Project 2 Report

Yuan Li (624329), Hao Wang (654727)

October 27, 2014

1 Stage One

In this section, we describe our approach and discuss several tricks we use to accelerate the training process.

1.1 Approach Outline

In our implementation, we use hyperbolic tangent $\tanh(x)$ as activation function in hidden units, softmax function in output units, and cross entropy as loss function. We use stochastic gradient descent to optimize the loss function and use backpropagation to calculate the gradient. We initialize

During the training process, we set checkpoint after every $20k$ training instances are used. At every checkpoint, we calculate the accuracy on training set and record it. If an accuracy larger that all previous ones is achieved, we will record the weights matrices together with the accuracy. When the number of iteration reach a fixed number (e.g. $60k$, per iteration 10 instances are used, in total $600k$ are used), the training function returns the highest accuracy together with weights matrices. Because the cost of computing accuracy on the whole dataset is considerable, so we choose the number $20k$.

If we are provided with a training set and a validation set, then accuracy on the validation set will also be calculated and recorded at every checkpoint. However, we will not make use of accuracies on validation set during the training process.

This approach has been implemented as a function in `stage1.py` called `train_nn` whose inputs are structure of neural network, training set and validation set (can be empty).

1.2 Widely Used Tricks

We have implemented several widely used tricks such as normalizing the inputs, weight-decay, mini-batch, and momentum.

Normalizing the inputs Comparing with the outputs of activation function (in $[-1, 1]$) which are the input of subsequent layer, the training instances are relatively large, so the weights connect input layer and first hidden layer are relatively small. However, since the gradient of a weight is proportional to the input value it connects, the weights connect input layer and first hidden layer have relatively large gradients. Relatively small value with relatively large gradient will make the training process unstable, therefore we normalize $\vec{x} = (x_1, x_2, \ldots, x_{25})$ to $\vec{x'} = \dfrac{\vec{x}}{\max_i |x_i|}$, so that $x'_i \in [-1, 1]$.

Weight-decay Weight-decay is to add a penalty term $\gamma \sum w_i^2$ to the objective function. Adding this term can improve generalization to new data by reducing overfitting to training data. After adding this term and set γ to be 0.00001, the sum square of weights will still grow but much slower.

Mini-batch Stochastic gradient descent is more computational efficient, however, batch provides more reliable gradient estimate. As a compromise, we use mini-batch which randomly takes 1 instances from each class (10 in total) to calculate average gradient and update weights at every iteration.

Momentum When updating the weight matrices, steepest decent on the gradient of loss function is the simplest approach, however it also brings fluctuations that could slows the process of convergence. Adding momentum helps solve the problem. It simply adds a fraction m of the previous weights update to the current one, used to prevent system from converging to a local minimum, in the mean while increases the speed of convergence.

1.3 Fixed versus Adaptive Learning Rate

We propose a simple mechanism to make the learning rate adaptive to the training process. When we arrive at a new checkpoint, we will compare the new accuracy with the previous one, if the new accuracy is larger than the previous one, then keep going, otherwise divide the learning rate by 2 and "rollback" to the previous checkpoint (restore all the weights). Comparing with fixed learning rate, this mechanism is more greedily that once a local maximam is met it will reduce the step size so as to explore the local region more carefully. Fig. 1 shows the different behaviors of these two mechanisms after a decrease of accuracy is met.

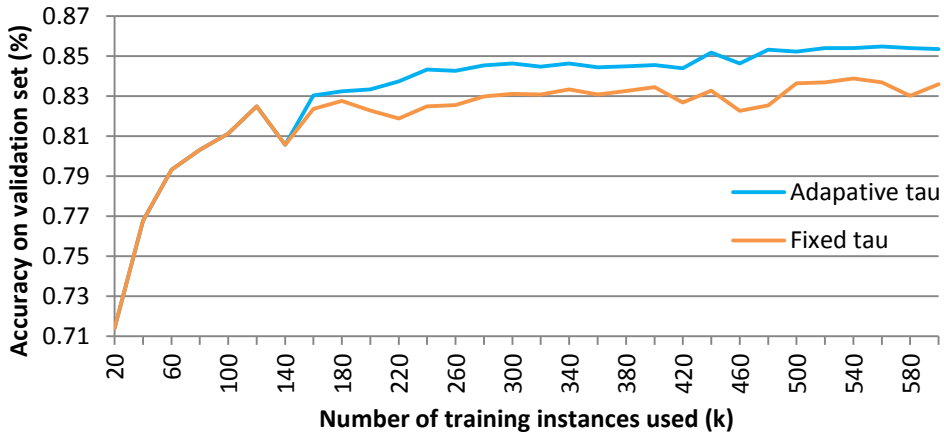

Figure 1: Performances of fixed and adaptive learning rates.

2 Stage two

2.1 Approach Outline

We adopt the same approach described in assignment specification. Specifically, we (1) first divide the whole training set into sub-training set (90%) and validation set (10%), (2) then run genetic algorithm and obtain the structure with highest fitness, (3) finally retrain the identified structure on the whole training set and make prediction for the test set. This approach has been implemented in the main function of `stage2.py`. Next, we describe our genetic algorithm in the following aspects:

Representation The structure of network is encoded in a list of numbers describing how many units in layers except input layer and output layer, e.g. a list $[57, 40]$ represents 4-layer structure 25-57-40-10.

Fitness function We define fitness of a network as the accuracy it achieves on a validation set after training. To calculate the fitness, we run the training function on the sub-training set to optimize the weights matrices, and finally calculate the accuracy on the validation set as fitness.

Population Since the calculation of fitness is very expensive (the cost depends on the number of layers and hidden units, typically half an hour), we choose the population size to be a very small number, 5 in our implementation. At the beginning of GA, we initialize these five individuals with random number of layers and random number of nodes for each layer.

Crossover When two individuals are selected, we align them to the left, then randomly choose a crossover position in the shorter one, finally cross over at that position and generate two offsprings, e.g. two individuals $[a_1, a_2, a_3]$ and $[b_1, b_2, b_3, b_4, b_5]$ may generate two offsprings $[a_1, a_2, b_3, b_4, b_5]$ and $[b_1, b_2, a_3]$. Since the length of individuals are relatively short, we don't consider k-point crossover, $k \geq 2$.

Mutation Offsprings generated by crossover have exactly lengths with their parents (suppose two parents have lengths l_1 and l_2, then there must one l_1 and one l_2 in their two offsprings), so crossover doesn't bring in new number of layers to the population. To make mutation in both number of layers and number of nodes in a certain layer, we consider three types of mutation: deletion, insertion, and substitution (borrowed from three types of string manipulation). Deletion is to remove a layer randomly; insertion is to add a layer of random number of nodes at a random position; substitution is to replace the number of a randomly selected layer by a random number. We also set possibilities for these three types of mutation, 0.2 (del), 0.2 (ins), and 0.6 (sub) in our implementation.

Selection Since the population size is small and the fitness between each individual are quite close, we make uniform selection so that every individual has the same probability to crossover with others. As for mutation, we set the mutation possibility of an individual to be 0.5. After mutation, we keep both the original individual and its mutated offspring.

Halting condition In our implementation, the algorithm will halt after a fixed number of iterations (10) due to the extremely high computation cost. 10 iterations will almost take two days. After parallelizing the fitness evaluation part, it still takes several hours on a 8-core desktop (actually 4 physical cores with hyper-threading on).

The genetic algorithm described above is summarized in Algorithm 1.

Algorithm 1: Genetic Algorithm

Generate initial population $P(0)$ randomly, initialize a dictionary *dict* to be empty, $t \leftarrow 0$;
repeat
  repeat
    Randomly select two individuals, apply crossover, then add two offsprings to $P(t)$;
  until *crossover* 5 *times*;
  foreach *individual in* $P(t)$ **do**
    Mutate *individual* and add offspring to $P(t)$ with probability 0.5;
  Evaluate all the individuals in $P(t)$ whose fitness are not recorded in *dict*;
  Add new individual-fitness pairs to *dict*;
  Rank all individuals in $P(t)$ and select top-5 to form $P(t+1)$, $t \leftarrow t+1$;
until $t \geq 10$;
Output the individual with highest fitness in $P(10)$;

2.2 Experiment Result

The optimal structure that our genetic algorithm finds after 10 iterations is 25-96-94-10 with fitness 87.08. Other surviving structures are 25-96-88-54-10 (86.94), 25-96-88-93-94-10 (86.92), 25-90-88-10 (86.86), and 25-99-94-10 (86.84). As we can see, their common characteristics are (1) not very deep (2-4 hidden layers) and (2) number of nodes in hidden layers are very large. This result can be explained as follows. Because they are not very deep, they can be effectively trained by our algorithm and don't suffer from gradient vanishing problem. Large number of hidden units per layer gives the network high ability of approximation.

3 Other tricks

In this section, we will describe two other tricks we use in training the neural network. However, because we haven't found an automatic way to perform them (we need to monitor the process and make manual modifications at present), these two tricks are not integrated into `stage1.py` and `stage2.py`.

3.1 Data Cleaning

Since the distribution of labels in the original MNIST training are uniform but the distribution of labels in our training set are not, we believe that our training set contains a lot of noise and feel necessary to clean the dataset by removing "suspicious" instances. We think an instance is "suspicious" if we can find very few instances with same label among its neighbors. We hope this preprocessing could give us some advantages in predicting labels for test set.

3.2 Pre-training

To address the problem that deep neural networks are very hard to effectively trained, two kinds of greedy layer-wise pre-training methods have been proved to be effective, namely restricted Boltzmann machine (RBM) and auto encoder. However, RBM is originally designed to be used for binary input, whereas auto encoder accepts real value input and can be trained just like a 3-layer neural network, so we choose auto encoder to help us pre-train networks.