Natural Language and Speech Processing

Lecture 7: Neural networks

Tanel Alumäe

Contents

- Introduction to neural network architectures
- Loss functions
- How neural networks are trained

- Largely based on http://neuralnetworksanddeeplearning.com

Human visual system

- Consider the following sequence of handwritten digits

- Most people effortlessly recognize them as 504192
- However, it’s difficult to write a computer program that can do the same thing
- Simple intuitions (e.g., *9* has a loop at the top, and a vertical stroke at the bottom right) are not so simple to express algorithmically
- Even when trying to write rules, we would quickly get lost with all the exceptions, caveats and special cases

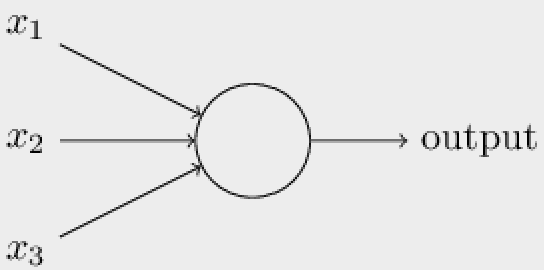
Neural networks

- Neural networks approach the problem differently
- Neural networks use the training data to automatically infer rules for recognizing handwritten digits
- By increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy

Perceptrons

- Perceptrons were developed in the 1950s and 1960s
- Perceptrons take several **binary** inputs and produce a **single binary** output
- Each input is multiplied with a weight
- Output is calculated as:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{theshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{theshold} \end{cases}$$

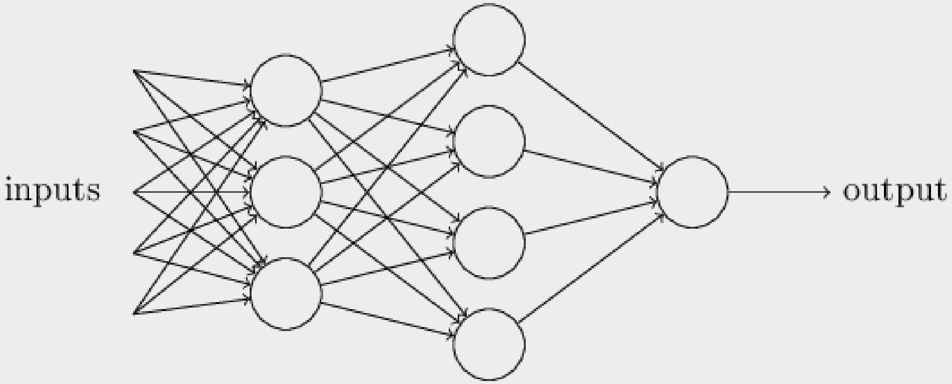


Perceptron: simple example

- A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence
- Example: cheese festival is coming up nearby. You might make your decision by weighing up three factors (x_1, x_2, x_3):
  - Is the weather good?
  - Does your boyfriend or girlfriend want to accompany you?
  - Is the festival near public transit? (You don't own a car)
- You assign weights to those factors:
  - $w_1=6$ $w_2=2$ $w_3=2$
- Finally, you choose threshold 5 for the perceptron
- Note that with these choices, the output of the perceptron only depends on the weather

Connecting perceptrons

- Obviously, perceptron is not a very flexible model
- But we can have many **layers** of perceptrons
- What about the perceptrons in the second layer?
  - Each of those perceptrons is making a decision by weighing up the results from the first layer
- Even more complex decisions can be made by the perceptron in the third layer
- Note that each perceptron still has only one output



Simplifying notation

- Let's simplify the way we describe perceptrons
- Let’s represent weight-multiplied inputs as a dot product:

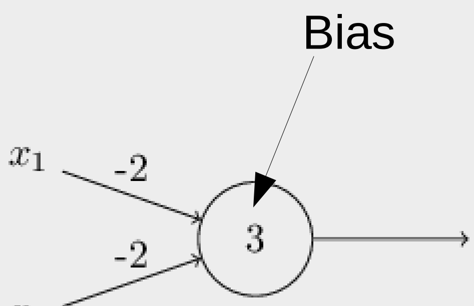
$$\sum_j w_j x_j \equiv w \cdot x$$

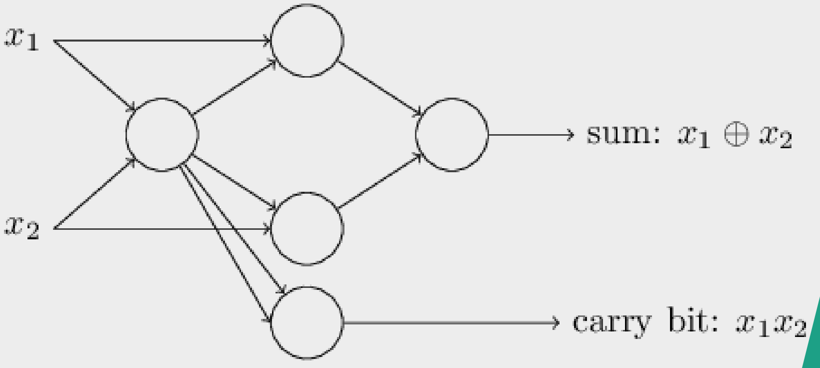
- Let’s also move the threshold to the other side of the inequality, and replace it by what's known as the **perceptron's bias**
- Using the bias instead of the threshold, the perceptron rule can be rewritten as:

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

Perceptron for logic gates

- Perceptrons can be used is to compute the elementary logical functions AND, OR and NAND
- For example, the perceptron on the right implements a NAND gate (NAND=NOT AND)
- NAND(0, 0) = 1
  NAND(0, 1) = 1
  NAND(1, 0) = 1
  NAND(1, 1) = 0
- **Because NAND gates are universal for computation, it follows that perceptrons are also universal for computation**
- For example, the network of NAND gates on the right implements bitwise addition



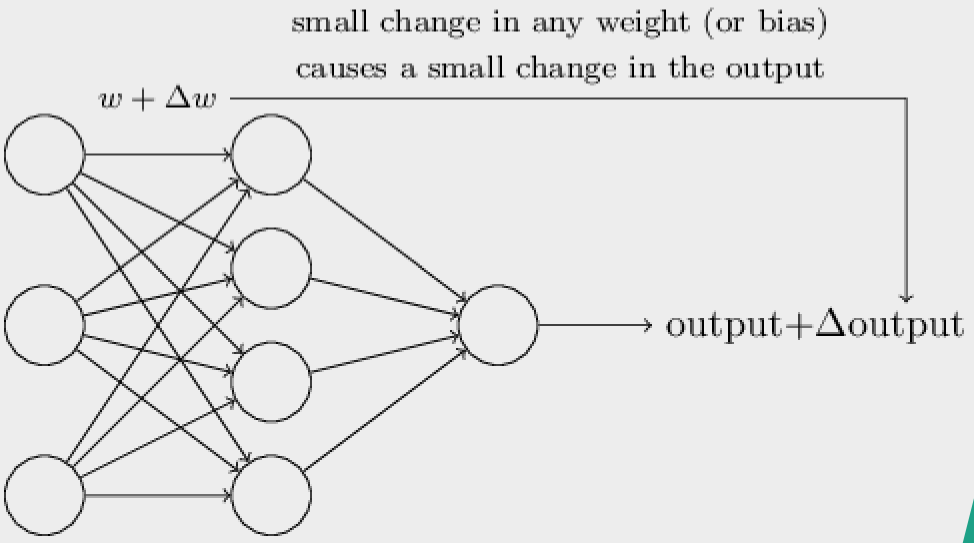


Perceptrons as universal functions

- However, perceptrons are much more than just NAND gates
- Learning algorithms which can automatically tune the weights and biases of a network of perceptrons
- Learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates
- Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems

Problem with perceptrons

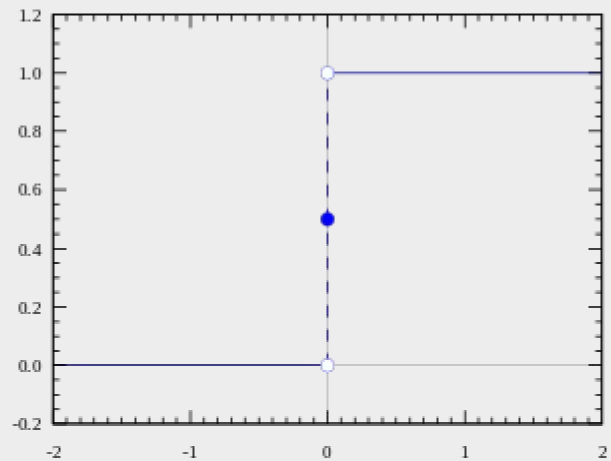
- To see how learning might work, we make a small change in some weight in the network
- We want this small change in weight to cause only a small change in the output
  - This actually makes learning possible
- However, with perceptrons, small changes in weights can cause big changes in output (output changes 0 → 1)
- That flip may then cause the behaviour the network to change in some very complicated way
- To overcome this, we use **sigmoid** neurons

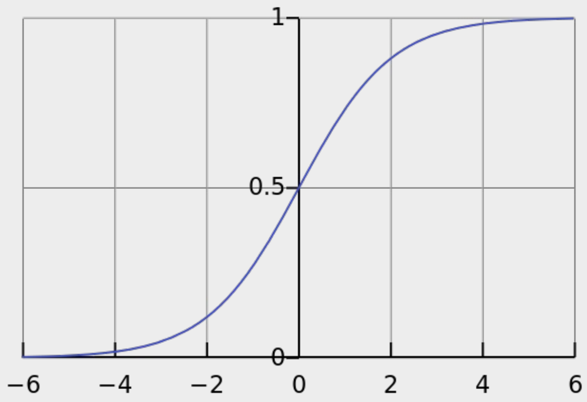


Sigmoid

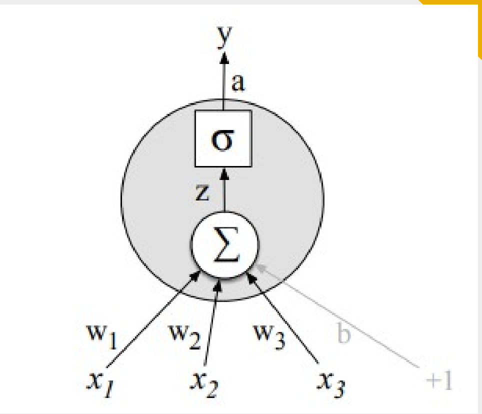
- Sigmoid function is similar to step function used in the perceptron
- But modified to be smooth around *0*
- Thus, small changes in the input cause only a small change in their output
- Inputs to and outputs of sigmoid neurons can be between 0 and 1
  - As opposed to perceptrons, where inputs and outputs are binary
- Definition:

$$\text{output} = \frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}$$





Activation function



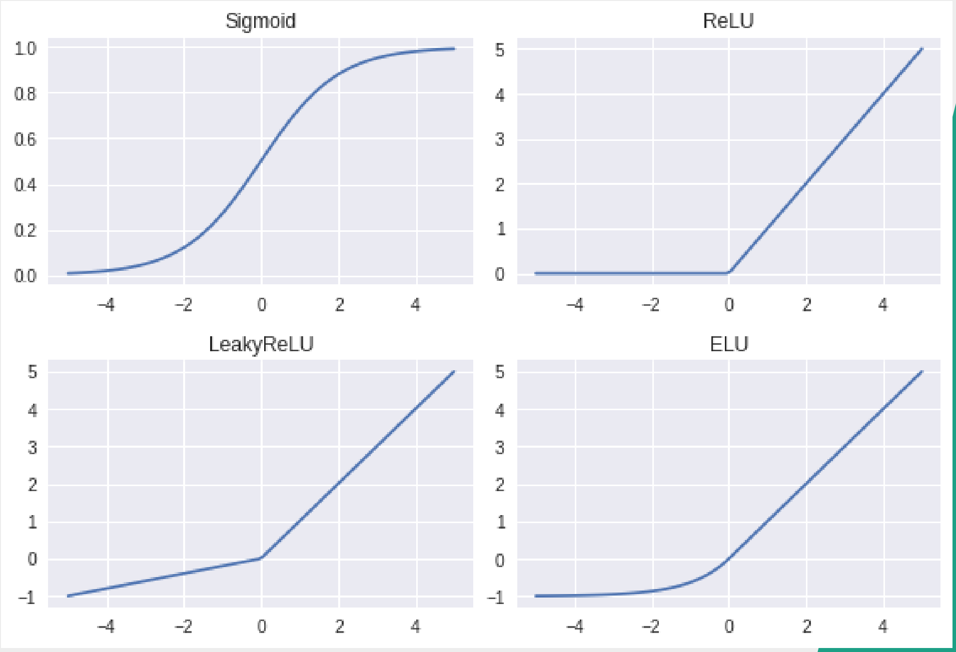
- The function that we use to transform input to output is called activation function
- Usually, the activation function is applied to the sum of weighted inputs z
- Sigmoid:

$$z = \sum_j w_j x_j + b$$

$$\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$$

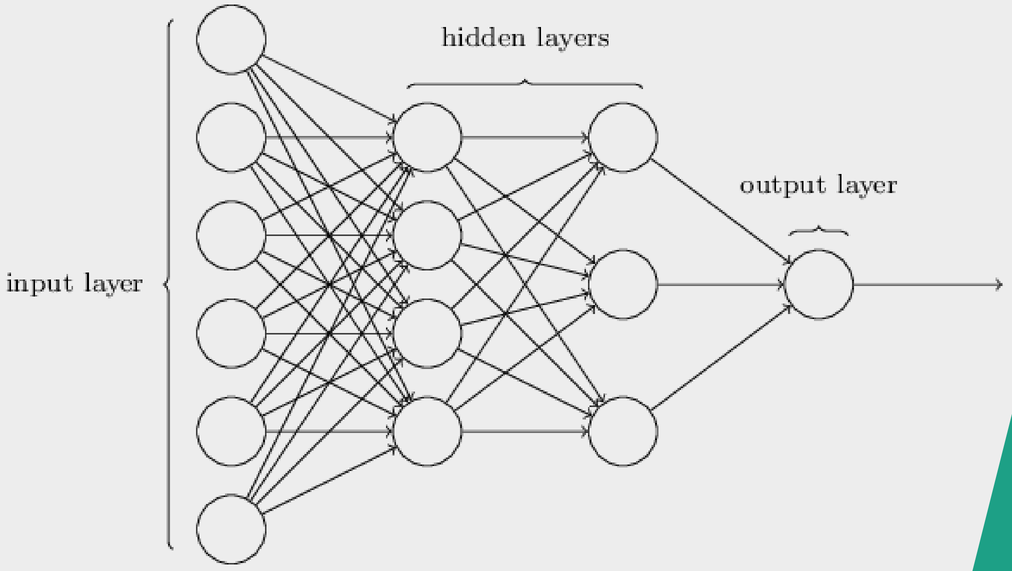
- Other activation functions:

$$\begin{aligned}
\text{ReLU}(z) &= \max(0, z) \\
\text{LeakyReLU}(z) &= \begin{cases} z & \text{for } z \geq 0 \\ \alpha z & \text{for } z < 0 \end{cases} \\
\text{ELU}(z) &= \begin{cases} z & \text{for } z \geq 0 \\ \alpha(e^z - 1) & \text{for } z < 0 \end{cases}
\end{aligned}$$



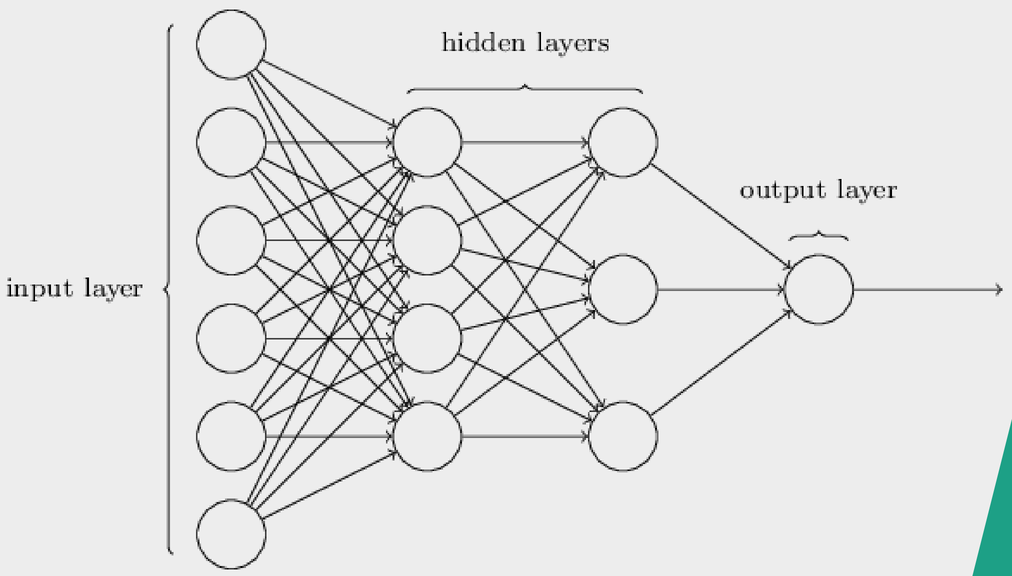
Architecture of neural networks

- Neural network has:
  - Input layer
  - Output layer
  - Hidden layer(s)
- Input neurons corresponds to input features, and they are simply scalar values
- Output layer corresponds to the output of the function that the network has to learn
  - Can be more than one neuron
  - Later we will see that the neural network can even have more than one output layers
- In simple neural networks, layers are connected sequentially, and **fully connected** with each other, but it doesn't have to be so
- Later we will learn that there can even be **feedback loops**, resulting in **recurrent neural networks**



Architecture of neural networks

- Often, it's straightforward to design input and output layers
  - They depend on the task
- Design of hidden layers is more complicated
  - How many hidden layers?
  - How many neurons in each layer?
  - Which activation function to use?
  - How to regularize the hidden layers?
- Neural network with more than one hidden layer is called a **deep neural network**
- **Deep learning**: training deep neural networks



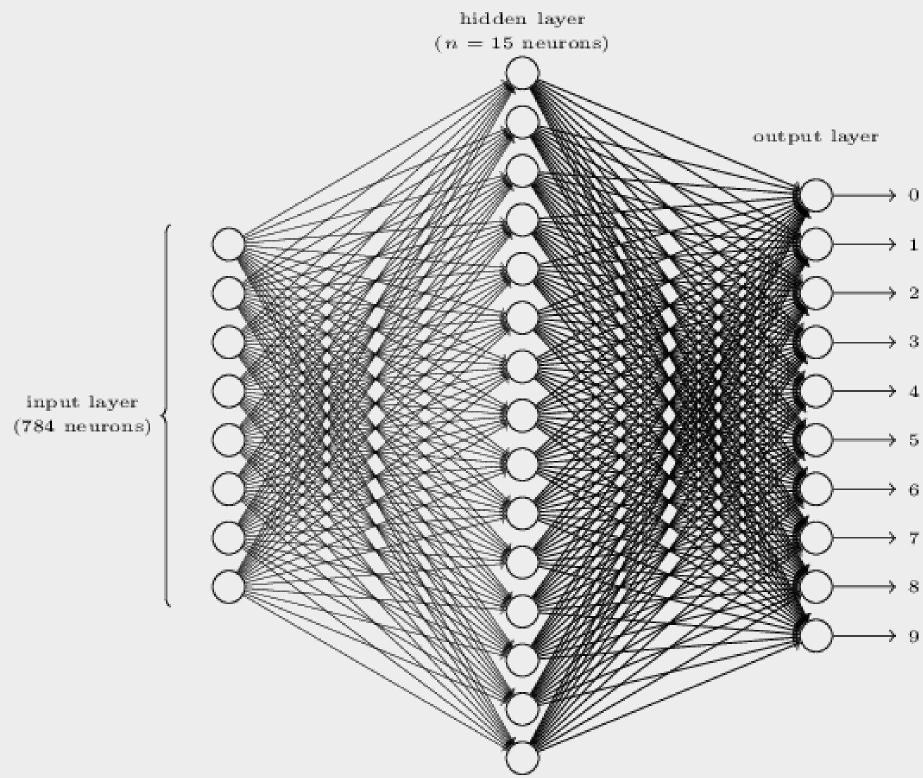
Matrix operations

- Each single neuron has parameters *w* (weight vector) and *b* (bias)
- We can represent the parameters of the entire hidden layer by combining the weight vector w_i and bias b_i to a single matrix W and bias vector b
- Each element W_{ij} represents the weight of the connection from the ith input unit x_i to the the jth hidden unit h_j
- Now, the hidden layer computation can be done very efficiently with simple matrix operations:

$$h = \sigma(Wx + b)$$

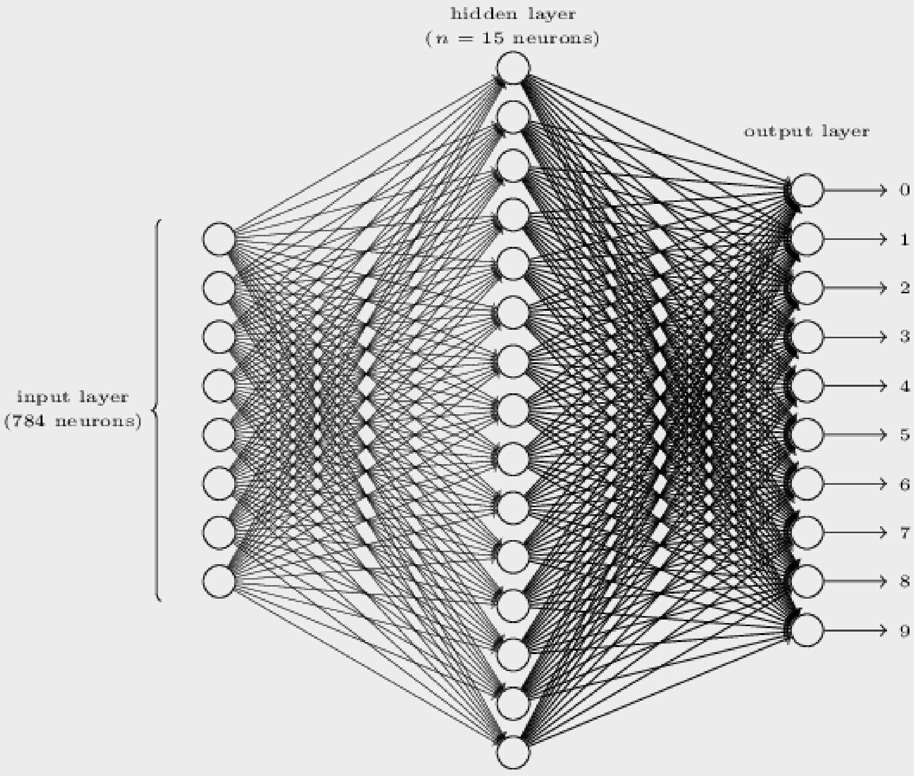
Handwritten digits classification

- Let’s look at how to do handwritten digit classification with a simple neural network
- Task:  ⟶ 5
- We will use a neural network with one hidden layer



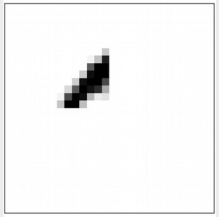
Handwritten digits classification

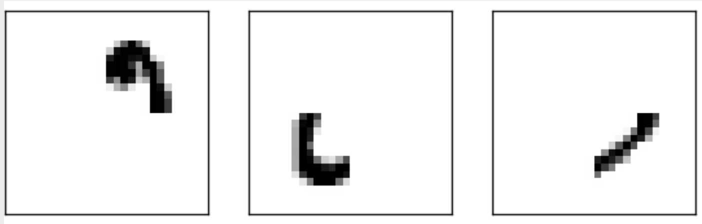
- Input neurons corresponds to grayscale values of pixels
  - Input image 28x28 pixels → 784 values
  - 0=white, 1=black
- Output layers has 10 neurons, one for each digit
  - Classification: select the output neuron with the highest value



Hidden layers as feature extractor

- How does the output neurons decide whether the input picture corresponds to their digit?
- Hidden layer(s) acts as a feature extractor, learning to detect simple component shapes
- Suppose the first four neurons in the hidden layer learn to detect whether image segments like on the left are present on the picture
- If all the four neurons in the hidden layer are “firing” (and other neurons a not), then it’s likely 0
- **The above is just a heuristic!**





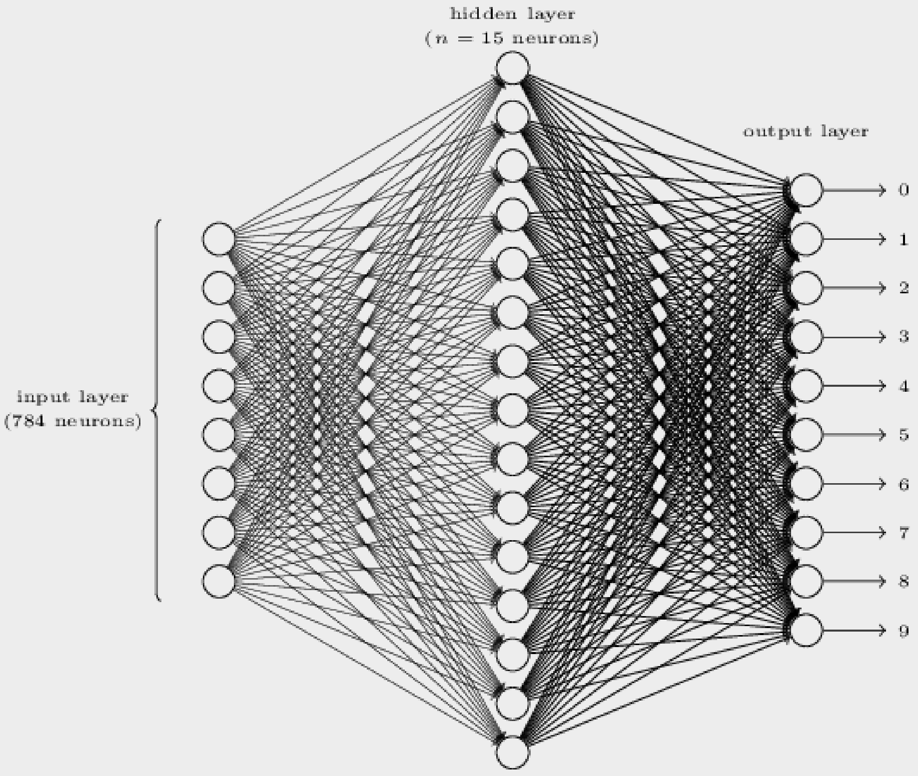
Softmax

- We can use any activation function (sigmoid, ReLU) in the hidden layer
- But what to use in the output layer?
- We would like the emitted values of output neurons to correspond to probabilities
- Sigmoid ensures that the outputs are between 0 and 1
- But we also want them to sum to 1
- Solution: softmax activation function

$$z = \sum_j w_j x_j + b$$

$$a_i = \frac{e_i^z}{\sum_k e_k^z}$$

- See a demo at http://neuralnetworksanddeeplearning.com/chap3.html#softmax



Learning

- For training, we need a **training set**
- Training set consists of inputs $\boldsymbol{x}$ and corresponding desired outputs $\boldsymbol{y}$
  - Desired outputs are encoded as 10-dimensional vectors, e.g.
    $y = [0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0]^T$
- Training finds weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify, how well the network is doing, we use a **cost function**, also know as **loss function**

Loss function

- We need a metric to quantify how well the network is doing
- A direct measure for loss is the the classification error rate
- However, classification error rate is not smooth: very small modifications to weights cause no change in error rate
  - This makes training difficult
- Thus, the loss function should be smooth
- The choice of a loss function depends on the task

Loss functions

- Loss function should be **summable**: loss of the training data must be the sum over losses of single training samples
- For neural networks with one output (doing regression):
  - Mean absolute error
  - Mean square error
- For classification (with softmax output layer):
  - Cross entropy
    - Essentially, it's the sum over the (negative) log probabilities of the network outputs corresponding to the desired class (where *y=1*)
    - Idea: we are interested in maximizing the probability that the network assigns to the desired class (and thus minimizing the probability of other classes)
    - The larger the (log) probability, the smaller the loss → thus negative (log) probability
    - But why log? So that we could **sum** over losses of individual samples
    - Beautiful loss metric: goes from 0 (model predicts perfectly) to infinity (model assigns 0 probability to the desired class)

Mean absolute error

$$L=\frac{1}{N}\sum_{i=1}^{N}\left|y^{(i)}-\hat{y}^{(i)}\right|$$

Mean square error

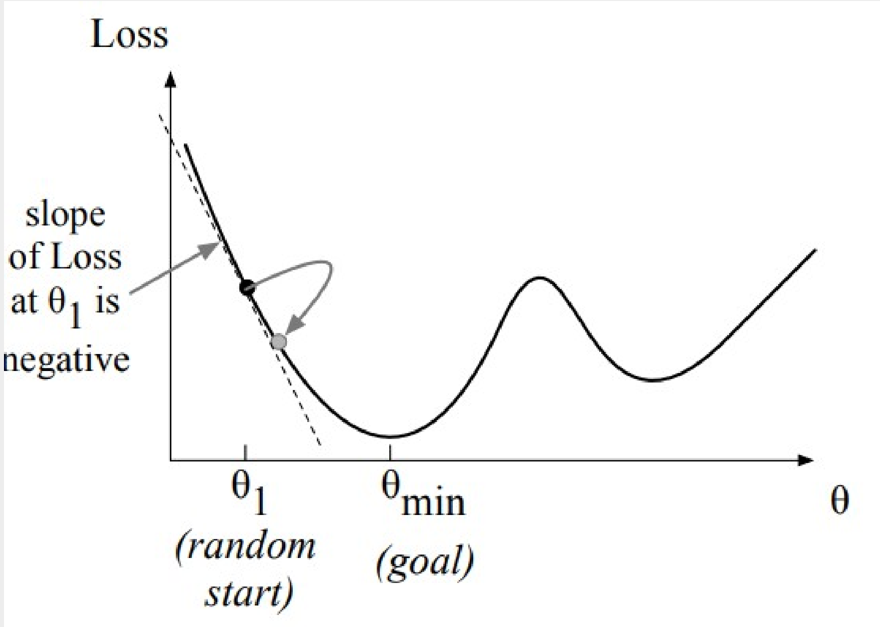
$$L=\frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)}-\hat{y}^{(i)}\right)^2$$

Cross entropy

$$L=-\frac{1}{N}\sum_{i=1}^{N}y\log\left(\hat{y}^{(i)}\right)$$

Gradient Descent

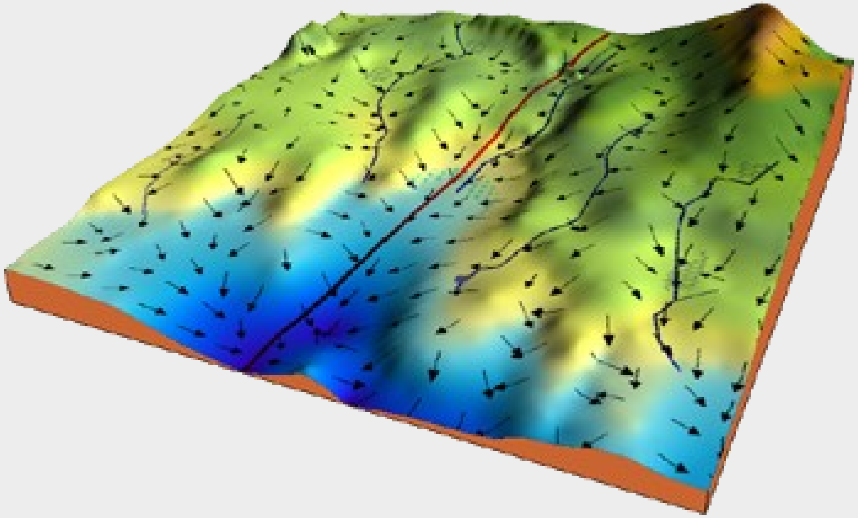
- Goal of training: minimize the loss function
- Gradient descent: figure out in which direction (in parameter space) the loss function increases most rapidly, and move in opposite direction
- This is done by finding the gradients of the loss function with regard to the parameters
- The gradient of any function $f(x_1, x_2, \ldots, x_n)$ at some certain point is a vector pointing to the direction of greatest change in function

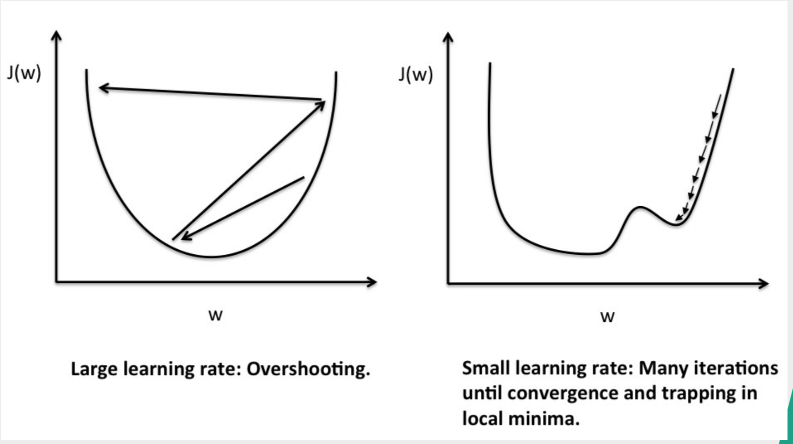


Gradient Descent

- Gradient descent is done in small steps
- Step size is determined by the learning rate
  - Large learning rate: move fast, but risk of overshooting
  - Small learning rate: slow movement (training)
- Gradient descent for function with one parameter:

$$\begin{aligned}&Given:\\&Parameter\ \theta_t\ at\ time\ t\\&Learning\ rate\ \lambda\\&\theta_{t+1}=\theta_t-\lambda\frac{d f\left(\theta_t\right)}{d\theta}\end{aligned}$$

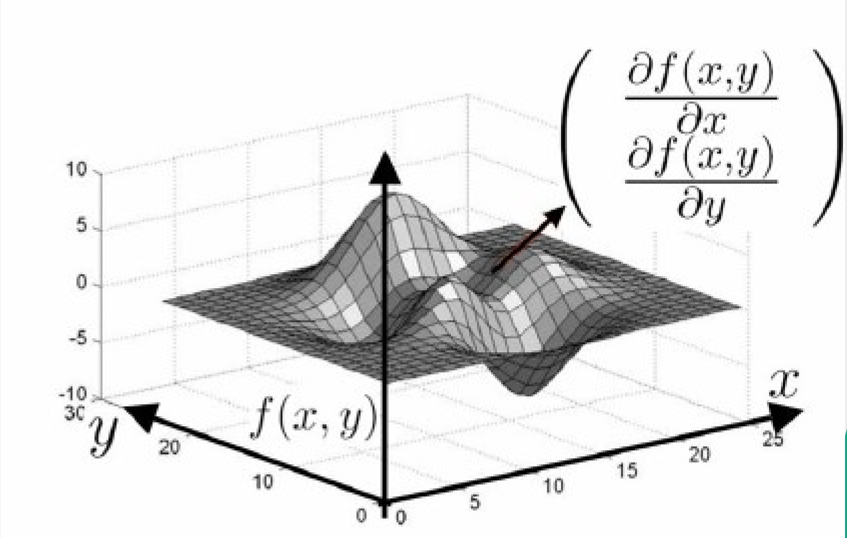




Gradient Descent

- We want to minimize the loss function ***L*** that is applied to the neural network function ***f(x; θ)***
- For more than one parameters we need partial derivatives:

$$\nabla_\theta L(f(x;\theta),y)=\begin{bmatrix}\frac{\partial}{\partial\theta_1}L(f(x;\theta),y)\\ \frac{\partial}{\partial\theta_2}L(f(x;\theta),y)\\ \vdots\\ \frac{\partial}{\partial\theta_m}L(f(x;\theta),y)\end{bmatrix}$$

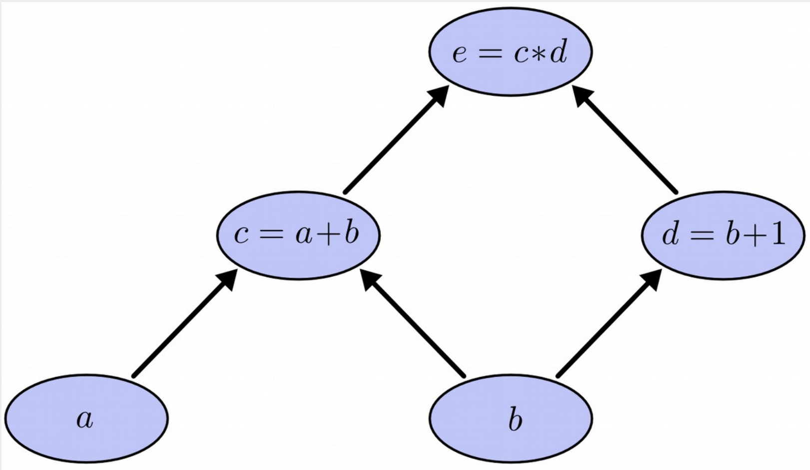


- Final gradient descent formula:

$$\theta_{t+1}=\theta_t-\lambda\nabla L(f(x;\theta),y)$$

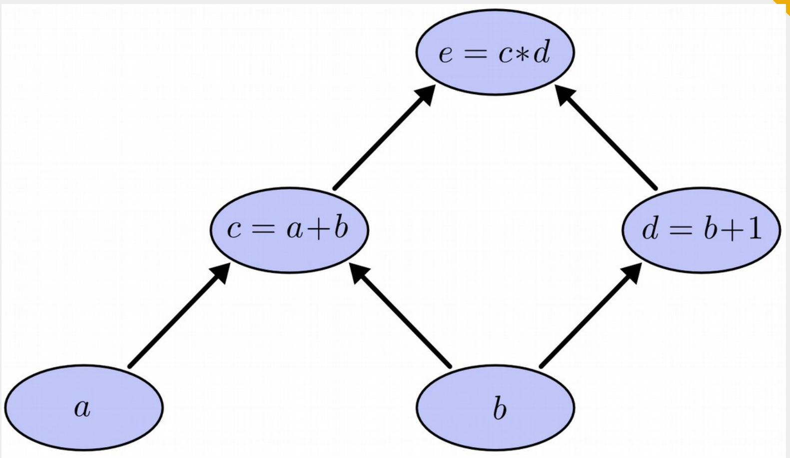
Backpropagation

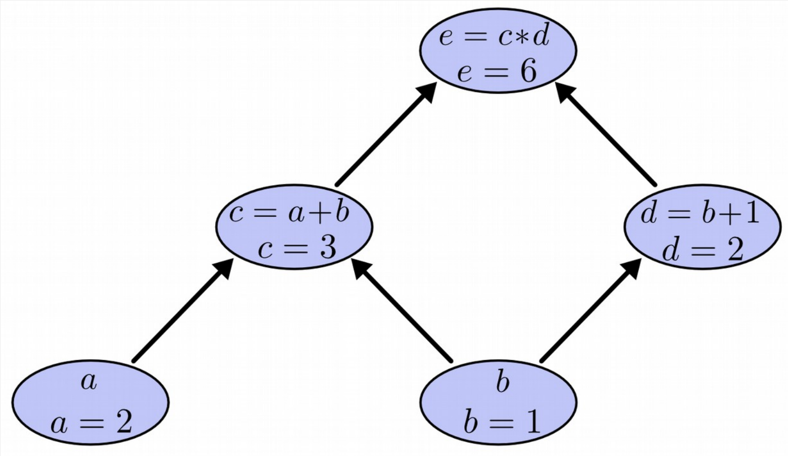
- But how to find the gradients of the complicated function that the neural network implements?
- Answer: **backpropagation**
- Backpropagation uses the **computational graph** of the network to calculate derivatives quickly
- Good explanation: http://colah.github.io/posts/2015-08-Backprop/



Computational Graph

- Consider the mathematical expression
  $e=(a+b)*(b+1)$
- We can represent it using the computational graph on the right
- Let’s introduce two intermediary variables
  $c=a+b$ and $d=b+1$
- Let’s set input variables:
  a=2 and b=1
- The expression evaluates to 6
- How to find derivatives of c with regard to a and b?





Derivatives in computation graph

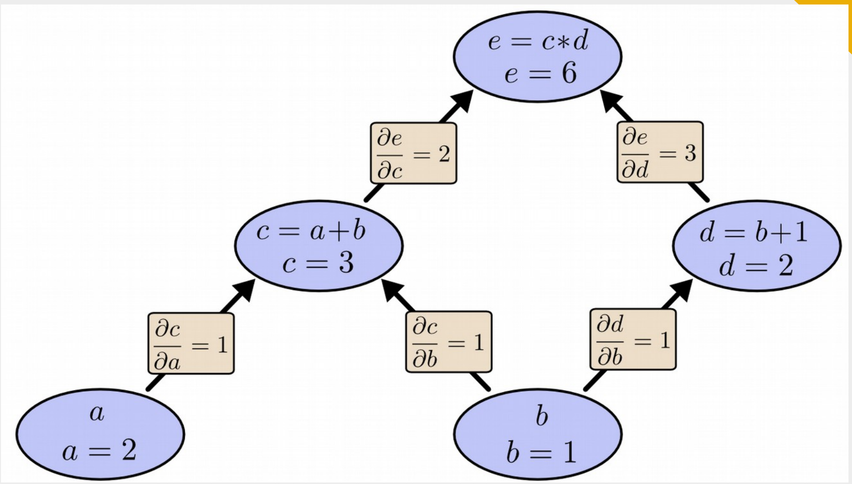
- We use the sum rule and product rule of of derivatives

$$\frac{\partial}{\partial} a(a+b)=1$$

$$\frac{\partial}{\partial} a(uv)=v$$

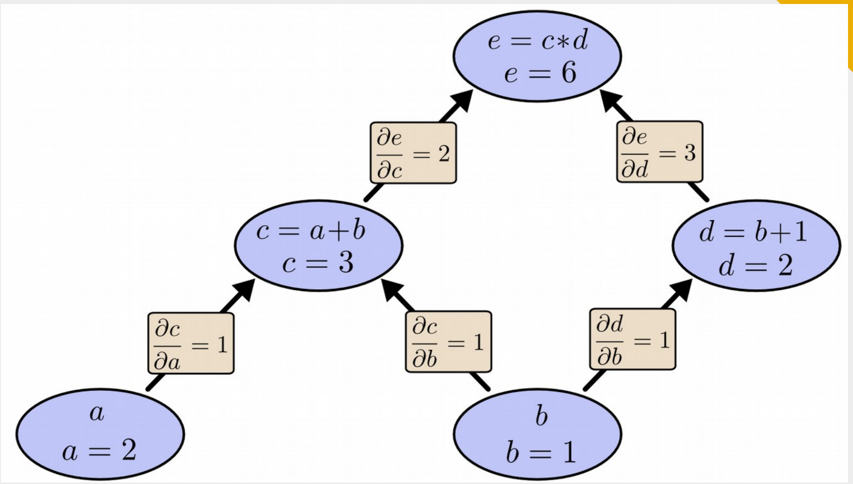
- On the left, the graph has the derivative on each edge labeled
- To find derivative of **e** w.r.t **b**, we multiply the derivatives of each path leading from e to b, and sum over all possible paths

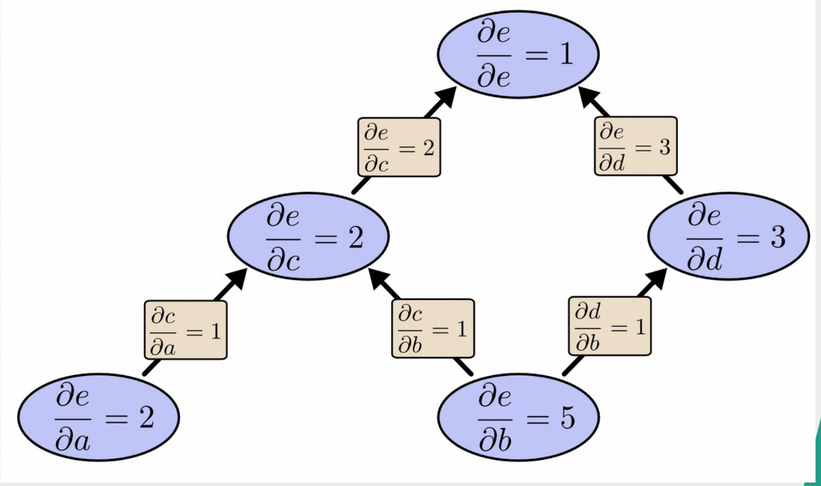
$$\frac{\partial e}{\partial b}=1*2+1*3=5$$



Backpropagation

- The partial derivative tells, e.g., how b affects c
- If derivative of c w.r.t. is 5, then it tells:
  - If we increase b by 1, changes by 5
- It’s a bit more complicated if we want to make it efficiently, considering that in neural networks, there can be thousands of paths from input to output
- In order to do it efficiently, we calculate the derivatives of e with respect to **every node** (every intermediary variable), starting from the top
- This is called **reverse-mode differentiation,** or **backpropagation**





(Batch) Stochastic Gradient Descent

- **Gradient Descent** requires calculating the gradients for the whole training set, and only then updating weights
  - This is very slow, as we usually need thousands of updates to converge
- Solution: **Stochastic Gradient Descent**
  - Take a random sample from the training set, and update the weights based on it
  - Training is much faster but noisy (accuracy on the validation set fluctuates wildly)
- Solution: **Batch Stochastic Gradient Descent**
  - Train using batches (where batch size is usually between 32 and 512) of randomly selected training examples
  - Fast and relatively stable training
  - Shuffling of training examples (or batches) on each **epoch** is important

Training

- Neural networks are usually trained until convergence
- Convergence: when the loss on a **validation set** stops to decrease
- If we train more than that, we will get an **overtrained** model