6.867: Homework 3

1. Neural Networks

In this section, we explore neural networks and various choices for the number of hidden layers, the number of neurons per hidden layer, regularization, binary classification, and multiclass classification. We investigate how our choice of these variables impacts the overall classification rates and how these decisions are incorporated into our implementation of neural networks.

1.1. ReLU + Softmax

We have here implemented a neural network in which all the hidden units have a ReLu activation function and the final output layer has a Softmax activation function. This is incorporated into the implementation where in the output layer, the activation α_i for class i is determined by the Softmax function:

$$f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Thus, for a k-class classification problem, the output layer has k neurons each with Softmax activation. The prediction is therefore given by the argmax of the *alpha* values calculated by the Softmax for each of the k neurons.

The cross entropy loss is the loss function used here and is given by:

$$-\sum_{i=1}^{k} y_i \log(f(z)_i)$$

This formulation of the loss function is essentially a calculation of our confidence in the correct classification. In other words, the loss is the log likelihood of the correctly classified class by the weights assigned in our training of the neural network model. This loss is incorporated into the implementation via the output error. This output error is then involved in backpropagation to determine δ values for each layer in the neural network, which is subsequently used to calculate the gradient used for stochastic gradient descent. Thus, our choice of the Softmax function as the output activation function and the cross entropy loss as the loss function in this case is critically important in our overall implementation. Different choices of these functions could fundamentally change the training and thus classification accuracy of our overall model.

1.2. Initialization

We can initialize the weights to be randomly selected from a Gaussian distribution with zero mean and standard deviation $\frac{1}{\sqrt{m}}$ for an mxn weight matrix. Choosing to initialize weights to these values is a reasonable decision because m represents the number of input neurons in each of our layers. This is selected to control the variance of the distribution in order to ensure that each layer in the neural network is agnostic to the number of inputs it receives, and instead depends on the values of the inputs.

1.3. Regularization

This would impact the pseudocode because now we must also incorporate this additional regularization term when taking the gradient of the loss function. Since this gradient is used in the stochastic gradient descent update it ultimately impacts the entire algorithm. More specifically, the update step before regularization was:

$$\theta \leftarrow \theta - \eta \frac{\delta l(y, F(x;\theta))}{\delta \theta}$$

With regularization, the update step now becomes:

$$\theta \leftarrow \theta - \eta \frac{\delta J}{\delta \theta}$$

where

$$J(\theta) = L(\theta) + R(\theta)$$

and

$$R(\theta) = \lambda * (\sum_{ij} {w_{ij}^{(1)}}^2 + \sum_{ij} {w_{ij}^{(2)}}^2)$$

Furthermore,

$$J'(\theta) = L'(\theta) + R'(\theta)$$

$L'(\theta)$ has already been calculated and $R'(\theta)$ is given by:

$$R'(\theta) = 2 * \lambda * W$$

	(5)	(100)	(5,5)	(100,100)
1	0.995	0.997	0.945	0.997
2	0.819	0.835	0.785	0.840
3	0.965	0.961	0.948	0.968
4	0.934	0.951	0.919	0.951

Table 1. Training accuracy, l layers, n neurons/layer

	(5)	(100)	(5,5)	(100,100)
1	0.993	0.997	0.945	0.995
2	0.799	0.813	0.768	0.823
3	0.961	0.962	0.945	0.948
4	0.942	0.956	0.925	0.958

Table 2. Testing accuracy, l layers, n neurons/layer

1.4. Binary classification

We test our implementation on the 2D datasets provided in HW2. The accuracies for the training and testing data sets are presented for different architectures of the neural network in Tables 1 and 2. Interestingly, we found that our implementation of the neural network exhibited significant variability and instability, even for small networks. This could be attributed to the randomly initialized weights, as well as the fact that the algorithm terminates when the validation error no longer decreases. Other choices of the termination criterion would have affected the overall accuracy of the algorithm however. The numbers presented in Tables 1 and 2 therefore illustrate the average accuracies when each neural net training process was run 20 times with a learning rate of 0.1.

In general we found that the training and testing accuracies were fairly comparable with each other. Furthermore, the number of neurons and the number of layers did not significantly affect the accuracy rates, although the classification rate did suffer slightly in a smaller neural net (1 layer with 5 neuron per layer and 2 layers with 5 neurons per layer). The neural network consistently had the worst performance for dataset 2, which makes intuitive sense as dataset 2 was not linearly separable and had many overlapping samples of different classes.

We additionally found that learning rate made a significant difference on the performance of the algorithm. This is plotted in Figure 1, in which the learning rates for different neural net architectures is compared with the average (over 20 runs as before) testing classification accuracy. It is interesting to note that the general shape of the curve plotting learning rate against testing accuracy is generally the same even for different architectures of the neural network. More specifically, accuracy peaks at a learning rate of about 0.1 to 1, and suffers quite significantly for greater and smaller learning rates. Furthermore, for suboptimal choices of the learning rate, the accuracy sometimes dropped below 50%, at which point, the classifier effectively performs no better than random guess. In this case, these results were generated based on dataset 1, but similar results are fairly likely for other datasets.

	Logistic	Linear SVM	RBF SVM	Neural net
1	0.99	0.99	0.99	0.997
2	0.805	0.81	0.835	0.813
3	0.97	0.97	0.96	0.962
4	0.50	0.508	0.963	0.956

Table 3. Testing accuracies of different algorithms

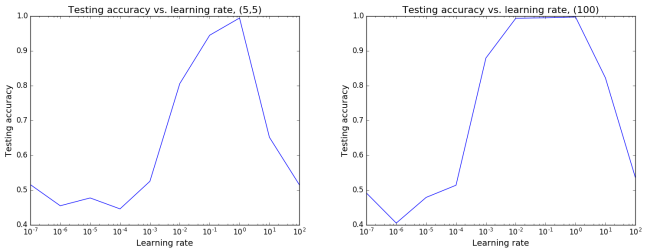

Figure 1. Learning rate vs. testing accuracy

Comparing these results with the accuracies from HW2, we see that the neural network performed significantly better than logistic regression with various types of regularization for non-linearly separable data. For linearly separable data, the accuracies were very comparable. Comparing these results with support vector machines, the neural network performed approximately as well as an SVM with a linear kernel for all the datasets with the exception of dataset 4, which displayed very poor performance. For SVMs with a radial basis kernel however, we were able to improve on the performance of a linear kernel SVM for dataset 4, and overall, the radial basis SVM displayed comparable, if not slightly improved performance, over neural nets. This comparative analysis is illustrated in Table 3, where the neural net accuracies are generated based on a neural net of one layer with 100 neurons per layer.

1.5. Multi-class classification

2. Convolutional Neural Networks

In this section, we explore the application of convolutional neural networks in performing image classification.

2.1. Convolutional filter receptive field

The dimensions of the receptive field for a node in Z_2 is $7x7$. With this in mind, we see that it is effective to build convolutional networks deeper in order to increase the receptive fields for subsequent nodes. This is effective because it allows the network to learn additional features that take into account larger and larger amounts of the original image's pixels.

2.2. Run the Tensorflow conv net

In the "define_tensorflor_graph" graph function we see that there could be a maximum of 6 layers and a minimum of 4 layers. Only 2 of the layers are convolutional. 2 other ones are regular hidden ones and the 2 optional ones are pooling layers. The relu activation function is used on the hidden nodes. The softmax loss function with cross entropy is being used to train the network. The loss is being minimized with gradient descent. When the convolutional network is run, we get a batch training accuracy of 100% and a validation accuracy of 63.2%. What this means is that the network is classifying extremely well on the training data, while not so well on the validation datasets, which points towards overfitting.

2.3. Add pooling layers

By adding pooling layers to our convolutional network, our results definitely changed. Specifically, we used a layer 1 pool filter size of 5, a layer 1 pool stride of 2, a layer 2 pool filter size of 2, and a layer 2 pool stride size of 2. After the 1500th step, this achieved a batch training accuracy of 100% and a validation accuracy of 69%. Interestingly enough, after the 1400th step, the training accuracy was 70% and the validation accuracy was also 70%. With these results it appears that max pooling marginally reduces the training accuracy while improving the validation accuracy. In a sense, it almost serves as a form of regularization that is able to prevent overfitting because now instead of the final layer processing information from all the original pixels the pooling allows clusters of pixels to be expressed as one value which enables better generalization.

2.4. Regularize your network!

2.4.1. Dropout

Dropout is a form of regularization that involves getting rid of certain activations. This prevents the units from co-adapting too much and is a good strategy to prevent overfitting. Through testing a variety of dropout values ranging from 0.5 to 1.0, it was discovered that optimum dropout rates for this convolutional network were in the range from 0.8 to 0.9. For a dropout rate of 0.9, we were able to get 70% training accuracy and 71.3% validation accuracy. Once the dropout rate hit 0.7, the results were noticeably worse (60% training accuracy and 59.6% validation accuracy). The one interesting thing we did notice is that the lower the dropout rate the less time it takes for the algorithm to complete. This makes intuitive sense because it means less calculations need to be made.

2.4.2. Weight regularization

Another way to avoid overfitting is to simply add an additional term to the loss function that penalizes the norm of the network's weights. This is a technique that we've used countless times in the class already. Through testing a variety of weight penalties from 0.0 to 1.0, it was discovered that the optimum weight penalty is between 0.01 and 0.05. With a penalty of 0.02, the training accuracy was 70% while the validation accuracy was 74.7%. Any penalty greater than 0.1 started causing poor performance. For example, a penalty of 0.2 resulted in a training accuracy of 60% and a validation accuracy of 49.4%.

2.4.3. Data augmentation

Another effective regularization technique is to augment the data with transformed versions of the original samples. When we use an augmented dataset that contains randomly flipped and distorted versions of the original training images as well as increase the number of training steps to 6001, we were able to get a batch training accuracy of 90% and a validation accuracy of 71.3%. This definitely is better than the original validation accuracy we got, which leads me to believe that it did in part solve the overfitting problem.

2.4.4. Early stopping

The final regularization technique that we tested was early stopping. The technique is quite simply and simply involves stopping the training as the validation accuracy starts to plateau or decrease. Through many tests, it was found that the validation accuracy starts to plateau at 1200 steps. At this time, the training

accuracy is 70% and the validation accuracy is 70.1%.

2.4.5. Discussion

While all regularization techniques were moderately effective, it seems that from rigorous testing that weight regularization was just slightly better than the rest by a couple percentage points.

2.5. Experiment with your architecture

In this section we will investigate other means to tweaking our architecture in order to improve the performance of the convolutional network. First we will look at what happens when we increase the filter size, stride, and depth of the convolutional layers.

Filter size: When the filter size is increased it is clear that there are two predominant effects. First, the total algorithm run time increases. When the filter size is 5 for both layers, the algorithm finishes in 21.78 seconds while when the filter size is 15 for both layers, the algorithm finishes in 96.24 seconds. The second effect that I noticed is that the validation error plateaus much earlier. For filter sizes of 5 and 7, the validation error plateaus at around 1200 steps while for filter sizes of 10 and 15, the validation error plateaus at around 900 steps. The best validation error was achieved with a filter size of 10, which was able to get a training accuracy of 100% and a validation accuracy of 75.9%.

Stride: Increasing the stride seems to have the exact opposite impacts compared with increasing filter size. Specifically, for a given max number of training steps, the higher the stride, the faster the algorithm takes to complete. For example, with 3001 steps, it takes the algorithm 48.92 seconds to complete when the stride is 2 while only 23.09 seconds to complete when the stride is 4. This makes intuitive sense because with a larger stride, there are less patches to deal with. The other discovery we made regarding stride is that with a higher stride it take the algorithm many more steps before the validation accuracy plateaus. With a stride of 2, the validation error plateaus at around 1200 steps while with a stride of 4, the validation error doesn't plateau until step 2800.

Depth: Increasing the depth seems to yield effects very similar to increasing filter size. First off, as depth increases, it takes the algorithm a longer time to run to completion. For example, with total training steps set to 1501, it took the algorithm 26.31 seconds to finish when the depth was 16 and 64.76 seconds to finish when the depth was 64. Additionally, it was observed that increasing the depth made the validation error plateau faster. At a depth of 16, the validation error plateaued at 1400 steps, at a depth of 32, the validation error plateaued at 1100 steps, and at a depth of 64, the validation error plateaued at 500 steps.

After we investigated the impact of each of the above factors independently, we then proceeded to test combinations of the factors together as architectures. We tested a pyramidal shaped architecture with the feature maps gradually decreasing in height and width while increasing in depth, a flat architecture, and an architecture with an inverse shape. Here's what we found:

Pyramid-shaped: The pyramid-shaped architecture performed relatively well. It completed 1500 steps in 54.53 seconds while converging (validation error plateaus) at the 1100 step mark to yield a training accuracy of 100% and a validation accuracy of 71.3%.

Flat: The flat architecture did slightly better compared with the pyramid-shaped one. It completed 1500 steps in 52.36 seconds while converging at the 900 step mark to yield a training accuracy of 100% and a validation accuracy of 75.9%.

Inverse pyramid-shaped: The inverse pyramid-shaped architecture is actually impossible to create because while you can maintain the original width and height, there is no way to increase it.

2.6. Optimize your architecture

The best performance we were able to achieve was a training accuracy of 90% and a validation accuracy of 78.2%. This was achieved through using the original data set, setting the layer 1 filter size to 5, the layer 1 depth to 64, the layer 1 stride to 2, the layer 2 filter size to 10, the layer 2 depth to 16, and the layer 2 stride to 2. Furthermore, the layer 1 pool filter is 5, the layer 1 pool stride is 2, the layer 2 pool filter is 2, and the layer 2 pool stride is 2. Finally, we also had a weight penalty of 0.01 and a maximum training steps of 1500.

2.7. Test your final architecture on variations of the data

With the optimal architecture that we found in the previous section, we tested our convolutional network on the transformed datasets. The following are the results we obtained for each of the transformations:

From these results there are two particular ones that surprised me. First, I was pretty surprised by the huge accuracy discrepancy between brightened data and darkened data. Another result that really intrigued me was seeing that inverted data only contained a 8%

Image Distortion	Validation Accuracy
Translated	44.8
Brightened	37.9
Darkened	69.0
High Contrast	37.9
Low Contrast	69.0
Flipped	51.7
Inverted	8.0

Table 4. Performance on distorted data

accuracy. While I understand how the network would have problem with geometrical transformations, 8% is literally how well a random algorithm would have performed. The transformations that my network is most invariant to are darken and low-contrast transformations. The transformations that lead to my network performing poorly are brightened data, high contrast data, and inverted data. What this means is my network has prioritized features that relate to the hue of the pixels. Specifically, the features probably depend on the overall magnitude of the hue as well as the relative hues of neighboring pixels. This explains why darkened and low contrast was able to perform well while brightened, high contrast, and inverted not so much. Additionally, it seemed like the network also learned some features related to the relative local positions of the pixels which is why translated and flipped were also able to do decently.