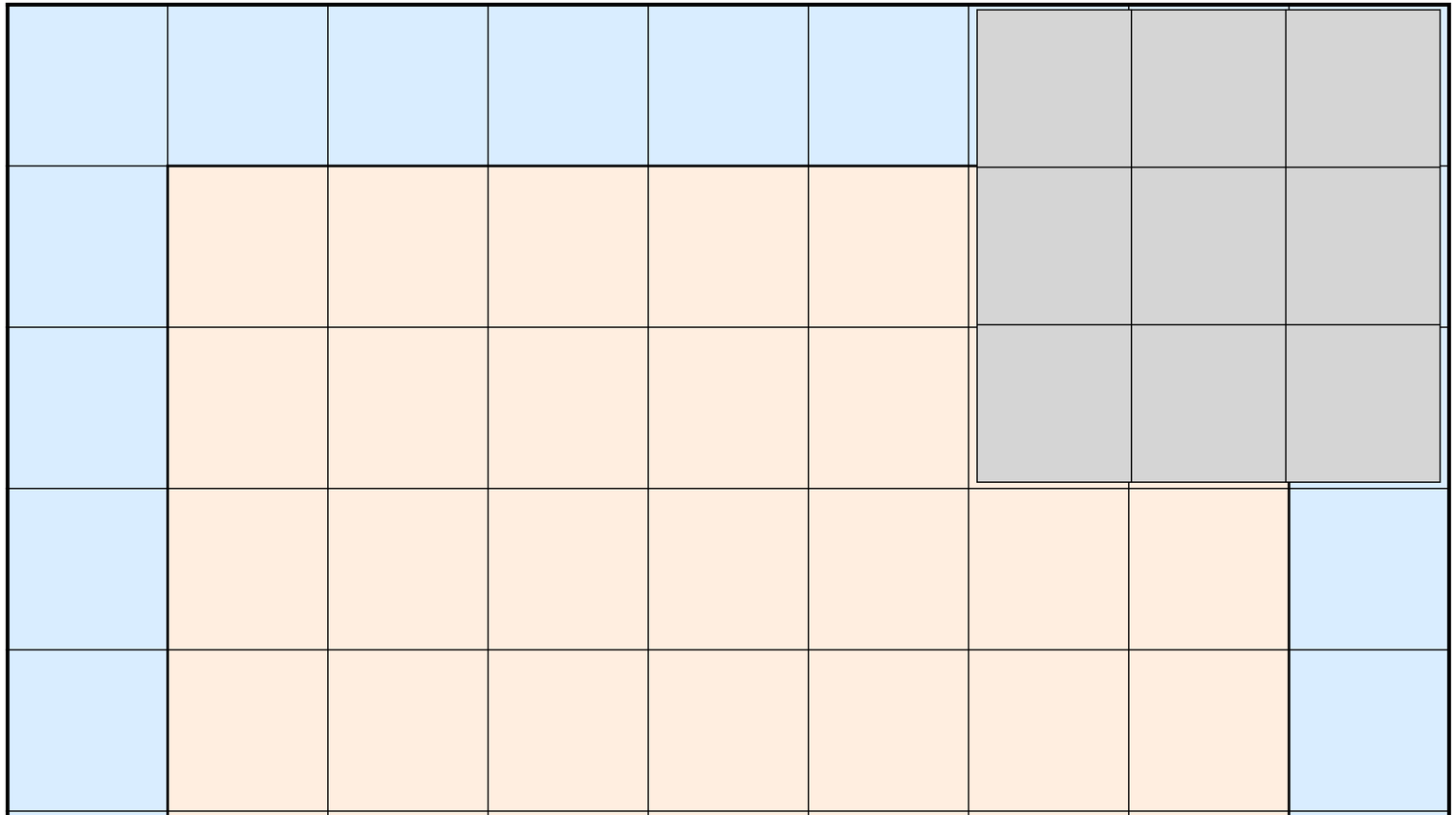
Stride

- I’ve said the convolution is applied everywhere, but it doesn’t have to be. The step by which the filter is scanned across the image is known as the stride, and it can be greater than 1.



Stride

- I’ve said the convolution is applied everywhere, but it doesn’t have to be. The step by which the filter is scanned across the image is known as the stride, and it can be greater than 1.



Stride

- I’ve said the convolution is applied everywhere, but it doesn’t have to be. The step by which the filter is scanned across the image is known as the stride, and it can be greater than 1.
- In this case the output image will again be smaller than the input.

Calculating the output size

- For kernel size K, input size D_{in}, padding P (each side) and stride S, we can calculate the output dimension D_{out} as:

$$D_{out} = \frac{D_{in} - K + 2P}{S} + 1$$

  → It is *possible* to have different values across and down, but we'll normally assume everything is square

- A common tactic is to have S=1 and K=2P+1, so that the output is the same size (spatially) as the input

Pooling

- We are usually interested in compound features of increasing spatial complexity and extent
- Pooling aggregates features over some (usually small, eg 2×2) spatial region into higher level “summary” features
- Most common pooling operation is max pooling: forward propagate the highest activation from a region
  - → If there’s a cat at any location in the region, there’s a cat in the region
  - → NB: backpropagated gradient through a max pool is zero for all the non-max paths
- Alternative is average pooling: forward propagate the mean
  - → Backprops everywhere but potentially weakens the forward activation signal

Pooling vs stride

- Both pooling and stride can “shrink” the image
- Stride is faster because it calculates the dot product fewer times
- But it may also step over interesting locations
- Pooling receives information from every filter at every location, and then decides what to do with afterwards
  - → It may still throw away information, but it shouldn’t (eg) miss a big peak
- Pooling is more commonly used, but as usual it’s a tradeoff

Receptive fields

- Extent of the input to which a filter is sensitive is called its receptive field
- Sensitivity to immediate input is bounded by kernel size (may be smaller)
- But pooling (or striding) shrinks the spatial dimensions, so filter is exposed to a larger region of the original image space (at reduced resolution)
- So filters at deeper layers can aggregate features over a much larger receptive field in the original image

Outputs

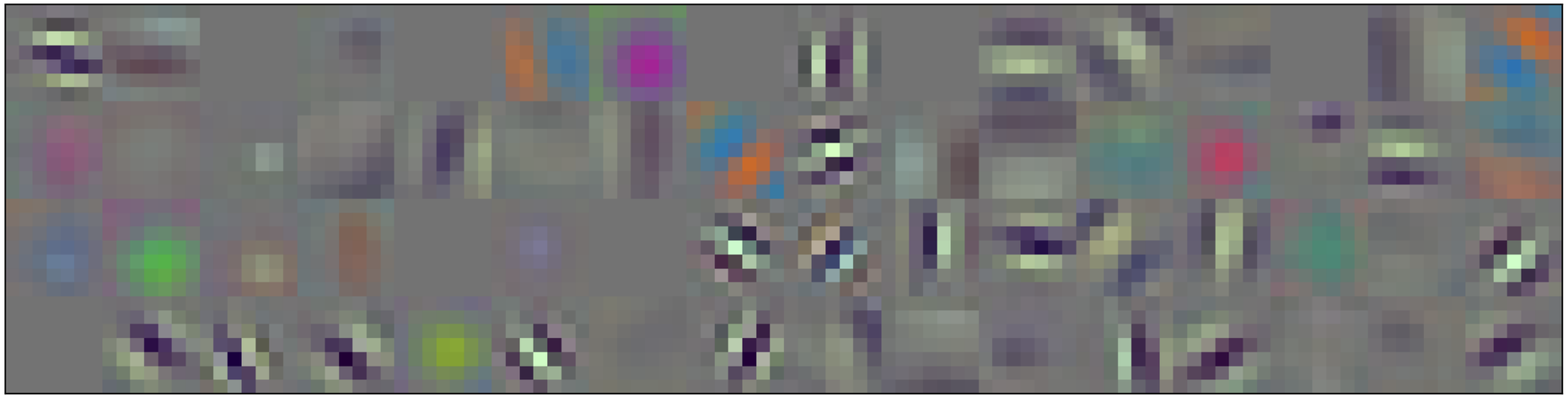
- Output activations from a trained convolutional layer constitute a projection of the image into a new basis, optimised with respect to the network's task
  - → If the problem is image classification, the output features should be such that the classes are more easily separable
  - → If the problem is some image-to-image regression problem, such as denoising, the output features should make it easier to reconstruct the desired parts of the image without the undesired parts (eg, signal not noise)
- Typically, this new representation is passed to some other network structure to perform the actual task — eg, a fully connected layer for classification
  - → Same interpretation applies at every layer, though — output of first conv layer is a new set of features that are better for building the feature maps at layer 2, and so on

Outputs

- What constitutes a good convolutional transformation will depend on the task
- But, because there's a lot of common structure in natural images, it turns out that some basic features are useful for many problems
- The early layers of convolutional networks tend to exhibit a kind of convergent evolution, coming up with filters that look pretty similar to the sort of edge and blob and grating detectors that vision researchers might explicitly design
- Deeper layers tend to be a bit less interpretable and more task-specific

Interpretation & visualisation

- Interpretation of the features and decision processes of CNNs tends to be quite difficult beyond the first few layers, as compound features made up of compound features made up of compound features all get tangled up
- For early layers we can (maybe) interpret by visualising the filters directly:



Interpretation & visualisation

- Later layers are pretty much impossible to interpret that way, because it's pretty much impossible to comprehend what the inputs are that are being combined, let alone what the combinations mean
- However, there's a neat trick for at least visualising (if not necessarily understanding) neuronal responses in deep networks, that comes as a sort of byproduct of backpropagation

Activity maximisation and DeepDream

- In backpropagation we compute gradients with respect to all the parameters and also all the inputs
- For training, we use the gradients to update the weights, but the inputs remain fixed — it would make no sense to fit the training data to the model
- But to visualise the behaviour of a trained model, we can keep the weights fixed and update the inputs to optimise for activation of one or more neurons
- In other words: generate images that strongly embody what a neuron or layer or prediction class is tuned to detect
- Popularised by DeepDream's psychedelia but also potentially informative about what a neuron is really responding to — not always what we'd guess

"Feature Visualization" Ohla et al 2017, Distill

ConvNet roundup

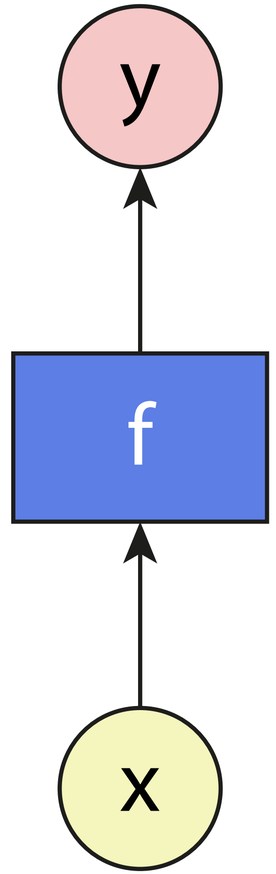
- Precursors date back to beginning of 1980s, and Yann LeCun's handwritten digit recogniser was successful in 1989, but the recent CNN renaissance follows success of AlexNet at ILSVRC 2012
- Subsequent progress in image classification, object detection, segmentation, autencoders, GANs etc has been very rapid and CNNs are still probably the king of the hill for image tasks (though with competition from RNNs, up next)
- They're also very well supported by standard libraries and frameworks (§6.4 & lab exercises), so they're easy to put to work
- Like everything in Deep Learning they're computationally intensive and extremely data hungry, but partly due to their popularity there are some tricks for mitigating the latter which we'll come back to in §6.4

6.3: Recurrent Networks

COMP0088 Introduction to Machine Learning · UCL Computer Science

Sequences

- All of the models that we have looked at so far — not just neural nets, but SVMs, random forests, logistic regression, nearest neighbours, AdaBoost — have been memoryless, context-free, one-to-one
- They take a single (albeit potentially complex and multi-valued) input, and they return a single (albeit ditto) output
- Each input is its own self-contained thing, independent from any that may have gone before and with no effect on any that may come after
- You put in an **x**, you get out a **y**



Context matters

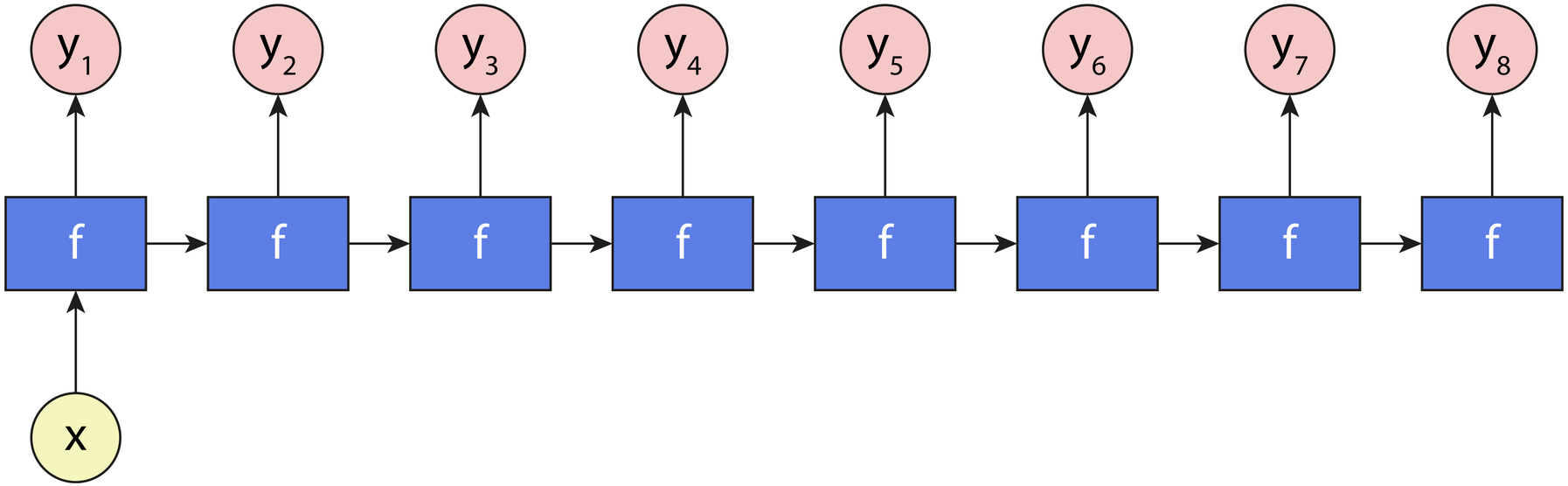
- For plenty of problems we might like to throw ML at, the data comes sequentially
- The order of arrival is important, the past affects the present, context matters
- Maybe there are an unknown number of inputs and you need to remember what you've received before
- Maybe there are an unknown number of outputs and you need to keep track of what you've said already
- Maybe both

There’s more to life than books, you know…

- …but not much more
- Sequence learning is relevant to many problems, but the ne plus ultra is natural language processing
- Words are variable length and often have order-dependent roles
- Sentences may be long or short, with nested clauses and forward references and dangling modifiers and poetic licences long overdue for renewal
- So we’ll tend to use language at least at one end of the following examples, though it is certainly possible to use RNNs for non-NLP content too

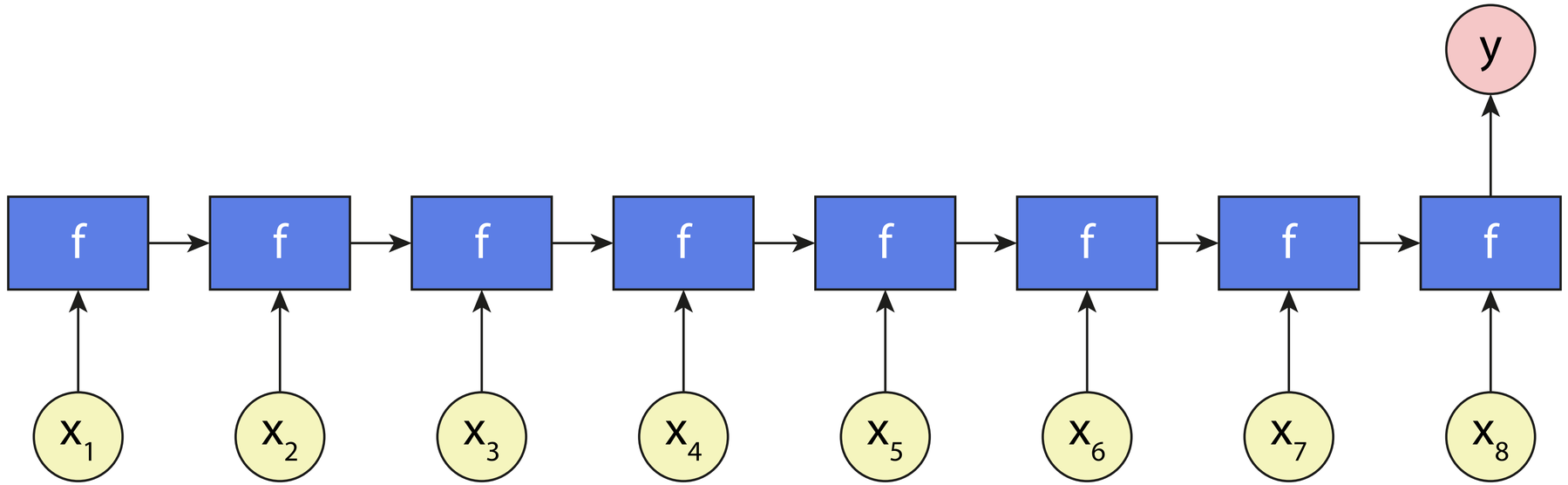
One to many: image captioning

- A picture is worth a thousand words, but let's not get carried away
- Input is one image
- Output is as much or as little text as the image merits



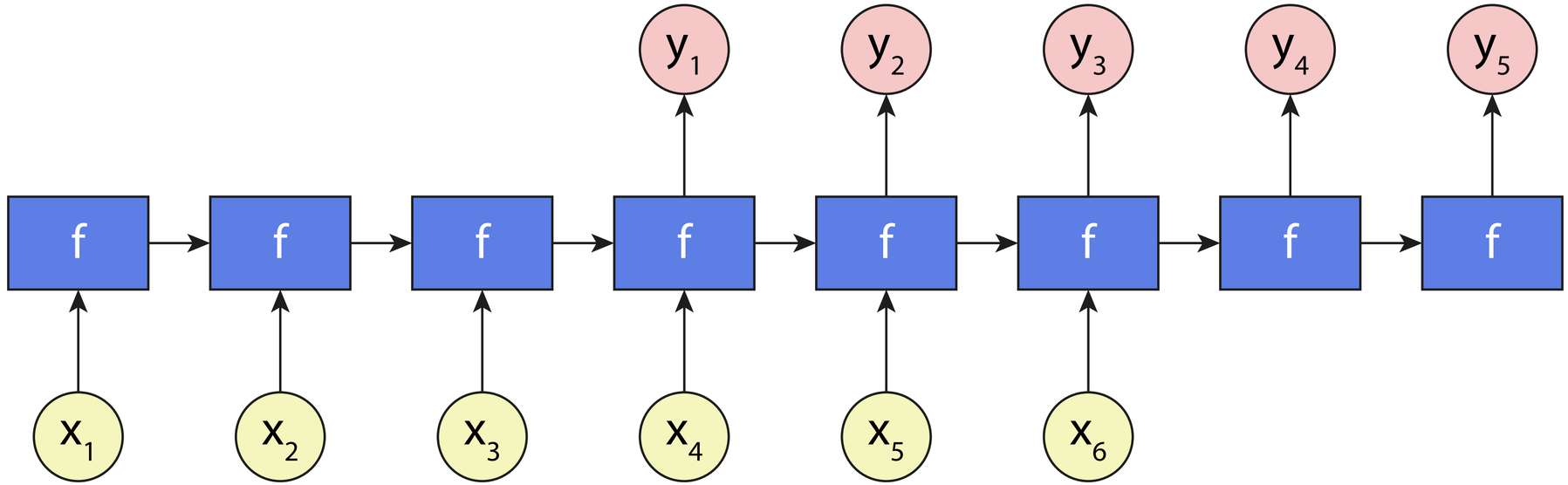
Many to one: sentiment classification

- Who reviews the reviewers? Who trolls the trolls?
- Input is a chunk of text
- Output is a class prediction: is it positive or negative? Is it fake or real?



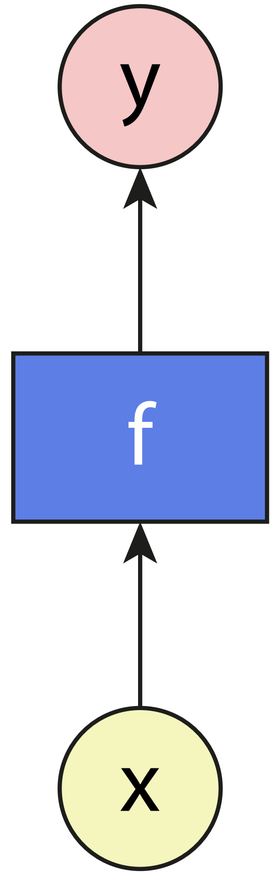
Many to many: machine translation

- You have not experienced Shakespeare until you have read him in the original Klingon
- One language may imply with a terse but nuanced syllable something it takes a painstaking paragraph to express in another



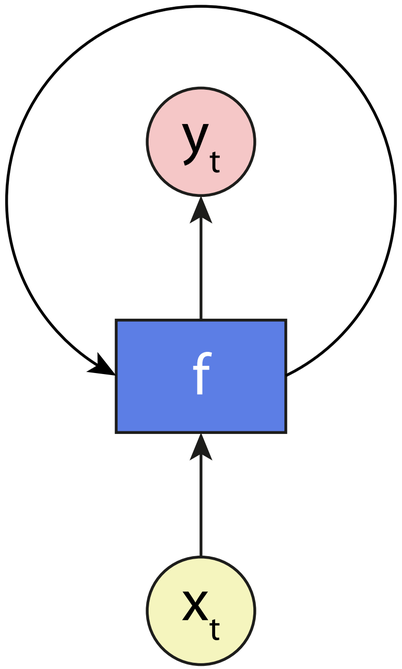
Good ol’ one to one

- I’ve described these as if variable length data is the be all and end all of recurrent nets, but many fixed size problems can also be tackled better or more efficiently in a recurrent or context-aware fashion
- This is not a significant example, but you’ll apply RNNs to ordinary image classification in this week’s lab exercises — treating an image as a sequence of vectors, rather than a grid of pixels



State Department

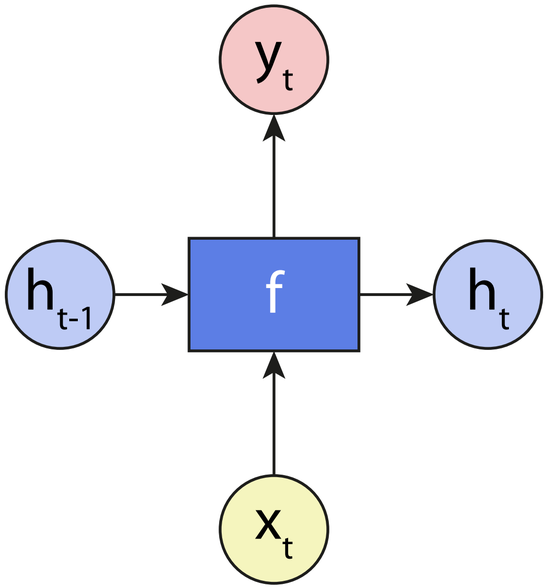
- The defining characteristic of recurrent neural networks is that some kind of state is carried over from processing one element of a sequence to the next



State Department

- Useful to represent state separate from function or model (which does not change), as an output from one step fed as input to the next

  → Important to distinguish internal hidden output **h** from external output **y**



Vanilla RNN functional form

- If we overlook this **h**/**y** double output, the general functional form is not that dissimilar from other neural networks we've seen
- As usual, we'll multiply some vector of inputs with a weights matrix to create some vector of outputs, applying some non-linear activation to the results
- In our previous nets that might look like this:

$$\mathbf{y} = g(\mathbf{x}\mathbf{W})$$

Vanilla RNN functional form

- However, in this case the vector of inputs is a concatenation of the old hidden state $\mathbf{h}_{t-1}$ with the new input data $\mathbf{x}_t$ and the (tanh) activations don't go directly to $\mathbf{y}_t$ but instead are saved to the new hidden state $\mathbf{h}_t$

$$\mathbf{h}_t = \tanh\left(\left[\mathbf{h}_{t-1}, \mathbf{x}_t\right]\mathbf{W}_h\right)$$

- The external output $\mathbf{y}_t$ is calculated as a further linear transformation of $\mathbf{h}_t$

$$\mathbf{y}_t = \mathbf{h}_t\mathbf{W}_y$$

Vanilla RNN functional form

- This is all repeated as many times as necessary to process the whole sequence
- The same weight matrices $\mathbf{W}_h$ and $\mathbf{W}_y$ are used at every step
- The initial state $\mathbf{h}_0$ before any data is received is typically just set to zeros

How do you fit this?

- As with other neural nets, we'll optimise the weights by gradient descent, using backpropagation to calculate the gradients
- We already know how to do this for all the constituent operations
  - → Technically we haven't covered tanh in the lectures or labs exercises, but it's a standard elementwise activation function with a well-known derivative
- But there are a couple of things to note

Hidden state

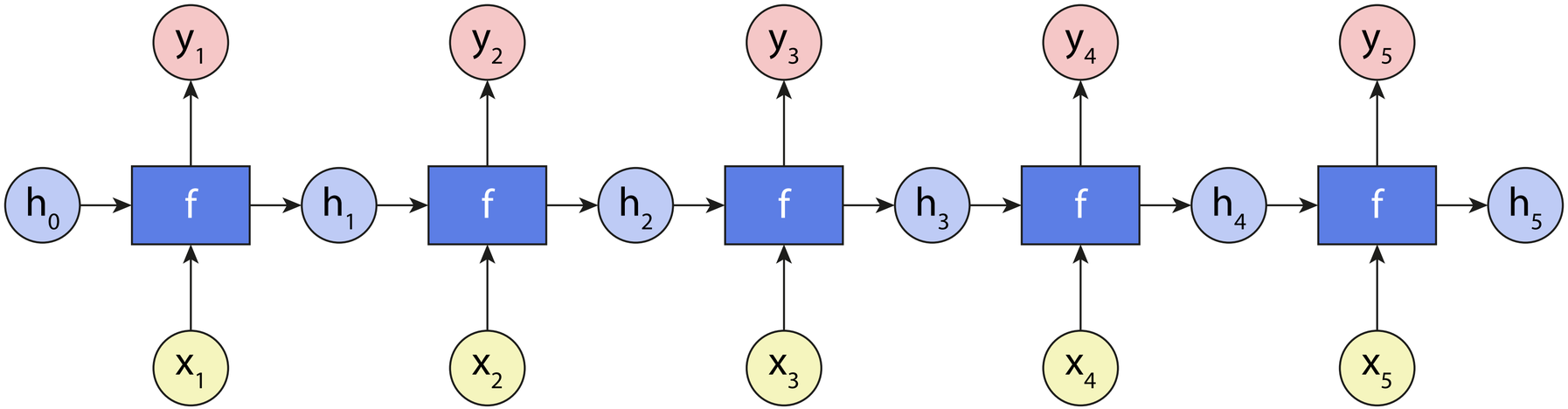
- The hidden state vector **h** is not directly supervised
  - → We do not specify what it is for or what values it should take. It just provides storage that the network can learn to use in whatever way it finds helpful.
  - → The idea of a hidden state shouldn't be worrying: we equally don't specify what the weights should be used for in hidden layers of an ordinary neural net, we just let them find their own level
- But the supervised outputs **y** depend on it, so there is one-step-removed supervision in conjunction with the output weights $\mathbf{W}_y$
- Since $\mathbf{h}_{t-1}$ is effectively an input to the network at time *t*, backpropagating into it is like backpropagating into the intermediate inputs **x** in an FC layer

Time travel

- A more substantive concern is: how can we propagate gradients from the future?
- But during training we already have all the data for future sequence steps
- What we’re calling “time” is just connections in the operation graph

Time travel

- We can unroll the sequence so that the network state at each time step is a separate subnet of a larger graph containing the full sequence — with that step's hidden state and the corresponding inputs and outputs too



- We can then propagate gradients through the "time" connections in this graph in the same way as through all the other connections

Weight sharing redux

- Just as with convolutional filters, all the time steps in the sequence use the same weights
- During backpropagation, each of the unrolled steps will have its own inputs, outputs and hidden state, and therefore also its own gradients with respect to those things
- When updating the weights, all those different gradients across the sequence need to be accounted for
- Again, just as with convolutional filters, we sum over the gradients from all the different iterations and update the weights using the aggregate values

Truncation

- In the full version of backpropagation through time the forward and backward passes go through the complete unrolled sequence each time
- Truncated BPTT instead runs only a chunk of the sequence at a time, running forward and backward through that subsequence before continuing with the next chunk
- Hidden state is carried forward from one chunk to the next, but gradients are not propagated back — we're already done with the chunk by then

Vanishing & exploding redux

- RNNs are highly susceptible to vanishing and exploding gradients because of the recurrence structure
- Backpropagation through the RNN involves not only a long chain of multiplications, but a long chain of multiplications by the same factor — the weights matrix $\mathbf{W}_h$
  - → Recall that to calculate the gradient with respect to the input we need to multiply by the weights. In this case **h** is the relevant input, and $\mathbf{W}_h$ is the same at every step
- If all the factors are different there is scope for balancing out, but multiplying the same one many times must explode or vanish unless the value is exactly 1 (the criterion is slightly different for matrices but the principle is the same)

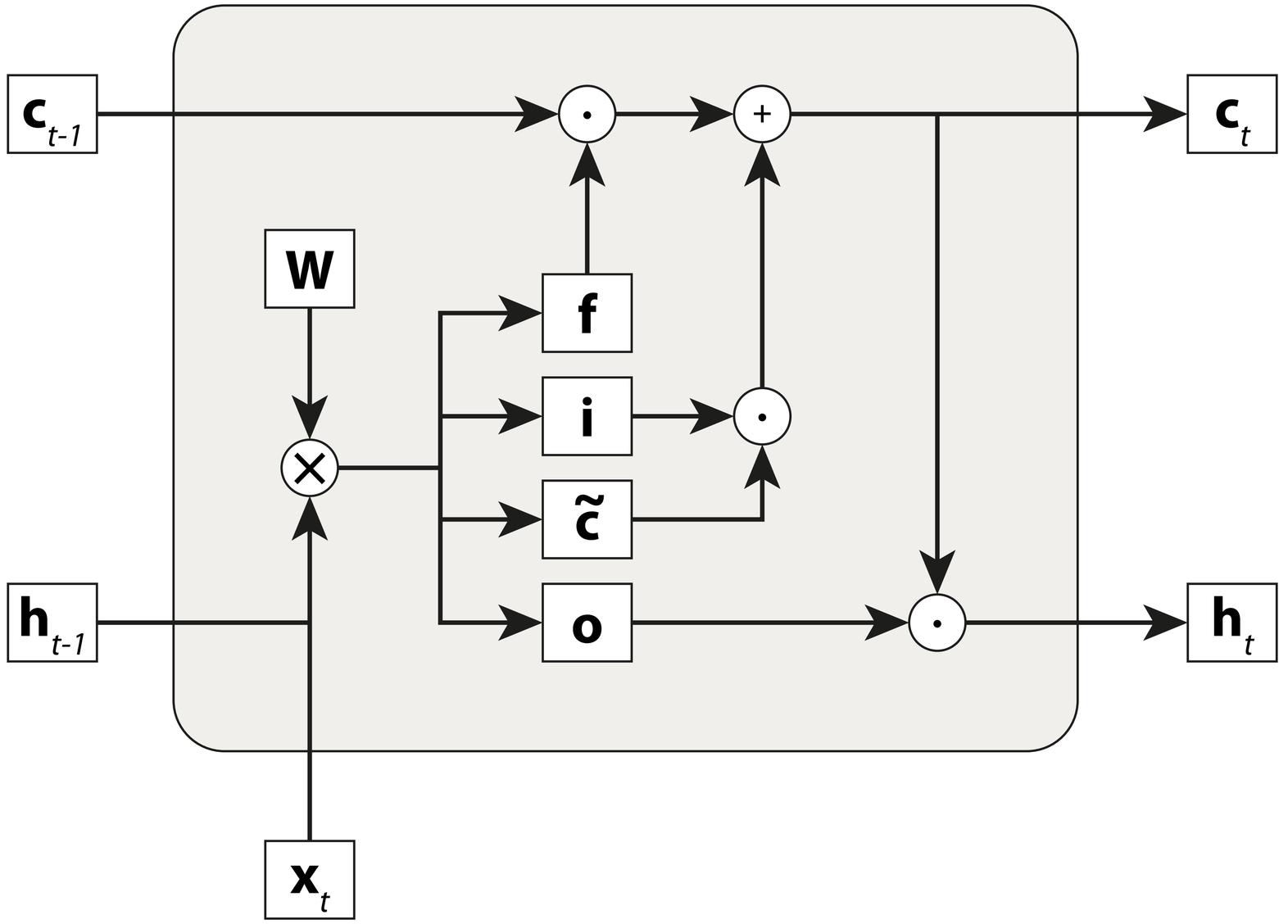
- Truncation helps a bit with this problem just by shortening the chain of multiplications
- But either way it means a vanilla RNN finds it very difficult to learn anything long term — the further into the past you go, the more the gradients go awry
- The solution is to introduce a more complex recurrence relation with different gradient pathways, providing a route for memory that doesn't just all go through the same cascade of multiplications
- We'll briefly consider two of these, LSTM and GRU

LSTM

- Long short-term memory
- Two separate hidden state vectors: hidden state and cell state
- Input [$\mathbf{h}_{t-1}$, $\mathbf{x}_t$] is multiplied by weights to produce 4 different intermediate "gating" vectors (forget, input, output, cell)
- sigmoid activation is applied to f, i, o; tanh to c
- gates are combined to update the cell and hidden states
- cell state updates use only elementwise multiplication and addition
- gradients still vanish through hidden state but repeated matrix multiplications by W are bypassed for the cell state allowing better gradient flow

LSTM recurrence equations

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\big([\mathbf{h}_{t-1}, \mathbf{x}_t]\mathbf{W}_f\big) \\
\mathbf{i}_t &= \sigma\big([\mathbf{h}_{t-1}, \mathbf{x}_t]\mathbf{W}_i\big) \\
\mathbf{o}_t &= \sigma\big([\mathbf{h}_{t-1}, \mathbf{x}_t]\mathbf{W}_o\big) \\
\tilde{\mathbf{c}}_t &= \tanh\big([\mathbf{h}_{t-1}, \mathbf{x}_t]\mathbf{W}_c\big) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \mathbf{o}_t \cdot \tanh\big(\mathbf{c}_t\big)
\end{aligned}
$$


c_{t-1}
c_t
W
f
i
$\tilde{c}$
o
h_{t-1}
h_t
x_t

GRU

- Gated recurrent unit
- Basically a cut-down LSTM, with no cell state or output gate but more complex update gating for hidden state
- A bit smaller and faster, has been shown to have similar performance for at least some tasks

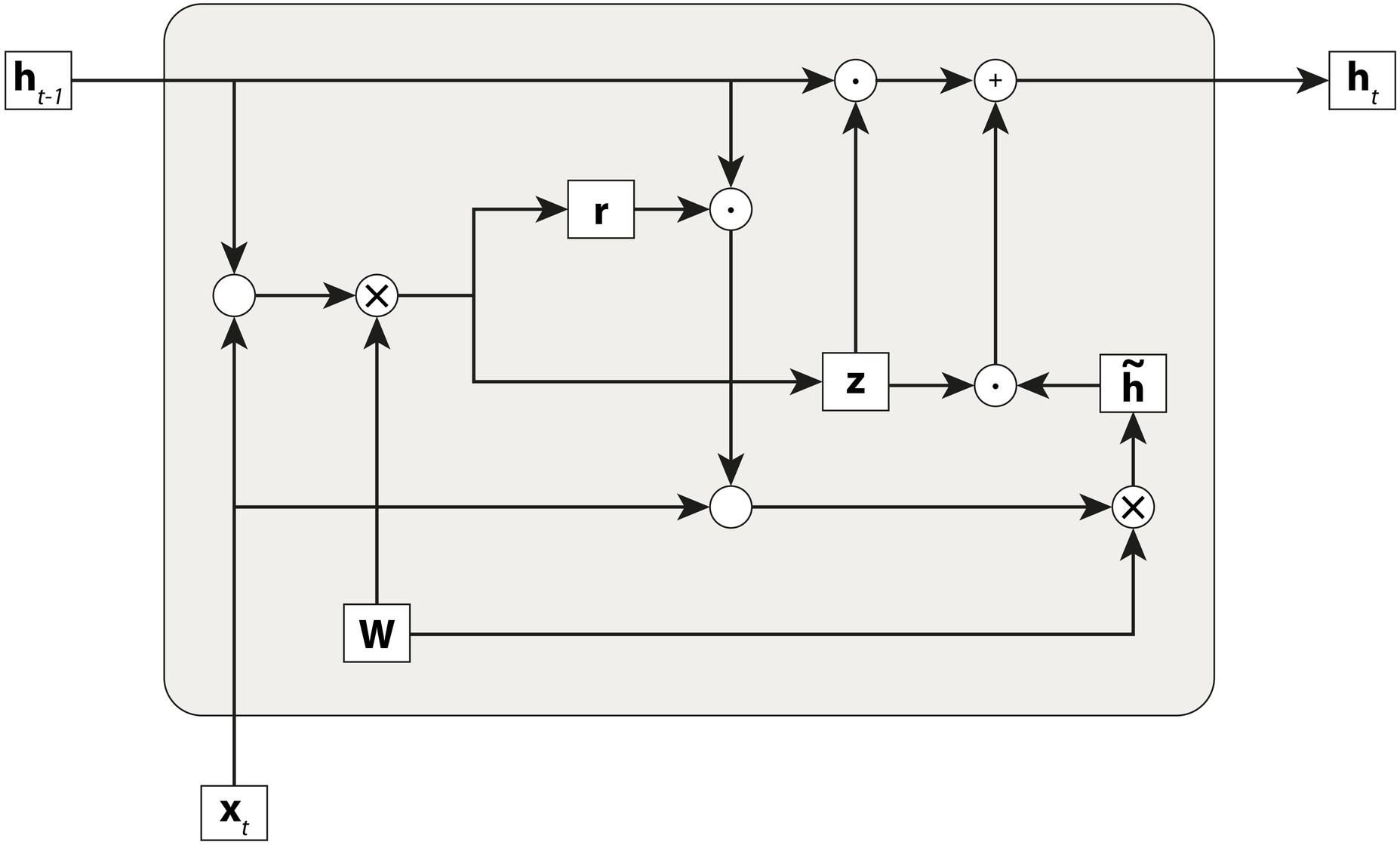
GRU recurrence equations

$$\mathbf{r}_t = \sigma\left(\left[\mathbf{h}_{t-1}, \mathbf{x}_t\right]\mathbf{W}_r\right)$$

$$\mathbf{z}_t = \sigma\left(\left[\mathbf{h}_{t-1}, \mathbf{x}_t\right]\mathbf{W}_z\right)$$

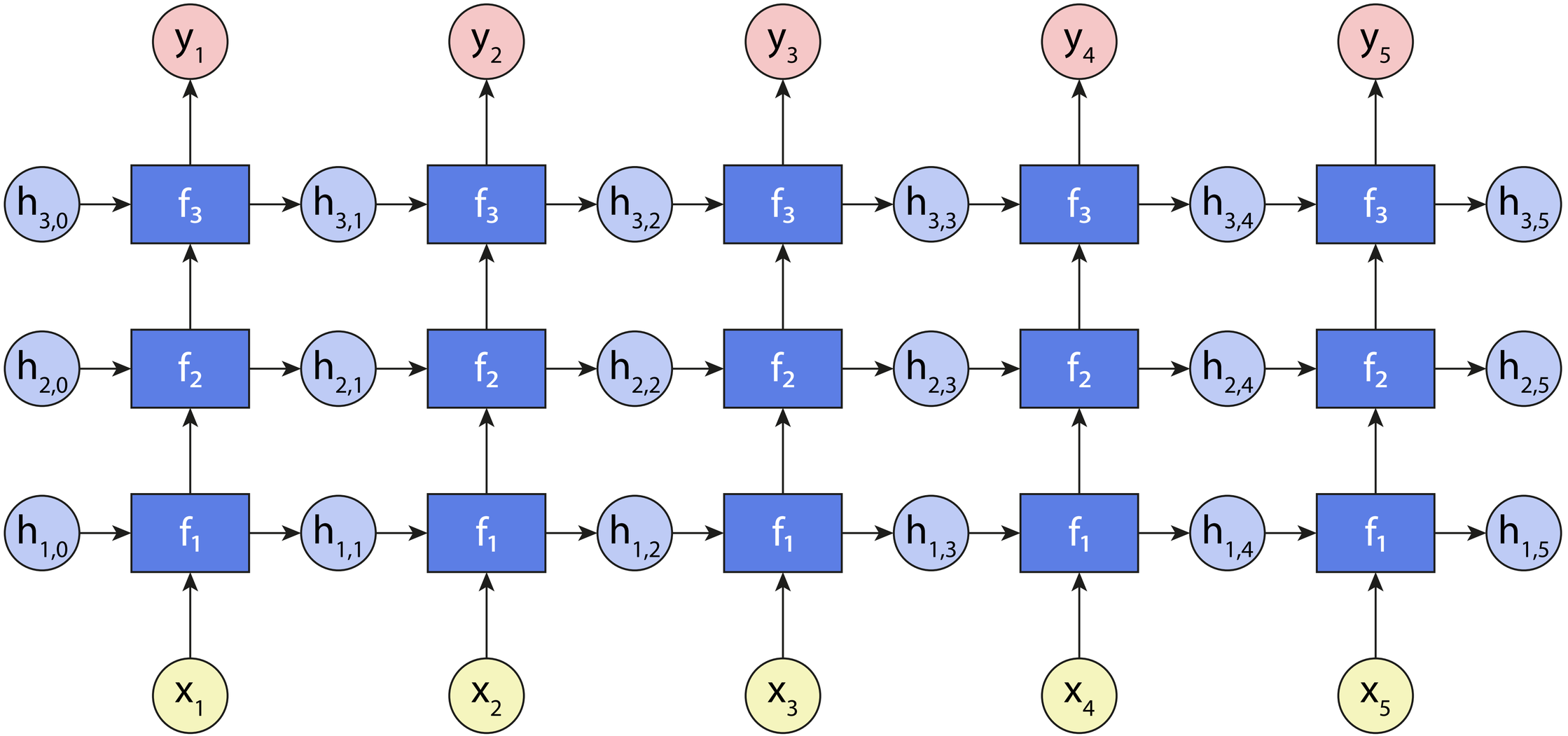
$$\tilde{\mathbf{h}}_t = \tanh\left(\left[\left(\mathbf{r}_t \odot \mathbf{h}_{t-1}\right), \mathbf{x}_t\right]\mathbf{W}_c\right)$$

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + \left(1 - \mathbf{z}_t\right) \odot \tilde{\mathbf{h}}_t$$

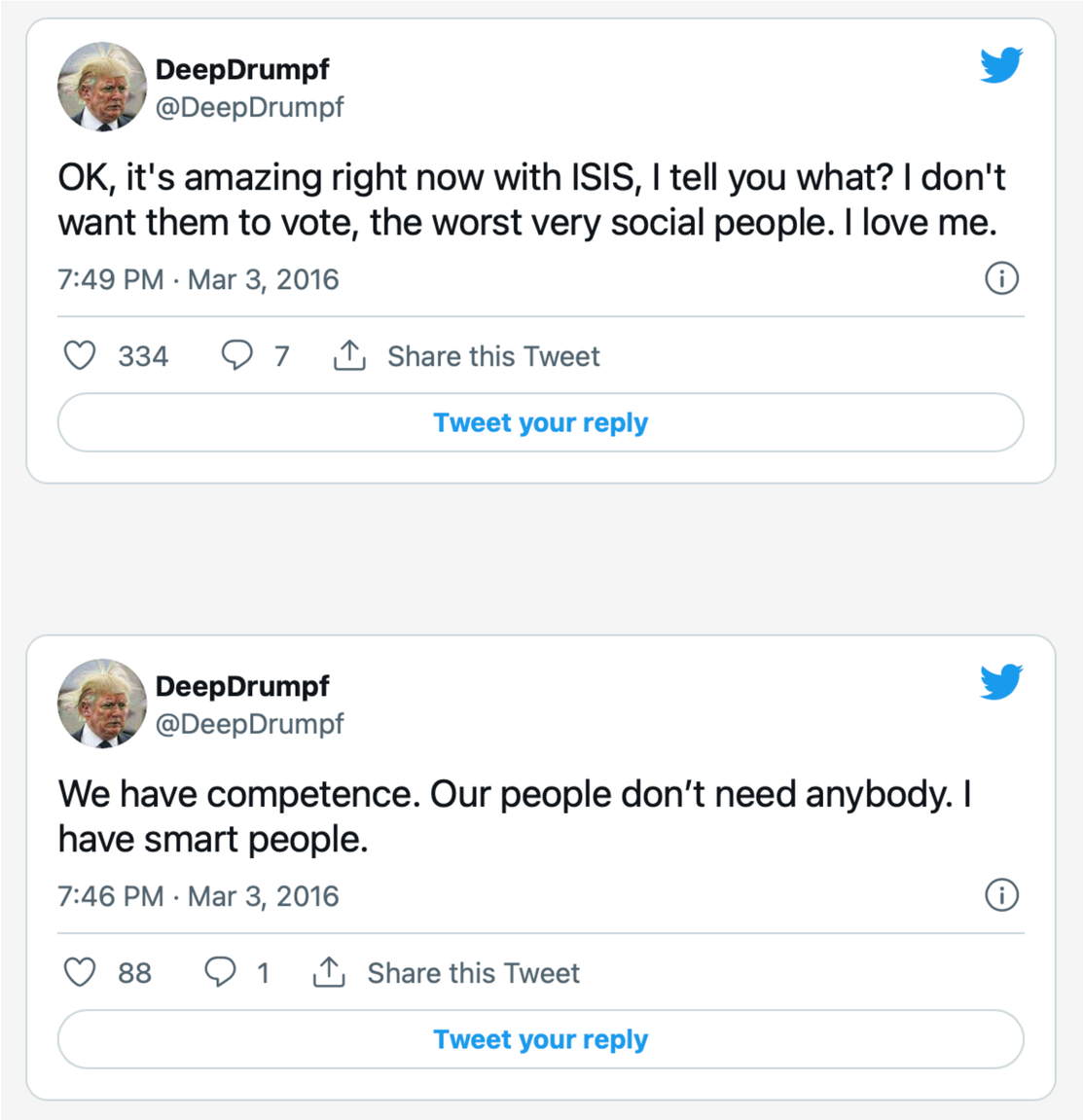

h_{t-1}
h_t
r
z
$\tilde{h}$
W
x_t

Multi-layer RNNs

- RNNs can be stacked together such that the ys from one become the xs to the next
  - → Analogous to adding multiple FC or conv layers in a deep feedforward network
- All the ideas of unrolling, BPTT, shared weights etc remain the same
  - → Weights are not shared between layers
- As with MLPs and CNNs, deeper RNNs have greater capacity, can learn more complex things, but are slower and more difficult to train and more susceptible to overfitting


y1
y2
y3
y4
y5
h3,0
f3
h3,1
f3
h3,2
f3
h3,3
f3
h3,4
f3
h3,5
h2,0
f2
h2,1
f2
h2,2
f2
h2,3
f2
h2,4
f2
h2,5
h1,0
f1
h1,1
f1
h1,2
f1
h1,3
f1
h1,4
f1
h1,5
x1
x2
x3
x4
x5

Recurrent nets power language silliness



RNN summary

- Recurrent networks are widely used for language tasks, and also for other sequence-based tasks such a generative music and video subtitling
- Vanilla RNNs are prone to vanishing and exploding gradients, so LSTMs tend to be the recurrence of choice, though GRUs may also perform well
- Sequential/contextual memory is a powerful tool for learning, enabling complex structures to be learned with less explicit *a priori* inductive bias
- The mechanics of RNNs, with some additional wrinkles, underpin very large language models such as BERT and GPT-3, which have gained a lot of media attention and which we'll come back to in week 9

6.4: Practicalities

COMP0088 Introduction to Machine Learning · UCL Computer Science

Regularisation

- Deep neural networks are often very big, with a huge number of parameters and great capacity for memorisation
- As a result, they can very easily overfit
- Classic weight norm regularisation methods can be applied — L_1 and L_2 are both commonly used
- The latter is often called weight decay — it can be framed as a downscaling of all the weights before applying a gradient update

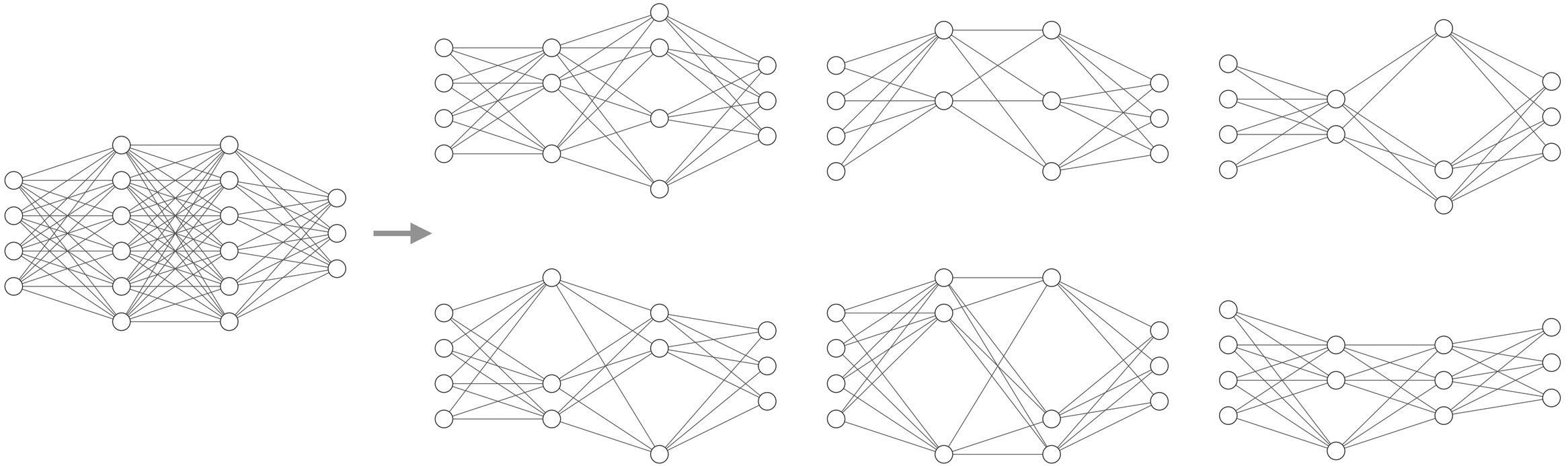
  → Penalty update is local to the layer, independent of backpropped global loss

Dropout

- Another pretty effective regularisation method, it randomly “switches off” a fraction of all neurons for each sample
  - → In practice implemented as elementwise multiplication by matrix of random 1s and 0s
- During backprop, gradients are averaged over the mini-batch, so should typically not be wholly switched off for most neurons most of the time
- Neuron-by-neuron dropout doesn’t make sense for convolutional layers, but there’s a convolution-specific equivalent that switches whole filters on and off

Dropout

- Akin to bagging in that it effectively aggregates over a population of different models, each using a different subset of neurons



Dropout

- Prevents network from relying too heavily on any individual neuron, improving robustness as well as generalisation
- At test time, activations must be scaled down to account for the fact that the whole population is always active whereas during training only part of it was at any one time

Batch normalisation

- Learning in a multi-layer network is predicated on the input data distribution at each layer — which comes from the layer before
- But training adjusts the weights and changes each layer's output distribution
- So each layer is trying to hit a moving target

  → Ioffe & Szegedy (2015) term this internal covariate shift

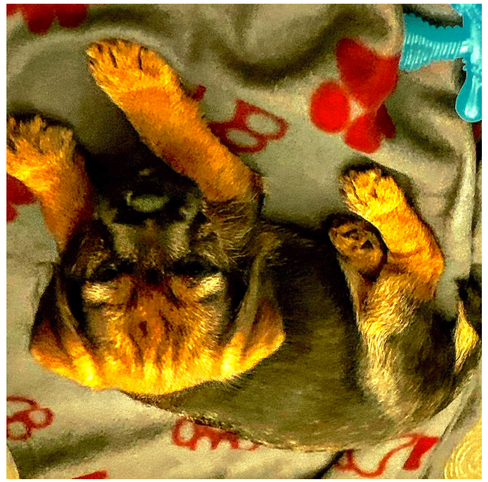
- Batch normalisation scales and offsets the outputs from each layer at every mini-batch, so they have a consistent (learned) mean and variance

Batch normalisation

- Speeds up learning
- Reduces dependence on careful initialisation
  - → Activations are always rescaled into a reasonable range rather than exploding or collapsing
- Also helps regularise
  - → Training samples are affected by mini-batch neighbours so are never the same twice
- Significantly increases model params — learns additional scale and offset at every connection
- Mini-batch stats are replaced by (fixed) training population stats at test time

Data augmentation

- Deep networks are extremely data hungry
- Need a lot of data to constrain the huge number of parameters
- But data can be scarce and expensive
- Data augmentation expands the training set by applying perturbations
  - → Eg, for image data: rotations, reflections, translation, blurring, noise, colour shifts
- Also has a regularising effect, reducing dependence on superficial details
  - → More general properties (catness) survive better across variations



Transfer learning

- As mentioned in §6.2, there are a lot of commonalities across images, and CNNs tend to converge on at least some kinds of universal feature
- Similar commonalities exist in other data types such as text and sound
- Many performant models exist that have already been trained for huge amounts of time on huge datasets at great computational expense
- These probably don't perform exactly the task that you want, but they embody a lot of detailed knowledge about their data modality (again, images are an important case in point)
- You can exploit that knowledge via transfer learning

Fine tuning

- Using pre-trained models as a starting point gives a big leg-up in training
- Instead of starting from a random initialisation that knows nothing, you start from a state that knows a lot about a possibly slightly-different task
- You keep as much or as little of that as you want

  → A common strategy is to keep most or all of the convolutional layers, which build a representation, and discard the final fully connected layers that do (eg) classification

- Pre-trained layers can be fine tuned (retrained a bit or a lot on your presumably scarcer task-specific data) or frozen (kept exactly as they are)

Commodification of deep learning

- We’ve covered a lot of ground in our two weeks on neural nets
- A decade ago or even more recently, much of this stuff was at the frontiers of research
- If you wanted to actually put it to practical use you would have to hand-build everything almost from scratch, wrestling with low-level numerical computing issues and libraries and graphics drivers etc
- That is definitely no longer the case

Deep learning frameworks

- Many of the structures and algorithms underlying neural networks and deep learning are well understood, with established standards and conventions about how to do things
- In the language of operation graphs and layers and flows of activations and gradients, many of the functional units are explicitly conceptualised as pluggable components, like Lego blocks — local units that only need to know about their immediate connections
- Frameworks can — and very much do — provide that functionality off the peg

  → Much more easily and also much better than you could do yourself

Layers and more

- A big chunk of such frameworks consists of implementations many types of layers that can just be wired together
  - → All the kinds of things we've talked about as network elements: linear layers, convolutions, recurrent units, pooling layers, regularising
- Also optimisation algorithms, data management utilities, standard implementations of well-known models with pre-trained weights, etc
- And, importantly, hardware support

GPUs

- Graphics is mostly linear algebra
- Making Doom run fast means optimising vector operations and doing them in a highly parallel fashion
- This turns out to be very useful for many kinds of more general number crunching, with machine learning a prime example
- GPU manufacturers noticed this and started providing driver support and interfaces for non-graphical computational tasks — NVidia's CUDA is the most widely known and used
- But coding directly to CUDA isn't much fun, and it's a bit of a distraction from your neural net endeavours

Someone else’s problem

- The advantage of using an existing kit of parts is that someone else can figure out how to make each part work on your TITAN X or whatever, and you can just reap the benefit
- Popular frameworks like TensorFlow and PyTorch provide GPU support more or less under the hood — you need to tell them to use your card, but they mostly take it from there
- The vast bulk of your code for model building and training and testing will just work seamlessly on both CPU and GPU without having to think about it

Tensor graphs

- A big part of how they do this is by being slightly abstracted
- When you wrote your MLP in NumPy, you wrote functions that were functions, passed around arrays of data that were actually arrays of data
- In TensorFlow and PyTorch the code will look as if you're doing that too, but in many cases the arrays and functions will actually be placeholder objects called Tensors
- These capture what you're doing, as graphs of operations on containers of data, but the actual functions and actual data might be deferred to some other underlying implementation — for example, on the GPU
- So, eg, you generally can't call NumPy functions directly on Tensors — the data isn't really there

TensorFlow vs PyTorch

- In the case of Python, the two dominant DL frameworks are TensorFlow (including the higher level model interface Keras) and PyTorch
- These are the sort of choices that inspire theological fervour in techie circles, but I don't propose to get into a holy war
- In the early days of both they had different emphases and the ways of working with them were significantly different – in particular, TF required you to build and compile your whole model before you could use it
- These days there's not much to choose between them and you should us whichever is more convenient or familiar
- You'll see some simple examples of using PyTorch in the lab exercises

Envoi

- Pretty much everything we've discussed in weeks 5 & 6 is readily available in PyTorch, TensorFlow/Keras and any other framework you care to mention
- Once you grasp the fundamentals it's easy to put them into practice
- For most problems there will be many possible solutions, and the difficulty is often as much in narrowing down the choices as in making the chosen approach work
- We'll come back to a few more neural net and deep learning topics in the final two weeks of the module
- Next: no **y**s, no wherefore!