Introduction to Deep Learning with Tensorflow

Hector Andrade Loarca
Jan Macdonald

Technical University of Berlin
Institute of Mathematics

andrade@math.tu-berlin.de
macdonald@math-tu-berlin.de

Agenda

1. A Short Review of Deep Learning
2. Approximations with Neural Networks
3. Introduction to Tensorflow
4. Numerical Experiments

A Short Review of Deep Learning

Learning Problems

- input space $\mathcal{X}$, for example $\mathbb{R}^d$
- output space $\mathcal{Y}$, for example $\mathbb{R}$

Goal: Approximate $f\colon \mathcal{X} \to \mathcal{Y}$ from samples $((\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N))$

Hypothesis space: $\mathcal{H} \subseteq \mathcal{Y}^{\mathcal{X}}$, usually $\mathcal{H} \subseteq C(\mathcal{X}, \mathcal{Y})$

Empirical risk minimization: find a minimzer h^* of the empirical loss

$$h^* \in \operatorname*{argmin}_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} \ell(h(\mathbf{x}_i), \mathbf{y}_i)$$

for some loss function $\ell\colon \mathcal{Y} \times \mathcal{Y} \to [0, \infty]$

Learning Problems

Three main sources of errors:

- expressiveness of $\mathcal{H}$ (approximation error)

  How well can functions in $\mathcal{H}$ approximate f?

- generalization (sample error)

  How well does the sample represent the space $\mathcal{X} \times f(\mathcal{X})$?

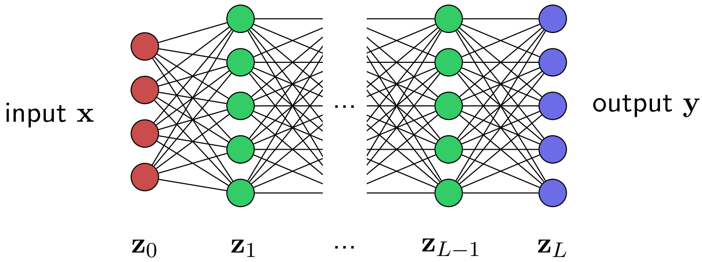
- optimization (training error)

  How well can we solve the minimization problem?

Deep Learning

- in deep learning $\mathcal{H}$ are neural networks with fixed architecture
- fully-connected networks
- convolutional networks
- optimization via (variants of) stochastic gradient descent
- ...

Neural Networks



$$\mathbf{z}_0 = \mathbf{x}$$

$$\mathbf{z}_i = \varrho(\mathbf{A}_i \mathbf{z}_{i-1} + \mathbf{b}_i), \quad i = 1, \ldots, L-1$$

$$\mathbf{y} = \mathbf{z}_L = \mathbf{A}_L \mathbf{z}_{L-1} + \mathbf{b}_L$$

Neural Networks Notation

- feed-forward neural network $\Phi = ((\mathbf{A}_1, \mathbf{b}_1), \ldots, (\mathbf{A}_L, \mathbf{b}_L))$
- weight matrices $\mathbf{A}_i \in \mathbb{R}^{d_i \times d_{i-1}}$ and bias vectors $\mathbf{b}_i \in \mathbb{R}^{d_i}$
- input dimension $d = d_0$ and output dimension $d_L = 1$
- number of layers $L(\Phi) = L$
- number of weights $W(\Phi) = \sum_{i=1}^{L} \|\mathbf{A}_i\|_0 + \|\mathbf{b}_i\|_0$
- denote the set of all such networks by $\mathcal{NN}_{d,L}$
- $\mathcal{NN}_d = \bigcup\limits_{L \in \mathbb{N}} \mathcal{NN}_{d,L}$

Notation inspired by P. Petersen & F. Voigtlaender (2017)

Neural Networks Notation

- the realization of Φ with activation function $\varrho\colon \mathbb{R} \to \mathbb{R}$ is the function

$$\Phi_\varrho(\mathbf{x}) = \mathbf{A}_L \varrho\left(\mathbf{A}_{L-1} \varrho\left(\dots \varrho\left(\mathbf{A}_1 \mathbf{x} + \mathbf{b}_1\right) + \dots\right) + \mathbf{b}_{L-1}\right) + \mathbf{b}_L$$

- denote the set of realizations as $\mathcal{NN}_{d,L,\varrho} = \{\, \Phi_\varrho \,:\, \Phi \in \mathcal{NN}_{d,L} \,\}$

Remarks:

- the activation function ϱ is applied componentwise
- commonly used activation function $\varrho(x) = \max\{0, x\}$ (ReLU)

Notation inspired by P. Petersen & F. Voigtlaender (2017)

Approximations with Neural Networks

Universal Approximations

"Feed-forward neural networks with one hidden layer and finite number of neurons can approximate many classes of functions on compact subsets of $\mathbb{R}^n$ arbitrarily well."

- **G. Cybenko (1981)**, sigmoid activation, approximations in $C([0,1]^d)$
- **K. Hornik (1991)**, bounded continuous non-constant activation, approximations in $L^p([0,1]^d)$
- **M. Leshno et al. (1993)**, locally bounded non-polynomial activation, approximations in $L^p([0,1]^d)$ and $C([0,1]^d)$

Universal Approximations

Activation function:

- $\varrho \in L^{\infty}_{\mathrm{loc}}(\mathbb{R})$
- closure of set of discontinuities of ϱ has measure zero

M. Leshno, V. Y. Lin, A. Pinkus, S. Schocken (1993)

For $d \in \mathbb{N}$ and $1 \le p < \infty$ the set $\mathcal{NN}_{d,\varrho,2}$ is dense in $L^p([0,1]^d)$ if and only if ϱ is not a polynomial (a.e.).

Also the set $\mathcal{NN}_{d,\varrho,2}$ is dense in $C([0,1]^d)$ if and only if ϱ is not a polynomial (a.e.).

Approximations with ReLU Networks

Sobolev spaces:

- $W^{r,p}((0,1)^d)$ Sobolev space of r-times weakly differentiable functions in $L^p((0,1)^d)$ with norm

$$\|f\|_{W^{r,p}((0,1)^d)} = \begin{cases} \left(\sum_{|\boldsymbol{\alpha}| \leq r} \|D^{\boldsymbol{\alpha}} f\|_p^p \right)^{1/p} & , \text{ if } 1 \leq p < \infty \\ \max_{|\boldsymbol{\alpha}| \leq r} \|D^{\boldsymbol{\alpha}} f\|_\infty & , \text{ if } p = \infty \end{cases}$$

- $F^{r,p,d}$ unit ball in $W^{r,p}((0,1)^d)$
- $F^{r,d} = F^{r,\infty,d}$

Approximations with ReLU Networks

Let $\varrho(x) = \max\{0, x\}$ be the ReLU activation function.

D. Yarotsky (2017)

For $d, r \in \mathbb{N}$, $\epsilon \in (0,1)$, and $f \in F^{d,r}$ there exists a neural network $\Phi \in \mathcal{NN}_d$ with $L(\Phi) \in \mathcal{O}(\ln \frac{1}{\epsilon} + 1)$ and $W(\Phi) \in \mathcal{O}(\epsilon^{-d/r}(\ln \frac{1}{\epsilon} + 1))$ such that $\|\Phi_\varrho - f\|_\infty \leq \epsilon$.

D. Yarotsky (2017)

For $d, r \in \mathbb{N}$ and $\epsilon \in (0,1)$ any ReLU network architecture capable of approximating all functions in $F^{d,r}$ with L^∞ error ϵ must have $\Omega(\epsilon^{-d/(2r)})$ weights.

Jupyter Notebook Time

https://github.com/arsenal9971/daedalus-intensive

Approximations with Sigmoid Networks

Let $\varrho(x) = 1/(1 + e^{-x})$ be the sigmoid activation function.

H. N. Mhaskar (1996)

For $d, r \in \mathbb{N}$, $\epsilon \in (0, 1)$, and $f \in F^{d,r}$ there exists a neural network $\Phi \in \mathcal{NN}_{d,2}$ with $W(\Phi) \in \mathcal{O}(\epsilon^{-d/r})$ such that $\|\Phi_\varrho - f\|_\infty \leq \epsilon$.

Observation: Deep ReLU neural networks and shallow sigmoid neural networks have a similar approximation behaviour for functions in $F^{d,r}$.

Can we verify this numerically?

Introduction to Tensorflow

Deep Learning Frameworks

Caffe2

PyTorch

mxnet

Keras

TensorFlow

DEEPLEARNING4J

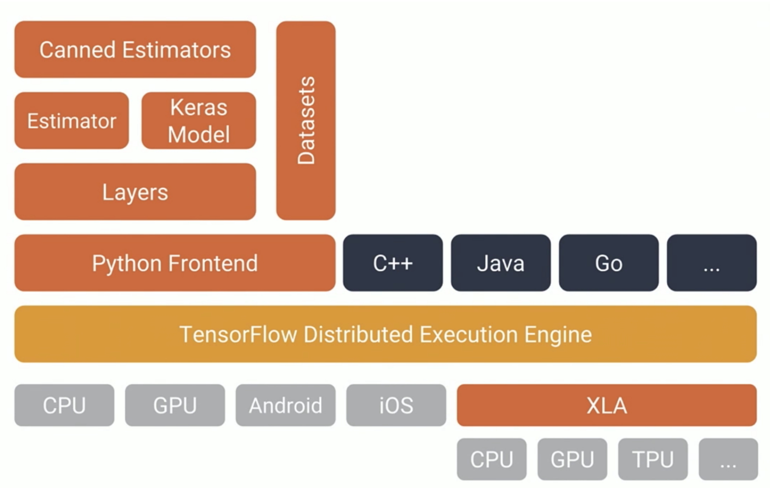
Microsoft CNTK

torch

theano

Chainer

What is Tensorflow?



Graphic: TensorFlow Fronties at Google I/O 2017

How to use Tensorflow?

```python
# importing the tensorflow python package
import tensorflow as tf

# also load numpy for convenience
import numpy as np
```

- in Tensorflow data is stored in tensor objects
- tensors can be thought of as multi-dimensional arrays
- they have a rank (number of array dimensions)
- they have a shape (list of number of components in each dimension)

How to use Tensorflow?

Three main types of tensors:

- constant tensors (have initial values, not changed during training)
- variable tensors (have initial values, can be changed during training)
- placeholder tensors (have no initial values, e.g. input layer)

How to use Tensorflow?

Computing with tensors:

- tensors can be combined with operations to produce new tensors
- reshaping, matrix multiplication, vector addition, convolutions, ...
- operations define a so called computation graph capturing the dependencies between tensors

How to use Tensorflow?

```
# create a constant scalar (0D) tensor
# with value 5, rank 0, shape ()
constant_scalar = tf.constant(5.0)

# create a constant matrix (2D) tensor
# with random values, rank 2, shape (3,2)
constant_matrix = tf.constant(
  np.random.randn(3,2)
)
```

How to use Tensorflow?

```
# create a variable matrix (2D) tensor
# with random inital values
# rank 2, shape (3,2)
variable_matrix = tf.Variable(
  np.random.randn(3,2),
  dtype=tf.float32,
  name='my_matrix'
)
```

How to use Tensorflow?

```
# create a variable vector (1D) tensor
# with random inital values
# rank 1, shape (3,)
variable_vector = tf.Variable(
  np.random.randn(3),
  dtype=tf.float32,
  name='my_vector'
)
```

How to use Tensorflow?

```python
# create a placeholder vector (1D) tensor
# with rank 1, shape (2,)
placeholder_input = tf.placeholder(
  dtype=tf.float32,
  shape=(2,),
  name='my_input'
)
```

How to use Tensorflow?

```
# reshape input as matrix
mat_input = tf.reshape(
    placeholder_input, [2, 1])

# compute a matrix product of two tensors
mat_product = tf.matmul(
    variable_matrix, mat_input)

# reshape product as vector
vec_product = tf.reshape(mat_product, [3])
```

How to use Tensorflow?

```
# compute vector addition of two tensors
vector_sum = vec_product + variable_vector

# compute ReLU activation of a tensor
relu_vector = tf.nn.relu(vector_sum)
```

Jupyter Notebook Time

https://github.com/arsenal9971/daedalus-intensive

How to use Tensorflow?

- using the low-level Tensorflow API is flexible but often cumbersome
- there are several high-level APIs wrapping standard operations
- easier to use but offer less control
- `tf.layers` module (will be removed in future versions)
- `tf.estimator` module
- `tf.keras` module (recommended API in future versions)

How to use `tf.keras`?

```python
# define a fully connected layer
# with 128 neurons
from tf.keras.layers import Dense
fully = Dense(128, activation=tf.nn.relu)

# apply the layer to a tensor
out_tensor = fully(in_tensor)
```

How to use `tf.keras`?

```python
# define a 2D convolutional layer
# with 32 output channels
from tf.keras.layers import Conv2D
conv = Conv2D(32, kernel_size=(3, 3),
    activation=tf.nn.relu)

# apply the layer to a tensor
out_tensor = conv(in_tensor)
```

How to use `tf.keras`?

```python
# define a placeholder input layer
# with shape (64, 64)
from tf.keras.layers import Input
input_tensor = Input(shape=(64, 64))
```

How to use `tf.keras`?

```python
# assemble tensors into a model
from tf.keras.models import Model
model = Model(inputs=input_tensor,
    outputs=output_tensor)
```

How to use `tf.keras`?

```python
# compile the model for training
from tf.keras.optimizer import SGD
from tf.keras.losses import MSE
from tf.keras.metrics import MAE
model.compile(
  optimizer=SGD,
  loss=MSE,
  metrics=MAE,
)
```

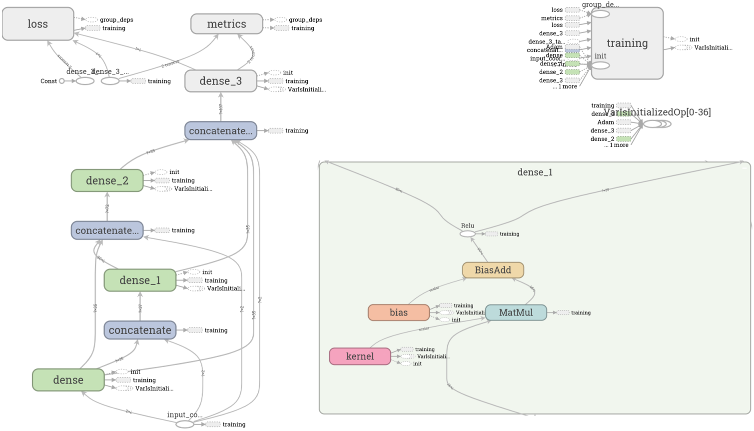
How to use `tf.keras`?

```
# train the model
model.fit(
  input_data,
  output_data,
  batch_size=32,
  epochs=1
)
```

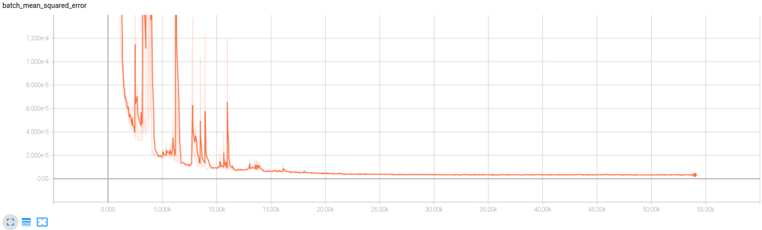
Visualizations of Training and Models

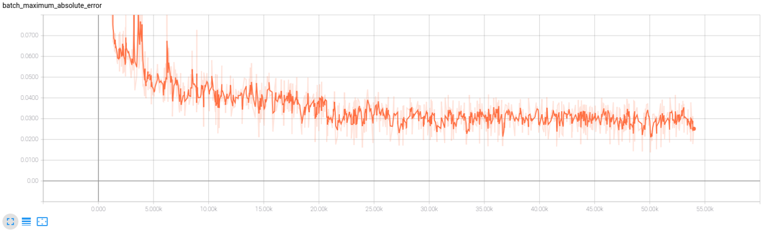
- the Tensorboard module is used for visualizations
- can show the computation graph
- can show the training progress
- ...
- can be easily integrated using the `tf.keras` callback API

Visualizations of Training and Models

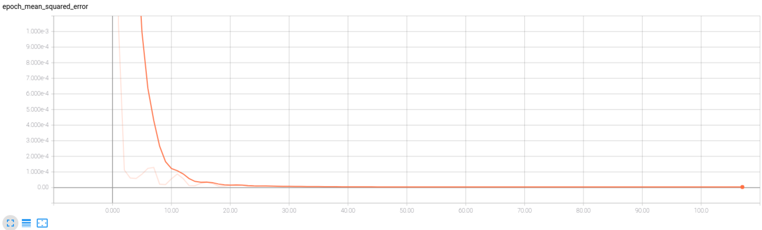


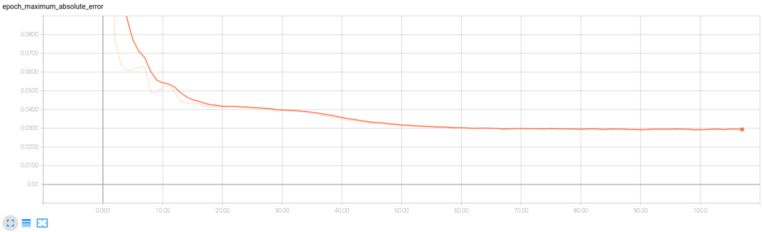
Visualizations of Training and Models





Visualizations of Training and Models





Numerical Experiments

Experimental Setup

- use deep skip-networks as proposed by D.Yarotsky (2017)

activation function	{ReLU, sigmoid}
depth (number of layers L)	$\{2, 3, \ldots, 9\}$
width (neurons per layer)	$\{10, 20, \ldots, 500\}$

- compare to shallow standard networks as in H.N. Mhaskar (1996)

activation function	{ReLU, sigmoid}
depth (number of layers L)	$\{2\}$
width (neurons per layer)	$\{10, 30, \ldots, 9990\}$

Experimental Setup

- target function in $F^{2,2}$ (see Jupyter notebook)
- generate test data from uniform random samples on domain $[0, 1]^2$
- train 200 epochs with 500 batches of size 8192 each
- use $\| \cdot \|_2^2$ as loss function
- use $\| \cdot \|_\infty$ as validation metric
- validate on regular grid with resolution 200×200
- early stopping
- initial learning rate $3 \cdot 10^{-2}$, reduced on plateau

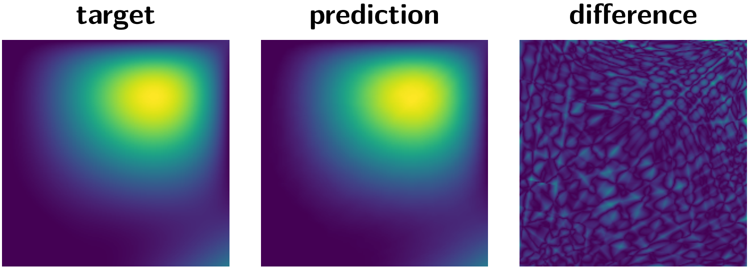
Jupyter Notebook Time

https://github.com/arsenal9971/daedalus-intensive

Results

Best deep ReLU network:

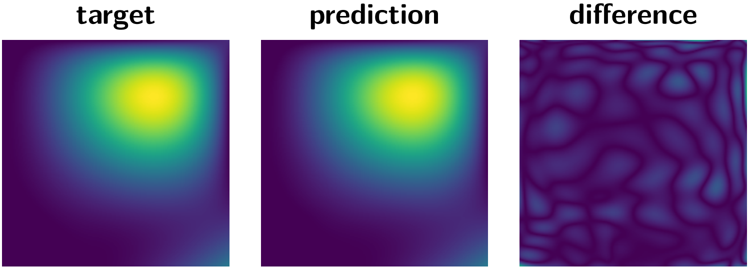
- depth: 6, width: 120, connections: 145802
- $\| \cdot \|_\infty$ error: $3.86 \cdot 10^{-3}$



Results

Best deep sigmoid network:

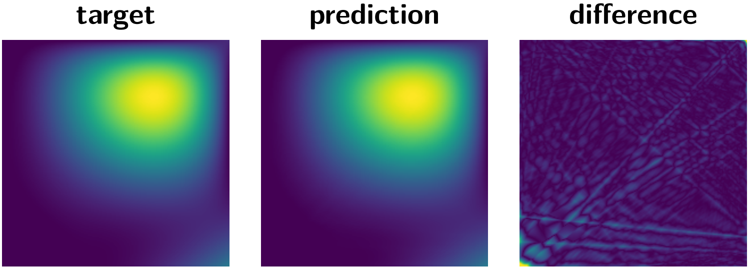
- depth: 6, width: 360, connections: 1301402
- $\|\cdot\|_\infty$ error: $3.12 \cdot 10^{-3}$



Results

Best shallow ReLU network:

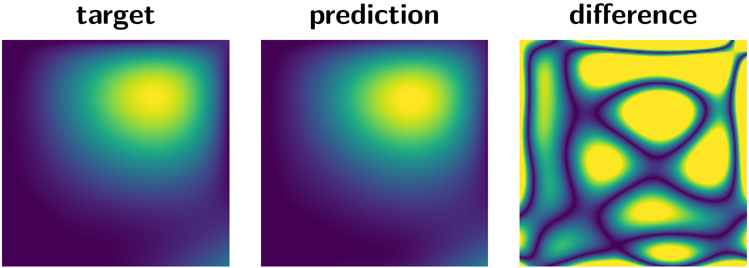
- depth: 2, width: 1790, connections: 5370
- $\| \cdot \|_\infty$ error: $6.88 \cdot 10^{-3}$



Results

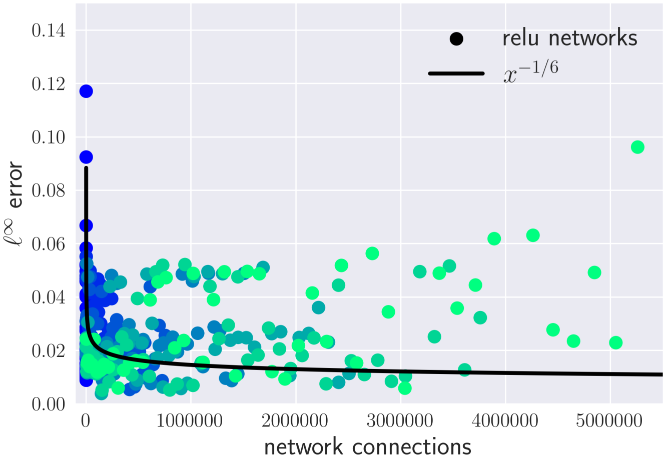
Best shallow sigmoid network:

- depth: 2, width: 4570, connections: 13710
- $\|\cdot\|_\infty$ error: $3.68 \cdot 10^{-2}$



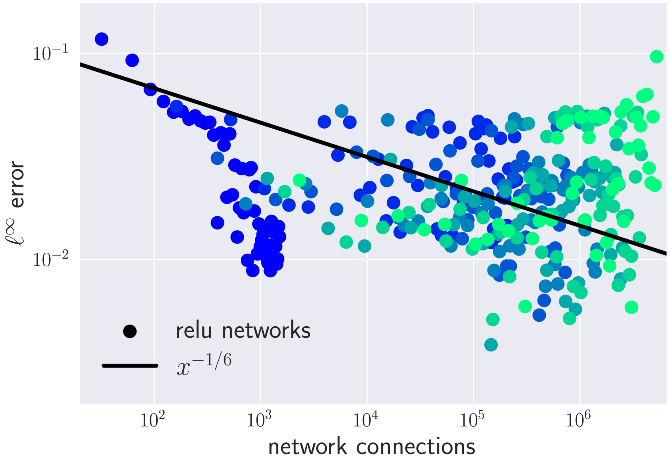
Results

Deep ReLU networks:



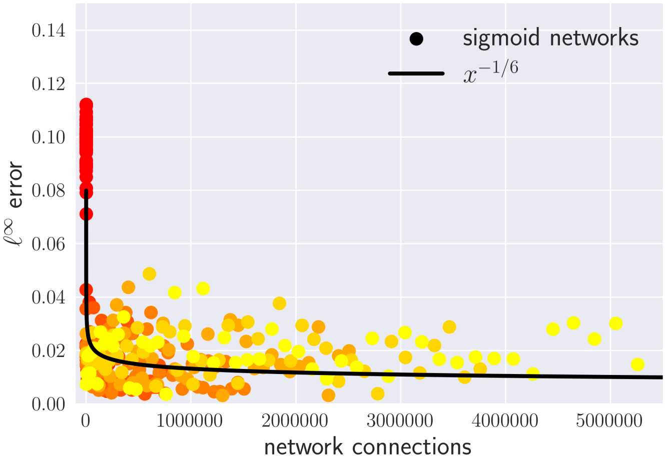
Results

Deep ReLU networks:



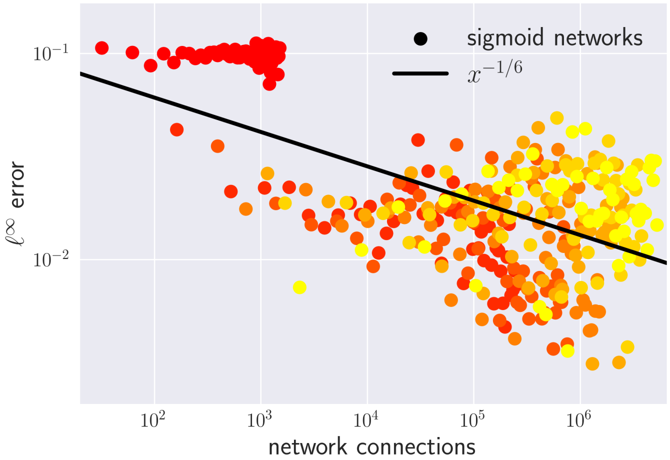
Results

Deep Sigmoid networks:



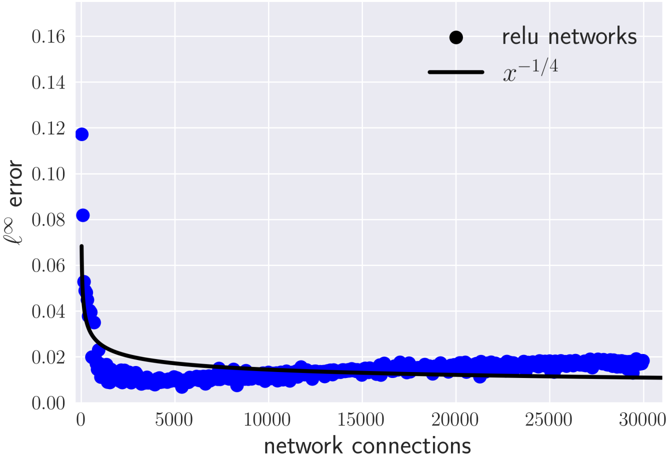
Results

Deep Sigmoid networks:



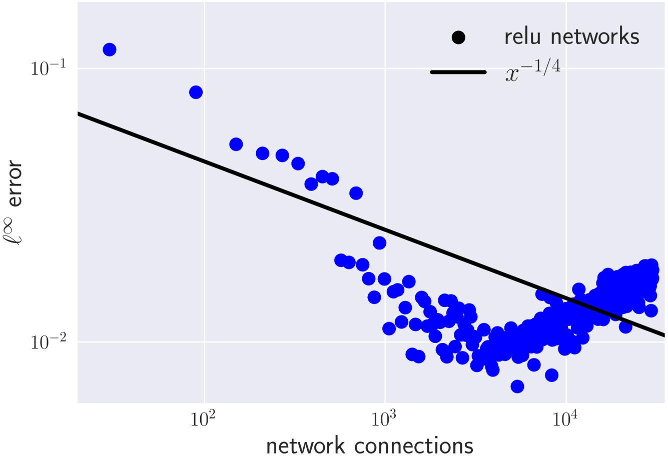
Results

Shallow ReLU networks:



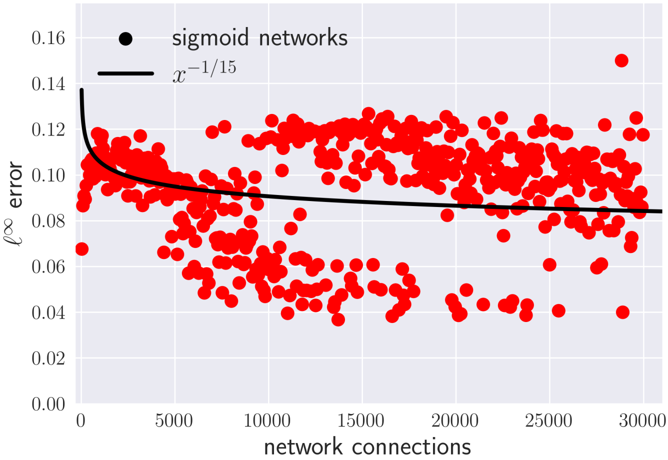
Results

Shallow ReLU networks:



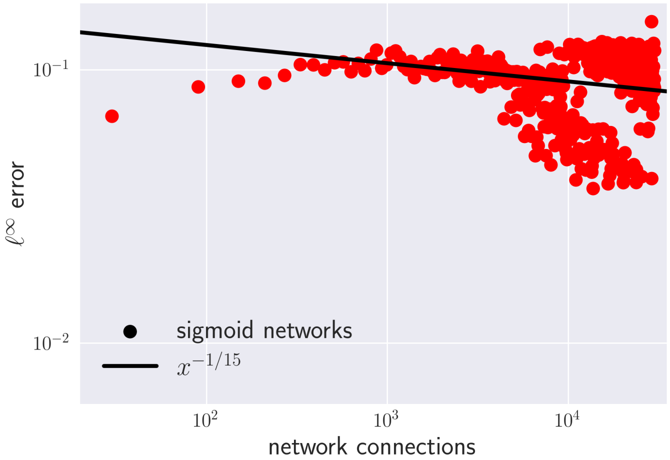
Results

Shallow Sigmoid networks:



Results

Shallow Sigmoid networks:



Conclusions

- similar theoretical approximation rates for deep ReLU networks and shallow sigmoid networks
- difficult to numerically reproduce approximation rates
- large sigmoid networks particularly hard to train

Conclusions

- similar theoretical approximation rates for deep ReLU networks and shallow sigmoid networks
- difficult to numerically reproduce approximation rates
- large sigmoid networks particularly hard to train

Thank you!

References

- M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken.
  Multilayer feedforward networks with a nonpolynomial activation function can approximate any function.
  Neural Networks, 6(6):861 – 867, 1993.
- H. N. Mhaskar.
  Neural networks for optimal approximation of smooth and analytic functions.
  Neural Comput., 8(1):164–177, Jan. 1996.
- P. Petersen and F. Voigtlaender.
  Optimal approximation of piecewise smooth functions using deep ReLU neural networks.
  ArXiv e-prints, Sept. 2017.
- D. Yarotsky.
  Error bounds for approximations with deep relu networks.
  Neural Networks, 94:103 – 114, 2017.