Evolving Architectures of Deep Neural Networks

Petra Vidnerová

Institute of Computer Science
The Czech Academy of Sciences

2018

Outline

Evolving architectures

- Introduction
  - Deep Neural Networks, KERAS library, Related Work
- Our Approach
  - Evolution Strategies, Individuals, Genetic Operators
- Experiments
  - Sensor Data Set, MNIST

Deep RBF Networks

- Introduction
  - RBF Networks, Adversarial Examples
- Deep Neural Network with RBF Layer
- Experiments

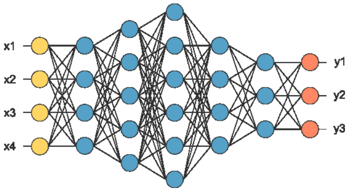
Introduction

Deep Neural Networks

- neural networks with more hidden layers
- convolutional networks - convolutional layers
- our work: feed-forward neural networks, fully connected

Network Architecture

- typically designed by humans
- trial and error method
- our goal: automatic design



Related Work

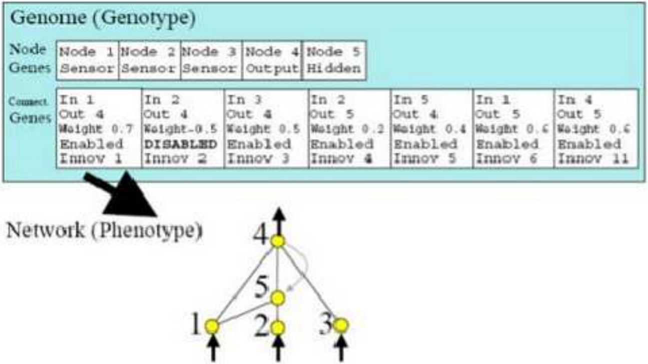
- quite many attemps on architecture optimisation via evolutionary process (NEAT, HyperNEAT, COSyNE)
- neuroevolution - evolving both topology and weights

- various method to represent the architecture by genom
- **direct encoding**
  - binary encoding (Dasgupta and McGregor, 1992, Structured Genetic Algorithm)
  - graph encoding (Pujol and Poli, 1997, Parallel Distributed Genetic Programming)
- **indirect encoding**
  - genoms are programs written in specialized graph transformation language
  - Gruau, 1993, cellular encoding

Related Work

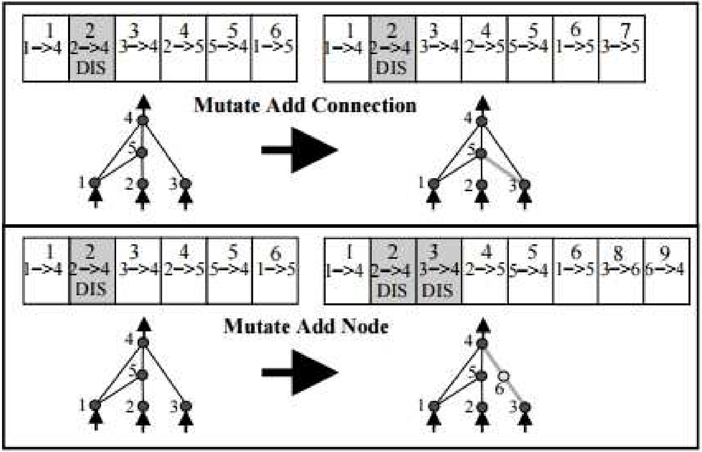
NEAT - NeuroEvolution of Augmenting Topologies

- Ken Stanley, 2002,
  www.cs.ucf.edu/~kstanley/neat.html



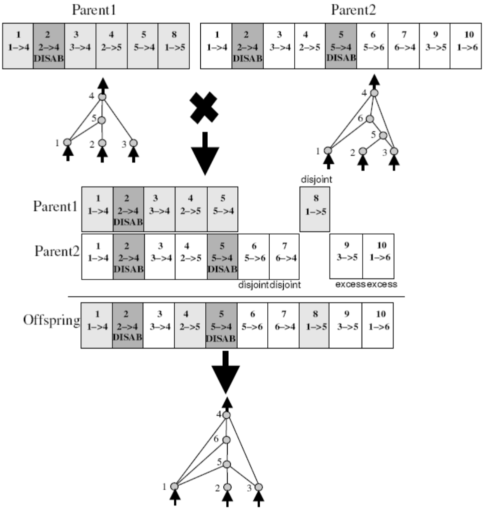
Related Work

NEAT - Mutation



Related Work

NEAT - Crossover



Related Work

NEAT - Main Ideas

- How to crossover disparate topologies in a meaningful way?
  – historical markings to line up genes with the same origin
- How to protect new topological innovations?
  – separate each innovation into a different species
- How to minimize topologies?
  – start from a minimal structure and grow only when necessary

Related Work

Deep Neural Networks

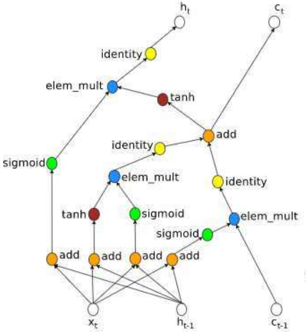
- architecture optimisation for DNN is very time consuming
- works focus on parts of network design
  - I. Loshchilov and F. Hutter, *CMA-ES for hyperparameter optimization of deep neural networks,* 2016
    number of layers fixed, only optimised number of neurons in individual layers, dropout rates, learning rates
  - J. Koutník, J. Schmidhuber, and F. Gomez, *Evolving deep unsupervised convolutional networks for vision-based reinforcement learning,* GECCO '14.
    architecture is fixed, only a small controller evolved

Related Work

- optimising deep learning architectures through evolution
  - R. Miikkulainen, J. Z. Liang, E. Meyerson, A. Rawal, D. Fink, O. Fran- con, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, *Evolving deep neural networks,* 2017
  - DeepNEAT - extending NEAT do deep networks, nodes are layers
  - CoDeepNEAT - two coevolving populations, one of modules, one of blueprints

Related Work

- Google: neural network for neural network design
- NLP task
- artificially designed network gives better results than the one designed by humans



KERAS Library

- widely used tool for practical applications of DNNs

```
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])
```

Our Approach

Keep the search space as simple as possible.

- only architecture is optimized, weights are learned by gradient based technique
- the approach is inpired by and designed for KERAS library
- architecture defined as list of layers, each layer fully connected with next layer (dense layers)
- layer defined by number of neurons, activation function, type of regularization

- future work: add convolutional and max-pooling layers
- metaparameters of learning algorithm (type of algorithm, learning rate, etc.)

Evolutionary Algorithms

- robust optimisation techniques
- work with population of *individuals* representing feasible solutions
- each individual has assigned a *fitness* value
- population evolves by means of *selection, crossover, and mutation*

Evolution Strategies

- initially designed for problems with continuous attributes
- Gaussian mutation is the key operator
- (n + m)-ES or (n, m)-ES

Evolution Strategies

$t = 0$
initialize($P(t)$) # n individuals
evaluate($P(t)$)
while not terminating_criterion **do**
    $P_c(t) \leftarrow$ reproduce(m, $P(t)$)
    evaluate($P_c(t)$)
    if PlusStrategy **then**
        $P(t+1) \leftarrow P_c(t) \cup P(t)$
    else
        $P(t+1) \leftarrow P_c(t)$
    end if
    $P(t+1) \leftarrow$ selectBest(n, $P(t+1)$)
    $t \leftarrow t + 1$
end while

Gaussian mutation

$$\sigma_i \leftarrow \sigma_i \cdot (1 + \alpha \cdot N(0,1))$$
$$x_i \leftarrow x_i + \sigma_i \cdot N(0,1)$$

Individuals for Keras Architectures

- individual - deep neural network architecture

$$
\begin{aligned}
I = (\quad & [size_1, drop_1, act_1, \sigma_1^{size}, \sigma_1^{drop}]_1, \ldots, \\
& [size_H, drop_H, act_H, \sigma_H^{size}, \sigma_H^{drop}]_H \quad),
\end{aligned}
$$

$$
\begin{aligned}
H & \ \ldots \text{number of hidden layers} \\
size_i & \ \ldots \text{size of layer} \\
drop_i & \ \ldots \text{dropout rate} \\
act_i & \ \ldots \text{activation function} \\
\sigma_i^{size}, \sigma_i^{drop} & \ \ldots \text{strategy parameteres}
\end{aligned}
$$

- output layer is softmax or linear (classification or regression task)

Crossover

- one-point crossover working on the whole blocks (layers)

Parents:

$$I_{p1} = (B_1^{p1}, B_2^{p1}, \ldots, B_k^{p1})$$

$$I_{p2} = (B_1^{p2}, B_2^{p2}, \ldots, B_l^{p2}),$$

Offspring:

$$I_{o1} = (B_1^{p1}, \ldots, B_{cp1}^{p1}, B_{cp2+1}^{p2}, \ldots, B_l^{p2})$$

$$I_{o1} = (B_1^{p2}, \ldots, B_{cp2}^{p2}, B_{cp1+1}^{p1}, \ldots, B_k^{p1}).$$

Mutation

- random changes to the individual

Roulette wheel selection of:

- mutateLayer - modifies one randomly selected layer
- addLayer - adds one random layer
- delLayer - deletes one random layer

mutateLayer

- change layer size . . . Gaussian mutation
- change dropout . . . Gaussian mutation
- change activation . . . random choice

Fitness and Selection

Fitness Evaluation

- create network defined by individual
- evaluate crossvalidation error on trainset
- KFold crossvalidation
- for each fold train network using gradient based technique

Tournament selection

- k individuals selected at random, the best one selected for repreduction

Experiment 1: Sensor Data

Target application - Air Pollution Prediction

- a real-world data set from the application area of sensor networks for air pollution monitoring
- concentration of several gas pollutants
- 8 input values - 5 sensors, temperature, absolute and relative humidity
- 1 predicted value - concentration of CO, NO2, NOx, C6H6, and NMHC

Sensor Data Set

- First task - whole time period divided into five intervals, one for training, the rest for testing
- Second task - data for training and testing selected at random

	First experiment		**Second experiment**	
Task	train set	test set	train set	test set
CO	1469	5875	4896	2448
NO2	1479	5914	4929	2464
NOx	1480	5916	4931	2465
C6H6	1799	7192	5994	2997
NMHC	178	709	592	295

Parameter setup

Main GA	n	(n,m) ES	10
	m	(n,m) ES	30
	ng	number of generations	100
	p_{cx}	crossover probability	0.6
	p_{mut}	mutation probability	0.2
Individual	$nlayers$	max number of layers	5
	max_lsize	max layer size	100
	min_lsize	minimum layer size	5
Fitness	k	k-fold crossover	5
Selection	k	tournament of k individuals	3

Activation functions: relu, tanh, sigmoid, hard sigmoid, linear
Learning algorithm: RMSprop

Experimental Results: ES vs. GA

	GA				ES			
	avg	std	min	max	avg	std	min	max
CO part1	**0.209**	0.014	0.188	0.236	0.229	0.026	0.195	0.267
CO part2	0.801	0.135	0.600	1.048	**0.657**	0.024	0.631	0.694
CO part3	0.266	0.029	0.222	0.309	**0.256**	0.045	0.199	0.349
CO part4	**0.404**	0.226	0.186	0.865	0.526	0.108	0.308	0.701
CO part5	0.246	0.024	0.207	0.286	**0.235**	0.025	0.199	0.277
NOx part1	2.201	0.131	1.994	2.506	**2.132**	0.086	2.021	2.284
NOx part2	1.705	0.284	1.239	2.282	**1.599**	0.077	1.444	1.685
NOx part3	**1.238**	0.163	0.982	1.533	1.339	0.242	1.106	1.955
NOx part4	**1.490**	0.173	1.174	1.835	1.610	0.164	1.435	2.041
NOx part5	**0.551**	0.052	0.456	0.642	0.622	0.075	0.521	0.726
NO2 part1	1.697	0.266	1.202	2.210	**1.506**	0.217	1.132	1.823
NO2 part2	2.009	0.415	1.326	2.944	**1.371**	0.048	1.242	1.415
NO2 part3	**0.593**	0.082	0.532	0.815	0.660	0.078	0.599	0.863
NO2 part4	**0.737**	0.023	0.706	0.776	0.782	0.043	0.711	0.856
NO2 part5	1.265	0.158	1.054	1.580	**0.730**	0.111	0.520	0.905
C6H6 part1	**0.013**	0.005	0.006	0.024	**0.013**	0.004	0.007	0.018
C6H6 part2	0.039	0.015	0.025	0.079	**0.034**	0.010	0.020	0.050
C6H6 part3	**0.019**	0.011	0.009	0.041	0.048	0.015	0.016	0.075
C6H6 part4	0.030	0.015	0.014	0.061	**0.020**	0.010	0.010	0.042
C6H6 part5	**0.017**	0.015	0.004	0.051	0.027	0.011	0.014	0.051
NMHC part1	1.719	0.168	1.412	2.000	**1.685**	0.256	1.448	2.378
NMHC part2	**0.623**	0.164	0.446	1.047	0.713	0.097	0.566	0.865
NMHC part3	1.144	0.181	0.912	1.472	**1.097**	0.270	0.775	1.560
NMHC part4	1.220	0.206	0.994	1.563	**1.099**	0.166	0.898	1.443
NMHC part5	1.222	0.126	1.055	1.447	**1.023**	0.050	0.963	1.116
	11				**15**			
	44%				**60%**			

Experiments Results: Evolved vs. SVR

Testing errors

Task	Evolved				SVR			
	avg	std	min	max	linear	RBF	Poly.	Sigmoid
CO_part1	**0.229**	0.026	0.195	0.267	0.340	0.280	0.285	1.533
CO_part2	0.657	0.024	0.631	0.694	0.614	**0.412**	0.621	1.753
CO_part3	**0.256**	0.045	0.199	0.349	0.314	0.408	0.377	1.427
CO_part4	**0.526**	0.108	0.308	0.701	1.127	0.692	0.535	1.375
CO_part5	0.235	0.025	0.199	0.277	0.348	0.207	**0.198**	1.568
NOx_part1	2.132	0.086	2.021	2.284	**1.062**	1.447	1.202	2.537
NOx_part2	1.599	0.077	1.444	1.685	2.162	1.838	**1.387**	2.428
NOx_part3	1.339	0.242	1.106	1.955	**0.594**	0.674	0.665	2.705
NOx_part4	1.610	0.164	1.435	2.041	0.864	0.903	**0.778**	2.462
NOx_part5	**0.622**	0.075	0.521	0.726	1.632	0.730	1.446	2.761
NO2_part1	**1.506**	0.217	1.132	1.823	2.464	2.404	2.401	2.636
NO2_part2	**1.371**	0.048	1.242	1.415	2.118	2.250	2.409	2.648
NO2_part3	**0.660**	0.078	0.599	0.863	1.308	1.195	1.213	1.984
NO2_part4	**0.782**	0.043	0.711	0.856	1.978	2.565	1.912	2.531
NO2_part5	**0.730**	0.111	0.520	0.905	1.0773	1.047	0.967	2.129
C6H6_part1	**0.013**	0.004	0.007	0.018	0.300	0.511	0.219	1.398
C6H6_part2	**0.034**	0.010	0.020	0.050	0.378	0.489	0.369	1.478
C6H6_part3	**0.048**	0.015	0.016	0.075	0.520	0.663	0.538	1.317
C6H6_part4	**0.020**	0.010	0.010	0.042	0.217	0.459	0.123	1.279
C6H6_part5	**0.027**	0.011	0.014	0.051	0.215	0.297	0.188	1.526
NMHC_part1	1.685	0.256	1.448	2.378	1.718	1.666	**1.621**	3.861
NMHC_part2	**0.713**	0.097	0.566	0.865	0.934	0.978	0.839	3.651
NMHC_part3	**1.097**	0.270	0.775	1.560	1.580	1.280	1.438	2.830
NMHC_part4	**1.099**	0.166	0.898	1.443	1.720	1.565	1.917	2.715
NMHC_part5	1.023	0.050	0.963	1.116	1.238	**0.944**	1.407	2.960
	17				2	2	4	
	68%				8%	8%	16%	

Experimental Results: Evolved Architectures

- evolved networks are quite small
- typical network:
  - one hidden layer of about 70 neurons
  - dropout rate 0.3
  - ReLU activation function.

Experiments Results: Evolved vs. fixed architecture

Testing errors

Task	Evolved avg	Evolved std	50-1 avg	50-1 std	30-10-1 avg	30-10-1 std	30-10-30-1 avg	30-10-30-1 std
CO_part1	**0.229**	0.026	0.230	0.032	0.250	0.023	0.377	0.103
CO_part2	**0.657**	0.024	0.861	0.136	0.744	0.142	0.858	0.173
CO_part3	**0.256**	0.045	0.261	0.040	0.305	0.043	0.302	0.046
CO_part4	0.526	0.108	0.621	0.279	0.638	0.213	**0.454**	0.158
CO_part5	**0.235**	0.025	0.283	0.072	0.270	0.032	0.309	0.032
NOx_part1	2.132	0.086	2.158	0.203	**2.095**	0.131	2.307	0.196
NOx_part2	**1.599**	0.077	1.799	0.313	1.891	0.199	2.083	0.172
NOx_part3	1.339	0.242	1.077	0.125	1.092	0.178	**0.806**	0.185
NOx_part4	1.610	0.164	**1.303**	0.208	1.797	0.461	1.600	0.643
NOx_part5	**0.622**	0.075	0.644	0.075	0.677	0.055	0.778	0.054
NO2_part1	1.506	0.217	1.659	0.250	**1.368**	0.135	1.677	0.233
NO2_part2	**1.371**	0.048	1.762	0.237	1.687	0.202	1.827	0.264
NO2_part3	0.660	0.078	0.682	0.148	**0.576**	0.044	0.603	0.069
NO2_part4	0.782	0.043	1.109	0.923	**0.757**	0.059	0.802	0.076
NO2_part5	0.730	0.111	**0.646**	0.064	0.734	0.107	0.748	0.123
C6H6_part1	0.013	0.004	**0.012**	0.006	0.081	0.030	0.190	0.060
C6H6_part2	**0.034**	0.010	0.039	0.012	0.101	0.015	0.211	0.071
C6H6_part3	0.048	0.015	**0.024**	0.007	0.091	0.047	0.115	0.031
C6H6_part4	**0.020**	0.010	0.026	0.010	0.051	0.026	0.096	0.020
C6H6_part5	0.027	0.011	**0.025**	0.008	0.113	0.025	0.176	0.058
NMHC_part1	**1.685**	0.256	1.738	0.144	1.889	0.119	2.378	0.208
NMHC_part2	0.713	0.097	**0.553**	0.045	0.650	0.078	0.799	0.096
NMHC_part3	1.097	0.270	1.128	0.089	0.901	0.124	**0.789**	0.184
NMHC_part4	1.099	0.166	1.116	0.119	0.918	0.119	**0.751**	0.096
NMHC_part5	1.023	0.050	0.970	0.094	0.889	0.085	**0.856**	0.074
	10		6		4		5	
	40%		24%		16%		20%	

Experimental Results: Sensors Second Task

Testing errors

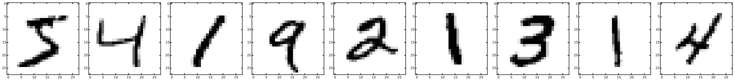
Task	Evolved		SVR			
	avg	std	linear	RBF	Poly.	Sigmoid
CO	**0.115**	0.006	0.200	0.152	0.157	1.511
NOx	**0.128**	0.016	0.328	0.211	0.255	1.989
NO2	**0.276**	0.012	0.494	0.368	0.406	2.046
C6H6	**0.001**	0.001	0.218	0.110	0.194	1.325
NMHC	**0.247**	0.058	0.688	0.383	0.513	3.215

- evolved networks several layers, dominating activation function ReLU

Experiment 2: MNIST

Data Set

- well known data set, classification of hand written digits
- 28 $\times$ 28 pixels
- 60000 for training, 10000 for testing



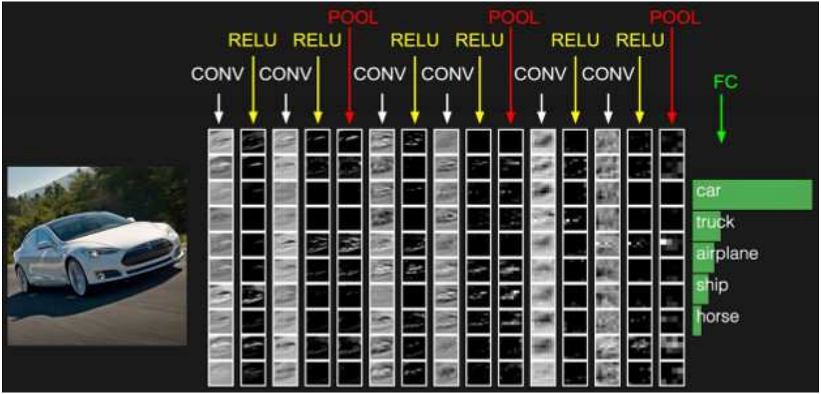
Results

- ES run for 30 generations, n = 5, m = 10

model	avg	std	min	max
baseline	98.34	0.13	98.18	98.55
evolved	**98.64**	0.05	98.55	98.73

Convolutional Neural Networks

- convolutional layers
- max-pooling layers



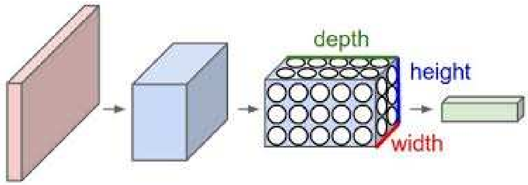
Convolutional Networks in Keras

```
model = Sequential()
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1],
                        border_mode='valid',
                        input_shape=input_shape))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

- each block has its type: convolutional, max-pooling, dense

Convolution Networks - Individuals



- individuals consists of two parts convolutional and dense
- convolutional part - convolutional and max-pooling layers - feature extraction
- dense part - only dense layers - classification

Individuals

Convolutional layer

- number of filters
- kernel size
- activation function type

Max-pooling layer

- pool-size

Genetic operators

- crossover convolutional and dense part separately
- mutation stays the same

Parallel approach

Classic approach

- very time consuming
- each fitness evaluation includes crossvalidation

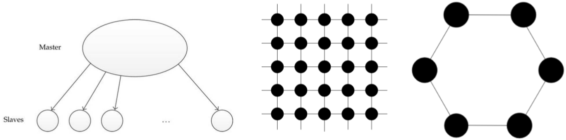
Parallel approach

- GA are easy to paralelize
- fitness evaluations are independent
- can be done in parallel

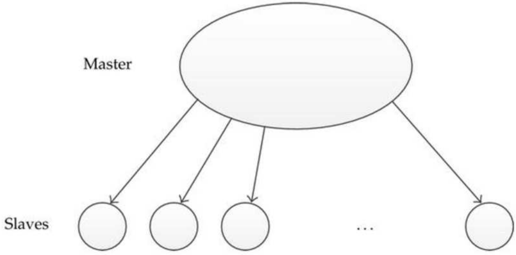
Parallel GA

- basic idea of parallel programs - divide-and-conquer approach
- can be applied to GAs in many different ways

- three main types of parallel GA:
  - global single-population master-slave GAs
  - single population fine-grained
  - multi-population coarse-grained



Master-slave parallel GA



- one population stored on the master
- master executes GA operations
- slaves only evaluate the fitness of individuals
- does not effect the algorithm
- easy to implement

Single Population Fine-grained GA

- suited for massively parallel computers
- one population spread over processors
- spatial structure that limits interactions between individuals
- individual only competes and mates with its neighbors
- neighborhoods overlap - good solutions may disseminate across the entire population
- ideal case - only one individual for every processing element

Multi-Population (Coarse Grained) GA

- consist of several subpopulations which exchange individuals occasionally (migration)
- e.g. copy of the best individual in each deme is sent to all neighbors every generation
- simple extension of the serial GA
- parameters:
  - topology
  - migration rate
  - migration interval
- migration - synchronous, asynchronous
- topology - how fast a good solution disseminates to other demes

Asynchronous evolution

- individuals evaluated one by one
- no notion of generation
- as soon as there is an idle processor, new individual is created
- arbitrary number of processors

1. get evaluated individual I
2. append I to the population
3. discard the worst individual
4. generate new individual I' by genetic operators
5. send I' for fitness evaluation

MNIST Results

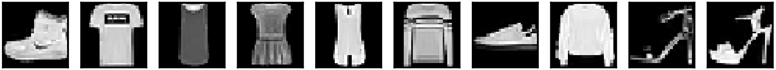
- asynchronous evolution
- population size 20
- 20 *generations*

model	avg	std	min	max
baseline	98.97	0.07	98.84	99.13
evolved	**99.17**	0.11	98.92	99.36

Fashion-MNIST Results

Data Set

- alternative to MNIST
- 28×28 pixels, 10 classes
- 60000 for training, 10000 for testing



Results

model	avg	std	min	max
baseline	91.64	0.37	90.77	91.97
evolved	**92.32**	0.52	91.07	92.86

Conlusion and Future Work

- proposed ES for DNN architecture design
- demonstrated the algorithm on experiments
- works for feed-forward DNN with dense layers, convolutional networks

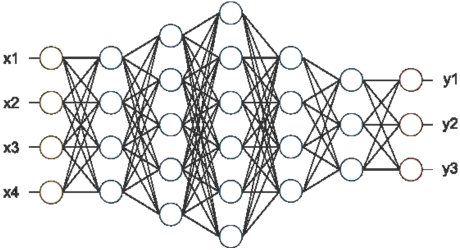
Future Work

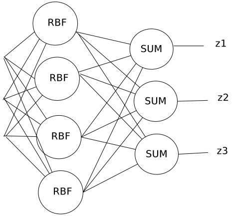
- evolve also other parameters of learning
- speed up the evolution - asynchronous evolution, surrogate modeling

Deep Networks and RBF Networks

- combinations of Deep Networks and RBF Networks
- RBF layers can be also included in evolution

- RBF networks less vulnerable to adversarial examples
- Does add RBF layers to deep network help to prevent adversarial examples?





RBF Networks

- feed-forward neural networks with one hidden layer of RBF units
- local units alternative to MLP
- RBF unit:

$$y = \varphi(\xi); \quad \xi = \beta||\vec{x} - \vec{c}||^2$$

  where $\varphi : \mathbb{R} \to \mathbb{R}$ is suitable activation function, typically Gaussian $\varphi(z) = e^{-z^2}$.
- the network computes the function $\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$:

$$f_s(\vec{x}) = \sum_{j=1}^{h} w_{js}\varphi\left(\beta_j \parallel \vec{x} - c_j \parallel\right)$$

RBF Networks Learning

- wide range of methods

Three Step Learning

1. set the centers - approximate the distribution of training samples
   - random or uniform samples, various clustering methods
2. set the widths - cover the input space by unit's fields
   - heuristics (k-neighbours)
3. compute the output weights
   - linear system, pseudoinverse

Gradient Learning

- analogous to backpropagation for MLP

Adversarial examples

- Applying an imperceptible non-random perturbation to an input image, it is possible to arbitrarily change the machine learning model prediction.

57.7% Panda 99.3% Gibbon

Figure from *Explaining and Harnessing Adversarial Examples* by Goodfellow et al.

- Such perturbed examples are known as *adversarial examples*. For human eye, they seem close to the original examples.
- They represent a security flaw in classifier.

Crafting adversarial examples

- several methods for crafting adversarial examples
- FGSM - Fast Gradient Sign Method
  Ian J. Goodfellow et al., Explaining and Harnessing Adversarial Examples, 2014, arXiv:1412.6572

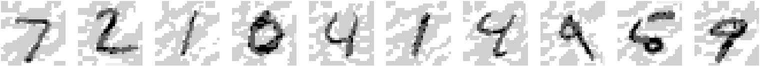
$$\eta = \varepsilon \, \mathrm{sgn}(\bigtriangledown_x J(\theta, x, y))$$

- our work - using Genetic Algorithm to craft adversarial examples
  does not need access to models weights

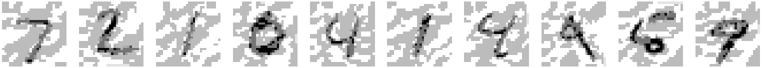
Adversarial examples by FGSM

Legitimate samples:

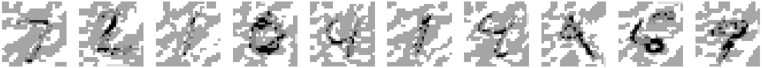
Adversarial samples $\epsilon = 0.2$:



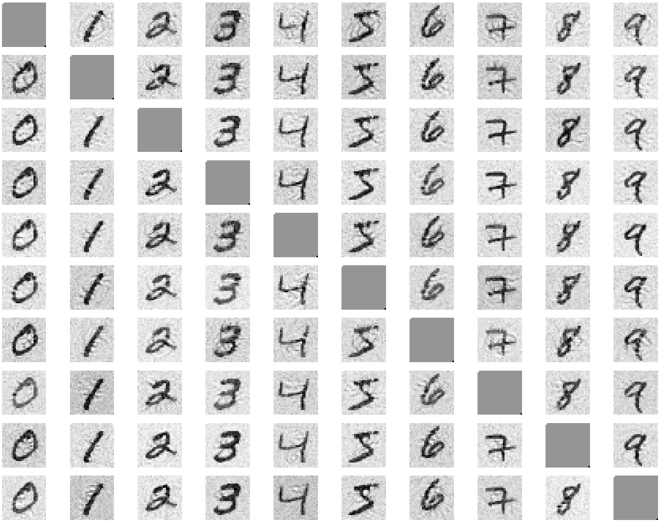
Adversarial samples $\epsilon = 0.3$:



Adversarial samples $\epsilon = 0.4$:

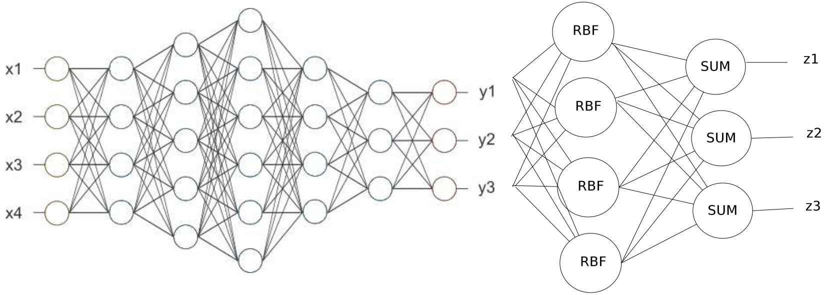


Adversarial examples by GA



Proposed architecture DNNRBF

- stacking deep neural network and RBF network



DNNRBF learning

1. train the *DNN*
2. set the centers of *RBF* randomly, drawn from uniform distribution on $(0, 1.0)$
3. set the parameters β to the constant value
4. init the weights of RBF output layer to random small values
5. retrain the whole network DNNRBF (by back propagation)

Experiments

Architectures

- MLP
  - dense layer of 512 ReLU
  - dense layer of 512 ReLU
  - dense layer of 10 softmax units

- CNN
  - convolutional layer with 32 3x3 filters and ReLU activation
  - convolutional layer with 32 3x3 filters and ReLU activation
  - 2x2 max pooling layer
  - dense layer of 128 ReLU
  - dense layer of 10 softmax units

Experiments

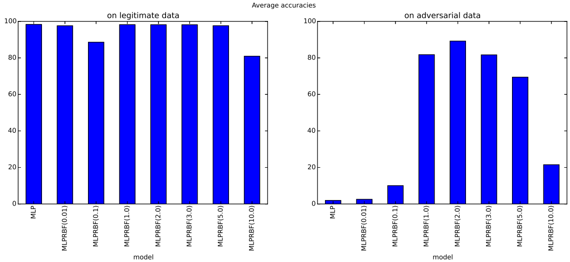
Implementation

- FGSM for crafting adversarial examples
  Cleverhans library: *cleverhans v2.0.0: an adversarial machine learning library, Nicolas Papernot, et al., arXiv preprint arXiv:1610.00768, 2017*
- Keras for MLP and CNN
  Keras, François Chollet, `https://github.com/fchollet/keras`*, 2015*
- our implementation of RBF Keras layers
  `http://github.com/PetraVidnerova/rbf_keras`

`http://github.com/PetraVidnerova/rbf_tests`

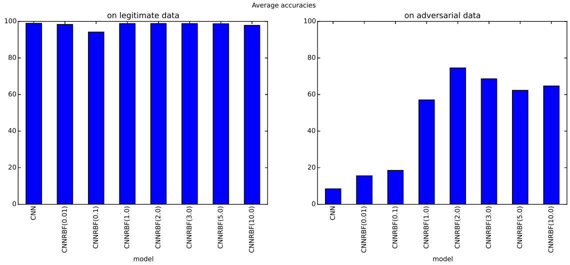
Experiments Results- MLP

	Legitimate samples				Adversarial samples			
model	mean	std	min	max	mean	std	min	max
MLP	**98.35**	**0.12**	**98.04**	**98.59**	**1.95**	**0.41**	**1.30**	**2.86**
MLPRBF(0.01)	97.62	2.43	88.44	98.65	2.56	2.09	1.16	10.71
MLPRBF(0.1)	88.61	8.56	69.91	98.36	10.04	6.45	1.71	23.10
MLPRBF(1.0)	98.23	0.10	98.08	98.48	81.77	7.84	64.18	94.06
MLPRBF(2.0)	**98.19**	**0.14**	**97.91**	**98.38**	**89.21**	**5.03**	**66.28**	**94.83**
MLPRBF(3.0)	98.18	0.14	97.88	98.45	81.66	4.38	70.13	87.23
MLPRBF(5.0)	97.64	2.09	89.34	98.36	69.47	13.26	13.01	81.95
MLPRBF(10.0)	80.94	11.82	58.57	98.33	21.49	16.32	2.48	65.11



Experiments Results - CNN

	Legitimate samples				Adversarial samples			
model	mean	std	min	max	mean	std	min	max
CNN	**98.97**	**0.07**	**98.84**	**99.13**	**8.49**	**3.52**	**3.11**	**16.43**
CNNRBF(0.01)	98.36	1.73	89.12	99.01	15.60	4.28	10.26	28.44
CNNRBF(0.1)	94.19	8.21	58.88	98.92	18.58	6.42	6.01	31.29
CNNRBF(1.0)	98.83	0.13	98.46	99.04	57.09	9.23	33.39	78.99
CNNRBF(2.0)	**98.85**	**0.13**	**98.38**	**99.09**	**74.57**	**7.69**	**53.07**	**84.67**
CNNRBF(3.0)	98.82	0.14	98.55	99.10	68.65	7.77	44.36	80.13
CNNRBF(5.0)	98.74	0.11	98.49	98.94	62.35	7.04	48.03	77.04
CNNRBF(10.0)	97.86	2.24	89.33	98.84	64.71	8.32	46.61	79.89



Experiments Results

model	Accuracy on adversarial data					
	$\epsilon = 0.2$		$\epsilon = 0.3$		$\epsilon = 0.4$	
	avg	std	avg	std	avg	std
CNN	33.85	7.58	8.49	3.52	4.34	1.71
CNNRBF	76.88	6.25	74.57	7.69	73.51	8.08
MLP	3.01	0.69	1.95	0.41	1.66	0.38
MLPRBF	90.14	4.82	89.21	5.03	88.27	5.14

Thank you!   Questions?