Exercise 6

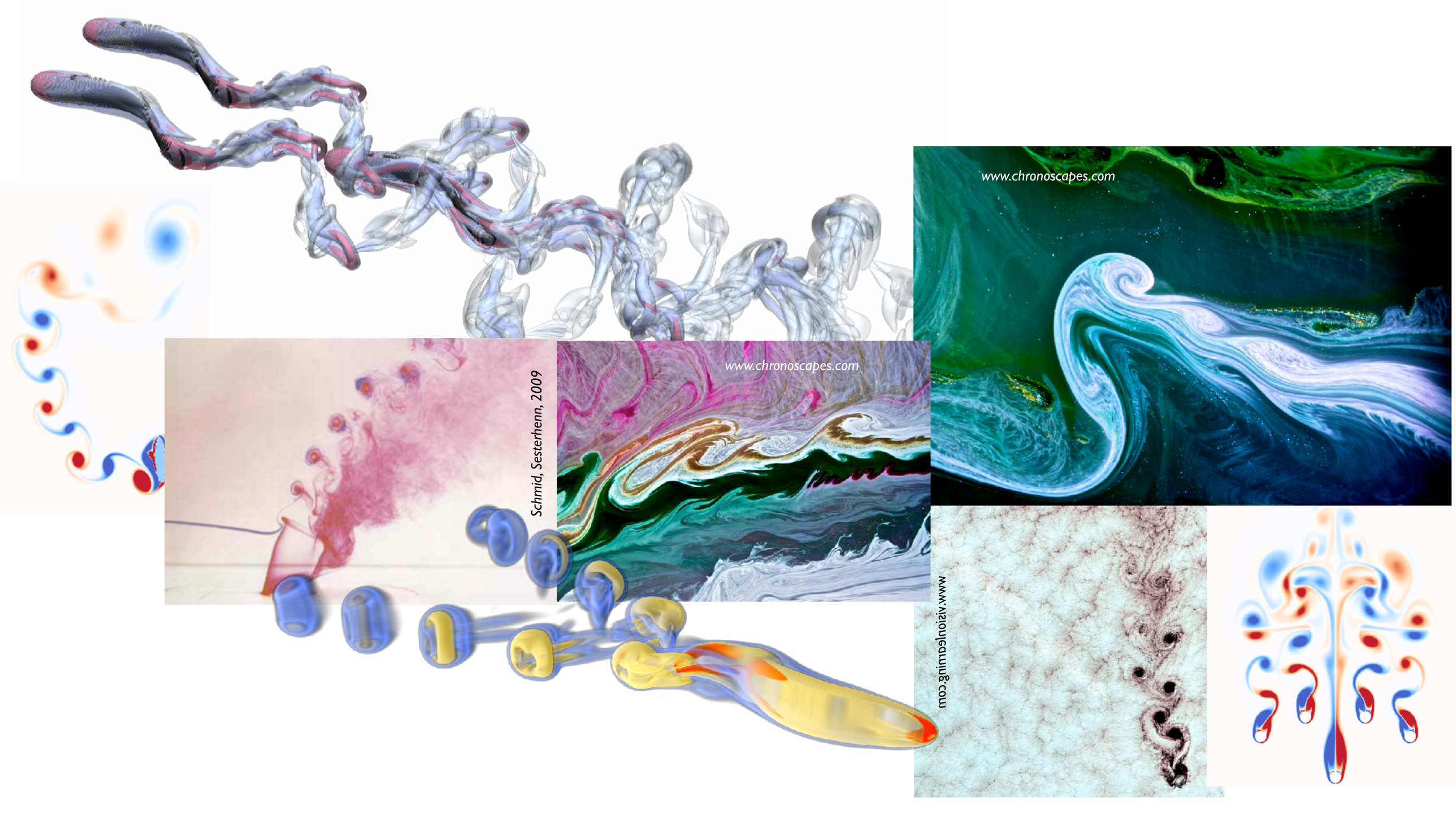
High Performance Computing for Science and Engineering

Guido Novati

November 10, 2017

CSElab
Computational Science & Engineering Laboratory
http://www.cse-lab.ethz.ch

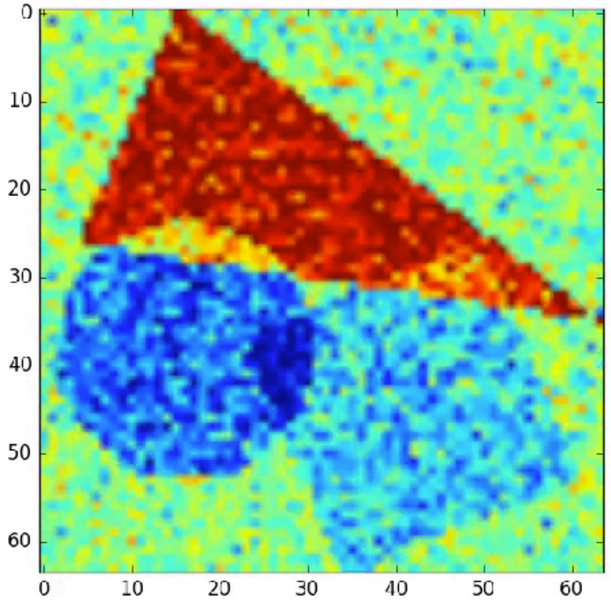
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich


www.chronoscapes.com
Schmid, Sesterhenn, 2009
www.chronoscapes.com
www.visionlearning.com

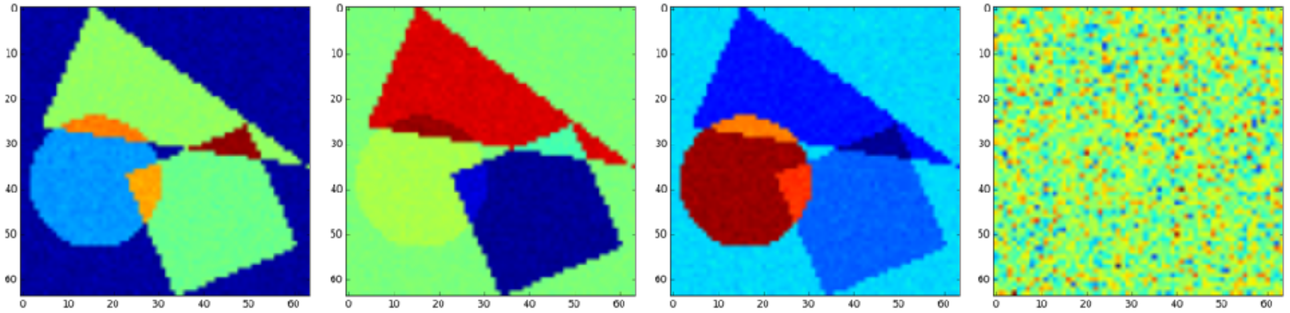
Principal Component Analysis

Motivation:

- Extraction of *modes* from experimental or simulation data to describe underlying mechanisms ("how many important things are happening in the dataset?")
- Projection of high dimensional problem onto a lower dimensional system (**model reduction**)



PCA modes:



1 2 3 all others

PCA with Auto-associative Neural Network

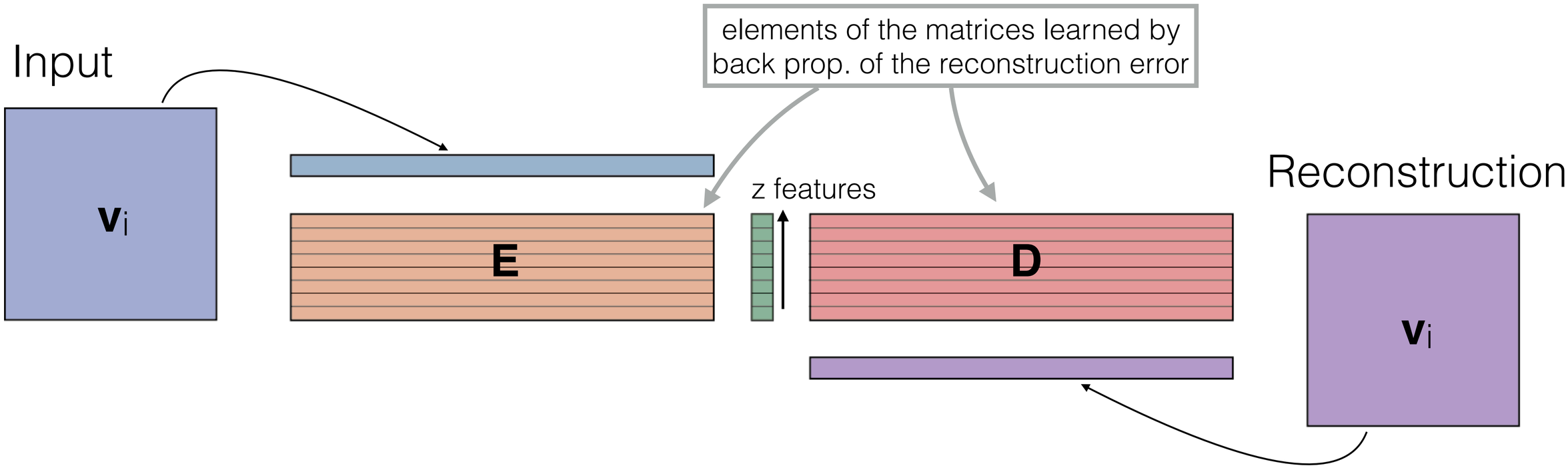
Start simple:

- linear fully connected neural network
- train network to reproduce input with mean squared error loss function:

$$\mathcal{L} = \sum_i (\mathbf{v}_i - \tilde{\mathbf{v}}_i)^2$$

$$\mathbf{DE}\mathbf{v}_i = \tilde{\mathbf{v}}_i$$

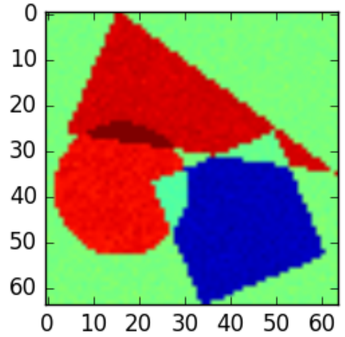
M x z decoder matrix
z x M encoder matrix
M x 1 input sequence



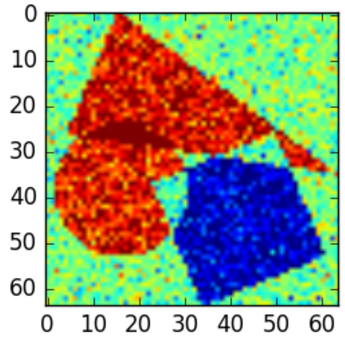
Example of PCA with Auto-associative Net

Sample reconstruction:

output

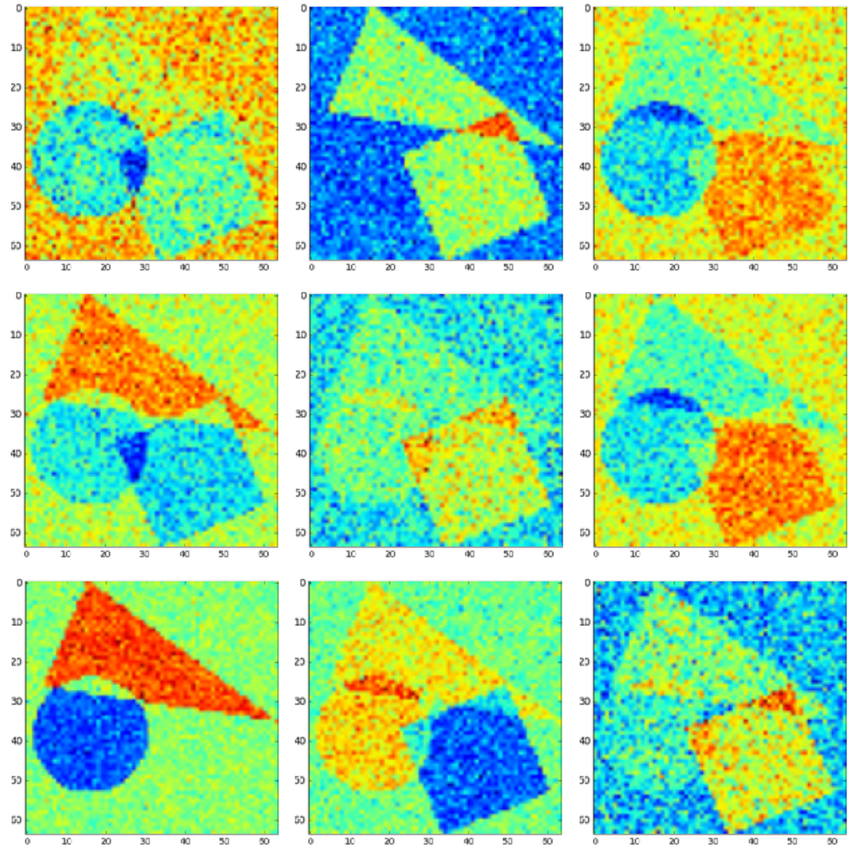


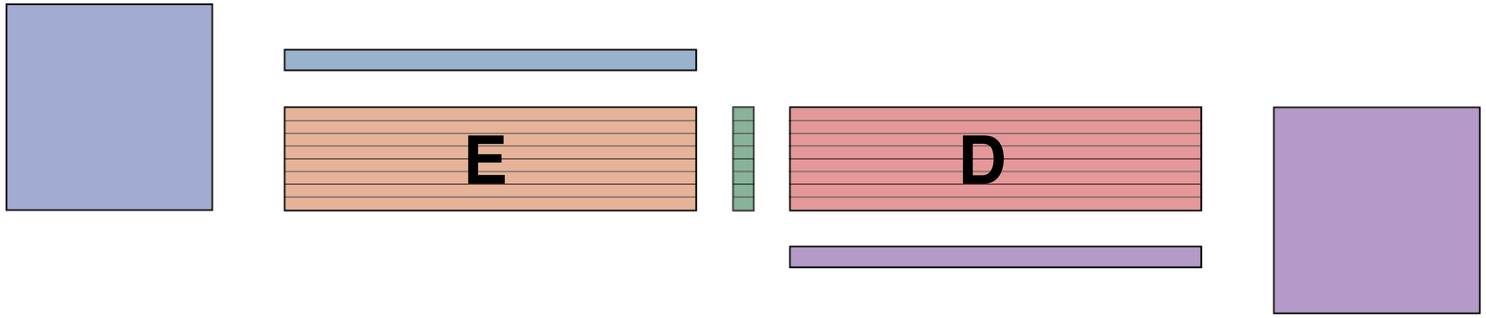
input



- Network learns to reproduce input filtering the added noise

- Chosen compression layer of size 9
- Once trained I can visualize output if I turn on only one unit in compression layer
- Each resulting output is a linear combination of PCA modes
- In this case, optimal compression later size would be 3





Exercise Introduction

- Ex 6 and 7 are a toy example of a "deep-learning" library
- We want to write a code that evaluates and optimizes the model given a cost function

- Have a look at deep learning libraries such as caffe: https://github.com/BVLC/caffe
- Core of the library is split in two categories:
  - Optimizers
  - Layers
    - For each layer type, two operations are defined:
      - Forward
      - Backward

Implementing Neural Networks

- NN is composition of operations:

$$\mathbf{x} \longrightarrow \mathbf{h}^{(1)} \longrightarrow \mathbf{h}^{(2)} \longrightarrow \cdots \longrightarrow \mathbf{h}^{(K)}$$

- Arrows denote operations that are differentiable:
  - Linear algebra

$$\mathbf{h}^{(k+1)} = \mathbf{h}^{(k)} \; W^{(k+1)}$$

  - Non linear operations:

$$\mathbf{h}^{(k+2)} = f\left(\mathbf{h}^{(k+1)}\right)$$

- Cost function, eg Eulerian Squared Distance:

$$\mathcal{L} = \frac{1}{2}\left(\mathbf{y}^* - \mathbf{h}^{(K)}\right) \cdot \left(\mathbf{y}^* - \mathbf{h}^{(K)}\right)$$

Forward-propagation

- Computing the output of a network: "prediction" or "forward-propagation"

$$\mathbf{h}^{(k+1)} = \mathbf{h}^{(k)}\ W^{(k+1)}$$

$$\mathbf{h}^{(k)} = \{h_1^{(k)}, h_2^{(k)}, \ldots, h_{n_k}^{(k)}\}$$

- Layer **k** has $\mathbf{n_{k+1}}$ outputs and received $\mathbf{n_k}$ inputs
- Parameters of the layer (to be optimized) constitute a matrix of $\mathbf{n_k}$ by $\mathbf{n_{k+1}}$ elements.

$$W^{(k+1)} = \begin{bmatrix} w_{1,\ 1}^{(k+1)} & w_{1,\ 2}^{(k+1)} & \cdots & w_{1,\ n_{k+1}}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_k,\ 1}^{(k+1)} & w_{n_{k+1},\ 2}^{(k+1)} & \cdots & w_{n_{k+1},\ n_{k+1}}^{(k+1)} \end{bmatrix}$$

- NOTES:
  - Superscript denotes layer ID, subscripts iterate over neurons.
  - For W, first index is output, second index is input neuron.

Back-propagation (I)

- Assume the Squared Distance Loss function of the auto-associative NN:

$$\mathcal{L} = \frac{1}{2}\left(\mathbf{y}^* - \mathbf{h}^{(K)}\right) \cdot \left(\mathbf{y}^* - \mathbf{h}^{(K)}\right)$$

- The gradient of the loss with respect to the NN's output:

$$\boldsymbol{\delta}^{(K)} = \frac{d\mathcal{L}}{d\mathbf{h}^{(K)}} = (\mathbf{h}^{(K)} - \mathbf{y}^*)$$

- This is a vector of the same size as output layer!
  - In general we will assume that Loss function is a scalar function
  - Gradient of a scalar with respect to a N-dimensional array is an N-dimensional array
  - Remembering this will be very helpful!

Back-propagation (II)

- Back-propagation denotes computing the gradients of:
  - Error with respect of layer: $\boldsymbol{\delta}^{(k)} = \left.\frac{d\mathcal{L}}{d\mathbf{h}^{(k)}}\right|_{\mathbf{x},W}$ (this is a vector)
  - Error with respect of weights $\left.\frac{d\mathcal{L}}{dW^{(k)}}\right|_{\mathbf{x},W}$ (this is a matrix)
  - Given a certain input and the current parameters.

- Back-propagation does not change the parameters, for that we will need to pick some optimization scheme.
- With back-propagation we aim to compute all derivatives, with similar computational complexity as for forward-propagation
- We proceed backwards…

$$\mathbf{x} \leftarrow \mathbf{h}^{(1)} \leftarrow \mathbf{h}^{(2)} \leftarrow \cdots \leftarrow \mathbf{h}^{(K)}$$

Back-propagation (III)

- Compute the derivative of the cost function with respect to network output

$$\boldsymbol{\delta}^{(K)} = \frac{d\mathcal{L}}{d\mathbf{h}^{(K)}} = (\mathbf{h}^{(K)} - \mathbf{y}^*)$$

- By chain differentiation, the derivative of the Loss w.r.t. the output of any layer is

$$\frac{d\mathcal{L}}{d\mathbf{h}^{(j)}} = \frac{d\mathcal{L}}{d\mathbf{h}^{(K)}} \frac{d\mathbf{h}^{(K)}}{d\mathbf{h}^{(K-1)}} \cdots \frac{d\mathbf{h}^{(j+1)}}{d\mathbf{h}^{(j)}}$$

- We don't need to remember the gradients of previous layers!

- Gradient of L w.r..t to layer's inputs and parameters does not depend on previous layers if we know the grad of L w.r..t to layer outputs!

$$\frac{d\mathcal{L}}{d\mathbf{h}^{(j)}} = \frac{d\mathcal{L}}{d\mathbf{h}^{(j+1)}} \frac{d\mathbf{h}^{(j+1)}}{d\mathbf{h}^{(j)}} \qquad \frac{d\mathcal{L}}{dW^{(j+1)}} = \frac{d\mathcal{L}}{d\mathbf{h}^{(j+1)}} \frac{d\mathbf{h}^{(j+1)}}{dW^{(j+1)}}$$

- Also second eq. is chain differentiation: the loss depends on $W^{(j+1)}$ (the parameters on layer j+1) only through $\mathbf{h}^{j+1}$ (the output of layer j+1).

Optimization algorithm (I)

- Once we performed back-propagation, we have $\left.\frac{d\mathcal{L}}{dW^{(k)}}\right|_{\mathbf{x},W}$
- Weights optimised by stochastic gradient descent: $W \leftarrow W - \frac{\beta}{B}\sum_{i}^{B}\left.\frac{d\mathcal{L}}{dW}\right|_{\mathbf{x},W}$
- We want to compute the loss function for a mini-batch of B samples at the time and compute the average gradient
- Therefore NN will perform B forward-propagation at the same time
- Back-propagation will compute $G = \sum_{i}^{B}\left.\frac{d\mathcal{L}}{dW}\right|_{\mathbf{x}_i,W}$

  G = unnormalised gradient estimate from the minibatch (i.e. factor 1/B is missing)

Back-propagation of Linear layer

- We consider, like most libraries, mini-batches of evaluations

$$H^{(k)} = \begin{bmatrix} \mathbf{h}_1^{(k)} \\ \vdots \\ \mathbf{h}_B^{(k)} \end{bmatrix}$$

- Forward operation: $H^{(k+1)} = H^{(k)}\ W^{(k+1)} + \mathbf{b}^{(k+1)}$
  - **GEMM**!! [**B** by $\mathbf{n_{k+1}}$] = [**B** by $\mathbf{n_k}$] × [$\mathbf{n_k}$ by $\mathbf{n_{k+1}}$], plus bias
  - Same bias of size [**1** by $\mathbf{n_{k+1}}$] is added to each column
- Backward operation is:

$$D^{(k)} = \begin{bmatrix} \boldsymbol{\delta}_1^{(k)} \\ \vdots \\ \boldsymbol{\delta}_B^{(k)} \end{bmatrix}$$

  - Gradient of L w.r.t. to input: [**B** × $\mathbf{n_k}$] = [**B** by $\mathbf{n_{k+1}}$] × [$\mathbf{n_{k+1}}$ by $\mathbf{n_k}$]

$$D^{(k)} = D^{(k+1)} \left(W^{(k+1)} \right)^T$$

  - Gradient of L w.r.t. to Parameters: [$\mathbf{n_k}$ by $\mathbf{n_{k+1}}$] = [$\mathbf{n_k}$ by **B**] × [**B** by $\mathbf{n_{k+1}}$]

$$G_{W^{(k+1)}} = (H^{(k)})^T D^{(k+1)}$$

Unnorm. grad of weights

$$G_{\mathbf{b}^{(k+1)}} = \sum_{b=1}^{B} \boldsymbol{\delta}_b^{(k+1)}$$

Unnorm. grad of bias

Back-propagation of Non-linearity

- Generally we use two types of non-linearities:
- Element-wise non-linearities (e.g. hyperbolic tangent):
  - The same single-input non-linear function is applied to each input:

$$\mathbf{h}^{(k+2)} = f(\mathbf{h}^{(k+1)})$$

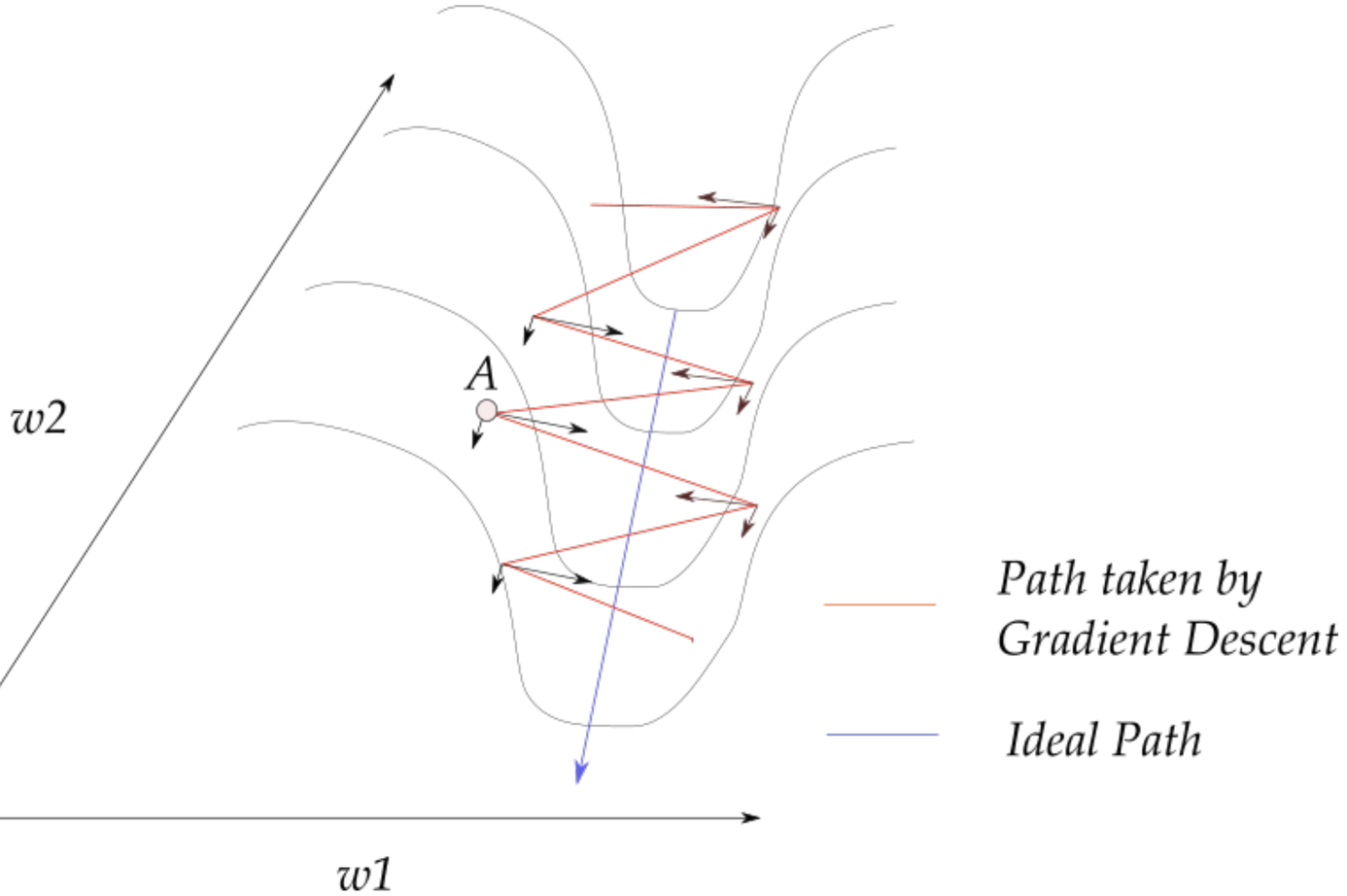
  - Derivative is also done element-wise:

$$\boldsymbol{\delta}^{(k+1)} = \boldsymbol{\delta}^{(k+2)} \circ f'(\mathbf{h}^{(k+1)})$$

    - We are denoting the Hadamard (element-wise) product
      [**I** by $\mathbf{n_{k+1}}$] = [**I** by $\mathbf{n_{k+1}}$] o [**I** by $\mathbf{n_{k+1}}$]
- Vector to scalar operations (like the loss function) are similar, but instead of element-wise product you compute the Jacobian
- With mini-batches this will result in an additional loop over B

Optimization algorithm (II)

- We do not really want to use vanilla Stochastic Gradient Descent (SGD)
- We will implement second-easiest algorithm (same rules for both W and b)
- Momentum-SGD tries to keep track of previous update directions



- 1) Backprop to compute $G = \sum_i^B \frac{d\mathcal{L}}{dW}\Big|_{\mathbf{x}_i, W}$
- 2) Compute average direction of update (referred to as "momentum") $M \leftarrow \beta M - \frac{\eta}{B} G$
- Here learning rate η (usually less than 0.001) and momentum factor β (0.9)
- 3) Update weights: $W \leftarrow W + M$

Network building blocks (I)

- First, we introduce two memory allocations:
  - `Params` defined in `network/Params.h` .
    - Each layer can allocate **Weights** and **Biases**
    - For linear layer `nWeights` = `nInputs` x `nOutputs`, `nBiases` = `nOutputs`
    - Non-linear layer (we will implement `Tanh`) does not allocate `Params`
  - `Activation` defined in `network/Activation.h`
    - `layerSize` is the number of outputs of the corresponding layer
    - **Output** and **Grad** of loss w.r.t output
    - both arrays have the same size, as we discussed in "Back-propagation (I)"

```
struct Params
{
  const int nWeights, nBiases;
  Real* const weights; // size is nWeights
  Real* const biases;  // size is nBiases
```

```
struct Activation
{
  const int batchSize, layersSize;
  //matrix of size batchSize * layersSize with layer outputs:
  Real* const output;
  //matrix of same size containing:
  Real* const dError_dOutput;
```

Network building blocks (II)

- Layer has a size (number of outputs) and an ID (index from 0 to K)

```
struct Layer
{
  const int size, ID;
```

- E.g. Linear layer
  - `size == nOutputs`
  - Use `ID` to access `Activation`
- Exercise:

Implement forward and backward with gemm (and for loops for bias)

```
template<int nOutputs, int nInputs>
struct LinearLayer: public Layer
{
  LinearLayer(const int _ID) : Layer(nOutputs, _ID) {
    printf("(%d) Linear Layer of Input:%d Output:%d\n", ID, nInputs, nOutputs);
    assert(nOutputs>0 && nInputs>0);
  }

  void forward(const std::vector<Activation*>& act,
               const std::vector<Params*>& param) const override {
    const int batchSize = act[ID]->batchSize;
    //array of outputs from previous layer
    const Real*const inputs = act[ID-1]->output; //size is batchSize * nInputs

    //weight matrix and bias vector:
    const Real*const weight = param[ID]->weights; //size is nInputs * nOutputs
    const Real*const bias   = param[ID]->biases; //size is nOutputs

    //return matrix that contains layer's output
    Real*const output = act[ID]->output; //size is batchSize * nOutputs
```

Network building blocks (III)

- Network is a list of Layers, memory workspace to compute, parameters and gradients.
- **forward** takes a vector of input vectors (mini-batch) and returns a mini-batch of outputs
- (optional we can give input to intermediate layer, eg to see PCA)
- **bckward** takes a vector of gradient of error w.r.t. to outputs and perform back-prop to get all gradients
- does not change weights yet…

```
struct Network
{
  std::mt19937 gen;
  // Vector of layers, each defines a forward and bckward operation:
  std::vector<Layer*>  layers;
  // Vector of parameters of each layer (two vectors must have the same size)
  // Each Params contains the matrices of parameters needed by the corresp layer
  std::vector<Params*> params;
  // Vector of grads for each parameter. By definition they have the same size
  std::vector<Params*>  grads;
  // Memory space where each layer can compute its output and gradient:
  std::vector<Activation*> workspace;
```

```
std::vector<std::vector<Real>> forward(
      // one vector of input for each element in the mini-batch:
      const std::vector<std::vector<Real>> I,
      // layer ID at which to start forward operation:
      const size_t layerStart = 0 // (zero means compute from input to output)
  )
```

```
void bckward(
  // vector of size of mini-batch of gradients of error wrt to network output
  const std::vector<std::vector<Real>> E,
  // layer ID at which forward operation was started:
  const size_t layerStart=0 // (zero means compute from input to output)
) const
```

Network building blocks (IV)

- Optimizer is some algorithm that loops over all Params and changes weights and biases

```
template<typename Algorithm>
struct Optimizer
{
  Network& NET;
  const Real eta, beta_1, beta_2, lambda;
  // grab the reference to network weights and parameters
  std::vector<Params*> & parms = NET.params;
  std::vector<Params*> & grads = NET.grads;
```

```
for (size_t j = 0; j < parms.size(); j++)
{
  if (parms[j] == nullptr) continue; //layer does not have parameters

  if (parms[j]->nWeights > 0)
  {
    algo.step(parms[j]->nWeights,
              parms[j]->weights, grads[j]->weights,
              momentum_1st[j]->weights, momentum_2nd[j]->weights);
    grads[j]->clearWeight(); // reset for next step
  }

  if (parms[j]->nBiases > 0)
  {
    algo.step(parms[j]->nBiases,
              parms[j]->biases, grads[j]->biases,
              momentum_1st[j]->biases, momentum_2nd[j]->biases);
    grads[j]->clearBias(); // reset for next step
  }
}
```

- Exercise: implement Momentum SGD

```
// perform gradient update for a parameter array:
inline void step (
      const int size,     // parameter array's size
      Real* const param,  // parameter array
      Real* const grad,   // parameter array gradient
      Real* const mom1st, // parameter array gradient 1st moment
      Real* const mom2nd  // parameter array gradient 2nd moment (unused)
    ) const
{
```

Code that uses the network

- Use `main_testGrad.cpp` for testing. It checks that:

$$\frac{dh^{(K)}}{dW} = \frac{h^{(K)}\big|_{\mathbf{x},W+\epsilon} - h^{(K)}\big|_{\mathbf{x},W-\epsilon}}{2\epsilon}$$

  - ie. it computes the gradient through finite differences and compares with back-prop.
  - Well documented: read it to see why back-prop is so popular
- Exercise mostly focuses on `main_backprop.cpp`:
  - Loads MNIST dataset (get by using `setup_mnist.sh`)
  - Trains Auto-associative Net to reconstruct MNIST
  - Writes to file the network outputs by activating only one unit in the compression layer at the time (view with `visualize_components.py`)