Introduction to Machine Learning

Lecture 16
Deep Learning III
Gradient descent/Regularization/Layer types

Goker Erdogan
26 – 30 November 2018
Pontificia Universidad Javeriana

Training a neural network

- Use backprop to get the gradient of error function $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$
- Gradient descent to update the weights

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \gamma \nabla_{\boldsymbol{w}} E(\boldsymbol{w})$$

- Remember that error is a sum over training samples

$$E(\boldsymbol{w}) = \frac{1}{N} \sum_{n} E_n(\boldsymbol{w})$$

- Batch (deterministic) gradient descent
  - Use all the training samples to calculate the gradient
  - Inefficient if the dataset is large
    - Can get a good estimate of the gradient without using the full dataset

Stochastic gradient descent

- Stochastic gradient descent
  - Pick a training sample randomly
  - Estimate gradient from a single sample

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \gamma \nabla_{\boldsymbol{w}} E_n(\boldsymbol{w})$$

- Estimates will be noisy
  - Need to keep learning rate γ small
  - But works well in practice
  - Rather inefficient
    - Might take a long time to train

Minibatch gradient descent

- Minibatch gradient descent
  - Calculate gradients using a minibatch of B < N samples

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \gamma \frac{1}{B} \sum_b \nabla_{\boldsymbol{w}} E_b(\boldsymbol{w})$$

  - B: (mini)batch size

- A good middle ground between stochastic and batch gradient descent
  - B usually ~ 32, 64, 128, 256, …
    - GPU memories handle such sizes more efficiently
  - Larger usually better
    - But constrained by memory
  - Gradient estimates are good enough
    - Data is redundant

Minibatch gradient descent

Algorithm

- Given a training set of N samples, $\{x, t\}_{n=1, 2, \ldots, N}$
- (Mini)batch size: B
- Repeat until you meet stopping criteria
  - Randomly sample B samples from training set
  - Calculate $\sum_b \nabla_{\boldsymbol{w}} E_b(\boldsymbol{w})$ using backprop
  - Update weights

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \gamma \frac{1}{B} \sum_b \nabla_{\boldsymbol{w}} E_b(\boldsymbol{w})$$

Optimization using gradient descent

- Most important hyperparameter is learning rate γ
  - Not too large or too small
  - Values around 1e-3, 1e-4 are common

- May want to decrease learning rate over time
  - Divide it by some factor every M updates
  - Set a minimum learning rate

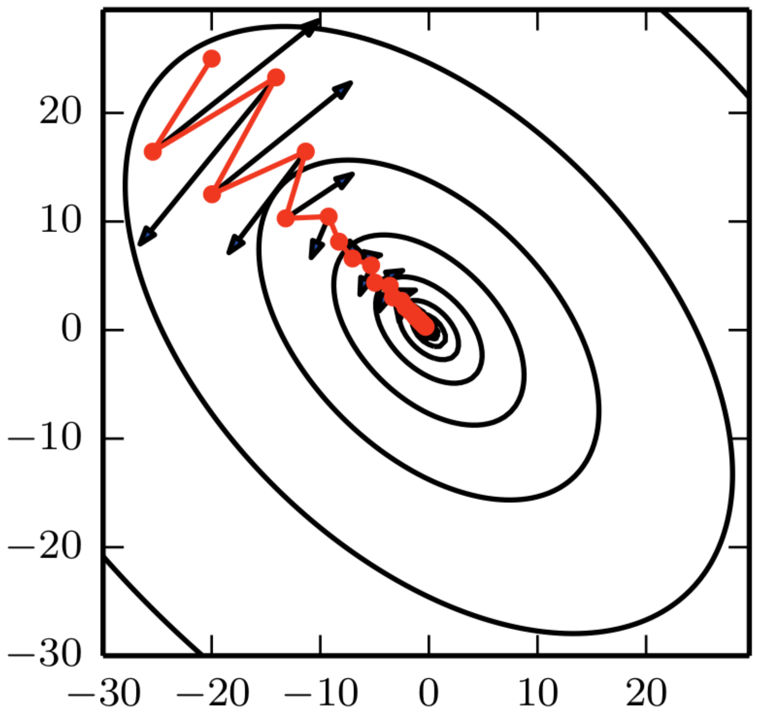
- Adaptive learning rates
  - Various techniques
  - e.g., adapt based on gradient magnitudes

Popular variants of gradient descent

- Momentum
  - Remember previous gradients

$$\Delta \boldsymbol{w} = \alpha \nabla_{\boldsymbol{w}}^{(t-1)} E(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}^{(t)} E(\boldsymbol{w})$$

  - α usually 0.5, 0.9, 0.99
  - Consistent directions over time are kept. Inconsistent directions cancel out



Popular variants of gradient descent

- Rmsprop (root-mean-square prop)
  - Keep sum of the squares of gradients over time
  - Divide gradient by square root of that sum
    - Moves slower along directions of large gradient
  - Different learning rate for each parameter

- Adam (adaptive moments)
  - Combination of momentum and rmsprop
  - Works well in practice

Initialization

- Remember gradient descent is an iterative procedure
  - Weights should have some initial values

- How should we initialize weights?
  - What not to do
    - Initialize to 0
    - Initialize weights for units that have the same inputs to the same value
  - Best practice
    - Initialize weights to small random values
      - e.g., Uniform(0.07), Normal(std=0.01)
    - Can scale them by the number of units in a layer
      - Popular variant: Xavier initialization

Regularization

- A large neural network has high capacity
  - Risk of overfitting
  - Various regularization schemes

- Simple norm-based regularization
  - l_2 regularization, also known as weight decay

$$E(\boldsymbol{w}) + \lambda||\boldsymbol{w}||_2^2$$

  - l_1 regularization

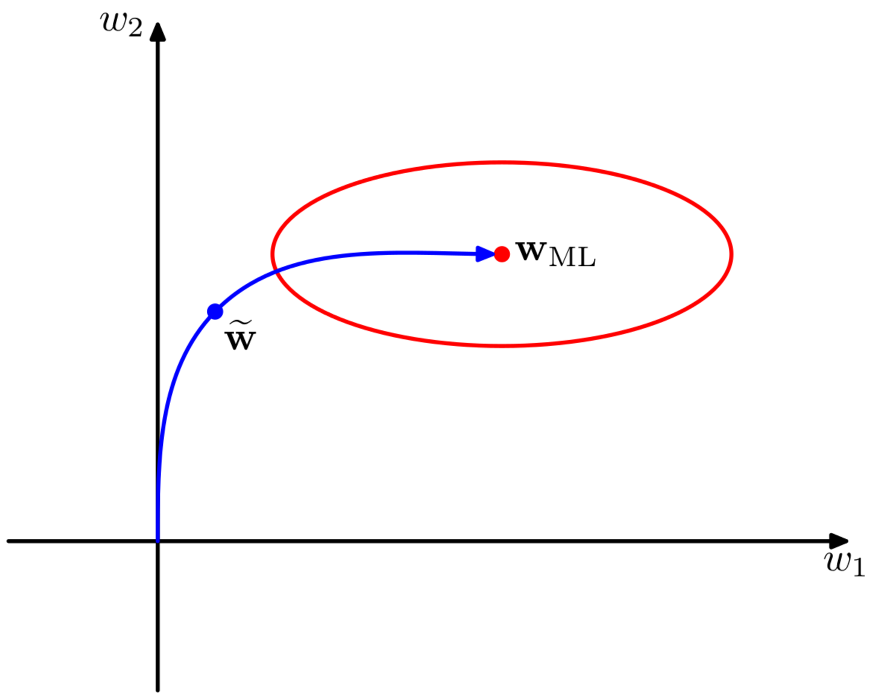
$$E(\boldsymbol{w}) + \lambda||\boldsymbol{w}||_1^2$$

    - Not very common

Early stopping

- Early stopping
  - Simple and efficient way of regularization
  - Watch performance on validation set
    - Stop training when the error on validation set doesn’t improve for K training epochs
  - Similar to weight decay

Epoch: one pass over the training dataset



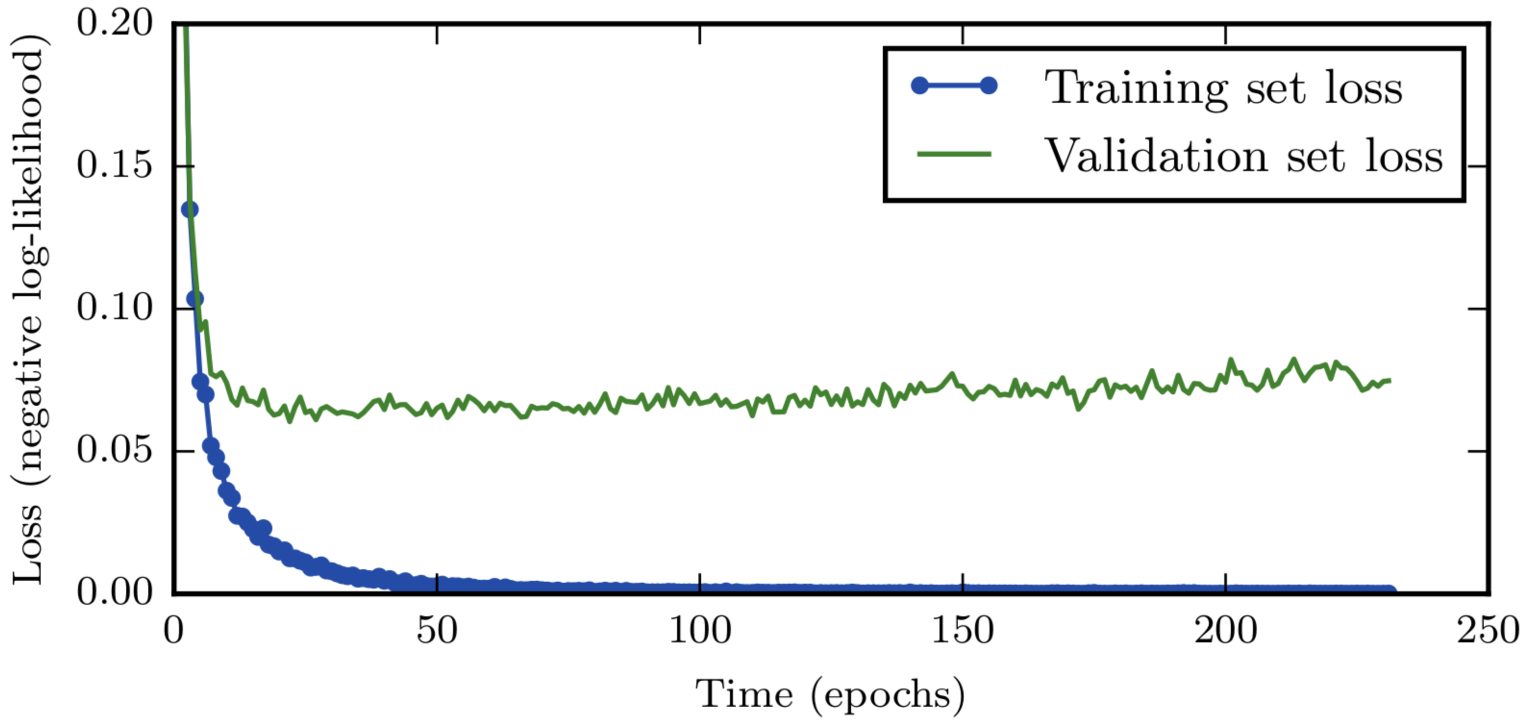


Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or **epochs**). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

Regularization

- Parameter sharing/tying
  - Make some units use the same weights
    - Reduces the number of parameters
  - Good example is convolutional neural networks

- Data augmentation
  - Generate training data making use of domain knowledge
  - e.g., for object recognition, flip/rotate/scale the image.
    - Labels don’t change. Use these as training data.

- Dropout
  - Drop out some units during training randomly
    - During test (evaluation), use all of them
  - Easy to apply and works well

Neural network types

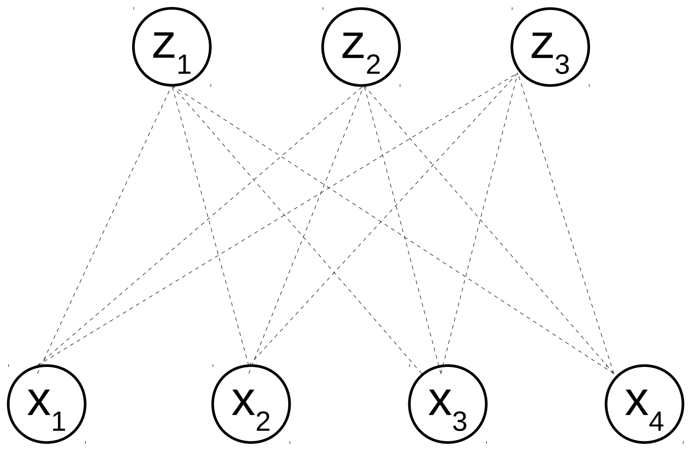
- Three main types
  - Multilayer perceptron (MLP)
  - Convolutional neural networks (CNN)
  - Recurrent neural networks (RNN)

- These days it makes more sense to talk about different types of layers
  - Dense/fully-connected
  - Convolutional
  - Recurrent
    - LSTM, GRU etc.

Dense/fully-connected layers

- Dense/fully-connected layer
  - All inputs are connected to each output unit

$$z_j = \sum_i w_{ij} x_i$$

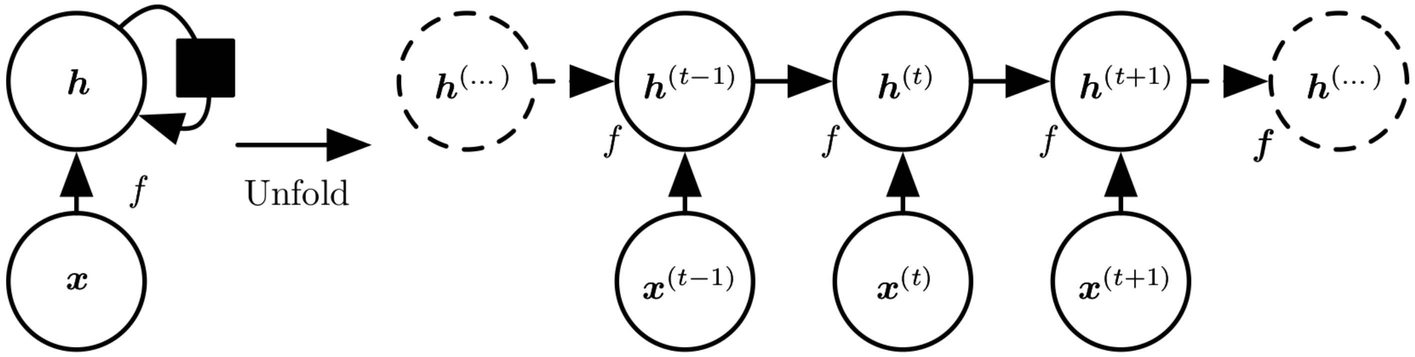


  - Multilayer perceptron (MLP)
    - A network consisting of dense layers

Recurrent layers

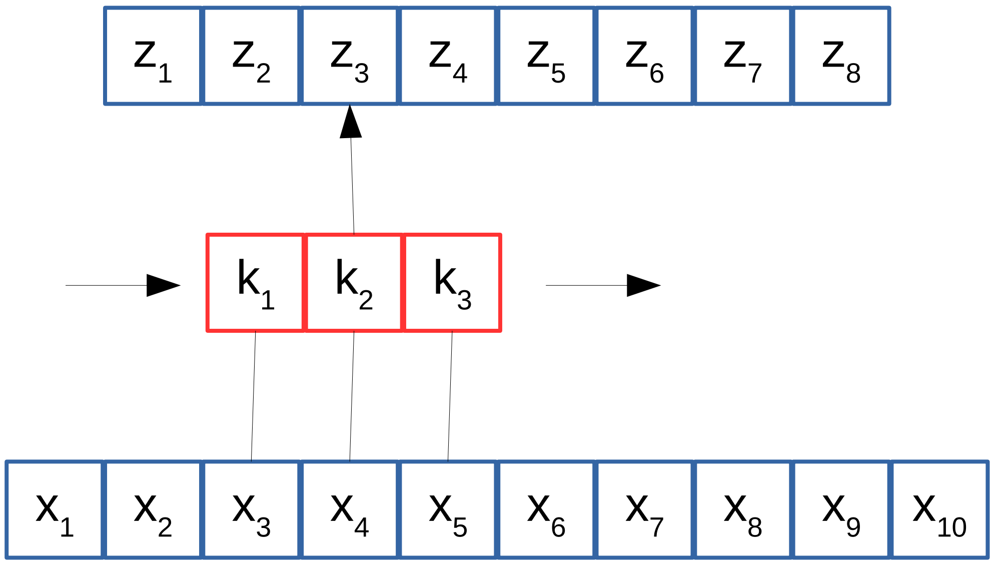
- Recurrent layer
  - Applied to sequence data
    - e.g., natural language, music, video
  - Input is a sequence over time $x^{(1)}, x^{(2)}, x^{(3)}, \ldots$
  - Hidden state of layer at time t: $h^{(t)}$
  - Many types of recurrent layers: LSTM, GRU, etc.

$$\begin{aligned}\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)},\\ \boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}),\end{aligned}$$



Convolutional layers

- Convolutional layer
  - A set of weights applied in a sliding window fashion



z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8
k_1 k_2 k_3
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}

$$z_j = \sum_{i=1}^{3} k_i x_{j+i-1}$$

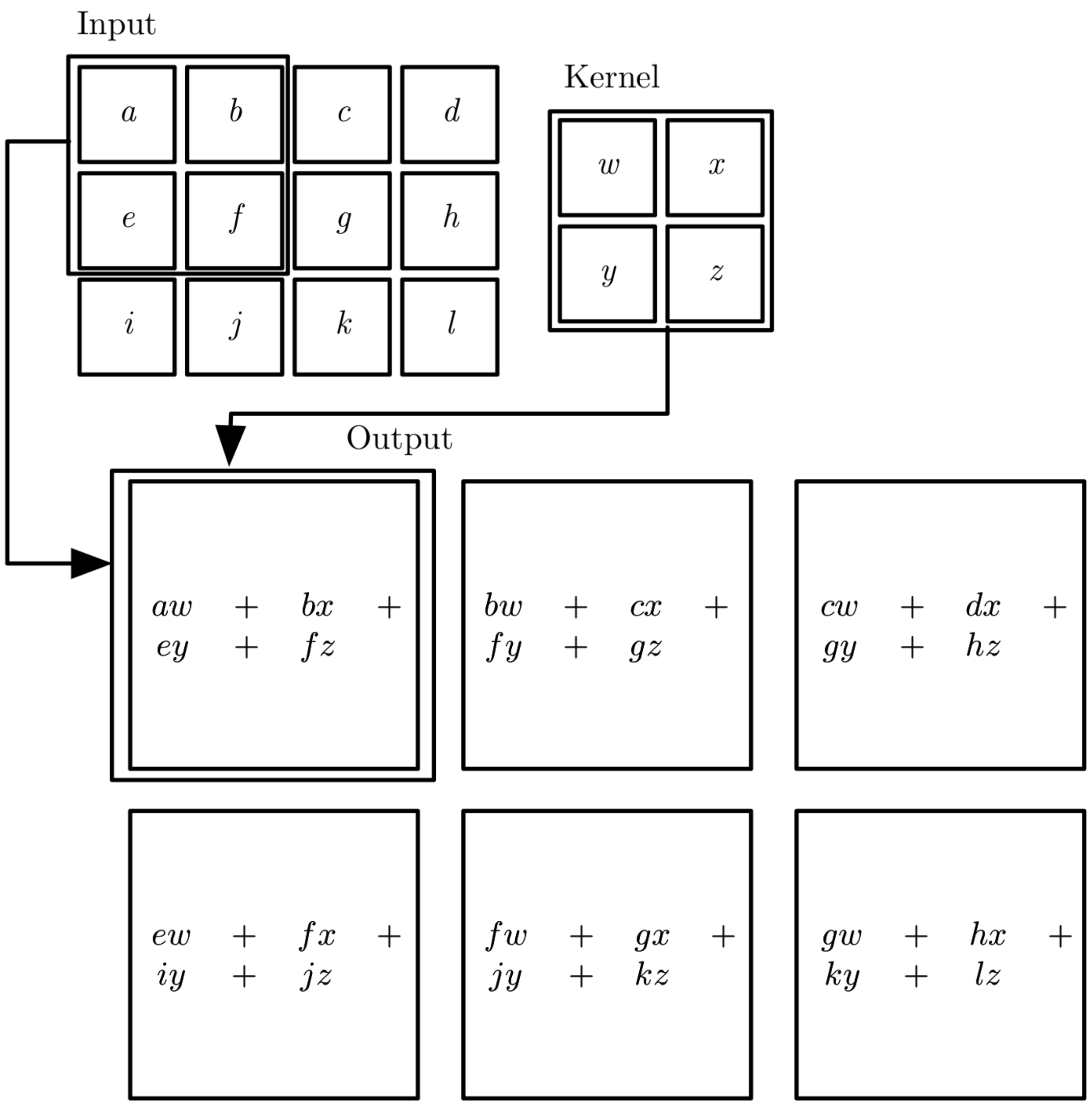
Convolutional layers

- Convolutional layer
  - x: input
  - k: kernel
  - z: feature map (output)

$$z_j = \sum_{i=1}^{3} k_i x_{j+i-1}$$

- This is an example of 1D convolution
  - Usually applied to 2D inputs (e.g., images)

- Note same kernel is applied to different inputs
  - A form of parameter sharing/tying



$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n).$$

Convolutional layers

- Motivation
  - An important feature can be anywhere in the input
    - e.g., an object can be anywhere in the image
  - Building in an assumption about the domain
    - Spatial translation shouldn’t change predictions
      - Invariant to translation

- Works well in practice
  - Computer vision models use CNNs heavily
  - Reduces the number of parameters significantly
    - Compare to a fully connected layer

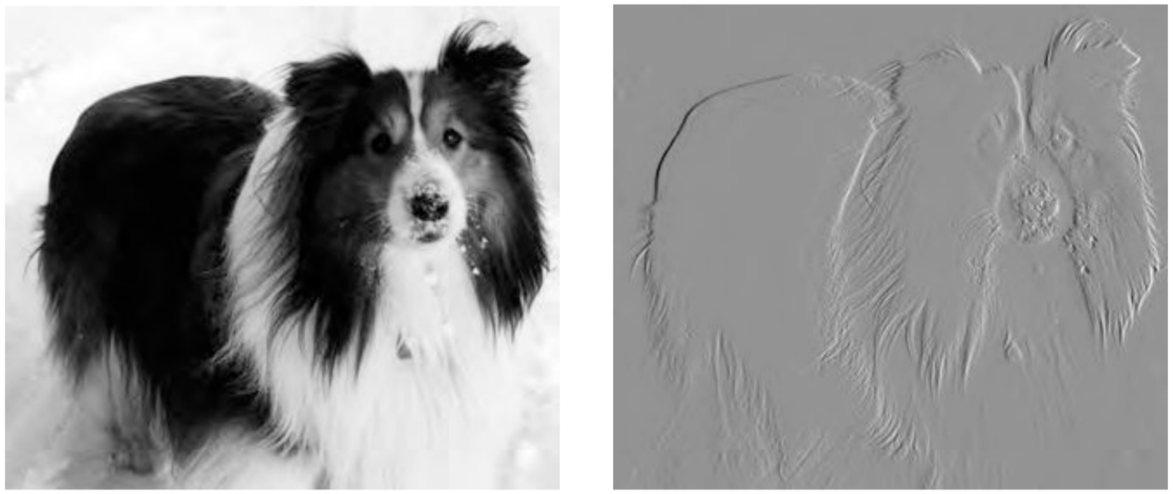
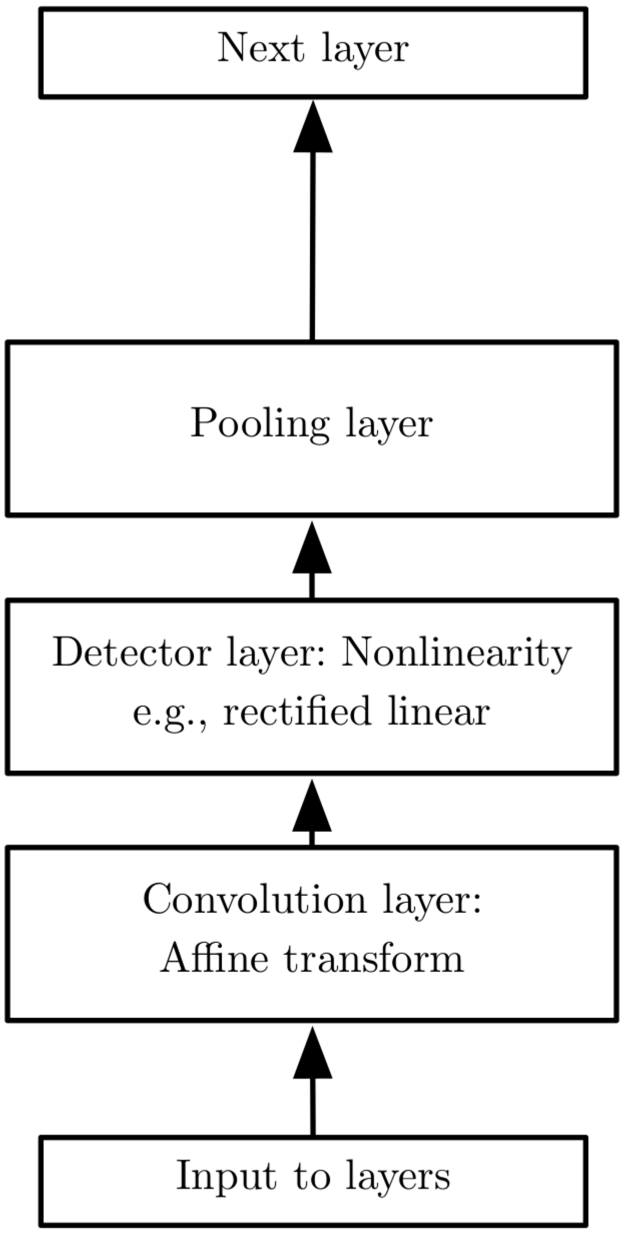


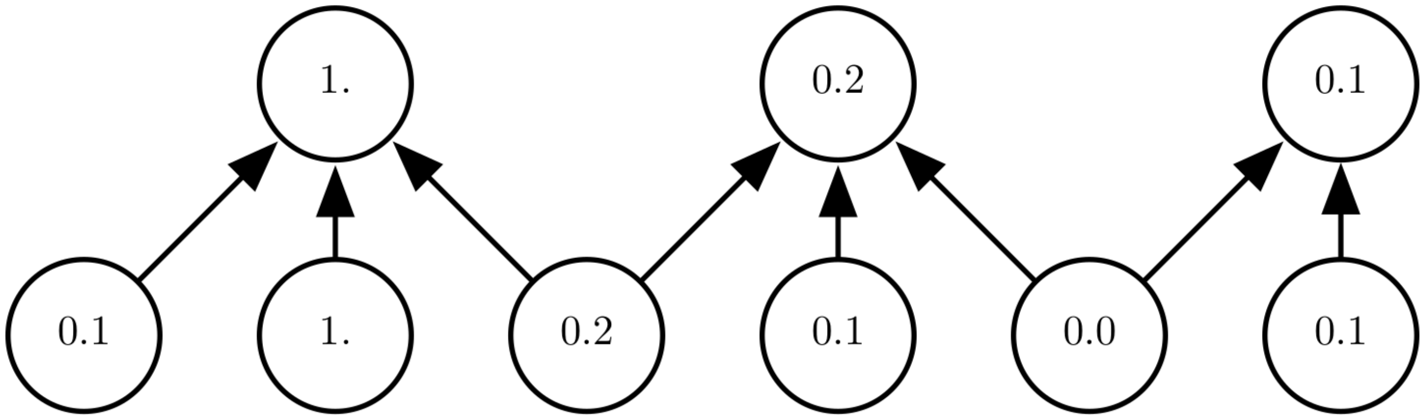
Figure 9.6: Efficiency of edge detection. The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide, while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing two elements, and requires $319 \times 280 \times 3 = 267,960$ floating-point operations (two multiplications and one addition per output pixel) to compute using convolution. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over eight billion, entries in the matrix, making convolution four billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over sixteen billion floating point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating-point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small local region across the entire input. Photo credit: Paula Goodfellow.

Simple layer terminology

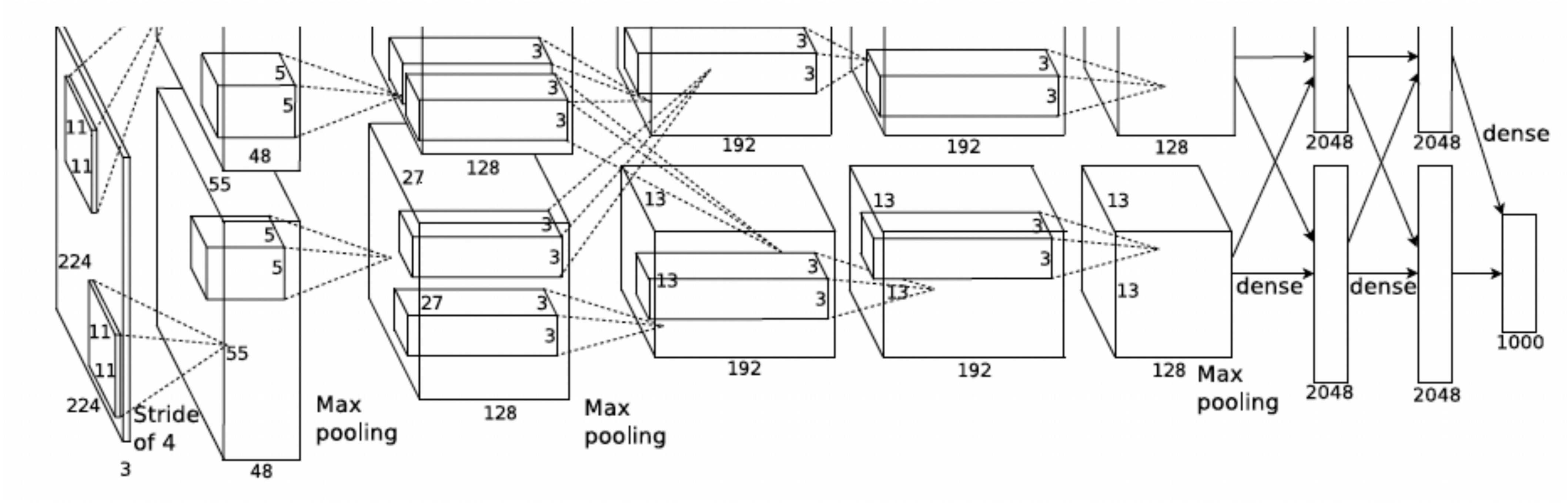


Pooling layer

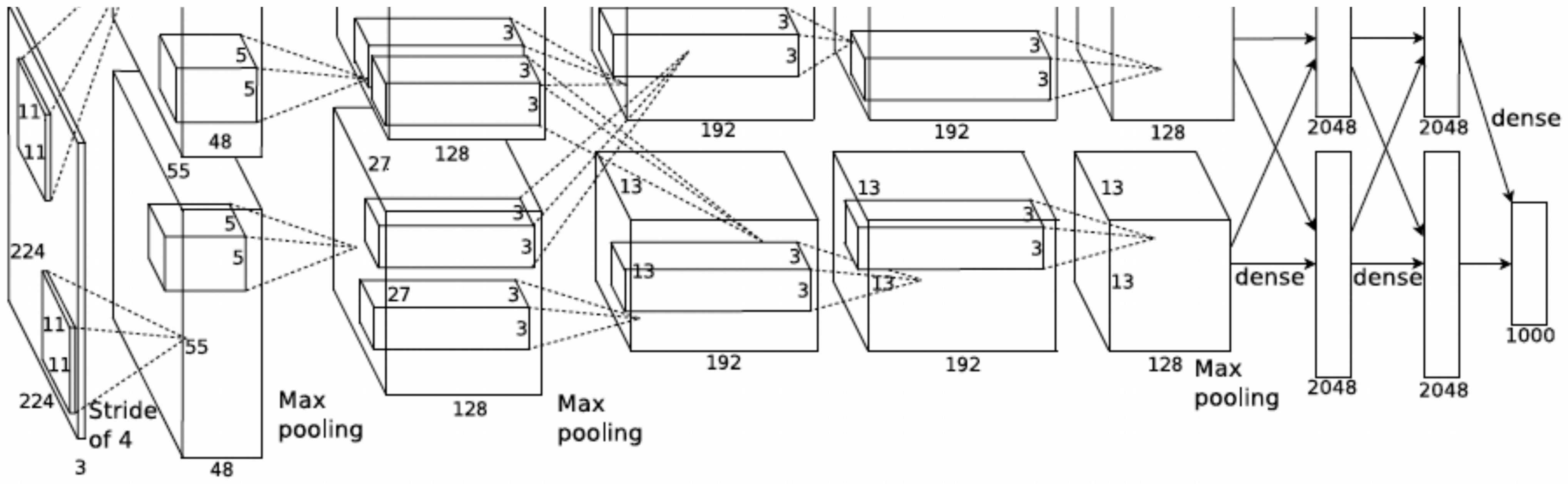
- Pooling layer
  - Used generally with convolutional layers
  - Slide a window and apply some function to map it to a single value
    - Max-pooling
    - Average-pooling
  - Move window by s steps to downsample by s
    - s is called the stride

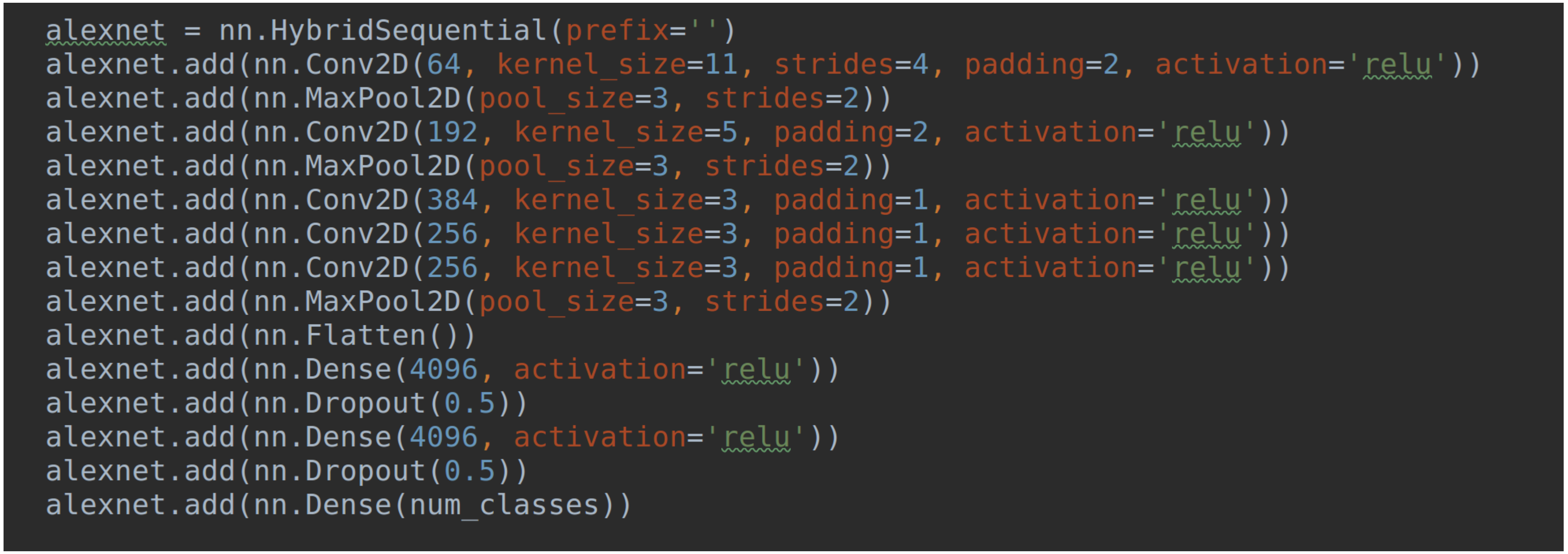


Max-pooling with s=2



AlexNet, ImageNet challenge 2012 winner





```
alexnet = nn.HybridSequential(prefix='')
alexnet.add(nn.Conv2D(64, kernel_size=11, strides=4, padding=2, activation='relu'))
alexnet.add(nn.MaxPool2D(pool_size=3, strides=2))
alexnet.add(nn.Conv2D(192, kernel_size=5, padding=2, activation='relu'))
alexnet.add(nn.MaxPool2D(pool_size=3, strides=2))
alexnet.add(nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'))
alexnet.add(nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'))
alexnet.add(nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'))
alexnet.add(nn.MaxPool2D(pool_size=3, strides=2))
alexnet.add(nn.Flatten())
alexnet.add(nn.Dense(4096, activation='relu'))
alexnet.add(nn.Dropout(0.5))
alexnet.add(nn.Dense(4096, activation='relu'))
alexnet.add(nn.Dropout(0.5))
alexnet.add(nn.Dense(num_classes))
```

Summary

- Training a neural network
  - Stochastic and minibatch gradient descent
  - Gradient descent variants
    - Momentum
    - Rmsprop
- Initialization
- Regularization
  - Weight decay
  - Early stopping
  - Dropout
- Neural network layer types
  - Fully-connected, recurrent, convolutional

- No exercises!

References

[1] Goodfellow I., Bengio Y., Courville A. Deep Learning.
https://www.deeplearningbook.org/

[2] Bishop C. Pattern Recognition and Machine Learning. Chapter 5.