Stacked Autoencoder Neuroevolution

Tim Silhan
Carl von Ossietzky Universität, Oldenburg
Tim.Silhan@uni-oldenburg.de

Abstract—Choosing good hyperparameters for neural networks is not easily achieved. This paper proposes a method that automatically initializes and adjusts these hyperparameters during the training process of a stacked autoencoder architecture. Autoencoders are constructed starting from one layer in the en- and decoder and adding layers during the neuroevolution process. The fitness function of the evolutionary algorithm can also be used to optimize the quality of the dimensionality reduction. Experiments show the most significant hyperparameters and analyze their lineage during the training process. The results suggest that the method proposed in this paper outperforms the baseline on multiple datasets.

Index Terms—Neuroevolution, Autoencoders, MNIST, Fashion-MNIST, Neural Network, Evolutionary Algorithm

1 Introduction

With the rise in computational power and easy access to vast computing resources, training of deep neural networks became more feasible. Though even with the computing power available today, the curse of dimensionality [1] often limits the depth and width of the network because of the increase in training time. Furthermore neural networks (NNs) usually have to be trained multiple times, adjusting the hyperparameters and starting with different initial weights to achieve a satisfactory solution. Finding a good set of hyperparameters is not easily achieved and currently often done using rules of thumb and general guidelines, paired with manual experimentation. Due to the popularity and success of NNs they are used by an increasing number of non-experts who use the aforementioned methods of configuration resulting in suboptimal configurations.

The problem of high dimensional data is well known and multiple solutions for example by van der Maaten et al. [2], Lueks et al. [3] and Lee and Verleysen [4], [5] have been proposed. Hyperparameter optimization is a major challenge when working with neural networks and was also subject of various papers such as the work of Diaz et al. [6], Feurer et al. [7], Mendoza et al. [8] and Loshchilov and Hutter [9] with different approaches to finding a solution. The choice of hyperparameters may lead to significant performance improvements in the resulting network and according to Bergstra et al. [10] can, in some cases, be even more useful than optimizing the machine learning algorithm or architecture that is used.

In this paper these problems will be addressed by combining a dimensionality reducing NN, called an autoencoder (AE), which was introduced by Rummelhart et al. in [11], with an evolutionary algorithm (EA) to optimize the hyperparameters of the AE. Using AEs as a means of preprocessing the input data and then feeding the reduced data into another NN can speed up the overall training process immensely as well as allowing the following network to work on a more abstract representation of the data since features are combined by the preprocessing. The focus in this paper is finding a low-dimensionsal representation of the input data rather than the possibilities of reconstructing it.

Further this preprocessing process will be optimized using neuroevolution. During training, the hyperparameters of the AE will be evolved by an EA. Since the initial configuration is chosen at random within a certain range and evolved during training, the evolutionary algorithm not only tries to find a good starting configuration but also an optimal path for each hyperparameter throughout the generations of training. The evolutionary process also aids the dimensionality reduction (DR) in that individuals who have a higher fitness according to a DR metric are chosen for reproduction rather than individuals with a better reconstruction error.

Due to the inherent structure of AEs, generally reducing a larger amount of inputs to a smaller amount of outputs in each layer, they can be trained layer by layer by using the outputs of the previous layer as inputs to the next layer. The neuroevolution process can therefore be applied to each layer of the AE and a maximum number of remaining output nodes can be defined as the stopping criteria, enhancing the flexibility of the approach.

The remainder of the paper is structured as follows: In Section 2 the methods of dimensionality reduction and hyperparameter optimization that are used in this paper are reviewed alongside other tools that represent the foundation of the neuroevolution process. Thereafter, the neuroevolution process that this paper proposes is presented in Section 3. In Section 4 experiments are conducted to analyse the stacked autoencoder neuroevolution and to evaluate its perfomance. Section 5 summarizes the paper.

2 Related Work

2.1 Dimensionality Reduction

The goal of DR is to find a meaningful low-dimensional representation of a high-dimensional data set. In a usual setting a high-dimensional data set $\Xi = \{\xi_1, \xi_2, ..., \xi_N\} \subset \mathbb{R}^K$ is transformed into a low-dimensional data set $X =$

$\{x_1, x_2, ..., x_N\} \subset \mathbb{R}^L$ with $L < K$. To visualize the result $L = 2$ or $L = 3$ can be used.

According to Lee and Verleysen [4] DR methods can be classified into linear methods such as principal component analysis (PCA) [12] and non-linear methods, for example NLDR [13] and AEs [11]. In this approach AEs are used.

In order to assess the quality of the DR it is tested if the structure of the data set is preserved. This can be achieved by analyzing similarities and neighborhood relationships by measuring the pairwise distances in Ξ and X. Though, instead of formulating a metric on the concrete pairwise distances, more recent criteria use the ranks of the distances. These criteria then analyze the K-ary neighborhood to measure the quality of the DR [4].

In this paper two rank-based criteria are used, namely, Q_{NX}, proposed by Lueks et al. [3] and the local continuity meta-criterion (LCMC) proposed by Chen and Buja [14]. Both of these criteria can be calculated using the co-ranking matrix proposed by Lee and Verleysen [4]. It is defined using the ranks of the pairwise distances from ξ_i to ξ_j denoted as ρ_{ij} in the high-dimensional space and x_i to x_j denoted as r_{ij} in the low-dimensional space. The co-ranking matrix $Q^{N-1 \times N-1}$ can then be defined as:

$$Q = [q_{ab}]_{1 \leq a,b \leq N-1},\ q_{ab} = |\{(i,j) : \rho_{ij} = a\ and\ r_{ij} = b\}|$$

To extract a single value as a measurement of the quality of the DR

$$Q_{NX}(K) = \frac{1}{KN} \sum_{a=1}^{K} \sum_{b=1}^{K} Q_{ab}$$

can be used. It counts the points that stay in the the K-ary neighborhood during the projection, meaning that the absolute difference $|\rho_{ij} - r_{ij}|$ of the ranks ρ_{ij} and r_{ij} of two corresponding distances $dist(\xi_i, \xi_j)$ and $dist(x_i, x_j)$ in the high- and low-dimensional space is less than K. This results in a value of 1 for a mapping where all points stay in the K-ary neighborhood. The

$$LCMC(K) = Q_{NX}(K) - \frac{K}{N-1}$$

differs from Q_{NX} only in a linear term that accounts for a random mapping [3].

2.2 Autoencoders

The idea of AEs was introduced by Rummelhart et al. [11] to solve an encoding problem where the input of the network had to be encoded with fewer nodes and then be reconstructed from that encoding. Essentially an AE is a multilayer feed-forward neural network with a low-dimensional central layer.

As Figure 1 demonstrates, it can also be seen as two separate networks: The encoder network that reduces the dimensionality of the input vector to the size of the central or code layer and the decoder which reconstructs the high-dimensional input vector from the encoding. Due to the non-linearities introduced by the activation functions between the layers the AEs dimensionality reduction is also non-linear [16]. An AE does not necessarily have to be symmetric around the encoding, though in this paper it will always be symmetrical.

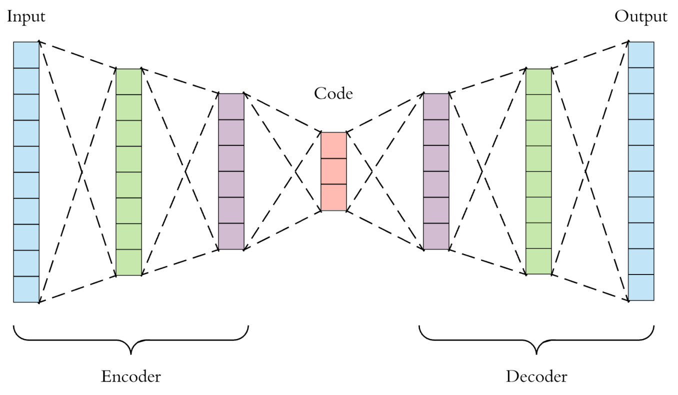

Fig. 1. Structure of an Autoencoder [15]

Hinton et al. showed in [16] that it is beneficial for deep autoencoders to pre-train each layer separately before optimizing the whole network, thus learning one layer of features at a time. They state that starting with large initial weights in the deep architecture results in only finding poor local minima and starting with small initial weights results in tiny gradients which makes it infeasible to train AEs with many hidden layers. The pre-training, to find good initial weights, is done by using a two layer network called a restricted Boltzmann machine for each layer and then stacking the layers on top of each other to construct the deep AE. The resulting AE can then be fine-tuned using gradient descent. This architecture yields better results in dimensionality reduction than principal component analysis according to the authors.

In this paper a similar approach is used, building the deep architecture layer by layer although the training algorithm differs from what is proposed in [16]. Section 3 will describe the method that is used in more detail.

2.3 Gradient Descent

Most often NNs are optimized using gradient descent based algorithms. Sebastian Ruder compared the most widely used gradient descent variations and evaluated their strengths and weaknesses in [17]. Only mini-Batch variations are discussed here, since other variants are not used in this paper. According to the author four challenges arise when using standard mini-batch gradient descent. First of all it is difficult to find a good learning rate (LR). This can lead to slow convergence or divergence during training. Secondly the appropriate LR usually changes during training so it has to be adapted or a schedule has to be used. When using a schedule it has to be determined before training starts and can therefore not adapt to the datasets characteristics. Furthermore not all parameters should be updated with the same LR. Infrequently occurring features should result in larger updates and the updates of frequently occurring features should be dampened. The last challenge the author describes is that while optimizing highly non-convex error functions neural nets often get trapped in a suboptimal local minimum.

These challenges were addressed in the RMSProp algorithm which was proposed by Hinton et al. in [18]. It calculates the LR for each parameter individually by using a decaying average $E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$ of the squared gradient g_t^2 of that parameter, weighed with a factor γ which usually is 0.9. This solves the problems of the first three proposed challenges. Additionally a momentum term is used to to avoid getting stuck in local minima. The final parameter update of a parameter θ at time t with a base LR of η is given by

$$\nabla\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t, \theta_{t+1} = \theta_t + \nabla\theta_t$$

In this paper it is used to train the AEs. Its ability to adapt the LR helps to optimize it in addition to the EA. This is especially useful since only rather small populations can be used due to training times. Other algorithms such as Adadelta [19] by Matthew Zeiler and Adam [20] by Kingma and Ba also solve the aforementioned challenges. Adadelta does not require a LR, which in turn means that the LR cannot be optimized by an EA. Adam adds a form of momentum to the parameter update which is done using the momentum proposed by Qian [21] in this paper. Further, initial experiments showed that RMSProp performs best on the given problem.

2.4 Dropout

Dropout is a regularization technique that prevents overfitting and also yields better training results due to model combination. It was proposed by Srivastava et. al in 2014 [22] and is especially well suited for deep architectures. Dropout hereby refers to temporarily disabling a node in a NN along with all its incoming and outgoing connections. Nodes are retained with a fixed probability p usually around 0.5 or if applied to the inputs of a network closer to 1. This means that each update is calculated on a different subset of the network. During testing, dropout is disabled resulting in a single unthinned network that effectively combines the networks from the training process. The retained nodes are scaled by $\frac{1}{1-p}$ during training so that the sum of outputs remains the same while training and testing.

Dropout is used in the SAENE method to reduce overfitting and the retention rates are optimized in the EA so that they are adapted throughout the training process.

2.5 Evolutionary Algorithms

Evolutionary algorithms are optimization algorithms which use biologically inspired mechanisms such as mutation, crossover, and fitness based selection to iteratively refine a population of solutions. An advantage of EAs is that they are black-box optimizers, meaning that very few assumptions are made about the underlying objective function. Additionally little insight to the structure of the problem space is required resulting in good solutions even if manual construction of a heuristic is infeasible.

A solution is defined by a genotype which stores all relevant information to apply it to the optimization problem by mapping it to its phenotype (genotype-phenotype mapping). The fitness of the phenotype can then be calculated using a fitness function. With this fitness value the solutions from which the next generation of candidates is generated can be selected [23].

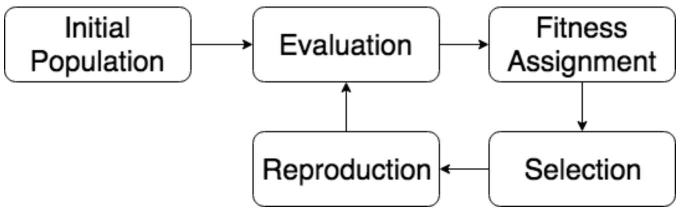

Fig. 2. Basic lifecycle of an evolutionary algorithm [23]

The basic lifecycle of an EA is illustrated in Figure 2. First an initial population of genotypes is generated randomly from the set of possible of genomes. Thereafter the EA enters a cycle which continues until a stopping criterion, for example a certain number of generations or a desired fitness is reached. This cycle consists of four stages, namely evaluation, fitness assignment, selection, and reproduction. In the evaluation stage, the genotype is mapped to the phenotype and applied to the problem at hand (objective function). Each individual is then assigned a fitness value based on the objective value. With all fitness values assigned a set of solutions is selected from the population that will be used to generate new offspring solutions. Individuals that have not been selected are discarded and the remaining individuals are recombined and mutated to fill up the population to its original size. The cycle then begins anew until the stopping criterion is met [23].

EAs have been used in many areas of optimization including hyperparameter optimization for example by Friedrichs and Igel [24] as well as Bochinski et al. [25] achieving state of the art performance. Bochinski et al. optimize the hyperparameters of a convolutional NN on a classification task whereas in this paper the parameters of a non-convolutional AE are optimized. Further they focus on optimizing the configuration of the convolutions and the layer sizes of the fully-connected layers and do not optimize the LR or dropout rates of the network. In Friedrichs and Igels work the hyperparameters of support vector machines are optimized. In [26] EAs are used to train the weights of a NN replacing other training algorithms such as gradient descent.

An approach similar to [25] is taken in this paper, although it is applied to a different kind of neural network and other hyperparameters are chosen for optimization.

3 Stacked Autoencoder Neuroevolution

The stacked autoencoder neuroevolution (SAENE) is a method that combines an AE with an EA to optimize the DR. This section is therefore split into two subsections describing the EA and the AE architecture respectively. The SAENE algorithm is visualized in Figure 3.

3.1 Evolutionary Algorithm

The EAs goal is to optimize the configuration of the NN. The genotype consists of six hyperparameters that are used during the training of the NN. They are depicted in Table 1. When the EA is initialized a population of NN configurations is selected randomly. The range of the initialization varies between experiments and will be detailed in Section 4.

```
1:  $pop \leftarrow init(pop\_size)$
2:  while $layer\_size < target\_size$ do
3:      $layer\_size \leftarrow layer\_size \cdot layer\_ratio$
4:      for all aes in pop do
5:          $ae.append\_layer(layer\_size)$
6:      end for
7:      for gens_per_layer do
8:          for aes in pop do
9:              $ae.restore\_layers()$
10:             $loss \leftarrow MSE(input, ae.fwd\_pass(input))$
11:             $ae.set\_params(dropout\_rates, activations)$
12:             $opt \leftarrow RMSProp(ae.lr, ae.momentum, loss)$
13:             for num_training_steps do
14:                 $batch \leftarrow get\_batch(batch\_size)$
15:                 $update\_weights(opt, batch, ae)$
16:             end for
17:         end for
18:         $evalute\_and\_sort\_by\_fitness(population)$
19:         $pop \leftarrow pop.take(pop\_size \cdot selection\_ratio)$
20:         $save\_histories()$
21:         $refill\_and\_mutate(pop)$
22:     end for
23: end while
```

Fig. 3. SAENE Algorithm

TABLE 1
Genome of the EA / Hyperparameters of the AE

Parameter
Learning Rate
Momentum
Training steps per generation
Batch size
Activation functions
Dropout rates

TABLE 2
Hyperparameters of the EA

Parameter
Population size
Generations per Layer
Selection Ratio
Layer Ratio
Layer Size
Target Size

In each generation of the EA, every genome of the population is mapped to an AE (genotype-phenotype mapping) and a training session will be run. Lines 7 to 22 in Figure 3 depict the generation loop. The weights of the AEs are always restored from the training session of the previous generation. After the training session the fitness of each individual is evaluated. Multiple options were be considered as fitness functions. The most intuitive fitness function is the reconstruction error of the AE after training. As the AE tries to reproduce its input Y_i at its output $\hat{Y}_i$ from the latent space, a metric such as the Mean-Squared-Error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

could be used for the fitness evaluation. Another option more specific to the problem at hand is to use a DR metric such as the LCMC as described in Subsection 2.1. Such a fitness function can directly select individuals with better DR results instead of indirectly through the reconstruction error. The LCMC cannot be used as a loss function during the training process with gradient descent. Therefore it increases the influence of the selection process of the EA and represents an advantage in comparison to standard training without the optimization of an EA. Both options are compared in the experiments in Section 4. The fitness is always evaluated on a separate evaluation set and not on the training data to prevent overfitting.

To control the evolution the EA has to be configured. Table 2 shows the hyperparameters of the EA. These hyperparameters are not evolved and stay the same throughout the run of the EA. It can be seen that most hyperparameters are specific to optimizing a stacked AE architecture.

After the fitness of each individual has been evaluated, the selection ratio hyperparameter of the EA config is used as the retention ratio of the population (see line 19 in Figure 3). The population is split according to this ratio and the more unfit individuals are discarded. The population is then refilled by producing offspring from the remaining individuals in a round-robin fashion, starting with the fittest individual. This means that the fittest individual has a higher chance of producing offspring since it is used first and the process stops once the population reaches its original size. Because populations tend to be rather small in this case (in the range of 10 individuals) this can be a significant advantage for the fittest individuals.

Offspring are created from only one parent using only mutation and no crossover. Crossover is not used to simplify the evolutionary process and as Osaba et. al state in [27] mutation is generally the driving force in the evolutionary process. Experiments also show that optimal hyperparameter values change from generation to generation, requiring the individuals to adapt rather than to settle into a final optimal solution which would be aided by crossover. The EA still has to be able to preserve successful hyperparameters. Therefore retained individuals are not mutated and since offspring are created through mutation they always count as mutated. This implicitly defines the mutation rate by the selection ratio and population size and explains why no explicit mutation rate is used. The strength of the mutation is set individually for each hyperparameter that is mutated by changing the variance of the gaussian mutation which is applied. Offspring inherit the current weights of the parents and continue training with mutated hyperparameters in the next generation. The mutation targets the LR, momentum and dropout rates of the AE. Activation functions are not mutated because a change of the activation function would have to be compensated by large changes in the weights and thereby undoing the optimization done in the previous generation. The batch size and training steps are also not mutated. They can be seen as more general hyperparameters to control the amount of time each training session takes and modifying them would also lead to problems when comparing the results of different experiments.

Although the activation functions are not mutated, a random activation function is chosen each time a layer is appended to the AE. The hyperparameter generations per layer specifies how many generations the EA runs before adding another layer to the AE. A random activation function and random dropout rates for the new layer are added to the genome and the EA enters its generation loop again. The size of the new layer is calculated using the layer ratio hyperparameter which is multiplied with the current layer size to get the size of the new layer (see line 3 in Figure 3). The EA terminates when the currently smallest layer is smaller or equal to the size specified by the target size hyperparameter when attempting to add a new layer. The target size represents the maximum size of the final layer and not necessarily its exact size.

Through the evolutionary process the structure of the AE is generated by continuously adding layers to it, creating a stacked AE architecture. The inner workings of this architecture are discussed in the following subsection.

3.2 Autoencoder

The stacked AE architecture used here is implemented in Python using the Tensorflow framework [28], which was originally developed by the Google Brain team and is now continued as an open source project on Github.

AEs in this paper are configured using the hyperparameters shown in Table 1 and created from a list of layer sizes from which the encoder and decoder layers are initialized. Usually they are initialized with one encoder and one decoder layer. Layers are always added in the middle of the AE, meaning that the size of the latent space changes when a layer is added. Two new sets of weights, one for the encoder to the new latent space and one for the decoder from the new latent space are added. Further, new dropout rates and activation functions are appended to the configuration.

The AE also provides the ability to freeze layers, fixing the weights that correspond to them. This technique can be used to speed up the training process since fewer gradients have to be calculated and less weights updated. Freezing is not used in this paper though because preliminary experiments showed a loss of accuracy and minimal speed up. Another feature is the ability to apply an activation function on the output of the en- and decoder. Depending on the data set and the preprocessing of the data it can be beneficial to apply an activation function on the output, for example if the target values range from 0 to 1. This is investigated in the experiments in Section 4. Dropout is used on every layer during training, including the input, mainly to reduce overfitting. When the fitness of the AEs is evaluated dropout is deactivated to get the best accuracy.

To train the weights of the AE RMSProp with momentum is used. As the loss function the MSE of the original input and the reconstructed output is used. Prior to the training session the weights from previous sessions are restored if they are available. Lines 8 to 17 in Figure 3 show the training loop for the AEs. The configuration used for the session is saved to be able to analyse the lineage of the hyperparameters. Using this architecture layers are not trained totally independent. However, the architecture starts with only one layer, learning to extract features from the current latent space. After a layer is appended, the previous layer is not frozen, as is done in other stacked architectures. Instead, it is allowed to adapt to the new layer. Using dropout might also be beneficial here since the EA can optimize the dropout rates in such a way that a high LR which is needed for the new layer does not disrupt the features that were already learned in the previous layer. Multiple training sessions are run for each layer with configurations that did well in the last session and mutated new ones allowing the hyperparameters to adapt as opposed to a static baseline that uses the same hyperparameters throughout training. This results in a dynamic architecture that not only searches for the best initial configuration but rather for the best lineage of configurations.

4 Experiments

The experiments compare the SAENE architecture on three datasets. Most experiments use four layers, halving the input twice in the encoder and decoder. Further, experiments with reductions to 10 and 2 output nodes on MNIST are conducted. The experiments on MNIST are compared to experiments with the same layer ratio on the Year Prediction MSD dataset. Then a higher layer ratio is used to create a network with 6 instead of 4 layers, reducing to the same number of output nodes to see if this improves the performance. Lastly it is tested if the use of MSE instead of LCMC in the fitness evaluation of the EA decreases the perfomance. These results are compared to baseline experiments on the datasets that were trained without the EA.

4.1 Datasets

To conduct the experiments in this section, three datasets are used: MNIST, Fashion-MNIST and the Year Prediction MSD dataset. On each of the datasets 5000 samples are reserved for the validation set.

The MNIST dataset consists of a training set of 60,000 examples and a test set of 10,000 examples, representing hand-written digits [29]. Each sample consists of 28x28, so 784, gray-scale features ranging from 0 to 255. Additionally every sample is labeled with the digit that it represents. In this paper the values are normalized to a range from 0 to 1.

Fashion-MNIST is designed as a more difficult drop-in replacement for MNIST [30]. It has the same number and split of examples as well as the same shape. The samples are Zalando's article images and labeled with the article class that they represent. The same pre-processing as for MNIST is used.

The third dataset used in this paper is the Year Prediction Million Song Database (YPMSD) dataset [31]. There are 515,345 total samples with 463,715 training samples and 51,630 test samples. Each sample contains 90 real-valued features derived from the timbre information of the song. No pre-processing is done on this datasset.

4.2 Parameter Lineage

The parameters of the AEs are optimized and change during the evolutionary process. These lineages of the parameters are analyzed in this subsection. Tables 3 and 4 show the base configuration and mutation rates that are used throughout the experiments. Deviations from these configurations are mentioned when a specific experiment is discussed.

4.2.1 Learning Rate

First, the lineage of the LR is analyzed. In general the learning rate decreases quickly from its initial value and in most cases does not reach a value as high as the initial value again. Minor spikes occur but the downward trend is rarely broken. When a new layer is appended to an AE, the LR increases significantly in most cases, resulting in a spike after generation 10 for a four layer model (two encoder and

TABLE 3
Base Configuration and Mutation of the AE

Parameter	Initial Value	Mutation
Learning Rate	$\mathcal{N}(0.05, 0.05)$	$\mathcal{N}(0.00, 0.05)$
Momentum	$\mathcal{N}(0.1, 0.05)$	$\mathcal{N}(0.00, 0.05)$
Steps per generation	5000	
Batch size	256	
Activation functions	relu, tanh, sigmoid	
Dropout rates	$\mathcal{N}(0.1, 0.05)$	$\mathcal{N}(0.00, 0.05)$

TABLE 4
Base Configuration of the EA

Parameter	Value
Population size	10
Generations per Layer	10
Selection Ratio	0.5
Layer Ratio	0.5
Target Size	196
Fitness function	LCMC (N=100, K=50)

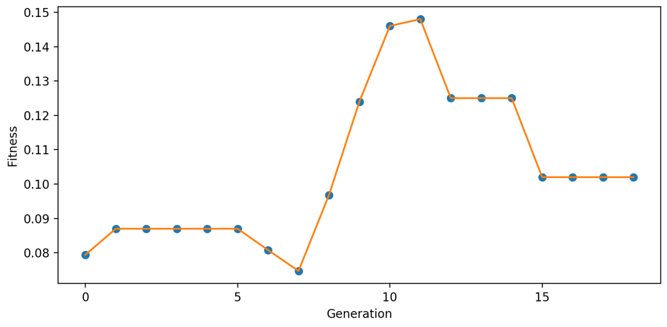

Fig. 4. LR lineage on MNIST reduction to 196 nodes

two decoder layers). After this spike the downward trend is resumed. Figure 4 shows the lineage of the LR throughout the evolutionary process. The median of the LR on the reduction to 196 nodes on MNIST is at approximately 0.047 in generation 1 then decreases to 0.00847 in generation 10 and jumps to 0.0115 in generation 11. In generation 20 the median dropped to 0.00055. These values are representative for all experiments that were conducted differing slightly in quantity but holding true in quality. The only exception is the reduction to 2 nodes on MNIST experiment. This experiment showed no dominant trend in the LR at all which can be contributed to the fact that the results were very bad throughout the whole process and therefore no good selection or optimization could be achieved. This will be discussed further in Subsection 4.3.

In many cases the LR reaches its minimum before the last generation of the layer is reached as can be seen in Figure 5. The minimum is set to 0.00001 in most experiments and used to prevent negative or zero values during mutation. The mutation rate for the LR is set to a relatively large value. This helps to increase the influence of the EA since the LR is adapted during training by RMSProp and the EA only provides the initial value. The high mutation rate contributes to the reason why the minimum is reached

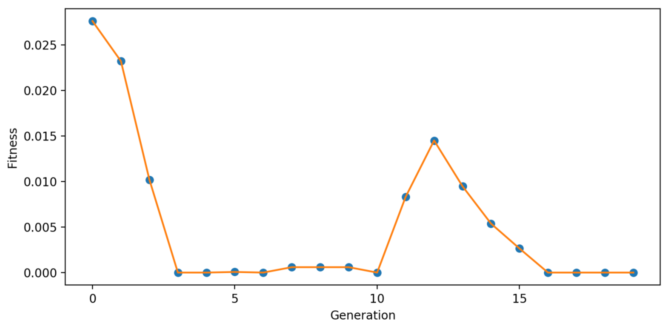

Fig. 5. Median LR on YPMSD reduction to 22 nodes

frequently even if a slightly higher LR may yield a better result. Although when the mutation strength is decreased the minimum is still reached often, meaning that small LR is indeed beneficial. A high mutation rate allows for better adaption for example when adding a layer. In some cases the LR starts to spike a generation or two before a layer is appended. The reason for this is that the AE might not be the top performer in the generation before the layer is appended because its LR is too high but it is good enough to be kept in the population. In the next generation, when a layer is appended, its higher LR then causes it to outcompete the other individuals. This can also be seen in Figure 4.

4.2.2 *Momentum*

The lineage of the momentum shows fewer patterns. In some experiments it shows a similar pattern to the LR but with much smaller fluctuations. In the experiments with the Year Prediction MSD (YPMSD) dataset it showed an upward trend. There, in generation 1 the median was always close to 0.1 and continued rising to approximately 0.2 in generation 20. In most other cases the momentum fluctuated around its initial value, indicating a random development. Hinton also stated that momentum does not help RMSProp as much as is usually does [18]. He also states that the learning rate should be increased toward the end of the training to help the gradient descent algorithm to navigate ravines. His suggested values are much higher (0.5 - 0.99) than what was used in this paper though. In the case of the SAENE architecture momentum does not seem to play a major role. Either it is neglected by the EA to optimize more important parameters such as the LR or its effect is too small to have a significant effect on the results.

4.2.3 *Activation Functions*

The activation functions have a more significant impact on the performance and patterns can be seen clearly. For both the standard and fashion MNIST data sets the sigmoid activation is dominant with a usage rate of approximately 70% across all layers. It is even more dominant when only the first and last layers are considered. In the fist layer it used in 75% of the cases and in the last layer in 98.33% of the cases. The dominance in the last layer can be explained by the range of values in the data set. All values are in the range from 0 to 1 which coincides with the range of the sigmoid activation. Further, most pixels are either black or white. This lets the network optimize the few pixels that are in the medium range, since even if other outputs would get very large or very small they are then correctly mapped to 1 or 0 respectively. This is also the reason why a linear output function performed worse on MNIST.

The experiments with the YPMSD data set showed dominance of the ReLU activation. It was used in approximately 90% of the cases. The output of the YPMSD networks was fixed to a linear activation because the data was not normalized, so the output layer cannot be considered here. The dominance can be explained by the range of the input values not being constrained to $(0, 1)$ and having a large difference in magnitude from around 0 to multiple thousands. It can also be seen that when the output of the encoder is also forced to use a linear activation, the results are more inconsistent. A clear correlation between the performance and the use of ReLU can be seen but the usage rate of ReLU in the first layer of the encoder is only 30%.

This is an inherent problem of the SAENE architecture. Since layers are always appended in the middle of the network, between the encoder and the decoder and activation functions are chosen when a layer is appended and then inherited but not mutated, the activations of the encoder cannot be optimized when a linear output is used there. Starting from a network with one encoder layer and one decoder layer, they both will be assigned an activation function but they will not be used since a linear output is used. After appending a layer, the old decoder layer is still connected to the output of the network, hence still using the linear activation. The old encoder layer though, now maps from the input to a hidden layer and the hidden layer maps to the output of the encoder. This means that the activation function of the old encoder layer is now used but was not optimized in the previous layer, hence no selection of the best activation function could take place. This leads to more inconsistent results as the YPMSD experiment has shown.

Experiment runs where the activation function was ReLU by chance performed better than the other options. See Subsection 4.3 for more details on the results.

4.2.4 *Dropout Rates*

The dropout rates do not change significantly throughout a training session. In most cases they fluctuate around their initial value. There is no common pattern to the lineage of the dropout rates. Since they are initialized from a gaussian distribution centered at 0.1 with a variance of 0.05 they usually fluctuate in ranges from $3 - 17\%$ as would be expected by a random development. The conclusion is similar to the momentum. Either the dropout has no significant effect in the experiments or its effect is too small and therefore neglected by the EA to optimize more important parameters. Another reason could be that the initial value is too small though one would expect that in some cases through mutation it would then drastically increase and show a consistent upward trend. This behavior could not be seen in the experiments.

4.2.5 *Fitness*

The lineage of the LCMC fitness depicts the inverse of the lineage of the learning rate. A consistent upward trend can be seen throughout the training of a layer. When a new layer is appended, the fitness plummets due to the new randomly

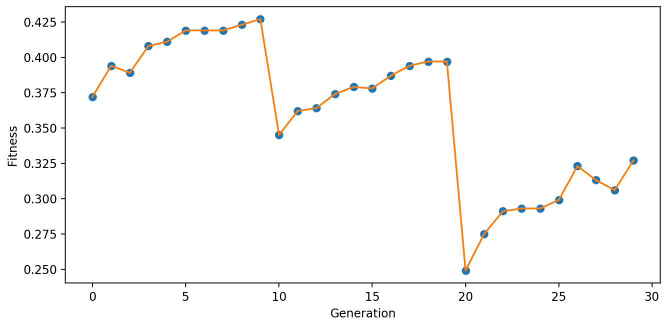

Fig. 6. Median LCMC Fitness on MNIST reduction to 10 nodes

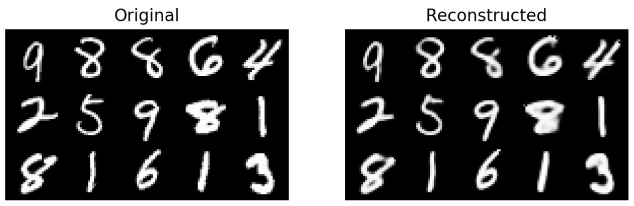

Fig. 7. Reconstruction of MNIST test set images from 196 nodes

initialized weights that are added for the new layer. It then resumes its upward trend but usually does not reach the peak of the previous layer, since the DR problem is more difficult because fewer nodes are available for the encoding. The resulting stair case shape of the lineage can be seen in Figure 6. If the MSE is used as fitness, the values have to be negated to match the patterns of the LCMC fitness because the MSE is an error metric, meaning that small values are better.

4.3 Test Set Results

To compare the results the LCMC fitness with $N = 100$ and $K = 50$ was calculated using the trained network and the test set for each experiment even if the fitness used during training was MSE. Also the focus of this paper is the quality of the dimensionality reduction not the quality of the reconstruction, although the two correlate as the experiments showed. Table 5 shows an overview of the results.

First the experiments that reduced the 784 inputs of the MNIST data sets to 196 outputs using two layers for the encoder and two layers for the decoder are examined. The parameters can be seen in Tables 3 and 4. The average LCMC score on MNIST using LCMC fitness on the test set is only slightly less than the average training score. An example of digits reconstructed with the best network can be seen in Figure 7. Interestingly the average score on the test set was higher than on the training set when MSE was used. Since this was not the case when using the LCMC during training, it could mean that using LCMC fitness overfits the networks to the validation set.

The best performing network overall was trained using MSE. The difference in the best performing networks is not significant though since the LCMC score is calculated using

TABLE 5
Test-Set LCMC scores of the experiments

Dataset	#Layers	#Outputs	Fitness F.	Avg	Std. Dev.	Min	Max
MNIST	4	196	LCMC	**0.3940**	**0.0221**	**0.3537**	0.4273
			MSE	0.3875	0.0327	0.3169	**0.4297**
			Baseline	0.1108	0.1385	0.0041	0.3761
		10	MSE	0.2852	0.0292	0.2337	0.3249
			LCMC	0.2804	0.0424	0.1809	0.3145
	6		LCMC	0.3133	0.0256	0.2505	0.3385
	4	2	LCMC	0.1768	0.0224	0.1449	0.2105
Fashion-MNIST	4	196	LCMC	**0.3858**	**0.0193**	**0.3441**	**0.4097**
			Baseline	0.0640	0.0650	0.0158	0.2153
		10	LCMC	0.3721	0.0285	0.3169	0.4065
YPMSD	4	22	LCMC	**0.3088**	0.1147	0.0745	0.4049
	(Lin. Enc)		LCMC	0.2814	0.1244	**0.1049**	**0.4554**
			Baseline	0.0482	**0.0331**	0.0025	0.0921

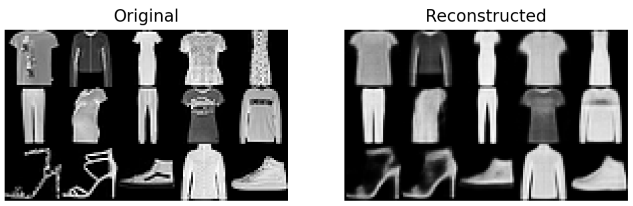

Fig. 8. Reconstruction of Fashion-MNIST test set images from 200 nodes

random samples of the test set and some variance naturally arises.

The experiment shows that using the MSE as fitness function results in a similar quality of the DR even though the EA does not specifically optimize for it. This is also true when optimizing to a smaller latent space with only 10 outputs. This leads to the conclusion that optimizing to minimize the reconstruction error implicitly produces a good encoding in terms of the quality of the DR. Or in order to perform a good reconstruction a good low-level representation is needed. Also the networks are trained for 5000 training steps per generation in the AE using the MSE as the error function of the optimization and only after each generation the LCMC is used as fitness in the EA to select individuals. This means that the majority of optimizing is still done using the MSE especially considering the relatively small population size of 10 individuals.

Figure 8 shows an example of images reconstructed using the best Fashion-MNIST network with 196 outputs. Some details of the images are lost in the reconstruction but the different classes are still clearly distinguishable. The LCMC scores are lower than for the standard MNIST data set due to the higher complexity of the images.

With only 10 output nodes the highest score is significantly lower than the worst performing network using 196 outputs for MNIST but very similar for Fashion-MNIST. Reconstructions can be seen in in Figure 9 and Figure 10. This shows that Fashion-MNIST is much more resilient in terms of the number of nodes needed for the reconstruction.

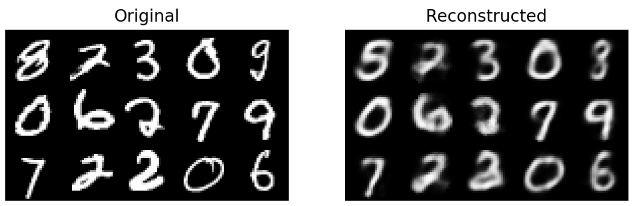

Fig. 9. Reconstruction of MNIST test set images from 10 nodes

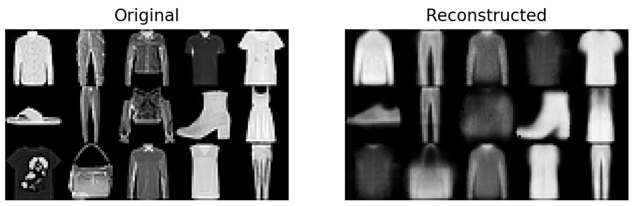

Fig. 10. Reconstruction of Fashion-MNIST test set images from 10 nodes

Decreasing the size of the layers more slowly, using a higher layer-ratio, resulting in a six layer network the performance increased. Figure 11 shows reconstructed images with a six layer network. This improvement in performance caused the training time to increase from an average of 6605 seconds to 21503 seconds, meaning that the training time increased by a factor of about 3.25.

With 2 output nodes available the quality of the reconstruction and DR decreased and artifacts appeared in the images. Also Similar digits such as 7, 4 and 9 were reconstructed to a 9. The digit 7 was also reconstructed to a 1 in many cases as can be seen in Figure 12.

In the YPMSD experiments all the networks had a linear output on the decoder. The variance in these experiments was higher than in the MNIST experiments. With a linear output on the encoder the best results outperformed the non-linear output on the encoder but the average was significantly lower. This leads to the conclusion that a linear

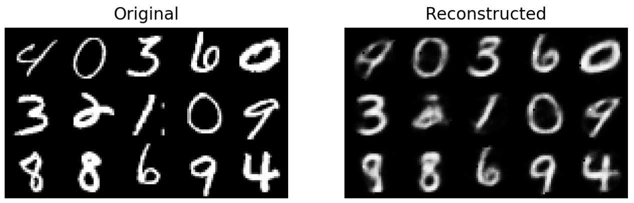

Fig. 11. Reconstruction of MNIST test set images from 10 nodes with six layers

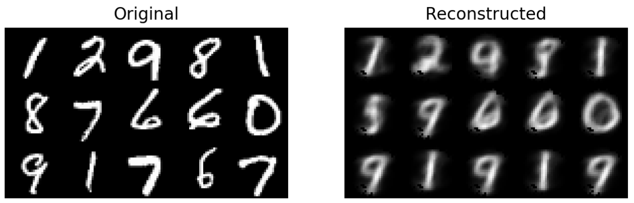

Fig. 12. Reconstruction of MNIST test set images from 2 nodes

output for the encoder is beneficial in case of the YPMSD data set, but due to problems with that approach described in Subsection 4.2 the results are inconsistent.

4.4 Baseline Comparison

For the baseline comparison the three data sets were trained without the EA. This means that all the parameters were initialized from their range when the AE was created and not changed during training. The network is also trained layer-wise with the same number of training steps as the EA would have performed in its generations. Also without the EA there is no population so the baseline experiments trained one network at a time. The input size was halved twice meaning on the MNIST data sets it was reduced from 784 to 196 and on the YPMSD data set from 90 to 22. Each resulting network consisted of 4 layers.

The average LCMC score of the baseline networks is much lower than when using the SAENE architecture. The results are very inconsistent and depend heavily on the chance that the initial parameters are good.

For the YPMSD data set the baseline had the lowest average fitness values. This can be explained by the linear output that is used on the decoder. On the MNIST data sets the output can be mapped to the range from 0 to 1 by the sigmoid activation function so a random mapping would be much closer to the original than when the range is larger.

5 Conclusion

This paper introduced the SAENE neuroevolution method and shows that using it to train stacked autoencoders results in significantly better quality of the DR compared to the baseline without an EA. It automatically finds good starting parameters and parameter lineages throughout the training process and has shown to be successful on multiple datasets.

The learning rate and activation functions have proven to have a higher impact on the performance than the momentum and dropout rates. Further, using the LCMC as the fitness function in the EA did not show much better results than using the MSE of the reconstructed data.

Newtorks performed better on MNIST with a large latent layer, but were surpassed by Fashion-MNIST with a small latent layer, meaning that Fashion-MNIST proved to be more resilient in that regard.

In future work the SAENE method could be extended to other AE variants such as variational AEs. The EA could be refined to make its hyperparameters such as the number of training steps more dynamic to avoid overfitting and speed up the training process. It could also be tested if the approach benefits from crossover. Further, experiments that use the SAENE method as preprocessing for their datasets could be compared to the same experiments without the SAENE preprocessing.

References

[1] R. Bellman, R. Corporation, and K. M. R. Collection, *Dynamic Programming*, ser. Rand Corporation research study. Princeton University Press, 1957. [Online]. Available: https://books.google.it/books?id=wdtoPwAACAAJ

[2] L. van der Maaten, E. Postma, and H. Herik, "Dimensionality reduction: A comparative review," vol. 10, 01 2007.

[3] W. Lueks, B. Mokbel, M. Biehl, and B. Hammer, "How to evaluate dimensionality reduction? - improving the co-ranking matrix," *CoRR*, vol. abs/1110.3917, 2011. [Online]. Available: http://arxiv.org/abs/1110.3917

[4] J. A. Lee and M. Verleysen, "Rank-based quality assessment of nonlinear dimensionality reduction," in *ESANN*, 2008.

[5] J. Lee and M. Verleysen, "Quality assessment of nonlinear dimensionality reduction based on k-ary neighborhoods," in *Workshop at ECML/PKDD 2008*, Y. Saeys, H. Liu, I. Inza, L. Wehenkel, and Y. V. de Pee, Eds., vol. 4. Antwerp, Belgium: PMLR, 15 Sep 2008, pp. 21–35. [Online]. Available: http://proceedings.mlr.press/v4/lee08a.html

[6] G. I. Diaz, A. Fokoue, G. Nannicini, and H. Samulowitz, "An effective algorithm for hyperparameter optimization of neural networks," *CoRR*, vol. abs/1705.08520, 2017. [Online]. Available: http://arxiv.org/abs/1705.08520

[7] M. Feurer, J. T. Springenberg, and F. Hutter, "Initializing bayesian hyperparameter optimization via meta-learning," in *AAAI Conference on Artificial Intelligence*. AAAI Press, 2015, pp. 1128–1135. [Online]. Available: http://dl.acm.org/citation.cfm?id=2887007.2887164

[8] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter, "Towards automatically-tuned neural networks," in *Workshop on Automatic Machine Learning*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., vol. 64. New York, New York, USA: PMLR, 24 Jun 2016, pp. 58–65. [Online]. Available: http://proceedings.mlr.press/v64/mendoza_towards_2016.html

[9] I. Loshchilov and F. Hutter, "CMA-ES for hyperparameter optimization of deep neural networks," *CoRR*, vol. abs/1604.07269, 2016. [Online]. Available: http://arxiv.org/abs/1604.07269

[10] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=104279.104293

[12] I. Jolliffe and J. Cadima, "Principal component analysis: A review and recent developments," vol. 374, p. 20150202, 04 2016.

[13] J. A. Lee and M. Verleysen, *Nonlinear Dimensionality Reduction*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[14] L. Chen and A. Buja, "Local multidimensional scaling for nonlinear dimension reduction, graph drawing, and proximity analysis," vol. 104, pp. 209–219, 03 2009.

[15] A. Dertat, "Applied deep learning - part 3: Autoencoders," 2017, accessed: 2018-08-13. [Online]. Available: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[16] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006. [Online]. Available: http://science.sciencemag.org/content/313/5786/504

[17] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[18] G. Hinton, N. Srivasta, and K. Swersky, "Lecture 6e: rmsprop," accessed: 2018-08-14. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[19] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012. [Online]. Available: http://arxiv.org/abs/1212.5701

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[21] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145 – 151, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608098001166

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[23] T. Weise, *Global Optimization Algorithms - Theory and Application*, 2nd ed. Self-Published, 2009, online available at http://www.it-weise.de/. [Online]. Available: http://www.it-weise.de/

[24] F. Friedrichs and C. Igel, "Evolutionary tuning of multiple svm parameters," *Neurocomputing*, vol. 64, pp. 107 – 117, 2005, trends in Neurocomputing: 12th European Symposium on Artificial Neural Networks 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231204005223

[25] E. Bochinski, T. Senst, and T. Sikora, "Hyper-parameter optimization for convolutional neural network committees based on evolutionary algorithms," in *2017 IEEE International Conference on Image Processing (ICIP)*, Sept 2017, pp. 3924–3928.

[26] H. Okada, "Neuroevolution of autoencoders by genetic algorithm." Copenhagen, Denmark: International Journal of Science and Engineering Investigations, 2017, pp. 127–131. [Online]. Available: http://www.ijsei.com/papers/ijsei-66517-21.pdf

[27] E. Osaba, R. Carballedo, F. Diaz, E. Onieva, I. de la Iglesia, and A. Perallos, "Crossover versus mutation: A comparative analysis of the evolutionary strategy of genetic algorithms applied to combinatorial optimization problems," *Scientific World Journal*, 2014. [Online]. Available: https://www.hindawi.com/journals/tswj/2014/154676/

[28] "Tensorflow," accessed: 2018-08-22. [Online]. Available: https://www.tensorflow.org/

[29] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[30] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[31] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml