Evolutionary Computation for Deep Neural Networks

Speaker:
Francisco Erivaldo Fernandes Junior
Ph.D. Candidate in Electrical Engineering
feferna@okstate.edu

Instructor:
Dr. Gary G. Yen
Regents Professor
gyen@okstate.edu

School of Electrical and Computer Engineering
Oklahoma State University
October 31, 2019

Outline

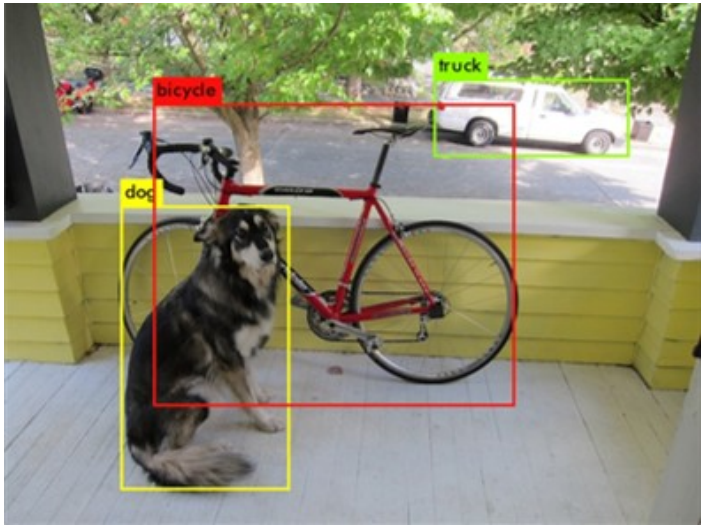
- Introduction and Motivation
- Related Works
- Convolutional Neural Network Architecture Search Based on Particle Swarm Optimization
- Deep Neural Network Pruning with Evolution Strategy
- Conclusions

Outline

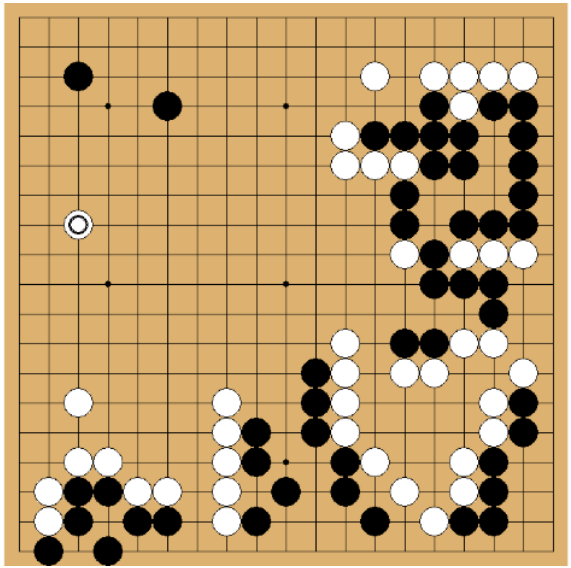
- Introduction and Motivation
- Related Works
- Convolutional Neural Network Architecture Search Based on Particle Swarm Optimization
- Deep Neural Network Pruning with Evolution Strategy
- Conclusions

Introduction and Motivation

- Deep Neural Networks are currently the most used models to solve a variety of difficult real-world problems.



Object classification and localization[1]



Intelligent game playing[2]

Machine translation[3]

[1]Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: arXiv (2018)
[2]Jeffrey Barratt and Chuanbo Pan. "Playing Go without Game Tree Search Using Convolutional Neural Networks". In: arXiv:1907.04658 [cs] (July 2019)
[3]https://blog.webcertain.com/machine-translation-technology-the-search-engine-takeover/18/02/2015/

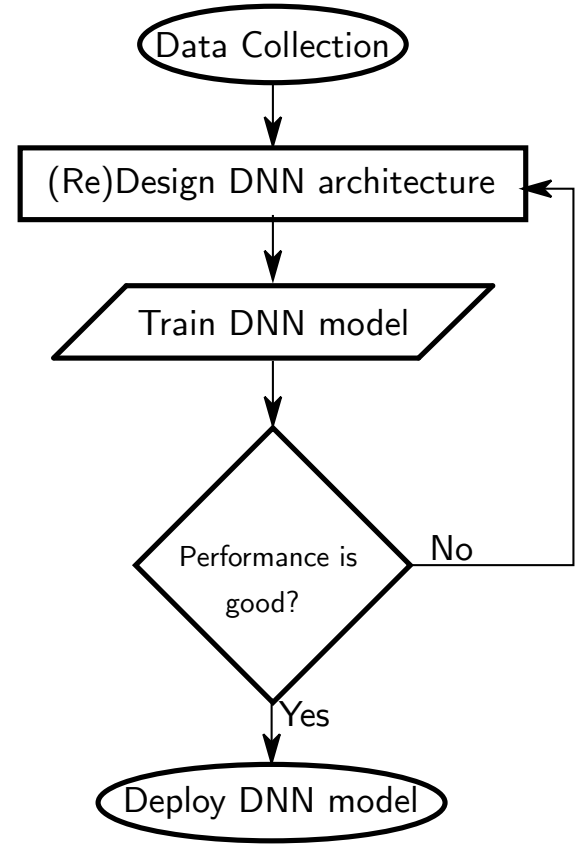
Introduction and Motivation

- Deep neural networks (DNNs) are models comprised of stacked layers of weight operations and activation functions.

- **Weight Operations:**
  - Matrices Multiplications;
  - Convolutional Operations;
  - Downsampling Operations.
- **Activation Functions:**
  - Hyperbolic Tangent;
  - Sigmoid;
  - Rectified Linear Unit (ReLU).

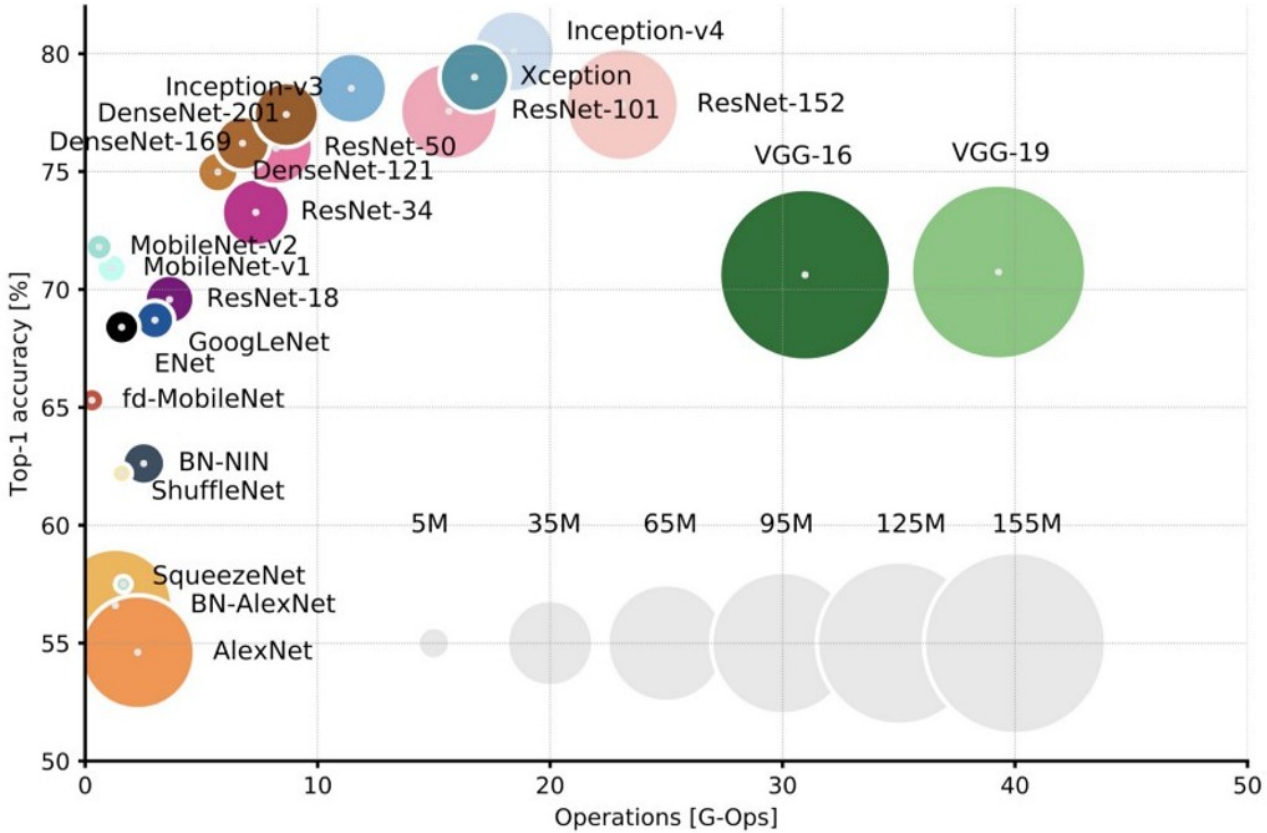
DNN Designing Challenges

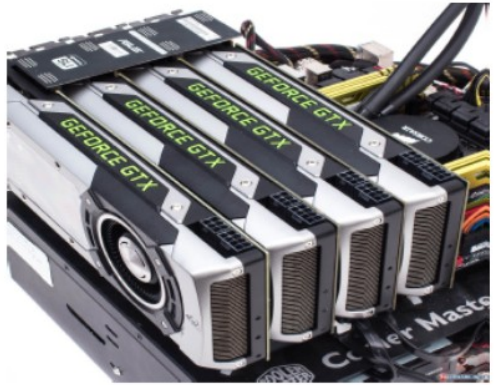
- The development and deployment of DNN architecture are a tedious process relying in constant trial and error to find the ideal model for a given problem.



DNN Designing Challenges

- Although an excellent DNN architecture can be found, there is no guarantee that it will fit within the constraints of the target hardware.





DNN Designing Challenges

- To make a DNN learn from real-world data, millions of data samples is required.

- **For example:**
  - **CIFAR10:** 60,000 color images with 32 x 32 pixels of resolution distributed in 10 classes[1].
  - **ImageNet:** 14 million color images with variable resolution distributed in 1,000 classes[2].

[1] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical Report. University of Toronto, Apr. 2009

[2] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: 2009 IEEE Conference on Computer Vision and Pattern Recognition. Miami, FL: IEEE, June 2009, pp. 248–255

Problem Statement

- **Two desired characteristic of a Deep Neural Network:**
  - Highly Accurate;
  - Small Computational Complexity.

- **The development and deployment of DNN architectures can be seen as a discrete combinatorial optimization:**
  - Evolutionary computation (EC) algorithms are good candidates for this type of problems.

- **Thus, the development of algorithms based on evolutionary approaches to design compact and highly accurate DNN architectures is needed to further advance the field of deep learning.**

Outline

- Introduction and Motivation
- Related Works
- Convolutional Neural Network Architecture Search Based on Particle Swarm Optimization
- Deep Neural Network Pruning with Evolution Strategy
- Conclusions

DNN Architecture Search

- **DNN architecture search can be achieved by:**
  - Only searching for weights;
  - Only searching for architectures;
  - Searching for both weights and architectures at the same time.

- **Classical approaches used in DNN architecture search:**
  - Genetic Algorithms;
  - Particle Swarm Optimization;
  - Evolution Strategy;
  - Etc.

DNN Architecture Search

- **Evolutionary approaches that search for weights and architectures at the same time:**
  - Neuroevolution of Augmenting Topologies (NEAT)[1].
  - Evolving deep Convolutional Neural Networks (EvoCNN)[2].
- **Evolutionary approaches that search only for architectures:**
  - Large-Scale Evolution of Classifiers[3].
  - Genetic CNN[4].

[1] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". en. In: Evolutionary Computation 10.2 (June 2002), pp. 99–127. doi: 10.1162/106365602320169811

[2] Yanan Sun et al. "Evolving Deep Convolutional Neural Networks for Image Classification". In: IEEE Transactions on Evolutionary Computation (2019), pp. 1–1. doi: 10.1109/TEVC.2019.2916183

[3] Esteban Real et al. "Large-scale Evolution of Image Classifiers". In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. ICML'17. event-place: Sydney, NSW, Australia. JMLR.org, 2017, pp. 2902–2911

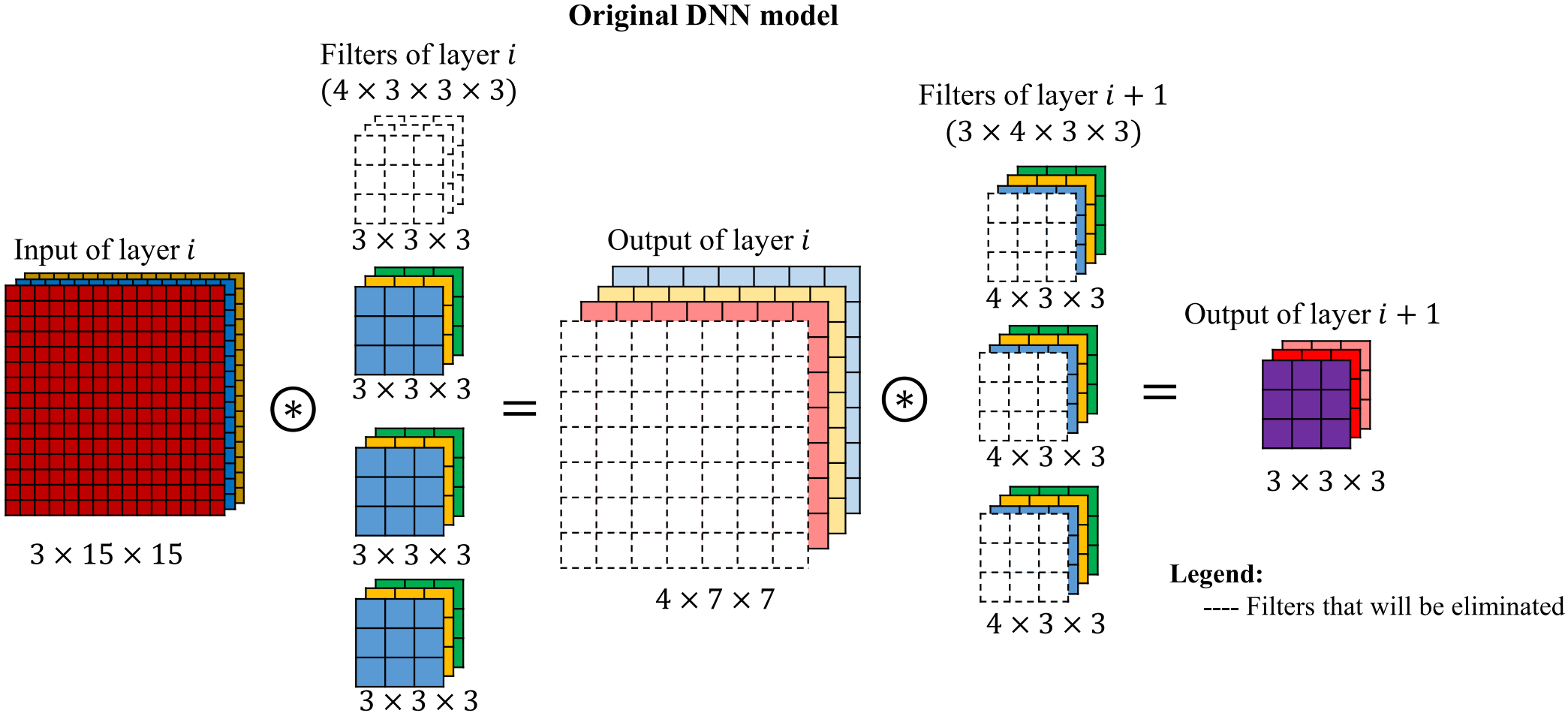
[4] Lingxi Xie and Alan Yuille. "Genetic CNN". In: The IEEE International Conference on Computer Vision (ICCV). Oct. 2017

DNN Architecture Pruning

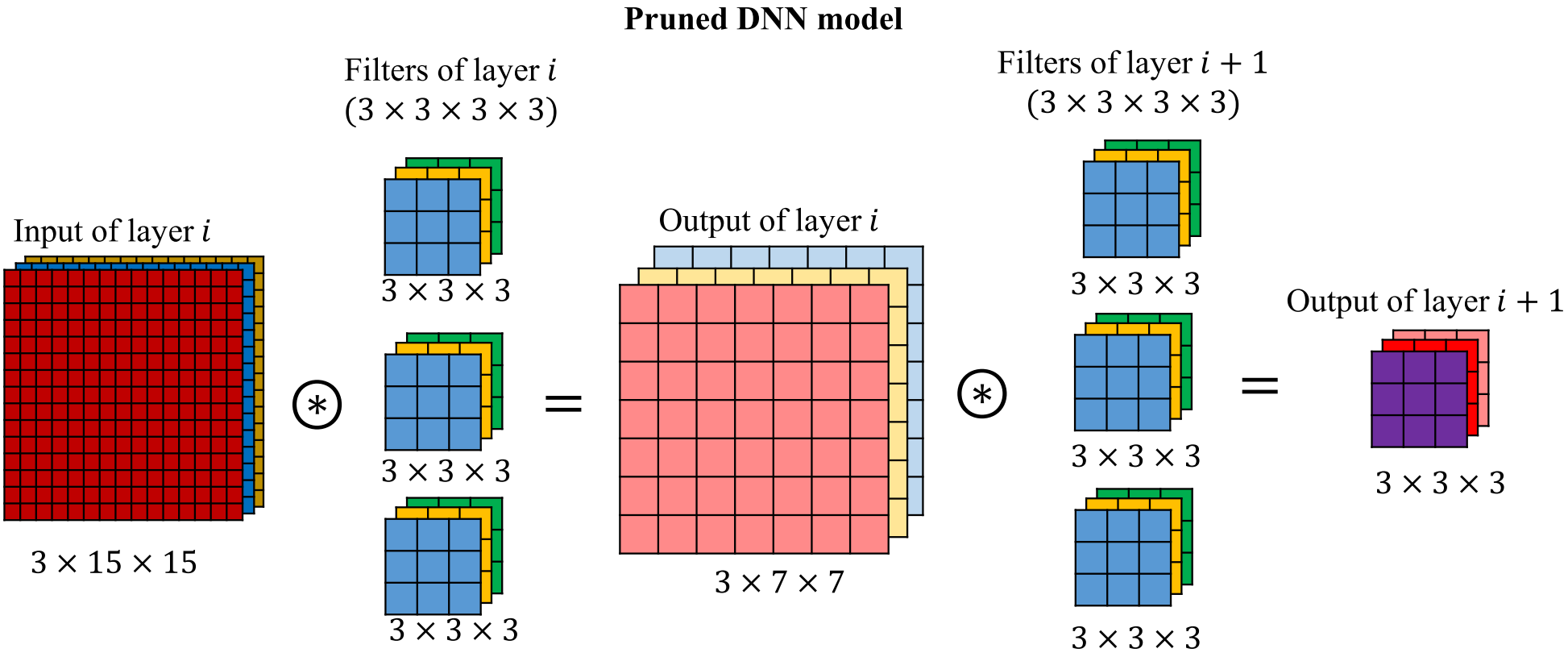
- Pruning or compression of DNN models is one solution for researchers and experts with limited computational power available.
- The hypothesis is that many DNN models handcrafted by humans have too many unnecessary parameters, and the identification and elimination of such parameters can reduce the model complexity without degrading its performance.
- Convolutional layers are responsible for most of the computational complexity of a DNN model[1].
- Thus, **Convolutional Filter Pruning** is the most used technique.

[1]Hao Li et al. "Pruning Filters for Efficient ConvNets". In: International Conference on Learning Representations. Toulon, France, 2017.

DNN Architecture Pruning



DNN Architecture Pruning



DNN Architecture Pruning

- **Filters are selected for elimination based on some criteria:**
  - **Mean activation:** the mean activation of each filter from a given layer is computed, and only the filters with the highest mean activation are kept[1].
  - **l1-norm:** the importance of a filter is determined by its l1-norm, i.e., the sum of the absolute values of the filter's weight. Filters with smaller l1-norm are considered unimportant compared with filters with large l1-norm[2].
  - **Average Percentage of Zeros (APoZ):** it eliminates the filters with a large number of zero activations during inference time[3].
  - **Random:** It randomly select filters to be eliminated without using any prior knowledge of the DNN architecture or filters. Mittal *et al.* showed that eliminating filters at random produced comparable results with other selection criteria[4].

[1] Pavlo Molchanov et al. "Pruning Convolutional Neural Networks for Resource Efficient Inference". In: arXiv:1611.06440 (Nov. 2016)
[2] Hao Li et al. "Pruning Filters for Efficient ConvNets". In: International Conference on Learning Representations. Toulon, France, 2017
[3] Hengyuan Hu et al. "Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures". In: arXiv:1607.03250 (July 2016)
[4] Deepak Mittal et al. "Studying the plasticity in deep convolutional neural networks using random pruning". In: Machine Vision and Applications 30.2 (Mar. 2019), pp. 203–216

Outline

- Introduction and Motivation
- Related Works
- Convolutional Neural Network Architecture Search Based on Particle Swarm Optimization
- Deep Neural Network Pruning with Evolution Strategy
- Conclusions

CNN Architecture Search on PSO

- **To reduce the time of the DNN development phase:**
  - An algorithm capable of automatically discovering DNN architectures is desirable.
  - The algorithm should also have a reduced running time compared to peer competitors.
- **Particle Swarm Optimization (PSO):**
  - A global optimization algorithm based on the flying pattern of bird flocks.
  - Faster convergence than standard genetic algorithms.
  - Thus, PSO is a good candidate approach for DNN architecture search.

F. E. Fernandes Junior and G. G. Yen, "Particle swarm optimization of deep neural networks architectures for image classification," Swarm and Evolutionary Computation, vol. 49, pp. 62–74, Sep. 2019.

Algorithm 1: Proposed psoCNN

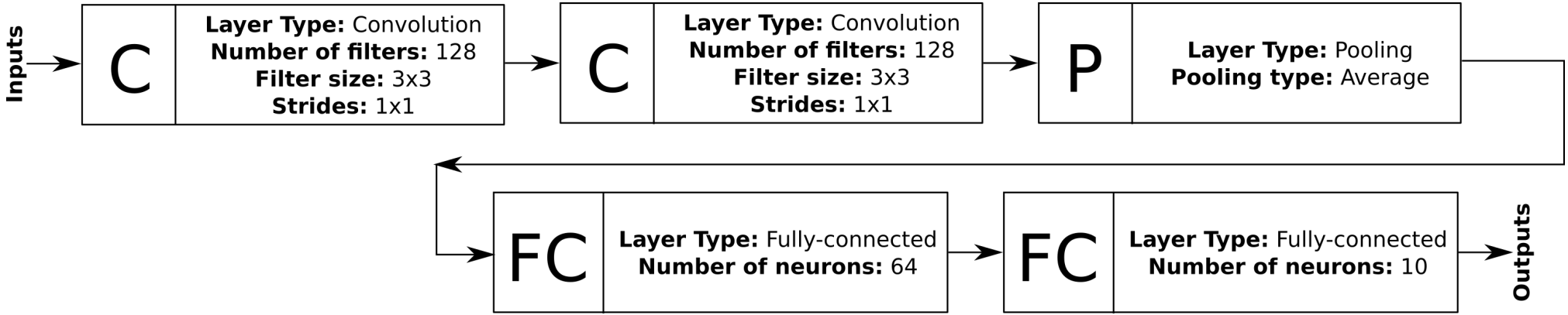
$S = \{P_1, ..., P_N\} \leftarrow InitializeSwarm(N, l_{max}, filter_{max}, k_{max}, n_{max}, n_{out})$;
$P_1.pBest \leftarrow P_1$;
$P_1.loss, P_1.pBest.loss \leftarrow ComputeLoss(P_1, X, e_{train})$;
Initialize $gBest \leftarrow P_1$;
$gBest.loss \leftarrow P_1.loss$;
for $i = 2$ *to* N **do**
    $P_i.pBest \leftarrow P_i$;
    $P_i.loss, P_i.pBest.loss \leftarrow ComputeLoss(P_i, X, e_{train})$;
    if $P_i.loss \leq gBest.loss$ **then**
        $gBest \leftarrow P_i$;
    end
end

Proposed Algorithm – *psoCNN* - 2/2

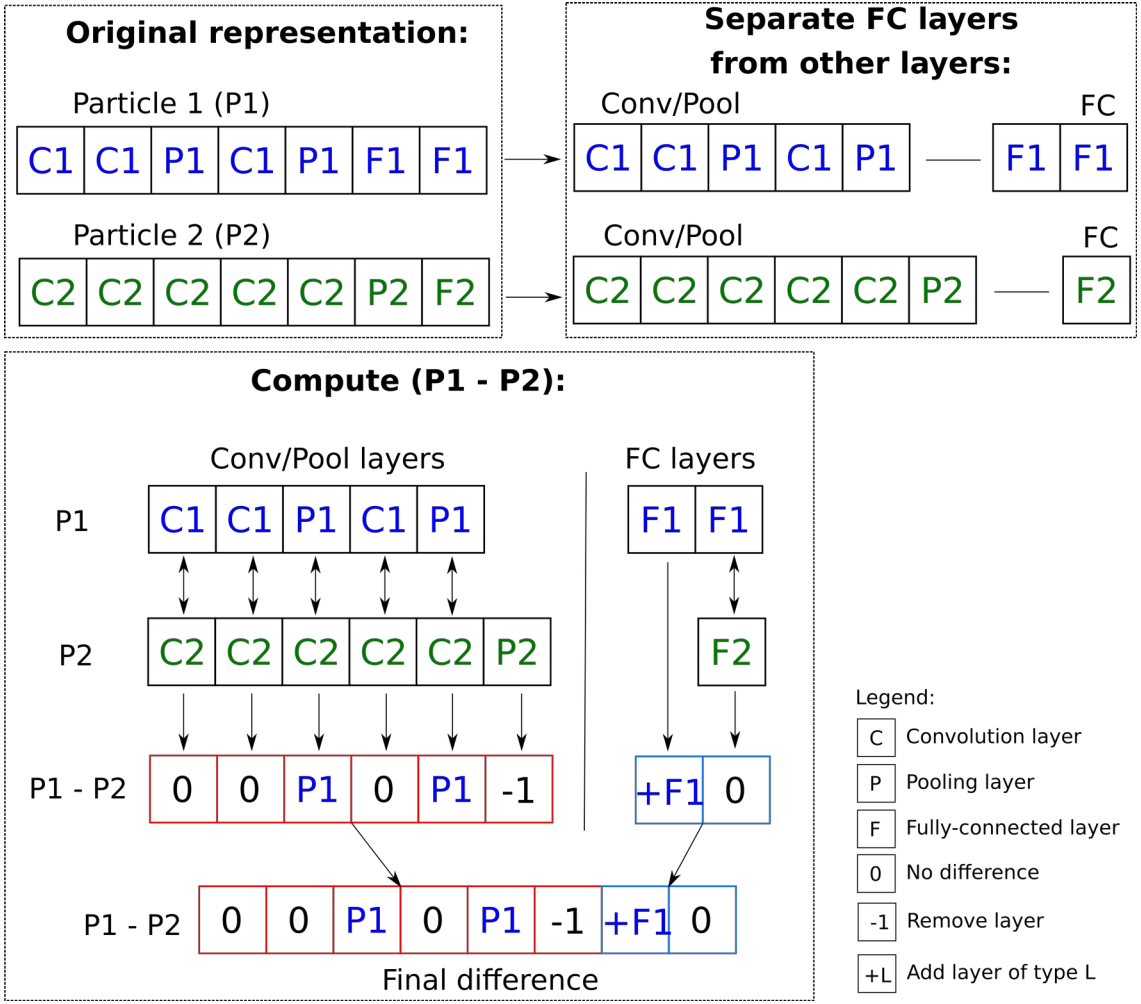
Algorithm 1: Proposed psoCNN

```
for iter = 1 to iter_max do
    for i = 1 to N do
        P_i.velocity ← UpdateVelocity(P_i, Cg) ;
        P_i ← UpdateParticle(P_i) ;
        P_i.loss ← ComputeLoss(P_i, X, e_train) ;
        if P_i.loss ≤ P_i.pBest.loss then
            P_i.pBest ← P_i ;
            P_i.pBest.loss ← P_i.loss ;
            if P_i.pBest.loss ≤ gBest.loss then
                gBest ← P_i.pBest ;
                gBest.loss ← P_i.pBest.loss ;
            end
        end
    end
end
gBest ← ComputeLoss(gBest, X, e_test) ;
return gBest, gBest.loss ;
```

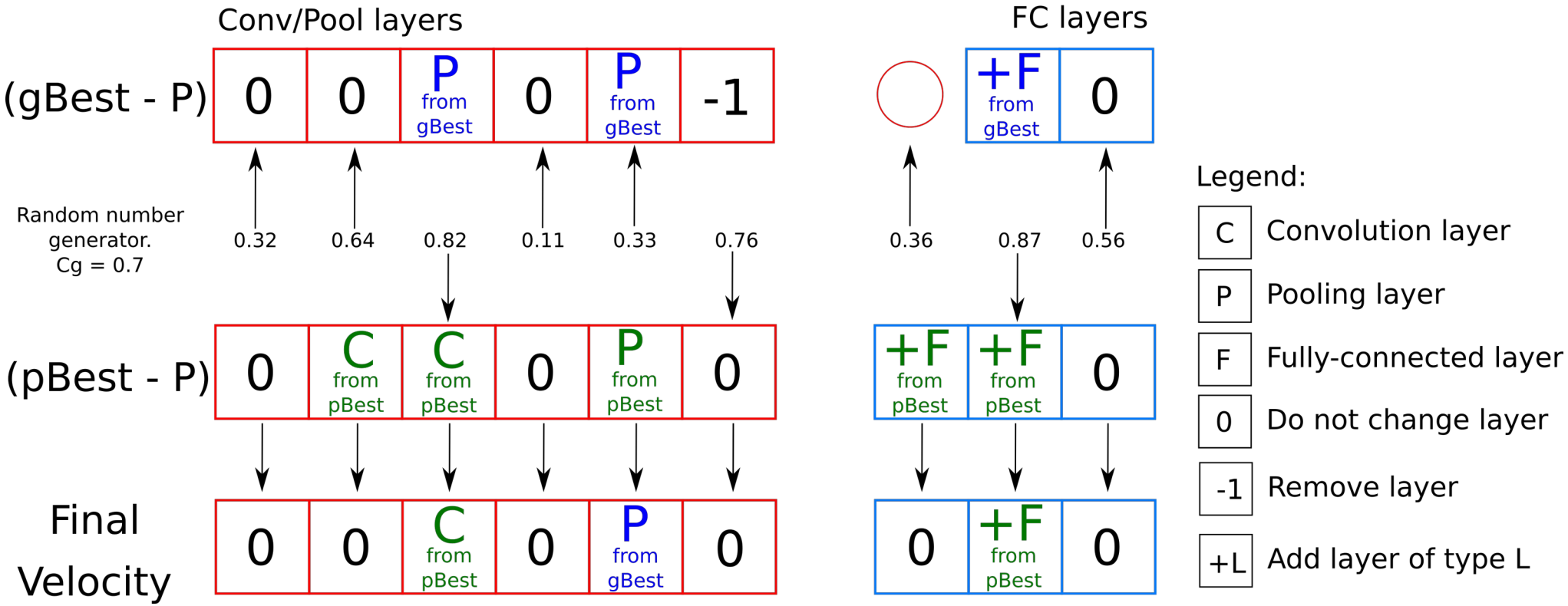
psoCNN - Representation

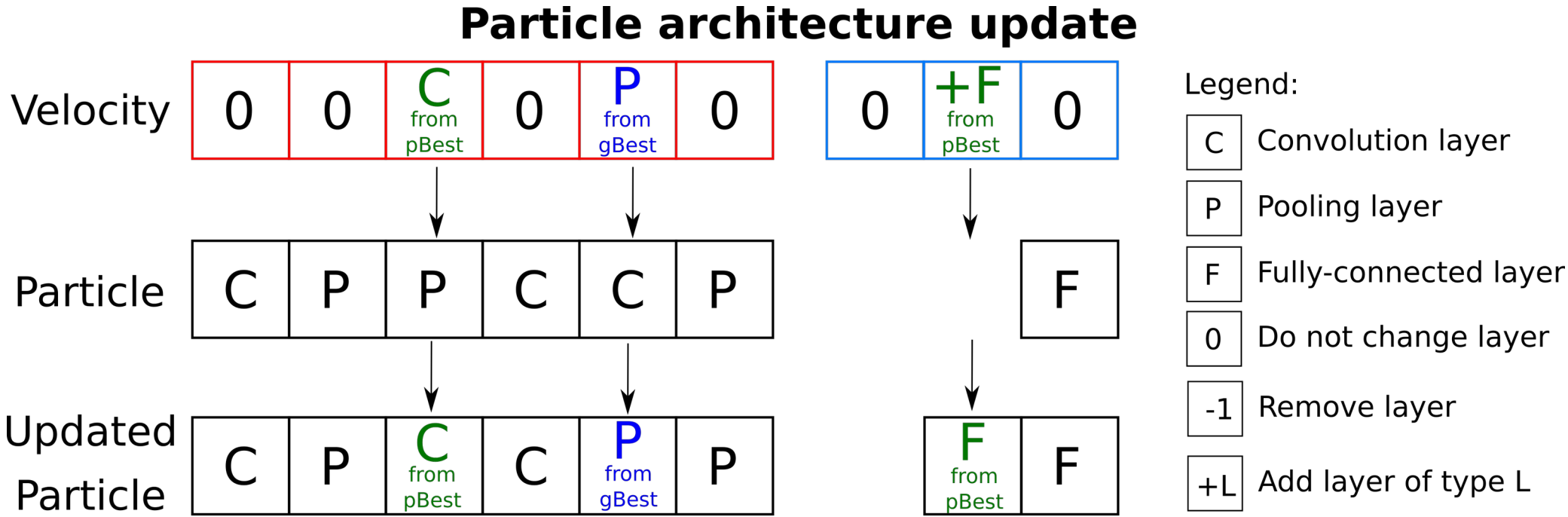


psoCNN - Difference between two particles

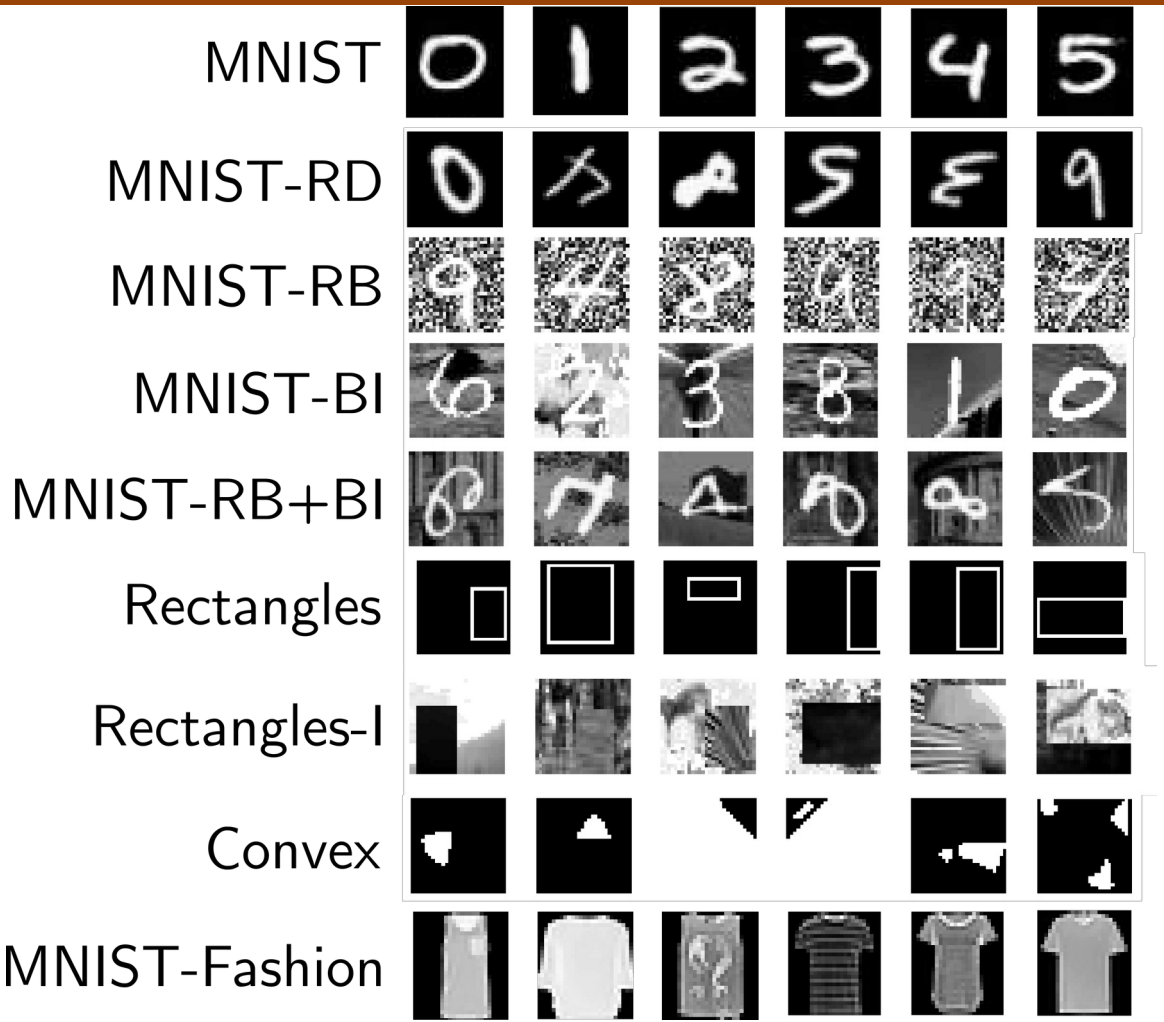


Velocity computation




Particle architecture update
Velocity
0
0
C
from
pBest
0
P
from
gBest
0
0
+F
from
pBest
0
Particle
C
P
P
C
C
P
F
Updated
Particle
C
P
C
from
pBest
C
P
from
gBest
P
F
from
pBest
F
Legend:
C
Convolution layer
P
Pooling layer
F
Fully-connected layer
0
Do not change layer
-1
Remove layer
+L
Add layer of type L

Datasets used to test the proposed *psoCNN*



psoCNN – Experimental Results

Table: Test results on the *MNIST*, *MNIST-RD*, *MNIST-RB*, *MNIST-BI*, *MNIST-RD+BI*, *Rectangles*, *Rectangles-I*, and *Convex* datasets.

Model	MNIST	MNIST-RD	MNIST-RB	MNIST-BI	MNIST-RD+BI	Rectangles	Rectangles-I	Convex
LeNet-1	1.7% (+)	-	-	-	-	-	-	-
LeNet-4	1.1% (+)	-	-	-	-	-	-	-
LeNet-5	0.95% (+)	-	-	-	-	-	-	-
RCNN	0.31% (-)	-	-	-	-	-	-	-
DropConnect	0.21% (-)	-	-	-	-	-	-	-
CAE-1	2.83% (+)	11.59% (+)	13.57% (+)	16.7% (+)	48.10% (+)	1.48% (+)	21.86% (+)	-
CAE-2	2.48% (+)	9.66% (+)	10.90% (+)	15.5% (+)	45.23% (+)	1.21% (+)	21.54% (+)	-
PCANet-2	1.06%(+)	8.52% (+)	6.85% (+)	11.55% (+)	35.86% (+)	0.49% (+)	13.39% (+)	4.19% (+)
RandNet-2	1.27% (+)	8.47% (+)	13.47% (+)	11.65% (+)	43.69% (+)	0.09% (+)	17.00% (+)	5.45% (+)
LDANet-2	1.40% (+)	4.52% (+)	6.81% (+)	12.42% (+)	38.54% (+)	0.14% (+)	16.20% (+)	7.22% (+)
EvoCNN (best)	1.18% (+)	5.22% (+)	2.8% (+)	4.53% (+)	35.03% (+)	0.01% (-)	5.03% (+)	4.82% (+)
EvoCNN (mean)	1.28% (+)	5.46% (+)	3.59% (+)	4.62% (+)	37.38% (+)	0.01% (-)	5.97% (+)	5.39% (+)
IPPSO (best)	1.13% (+)	-	-	-	34.50% (+)	-	-	8.48% (+)
IPPSO (mean)	1.21% (+)	-	-	-	33% (+)	-	-	12.06% (+)
psoCNN (best)	0.32%	3.58%	1.79%	1.90%	14.28%	0.03%	2.22%	1.7%
psoCNN (mean)	0.44%	6.42%	2.53%	2.40%	20.98%	0.34%	3.94%	3.9%

psoCNN – Experimental Results

Table: Test results on the *MNIST-Fashion* dataset.

Model	Test error	# parameters
Human performance	16.5% (+)	-
MLP 256-128-100	11.67% (+)	3M
GRU + SVM	11.2% (+)	-
HOG + SVM	7.4% (+)	-
AlexNet	10.1% (+)	62.3M
3CONV + 3FC	6.6% (+)	500k
VGG16	6.5% (+)	26M
GoogLeNet	6.3% (+)	23M
evoCNN (best)	5.47% (-)	6.68M
evoCNN (mean)	7.28% (+)	6.52M
MobileNet	5% (-)	4M
psoCNN – dropout – BN (best)	8.1% (+)	1.4M
psoCNN – dropout – BN (mean)	9.15% (+)	1.8M
psoCNN + dropout + BN (best)	5.53%	2.32M
psoCNN + dropout + BN (mean)	5.90%	2.5M

Best CNN for the MNIST dataset

Best CNN for the MNIST-BI dataset

Best CNN for the MNIST-RD dataset

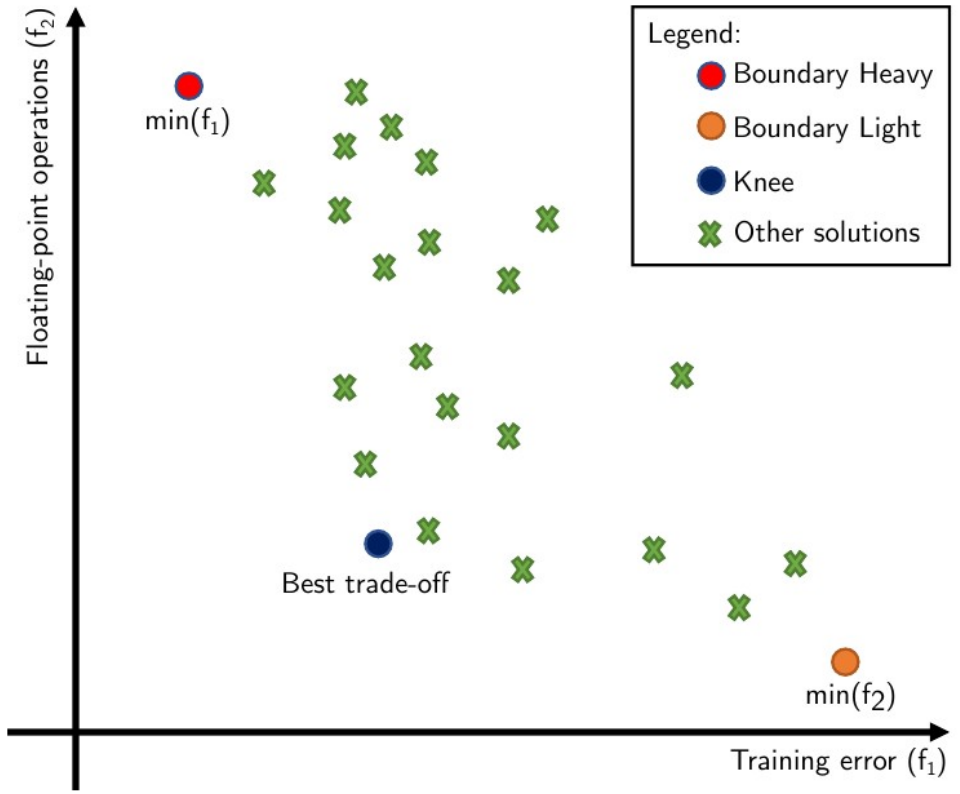
Best CNN for the Rectangles dataset

Outline

- Introduction and Motivation
- Related Works
- Convolutional Neural Network Architecture Search Based on Particle Swarm Optimization
- Deep Neural Network Pruning with Evolution Strategy
- Conclusions

Pruning DNN Architectures with ES

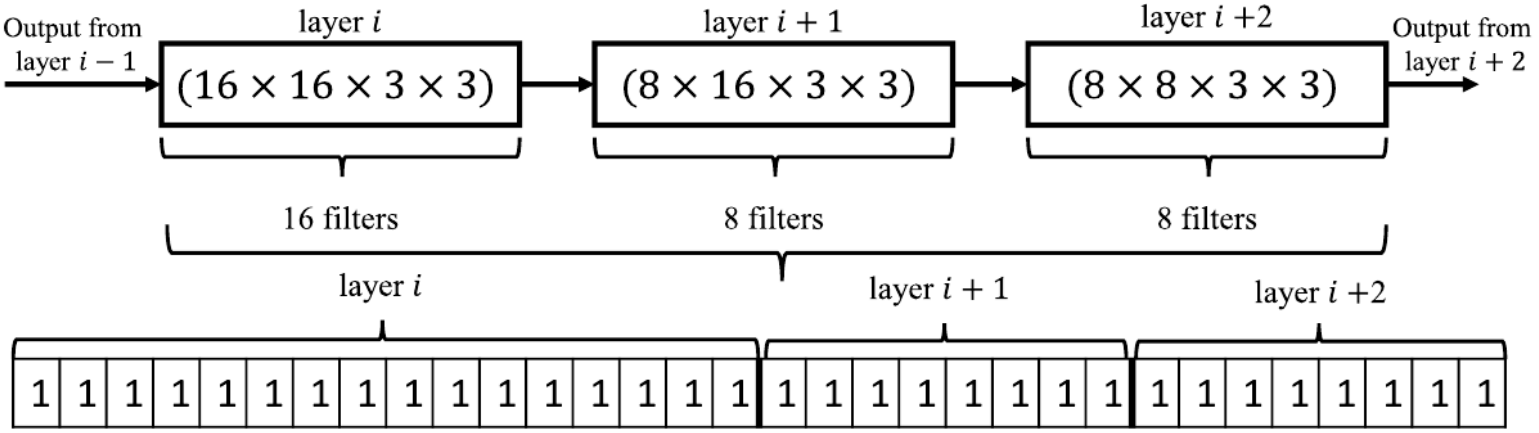
- **Evolution Strategy** is used to select filters for elimination at random, called *DeepPruningES*.
- Uses **Multi-Criteria Decision Making (MCDM)**:
  - Tries to **minimize** the **number of floating operations** and the **training error** of a given DNN model at the same time.
  - From a final population of N individuals, it outputs **three pruned DNN architectures** for use by decision makers (DMs):
    - **Boundary Heavy:** A pruned model with the lowest training error.
    - **Boundary Light:** A pruned model with the lowest number of floating operations.
    - **Knee:** A pruned model with the best trade-off between training error and number of floating operations.



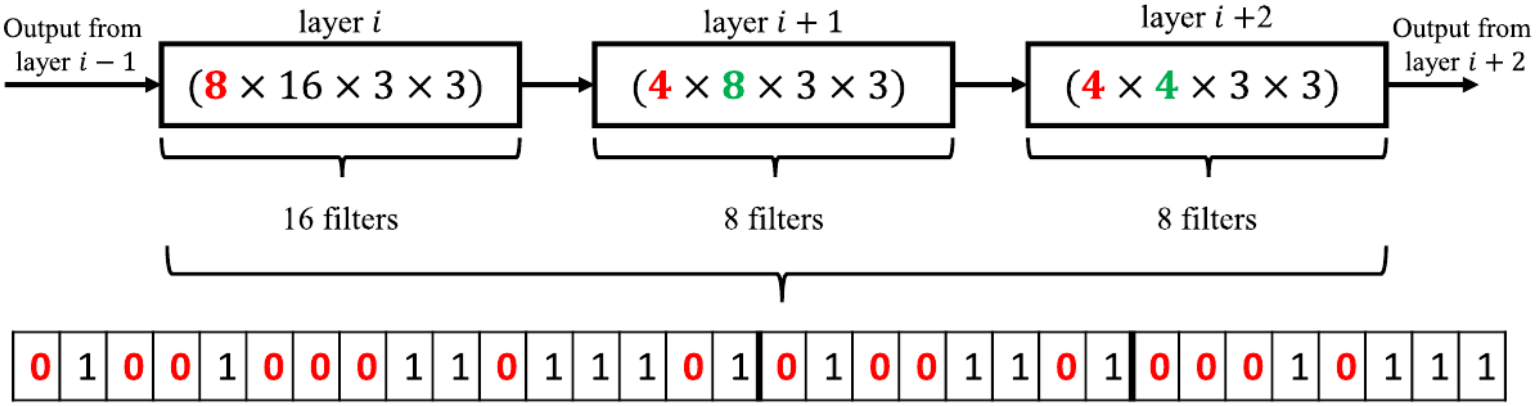
Representation - *DeepPruningES*

- **Works with:**
  - Convolutional Neural Networks (CNNs);
  - Residual Neural Networks (ResNets);
  - Densely Connected Neural Networks (DenseNets).
- **Filters of each convolutional layer are represented with a binary string:**
  - **1:** do not eliminate filter;
  - **0:** eliminate filter.

DeepPruningES – CNN Filter Representation

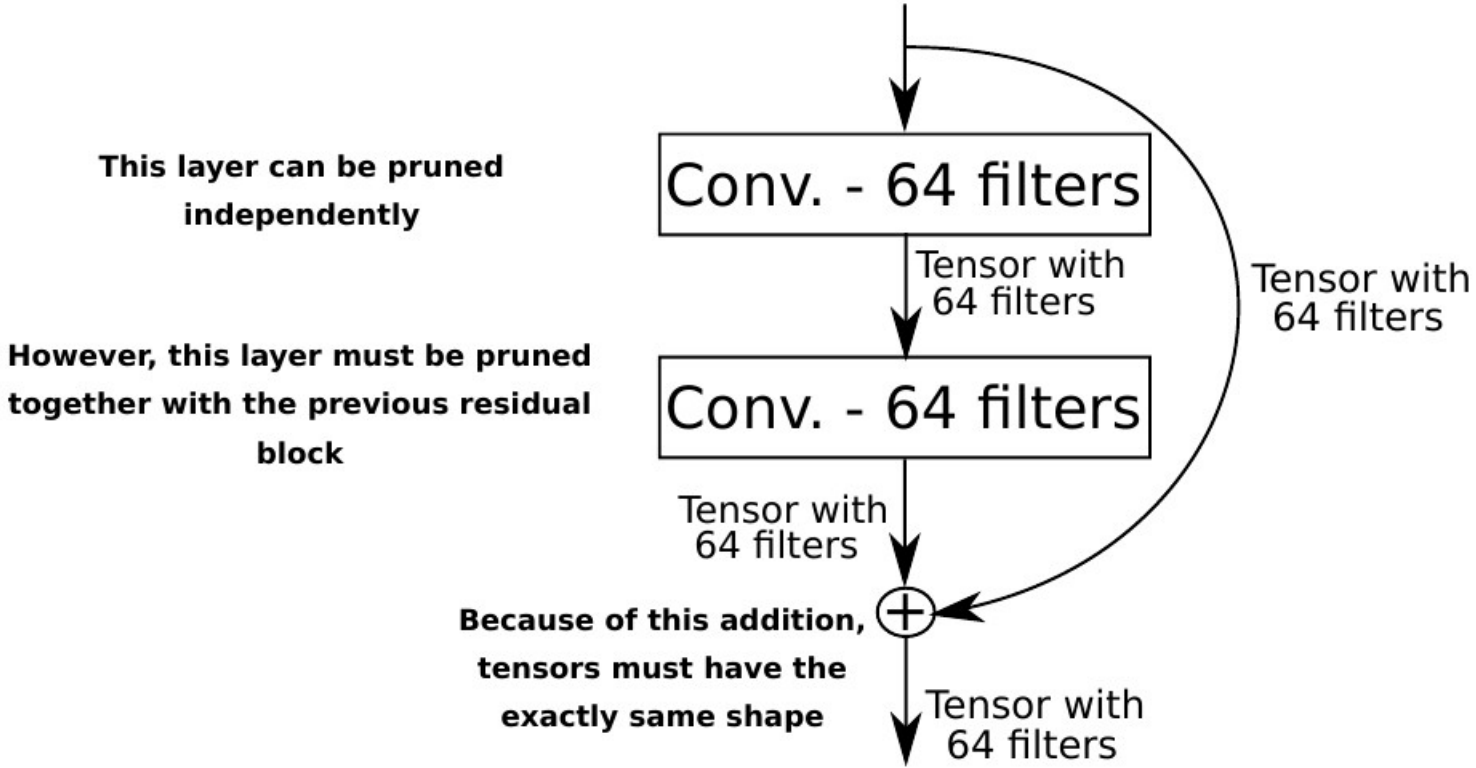


Binary string to represent a total of 32 filters

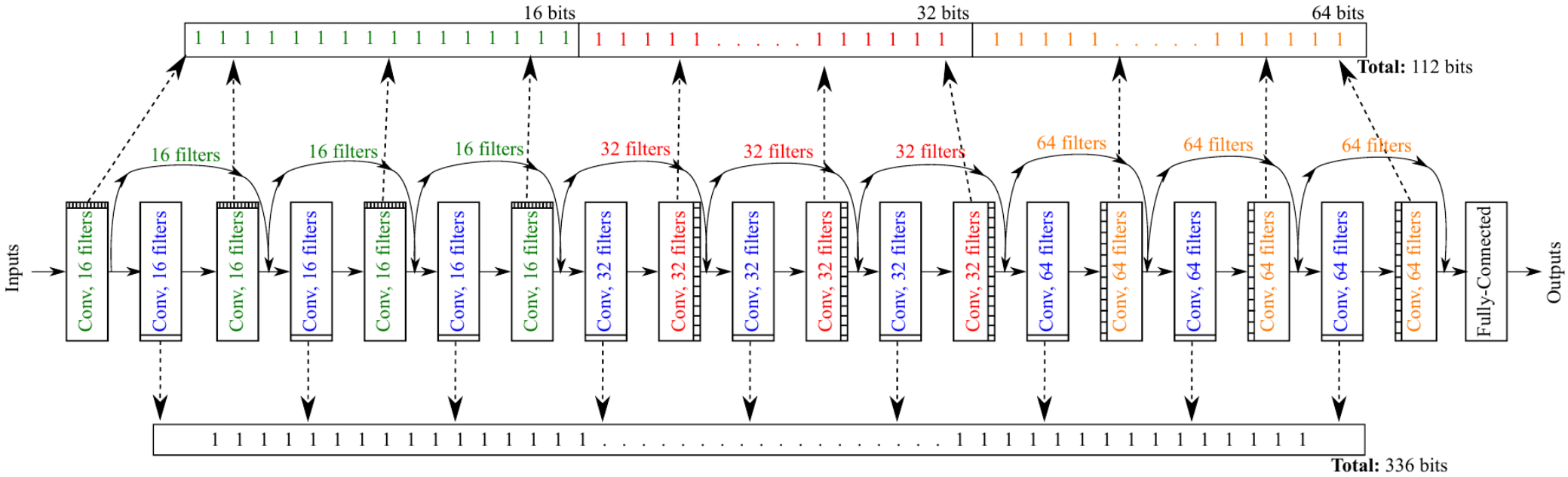


Binary string still has a total of 32 bit, but only 16 filters are kept

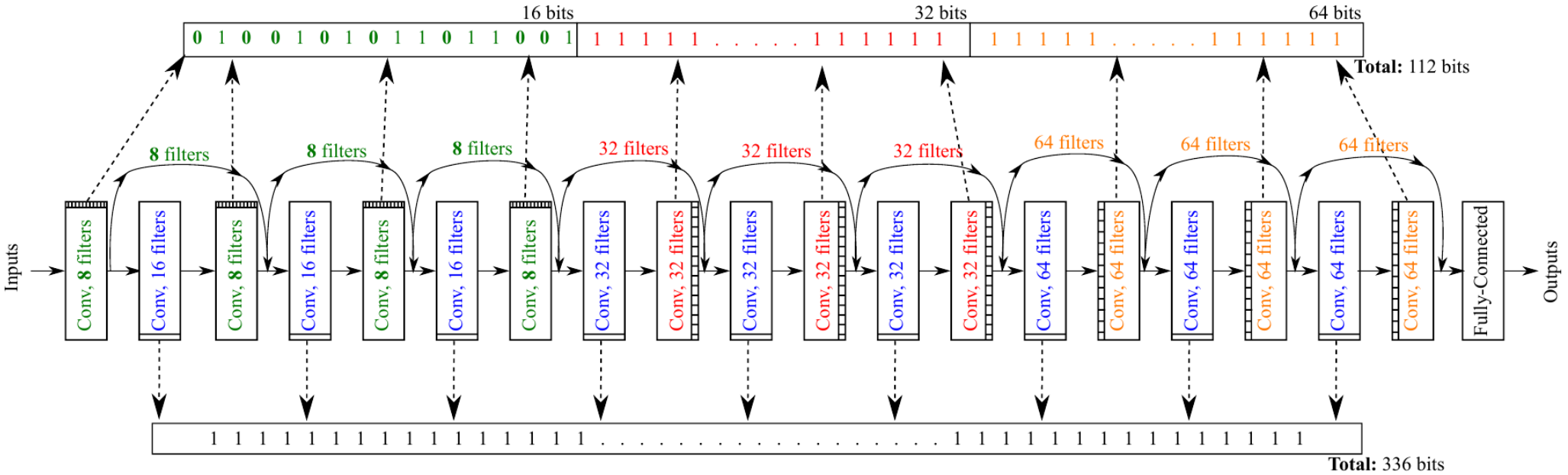
DeepPruningES – ResNet Filter Representation



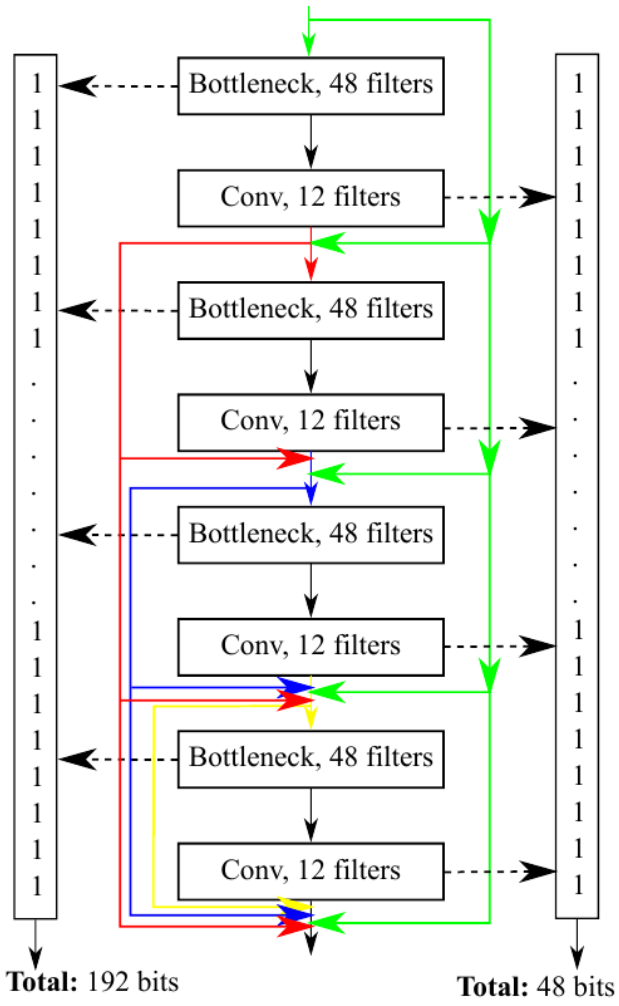
DeepPruningES – ResNet Filter Representation



DeepPruningES – ResNet Filter Representation



DeepPruningES – DenseNet Filter Representation



Algorithm 2: Proposed *DeepPruningES*

Input : Offspring size (λ_{size}), maximum number of generations (gen), mutation probability (p_m), original DNN model (dnn), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}), number of epochs for fine-tuning (e_{fine}), learning rate for fine-tuning (α_{fine}).

Output: Three DNN models: knee solution ($\mu.knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

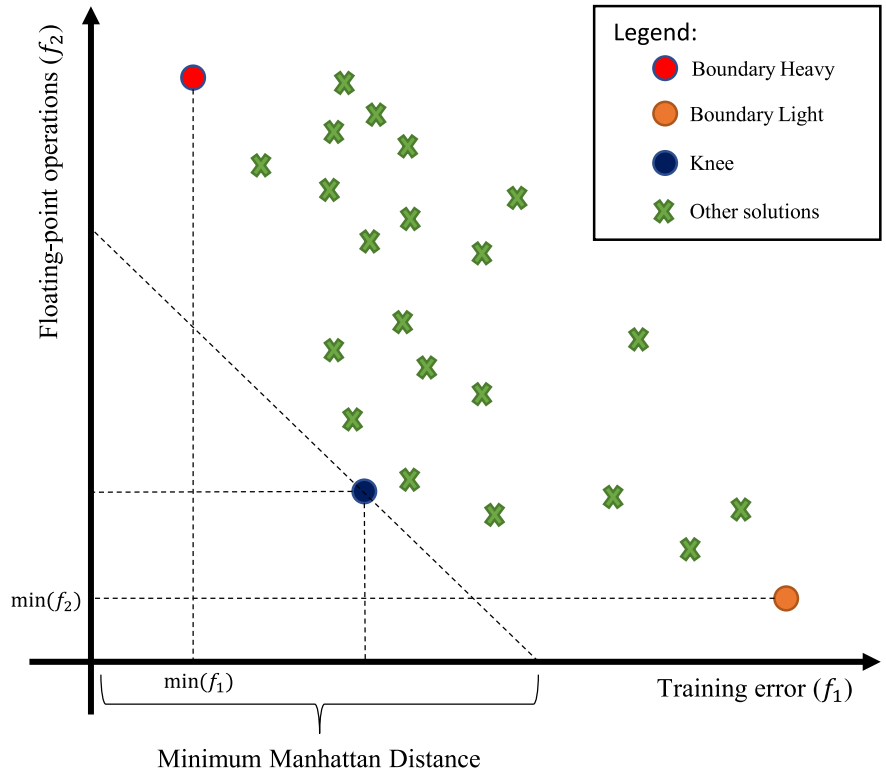
$\mu, \lambda \leftarrow$ *Initialize Population*(λ_{size}, dnn);
for $g = 1$ *to* gen **do**
  $\mathbf{P} \leftarrow \boldsymbol{\mu} + \boldsymbol{\lambda}$
  $\mu \leftarrow$ *Knee Boundary Selection*($\mathbf{P}$, dnn, e_{eval}, α_{eval});
  $\lambda \leftarrow$ *Offspring Generation*($\boldsymbol{\mu}$, λ_{size}, p_m, dnn);
end
Fine-tuning($\boldsymbol{\lambda}$, dnn, e_{fine}, α_{fine});
return $\mu.knee$, $\mu.heavy$, $\mu.light$;

Algorithm 3: Knee and Boundary Selection

Input : Individuals in the Population (**P**), original DNN model (*dnn*), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: Three individuals: knee solution ($\mu.knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

1 *Evaluate Population*(**P**, *dnn*, e_{eval}, α_{eval});
2 Find $min(f_1), min(f_2), max(f_1), max(f_2)$;
3 $\mu.heavy \leftarrow P_i$, where $f_1(P_i) = min(f_1)$;
4 $\mu.light \leftarrow P_j$, where $f_2(P_j) = min(f_2)$;
5 **for** $k = 1$ *to len*(**P**) **do**
6 $\quad dist(k) = \frac{f_1(P_k) - min(f_1)}{max(f_1) - min(f_1)} + \frac{f_2(P_k) - min(f_2)}{max(f_2) - min(f_2)}$;
7 **end**
8 $\mu.knee \leftarrow P_k$, where P_k has the minimum $dist(k)$;
9 **return** $\mu.knee$, $\mu.heavy$, $\mu.light$;



DNN models tested with *DeepPruningES*

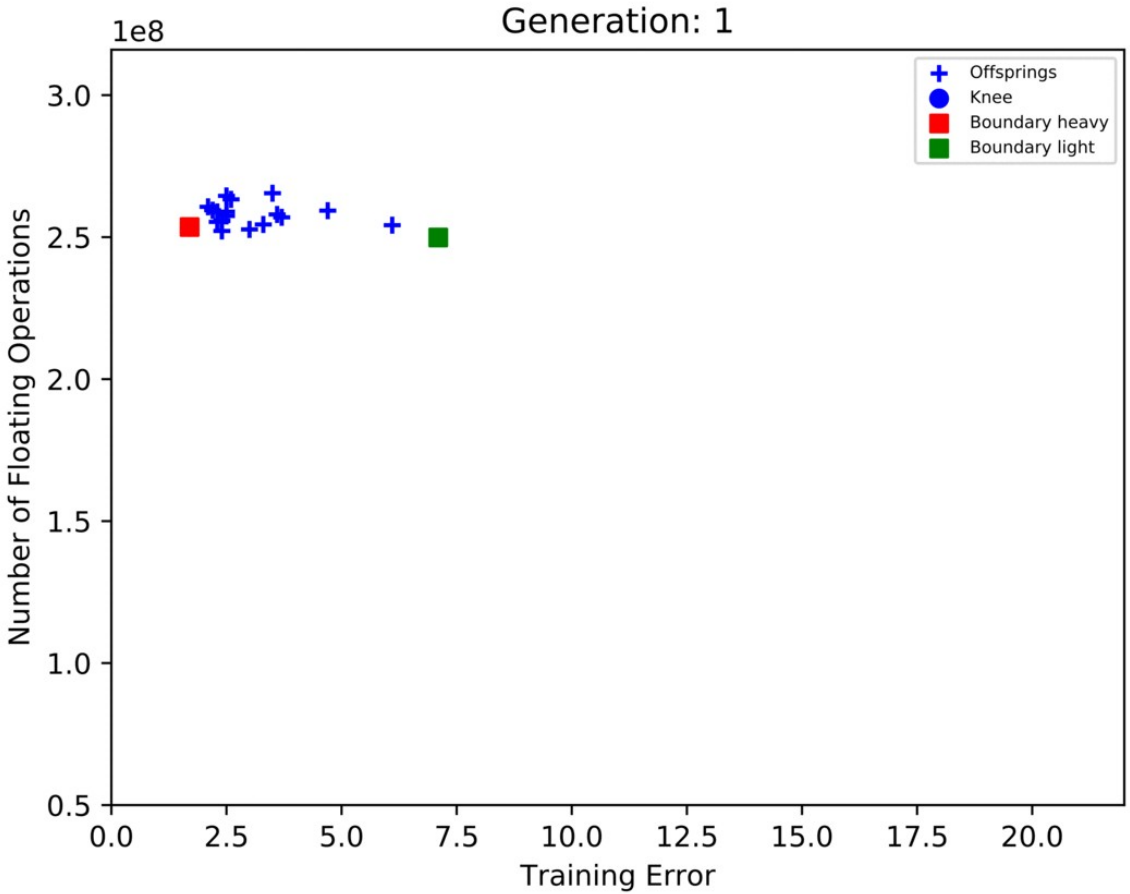
Table: Overview of the DNN architectures used to evaluate the proposed *DeepPruningES*.

DNN	# layers	# FLOPs	Test error on CIFAR10
VGG16	16	3.15×10^8	6.06%
VGG19	19	4.01×10^8	6.18%
ResNet56	56	1.27×10^8	6.63%
ResNet110	110	2.57×10^8	6.2%
DenseNet50	50	0.93×10^8	6.92%
DenseNet100	100	3.05×10^8	5.66%

Pruning results obtained with the proposed *DeepPruningES* on CIFAR10

DNN Model	DeepPruningES						
VGG16	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
	Knee	9.04%	9.58%	1.09×10^8	1.22×10^8	65.49%	61.58%
	Boundary Heavy	8.21%	8.6%	2.15×10^8	2.49×10^8	32.01%	20.88%
	Boundary Light	10.51%	11.41%	0.88×10^8	0.9×10^8	72.17%	71.36%
VGG19	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
	Knee	9.04%	9.87%	1.53×10^8	1.72×10^8	61.86%	57.15%
	Boundary Heavy	8.21%	8.77%	2.7×10^8	3.12×10^8	32.56%	22.28%
	Boundary Light	10.53%	12.03%	1.13×10^8	1.18×10^8	71.74%	70.69%
ResNet56	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
	Knee	9.28%	9.98%	0.432×10^8	0.523×10^8	66.23%	59.15%
	Boundary Heavy	8.11%	8.77%	1.01×10^8	1.08×10^8	21.31%	15.23%
	Boundary Light	11.42%	13.36%	0.244×10^8	0.286×10^8	80.89%	77.67%
ResNet110	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
	Knee	8.66%	9.42%	0.905×10^8	1.03×10^8	64.84%	59.89%
	Boundary Heavy	7.43%	7.93%	2.14×10^8	2.21×10^8	16.72%	14.14%
	Boundary Light	10.27%	12.9%	0.43×10^8	0.56×10^8	83.29%	77.86%
DenseNet50	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
	Knee	9.8%	10.43%	0.41×10^8	0.466×10^8	56.05%	50.15%
	Boundary Heavy	8.91%	9.26%	0.756×10^8	0.779×10^8	19.16%	16.59%
	Boundary Light	13.04%	14.8%	0.229×10^8	0.244×10^8	75.53%	73.91%
DenseNet100	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
	Knee	9.04%	9.37%	1.11×10^8	1.21×10^8	63.64%	60.31%
	Boundary Heavy	8.34%	8.39%	2.46×10^8	2.49×10^8	19.33%	18.24%
	Boundary Light	10.47%	11.90%	0.82×10^8	0.879×10^8	73.09%	71.16%

DeepPruningES - Population Evolution for Nine Generations



Conclusions

- Deep Neural Networks (DNNs) are very popular models.
  - However, their development requires expert knowledge about the problem at hand and lots of tedious work.
- To address this problem, the task of creating DNN architectures is characterized as a discrete combinatorial problem.
  - **Particle Swarm Optimization** was used to search for meaningful CNN architectures with competitive results.
- Another problem that can hindering the use of DNNs is their massive computing requirements.
  - **Evolution Strategy** and **Multi-Criteria Decision Making** were used tools to allow the pruning of unnecessary parameters.
  - The algorithm can efficiently find three pruned DNN models with different trade-offs between computational complexity and accuracy.

DNN Implementation with Python

- **Deep Learning Frameworks:**

https://pytorch.org/

http://caffe.berkeleyvision.org/

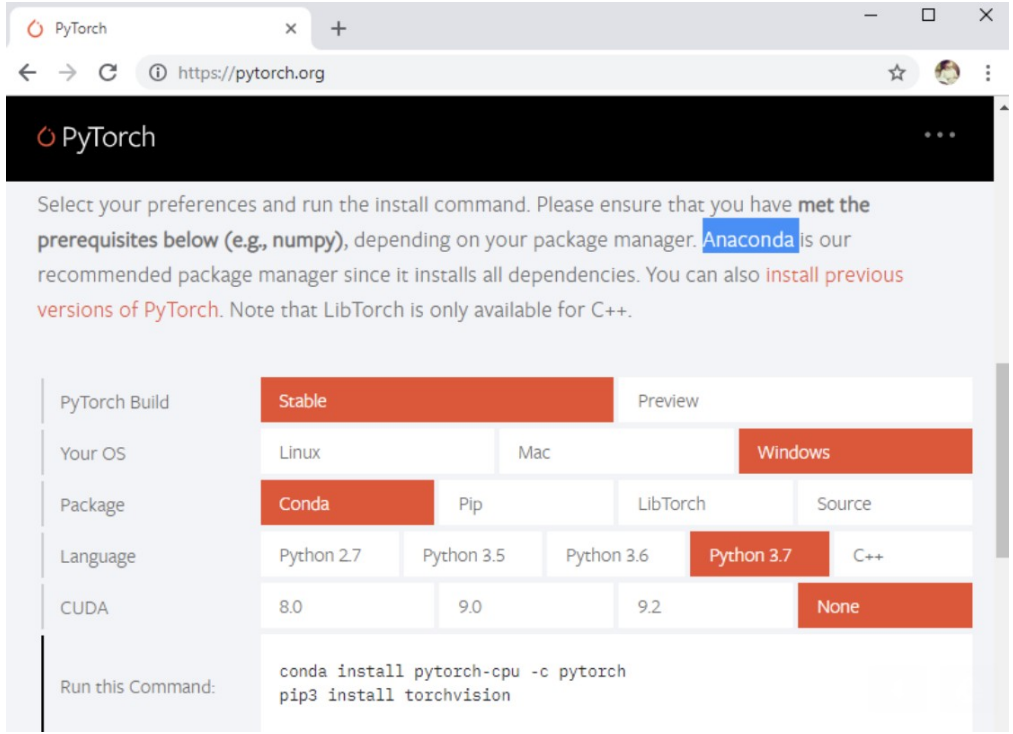
https://www.tensorflow.org

https://mxnet.apache.org/

https://keras.io/

PyTorch Installation

- **Step 1:**



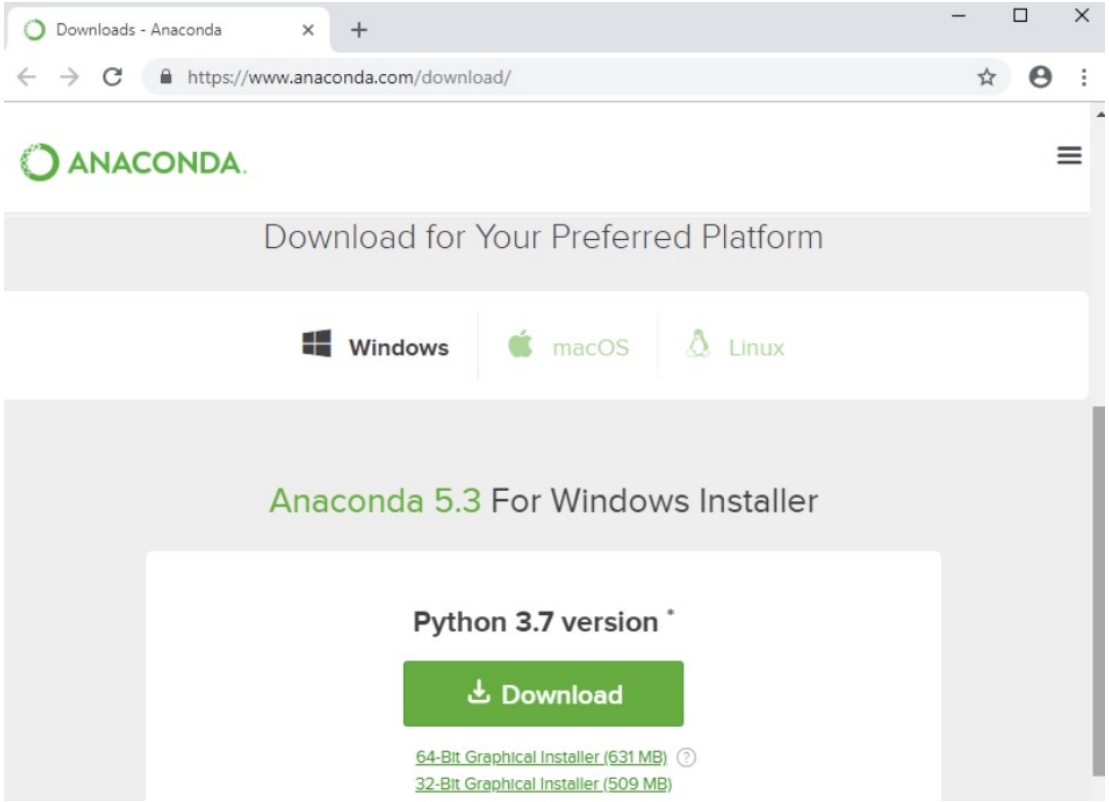
Install *Anaconda* First

For Linux and Mac, see instructions on https://pytorch.org/

PyTorch Installation

- **Step 2:**

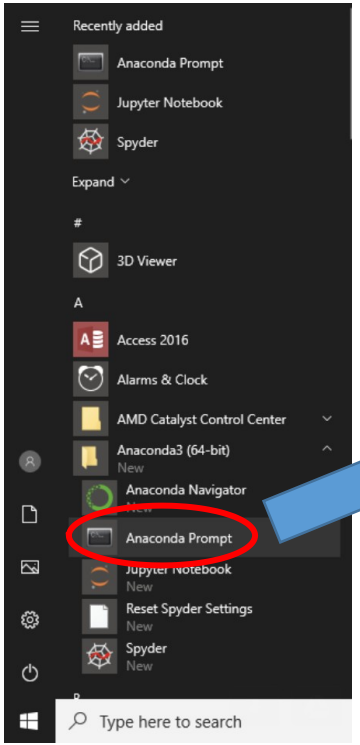
ANACONDA https://www.anaconda.com/download



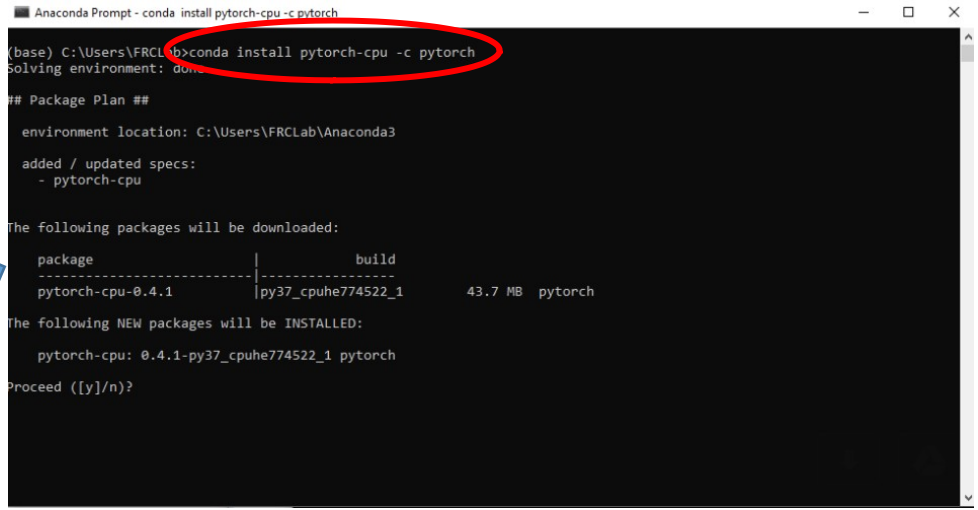
- Download *Anaconda*
- Install it

PyTorch Installation

- **Step 3:**



After installed Anaconda on your computer,
open **Anaconda Promopt**, and type the following commands:
conda install pytorch-cpu -c pytorch
pip install torchvision



Type 'y' when asked by "Proceed ([y]/n)?"

If you have GPU on your computer, please refer to instructions on pytorch.org.

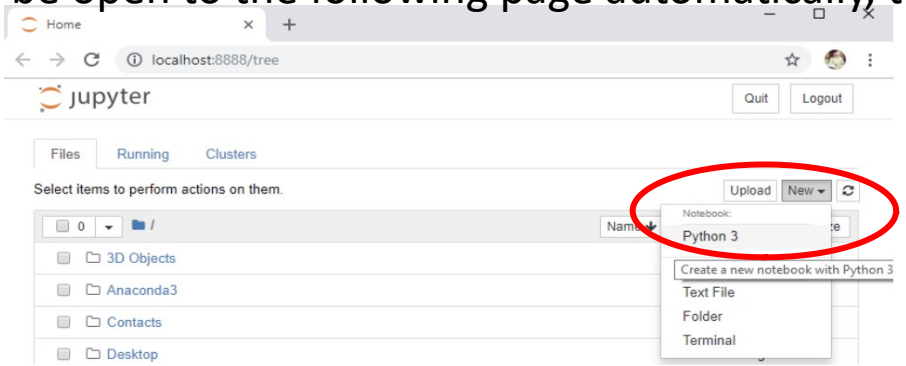
How to programming with PyTorch?

Not familiar with Python?
Read the following short tutorial
http://cs231n.github.io/python-numpy-tutorial/

1. Type *jupyter notebook* in Anaconda Prompt

```
base) C:\Users\FRCLab>jupyter notebook
I 19:06:56.433 NotebookApp] Writing notebook server cookie secret to C:\Users\FRCLab\AppData\Roaming\jupyter\runtime\no
tebook_cookie_secret
I 19:07:00.582 NotebookApp] JupyterLab extension loaded from C:\Users\FRCLab\Anaconda3\lib\site-packages\jupyterlab
I 19:07:00.583 NotebookApp] JupyterLab application directory is C:\Users\FRCLab\Anaconda3\share\jupyter\lab
I 19:07:00.650 NotebookApp] Serving notebooks from local directory: C:\Users\FRCLab
I 19:07:00.650 NotebookApp] The Jupyter Notebook is running at:
I 19:07:00.653 NotebookApp] http://localhost:8888/?token=d95403cbad925ff90a880337f50e5615a8833954474a8471
I 19:07:00.653 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
C 19:07:00.708 NotebookApp]

   Copy/paste this URL into your browser when you connect for the first time,
   to login with a token:
       http://localhost:8888/?token=d95403cbad925ff90a880337f50e5615a8833954474a8471
```

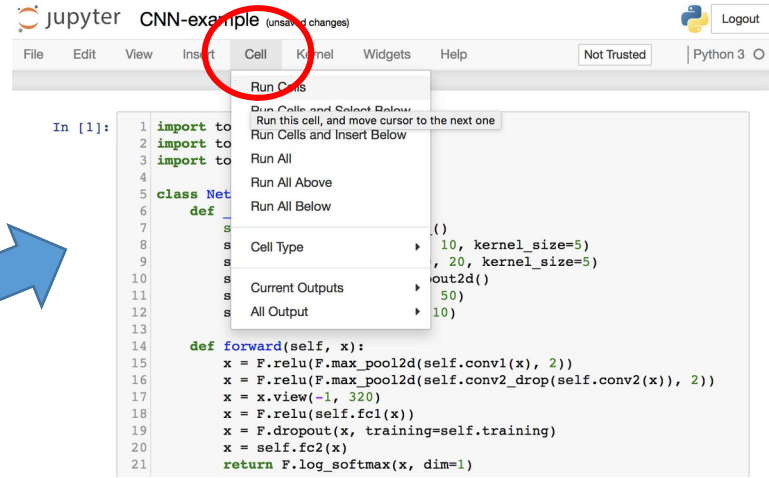
2. Your browser will be open to the following page automatically, then click “New” => “Python3”



How to programming with PyTorch?

- Write code in notebook cells, and run them.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):
        def __init__(self):
                super(Net, self).__init__()
                self.conv1 = nn.Conv2d(1, 10, kernel_size=5)

                self.conv2 = nn.Conv2d(10, 20, kernel_size=5)

                 self.conv2_drop = nn.Dropout2d()
                self.fc1 = nn.Linear(320, 50)
                self.fc2 = nn.Linear(50, 10)
        def forward(self, x):
                x = F.relu(F.max_pool2d(self.conv1(x), 2))
                x =
                F.relu(F.max_pool2d(self.conv2_drop(self.conv
                2(x)), 2))
                x = x.view(-1, 320)
                x = F.relu(self.fc1(x))
                x = F.dropout(x, training=self.training)
                x = self.fc2(x)
                return F.log_softmax(x, dim=1)
```



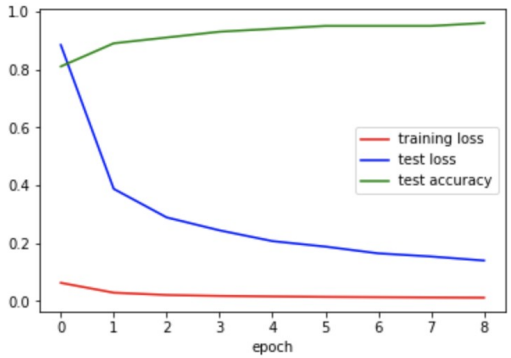
PyTorch implementation of a CNN,
which consists of 2 conv layers and 2 fully connected layers.

How to programming with PyTorch?

- Training the CNN on MNIST[1] dataset for handwritten digit recognition

```python
import torch.optim as optim
from torchvision import datasets, transforms

def train(model, device, train_loader, optimizer, epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
    train_loss /= len(train_loader.dataset)
    print('Train Epoch: {} \tLoss: {:.6f}'.format(epoch, train_loss))
    return train_loss

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up batch
            pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    test_acc = 100. * correct / len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset), test_acc))
    return test_loss, test_acc
```

```python
def dataloaders(batch_size=32, test_batch_size=32):
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=False, transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=test_batch_size, shuffle=True)
    return train_loader, test_loader

device = torch.device("cpu")
model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.5)
train_loader, test_loader = dataloaders(batch_size=32, test_batch_size=32)

for epoch in range(1, 10):
    train_loss = train(model, device, train_loader, optimizer, epoch)
    test_loss,test_acc = test(model, device, test_loader)
```



99+% accuracy, significantly better than many other machine learning techniques!!!

[1] http://yann.lecun.com/exdb/mnist/