Deep Neural Networks Survey

Survey

- Chapter 11 – Training Deep Neural Networks
- Chapter 13 – Encoding Categorical Features
- Chapter 14 – Deep Computer Vision using CNNs
- Chapter 15 – Processing Sequences using RNNs
- Chapter 17 – Representation Learning and Generative Learning

Training Deep Neural Networks

Chapter 11

Vanishing & Exploding Gradients

- Vanishing Gradients
  - Gradient updates for each step cause the gradients to become smaller and smaller towards the lower layers
  - Training never converges
- Exploding Gradients
  - The gradients become larger and larger
  - Algorithm diverges

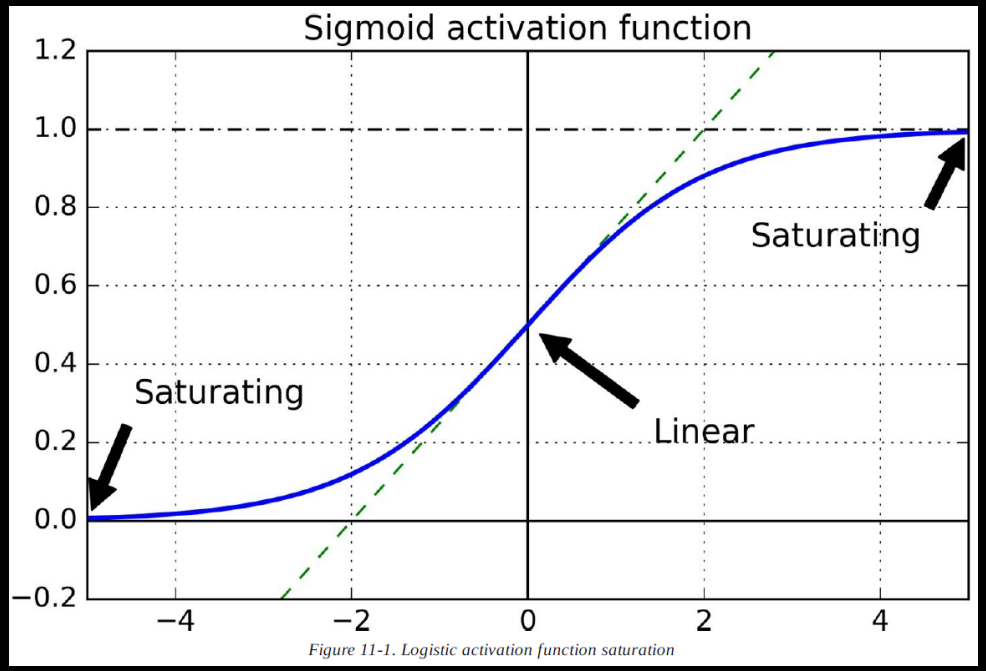


Figure 11-1. Logistic activation function saturation

Initialization

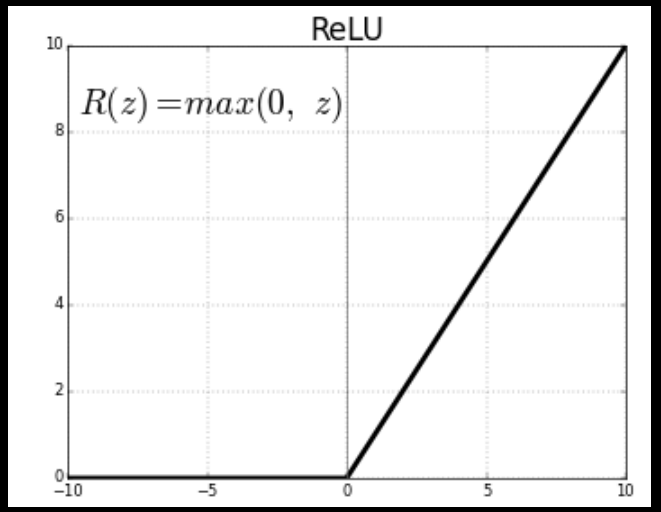
- To avoid vanishing/exploding gradients, the variance of the outputs of each layer should be equal to the variance of its inputs
  - It's not possible unless the layer has equal number of input & output connections
  - Good compromise is to initialize the weights randomly (Xavier/Glorot initialization strategy)

Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

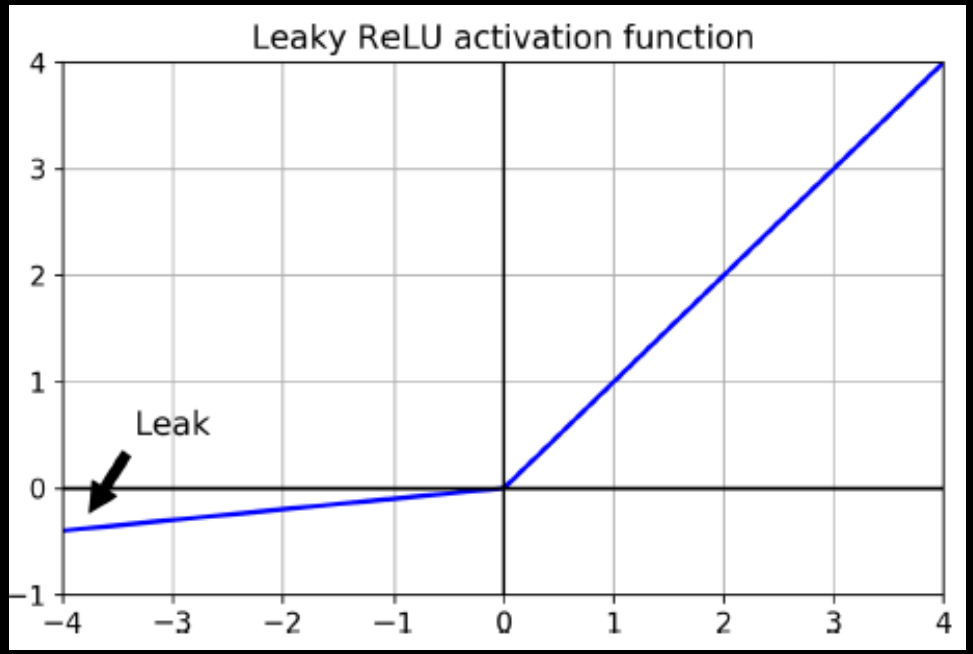
Activation Functions

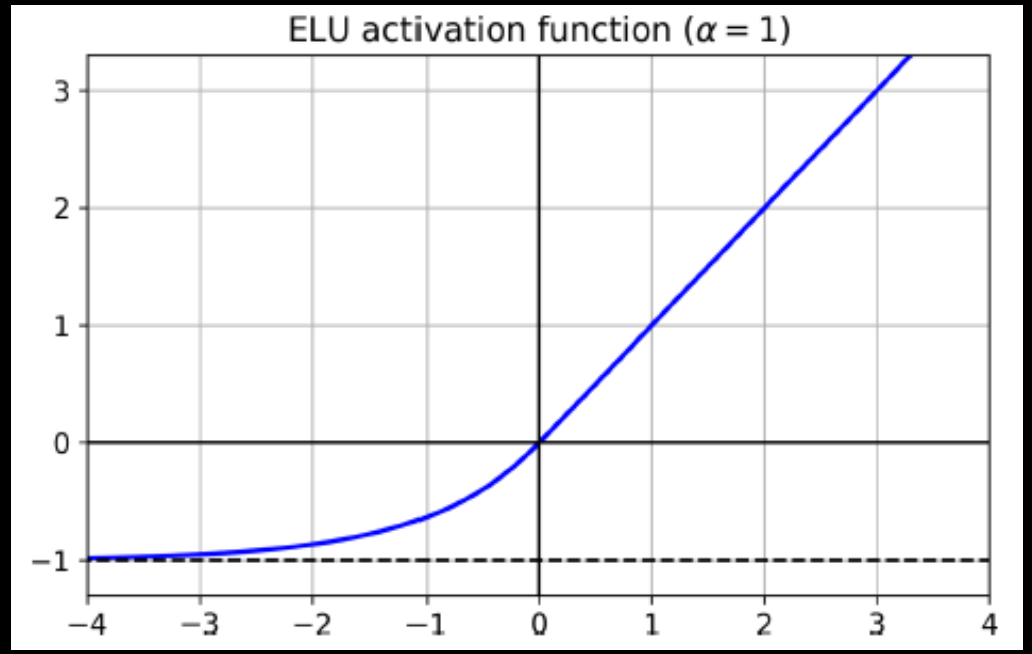
- Other activation functions than sigmoid function
- ReLU (Rectified Linear Unit)
  - Does not saturate for positive values
  - Fast to compute
  - Some neurons die if the inputs are negative



Activation Functions (continued)

- Leaky ReLU
- ELU (Exponential Linear Unit)





Batch Normalization

- Normalize the input into the activation function of each layer
- Considerably improves vanishing gradient problem
- Acts like a regularizer as well
- Model becomes more complex (slower)

1. $$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x^{(i)}$$
2. $$\sigma_B{}^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$$
3. $$\boldsymbol{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$$
4. $$z^{(i)} = \gamma \boldsymbol{x}^{(i)} + \beta$$

Gradient Clipping

- Clips the gradients during backpropagation

Transfer Learning

- Reuses pre-trained layers
  - Freeze lower layers
  - Tweak/drop/replace upper layers
- Works well if the base and target data sets have similar low-level features

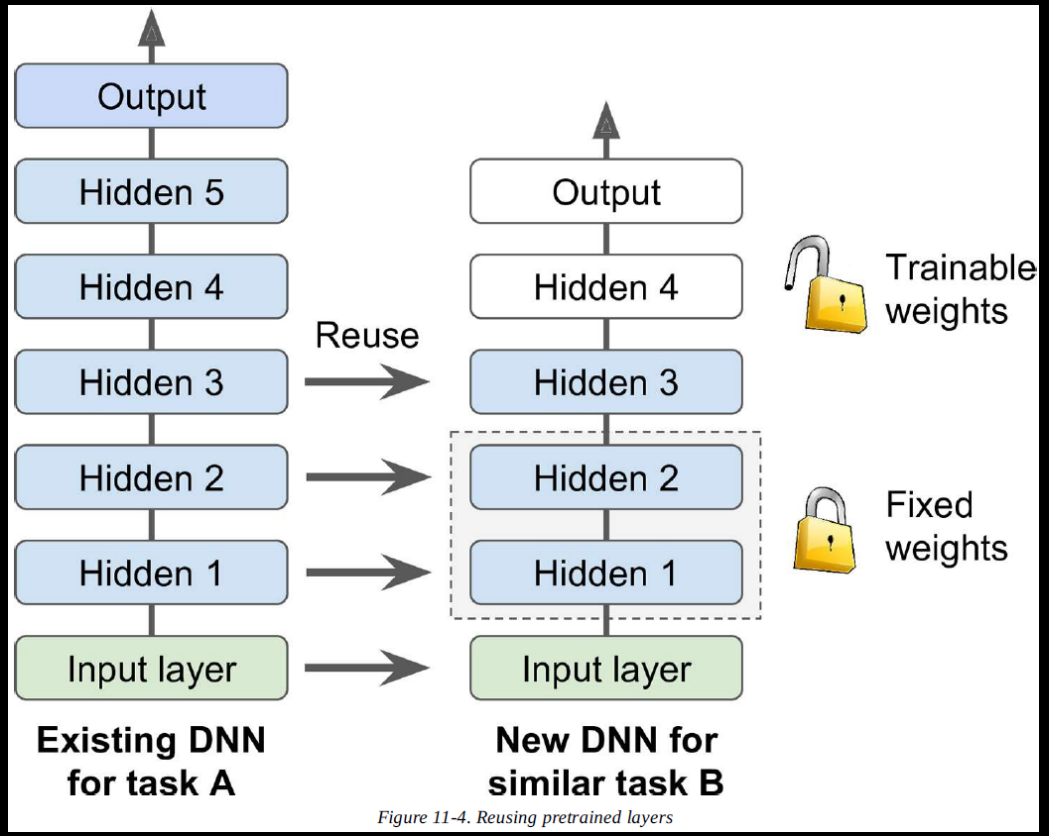


Figure 11-4. Reusing pretrained layers

Faster Optimizer

- To speed up training:
  - Good initialization strategy
  - Good activation function
  - Batch Normalization
  - Pretrained network

Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

```
1 import tensorflow as tf
2 import numpy as np
3 x = np.array([-10, 7, 5, -33])
4 y = np.array([10.3, 68.7, 61.8, -68.7], dtype=float)
5
6 model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])
7 model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(0.1))
8 history = model.fit(x, y, epochs=2000, verbose=False)
```

Gradient Descent (continued)

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

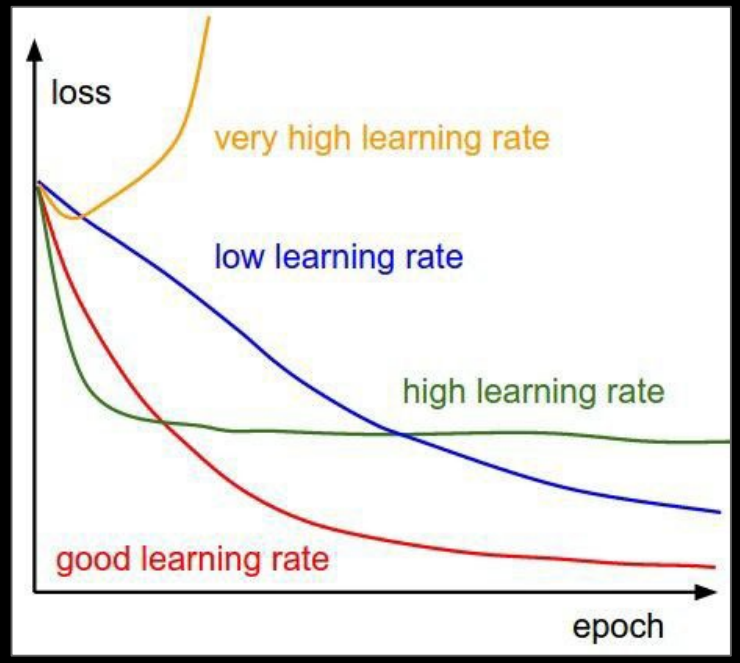
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3 or 5e-4
is a great starting point for many models!

Learning Rate

- Finding good learning rate is not easy



Learning Rate (continued)

- Predetermined piecewise constant learning rate
  - Set learning rate to $\eta = 0.1$ initially, then to $\eta = 0.001$ after 50 epochs
- Performance scheduling
  - Measure validation error every M steps, then reduce the learning rate by a factor of λ after the error stops dropping
- Power scheduling
  - Set learning rate as a function of the iteration number t, the initial learning rate η_0, the power c and the steps s
  - $\eta(t) = \eta_0/(1 + t/s)^c$
- Exponential scheduling:
  - Set learning rate to $\eta(t) = \eta_0 \times 0.1^{t/s}$

Regularization

- To avoid overfitting
  - Early Stopping
  - ℓ_1, ℓ_2 Regularization
  - Dropout
  - Max-Norm Regularization
  - Data Augmentation

Regularization (continued)

- Early stopping
  - Stop the training when the performance on the validation error starts increasing

Regularization (continued)

- Dropout
- Very good regularization technique
- At every training step, every neuron can be dropped according to a hyperparameter *p* (*dropout probability*)

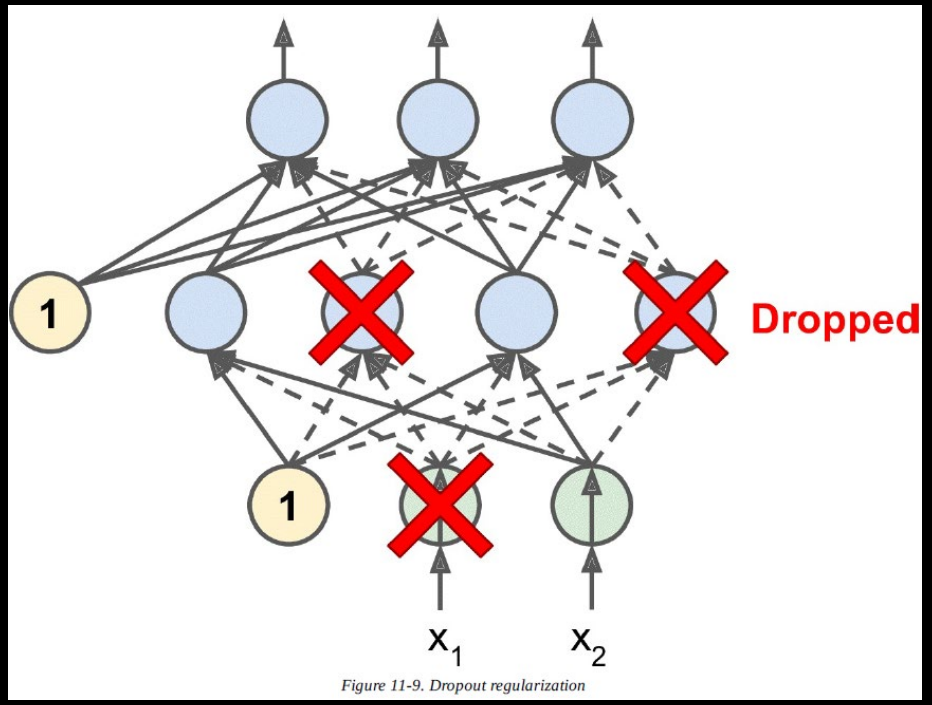


Figure 11-9. Dropout regularization

Regularization (continued)

- Max-Norm Regularization
  - Constrain the weights $\boldsymbol{w}$ such that $\|\boldsymbol{w}\|_2 \leq r$ where *r* is a hyperparameter

Regularization (continued)

- Data Augmentation
  - Generate new training examples from the existing ones

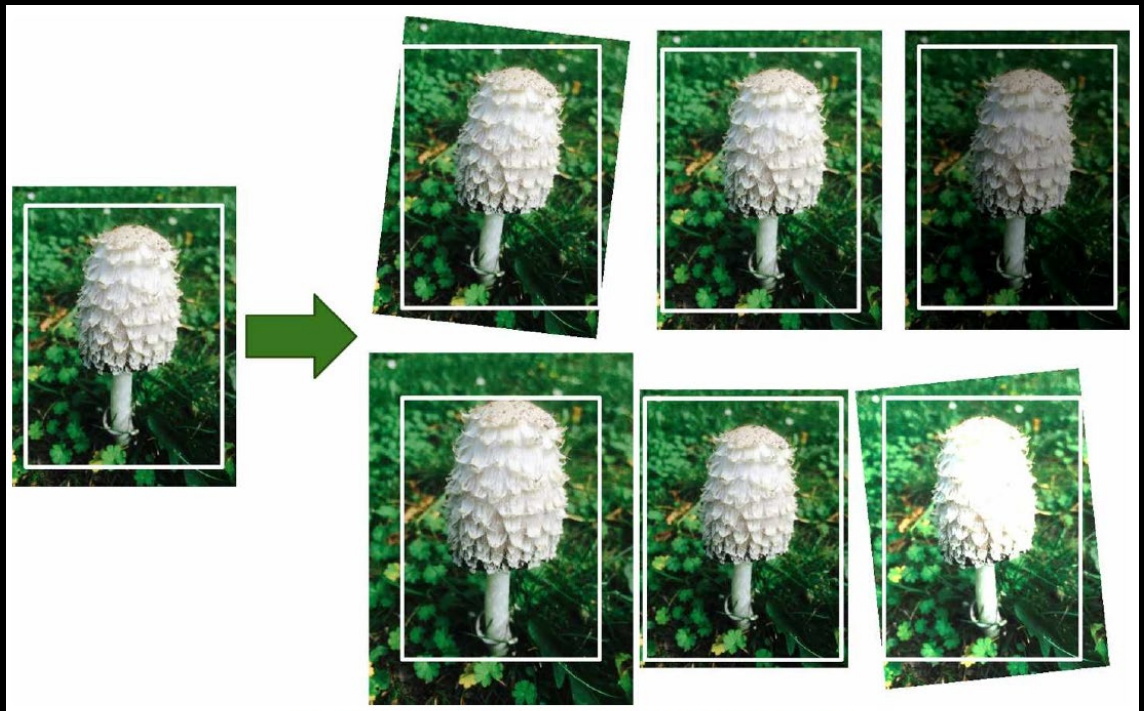


Figure 11-10. Generating new training instances from existing ones

Encoding Categorical Features

Chapter 13

Categorical Features

- Vocabulary = [OCEAN, INLAND, NEAR OCEAN, NEAR BAY, ISLAND]

- How to encode categorical features?
  - Using one-hot vectors
  - Using embeddings

Using One-Hot Vectors

- A vector that contains all zeros except at the index of the feature of interest

- Vocabulary = [OCEAN, INLAND, NEAR OCEAN, NEAR BAY, ISLAND]

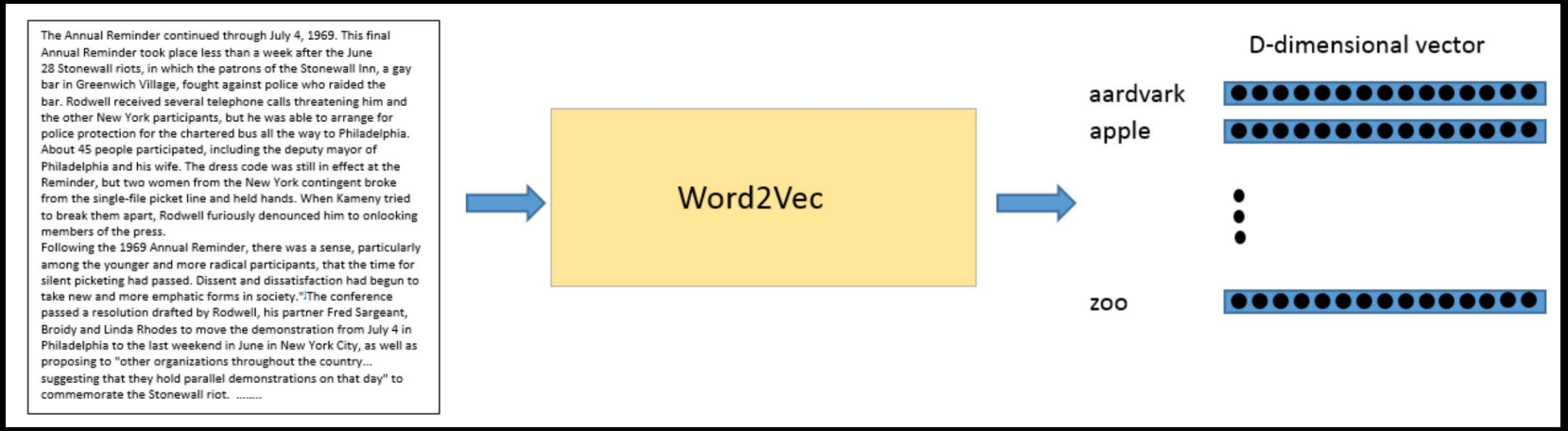
```
[0., 0., 0., 1., 0., 0.],    // NEAR BAY
[0., 0., 0., 0., 0., 1.],    // DESERT (out-of-vocabulary)
[0., 1., 0., 0., 0., 0.],    // INLAND
[1., 0., 0., 0., 0., 0.]],   // OCEAN
```

Word Embeddings

- Idea: use a multi-dimensional vector to represent a word
- Semantically related words end up clustered together
  - The meaning of a word is often determined by its context, i.e., the words that appear nearby (within a fixed size window)

Word2Vec

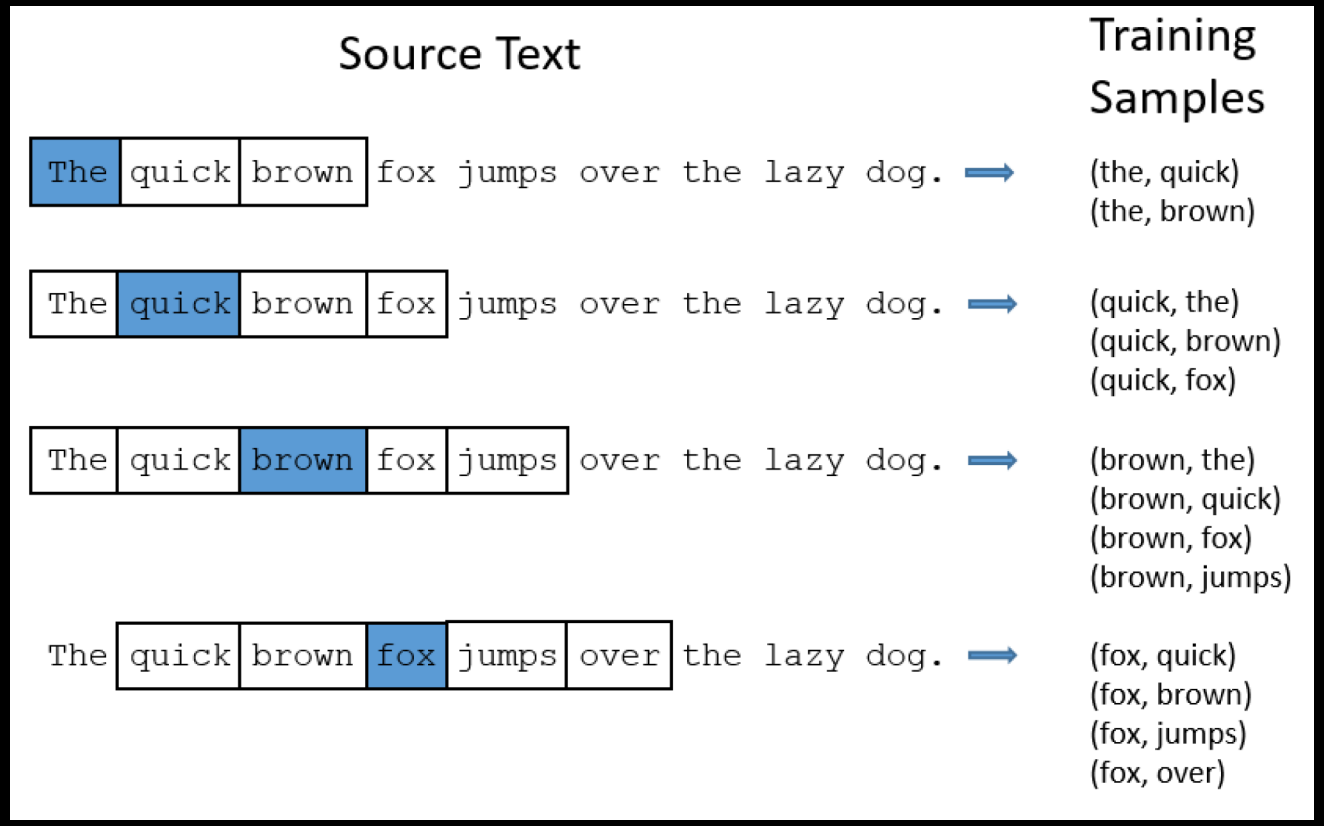
- Feed training words to a single hidden layer neural network
- The objective is to learn the weights of the hidden layer, i.e., the *word vectors*



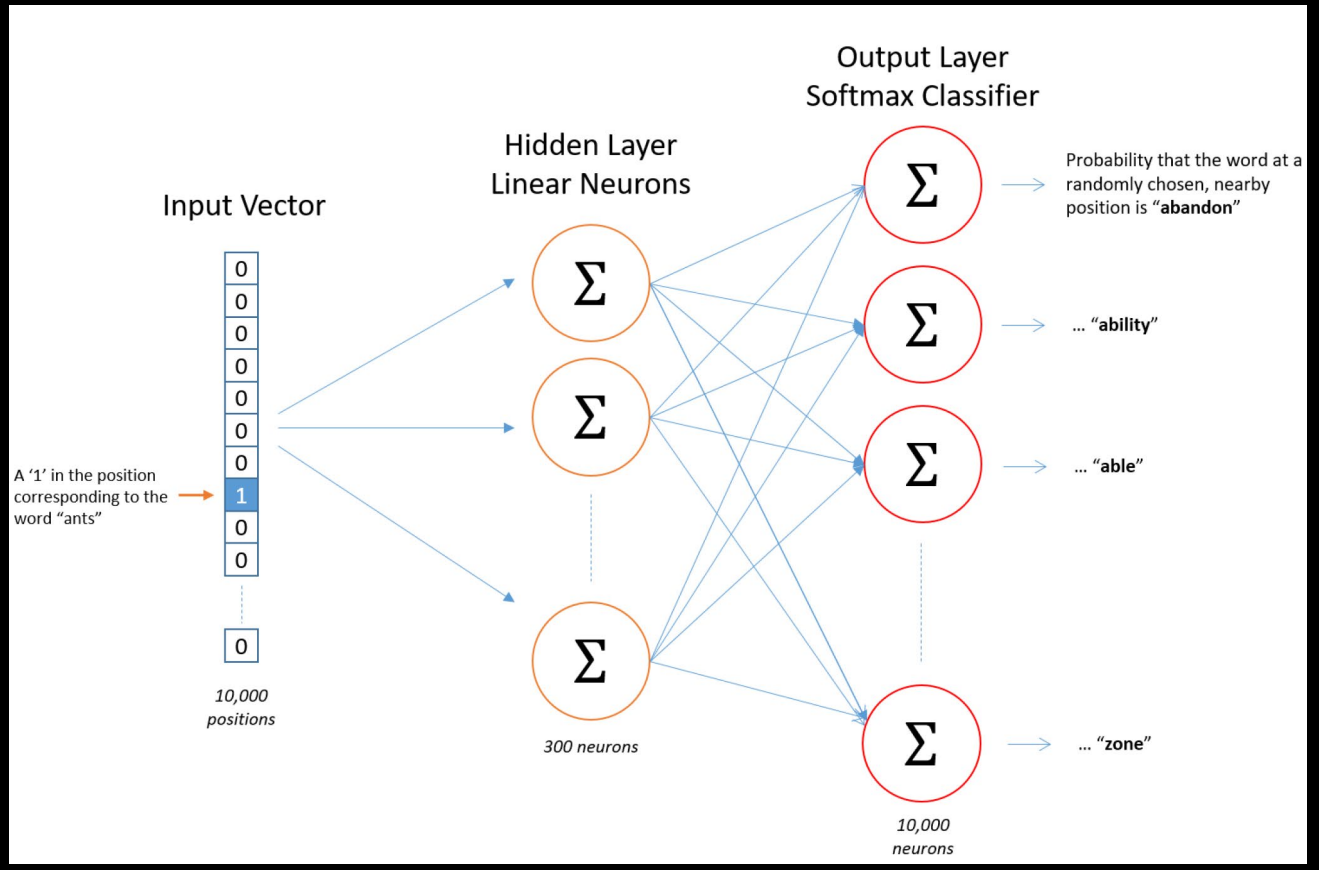
- Google trains Word2Vec model on Google News dataset containing 100 billion words creating 3 million word vectors, each with 300 dimensions.

- References*:*
  - *Distributed Representations of Words and Phrases and their Compositionality*, Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean
  - *Word2Vec Tutorial – The Skip-Gram Model,* Chris McCormick
  - *Word2Vec Tutorial Part 2 – Negative Sampling,* Chris McCormick

- Create word pairs (input/center word and outside/context word)
- Select window size
  - For example if window size = 2, then consider up to 2 words before and after the center word



- Represent input word and output word as one-hot vectors
  - If the vocabulary is 10K words, then the vector will contain a ‘1’ in the position of the corresponding word, the rest are 0’s
  - 300 neurons in the hidden layer



- Lookup Table
  - Once trained, the hidden layer functions as a lookup table

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

Deep Computer Vision Using Convolutional Neural Networks

Chapter 14

- CNN model is based on the visual cortex in image recognition
- Neurons in visual cortex have small local receptive field

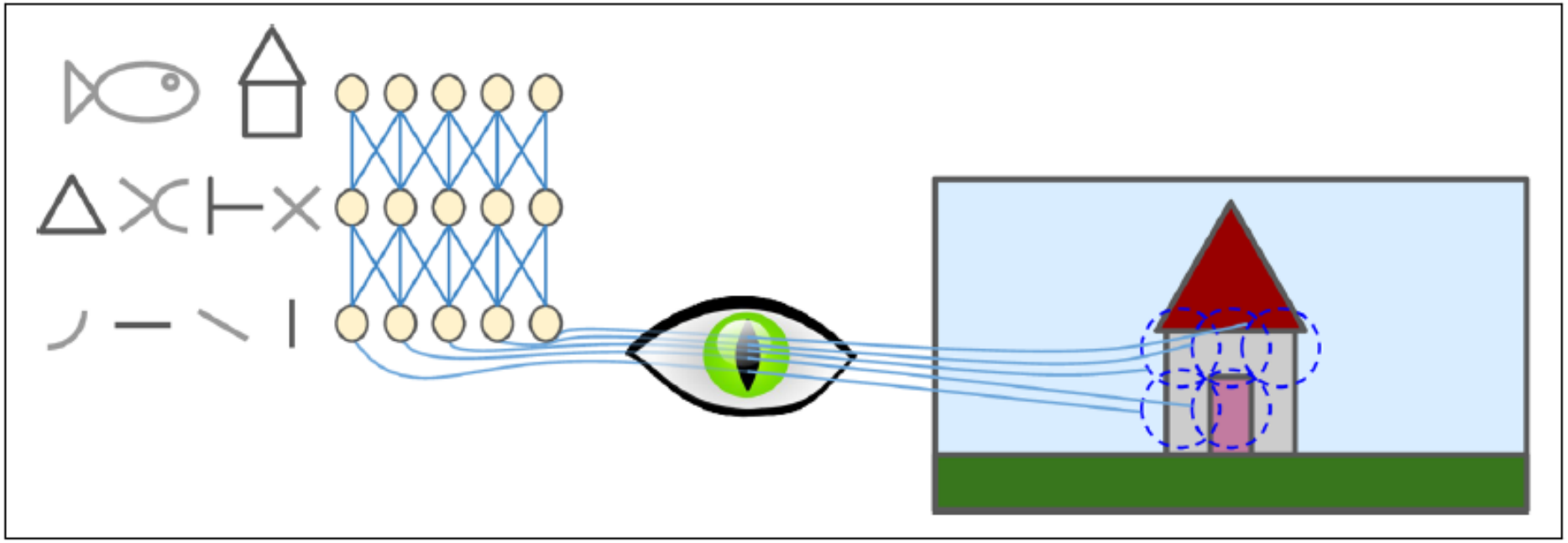


Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields.

Convolutional Layers

- A neuron in a layer is connected only to pixels of the previous layer that belong to its receptive field

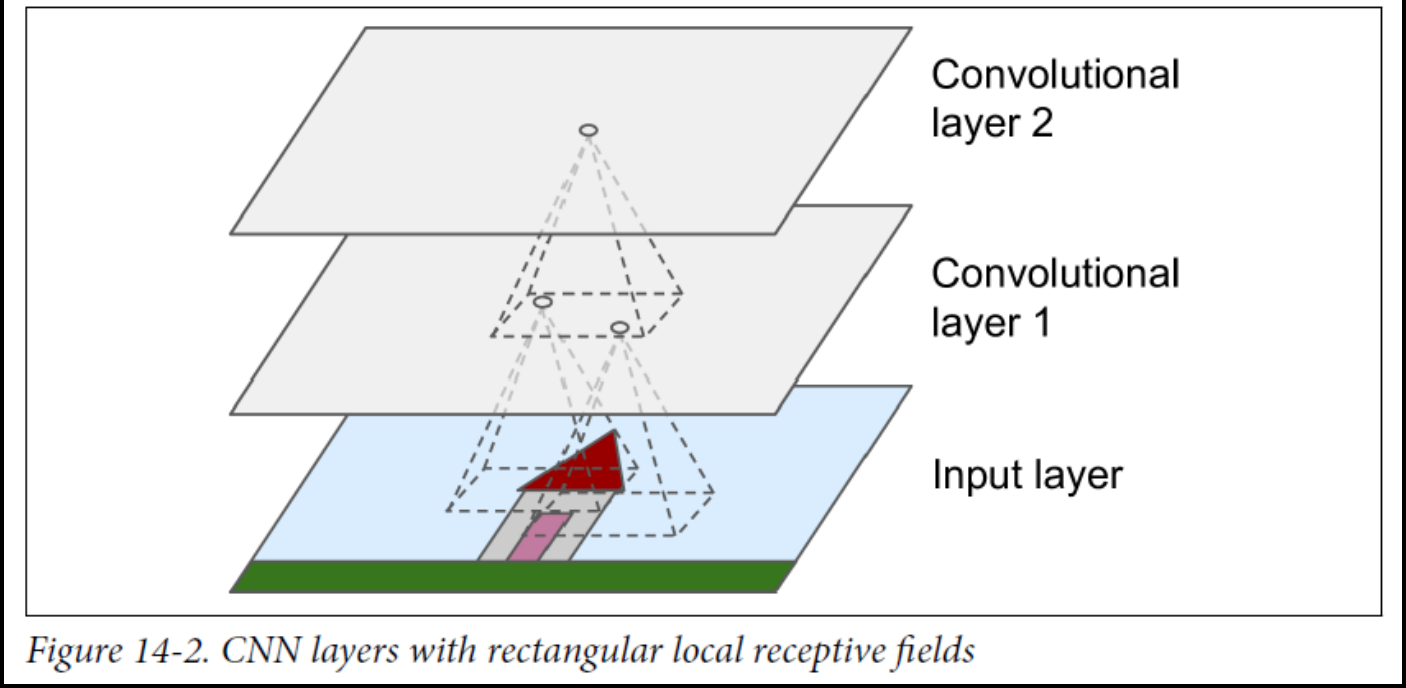


Figure 14-2. CNN layers with rectangular local receptive fields

- Finds small low-level features in 1st hidden layer. Then larger higher-level features in the next layer etc

- Typically convolutional layer has multiple filters and outputs one feature map per filter.
- All neurons within a feature map share the same parameters
- Receptive field extends across all of the previous layers' feature maps

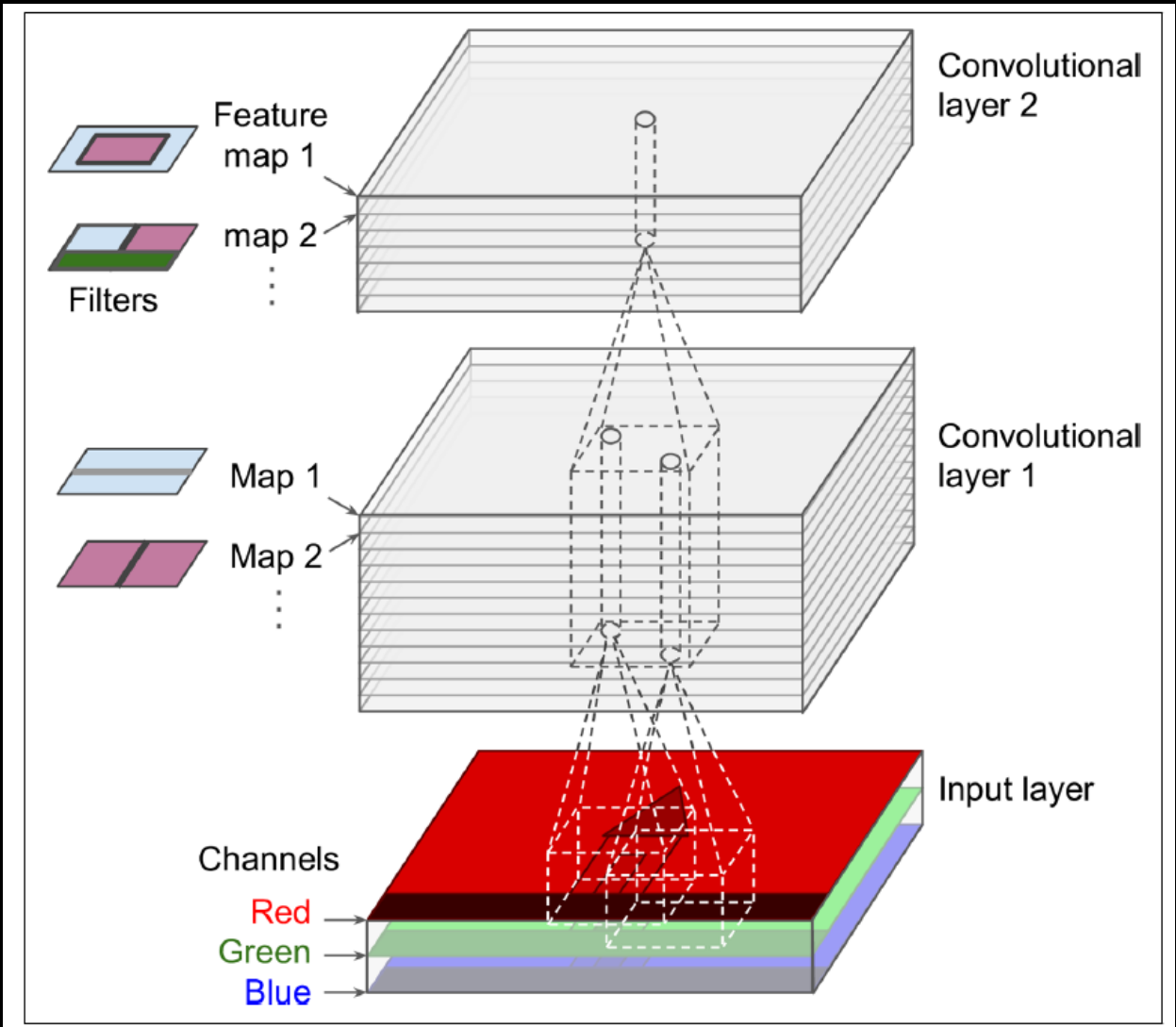


Figure 14-6. Convolutional layers with multiple feature maps, and images with three color channels

Zero Padding

- Padding parameter: *valid* or *same*

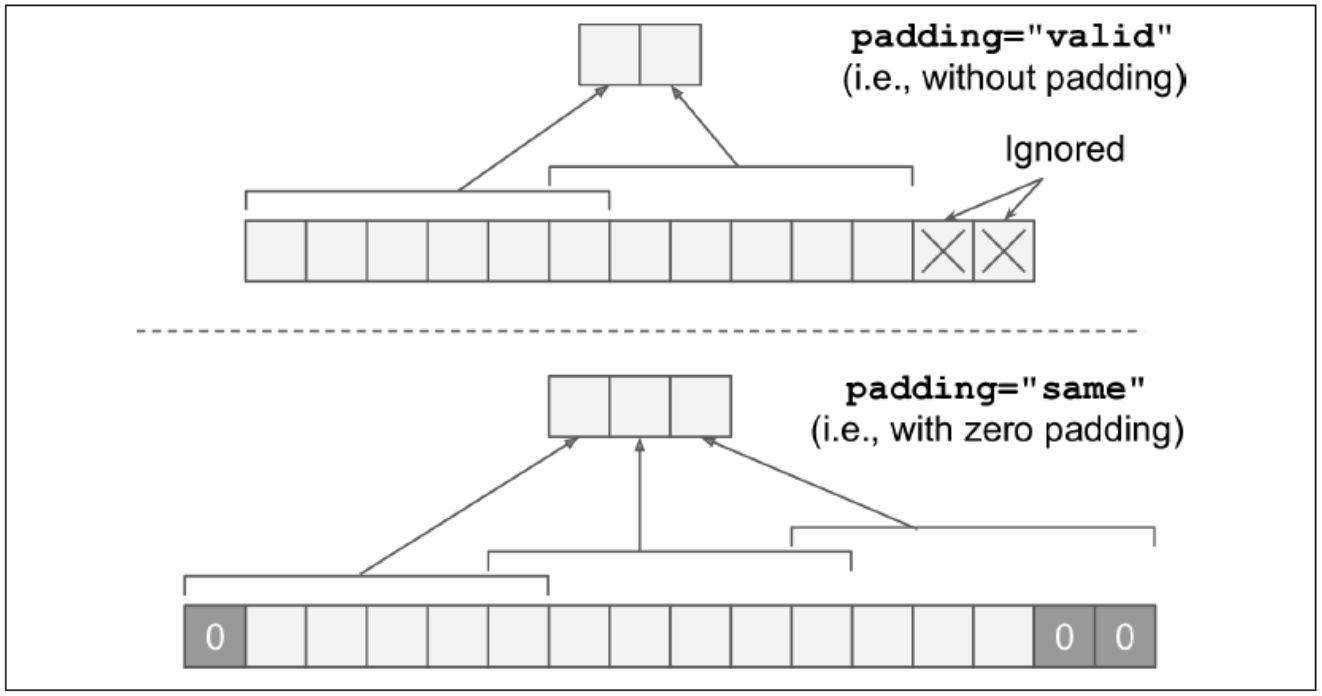
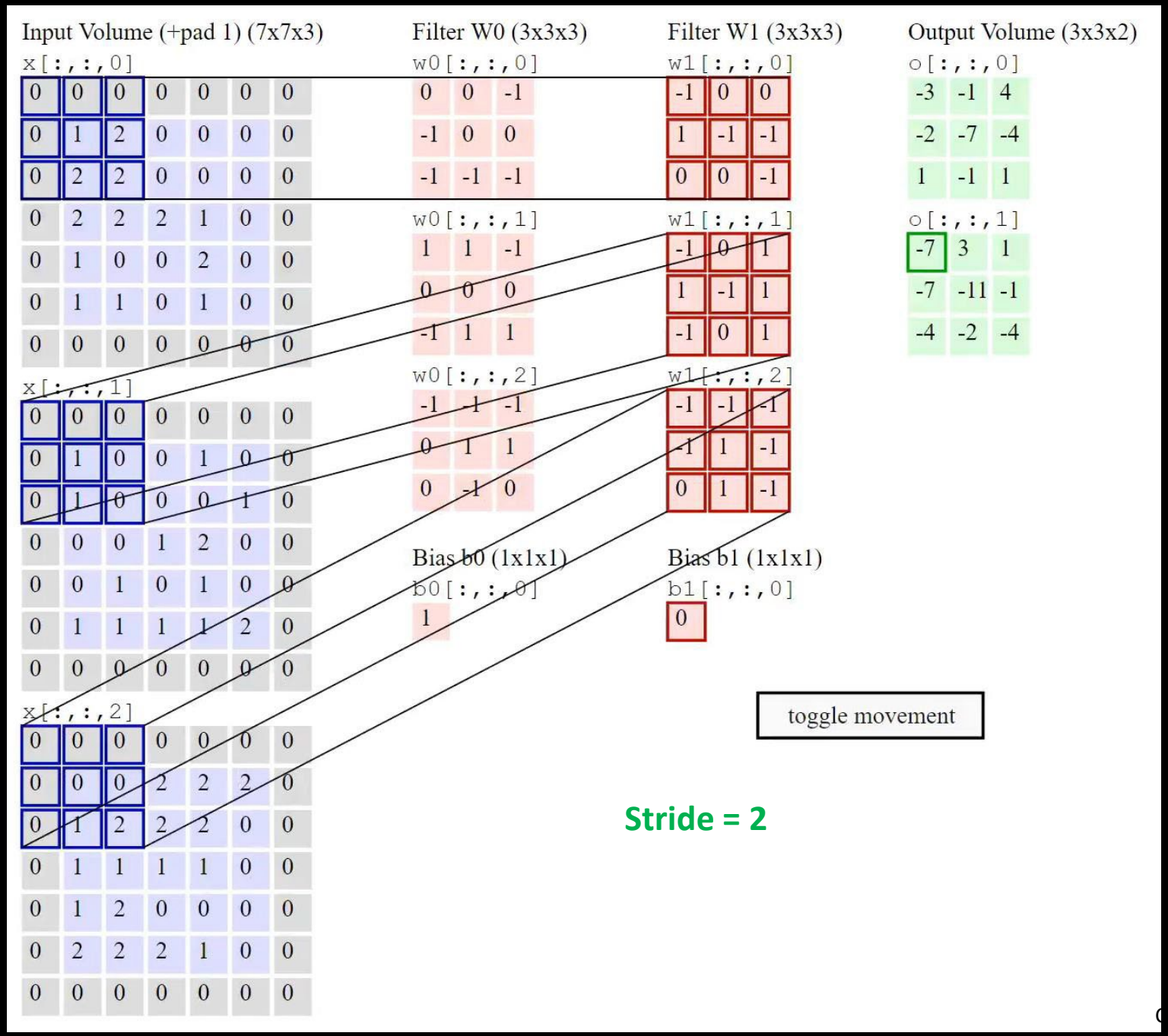


Figure 14-7. "SAME" or "VALID" padding (with input width 13, filter width 6, stride 5)



Convolutional Layer

Example:

0x-1 + 0x0 + 0x0 +
0x1 + 1x-1 + 2x-1 +
0x0 + 2x0 + 2x-1 +

0x-1 + 0x0 + 0x1 +
0x1 + 1x-1 + 0x1 +
0x-1 + 1x0 + 0x1 +

0x-1 + 0x-1 + 0x-1 +
0x-1 + 0x1 + 0x-1 +
0x0 + 1x1 + 2x-1

= -7

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.

W1 = H1 = 5
K = 2, F = 3, S = 2, P = 1

$$\mathbf{W2 = H2 = \frac{(5 - 3 + 2)}{2} + 1 = 3}$$

$$\mathbf{D2 = 2}$$

Pooling Layer

- Goal is to subsample (shrink) the input image
- Each neuron in this layer is connected to a small number of neurons in the previous layer that fall in its receptive field

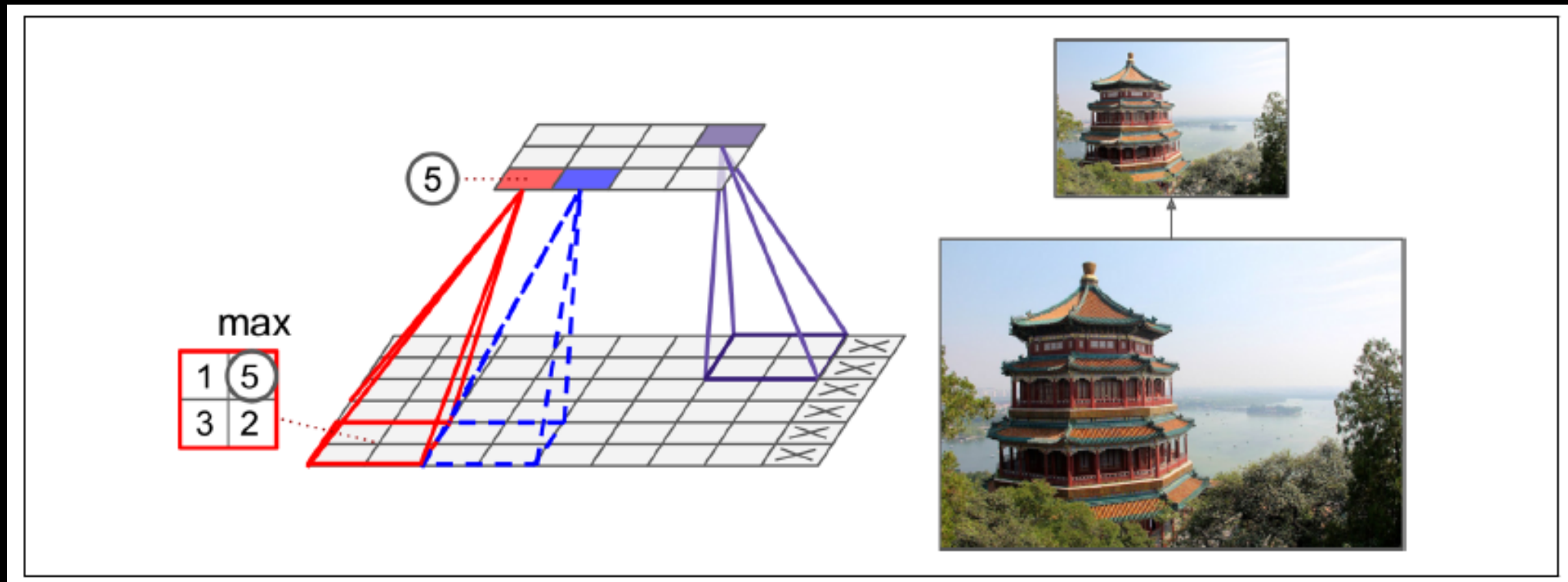


Figure 14-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

W1 = H1 = 4
F = 2, S = 2

$$\mathbf{W2 = H2 = \frac{(4-2)}{2} + 1 = 2}$$

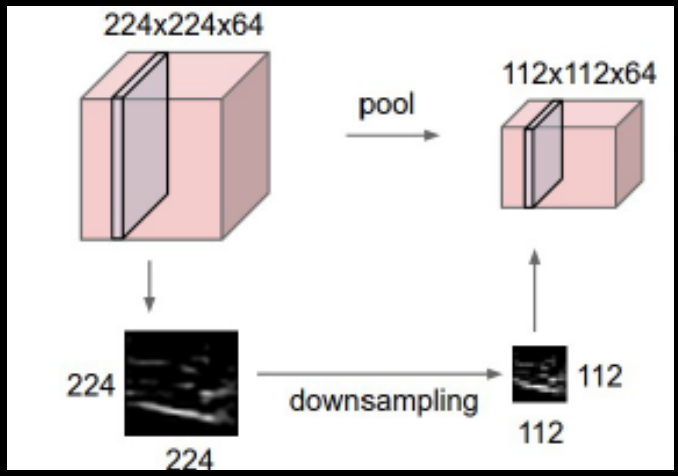
Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2 →

6	8
3	4

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input



CNN Architectures

- A stack of conv layers – pooling layer – conv layer – pooling layer etc
- Full connected layers at the top of the stack
- ReLU's for intermediate layers; Softmax in the output layer

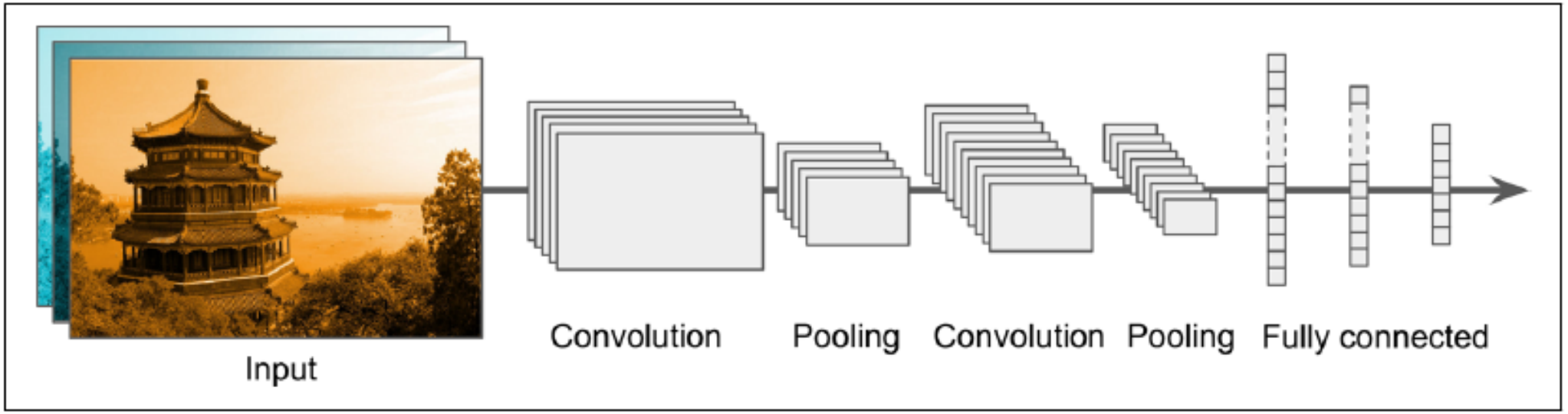


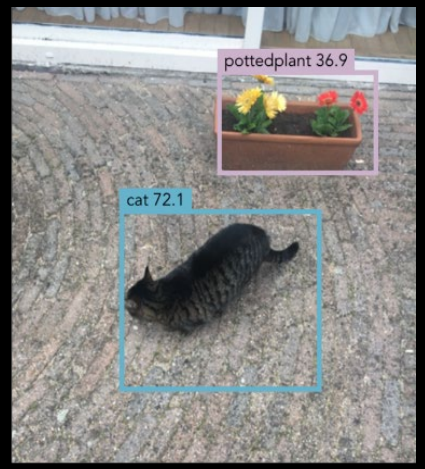
Figure 14-11. Typical CNN architecture

VGG Model

LAYER	OUTPUT SHAPE (stride=1)
Input	(224, 224, 3)
CONV (3X3X64)	(224, 224, 64)
CONV (3X3X64)	(224, 224, 64)
POOL (2X2)	(112, 112, 64)
CONV (3X3X128)	(224, 224, 128)
CONV (3X3X128)	(224, 224, 128)
POOL (2X2)	(56, 56, 128)
CONV (3X3X256)	(56, 56, 256)
CONV (3X3X256)	(56, 56, 256)
CONV (3X3X256)	(56, 56, 256)
POOL (2X2)	(28, 28, 256)
CONV (3X3X256)	(28, 28, 512)
CONV (3X3X256)	(28, 28, 512)
CONV (3X3X256)	(28, 28, 512)
POOL (2X2)	(28, 28, 512)
CONV (3X3X512)	(14, 14, 512)
CONV (3X3X512)	(14, 14, 512)
CONV (3X3X512)	(14, 14, 512)
POOL (2X2)	(7, 7, 512)
AFFINE (4096 UNITS)	(4096, 1)
AFFINE (4096 UNITS)	(4096, 1)
AFFINE (100 UNITS)	(100, 1)

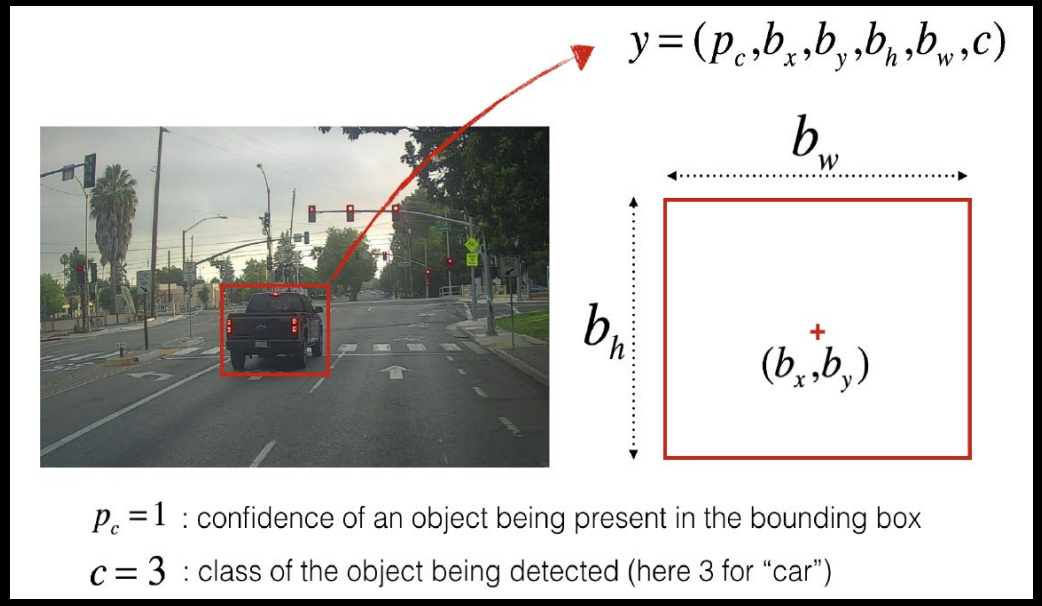
Object Detection

- Object classification recognizes if an object is present or not.
- Object detection classifies and localizes multiple objects in an image by drawing bounding boxes



YOLO (You Only Look Once)

- Runs forward pass once
- Divide the input image into *SxS* grid
- Each grid cell predicts B bounding boxes and their corresponding confidence scores p_c and class probabilities c_k
  - Each bounding box is defined by 6 numbers



Problem with Deep CNN

- Deeper CNN achieves better generalization (can represent complex functions at the deeper layers)
- However, after a certain point deeper networks have higher training and test error

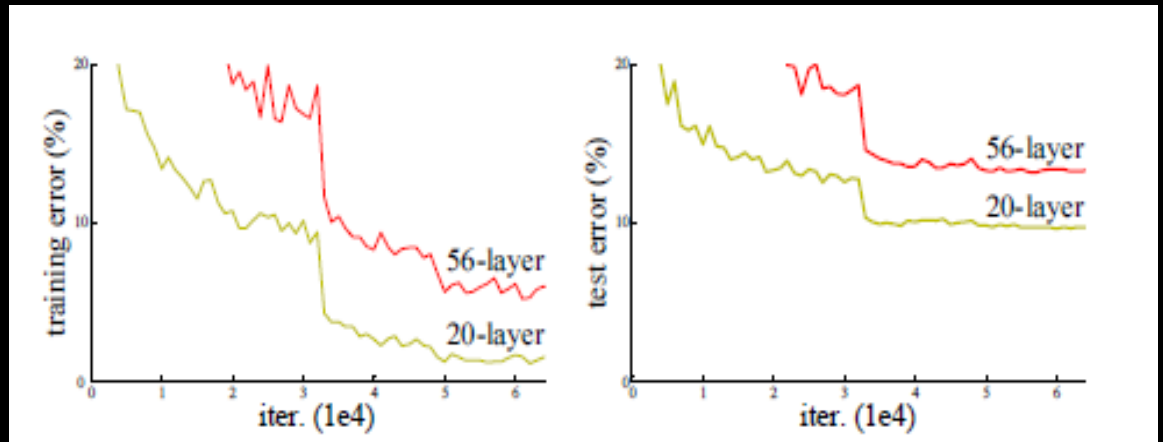
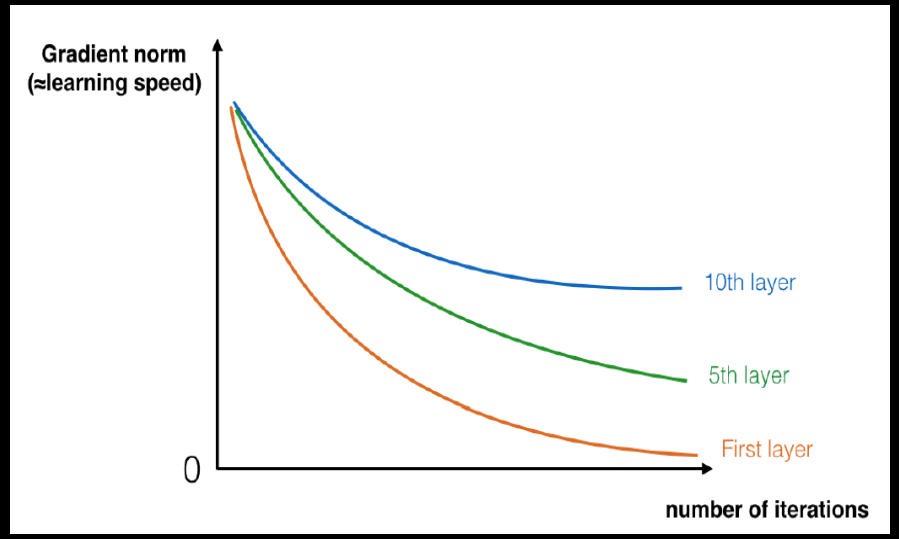


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

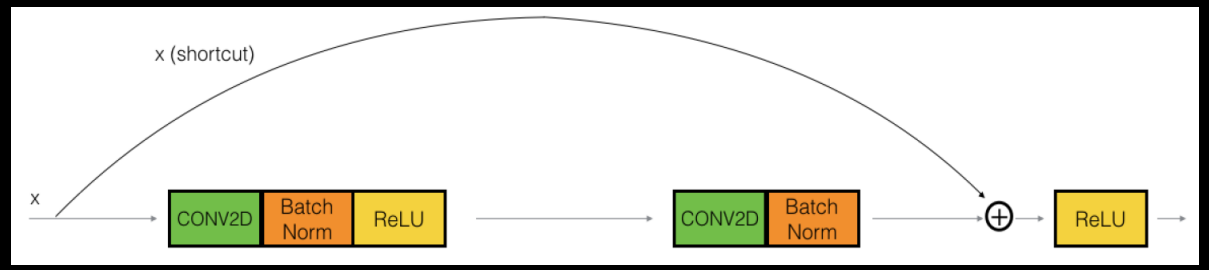
Problem with Deep CNN (continued)

- Degradation is not caused by overfitting but by vanishing gradient
- The system cannot be optimized anymore during back propagation because the gradient is multiplied by the weight matrix on each step from the final layer back to the first layer. As result, the gradient decreases exponentially toward zero

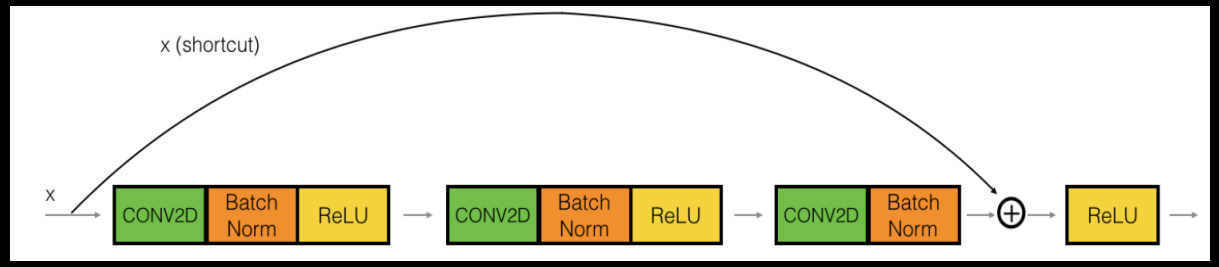


ResNet

- Solves vanishing gradient problem
- Consists of main path and short cut connection to main path
- Gradients are directly back-propagated by shortcut(skip) connections



2-layer skip over



3-layer skip over

- Error Rates

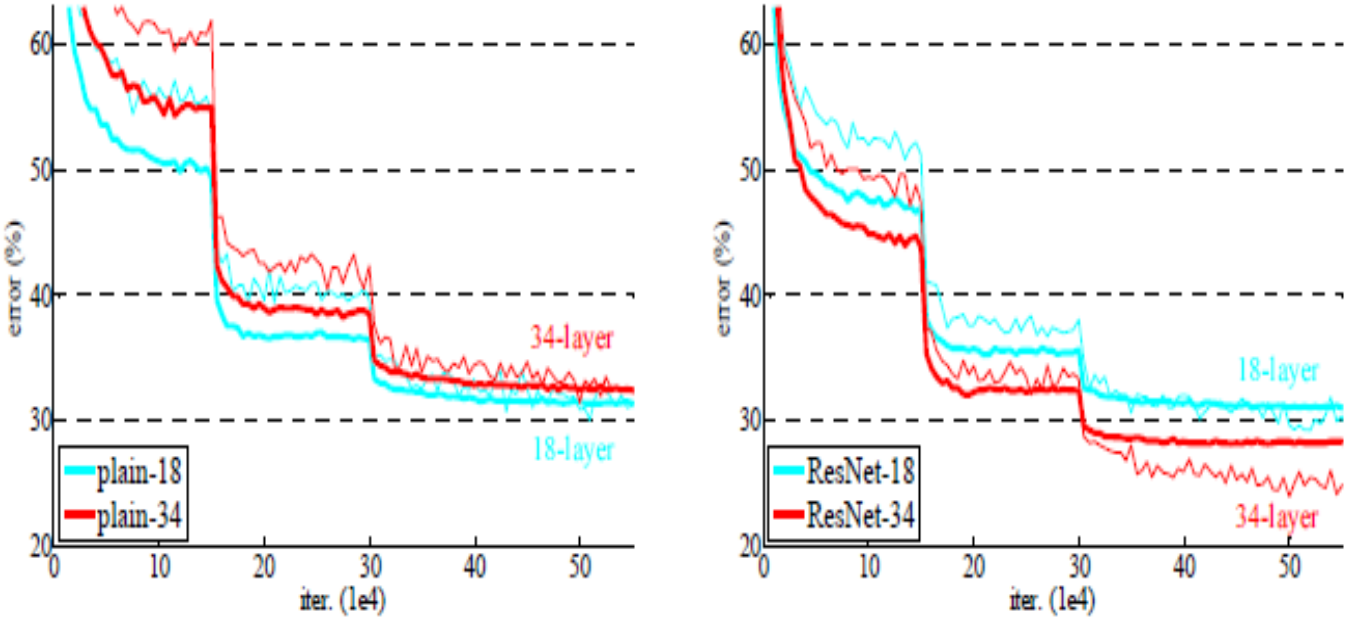


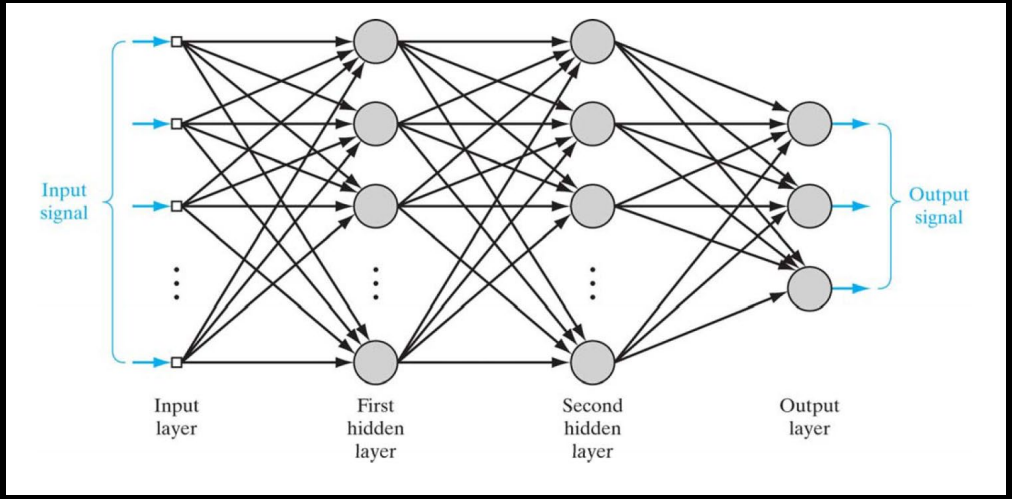
Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Processing Sequences Using RNNs

Chapter 15

Sequential Information

- Sequence is important
  - We understand a word in someone's speech in the context of the previous words
  - We interpret the current scene in a movie based on what we saw earlier
- Typical neural networks are stateless
  - Decisions made at time *t* are independent of those made at time *t-1*



Recurrent Neural Networks Predict The Future

- Finish someone’s sentence
- Analyze time series data
- Anticipate car trajectories
- Natural language processing (NLP) applications

Recurrent Neural Network (RNN)

- Looks like a feedforward NN
- Uses the state of the hidden layers from the last step
- Has connections pointing back in time
- Each time step, a neuron receives an input, produces an output, sends the output to itself

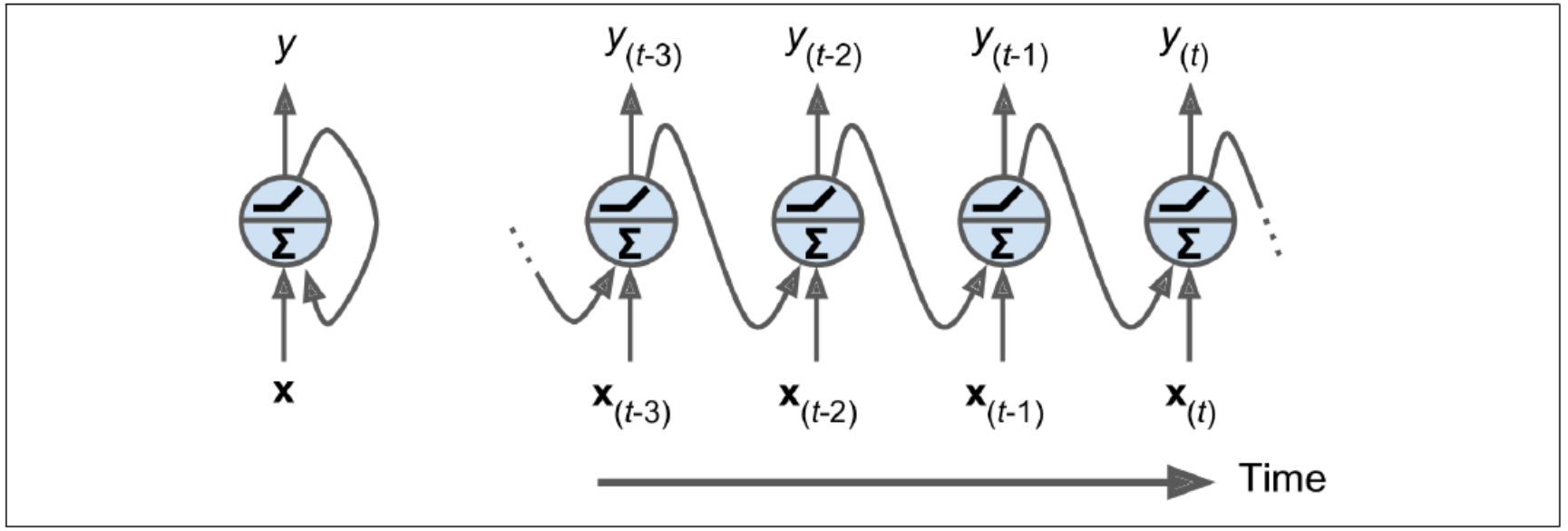


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

Memory Cell

- Part of the neuron (or network) that preserves the state across time
  - $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$

- The output $\mathbf{y}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$

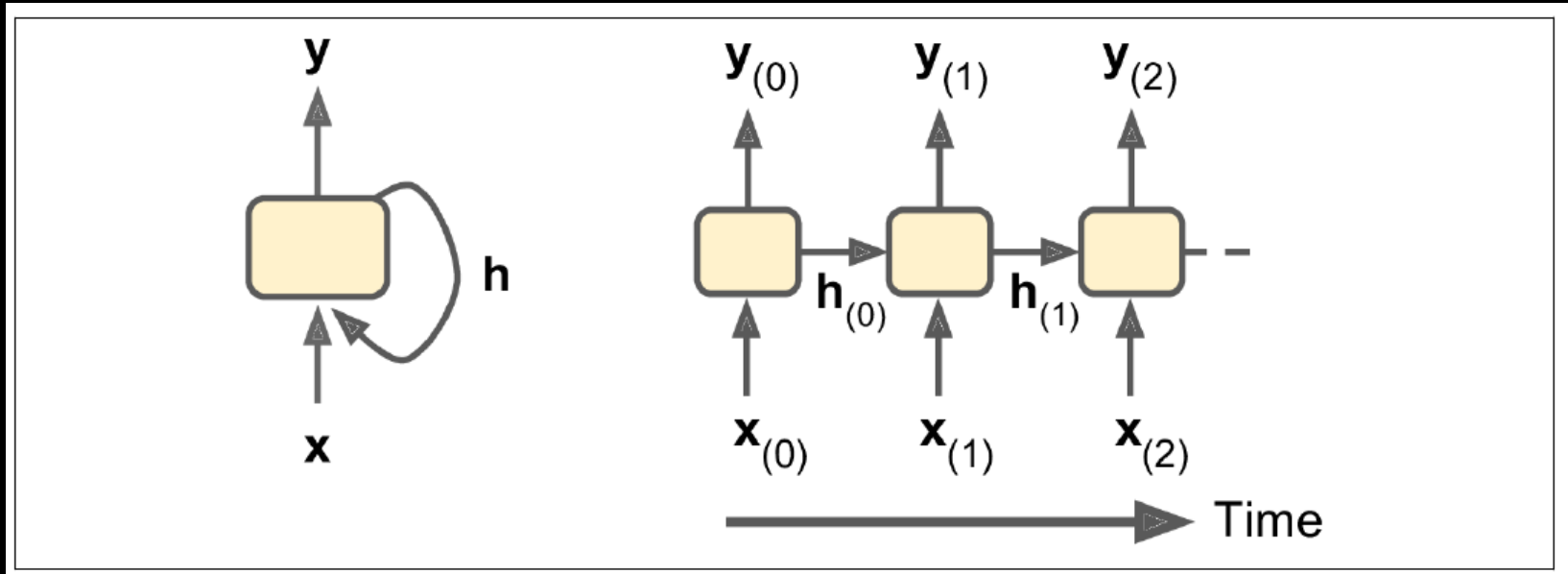


Figure 15-3. A cell's hidden state and its output may be different

Training RNNs

- Back-propagation Through Time (BPTT)
  - Forward pass through the unrolled network (dashed arrows)
  - Back propagate gradient through the unrolled network (solid arrows)
  - Update model parameters

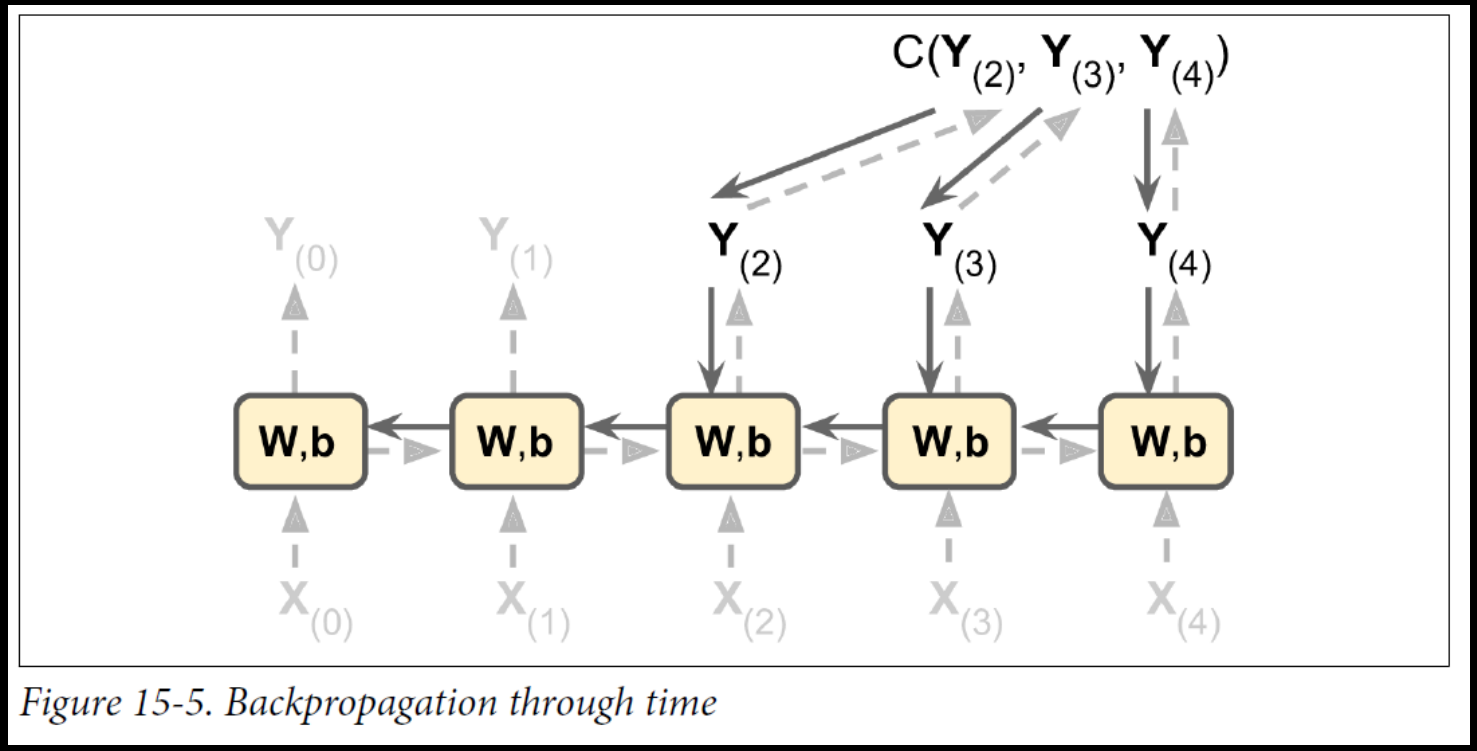


Figure 15-5. Backpropagation through time

Short-Term Memory Problem

- Loss of information at each time step → RNN's state eventually contains no trace of the first inputs
- Mitigation strategy: use LSTM (Long Short-Term Memory) cell

- Idea is the cell learns:
  - What to store
  - What to discard
  - What to read

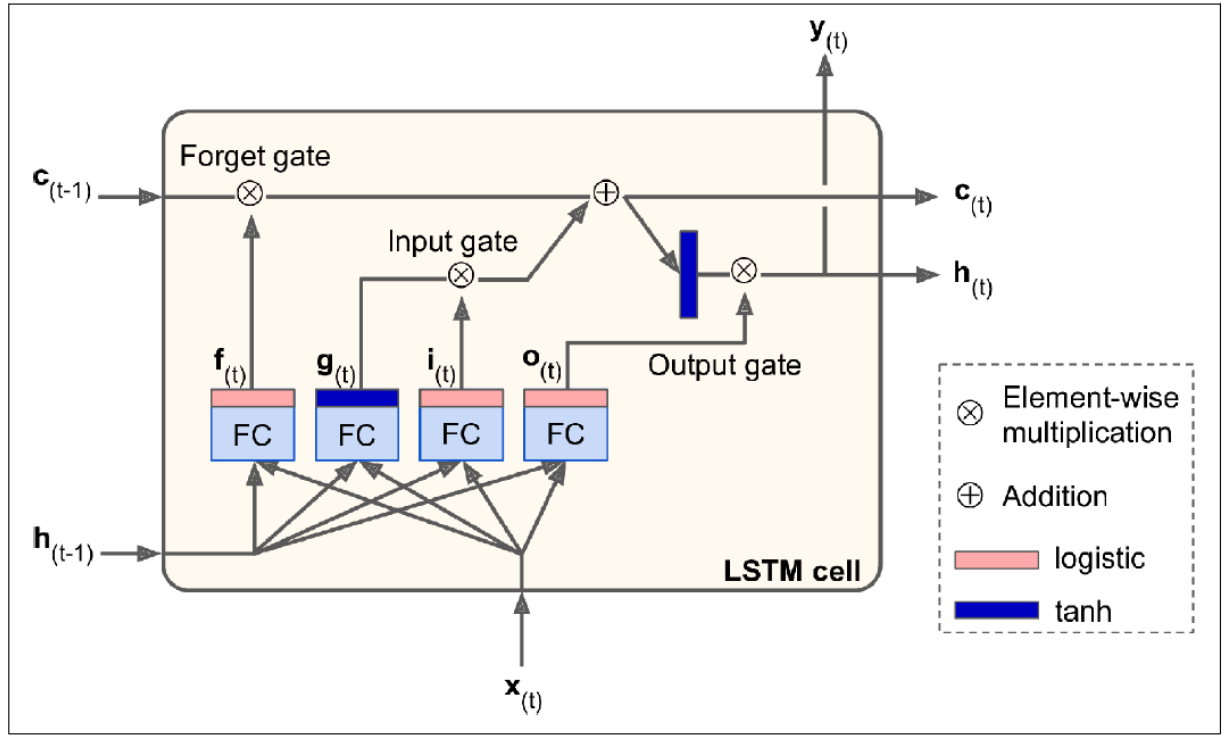


Figure 15-9. LSTM cell

Representation Learning and Generative Learning

Chapter 17

Autoencoders

- Unsupervised neural networks that learn latent representations (codings) of the input data
- Autoencoder converts the input to efficient latent representations then reproduces an output that looks like the input

- Applications
  - Dimensionality reduction
  - Feature detectors
  - Unsupervised pretraining of DNNs
  - Generative models

Autoencoders

- Encoder (recognition network)
  - Converts inputs to some latent representation
- Decoder (generative network)
  - Converts latent representation to outputs

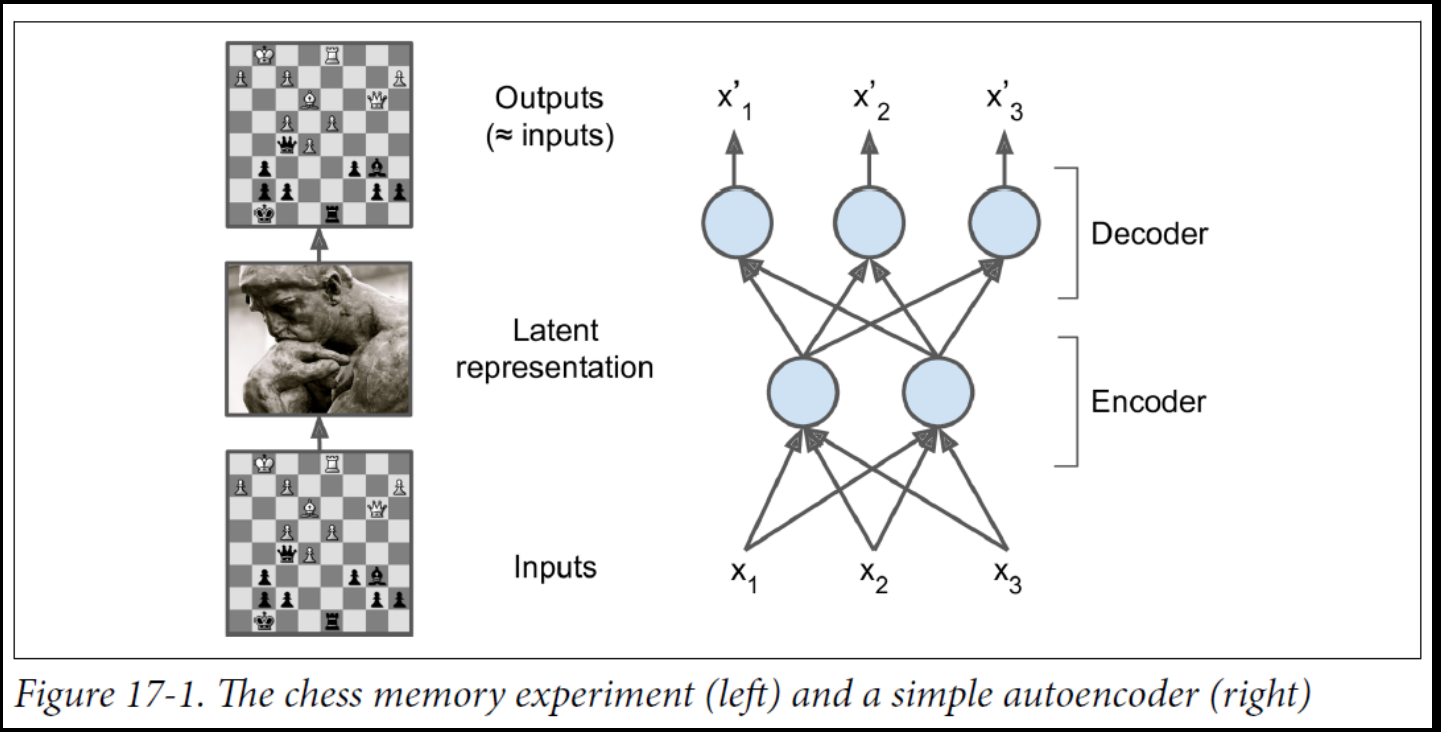


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

- Undercomplete if the dimensionality of the internal representation (coding layer) is less than that of the input data
- Overcomplete if the dimensionality of the internal representation is more than that of the input data

Stacked Autoencoders

- aka Deep Autoencoders
  - Have multiple hidden layers
  - Symmetrical relative to the central hidden layer (coding layer)

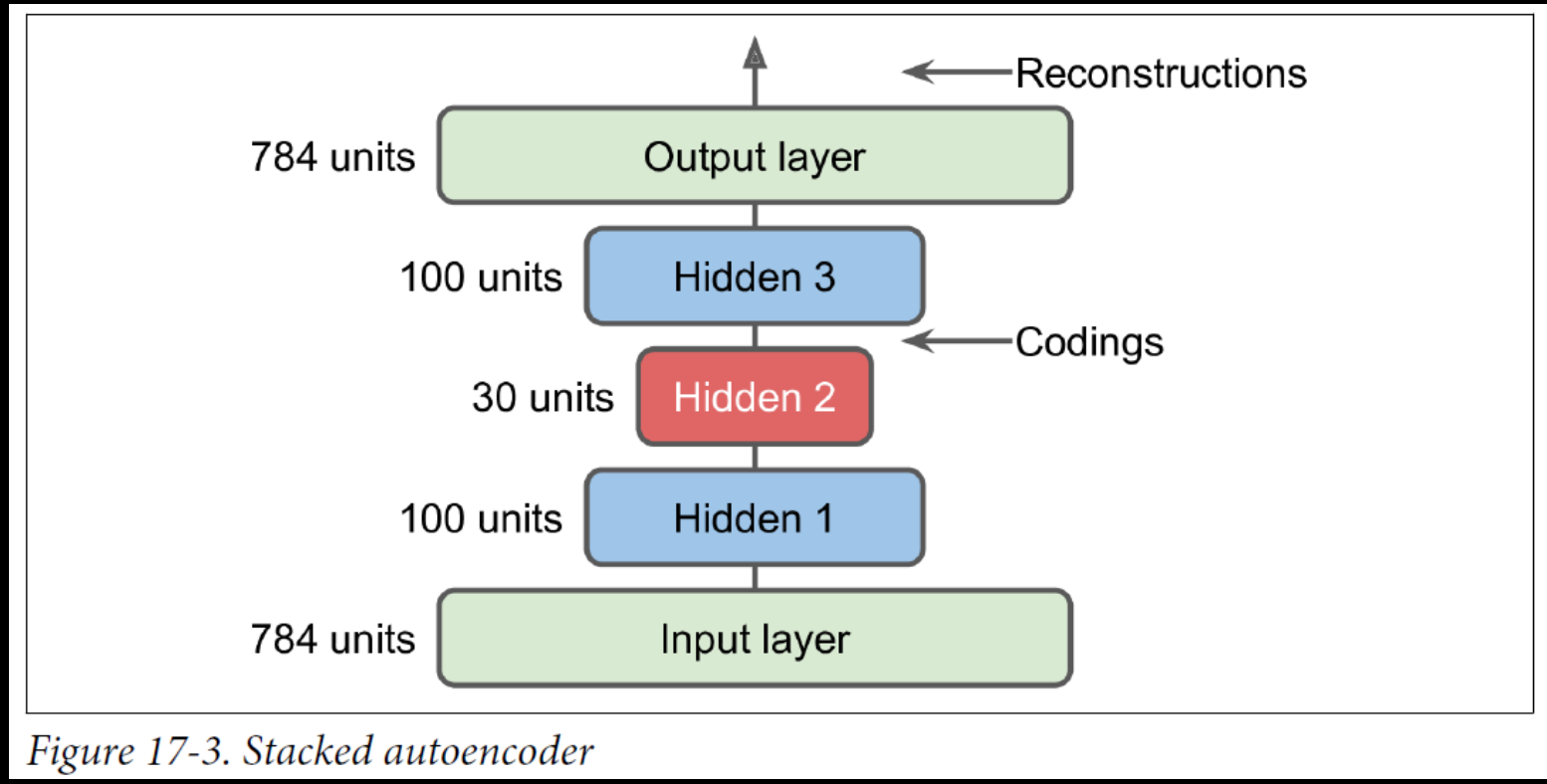


Figure 17-3. Stacked autoencoder

- Should not make the autoencoder *too powerful* otherwise it will not learn useful data representation

Training One Autoencoder at a Time

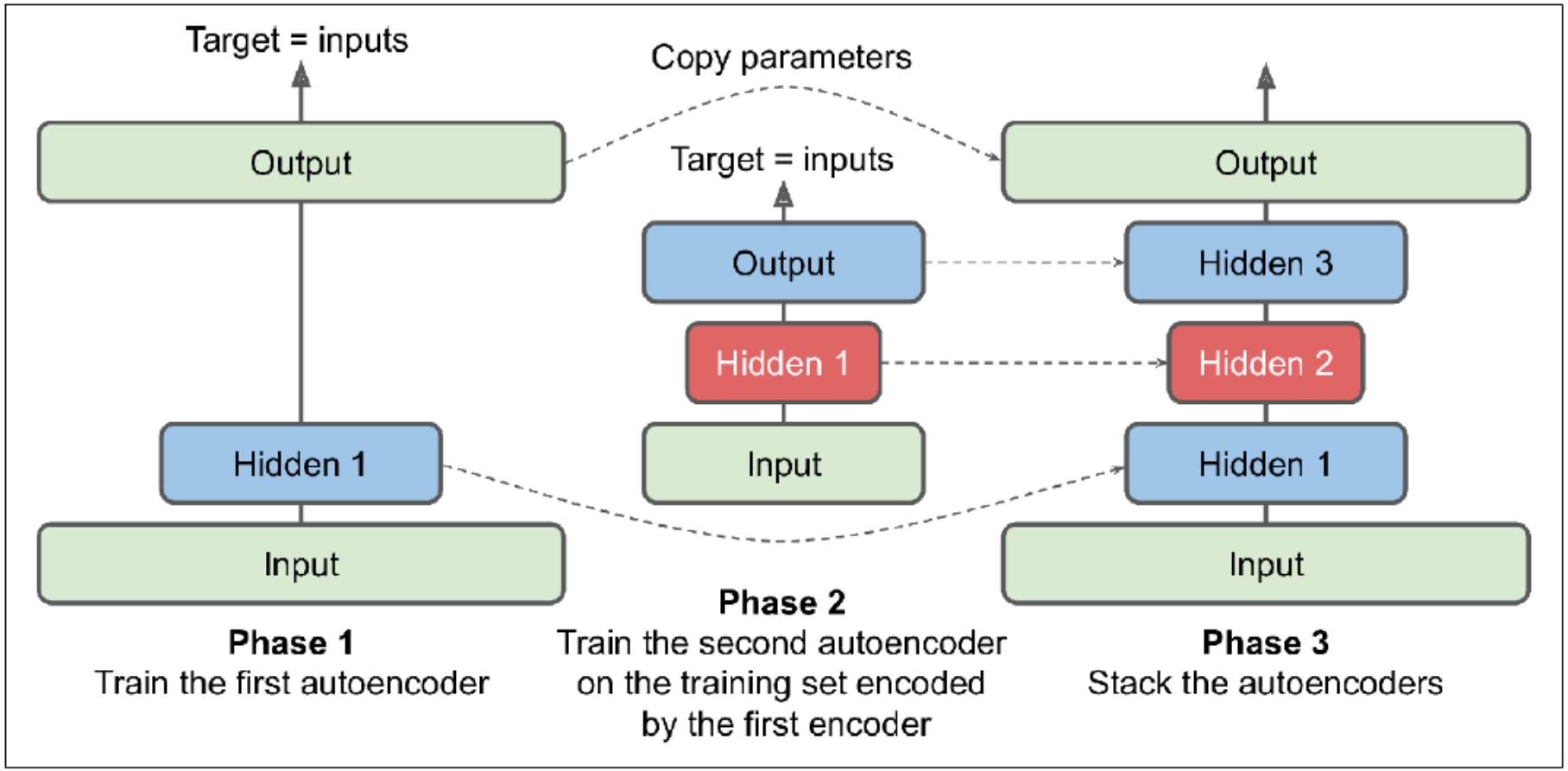


Figure 17-7. Training one autoencoder at a time

Unsupervised Pretraining

- Train a stacked autoencoder on unlabeled data
- Freeze & reuse the lower layers for a neural network that can be trained by labeled data

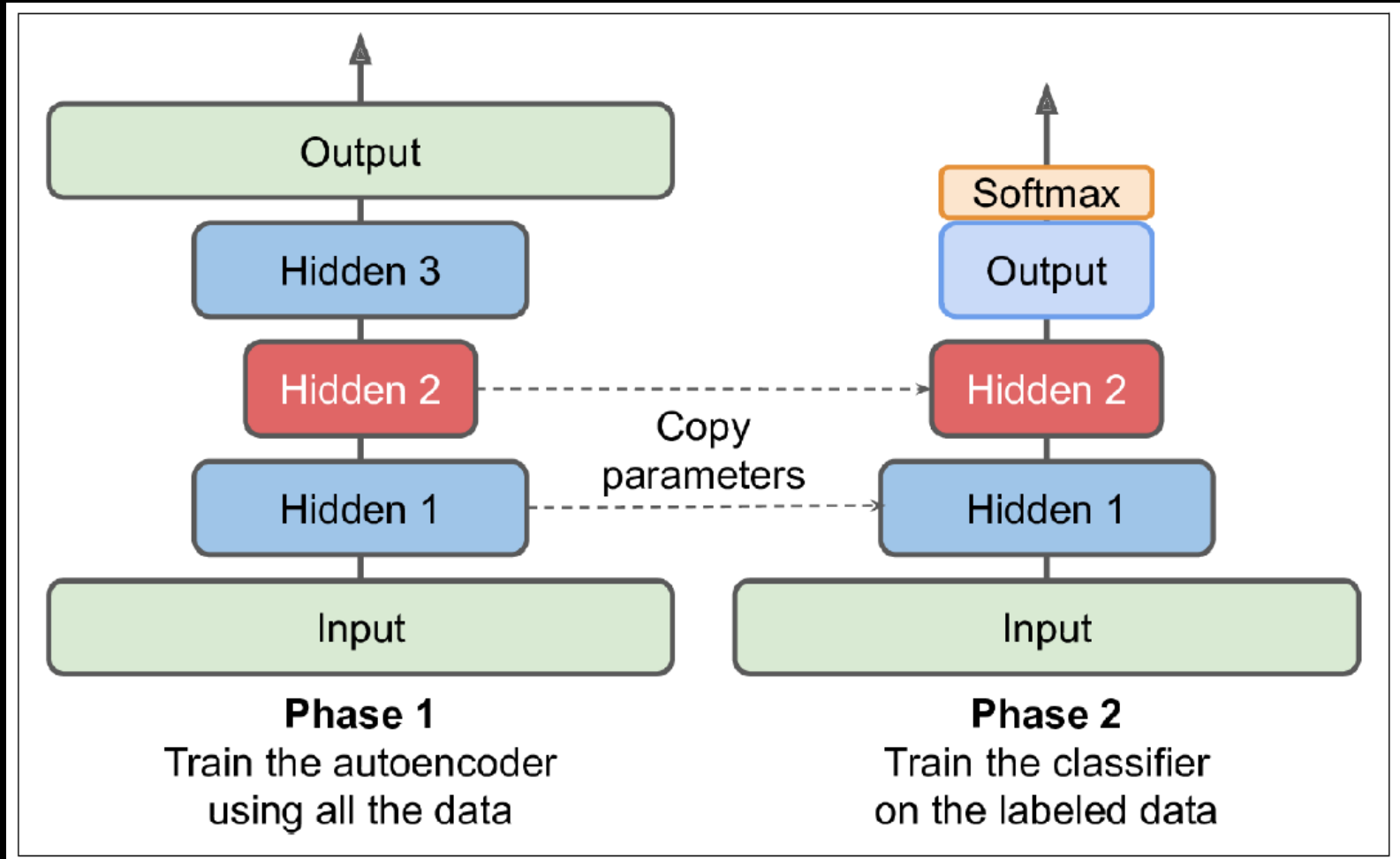


Figure 17-6. Unsupervised pretraining using autoencoders

Generative Adversarial Networks (GANs)

- Composed of 2 competing NNs
  - Generator – generates data that looks as similar to training data as possible
  - Discriminator – tries its best to distinguish the real data from the fake data
- Applications
  - Dataset augmentation
  - Image editing
  - Text, audio, video generation

Generative Adversarial Networks

- Generator
  - Takes a random distribution as input and outputs some data (image)
- Discriminator
  - Takes a fake image from Generator or a real image from training set
  - Must determine if the image is fake or real

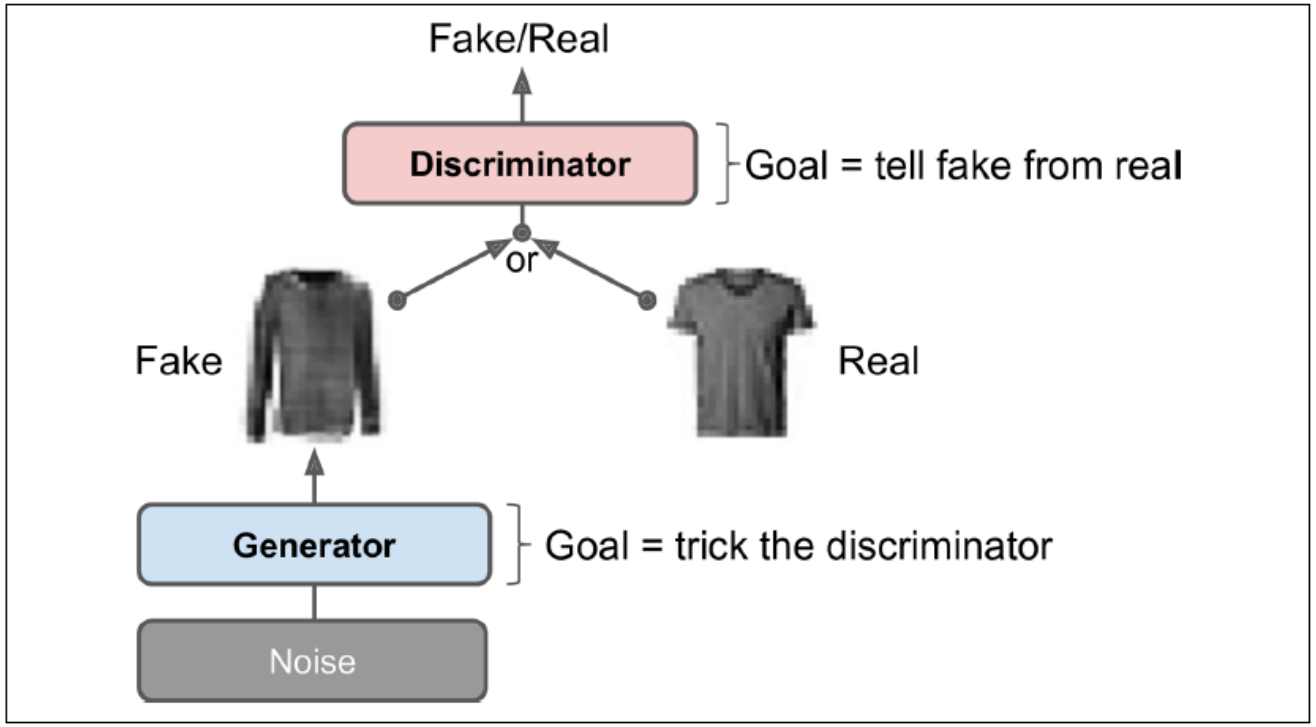


Figure 17-15. A generative adversarial network

- Good reading: https://developers.google.com/machine-learning/gan

Training GANs

- Training Discriminator
  - Freeze the Generator weights
  - Feed real and fake images with labels (0 and 1)
  - Optimize the weights of Discriminator only

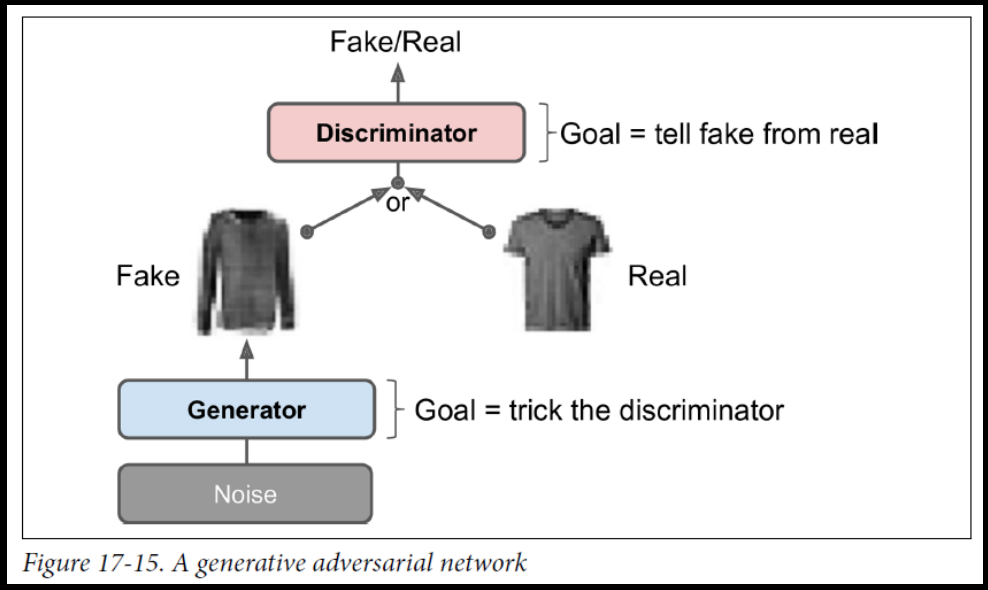


Figure 17-15. A generative adversarial network

- Training Generator
  - Freeze the Discriminator weights
  - Feed fake images to Discriminator to cause it to *wrongly* believe to be real
  - Optimize the weights of Generator only
  - Generator gets the gradients through Discriminator → it learns how to produce convincingly fake images
    - Note: Generator never sees any real images

These places & objects do not exist …

- Source: https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/

None of these people are real …

- Source: https://futurism.com/incredibly-realistic-faces-generated-neural-network