PHY604 Lecture 25

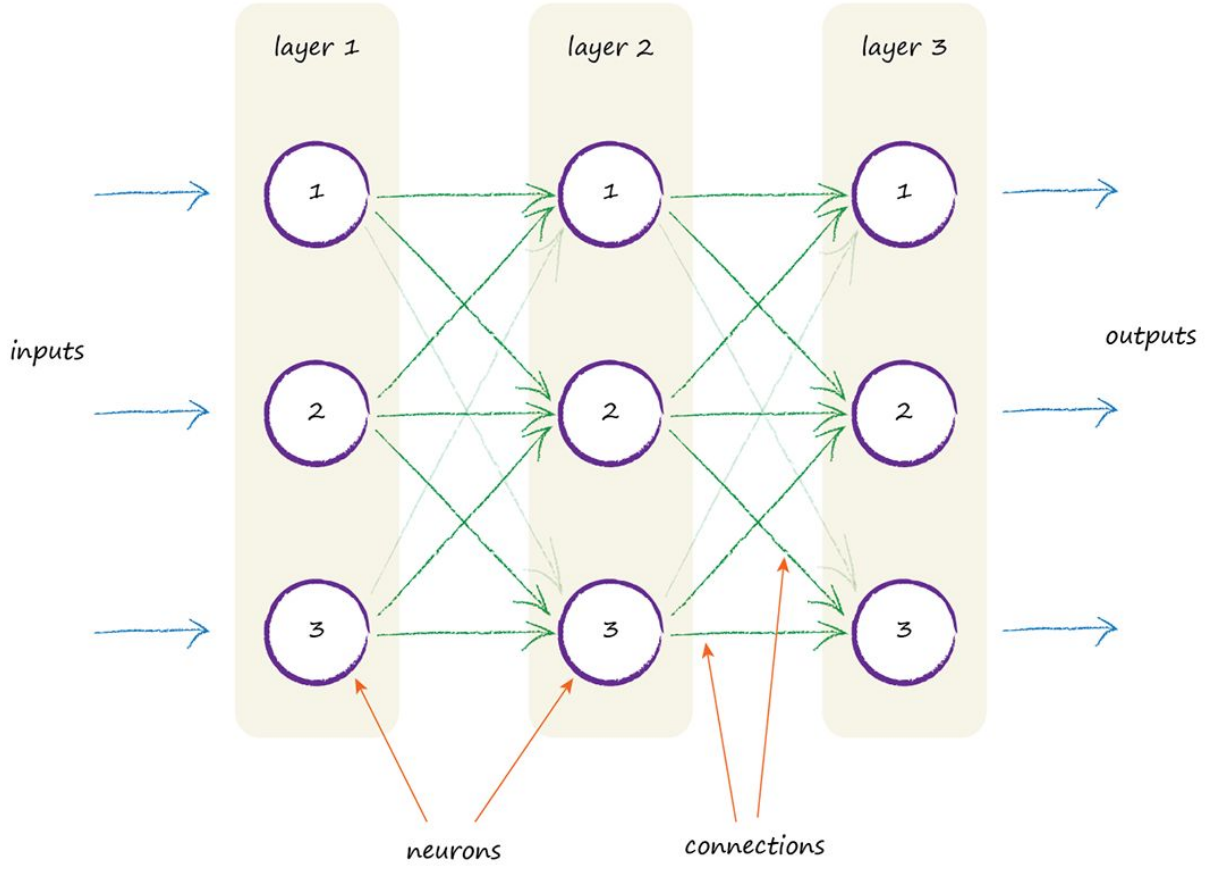
November 18, 2021

Review: Pattern recognition with computers

- Classic problem: Identify pictures of dogs versus cats
  - Easy for human, difficult for computer

Review: Nonlinear functions at the basis of neural networks

- Neural networks are divided into *layers*
  - Input layer accepts the input
  - Output layer outputs results
- Each layer has neurons (or nodes)
  - For input, one node for each input variable
  - Every node in the first layer connects to every node in the next layer
- Weight associated with the connection can be adjusted
  - These are the matrix elements
- Operations at neurons given by nonlinear activation function



Make Your Own Neural Network, Tariq Rashid

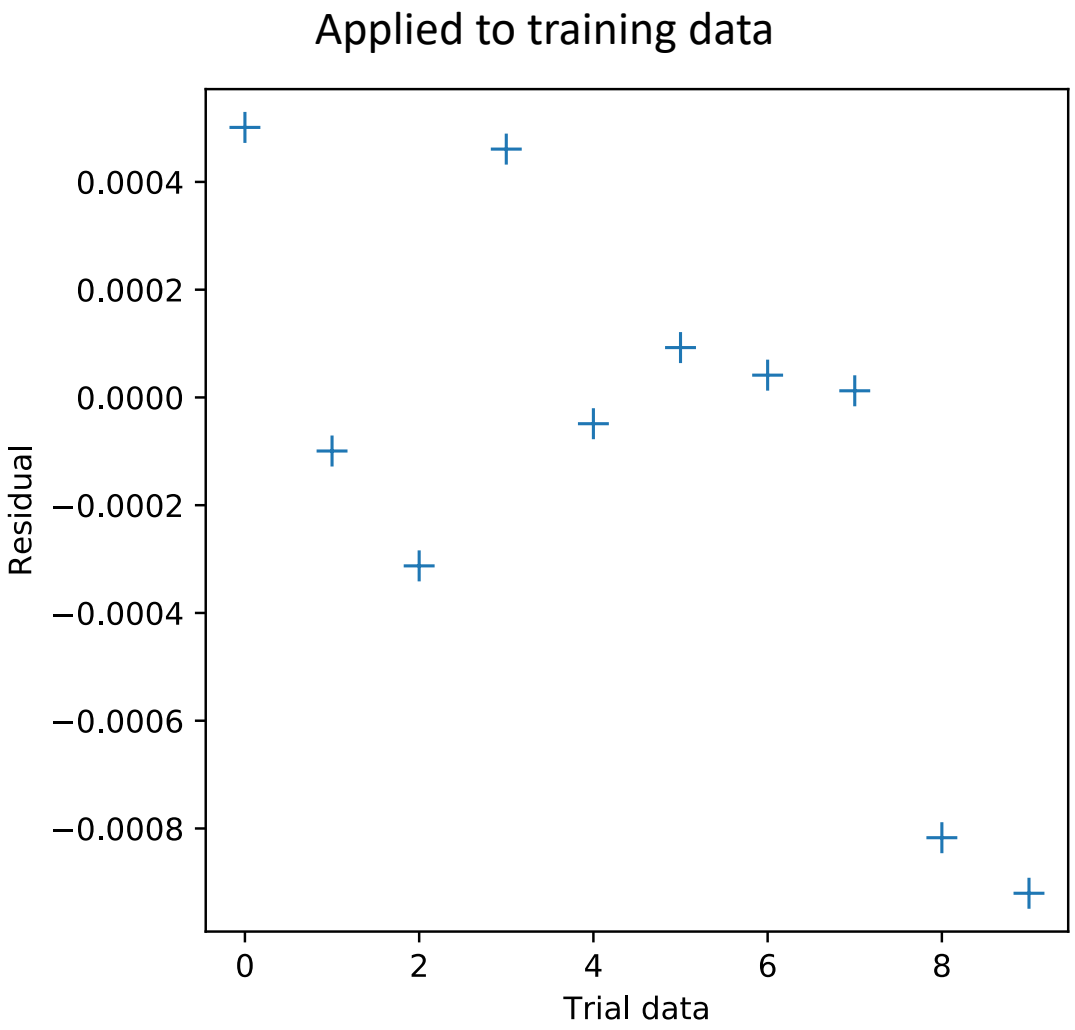
Review: Procedure for doing "Machine Learning" with neural network

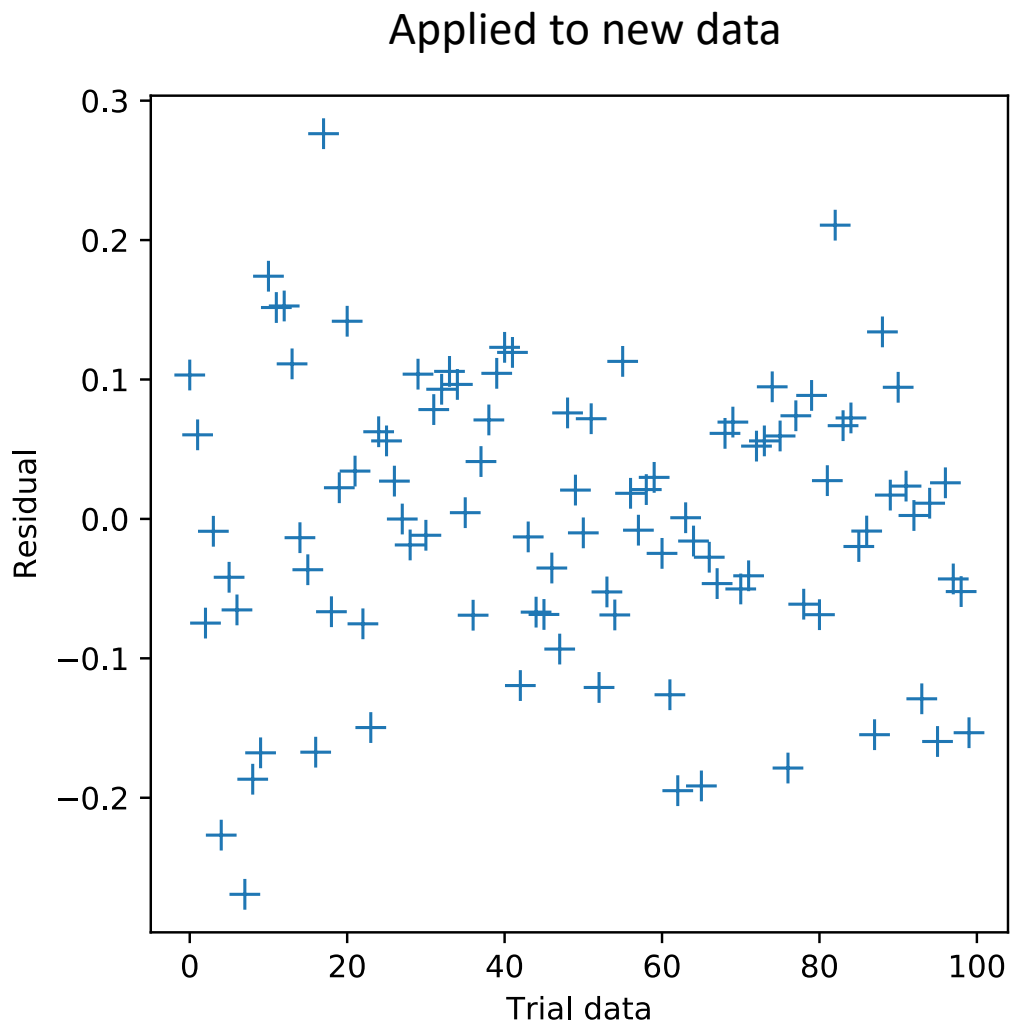
- 1. Choose a nonlinear activation function (in our case, find α)
- 2. Choose/generate *t* input/output pairs for training
- 3. Repeat the set from step 2 *N* times (epochs) to get a training set of *T*=*Nt* pairs
- 4. Run the training set through the neural net at random, performing the steepest descent minimization for each
- 5. To test the training in step 4, run the *t* examples through and calculate the residual:

$$g(\mathbf{A}x_j) - z_j$$

- 6. Use the neural net on some new data

Review: Results choosing the 10th number with a NN





$$\mathbf{A} = \begin{bmatrix} -0.04 & 0.42 & 0.15 & -0.23 & 0.13 & 0.06 & 0.19 & -0.42 & 0.48 & 4.45 \end{bmatrix}$$

Today's lecture:
Neural networks and parallel computing

- Neural network examples
  - Interpreting a noisy signal
  - Identifying hand-written characters

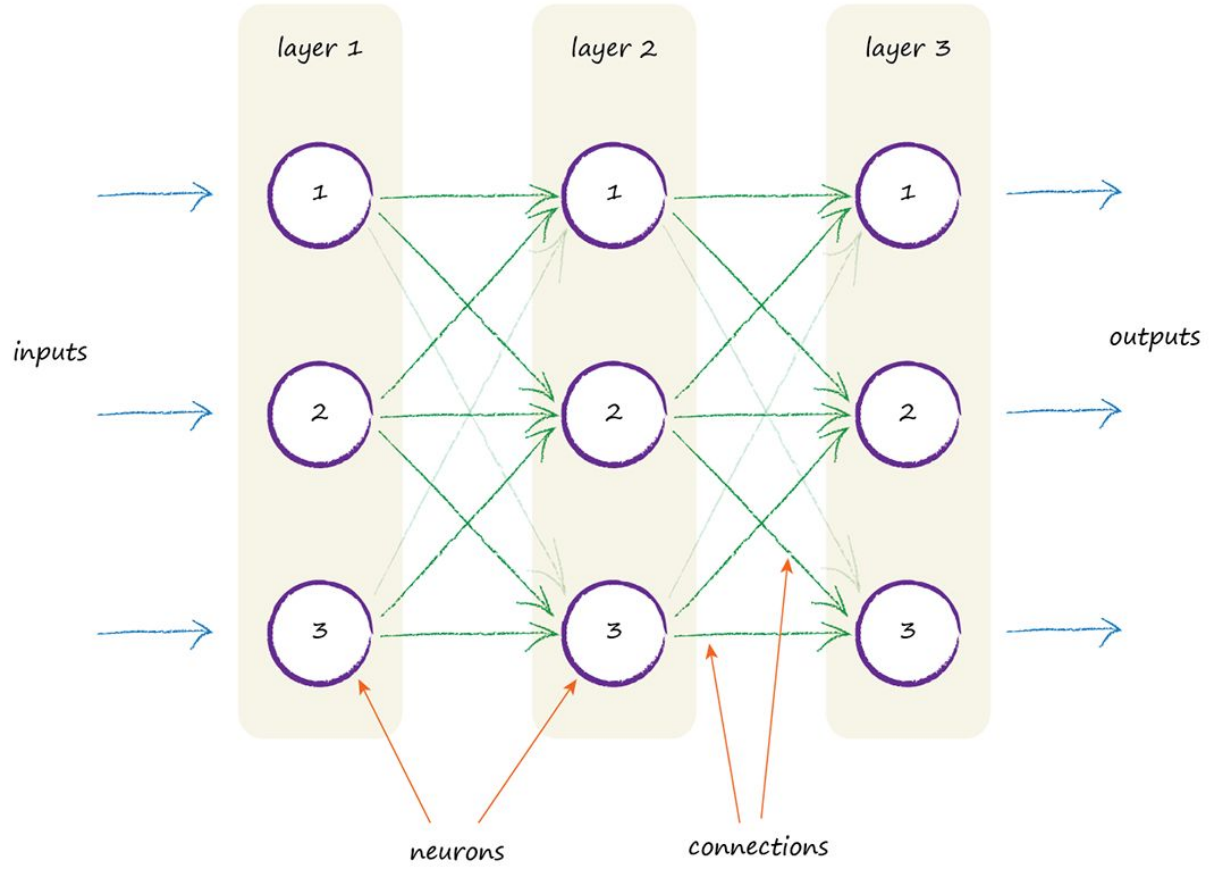
- Parallel computing
  - OpenMP

Adding additional degrees of freedom

- In the previous example, the number of adjustable parameters is constrained by the size of the input and output

- To overcome this limitation, we can add hidden layers to our neural net

- Will need an additional matrix and an additional evaluation of out nonlinear function



Make Your Own Neural Network, Tariq Rashid

Hidden layers

- Take as input a vector *x* of length *n*
- Take as output a vector *z* of length *m*
- Consider a *k* x *n* matrix **B** and a *m* x *k* matrix **A**
- Construct the output as:

$$\widetilde{z} = g(\mathbf{B}x) - \frac{1}{2}, \qquad z = \widetilde{g}(\mathbf{A}\widetilde{z})$$

- Note that we differentiate the applications of *g* because they may have different α's
- Extra shift of ½ is to recenter the data around 0 to put it in the nonlinear range of *g*
- Key: *k* is independent of the size of input/output!
  - Can train *k*(*m*+*n*) total elements

Implementing the hidden layer

- We still want to minimize our cost function *f*:

$$f(A_{rs}, B_{ij}) = \sum_{r=1}^{m} (z_r - y_r)^2$$

- Now we have to do two interrelated steepest descent minimizations:

$$A_{pq} = A_{pq} - \eta \frac{\partial f}{\partial A_{pq}}, \qquad B_{pq} = B_{pq} - \eta \frac{\partial f}{\partial B_{pq}}$$

- Where:

$$\frac{\partial f}{\partial A_{pq}} = 2\widetilde{\alpha}(z_p - y_p) z_p (1 - z_p) \widetilde{z}_q \equiv \sigma_p \widetilde{z}_q$$

$$\frac{\partial f}{\partial B_{pq}} = \sum_{r=1}^{m} \sigma_r A_{rp} \alpha \left(\frac{1}{2} + \widetilde{z}_p\right) \left(\frac{1}{2} - \widetilde{z}_p\right) x_q$$

Back propagation

- Note that we are optimizing simultaneously **A** and **B**:

$$\frac{\partial f}{\partial A_{pq}} = 2\widetilde{\alpha}(z_p - y_p)z_p(1 - z_p)\widetilde{z}_q \equiv \sigma_p \widetilde{z}_q$$

$$\frac{\partial f}{\partial B_{pq}} = \sum_{r=1}^{m} \sigma_r A_{rp} \alpha \left(\frac{1}{2} + \widetilde{z}_p\right)\left(\frac{1}{2} - \widetilde{z}_p\right) x_q$$

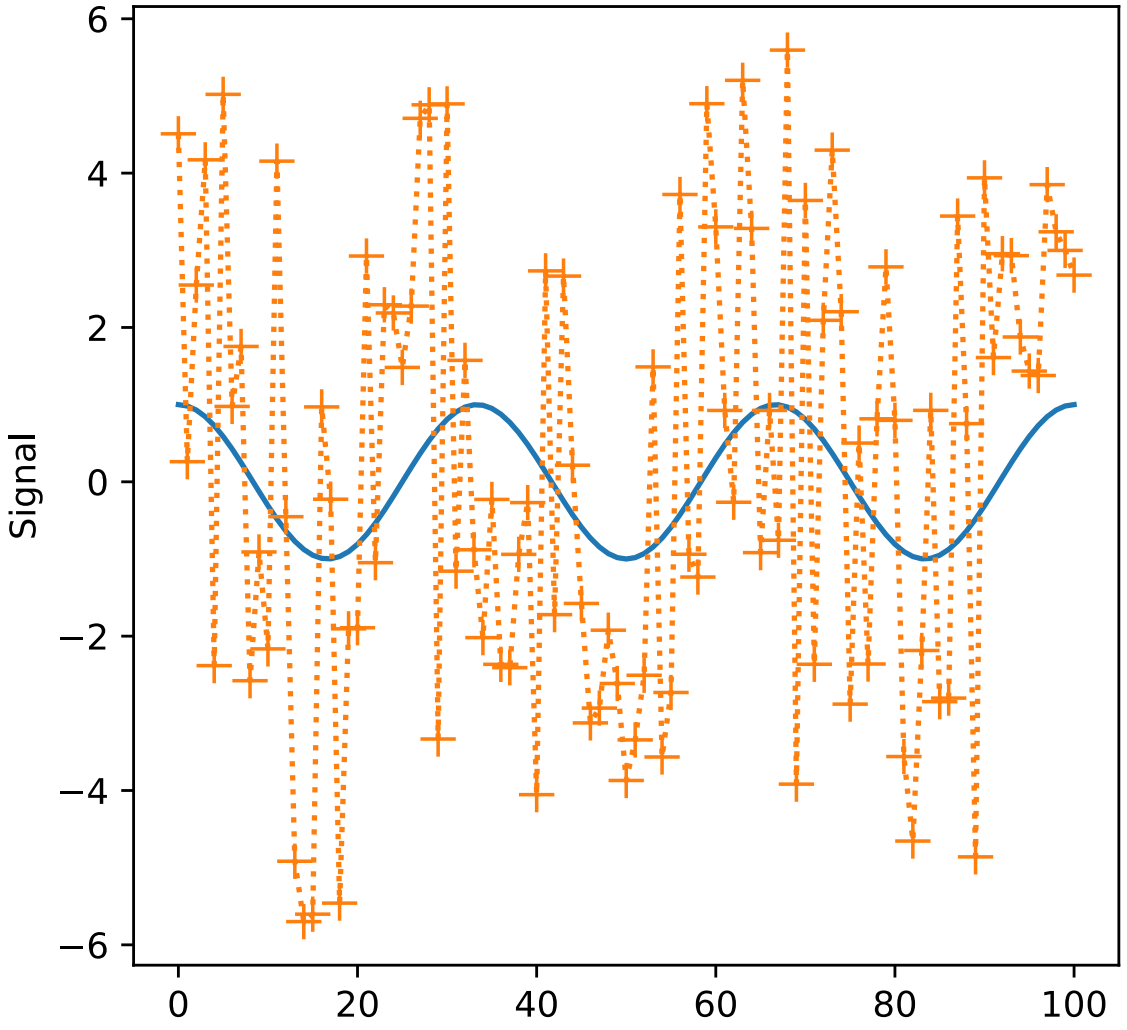
- So, the errors are “backpropagated” through the output and hidden layers

Example: Signal analysis

- Given a noisy signal expected to be one of four frequencies
  - f ={1,2,3,4} Hz
- Noise is significantly larger than the underlying signal:

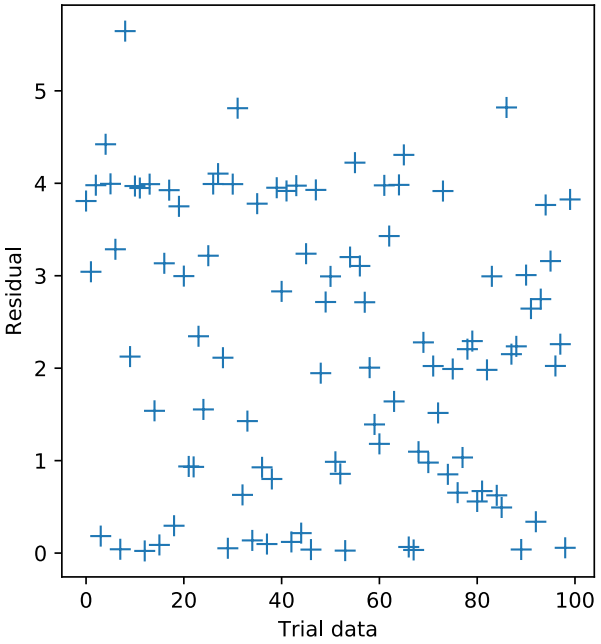
$$s(t) = \cos(2\pi f t) + 5\xi$$

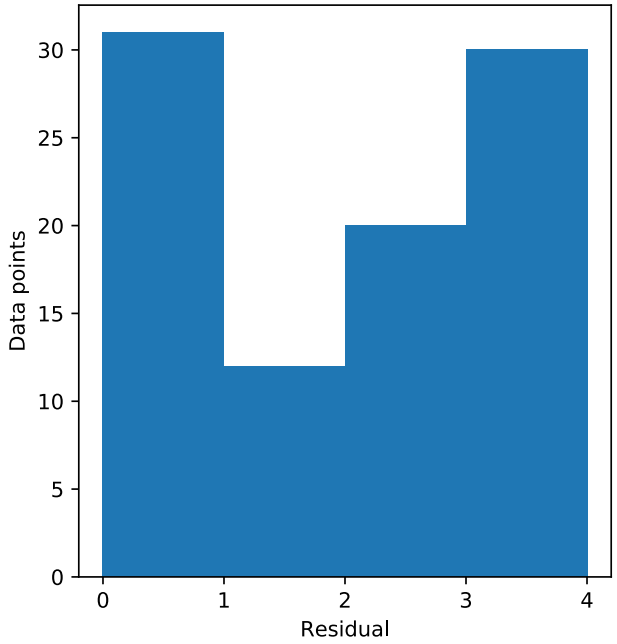
  - ξ are random numbers in [-1,1]
- Can we identify the frequency?



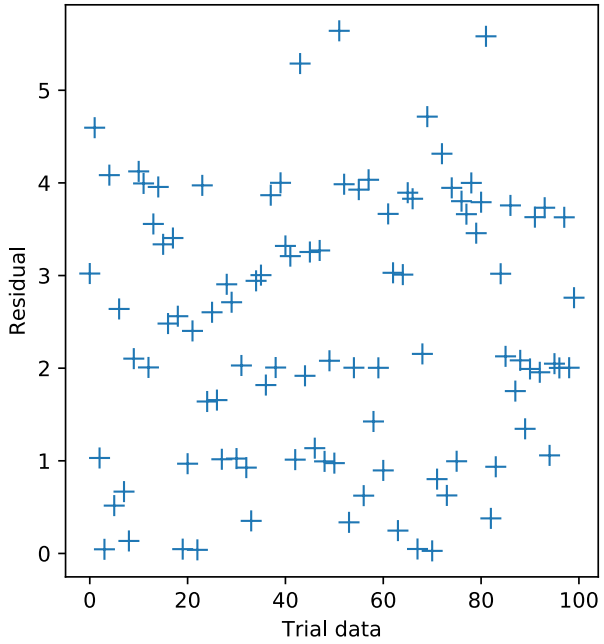
Signal analysis: Hidden layers size 2

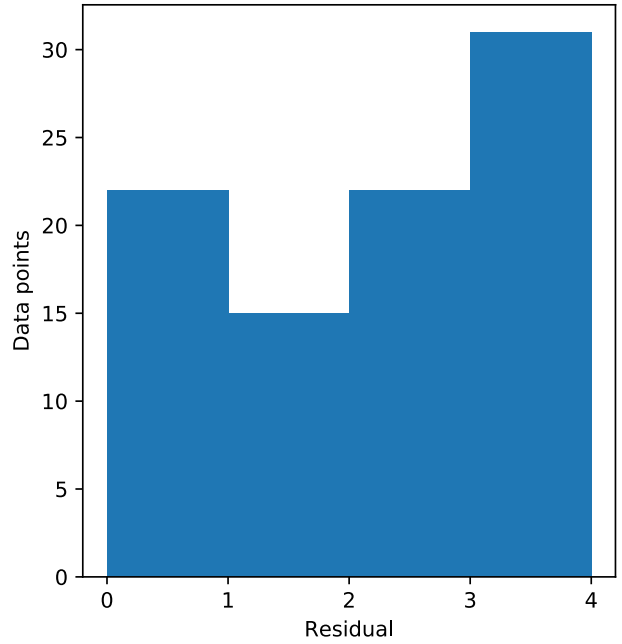
Test on the training set





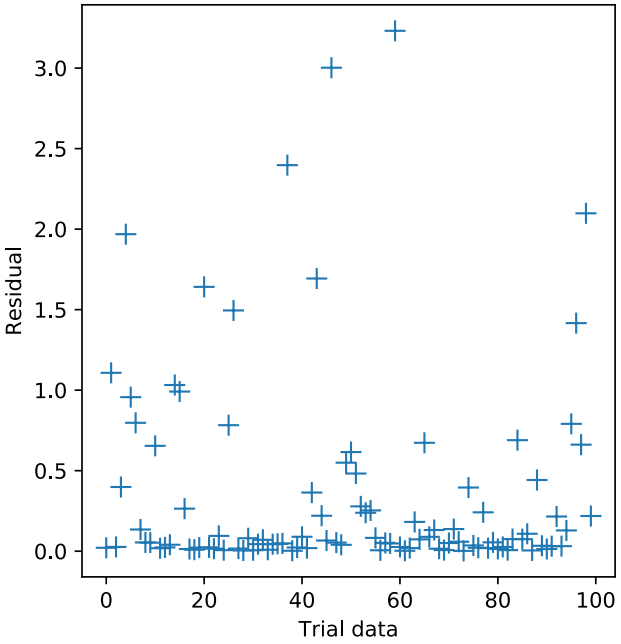
Test on new data

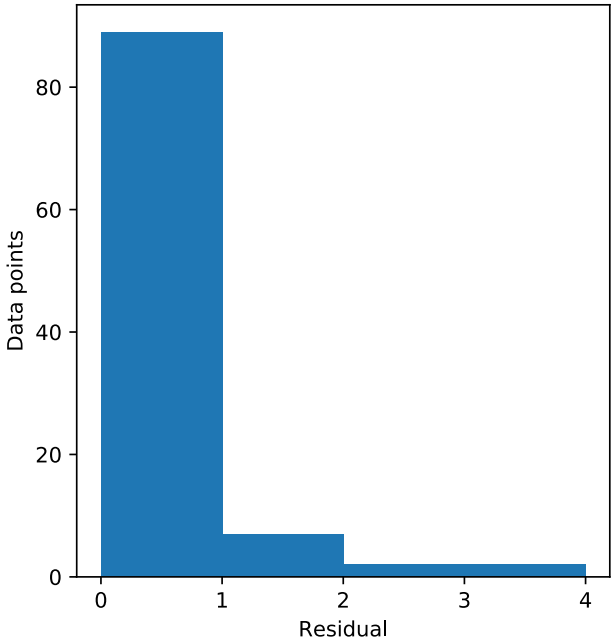




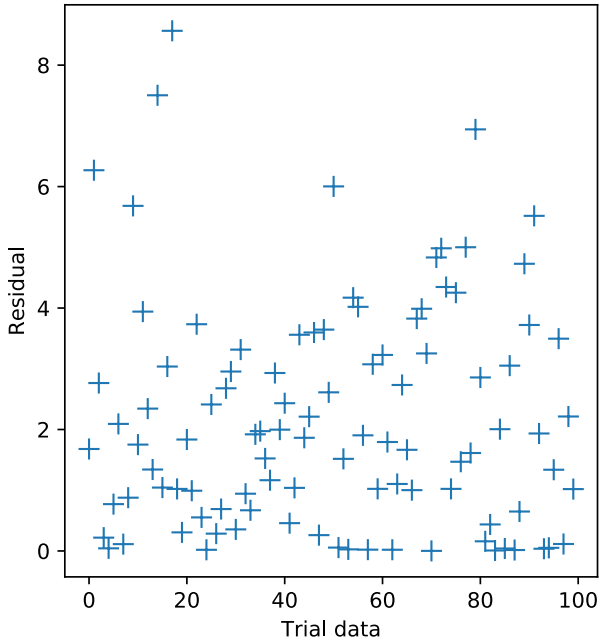
Signal analysis: Hidden layers size 3

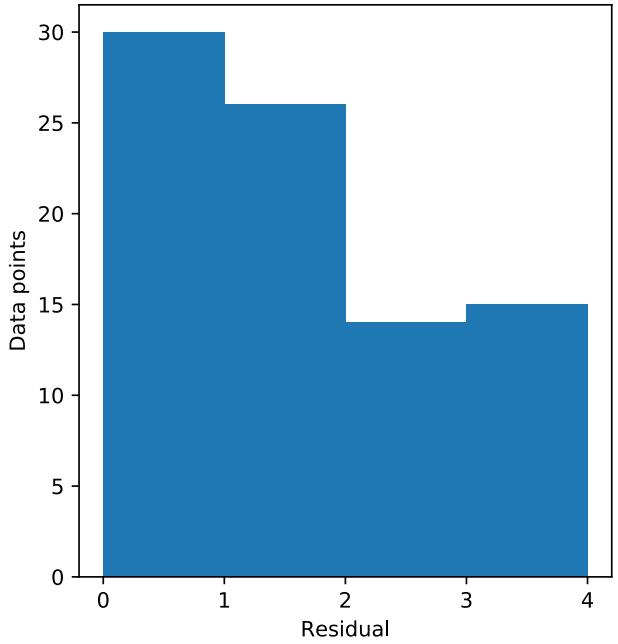
Test on the training set





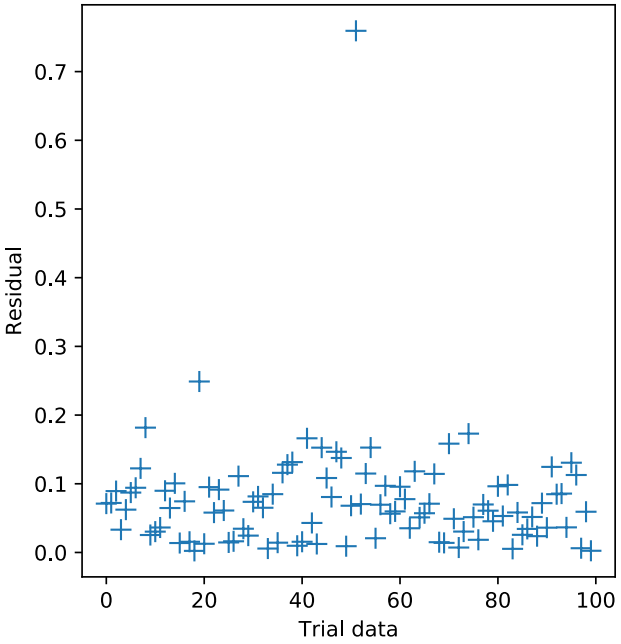
Test on new data

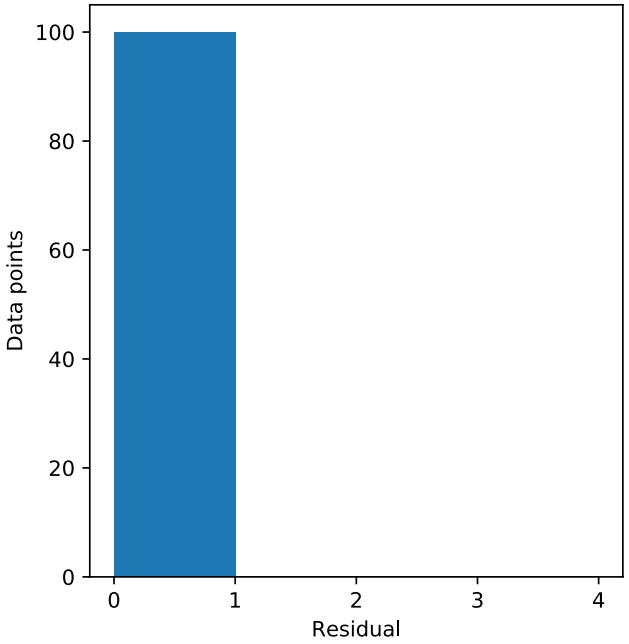




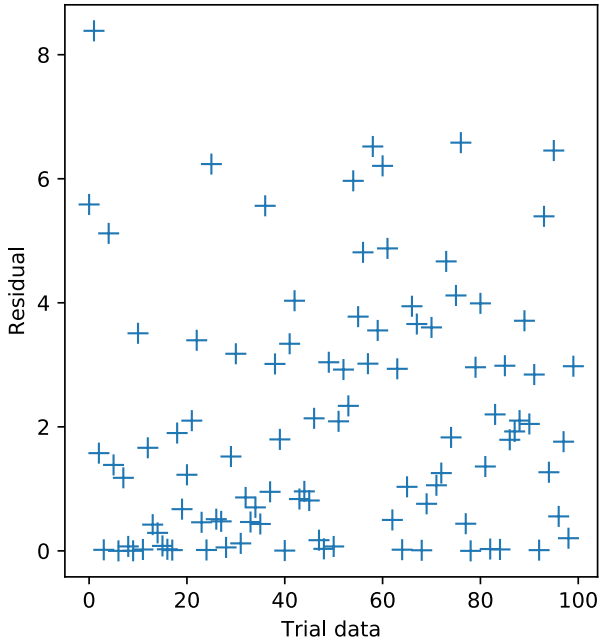
Signal analysis: Hidden layers size 4

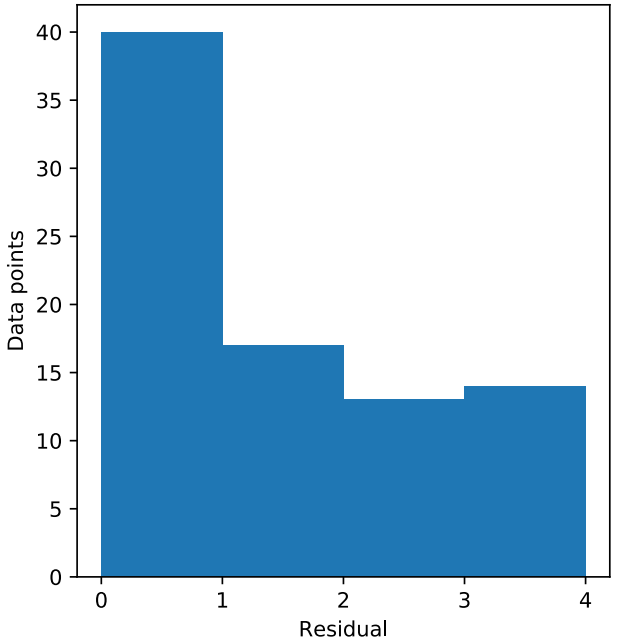
Test on the training set





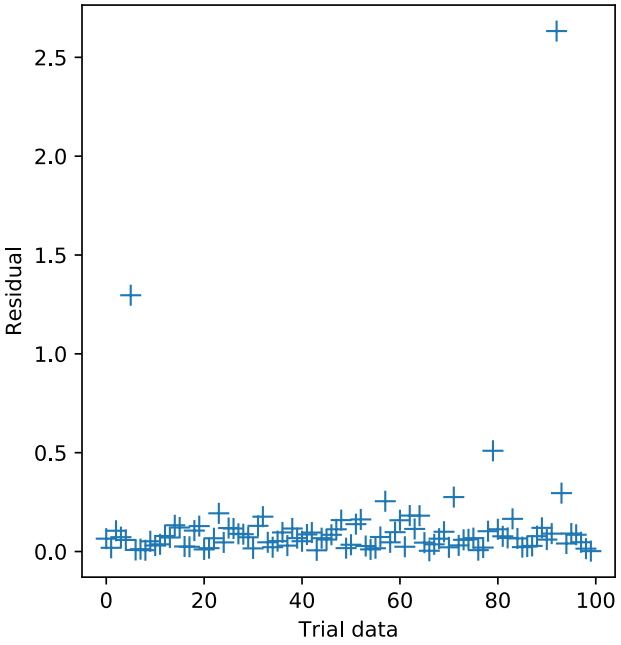
Test on new data

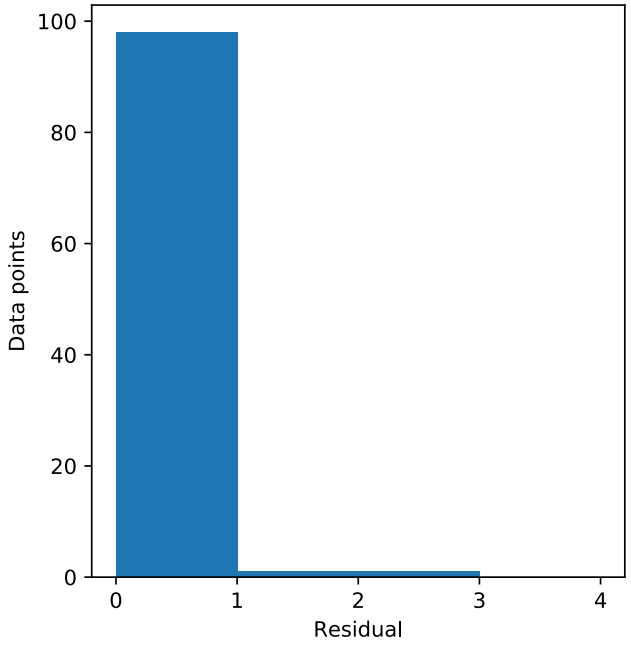




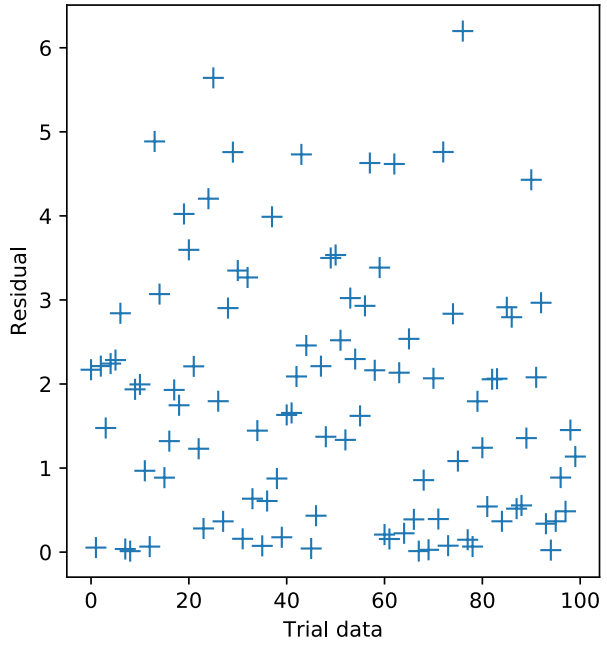
Signal analysis: Hidden layers size 8

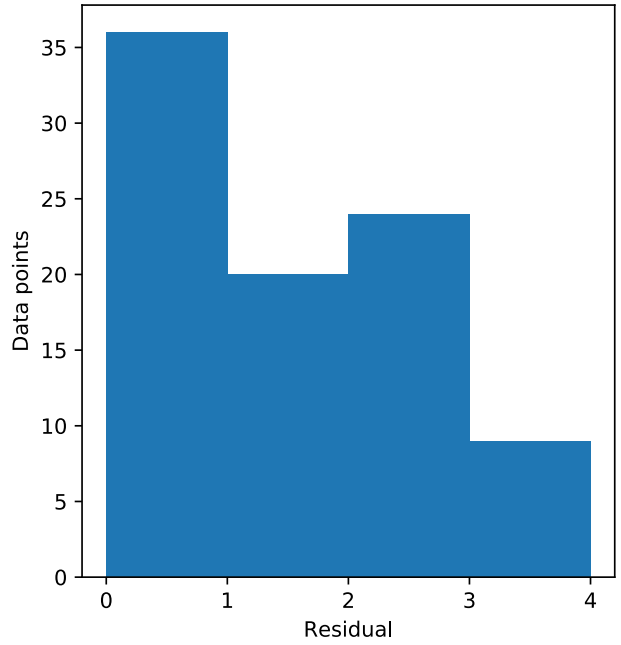
Test on the training set



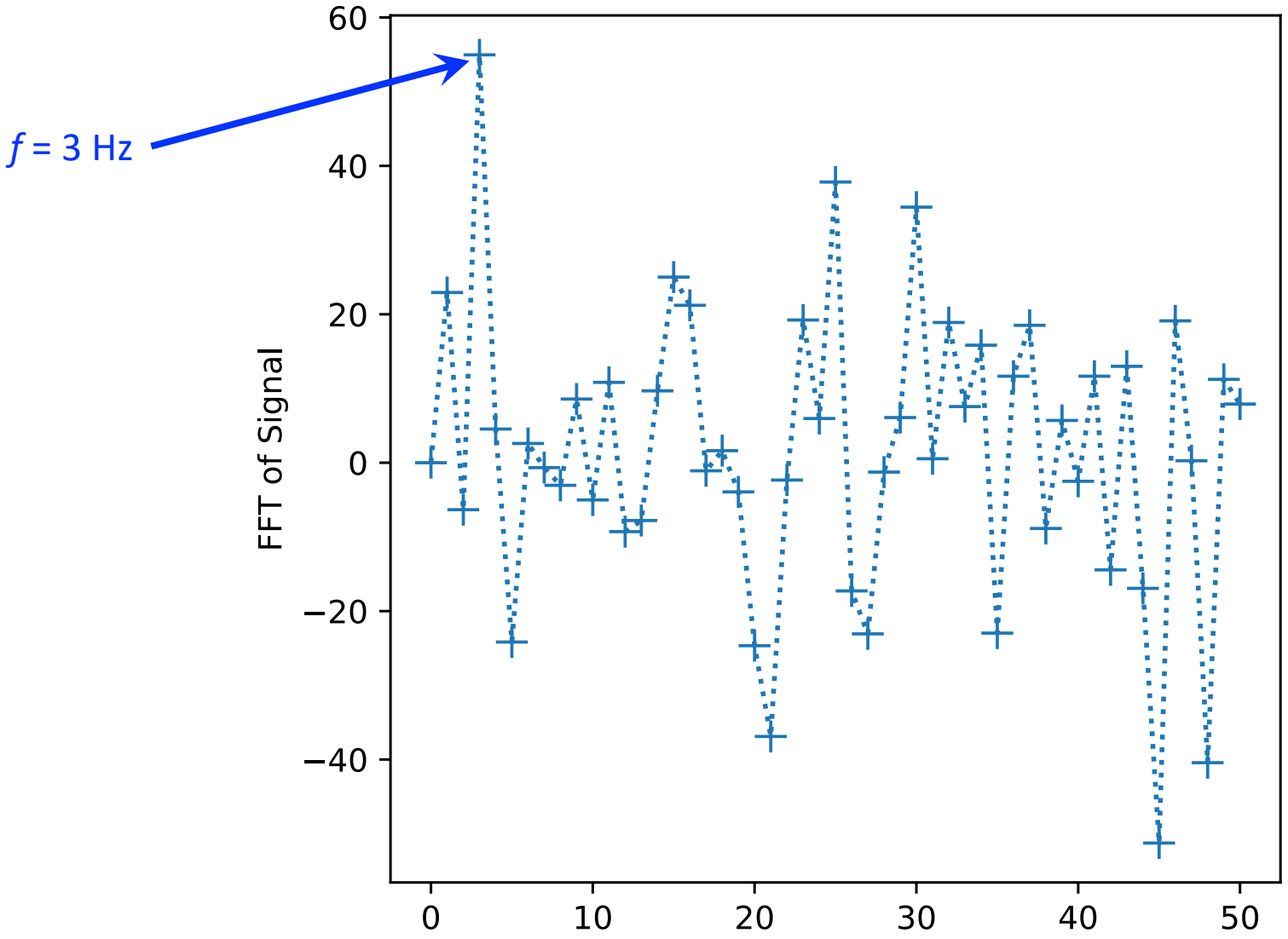


Test on new data





Can we do the same with an FFT



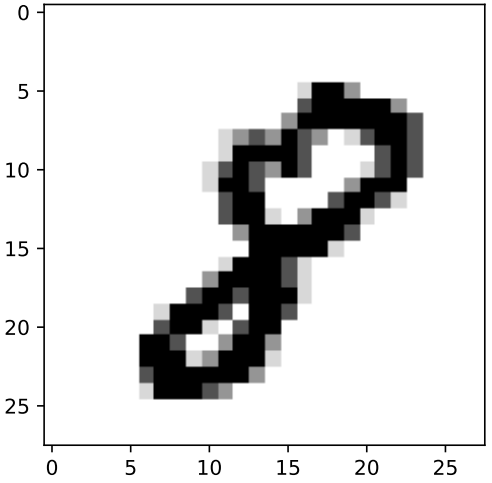
Another example: Recognizing written characters

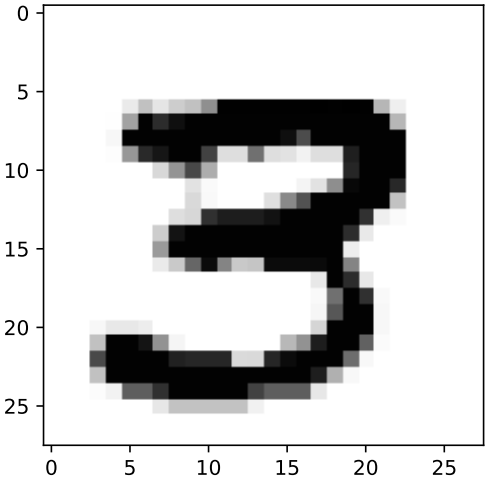
- We'll try to recognize a digit (0 – 9) from an image of a handwritten digit.
- MNIST dataset (http://yann.lecun.com/exdb/mnist/)
  - Popular dataset for testing out machine learning techniques
  - Training set is 60,000 images
  - Approximately 250 different writers
  - Test set is 10,000 images
  - Correct answer is known for both sets so we can test our performance
- Image details:
  - 28 × 28 pixels, grayscale (0 – 255 intensity)

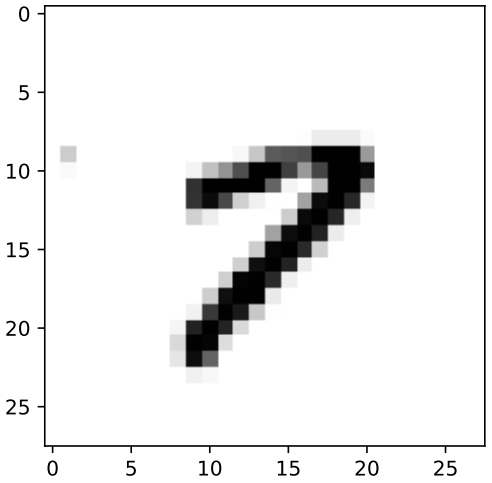
- We'll use a small subset

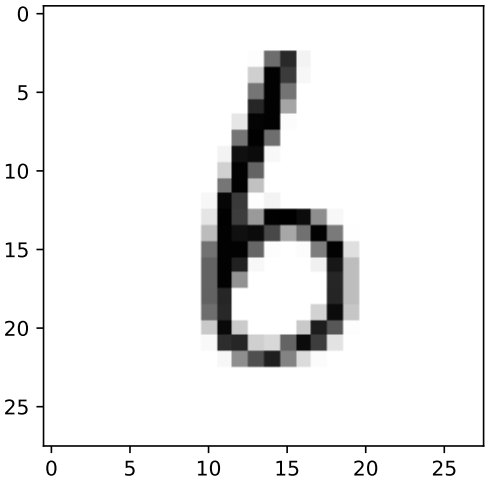
Another example: Recognizing written characters

- Input layer: 784 nodes (number of pixels)
- Output layer will be 10 nodes
  - Array with an entry for each possible digit
- Hidden layer size of 100
- 10 epochs
- We'll train on the training set, using 1000 images
- Rescale the input to be in [0.01, 1]
- We'll test on the test set of 1000 images



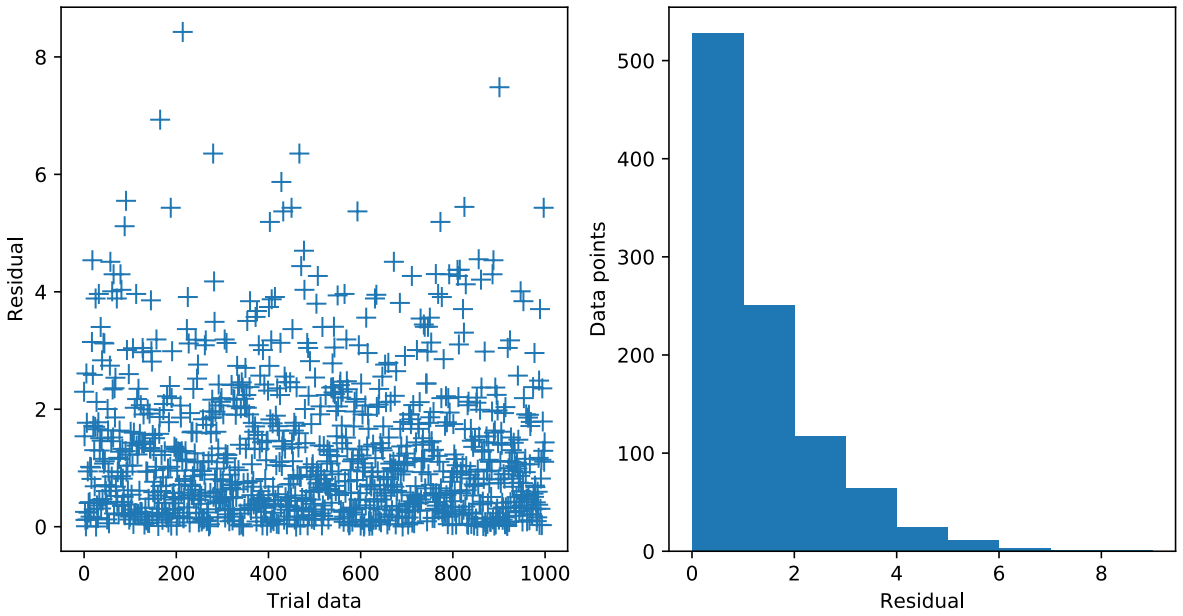




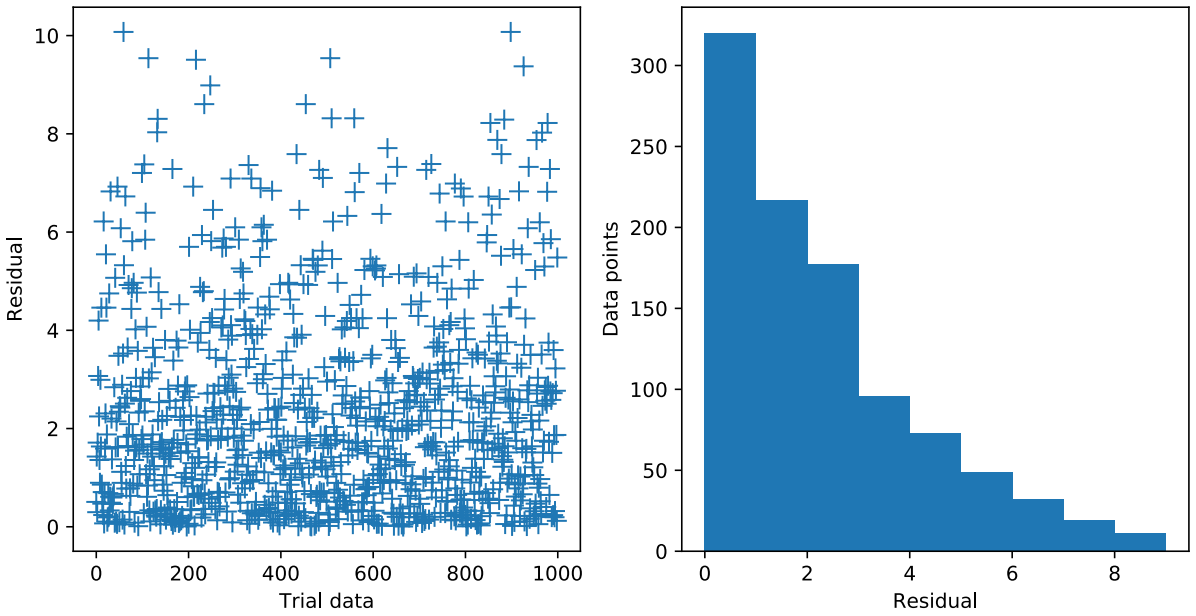


Another example: Recognizing written characters

Test on the training set



Test on new data



Today's lecture:
Neural networks and parallel computing

- Neural network examples
  - Interpreting a noisy signal
  - Identifying hand-written characters

- Parallel computing
  - OpenMP

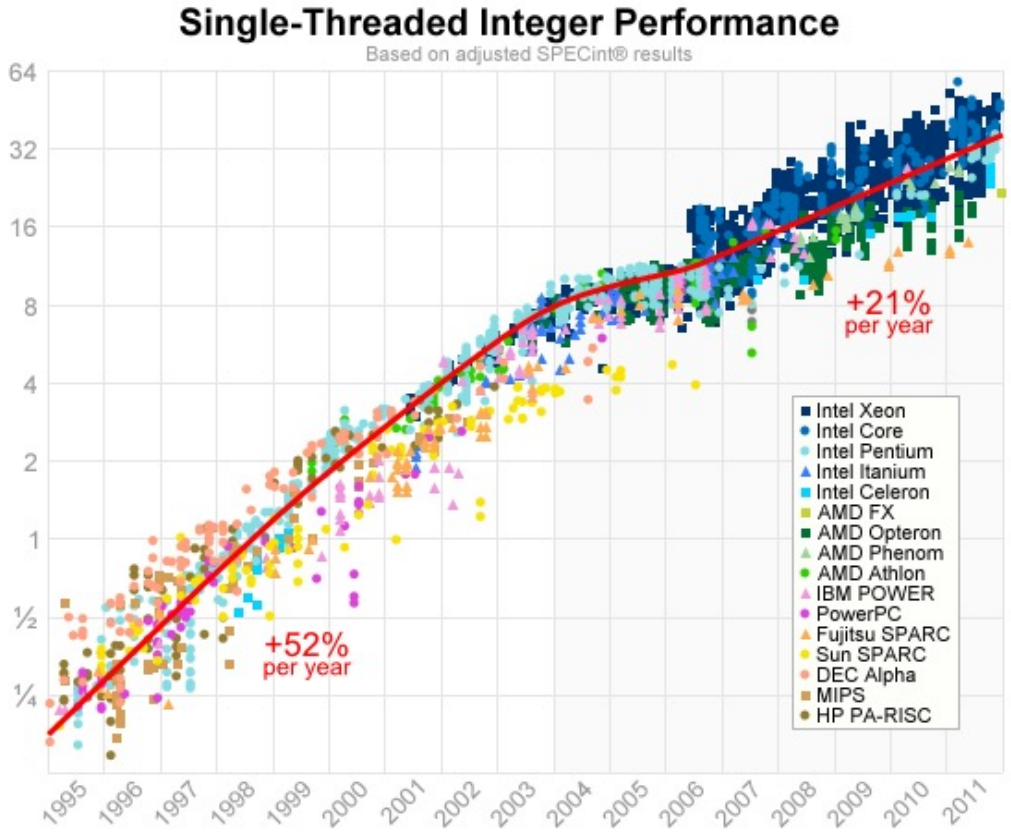
Why do we care about high performance computing?

- The more you can compute, the more interesting physics problems you can address

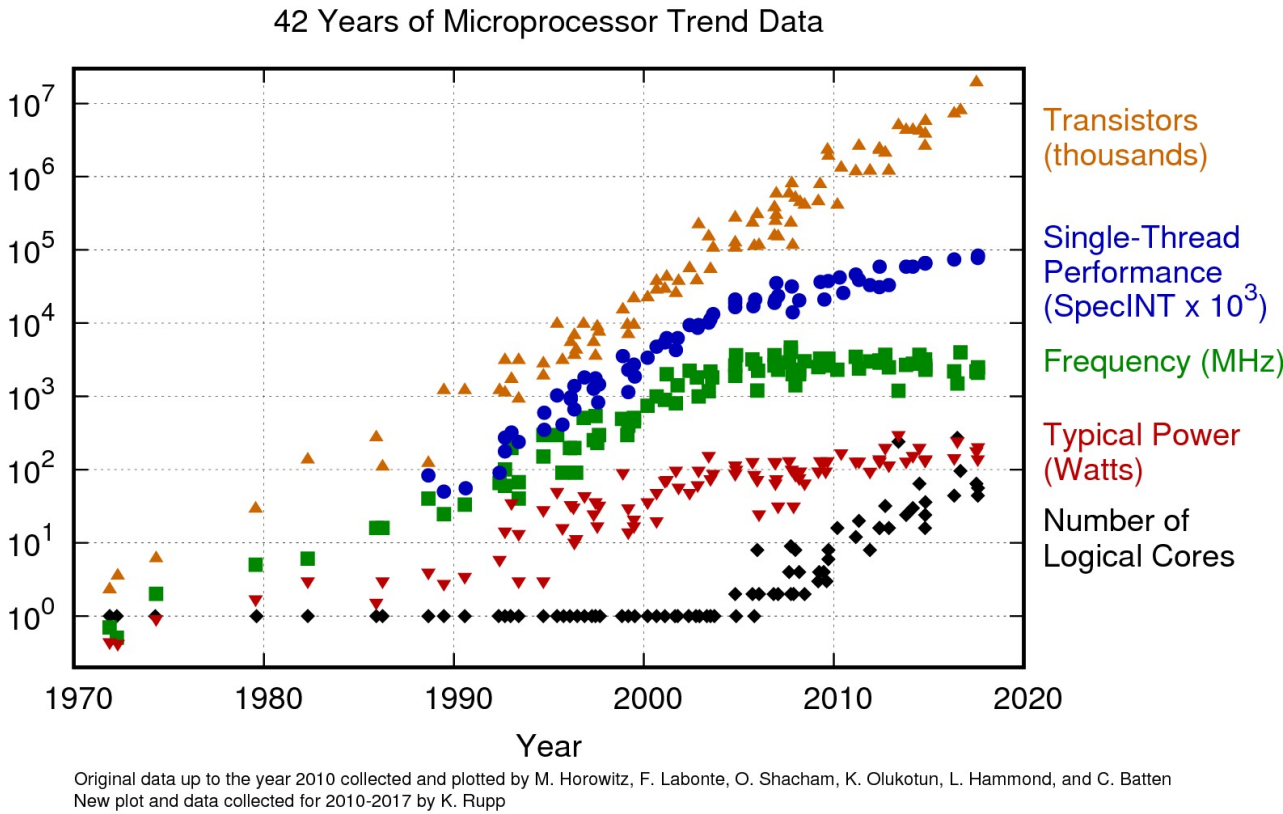
- Algorithmic advances allow us to do more with the same resources
  - This is what we have discussed over the course of the semester

- Another important aspect: Effectively taking advantage of more powerful computing hardware, e.g., supercomputers, GPUs

Performance of CPUs



https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/



How can we do better? Superconducting electronics

IEEE TRANSACTIONS ON APPLIED SUPERCONDUCTIVITY, VOL. 1, NO. 1, MARCH 1991

RSFQ Logic/Memory Family: A New Josephson-Junction Technology for Sub-Terahertz-Clock-Frequency Digital Systems

K. K. Likharev and V. K. Semenov

Invited Paper

- Record for clock speed: 100-300 GHz
  - Normal CPUs 3-5 GHz

How can we do better? Parallel computing

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371

How do we write parallel programs?

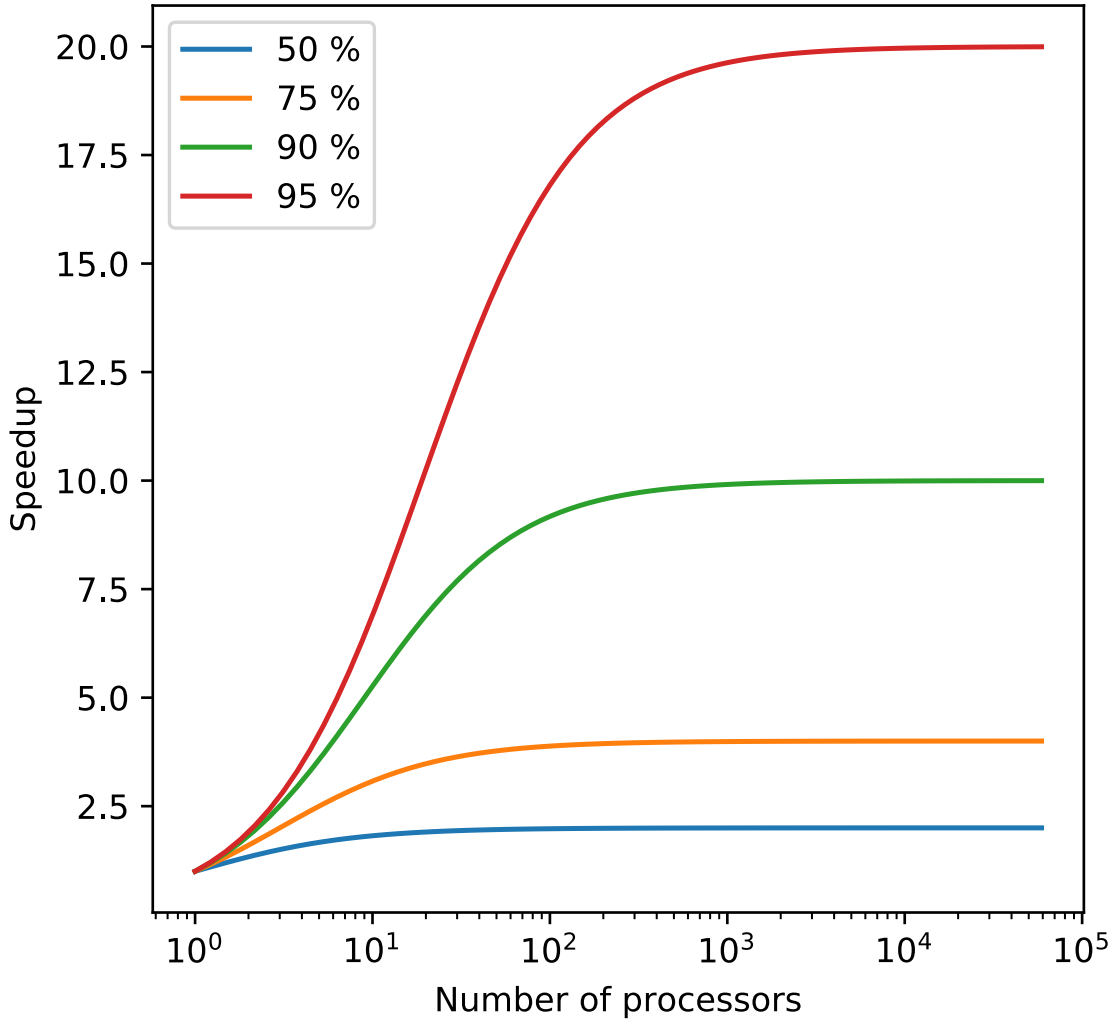
- Determine what parts of the program are limiting
  - Profilers can help
- Determine the type of hardware you will be running your program on
- Decide how to parallelize limiting parts
- Test the scaling of your program with number of parallel processes

How parallel does the program need to be?

- The speedup depends on what portion of the program is parallel
  - Often, certain algorithms or parts of algorithms are not easy to parallelize
  - Not very useful if parallel parts are not the limiting parts

- Amdahl's law: speedup attained from increasing the number of processors, *N*, given the fraction of the code that is parallel, *P* :

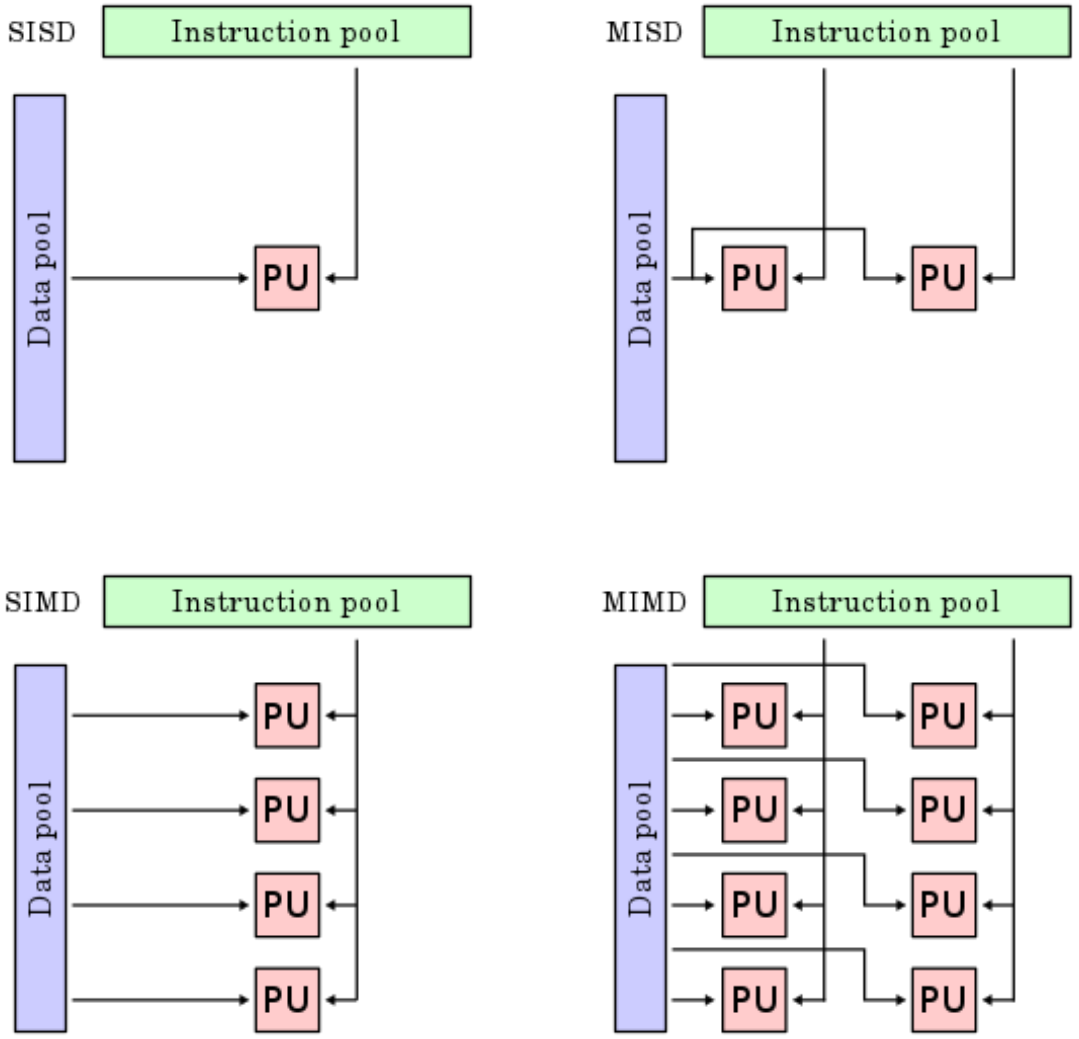
$$S = \frac{1}{(1-P) + (P/N)}$$



Types of parallelism: Flynn's taxonomy: (wikipedia)

- Classification based on how the CPU receives instructions and data

- Single instruction stream, single data stream (SISD)
  - No parallelism, instructions and data accessed serially
- Single instruction stream, multiple data streams (SIMD)
  - Single instruction applied to multiple data streams
  - Structure of old vector processors, modern GPUs
- Multiple instruction streams, single data stream (MISD)
  - Multiple instructions applied to one data stream
  - Not very common
- Multiple instruction streams, multiple data streams (MIMD)
  - Multiple processors simultaneously executing different instructions on different data
  - This is what we normally consider parallel computing

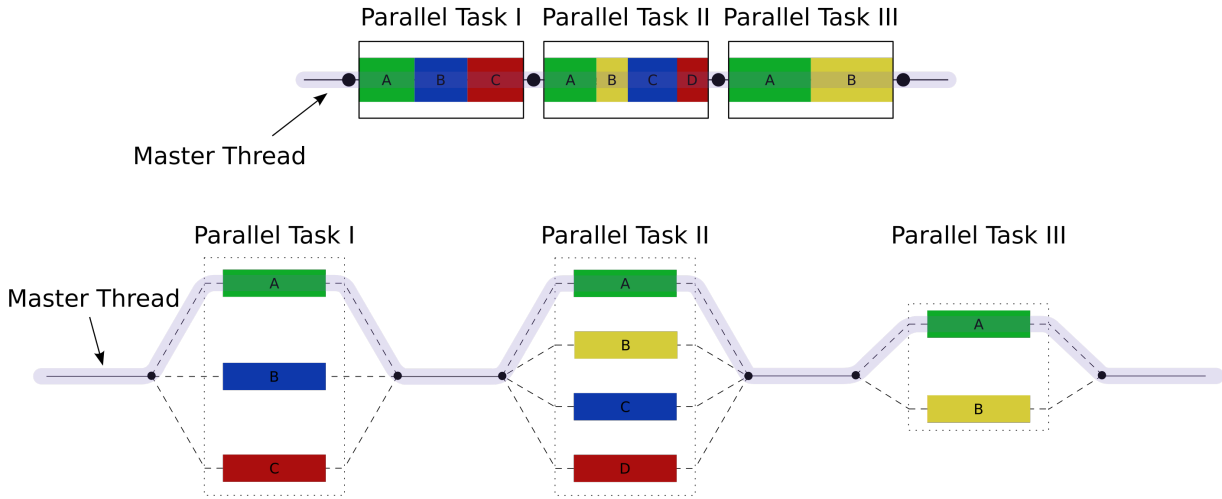


MIMD versus SPMD versus SIMD

- Normally, we do a specific type of MIMD:
  - Run a single program on multiple pieces of data (SPMD)
  - Different than SIMD since it does not require specialized hardware and does not require the processes to be in sync
- Two approaches to SPMD:
  - Distributed memory:
    - Machine has separate nodes which are essentially their own computer
    - Data (messages) are passed between nodes with interconnects, including commands to synchronize
    - Usually use MPI standard message passing interface (next time)
  - Shared memory:
    - Several CPUs have access to the same memory
    - Messages sent via the shared memory
    - Program starts executing on one processor, execution splits in a parallel region, then resynchronized
    - This is done via OpenMP (today)

OpenMP and multithreading

- Start process on a master thread
- Each time a piece of code is reached that is parallelized, the master thread forks subthreads to run the task
- After the parallel task is completed, threads join back to the master



Wikipedia

Examples of OpenMP

- To use OpenMP, we modify our program with directives or pragmas
  - Look like comments, but tells the compiler how it should process the input
- We also need to compile with flags for OpenMP
  - E.g., `gfortran –fopenmp`
- Finally, environment variables can also set properties
  - `OMP_NUM_THREADS` sets the number of threads
- Simplest example:

```
program hello

  !$OMP parallel
  print *, "Hello world"
  !$OMP end parallel

end program hello
```

OMP functions

- OMP provides some functions to get the number of threads, current thread, etc.:

```
program hello

  !integer, external :: omp_get_num_threads, omp_get_thread_num
  use omp_lib

  write(*,*) "outside of parallel region, num threads = ", omp_get_num_threads()

  !$OMP parallel
  write(*,*) "Hello world", omp_get_thread_num()
  !$OMP end parallel

end program hello
```

How can we get a speedup from threading?

- In the last example, we just did the same thing multiple times
- Simplest code to nontrivially parallelize are loops:

```
!$omp parallel private(i, j)
!$omp do
do j = 1,N
   do i = 1,N
      a(i,j)=0.
   end do
end do
!$omp end do
!$omp end parallel
```

Parallel calculations with OpenMP

- Can set the number of threads available with `OMP_NUM_THREADS`
  - Your code will most likely still run if you ask for more threads than cores, but you will not get a speedup!
- There is some overhead for spawning the threads
- As discussed before, best to parallelize limiting parts of the program first

Private versus shared data

- Inside the parallel tasks, can specify if each thread has it's own copy (private) or not (shared)
  - Shared is the default for variables declared outside of the parallel task
  - Be careful modifying shared data!

- Consider the two pieces of code:

```
k=0
!$omp parallel shared(k)
k=omp_get_thread_num()
!$omp end parallel
write(*,*) 'k=',k
```

```
k=0
!$omp parallel private(k)
k=omp_get_thread_num()
!$omp end parallel
write(*,*) 'k=',k
```

- The first one shares k over the nodes, the second makes private copies on all nodes

Combining data with reduction

- We can combine the results from all threads automatically via `reduction`:

```
k=0
!$omp parallel reduction(+:k)
k=omp_get_thread_num()
!$omp end parallel
write(*,*) 'k=',k
```

- Will add the results of each thread

Controlling ordering an synchronization of threads

- As we saw before the threads will proceed in some unknown order

- We may want to have more control than this

- Some useful directives:
  - `!$omp critical`: Make sure only one thread at a time can changes things in critical sections
    - Creates a “thread safe” section of code
    - Important when, e.g., incrementing shared variables
  - `!$omp barrier`: Stops processes from proceeding until all processes catch up

Some comments on OpenMP

- We just covered some basics, OpenMP has quite a bit of functionality
- Have to make sure that your program is threadsafe:
  - Don’t inadvertently overwrite data
  - Don’t have issues because of the order or synchronization of processes
  - Always good unit test in serial and parallel to make sure the results are the same
- Fairly straightforward to parallelize your code piece by piece
  - Because directives look like comments, it is easy to run with and without parallelization (just change compiler flags)
- Works well for fortran, C, C++
- Will NOT work for pure python
  - “Global interpreter lock” that means only one thread can talk to the interpreter at any one time
  - Can use OpenMP with, e.g., Cython

After class tasks

- First draft of first two sections of writeup due today
- Tuesday Nov 23 will be our last normal class
- Presentations Tues Nov. 30 and Thurs. Dec 2
  - Chosen at random, so please be prepared on Tuesday!
- Readings:
  - *Computational Methods for Physics*, Joel Franklin, Chapter 14
  - *Make Your Own Neural Network*, Tariq Rashid
  - http://playground.tensorflow.org