Computer Vision and Deep Learning

Adrian Opheim

April 11, 2019

Contents

1 Lecture 2: Image Classification Pipeline

- Image classification: Given a set of labels, choose the one that classifies the image.
- **The semantic gap:** The difference between our semantic understanding of the image and what the computer sees.
- **Data-driven approach:** Provide the computer with many examples of each class and then develop learning algorithms that look at these examples and learn about the visual apperance of each class.
- **Nearest neighbour**: Given a test image, find the most similar training images. Will find the closest example from the training set. Will find the visually most similar image.
- **Distance metric to compare images:** Figure 1 shows the L_1 distance, given by

$$d_1(I_1, i_2) = \sum_p |I_1^p - I_2^p| \tag{1.1}$$

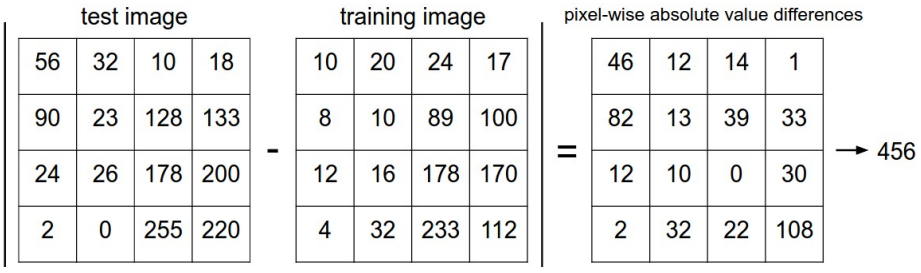


Figure 1: Distance metric

```
1  import numpy as np
2
3  class NearestNeighbor(object):
4    def __init__(self):
5      pass
6
7    def train(self, X, y):
8      """ X is N x D where each row is an example. Y is 1-dimension of
           size N """
9      # the nearest neighbor classifier simply remembers all the
           training data
10     self.Xtr = X
11     self.ytr = y
12
13   def predict(self, X):
14     """ X is N x D where each row is an example we wish to predict
           label for """
15     num_test = X.shape[0]
16     # lets make sure that the output type matches the input type
17     Ypred = np.zeros(num_test, dtype = self.ytr.dtype)
18
19     # loop over all test rows
20     for i in xrange(num_test):
21       # find the nearest training image to the i'th test image
22       # using the L1 distance (sum of absolute value differences)
```

```
23        distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
24        min_index = np.argmin(distances) # get the index with smallest
              distance
25        Ypred[i] = self.ytr[min_index] # predict the label of the
              nearest example
26
27    return Ypred
```

- Not really good, because you want the classifying to be fast, and training can be slow, aka the reverse of this technique.
- The L_2 distance is the Euclidean distance, given by

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2} \tag{1.2}$$

1.1 k-Nearest Neighbour

- Instead of finding the single closest image in the training set, we will find the top k closest images, and have them vote on the label of the test image.

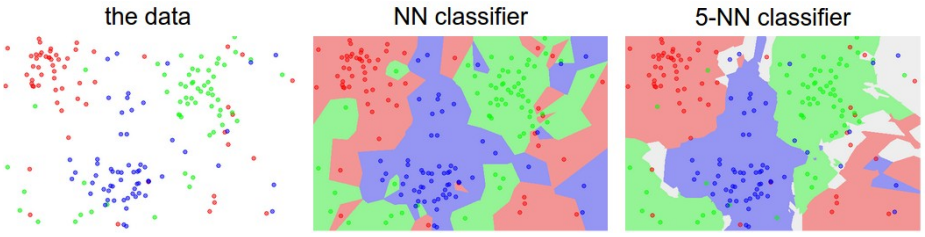


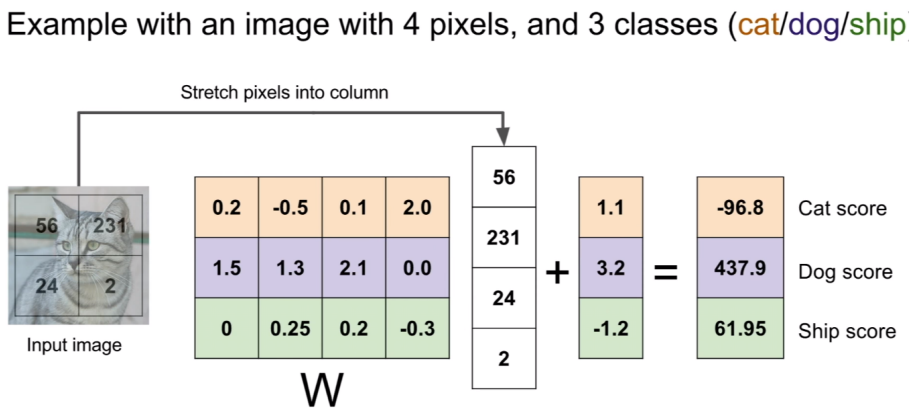
Figure 2: K-Nearest neighbours. An example of the difference between Nearest Neighbor and a 5-Nearest Neighbor classifier, using 2-dimensional points and 3 classes (red, blue, green). The colored regions show the decision boundaries induced by the classifier with an L2 distance. The white regions show points that are ambiguously classified (i.e. class votes are tied for at least two classes). Notice that in the case of a NN classifier, outlier datapoints (e.g. green point in the middle of a cloud of blue points) create small islands of likely incorrect predictions, while the 5-NN classifier smooths over these irregularities, likely leading to better generalization on the test data (not shown). Also note that the gray regions in the 5-NN image are caused by ties in the votes among the nearest neighbors (e.g. 2 neighbors are red, next two neighbors are blue, last neighbor is green).

- **Hyperparameters:** What is the best value of `k/distance` to choose for your algorithm? Very problem-specific. Try and fail.
- **Important:** We cannot use the test set for the purpose of tweaking hyperparameters. The test set should be seen as a very precious resource that should ideally never be touched until one time at the very end.
- To set hyperparameters, split the data in three: **train, validation, test**. The validation test is then used as a "fake" test to tune the hyper-parameters. Choose hyperparameters on validation, and evaluate on test data.

1.2 Linear classifiers

- In the parametric approach, we summarize our test data in the weights `W`, and no longer need the test data when testing. Improves run time.

- The linear classifier uses $f(x, W) = Wx + b$ where $W = (10 \times 3072), x = (3072 \times 1), b = (10 \times 1)$



- The linear classifier can only use one template for each category.

2 Lecture 22. January

- Deep learning is a subfield of machine learning and artificial intelligence.
- Computer vision: Makes the computer able to see. AI makes the computer think. AI tries to automate intellectual tasks normally performed by humans.
- Machine learning: Learn by examples. The sytem is **trained** instead of explicitly programmed.
- **Deep learning:** Learning successive **layers** of increasingly meaningful representations (depth of the model). It learns representations via models called **neural networks**.

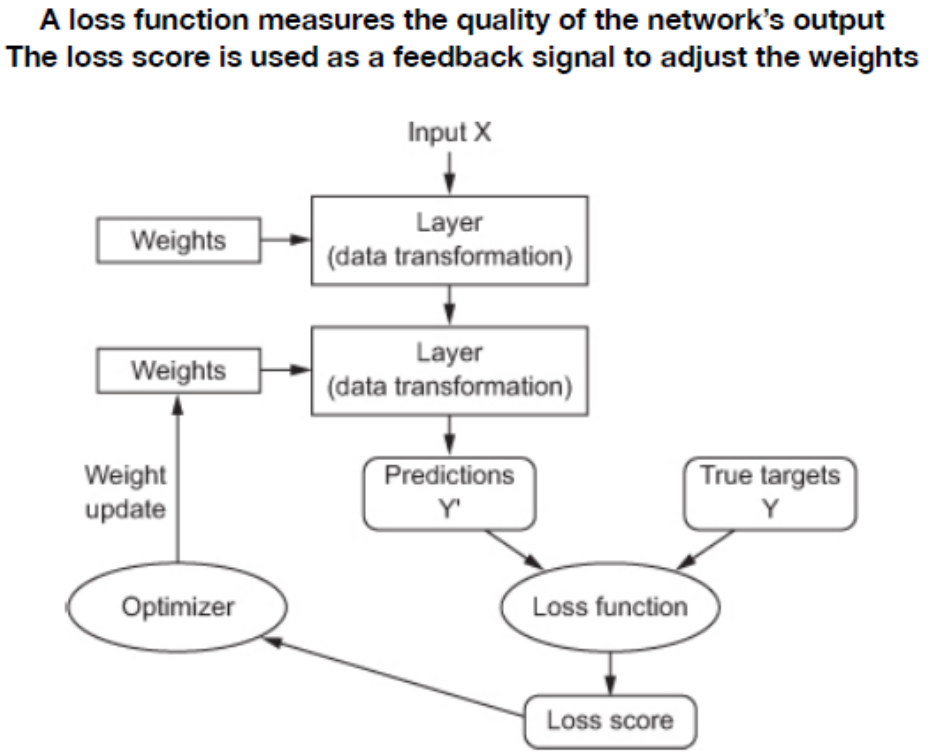


Figure 3: Loss function in a neural network

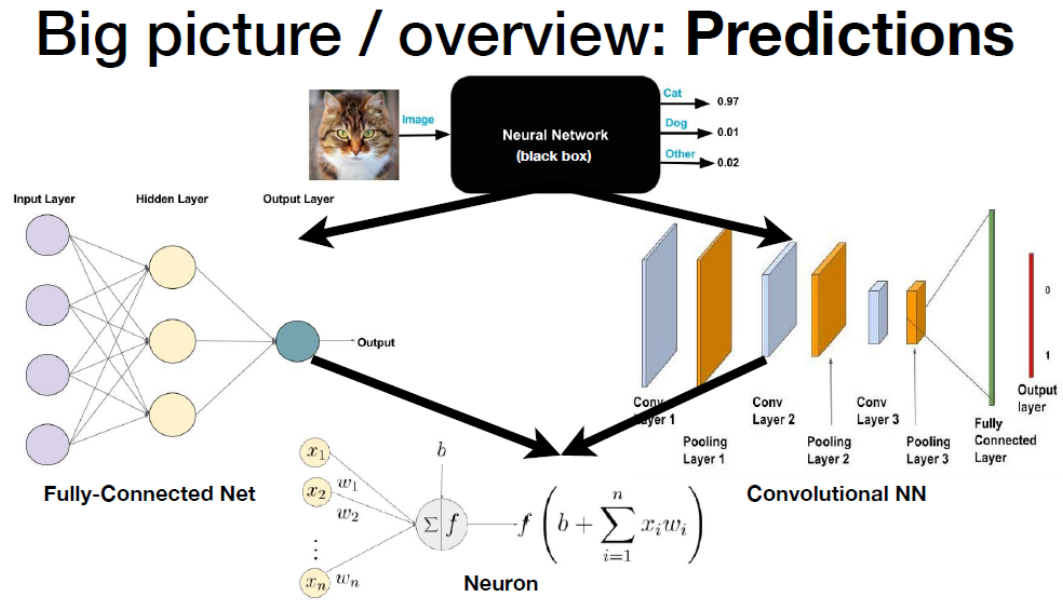


Figure 4: Preconditions

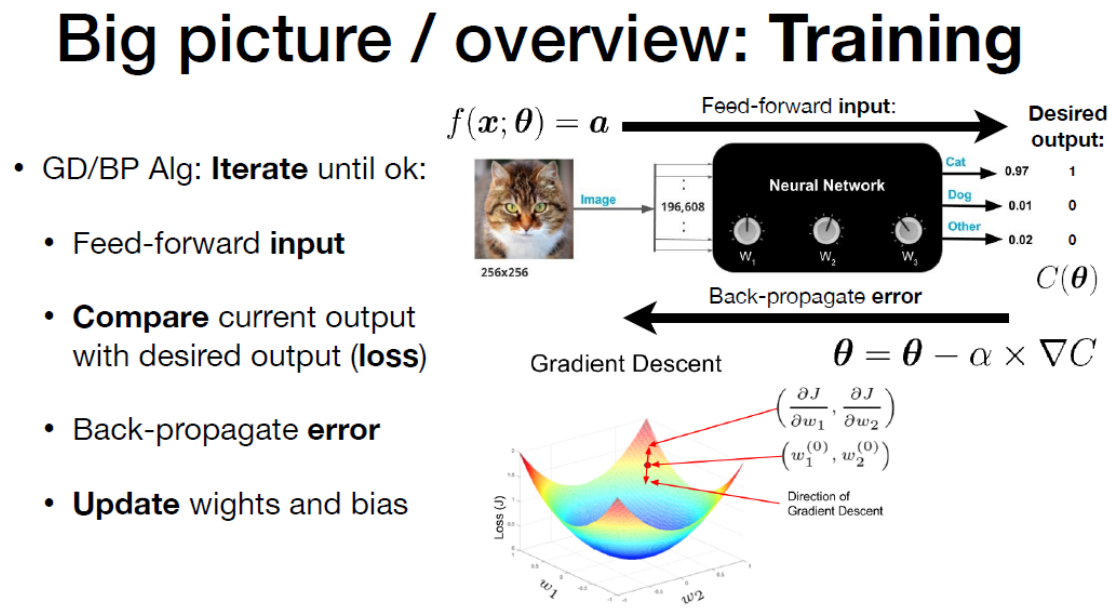


Figure 5: Training a neural network

2.1 Shallow Fully-Connected Feed-Forwards

- An Artificial Neural Network (ANN) consists of multiple **layers**. Each layer consists of multiple **neurons.**
- Fully connected network: A given neuron is connected to all neurons in previous and next layer. **Feed-forward:** ANNs: Only forward links.

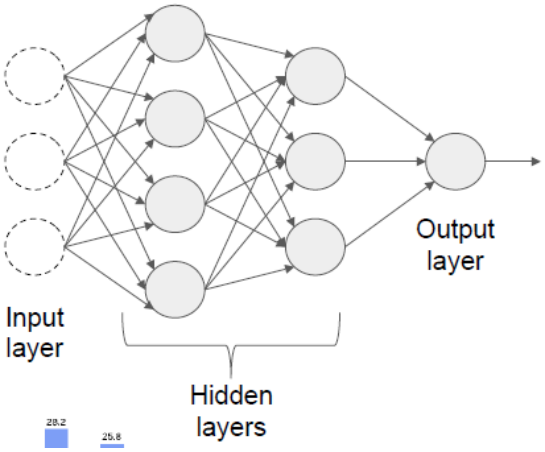


Figure 6: Layers and neurons in an ANN.

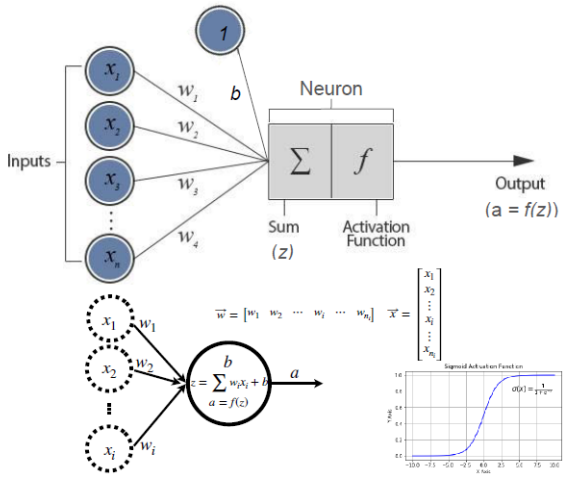


Figure 7: Artificial Neurons

- Each neuron have two operations: A weighted input, $z = Wx + b$. Activation: $a = f(z)$. The b is the bias that tells us how much we should remove of the weights. Weights and bias are learnable parameters.
- The bias is the b term in $y = ax + b$. Meaning the intersection with the y axis.

2.2 Activation Functions

- Rectified Linear Unit. $f(x) = max(0, x)$

$$f(x) = \max(0, x)$$

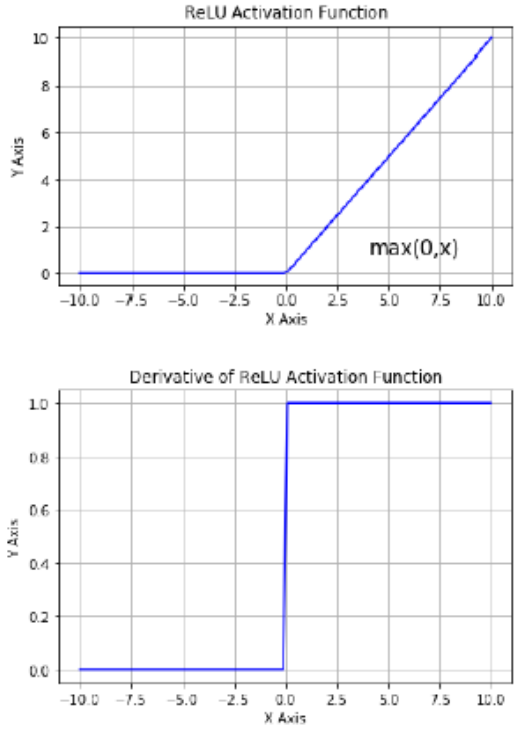


Figure 8: Rectified Linear Unit (ReLU)

2.3 Matrix-based Notation

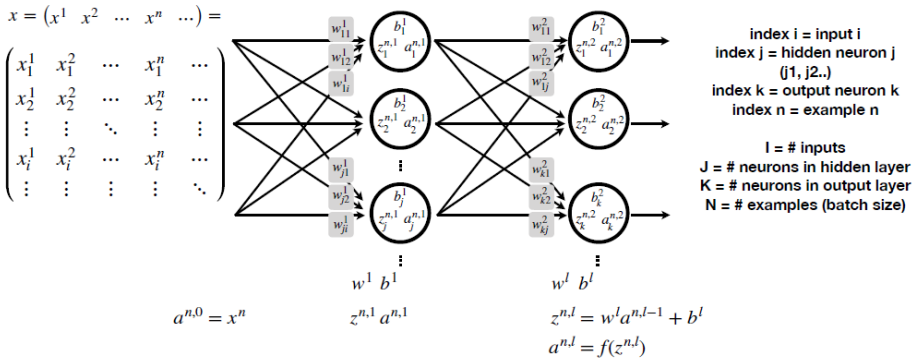


Figure 9: The matrix-based notation used.

3 Chapter 1: Using neural networks to recognize handwritten digits

3.1 Perceptrons

- The main neuron model used today is called the *sigmoid neuron.*
- A perceptron takes several binary inputs $x_1, x_2, x_3, ...$ and produces a single binary output:

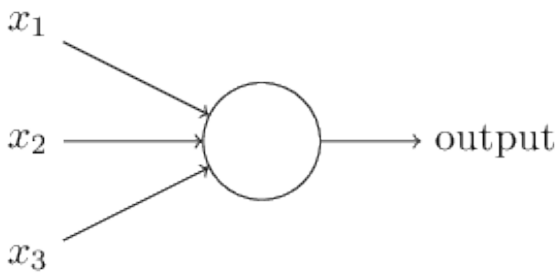


Figure 10: A perceptron

- By introducing *weights* $w_1, w_2, w_3, \ldots$ corresponding to the respective input's importance to the output.
- The output is 1 if the sum of the input multiplied with its weight is greater than a threshold value:

$$\text{output} = \begin{cases} 0 & \text{if} \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \sum_j w_j x_j \geq \text{threshold} \end{cases} \tag{3.1}$$

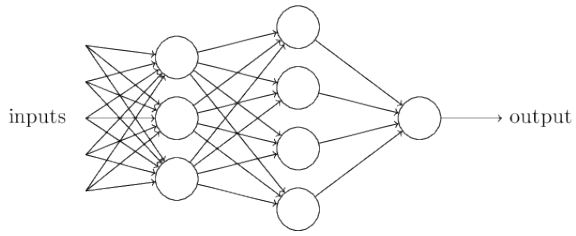


Figure 11: The coloumns are here perceptrons. Each one has still a single output. The arrows indicate that the output from a perceptron is being used as the input to several other perceptrons.

- Rewriting Eq. 3.1 by saying $\sum_j w_j x_j = w \cdot x$, where w and x are vectors, whose components are the weights and inputs, respectively. We can also move the threshold over, and then call it the perceptron's *bias*. $b =$ threshold. The calculation of a single perceptron's output can the be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases} \tag{3.2}$$

3.1.1 Simple perceptron example

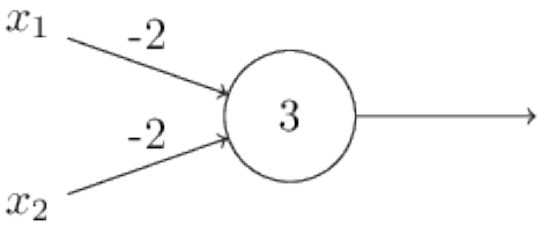


Figure 12: Example of a simple calculation

- If we use the input 00, we get the output 1. That is because $-2 \cdot 0 + (-2) \cdot 0 + 3 = 3$, which is positive.
- Further on, we can see that input 01 and 10 produces output 1. But input 11 produces output 0: $-1 \cdot 1 + (-2) \cdot 1 + 3 = -1$. Therefore, our perceptron implements a NAND gate!
- Using this, we see that it is possible to for example build a circuit that adds two bits, x_1 and x_2 using perceptrons set as NAND gates.

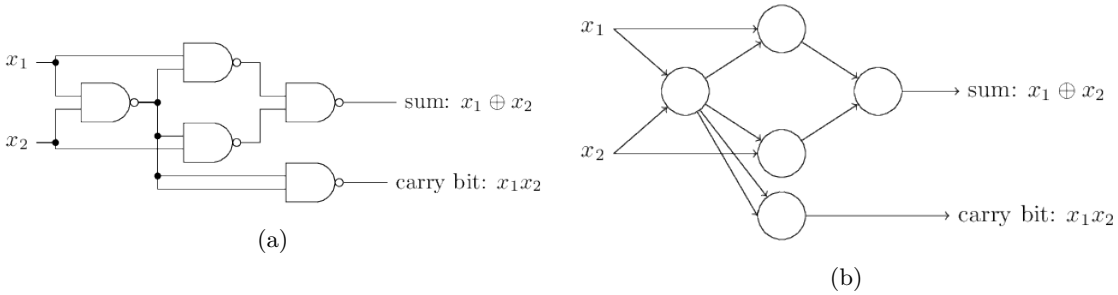


Figure 13: The corresponding network of NAND perceptrons to make a adding bits network

3.2 The sigmoid neuron

- In order to make a neural network with perceptrons learn, we must introduce a sigmoid neuron:

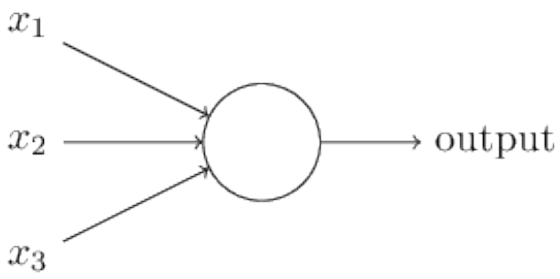


Figure 14: A sigmoid neuron

- The sigmoid has the same inputs $x_1, x_2, x_3, \ldots$ but instead of the *input* just being 0 and 1, they can take any values *between* 0 and 1. 0.638 is therefore a valid input for a sigmoid neuron.
- The sigmoid has also weights $w_1, w_2, w_3, \ldots$ corresponding to the input and a bias b. But the output is $\sigma(w \cdot x + b)$, where σ is called the sigmoid function, defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.3}$$

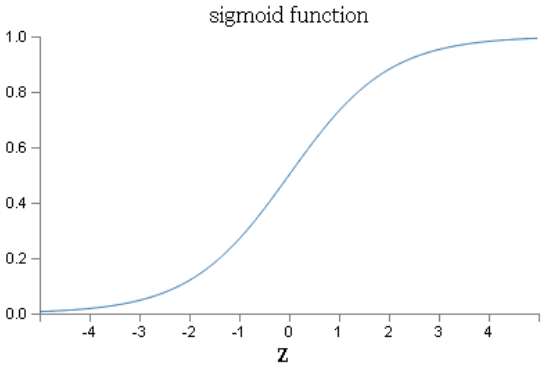


Figure 15: The sigmoid function. We see that when z is large, $\sigma(z) \approx 1$ and when z is small, $\sigma(z) \approx 0$. It is ony when $z = wx + b$ is of modest size, there is much deviation from the perceptron model. We see that the sigmoid gives a "smoothed out" perceptron.

- The smoothness of σ means that small changes in the weights, Δw_j and Δb in the bias will produce a small change Δoutput in the output from a neuron. This means that the output can be approximated by

$$\Delta\text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b \tag{3.4}$$

- This tells us that Δoutput is a linear function of the changes in the weights and bias.
- Sigmoid neurons will output any real number between 0 and 1.

3.3 The architecture of neural networks

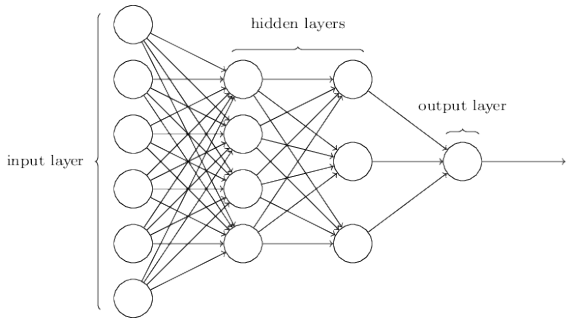


Figure 16: Notation for a four-layer neural network.

- If we wanted our neural network to determine the value of a handwritten number, we would use the pixels of the image in the input layer. If it is a 64×64 greyscale image, we would have $64 \times 64 = 4096$ input neurons, with intensities scaled appropriately between 0 and 1.
- Up to now, we have seen *feed-forward* neural networks, where output from one layer is used as input in the next layer. Infromation is always fed forward, not back.
- In **recurrent neural networks**, feedback loops is possible.
- A simple neural network for recognizing hand-written digits could be:

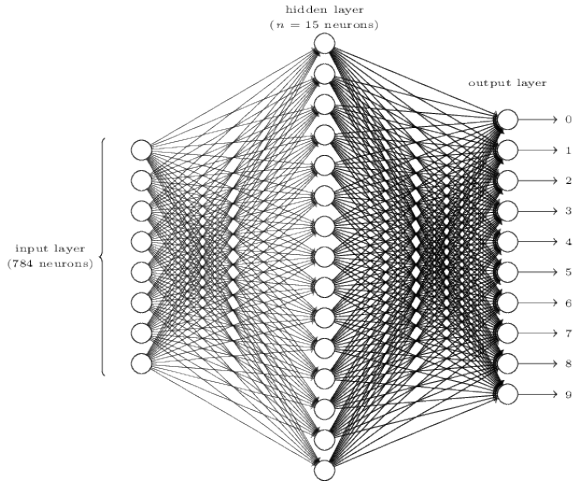


Figure 17: Three-layered neural network for recognizing hand-written digits.

- The input pixels are greyscale, where a value of 0.0 means white, and 1.0 represents black.
- A training input will have the notation x. x will be a $28 \times 28 = 784$-dimensional vector, where each entry represents the grey value for a single pixel in the image. The corresponding **output** will be denoted $y = y(x)$, where y is a 10-dimensional vector. For example, if the training image x represents a 6, then $y(x) = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]^T$ is the desired output from the network.

3.4 Cost function

- We want to check how well the output from the network aproximates $y(x)$ for all training inputs x. The**cost function** C quantifies this:

$$C(w,b) = \frac{1}{2n} \sum_x ||y(x) - a||^2 \tag{3.5}$$

w: all the weights in the network
b: all the biases
n: the total number of training inputs
a: the vector of outputs from the network when x is input.

- We call C the quadratic cost function. It is also sometimes known as the *mean squared error*, MSE.
- $C(w,b)$ is non-negative, since every term in the sum is non-negative.
- $C(w,b) \approx 0$ precisely when $y(x)$ is approximately equal to the output, a, for all the training inputs x.
- Therefore, we would like to minimize the cost function C as a function of the weights and biases. This is done by an algorithm called *gradient descent*.
- In total: The goal in training a neural network is to find weights and biases which minimize the quadratic cost function $C(w,b)$

3.5 Minimizing the cost function $C(v)$

- Here, the w notation is replacing w and b, just to emphasize that this could be any function. Meaning v could be a function of many variables: $v = v_1, v_2, ...$

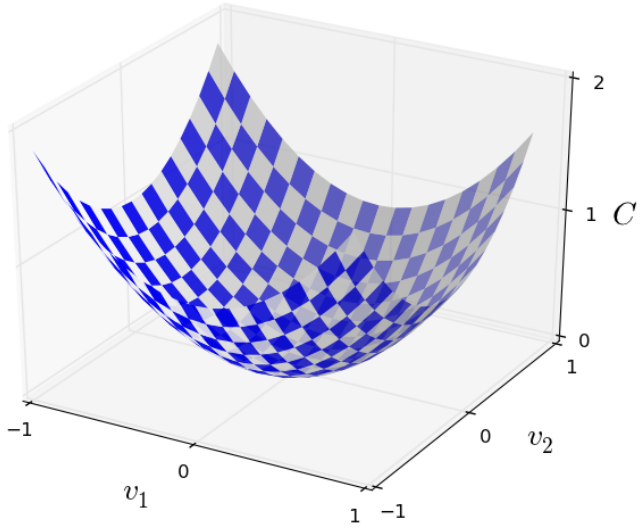


Figure 18: The cost function $C(v)$ in which we want to find a minimum

- One might think of the task of minimizing the cost function as similar to letting a ball roll down the function, as it will then find the bottom.

- Moving the ball a small amount Δv_1 in the v_1 direction and Δv_2 in the v_2 direction:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{3.6}$$

- This means that we may write the gradient of C to be the vector of partial derivatives:

$$\nabla C = \begin{pmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{pmatrix} \tag{3.7}$$

- Using this, we may rewrite Eq. 3.6 to

$$\Delta C \approx \nabla C \cdot \Delta v \tag{3.8}$$

- We see then that ∇C relates changes in v to changes in C.

  In order to make ΔC negative, we may choose *Deltav* as

$$\Delta v = -\eta \nabla C \tag{3.9}$$

  where η is a small, positive parameter known as the *learning rate*. This is used to express the "law of motion" for the ball in the gradient ascent algorithm. In the way that we will calculate a vlaue for Δv, then move the ball's positon v by that amount:

$$v \to v' = v - \eta \nabla C \tag{3.10}$$

  We use this update rule again and again, to make another move. Until we reach a global minimum - hopefully.

- The gradient descent algorithm will work, even if C is a function of many variables.

- For a neural network, gradient descent algorithm is used to find the weights w_k and biases b_l which minimize the cost function $C(w, b)$. The gradient descent update rule then becomes

$$\begin{aligned} w_k \to m'_k &= w_k - \eta \frac{\partial C}{\partial w_k} \\ b_l \to b'_l &= b_l - \eta \frac{\partial C}{\partial b_l} \end{aligned} \tag{3.11}$$

3.6 Using the MNIST database

4 Chapter 2: How the backpropagation algorithm works

- This chapter is about how one can compute the gradient of the cost function C. The **backpropagation algorithm** is used for finding such gradients.

- The algorithm gives us a detailed insight in how changing the weights and biases changes the overall behaviour of the network.

4.1 Notation

$$w^l_{jk} \tag{4.1}$$

This is the weight for the connection *from* the k^{th} neuron in the $(l-1)^{\text{th}}$ layer *to* the j^{th} neuron in the l^{th} layer.

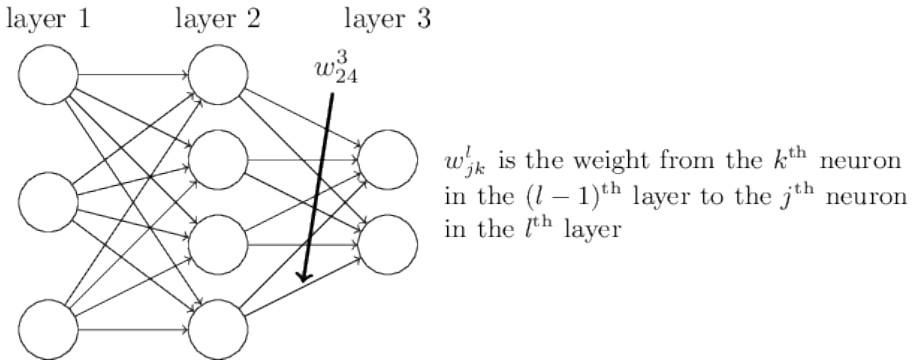


Figure 19: The weight of a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network

- A similar notation is used for the network's biases and activations.

$$b^l_j \tag{4.2}$$

This is the j^{th} bias in the l^{th} layer.

$$a^l_j \tag{4.3}$$

This is the activation of the j^{th} neuron in the l^{th} layer.

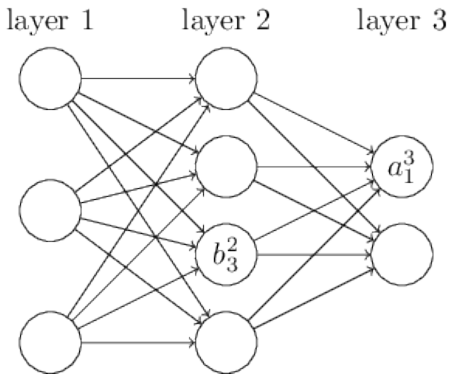


Figure 20: Example of the notation used for biases and activation.

The activation a^l_j is related to the activations in the $(l-1)^{\text{th}}$ layer by the equation

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) \tag{4.4}$$

where the sum is over all neurons k in the $(l-1)^{\text{th}}$ layer.

- To rewrite this in matrix form, we define a *weight matrix* w^l for each layer l. All entries are the weights connecting to the l^{th} layer of neurons. Meaning that the entry in the j^{th} row and k^{th} column is w^l_{jk}.
- Similarly, we define a *bias vector* b^l for each layer. The components are the values b^l_j, one component for each neuron in the l^{th} layer.
- The *activation vector* a^l whose components are the activations a^l_j

- When vectorizing a function like σ, we want to apply the function to every element in a vector v. $\sigma(v)$ is used to denote this elementwise application of a function. If we have the function $f(x) = x^2$ then the vectorized form of f has the effect

$$f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix} \tag{4.5}$$

- Then we can rewrite Eq. 4.4 to the vectorized form

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{4.6}$$

- We introduce $z^l = w^l a^{l-1} + b^l$, and call z^l the *weighted input* to the neurons in layer l. z^l has components $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$

4.2 The cost function used in backpropagation algorithm

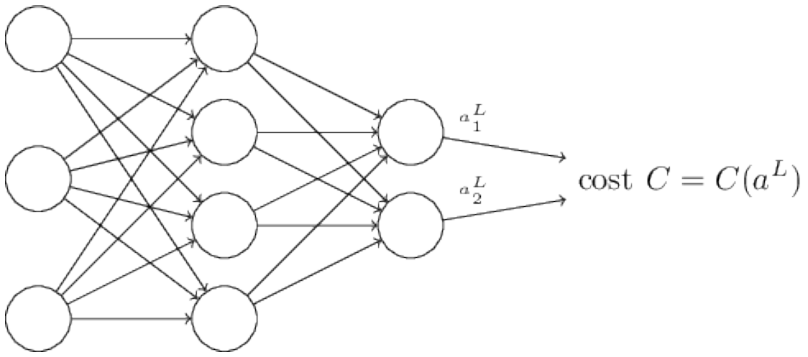
- We will use the quadratic cost function, which has the form

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{4.7}$$

n: total number of training examples
x: individual training samples. The sum is over these.
$y = y(x)$: the corresponding desired output.
L denotes the number of layers in the network
$a^L = a^L(x)$: is the vector of activations output from the network when x is input.

- We assume that the cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over the cost functions C_x for individual training examples, x.

- We also assume that the cost can be written as a function of the outputs from the neural network:



4.3 The Hadamard product $s \odot t$, elementwise product of two vectors

The components of $s \odot t$ are just $(s \odot t)_j = s_j t_j$. For example:

$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1 \cdot 3\\2 \cdot 4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix} \tag{4.8}$$

4.4 The four fundamental equations behind backpropagation

- We introduce an intermediate quantity δ_j^l which we call the *error* in the j^{th} neuron in the l^{th} layer.
- Looking at one neuron j in layer l, a little change Δz_j^l is added to the neuron's weighted input. So that instead of outputting $\sigma(z_j^l)$, the neuron instead outputs $\sigma(z_j^l + \Delta z_j^l)$. This propagates through the network, causing the overall cost to change by an amount $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$.
- One could therefore interpret $\frac{\partial C}{\partial z_j^l}$ to being the measure of the error in the neuron, defined as the quantity δ_j^l:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{4.9}$$

4.4.1 The error in the output layer, δ^L

The components of δ^L are given by

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{4.10}$$

$\frac{\partial C}{\partial a_j^L}$ measures how fast the cost is changing as a function of the j^{th} output activation.
$\sigma'(z_j^L)$ measures how fast the activation function σ is changing at z_j^L.

- The component $\frac{\partial C}{\partial a_j^L}$ is normally quite easy to calulate. Using the quadratic cost function, $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$, and so $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$
- The matrix-based form of Eq. 4.10 is

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{4.11}$$

Here, $\nabla_a C$ is a vector whose components are the partial derivatives $\frac{\partial C}{\partial a_j^L}$. May be thought of as expressing the rate of change of C with respect to the output activations.

- When using the quadratic cost function, we have $\nabla_a C = (a^L - y)$, giving the fully matrix-based form

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \tag{4.12}$$

4.4.2 The error δ^l in terms of the error in the next layer δ^{l+1}

$$\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l) \tag{4.13}$$

Suppose we know the error δ^{l+1} at the $(l+1)^{\text{th}}$ layer. When applying the transpose weight matrix $(w^{l+1})^T$, we can think of it as moving the error *backward* throught the network, giving us a measure of the error at the output of the l^{th} layer. Then the product $\odot\sigma'(z^l)$ moves the error backward through the activation function in layer l, giving us the error δ^l in the weighted input to layer l.

- By combining Eq. 4.11 with Eq. 4.13, we can compute the error δ^l for any layer in the network. By using (4.11) to compute δ^L, then using (4.13) to compute δ^{L-1}. Then we can continue using (4.13) to find δ^{L-2}, and so on.

4.4.3 The rate of change of the cost with respect to any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{4.14}$$

- This means that the error of neuron j in layer l is *exactly equal* to the rate of change $\frac{\partial C}{\partial b_j^l}$

4.4.4 The rate of change of the cost with respect to any weight in the network

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{4.15}$$

- Can be rewritten in a less index-heavy notation as

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}} \tag{4.16}$$

where a_{in} is the activation of the neuron input to the weight w, and δ_{out} is the error of the neuron output from the weight w.

4.5 Insights taken from the backpropagation formulas

- Looking at the output layer, and the term $\sigma'(z_j^L)$. From the graph of the sigmoid function, we know that the graph is very flat when $\sigma(z_j^L)$ is approximately 0 or 1. Meaning that in these cases, we have $\sigma'(z_j^L) \approx 0$. In this case we say the neuron has *saturated*, and as a result, the weight has stopped learning (or is learning slowly).
- Using this, we see from Eq. 4.13 that δ_j^l is likely to get small if the neuron is near saturation.
- In total, a weight will learn slowly if either the input neuron is low-activation, or if the output neuron has saturated.

4.6 The backpropagation algorithm

The algorithm lets us calculate the gradient of the cost function.

1. **Input** x: Set the corresponding activation a^1 for the input layer.
2. **Feedforward**: For each $l = 2, 3, ..., L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = \big((w^{l+1})^T\delta^{l+1}\big)\odot \sigma'(z^l)$
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_k^l} = \delta_j^l$

This computes the gradient of the cost function for a single training example, $C = C_x$. Normally, we combine backpropagation with a learning algorithm such as gradient descent. Given a mini-batch of m training examples, the following algrithm applies a gradient descent learning step based on that mini-batch:

1. Input a set of training examples
2. **For each traning sample** x**:** Set the corresponding input activation $a^{x,1}$, and perform the following steps:
   (a) **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$
   (b) **Output error** $\delta^{x,L}$: Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$

(c) **Backpropagate the error**: For each $l = L-1, L-2, \ldots, 2$, compute $\delta^{x,l} = \left((w^{l+1})^T \delta^{x,l+1}\right) \odot \sigma'(z^{x,l})$

3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule

$$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$$

and the biases according to the rule

$$b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

4.7 Backpropagation implementation in Python

```
class Network(object):
...
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini
            batch.
        The "mini_batch" is a list of tuples "(x, y)", and "eta"
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
                delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
                delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple "(nabla_b, nabla_w)" representing the
        gradient for the cost function C_x.  "nabla_b" and
        "nabla_w" are layer-by-layer lists of numpy arrays, similar
        to "self.biases" and "self.weights"."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer
            by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
```

```
37              sigmoid_prime(zs[-1])
38          nabla_b[-1] = delta
39          nabla_w[-1] = np.dot(delta, activations[-2].transpose())
40          # Note that the variable l in the loop below is used a
                little
41          # differently to the notation in Chapter 2 of the book.
                Here,
42          # l = 1 means the last layer of neurons, l = 2 is the
43          # second-last layer, and so on.  It's a renumbering of the
44          # scheme in the book, used here to take advantage of the
                fact
45          # that Python can use negative indices in lists.
46          for l in xrange(2, self.num_layers):
47              z = zs[-l]
48              sp = sigmoid_prime(z)
49              delta = np.dot(self.weights[-l+1].transpose(), delta) *
                    sp
50              nabla_b[-l] = delta
51              nabla_w[-l] = np.dot(delta, activations[-l-1].transpose
                    ())
52          return (nabla_b, nabla_w)
53
54  ...
55
56      def cost_derivative(self, output_activations, y):
57          """Return the vector of partial derivatives \partial C_x /
58          \partial a for the output activations."""
59          return (output_activations-y)
60
61  def sigmoid(z):
62      """The sigmoid function."""
63      return 1.0/(1.0+np.exp(-z))
64
65  def sigmoid_prime(z):
66      """Derivative of the sigmoid function."""
67      return sigmoid(z)*(1-sigmoid(z))
```

4.8 Backpropagation: the big picture

5 Chapter 3: Improving the way neural networks learn

- This chapter is about techniques to improve the implementation of backpropagation.

5.1 The cross-entropy cost function

5.1.1 Problem description

- We want the neurons to learn fast from their errors.

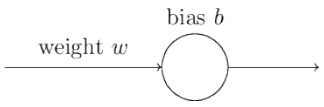


Figure 21: The toy example used

- We want this neuron to do something very easy: take the input 1 to output 0. This is of course much easier to do by choosing the weight and biases correctly, but is done for illustrational purposes.
- The initial weight is chosen to be 0.6 and the initial bias is 0.9. This gives an initial output of 0.82
- An illustration of the learning process is seen here: `http://neuralnetworksanddeeplearning.com/chap3.html`
- This neuron learns very rapidly.
- Choosing the starting weight and bias to be 2.0, which gives an initial output of 0.98, makes the learning process go much slower, even though we are using the same learning rate $\eta = 0.15$. Here for the first ≈ 150 epochs, the weights and biases do not change much at all.
- We see that the neuron learns slowly even though it is very at guessing in the initial phase.
- **Why is learning slow?** We know that the neuron learns by changing the weight and bias according to the partial derivatives of the cost function: $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. If the learning is slow, these partial derivatives must be small.
- The cost function used is

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a||^2 \tag{5.1}$$

  When it is applied to our example neuron, it is expressed

$$C = \frac{(y-a)^2}{2} \tag{5.2}$$

  where a is the neuron's output when the training input $x = 1$ is used, and $y = 0$ is the corresponding desired output.
- Written more explicitly, we know that $a = \sigma(z)$ and $z = wx + b$
- The partial derivatives are then

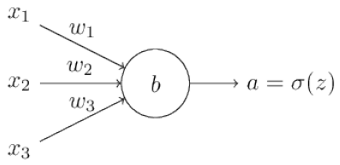
$$\begin{aligned} \frac{\partial C}{\partial w} &= (a-y)\sigma'(z)x = a\sigma'(z) \\ \frac{\partial C}{\partial b} &= (a-y)\sigma'(z) = a\sigma'(z) \end{aligned} \tag{5.3}$$

  Here, $x = 1$ and $y = 0$ is substituted.
- We see that the reason for the slowdown is that when the neuron's output is close to 1, $\sigma'(z)$ (the derivative of $\sigma(z)$) get small.

5.2 Introducing the cross-entropy cost function

- We can solve the problem by replacing the quadratic cost function with a different one.

- Now we want to train a neuron with multiple input variables:



- The output is $a = \sigma(z)$ where $z = \sum_j w_j x_j + b$
- We define the cross-entropy cost function for this neuron by

$$C(w, b) = -\frac{1}{n} \sum_x [y \ln(a) + (1 - y) \ln(1 - a)] \tag{5.4}$$

where n is the total number of items of training data, the sum is over all training input x, and y is the desired output.

- **Why can this be a cost function?**
  1. The expression is non-negative, $C > 0$.
  2. If the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy will be close to zero.
- We can calculate the partial derivatives (where $\sigma(z) = 1/(1+e^{-z}) \quad \Longrightarrow \quad \sigma'(z) = \sigma(z)(1-\sigma(z))$

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= \frac{1}{n} \sum_x x_j (\sigma(z) - y) \\ \frac{\partial C}{\partial b} &= \frac{1}{n} \sum_x (\sigma(z) - y) \end{aligned} \tag{5.5}$$

It tells us that the rate at which the weight learns is controlled by $(\sigma(z) - y)$, i.e. the error in the output. Meaning that the larger the error, the faster the neuron will learn.

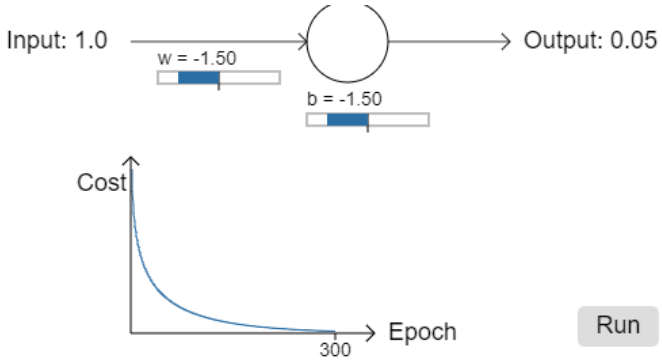


Figure 22: Learning happens very fast, even with badly chosen initial weights and biases.

5.2.1 The cross-entropy cost function for many-neuron multi-layer network

- Suppose that $y = y_1, y_2, \ldots$ are the desired values at the output neurons, i.e. the neurons in the final layer, while $a_1^L, a_2^L, \ldots$ are the actual outout values. The cross-entropy is then

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] \tag{5.6}$$

- The cross-entropy is almost always a better choice to use than the quadratic cost, provided the output neurons are sigmoid neurons.

5.3 Using the cross-entropy to classify the MNIST digits

- A network with
  - 30 hidden neurons
  - Mini-batch size of 10
  - Learning rate $\eta = 0.5$
  - Train for 30 epochs

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.
    CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
... monitor_evaluation_accuracy=True)
```

These settings give us a network with 95.49% accuracy, close to the results using the quadratic cost, with 95.42%.

- Using the same parameters, but instead 100 hidden neurons give a much better accuracy, on 96.82%

5.4 What does the cross-entropy mean?

- We know that the origin for the learning slowdown is due to the $\sigma'(z)$ term in the equations

$$\begin{aligned} \frac{\partial C}{\partial w} &= a\sigma'(z) \\ \frac{\partial C}{\partial b} &= a\sigma'(z) \end{aligned} \tag{5.7}$$

- Optimally, we would like the $\sigma'(z)$ term to disappear, so that the cost $C = C_x$ for a single training example x would satisfy

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= x_j(a - y) \\ \frac{\partial C}{\partial b} &= (a - y) \end{aligned} \tag{5.8}$$

- Using some algebra, and integration, we can get the cross-entropy expression.

5.5 Softmax

- The idea of softmax is to define a new type of output layer for the neural network.
- For softmax, we don't apply the sigmoid function to get the output. Instead, in a softmax layer we apply the *softmax function* to the z_j^L.
- The softmax function says that the activation a_j^L of the jth output neuron is

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{5.9}$$

where z_j^L is the weighted input, and we sum over all the output neurons z_k^L

- An advantage with the softmax function is that the output activations always sum up to 1:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1 \tag{5.10}$$

- As a result, if a_4^L increases, the other output activations must decrease by the same total amount to maintain the sum to be 1.
- This means that the output from the softmax layer can be thought of as a probability distribution.
- This is convenient in the way that we can interpret a_j^L as the network's estimated probability that the correct digit classification is j.

5.6 The log-likelihood cost function

- The log-likelihood associated to this training input is

$$C = -\ln a_y^L \tag{5.11}$$

  This means that we are using the MNIST database, and input an image of 7, the log.likelihood cost is $-\ln a_7^L$.
- This makes sense: If the network estimated a correct value, the output is $\approx 1 \implies -\ln a_7^L$ will be small.
- Opposite: An output close to zero will give a larger log-likelihood cost.
- As a general point of principle, softmax plus log-likelihood is worth using whenever you want to interpret the output activations as probabilities.

5.7 Overfitting and regularization

- Our goal is to find a model that is able to make predictions in situations it hasn't been exposed to before.
- To illustrate our challenge, we will not train the network on all 50 000 MNIST training images, but just use the first 1 000.

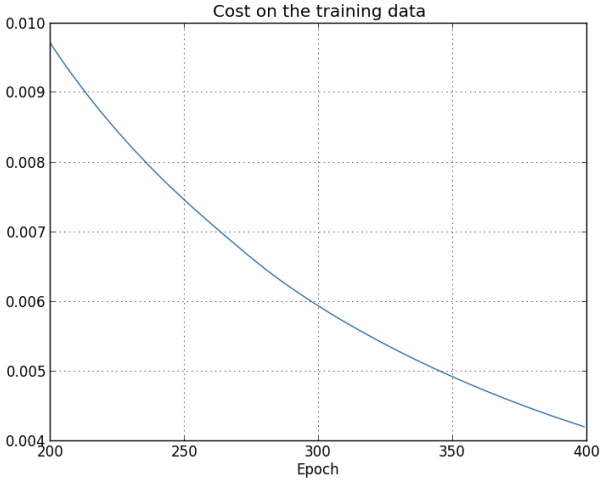


Figure 23: Cost function on training data

- The fact that the cost is decreasing seems encouraging, but seeing the classification accuracy changes that:

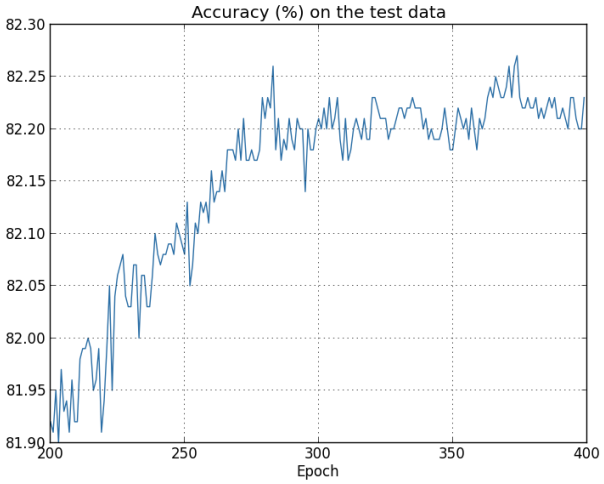


Figure 24: Accuracy of our classification more or less stops imporving after epoch 280.

- This tells us that what the network learns after epoch 280 no longer generalizes to the test data. It is not useful learning. We say that the network is *overfitting* or *overtraining* beyond epoch 280.

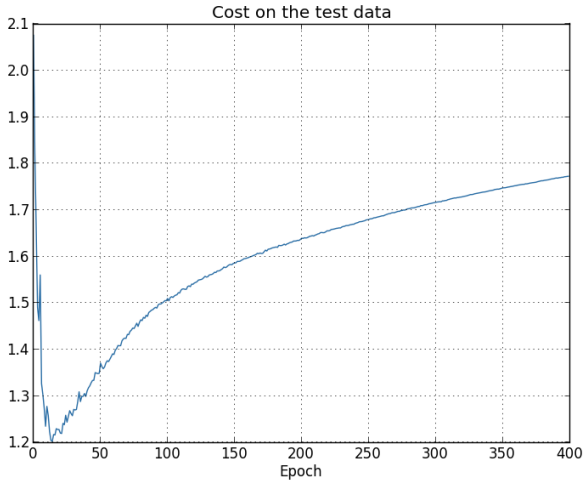


Figure 25: Cost on the test data improves until epoch 15, then gets worse. This is another sign that our model is overfitting.

- A similar sign of overfitting can be seen on the accuracy of the training data:

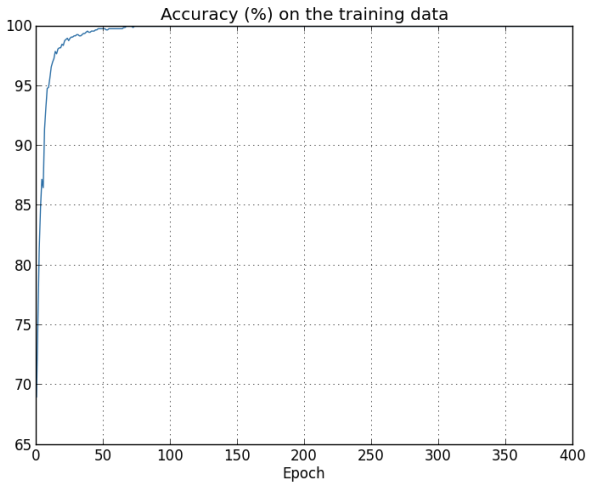


Figure 26: Accuracy on the training data rises all the way up to 100 percent. But the test accuracy tops out at just 82.27 percent. This tells us that our network is learning about peculiarites of the training set, not just recognizing digits in general.

- We want a way to detect when the network is overfitting.
- We split the MNIST data in three: `training_data, validation_data, test_data`
- `validation_data` contains 10 000 iamges of digits, different from the 50 000 in the training set and 10 000 in the test set.
- The strategy is to use `validation_data` instead of `test_data` to prevent overfitting. We will compute the classification accuracy after each epoch, and when the `validation_data` has saturated, we stop training.
- One can think of the validation data as a type of training data that helps us learn good hyper-parameters.
- The best way to reduce overfitting is to increase the size of the training data.

5.8 Regularization

- Regularization is a technique used to reduce overfitting, even when we have a fixed network and fixed training data.
- The most common is known as **weight decay** or **L2 regularization**, where we add an extra regularization term to the cost function.
- The regularized cross-entropy:

$$C = -\frac{1}{n}\sum_{x,j}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2 \tag{5.12}$$

  $\lambda > 0$ is known as the regularization parameter, and n is the size of our training set.
- The same regularization can be done with the quadratic cost and other cost functions. In general we can write

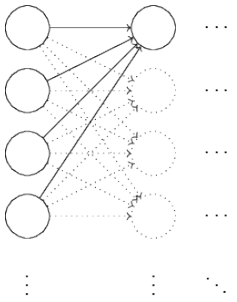
$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2 \tag{5.13}$$

  where C_0 is the original, unregularized cost function.

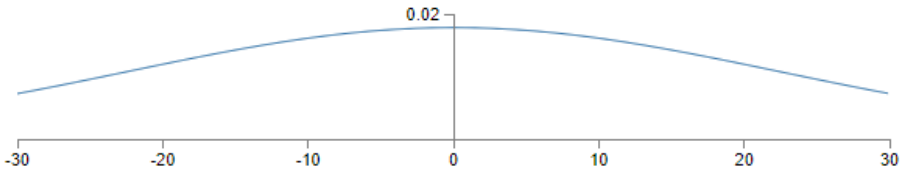
- Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowd if the considerable improve the first part of the cost function. Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function.

5.9 Weight initialization

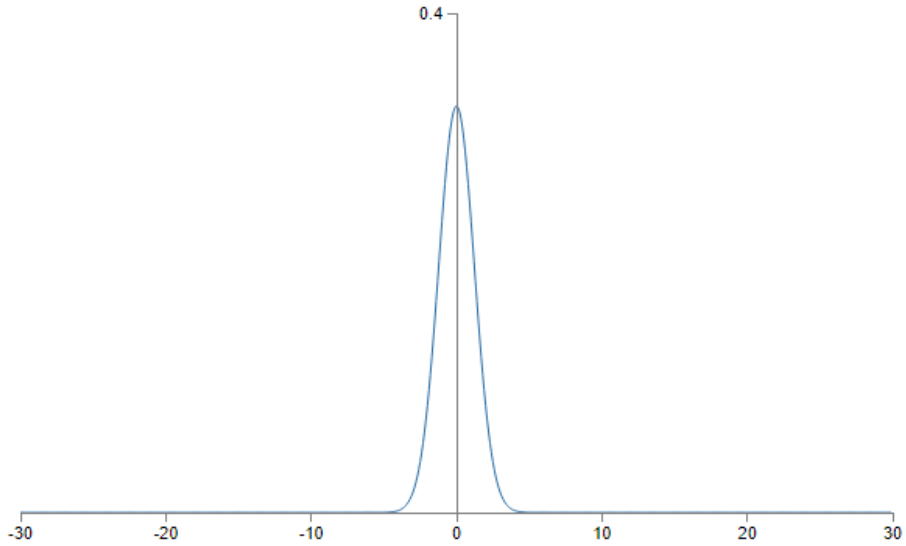
- We want another strategy for initializing our weights than the random initialization first used.
- We will focus on the weights connecting the input neurons to the first neuron in the hiden layer, and ignore the rest of the network:



- For simplicity, we use a training input x in which half the neurons are on, i.e. set to 1, and half the input neurons are off, i.e. set to 0.
- When considering the weighted sum $z = \sum_j w_j x_j + b$ of inputs to our hidden neuron. 500 terms will here vanish, because the corresponding input x_j is zero. Meaning that this sum is over 501 normalized Gaussian random variables, accounting for the 500 weight terms and the 1 extra bias term.
- Meaning that z itself is distributed as a Gaussian with mean zero and standard deviation $\sqrt{501} \approx 22.4$:



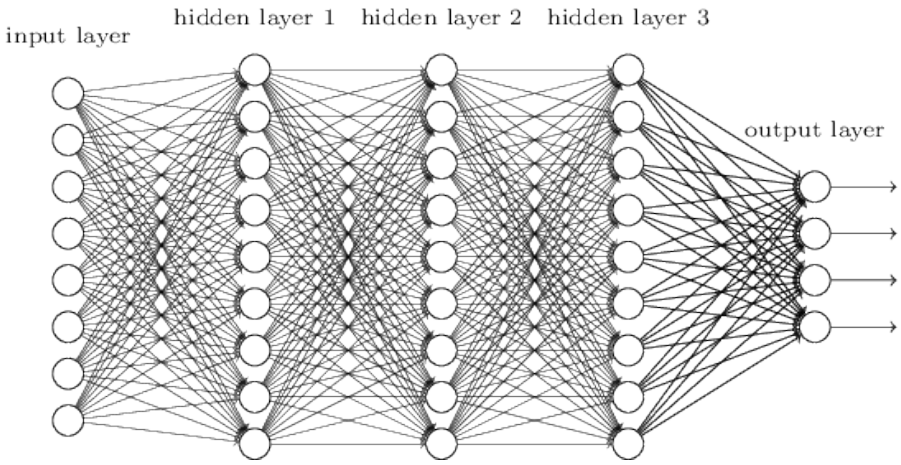
- This means that it is quite likely that $|z|$ will be pretty large, i.e. $z >> 1$ or $z << -1$. Meaning that the output $\sigma(z)$ from the hidden neuron will be very close to 0 or 1. That means the hidden neuron has saturated. As a result, those weights will only learn very slowly when we use the gradient decent algorithm.
- How should we choose the weights better? Suppose we have a neuron with n_{in} input weights. Then we shall initialize those weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{\text{in}}}$ thereby making it less likely that our neuron will saturate.
- Then, the weighted sum z will again be a Gaussian random variable, but with a much sharper peak than before:



6 Chapter 6: Deep learning

6.1 Introducing convolutional networks

- Previously, we have used networks in which adjacent network layers are fully connected to one another. Every neuron in the network is connected to every neuron in the adjacent layers:



- For the 28×28 images used, this means that the network has $784 (= 28 \times 28)$ input neurons.
- For images, it is strange to use networks with fully-connected layers to classify images. Because such an architecture does not take into account the spatial structure of the images. For instance, it treats input pixels which are far apart and close together on exactly the same footing.
- **Convolutional networks** tries to take advantage of the spatial structure. It is an architecture which is particularly well-adapted to classify images.
- Convolutional neural networks use three basic ideas: *local receptive fields, shared weights* and *pooling.*

6.1.1 Local receptive fields

- For the fully-connected layers, the inputs were depicted as a vertical line of neurons. In a convolutional net, we will instead think of the inputs as a 28×28 square of neurons, whose

values correspond to the 28×28 pixel intensities we are using as inputs:

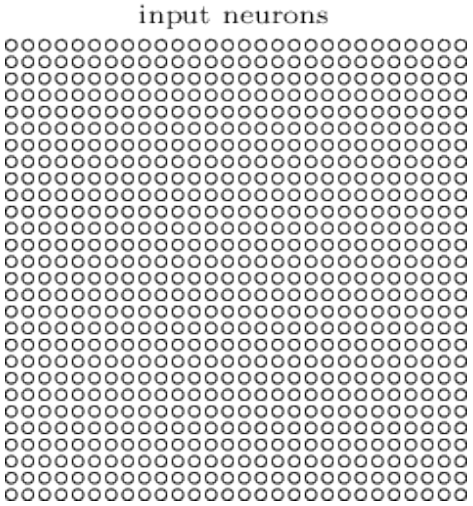


Figure 27: Input neurons of a convolutional net

- We will connect the input pixels to a layer of hidden neurons, but we will only make connections in small, localized regions of the input image. Each neuron in the first hidden layer will be connected to a small *region* of the input neurons:

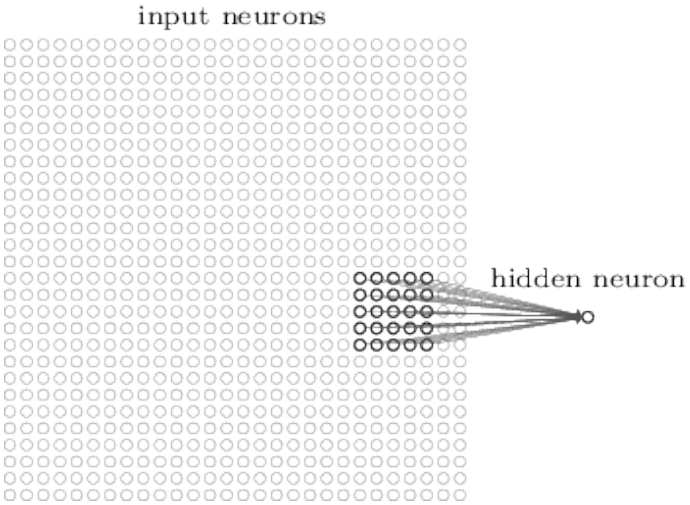
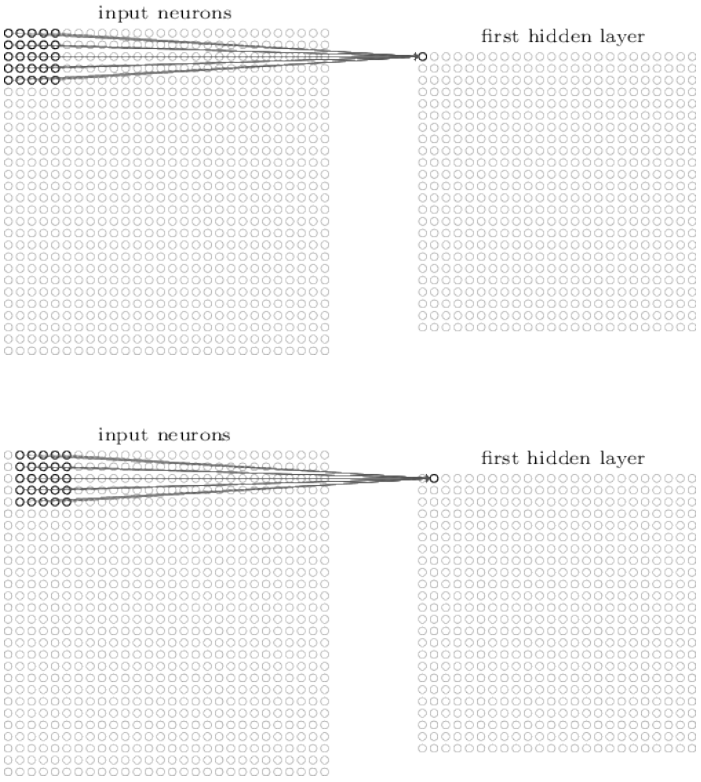


Figure 28: One neuron in the first hidden layer is connected to a 5×5 region in the input regions.

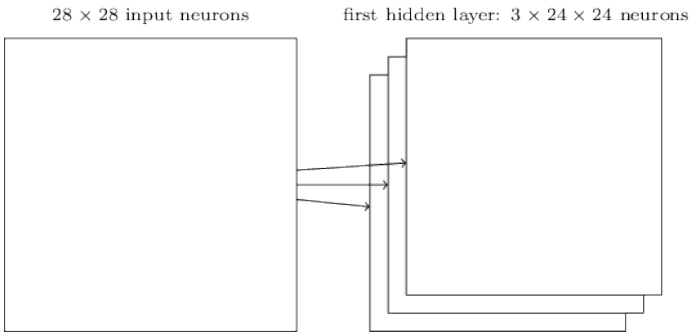
- The 5×5 region in the input image is called the **local receptive field** for the hidden neuron. Each connection learns a weight. And the hidden neuron learns a bias as well.
- The local receptive field is slided over by one pixel to the right to connect to a second hidden neuron:



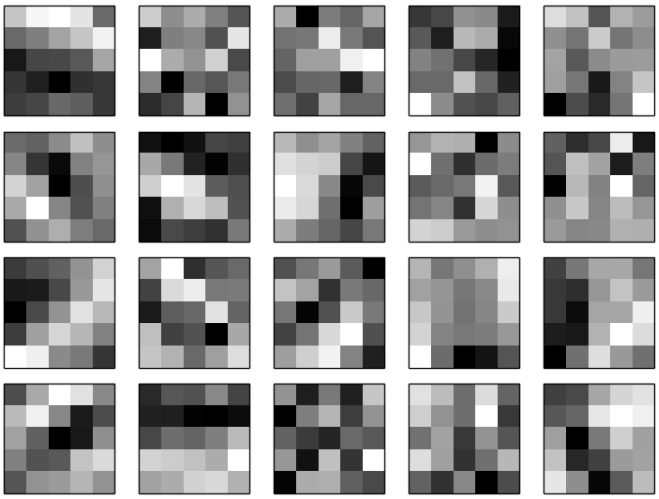
- This means that if we have a 28×28 input image and 5×5 local receptive fields, there will only be 24×24 neurons in the hidden layer.
- It is also possible to experiment with other stride lengths than one pixel, which is used here.
- Each hidden neuron has a bias and 5×5 weights connected to its local receptive field. We are going to use the *same* weights and bias for each of the 24×24 hidden neurons. Meaning that for the j, kth hidden neuron, the output is

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right) \tag{6.1}$$

  σ is the neural activation function. b is the shared value for the bias. $w_{l,m}$ is a 5×5 array of shared weights. And $a_{x,y}$ denotes the input activation at position x, y.
- This means that all the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image.
- Therefore, the map from the input layer to the hidden layer is sometimes called a *feature map*. The weights defining the feature map are called the *shared weights*. The bias defining the feature map is the *shared bias*.
- To perform well in image recognition, we need more than one feature map. Therefore, a complete convolutional layer consists of several different feature maps:



- Here, there are 3 feature maps. Each is defined by a set of 5×5 shared weights, and a single shared bias. Meaning that the network can detect three different kinds of features, with each feature being detectable across the entire image.
- It is also normal to use more feature maps.
- Here are some of the features which are learned:

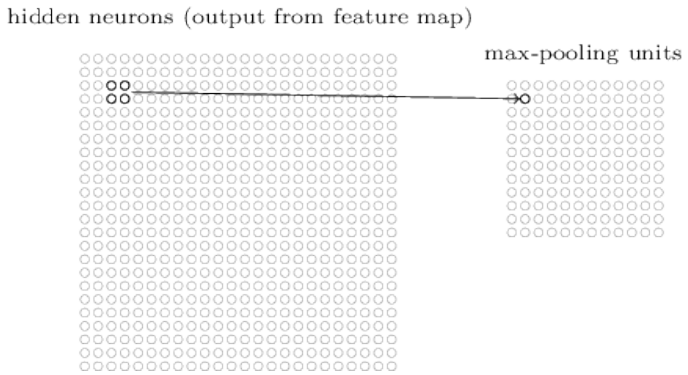


- The 20 images correspond to 20 different feature maps. Each map is represented as a 5×5 block image, corresponding to the 5×5 weights in the local receptive field.
- Whiter blocks means smaller weight. So the feature map responds less to corresponding input pixels. And vice versa.
- The convolutional network will have much less parameters than a fully-connected network: Each feature map needs $25 = 5 \times 5$ shared weights, plus a single shared bias. Meaning that each feature map requires 26 parameters. Having 20 feature maps gives us a total of $20 \times 26 = 520$ parameters defining the convolutional layer.
- For a fully-connected network with $28 \times 28 = 784$ input neurons, and (modest) 30 hidden neurons gives a total of 784×30 weights plus 30 biases, a total of 23550 parameters. 40 times as many as the convolutional layer.

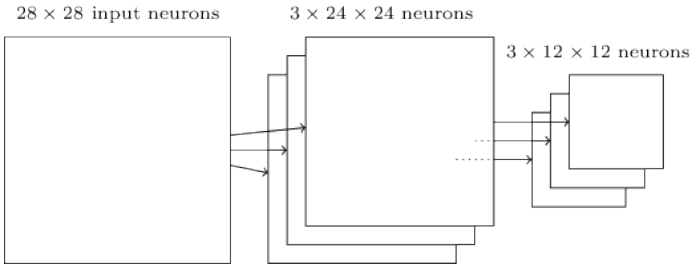
6.1.2 Pooling layers

- Pooling layers are usually used immediately after convolutional layers. The pooling layer simplifies the information in the output from the convolutional layer.

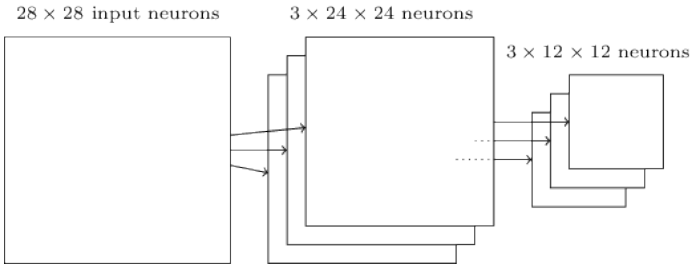
- A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map. For instance, each unit in the pooling layer may summarize a region of (say) 2×2 neurons in the previous layer.
- A common procedure is *max-pooling*, which a pooling unit simply outputs the maximum activation in the 2×2 input region:



- In this example, since we have 24×24 neurons output from the convolutional layer, after pooling we have 12×12 neurons.
- Pooling is applied to each feature map separately, so if we have three feature maps, the combined convolutional and max-pooling layers would look like:



- *L2 pooling* is also a technique used for pooling. Here, we take the square root of the sum of the squares of the activations in the 2×2 region.
- Putting it all togethe, we form a complete convolutional neural network. Here, one layer is added, which is the output layer of 10 output neurons, corresponding to 10 possible values for the MNIST digits.



- The final layer of connections in the network is a fully-connected layer. That is, this layer connects *every* neuron from the max-pooled layer to every one of the 10 output neurons.
- The network will be trained using stochastic gradient descent and backpropagation.

7 E1 - notes

7.1 T1 & T2: Forward and backwards pass (learning)

7.1.1 Perceptron algorithm

- The Perceptron algorithm is given by

$$z = \begin{cases} 0 & \text{if } \mathbf{wx} + \mathbf{b} \leq 0 \\ 1 & \text{if } \mathbf{wx} + \mathbf{b} > 0 \end{cases} \tag{7.1}$$

7.1.2 Learning rate

- When learning by gradient descent 7.1.4, weights are moved in the direction of the cost function. The size of each step is governed by the learning rate, η:

$$\Delta v = -\eta \nabla C \tag{7.2}$$

7.1.3 Learning

- **Mini-batch:** Number of examples chosen for cost estimate
- **Epoch**: Number of times to iterate through all of the training examples.
- A network using stochastic gradient descent with the following parameters:
  - epochs = 10
  - mini-batch: 20

  You can say the following of the network: "Choose 20 examples to calculate an estimate of the cost. Backpropagate the error. Repeat until all training examples are exhausted. Repeat all of this 10 times."

7.1.4 Gradient descent

- Gradient descent updates the weights after each epoch, that is, after every pass over the training set, calculating the entire gradient.
- Stochastic gradient descent updates the weights after each training sample.
- **Stochastic gradient descent (SGD)**: Only a few of the training data are used to calculate the cost function in each iteration.
  - Faster than gradient descent, as there are fewer elements to compute.
  - Less accurate. The number of iterations needed to reach a minimum will increase.
  - More likely to find a global minimum than standard gradient descent.
- In normal gradient descent, the full training data set is used.
- **Vanishing gradient problem**: When the gradient becomes close to zero, the weights in the network are prevented from changing its value. May stop the network from further training.
- Gradient descent is an optimization algorithm where you learn by moving the parameters in the opposite direction of the gradient of the cost function. Doing this decreases the loss values.

7.1.5 Activation function (general)

- An activation function is used in each neuron after the summation of the inputs and the bias to determine the input to the next neuron. A non-linear activation function makes the network able to learn arbitrary complex functions. A network with a linear activation function can only learn linear functions.

7.1.6 Sigmoid function

$$a = \sigma(z) = \frac{1}{1 + e^{-x}} \tag{7.3}$$

- The **Sigmoid function** is well suited for classifying values to different classes. But the gradient is close to 0 for both small and large values of x. This is the vanishing gradient problem.

7.1.7 Logistic sigmoid function

- The **logistic sigmoid** has a problem with vanishing gradients.

7.1.8 tanh activation function

$$a = \sigma(z) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{7.4}$$

- Has problems with vanishing gradients.
- Is zero-centered.

7.1.9 ReLU activation function

$$a = \sigma(z) = \max(0, z) \tag{7.5}$$

- Has an undefined point in the derivative
- Is notably efficient to compute.

7.1.10 Softmax activation function

$$a = \sigma(z) = \frac{e^{z_i}}{\sum_{i=1}^{m} e^{z_i}} \tag{7.6}$$

- Depends on all perceptron outputs of a layer. Other activation functions depend only on the output of a single perceptron.
- Its output is summed to 1.

7.1.11 Identity activation function

$$a = \sigma(z) = z \tag{7.7}$$

-

7.1.12 Backpropagation

- The goal of backpropagation is to compute the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ with respect to any weight w or bias b in the network.
- Lets us find the gradient of a network's cost function, with respect to every individual weight and bias in the network. Thus, we know the contribution of each weight to the error of the output and can therefore train the network to minimize the error.
- The two main assumptions of the backpropagation algorithm is

  1. The cost function C can be written as an average, $C = \frac{1}{n}\sum_x C_x$ over the cost functions C_x for individual training examples x. For the quadratic cost function, $C_x = \frac{1}{2}||y - a^L||^2$
  2. The cost function must be possible to write as a function of the outputs from the neural network. This is possible for the quadratic cost function:

$$C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

- The error in terms of the error in the next layer is represented by

$$\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l)\right) \tag{7.8}$$

- The error in the output layer is given by

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \tag{7.9}$$

  $\frac{\partial C}{\partial a_j^L}$ tells us how much the cost function is changing with respect to the activation of the j^{th} neuron in the output layer. $\sigma'(z_j^L)$ tells us how much the activation function changes with respect to the weighted input z. This term approaches zero as the neuron saturates.

- Backpropagation can be summarized in 5 steps:

  1. **Input**: Start by defining a_0 as the input.
  2. **Calculate**: Calculate z and a for all neurons in each layer.
  3. **Output error**: Set $y = a^L$ and calculate its output error by the formula $d_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$
  4. **Backpropagate the error:** $\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l)\right)$ and calculate the partial derivatives as $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$

7.1.13 Supervised learning

- Both input and output variables are known beforehand, and you can use an algorithm to learn the mapping from input to output. With new input data, the goal is to approximate the mapping function so well that you can predict the output variables. Supervised learning can be divided into two groups: **classification** and **regression**.
- All data is labeled.
- Examples of supervised learning algorithms:
  - linear regression
  - support vector machines for classification problems.

7.1.14 Unsupervised learning

- Only input variables are known, but no outputs. Therefore you have no supervision of the learning. Unsupervised learning can be divided into two groups: **clustering** and **association**.
- All data is unlabeled.
- Examples of unsupervised learning algorithms:
  - K-means clustering, Apriori algorithm for association rule learning problem.

7.2 T5: Regularization

- Regularization solves the problem of overfitting (at least partially). Overfitting is the result of training to the point where the model is fit to peculiarities in the training set instead of the general patterns. Overfitting leads to poor results on data outside the training set, such as validation sets. Overfitting typically occurs with overtraining, limited data, and/or excessive model capacity.
- Regularization is done by adding terms according to some lp-norm to the cost function, for instance the sum of the squares of the weights, multiplied by some regularization constant. This means that the weights that have a limited influence on the performance of the net on the training set will be driven close to zero, which tend to increase the ability to generalize.

8 E2 - notes

8.1 T4: CNNs and image classification

8.1.1 Data augmentation

- The practice of using small, random adjustments to your training data before letting the network train on it. Normal adjustments: flipping, rotating, stretching, zooming, translating.
- Helps combat overfitting, as it never lets the network train on the same data twice.

8.1.2 Max pooling

- Takes the highest values from a kernel, as we are most interested in what features have a strong presence in an area rather than having the average intensity of multiple features.
- Enhances the presence of certain features.
- **Pooling layer**: Used as a way to reduce the number of subsequent layers, usually between multiple convolutional layers. Pooling layers make us look at increasingly large windows when we move down the network. In this way, we can recognize increasingly large spatial structures by combining smaller structures in the previous layers.

8.1.3 Dropout operation

- Outputs the same tensor/matrix as the input, but with a percentage of the elements set to 0.

8.1.4 Network architecture

```
model = Sequential()  # Initalize a new model
model.add(Conv2D(64,(3,3),activation='relu',input_shape=(img_width,img_height,1)))
model.add(Conv2D(32,(3,3),activation='relu'))
model.add(Conv2D(32,(3,3),activation='relu'))
model.add(Flatten())
model.add(Dense(64,activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
```

- What improvements could be made on this network architecture?
  - Have a MaxPooling2D layer between each convolutional layer. This will extract elements that contain the most information $\longrightarrow$ Better recognition. Decreases number of parameters in the network $\longrightarrow$ Better run time.
  - Add dropouts after each layer. The dropout rate should be increased as one goes to the bottom of the network. Because bottom layers often get changed more than the top ones, because of the "vanishing gradient". Therefore deactivating the bottom neurons will prevent overfitting.
- If you are using a **pretrained network** for real-time facial detection, what properties should be changed?
  - Freeze the top layers. These will already be good at finding general patterns. The bottom layers could be specialized on the specific problem at hand.
  - Add a classifier as the bottom/output layer. In order to specialize the network on the faces you want to recognize.
  - Lower the **learning rate**. In order to fine-tune the network for better accuracy, and avoiding overfitting.

- Convolutional Neural Networks (CNN) are normally preferred over fully connected layers for image classification. One reason for this is that fully connected layers do not consider the spatial arrangement of pixels. In addition, an image contains a lot of information and many pixels, leading to expensive computations in a fully connected network.
- Giving the weights the same initial value is a bad idea. By doing so, each neuron will compute the same gradient, and thus update the weights to exactly similar values.

8.1.5 Layers and their properties

- In a CNN, you use two types of spatial structures/layers for feature extraction: **convolutional layer** and **pooling layer**. The **convolutional layer** takes an input which it thereafter performs a specified amount of filters on by traversing a kernel over the input. This produces a **feature map** for each filter which can be fed into a pooling layer.
  The **pooling layer** decreases the computational requirements by reducing the dimensinality of the feature maps while preserving the important information. If the pooling layer was the last, its output could be sent to a fully connected layer for classification.
- The number of **filters** in a convolutional layer, can be seen as the number of patterns the network is able to recognize.
- The **window size** determines the subarea of an image in which the layer looks for patterns. Having a window size of 5×5 means that the layer scans the image looking for patterns within boxes of 5×5 pixels. This means that the larger the window size, the larger pattern the network is able to recognize.
- **Depth** in a convolutional layer describes the number of filters each layer has.
- The **receptive field size** represent how many neurons that are connected to a neuron in the next layer.
- **Feature extraction** is the process where you use the representations learned by the pretrained convolutional base of the network (pooling and convolutional layers), run new data through it, and thereafter train a new classifier on top of the output. Representations from the convolutional base are likely to be more generic and therefore more reusable.

8.1.6 Calculations on layers/networks

A network takes in images of size $(227 \times 227 \times 3)$ (RGB images). The network' first convolutional layer has 96 filters with size (11×11), and a stride of 4.

- The **output size** will be given by the formula

$$O = \frac{W - K + 2P}{S} + 1 \tag{8.1}$$

  where O is the output height/length, W is the input height/length, K is the filter size, P is the padding and S is the stride. The output size for our example will then be

$$O = \frac{227 - 11 + 2 \cdot 0}{4} + 1 = 55$$

  With 96 filters, the output volume size will be $(55 \times 55 \times 96)$
- The **number of parameters** in this layer will be $(11 \times 11 \times 3) \times 96 = 34848$.
- The number of parameters in a pooling layer will be zero, as it only changes the dimensions of the layer.

Given (28×28) size images. Calculate the number of weights needed for a:

- **Fully connected network**
  - 784 input neurons
  - Layer 1: 20 neurons
  - Layer 2: 10 neurons
  - Output layer: 5 neurons

  Each neuron will be connected to all neurons in the previous layer with a weight. Number of weights W is then: $W = 20 \cdot 784 + 10 \cdot 20 + 5 \cdot 10 = 15930$ and the number of biases, B, is $B = 20 + 10 + 5 = 35$. The total number of parameters in the model is then $N = W + B = 15930 + 35 = 15965$.

- **Convolutional network**
  - (5×5) local receptive field in the kernel.
  - 20 feature maps in first layer
  - (2×2) max pooling layer
  - 10 feature maps in the second layer

  The number of parameters will be:
  Kernel $\longrightarrow$ 1st layer: $(5 \cdot 5 + 1) \cdot 20 = 520$
  Max pooling $\longrightarrow$ 2nd layer: $(5 \cdot 5 + 1) \cdot 10 = 260$
  Total: $520 + 260 = 780$
  An important thing to note is that the number of parameters in the convolutional network will not change if the image size is increased.

- There are different ways to calculate distances in a convolutional layer:
  The L_1 **distance** takes the sum of the absolute distance between each element of the vectors:

$$L_1(I_1, I_2) = \sum |I_1^p - I_2^p| \tag{8.2}$$

  The L_2 **distance** computes the euclidean distance between two vectors:

$$L_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2} \tag{8.3}$$

  The L_2 distance is much more unforgiving. It will punish a single difference, than many small differences.

8.2 T5/T6: Object detection / segmentation

8.2.1 R-CNN: Region-Based CNN

- R-CNN is used for object detection and localization. An image is given to the network, which then proposes "boxes" of different sizes in the image, known as regions of interest. Each box/region is then passed into a CNN to check if it corresponds to any object, thereafter classifying the objects that may have been found. The boundary boxes are then run through a linear regression model to improve the boundary boxes around the objects.

- The main problem of R-CNN is that it is very computaionally expensive, as every region found by selective search has to be processed by a CNN.

- The main bottleneck was the selective search for region proposals.

- R-CNN is slow because a forward pass is needed for every proposal for every image and it has to train the CNN, the prediction and the regression as three different models at the same time.

- The three main steps of R-CNN are:
  1. Generate a set of region proposals from an image. R-CNN uses **selective search** to create these.
  2. Process the region proposals using a CNN. (Feature extraction)
  3. Compute task-specific outputs. (Classification). R-CNN classifies the region proposals using Support Vector Machine (SVM) and linear regression.

8.2.2 Fast R-CNN

- The placement of objects within an image is returned as a **bounding box**. Defined by the position of a corner, the width and the height.
- Region of Interest Pooling (RoIPool) was introduced with Fast R-CNN. **RoI max pooling** is a processing step that takes a RoI of any size and outputs a fixed-length feature vector which is used as an input for a fully-connected network for classification. This is necessary because a Fully-Connected Network (FCN) has a fixed input size.
  The three steps of Fast R-CNN
  1. Region proposal is created using selective search
  2. Region proposals are processed with a CNN
  3. Classification of region proposals are done with CNN

8.2.3 Faster R-CNN

- Faster R-CNN is faster than Fast R-CNN and R-CNN because it shares convolutional layers with the detection network, making the region proposals almost cost free in regards to time.
- Faster R-CNN uses Region Proposal Networks (RPN) when defining regions containing objects in the image, rather than "traditional" algorithms such as selective search.

  The three steps of Faster R-CNN
  1. Creates region proposals using a CNN
  2. Processes region proposals using a CNN
  3. Classifies the region proposals using a CNN

8.2.4 Mask R-CNN

- Returns a detailed mask of each object, defining which pixels in the image belongs to the object.
- Mask R-CNN can do
  - Object detection
  - Object classification
  - Instance segmentation

8.2.5 YOLO

- The main difference between YOLO (You Only Look Once) is that YOLO only looks at the image one time to detect objects. It divides the image into grid cells and performs predictions of class label confidence and bbox regressor for each cell, and combines them afterwards to detect objects.

- YOLO will struggle when an image contains many small objects. This is because it divides the image into grid cells, and it can only predict two boxes and one class per grid cell. Meaning that if there are multiple objects within the grid cell, it will fail to classify them all.
- YOLO uses fully connected layers with input only from the last convolutional layer for its bounding box regression and object category prediction.

8.2.6 SSD

- Uses small convolutional filters to predict object categories and offsets in bounding box locations. Applies these filters to multiple feature maps from the later stages of a network in order to perform detection at multiple scales.
- SSD has achieved significantly better results in real-time object detection than YOLO.

8.2.7 Object detection

- The difference between **classification** and **object detection** in images is that classification only tries to figure out if images contain the classes or not. Object detection has to detect and find all the classes in the image, and localize them using for example a bounding box. Meaning that object detection **localizes** and **classifies** multiple objects in an image.
- **mAP** (mean Average Precision) is a measure of how good your network is at classifying the objects/classes in an image. It gives you the percentage of all input images where the correct object was proposed in the correct location.
- **Calculating mAP**. We have $A_1 = 189$ and $A_2 = 31$ proposed pixels for classes 1 and 2 from the network on a 2-class image. And $B_1 = 201$, $B_2 = 50$ of true object pixels for classes 1 and 2, where 185 of the pixels of A_1 and B_1 intersect, and 23 pixels of A_2 and B_2 intersect.
  In order to calculate mAP for the image, we must first find the **Intersection over Union (IoU)** for the different classes. The IoU is defined as

$$IoU = \frac{\text{Area of Overlap}}{\text{Area of Union}} \tag{8.4}$$

  which here becomes

$$IoU_1 = \frac{185}{(189 + 201) - 185} = 0.902$$
$$IoU_2 = \frac{23}{(31 + 50) - 23} = 0.397$$

  The mAP is then just the average of the IoU of all classes:

$$mAP = \frac{1}{2} \cdot (0.902 + 0.397) = 0.65$$

- The advantage all methods of object detection have in common is that they can use a pre-trained CNN for classification. Like VGG or ImageNet, etc.

8.2.8 Segmentation

- **Segmentation** can be seen as per-pixel classification. And is an addition to object detection.
- There exists two types of segmentation:

  1. Semantic segmentation
     Here, all each pixel is classified to a class. Meaning that all persons would be in the same class if you were to process an image of persons. The Fully Convolutional Network (FCN) is a semantic segmentation network. The output from FCN is very coarse, and can be extended with random fields to get better performance.

2. Instance segmentation
   Instance segmentation will distinguish between different individuals.

- A segmantation approach can easily be extended to Human Pose Estimation, where human joints are estimated in the segmented region. This is done by predicting one mask for each keypoint of the joints.
- RoI Pooling vs segmentation for object detection: For a mask, the drop in accuracy is higher for small translations than for a bounding box. Making masks more sensitive to small misalignments than a bounding box.

8.2.9 Keras

- `fit`: Train the network for iterations on a dataset
- `evaluate`: Check that the model performs well on an unseen test set
- `predict`: Generates output predictions for input samples
- `compile`: Configures a model for training.
- **Dense** layer consists of global patterns (from all pixels). **Convolutional layer** consists of local patterns (small 2D windows)