Lecture 9: Deep Learning (Part 1)

Recap: Data representations

- Data heterogeneity
- Data size
  - E.g., Vectorization of images

Size: 536x356x3

$$\mathbf{x}_i \in \mathbb{R}^{536 \cdot 356 \cdot 3} = \mathbb{R}^{572448}$$

Size: 500x523x3

$$\mathbf{x}_j \in \mathbb{R}^{500 \cdot 523 \cdot 3} = \mathbb{R}^{784500}$$

- Data noisiness
  - Not every piece of data encodes valuable information

Recap: Attributes

- E.g., UCI Abalone dataset
  - Predict the age of abalone from measurements

Name	Data Type	Measurement	Description
Sex	Categorical		M, F, and I (infant)
Length	Numeric (continuous)	mm	Longest shell measurement
Diameter	Numeric (continuous)	mm	perpendicular to length
Height	Numeric (continuous)	mm	with meat in shell
WT_Whole	Numeric (continuous)	grams	whole abalone
WT_Shuck	Numeric (continuous)	grams	weight of meat
WT_Vscra	Numeric (continuous)	grams	gut weight (after bleeding)
WT_Shell	Numeric (continuous)	grams	after being dried
Rings	Numeric (integer)		+1.5 gives the age in years

- With attributes, all samples have the same length

Recap: Bag of words

- Idea: For each sample, count the number of times each word in the dictionary appears
- Example (from Wikipedia):
  - 2 samples

```
(1) John likes to watch movies. Mary likes movies too.
```

```
(2) John also likes to watch football games.
```

  - Corresponding counts:

```
BoW1 = {"John":1,"likes":2,"to":1,"watch":1,"movies":2,"Mary":1,"too":1};
BoW2 = {"John":1,"also":1,"likes":1,"to":1,"watch":1,"football":1,"games":1};
```

Recap: Bag of words

- With D words in the dictionary, each sample can be represented as a vector in $\mathbb{R}^D$
- Example:
  - With dictionary ($D = 9$)

```
"John","likes","watch","movies","Mary","too","also","football","games"
```

  - the BoWs

```
BoW1 = {"John":1,"likes":2,"to":1,"watch":1,"movies":2,"Mary":1,"too":1};
BoW2 = {"John":1,"also":1,"likes":1,"to":1,"watch":1,"football":1,"games":1};
```

  - become

$$\tilde{\mathbf{x}}_1 = [1 \quad 2 \quad 1 \quad 2 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0]$$
$$\tilde{\mathbf{x}}_2 = [1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1]$$

Recap: Histograms

- After normalization, the final BoW representation

$$\mathbf{x}_1 = [1/8 \quad 1/4 \quad 1/8 \quad 1/4 \quad 1/8 \quad 1/8 \quad 0 \quad 0 \quad 0]$$

- is called a **histogram**

- It is such that

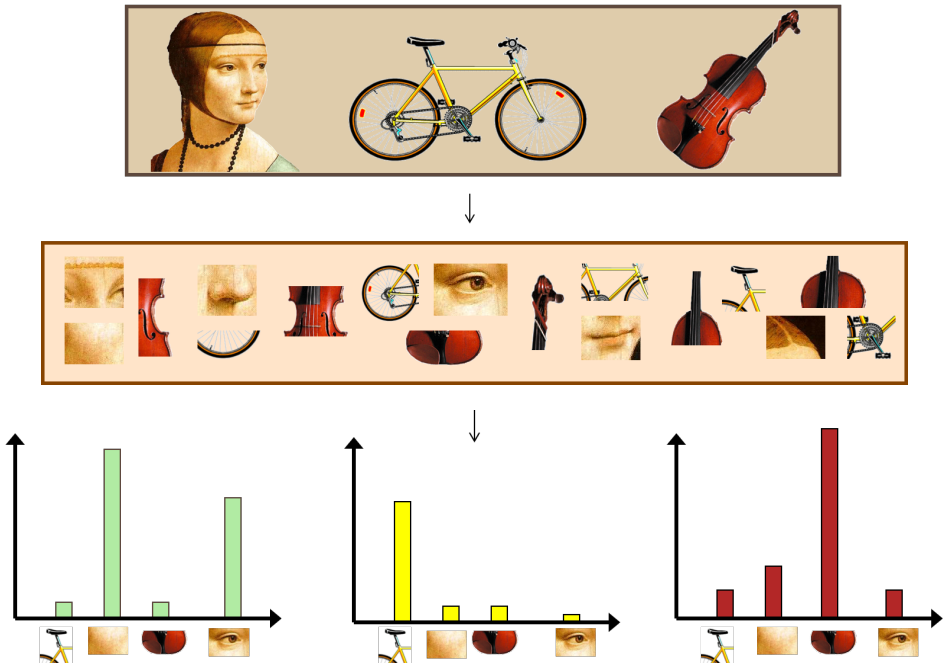
$$x_1^{(d)} \in [0,1]$$

$$\sum_{d=1}^{D} x_1^{(d)} = 1$$

- In this way, each sample is represented with a vector in $\mathbb{R}^D$

Recap: Bag of visual words

- For each patch in an image
  - find the nearest visual word
  - add one to the corresponding element in a BoW vector
- Normalize the BoW vectors to obtain histograms



Recap: Discussion

- We have seen that text and images can be represented as Bags of Words. What other type of data can you represent in this way? How would you do it?
  - For example, sound/speech. One can build a dictionary by breaking the training samples into short sound snippets and clustering them based on a notion of similarity. Then, each training sample can be expressed as a bag of audio word by associating each snippet to one cluster.

Recap: Data pre-processing

- Even with a good representation, the data might benefit from being pre-processed
- This is because, e.g.,
  - Some attribute values are missing
  - Some attribute values are noisy
  - The feature dimensions are of incommensurate magnitudes

Recap: Missing data

- Some attribute values were not recorded
  - E.g., the Air Quality UCI dataset contains missing values, which are tagged with a -200 value
- Potential solutions
  - Ignore the corresponding sample
  - Fill in the missing values with a global constant
  - Fill in the missing values with the corresponding attribute average over the dataset
  - Fill in the missing values with the corresponding attribute average over the samples belonging to the same class

Recap: Noisy data

- Noise:
  - Random measurement error
- Incorrect values
  - Faulty data collection
  - Data entry problems
  - Data transmission problems
- Other problems
  - Duplicate entries
  - Inconsistent data
- Solution: Clean the data

Recap: Cleaning noisy data

- Binning
  - Partition the data into bins and smooth it by replacing the data values with the mean or median of their respective bin
- Clustering
  - Detect outliers as the clusters with too few elements and remove these samples
- What other solution: Leverage human knowledge
  - Some values are not possible for certain attributes. E.g.,
    - Blood pressure cannot be 0
    - Salary cannot be negative

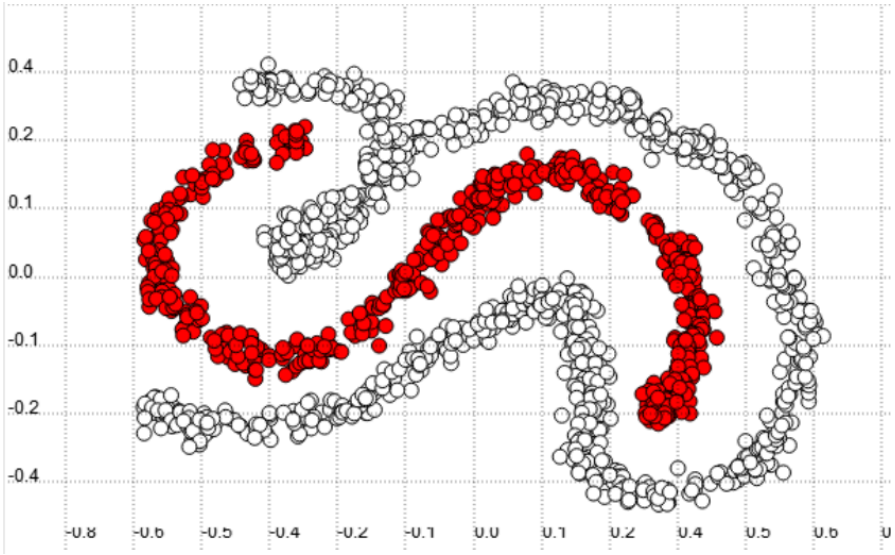
Recap: Data normalization

- Goal: Scale each individual attribute/feature dimension to fall within a specified range

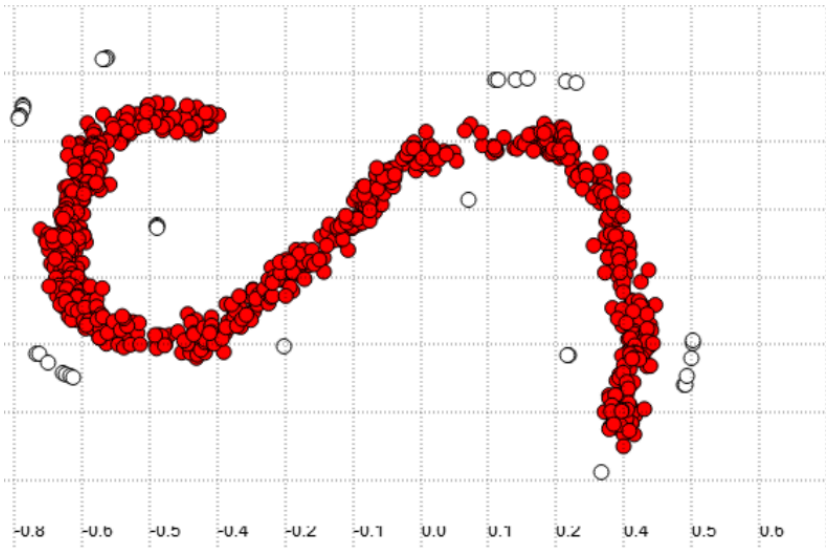
- Min-max normalization
- z-score normalization
- Max normalization
- Normalization by decimal scaling

Recap: Imbalanced vs balanced data

- Between-class imbalance



Balanced data



Imbalanced data

- Two solutions:
  - Act on the data
  - Act on the empirical risk

Goals of today's lecture

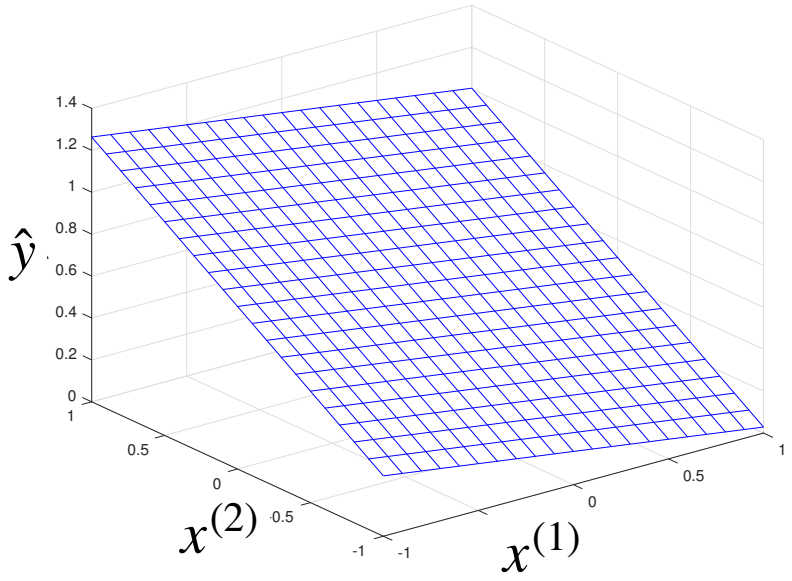
- Introduce the multilayer perceptron (MLP) model

- Explain the training algorithm for an MLP

Lessons learned until now

- Linear models:

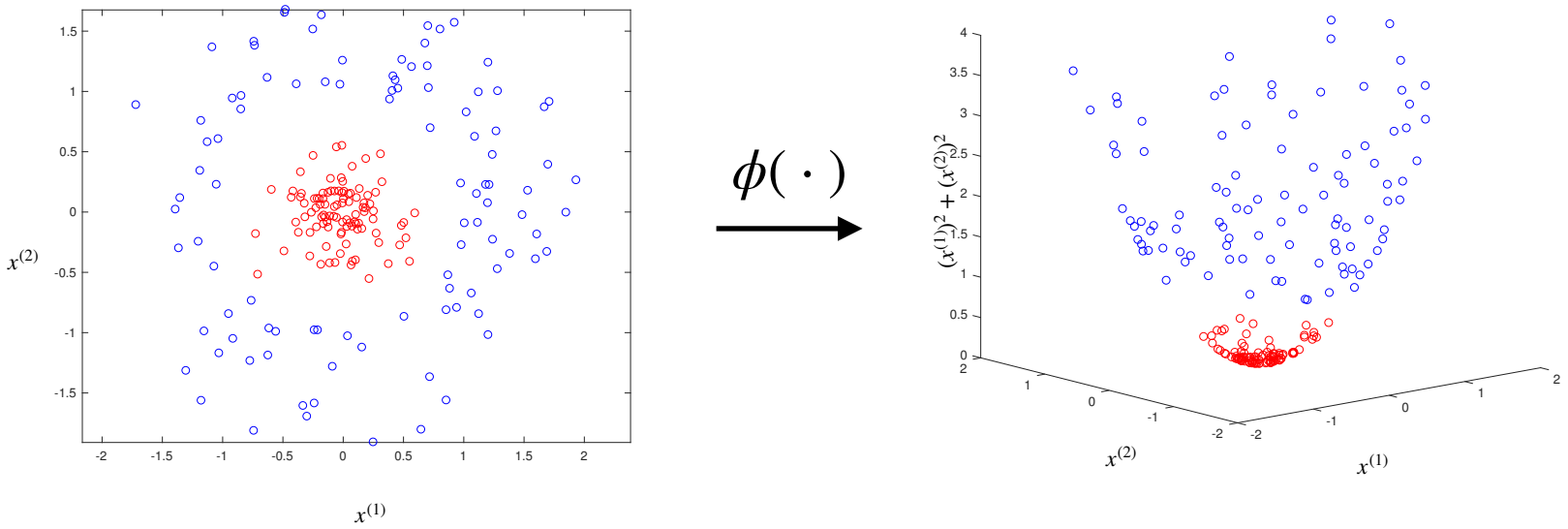
$$\hat{y}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$$



- Lesson learned: Simple to use and often effective

Lessons learned until now

- Feature expansion and kernel methods (inherently) map the data to a different representation



- Lesson learned: Transforming the input nonlinearly gives more flexibility

From handcrafted representations to learned ones

- The representations and data processing strategies we have discussed so far were all manually designed
  - The transformation corresponding to a kernel is still defined manually
- There is, however, no guarantee that they are the best representations for the task at hand
- Ideally, one would therefore like to learn the representation jointly with the classifier
  - Artificial neural networks are a way to do so

Motivating example: Curve fitting

- In the 1D input and 1D out scenario, we would like to find a mapping from x to y that approximates the curve below

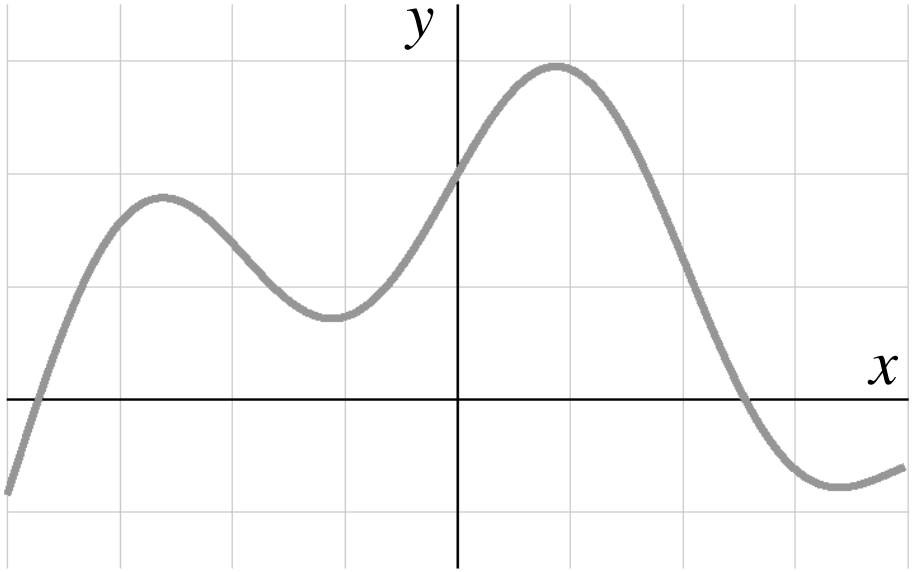


Figure by François Fleuret

Curve fitting: Feature expansion

- In a previous lecture, we have attempted to do so by performing a polynomial feature expansion

$$x \longrightarrow \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^M \end{bmatrix}$$

- We then applied a linear model to the expanded features
- In this case, the feature expansion was manually defined, independently of the linear model training
  - Instead, we would like to learn the feature expansion and the linear model jointly

Parametric feature expansion

- To achieve this, we can define a parametric feature expansion strategy
- Because we like linear models, let's use one for this purpose
- Naively, we could then write the mapping

$$x \longrightarrow \mathbf{W}_{(1)}^{T} \begin{bmatrix} 1 \\ x \end{bmatrix} , \quad \text{with } \mathbf{W}_{(1)} \in \mathbb{R}^{2 \times M}$$

  and then apply a linear model with parameters $\mathbf{w}_{(2)} \in \mathbb{R}^{M \times 1}$ to the resulting representation to predict y

- However, as mentioned in a previous lecture, linear feature expansion is useless; you can achieve the same results with just one linear model

Parametric feature expansion

- We therefore need to add nonlinearity in the process
- To achieve this, we will use a nonlinear function $f : \mathbb{R} \rightarrow \mathbb{R}$
- To this end, let's define

$$\mathbf{a}_{(1)} = \mathbf{W}_{(1)}^{T} \begin{bmatrix} 1 \\ x \end{bmatrix} = \begin{bmatrix} W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x \\ W_{(1)}^{(2,0)} + W_{(1)}^{(2,1)} x \\ \vdots \\ W_{(1)}^{(M,0)} + W_{(1)}^{(M,1)} x \end{bmatrix} = \begin{bmatrix} a_{(1)}^{(1)} \\ a_{(1)}^{(2)} \\ \vdots \\ a_{(1)}^{(M)} \end{bmatrix}$$

Parametric feature expansion

- Then, our parametric feature expansion process can be expressed as

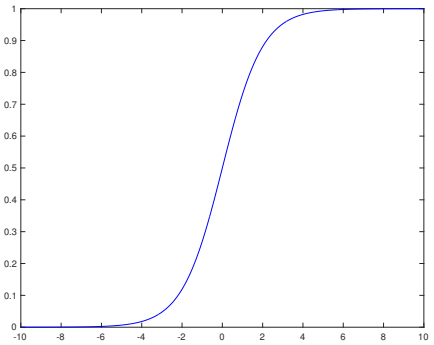
$$x \longrightarrow \begin{bmatrix} f(a_{(1)}^{(1)}) \\ f(a_{(1)}^{(2)}) \\ \vdots \\ f(a_{(1)}^{(M)}) \end{bmatrix} = f\left(\mathbf{W}_{(1)}^{T} \begin{bmatrix} 1 \\ x \end{bmatrix}\right)$$

  where the final notation applies the function $f()$ in an elementwise manner to the M-dimensional vector output by the linear mapping

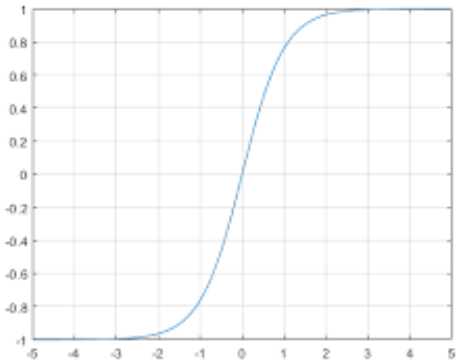
- In the context of artificial neural networks the function $f()$ is referred to as the *activation function*
  - Note that the same function is applied to every $a_{(1)}^{(j)}$, which avoids having to design too many things manually

Activation function: Examples

- Sigmoid: $f(a) = \dfrac{1}{1+\exp(-a)}$

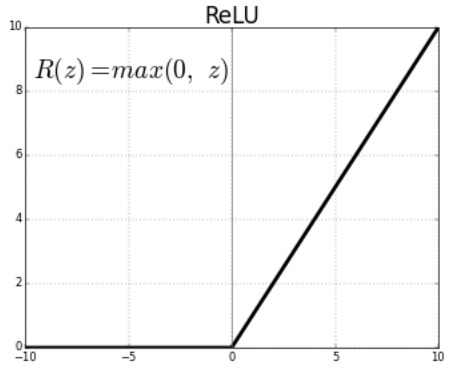


- Hyperbolic tangent: $f(a) = \tanh(a)$



- Rectified Linear Unit (ReLU)

$$f(a) = \begin{cases} a\,, & \text{if } a > 0 \\ 0\,, & \text{otherwise} \end{cases} = \max(a,0)$$



Hidden representation

- The M-dimensional vector

$$\mathbf{z}_{(1)} = \begin{bmatrix} f(a_{(1)}^{(1)}) \\ f(a_{(1)}^{(2)}) \\ \vdots \\ f(a_{(1)}^{(M)}) \end{bmatrix} = f\left(\mathbf{W}_{(1)}^{T} \begin{bmatrix} 1 \\ x \end{bmatrix} \right)$$

  is commonly referred to as *hidden representation* (or *hidden layer*)

- Each one of its M elements is referred to as a *hidden unit*

Complete model

- We can then perform prediction by applying a second linear model to the hidden representation
- This gives

$$\hat{y} = \mathbf{w}_{(2)}^T \mathbf{z}_{(1)} = \mathbf{w}_{(2)}^T f\left(\mathbf{W}_{(1)}^T \begin{bmatrix} 1 \\ x \end{bmatrix}\right) = \mathbf{w}_{(2)}^T \begin{bmatrix} f(a_{(1)}^{(1)}) \\ f(a_{(1)}^{(2)}) \\ \vdots \\ f(a_{(1)}^{(M)}) \end{bmatrix}$$

$$= \sum_{j=1}^{M} w_{(2)}^{(j)} f\left(W_{(1)}^{(j,0)} + W_{(1)}^{(j,1)} x\right)$$

- Let's get back to our curve fitting example and see what we can approximate with this formulation

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions

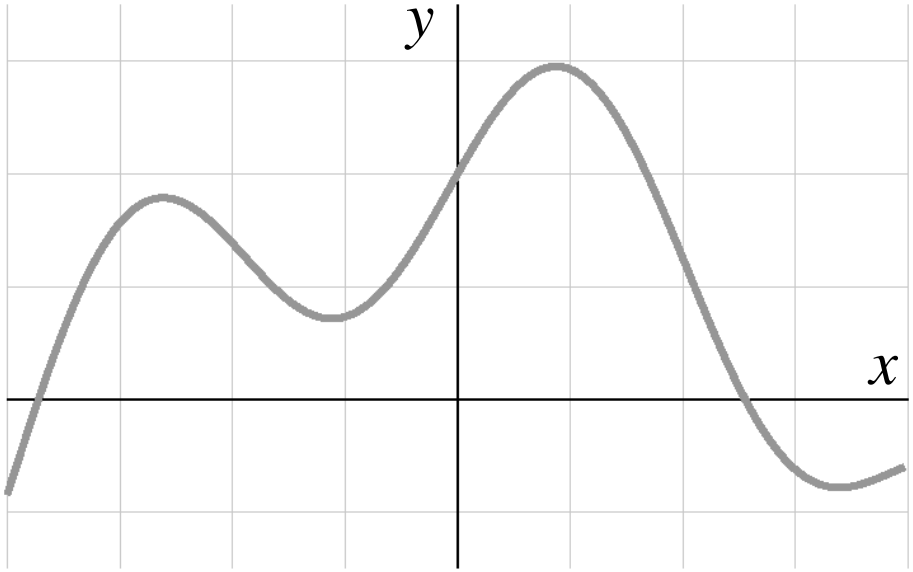


Figure by François Fleuret

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions $f(\cdot)$

$$\hat{y} = w_{(2)}^{(1)} f\left(W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x\right)$$

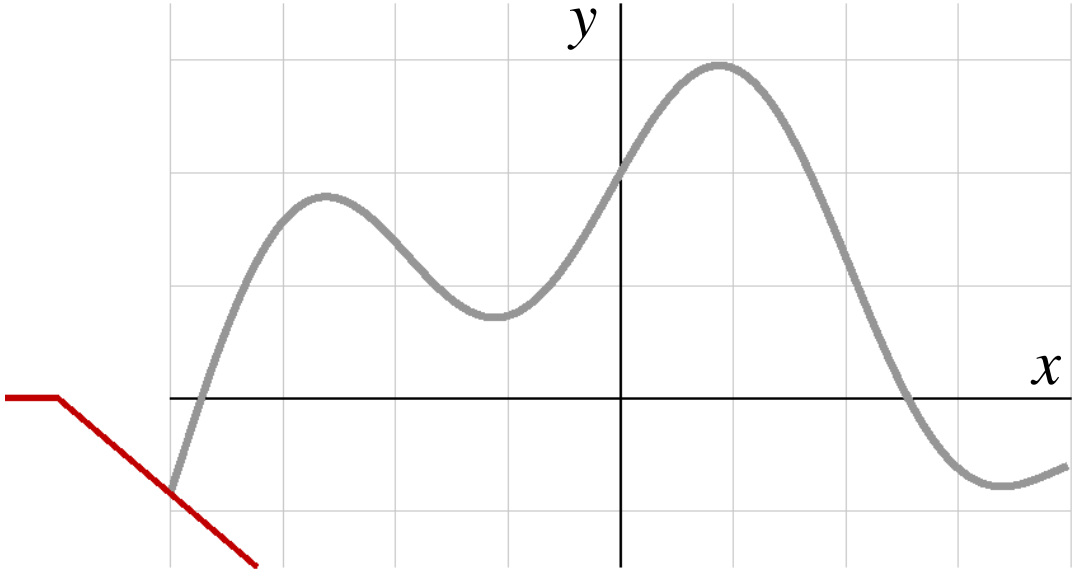


Figure by François Fleuret

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions $f(\cdot)$

$$\hat{y} = w_{(2)}^{(1)} f\left(W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x\right) + w_{(2)}^{(2)} f\left(W_{(1)}^{(2,0)} + W_{(1)}^{(2,1)} x\right)$$

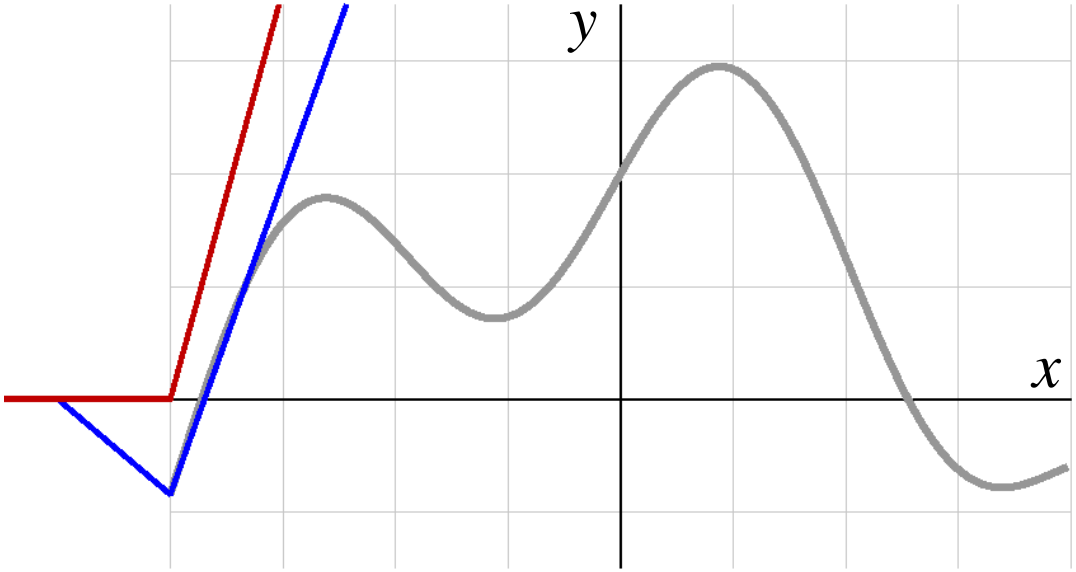


Figure by François Fleuret

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions $f(\cdot)$

$$\hat{y} = w_{(2)}^{(1)} f\left(W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x\right) + w_{(2)}^{(2)} f\left(W_{(1)}^{(2,0)} + W_{(1)}^{(2,1)} x\right) + w_{(2)}^{(3)} f\left(W_{(1)}^{(3,0)} + W_{(1)}^{(3,1)} x\right)$$

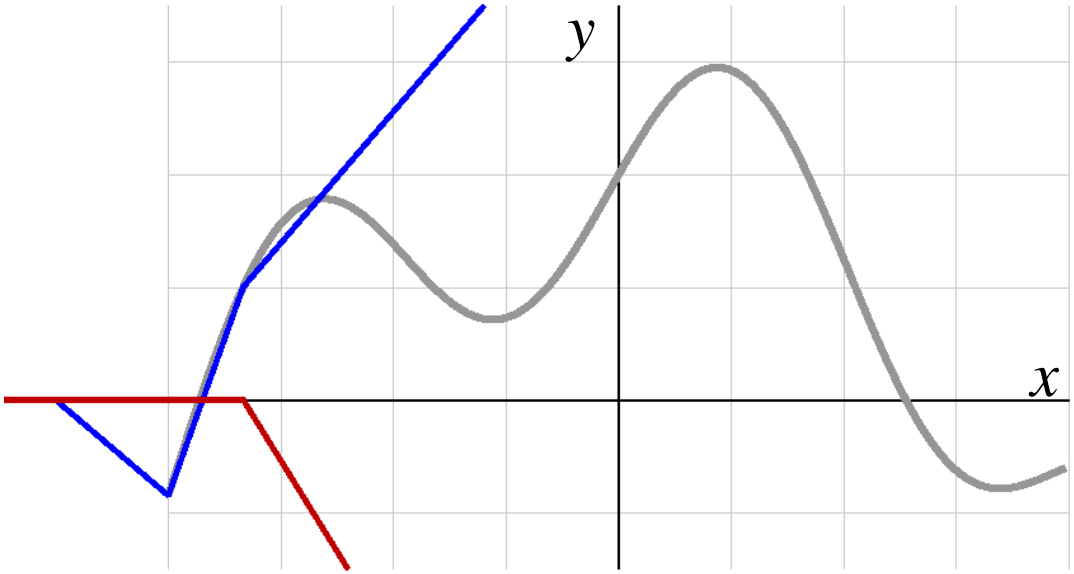


Figure by François Fleuret

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions $f(\cdot)$

$$\hat{y} = w_{(2)}^{(1)} f\left(W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x\right) + w_{(2)}^{(2)} f\left(W_{(1)}^{(2,0)} + W_{(1)}^{(2,1)} x\right) + w_{(2)}^{(3)} f\left(W_{(1)}^{(3,0)} + W_{(1)}^{(3,1)} x\right) + \cdots$$

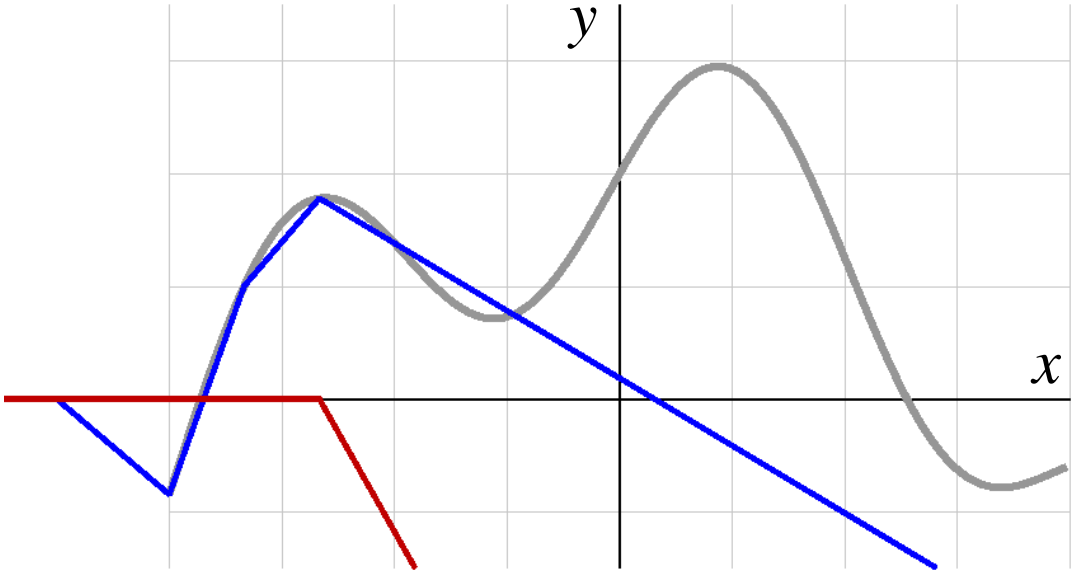


Figure by François Fleuret

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions $f(\cdot)$

$$\hat{y} = w_{(2)}^{(1)} f\left(W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x\right) + w_{(2)}^{(2)} f\left(W_{(1)}^{(2,0)} + W_{(1)}^{(2,1)} x\right) + w_{(2)}^{(3)} f\left(W_{(1)}^{(3,0)} + W_{(1)}^{(3,1)} x\right) + \cdots$$

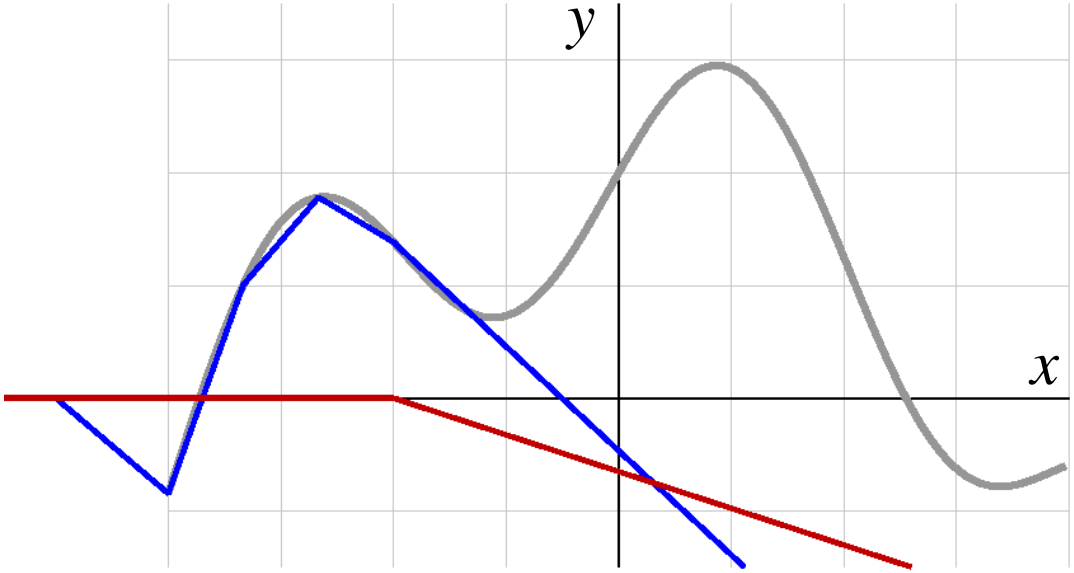
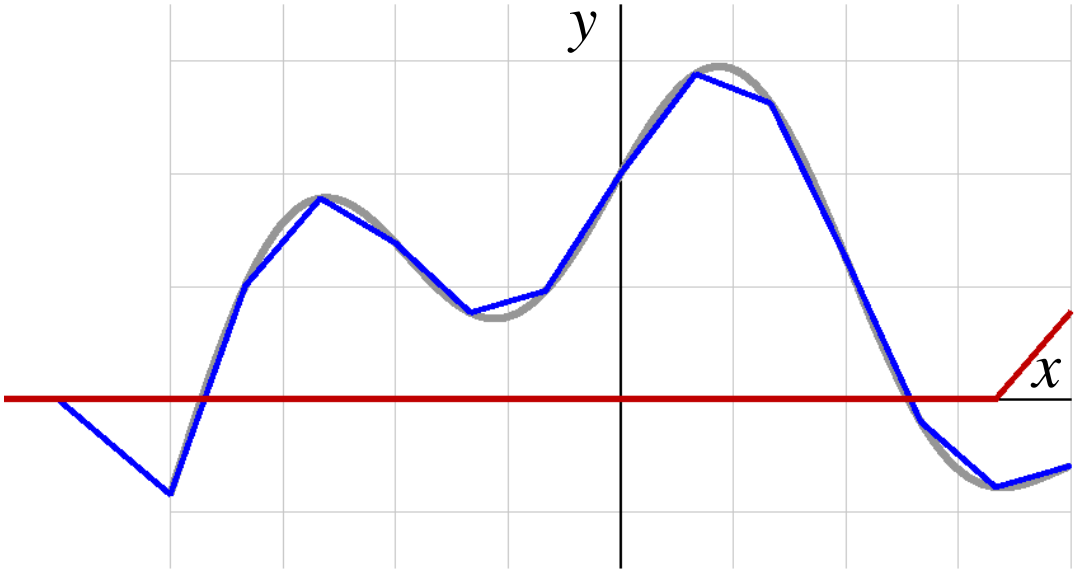


Figure by François Fleuret

Universal approximation

- We can approximate any continuous function with a linear combination of translated/scaled ReLU functions $f(\cdot)$

$$\hat{y} = w_{(2)}^{(1)} f\left(W_{(1)}^{(1,0)} + W_{(1)}^{(1,1)} x\right) + w_{(2)}^{(2)} f\left(W_{(1)}^{(2,0)} + W_{(1)}^{(2,1)} x\right) + w_{(2)}^{(3)} f\left(W_{(1)}^{(3,0)} + W_{(1)}^{(3,1)} x\right) + \cdots$$



- This is also true for other activation functions under mild assumptions

Figure by François Fleuret

Dealing with multiple input dimensions

- Handling $D > 1$ input dimensions makes very little difference in the model formulation
- From the 1D input formulation

$$\hat{y} = \mathbf{w}_{(2)}^T f\left(\mathbf{W}_{(1)}^T \begin{bmatrix} 1 \\ x \end{bmatrix}\right)$$

- we simply write the D-dimensional case as

$$\hat{y} = \mathbf{w}_{(2)}^T f\left(\mathbf{W}_{(1)}^T \mathbf{x}\right)$$

where $\mathbf{x} \in \mathbb{R}^{D+1}$ to account for the bias, and $\mathbf{W}_{(1)}$ is of dimension $(D+1) \times M$ instead of $2 \times M$

Dealing with multiple output dimensions

- To handle $C > 1$ output dimensions, we can proceed as in the case of linear models
- We simply need to replace the vector of parameters $\mathbf{w}_{(2)}$ with a matrix of parameters $\mathbf{W}_{(2)} \in \mathbb{R}^{M \times C}$

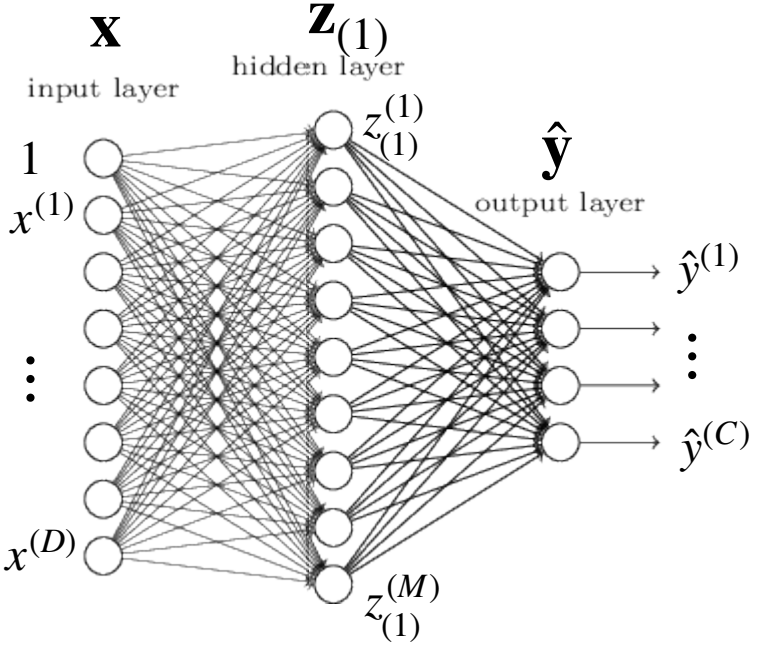
- The prediction can then be written as

$$\hat{\mathbf{y}} = \mathbf{W}_{(2)}^T f\left(\mathbf{W}_{(1)}^T \mathbf{x}\right)$$

where $\hat{\mathbf{y}} \in \mathbb{R}^C$ is now a vector of dimension C

Simple artificial neural network

- The resulting simple artificial neural network with one hidden layer can be depicted as shown below



- Because we use a matrix-vector product for each layer, the layers are called *fully-connected*
  - Each hidden dimension (hidden unit) $z_{(1)}^{(j)}$ depends on all input dimensions $x^{(1)}, \ldots, x^{(D)}$
  - Each output dimension $\hat{y}^{(k)}$ depends on all hidden units $z_{(1)}^{(1)}, \ldots, z_{(1)}^{(M)}$

Simple network: Function approximation

- Output of ReLU activation (b in the figure corresponds to $w^{(0)}$)

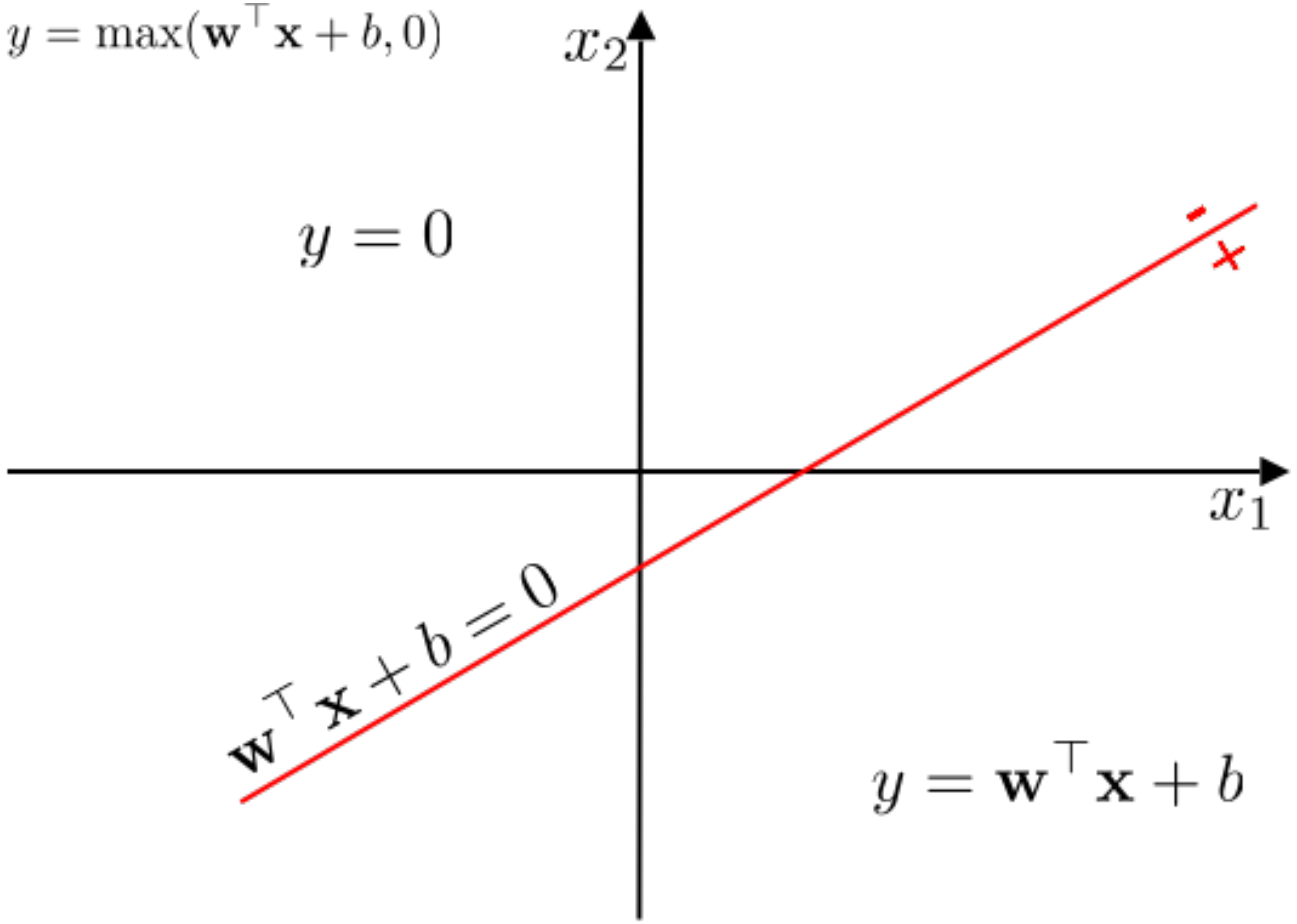


Figure by Vincent Lepetit

Simple network: Function approximation

- Two hidden units with ReLU activation ($\mathbf{h}$ corresponds to $\mathbf{z}$)

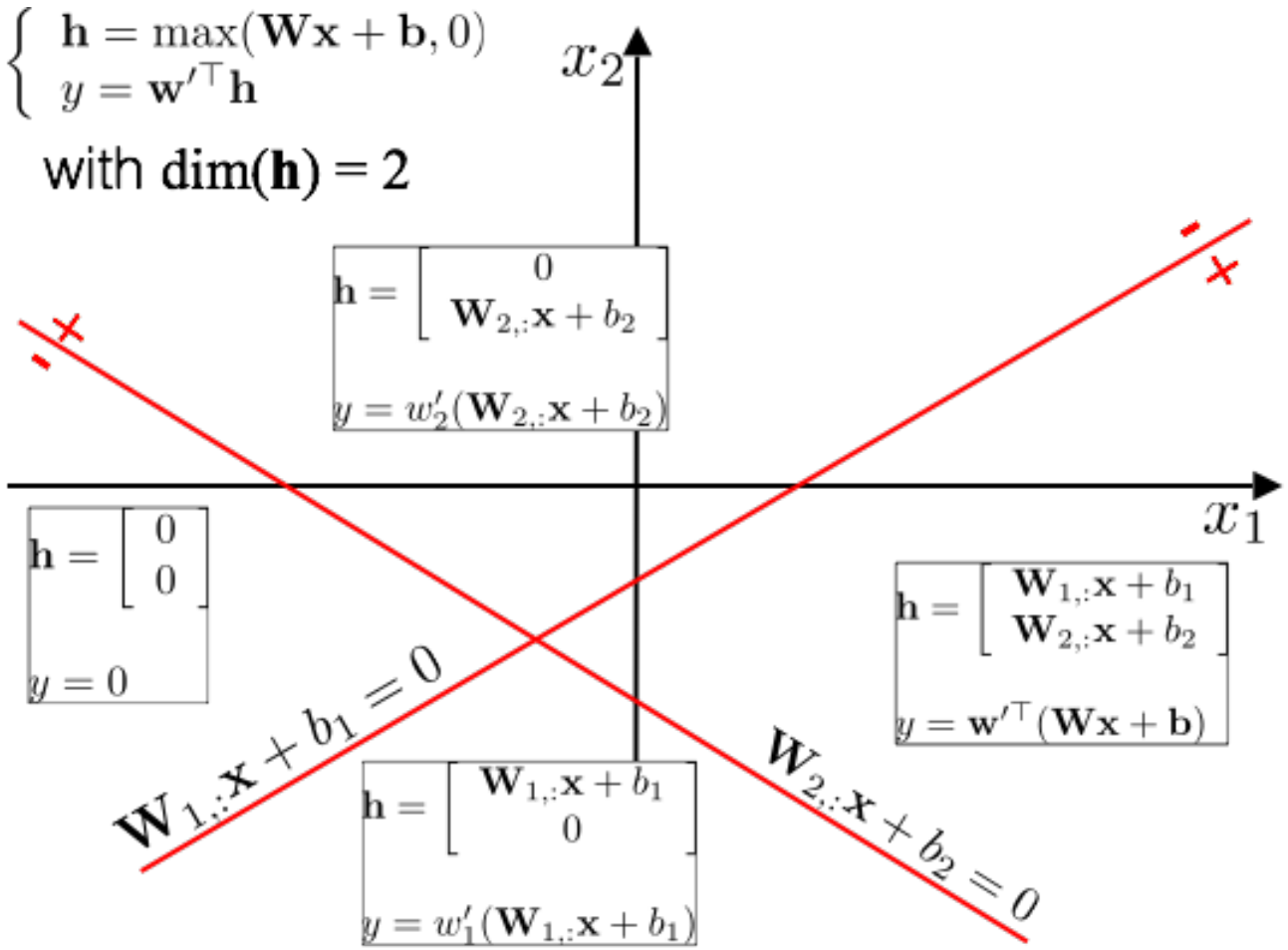


Figure by Vincent Lepetit

Simple network: Function approximation

- Three hidden units with ReLU activation ($\mathbf{h}$ corresponds to $\mathbf{z}$)

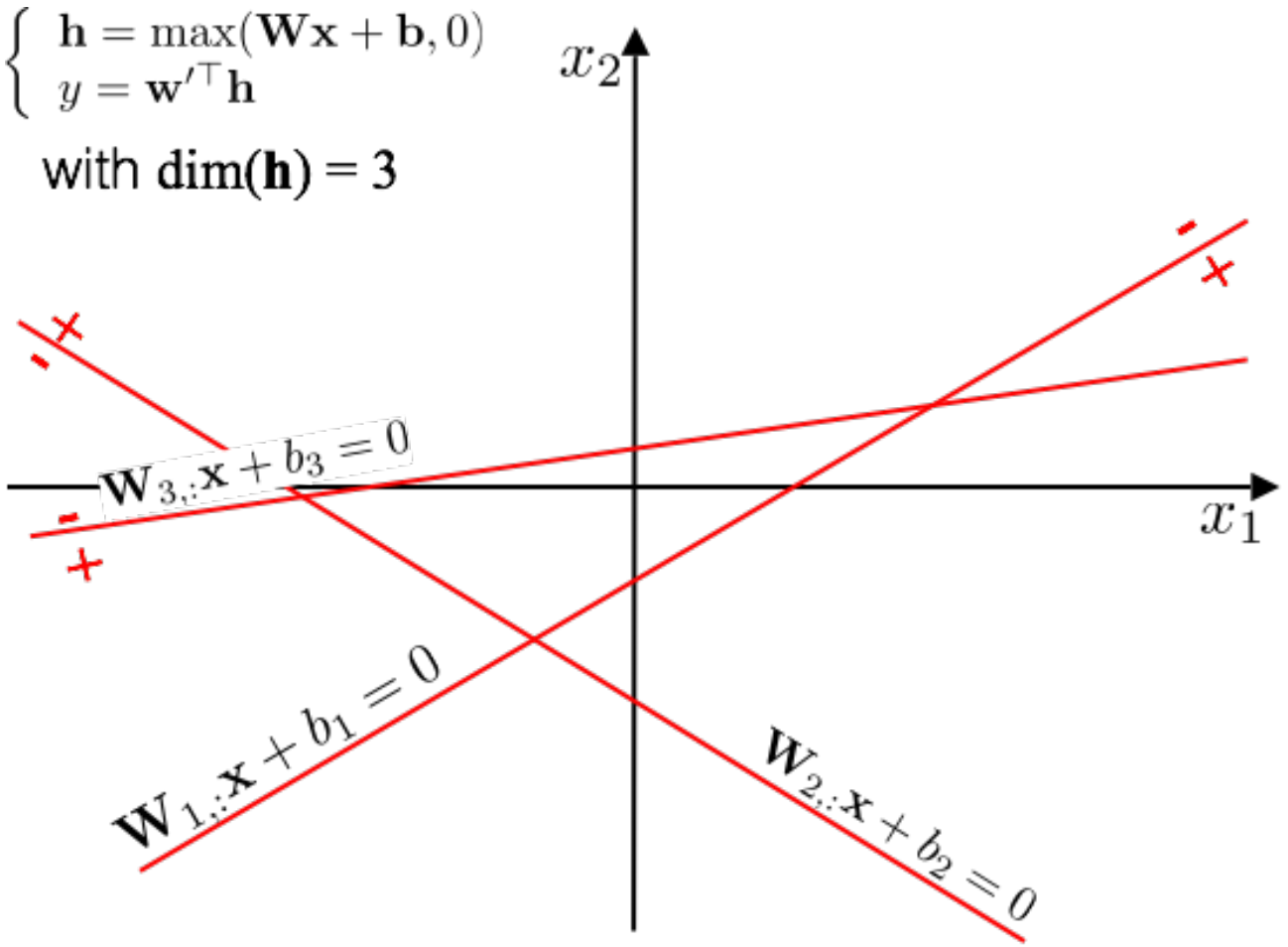
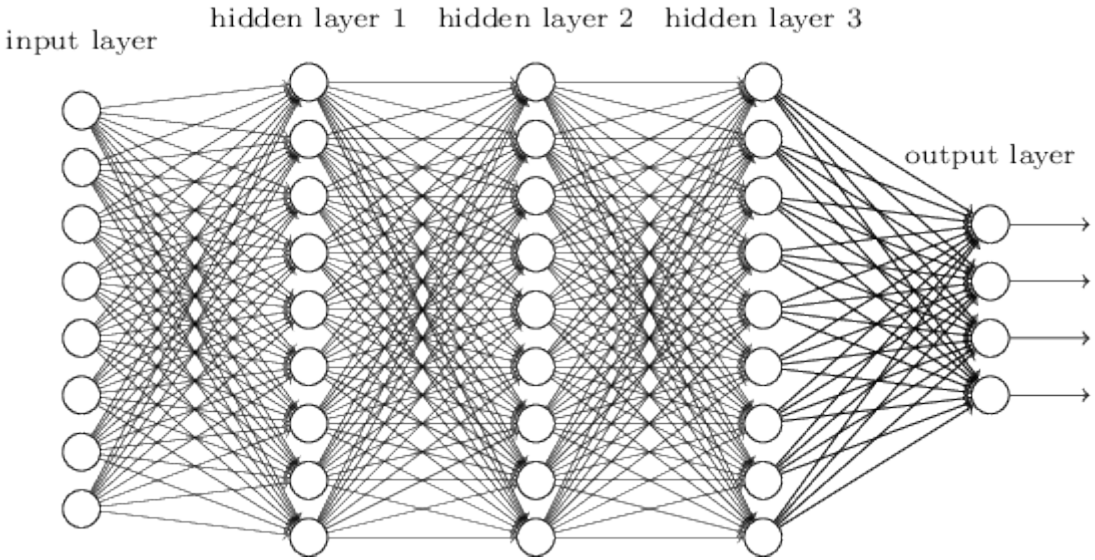


Figure by Vincent Lepetit

Multilayer Perceptron

- With a single hidden layer, one can already model complicated functions
- However, why stop at just one hidden layer?



- Slightly misleading name:
  - The perceptron uses a step function as activation function
  - MLPs typically rely on continuous functions (e.g., sigmoid, ReLU)

Multilayer Perceptron

- In essence, each hidden layer takes as input the output of the previous layer
- For the first hidden layer, we still have

$$\mathbf{z}_{(1)} = f\left(\mathbf{W}_{(1)}^{T}\mathbf{x}\right)$$

- For any subsequent layer l, we have

$$\mathbf{z}_{(l)} = f\left(\mathbf{W}_{(l)}^{T}\mathbf{z}_{(l-1)}\right)$$

- Finally, for a network of L layers, we compute the output as

$$\hat{\mathbf{y}} = \mathbf{W}_{(L)}^{T}\mathbf{z}_{(L-1)}$$

- The process of going from the input to the output is called the *forward pass* of the network

Multilayer Perceptron

- If we wanted to write the prediction as a single equation, it would look like

$$\hat{\mathbf{y}} = \mathbf{W}_{(L)}^T f\left(\mathbf{W}_{(L-1)}^T f\left(\mathbf{W}_{(L-2)}^T f\left(\cdots f\left(\mathbf{W}_{(1)}^T \mathbf{x}\right)\right)\cdots\right)\right)$$

- This is not very convenient, and it is easier to simply think of $\hat{\mathbf{y}}$ as a function of all parameters

$$\hat{\mathbf{y}} = \hat{\mathbf{y}}(\mathbf{W})$$

where $\mathbf{W} = \left\{\mathbf{W}_{(l)}\right\}$ encompasses the parameters of all the layers

Training an MLP

- Given a set of N training samples $(\mathbf{x}_i, \mathbf{y}_i)$, we would then like to find the best set of parameters $\mathbf{W}^*$
- As usual, this is done by minimizing an empirical risk w.r.t. all the parameters $\mathbf{W}$
- For regression, the loss function is typically the square loss

$$R(\mathbf{W}) = \frac{1}{N} \sum_{i=1}^{N} \|\hat{\mathbf{y}}_i(\mathbf{W}) - \mathbf{y}_i\|^2$$

where the dependence of $\hat{\mathbf{y}}_i$ on the parameters $\mathbf{W}$ is denoted as in the previous slide

Training an MLP

- Because an MLP stacks multiple layers, each of which has a nonlinear activation function, the problem does not have a closed-form solution and is not convex

- Learning is therefore done by gradient descent
  - Question: Can you think of a drawback of this strategy?

- While the gradient might seem complicated, it can in fact be computed in a nice manner
  - This is called backpropagation

Gradient-based learning

- Because the empirical risk is an average over the training samples, its gradient will also be an average of gradients
- Thus, we can focus on a single loss term $\ell(\hat{\mathbf{y}}_i, \mathbf{y}_i)$, where

$$R(\{\mathbf{W}_{(l)}\}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{N}\sum_{i=1}^{N} \ell_i$$

- For simplicity, I will assume that all hidden layers use the same type of activation function, denoted by $f(\cdot)$

Gradient-based learning

- For the k^{th} dimension of the output of any layer l, let us define

$$a_{(l)}^{(k)} = \sum_{j} W_{(l)}^{(k,j)} z_{(l-1)}^{(j)}$$

  which denotes the dot product between the k^{th} column of $\mathbf{W}_{(l)}$ and $\mathbf{z}_{(l-1)}$

- What we aim to compute is the derivative of the loss function ℓ_i w.r.t. any parameter $W_{(l)}^{(k,j)}$

- Following the chain rule, we can write such a derivative as

$$\frac{\partial \ell_i}{\partial W_{(l)}^{(k,j)}} = \frac{\partial \ell_i}{\partial a_{(l)}^{(k)}} \frac{\partial a_{(l)}^{(k)}}{\partial W_{(l)}^{(k,j)}}$$

Gradient-based learning

- Let us now consider the second term of the previous formula

- From the definition from the previous slide (repeated here)

$$a_{(l)}^{(k)} = \sum_j W_{(l)}^{(k,j)} z_{(l-1)}^{(j)}$$

we can see that

$$\frac{\partial a_{(l)}^{(k)}}{\partial W_{(l)}^{(k,j)}} = z_{(l-1)}^{(j)}$$

Gradient-based learning

- Furthermore, let us define

$$\delta_{(l)}^{(k)} = \frac{\partial \ell_i}{\partial a_{(l)}^{(k)}}$$

  which directly encodes the first term of the derivative

- So now we can rewrite the derivative

$$\frac{\partial \ell_i}{\partial W_{(l)}^{(k,j)}} = \delta_{(l)}^{(k)} z_{(l-1)}^{(j)}$$

- Because the hidden values $\{z_{(l-1)}^{(j)}\}$ are computed in the forward pass, we only need to compute the $\{\delta_{(l)}^{(k)}\}$ to obtain the derivative of the loss w.r.t. any parameter

Gradient-based learning

- For the last layer (Layer L), we have (assuming no final activation)

$$a_{(L)}^{(k)} = \hat{y}^{(k)}$$

- This means that, for the last layer, $\delta_{(L)}^{(k)}$ can be computed explicitly as the partial derivative of the loss function w.r.t. the network's output

- E.g., for the square loss, we have

$$\delta_{(L)}^{(k)} = 2(\hat{y}_i^{(k)} - y_i^{(k)})$$

Gradient-based learning

- For the other layers ($l < L$), it can be shown that $\delta_{(l)}^{(k)}$ can be computed from:
  - the $\{\delta_{(l+1)}^{(j)}\}$ (from the subsequent layer $l+1$)
  - the derivative of the activation function w.r.t. its input
  - the parameters $\mathbf{W}_{(l+1)}$ (of the subsequent layer $l+1$)

Backpropagation

- In other words, given $\delta_{(l+1)}$, one can compute $\delta_{(l)}$
- Because we can explicitly compute $\delta_{(L)}$ for the last layer, we can in turn compute any $\delta_{(l)}$
- Then, this gives us a way to compute the gradient

$$\frac{\partial \ell_i}{\partial W_{(l)}^{(k,j)}} = \delta_{(l)}^{(k)} z_{(l-1)}^{(j)}$$

at any layer, starting from the last one

Backpropagation: Algorithm

1. Propagate $\mathbf{x}_i$ forward through the network
2. Compute $\delta_{(L)}$ depending on the loss
3. Propagate $\delta_{(L)}$ backward to obtain each $\delta_{(l)}$
4. At each layer, compute the partial derivatives

$$\frac{\partial \ell_i}{\partial W_{(l)}^{(k,j)}} = \delta_{(l)}^{(k)} z^{(j)}_{(l-1)}$$

where $\mathbf{z}_{(l-1)}$ has been computed during the forward pass

Gradient descent

- Once the gradient is computed, the update at iteration k is performed as

$$\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \eta \nabla_{\mathbf{W}} R(\mathbf{W}_{k-1})$$

where $\mathbf{W}$ encompasses the parameters of all layers

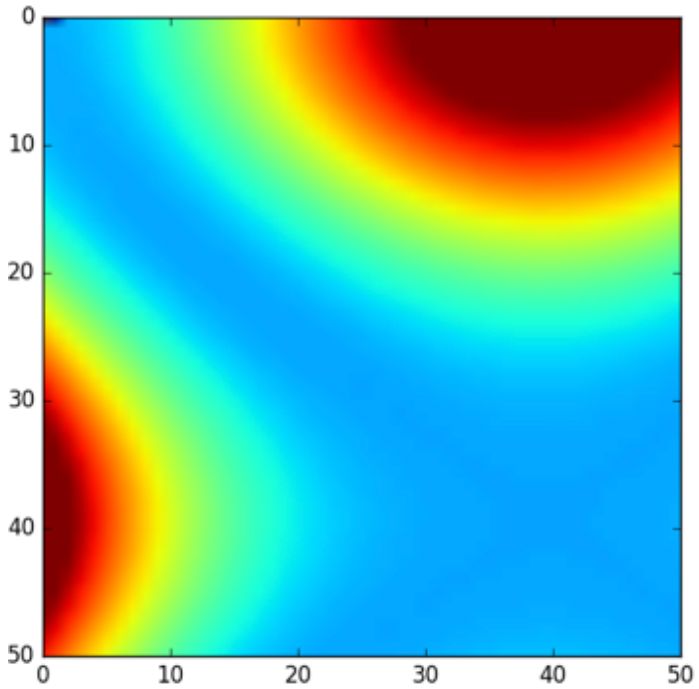
- You can also think of this as updating the parameters in each layer l as

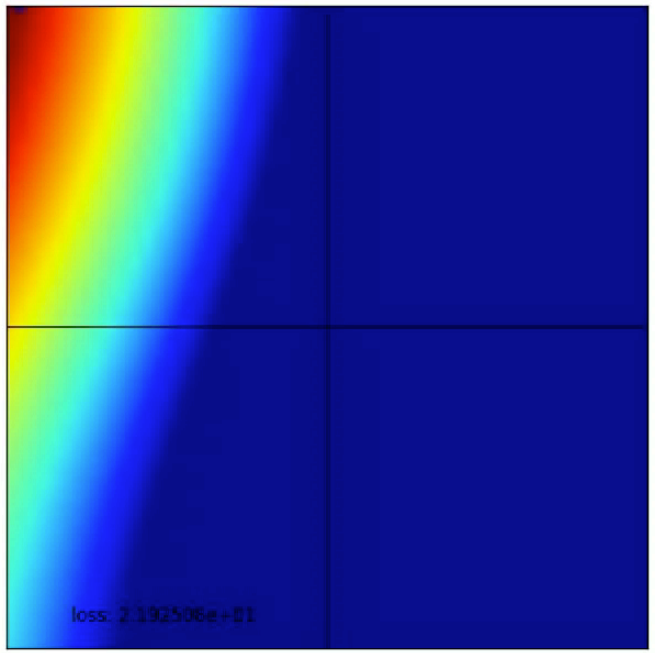
$$\mathbf{W}_{(l),k} \leftarrow \mathbf{W}_{(l),k-1} - \eta \nabla_{\mathbf{W}_{(l)}} R(\mathbf{W}_{k-1})$$

where $\mathbf{W}_{(l),k}$ represents the parameters of layer l at iteration k of gradient descent

Simple network training: Regression example

- Interpolate a surface
  - Input: 2D coordinate ($x^{(1)}$, $x^{(2)}$)
  - Output: 1D value ($y = 100(x^{(2)} - (x^{(1)})^2)^2 + (1 - x^{(1)})^2$, shown as color)



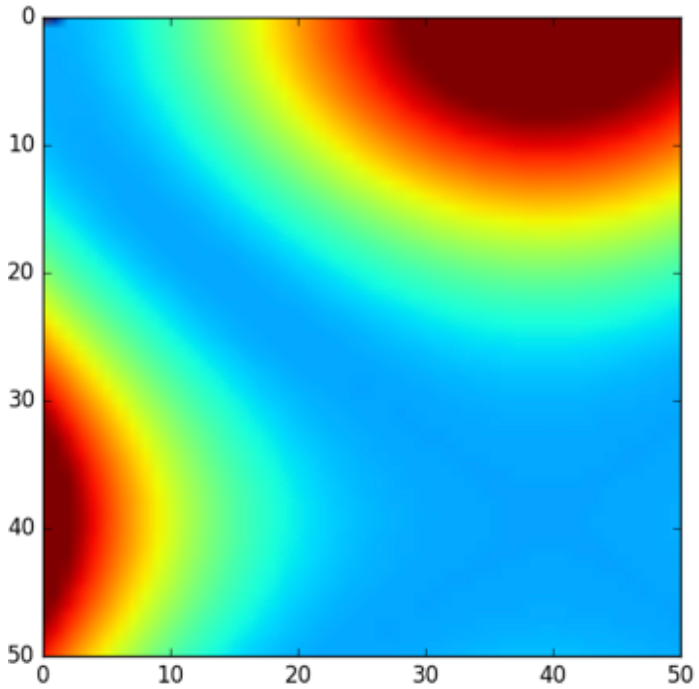


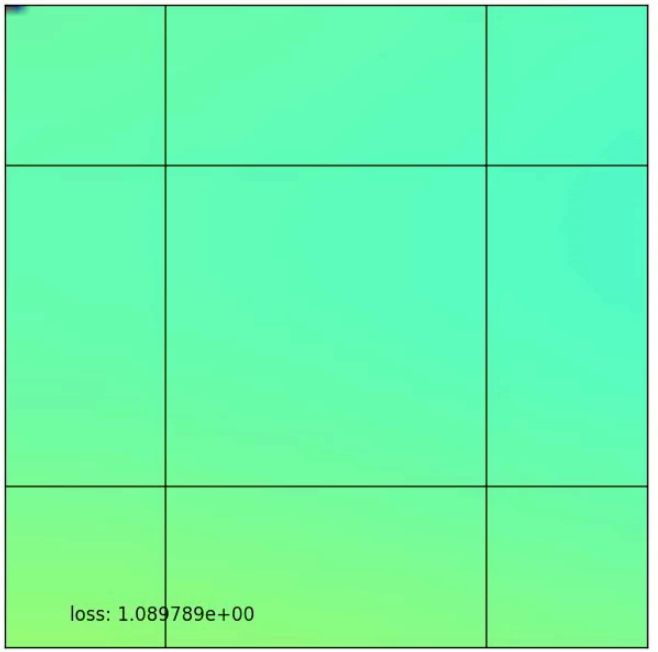
3 hidden units

Figure by Pascal Fua

Simple network training: Regression example

- Interpolate a surface
  - Input: 2D coordinate ($x^{(1)}$, $x^{(2)}$)
  - Output: 1D value ($y = 100(x^{(2)} - (x^{(1)})^2)^2 + (1 - x^{(1)})^2$, shown as color)



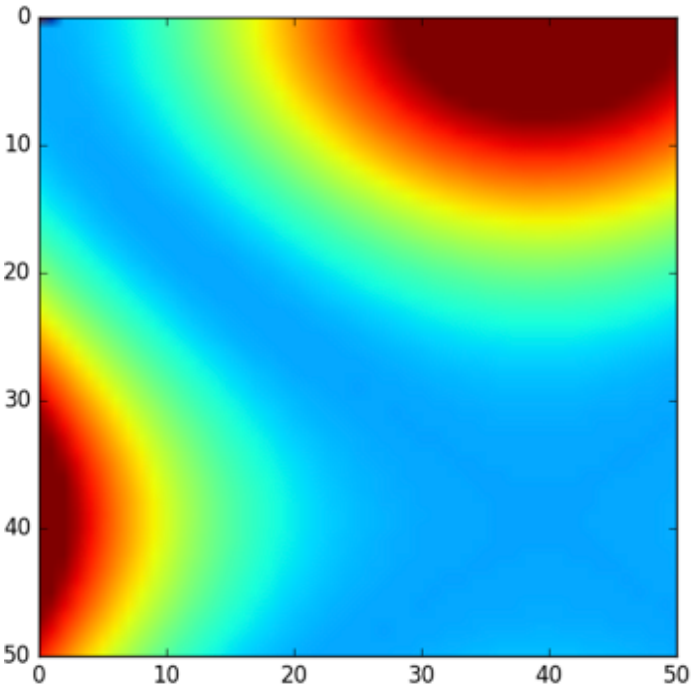


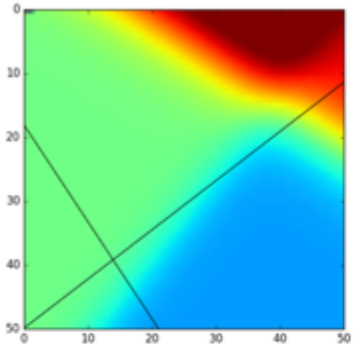
4 hidden units

Figure by Pascal Fua

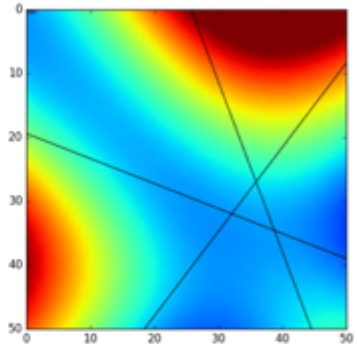
Simple network training: Regression example

- Interpolate a surface
  - Input: 2D coordinate $(x^{(1)}, x^{(2)})$
  - Output: 1D value ($y = 100(x^{(2)} - (x^{(1)})^2)^2 + (1 - x^{(1)})^2$, shown as color)

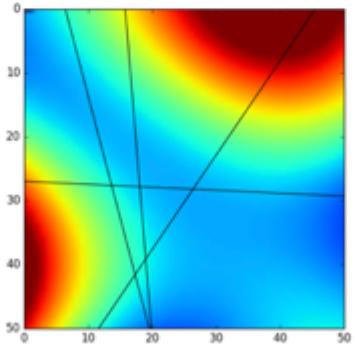




2 nodes -> loss 3.02e-01



3 nodes -> loss 2.08e-02

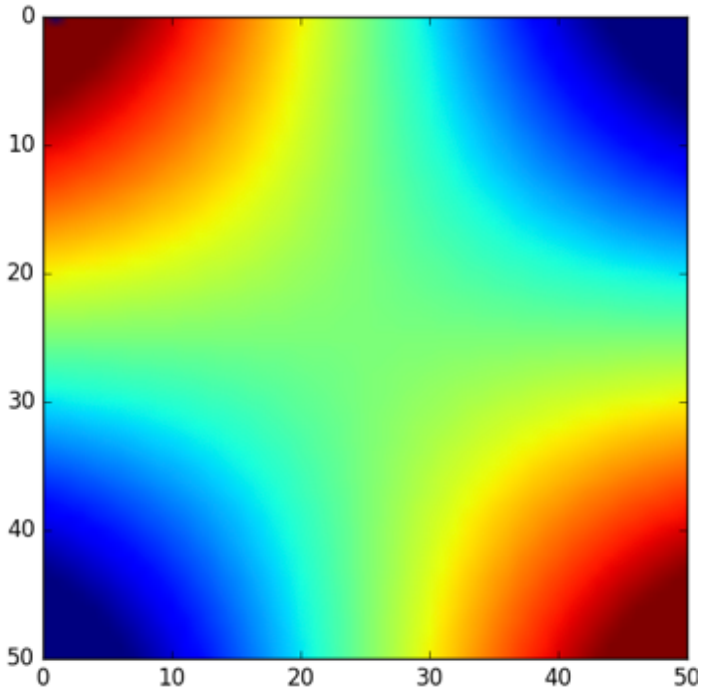


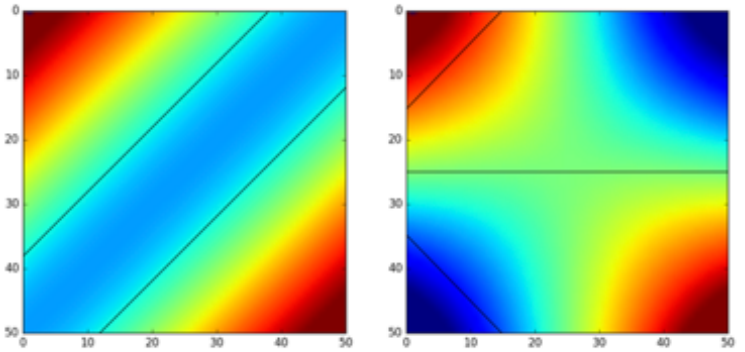
4 nodes -> loss 8.27e-03

Figure by Pascal Fua

Simple network training: Regression example

- Interpolate a surface
  - Input: 2D coordinate ($x^{(1)}$, $x^{(2)}$)
  - Output: 1D value ($y = \sin(x^{(1)})\sin(x^{(2)})$, shown as color)





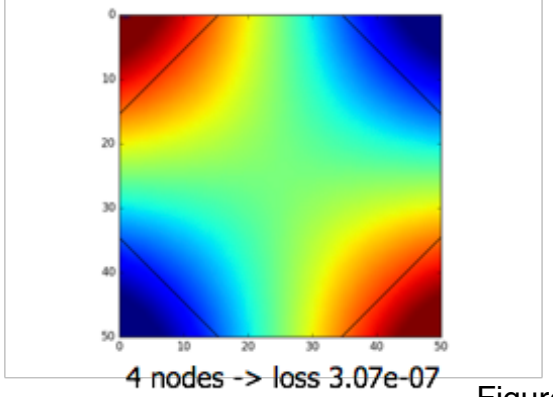


Figure by Pascal Fua

Gradient descent

- The empirical risk is the average of the loss function value over all training samples

- Because the gradient of an average is the average of the gradients, we can write also a standard gradient update as

$$\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{W}} \ell_i(\mathbf{W}_{k-1})$$

- This means that, for each gradient descent iteration, one needs to perform a forward pass and a backward pass through the network for all the training samples
  - This becomes expensive for large training sets

Stochastic gradient descent

- *Stochastic* gradient descent updates the parameters $\mathbf{W}$ based on the gradient of a single sample in each iteration

$$\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \eta \nabla \ell_{i(k)}(\mathbf{W}_{k-1})$$

At each iteration, we use the gradient of a single loss term.
The index of the sample depends on the iteration number k

- In practice, one then iterates over the training samples, either in a fixed order or in a random one

SGD with mini-batches

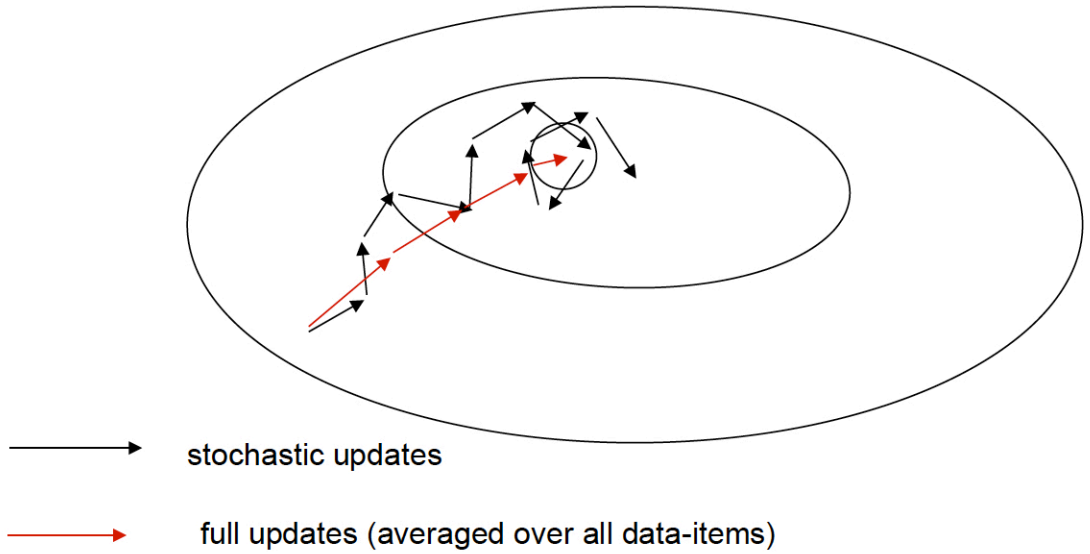
- The gradient obtained by a single sample might be a poor approximation of the true, complete gradient
  - E.g., an individual sample might benefit from the increase of one specific parameters, whereas most of the other samples would benefit from a decrease of this parameter
- To obtain a more reliable gradient estimate, one typically uses mini-batches of B samples
  - The parameters are then updated as

$$\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \eta \sum_{b=1}^{B} \nabla \ell_{i(b,k)}(\mathbf{W}_{k-1})$$

The index of the samples used for the update now depends on the iteration number k and on the index b of the samples in the mini-batch

SGD behavior

- Because it approximates the full gradient, SGD will move the parameters less directly towards a local minimum
  - The parameters dance around the minimum, which may require decreasing the learning rate
  - Nevertheless, SGD can help to avoid bad local minima



SGD behavior

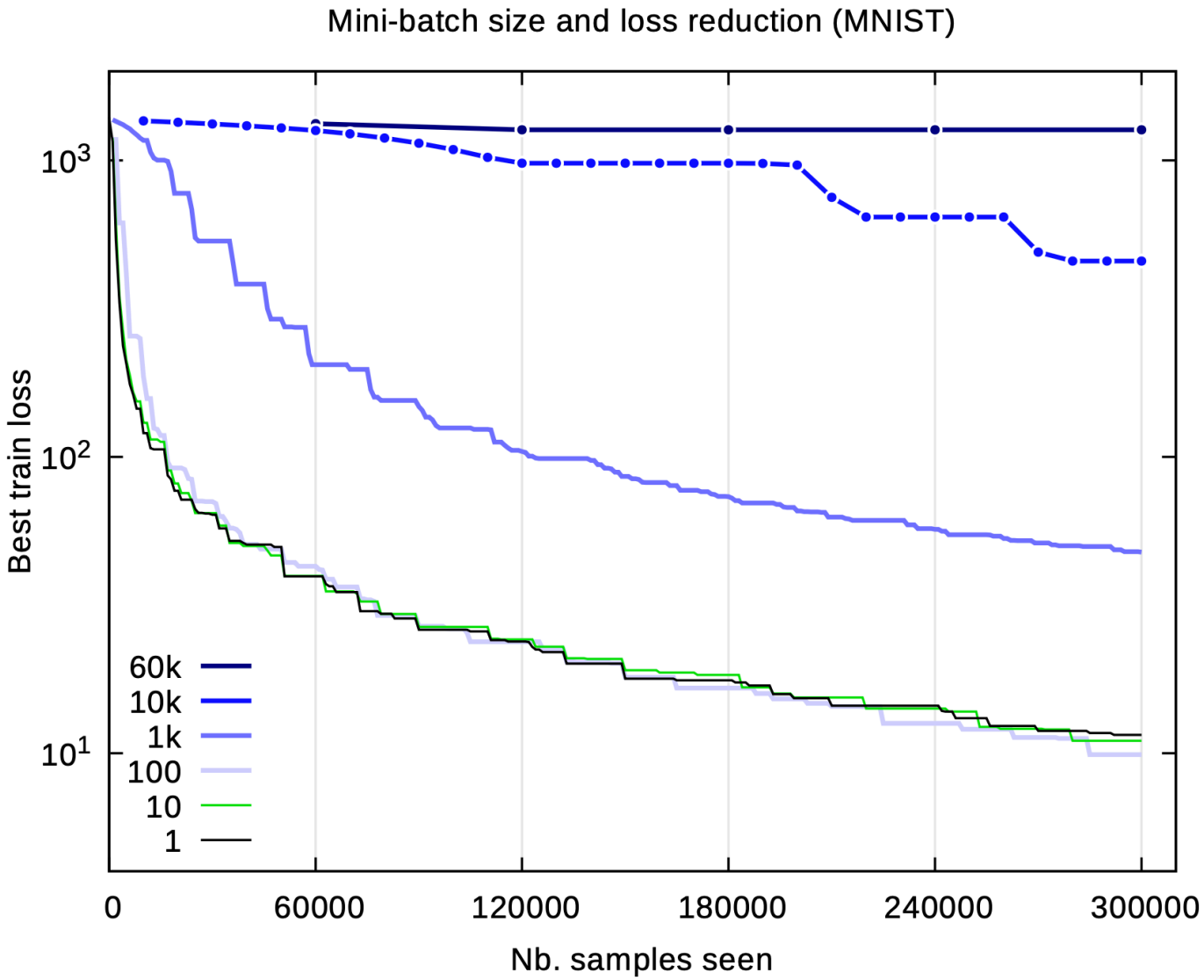
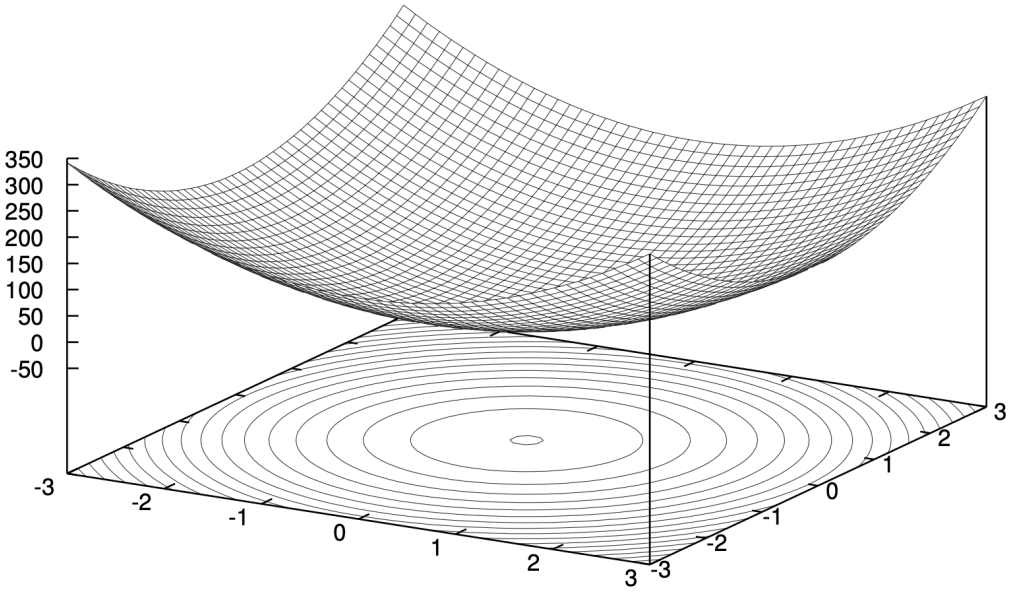


Figure by François Fleuret

Gradient descent: Limitations

- Gradient descent makes strong assumption about the magnitude and isotropy of the local curvature so that the same step size can be used in all directions



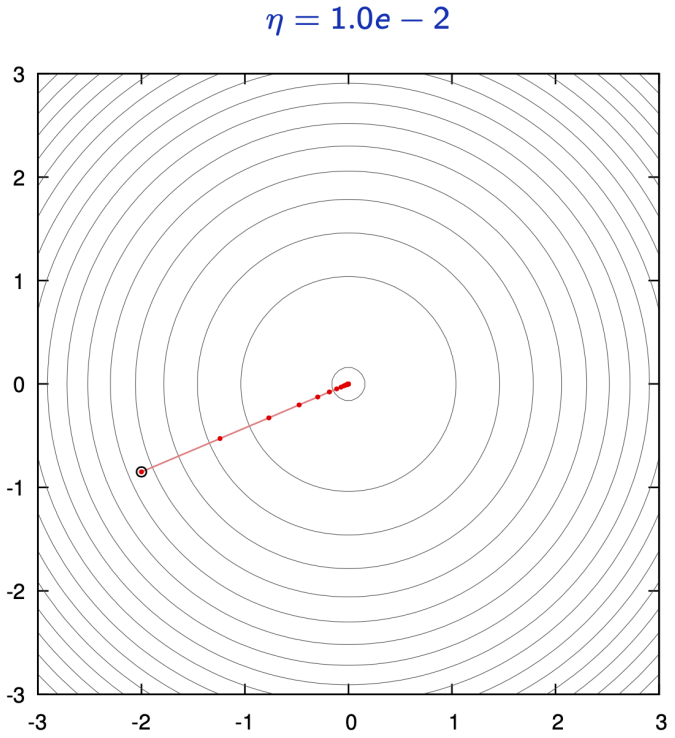


Figure by François Fleuret

Gradient descent: Limitations

- Gradient descent makes strong assumption about the magnitude and isotropy of the local curvature so that the same step size can be used in all directions

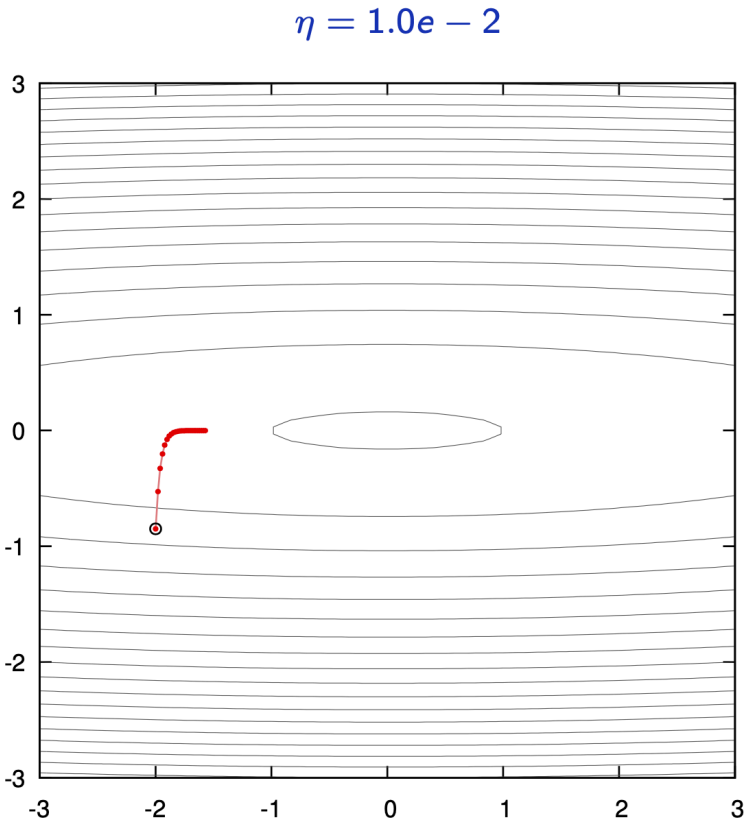


Figure by François Fleuret

Gradient descent: Limitations

- Gradient descent makes strong assumption about the magnitude and isotropy of the local curvature so that the same step size can be used in all directions

$\eta = 2.0e-2$

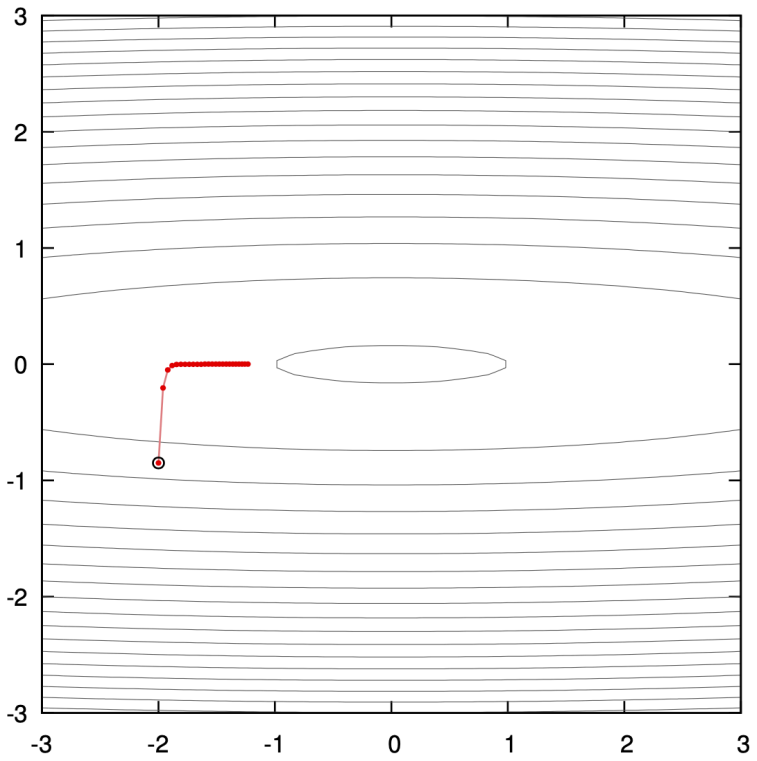


Figure by François Fleuret

Gradient descent: Limitations

- Gradient descent makes strong assumption about the magnitude and isotropy of the local curvature so that the same step size can be used in all directions

$\eta = 4.0e-2$

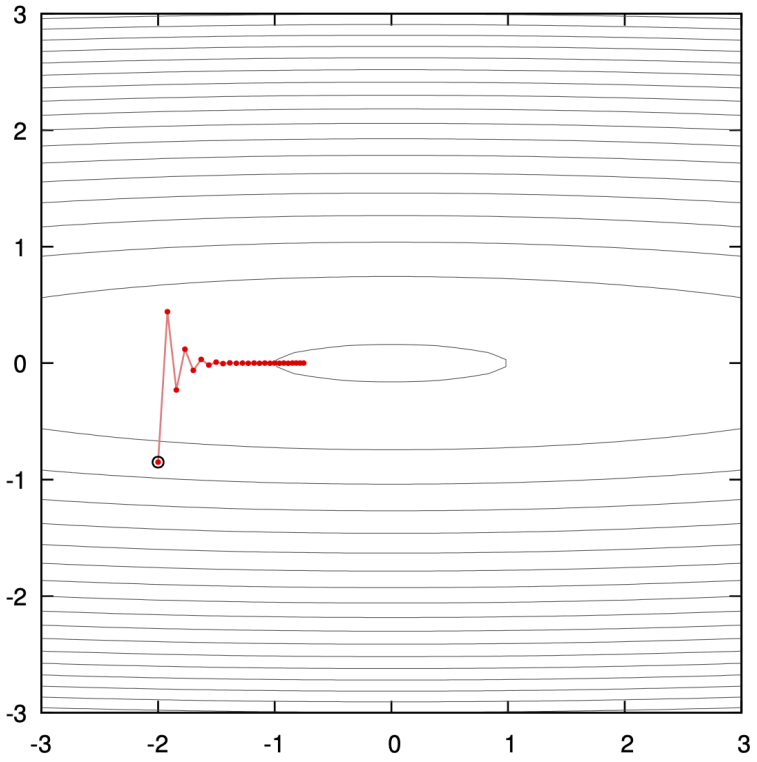


Figure by François Fleuret

Gradient descent: Limitations

- Gradient descent makes strong assumption about the magnitude and isotropy of the local curvature so that the same step size can be used in all directions

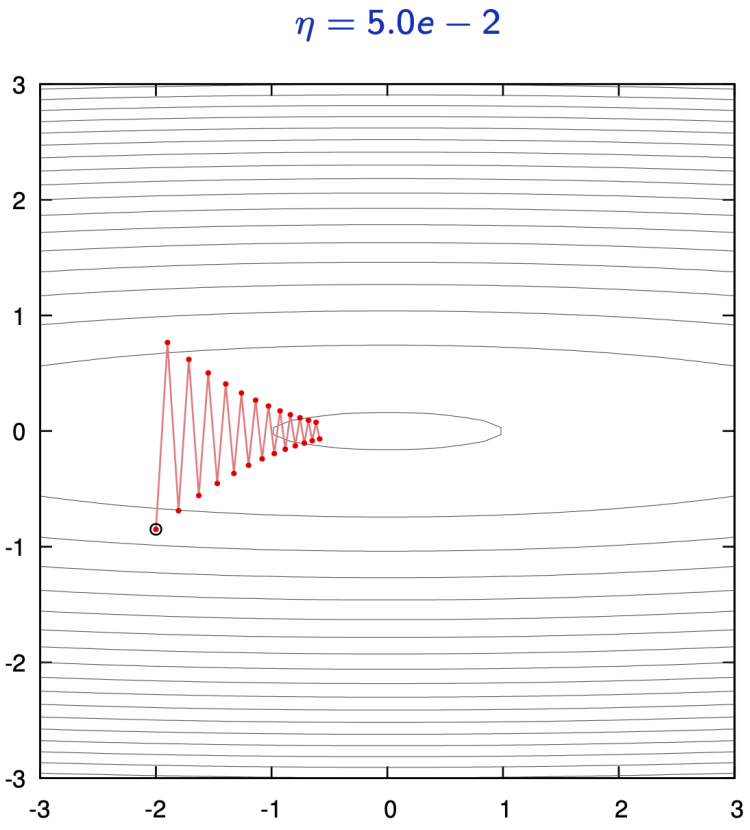


Figure by François Fleuret

Gradient descent: Limitations

- Gradient descent makes strong assumption about the magnitude and isotropy of the local curvature so that the same step size can be used in all directions

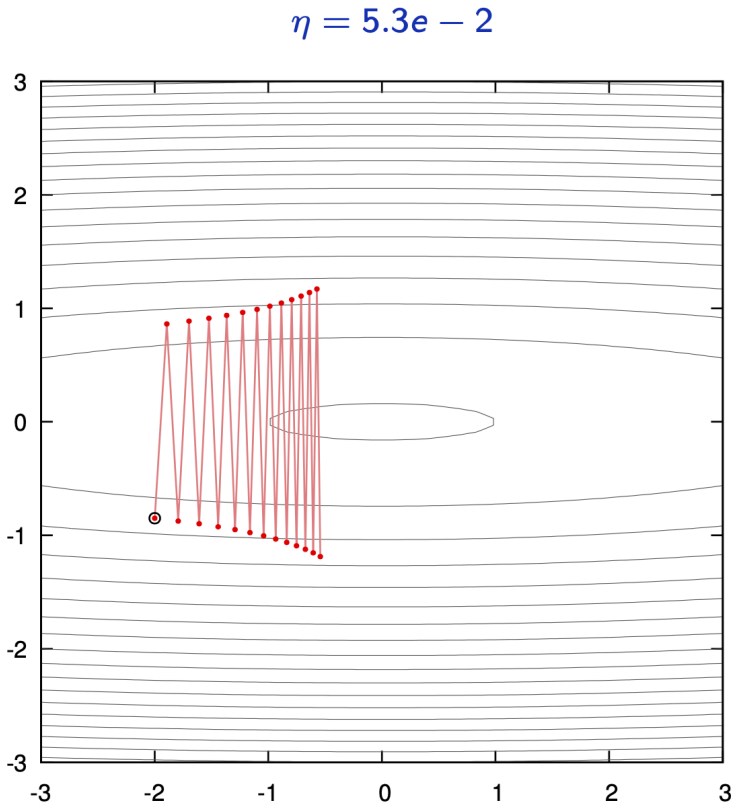


Figure by François Fleuret

SGD Extensions

- Momentum:
  - Add inertia to the parameter updates

$$\nu \leftarrow \gamma\nu + \eta \nabla R(\mathbf{W})$$

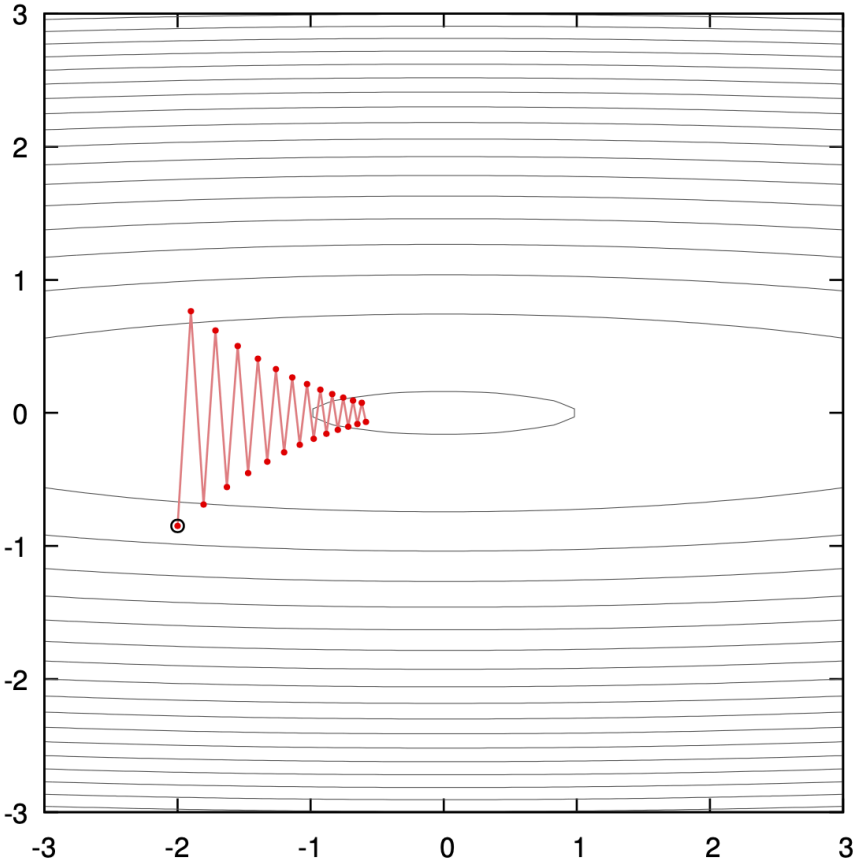
$$\mathbf{W} \leftarrow \mathbf{W} - \nu$$

- With $\gamma > 0$, momentum
  - Accelerates if the gradient does not change too much
  - Dampens the oscillations in narrow valleys

Momentum

Without momentum

$\eta = 5.0e-2, \gamma = 0$



With momentum

$\eta = 5.0e-2, \gamma = 0.5$

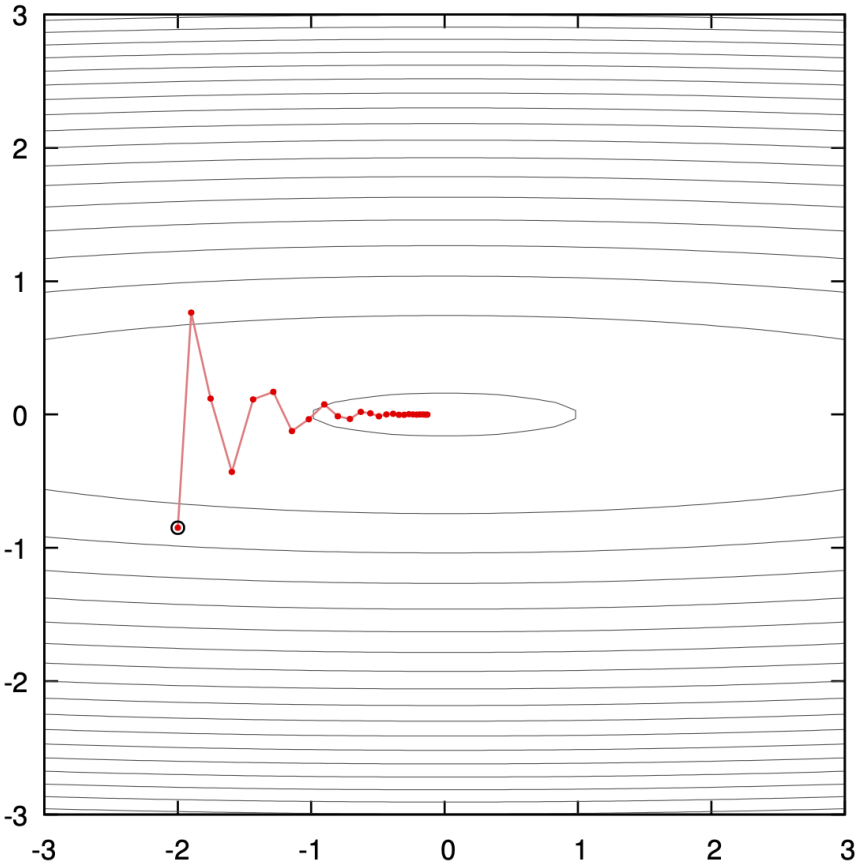


Figure by François Fleuret

SGD Extensions

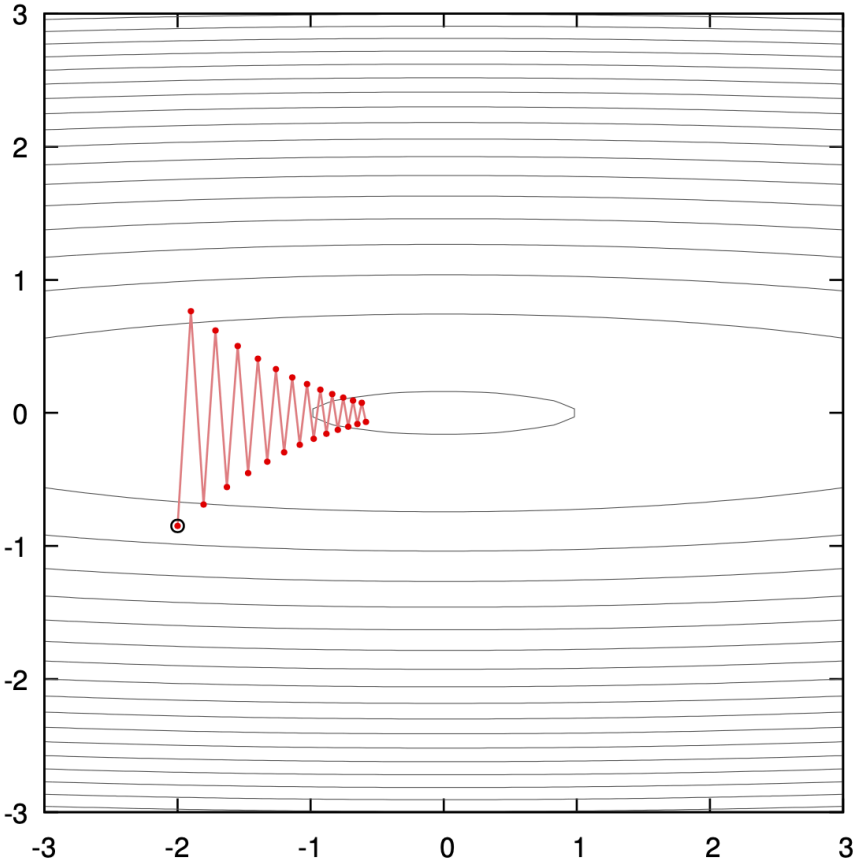
- Adaptive learning rate:
  - Adagrad (Duchi et al., JMLR 2011)
  - Adadelta (Zeiler, 2012)
  - RMSprop (Hinton, 2012)
  - ADAM (Kingma et al., ICLR 2015)

- For example, ADAM uses a moving average of each coordinate to rescale each coordinate separately

Momentum

Without ADAM

$\eta = 5.0e-2, \gamma = 0$



With ADAM

ADAM has several hyper-parameters that I will not list here

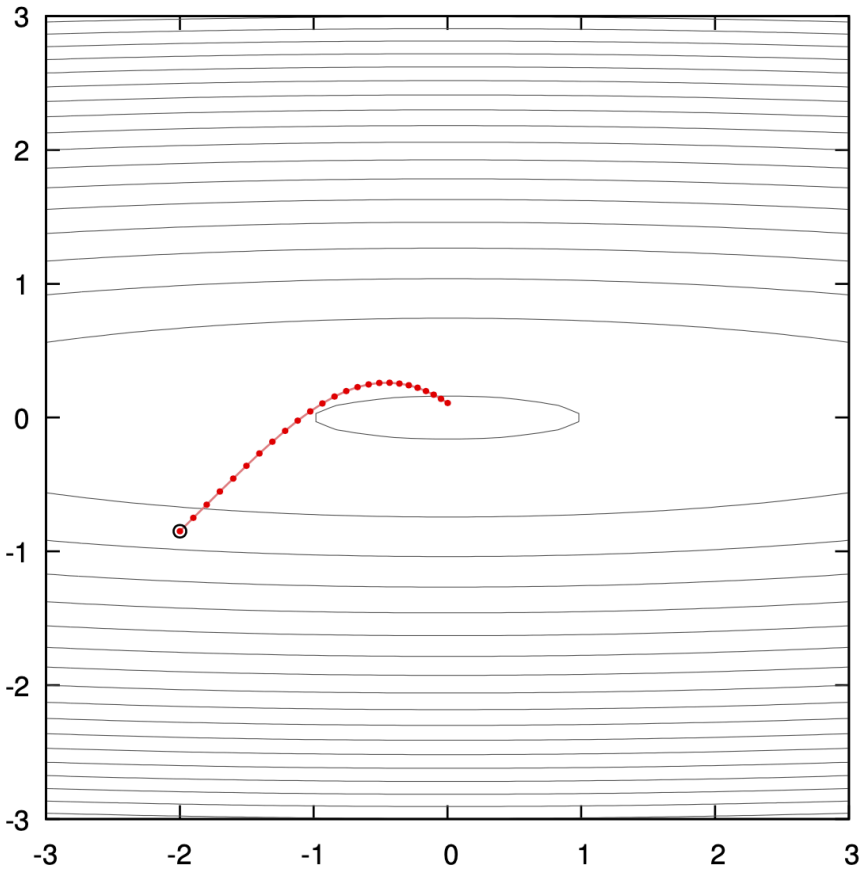
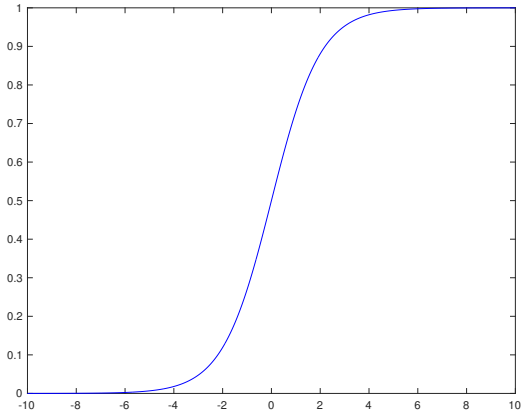


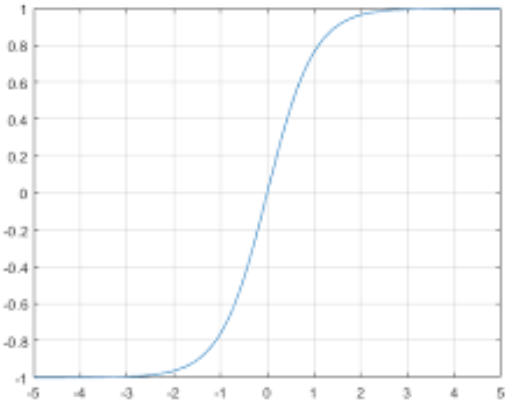
Figure by François Fleuret

Gradient vanishing & initialization

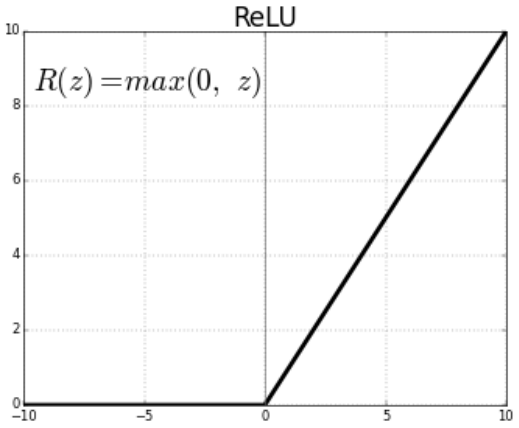
- The activation functions often have saturating regions where the gradient becomes (close to) 0



Sigmoid



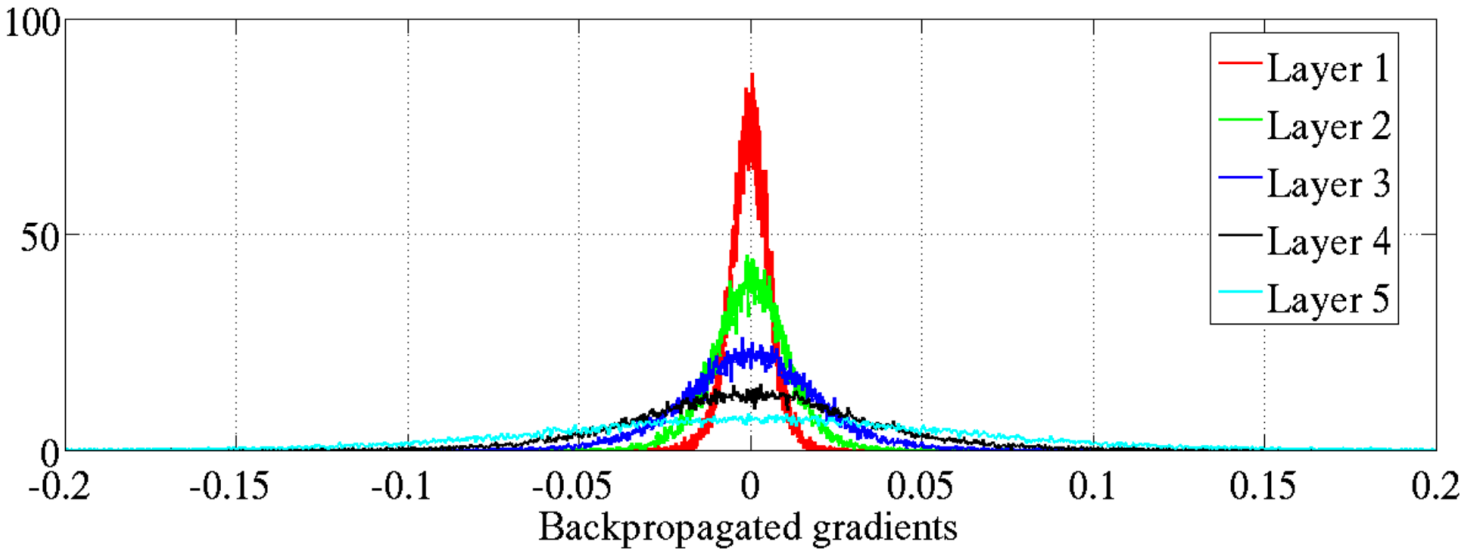
tanh



ReLU

Gradient vanishing & initialization

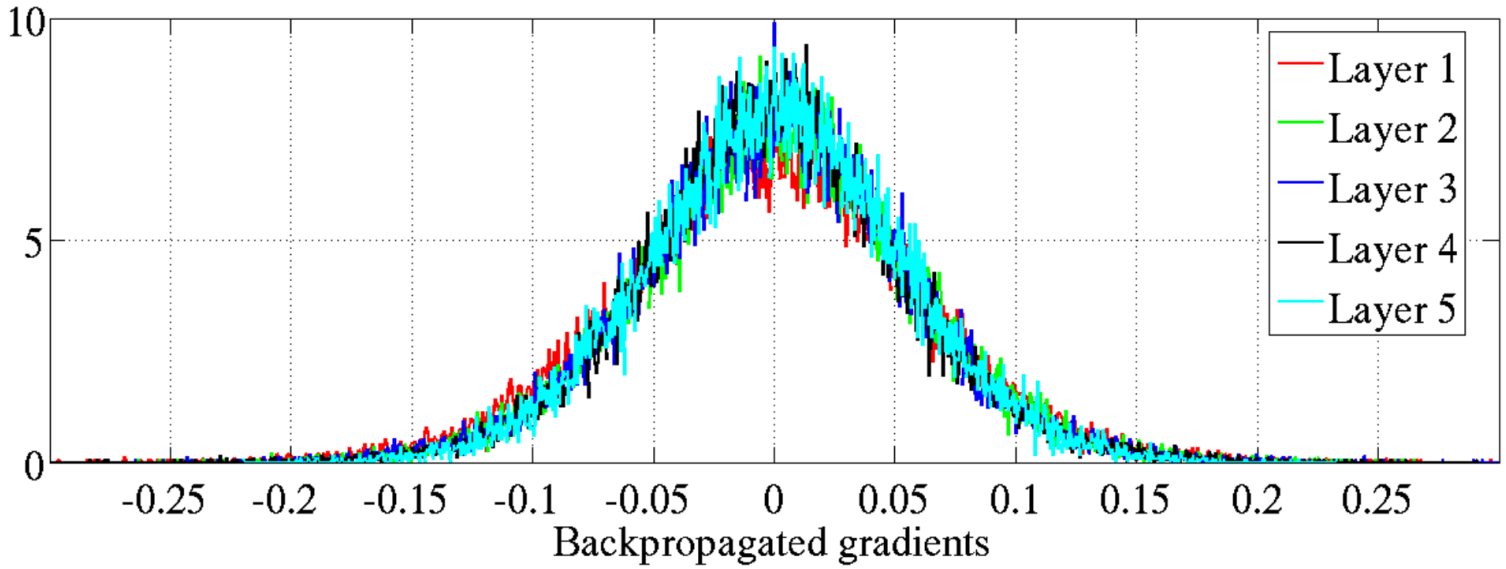
- Glorot & Bengio observed that the gradient vanishes exponentially with the (inverse) depth if the weights are such that they tend to often saturate the activation functions
  - X. Glorot & Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, AISTATS, 2010



Histograms of the gradient values

Gradient vanishing & initialization

- To overcome this, Glorot & Bengio introduced a weight initialization strategy where the range of the initial, random values of the weights is layer dependent
  - This is referred to as Xavier (or Glorot) initialization



Histograms of the gradient values

Regularization

- Deep networks have many parameters and can therefore overfit
  - In practice, this seems to happen surprisingly rarely
- One strategy to address this is called Dropout (Srivastava et al., JMLR 2014):
  - Randomly remove units and the connections during training
- Based on what we have learned so far, can you think of two other strategies to prevent overfitting?

MLP regression demo

- http://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-gauss&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4&seed=0.44368&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=regression&initZero=false&hideText=false

From regression to classification

- So far, we have focused on the case where the prediction is taken directly as the output of a linear model applied to the last hidden representation, i.e.,

$$\hat{\mathbf{y}} = \mathbf{W}_{(L)}^T \mathbf{z}_{(L-1)}$$

- While this is fine for a regression task, it is not ideal for a classification one, where we would like each $\hat{y}^{(k)}$ to represent the probability (score) for class k

- This can be achieved via the softmax function (as in multi-class logistic regression)

$$\hat{y}^{(k)} = \frac{\exp(\mathbf{w}_{(L)(k)}^T \mathbf{z}_{(L-1)})}{\sum_{j=1}^{C} \exp(\mathbf{w}_{(L)(j)}^T \mathbf{z}_{(L-1)})}$$

where $\mathbf{w}_{(L)(j)}$ indicates the j^{th} column of $\mathbf{W}_{(L)}$

From regression to classification

- In this case, the empirical risk is typically defined as the cross-entropy

$$R(\mathbf{W}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} y_i^{(k)} \ln \hat{y}_i^{(k)}$$

- Backpropagation then requires computing the derivative of the cross-entropy w.r.t. each $\hat{y}^{(k)}$ and the derivative of the softmax function w.r.t. each $W_{(L)}^{(j,k)}$
  - Note that this is similar to the derivatives computed for multi-class logistic regression

MLP classification demos

- http://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4&seed=0.20666&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

- https://adamharley.com/nn_vis/mlp/2d.html

Working with images

- Treating an image as a big vector seems counter-intuitive
  - An image of a reasonable size yields a huge vector, thus a huge amount of parameters in an MLP
  - Small image regions have similar property and should not be processed independently
  - A translation should not affect the results

$$\mathbf{x}_i \in \mathbb{R}^{536 \cdot 356 \cdot 3} = \mathbb{R}^{572448}$$

- Next week, we will see how we can reduce the number of parameters and improve invariance