Final Report of CSE 891 Section 002: Deep Learning in Biometrics

Evolving Deep Networks for Computer Vision Tasks (Updated Version)

Honglin Bao

NSF BEACON Center for the Study of Evolution in Action & CSE Department
Michigan State University
East Lansing, Michigan USA 48823

* Considering the intention is to complete a technical report, i did not strictly follow academic conference/journal writing formate.
* Code available: https://github.com/hlbao/Neuro-Evolution
* My GitHub: https://github.com/hlbao, Email: baohlcs@gmail.com for discussion.

1. Mainstream Limitation and Motivation

Deep learning has been making a huge success recently. Under the surface of deep learning is the latest form of a technology that has been around for decades. It is an artificial neural network (ANN). ANN researchers used programs to simulate these neurons and the signals transmitted between them, and obtained a relatively vague process of simulating what is happening in the brain. Of course, there are many differences between the two. But the real challenge is that simply connecting a bunch of neurons to each other and letting them share signals with each other does not produce intelligence. On the contrary, *intelligence comes precisely from how neurons are connected*.

For example, a neuron that strongly affects another neuron is said to have a large weight connected to its partner. Here, the weight of this connection determines how the neurons affect each other, resulting in a specific pattern of neural activation in the neural network in response to input to the neural network (e.g., input from the eye). To get an intelligent network, the corresponding challenge becomes *how to determine the weight of the network connection.*

Usually, no one will manually calculate the weight of the connection (considering that modern ANNs generally have millions of connections, we can understand why the manual method is not realistic). Conversely, finding the right connection weight for a task is regarded as a learning problem. In other words, the researchers spent a lot of energy inventing a method for ANN to learn the best weights for specific tasks. The most common way to learn weights is to compare the output of the ANN (for example, that looks like a dog) with the standard answer, and then change the weights through gradient descent (just a classic method), so that the next time the possibility of correct output higher. I personally would say, this way to go about finding the right hyper parameters is through brute force trial and error. After training for such blind countless comparison examples (there may be millions of them), the neural network can begin to assign the correct weights to accurately answer various questions. Frequently, the ability of neural networks can even be generalized to answer questions that it has not encountered, as long as these questions are not so different from the questions seen in previous training. So far, ANN has basically learned to deal with specific problems. A common method of adjusting weights is the stochastic gradient descent method, which is a very popular component in deep learning mentioned earlier. The implementation of deep learning in recent years is basically a massive training of ANNs composed of very many layers of neurons until it can be so called ‘deep’. This is also because of the help of powerful computing hardware that has emerged in recent years.

But here is a question that I have not mentioned yet, that is, how do we first decide who (neural) is connected to whom? In other words, our brain is *not only determined by the connection weight, but also by the internal structure*. Stochastic gradient descent cannot even solve the structure, but just tries its best to optimize the weight of connections. To put it more clearly, deep learning has traditionally focused on learning by programming ANN structure, while neuron evolution focuses on the original architecture of the brain network itself. It can include who connects with whom, the weight of the connection, and (sometimes) how these connections are allowed to be changed. This is the motivation of this project, where I want to apply evolutionary search to evolve a DNN for computer vision task.

2. Genetic Algorithm, a very quick recall.

A very basic genetic algorithm can be shown in the following:

Create initial population and encode them. There are some skills to encode population. For example, how to move toward your target even at the very starting stage through a proper encoding schema without significantly increasing computational complexity? Then, evaluate the fitness of individuals in a population based on the principle of proportionality (the chance to be selected for the next generation of an individual with a high fitness score is also higher). The reason for not only picking the highest score individual is that doing so may converge to the local optima point, not the global. So in this selection process we must consider not only fitness scores but also diversity. Then we add variation into the population (crossover and mutation), to generate next generation of offspring and evaluate them again, repeat this process until the conditions for stopping the iterations are met.

3. Dataset

In this project, i plan to study a face classification problem on dataset FER-2013.
for the dataset processing part, i refer the code in github: https://github.com/gitshanks/fer2013

data set: https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data

The data consists of 48x48 pixel grayscale images of faces. The faces have been automatically registered so that the face is more or less centered and occupies about the same amount of space in each image. The task is to categorize each face based on the emotion shown in the facial expression in to one of seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). [1] so we actually treat this as a classification task with the contents of computer vision/face recognition.

4. How to apply GA to evolve a DNN?

First of all I want to describe my ideas generally, in the next section I will elaborate from the implementation details.

4.1 General Idea

We will try to develop a fully connected network (MLP) and Conventional Neural Network (CNN). Our goal is to find the best network configuration parameters for face classification, especially expression classification tasks. We will adjust four parameters (similar to hyper parameter tuning):
a. 'nb_neurons': [32, 64, 128, 256, 512, 768, 1024], /layer
b. 'nb_layers': [1, 2, 3, 4],
c. 'activation': ['elu', 'relu', 'sigmoid', 'tanh'], dense layer activation function
d. 'optimizer': ['sgd', 'adam', 'adagrad',
'adadelta', 'adamax', 'nadam', 'rmsprop']

So generally speaking, how to apply GA to evolve a proper NN architecture for this kind of classification task?

First we initialize N random networks to create our population.

Second, grade each network. We train the parameter weights of each network and then see how well it performs when classifying in the test set. Since this is actually a classification task, we use classification accuracy as the fitness function.

Third, sort all networks in our population by the score (accuracy). We will retain a certain percentage of top-level networks to make it as the next generation and generate offspring. We will also randomly reserve some non-top networks. This helps to find the potential combination between the under-performer and the best performer, and also helps prevent us from falling into a local maximum. This is a point of view borrowed from the exploration in the field of reinforcement learning. Although theoretically, the standard of evaluation (fitness) is the classification accuracy of the network, but in fact we have considered both accuracy and diversity in the evaluation process.

Forth, now that we have identified the networks to be retained. For the forth step, we randomly changed certain parameters (mutations) on some networks. Suppose we started with 40 networks, we retained the top 25% (10 networks), and randomly retained 4 networks which are not quite good, as described in step 3, and some of them were mutated. We let totally 26 other networks die from our training. In order for our population to maintain totally 20 networks, we need to fill 6 new-generated networks. These six networks are derived from the crossover of parents.

Fifth: crossover
Breeding means that we select two members from a population and then give birth (generate) to one or more children, which represents a combination of their parents.
In our neural network case, each child is a combination of various random parameters from its parents. For example, a child may have the same network depth as mother, while other parameters come from father. The second child of the same parents may have the same activation function as the father, and the other as the mother. From the theory of genetic algorithm this will quickly lead to an optimized network. Here we are actually considering asexual reproduction, that is to say, assuming two parent networks, the children inherit some parameter features from one and another part from the other, and do not strictly distinguish the "gender" of the two parent networks.

4.2 Technical Implementation

in this section, i will mainly talk about the implementation details of this model.

a. how to classify?

the classify procedure as shown in figure 1, is mainly borrowed from https://github.com/hlbao/Neuro-Evolution/tree/master/fer2013-master
https://github.com/gitshanks/fer2013

i directly use confmatrix.py for classification also submitted that package (fer2013-master folder) to my own github repository. this part is not my original work, so i will describe other parts of this project with more efforts.

b. how we train and test the model? (see train.py)

i. for MLP

first i would clarify some parameters here, num_classes = 7 indicates seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). batch_size = 64, width, height = 48, 48 (according to FER-2013 data set 48x48

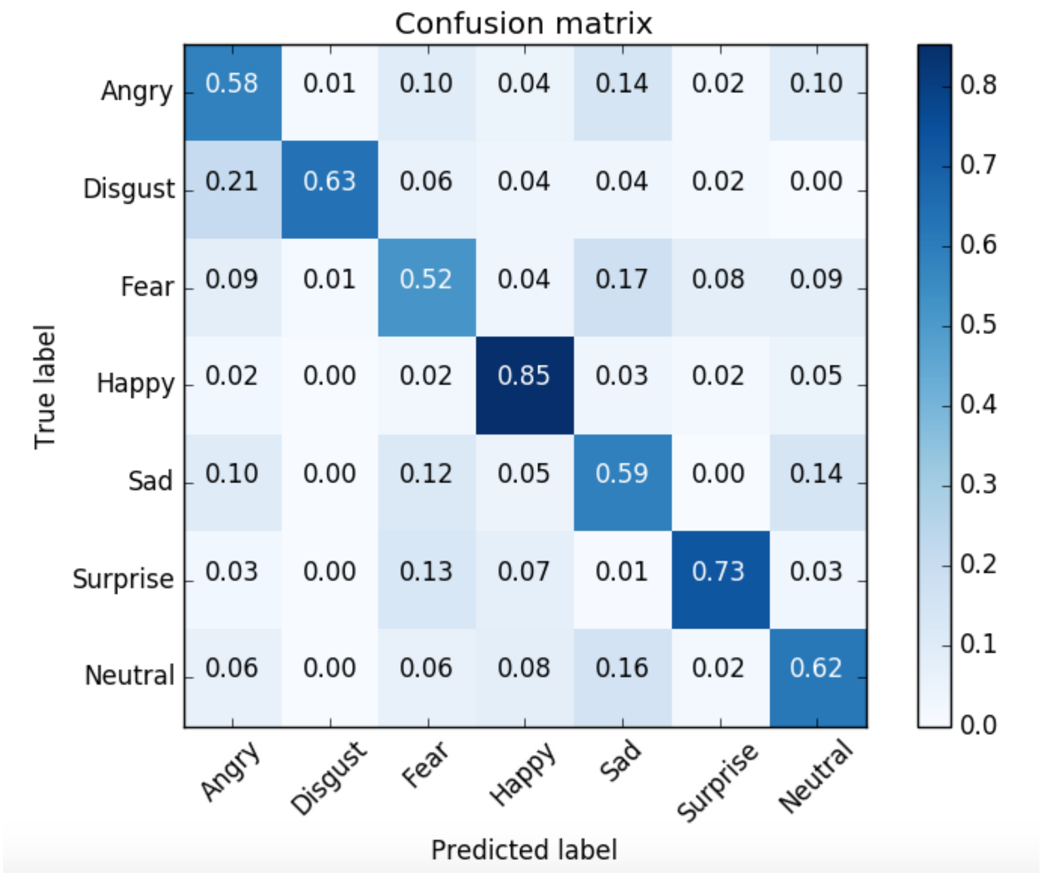


figure 1. face classification metrics.

pixel grayscale images of faces). We also split dataset into training testing data (training 90% testing 10%)

Compiling the sequential model which is a neural net, we separate sequences into input (x) and output (y) elements. We can do this with array slicing.

for example np.array(y_train),

pass the input shape into the first layer, we choose dense as the first layer, also hard-coded dropout rate for each layer being set as 0.2 to overcome overfitting limitation. For the output layer, we also set it as dense and set activation function as softmax (why?). we do not adapt any popular, well-packaged optimizer, e.g., adam, to search the best architecture. We alternatively regard the selection of optimizer as an optimization result. Since we hand-coded an evolutionary optimizer for searching, which will be described in the next section.

The evaluation metrics is accuracy (fitness). For the optimization target (i.e., the supervision of training) in this work, we choose categorical cross entropy (why?).

OK, right now we have two questions but with the same answer. Why we choose softmax as activation function and why we choose categorical cross entropy as loss function? this is because we regard this as a multi-class classification problem. let's move back to the data set. The task is to categorize each face based on the emotion shown in the facial expression in to one of seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). so we actually treat this as a classification task with the contents of computer vision/face recognition. we should also note that we assume each sample only belongs to one category rather than multiple categories (you can not say one person is laughing meanwhile crying).

so formalize this problem as one-of-many classification. Each sample can belong to ONE of C classes. The deep network will have C output neurons that can be gathered in a vector s (Scores). The target (ground truth) vector t will be a one vector with a positive class and C−1 negative classes. This task is treated as a single classification problem of samples in one of C classes. the model is compiled specifying the categorical cross entropy loss needed to fit the model. The answer of that question is quite simple: since we find categorical cross entropy loss is suitable and quite widely used in multiple classification problem in existing literatures, as a result, we simply fill in the loss='?' with categorical_crossentropy. Meanwhile, softmax is proper for the case of one-vs-all classification. If the problem is a multiple-vs-all, the activation function should be changed, such as logistic regression.

ii. For CNN

CNN is widely used in face expression classification. So for the first layer, 2D convolution layer (e.g. spatial convolution over images). If activation is not None, it is applied to the outputs as well. When using this layer as the first layer in a model, i also provide the keyword argument input_shape (tuple of integers). padding: one of "valid" or "same" (case-insensitive) and i choose same. For other layers, we hit zero and apply Max pooling operation for spatial data. Next is Flatten which converts the pooled feature map to a single column that is passed to the fully connected layer (dense).

Loss function is also categorical_crossentropy. Meanwhile, softmax is proper for the case of one-vs-all classification. Dropout is also applied for overfitting.

c. how we optimize the model training?

we apply an evolutionary search for MLP optimizer. see optimizer.py.

We start by creating a random population. This instantiates "count" networks with randomly initialized settings, and adds them to our "pop" list. This is the seed for all generations.

here we go:

```
def __init__(self, nn_param_population, retain=0.4,
            random_select=0.1, mutate_chance=0.2):
```

nn_param_population (dict): all possible network parameters when we run different generation. we want to choose a best-performer and choose these parameters accordingly. this is a combination of the four followings:

```
    'nb_neurons': [32, 64, 128, 256, 512, 768, 1024],
    'nb_layers': [1, 2, 3, 4],
    'activation': ['elu', 'relu', 'sigmoid', 'tanh'],
    'optimizer': ['sgd', 'adam', 'adagrad',
                  'adadelta', 'adamax', 'nadam', 'rmsprop']
```

retain (float): Percentage of population to retain after each generation, here is top 40%
random_select (float): Probability of a low performing network remaining in the population, here is 0.1. so actually we choose totally 0.6 \times 0.1 =0.06 of low-performer and put them into next generation together with top 0.4 high-performer, to keep a balance of accuracy and diversity.
mutate_chance (float): Probability 0.2 that a network will be randomly mutated

Current we have a naive version of this evolutionary search. this kind of "naive" is shown in initialized population generation and mutation. as i mentioned, the initial population is random. could we modify it to make the generalized population move towards the final target even at the very early starting stage? also, current version of mutation is like:

```
 # randomly mutate one of the params.
   network.network[mutation] = random.choice(self.nn_param_population[mutation])
```

could we introduce some "good" mutation rather than pure random to additionally speed up the search process? all these points should be the future work of this project.

5. Results

in this section, i will mainly talk about the result of this work from two perspectives, efficiency and accuracy. Here, we try to develop the network by applying genetic algorithms to improve the brute force search method, the purpose is to achieve the best hyper parameters within a small part of the brute force search time. This is the term of efficiency. we also focus on the accuracy to see how it successfully classifies different faces.

One point is CNN's result is really bad, i don't know the reason, but i guess the parameter candidate for search should be different for CNN and MLP. They probably also need different preprocessing of data set. In this section, i will mainly report the experiment result of MLP so how much faster and accurate is so? Let us analyze from both theoretical and experimental aspects.

Experimental perspective:

The reference group is a brute force search, which will start by running a brute force algorithm to find the best network architecture. That is, we will train and test all possible combinations of the network parameters which is listed in nn_param_population. The experimental group is a heuristic search. We want to know how much faster the experimental group is than the reference group, from the experimental results.

we run the genetic algorithm from 20 randomly initialized networks and run it for 10 generations. see main.py. for this case, the reported result is here:

52.12% accuracy, Number of layers: 2 Neurons: 512 Activation: relu Optimizer: adam (kaggle winner for this task-71.16% accuracy)

the purpose of this experiment is to compare the efficiency, so i should run the brute force search, which will start by running a brute force algorithm to find the best network. That is, we will train and test all possible combinations of the network parameters we provide. however, it takes so long time, i did not get the result of blind search so far. it should be an evidence that heuristic search of best architecture of neural nets is indeed much quicker. “With the increase of parameter complexity, genetic algorithm may bring exponential speed advantage [2]”.

the limitation right now is that accuracy is not as good as the state of the art. it may be caused by many reasons. hardware? whether being trained completely, whether the model could be more complex or deep (right now only 4 layers maximum)? time limitations? loss? all these factors will significantly influence the accuracy.

Theoretical perspective:

let’s analyze why heuristic search is faster than blind search in theory.

let’s move back to parameter set again:

a. ’nb_neurons': [32, 64, 128, 256, 512, 768, 1024], /layer
b. ’nb_layers': [1, 2, 3, 4],
c. ’activation': ['elu', 'relu', 'sigmoid', ‘tanh'], dense layer activation function
d. ’optimizer': ['sgd', 'adam', 'adagrad',
'adadelta', 'adamax', 'nadam', ‘rmsprop']

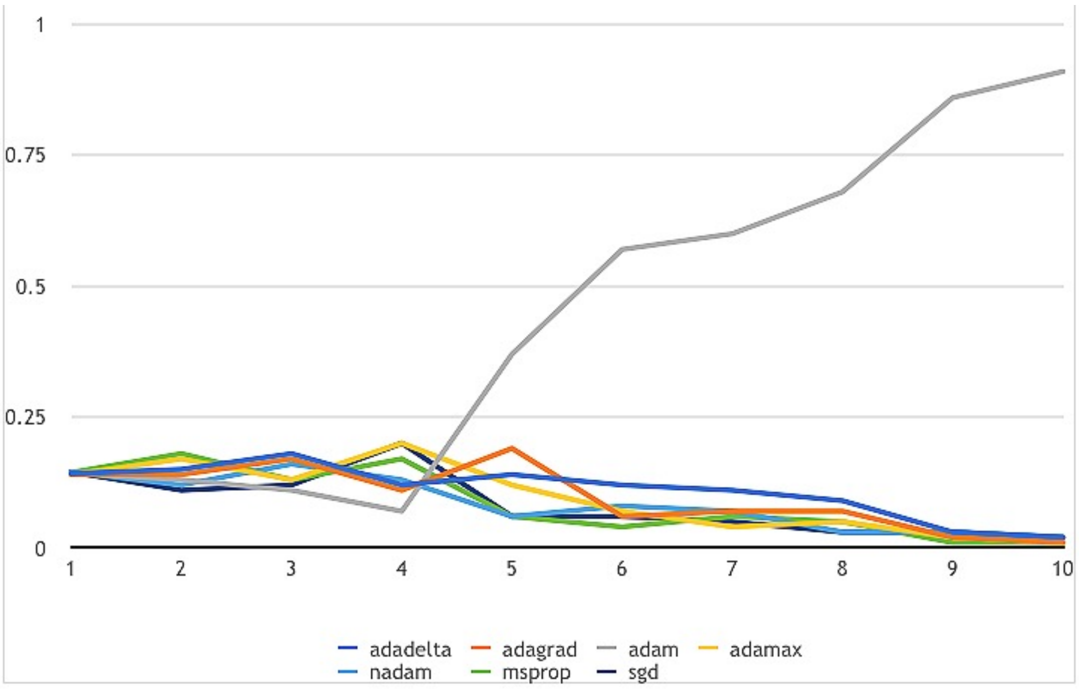
ok, so right now we have four parameters with 7, 4, 4, 7 possible settings each. so we totally have 7+4+4+7=22 candidates. Let's say it takes t minutes to train and evaluate a single network candidate on fer-2013 dataset. so try them all would take us totally 22t \times 10runs =220t minutes.
next, let's move to heuristic search part, we run the genetic algorithm from 20 randomly initialized networks and run it for 10 generations. see main.py. we keep top 40% in one generation (selection) plus 6% low-performer (diversity) into next generation. lets keep in mind these data.

Let's also say it takes t minutes to train and evaluate a single network candidate on fer-2013 dataset. for the first generation, we need to evaluate 20 networks spending 20t minutes. For the every next generation after that, it only requires (20 \times 0.46 =9.2) candidates to evaluate. so the total time should be:

20t + 9.2t \times 9runs = 102.8t minutes, around only 47% of blind search.

optimizer dynamics

in this section we explore the optimizer dynamics with fixing other parameters as neurons per layer=512, total layers=2, activation of dense layer=rerlu. so finally adam increasing from average to dominant optimizer.



As my described perviously, make deeper networks, more experiments, design novel loss function (combine accuracy and diversity), should be the future work. Since i really don’t have time for more experiments right now, even HPCC we also need to wait for submission. it is indeed a pity.

References

[1] Rose, Nectarios. "Facial expression classification using gabor and log-gabor filters." *7th International Conference on Automatic Face and Gesture Recognition* (FGR06). IEEE, 2006.
[2] Stanley, Kenneth O., et al. "Designing neural networks through neuroevolution." *Nature Machine Intelligence* 1.1 (2019): 24-35.