Mathematics of AI

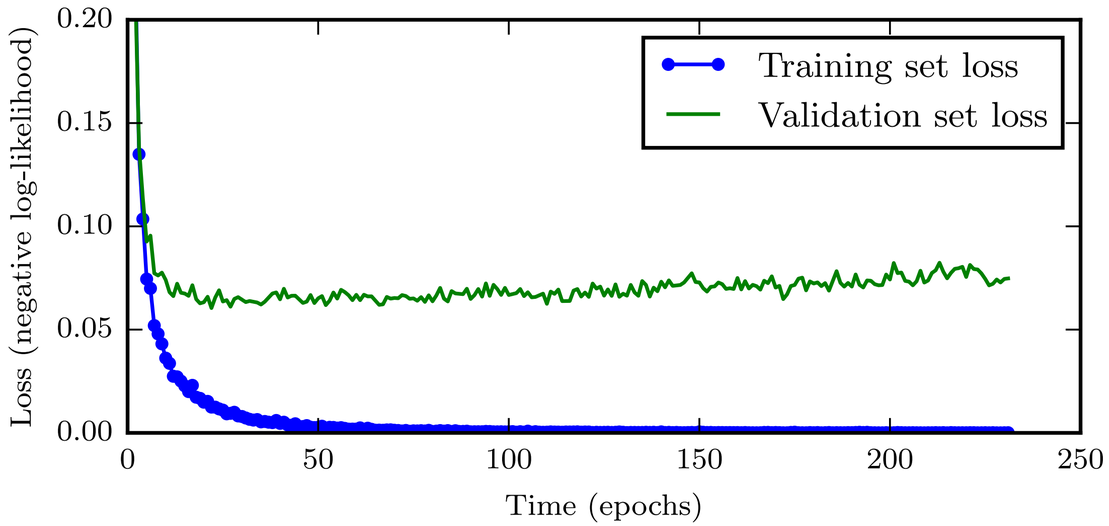
Convolutional neural networks (CNNs)

Goodfellow *et al*, *Deep Learning*
Liquet *et al, MEML*
Prince, *Understanding Deep Learning (ch 10)*
Random papers, textbooks, and online resources

Regularisation

In deep learning

- “Just get more data” —Guy from Google
- Early stopping, i.e., “cheating”
- … (bagging, multi-task learning, …)
- Sparse representations! (cf. compressive sensing…)
- Dropout



Definition

- “Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.”

Sparse Representations

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \tag{7.46}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad$ $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ $\quad$ $\boldsymbol{x} \in \mathbb{R}^n$

Sparse Representations

$$\underbrace{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}}_{\boldsymbol{y} \in \mathbb{R}^m} = \underbrace{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}}_{\boldsymbol{B} \in \mathbb{R}^{m \times n}} \underbrace{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}_{\boldsymbol{h} \in \mathbb{R}^n} \tag{7.47}$$

Sparse Representations

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \tag{7.47}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad$ $\boldsymbol{B} \in \mathbb{R}^{m \times n}$ $\quad$ $\boldsymbol{h} \in \mathbb{R}^n$

Sparser representation of data ==> greater generalisability

$$\arg\min_{\boldsymbol{h}, \|\boldsymbol{h}\|_0 < k} \|\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}\|^2,$$

Sparse Representations

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \tag{7.47}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad$ $\boldsymbol{B} \in \mathbb{R}^{m \times n}$ $\quad$ $\boldsymbol{h} \in \mathbb{R}^n$

Learn:
- **Dictionary W (or D)**
- **Representation h (or f)**

$$\arg\min_{\boldsymbol{h}, \|\boldsymbol{h}\|_0 < k} \|\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}\|^2,$$

Sparse Representations

The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization

Adam Coates ACOATES@CS.STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Stanford University, 353 Serra Mall, Stanford, CA 94305

Train / Encoder	Test Acc.
RP / T	79.1%
SC / SC	78.8%
SC / T	78.9%
OMP-1 / SC	78.8%
OMP-1 / T	79.4%
OMP-10 / T	80.1%
OMP-1 / T ($d = 6000$)	**81.5**%
(Coates et al., 2011) 1600 features	77.9%
(Coates et al., 2011) 4000 features	79.6%
Improved LCC (Yu & Zhang, 2010)	74.5%
Conv. DBN (Krizhevsky, 2010)	78.9%
Deep NN (Ciresan et al., 2011)	80.49%

Lots of ways to learn dictionaries and representations!

https://openreview.net/forum?id=By4afoWO-B

results using dictionaries created from random noise, randomly sampled exemplars, and vector quantization and show that the last two yield perfectly usable dictionaries in every case.

3. Learning framework

Given an unsupervised learning algorithm, we learn a new feature representation from unlabeled data by employing a common framework. Our feature learning framework is like the patch-based system presented in (Coates et al., 2011) with a few modifications. It shares most of its key components with prior work in visual word models (Csurka et al., 2004; Lazebnik et al., 2006; Agarwal & Triggs, 2006).

In order to generate a set of features, our system first accumulates a batch of small image patches or image descriptors harvested from unlabeled data. When learning from raw pixels, we extract 6 pixel square patches, yielding a bank of vectors that are then normalized[1] and ZCA whitened (Hyvarinen & Oja, 2000) (retaining full variance). If we are learning from SIFT descriptors, we simply take single descriptors to form a bank of 128-dimensional vectors.

Given the batch of input vectors, $x^{(i)} \in \mathbb{R}^n$, an unsupervised learning algorithm is then applied to learn a dictionary of d elements, $D \in \mathbb{R}^{n \times d}$, where each column $D^{(j)}$ is one element. In order to make all of our algorithms consistent (so that we may freely change the choice of encoder), we will make certain that each of the algorithms we use produces normalized dictionary elements: $||D^{(j)}||_2^2 = 1$.

In this work, we will use the following unsupervised learning algorithms for training the dictionary D:

1. **Sparse coding (SC):** We train the dictionary using the L1-penalized sparse coding formulation. That is, we optimize

$$\min_{D,s(i)} \sum_i ||Ds^{(i)} - x^{(i)}||_2^2 + \lambda ||s^{(i)}||_1 \qquad (1)$$
$$\text{subject to } ||D^{(j)}||_2^2 = 1, \forall j$$

   using alternating minimization over the sparse codes, $s^{(i)}$, and the dictionary, D. We use the coordinate descent algorithm to solve for the sparse codes (Wu & Lange, 2008).

2. **Orthogonal matching pursuit (OMP-k):** Similar to sparse coding, the dictionary is trained using an alternating minimization of

$$\min_{D,s(i)} \sum_i ||Ds^{(i)} - x^{(i)}||_2^2 \qquad (2)$$
$$\text{subject to } ||D^{(j)}||_2^2 = 1, \forall j$$
$$\text{and } ||s^{(i)}||_0 \leq k, \forall i$$

   where $||s^{(i)}||_0$ is the number of non-zero elements in $s^{(i)}$. In this case, the codes $s^{(i)}$ are computed (approximately) using Orthogonal Matching Pursuit (Pati et al., 1993; Blumensath & Davies, 2007) to compute codes with at most k non-zeros (which we refer to as "OMP-k"). For a single input $x^{(i)}$, OMP-k begins with $s^{(i)} = 0$ and at each iteration greedily selects an element of $s^{(i)}$ to be made non-zero to minimize the residual reconstruction error. After each selection, $s^{(i)}$ is updated to minimize $||Ds^{(i)} - x^{(i)}||_2^2$ over $s^{(i)}$ allowing only the selected elements to be non-zero.

   Importantly, OMP-1 is a form of "gain-shape vector quantization" (and is similar to K-means when the data $x^{(i)}$ and the dictionary elements $D^{(j)}$ all have unit length). Specifically, it chooses $k = \arg\max_j |D^{(j)\top} x^{(i)}|$, then sets $s_k^{(i)} = D^{(k)\top} x^{(i)}$ and all other elements of $s^{(i)}$ to 0. Holding these "one hot" codes fixed, it is then easy to solve for the optimal D in (2).

3. **Sparse RBMs and sparse auto-encoders (RBM, SAE):** In some of our experiments, we train sparse RBMs (Hinton et al., 2006) and sparse auto-encoders (Ranzato et al., 2007; Bengio et al., 2006), both using a logistic sigmoid nonlinearity $g(Wx + b)$. These algorithms yield a set of weights W and biases b. To obtain the dictionary, D, we simply discard the biases and take $D = W^\top$, then normalize the columns of D.

4. **Randomly sampled patches (RP):** We also use a heuristic method for populating the dictionary: we fill the columns of D with normalized vectors sampled randomly from amongst the $x^{(i)}$.

5. **Random weights (R):** It has also been shown that completely random weights can perform surprisingly well in some tasks (Jarrett et al., 2009; Saxe et al., 2010). Thus, we have also tried filling the columns of D with vectors sampled from a unit normal distribution (subsequently normalized to unit length).

After running any of the above training procedures, we have a dictionary D. We must then define an "encoder" that, given D, maps a new input vector x to

[1] We subtract the mean and divide by the standard deviation of the pixel values.

algorithms consistent (so that we may freely change the choice of encoder), we will make certain that each of the algorithms we use produces normalized dictionary elements: $||D^{(j)}||_2^2 = 1$.

In this work, we will use the following unsupervised learning algorithms for training the dictionary D:

1. **Sparse coding (SC):** We train the dictionary using the L1-penalized sparse coding formulation. That is, we optimize

constrained optimisation

$$\min_{D,s(i)} \sum_i ||Ds^{(i)} - x^{(i)}||_2^2 + \lambda||s^{(i)}||_1 \qquad (1)$$

$$\text{subject to } ||D^{(j)}||_2^2 = 1, \forall j$$

   using alternating minimization over the sparse codes, $s^{(i)}$, and the dictionary, D. We use the coordinate descent algorithm to solve for the sparse codes (Wu & Lange, 2008).

2. **Orthogonal matching pursuit (OMP-k):** Similar to sparse coding, the dictionary is trained

3. **Sparse RBMs and sparse auto-encoders (RBM, SAE):** In some of our experiments, we train sparse RBMs (Hinton et al., 2006) and sparse auto-encoders (Ranzato et al., 2007; Bengio et al., 2006), both using a logistic sigmoid nonlinearity $g(Wx + b)$. These algorithms yield a set of weights W and biases b. To obtain the dictionary, D, we simply discard the biases and take $D = W^\top$, then normalize the columns of D.

4. **Randomly sampled patches (RP):** We also use a heuristic method for populating the dictionary: we fill the columns of D with normalized vectors sampled randomly from amongst the $x^{(i)}$.

5. **Random weights (R):** It has also been shown that completely random weights can perform surprisingly well in some tasks (Jarrett et al., 2009; Saxe et al., 2010). Thus, we have also tried filling the columns of D with vectors sampled from a unit normal distribution (subsequently normalized to unit length).

After running any of the above training procedures, we have a dictionary D. We must then define an "encoder" that, given D, maps a new input vector x to

[1] We subtract the mean and divide by the standard deviation of the pixel values.

results using dictionaries created from random noise, randomly sampled exemplars, and vector quantization and show that the last two yield perfectly usable dictionaries in every case.

3. Learning framework

Given an unsupervised learning algorithm, we learn a new feature representation from unlabeled data by employing a common framework. Our feature learning framework is like the patch-based system presented in (Coates et al., 2011) with a few modifications. It shares most of its key components with prior work in visual word models (Csurka et al., 2004; Lazebnik et al., 2006; Agarwal & Triggs, 2006).

In order to generate a set of features, our system first accumulates a batch of small image patches or image descriptors harvested from unlabeled data. When learning from raw pixels, we extract 6 pixel square patches, yielding a bank of vectors that are then normalized[1] and ZCA whitened (Hyvarinen & Oja, 2000)

using an alternating minimization of

constrained optimisation

$$\min_{D,s(i)} \sum_i ||Ds^{(i)} - x^{(i)}||_2^2 \qquad (2)$$
$$\text{subject to } ||D^{(j)}||_2^2 = 1, \forall j$$
$$\text{and } ||s^{(i)}||_0 \leq k, \forall i$$

where $||s^{(i)}||_0$ is the number of non-zero elements in $s^{(i)}$. In this case, the codes $s^{(i)}$ are computed (approximately) using Orthogonal Matching Pursuit (Pati et al., 1993; Blumensath & Davies, 2007) to compute codes with at most k non-zeros (which we refer to as "OMP-k"). For a single input $x^{(i)}$, OMP-k begins with $s^{(i)} = 0$ and at each iteration greedily selects an element of $s^{(i)}$ to be made non-zero to minimize the residual reconstruction error. After each selection, $s^{(i)}$ is updated to minimize $||Ds^{(i)} - x^{(i)}||_2^2$ over $s^{(i)}$ allowing only the selected elements to be non-zero.

Importantly, OMP-1 is a form of "gain-shape vec-

Constrained optimisation <=> regularisation

Matt's notes

Reconstruction, inverse problems, and regularisation

Transformation is just

$$\mathbf{z} = W\mathbf{x}, \tag{6}$$

but if this is an invertible transform then W is a square, non-singular matrix. But if we are working, for instance, with an overcomplete dictionary, the transform matrix W might be non-square. Also in many problems we are forced into measuring (by cost or other practicalities) such that we measure through some transformation.

Hence, a very common problem is reconstruction of a signal $\mathbf{x}$ from a transformed measurement $\mathbf{z}$. Classic problems in this form are sometimes called *tomography* problems by analogy to CAT (Computer Axial Tomomography) scans. These reconstruction problems are inverse problems (I shall stick to linear problems for now). They have a set of general characteristics:

- The equation above is more realistically $\mathbf{z} = W\mathbf{x} + \epsilon$, where ϵ is noise. Hence, there may not actually be an exact solution to the set of equations.
- The matrix W is usually not square and in many cases the length of $\mathbf{z}$ is much smaller than $\mathbf{x}$ so the equations are (massively) underconstrained.

So we can't solve this using the usual linear algebra you have been taught.

The easiest way to create a framework for all of these problems is to consider them as optimisation problems, i.e., we write them as

$$\underset{\mathbf{x}}{\operatorname{argmin}}\, f(\mathbf{x}) \text{ such that } \mathbf{z} = W\mathbf{x}. \tag{7}$$

but for our purposes we solve a new optimisation problem

$$\underset{\mathbf{x}}{\operatorname{argmin}}\, f(\mathbf{x}) + \lambda||\mathbf{z} - W\mathbf{x}||. \tag{8}$$

That is we solve a problem that tries to find a good solution that closely satisfies the measurment constaints (the extra term in the objective is a type of Lagrange multiplier).

Note the extra parameter λ. This let's you perform a tradeoff between your "model" implied by $f(\cdot)$ and your measurements. So

- If you think the measurements are very noisy, you migth chose a smaller λ.
- If you think the measurements are very accurate, you might chose a larger λ.

The secret of all of these, however, lies in the details of $f(\cdot)$ and the norm we use $||\cdot||$. There are very many alternatives.

cf. Compressive sensing

Matt's notes

Sensing and compression as one action = compressive sensings

Now we get to put all of these ideas together.

1. Remember that transformation and sensing can happen as one step in hardware (e.g., JPEG) so
    - I will assume that my signal $\mathbf{x}$ is compressible (after some tranformation of the raw signal), and
    - I will also use a random projection matrix A to perform a dimension reduction on the signal to get

$$\mathbf{y} = A\mathbf{x} \tag{9}$$

    where $\mathbf{x} \in R^n$ and $\mathbf{y} \in R^k$ where $k \ll n$ (i.e., A is a $k \times n$ matrix).

2. Now I want to reconstruct an approximation $\hat{\mathbf{x}}$ from $\mathbf{y}$ and I shall suggest the following:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \lambda||\mathbf{x}||_0 + ||\mathbf{z} - W\mathbf{x}||_2^2. \tag{10}$$

(actually use
an L1 norm)

Compressive sensing

Linkages throughout the course

- CS is a constrained optimisation problem
- … which itself can be cast as unconstrained optimisation with a regularisation (penalty term)
- CS (with the L1 penalty term) is essentially LASSO (L2 penalty is ridge regression)
- Also essentially the same thing as “sparse representations” —> regularisation for deep learning (and works well)
- “Deep” methods essentially do the same thing as CS (e.g., https://arxiv.org/abs/1905.06723)

Compressive sensing

Your turn!

- https://github.com/AdelaideUniversityMathSciences/MathsForAI/blob/main/Code/compressive_sensing_example.ipynb
- Adapted from an existing notebook — may have some issues to do with versioning!

Convolutions and neural networks

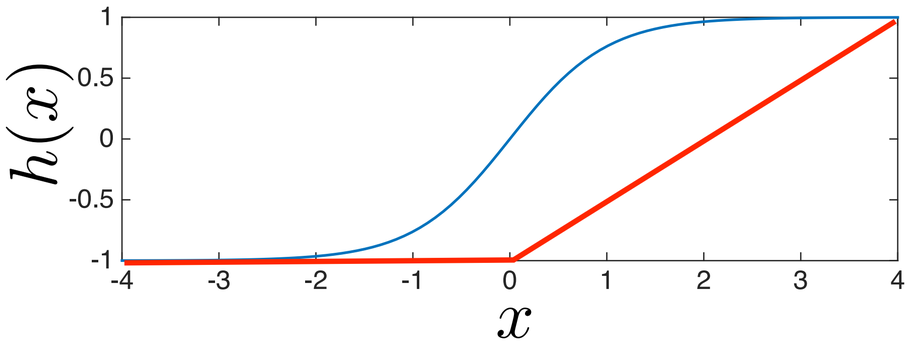
Key concepts

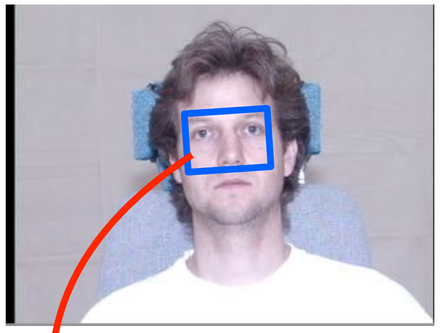
- For images (and audio, and …), wouldn’t DCT etc make sense for layers?
- Stride, padding, and dilation
- Pooling

Example in Code

```
class MLPNet(nn.Module):
    def __init__(self):
        super(MLPNet, self).__init__()

        # First fully connected layers input image is 28x28 = 784 dim.
        self.fc0 = nn.Linear(784, 256) # nparam = 784*256 = 38400
        # Two more fully connected layers
        self.fc1 = nn.Linear(256, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        # Flattens the image like structure into vectors
        x = torch.flatten(x, start_dim=1)

        # fully connected layers with activations
        x = self.fc0(x)
        x = F.relu(x)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
```

Why the Non-Linearity?





$$\mathbf{x} \in \mathbb{C}_1 \quad \gtrless 0 \quad \Leftarrow \quad [\mathbf{w}^{(2)}]^T h([\mathbf{W}^{(1)}]^T \mathbf{x}) \quad \Leftarrow$$

$$\mathbf{x} \in \mathbb{C}_2$$

Shallow vs. Deep Learning

- A shallow learner has **only one** hidden unit,

$$\mathbf{W}^{(2)}h(\mathbf{W}^{(1)}\mathbf{x})$$

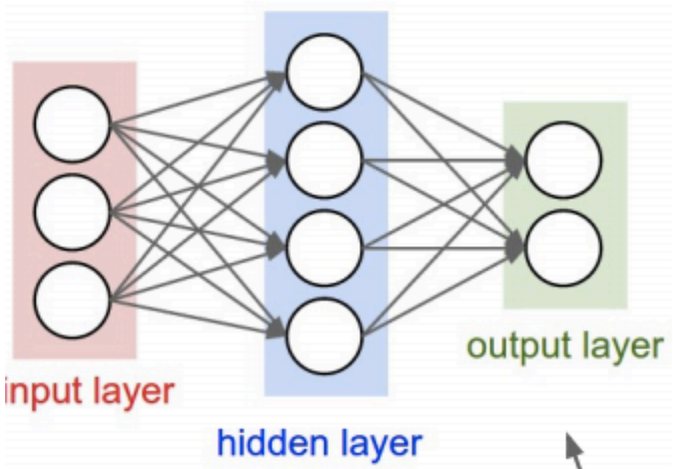
- A deep learner **has more than one** hidden unit,

$$\mathbf{W}^{(3)}h(\mathbf{W}^{(2)}h(\mathbf{W}^{(1)}\mathbf{x}))$$

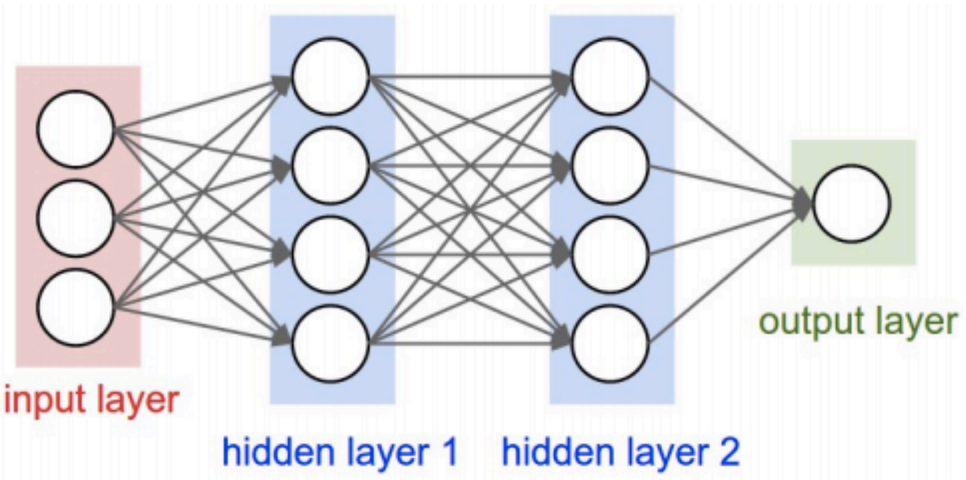
“Geoffrey Hinton”

Shallow vs. Deep Layers

“shallow”



“deep”



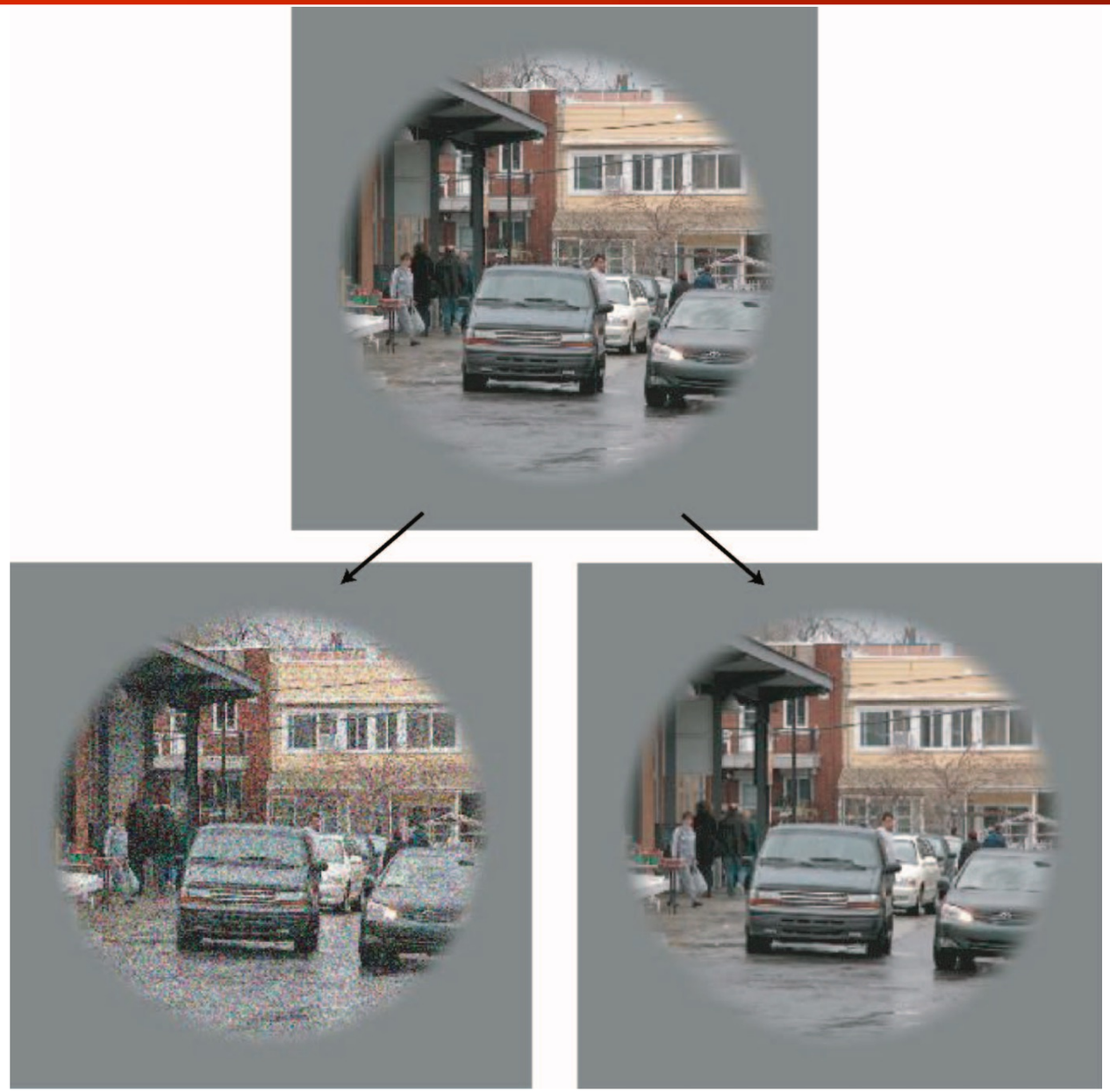
“2-layer neural net,” or
“1-hidden-layer neural net”

“Fully-connected” layers

“3-layer neural net,” or
“2-hidden-layer neural net”

Why deepness?

Question?



a)

c)

e)

b)

d)

f)

We want two properties here

- Invariance: $\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{f}[\mathbf{x}]$. (to translation, rotation, flipping, warping)
- Equivariance: $\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{t}\big[\mathbf{f}[\mathbf{x}]\big]$.

(UDL)

Translation invariance: The classification decision for each image is independent of the position of the animal on the image. A cat is a cat irrespective of whether it appears at the top or at the bottom of an image.

Locality: The classification decision does not really depend on a pixel that is far away from the animal on the image. A cat is still a cat irrespective of whether far away pixels correspond to a building or a tree.

(MEML)

We want two properties here

And convolutions have them!

Translation invariance: The classification decision for each image is independent of the position of the animal on the image. A cat is a cat irrespective of whether it appears at the top or at the bottom of an image.

Locality: The classification decision does not really depend on a pixel that is far away from the animal on the image. A cat is still a cat irrespective of whether far away pixels correspond to a building or a tree.

(MEML)

Linearity: For any two input signals $x_1(t)$ and $x_2(t)$ and scalars α_1 and α_2,

$$\mathcal{L}\left(\alpha_1 x_1 + \alpha_2 x_2\right) = \alpha_1 \mathcal{L}\left(x_1\right) + \alpha_2 \mathcal{L}\left(x_2\right).$$

Time invariance: When the shifted (delayed by τ) signal $\tilde{x}(t) = x(t-\tau)$ is given as an input, then the corresponding output signal $\tilde{y} = \mathcal{L}\left(\tilde{x}\right)$ is $\tilde{y}(t) = y(t-\tau)$, where $y = \mathcal{L}\left(x\right)$. Namely, the output of the shifted input is the the shifted output of the original input.

Convolutions and neural networks

Why

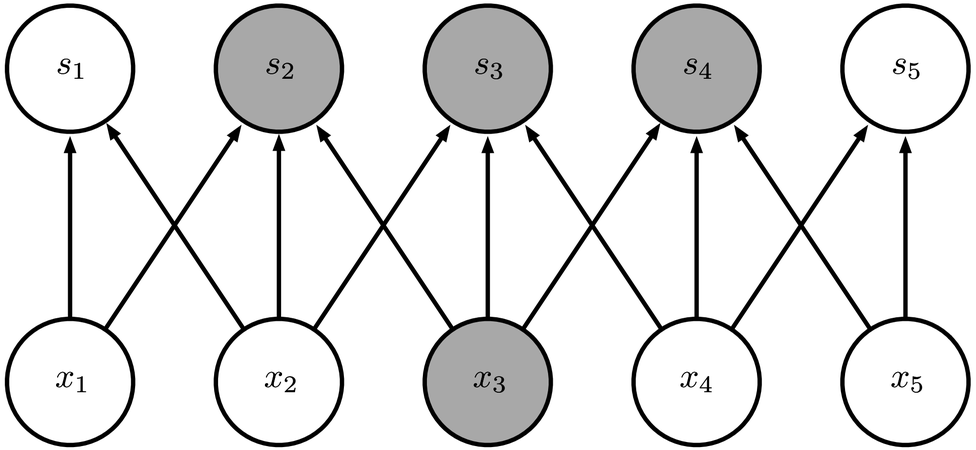
- Consider fully-connected network for speech classification: 2min of 8kHz audio, hidden layers with 10k, 1k neurons

$$\hat{f}(x) = \operatorname{sign}(2\sigma_{\text{sigmoid}}(W_3\sigma_{\text{relu}}(W_2\sigma_{\text{relu}}(W_1 x + b_1) + b_2) + b_3) - 1).$$

- Only need to use local data
- (Even worse for images!)

Sparse Connectivity

Sparse connections due to small convolution kernel



Dense connections

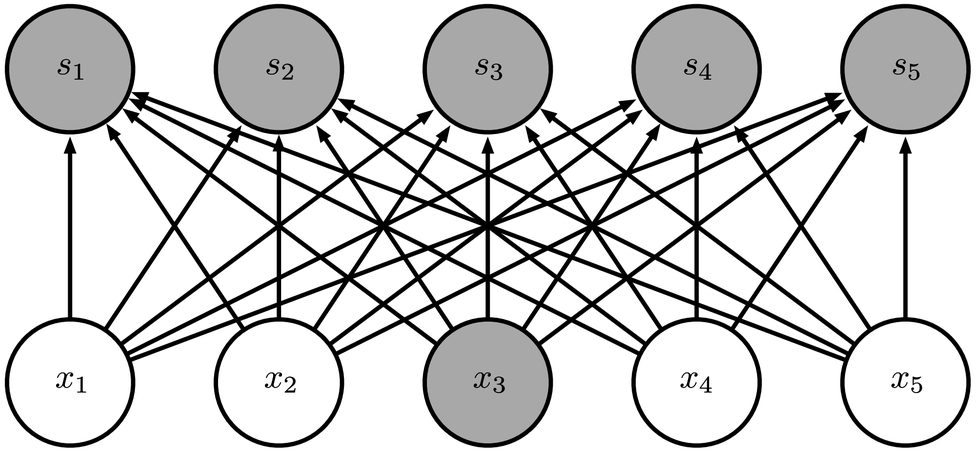
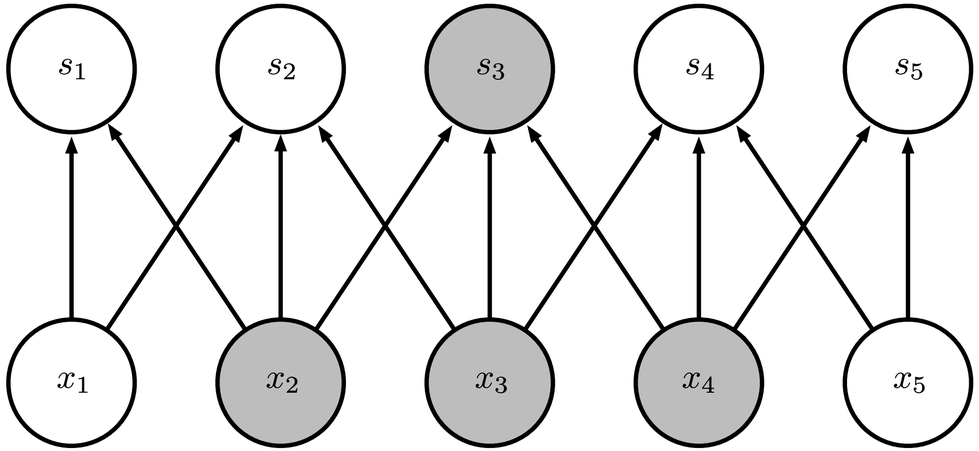


Figure 9.2

Sparse Connectivity

Sparse connections due to small convolution kernel



Dense connections

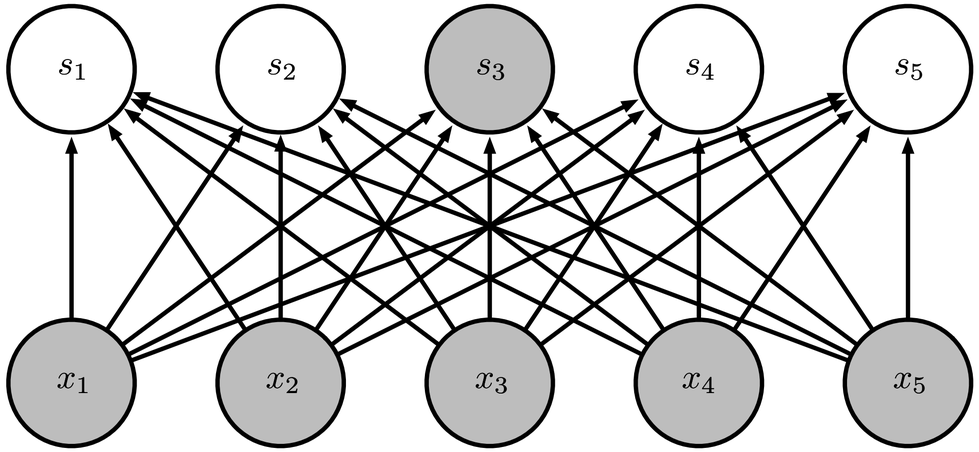
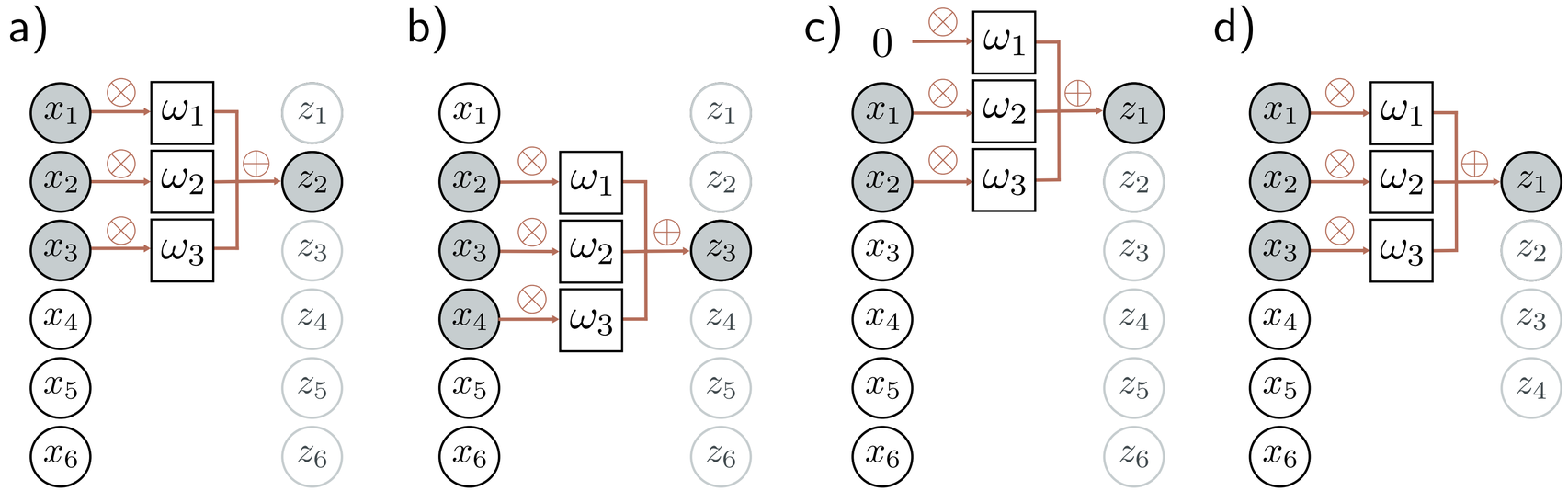


Figure 9.3

1D convolution

$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1},$$

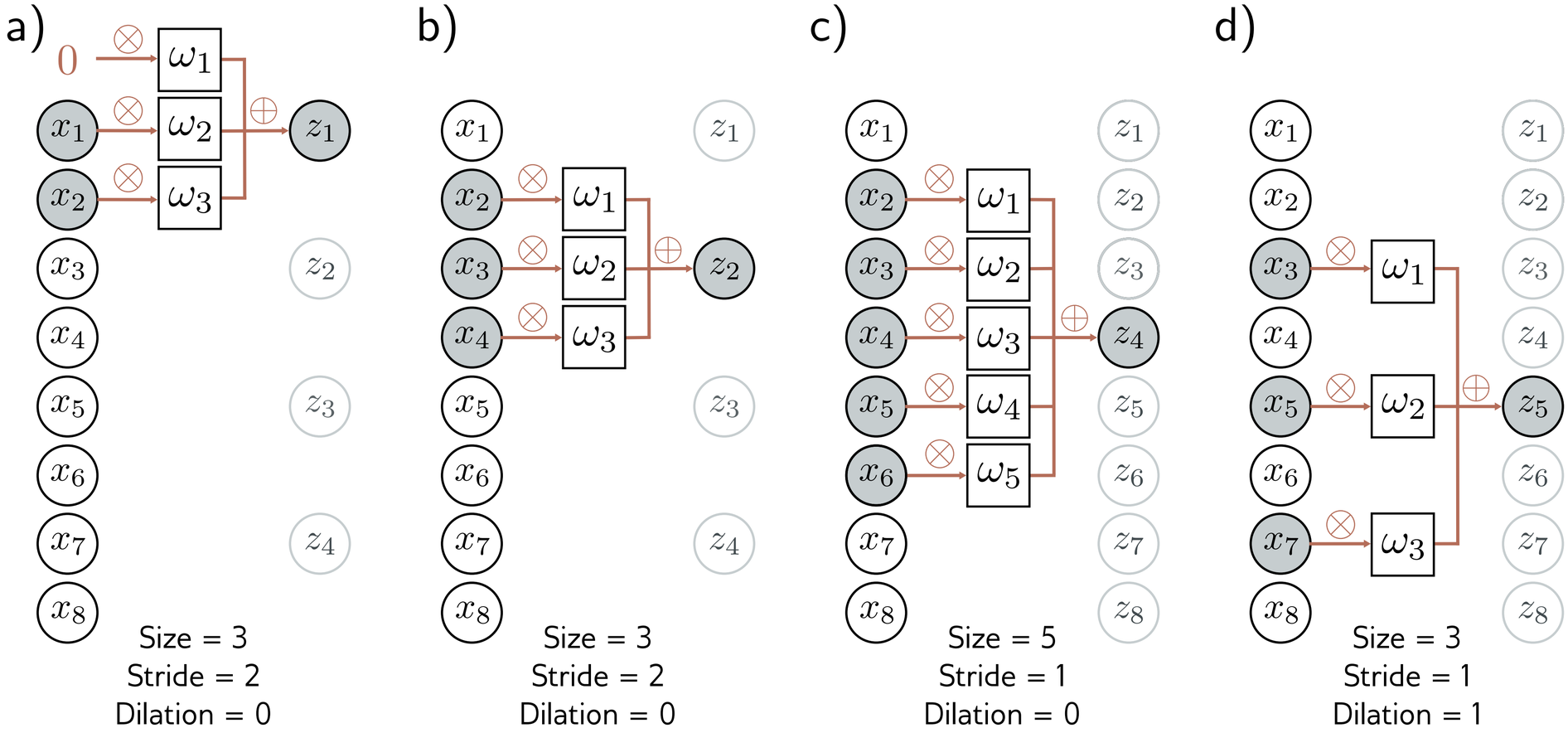
- Equivariant to translation!



1D convolution

Stride, kernel size, dilation

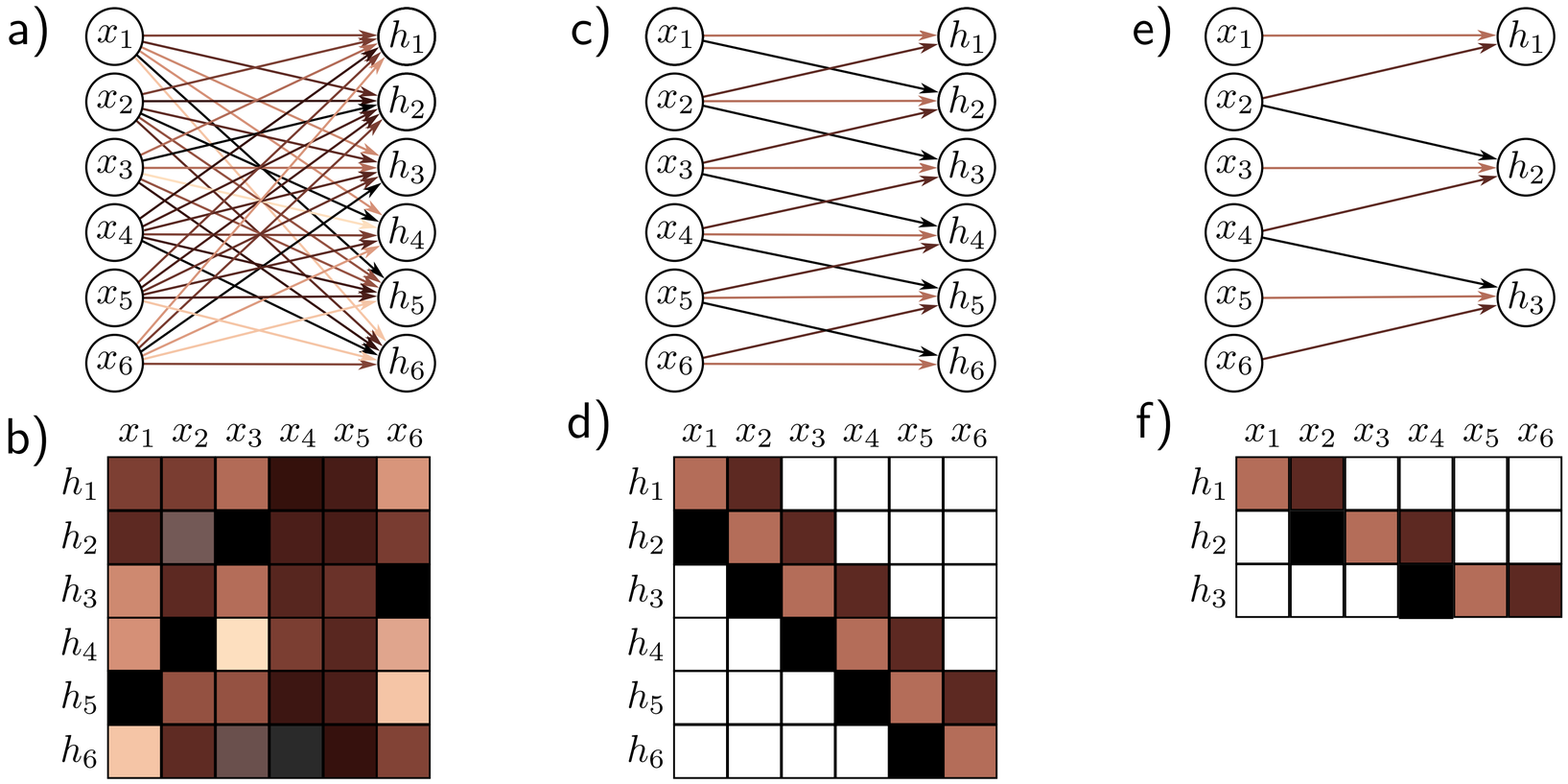
$$h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right].$$



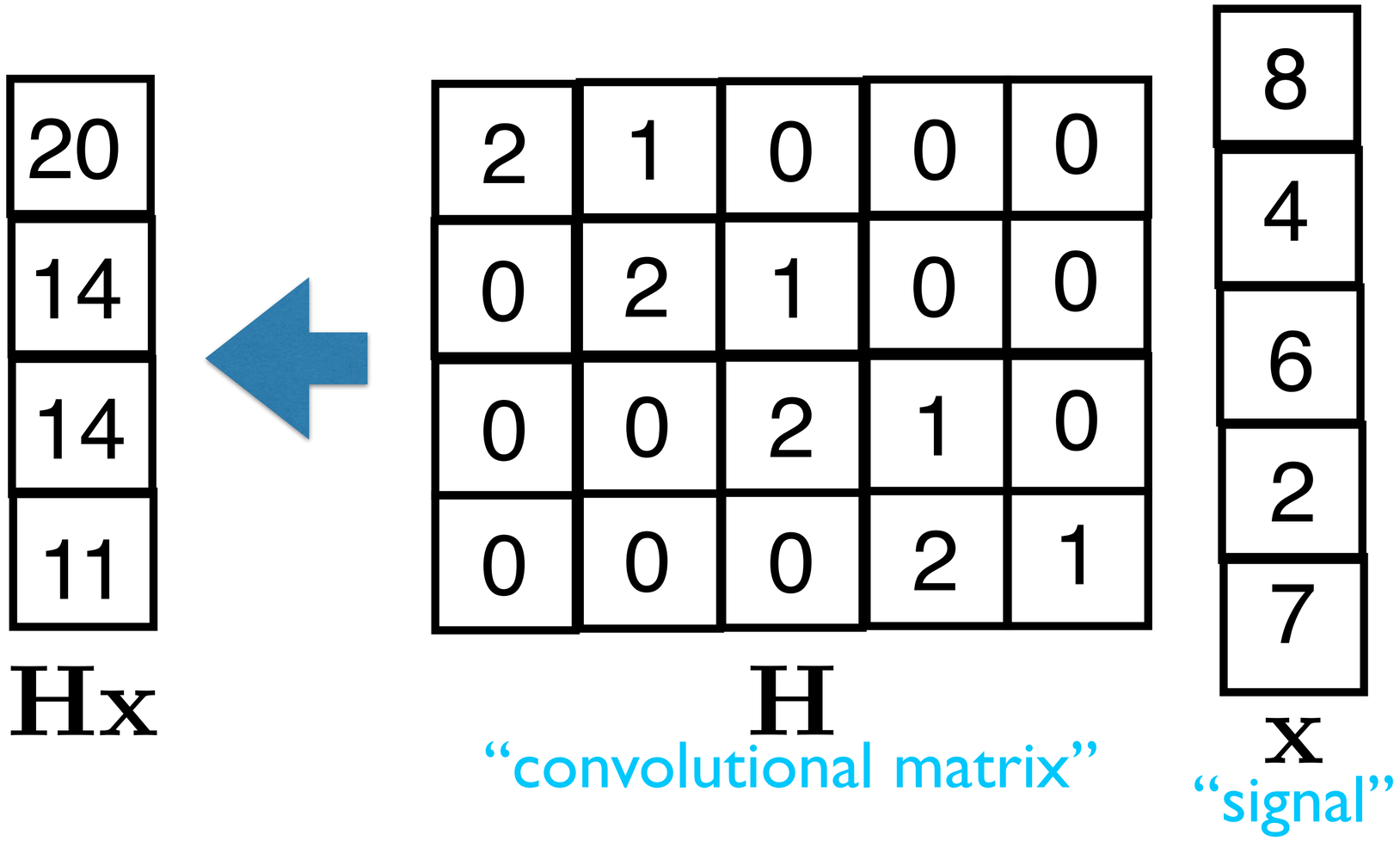
1D convolution

Dense vs convolutional layers

$$h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right].$$

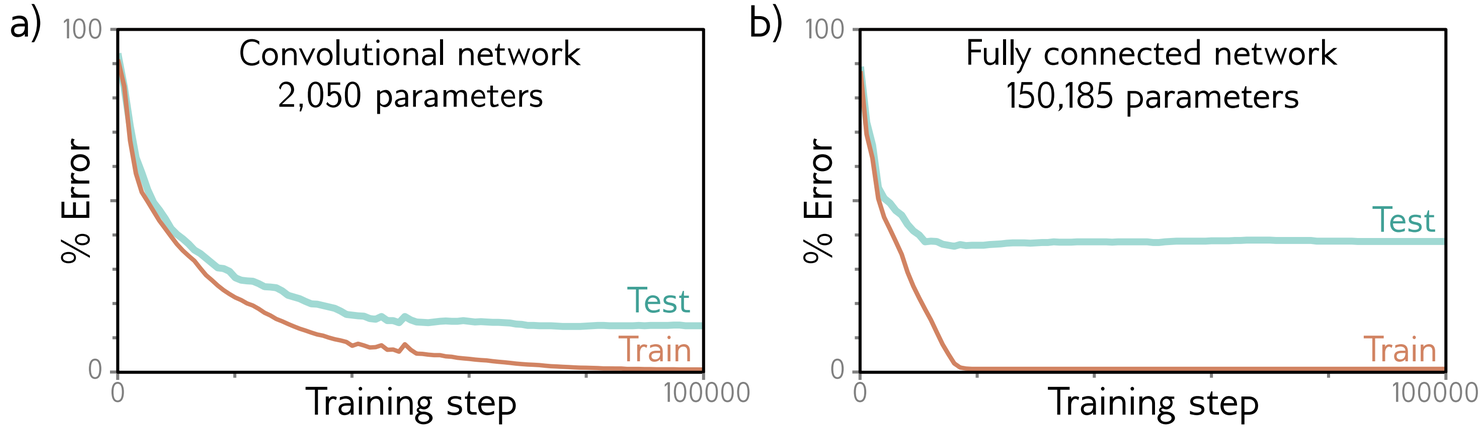


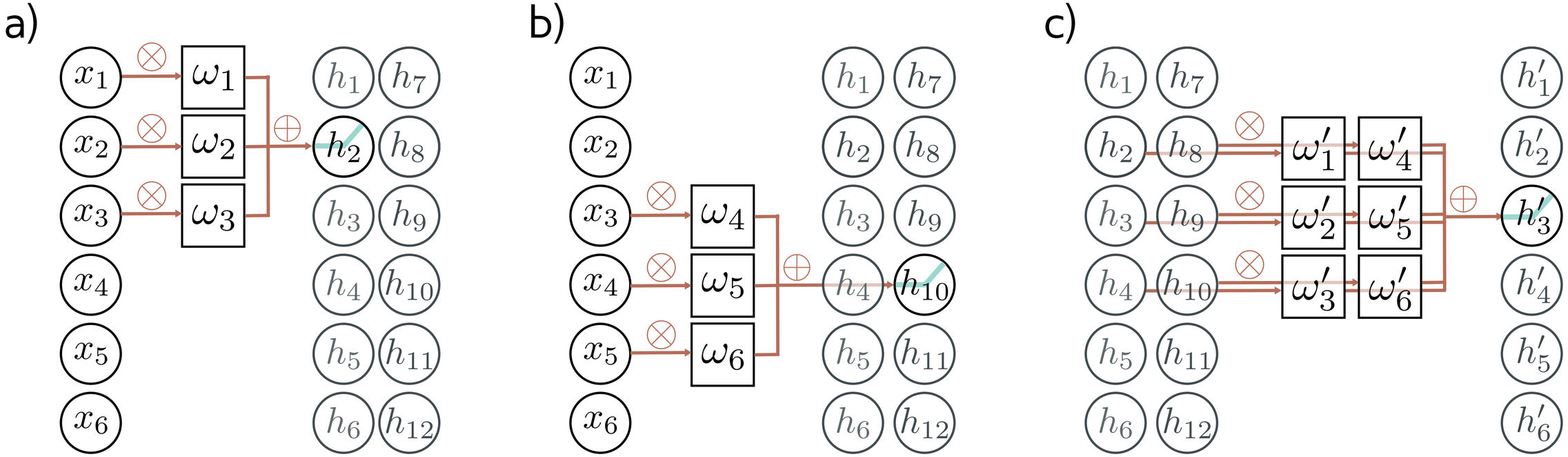
Reminder: Convolution



1D convolution

Channels





2D Convolution

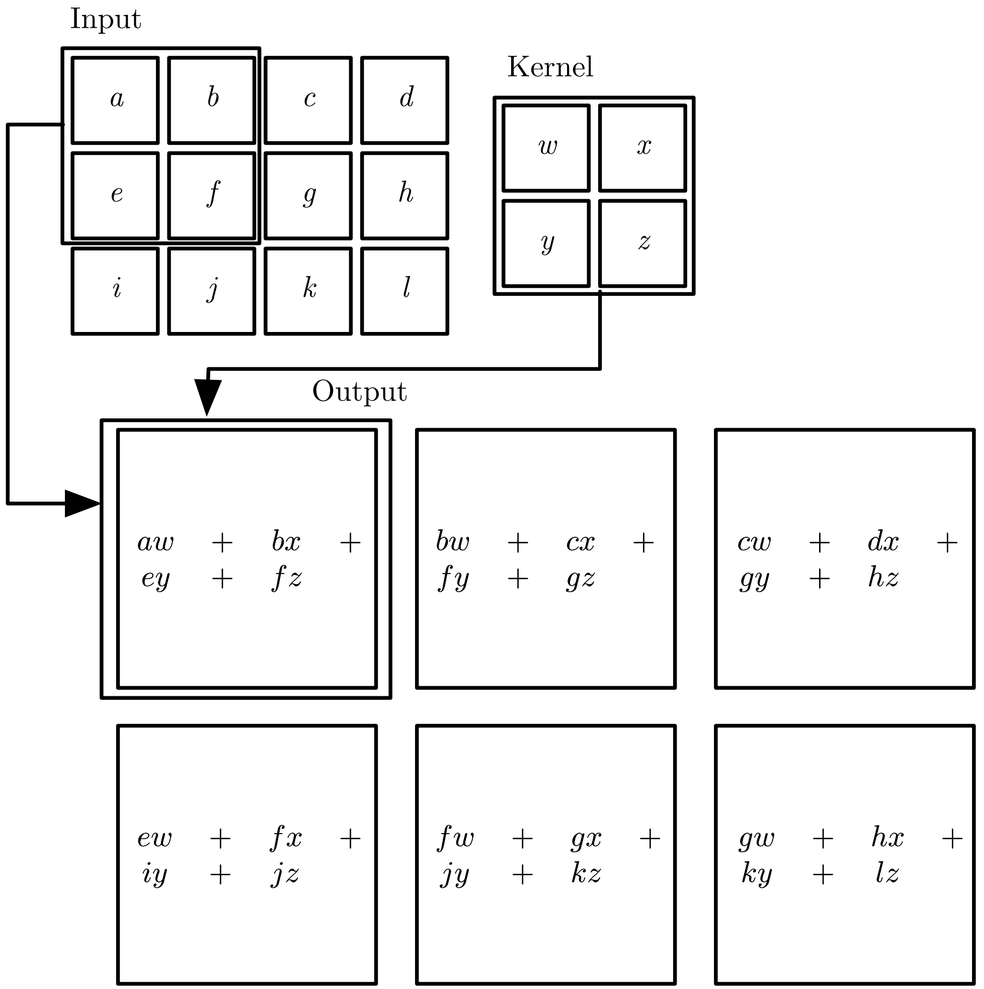


Figure 9.1

(Goodfellow 2016)

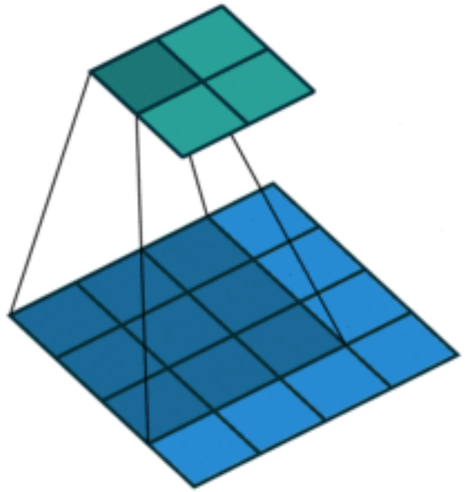
Convolutions in neural networks

Key concepts

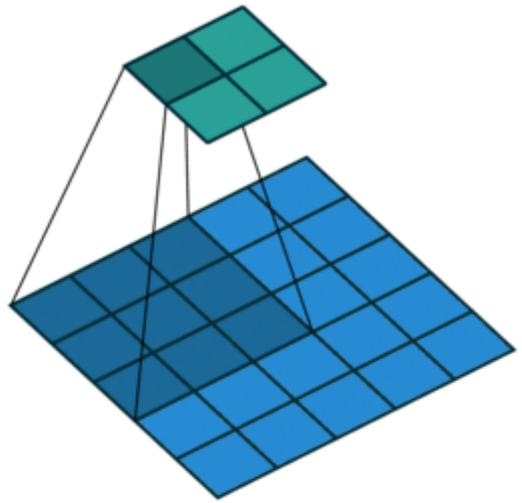
- Strides
- Padding
- Dilation
- Pooling

Convolutions in neural networks

Strides



Stride length = 1

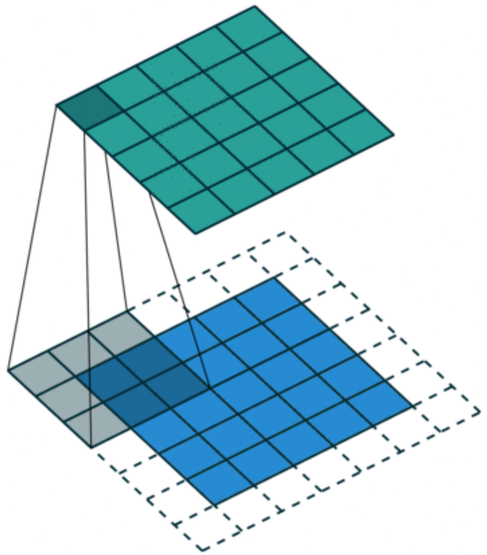


Stride length = 2

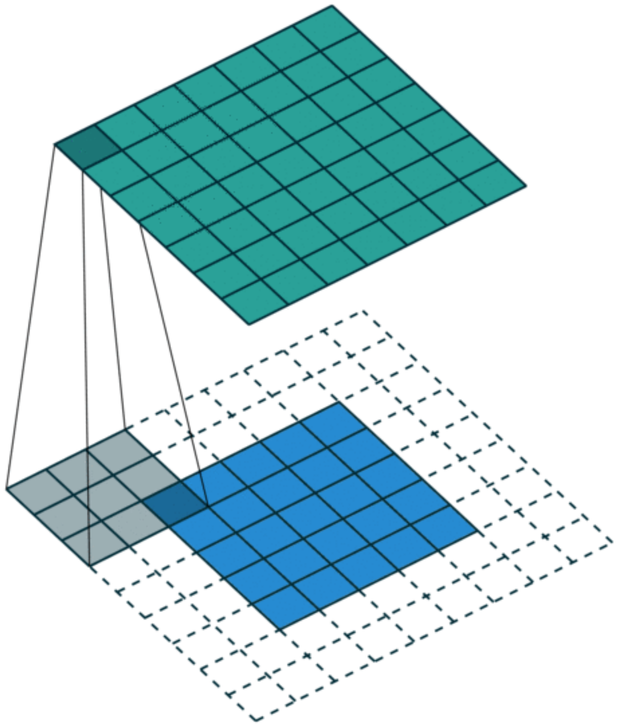
https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Padding



Padding length = 1

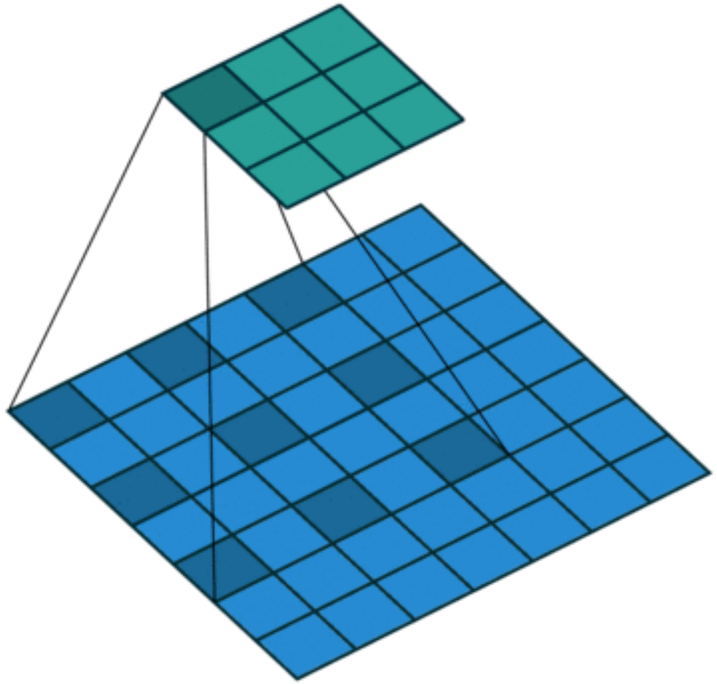


Padding length = 2

https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Dilation



Dilation factor = 2

https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Pooling

What do we do to get each pixel in the output?

- Max pooling
- Average pooling
- L2 norm (or your favourite)
- Weighted (e.g., Gaussian)

https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Pooling

0	1	2
2	2	0
0	1	2

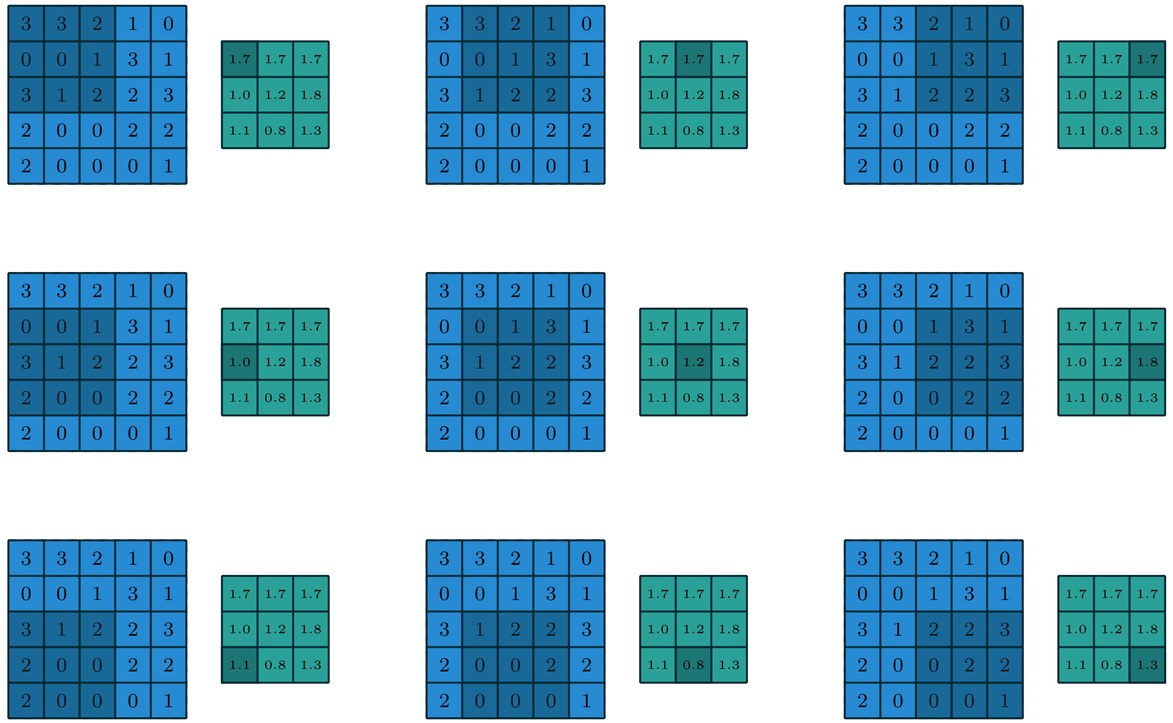
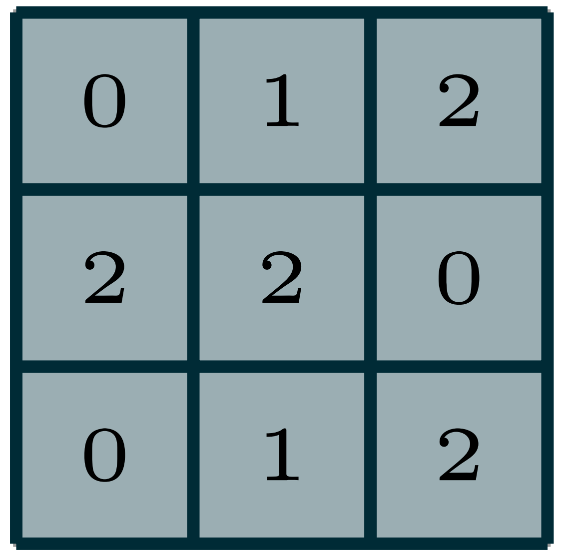


Figure 1.5: Computing the output values of a 3×3 average pooling operation on a 5×5 input using 1×1 strides.

https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Pooling



https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Pooling

0	1	2
2	2	0
0	1	2

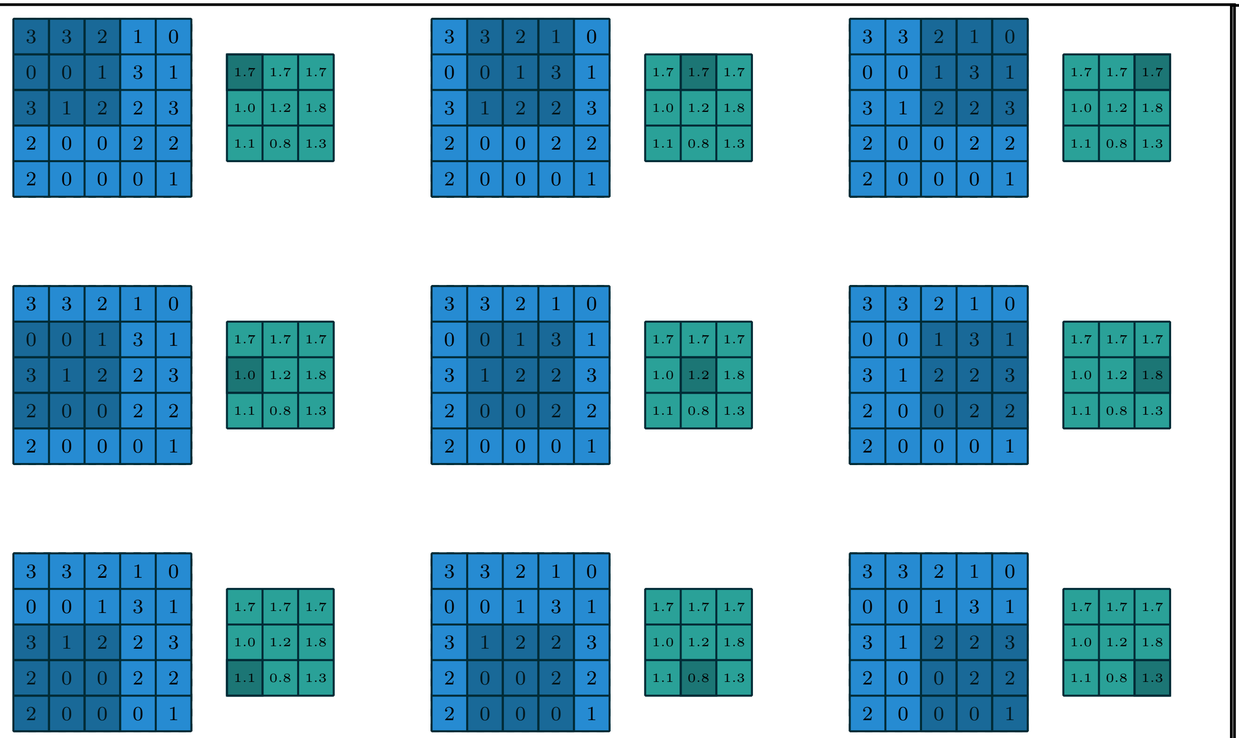


Figure 1.5: Computing the output values of a 3×3 average pooling operation on a 5×5 input using 1×1 strides.

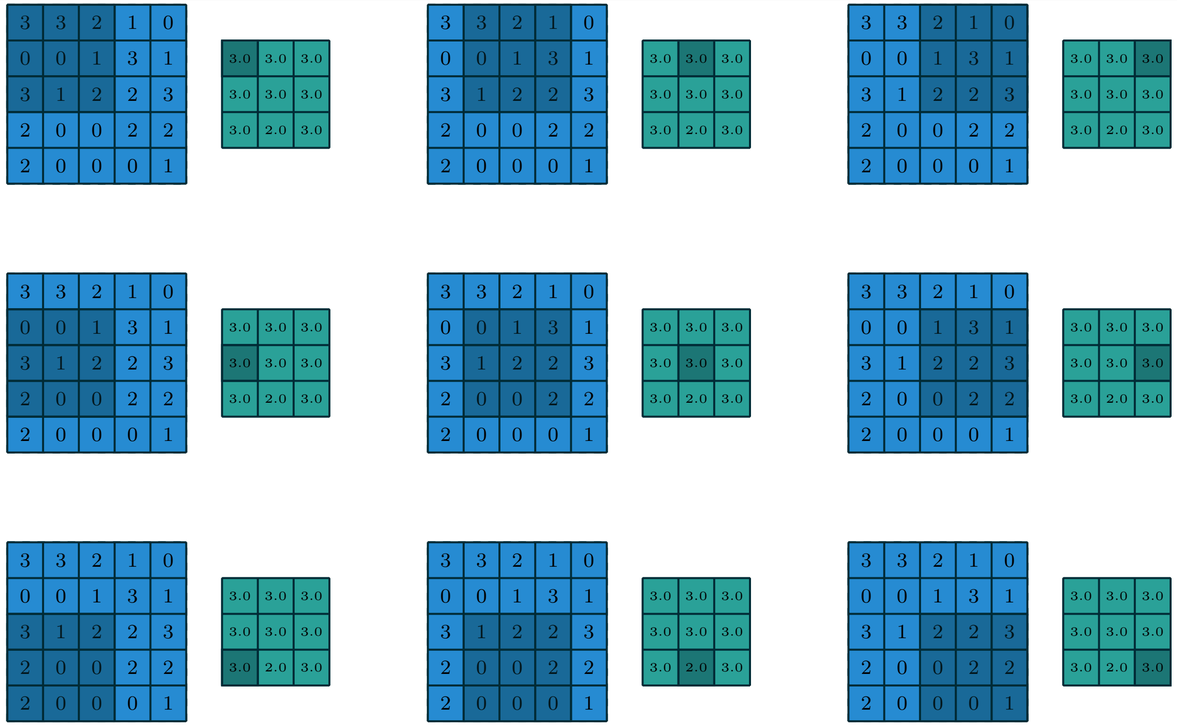
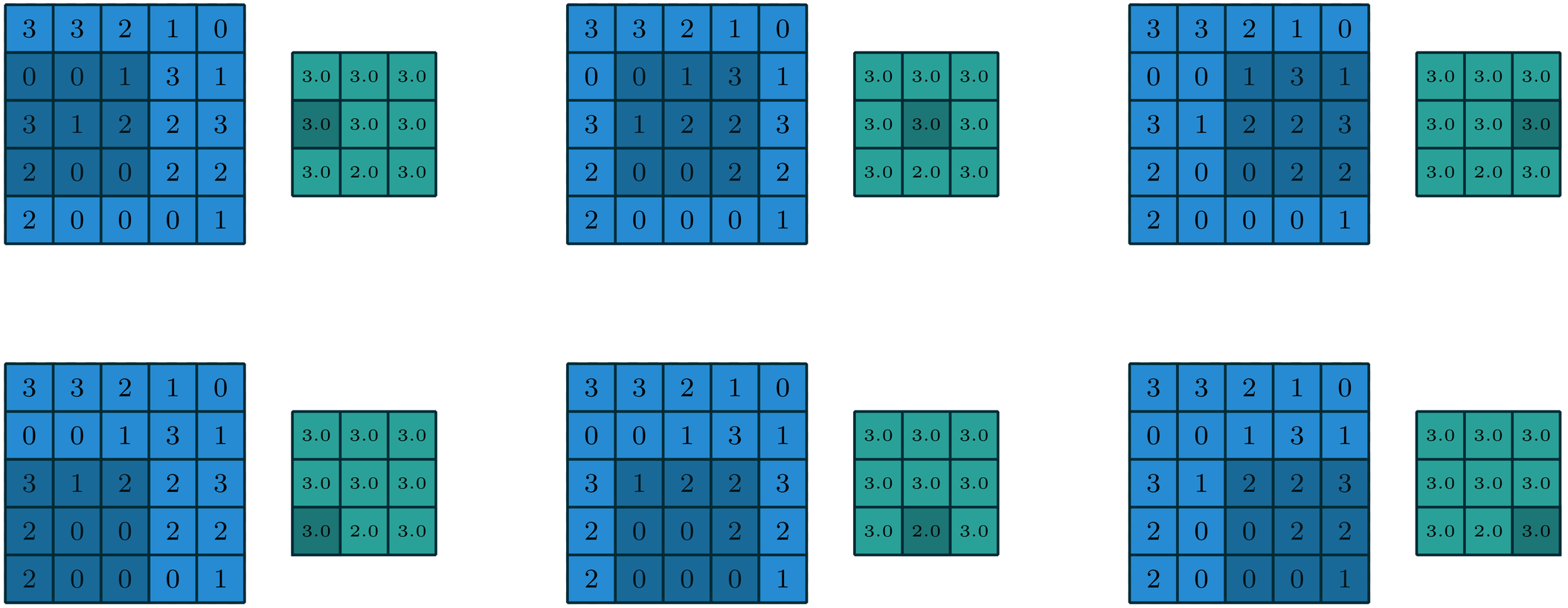


Figure 1.6: Computing the output values of a 3×3 max pooling operation on a 5×5 input using 1×1 strides.

https://arxiv.org/abs/1603.07285

Convolutions in neural networks

Pooling



https://arxiv.org/abs/1603.07285

Convolutional Network Components

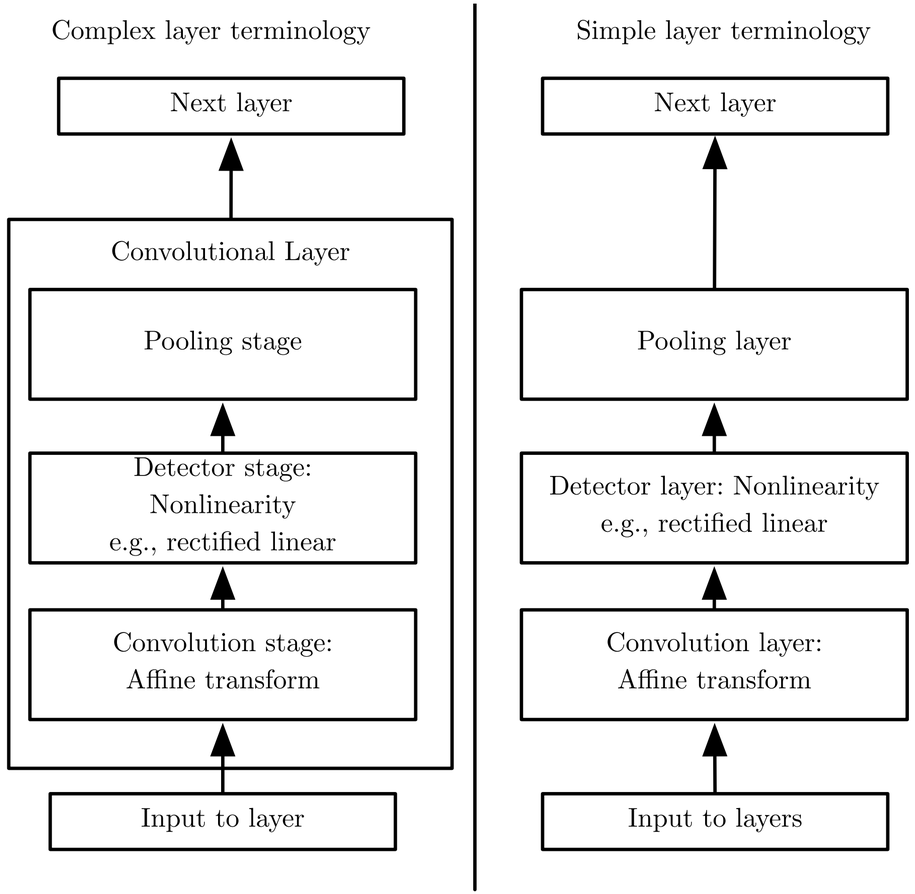


Figure 9.7

Example Classification Architectures

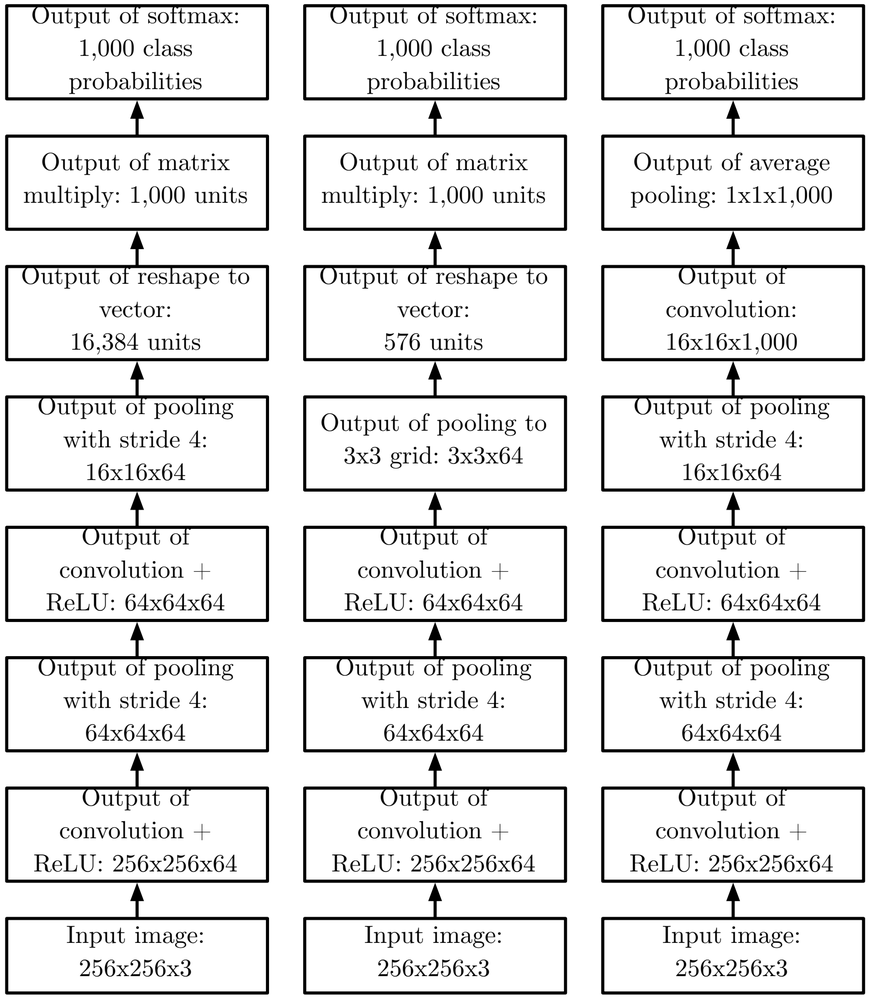


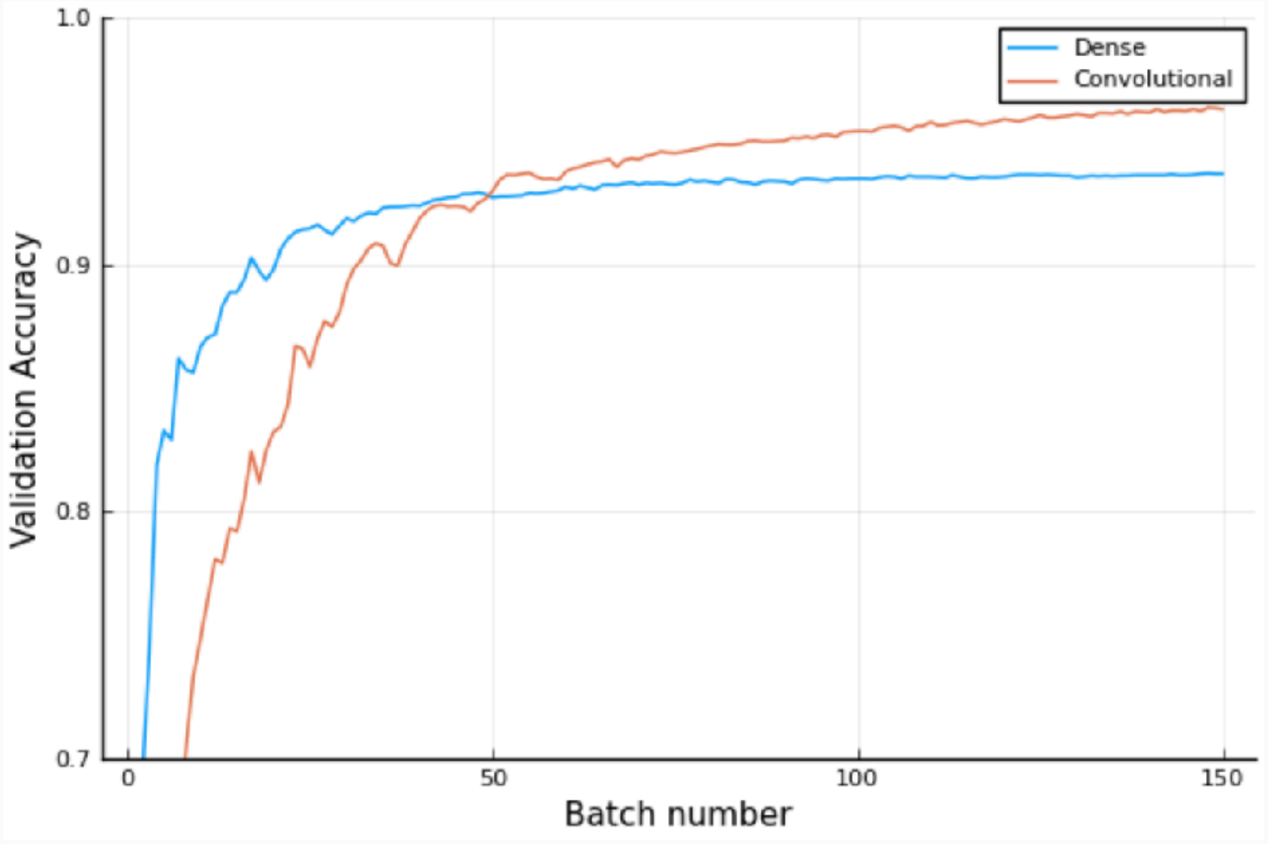
Figure 9.11

It works!

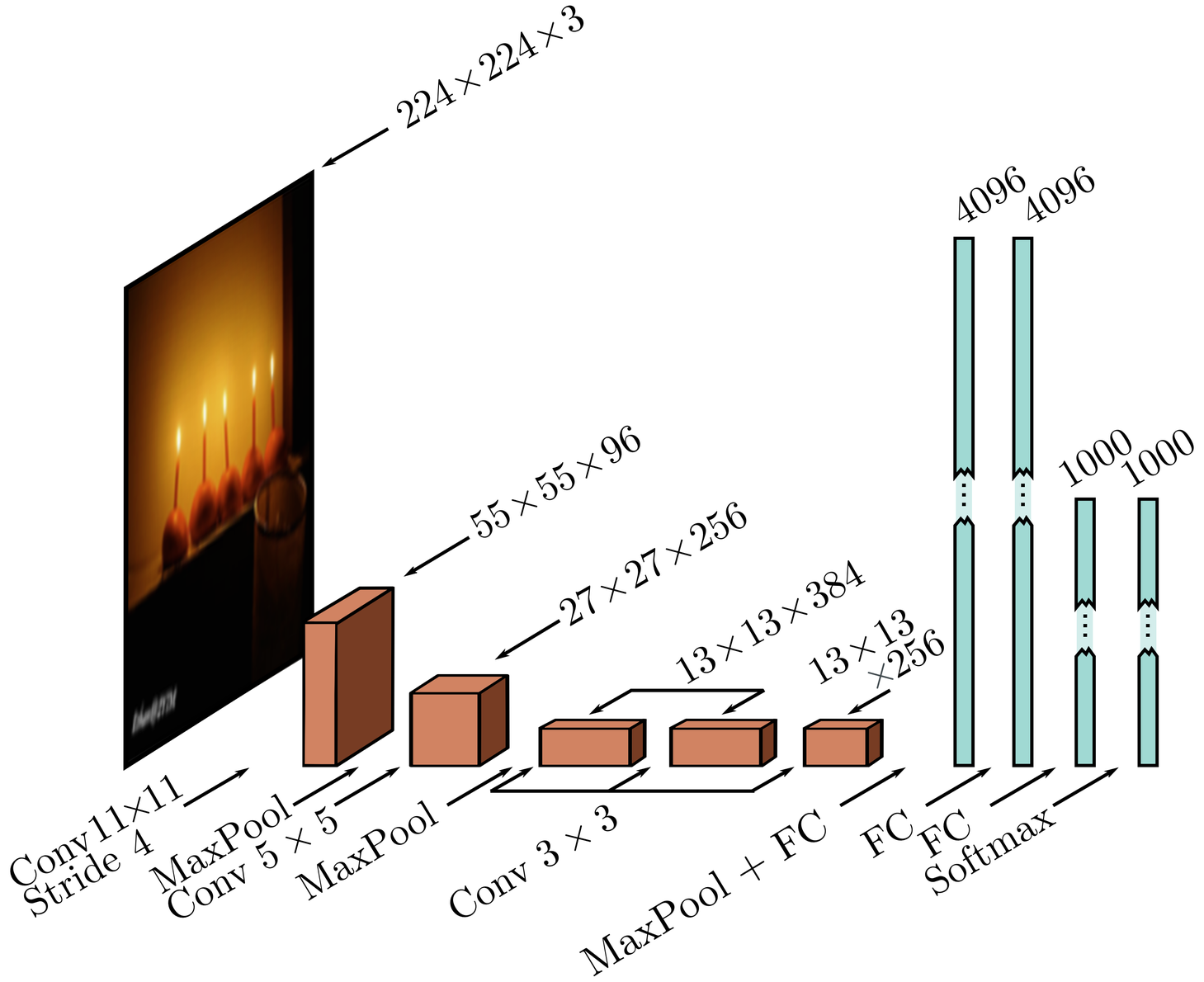
From *MEML*, Ch 5

```
#Julia code
using Flux
model1 = Chain(
  flatten,
  Dense(784, 200, relu),
  Dense(200, 100, tanh),
  Dense(100, 10),
  softmax)

model2 = Chain(
  Conv((5, 5),1=>8,relu,stride=(1,1),pad=(0,0)),
  MaxPool((2,2)),
  Conv((3, 3),8=>16,relu,stride=(1,1),pad=(0,0)),
  MaxPool((2,2)),
  flatten,
  Dense(400, 10),
  softmax)
```

Here is a plot of the validation accuracy for both `model1` and `model2` when trained on $5,000$ MNIST images (and validated on an additional $5,000$). This is with the ADAM optimizer with a learning rate of 0.005 and batch sizes of $1,000$.



AlexNet


224×224×3
55×55×96
27×27×256
13×13×384
13×13
×256
4096
4096
1000
1000
Conv11×11
Stride 4
MaxPool
Conv 5 × 5
MaxPool
Conv 3 × 3
MaxPool + FC
FC
FC
Softmax

AlexNet in PyTorch

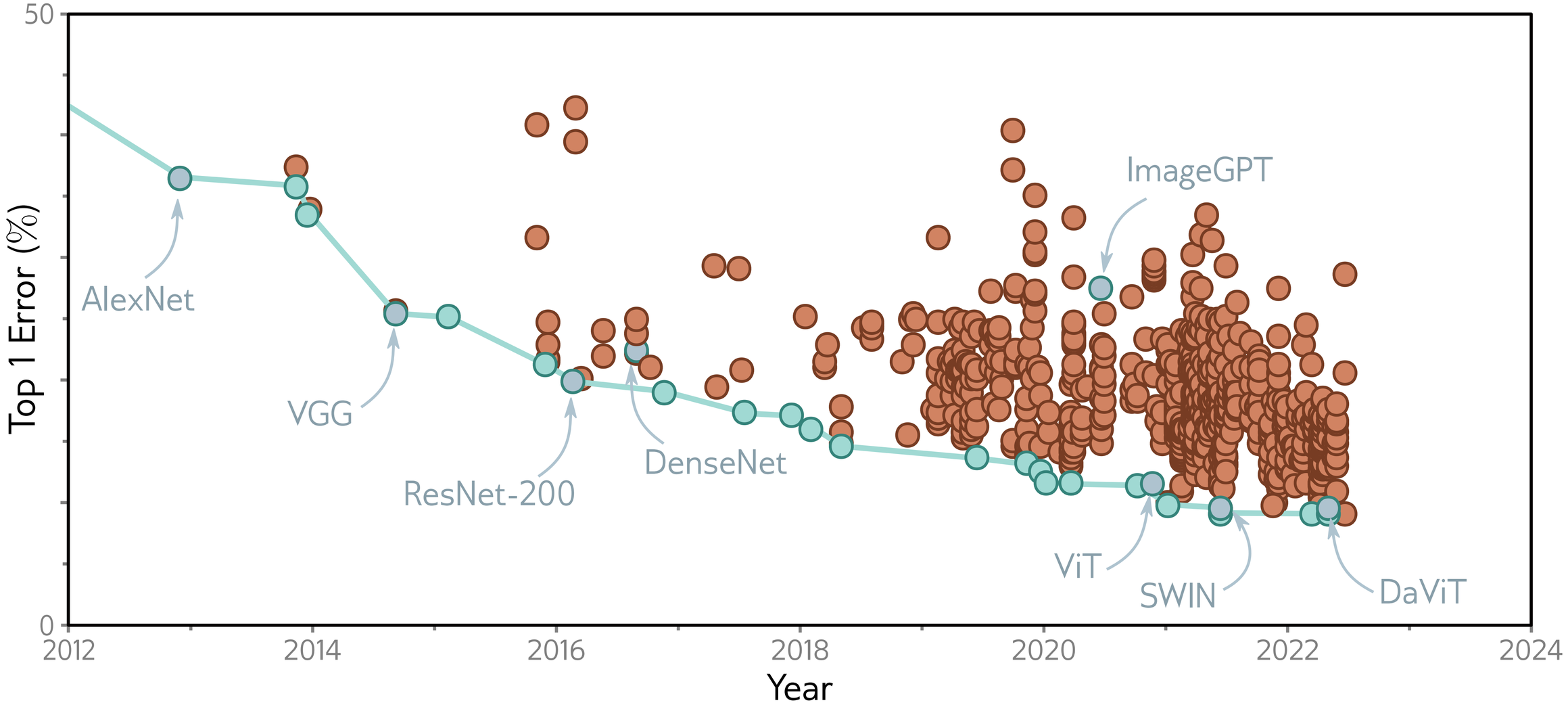
```
>>> from torchvision import models
>>> net = models.alexnet()
```

AlexNet in PyTorch

```
>>> net
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace)
    (3): Dropout(p=0.5)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

AlexNet

And beyond



Your turn!

CNNs in PyTorch

- Modified MNIST notebook: https://github.com/AdelaideUniversityMathSciences/MathsForAI/blob/main/Code/LeNetClassificationExcercise_2021.ipynb
- *Understanding Deep Learning* version: https://github.com/udlbook/udlbook/blob/main/Notebooks/Chap10/10_5_Convolution_For_MNIST.ipynb
- Do the TODO's in each

Mathematics of AI

Emerging themes at end of Module 2

- “Find sparsity”: sparse data representations help generalise
- It’s optimisation all the way down
- Regularisation <-> optimisation <-> sparsity (e.g., compressive sensing)
- Convolutions: the “most useful trick in applied probability”
  Fourier transforms ——> CNNs
- NNs == matrix multiplication + non-linearity; CNNs == matrix multiplication **with structure** + non-linearity