Automated CNN Architecture with Evolutionary Algorithm

Volha Leusha, Ruben Rodriguez Buchillon
Stanford - CS221
{leusha;rubenr2}@stanford.edu

Abstract

In this project we present a method that automatically searches for a promising Convolutional Neural Network (CNN) architecture for image classification task. Additionally we research into possibility to minimize training and prediction compute without significant impact on prediction accuracy. Our approach is based on Evolutionary Algorithm described in [3] and [2] with multiple modifications on our side. Specifically we introduce new combined methods to select networks from population for crossover, and compare performance of this methods to binary tournament and random selection. Moreover, we try to speed up the architecture search by adding prediction time to the fitness function.

Due to computational constraints we use relatively small dataset and restrict the model to six convolutional and two dense layers. However, the method is applicable for any dataset and is easily extendable to more layers by only adjusting two parameters in the code. Performance of the search method presented in this project proves to be comparable with well known benchmarks. It achieves similar to Google AutoML prediction accuracy with significant reduction in search time from 24h to 2h. Moreover, by adding prediction time into the fitness function the search time is decreased approximately by a factor of two compared to the algorithm presented in [2] with only slight drop in accuracy.

code.zip

`https://bit.ly/35k7gxl`
`https://drive.google.com/open?id=1HFNOFFdQf5wfkngjyfPQ_IL-6ZaeizCo`

Note: The two links are the same. The bit.ly is simply provided in case that the user cannot copy/paste and has to manually type the link
Note: There is no data.zip file as the framework handles retrieving the dataset

1. Introduction

In the last decade CNNs have gained dominant popularity in deep learning due to growing compute power and demonstrated superiority over other machine learning techniques in multiple real-world applications [4], [5], [6]. Developing neural network models often requires significant amount of effort and compute. There have been multiple successful attempts to achieve top accuracy by introducing new optimization methods, transformations, etc. into the architecture. Impressive results were achieved by [7], [8], [9]. As mentioned in [2], there is a trend for increasingly deeper architectures that is driven by an assumption that deeper networks can handle more data complexities. Some examples of successful deep neural networks are: DenseNet201 [10], Resnet152[11]. As performance of CNN is strongly related with the type of data set, it is not always possible to use existing models without modifications. Moreover, people who know the specifics of certain data are not always experts in the neural network architecture, and vice versa. Consequently, there is an increasing demand for automated systems. In general, algorithms to design CNN architectures are divided into two big groups: evolutionary algorithms [13], [14] and algorithms based on reinforcement learning [1], [12].

2. Approach

In this project we used random search and Evolutionary strategy to find a promising architecture, and compared performance for this approaches. Evolutionary algorithm is based on the concept from genetics: 'only best genes survive'. The basic idea is to first train a number of randomly selected parent CNNs and use some fitness function to determine how fit they are. Then only the best parents are selected and crossed to produce next population of networks. This procedure has to be repeated for multiple generations until requested accuracy is found or assigned computational resources are consumed. Open questions here are: which exactly fitness function and parent selection strategy to use. In existing papers the combination of prediction accuracy and binary tournament strategy is utilized. However, it does not always produce best accuracy and can result in a slow search. We tried to solve this problem by adding prediction time, and number of CNN parameters to the fitness function and combining tournament selection with various sampling strategies, such as Gibbs and epsilon-greedy.

2.1 Architecture Specification

To define the architecture of the CNN we used a sequence S of numbers that has predefined length and structure. In the project S is restricted to six convolutional and two dense layers due to computation constraints, though it is extendable to more layers. Each number in the sequence represents a specific building parameter to construct a network as shown on the image below.
The sequence starts with convolutional blocks, each followed by one maxpooling block. The output of the last pooling block is followed by a flatten layer. Then construction of dense blocks in done, followed by a last dense block with 'softmax' activation and $units = 10$ - amount of classification labels. Each block can be active or not active and this is controlled by $\{'on\_off'\}$ parameter. Moreover, convolutional blocks have additional parameters: $\{'n\_filters', filter\_sizes',' batch\_norm',' drop\_out'\}$, and dense block has additional parameters: $\{'units', batch\_norm',' drop\_out'\}$, where
$D('on\_off') = \{0, 1\}$
$D('n\_filters') = \{4, 8, 16, 32, 64, 128\}$
$D('filter\_sizes') = \{3, 5\}$
$D('batch\_norm') = \{0, 1\}$
$D('drop\_out') = \{0, 0.2, 0.3, 0.4, 0.5\}$
$D('units') = \{4, 8, 16, 32, 64, 128, 256, 512, 1024\}$
In order to have unbiased comparison we use the same architecture construction approach for all search strategies.

2.2 Random Search

Random search strategy is a naive solution model. We use the same random approach to construct population of sequences $P(S)$ for all iterations, where size of population in one iteration is 10.

2.3 Evolution Algorithm

Evolution algorithm used in this project is based on algorithms described in [2] and [3]. We use the general idea of crossover and mutation operation from the literature, with multiple upgrades and modifications of fitness function and parent selection method.

General Algorithm:

1. Create n random sequences of pre-defined length
2. Convert sequences to CNNs as shown on the Figure 1
3. For each CNN calculate fitness using specified fitness function
4. Select next generation of n sequences using a specified selection strategy
5. Crossover and mutate as shown on Figure 2
7. Go to step 2 and repeat for m iterations

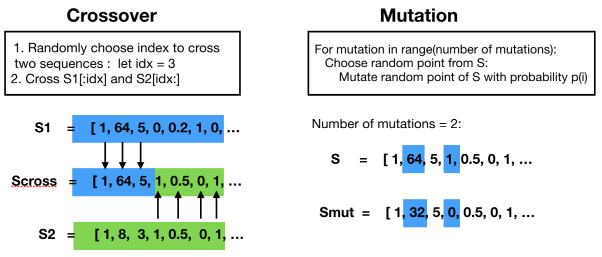

Crossover
1. Randomly choose index to cross two sequences : let idx = 3
2. Cross S1[:idx] and S2[idx:]
S1 = [1, 64, 5, 0, 0.2, 1, 0, ...
Scross = [1, 64, 5, 1, 0.5, 0, 1, ...
S2 = [1, 8, 3, 1, 0.5, 0, 1, ...
Mutation
For mutation in range(number of mutations):
Choose random point from S:
Mutate random point of S with probability p(i)
Number of mutations = 2:
S = [1, 64, 5, 1, 0.5, 0, 1, ...
Smut = [1, 32, 5, 0, 0.5, 0, 1, ...

Figure 2: Crossover and mutation operation

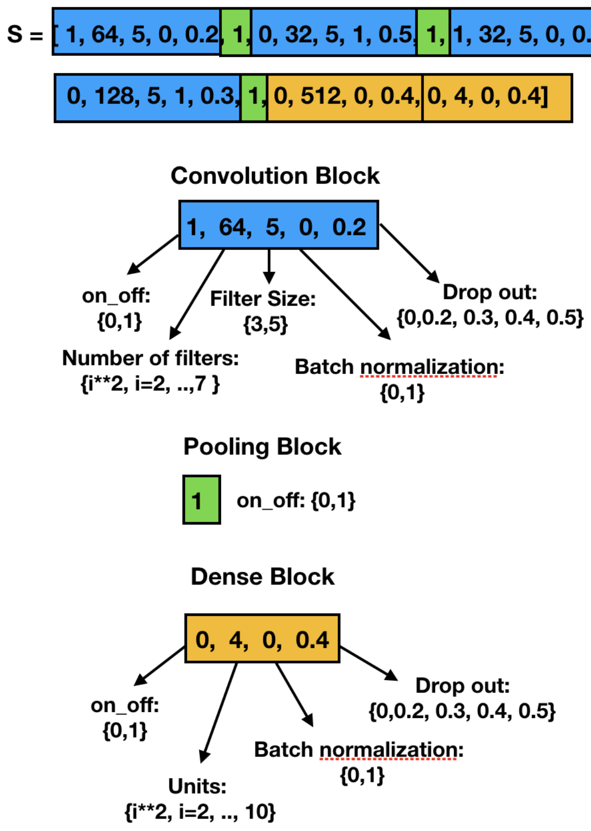
Mutation probability $p(i)$ for a certain point depends on the type of performed mutation: for example for mutation 'add/delete layer', probability of mutation is $p('on\_off') = 0.5$, as $'on\_off'$ parameter domain has only two possible values $\{0, 1\}$.

2.3.1 Fitness Function

In [3] and other papers prediction accuracy on the test data is used as a fitness function to evaluate how good a parent CNN is for further selection. This approach directs the network to achieve higher accuracy without taking into account prediction time, amount of network parameters and other important aspects. It can be beneficial to add other features to the fitness function. For example, addition of prediction time and number of network parameters can help to speed up the search. Also prediction speed and network size for certain applications can be as important as accuracy. One potential example of such application is high speed industrial line, where a couple of percent in prediction accuracy can be sacrificed to achieve faster speeds. Moreover, depending on the data specifics smaller models can produce better or comparable results (mostly due to over fitting, wrong parameter tuning, etc.). For example, in this project the model constructed from the biggest possible sequence S resulted in lower accuracy ($\sim 73\%$) than the model that has only one convolutional layer ($\sim 78\%$).
In this project we introduce prediction time to the fitness function as additional feature. The modified fitness function $fit2$ has feature vector:

$$\phi(x) = \{prediction\_accuracy, prediction\_time\}$$

where $prediction\_accuracy \in [0, 1]$, and $prediction\_time$ depends on the hardware and test data.
The open question is which exactly prediction time to use, eg. prediction time on one example or batch of examples. As the main objective of this research is to


S = [1, 64, 5, 0, 0.2, 1, 0, 32, 5, 1, 0.5, 1, 1, 32, 5, 0, 0.5, 1, 1, 8, 5, 1, 0.3, 1, 0, 64, 3, 1, 0.3, 1,
0, 128, 5, 1, 0.3, 1, 0, 512, 0, 0.4, 0, 4, 0, 0.4]
Convolution Block
1, 64, 5, 0, 0.2
on_off: {0,1}
Filter Size: {3,5}
Drop out: {0,0.2, 0.3, 0.4, 0.5}
Number of filters: {i**2, i=2, ..,7 }
Batch normalization: {0,1}
Pooling Block
1
on_off: {0,1}
Dense Block
0, 4, 0, 0.4
on_off: {0,1}
Drop out: {0,0.2, 0.3, 0.4, 0.5}
Batch normalization: {0,1}
Units: {i**2, i=2, .., 10}

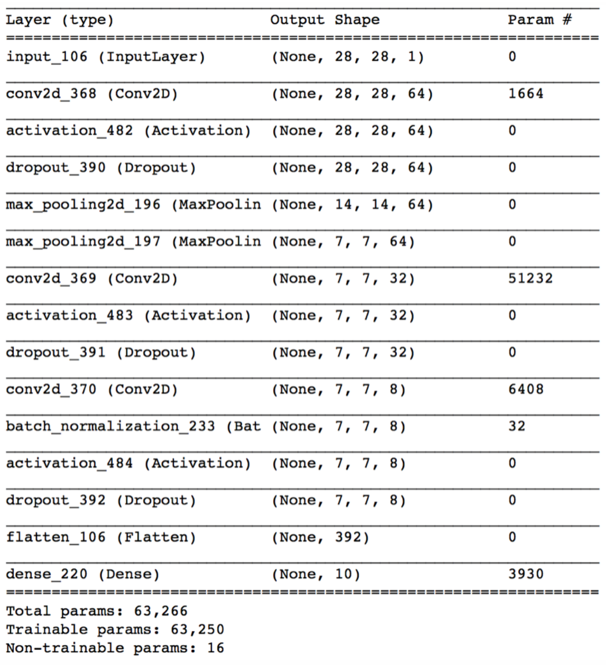

```
Layer (type)                 Output Shape              Param #
=================================================================
input_106 (InputLayer)       (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_368 (Conv2D)          (None, 28, 28, 64)        1664
_________________________________________________________________
activation_482 (Activation)  (None, 28, 28, 64)        0
_________________________________________________________________
dropout_390 (Dropout)        (None, 28, 28, 64)        0
_________________________________________________________________
max_pooling2d_196 (MaxPoolin (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_197 (MaxPoolin (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_369 (Conv2D)          (None, 7, 7, 32)          51232
_________________________________________________________________
activation_483 (Activation)  (None, 7, 7, 32)          0
_________________________________________________________________
dropout_391 (Dropout)        (None, 7, 7, 32)          0
_________________________________________________________________
conv2d_370 (Conv2D)          (None, 7, 7, 8)           6408
_________________________________________________________________
batch_normalization_233 (Bat (None, 7, 7, 8)           32
_________________________________________________________________
activation_484 (Activation)  (None, 7, 7, 8)           0
_________________________________________________________________
dropout_392 (Dropout)        (None, 7, 7, 8)           0
_________________________________________________________________
flatten_106 (Flatten)        (None, 392)               0
_________________________________________________________________
dense_220 (Dense)            (None, 10)                3930
=================================================================
Total params: 63,266
Trainable params: 63,250
Non-trainable params: 16
```

Figure 1: Architecture Specification

investigate how addition of prediction time feature can help to speed up the search we used prediction time of full batch of test data. This allows to account for parallel processing and check real running time for a big batch of data instead of only one example. However, for applications where prediction time of one example is more critical, it worth to use one example to calculate prediction time.
We also tried to look into 'number of parameters' as a feature for the fitness function, but results are not representative and require more investigation. Due to time constraints we left them out from this report and suggest it as one the next steps for further research.

Weights for Fitness Function:

Introducing time to the fitness function can produce instability in results. As was mentioned above *prediction_time* depends on the hardware and amount of test data. With increase in dataset size, prediction time also grows. Moreover, depending on the hardware even the same data volume can propagate with different speeds through the network.
The range between minimum and maximum prediction time and standard deviation can also vary. For example, we observed a bigger range for experiments on CPU versus GPU. This happens due to the parallel nature of GPU calculations. Also for different applications the acceptable prediction time and accuracy is different, therefore we can not suggest universal weights for the *prediction_time* feature applicable for all applications in this report. However, we provide a recipe and show how we approach weight selection for this project and Fashion-MNIST dataset in the 'Fitness Function Test' in section 3.4, and suggest to use $weight\_time = -1$ and normalized fit2 as default.

2.3.2 Selection Strategy

Selection strategy is a very important aspect in evolutionary algorithm performance. The first obvious choice is to pick two best networks and cross them with each other to produce children networks. This strategy can seam attractive, but this approach can easily lead to local maxima, especially taking into account that random selection is used for the first population. In general, in the literature binary tournament selection strategy [15] is used to select two superior parents from population P_i for child network creation:

```
for i=1, …, population size:
    Randomly choose two sequences from $P_i(S)$
    Choose best sequence from this two sequences $S_1$
    Randomly choose two sequences from $P_i(S)$
    Choose best sequence from this two sequences $S_2$
    Get $S_{crossed}$ = Crossover($S_1$, $S_2$)
    Mutate($S_{crossed}$) and add to new population
```

Even thought binary tournament selection strategy partially solves the local maximum problem, networks are still being selected only from the previous parent population and better architectures can be overlooked. As two sequences are chosen randomly from $P_i(S)$, the best models from the previous parent population can be missed in selection, eg. choosing the best model from two random models, cross them, and repeat n times does not guarantee to cross two best models in the population.
To solve this issues we try to add following strategies to the binary tournament selection:
1. Assure that two best models in parents sequences are crossed with each other

```
for i=1, …, population size:
  Choose two best sequences $S_{top2}$ in $P_i(S)$
  Get $S_{crossed}$ = Crossover($S_{top2}$)
  Mutate($S_{crossed}$) and add to new population
```

2. Add epsilon-greedy to the algorithm (with exploration probability of 20%)

```
for i=1, …, population size:
  Construct (0.2* population_size) new random
  sequences
  Mutate them and add to population
```

3. Use Gibbs sampling instead of randomly sampling two models in tournament selection. Gibbs sampling is performed from full population of explored so far networks and weights are equal to the normalized prediction accuracy.

```
Calculate Gibbs weights for parent population
for i=1, …, population size:
    Choose sequence from $P_i(S)$ with probability
        proportional to Gibbs weights
    Mutate($S_{crossed}$) and add to new population
```

Moreover, we also assess performance of random selection, pure Gibbs sampling, and Gibbs sampling combined with epsilon. Pure Gibbs sampling algorithm is presented below:

```
Calculate Gibbs weights for parent population
for i=1, …, population size:
    Choose two sequences from $P_i(S)$ with probabilities
        proportional to Gibbs weights
    Choose best sequence from this two sequences $S_1$
    Choose two sequences from $P_i(S)$ with probabilities
        proportional to Gibbs weights
    Choose best sequence from this two sequences $S_2$
    Get $S_{crossed}$ = Crossover($S_1$, $S_2$)
    Mutate($S_{crossed}$) and add to new population
```

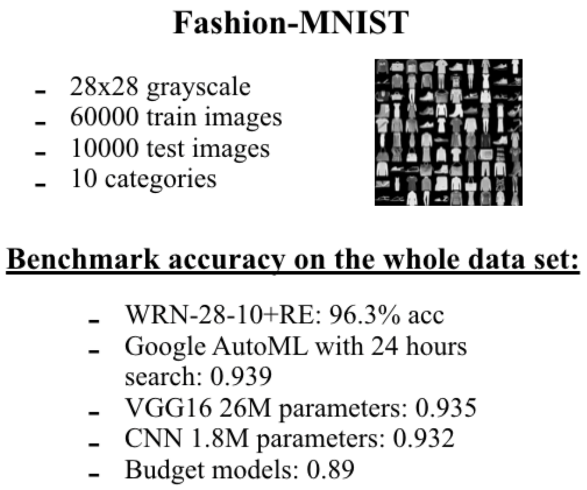
Stability test for each of this strategies compared to random and pure binary tournament and is presented in section 3.3.

3. Experiments and Results

In this section results for three tests are presented, namely stability, fitness function and performance test.

In stability test selection strategies are compared with each other. Fitness function test explains how weights and features are selected for the fitness function. Performance test is a final test to compare performance of all strategies described in this report.

3.1 Dataset



Fashion-MNIST

- 28x28 grayscale
- 60000 train images
- 10000 test images
- 10 categories

Benchmark accuracy on the whole data set:

- WRN-28-10+RE: 96.3% acc
- Google AutoML with 24 hours search: 0.939
- VGG16 26M parameters: 0.935
- CNN 1.8M parameters: 0.932
- Budget models: 0.89

Figure 3: Fashion_MNIST Benchmark [16],[17]

The Fashion-MNIST [17] dataset is chosen for experiments. It contains 60000 train and 10000 test 28x28 grayscale images which can be classified in 10 categories. The nature of this project requires to construct and train more than ten thousand of CNNs, therefore this dataset is the perfect choice due its relatively small size. Moreover, there is a lot of benchmark data available. The short summary is presented on the figure 3

3.2 Hardware Acceleration

For acceleration we explored two approaches: speed up individual experiments and speed up experimentation as a whole. To the former, we leveraged multi-processing to accelerate data loading. This achieved an improvement of 10% of total train time on small experiments. Increased RAM was also effective as data could stay in memory and no reloading between iterations was required. After addressing memory management issues in the framework, the system could run occupying at most 2.5 GB of RAM, and $\sim$ 1.5 2GHz CPU cores. Next, we tried to use two Nvidia K80 GPUs in one instance. The framework struggled to leverage multiple GPUs simultaneously (one remains underutilized), however, adds resource sharing. As RAM, CPU, and disk cost is shared across experiments it allowed for two experiments (one per GPU) to run concurrently, resulting in cost minimization.

With the introduction of prediction time into the fitness function, a controlled environment without resource sharing is needed to comparatively measure the performance. Thus, we returned to using one GPU (Nvidia P100) and one experiment per instance. We switched to the P100 GPU because it achieved 2x faster total search time vs K80, likely due to the increased compute and more VRAM availability.

To the second goal, the most effective way is to run multiple identical hardware instances in parallel and orchestrate different experiments across them - that way the resource isolation could be guaranteed without needing to run experiments in sequence.

3.3 Stability Test

The objective of the stability test is to understand the nature of selection methods, namely how a certain selection method influences the mean and the standard deviation of populations with increasing iterations (generations).

In the Figures 4-8 best prediction accuracy for each sample strategy and each iteration are shown together with the mean and standard deviation of all prediction accuracies across iterations.

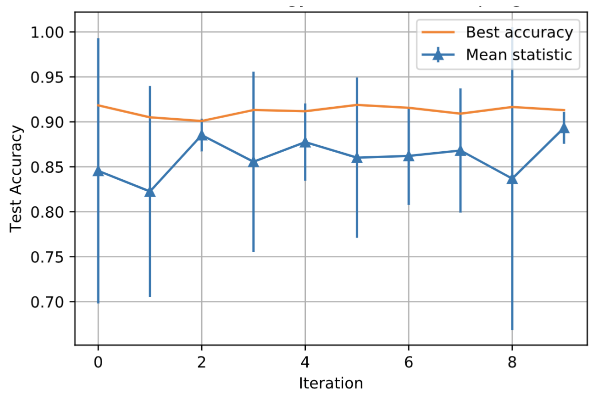


Figure 4: Stability Test for Random Selection

1. As expected random sampling is highly unstable due to the pure random nature of the mean and standard deviation. Therefore this selection strategy does not guarantee to find a model with good prediction accuracy.

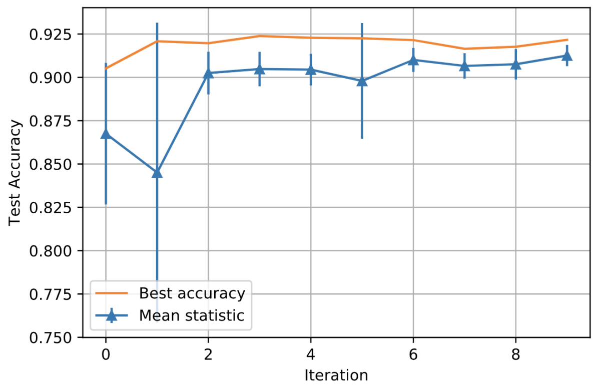


Figure 5: Stability Test for Tournament Selection

2. Tournament sampling has the most stable growing mean/best accuracy. As every strategy it starts with instability due to the random initial population, but stabilizes more with each new iteration. This happens as only high performing parents are crossed with each other. Oc-

casionally small jumps occur due to mutations and randomness in crossover index and binary parent selection. However, for binary tournament selection this happens infrequently. Therefore, it is fair to say that this selections method does not fully solve the problem of local maximum. This result is anticipated, as only networks from previous population are crossed.

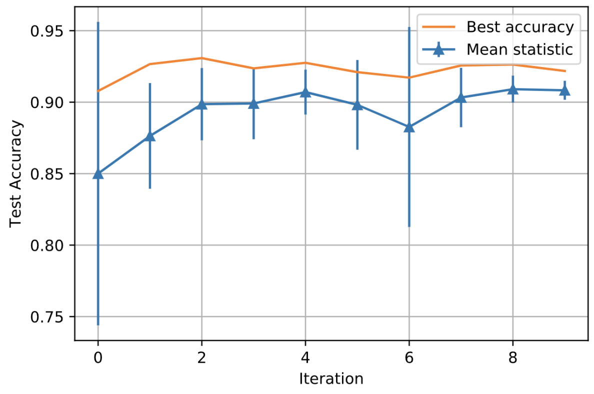


Figure 6: Stability Test for Tournament Selection With Epsilon-greedy

3. Epsilon-greedy introduces more instability to tournament and helps the model move out from the local maximum faster. The graph supports this conclusion by showing higher magnitude jumps in both accuracy mean and standard deviation. Across all experiments performed for the tournament-epsilon strategy, we observed these jumps to happen more frequently than for the pure tournament strategy, which is expected. The idea of epsilon-greedy is to explore the whole population of possible children with some exploration probability, e.g. add fully random networks to the children population in every iteration. This leads to totally new configurations that are not in the parent population, and lower mean and higher standard deviation, pushing the model out from the previous results.

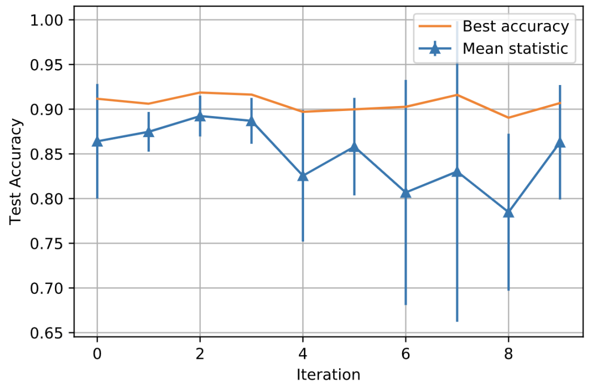


Figure 7: Stability Test for Gibbs Sampling

4. In initial iterations Gibbs sampling behaves very similar to the tournament selection. However, with growing numbers of iterations as we sample from the full population of explored networks, it behaves more like uniform random sampling from explored population. This happens due to normalization of weights i.e. with increasing iterations the population of explored networks gets bigger and weights get smaller and consequently very similar to each other. For example, if to assume sum of all accuracies to be equal to 40.9 (for a full population of 50),
$weight\_accuracy(acc = 0.93) = 0.93/40.9 = 0.023$, and
$weight\_accuracy(acc = 0.85) = 0.85/40.9 = 0.021$.
Even though accuracy= 85% is significantly lower than accuracy = 93%, weights 0.021 and 0.023 are very similar to each other.
The same behavior is observed when Gibbs sampling is used instead of random sampling to select models in tournament selection. Tournament with random binary sampling performs very similar to tournament with Gibbs binary sampling. Another, and probably better strategy, would be to normalize Gibbs weights and select parents only from the current parent population or only from a number of top performing and explored so far parents. We did not explore this sampling type in this project and suggest it as next step.

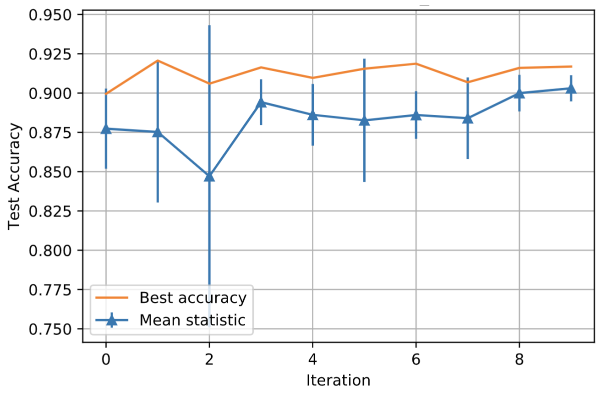


Figure 8: Stability Test for Gibbs Sampling

3.4 Fitness Function Test

The objective of the fitness function test is to look deeper into prediction time feature and choose applicable feature representation and weights for the performance test.
We selected two types of feature representation for prediction time: pure and normalized.

3.4.1 Pure Prediction Time

Initially we considered multiple representations of prediction time, e.g. log, x^2. As our main goal is to speed up the search, it is more natural to perceive time as a linear feature for this task, thus we decided to use pure prediction time.

$$fit2 = w * \phi_1(x)$$

$$\phi_1(x) = \{pred\_acc, pred\_time\}$$

The pure prediction time is useful for the real world applications when hardware can not be upgraded, and an engineer tries to find a solution that fits into the allowed time window. However, this approach can not be used without additional weights tuning. Weight needs to be

adapted to match with a mean absolute value of prediction time, and this value varies across applications.
For example, we observed the following prediction time spread for fashion-MNIST on CPU 3.1 GHz and GPU Nvidia Tesla P100. Change in image size, amount of test data and hardware effects this figures.
Table 3.4: 3

Hardware	Prediction Sample Size	Minimum Time, sec	Maximum Time, sec	Mean Time, sec	Standard Deviation, sec
CPU 3.1 GHz	1000	0.06	2	0.41	0.54
GPU Nvidia Tesla P100	10000	0.4	1	0.6	0.12

Weight Selection
To choose weights for the fitness function we need to answer the question on how much to reward or penalize a certain feature. To reward good accuracy and penalize high prediction time the weight for accuracy ($weight\_acc$) has to be positive while the weight for prediction time ($weight\_time$) has to be negative. We assumed that $weight\_acc = 1$ is constant and $weight\_time$ varies from -1 to -0.005, and performed 4 experiments on CPU for each weight. Results are presented on the figure 9.

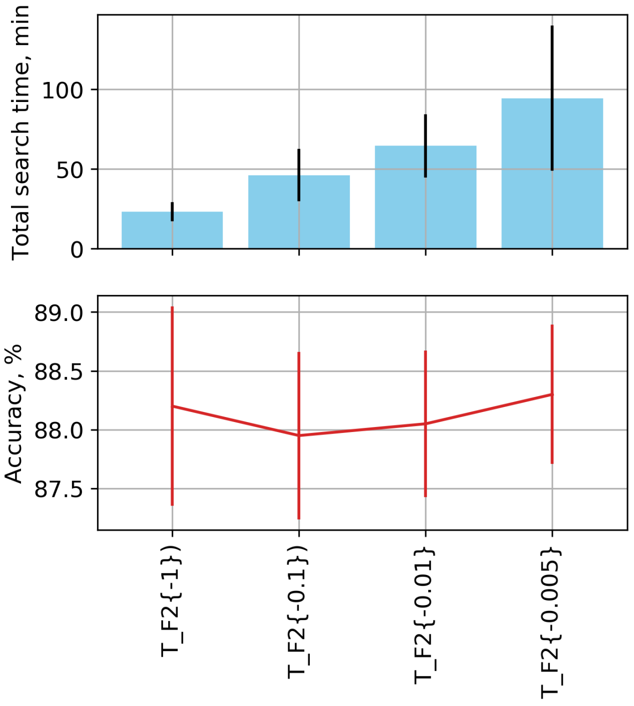


Figure 9: Total search time and accuracy vs Fit2 weight (on CPU)

From the figure following can be seen:

1. It can not be concluded that accuracies are different for different weights
2. Mean total search time is higher for smaller $|weight\_time|$ (absolute value). Moreover, mean search total time trend is growing with decreasing $|weight\_time|$.
   This result is not surprising as with reduction in $|weight\_time|$ the total influence of total search time on fitness function decreases and consequently, accuracy gets more important

3.4.2 Normalized Prediction Time

The pure prediction time, even if it is calculated on one example does not solve the issue with mean and standard deviation differences when various hardware and image size is used. To solve this issue we introduced normalized prediction time:

$$fit2n = w * \phi_2(x)$$

$$\phi_2(x) = \{pred\_acc, \frac{pred\_time - min\_pred\_time}{max\_pred\_time - min\_pred\_time}\}$$

The normalized feature representation is more universal and allows to speed up search process for any hardware and amount of given data as it is rescaling the range of features to a scale matching prediction accuracy range of [0, 1]. We didn't perform the full weight selection test for normalized function as similar to Figure 9 results are expected. Instead we ran an additional experiment called $'T\_F2\_norm\_0.01'$ with $weight\_time = 0.01$ (see performance test). For the further experiments we use $weights\_normalized = \{1, -1\}$

3.5 Performance Test

The objective of performance test is to compare performance of all described above sample strategies and fitness functions. Due to limited compute resources the test was organized in stages. In the first two stages the main goal is to compare performance of all described below methods, e.g:

Abbreviation	Decription
R^F1	Random with fit1(only accuracy)
T^F1	Tournament selection and fit1
T^E^F1	Tournament selection with epsilon-greedy and fit1
G^F1	Gibbs selection and fit1
T^G^F1	Tournament selection with Gibbs sampling and fit1
T^E^F2{-1}	Tournament selection with epsilon-greedy and fit2 (w_time=-1)
T^G^E^F2{-1}	Tournament selection with Gibbs sampling, epsilon-greedy and fit2 (w_time=-1)
T^F2n{-1}	Tournament selection with normalized fit2 (w_time=-1)
T^E^F2n{-1}	Tournament selection with epsilon-greedy and normalized fit2 (w_time=-1)
T^G^E^F2n{-1}	Tournament selection with Gibbs sampling, epsilon-greedy and normalized fit2 (w_time=-1)
T^F2n{-0.01}	Tournament selection and normalized fit2 (w_time=-0.01)

In stage 3 and 4 we target to achieve top accuracy to compare with benchmark networks, therefore we increase the amount of iterations, population size, epochs and perform tests only on one of the top performing strategies: tournament epsilon with fitness function fit1.

Stage 1:
Four experiments for each sample strategy are performed on the Fashion-MNIST dataset. $\sim 10\%$ of the dataset is used, i.e. 5000 train samples and 1000 test samples. The data is chosen for each label uniformly so that the resulting smaller dataset is representative. The experiment is performed using only a 3.1 GHz CPU and 16GB RAM. Running parameters are: population size $n = 10$, iterations (generations) $m = 10$, epochs = 3, *batch_size* = 32.
Results for the mean and standard deviation across 4 experiments are shown on the graphs below. Total search time and model accuracy for the best model found by the method vs selection strategy are presented on the figure 10. Comparison of this results clustered by fitness function type is done on the figure 11 :

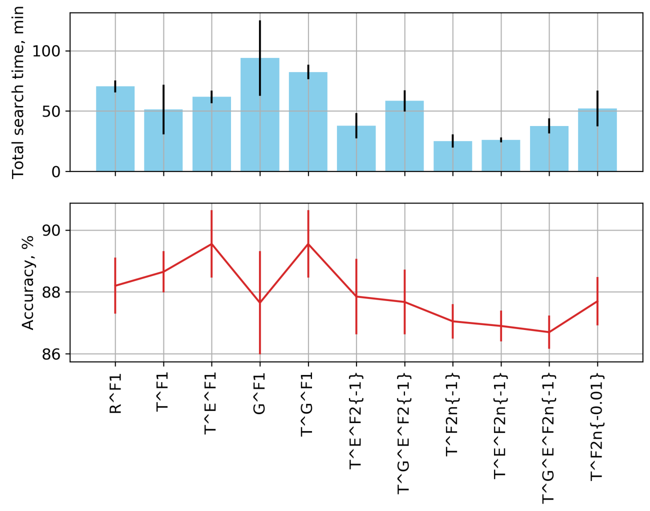


Figure 10: Accuracy and Total search time statistics for Stage 1

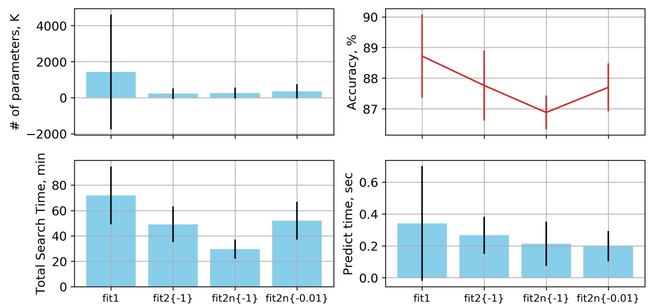


Figure 11: Comparison of fitness functions for Stage 1

From figure 10 and 11 following can be concluded:

1. Introduction of normalized prediction time with *weight_time* $= -1$ reduced mean training time by approximately a factor of two in performed experiments. Moreover, we observe slight reduction in accuracy, i.e. $\sim 1\%$ for fit2n{-1}

2. Observed lower total search time for fit2n{-0.01} and fit2n{-1} across preformed experiments

3. Comparing results for fit2n{-1} vs fit2n{-0.01} we observe an increase in total search time with growing $|weight_time|$. Moreover, there is increase in mean accuracy, but we can't state it confidently. This results aligns with the Fitness Function Test results.

4. Standard deviation of prediction time for fit1 is significantly bigger than standard deviation for fit2. This result is expected, as prediction time is only accounted for in fit2. The same can be concluded for the number of parameters. Even though we do not account directly for number of parameters in fit2, this metric is partially correlated with prediction time e.g. for smaller networks prediction times tend to be smaller. Remark: prediction time does not only depend on the number of parameters, but also on hardware type, amount on layers, etc., therefore smaller networks do not always result in smaller prediction time.

5. Tournament with epsilon and tournament with Gibbs sampling and fit1 are the top performers in four experiments we performed.

Stage 2:
Fours experiments for each sample strategy are performed on the full Fashion-MNIST dataset i.e. 60000 train samples and 10000 test samples. The experiment is done using Google cloud instances with: GPU Nvidia Tesla P100. Running parameters: Population size $n = 10$, iterations (generations) $m = 10$, epochs = 10, *batch_size* = 200. Results graphs are the same as in stage 1 and are presented on the images below:

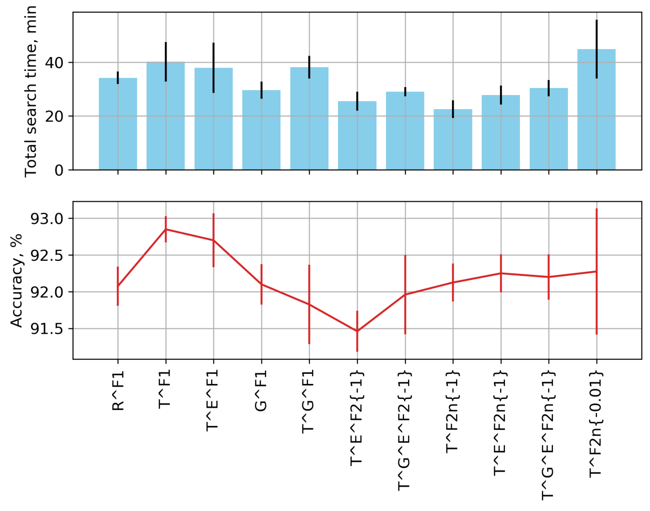


Figure 12: Accuracy and Total search time statistics for Stage 2

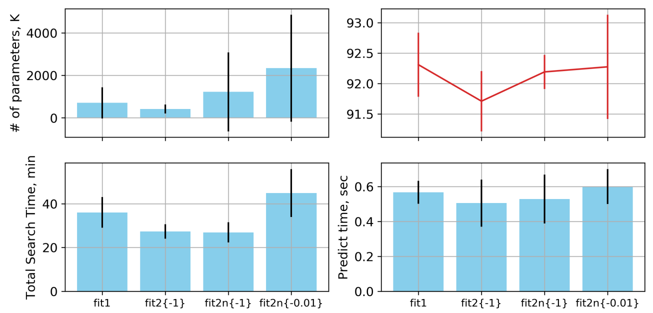


Figure 13: Comparison of fitness functions for Stage 2

From figure 12 and 14 following can be concluded:

1. Tournament and tournament-epsilon with fit1 fitness function are top performers in terms of accuracy in performed experiments. Taking into account results in stage 1 we will perform experiments for tournament-epsilon greedy strategy in stage 3

2. In performed experiments introduction of fit2 function with high penalty (e.g. $weight\_time = -1$) reduced search time by $\sim 20\%$. This is much lower than in the CPU experiment, however, not surprising. As was mentioned previously, this happens due to the parallel nature of GPU calculations combined with the fact that Fashion-MNIST is a relatively small dataset i.e. even though the mean prediction time for test set is similar, the standard deviation for GPU-backed experiments is smaller.

3. From the graphs we can see that ncrease in total search time follows growing $|weight\_time|$. This results aligns with all previous findings.

Stage 3:
One experiment is performed only for $T \wedge E \wedge F1$ strategy on full Fashion-MNIST dataset. The experiment is done using Google cloud instances with: GPU Nvidia Tesla P100
Running parameters: population size $n = 15$, iterations (generations) $m = 20$, epochs $= 15$, $batch\_size = 200$. The model achieved best accuracy of 93.7% with 1.7M parameters and 2h of search time. The best network is smaller and has higher accuracy than VGG16 with 26M parameters and $accuracy = 93.5\%$. Also, the search lasted only 2 hors on a single GPU, that is 12 times faster than the benchmark Google AutoML search. However, AutoML model is slightly more accurate: 93.9%.

Stage 4:
For the stage 4 the same experiment as in stage 3 was performed with increased number of iterations equal to 25 and population size of 20, resulting in search time increase to $\sim 4h$.
The goal of this test was to get higher accuracy than in stage 3, however 93.4% test accuracy for the best model was achieved. This is 0.3% less than the previous result. There can be multiple reasons why increase in accuracy was not observed. For example, due to randomness in selections, crossover and mutations it can be just misfortune. Another reason can be a wrong initial design of the sequence S (not enough layers, too simple structure, etc.) resulting in impossibility to achieve higher accuracy for any of the possible architectures.

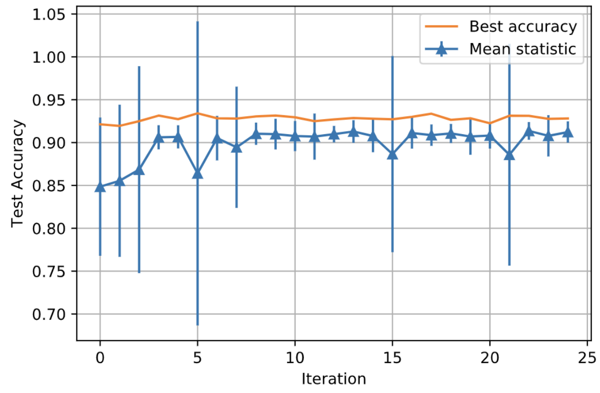


Figure 14: Stability statistic for Stage 4 experiment

From the stability statistics presented in the image above can be seen that the model after eights iteration becomes relatively stable and the best accuracy is not improving. After jumps (most probably due to epsilon-greedy) in 15^{th} and 21^{st} iterations, the model slightly moves out from the local maximum, but after some iterations comes back. Two jumps are not enough to conclude that model is not able to improve further, thus it would be interesting to observe the behavior of the model for more than 25 iterations. Taking into account that the model stabilizes around 93.5% most probably the maximum accuracy that predefined sequences S design is able to provide is not much higher. Therefore, to improve accuracy it is not enough to increase number of iterations, but rather change design of the sequence S architecture.

4. Conclusions

In this project automated CNN architecture search model for image classification is created. The model size is restricted to 6 convolutional and 2 dense layers due to computation constraints, however is extendable to more layers. By default the model uses tournament epsilon selection strategy and fitness function fit1 (only accuracy), however by changing run parameters one can choose any selection method and fitness function we described in this report. In case of fit2 we suggest to use default $weight\_time = -1$, but we included weights as run parameters for special cases.
With default settings of the model and 2h search we were able to find best CNN that achieves 93.7% accuracy and 1.7M parameters on Fashion-MNIST dataset. This result is better than VGG16(93.5%) and CNN with 1.8M(93.2%), and much faster than Google AutoML(93.9%) 24h search.
We compared different strategies with random and binary tournament strategy and came to the following conclusions:

1. Adding prediction time to the fitness function reduced run time by approximately a factor of two on CPU and 20% on GPU with only a $\sim 1\%$ drop in accuracy. However, this only works in case if $weight\_time$ is significant enough to reward faster networks, therefore to avoid parameter tuning it is better to use normalized prediction time with $weight\_time = -1$

2. Adding epsilon-greedy to tournament selection helps the model to jump out from a local maximum.

3. Gibbs sampling from full population performs similar to tournament selection only for initial iterations, but for later iterations behaves more like a random sampling.

4. The best accuracy models across experiments are tournament and tournament epsilon

5. Next Steps

Suggested next steps:

1. Longer run for a top performing model from stage 4

2. Perform more experiments for each strategy and stage to gain better statistical confidence

3. Different sequence S design. Potentially add more dense layers, include skip connections, use different optimization techniques, etc.

4. Gibbs sampling from only previous generation or from a number of top performers in full population

5. Include number of network parameters into fitness function

References

[1] Zoph, B., and Le, Q. V. *Neural Architecture Search with reinforcement learning* 2016.

[2] Yanan Sun., Bing. Xue, Mengjie Zhang, Gary G. Yen. *Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification* 2019.

[3] Petra Vidnerova, Roman Neruda. *Evolution Strategies for Deep Neural Network Models Design* 2017.

[4] Krizhevsky A., Sutkever I., and Hinton G.E. *Imegenet Classification with Deep Convolutional Neural Networks* in Advances in Neural Information Processing Systems, 2012, pp.1097-1105

[5] Clark C., Storkey A. *Training Deep convolutional Neural Networks to Play Go* in Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015, pp. 1766-1774

[6] Sutskever I., Vinyals O., and Le V. *Sequence to Sequence Learning with Neural Netwroks* in Advances in Neural Information Processing Systems, 2014, pp.3104-3112

[7] Simonyan K., Zisserman A. *Very Deep Convolutional Networks for Large-Scale Image Recognition* in International Conference on Learning Representation, 2015

[8] Xie S., R. Girshick, P.Dollar, Tu Z., and He K. *Aggregated Residual Transformations for Deep Neural Networks*2017

[9] Szegedy C., Vanhoucke V., Ioffe S., Shlens S., and Wojna Z. *Rethinking the Inception Architecture for Computer Vision* in IEEE Comference on Computer Vision and Pattern Recognition, 2016

[10] Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger *Densely Connected Convolutional Networks* 2018

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun *Deep Residual Learning for Image Recognition*, 2015

[12] Hien Pham, Melody Y. Guan, Barret Zoph, Quoc V.Le, Jeff Dean *Efficient Neural Architecture Search via Parameter Sharing* 2018

[13] Xie L., and Yuille A. *Genetic CNN* in Proceedings of 2017 IEEE International Conference on Computer Vision, 2017

[14] Liu H., Simonyan K., Vinyals O., et al. *Hierarchical Representations for Efficient Architecture Search* , 2018

[15] Miller B.L., Goldberg D.E., et al. *Genetic Algorithms, Tuornament Selection, and the Effect of Noise* , 1995

[16] Heinz S. *A performance benchmark of Google AutoML Vision using Fashion-MNIST* , 2018

[17] Kashif Rasul, Han Xiao *Fashion-MNIST* Research Project, Zalando Research, 2018