Deep Learning with PyTorch

Patrick Tirtapraja
Purdue University
ptpraja@purdue.edu

Abstract

This paper utilizes TensorFlow to train a deep convolutional neural network model with the MNIST dataset to obtain its accuracy. The first approach was to perform the experiment with a very simple model that yields a low accuracy of 91.86%. Hence, to improve the accuracy, a deep convolutional neural network was implemented and used to yield an accuracy of 99.27%. Throughout the experiment, the MNIST dataset was learned, softmax regressions were implemented, and basic machine learning model parameters were learned.

1. Introduction

Deep learning uses an artificial neural net that is composed of several levels arranged in a hierarchy to carry out machine learning processes. It has attracted a lot of attention recently because it is particularly good at analyzing data and has the potential to be very useful for real-world application [3]. After a model was trained, one of these asset is to be able to classify and predict certain classes or objects within a dataset. For this experiment, the MNIST, a simple computer vision dataset, was used. It consisted of images of handwritten digits ranging from 0 to 9 saved as a Python list of unsigned bytes. The labels are also in a Python array of unsigned bytes. Hence, several lines of code saved in "showImage.py" was ran to display some random numbers in the dataset as shown below.

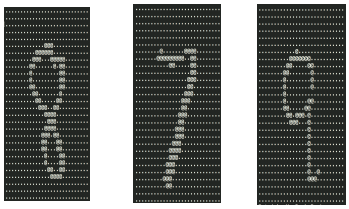

Figure 1: Images in the MNIST dataset

The dataset also includes the label for each image, which provides information as to what digit corresponds to the handwritten images. For example, in Figure 1, the labels are 8, 7, and 9 respectively.

In this experiment, a model, called the softmax regression, was trained to look at the images and predict what digits they are. A function was created to be the model to recognize the digits by having it look over hundreds or even thousands of these examples inside the MNIST dataset. This was carried out by running a TensorFlow session and the model's accuracy compared with the original dataset will be the output of the session.

2. Methods

2.1. MNIST Dataset

The MNIST data, short for Modified National Institute of Standards and Technology, was obtained from a website hosted by Yann LeCun [4]. The dataset has three different parts. The first part contains 55,000 data points of training data called "mnist.train", the second contains 10,000 points of test data called "mnist.test", and the third 5,000 points of validation data called "mnist.validation". Each image in the dataset is 28 pixels by 28 pixels flattened with 784 values representing each pixel's intensity of which darkest means that there is a huge correlation that the digit occupies that specific index in the array. In Figure 2 below, a random digit was pulled out by running "plotVisual.py". As can be seen from the figure, the image has a pre-defined label 8 corresponding to the image resembling a hand-written number 8 shown by the dark colored pixel.

Figure 2: Representing an image as a matrix of pixel's intensity

The array was flattened to a vector containing 784 numbers obtained from multiplying the array dimensions (28 and 28). The result will be a tensor (an n-dimensional array) called the "mnist.train.images" and will have a shape of [55000, 784]. The first dimension, of maximum size

55,000, represents the index of the specific image used in the dataset and the second dimension, of maximum size 784, represents the index of the pixel in the specific image. As mentioned above, the entry in the tensor is a pixel intensity between 0 and 1 for a particular pixel in the image.

Within the MNIST dataset, each image has its own corresponding label of a number between 0 and 9 to represent the number that corresponds to the image. The labels were used as a "one-hot vector" which is a vector that has a value of 0 in most of its dimensions, and 1 in a single dimension. The n^{th} digit will be represented as a vector with a value 1 in the n^{th} dimension. For example, the number 5 will be represented as [0, 0, 0, 0, 1, 0, 0, 0, 0, 0] and 7 will be represented as [0, 0, 0, 0, 0, 0, 1, 0, 0, 0].

2.2. Softmax Regressions

Softmax regression is a natural model where it has the ability to assign the probability of how close an object is in being one of several things and represented this probability between a value of 0 and 1. The softmax regression carried out two steps. The first one added up the evidence of an input being in certain classes and the second converts that evidence into its corresponding probability value. The evidence was added up by calculating the weighted sum of the pixel intensity on how close it correlates to a certain class, in this case, a number. The weight is positive if the evidence is in favor of the image being in the class and negative if it does not.

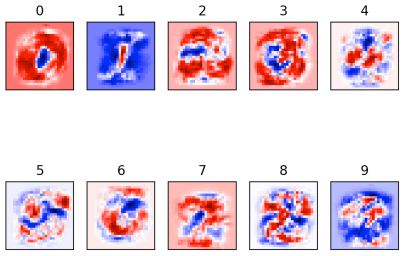

Figure 3: Representation of weights a model learns for the ten classes: red as positive weights, blue as negative weights

Figure 3 shows an imagery of how the model learns all of the ten different classes. The blue coloring represents negative weights, and red represents the positive weights. An additional evidence was also used, called the bias. A bias can be positive or negative and a large bias can lead to high output correlations [5]. The evidence for a class i given an input x can be represented as (1):

$$\text{evidence}_i = \sum_j W_{i,\ j} x_j + b_i \qquad (1)$$

Where W_i is the weight and b_i is the bias for class i. Indices j represent the indices of the pixel location in the input image. Since the probability distribution over 10 cases was desired, a "softmax" function was used to serve as an activation function to shape the output of the initial linear function to the desired distribution. The softmax function was used to convert the tallied evidence to the predicted probabilities demonstrated by the following equation (2):

$$y = \text{softmax}(\text{evidence}) \qquad (2)$$

Where y is the predicted probabilities where the softmax function itself can be represented with equation (3) below.

$$\text{softmax}(x)_i = \text{normalize}(\exp(x_i)) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \qquad (3)$$

The softmax exponentiates its input first then normalizes them. This means that more unit of evidence increases the weight given to any hypothesis significantly while one less unit of evidence does the opposite. Since no hypothesis has zero or negative weights, given that there is significant evidence that an input is a specific class, the certainty will be amplified by the softmax function. The weights were then normalized so that it adds up to one to form a valid probability distribution.

For each output, softmax was applied to the weighted sum of input x added with a bias. This can be represented as equation (4) below.

$$y = \text{softmax}(Wx + b) \qquad (4)$$

With increased training size, the weights will change as well, enabling the model to classify certain classes with higher certainty.

3. Experiments

3.1. Regression Implementation

To start the implementation of the softmax regression, TensorFlow was first imported as *tf*. A symbolic variable, x, was used as a *placeholder*, to be given an input when TensorFlow was ran. The placeholder will be able to take in any number of MNIST images representaed as a 2-D tensor of floating-point numbers with shape [None, 784]. None means that the first dimension can be of any length and 784 is the dimension of the flatted vector of the images. TensorFlow's *Variable* is a tensor that can be modified and is in TensorFlow's interacting operations. The weight, W, and bias, b, was created as a *Variable* by giving *tf.Variable* the initial value of each *Variable*. They are both initialized

as tensors full of zeros as the value will get modified as the model gets trained.

The model was implemented with the following line of code:

y = tf.nn.softmax(tf.matmul(x, W) + b)

where matmul is the multiplication function of TensorFlow. The softmax was then applied by using tf.nn.softmax to the product of W and x in addition with b.

3.2. Training

The cost or loss of the model represents how far the model is from the desired outcome. For better results, it is important to minimize the error in the model obtaining a smaller loss value. To calculate the loss of a model, a function called "cross-entropy" was used represented as the equation (5) below.

$$H_{y'}(y) = \sum_i y_i' \log(y_i) \quad (5)$$

Where y is the predicted probability distribution, and y' is the true distribution represented as a one-hot vector with labels of the digit. This was then implemented by first adding a new placeholder to represent the correct answers y_i'. Then the cross entropy was calculated by implementing the above equation where the mean over all the examples in the batch is going to be calculated. TensorFlow uses the backpropagation algorithm to determine how the different variables affect the losses that wanted to be minimized. A gradient descent optimizer algorithm was then applied to modify the variables to reduce such losses. Each time TensorFlow was run, it does a step of gradient descent training, which tweaked the variables slightly to reduce the losses. Hence, the more frequent the training step was performed, the more times the variables got modified to reduce the losses.

3.3. Evaluation

To determine how well the model has performed, a check was performed to determine if the predictions matches the truth. This was done by using tf.argmax, a function that gives the index of the highest entry in a tensor along a given axis. For example, tf.argmax(y, 1) is the label the model thinks is the right class for a given input while tf.argmax(y_, 1) is the actual correct label. The check was performed by applying the following line of equation code:

correction_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))

which in turn gives a list of Booleans converted to 1's (True) and 0's (False). The average value was casted as a floating point number and the average value was taken.

With the simple implementation of the softmax regression and running the training step 1000 times, an accuracy of 91.86%. The number of iterations affects the accuracy as well. An accuracy of 92.23% was obtained the training step was run 10,000 times.

3.4. Multilayer Convolutional Network

However, obtaining merely 92% on the MNIST dataset was below the known standard. Hence, one way to obtain a more accurate representation was to use a convolutional neural network. ReLU neurons were used and they were initialized with a slightly positive bias to avoid "dead neurons". To achieve better results, a three layer convolutional network model was used. The convolutions used a stride of one with zero padding so that the output would have the same size as the input. The pooling used was max pooling over 2x2 blocks.

The first convolutional layer consist of convolution followed by max pooling. It computed 32 features for each 5x5 patch and its weight tensor have a shape of [5, 5, 1, 32]. Dimensions one and two are the patch size, the third is the number of input channel, and the last is the number of output channels. To apply this, the input, x, was first reshaped to a 4d tensor by applying the following code:

x_image = tf.reshape(x, [-1, 28, 28, 1])

where the second and third dimensions are the image width and height. It is the convolve with a weight tensor, added with the bias, and applied with the ReLU function, before performing the max pool. By this point, the image was reduced to the size of 14x14.

The second convolutional layer have 64 features for each 5x5 patch. This further reduced the size of the image to 7x7 and a fully-connected layer with 1024 neurons were added to allow processing of the entire image. The tensor was then reshaped from the pooling layer into a batch of vectors, multiplied with the weight, added with the bias, before a ReLU was applied. Dropout was also applied to reduce overfitting. However, since the neural network is not too deep, the absence of a dropout will not lead to significant changes on the final result. A final layer, similar to the previous model of softmax regression, was then added.

The model was then ran for 20,000 training iterations and an accuracy of 99.27% was obtained.

4. Conclusion

In conclusion, this experiment was a great introduction to TensorFlow, some of its standard libraries, implementation, and terms. It has also

introduced some familiarity with the MNIST dataset and the basics of how to train a model of neural network, both deep, or simply just by modifying the weights and loss values, to classify a certain image to its corresponding class. It was discovered that by increasing the number of iterations, the accuracy of classification may improve slightly. Although the simple model was able to give an accuracy of close to 92%, this value is a pretty low score in terms of accuracy using an MNIST dataset. Hence, a three-layered convolutional network was used to improve the results. This yielded a much better result of around 99.27%.

5. References

[1] "MNIST for ML Beginners | TensorFlow, www.tensorflow.org/versions/r1.1/get_started/mnist/beginners.

[2] "Deep MNIST for Experts | TensorFlow, https://www.tensorflow.org/versions/r1.1/get_started/mnist/pros.

[3] Murnane, Kevin. "What Is Deep Learning And How Is It Useful?" Forbes, Forbes Magazine, 5 Apr. 2016, https://www.forbes.com/sites/kevinmurnane/2016/04/01/what-is-deep-learning-and-how-is-it-useful/#489c92c1d547.

[4] "THE MNIST DATABASE." MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges, yann.lecun.com/exdb/mnist/.

[5] Collis, Jaron. "Glossary of Deep Learning: Bias – Deeper Learning – Medium." Medium, Deeper Learning, 14 Apr. 2017, medium.com/deeper-learning/glossary-of-deep-learning-bias-cf49d9c895e2.