6.867 Machine Learning HW3

November 9, 2016

1 Fully Connected Neural Net

1.1 Softmax and Cross Entropy

For the multi-class classification problem, we employ one-of-K coding scheme for the target variable $t_k \in 0, 1$. Thus, for a training set of N samples $(\boldsymbol{X}, \boldsymbol{t})$ where $t_k = 1$ indicates a particular training sample labeled with class k and $t_k = 0$ otherwise. Assuming data are distributed i.i.d, we can then write the joint likelihood of the target variable according to softmax:

$$p(\boldsymbol{t}|\boldsymbol{X}, \boldsymbol{w}) = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k(\boldsymbol{x_n}, \boldsymbol{w})^{t_{kn}} \tag{1}$$

where we interpret $y_k(\boldsymbol{x_n}, \boldsymbol{w}) = p(t_k = 1|\boldsymbol{x_n}, \boldsymbol{w})$ as the conditional probability of the target variable given the nth training sample and the true label t_{nk}.

The cross-entropy loss function is hence derived from the negative log-likelihood:

$$J(\boldsymbol{w}) = LL(\boldsymbol{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{kn} \ln y_k(\boldsymbol{x_n}, \boldsymbol{w}) \tag{2}$$

where we think of $w = \{W^1 ... W^L\}$ which consist of the weight matrices for the L layers.

After defining the loss function, we need to compute the gradients $\nabla_{W^{(1)}} J(\boldsymbol{w}) ... \nabla_{W^{(L)}} J(\boldsymbol{w})$ as we follow the pseudo-code in lecture notes.

For stochastic gradient descent, we update the gradient with only 1 training sample:

$$W^l_{t+1} = W^l_t - \eta \frac{\partial E(w)}{\partial W^l} \tag{3}$$

$$b^l_{t+1} = b^l_t - \eta \frac{\partial E(w)}{\partial b^l}$$

However, in actual implementation, we find using a variant of SGD - mini-batch to be not only computational more efficient but also yield smoother, faster convergence.

We write down the gradient with respect to softmax input z_i as follows:

$$\nabla E_w(w) = -\sum_{k=1}^{K} t_k \frac{\partial \ln y_k}{\partial z_i} = -\sum_{k=1}^{K} t_k \frac{1}{y_j} \frac{\partial y_j}{\partial z_i} = y_i - t_i \tag{4}$$

Or in vectorized form,

$$\nabla E_w(w) = Y - T$$

Then we can calculate the backward-propagating δ^l as in lecture notes accordingly where

$$\frac{\partial E(w)}{\partial W^l} = \alpha^{l-1} (\delta^l)^T \qquad \frac{\partial E(w)}{\partial b^l} = (\delta^l)$$

1.2 Initialization, Regularization and Learning Rate

Initialization A common way of initializing the weights for a neural net is to use random weights distributed as a Gaussian with zero mean and standard deviation $\frac{1}{\sqrt{m}}$ for a weight matrix with dimensions $[m, n]$. This way we not only break the symmetry of all incoming m but also guarantee the variance of the units to be independent of m. For example, if we had all the units as sigmoids, with a constant variance, then large m inputs will have a higher chance of generating large variance. Since sigmoid function map input to [0,1], the output $\sigma(x)$ with large inputs will then be close to 1 or 0, and the gradient of sigmoid $\sigma(x)(1 - \sigma(x))$ will be vanishing in the process of back-propagation fairly quickly, resulting in no update for the weights. If we scale down variance by a factor of m, we make sure this undesirable effect of vanishing gradient would be mitigated to a great extent.

Regularization The regularized loss is defined as :

$$J(\boldsymbol{w}) = LL(\boldsymbol{w}) + \lambda(||\boldsymbol{w}^{(1)}||^2_F + ||\boldsymbol{w}^{(L)}||^2_F)$$

The gradient for the update equation will be different:

$$W^l_{t+1} = W^l_t - \eta(\frac{\partial E(w)}{\partial W^l} + 2\lambda W^l) \tag{5}$$

$$b^l_{t+1} = b^l_t - \eta \frac{\partial E(w)}{\partial b^l}$$

But we do not penalize for the bias so the update for the bias is the same.

Learning Rate Learning rate is crucial for successful convergence in stochastic gradient descent. To improve upon a fixed learning rate, we can dynamically use adaptive learning rate for faster and potential convergence to a global optimum. To achieve this, we will use decay factor d and momentum V :

$$\eta_{t+1} = \eta_t \frac{1}{1 + td}$$

For momentum, our update equation will become

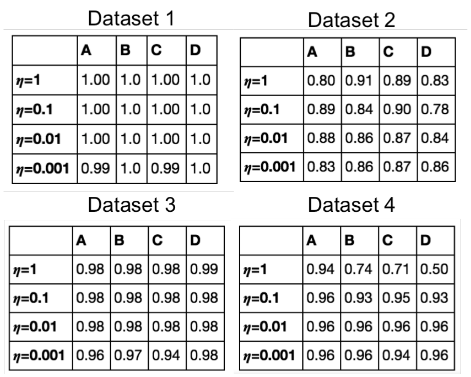
$$W^l_{t+1} = W^l_t - v_t \frac{\partial E(w)}{\partial W^l} \tag{6}$$

where

$$v_t = \gamma v_t + \eta_t \nabla J(w)$$

1.3 Implementation on 2D Datasets

We implement the neural network using back-propagation algorithm and initialized as above on the four 2D data sets from HW2. We experiment with four architectures: single small, single large, two small, and two large hidden layers (denoted by $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$, where small means 10 and large means 100 hidden units. From the rest of papers, we use these hyper-parameters unless noted otherwise. size of mini-batch = 30 samples, number of iterations = 100. We also vary learning rate from 1 through 0.001 on our mini-batch algorithm, and report the final accuracy on tables 1 and 3. We can clearly see that different architectures and data sets require fine-tuning learning rates, especially for difficulty classification problems such as 2 and 4, which are linearly inseparable. For one dataset or application, one set of parameters may be optimal while for others, they are overfitting, as in Dataset 3 versus Dataset 4 when $\eta = 1$ with D. Compared with our SVM classifiers in HW2, our best-performing architectures on the four datasets have accuracy: 100%, 91%, 99% and 96%(training) and 100%, 88%, 97% and 96% (test), which are modest improvements from SVM.



Dataset 1

	A	B	C	D
η=1	1.00	1.0	1.00	1.0
η=0.1	1.00	1.0	1.00	1.0
η=0.01	1.00	1.0	1.00	1.0
η=0.001	0.99	1.0	0.99	1.0

Dataset 2

	A	B	C	D
η=1	0.80	0.91	0.89	0.83
η=0.1	0.89	0.84	0.90	0.78
η=0.01	0.88	0.86	0.87	0.84
η=0.001	0.83	0.86	0.87	0.86

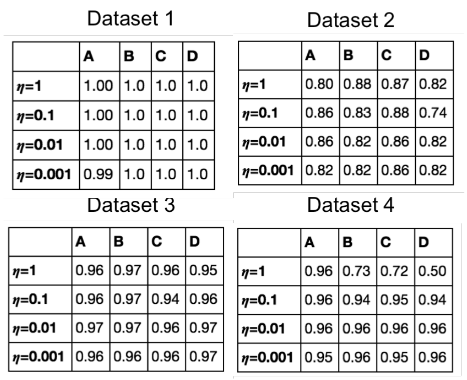
Dataset 3

	A	B	C	D
η=1	0.98	0.98	0.98	0.99
η=0.1	0.98	0.98	0.98	0.98
η=0.01	0.98	0.98	0.98	0.98
η=0.001	0.96	0.97	0.94	0.98

Dataset 4

	A	B	C	D
η=1	0.94	0.74	0.71	0.50
η=0.1	0.96	0.93	0.95	0.93
η=0.01	0.96	0.96	0.96	0.96
η=0.001	0.96	0.96	0.94	0.96

Figure 1: Training accuracy against various η in the four 2D-datasets. Columns $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ correspond to single small, single large, two small, two large hidden layers.



Dataset 1

	A	B	C	D
η=1	1.00	1.0	1.0	1.0
η=0.1	1.00	1.0	1.0	1.0
η=0.01	1.00	1.0	1.0	1.0
η=0.001	0.99	1.0	1.0	1.0

Dataset 2

	A	B	C	D
η=1	0.80	0.88	0.87	0.82
η=0.1	0.86	0.83	0.88	0.74
η=0.01	0.86	0.82	0.86	0.82
η=0.001	0.82	0.82	0.86	0.82

Dataset 3

	A	B	C	D
η=1	0.96	0.97	0.96	0.95
η=0.1	0.96	0.97	0.94	0.96
η=0.01	0.97	0.97	0.96	0.97
η=0.001	0.96	0.96	0.96	0.97

Dataset 4

	A	B	C	D
η=1	0.96	0.73	0.72	0.50
η=0.1	0.96	0.94	0.95	0.94
η=0.01	0.96	0.96	0.96	0.96
η=0.001	0.95	0.96	0.95	0.96

Figure 2: Test accuracy against various η in the four 2D-datasets. Columns $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ correspond to single small, single large, two small, two large hidden layers.

The tables above give the optimized hyper-parameters such as learning rate η and architectures (M for number of hidden units and L for number of hidden layers, which may be different for each dataset). Given the parameters, we visualize the best results in Figure 3. The left columns display the convergence of loss function for training and validation. We can clearly see that for the easy datasets such as Dataset 1, convergence is fast (with only 1 iteration). In other cases, we can see the early stopping rule in action that we wait three consecutive iterations of non-increasing loss on validation set to avoid overfitting. The right columns are decision boundary, and show quite satisfying results as compared with our SVM in HW2.

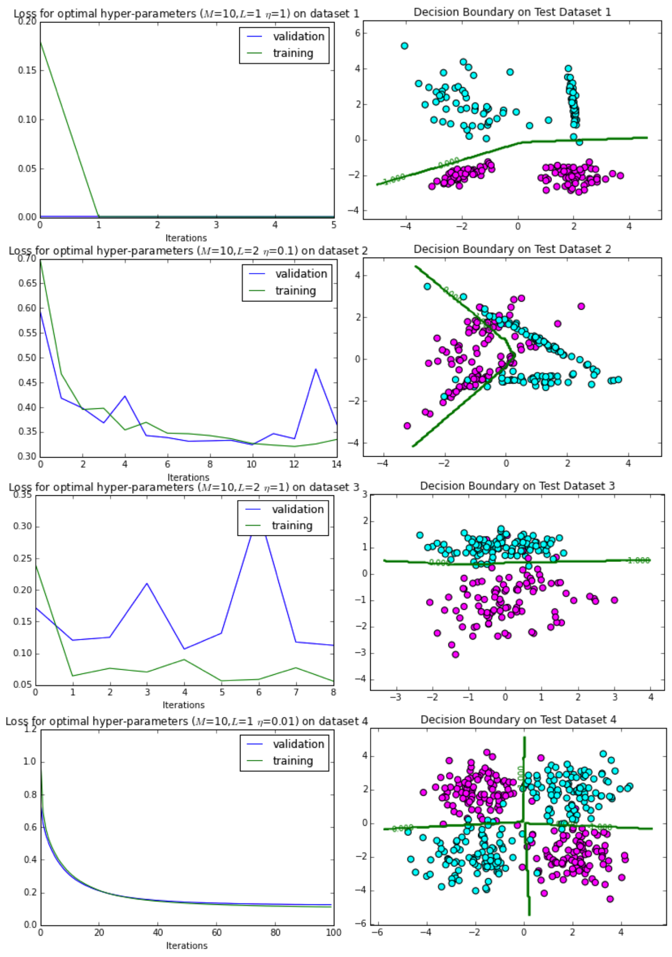


Figure 3: LEFT: Convergence of loss function RIGHT: Decision Boundary with optimal hypermeters

1.4 Implementation on MNIST

We implement the neural net on the MNIST to predict the digit 0 through 9 directly as a multi-class classification. We train 200 samples, validate 150 samples, and 150 test samples per digit. To test performance, we simply ask the neural net to predict 150 test samples of 10 digits, and calculate the total accuracy for all digits. Since this is much more complex dataset than the previous one, we also fine-tune the learning rate with decay factor d and momentum, of which we detail the process in the next paragraph. We also train 400 samples per digit with fine-tuned parameters. We find that normalization is very helpful for tuning, since without normalization, most of the time the neural net gives out accuracy less than 10% and in the end giving suboptimal performance.

Here we report the results: Fine-tuned, 200 samples: 93.6%; fine-tuned, 400 samples: 96.0%; un-normalized 200 samples: 86.7%, which clearly indicates the benefit of more, normalized training samples.

Although we cannot directly compare with the binary classification tasks in HW2, we can still see that predicting digits are strictly harder than binary classification. Therefore, we find the base result 93%, which seems at par with our SVM results, is based on a harder task and should be given more credits.

Fine-tuning the Network Now we have to fine-tune a large set of hyper-parameters: learning rate, decay factor, momentum, number of layers L, number of hidden units per layers M. For the first three parameters related to learning rate, which we found to be most sensitive, we combine a greedy and binary search approach for our validation process. Take learning rate η as an example. We first search in the range (1e2,1e1 ... 1e-3,1e-4), and obtain optimal η_1 from the validation set. Then we search again in the subinterval centering around η_1 and obtain η_2. Given one optimal parameter, we then fix it and search for the next parameter in the same fashion. The search process searches, in this order, learning rate ranging 1e2 through 1e-4, decay factor ranging 1e-1 through 5e-4, momentum ranging 0.1 through 0.9, L from 1 through 5 and 10, and M ranging 8,16,...512, and is visualized in 6. Each graph points out a unique maximum within the range we search in with fixed parameters given in the title. We in the end find $\eta = 0.01$, decay factor $= 0.0004$, momentum=0.9, $L = 2, M = 256$ to yield the best performance.

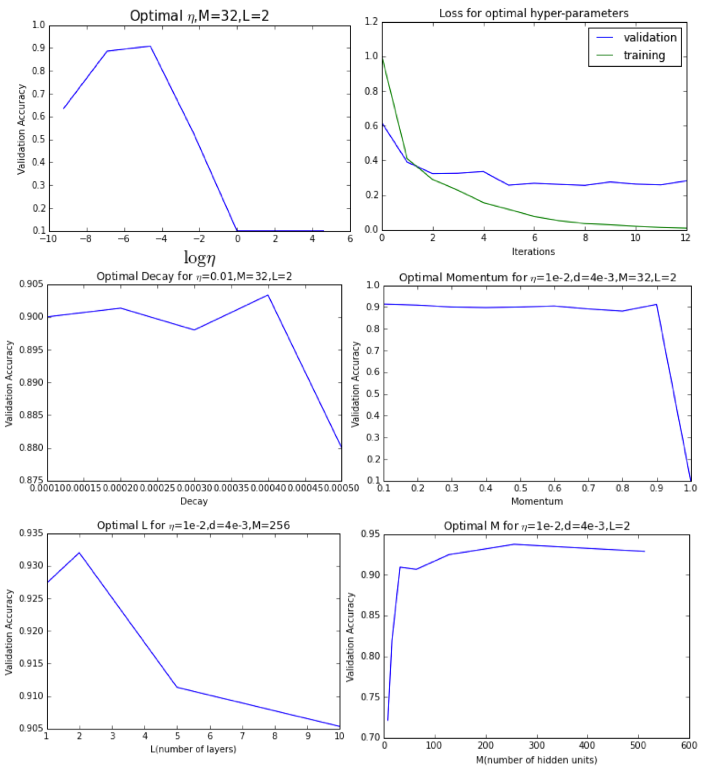


Figure 4: Process of fine-tuning the hyperparameters. Smooth convergence of loss function given the optimized hyperparameters is shown on top right

Predicting Individual Digits We can find interesting patterns in the accuracy of predicting individual digits. With our greedy approach, using and tuning decay factor and momentum is a difficult task. From our experience, adding momentum and schedule for learning rate makes tuning significantly harder. The following graphs highlight the stability of results between the two cases. We see predicting digits like 3 seems harder and more sensitive parameter change than others. With fine-tune learning rate, we often see unstable results producing accuracy from 10% 24%.

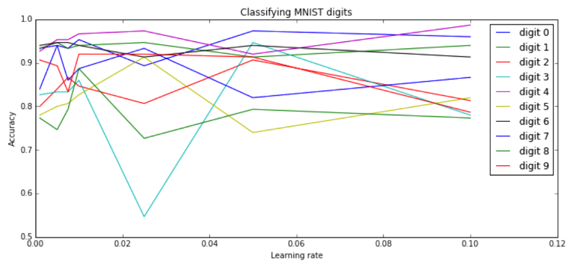


Figure 5: Untuned results: M = 32, L = 2, learning rate = 0.01, no decay factor or momentum

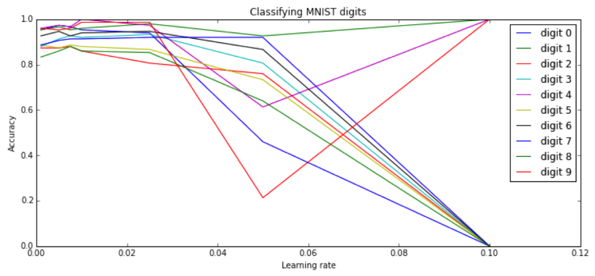


Figure 6: Fine-tuned results with our optimized parameters

We remark that given a particular set of decay factor and momentum, weights can be very sensitive to learning rate, possibly causing significant overfitting and unstable out-of-sample performance. Compare the summary tables **??** and 2. We see a greater range of performance if we use decay factor and momentum than otherwise. There are cases where test accuracy is boosted to 100% yet, whereas in other cases, the test performance is worse than that without decay factor and momentum, indicating overfitting behavior. The mixed again highlights the importance and difficulty of tuning. In addition, we observe in 3 that more training samples show improvements in performance.

Digits	0	1	2	3	4
Max acurracy	97.3%	94.7%	90.7%	94.7%	98.7%
Min accuracy	89.3%	91.3%	80.0%	54.7%	92.0%
Digits	5	6	7	8	9
Max acurracy	91.3%	94.7%	94.0%	88.7%	92.0%
Min accuracy	74.0%	91.3%	82.0%	72.7%	78.7%

Table 1: Range of accuracy against η (untuned). M = 256, L=2

Digits	0	1	2	3	4
Max accuracy	97.3%	100.0%	88.0%	93.3%	100.0%
Min accuracy	0.0%	92.7%	0.0%	0.0%	61.3%
Digits	5	6	7	8	9
Max accuracy	88.7%	94.7%	92.0%	88.0%	100.0%
Min accuracy	0.0%	0.0%	0.0%	0.0%	21.3%

Table 2: Range of accuracy against η (tuned). Decay factor = 4e-4, and momentum=0.9. M = 256, L=2

Digits	0	1	2	3	4
Max accuracy	96.67%	97.33%	90.00%	91.33%	99.33%
Min accuracy	92.67%	92.67%	84.00%	85.33%	92.00%
Digits	5	6	7	8	9
Max accuracy	90.67%	96.00%	96.67%	88.67%	96.67%
Min accuracy	84.00%	92.67%	88.00%	80.67%	85.33%

Table 3: Range of accuracy against η (tuned). With 400 training samples

2 Convolutional Neural Networks(CNN)

In this section, we will implement convolutional neural networks to perform image classification. The dataset we will use for this assignment was created by Zoya Bylinskii, and contains 451 works of art from 11 different artists all down-sampled and padded to the same size(50*50 pixels).

2.1 Introduction to CNN

CNN uses a special architecture which is particularly well-adapted to classify images. Suppose we have 50*50 pixel input, instead of using fully connected neurons, we used a 50*50 square neurons to connect to the input. Then we add two convolutional layers, one is 5*5 and another is 3*3. The first convolutional layer sums up every weighted(w matrix) 5*5 pixel region, the stride is 1, which is very similar to the operation of convolution. After the first convolutional layer, we get feature map Z_1. The second layer will do the same thing on Z_1, but the region size changes to 3*3, which we call feature map Z_2. Then for each node in Z_2, the receptive field is 7*7. The more layers you add, the bigger receptive field will be. Then if the input image has very large object, the larger receptive field will be more likely to cover the whole object, which makes the CNN easier to get useful information and make right decision.

2.2 Test Tensorflow CNN

We used tensorflow package in python to implement the CNN. We build 4 layers in the neuron network(NN). The first two layers are convolutional layers and the other two layers are fully connected layers containing 64 neurons each. For the convolutional layer, we use 5*5 filter size and the stride is 2. The depth of convolutional layers is 16, which means we use 16 different kinds of filters. The reLU function is used as the activation function in the hidden units. The loss function is cross entropy function:

$$H(p,q) = -\sum_x p(x)\log(q(x)) \tag{7}$$

The output of CNN is the probability for each artist. We used stochastic gradient decent to minimized the loss. The step size is 0.01, batch size is 10. The training steps we use is 1500.

We use 364 rgb images as training data and 87 images as the validation data. The raining accuracy we get is 95.0% and validation accuracy is 66.8%. Obviously, we over fit the data. Note that since there are variations with different trial of training due to the random initialization of the weight vector, we repeated the code for 10 times and averaged the accuracy. And all of following accuracy are also averaged ones.

2.3 Add pooling layers

In addition to the convolutional layers just described, convolutional neural networks also contain pooling layers. Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer. For instance, each unit in the pooling layer may summarize a region of (say) $2*2$ neurons in the previous layer. Here, we use max pooling, which simply outputs the maximum activation in the $2*2$ input region. Different size of pooling region and different strides may result in different accuracy. We list the results in the following table.

Size/Stride	1	2	3	4
2	92.5%/67.4%	77.5%/65.7%	NA	NA
3	90.8%/70.8%	79.0%/67.6%	66.9%/55.2%	NA
4	88.3%/67.8%	77.1%/67.5%	69.6%/60.3%	58.6%/49.1%

Table 4: Training/validation accuracy with different pooling sizes and strides

When we choose proper pooling size and stride, the validation accuracy increase to 70.8%, the training accuracy gets to 90.8% when pooling size equals to 3, stride equals to 1. But if the pooling size and stride are not proper, there is no improvement or the accuracy get even worse. Adding pooling layer and choose proper size and stride helps to increase accuracy.

2.4 Regularize the network

When the dataset is small, it is very easy for the CNN to overfit the data. We will try four different ways to regularize the network. 1) Dropout. The key idea of dropout is to randomly drop units (along with their connections) from the neural network during training. There is a keep probability for each unit, you can set it from 0.5 to 1. We try different values of the keep probability in our CNN to prevent over-fitting problem, the change of training accuracy and validation accuracy with it is shown in Figure 7. When keep probability equals to 0.95, we get the maximum accuracy, but the improvement is not obvious.

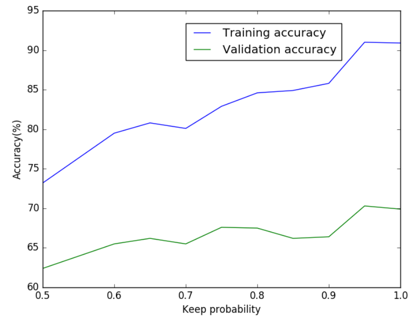


Figure 7: Training and validation accuracy with different keep probabilities

2) Weight regularization. Like what we did in ridge regression or LASSO, we can add a penalty term in the loss function to regularize the weight function. Here we used $L2$ norm penalty. The hyper-parameter λ we try ranges from 0.0001 to 1. The following figure shows the training accuracy and validation accuracy with different λ. As the weight penalty gets stronger, both training accuracy and validation accuracy get lower. So weight regularization doesn't works well for our CNN.

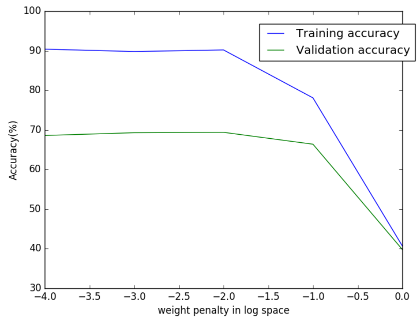


Figure 8: Training and validation accuracy with different weight penalty

3) Data augmentation. When the amount of data set is small, we can make new data by doing some operation to the training images, like randomly flip the image or do some distortion. But here the validation data is unchanged. After we do augmentation to the training data, the data set becomes 4 times bigger and we increase the step number to 1800. The training accuracy we get is 69.0%, validation accuracy is 62.3%. Data augmentation works not very well.

4) Early stopping. Stop training your model after your validation accuracy starts to plateau or decrease. Originally we are using 1500 steps. To figure out the best stopping step number, we record the training and validation accuracy for every 100 steps. Then plot in Figure 9. From the figure, we can tell the training accuracy increase when the step number get bigger. However, the validation accuracy get to peak when step number equals to 900, and after that decrease. Controlling the step number is a very effect way to avoid over fitting.

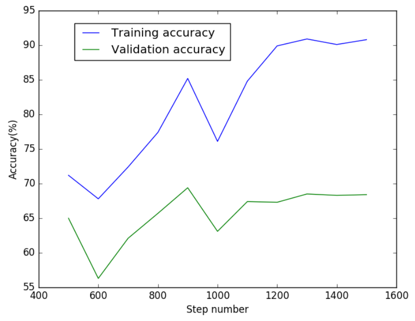


Figure 9: Training and validation accuracy with different step number

Overall, dropout and controlling proper step number are effective ways to avoid over fitting problem for our CNN. Weight regularization and Data augmentation seems doesn't help much.

2.5 Experiment with the architecture

The architecture of CNN could also affect the accuracy a lot. Here we analyzed the effect of filter size, stride and depth(kind of filter) in the convolutional layers, pyramidal-shaped network and reverse pyramidal-shaped network. The training and validation accuracy for different filter size is listed in Figure 10. From the curve, we can tell that there is no big different when filter size varies from 5 to 9, but when filter size gets too big, when validation accuracy decrease dramatically. For effect of different stride is also shown in Figure 10 (right). We fix filter size equals to 5, test stride with 2, 3, 4. 2 and 3 give similar result, but when stride is too big, the accuracy decrease, which makes sense since small overlapping area may lose useful information.

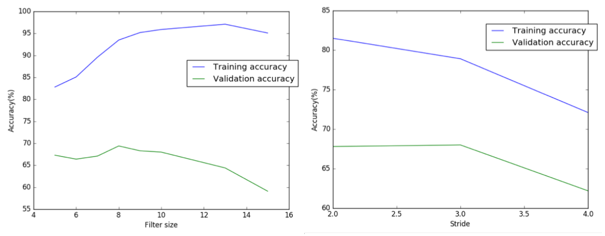


Figure 10: Training and validation accuracy with different filter size(left), with different strides (right)

The effect of different depths of the convolution layers to training and validation accuracy is shown in the following table. We can tell that when the depth of the first is 28 and second layer is 16, we achieve the best performance. The validation accuracy is 69.9%. Basically it is a reverse pyramid-shape network.

layer 1 / 2	16	20	24	28	32	36
16	81%/66.2%	81.5%/61.4%	80.8%/64.8%	75.6%/63.4%	77.3%/66%	74.4%/62.5%
20	84%/69.6%	82.1%/69.5%	NA	NA	NA	NA
24	84.1%/69.4%	NA	81.7%/65.9%	NA	NA	NA
28	87.4%/69.6%	NA	NA	82.0%/64.4%	NA%	NA
32	86.6%68.5%	NA	NA	NA	81.8%/64.4%	NA
36	86.7%68.0%	NA	NA	NA	NA	84.1%/65.5%

Table 5: Training / validation accuracy with different depths in layer 1 and layer 2

2.6 Optimize the architecture and test the final architecture

The best architecture we are using contains two convolutional layers, 2 pooling layers and 2 fully-connected layers. The filter size of the two convolutional layers is 5 and stride is 2. The depth of the first layer is 28 and the second layer is 16. The two pooling layer used max pooling with size 3*3 and stride equals to 1. The two fully-connected layer contains 128 neurons each. To overcome the over fitting problem, we use dropout (keep probability is 0.95), data augmentation and early stopping (step number is 900). The best validation accuracy we achieve is 77.0%, training accuracy is 91.2%. To test the final architecture, we used some data with variations, like color-inverted, translated, flipped, brightened, darkened, high-contrast, low-contrast, as the test data. The results are accuracy on normal validation data: 72.4%, accuracy on translated data: 36.8%, accuracy on brightened data: 62.1%, accuracy on darkened data: 71.3%, accuracy on high contrast data: 62.1%, accuracy on low contrast data: 71.3%, accuracy on flipped data: 46.0% and accuracy on inverted data: 10.3%. The performance of the CNN does surprise me in a bad way. For the normal data, the accuracy is fine. The lowest error we got is from the color-inverted images, which means our CNN uses a lot of color information of the images. Brightness and contrast affect a little bit, but not much, since we used data augmentation in the training set. Translated and inverted data give very bad performance, which means our CNN uses spatial location of the images and we probability should add the translated and flipped images to our training data.