Deep Learning for NLP

Session 3: Feed Forward Networks

Sharid Loáiciga — May 5th, 2020

Recap

- $y = X\beta + \beta 0$ — linear regression
- $P(Y|X) = \dfrac{1}{1 + e^{-(\beta X + \beta_0)}}$ — logistic classification
- $lo(Y|X) = \beta X + \beta_0$ — equivalent (logit) form
- $H(X|Y) = \sum_{x,y} p(x,y) log_2(\dfrac{1}{p(x|y)})$ — cross-entropy loss: measures “distance" between predictions and gold standard
- SGD — algorithm to evaluate the loss function wrt the parameters of the model

Recap
SGD

- Find the slope of the loss function with respect to each parameter/feature. In other words, compute the gradient of the function.
- Pick a random initial value for the parameters.
- Update the gradient function by plugging in the parameter values.
- Calculate the step sizes for each feature as : step size = gradient * learning rate.
- Calculate the new parameters as: new params = old params - step size
- Repeat steps 3 to 5 until gradient is almost 0.

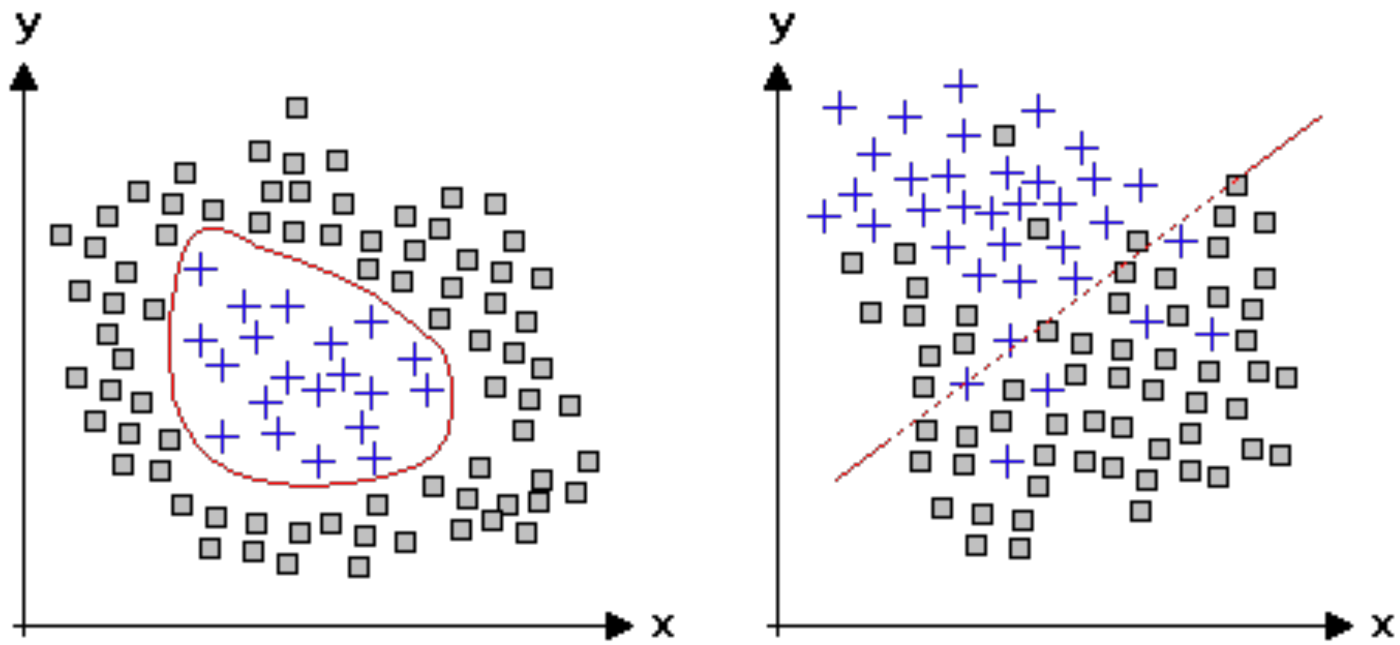
https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

Slides 3.b. by
Mike Silfverberg & Hande Celikkanat,
University of Helsinki

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Feed-forward networks

Many Datasets are not Linearly Separable

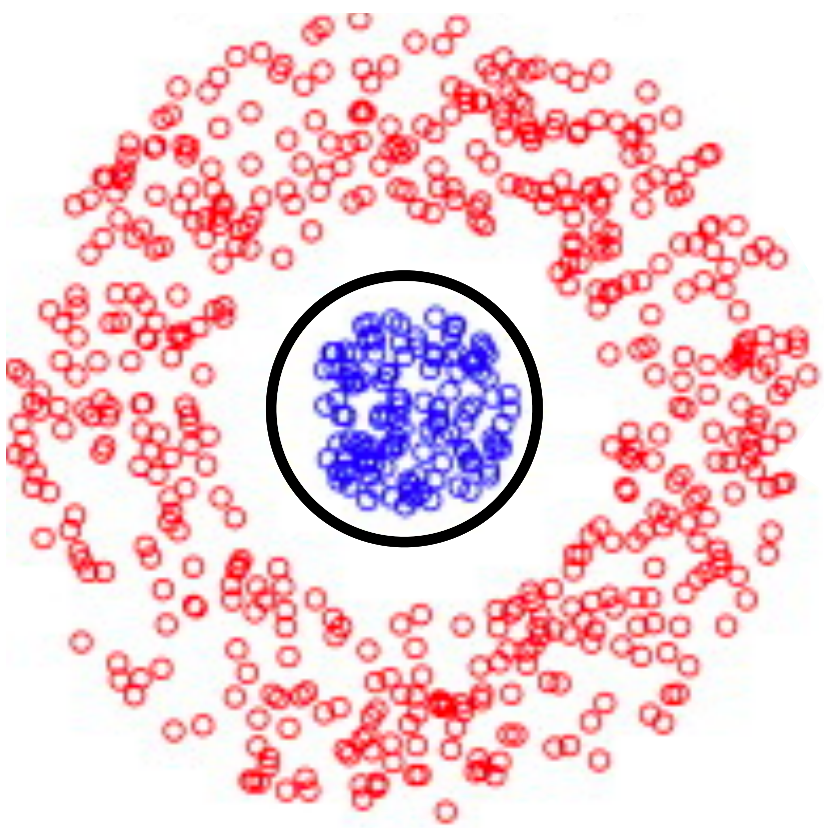


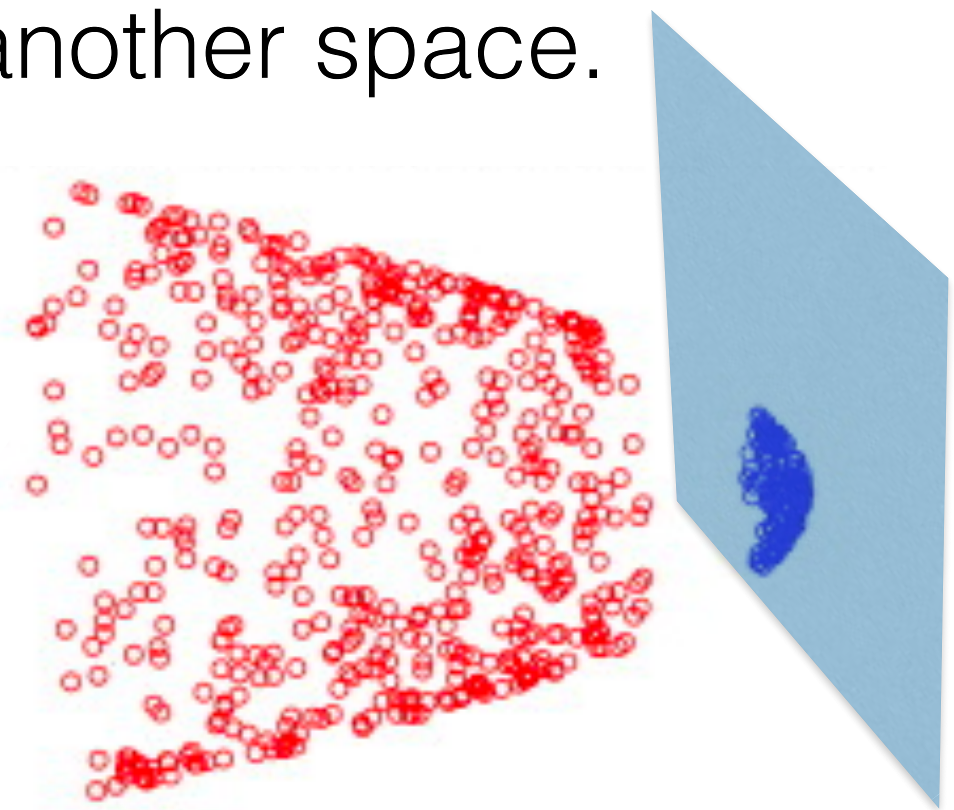
Sometimes the data is simply noisy and there's no linear decision boundary but a linear classifier will still work quite well.

Sometimes the appropriate decision boundary is simply not linear.

Often We Can Make the Problem Linearly Separable

A dataset which is not linearly separable may become linearly separable in another space.

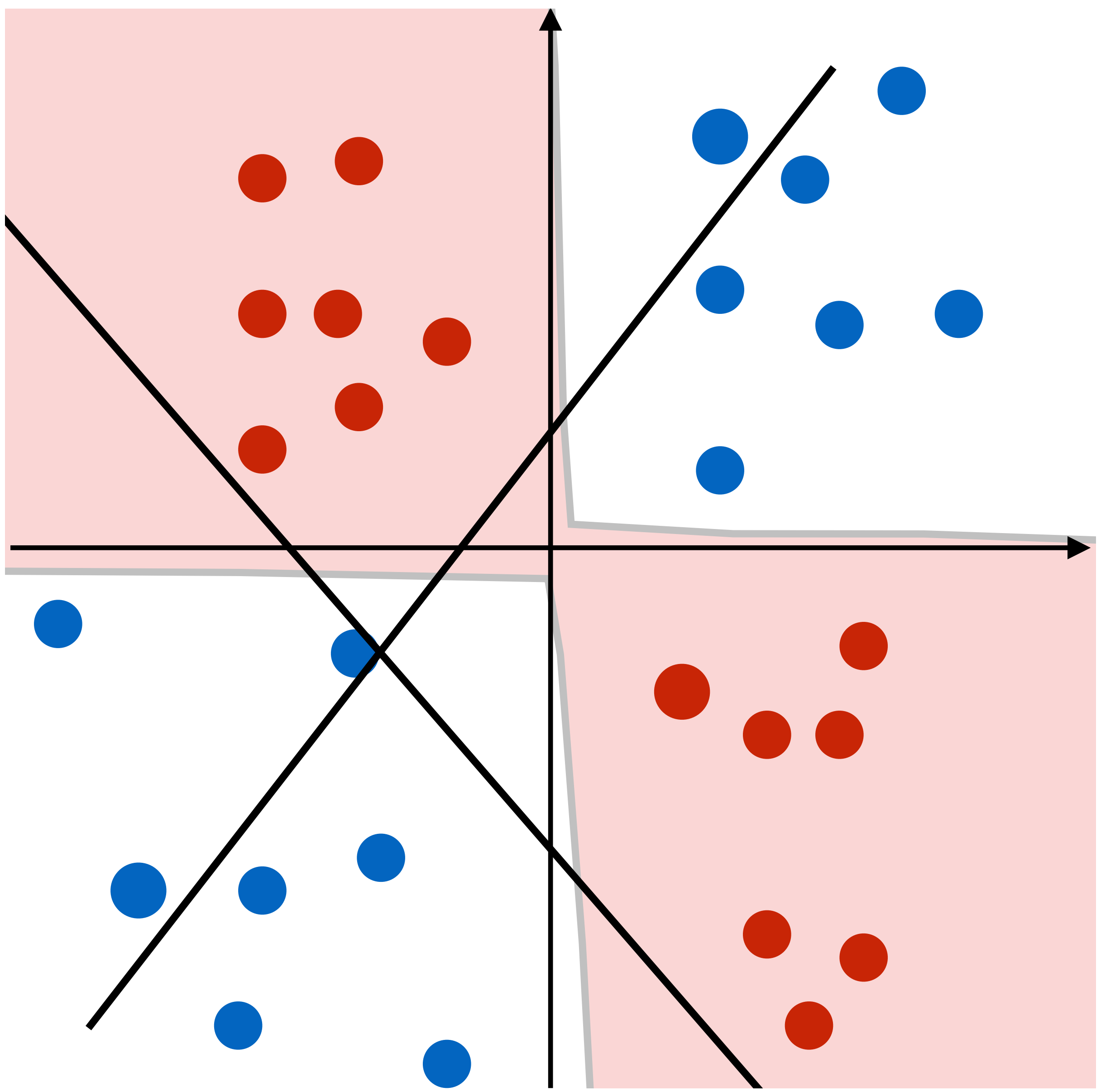




$$(x_1, y_2) \mapsto (x_1, x_2, x_1^2 + x_2^2)$$

Exclusive OR (XOR)

A classical example of a non-linear problem is XOR

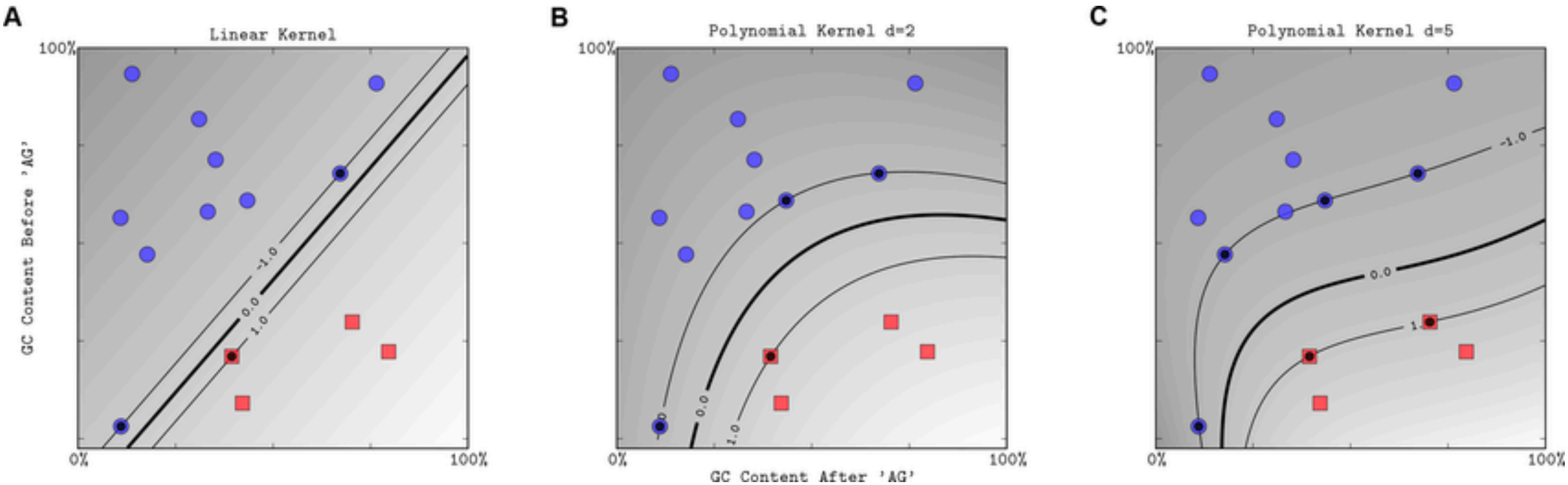


	x1	x2
-	-1	-1
+	-1	1
+	1	-1
-	1	1

Kernel Methods

Polynomial kernel of degree 2 from 2D to 3D space:

$$(x_1, x_2) \mapsto (x_1 \cdot x_1, x_1 \cdot x_2, x_2 \cdot x_2)$$



Kernel methods can define non-linear decision boundaries in the original space.

Problem with Kernel Methods

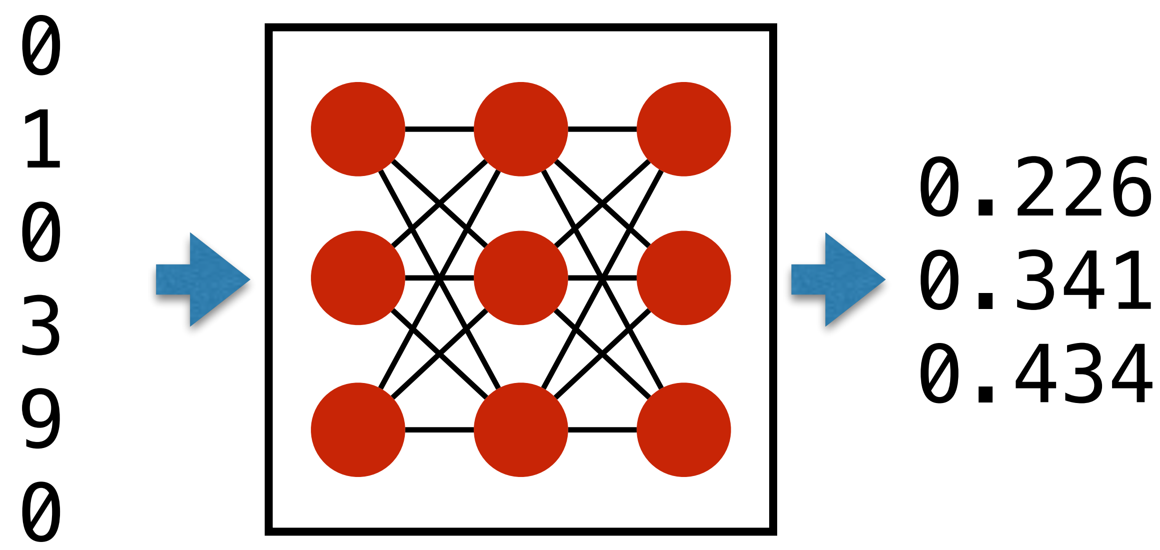
Polynomial kernel from 3D to 6D space:

$$(x_1, x_2, x_3) \mapsto (x_1 \cdot x_1, x_1 \cdot x_2, x_1 \cdot x_3, x_2 \cdot x_2, x_2 \cdot x_3, x_3 \cdot x_3)$$

Polynomial kernel from 50000D space (typical size of a vocabulary):
199,990,000 dimensions

The number of dimensions grows very fast.
The problem becomes linear but there is severe danger of over-fitting because of high dimensionality.

Feed-Forward Neural Networks

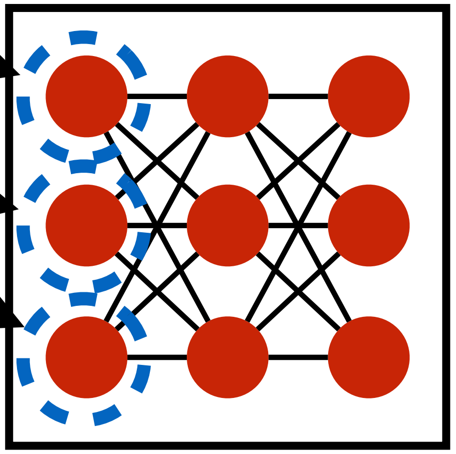


General feed-forward networks can have multiple layers.

Defining a Network Using Vectors

0
1
0
3
9
0

- [0.365, 0.482, 0.156, 0.294, 0.339, 0.053] = 4.41
- [-0.121, 0.013, 0.751, 0.880, -1.401, 0.001] = -9.95
- [1.561, -0.045, -0.651, 0.02, 0.522, -0.511] = 4.71



What is Deep Learning?

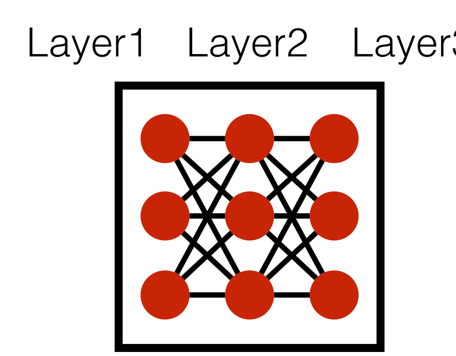
Input for the next layer

tanh(x)

Input dimension = 6

```
0   • [0.365, 0.482, 0.156, 0.294, 0.339, 0.053] = 4.41      tanh(4.41) =   0.9997
1
0   • [-0.121, 0.013, 0.751, 0.880, -1.401, 0.001] = -9.95   tanh(-9.95)= -1.0000
3
9
0   • [1.561, -0.045, -0.651, 0.02, 0.522, -0.511] = 4.71    tanh(4.71) =   0.9997
```

An **activation function** like hyperbolical tangent is applied to the outputs of the dot products.



Parameter vectors in layer 2 have dimension 3.

Defining a Network Using Matrices

$$\vec{\mathbf{x}}\mathbf{W}^1 + \vec{\mathbf{b}}^1$$

```
w1 = [0.365, 0.482, 0.156, 0.294, 0.339, 0.053]
w2 = [-0.121, 0.013, 0.751, 0.880, -1.401, 0.001]
w3 = [1.561, -0.045, -0.651, 0.02, 0.522, -0.511]
```

$$[0\ 1\ 0\ 3\ 9\ 0] \times \begin{bmatrix} 0.365 & -0.121 & 1.561 \\ 0.482 & 0.013 & 0.045 \\ 0.156 & 0.751 & -0.651 \\ 0.294 & 0.880 & 0.020 \\ 0.339 & -1.401 & 0.522 \\ 0.053 & 0.001 & -0.511 \end{bmatrix} + [1.1\ 3.0\ -0.5] = [5.51\ -6.95\ 4.21]$$

$\vec{\mathbf{x}}$ - input vector

$\mathbf{W}^1$ - weight matrix in layer 1

$\vec{\mathbf{b}}^1$ - bias term in layer 1

Defining a Network using Matrices

2-layer feed-forward network (multi-layer perceptron) with **one hidden layer**:

$$\mathrm{NN}_{\mathrm{MLP1}}(\boldsymbol{x}) = g(\vec{\boldsymbol{x}}\boldsymbol{W}^{\mathbf{1}} + \vec{\boldsymbol{b}}^{\mathbf{1}})\boldsymbol{W}^{\mathbf{2}} + \vec{\boldsymbol{b}}^{\mathbf{2}}$$

$$\vec{\boldsymbol{x}} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W}^{\mathbf{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \vec{\boldsymbol{b}}^{\mathbf{1}} \in \mathbb{R}^{d_1}, \quad \boldsymbol{W}^{\mathbf{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \vec{\boldsymbol{b}}^{\mathbf{2}} \in \mathbb{R}^{d_2}.$$

$\mathbf{W}^i$ - weight matrix in layer i

$\vec{\mathbf{b}}^i$ - bias term in layer i

g - non-linearity

`g is a pointwise non-linear function:`

$$g(\vec{\mathbf{y}}) = [g(y_1), \ldots, g(y_{d_1})]$$

How Do We Apply a Network?

$$\mathrm{NN}_{\mathrm{MLP1}}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W}^{\mathbf{1}} + \boldsymbol{b}^{\mathbf{1}})\boldsymbol{W}^{\mathbf{2}} + \boldsymbol{b}^{\mathbf{2}}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W}^{\mathbf{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \boldsymbol{b}^{\mathbf{1}} \in \mathbb{R}^{d_1}, \quad \boldsymbol{W}^{\mathbf{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \boldsymbol{b}^{\mathbf{2}} \in \mathbb{R}^{d_2}.$$

1. Compute $\vec{\mathbf{x}}\mathbf{W}^1$ and add $\mathbf{b}^1$ to the result.
2. Take the result $\vec{\mathbf{y}} = [y_1, \ldots, y_{d_1}]$ and apply non-linearity:

$$g(\vec{\mathbf{y}}) = [g(y_1), \ldots, g(y_{d_1})]$$

3. Now take the output of layer 1 $g(\vec{\mathbf{y}})$ and compute $g(\vec{\mathbf{y}})\mathbf{W}^2$ and again add the bias term $\vec{\mathbf{b}}^2$

If the network is a classifier, we often have a final softmax.

How Do We Train?

$$\mathrm{NN}_{\mathrm{MLP1}}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{1}} + \boldsymbol{b}^{\boldsymbol{1}})\boldsymbol{W}^{\boldsymbol{2}} + \boldsymbol{b}^{\boldsymbol{2}}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W}^{\boldsymbol{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \boldsymbol{b}^{\boldsymbol{1}} \in \mathbb{R}^{d_1}, \quad \boldsymbol{W}^{\boldsymbol{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \boldsymbol{b}^{\boldsymbol{2}} \in \mathbb{R}^{d_2}.$$

We train networks using stochastic gradient descent (SGD)

First, we need to choose a loss L (quite often cross-entropy)

We need to compute partial the derivatives:

$$\frac{\partial L}{\partial w} \quad \frac{\partial L}{\partial b}$$

w.r.t. all weight parameters w and bias parameters b

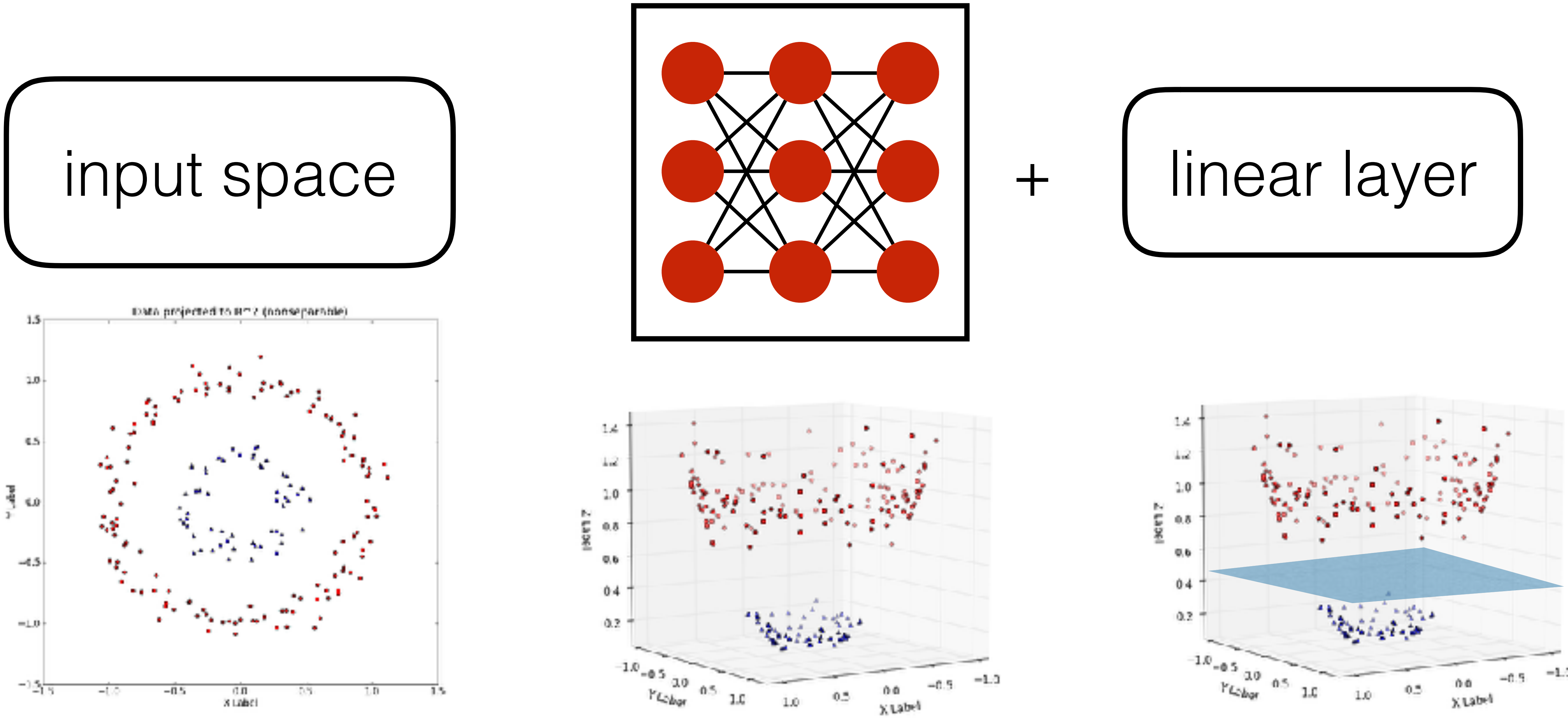
Problems with Naive SGD

We end up computing a lot of quantities again and again…

The solution is to use a dynamic programming algorithm called back propagation for computing the derivatives recursively.

We'll get to training networks in week 3

NN = Feature Extractor + Classifier



The input dataset is not linearly separable, our NN makes it separable and then a final linear layer learns a classifier.

Why Do We Need a Non-Linearity?

If you combine a number of linear layers,
you end up with a linear layer:

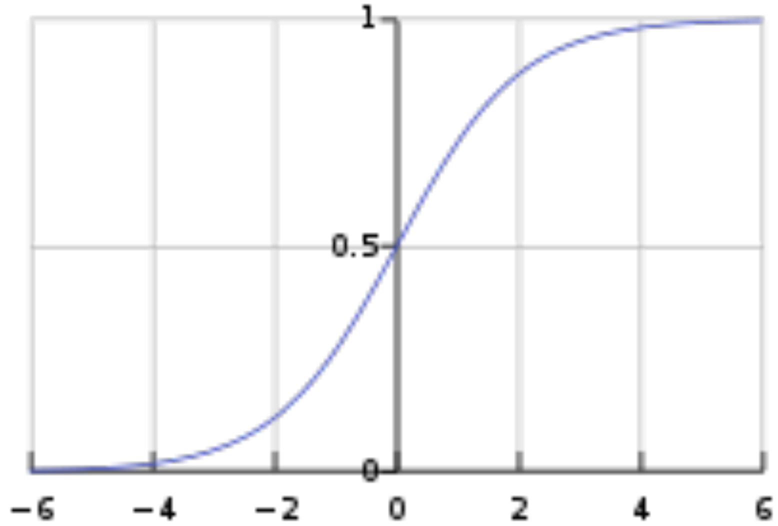
$$((xW_1 + b_1)W_2 + b_2)W_3 + b_3 = x(W_1W_2W_3) + (b_1W_2W_3 + b_2W_3 + b_3)$$

$$=$$

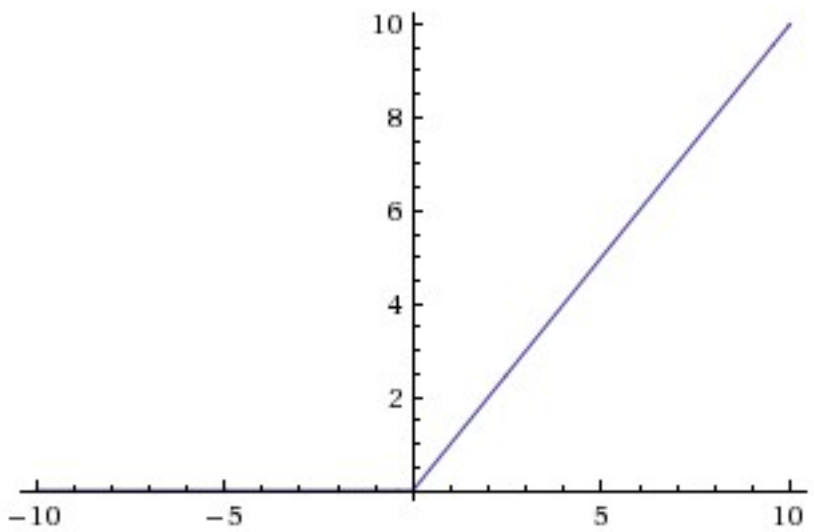
$$xW_4 + b_4$$

You cannot go beyond the expressive power of a single linear layer without introducing non-linearities

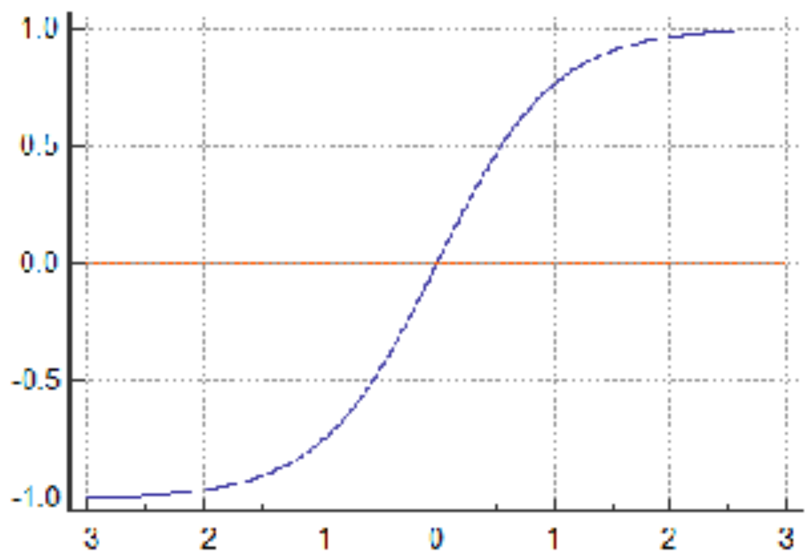
Common Non-Linearities



sigmoid



ReLU



tanh

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

A Concrete Example — XOR using ReLU

Worked Out Example: ReLU for XOR

XOR	x1	x2
-1	-1	-1
1	-1	1
1	1	-1
-1	-1	-1

$$\mathbf{W}^1 = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \quad \mathbf{W}^2 = \begin{bmatrix} -0.5 \\ 0.5 \\ 0.5 \\ -0.5 \end{bmatrix}$$

$$\vec{\mathbf{b}}^1 = [0\ 0\ 0\ 0]$$

$\vec{\mathbf{x}}$	$\vec{\mathbf{x}}\mathbf{W}^1 + \vec{\mathbf{b}}^1$	$\mathrm{ReLU}(\vec{\mathbf{x}}\mathbf{W}^1 + \vec{\mathbf{b}}^1)$	$\mathbf{W}^2\mathrm{ReLU}(\vec{\mathbf{x}}\mathbf{W}^1 + \vec{\mathbf{b}}^1)$
$[-1\ -1] \mapsto$		$\mapsto [2\ 0\ 0\ 0]$	$\mapsto -1$
$[-1\ 1] \mapsto$		$\mapsto [0\ 0\ 2\ 0]$	$\mapsto 1$
$[1\ -1] \mapsto$		$\mapsto [0\ 2\ 0\ 0]$	$\mapsto 1$
$[1\ 1] \mapsto$		$\mapsto [0\ 0\ 0\ 2]$	$\mapsto -1$

XOR implementation:

https://github.com/joosthub/PyTorchNLPBook/tree/master/chapters/chapter_4