Exploration of Neural Network

By Dexian Chen

Abstract

The report intends to explore the application of neural network in machine learning by applying it to a Fashion MNIST dataset to classify items in each image. Deep learning is nowadays a very easily accessible and powerful tools on training data and testing because more and more packages are developed and shared in internet.

Introduction and Overview

The project consists of two parts to explore the neural network in deep learning. Part one is about applying fully connected neural network on identifying difference fashion items in the dataset Fashion MNIST. Part two repeats what part one does but also explores convolutional neural network. Finally, the there will be a comparison between part 1 and part 2, including the results like accuracy and loss.

Theoretical Background

Artificial Neural Network is inspired by biological neural networks that constitutes animal brains. For example, our brains can recognize cats and dogs by manually labeling them, which is the "learning" process. After reasonable number of examples, we can identify cat and dog. This is also how we train or teach the machines to classify image in machine learning or deep learning.

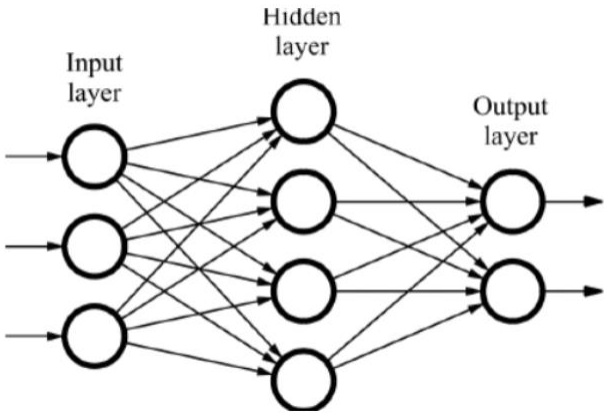


Figure 1: Artificial Neural Network Visualization

In many deep learning packages, we use TensorFlow in python for this project. **TensorFlow** is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It was developed by the Google Brain team. I'll only introduce a few of the most important algorithms related to our code or mostly commonly used, because TensorFlow is complicated and contains too many algorithms.

1. **Linear Regression**: Assume that we have a data set

$$(x_1, y_1), (x_2, y_2), \ldots\ldots, (x_N, y_N) \qquad (1)$$

Then we will have a model of x as an independent variable and y as a dependent variable

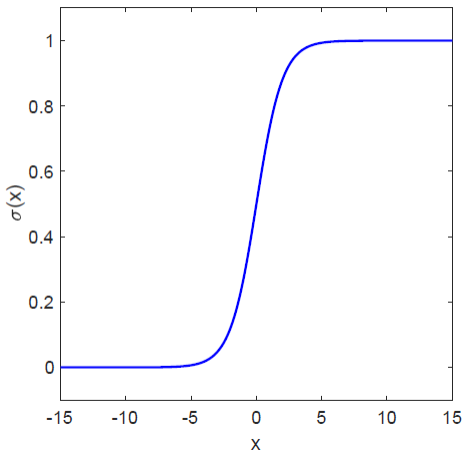
$$y = mx + b \qquad (2)$$

The loss function or objective function is what we use to determine which parameter best fits the data. The most common choice is mean square error(root mean squared error). If we denote $\hat{y} = mx + b$. where $\hat{y}$ is the corresponding point in the model for each independent x. Then the mean squared error is $\frac{1}{n}\sum_{j=1}^{N}(y_j - \hat{y}_j)^2$. The parameter m and b are what we find to minimize this error.

2. **Logistic Regression**: Assume that we have the same data as the above, but each y is either 0 or 1. we use a function as the model that will stay between 0 and 1, the logistic function (or sigmoid):

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad (3)$$

The function has a shape as the following:



The loss function is very different:

$$\frac{1}{N}\sum_{j=1}^{N}[\, y_j \ln(p_j) + (1-y_j) \ln(1-p_j) \,] \qquad (3)$$

where p is the probability of being correct. This is also called log likelihood, and we use log for technical reasons. However, we try to minimize the **negative of function (3)**, which is the called **cross-entropy loss** or **log loss**.

Algorithm Implementation and Development

For both part one and part two, we firstly import packages we need for our code numpy, tensorflow, matplotlib.pyplot, pandas and sklearn.metrics. Then we load our data from the fashion_mnist data set. Before we start to train our data, we need to prepare the data for training by extracting 5000 for validation and the rest for training in the "X" dataset.(See Appendix B) Also, we make them to be floating point numbers between 0 and 1 by dividing each by 255.0. Then I import partial from the functools, and then make my Sequential model. The models in part 1 and part 2 are different.

Part 1: I flatten the layers and use fully connected layers with an input size 28x28. Then I put 4 hidden layers of decreasing size, and the difference between two hidden layers will become smaller when it's deeper. Then I output layer is in a size of 10 because we have ten items with an **activation** function of "**softmax**". For each hidden layer, I keep everything as default but activation and kernel_regularizer. I used "relu" as activation and tf.keras.regularizers.l2 as 0.0005. Finally, I compile my model with a **learning rate** of 0.0001, set my mini patch, epochs, to be 30 for avoid over training, train my data and test it to get the result.

Part 2: I use an architecture similar to **LeNet-5** (see reference) for my Sequential, but I insert two hidden layers right after my first input layers to improve its performance. The first one is MaxPooling2D and another one is a convolutional layer with 16 filters and kernel size of 3. Also, I revise all of the kernel size to 3, and set my dense layer's kernel regularizer equal to **tf.keras.regularizers.l2(0.0005)**. I set my learning rate to be 0.0005 and epochs to be 32. Lastly, I train the data and evaluate the accuracy.

Computational Results

Part 1: I get an accuracy between 89% and 90% by trying different hyperparameters in the fully connected neural network. A layer width over 300 is likely to reduce the accuracy.

Part 2: I get an accuracy of around 91% with the convolutional layers in revised LeNet-5 architecture. Reducing kernel size from 5 to 3 increase my accuracy while having too many filters will decrease the accuracy. The effect of strides is vague but applying MaxPooling2D as the first pool of the model would help increase accuracy. This idea is from VGGnet by my searching. I find no clear change by adjusting the padding options, pool size or the strides so I set it as default.

Comparison: Compared with part 2, It's very hard for me to get a even higher accuracy with all fully connected layers in part 1 because I'm somehow blindly trying out different hyper parameters and there's some limitation with only using fully connected layers. Once I use convolutional layers mixed with fully connected layers, I can boost up my accuracy very easily. By trying out many times in both part 1 and part 2, I find in general:

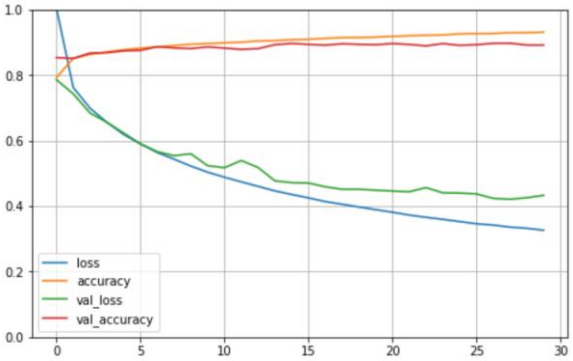
1. more layers will decrease the accuracy, so I decide to stick to the LeNet-5 architecture, because it's simply and easy to use. Furthermore, it's friendly to normal computers to run. Another reason I give up many of the other net like Alexnet is because it has too many layers in this situation and I've already found that more layers lead to lower accuracy.

2. Adjusting the learning rate and epochs are extremely important to avoid overtraining and get a good result. If the epochs or the learning rate is too big, the accuracy will likely meet a bound. Learning rate and epochs are somehow negatively related. For example, if we fix other parameters and lower the learning rate, we will very likely need to increase our epochs for training reasonable times.

3. Regularization parameter being lower than 0.0001 will reduce accuracy

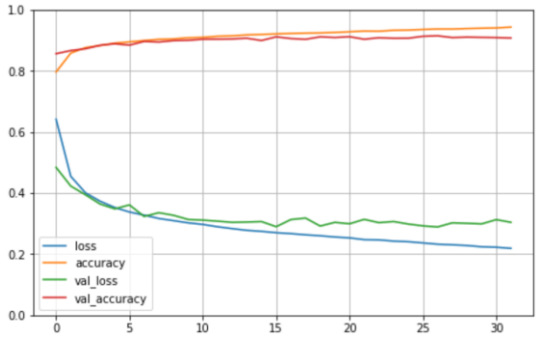
4. Activation function of ELU and RELU are very similar and switching them won't change the accuracy. Hyperbolic tangent works the best for convolutional layers.

The above is some of the common finding for both parts. I'm satisfied with both of the result because the validation accuracy is close enough to the accuracy and loss line is relatively smooth, instead of sliding too fast.



	0	1	2	3	4	5	6	7	8	9
0	899	1	6	23	4	1	62	0	4	0
1	5	969	0	20	3	0	3	0	0	0
2	27	0	767	21	116	0	69	0	0	0
3	23	5	7	933	16	0	15	0	1	0
4	2	1	64	53	827	0	52	0	1	0
5	0	0	0	1	0	973	0	11	0	15
6	168	2	64	34	78	0	650	0	4	0
7	0	0	0	0	0	30	0	952	0	18
8	11	0	4	7	6	4	9	3	956	0
9	0	0	0	0	0	6	1	36	0	957

Figure 2: Accuracy Line visualization & Confusion Matrix in part 1



	0	1	2	3	4	5	6	7	8	9
0	860	2	12	19	8	1	90	0	8	0
1	2	979	2	12	1	0	4	0	0	0
2	16	0	800	10	70	0	103	0	1	0
3	16	4	6	921	25	0	27	0	1	0
4	1	0	21	29	897	1	50	0	1	0
5	0	0	0	0	0	990	0	3	0	7
6	111	2	30	27	96	0	728	0	6	0
7	0	0	0	0	0	26	0	929	0	45
8	3	1	1	5	2	1	5	2	977	3
9	0	0	0	0	1	5	0	27	0	967

Figure 3: Accuracy Line visualization & Confusion Matrix in part 2

Summary and Conclusions

In summary, from my discovery, structure of the net, learning rate, epochs are extremely important for getting a good result with some many times of adjustment or “guessing”. Only depending on a simple architecture of fully connected layers makes it hard to reach high accuracy, so it’s better to mix both convolutional and fully connected layers. LeNet-5 is feasible on deep learning for relatively small dataset, and user friendly to people who have no access to powerful computers. Overall, relatively simple net can get a decent accuracy on classification within fashion MNIST dataset.

Reference

Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data by J. Nathan Kutz.

https://en.wikipedia.org/wiki/Artificial_neural_network

https://medium.com/@pechyonkin/key-deep-learning-architectures-lenet-5-6fc3c59e6f4

Appendix A

import: import packages from python

model.predict: Generates output predictions for the input samples.

model.fit: Trains the model for a fixed number of epochs (iterations on a dataset).

model.evaluate: Returns the loss value & metrics values for the model in test mode.

tf.kera.dataset: load TensorFlow dataset

tf.keras.layers.Dense: Fully connect layer. Just your regular densely connected NN layer.

tf.keras.Conv2D: 2D convolutional layer. (e.g. spatial convolution over images)

tf.keras.MaxPooling2D: Max pooling operation for spatial data

tf.keras.AveragePooling2D: Average pooling operation for spatial data

Appendix B

Part 1:

```
import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.metrics import confusion_matrix

fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

X_valid = X_train_full[:5000] / 255.0

X_train = X_train_full[5000:] / 255.0

X_test = X_test / 255.0



y_valid = y_train_full[:5000]

y_train = y_train_full[5000:]
```

```
from functools import partial


my_dense_layer = partial(tf.keras.layers.Dense, activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(0.0005))


model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    my_dense_layer(300),
    my_dense_layer(200),
    my_dense_layer(150),
    my_dense_layer(50),
    my_dense_layer(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid,y_valid))
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
```

```
plt.gca().set_ylim(0,1)

plt.show()

y_pred = model.predict_classes(X_train)

conf_train = confusion_matrix(y_train, y_pred)

print(conf_train)

model.evaluate(X_test,y_test)

y_pred = model.predict_classes(X_test)

conf_test = confusion_matrix(y_test, y_pred)

print(conf_test)
```

Part 2:

```
import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.metrics import confusion_matrix

fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

```
X_valid = X_train_full[:5000] / 255.0

X_train = X_train_full[5000:] / 255.0

X_test = X_test / 255.0



y_valid = y_train_full[:5000]

y_train = y_train_full[5000:]



X_train = X_train[..., np.newaxis]

X_valid = X_valid[..., np.newaxis]

X_test = X_test[..., np.newaxis]

from functools import partial



my_dense_layer = partial(tf.keras.layers.Dense, activation="tanh",

kernel_regularizer=tf.keras.regularizers.l2(0.0005))

my_conv_layer = partial(tf.keras.layers.Conv2D, activation="tanh", padding="valid")



model = tf.keras.models.Sequential([

    my_conv_layer(6,3,padding="same",input_shape=[28,28,1]),

    tf.keras.layers.MaxPooling2D(2),
```

```
    my_conv_layer(16,3),
    my_conv_layer(16,3),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(120,3),
    tf.keras.layers.Flatten(),
    my_dense_layer(84),
    my_dense_layer(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005),
              metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=32, validation_data=(X_valid,y_valid))
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()
y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
```

```
print(conf_train)

model.evaluate(X_test,y_test)

y_pred = model.predict_classes(X_test)

conf_test = confusion_matrix(y_test, y_pred)

print(conf_test)

fig, ax = plt.subplots()



# hide axes

fig.patch.set_visible(False)

ax.axis('off')

ax.axis('tight')



# create table and save to file

df = pd.DataFrame(conf_test)

ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10), loc='center',

cellLoc='center')

fig.tight_layout()

plt.savefig('conf_mat2.pdf')
```