# Introduction to Learning Deep Representations

Vlado Menkovski

Eindhoven University of Technology

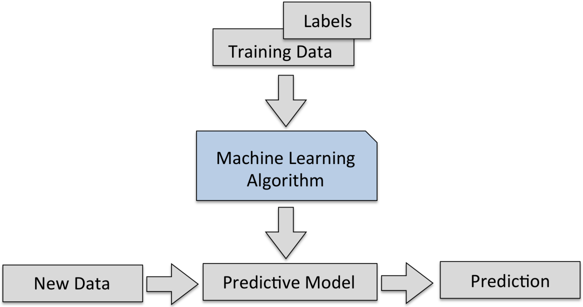
*v.menkovski@tue.nl*

March 22, 2018

Introduction to Learning Deep Representations

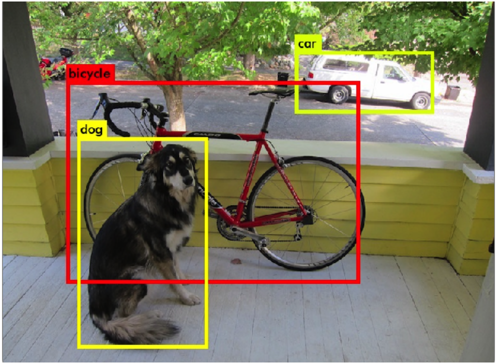
# Overview

# Supervised learning

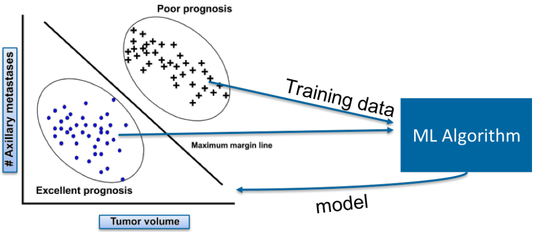


# Modeling speech

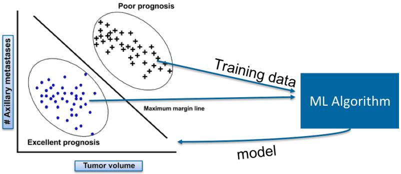
# Image processing - localization



# Deep Learning - motivation

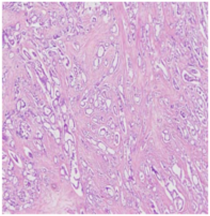


# Deep Learning - motivation

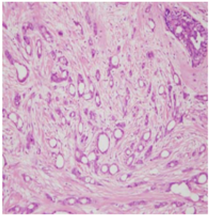



| Tumor volume | # Metastasis | Prognosis |
|---|---|---|
| 3 | 0 | good |
| 4 | 1 | good |
| 12 | 4 | not good |

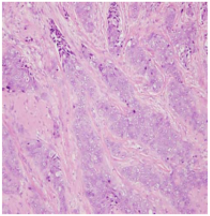
# Supervised learning - high dimensional data



**Tubule score = 1**



**Tubule score = 2**



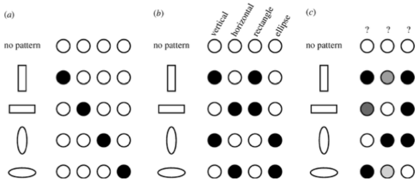
**Tubule score = 3**

| X0_0 | X01_ | X0_2 | ... | X1_0 | X1_1 | ... | ... | X512_511 | x_512_512 | Tubule score |
|---|---|---|---|---|---|---|---|---|---|---|
| (123, 223,075) | (123, 223,075) | (123, 223,075) | (123, 223,075) | (123, 223,075) | (123, 223,075) | (123, 223,075) | (123, 223,075) | ... | ... | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | 2 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | 2 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | 3 |

**262144**

# Distributed representation

Neural network representations: (a) localist, (b) semilocalist or feature-based and (c) distributed.




Kevin Gurney Phil. Trans. R. Soc. B 2007;362:339-353

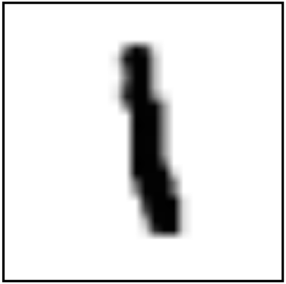


[1] *Allows for compositionality*

[1] Hinton, Geoffrey E., James L. McClelland, and David E. Rumelhart. "Distributed representations." Parallel distributed processing: Explorations in the microstructure of cognition 1.3 (1986): 77-109.

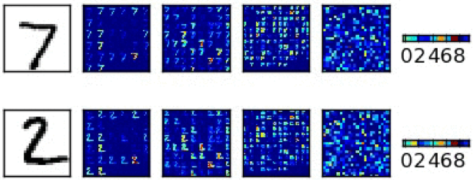
# Image data

# Image data

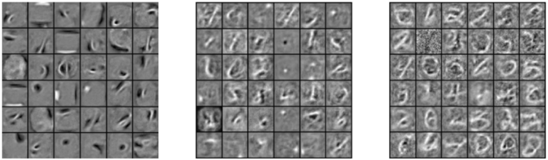


$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

# Neural network processing images



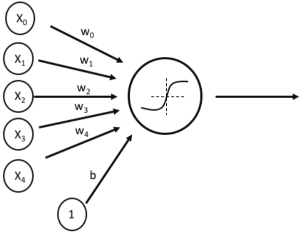
# Sensitivity of neurons to input



# Artificial Neuron

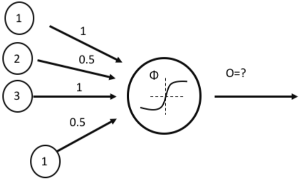
- The input to the neuron is defined by $x_0$ to $x_4$, noted as input vector $\mathbf{x}$
- Edges between nodes have parameters $\mathbf{w}$ and a bias term $b$
- A single neuron implements:

$$o(x;\theta) = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^T \mathbf{x})$$

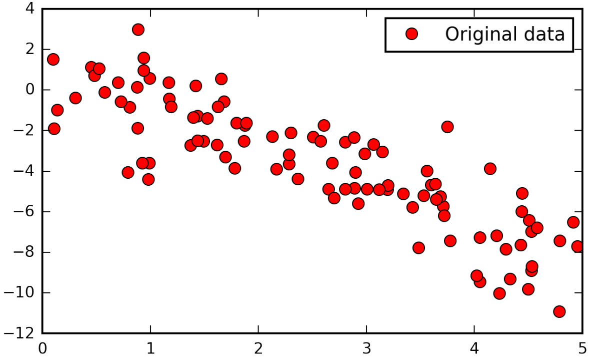


# Artificial Neuron

- If the input to the model is a vector $\mathbf{x} = [1, 2, 3]^\top$
- parameterized by $\mathbf{w} = [1, 0.5, 1]^\top, b = 0.5$
- Computes the following expression:
- $o = \phi(\mathbf{w}^\top \mathbf{x}) = \phi(1*1+2*0.5+3*1+0.5) = \phi(1+1+3+0.5) = \phi(5.5)$
- $\phi$ can be many different functions.
- E.g. $\phi(x) = x$
  - In which case: $o = \phi(5.5) = 5.5$



# Linear Regression




*How can we use a neuron to model this data?*

# Empirical Risk minimization

$$\arg\min_{\theta} \frac{1}{N} \sum_{i=0}^{N} L\left( f(\mathbf{x}^{(i)}; \theta), y^{(i)} \right)$$

# Empirical Risk minimization

$$\arg\min_{\theta} \frac{1}{N} \sum_{i=0}^{N} L\left(f(\mathbf{x}^{(i)}; \theta), y^{(i)}\right)$$

$$\arg\min_{\theta} \frac{1}{N} \sum_{i=0}^{N} L\left(o_{\theta}(\mathbf{x}^{(i)}), y^{(i)}\right)$$

# Gradient Descent Optimization

Empirical Risk minimization

$$\arg\min_{\theta} \frac{1}{N} \sum_{i=0}^{N} L\left(f(\mathbf{x}^{(i)}; \theta), y^{(i)}\right)$$

$$\arg\min_{\theta} \frac{1}{N} \sum_{i=0}^{N} L\left(o_{\theta}(\mathbf{x}^{(i)}), y^{(i)}\right)$$

- Given a set of $n$ examples $\{(\mathbf{x}, y)\}$.
- GD Update rule:
* repeat until convergence

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} L(\mathbf{x}; \theta)$$

# Gradient Descent Optimization

- Given a set of $n$ examples $\{(\mathbf{x}, y)\}$.
- GD Update rule:
  - repeat until convergence

$$w \leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b)$$

$$b \leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b)$$

$\alpha$ - learning rate

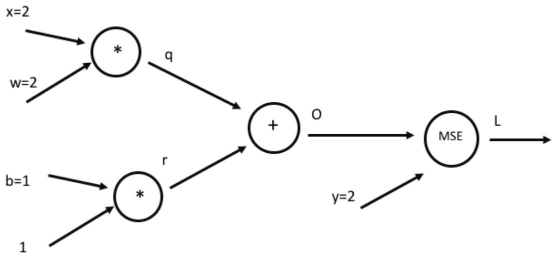
# Gradient Descent Optimization

*Requirements:*

- Model:
  - $o_\theta = \mathbf{w}^\top x$
  - $\theta : \{\mathbf{w}, b\}$
- Loss function:
  - $L(\mathbf{x}, y; \mathbf{w}, b) = \frac{1}{2n} \sum_{i=0}^{n-1} (o_\theta - y)^2$
- Gradient of $L$ wrt $\mathbf{w}$ and $b$:
  - $\frac{\partial}{\partial w} L(.)$
  - $\frac{\partial}{\partial b} L(.)$

# Computing the gradient of the loss

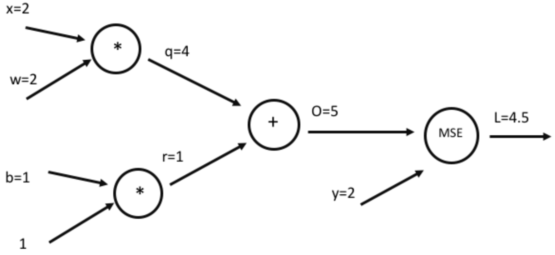
*Compute graph* - For a single data-point $\{x = 2, y = 2\}$



# Computing the gradient of the loss

*Compute graph (forward pass)*

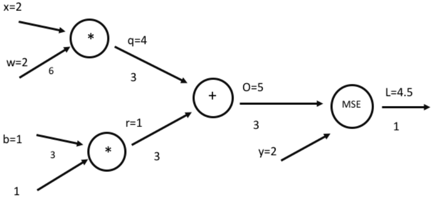
- $o = wx + b = 2 * 2 + 1 = 5$
- $L = \frac{1}{2}(o - y)^2 = \frac{1}{2}(5 - 2)^2 = 0.5 * 3^2 = 4.5$



# Computing the gradient of the loss

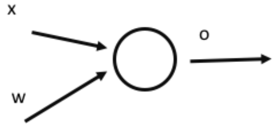
*Compute graph (backward pass)*

$$\frac{\partial L}{\partial L} = 1$$
$$\frac{\partial L}{\partial o} = \frac{2(o-y)}{2} * 1 = (o-y)$$
$$\frac{\partial L}{\partial q} = \frac{\partial o}{\partial q}\frac{\partial L}{\partial o} = 1 * (o-y)$$
$$\frac{\partial L}{\partial r} = \frac{\partial o}{\partial r}\frac{\partial L}{\partial o} = 1 * (o-y)$$
$$\frac{\partial L}{\partial w} = \frac{\partial q}{\partial w}\frac{\partial o}{\partial q} = x * (o-y)$$
$$\frac{\partial L}{\partial b} = \frac{\partial r}{\partial b}\frac{\partial o}{\partial r} = 1 * (o-y)$$



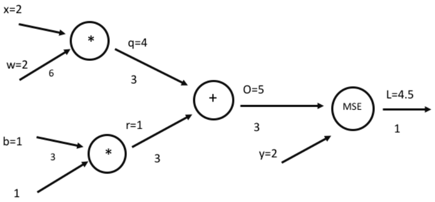
# Backpropagation

- Compute forward pass
  - $o = x(*)w$
- For each node compute the local derivative
  - $\frac{\partial o}{\partial x}$
  - $\frac{\partial o}{\partial w}$
- Backward pass the derivative
  - apply the chain rule



# Computing the gradient of the loss

*Compute graph (backward pass)*




$$w \leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b)$$

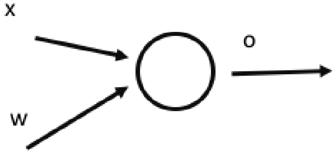
$$w \leftarrow 2 - 0.1 * 6$$

$$b \leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b)$$

$$b \leftarrow 1 - 0.1 * 3$$

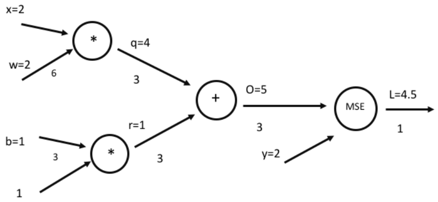
# Backpropagation

- Compute forward pass
  - $o = x(*)w$
- For each node compute the local derivative
  - $\frac{\partial o}{\partial x}$
  - $\frac{\partial o}{\partial w}$
- Backward pass the derivative
  - apply the chain rule



# Computing the gradient of the loss

*Compute graph (backward pass)*




$$w \leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b)$$

$$w \leftarrow 2 - 0.1 * 6$$

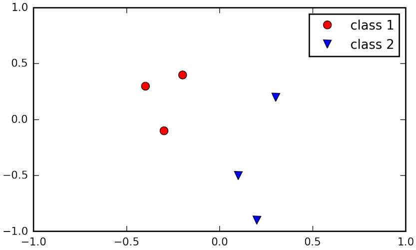
$$b \leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b)$$

$$b \leftarrow 1 - 0.1 * 3$$

# Tensorflow implementation

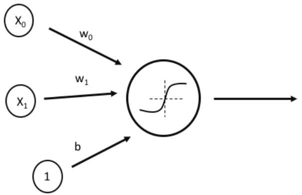
(browser)

# Classification




- Input: $x_0$, $x_1$
- Output is $f_c(\mathbf{x}) = P(y = c|\mathbf{x})$

# Artificial Neuron Classification



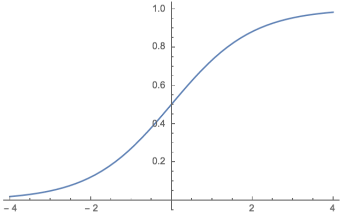

- Input: by $x_0$, $x_1$, vector notation $\mathbf{x}$
- Edge parameters: $\mathbf{w}$ + bias term $b$
- A single neuron implements:

$$o(x;\theta) = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^\top \mathbf{x})$$

$$\phi(\mathbf{x}) = logit(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

- Bounded from 0 to 1. Smooth, positive function.

# Artificial Neuron Classification




- Input: by $x_0$, $x_1$, vector notation $\mathbf{x}$
- Edge parameters: $\mathbf{w}$ + bias term $b$
- A single neuron implements:

$$o(x;\theta) = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^\top \mathbf{x})$$

$$\phi(\mathbf{x}) = logit(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

- Bounded from 0 to 1. Smooth, positive function.

# Binary classification

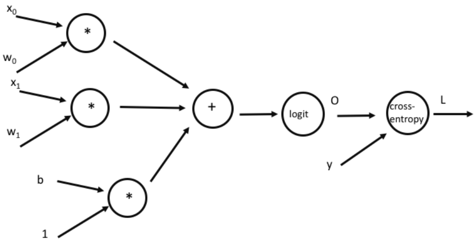
- Output is $o_c(\mathbf{x}) = P(y = c|\mathbf{x})$
- $o_1(\mathbf{x}) = P(y = 1|\mathbf{x}) = \phi(\mathbf{w}^\top \mathbf{x}) = logit(\mathbf{w} \top \mathbf{x})$
- $o_0 = (1 - o_1)$

*Gradient Descent Optimization:*

- Model:
  - $o_\theta = logit(\mathbf{w}^\top x + b) = \frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}+b}}$
  - $\theta : \{\mathbf{w}, b\}$
- Loss function:
  - $L(\mathbf{x}, y; \theta) = -\sum_c 1_{(y=c)} \log \; o_c = -log \; o_y$
  - Log for numerical stability and math simplicity
- Gradient of $L$ wrt $\mathbf{w}$ and $b$:
  - $\frac{\partial}{\partial w} L(.)$ for both $w_0$ and $w_1$
  - $\frac{\partial}{\partial b} L(.)$

# Compute Graph

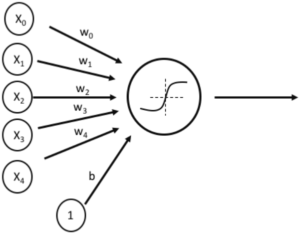
- $\frac{\partial L}{\partial L} = 1$
- $\frac{\partial L}{\partial o} = \frac{\partial - \log\ o}{\partial o} = -\frac{1}{o}$
- $\frac{\partial o_\theta(\mathbf{x})}{\partial \theta} = \frac{\partial logit(\mathbf{x};\theta)}{\partial \theta} = (1 - logit(\mathbf{x};\theta))(logit(\mathbf{x};\theta))$



# Perceptron

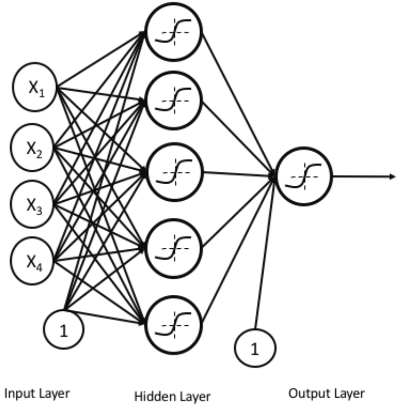
(browser)

# Artificial Neuron



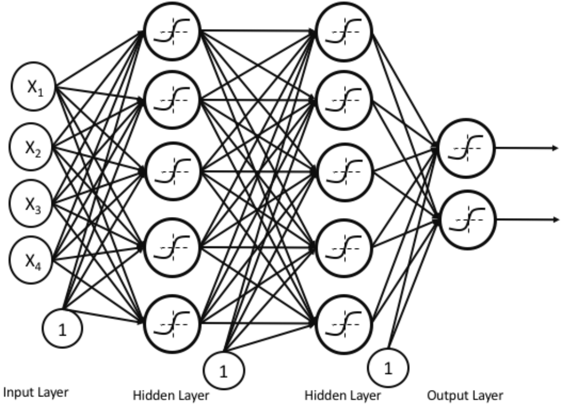
# Multilayered Perceptron

*Can be stacked*



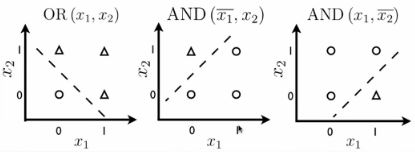
# Multilayered Perceptron

*Multiple layers of stacked neurons*

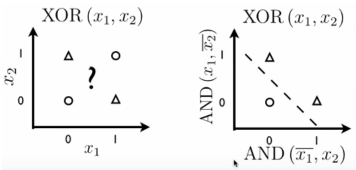


# Multilayered Perceptron

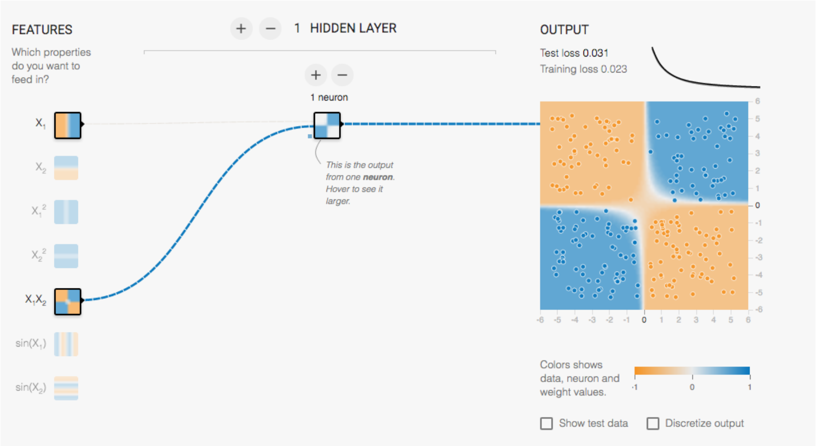
*Motivation* - Compositional features



# Multilayered Perceptron

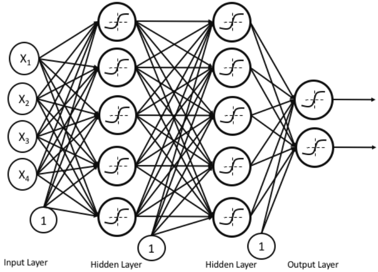


# Multilayered Perceptron



# Multilayer Perceptron

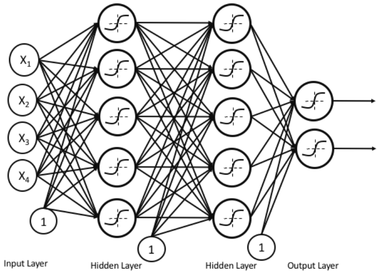
- Directed acyclic graph
- Nodes are artificial neurons
- Edges are connections between them
- Feedforward Neural Network
    - Neurons are organized in layers
    - No connection between neurons within a layer
    - All neurons in the same layer of the same type



# Multilayer Perceptron

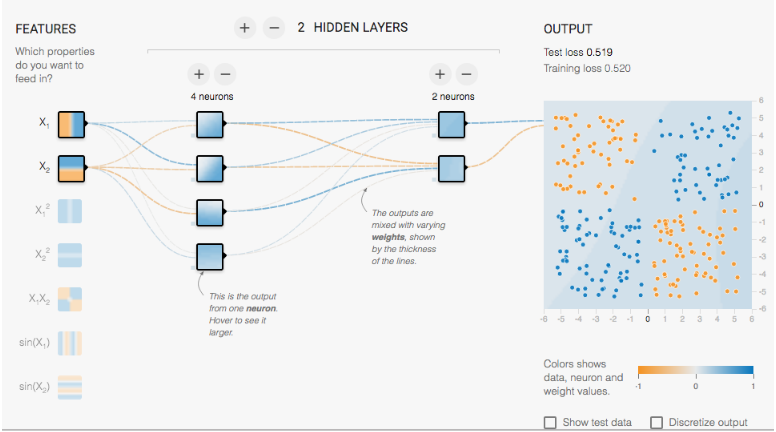
Each layer creates a new representation of the input data:
$h^{(0)} = f^{(0)}(\mathbf{x})$ $h^{(1)} = f^{(1)}(\mathbf{h^{(0)}})$
$y = f^{(2)}(\mathbf{h^{(1)}})$
Overall MLP is a function $f$
$y = f(x, \theta)$
Nested functions:
$f^{(3)}(f^{(2)}(f^{(1)}(x)))$

- First layer: $f^{(1)}$
- Second layer: $f^{(2)}$
- Third layer: $f^{(3)}$



# MLP & XOR

## Can solve XOR



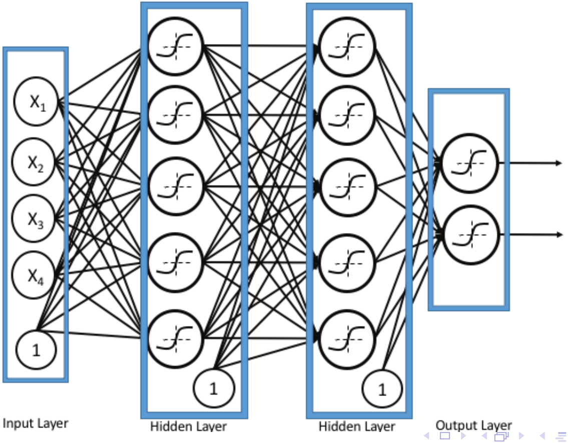
# MLP model

- Model:
  - $o_\theta = \phi_3(\mathbf{w_3}^\top \phi_1(\mathbf{w_2}^\top \phi_1(\mathbf{w_1}^\top x)))$
  - $\theta : \{\mathbf{W}\}$
- Loss function:
  - $L(\mathbf{x}, y; \mathbf{W}) = \frac{1}{2n} \sum_{i=0}^{n} (o_\theta - y)^2$
- Gradient of $L$ wrt $\mathbf{W}$:
  - $\frac{\partial}{\partial W} L(.)$

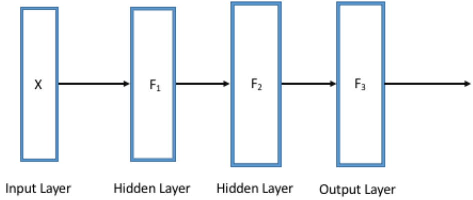
- biases omitted for brevity

# MLP consolidated

*Layered representation*

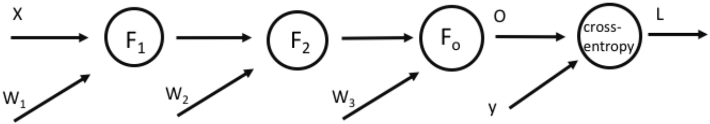


# MLP consolidated

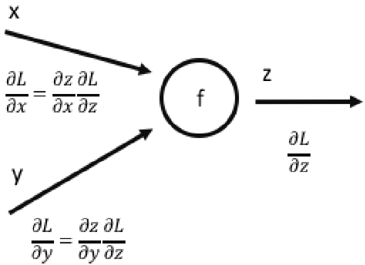


# MLP Compute Graph

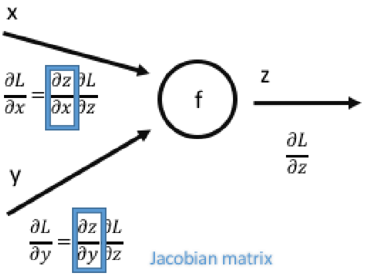
Compute Graph - Vectorized form



# Backprop Node



# Backprop Node



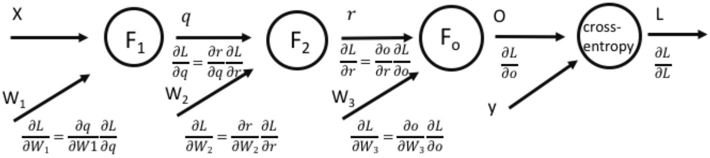
# Backprop MLP

$$J = \frac{\partial(\mathbf{F})}{\partial(\mathbf{W})} = \begin{vmatrix} \frac{\partial f_1}{\partial w_1} & \frac{\partial f_1}{\partial w_2} & \frac{\partial f_1}{\partial w_3} \\ \frac{\partial f_2}{\partial w_1} & \frac{\partial f_2}{\partial w_2} & \frac{\partial f_2}{\partial w_3} \\ \frac{\partial f_3}{\partial w_1} & \frac{\partial f_3}{\partial w_2} & \frac{\partial f_3}{\partial w_3} \end{vmatrix}$$

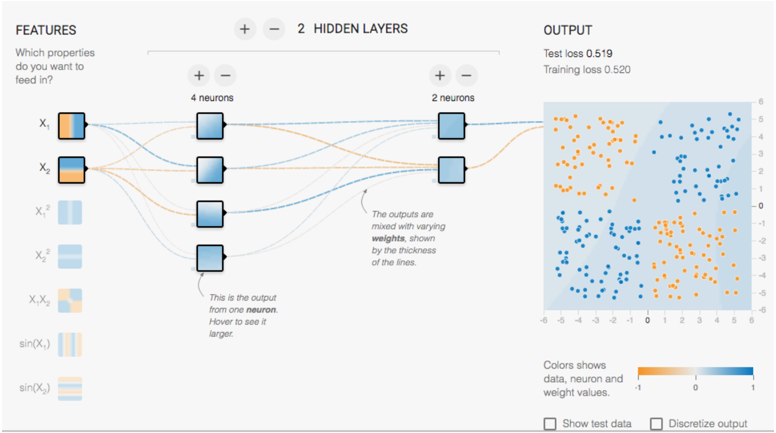
- Activation of neuron n:
  - $f_n$
- Parameters of neuron n:
  - $w_n$

$$J = \frac{\partial(\mathbf{F})}{\partial(\mathbf{W})} = \begin{vmatrix} \frac{\partial f_1}{\partial w_1} & 0 & 0 \\ 0 & \frac{\partial f_2}{\partial w_2} & 0 \\ 0 & 0 & \frac{\partial f_3}{\partial w_3} \end{vmatrix}$$

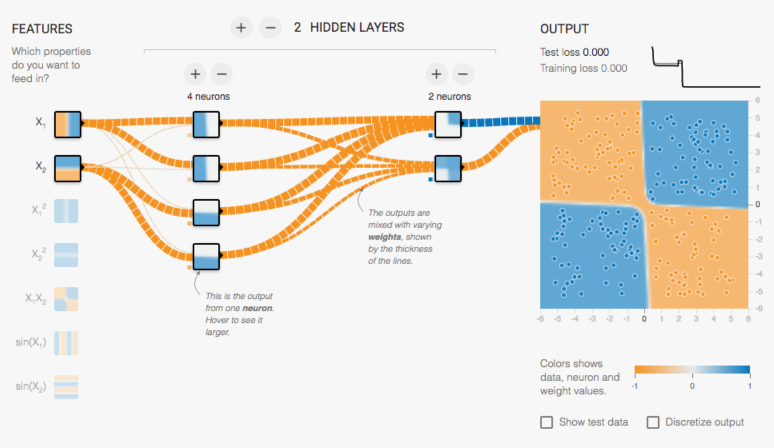
# MLP backprop compute graph



# MLP XOR Start



# MLP XOR Start



# MLP Classification

*MNIST dataset*

# Classification output Softmax

- We would like to output the probability distribution over a set of classes
- We need a output neuron for each class, such that each value corresponds to
  - $P(y = i|\mathbf{x})$ for computing the $i$ class with the $i$-th neuron
- Softmax units $\rightarrow$ Multinoulli output distributions
  - multiple neurons output the probability of each class
  - Normalized with the softmax function
  - $p(y = j|x) = \frac{e^{\mathbf{x}^{\top} w_j}}{\sum_{k=1}^{n} e^{\mathbf{x}^{\top} w}}$
  - strictly positive
  - sums to one

# MNIST

*Dataset*

- 784 dimensional vector $x$ for each datapoint
- 10 dimensional vector, $y$ for output
    - represents: probability distribution over the classes

*Design decisions*

- Output activation function
- Hidden layer activation function
- Architecture of the model
    - Number of hidden layers
    - Number neurons per layer
- Loss function
    - Properties of the loss function: differentiable (smooth)
    - Monotonically increasing with the distance from the target value

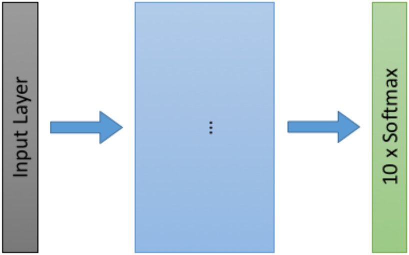
# Design decisions

*Design decisions*

- Output activation function : Softmax
  - $p(y = j|x) = \frac{e^{\mathbf{x}^\top w_j}}{\sum_{k=1}^{n} e^{\mathbf{x}^\top w}}$
- Hidden layer activation function: RelU
- Architecture of the model
  - Number of hidden layers : 2
  - Number neurons per layer: 256
- Loss function: Cross entropy
  - $L(\mathbf{x}, y; \theta) = -\sum_c 1_{(y=c)} \log\ o_c = -log\ o_y$

# Model Design

- Input format
- Output layer
- Loss function(s)
- Model Architecture
- Optimization parameters



# Output layer

*Regression*

- Linear units $\rightarrow$ Gaussian output distributions
  - Given a vector of feature activations $h$
  - $\hat{y} = W^T h + b$
  - $p(y|x) = \mathcal{N}(y; \hat{y}, I)$

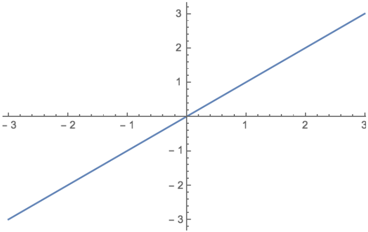
*Classification*

- Sigmoid units $\rightarrow$ Bernoulli output distributions
  - $p(y = 1|x)$
- Softmax units $\rightarrow$ Multinoulli output distributions
  - multiple neurons output the probability of each class
  - Normalized with the softmax function
  - $p(y = j|x) = \frac{e^{\mathbf{x}^\top w_j}}{\sum_{k=1}^{n} e^{\mathbf{x}^\top w}}$
  - strictly positive
  - sums to one

# Linear activation functions

$$g(z) = z$$

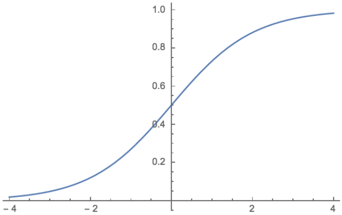
$$h = g(W^\top x + b)$$




- Usually used as a last layer activation for doing regression
- If all neurons are linear, the MLP is linear, which limits the generalization

# Sigmoid activation

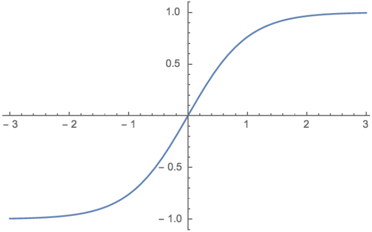
$\phi(z) = \frac{1}{1+e^{-z}}$
$h = \phi(W^\top x + b)$




Positive, bounded, strictly increasing

# Hyperbolic Tangent

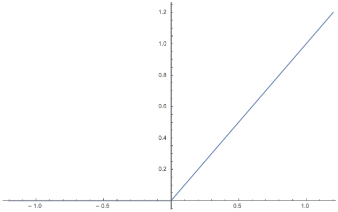
$$\phi(z) = \tanh(z)$$
$$h = \phi(W^\top x + b)$$




Positive, negative, bounded, strictly increasing

# Rectified Linear Unit

$$\phi(z) = max\{0, z\}$$
$$h = \phi(W^\top x + b)$$




- Bounded below by 0, no upper bound, monotonically increasing
- Not differentiable at 0
- Produces sparse activations
- Addresses the vanishing gradient problem
- Tip: Bias initialization to small positive values
- Variations: Leaky ReLU, PReLU, Maxout

# Model design - depth and width

*Depth and Width*

- Capacity
- Compositional features

# Model design - number of layers

*Number of layers*

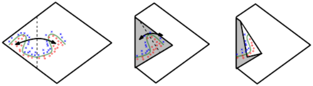
*Single hidden layer* - Universal approximation theorem (Hornik, 1991)

a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough units

- Capacity scales poorly
  - To learn a complex function the model needs exponentially many neurons

# Model capacity

- Shallow and deep network can learn the same functions
- Models with sequence of layers:
  - Each layer can partitioning the original input space piecewise linearly
  - Each subsequent layer recognizes pieces of the original input
  - Apply the same computation across different regions



- The segments grows:
  - exponentially with the number of layers
  - polynomial with the number of neurons
- Should we use very deep networks for any problem?

# Gradient Descent

* $\theta \leftarrow \theta - \alpha \nabla_\theta L(\mathbf{x}; \theta)$

- Overfitting
  - Early stop, Learning rate adaptation
  - Weight decay L1/L2 regularization (ridge regression)
- Momentum
  - $v \leftarrow \gamma v - \alpha \nabla_\theta L(\mathbf{x}; \theta)$
  - $\theta \leftarrow \theta - v$
- Nestorov momentum
- AdaGrad
- AdaDelta
- Adam
- RMSProp

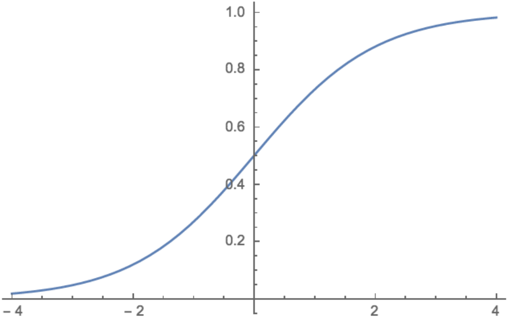
# Stochastic Gradient Descent

- Learn in Batches
- Reduce learning rate when it plateaus
    - Learning rate adaptation

# SGD algorithms

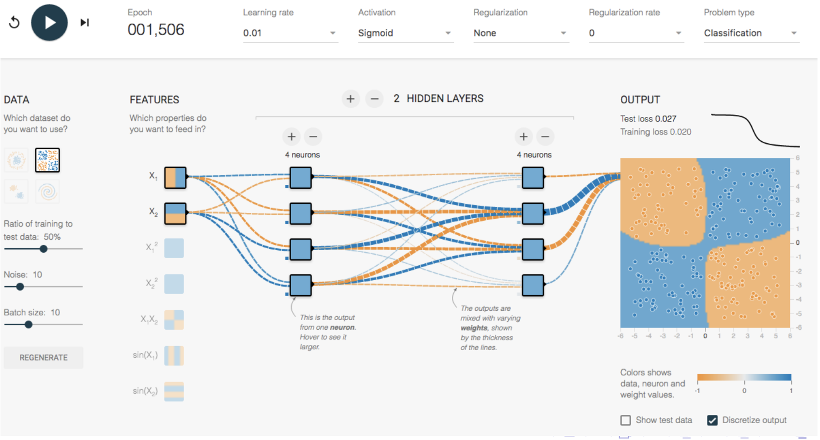
See animated GIF in browser

# Vanishing gradient



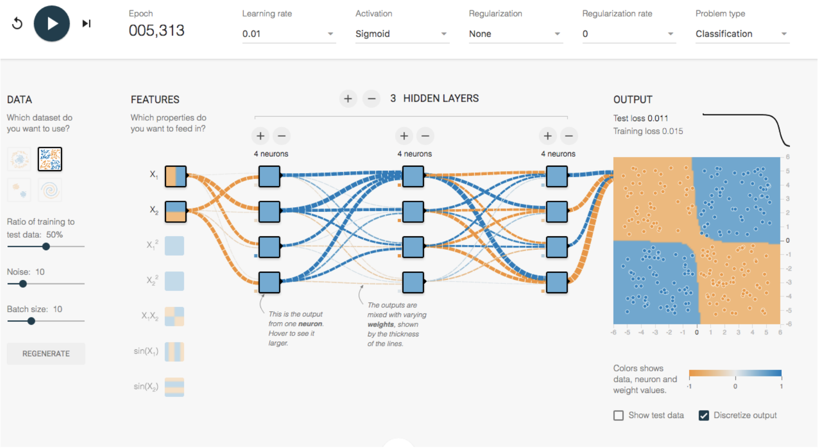
# Sigmoid activations 2 layers

## *Sigmoid activations 2 layers*



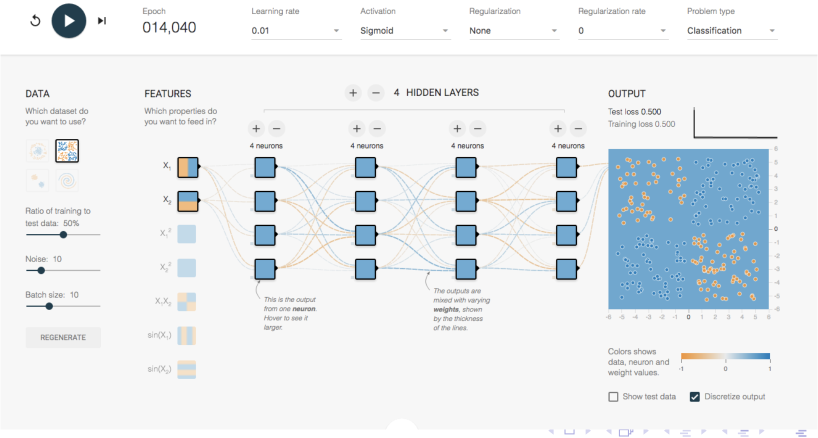
# Sigmoid activations 3 layers

## *Sigmoid activations 3 layers*



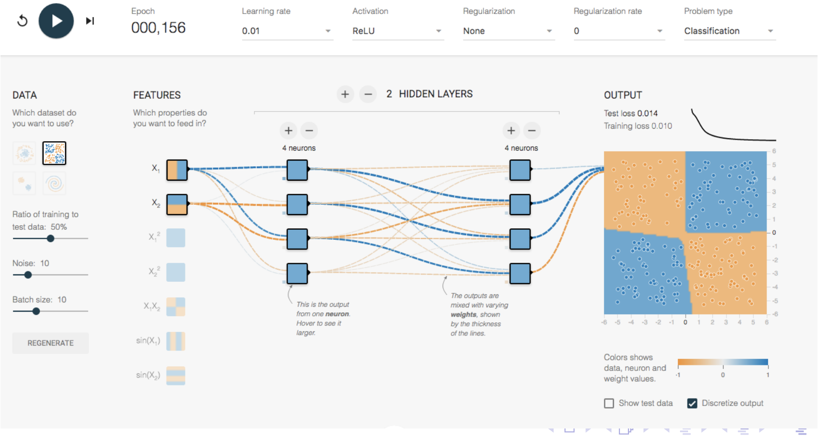
# Sigmoid activations 4 layers

*Sigmoid activations 4 layers*



# Sigmoid activations ReLU

## *Sigmoid activations ReLU*



# Regularization

*Regularization*

- L1/L2
  - Weights
  - Activations
- Sparsity
- `https://keras.io/regularizers/`
- `https://www.tensorflow.org/api_guides/python/contrib.layers#Regularizers`

# Initialization

*Initialization*

Depends on the activation function

- ReLU, small positive weights
- (Grolot et al. 2010)
- https://keras.io/initializers/https://keras.io/initializers/
- https://www.tensorflow.org/api_guides/python/contrib.layers#Initializers

# Dropout