Deep Learning

Lecture 3 – Deep Neural Networks

Prof. Dr.-Ing. Andreas Geiger

Autonomous Vision Group

University of Tübingen / MPI-IS

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

MAX-PLANCK-GESELLSCHAFT

ellis

European Laboratory for Learning and Intelligent Systems

Agenda

3.1 Backpropagation with Tensors

3.2 The XOR Problem

3.3 Multi-Layer Perceptrons

3.4 Universal Approximation

3.1
Backpropagation with Tensors

Recap: Backpropagation with Scalars

Forward Pass:

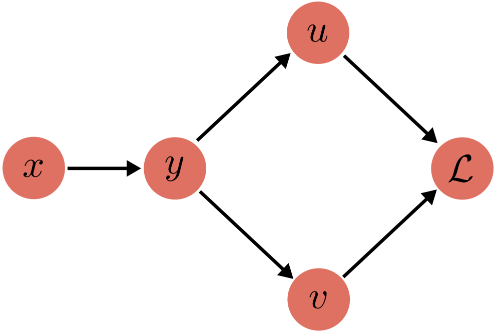
$$(1) \quad y = y(x)$$

$$(2) \quad u = u(y)$$

$$(2) \quad v = v(y)$$

$$(3) \quad \mathcal{L} = \mathcal{L}(u, v)$$

Loss: $\mathcal{L}(\, u(y(x)),\, v(y(x))\,)$



Recap: Backpropagation with Scalars

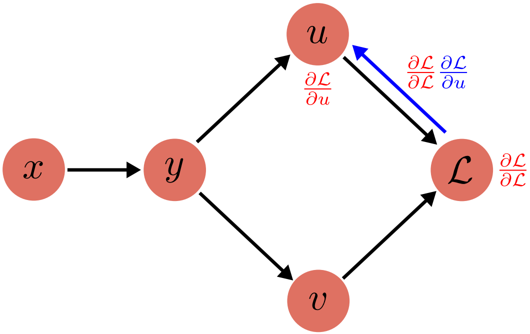
Forward Pass:

$$
\begin{aligned}
(1) \quad & y = y(x) \\
(2) \quad & u = u(y) \\
(2) \quad & v = v(y) \\
(3) \quad & \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u}$$

Loss: $\mathcal{L}(\, u(y(x)),\, v(y(x))\,)$



Recap: Backpropagation with Scalars

Forward Pass:

$$(1) \quad y = y(x)$$

$$(2) \quad u = u(y)$$

$$(2) \quad v = v(y)$$

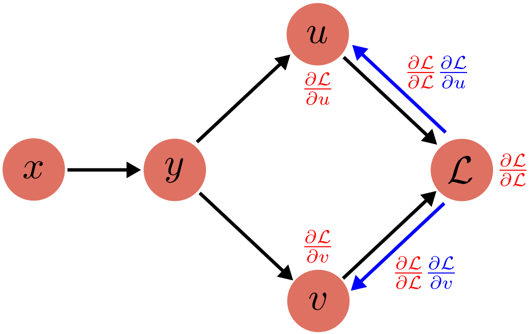
$$(3) \quad \mathcal{L} = \mathcal{L}(u, v)$$

Backward Pass:

$$(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u}$$

$$(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v}$$

Loss: $\mathcal{L}(\, u(y(x)), \, v(y(x))\,)$



Recap: Backpropagation with Scalars

Forward Pass:

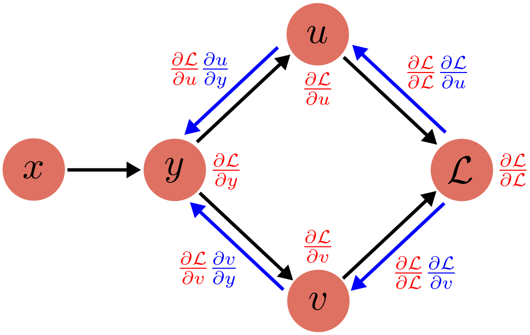
$$(1) \quad y = y(x)$$

$$(2) \quad u = u(y)$$

$$(2) \quad v = v(y)$$

$$(3) \quad \mathcal{L} = \mathcal{L}(u, v)$$

Loss: $\mathcal{L}(\, u(y(x)),\, v(y(x))\,)$



Backward Pass:

$$(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u}$$

$$(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v}$$

$$(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y}$$

Recap: Backpropagation with Scalars

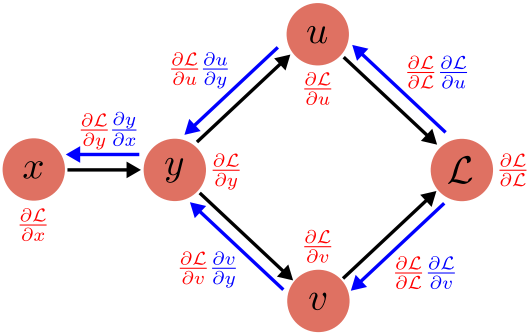
Forward Pass:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}
\end{aligned}
$$

Loss: $\mathcal{L}(\, u(y(x)),\, v(y(x))\,)$



Recap: Backpropagation with Scalars

Forward Pass:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}
\end{aligned}
$$

Implementation: Each variable/node is an object and has attributes `x.value` and `x.grad`. Values are computed **forward** and gradients **backward:**

$$
\begin{aligned}
\texttt{x.value} &= \texttt{Input} \\
\texttt{y.value} &= y(\texttt{x.value}) \\
\texttt{u.value} &= u(\texttt{y.value}) \\
\texttt{v.value} &= v(\texttt{y.value}) \\
\texttt{L.value} &= \mathcal{L}(\texttt{u.value}, \texttt{v.value})
\end{aligned}
$$

Recap: Backpropagation with Scalars

Forward Pass:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}
\end{aligned}
$$

Implementation: Each variable/node is an object and has attributes `x.value` and `x.grad`. Values are computed **forward** and gradients **backward:**

$$
\begin{aligned}
&\texttt{x.grad} = \texttt{y.grad} = \texttt{u.grad} = \texttt{v.grad} = 0 \\
&\texttt{L.grad} = 1 \\
&\texttt{u.grad} \mathrel{+}= \texttt{L.grad} * (\partial \mathcal{L}/\partial u)(\texttt{u.value}, \texttt{v.value}) \\
&\texttt{v.grad} \mathrel{+}= \texttt{L.grad} * (\partial \mathcal{L}/\partial v)(\texttt{u.value}, \texttt{v.value}) \\
&\texttt{y.grad} \mathrel{+}= \texttt{u.grad} * (\partial u/\partial y)(\texttt{y.value}) \\
&\texttt{y.grad} \mathrel{+}= \texttt{v.grad} * (\partial v/\partial y)(\texttt{y.value}) \\
&\texttt{x.grad} \mathrel{+}= \texttt{y.grad} * (\partial y/\partial x)(\texttt{x.value})
\end{aligned}
$$

Scalar vs. Matrix Operations

So far we have considered computations on **scalars:**

$$y = \sigma(w_1 x + w_0)$$

We now consider computations on **vectors** and **matrices:**

$$\mathbf{y} = \sigma(\mathbf{A}\mathbf{x} + \mathbf{b})$$

- Matrix $\mathbf{A}$ and vector $\mathbf{b}$ are objects with attributes `value` and `grad`
- `A.grad` stores $\nabla_{\mathbf{A}}\mathcal{L}$ and `b.grad` stores $\nabla_{\mathbf{b}}\mathcal{L}$
- `A.grad` has the same shape/dimensions as `A.value` (since $\mathcal{L}$ is scalar)

Backpropagation on Loops

The matrix/vector computation

$$\mathbf{y} = \sigma(\underbrace{\mathbf{A}\mathbf{x}}_{=\mathbf{u}} + \mathbf{b})$$

can be written as **loops over scalar operations:**

$$\begin{aligned}
&\text{for i} && \texttt{u.value}[i] = 0 \\
&\text{for i,j} && \texttt{u.value}[i] \mathrel{\texttt{+=}} \texttt{A.value}[i,j] * \texttt{x.value}[j] \\
&\text{for i} && \texttt{y.value}[i] = \sigma(\texttt{u.value}[i] + \texttt{b.value}[i])
\end{aligned}$$

Backpropagation on Loops

The backpropagated gradients for

$$\text{for i} \quad \mathtt{y.value}[i] = \sigma(\mathtt{u.value}[i] + \mathtt{b.value}[i])$$

are:

$$\text{for i} \quad \mathtt{u.grad}[i] \mathrel{+}= \mathtt{y.grad}[i] * \sigma'(\mathtt{u.value}[i] + \mathtt{b.value}[i])$$

$$\text{for i} \quad \mathtt{b.grad}[i] \mathrel{+}= \mathtt{y.grad}[i] * \sigma'(\mathtt{u.value}[i] + \mathtt{b.value}[i])$$

- **Red:** back-propagated gradients
- **Blue:** local gradients

Backpropagation on Loops

The backpropagated gradients for

$$\text{for i,j} \quad \texttt{u.value}[i] \mathrel{+}= \texttt{A.value}[i,j] * \texttt{x.value}[j]$$

are

$$\text{for i,j} \quad \texttt{A.grad}[i,j] \mathrel{+}= \texttt{u.grad}[i] * \texttt{x.value}[j]$$

$$\text{for i,j} \quad \texttt{x.grad}[j] \mathrel{+}= \texttt{u.grad}[i] * \texttt{A.value}[i,j]$$

- **Red:** back-propagated gradients
- **Blue:** local gradients

Backpropagation on Loops

In practice, all deep learning operations can be written using loops over scalar assignments. Example for a **higher-order tensor:**

for h,i,j,k $\quad \texttt{U.value}[h,i,j] \mathrel{+}= \texttt{A.value}[h,i,k] * \texttt{B.value}[h,j,k]$

for h,i,j $\quad \texttt{Y.value}[h,i,j] = \sigma(\texttt{U.value}[h,i,j])$

Backpropagation loops:

h,i,j $\quad \texttt{U.grad} \mathrel{+}= \texttt{Y.grad}[h,i,j] * \sigma'(\texttt{U.value}[h,i,j])$

h,i,j,k $\quad \texttt{A.grad} \mathrel{+}= \texttt{U.grad}[h,i,j] * \texttt{B.value}[h,j,k]$

h,i,j,k $\quad \texttt{B.grad} \mathrel{+}= \texttt{U.grad}[h,i,j] * \texttt{A.value}[h,i,k]$

Minibatching

Source code has two components:

- Slow part: Sequential operations (Python)
- Fast part: Vector/matrix operations (NumPy, BLAS, CUDA)

Goal:

- Fast part should dominate computation (wall clock time)
- Reduce the number of slow sequential operations (e.g., Python loops) by running the fast vector/matrix operations on several data points jointly
- This is called **minibatching** and used in stochastic gradient descent

Minibatching

Affine + Sigmoid: (applied to N data points simultaneously)

$$\mathbf{Y} = \sigma(\underbrace{\mathbf{X}\mathbf{A}}_{=\mathbf{U}} + \mathbf{B})$$

- Each row in $\mathbf{X} \in \mathbb{R}^{N \times D}$ is a data point, bias $\mathbf{b} \in \mathbb{R}^{M}$ is broadcast to $\mathbf{B} \in \mathbb{R}^{N \times M}$

Loops now include **batch index b:**

for b,i $\quad \texttt{U.value}[b, i] = 0$

for b,i,j $\quad \texttt{U.value}[b, i] \mathrel{\texttt{+=}} \texttt{X.value}[b, j] * \texttt{A.value}[j, i]$

for b,i $\quad \texttt{Y.value}[b, i] = \sigma(\texttt{U.value}[b, i] + \texttt{B.value}[i])$

- Only inputs and outputs depend on batch index b, not the parameters (e.g., `A`, `B`)
- By convention, the gradients are averaged over the batch

Implementation

Affine Transformation: (applied to N data points simultaneously)

$$\mathbf{Y} = \mathbf{X}\mathbf{A} + \mathbf{B}$$

- Each row in $\mathbf{X} \in \mathbb{R}^{N \times D}$ is a data point, bias $\mathbf{b} \in \mathbb{R}^{M}$ is broadcast to $\mathbf{B} \in \mathbb{R}^{N \times M}$

Implementation in EDF:

```
def forward(self):
    self.value = np.matmul(self.x.value,self.w.A.value) + self.w.b.value

def backward(self):
    self.x.addgrad(np.matmul(self.grad,self.w.A.value.transpose()))
    self.w.b.addgrad(self.grad)
    self.w.A.addgrad(self.x.value[:,:,np.newaxis] * self.grad[:,np.newaxis,:])
```

- Computation graphs are easy to understand using the loop notation
- Efficient implementation using NumPy primitives not always obvious

3.2
The XOR Problem

The XOR Problem

Logistic Regression Model:

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x}) \quad \text{with} \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

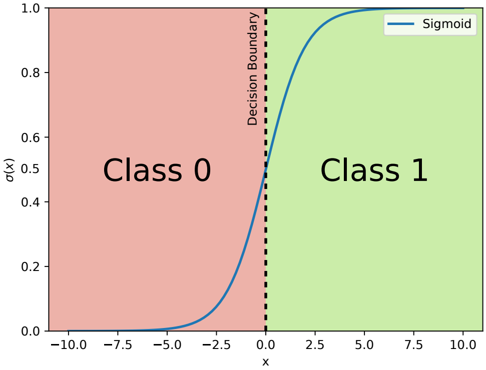
- Which problems can we solve with such a simple linear classifier?

The XOR Problem

Example: 2D Logistic Regression

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + w_0) \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

- Let $\mathbf{x} \in \mathbb{R}^2$
- Decision boundary: $\mathbf{w}^\top \mathbf{x} + w_0 = 0$
- Decide for class $1 \Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$
- Decide for class $0 \Leftrightarrow \mathbf{w}^\top \mathbf{x} < -w_0$



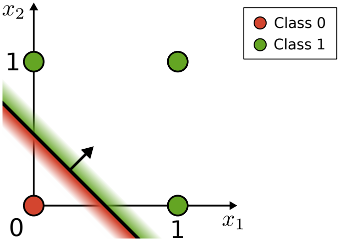
The XOR Problem

Linear Classifier:

Class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$

$$\underbrace{\begin{pmatrix} 1 & 1 \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{0.5}_{-w_0}$$

x_1	x_2	OR(x_1,x_2)
0	0	0
0	1	1
1	0	1
1	1	1



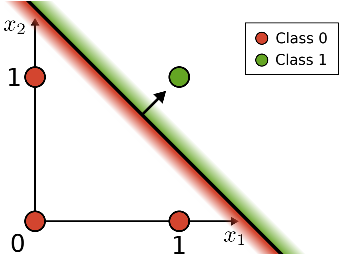
The XOR Problem

Linear Classifier:

Class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$

$$\underbrace{\begin{pmatrix} 1 & 1 \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{1.5}_{-w_0}$$

x_1	x_2	AND(x_1,x_2)
0	0	0
0	1	0
1	0	0
1	1	1



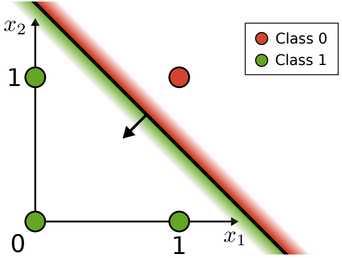
The XOR Problem

Linear Classifier:

Class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$

$$\underbrace{\begin{pmatrix} -1 & -1 \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{-1.5}_{-w_0}$$

x_1	x_2	NAND(x_1,x_2)
0	0	1
0	1	1
1	0	1
1	1	0



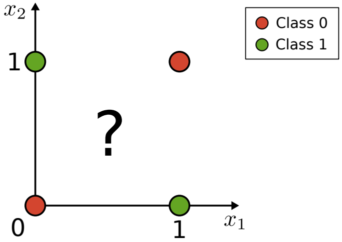
The XOR Problem

Linear Classifier:

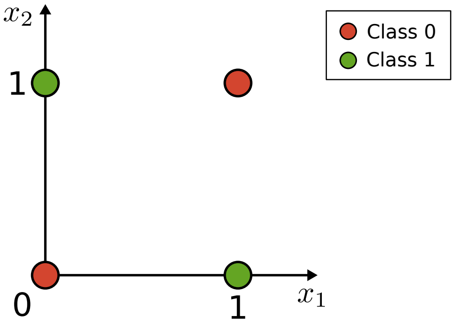
Class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$

$$\underbrace{\begin{pmatrix} ? & ? \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{?}_{-w_0}$$

x_1	x_2	XOR(x_1,x_2)
0	0	0
0	1	1
1	0	1
1	1	0

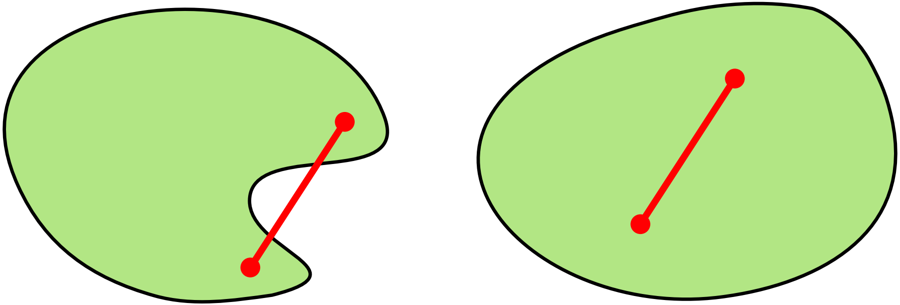


The XOR Problem



- Visually it is obvious that XOR is not linearly separable
- How can we formally prove this?

Convex Sets

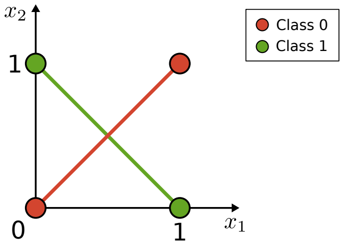


- A set $\mathcal{S}$ is **convex** if any line segment connecting 2 points in $\mathcal{S}$ lies entirely within $\mathcal{S}$:

$$\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{S} \quad \Rightarrow \quad \lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in \mathcal{S} \quad \text{for} \quad \lambda \in [0,1]$$

The XOR Problem

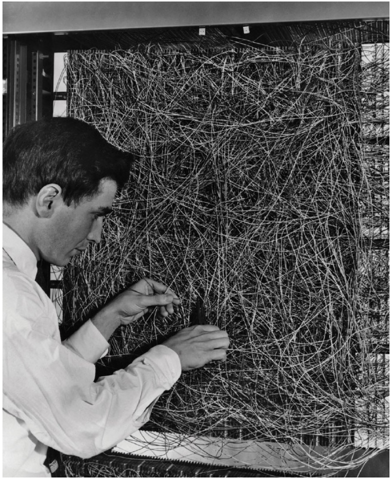
- Half-spaces (e.g., decision regions) are convex sets
- Suppose there was a feasible hypothesis. If the positive examples are in the positive half-space, then the green line segment must be as well.
- Similarly the red line must lie within the negative half-space.
- But the intersection can't lie in both half-spaces. Contradiction!



The XOR Problem

Some Historical Remarks:

- While linear classification showed some promising results in the 50s and 60s on simple image classification problems (Perceptron)
- But limitations became clear very soon (e.g., Minsky and Papert book “Perceptrons”, 1969)
- **XOR problem** is simple but cannot be solved as model capacity limited to linear decisions
- Led to decline in neural net research in the 70s
- How can we solve non-linear problems?

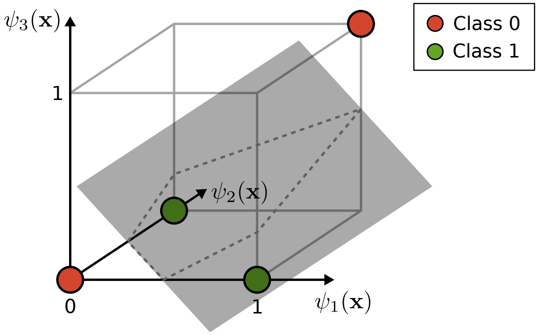


The XOR Problem

Linear classifier with non-linear features ψ:

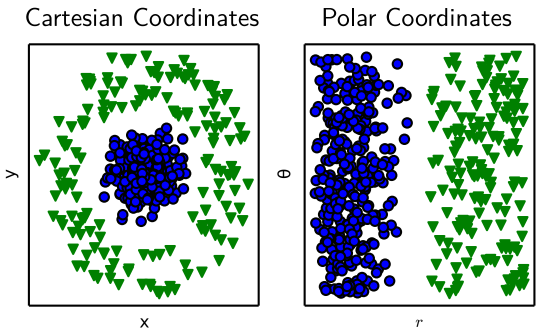
$$\mathbf{w}^\top \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}}_{\boldsymbol{\psi}(\mathbf{x})} > -w_0$$

x_1	x_2	$\psi_1(\mathbf{x})$	$\psi_2(\mathbf{x})$	$\psi_3(\mathbf{x})$	XOR
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0



- Non-linear features allow linear classifier to solve non-linear classification problems!
- Analogous to polynomial curve fitting

Representation Matters



- But how to choose the transformation? Can be very hard in practice.
- Yet, this was the dominant approach until the 2000s (vision, speech, ..)
- In this class we want to learn them $\Rightarrow$ **Representation learning**
- Human needs to choose the right function family rather than the correct function

The XOR Problem

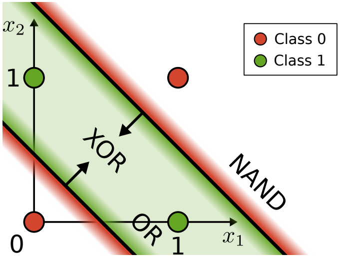
Linear Classifier:

Class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$

x_1	x_2	XOR(x_1,x_2)
0	0	0
0	1	1
1	0	1
1	1	0

XOR(x_1, x_2) =

AND(OR(x_1, x_2),NAND(x_1, x_2))



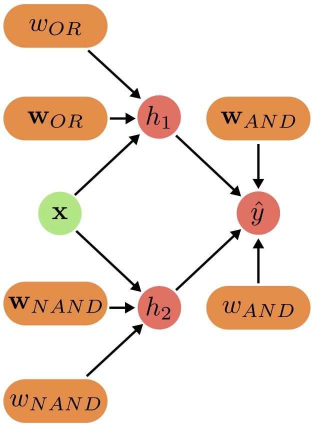
The XOR Problem

$$\text{XOR}(x_1, x_2) = \text{AND}(\text{OR}(x_1, x_2), \text{NAND}(x_1, x_2))$$

The above expression can be rewritten as a **program of logistic regressors:**

$$h_1 = \sigma(\mathbf{w}_{OR}^\top \mathbf{x} + w_{OR})$$
$$h_2 = \sigma(\mathbf{w}_{NAND}^\top \mathbf{x} + w_{NAND})$$
$$\hat{y} = \sigma(\mathbf{w}_{AND}^\top \mathbf{h} + w_{AND})$$

Note that $\mathbf{h}(\mathbf{x})$ is a non-linear feature of $\mathbf{x}$.
We call $\mathbf{h}(\mathbf{x})$ a **hidden** layer.



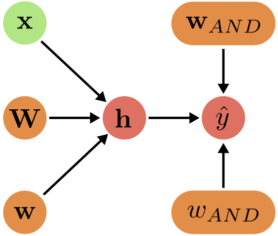
The XOR Problem

$$\text{XOR}(x_1, x_2) = \text{AND}(\text{OR}(x_1, x_2), \text{NAND}(x_1, x_2))$$

Writing the two 1D mappings $h_1(\mathbf{x})$ and $h_2(\mathbf{x})$ as a **single 2D mapping** $\mathbf{h}(\mathbf{x})$ yields:

$$\mathbf{h} = \sigma\left(\underbrace{\begin{pmatrix} \mathbf{w}_{OR}^\top \\ \mathbf{w}_{NAND}^\top \end{pmatrix}}_{\mathbf{W}} \mathbf{x} + \underbrace{\begin{pmatrix} w_{OR} \\ w_{NAND} \end{pmatrix}}_{\mathbf{w}}\right)$$

$$\hat{y} = \sigma(\mathbf{w}_{AND}^\top \mathbf{h} + w_{AND})$$



Parameters can be learned using backprop. This is our first **Multi-Layer Perceptron!**

3.3
Multi-Layer Perceptrons

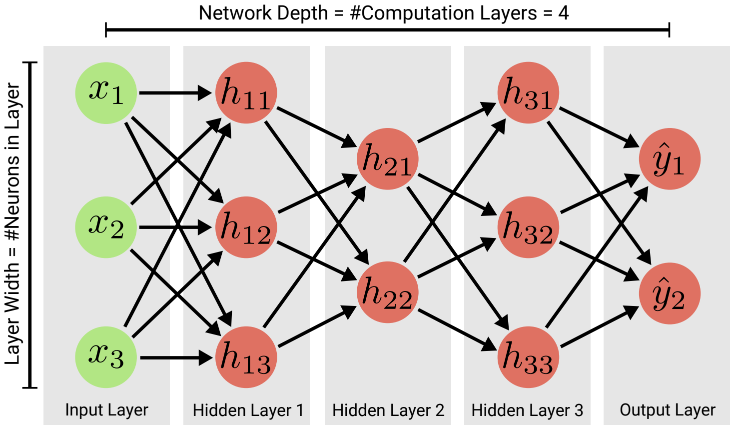
Multi-Layer Perceptrons

- MLPs are **feedforward** neural networks (no feedback connections)
- They **compose** several non-linear functions $\mathbf{f}(\mathbf{x}) = \hat{\mathbf{y}}(\mathbf{h}_3(\mathbf{h}_2(\mathbf{h}_1(\mathbf{x}))))$ where $\mathbf{h}_i(\cdot)$ are called **hidden layers** and $\hat{\mathbf{y}}(\cdot)$ is the **output layer**
- The data specifies only the behavior of the output layer (thus the name "hidden")
- Each layer i comprises multiple **neurons** j which are implemented as **affine transformations** ($\mathbf{a}^\top\mathbf{x} + \mathbf{b}$) followed by non-linear **activation functions** (g):

$$h_{ij} = g(\mathbf{a}_{ij}^\top \mathbf{h}_{i-1} + \mathbf{b}_{ij})$$

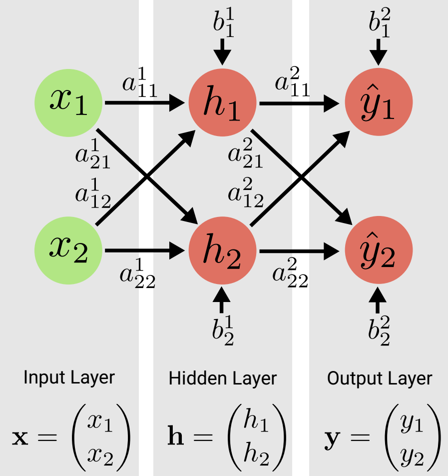
- Each neuron in each layer is **fully connected** to all neurons of the previous layer
- The overall length of the chain is the **depth** of the model $\Rightarrow$ "Deep Learning"
- The name MLP is misleading as we don't use threshold units as in Perceptrons

MLP Network Architecture



- Neurons are grouped into layers, each neuron **fully connected** to all prev. ones
- **Hidden layer** $\mathbf{h}_i = g(\mathbf{A}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$
- **Output layer** $\mathbf{y} = h(\mathbf{A}_L \mathbf{h}_{L-1} + \mathbf{b}_L)$ with **activ. function** $h(\cdot)$ and weights $\mathbf{A}_L, \mathbf{b}_L$

MLP Network Architecture



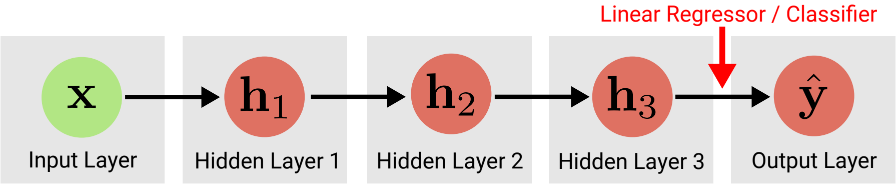
$$\begin{aligned} \mathbf{h} &= g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1) \\ \mathbf{y} &= h(\mathbf{A}_2\mathbf{h} + \mathbf{b}_2) \end{aligned}$$

$$\mathbf{A}_l = \begin{pmatrix} a_{11}^l & a_{12}^l \\ a_{21}^l & a_{22}^l \end{pmatrix}$$

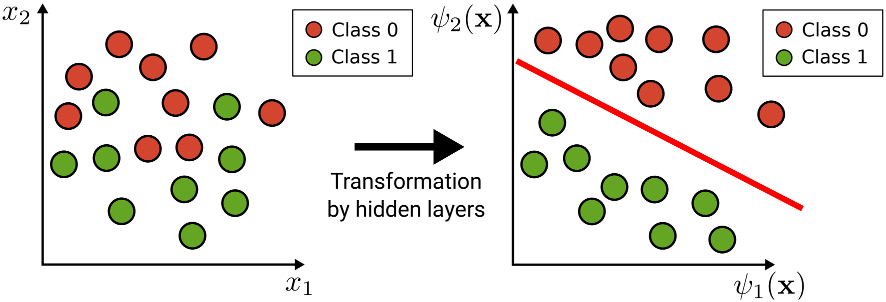
$$\mathbf{b}_l = \begin{pmatrix} b_1^l \\ b_2^l \end{pmatrix}$$

- Example MLP with $L = 2$ layers of width 2 and parameters $\{\mathbf{A}_1, \mathbf{A}_2, \mathbf{b}_1, \mathbf{b}_2\}$

Feature Learning Perspective



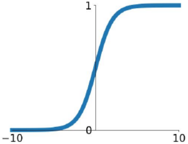
$$\mathbf{h}_3 = \psi(\mathbf{x})$$



Activation Functions g(·)

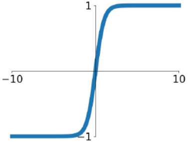
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



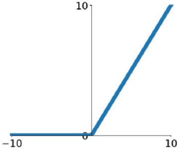
tanh

$\tanh(x)$



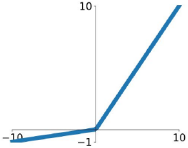
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

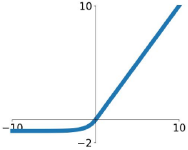


Maxout

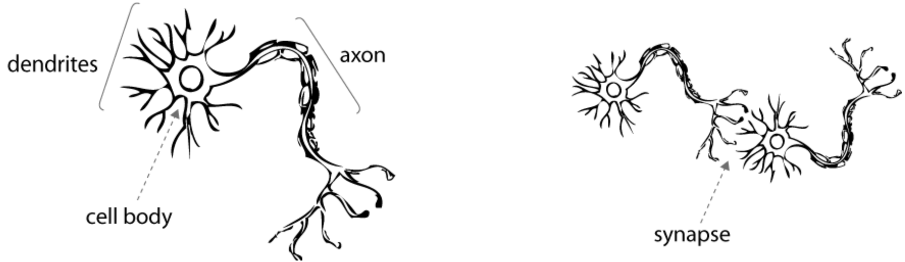
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural Motivation



- Neurons in the brain are structured in layers
- They receive input from many other units and compute their own activation
- The sigmoid activation function is guided by neuroscientific observations
- However, the architecture and training of modern networks differs radically
- Our main goal is not to model the brain, but to achieve statistical generalization

Training

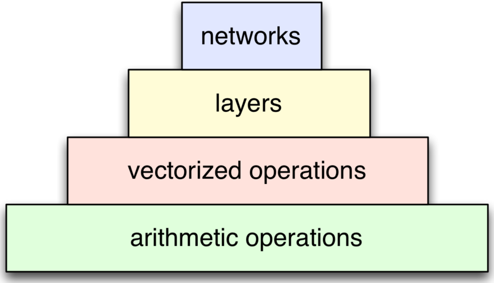
Algorithm for training an MLP using (stochastic) gradient descent:

1. Initialize weights $\mathbf{w}$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw (random) minibatch $\mathcal{X}_{\text{batch}} \subseteq \mathcal{X}$
3. For all elements $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}_{\text{batch}}$ of minibatch (in parallel) do:
   3.1 Forward propagate $\mathbf{x}$ through network to calculate $\mathbf{h}_1, \mathbf{h}_2, \ldots, \hat{\mathbf{y}}$
   3.2 Backpropagate gradients through network to obtain $\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|\mathcal{X}_{\text{batch}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{X}_{\text{batch}}} \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
5. If validation error decreases, go to step 2, otherwise stop

Remarks:

- Large datasets typically do not fit into GPU memory $\Rightarrow |\mathcal{X}_{\text{batch}}| < |\mathcal{X}|$
- Our examples on the next slides are small $\Rightarrow |\mathcal{X}_{\text{batch}}| = |\mathcal{X}|$

Levels of Abstraction



- When designing neural networks and machine learning algorithms, you'll need to simultaneously think at multiple level's of abstraction
- "The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large." [Donald E. Knuth]

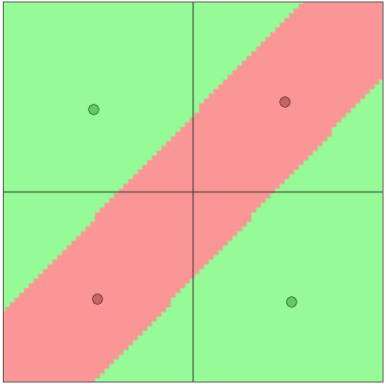
The XOR Problem

```
layer_defs = [];
layer_defs.push({type:'input', out_sx:1, out_sy:1, out_depth:2});
layer_defs.push({type:'fc', num_neurons:2, activation: 'tanh'});
layer_defs.push({type:'fc', num_neurons:1, activation: 'tanh'});
layer_defs.push({type:'softmax', num_classes:2});

net = new convnetjs.Net();
net.makeLayers(layer_defs);

trainer = new convnetjs.SGDTrainer(net, {learning_rate:0.01,
momentum:0.1, batch_size:10, l2_decay:0.001});
```

change network

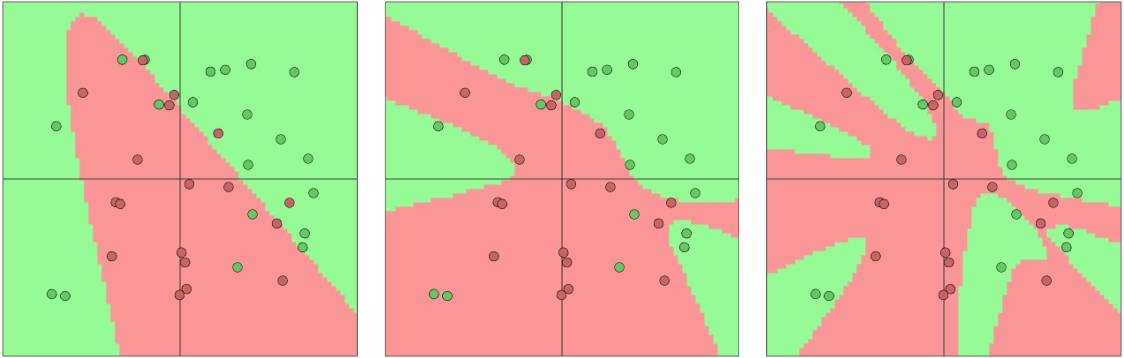


- Note that we have learned a boolean circuit! $\Rightarrow$ differentiable programming

`https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html`

A More Challenging Problem

2 Hidden Neurons | 5 Hidden Neurons | 15 Hidden Neurons



https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Expressiveness

This following two-layer MLP

$$\mathbf{h} = g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{y} = g(\mathbf{A}_2\mathbf{h} + \mathbf{b}_2)$$

can be written as

$$\mathbf{y} = g(\mathbf{A}_2\, g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

What if we would be using a linear activation function $g(\mathbf{x}) = \mathbf{x}$?

$$\mathbf{y} = \mathbf{A}_2\left(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2 = \mathbf{A}_2\mathbf{A}_1\mathbf{x} + \mathbf{A}_2\mathbf{b}_1 + \mathbf{b}_2 = \mathbf{A}\mathbf{x} + \mathbf{b}$$

- With linear activations, a multi-layer network can only express linear functions
- What is the model capacity of MLPs with non-linear activation functions?

3.4
Universal Approximation

Universal Approximation Theorem

Theorem 1

Let σ be any continuous discriminatory function. Then finite sums of the form

$$G(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{a}_j^\top \mathbf{x} + b_j)$$

are dense in the space of continuous functions $C(I_n)$ on the n-dimensional unit cube I_n. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(\mathbf{x})$ for which

$$|G(\mathbf{x}) - f(\mathbf{x})| < \epsilon \quad \text{for all} \quad \mathbf{x} \in I_n$$

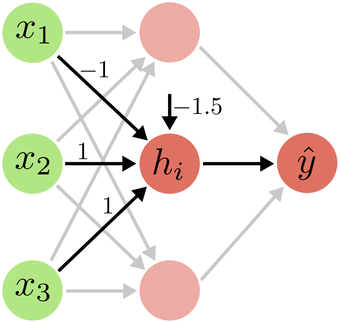
Remark: Has been proven for various activation functions (e.g., Sigmoid, ReLU).

Cybenko: Approximation by superposition of a sigmoidal function. Mathematics of Control, Signals, and Systems, 1989.

Example: Binary Case

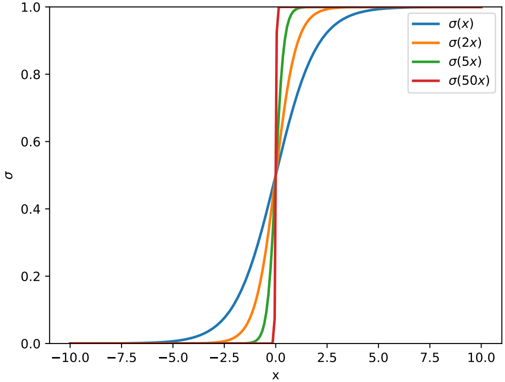
x_1	x_2	x_3	y
$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$

$$\hat{y} = \sum_i \underbrace{[\mathbf{a}_i^\top \mathbf{x} + b_i > 0]}_{h_i}$$



- Each hidden **linear threshold unit** h_i recognizes one possible input vector
- We need 2^D hidden units to **recognize** all 2^D possible inputs in the binary case

Soft Thresholds



Learning **linear threshold units** is hard as their gradient is 0 almost everywhere

- Solution: Replace hard threshold with **soft theshold** (e.g., sigmoid)
- Sigmoids approximate step functions when increasing the input weight

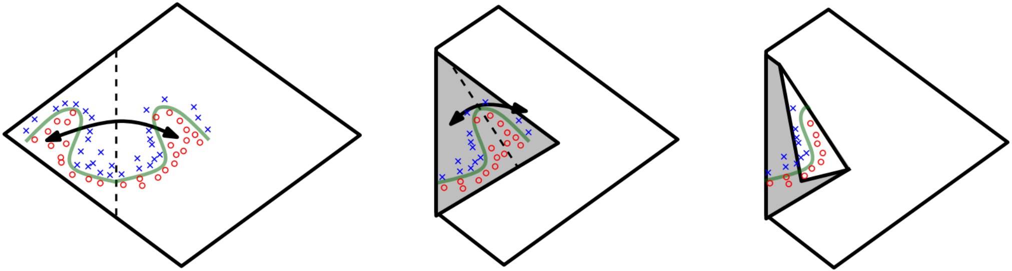
Network Width vs. Depth

- Universality of 2 layer networks is appealing but requires **exponential width**
- This leads to an exponential increase in memory and computation time
- Moreover, it doesn't lead to generalization $\Rightarrow$ network simply **memorizes inputs**
- **Deep networks** can represent functions more compactly (with less parameters)
- Inductive bias: Complex functions modeled as **composition of simple functions**
- This leads to **more compact** models and **better generalization** performance
- Example: The parity function

$$f(x_1, \ldots, x_D) = \begin{cases} 1 & \text{if } \sum_i x_i \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

requires an exponentially large shallow network but can be computed using a deep network whose size is linear in the number of inputs D.

Space Folding Intuition

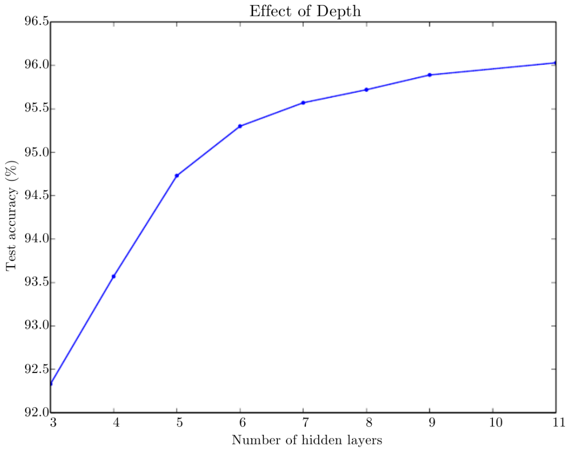


Space folding intuition for the case of **absolute value rectification units:**

- Geometric explanation of the exponential advantage of deeper networks
- Mirror axis of symmetry given by the hyperplane (defined by weights and bias)
- Complex functions arise as mirrored images of simpler patterns

Montufar, Pascanu, Cho and Bengio: On the Number of Linear Regions of Deep Neural Networks. NIPS, 2014.

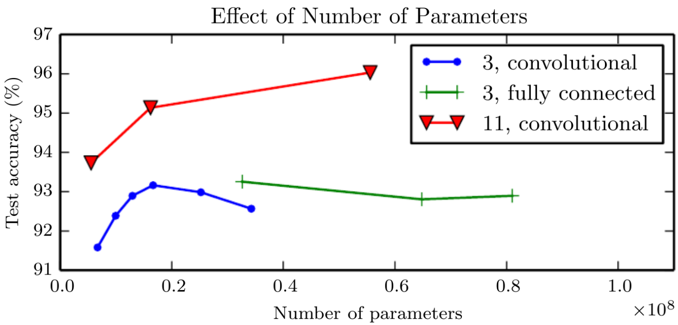
Effect of Network Depth



- Deeper networks generalize better (task: multi-digit number classification)

Goodfellow, Bulatov, Ibarz, Arnoud and Shet: Multi-digit number recognition from Street View imagery using deep convolutional neural networks. ICLR, 2014.

Effect of Network Depth



- Increasing the number of parameters is not as effective as increasing depth
- Shallow models even overfit at around 20 million parameters in this example
- Compositionality is a useful prior over the space of functions the model can learn

Goodfellow, Bulatov, Ibarz, Arnoud and Shet: Multi-digit number recognition from Street View imagery using deep convolutional neural networks. ICLR, 2014.