# Asynchronous Evolution of Convolutional Networks

Petra Vidnerová

Institute of Computer Science
The Czech Academy of Sciences

ITAT 2018

# Outline

- Introduction
  - Related Work
  - Convolutional Networks, Keras
- Our Approach
  - Coding of Individuals
  - Genetic Operators
- Parallel Approch
  - Asynchronous Evolution
- Experiments
  - MNIST, Fashion-MNIST
- Conclusion

# Introduction

## Convolutional Neural Networks

- subset of deep neural networks
- convolutional networks - convolutional layers
- our work: evolving architecture of convolutional networks

## 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
- 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

## Autokeras

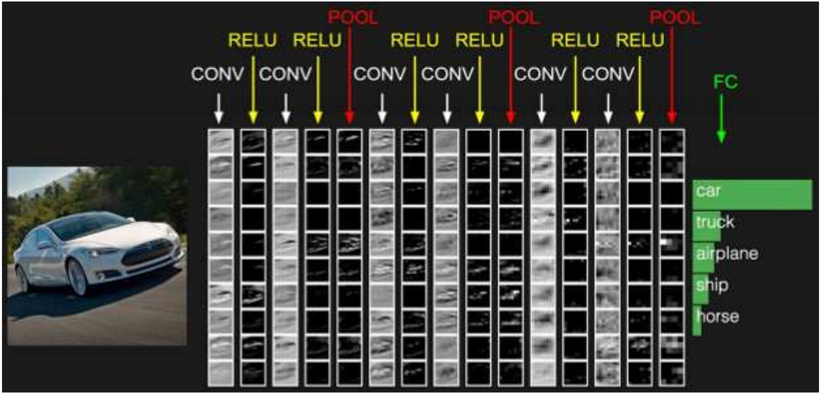
- *Efficient Neural Architecture Search with Network Morphism.* Haifeng Jin, Qingquan Song, and Xia Hu. arXiv:1806.10282.
- uses Bayesitan optimisation to select network morphism operation

```
import autokeras as ak

clf = ak.ImageClassifier()
clf.fit(x_train, y_train)
results = clf.predict(x_test)
```

# Convolutional Neural Networks

- convolutional layers
- max-pooling layers



# Convolutional Networks in Keras

- Keras - widely used tool for implementing deep neural networks

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

# 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
- dense, convolutional, max-pooling layers
- layer defined by number of neurons/number of filters, size of filter, size of pool, activation function, type of regularization

- future work: metaparameters of learning algorithm (type of algorithm, learning rate, etc.)

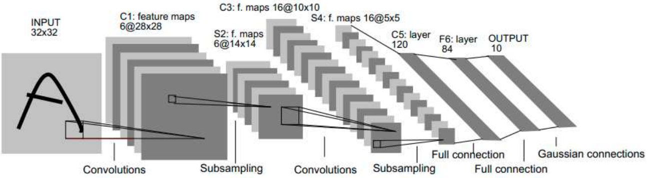
# Evolutionary Algorithms

- 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*

## Our previous work

- classical GA for DNN (FedCSIS 2017)
- evolution strategies for DNN (ITAT 2017)

# Convolution Networks - Individuals




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

# Coding of Individuals

$$
\begin{aligned}
I &= (I_{conv}, I_{dense}),\\
I_{conv} &= ([type, params]_1, \ldots, [type, params]_{H1})\\
I_{dense} &= ([size, dropout, act]_1, \ldots, [size, dropout, act]_{H2})
\end{aligned}
$$

- $I_1$ and $I_2$ - convolutional and dense part
- $H_1$ and $H_2$ corresponding number of layers

- $type \in \{\mathtt{convolutional}, \mathtt{max-pooling}\}$
- convolutional parameters: number of filters, size of filter, activation function
- max-pooling parameters: size of pool
- $act \in \{\mathtt{relu}, \mathtt{tanh}, \mathtt{sigmoid}, \mathtt{hardsigmoid}, \mathtt{linear}\}$

# 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, number of filters, filter size, pool size
- change dropout
- change activation

# 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

# Parallel approach

## Classic approach

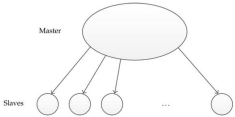
- very time consuming
- each fitness evaluation includes crossvalidation

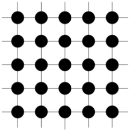
## Parallel approach

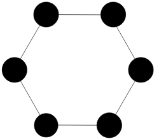
- GA are easy to paralelize
- fitness evaluations are independent
- can be done in parallel

# Parallel GA

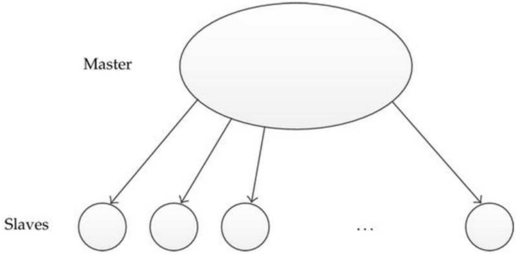
- basic idea of parallel programs - divide-and-conquer approach
- can be applied to GAs in many different ways
- three main types of parallel GA:
  - global single-population master-slave GAs
  - single population fine-grained
  - multi-population coarse-grained





# 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

# Our parallel implementation - Master-slave

- we use master-slave approach
- fitness is evaluated in parallel

## Disadvantage

- individuals are networks of different sizes
- some evaluate faster than others
- in each generation some processors idle for a period of time

# Asynchronous evolution

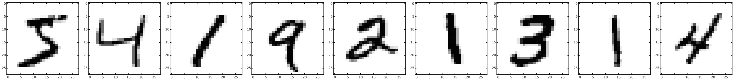
- individuals evaluated one by one
- no notion of generation
- as soon as there is an idle processor, new individual is created
- arbitrary number of processors
- slightly prefers smaller networks

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

# Experiments

## MNIST dataset

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



# 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 |

# MNIST Results – Architectures

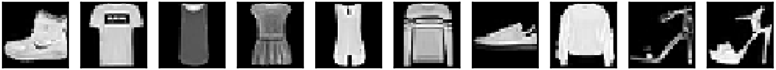
| Baseline network |
|---|
| conv #32 kernelsize=3 activation=relu |
| conv #32 kernelsize=3 activation=relu |
| pool poolsize=2 |
| dense #128 dropout=0.5 activation=relu |
| Trainable params: 600,810 |

| Evolved network |
|---|
| conv #22 kernelsize=2 activation=tanh |
| conv #31 kernelsize=5 activation=linear |
| pool poolsize=3 |
| conv #33 kernelsize=5 activation=relu |
| dense #143 dropout=0.4 activation=relu |
| dense #42 dropout=0.0 activation=tanh |
| Trainable params: 431,659 |

# Fashion-MNIST Results

## Data Set

- alternative to MNIST
- $28 \times 28$ 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 |

# Fashion MNIST Results – Architectures

| Baseline network |
| --- |
| conv #32 kernelsize=3 activation=leakyRelu |
| pool poolsize=2 |
| conv #64 kernelsize=3 activation=leakyRelu |
| pool poolsize=2 |
| conv #128 kernelsize=3 activation=leakyRelu |
| pool poolsize=2 |
| dense #128 dropout=0.3 activation=leakyRelu |
| Trainable params: 356,234 |

# Fashion MNIST Results – Architectures

| Evolved network |
|---|
| conv #46 kernelsize=3 activation=relu |
| conv #15 kernelsize=3 activation=relu |
| conv #36 kernelsize=4 activation=relu |
| conv #13 kernelsize=3 activation=relu |
| conv #36 kernelsize=3 activation=relu |
| pool poolsize=2 |
| dense #235 dropout=0.4 activation=hard_sigmoid |
| dense #130 dropout=0.3 activation=tanh |
| Trainable params: 1,714,219 |

## Synchronous vs. Asynchronous Approach

- both algorithms run on 5 processors for 4 days with population size 20
- asychronous approach: 140 fitness evaluations
- synchronous approach: 100 fitness evalutations

# Conlusion and Future Work

- proposed algorithm for CNN architecture design
- demonstrated the algorithm on experiments

## Future Work

- compare our approach and autokeras
- evolve also other parameters of learning
- multi-criteria evolution
- speed up the evolution - surrogate modeling

Thank you! Questions?