Neural Networks and Sparse Coding from the Signal Processing Perspective

Gerald Schuller

Ilmenau University of Technology
and Fraunhofer Institute for Digital Media Technology (IDMT)

April 6, 2016

Introduction

- Goal: Show Connections and shared principles between neural networks, sparse coding, and optimization and signal processing.
- You will see programming examples in Python
- This is for easier understandability,
- to test if and how algorithms work,
- and for reproducibility of results, to make algorithms testable and useful for other researchers.

Introduction Optimization

- Optimization is needed for Neural Networks, Sparse Coding, and Compressed Sensing
- Feasibility often depends on a fast and practical optimization algorithm

Introduction Optimization

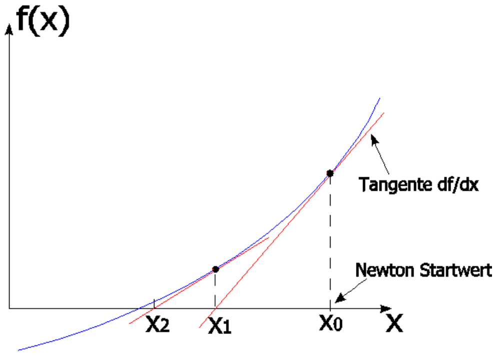
- The goal of optimization is to find the vector x which minimizes the error function $f(x)$.
- We know: in a minimum, the functions derivative is zero,

$$f'(x) := \frac{df(x)}{dx} = 0$$

.

Newtons Method

- An approach to iteratively find the zero of a function is **Newtons method**.
- Take some function f(x), where x is not a vector but just a number, then we can find its minimum as depicted in the following picture.



Newtons Method

- with the iteration

$$x_{new} = x_{old} - \frac{f(x_{old})}{f'(x_{old})}$$

- Now we want to find the zero not of $f(x)$, but of $f'(x)$, hence we simply replace $f(x)$ by $f'(x)$ and obtain the following iteration,

$$x_{new} = x_{old} - \frac{f'(x_{old})}{f''(x_{old})}$$

Newtons Method

- For a **multi-dimensional** function, where the argument $\mathbf{x}$ is a vector, the first derivative is a vector called Gradient, with symbol Nabla ∇, because we need the derivative with respect to each element of the argument vector $\mathbf{x}$,

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

(where n is the number of unknowns in the argiment vector $\mathbf{x}$).

Newtons Method

- For the second derivative, we need to take each element of the gradient vector and again take the derivative to each element of the argument vector. Hence we obtain a matrix, the **Hesse Matrix**, as matrix of second derivatives,

$$H_f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

- Observe that this Hesse Matrix is symmetric around its diagonal.

Newtons Method

- Using these definitions we can generalize our Newton algorithm to the multi-dimensional case. The one-dimensional iteration

$$x_{new} = x_{old} - \frac{f'(x_{old})}{f''(x_{old})}$$

turns into the multi-dimensional iteration

$$\mathbf{x}_{new} = \mathbf{x}_{old} - H_f^{-1}(\mathbf{x}_{old})\,\nabla f(\mathbf{x}_{old})$$

Gradient Descent

- For a minimum, $H_f(\mathbf{x})$ must be positive definite (all eigenvalues are positive).
- The problem here is that for the Hesse matrix we need to compute n^2 second derivatives, which can be computationally too complex, and then we need to invert this matrix.
- Hence we make the simplifying assumption, that the Hesse matrix can be written as a **diagonal matrix with identical values on the diagonal**.
- This leads to the widely used **Gradient Descent** or **Steepest Descent** method.

Gradient Descent

- We approximate our Hesse matrix as

$$\mathbf{H}_f(\mathbf{x}_k) = \frac{1}{\alpha} \cdot \mathbf{I}$$

- Observe that this is mostly is mostly a very crude approximation, but since we have an iteration with many small updates it can still work.
- The best value of α depends on how good it approximates the Hesse matrix.

Gradient Descent

- Hence our iteration

$$\mathbf{x}_{new} = \mathbf{x}_{old} - H_f^{-1}\left(\mathbf{x}_{old}\right)\nabla f\left(\mathbf{x}_{old}\right)$$

  with $H_f^{-1} = \alpha \cdot \mathbf{I}$ turns into

$$\mathbf{x}_{new} = \mathbf{x}_{old} - \alpha\nabla f\left(\mathbf{x}_{old}\right)$$

- which is much simpler to compute. This is also called "Steepest Descent", because the gradient tell us the direction of the steepest descent, or "Gradient Descent" because of the update direction along the gradient.

Gradient Descent

- We see that the update of $\mathbf{x}$ consists only of the gradient $\nabla f(\mathbf{x}_k)$ scaled by the factor α.
- In each step, we reduce the value of $f(\mathbf{x})$ by moving $\mathbf{x}$ in the direction of the gradient.
- If we make α larger, we obtain larger update steps and hence quicker convergence to the minimum, but it may oscillate around the minimum. For smaller α the steps become smaller, but it will converge more precisely to the minimum.

Gradient Descent Example

- Find the 2-dimensional minimum of the function

$$f(x_{0,}x_1) = \cos(x_0) - \sin(x_1)$$

- Its gradient is

$$\nabla f(x_{0,}x_1) = [-\sin(x_0), -\cos(x_1)]$$

- **Observe:** the Hessian matrix of 2nd derivatives has **diagonal** form (since it is a sum of 1-dim. functions), although not necessarily with the same entries on the diagonal, hence it is a **good fit** for the Gradient Descent

Gradient Descent Example in Python

```
ipython -pylab
alpha=1;
x=array([2,2])
#Gradient Descent update:
x= x -alpha*array([-sin(x[0]), -cos(x[1])])
print(x)
#[ 2.90929743 1.58385316]
x= x -alpha*array([-sin(x[0]), -cos(x[1])])
print(x)
#[ 3.13950913 1.5707967 ]
x= x -alpha*array([-sin(x[0]), -cos(x[1])])
print(x)
#[ 3.14159265 1.57079633]
print(pi, pi/2)
#(3.141592653589793, 1.5707963267948966)
```

Gradient Descent Example in Python

- **Observe:** after only 3 iterations we obtain π and $pi/2$ with 9 digits accuracy!
- **Keep in mind**: Gradient Descent works if its assumption of a **diagonal Hesse matrix** is true!

Gradient Descent Example 2 in Python

- Find the 2-dimensional minimum of the function

$$f(x_{0,}x_1) = exp(\cos(x_0) - \sin(x_1))$$

- Observe: it has the same minima as before, and has resemblance to non-linear functions in Neural Networks.
- Its gradient is

$$\nabla f(x_{0,}x_1) = exp(\cos(x_0) - \sin(x_1)) \cdot [-\sin(x_0), -\cos(x_1)]$$

- **Observe:** the Hessian matrix of 2nd derivatives now has **no diagonal** form (because of the non-linear exp function), hence it is **not a good fit** for the Gradient Descent anymore.

Gradient Descent Example 2 in Python

```
ipython --pylab
alpha=1;
x=array([2,2])
#Gradient Descent update:
x= x -alpha*exp(cos(x[0])-sin(x[1]))*array([-sin(x[0]), -cos(x[1])])
print(x)
#[ 2.24158659 1.88943607]
x= x -alpha*exp(cos(x[0])-sin(x[1]))*array([-sin(x[0]), -cos(x[1])])
print(x)
#[ 2.40434831 1.82434327]
x= x -alpha*exp(cos(x[0])-sin(x[1]))*array([-sin(x[0]), -cos(x[1])])
#[ 2.52613587 1.77890026]
print(pi, pi/2)
#(3.141592653589793, 1.5707963267948966)
```

Gradient Descent Example 2 in Python

- **Observe:** after 3 iterations we obtain maximally a single digit of accuracy!
- **Observe**: Gradient Descent may not work well if its assumption of a **diagonal Hesse matrix** is not true!

Artificial Neural Networks

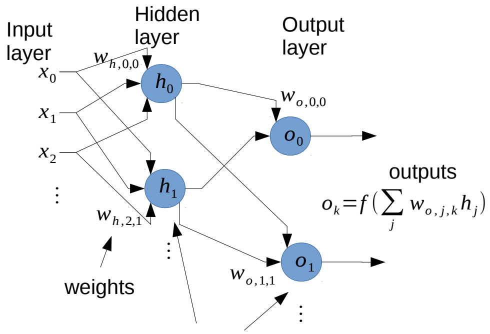
- "**(Artificial) Neural Networks**" use a weighted sum at its input and a non-linear function for its output
- They usually use several connected layers. If there are more than 3 layers, they are called "**Deep Neural Networks**", with "**Deep Learning**". These are current active research areas, for instance for speech recognition and image recognition.
- The non-linear function $f(x)$ is often the so-called **sigmoid**, (see also `https://en.wikipedia.org/wiki/Sigmoid_function`) which is defined as

$$f(x) := \frac{1}{1+e^{-x}}$$

- Its derivative is

$$f'(x) = \frac{d}{dx} f(x) = \frac{e^x}{(1+e^x)^2}$$

A Three Layer Neural Networks



$$o_k = f\left(\sum_j w_{o,j,k} h_j\right)$$

$$f(x) := \frac{1}{1+e^{-x}}$$

$$h_j = f\left(\sum_i w_{h,i,j} x_i\right)$$

A Three Layer Neural Networks

- We have the input layer with inputs x_i
- a hidden layer with outputs h_i
- an output layer with outputs o_k
- weights $w_.$
- and a desired output, also called the **target**, for the optimization of the weights, also called **training**.
- We use a quadratic error function or **loss function** for the training.

A Three Layer Neural Networks

- The output of our neural network depends on the weights **w** and the inputs **x**. We assemble the inputs in the vector **x** which contains all the inputs,
- $$\mathbf{x} = [x_{0,} x_{1,...}]$$
- and vector **w** which contains all the weights (from the hidden and the output layer),

$$\mathbf{w} = [w_{h,0,0}, w_{h,0,1}, ..., w_{o,0,0}, w_{o,0,1} ...]$$

- To express this dependency, we can rewrite the **output k** as

$$o_k(\mathbf{x}, \mathbf{w})$$

Backpropagation as Gradient Descent

- Now we would like to “train” the network, meaning we would like to determine the weights such that
- if we present the neural network with a training pattern in x, the output produces a desired value.
- We have **training inputs**, and **desired outputs** d_k.
- We use "Stochastic" **Gradient Descent** to obtain the weights $\mathbf{w}$.
- We define the **Error Function** or **Loss Function** of the k'th output as

$$Err_k(\mathbf{x}, \mathbf{w}) = 0.5 \cdot (o_k(\mathbf{x}, \mathbf{w}) - d_k)^2$$

Backpropagation as Gradient Descent

- we can now apply Gradient Descent for each output k,
- $\mathbf{w}_{new} = \mathbf{w}_{old} - \alpha \cdot \nabla Err_k(\mathbf{x}, \mathbf{w_{old}})$
- after some derivation using the chain rule we obtain for the output layer,

$$w_{o,j,new} = w_{o,j,old} - \alpha \cdot (o_k(\mathbf{x}, \mathbf{w}) - d_k) \cdot f'(s_{o,k}) \cdot h_j$$

  where $s_{o,k}$ is the weighted sum for the output layer before the non-linearity.
- This update says: **update** = **alpha** times **output difference** times **output derivative** times its **input** h_j from the hidden nodes.
- **Observe**: It only uses **local processing**, signals available at the neuron.

Backpropagation as Gradient Descent

- for the weights of the hidden layer we define a "**back propagated delta**" term for neuron j as

$$\delta_{h,j,k}(\mathbf{x},\mathbf{w}) := (o_k(\mathbf{x},\mathbf{w}) - d_k) \cdot f'(s_{o,k}) \cdot w_{o,j,k}$$

- after some derivations this results in the following update formula,

$$w_{h,i,j,new} = w_{h,i,j,old} - \alpha \cdot \delta_{h,j,k}(\mathbf{x},\mathbf{w}) \cdot f'(s_{h,j}) \cdot x_i$$

  where $s_{h,j}$ is the weighted sum for the hidden layer before the non-linearity, and x_i is the i'th input.
- Observe: this update looks quite similar to the one of the output layer.
- It says: **update** = **alpha** times **back propagated delta** times **derivative of hidden function** times its **input** x_i.

Backpropagation as Gradient Descent

- This is the famous **Backpropagation** algorithm made popular by Rummelhart and Hinton in the mid 80's.
- **Observe**: It is in principal just **Gradient Descent**.
- **We saw:** If the Hessian matrix has significant entries off its diagonal, it becomes **very slow**, as we saw in the example with the non-linearity.
- We have indeed a very similar non-linearity in our neural network case.
- Hence we can expect that Backpropagation becomes very slow.
- Hence optimization algorithms which **don't make the assumption** of a diagonal Hesse matrix could be superior in its convergence speed,
- for instance the method of **Conjugate Gradients**

Python Keras Example Neural Network

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
import numpy as np

def generate_dummy_data():
    #Method to generate some artificial data in an numpy array form in order to fit the network.
    #:return: X, Y numpy arrays used for training
    X = np.array([[0.5,1.,0], [0.2,0.7,0.3], [0.5,0,1.], [0,0,1.]])
    Y = np.array([[1], [0], [1], [1]])
    return X, Y

def generate_model():
    # Method to construct a fully connected neural network using keras and theano.
    # :return: Trainable object
    # Define the model. Can be sequential or graph
    model = Sequential()
    model.add(Dense(output_dim = 4, input_dim = 3, init="normal"))
    model.add(Activation("sigmoid"))
    model.add(Dense(output_dim = 1, input_dim = 3, init="normal"))
    model.add(Activation("sigmoid"))
    # Compile appropriate theano functions
    model.compile(loss='mse', optimizer='sgd')
    return model

if __name__ == '__main__':
    # Demonstration on using the code.
    X, Y = generate_dummy_data() # Acquire Training Dataset
    model = generate_model() # Compile an neural net
    model.fit(X, Y, nb_epoch=100, batch_size=4)
    model.predict(X) # Make Predictions
    model.save_weights('weights.hdf5') #save weights to file
```

Python Keras Example Neural Network

- "Keras" is a Deep Learning neural network library based on the libraries Theano or TensorFlow, including optimization/ training.
- optimizer='sgd' means "Stochastic Gradient Descent", or Backpropagation.
- Observe: Keras also has other optimizers, which might be more suitable to your problem
- Try the example with
  python kerasexamples.py
- **Observe**: The loss function is indeed minimized during training.
- The resulting weights can be written into a file.

Convolutive Neural Networks

- The neurons at the same layer have the same weights
- This corresponds to a *convolution* or *filtering* in signal processing
- Consider just one layer, and omit the non-linearity
- then we obtain *adaptive filters*
- apply Gradient Descent to this adaptive filter and we obtain the well known "Least Means Squares" (LMS) algorithm (Widrow, Hoff...)

Convolutive Neural Networks

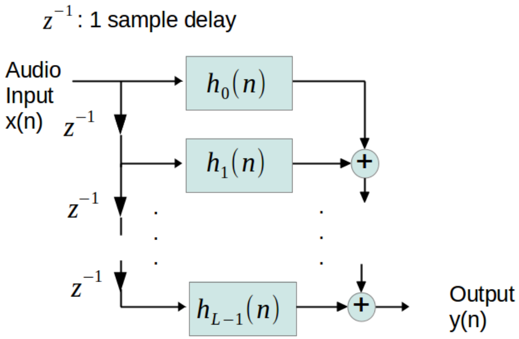


Figure : A convolutive neuron without non-linearity, or Finite Impulse Response (FIR) filter, with weights $h_k(n)$ adaptable with time n.

Convolutive Neural Networks

- Its output is $y(n) = \sum_{k=0}^{L-1} h_k(n) \cdot x(n-k)$
- Example: Adapt to a "predictable" signal like speech, have the filter or neurons trained or adapt such that they predict the next signal sample in the future. Hence $y(n)$ is the prediction for the next audio sample.
- This can be used to "de-noise" a signal, since noise often is non-predictable.
- $y(n)$ is the de-noised signal, since it is the prediction, hence the predictable part.

Adaptive Predictor

- Since our "neuron" or filter should be a predictor of the current sample, its input should be the preceding samples only.
- Hence our predicted sample $x(\hat{n})$ is computed starting with index $k = 1$,

$$\hat{x}(n) = \sum_{k=1}^{L} h_k(n) \cdot x(n-k)$$

- Our "prediction error" is the difference between the real and the predicted value, $e(n) := x(n) - \hat{x}(n)$.

Adaptive Predictor

- Our optimization goal is to minimize the expectation of the squared error as our loss function. Since we expect many iterations, we let them do the averaging and drop the expection function,

$$f(\mathbf{h}(n)) := e^2(n)$$

with the vector of weights

$$\mathbf{h}(n) := [h_1(n), h_2(n), \ldots, h_L(n)]$$

Adaptive Predictor

- We would like to apply **Stochastic** Gradient Descent (because we let the iteration do the averaging).
- For that we have to compute the first derivatives for the Gradient,

$$\frac{\partial f(\mathbf{h}(n))}{\partial h_k(n)} = 2 \cdot e(n) \cdot (-x(n-k))$$

- To check if Gradient Descent really works we need to verify that the (stochastic) Hessian Matrix of 2nd derivatives has a diagonal shape.
- The 2nd derivatives are,

$$\frac{\partial^2 f(\mathbf{h}(n))}{\partial h_k(n) \partial h_j(n)} = \frac{\partial^2 2 \cdot e(n) \cdot (-x(n-k))}{\partial h_j(n)} = 2 \cdot x(n-j) x(n-k)$$

Adaptive Predictor

- We take the expectation of the 2nd derivatives for the Hessian matrix,

$$2 \cdot E(x(n-j)x(n-k))$$

- Observe: the off-diagonal element of the Hessian are for $k \neq j$, and they only become zero or small when there is **no or little correlation** between neighboring samples.
- But then we cannot really predict the next sample
- We have a "catch-22": If our Gradient Descent works, our predictor doesn't really work, and if our predictor works, the Gradient Descent update doesn't really work well!
- But because of its simplicity we try anyway.

Adaptive Predictor

- The gradient is the vector of the first derivatives,

$$\nabla f = -2e(n) \cdot [x(n-1), \ldots, x(n-L)]$$

- The (Stochastic) Gradient Descent update is

$$\mathbf{h}(n+1) = \mathbf{h}(n) - \alpha \cdot \nabla f$$

- Absorbing the factor 2 into the α this becomes

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \alpha \cdot e(n) \cdot [x(n-1), \ldots, x(n-L)]$$

- This is the famous LMS update rule (Widrow, Hoff, 1960's)

Adaptive Predictor

- Amazingly the LMS usually works quite well, despite the violations of its assumptions, as the following example shows.
- If the non-diagonal Hessian matrix is also taken into account, it usuall results in a much faster and more robust convergence,
- for instance the "Recursive Least Squares" (RLS) algorithm,
- but it is more computationally complex.

Python Example LMS De-Noising

```
from pylab import *
import sound as snd

x, Fs=snd.wavread('fspeech.wav');#read in speech sound file in x, sample rate in Fs
x=array(x,dtype=float)/(2**15) #normalize to range -1...+1
noise=(rand(len(x))-0.5)*0.1 #uniform zero mean noise samples
x=x+noise   #add noise to speech
plot(x);xlabel('Time, Sample No.');ylabel('Sample Value');title("Noisy Speech");show()
snd.sound(x*2**15, 32000) #play de-normalized noisy speech sound
e=zeros(len(x));  #initialize
p=zeros(len(x));
h=zeros(10);   #10 weights for prediction

for n in range(10,len(x)): #for loop over sound file
   p[n]=dot(x[n-10:n], flipud(h)) #prediction using the adapted weights
   e[n]=x[n] - p[n]   #prediction error
   h= h + 1.0 *e[n]*flipud(x[n-10:n]); #LMS update rule, mu=1.0

#Plot and play out the prediction error and de-normalize:
plot(e,'r');title("Prediction Error");show()
snd.sound(e*2**15, Fs)
#Plot and play out the predicted signal:
plot(p);title("De-Noised Speech"); show()
snd.sound(p*2**15, Fs)
```

Python Example LMS De-Noising

- Start the example in a terminal window with:
  python lms_denoisesnd.py
- **Observe:** The input is speech which sounds noisy. The noise can also be seen in the plot in the speech pauses.
- The prediction error is just the noise and some parts of the speech.
- The predicted signal only contains the speech, with some distortions.

Introduction Compressed Sensing

- Compressive sensing is a relatively recent mathematical tool
- used where sampling, compression, and reconstruction is desired
- Example: Computer Tomography with as few X-ray images as possible
- it uses random sampling or random projections
- it is based on a "sparse" representation of the signal to measure in some domain
- uses so-called L1 norm minimization to find the sparse solution

Introduction

- Goal: capture the "essential" information of a signal with as few samples as possible
- Problem to solve: Regular sampling and subsequent compression need many samples and computational power for the encoder
- Approach: use non-regular sampling or combinations to capture the information with fewer samples, shift the computaional complexity to the decoder (use optimization for reconstruction)

Matching Pursuit for Overcomplete Representations

- The goal here is to approximate a given signal s(n) with a weighted sum of a minimum possible number of basis function out of a finite set of basis functions.

$$s(n) \approx \sum_{k=0}^{K-1} c_k \cdot f_k(n)$$

  with a minimum number of functions, K.
- We can also write this equation with matrices and vectors, with a signal of length L,

$$\mathbf{s} = [s(0), ..., s(L-1)]$$

Matching Pursuit for Overcomplete Representations

- and a matrix of basis vectors

$$\mathbf{T} = \begin{bmatrix} f_0(0) & f_1(0) & \cdots \\ f_0(1) & f_1(1) & \vdots \\ \vdots & & \\ f_0(L-1) & \cdots & f_{K-1}(L-1) \end{bmatrix}$$

- and our equation becomes

$$\mathbf{s}^T \approx \mathbf{T} \cdot \mathbf{c}^T$$

Matching Pursuit for Overcomplete Representations

- For instance, in case of **T** being the inverse DFT transform matrix, the vector **c** contains the frequency components for our signal **s** .
- These functions in **T** can be, for instance, the basis functions of a Discrete Cosine Transform or Discrete Sine Transform or a Discrete Fourier Transform. Basically the basis functions of **any transform(s)** in which the signal s(n) appears **"sparse"**, meaning it has only a few non-zero entries.

Matching Pursuit for Overcomplete Representations

- If we have a so-called over-complete representation, meaning more basis function that we would need to represent our signal, we obtain a space of possible solutions.
- In practice we are often looking for solutions which are “close enough” to the given target signal s. We capture this “close enough” by minimizing a quadratic norm, or L_2 norm, defined as

$$||\mathbf{s}||_2 = \left(\sum_{n=0}^{L-1} s(n)^2 \right)^{1/2}$$

Matching Pursuit for Overcomplete Representations

- Hence we are looking for all solutions with a very small L_2 norm of the difference

$$||\mathbf{s} - \mathbf{T} \cdot \mathbf{c}^T||_2$$

- We would now like to pick the solution with the **minimum number of non-zero entries** for our coefficient vector **c**.

Matching Pursuit for Overcomplete Representations

- We can formulate this as a minimization goal:
- **minimize** number of non-zero elements in c, **subject to** a minimum

$$||\mathbf{s} - \mathbf{T} \cdot \mathbf{c}^T||_2$$

- The function "number of non-zero elements.." is also called the L_0 norm, $||\mathbf{c}||_0$.

Matching Pursuit for Overcomplete Representations

- The problem is: we cannot use the L_0 norm in usual optimization routines, because we **cannot compute a derivative** for it (it is a non-continuous function).
- Hence we apply a trick: instead of using the L_0 norm, we use the closest thing to it which has a derivative in most places.
- This is the so-called L_1 norm, or $||\mathbf{c}||_1$, defined as the sum of the magnitudes of the coefficients in $\mathbf{c}$:

$$||\mathbf{c}||_1 = \sum_{k=1}^{K} |c_n|$$

Matching Pursuit for Overcomplete Representations

- This norm can now be used in usual optimization routines, and interestingly, it still converges to a sparse solution, with the minimum number of non-zero entries. So now we have the minimization formulation
- minimize $||\mathbf{c}||_1$ subject to a minimum in $||\mathbf{s} - \mathbf{T} \cdot \mathbf{c}^T||_2$

Matching Pursuit for Overcomplete Representations

- To simplify it, this is usually put in a so-called Lagrangian formulation with a Lagrange multiplier λ:
- find $\mathbf{c}$ that minimizes $||\mathbf{s} - \mathbf{T} \cdot \mathbf{c}^T||_2 + \lambda \cdot ||\mathbf{c}||_1$

An iPython Example

- Take a cosine signal with relatively high frequency, with normalized frequency of 12.5/16=0.78125 (normalized to the Nyquist frequency, which is half the sample frequency):
- `ipython -pylab; s=cos(pi/16*(arange(16))*12.5); plot(s);`

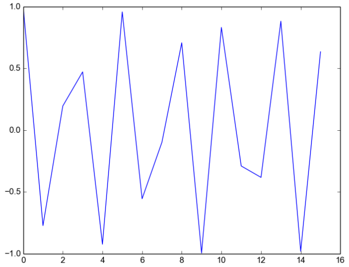


Figure : A sampled cosine signal

An iPython Example

- We use a standard the DCT Type 4, implelemented as a matrix with the following Python code,

```
def DCT4(N):
    #Calculate the DCTo (odd DCT with size NxN)
    #Args: N: (int)
    #Return: DCTo: (ndarray)

    DCT4Matrix=zeros((N, N))
    for n in range(N):
        for k in range(N):
            DCT4Matrix[n,k]=cos(pi/N*(k+0.5)*(n+0.5))
    return DCT4Matrix
```

An iPython Example

- If we transform our wave with this DCT4, in the transform domain it is not quite clear that it is a pure cosine wave:
- `from` addfunc import * #For DCT4, etc.
  specDCT = dot(s, inv(DCT4(16))); plot(specDCT);
  xlabel("DCT Coeffs.")

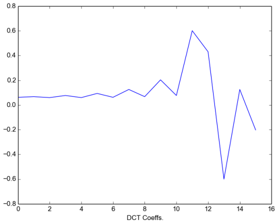


Figure : The DCT of that signal

An iPython Example

- Observe that most of the spectral coefficients are non-zero, so it is hard to say if we detected a sinusoidal signal, and if so, with which frequency and phase.
- Now we try a so-called over-complete transform, by concatenating the DCT4 and a DST4 (cosine replaced by sine) matrix, to also accommodate phase shifts
  ```
  T = hstack((DCT4(16), DST4(16)))
  ```
- Now we have an infinite number of solutions to represent our signal with a combination of basis vectors.
- But we would like to have the solution with the fewest number of non-zero coefficients for the basis vectors.

An iPython Example

- We apply optimization, which computes the coefficients for the DCT4 and DST4 basis vectors, such that the result is as close as possible to the observed signal.
- To obtain the lowest number of non-zero coefficients, we apply the L_1 norm in the minimization over all possible solutions.
- We use the Lagrange optimization, in iPython:
- `y=sum((s-dot(T, x))**2)+sum(abs(x))`
- Hence we write the following optimization function

An iPython Example

```
def optimfuncDSTDCTL1(x):
    #function to minimize, dim. of x is 32.
    #Example of matching pursiut, overcomplete transform with DCT and
        ↪ DST and L1 norm.
    #Args: x: (ndarray)
    #Return: optimfuncDSTDCTL1 : (ndarray)

    # Overcomplete transform:
    t = hstack((DCT4(16), DST4(16)))
    # Signal Example:
    s = cos(pi/16*(arange(16))*12.5)
    # Lagrange optimization:
    return sum((s-dot(t, x))**2)+sum(abs(x))
```

An iPython Example

- In iPython run the optimization with the Python function 'optimize' with a random starting point for the frequency domain coefficients:

-
  ```
  ipython --pylab
  from addfunc import *
  import scipy.optimize as opt
  xmin = opt.minimize(optimfuncDSTDCTL1, rand(32, 1))
  ```
- The output xmin contains the optimized frequency domain coefficients for the DCT and DST

An iPython Example

```
plot(xmin);xlabel("Coefficient");ylabel("Value");
title("Result of L1 Optimization")
```

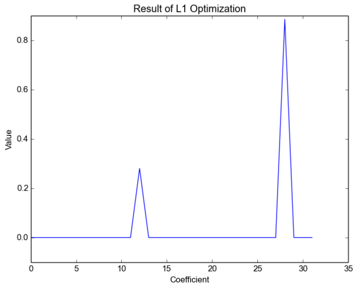


Figure : The DCT/DST coefficients from optimization. Observe: **only 2 non-zero coefficients**, 1 for the DCT, 1 for the DST matrix. This allows a precise estimate of the frequency and phase of our sinusoidal signal.

Random Sampling

- Now we can use this optimization framework for random sampling, or in general some random projections of samples (linear measurements)
- For that we define a sampling matrix Φ which produces the samples or the linear combination of samples $\mathbf{y}$,

$$\mathbf{y} = \phi \cdot \mathbf{s}^T$$

- ϕ is a $M \times L$ matrix, where M is the number of random samples or linear measurements, which is much smaller than the signal length L, $M << L$, regardless of Nyquist's Sampling Theorem!

Random Sampling

- For random sampling, this matrix contains a 1 in each row at a random position.
- **y** is the observed signal or “measurements” vector, containing our L measurements or samples.
- The goal is now to find a sparse vector of coefficients **c** such that the resulting signal is as close as possible to our observed or measured signal **y**,

$$\mathbf{y} = \phi \mathbf{s}^T \approx \phi \mathbf{T} \cdot \mathbf{c}^T$$

Random Sampling

- Hence our minimization task becomes
- find $\mathbf{c}$ that minimizes

$$||\mathbf{y} - \phi \cdot \mathbf{T} \cdot \mathbf{c}^T||_2 + \lambda \cdot ||\mathbf{c}||_1$$

Random Sampling

- We generate a reproducible or constant random sampling pattern with an average downsampling with a factor of 0.6 in iPython:
- `seed([1, 2, 3]); r=rand(16,1); randpat=r<0.6; bar(range(16), randpat); xlabel("Sample")`

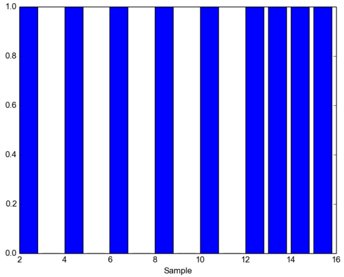


Figure : The generated random sampling pattern.

Random Sampling

- Instead of the matrix multiplication with ϕ we use a element-wise multiplication with our random vector r in this case.
- Observe that the average sampling **violates the Nyquist criterion** of our example signal, because the normalized frequency of 0.78125 of the signal is bigger than the sampling factor or new Nyquist frequency:$12.5/16 = 0.78125 > 0.6$!
- This means with regular sampling of this rate we could not detect or measure the sinusoids in the signal!

Random Sampling

- But now we try our L1 optimization approach. We use our optimization function, but now include this random sampling.
- For this we modify our Lagrange formulation to include the pseudo-random sampling pattern:
- `y = sum(((s'-T*x)` **.*randpat** `).^2) + sum(abs(x));`
- hence our optimization function now is as follows

Rand. Samp. iPython Example

```
def optimfuncDSTDCTL1randsamp(x):
    #function to minimize, dim. of x is 32
    #x is the sparse vector of unknown (DCT and DST) doefficients.
    #Example of matching pursiut and random sampling, overcomplete transform with DCT and DST and L1 (abs) norm.
    #Args: x: (ndarray)
    #Return: optmizing function value (ndarray)

    # Overcomplete transform:
    t = hstack((DCT4(16), DST4(16)))

    # Signal Example:
    s = cos(pi/16*(arange(16))*12.5)

    # random sampling with a constant pattern:
    seed([1, 2, 3])
    r = rand(16, 1)
    # only a fraction of 0.6 is randomly sampled, hence below Nyquist!:
    randpat = r < 0.6
    randpat.astype(int)

    # Lagrange optimization, with distance measure only for random samples:
    return sum(((s-dot(t, x))*randpat)**2)+sum(abs(x))
```

Rand. Samp. iPython Example

- Run the optimization:
- `from addfunc import *; import scipy.optimize as opt xmin = opt.minimize(optimfuncDSTDCTL1randsamp, rand(32, 1)); plot(xmin.x)`

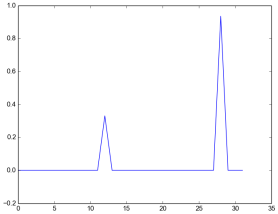


Figure : The DCT/DST coefficients from optimization, now with random sampling.

Rand. Samp. iPython Example

- Observe: We still perfectly estimated the sinusoidal components, even though our average sampling is below the Nyquist limit!
- Hence we can reconstruct the original from the randomly sampled version.
- There is a rule of thumb, saying that we need **about 5 samples per non-zero coefficient** in our representation. This is independent of the associated frequency!
- In our example we have 9 samples for 2 non-zero coefficients.

Example Random Combinations

- In cases where the signal itself is already sparse (for instance a time signal which consists of pulses), random sampling might miss those samples.
- Example: early reflections of a room impulse response
- Instead of random sampling pulses we inner products with random functions as our measurements.
- In our example we expect 2 non-zero coefficients, so we need about 5*2=10 random functions.
- Our program now becomes as follows

Rand. Comb. iPython Example

```
def optimfuncDSTDCTL1randcomb(x):
    #function to minimize, dim. of x is 32
    #x is the sparse vector of unknown (DCT and DST) doefficients.
    #Example of matching pursiut and random sampling, overcomplete transform with DCT and DST and L1 (abs) norm.
    #Args: x: (ndarray)
    #Return: optmizing function value (ndarray)

    # Overcomplete transform:
    t = hstack((DCT4(16), DST4(16)))

    # Signal Example:
    s = cos(pi/16*(arange(16))*12.5)

    # random sampling with a constant pattern:
    seed([1, 2, 3])
    # random measurement matrix PHI, with 5 measurements per non-zero coefficient (2 coeff)
    PHI = rand(10, 16)

    # Lagrange optimization, with distance measure only for random samples:
    return sum(((PHI*s)-(PHI*dot(t, x)))**2)+sum(abs(x))
```

Rand. Samp. iPython Example

- We let it run with:
- `xmin = opt.minimize(optimfuncDSTDCTL1randcomb, rand(32, 1))`
- After a much longer optimization time (matrix multiplication is more computational complex than sampling), we arrive at the same result, as expected: `plot(xmin.x); xlabel("DCT/DST Coefficient No.")`

Rand. Samp. iPython Example

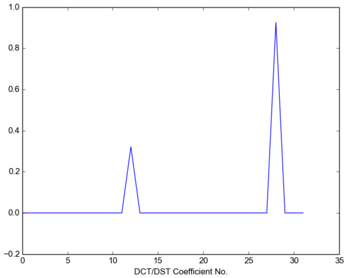


Figure : The DCT/DST coefficients from optimization, now with random sampling functions.

- Observe: This approach works for all signal representations or transforms, in which our signal has a sparse representation in some domain!

Conclusions

- We saw that Gradient Descent is a special case of the Newton Method with the assumption of a diagonal Hessian matrix of 2nd derivatives.
- Backpropagation results from the application of Gradient Descent to neural networks, even though the assumtion of a diagonal Hessian matrix is not fulfilled.
- In adaptive filters, a linear relative of convolutional networks, the application of Gradient Descent leads to the popular LMS algorithm, even though the Hessian matrix is also not diagonal in most cases.

Conclusions

- We saw: If our signal has a sparse representation in some domain:
- We can find the sparse representation in that domain with a method called Matching Pursuit, if we use the L_1 norm to minimize the number of non-zero coefficients.
- This method can be easily extended to the case of a randomly sampled signal, where the number of samples is about 5 times the expected number of non-zero coefficients, regardless of Nyquists Theorem.
- It also works with the same number of random measurement functions instead of samples, which is useful for already sparse signals.
- Slides and Python examples will be available at http://www.macsenet.eu/SpringSchool/index.php#1—

Some References

- P.M. Clarkson, "Optimal and Adaptive Signal Processing", CRC Press
- J.S. Lim, A.V. Oppenheim (Eds.), Advanced Topics in Signal Processing", Prentice Hall.
- J. Haupt and R. Nowak, "Adaptive sensing for sparse recovery, " November 2010, to appear in Compressed Sensing: Theory and Applications, Y. Eldar and G. Kutyniok eds., Cambridge University Press.
- Y.C. Eldar, G. Kutyniok (Eds.), "Compressed Sensing, Theory and Applications", Cambridge University Press.
- M. Davenport, "The Fundamentals of Compressive Sensing", IEEE Signal Processing Society Online Tutorial Library, April 12, 2013.