COMP5329 Assignment1

Dongdong Zhang
470161133
dzha4889

Quan Chen
470199228
qche7416

Rui Wang
470208162
rwan0699

Abstract

Multi-class classification has always been a classical problem in machine learning. It is a problem of classifying instances into at least three classes. Traditional approaches of machine learning such as k-nearest neighbors, naïve Bayes classifier and support vector machines can be used to address this problem. In addition to these methods, multilayer neural networks are extensively used today as a high-performance structure for classification. In this project, we propose a multilayer neural network to address the multi-class classification problem, which iteratively increase the performance using backward propagation. The network takes a dataset as an input and assign weights and bias to each data in the forward propagation. Then it utilizes mini-batch gradient descent in the backward propagation process to reduce the loss. We conduct comprehensive experiments using a dataset which includes 6,000 training data and 10 classes and reaches % accuracy.

1. Introduction

Classifying instances into multi-class has always been a trending problem in machine learning. This problem requires a discrimination of data among many classes. It is important because most applications in artificial intelligence requires multi-class classification, such as image recognition, natural language processing and data mining. All of these applications require a mass of data as input and then classify the data into several classes. Multilayer neural network has drawn extensive attention on multi-class classification since it produces state-of-art performance. The representation of features using multi layers of neurons has been proved to be extremely effective in fields of natural language processing and computer vision [1]. Therefore, it is important to understand how to build a multilayer neural network for classification.

In this project, we propose a multilayer neural network to finish a multi-class classification problem. Typically, the weights assigned to each input are set manually [1]. While in this project, we set a random matrix as the weight matrix with constrained maximum and minimum values. Then we update the weight matrix using backward propagation to reduce the cross-entropy loss. We use ReLU as activation function and weight normalization to normalize the weights. Dropout, softmax and other advanced modules are also utilized in this framework.

The rest of the report is organized as follows. Section 2 reviews the related work on multi-class classification. Section 3 introduces the proposed method. Section 4 describes the experiment and the results, followed by discussions and conclusions in section 5. The appendix section describes how to run our code.

2. Related work

The multi-class classification has always been the central problem in machine learning. Many approaches have been utilized to address this problem. Some strategies transform this problem into binary classification to simplify the classification. Such methods include one-vs.-all (OvA) and one-vs.one (OvO) [2]. Another novel approach called Hierarchal Deep Learning for Text Classification (HDLTC) was introduced in 2017 which divides the output space into a tree [3]. Other methods are basically extended from binary classification, which will be discussed in detail.

2.1. Support vector machines

Support vector machines (SVM) is one of the most classical algorithm in machine learning. The basic idea of SVM is to maximize the distance between the hyperplane and the input data. Nevertheless, traditional SVM is only used for binary classification since it only consists one hyperplane. Therefore, a typical method is to construct N SVMs for a N-class classification problem [4]. In this case, this method cannot classify the data in a simple way.

2.2. K-nearest neighbors

As a non-parametric classification method, k-nearest neighbors (KNN) calculates the distance between the data and the training data to get the output labels. It assumes that the distances between samples in the same class are smaller than those in other classes. This algorithm simplifies the process of classification, but it may not provide a remarkable accuracy.

2.3. Naive Bayes classifier

Naive Bayes classifier applies Bayes' theorem to predict the probability of the samples' labels based on the training samples. As a probability model, it assigns the probability of each label to the sample data, and the highest probability indicates the corresponding label to the data.

2.4. Decision trees

Decision tree is popular structure in machine learning. They can split the input data to different classes. Each leaf node of the tree can refer to a label. The algorithm is not complicated, yet the result is less accurate compared to neural networks.

3. Method

In this section, we will introduce the proposed structure of the multilayer neural network based on its' modules. To explain the structure of the network, we assume the input data is **X** where **X** is a d dimensional array. The actual data in our experiment are 60,000 128-dimension texts. We pro-process the data by subtracting the mean of the data and divide by their standard variance.

3.1. Hidden layers

Given the input data **X**, we feed it into the neural network as the input layer. Hidden layers are the neuron nodes stacked between the input and output of the neural network. More specifically, for one hidden layer, each node is

$$y_j = f\left(\sum_{i=1}^{d} x_i w_{ji} + b_j\right), \tag{1}$$

where y_j is the j_{th} node in that layer, f is the activation function, w_{ji} is the weight assigned to each element of **X** and b_j represents the bias. As Figure 1 indicates, this process is called feedforward in deep learning. In this part, only one hidden layer and one neuron node are used as an example. For multi-hidden layer structure, the output from one hidden layer can be fed into the next hidden layer as input. In our design, the number of hidden layers can be set by one parameter.

Another process is backpropagation. In this process, the weight matrix **W** is repeatedly adjusted to reduce the loss between the output and the actual label. This process will be explained in detail in section 3.4.

3.2. Activation function

In this design, the activation is represented by f in equation (1). Without the application of activation function, neural network is nothing but some linear operations. In that case, neural network can only be utilized as a linear

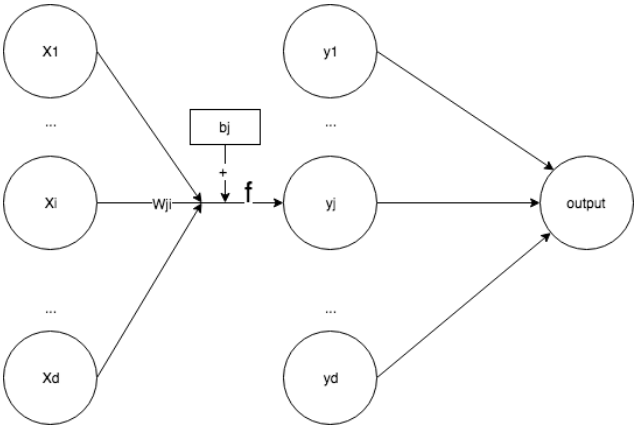

Figure 1: Feedforward process of neural network

classifier. Therefore, the activation function has to be non-linear. Secondly, the output of the activation function has to be bounded. This is because that value of the neuron will be used to represent a probability of labels. If the value of a neuron goes from minus infinity to infinity, we cannot determine which label the neuron refers to. Thirdly, the activation function should be differentiable and continuous almost everywhere. The reason to that is the derivative of the activation function will be calculated during the backpropagation process. Fourthly, the function should be monotonic. Otherwise one output from the activation function may correspond to multiple inputs. ReLU activation function is an extensively used activation function in machine learning. It is defined as

$$f(x) = \max(0, x), \tag{2}$$

where $\max(0, x)$ indicates the maximum value between 0 and x. ReLU function is one sided and can be used to avoid gradient vanishing problem where at some point, the gradient of the activation function becomes 0. Nevertheless, ReLU function indicates that the negative values of input are treated as 0s, which means some neurons will be dead. In our design, ReLU is choosen as the default activation function. There are also two activation functions as options in this project. One is sigmoid (logistic) function and the other one is tanh function. These two functions are also widely used in machine learning and they are expressed in equation (3) and (4) respectively.

$$f(x) = \frac{1}{1 + e^{-x}} \quad . \tag{3}$$

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad . \tag{4}$$

We test these three activation functions separately to determine which one provides the best performance.

3.3. Softmax and cross entropy loss

Before the output layer of the network, softmax function is utilized to assign conditional probabilities to each one of the target classes. The one with highest probability indicates the resulted label for the input data. The function can be expressed as

$$\hat{P}(class_k|x) = \frac{e^{net_k}}{\sum_{i=1}^{K} e^{net_i}} , \quad (5)$$

where net_k is $\left(\sum_{i=1}^{d} x_i w_{ji} + b_j\right)$, which is the input to the activation function. K indicates the total number of net_k. In reality, there will be some differences between the output **z** and the ground-truth **t**. To measure these errors and optimize the network, loss functions are defined. Typically, the Euclidean distance between **t** and **z** is chosen as the loss function. In practice, we discovered that cross-entropy function

$$J(\boldsymbol{t}, \mathbf{z}) = -\sum_{k=1}^{C} t_k \log(z_k) \quad (6)$$

results a better performance.

3.4. SGD, MBGD and momentum

In order to address the multiclass classification problem, the weights of the neural network should be updated repeatedly. According to the gradient descent theory, a function always decreases in the direction where the gradient decreases [5]. In this project, we tested both stochastic gradient descent (SGD) and mini-batch gradient descent (MBGD) as our approaches. SGD compute the gradient of each training sample, which is accurate yet time consuming. MBGD addresses this problem by compute the gradient for every mini-batch training data. Another problem of SGD is that the gradient may oscillates at steep areas, which are common local optima, and escapes slowly. To address this issue, we added momentum in our design to accelerate SGD. The momentum updates the gradient by

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) , \quad (6)$$

$$\theta_t = \theta_{t-1} - v_t , \quad (7)$$

where same direction gradients are increased and different direction gradients are decreased. Therefore, this approach helps the gradient descent to escape from oscillations.

3.5. Dropout

Overfitting is a common issue in deep learning. This problem means that the neural network could be overfitting with one dataset; once the data is changed, the accuracy of this network could decrease significantly. Dropout process sets some neurons of the network to be zero. In this case, equation (1) can be rewritten as

$$y_j = f\left((\sum_{i=1}^{d} x_i w_{ji}) r_j + b_j\right), \quad (8)$$

where r_j represents the Bernoulli random variable with probability p.

3.6. Batch and weight normalization

Neural network can be used for predicting results. However, since the distribution of activation could vary, leading to inaccuracy. This issue is called covariate shift. Another problem is that the gradients of activation function could vanish or explode since the derivative could be zero or extremely large. To address these two issues, we applied batch normalization in our model as follows:

i. Calculate the mini-batch mean of **X** $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$
ii. Calculate the mini-batch variance ${\sigma_B}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
iii. Normalize the value $\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}}$, where ϵ is small constant
iv. Scale and shift the normalized value to y $y_i = \gamma \hat{x}_i + \beta$, where γ and β can be learned [8]

Hence, batch normalization makes the activation similar to a Gaussian distribution with mean μ_B and variance ${\sigma_B}^2$. This approach accelerates the training.

Another module applied to our design is weight normalization. The weight of each layer is normalized by

$$\boldsymbol{w} = g\frac{\boldsymbol{v}}{||\boldsymbol{v}||} , \quad (9)$$

where g is a scalar and $\boldsymbol{v}$ is a vector in the same direction of the weight. This approach can be utilized to accelerate the convergence of SGD [6].

3.7. Weight decay

This module is applied to address two problems: firstly, it can choose the minimum weight vector to avoid irrelevant components; secondly, it can reduce some of the static noise on the targets [7]. This can be achieved by adding penalization to large weights:

$$\mathcal{L}_{new}(\boldsymbol{w}) = \mathcal{L}_{old}(\boldsymbol{w}) + \frac{1}{2}\lambda \sum_i w_i^2 . \quad (10)$$

4. Experiments and results

4.1. Dataset and implementation detail

The dataset we utilize is a .h5 file is a 128-dimension data. There are 10 corresponding classes which is label for this dataset. Overalll, traning data is (60000,128), traning label is (60000,1) and testing data is (10000,128). Before implementation, the data is normalized by subtracting the mean and diving by the standard variance. The experiments are implemented in python3 running on a MacBook Pro with 2 GHz Intel Core i5 processor and 8GB 1867 MHz memory.

4.2. Results and analysis

Through this experiment, we develop ReLU and Tanh activation, He weight initialization, MSE loss ans Softmax loss, Momentum, Dropout, weight normalization and batch normalization.

Due to the number of label is from 0 to 9, it is a 10 label classes, we translate it to one-hot representation. In those whole process, we apply ideal→Code→Experiment cycle to make all experiment and make table to record each result in order to adjust parameters. All of our experiment is completed in a disable time.

We randomly select 48000 samples from training data for training and last 12000 data for validation data. When there are good models in experiment, we use it as a final model and feed it with test data. This experiment combines different type of method and compare them in a list of tables which is in our **appendix**. All of the results are displayed in Table 1. There are experiment results and analysis below:

Basic model+ Soft max Cross Entropy Loss

Activation: Tanh, Mini-batch size: 256, Hidden layer size: 100, 80, 50, 40, 30, 20. Learning rate: 0.1 and 0.05. Epochs: 20,40, 60,80. Best accuracy time consuming: 68.98s (epochs = 80, learning rate = 0.05). Best accuracy: 85.6917%.

In this experiment, we just use Mini-batch without other method. It can be found that although the accuracy is almost same, our basic model may stay locally optimal solution. We need wo add some methods to help the result escape locally optimal solution and find a better solution.

Basic model + Soft max Cross Entropy Loss + Momentum

Activation: Tanh, Mini-batch size: 256, Hidden layer size: 100, 80, 50, 40, 30, 20. Learning rate: 0.1 and 0.05. Epochs: 20,40, 60,80. Best accuracy time consuming: 17.296554s (epochs = 20, learning rate = 0.05), Best accuracy: 86.6083%

We improve 1% accuracy in our model. It can be found that with the increasing epoch, no matter for 0.01 and 0.05 learning rate, accuracy has a decreasing tendency. Because we chose 80% of raw data as train data, overfitting may happen. This tell us sometimes it is not right to use large epochs, which will over-fit the training data and cause this model perform worse on validation data. In other words, this model will not generalized the real word enough.

Basic model + Soft max Cross Entropy Loss + Momentum + Batch Normalization + Weight Normalization

Activation: tanh, Mini-batch size: 512, Hidden layer size: 100, 80, 50. Learning rate: 0.03. Epochs: 20,40, 60,80. Best accuracy time consuming: 42.249059s (epochs = 80), Best accuracy: 87.4500%

Here we got a better accuracy which is 87.45% by adding batch normalization and weight normalization. It is noticeable that we need to use lower learning rate here. If we use large learning rate here we find with the training going, the accuracy stops at a worse grade and oscillate. This may meanings our learning rate is too large, which cause each training in our model jump a big step over the optimal point. By adding Batch Normalization and Weight Normalization, weight and batch is normalized and it let model's weight and batch in a good range to keep training going. It can be predicted that there are better accuracy here with the better learning rate and epochs.

Basic model + Soft max Cross Entropy Loss + Momentum+ Dropout with weight decay + Weight Normalization

Activation: relu, Mini-batch size: 512, Hidden layer size: 100, 80, 50. Learning rate: 0.0001. Epochs: 20,40, 60,80. Weight lambda=0.000004. Best accuracy time consuming: 28.026810s (epochs = 60), Best accuracy: 56.8333%

It is obvious that learning rate need to be very small when using relu activation. The reason may come from relu activation itself. For positive input, relu will not change it while thanh activation will change the original data. Small learning rate would keep taring going and find a considerable result. By using Softmax Cross Entropy Loss as loss function, we got a bad accuracy in our model compared with using MES Loss, which will show next part. It can be seen that when epochs increase, accuracy is improving but after 60 epochs the accuracy decrease. This model is stuck in locally optimal solution. Small learning rate and small Weight lambda will improve this model.

In our experiment, we find that Dropt out and batch normalization is exclusive. Both of them is used to avoid overfitting to some extent. When we use them together, we got a very bad accuracy.

Act.fun	Hidden Layers	LR	Epochs	Momentum	Dropout and Weight decay	Weight Norm	Batch Norm	MSE Loss	Softmax & Cross entropy	Accuracy	Training time (s)
relu	80, 90, 95	0.01	40	T	T	T	F	0.27244	F	84.55%	29.46
relu	100, 80, 50	0.001	40	T	T	T	F	0.36298	F	82.55%	27.32
relu	100, 80, 50	0.001	20	T	T	T	F	0.48432	F	79.30%	9.21
relu	100, 80, 50	0.0001	60	T	T	T	F	F	0.00397	56.83%	28.03
tanh	100, 110, 100	0.05	40	T	F	T	T	F	0.00193	85.16%	32.80
tanh	100, 80, 50	0.03	80	T	F	T	T	F	0.00193	87.45%	42.25
tanh	100, 80, 50, 40, 30, 20	0.1	20	T	F	F	F	F	0.00348	86.61%	17.30

Table 1: Results of all experiments

Change Dropt out to batch normalization

When using batch normalization, we just got 31.45% accuracy (epochs is 40). It performs worse than Dropt out. When the activation is tanh, it performs well (accuracy is 87.45%). Relu is very different with tanh, we need to use different method when we use different acativation to get a better accuracy.

Basic model + MSE Loss + Momentum+ Dropout with weight decay or batch normalization + Weight Normalization

Activation: relu, Mini-batch size: 512, Hidden layer size: 80, 90, 95. Learning rate: 0.01. Epochs: 20,40,60,80. Weight lambda=0.000004. Best accuracy time consuming: 44.188926s (epochs = 60), Best accuracy: 84.4417%.
By using Drop out with weight decay methods and MSE loss, we got 824.4417% accuracy in this model. Here we change hidden layer size to 80, 90, 95 because we find that the increasing layer size perform better. Compared with Soft max cross-entropy loss model, MSE loss perform well when activation is relu and use drop out. When using batch normalization, the accuracy is worse (64.8667%, epochs is 20), with the increasing epochs, accuracy is lower and lower. This may tell us batch normalization fit worse in relu and MES loss model.

Due to the special function Character in each activation function, we need to experiment each method by adding different learning rate, epochs and related parameter. For instance, model with relu is more sensitive than tanh, large learning rate will increase loss or oscillate around optimal point, drop out and batch normalization is exclusive to some extent, Large epochs will case overfitting in train data and let mode loss generalization in real world (test data).

In deep learning experiment, tuning parameters is essential. It needs to be compared with different method and parameters with the cycle ideal→Code→Experiment. Make a table for recoding and comparison is important. Through which we can analyze it and chose a better parameter and methods.
Overall, we have best 87.45% accuracy in our model by combining tanh activation, momentum, weight normalization, batch normalization and soft max cross-entropy loss.

5. Experiments and results

5.1. Discussions

During our work on the multi-class classification, we experienced the process to build neural networks. We experienced the procedure of planning, developing and finishing a classification project. Another important gain from this project is that we learned the details and principles of each modules in neural network. The influences of learning modules and writing codes on us are mutual. We write the codes of each module based on the equations. Meanwhile, the implementation process helps us understand the principle of each module better.

At the first stage of this project, all of the modules were implemented together. Later we figured out that this is inconvenient to debug and difficult to find the optimum result. In addition to that, some modules are conflicted in regard of optimizing. Hence, we decided to set the selections of modules as parameters. That is, we can choose which module can be utilized in the network. This enables us to try different combination of modules to get the best performance.

Overall, the experience of doing this project is positive. The whole procedure of implementation provides us the insight of how to build a neural network and why deep learning is one of the most cutting-edge technique in artificial intelligence.

5.2. Conclusions

In this project, we propose a neural network for multi-class classification. This proposed network utilizes two sets of modules which are:

i. ReLU, MSE, mini-batch gradient descent with momentum, dropout, weight normalization and weight decay.
ii. Tanh, cross-entropy, mini-batch gradient descent with momentum, batch normalization, weight normalization and softmax.

Extensive experiments demonstrate that these two combinations of modules result similar and superior performances. We take 60,000 128-dimensions data as

input and reach an accuracy of 87.45%. In the future, our work can be extended by applying more activation such as leaky ReLU and PReLU. Other works like applying and combining different and advanced model and method (such as adaptive learning rate) will be considered for next deep research work.

References

[1] J. Alvarez and M. Salzmann. Learning the Number of Neurons in Deep Networks. In *NIPS*, 2016.

[2] C. Bishop. Pattern Recognition and Machine Learning. page182, 2006.

[3] K. Kowsari, D. Brown, M. Heidarysafa, K.Meimandi, M. Gerber and L. Barnes. HDLTex: Hierarchical Deep Learning for Text Classification. In *ICMLA*, 2017.

[4] J. Platt, N. Cristianini and J. Taylor. Large Margin DAGs for Multiclass Classification. In *NIPS*, 1999.

[5] L. Bottou. Large-Scale Machine Learning with Stochastic. page 177-187, 2010.

[6] T. Salimans and D. Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. In *NIPS*, 2016.

[7] A. Krough and J. Hertz. A Simple Weight Decay Can Improve Generalization. In *NIPS,* 1991.

[8] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *NIPS*, 2015.

Appendix

Part of our testing

Activation	Hidden Layers	LR	Epochs	Momentum	Dropout and Weight decay	Weight Normalization	Batch Normalization	MSE Loss	Softmax and Cross entropy loss	Training time (s)	Accuracy
relu	100, 80, 50	0.001	20	T	T	T	F	0.484317	F	9.207963	79.30%
relu	100, 80, 50	0.001	20	T	F	T	T	0.491541	F	24.13245	64.87%
relu	80, 90, 95	0.01	40	T	T	T	F	0.272444	F	29.459971	84.55%
relu	100, 80, 50	0.001	40	T	T	T	F	0.362979	F	27.324177	82.55%
relu	100, 80, 50	0.0001	60	T	T	T	F	F	0.003973	28.02681	56.83%
relu	100, 80, 50	0.0001	80	T	T	T	F	F	0.004094	37.817429	39.70%
tanh	100, 110, 100	0.05	20	T	F	T	T	F	0.002	16.259895	84.26%
tanh	100, 110, 100	0.05	40	T	F	T	T	F	0.001929	32.802641	85.16%
tanh	100, 80, 50	0.03	60	T	F	T	T	F	0.001945	33.269108	86.26%
tanh	100, 80, 50	0.03	80	T	F	T	T	F	0.001932	42.249059	87.45%
tanh	100, 80, 50, 40, 30, 20	0.1	20	T	F	F	F	F	0.003478	17.296554	86.61%
tanh	100, 80, 50, 40, 30, 20	0.1	60	T	F	F	F	F	0.003293	66.032085	85.79%

Readme

the "mlp.ipynb" is our assignment work not only has predict data, but also has analysis every function and many experiments, if you want to review code, please use it in jupyter

if you only want to get predict test data label, please run "predict_mlp.py" in python3

in all data, if you want to change each modules and parameters, please change them at MLP and MLP.fit classes and functions, as follows

```
nn=MLP(input_data.shape[1], [100, 100, 100],
label_data.shape[1], 'activation function',
weight_norm=True/False, dropout=True/False,
keep_prob=0.8,
output_softmax_crossEntropyLoss=True/False,
weight_decay=True/False, weight_lambda=0.0008)

nn.fit(data, label, learning_rate=0.05, epochs=60,
gd='mini_batch', momentum=True/False,
gamma_MT=0.9, mini_batch_size=512,
batch_norm=True/False)
```

run all of file code need input data in the same folder