MBAN6500A_assignment2

October 3, 2019

1 AI 1 - Assignment 2

This assignment requires you to install Keras and Tensorflow. Keras is a high-level Deep Learning API, written in Python using TensorFlow, CNTK, or Theano as back-ends. Here, we will use Tensorflow as back-end.

This assignment is divided in two parts. In the first part you will learn about Keras with the help of the example below and the Keras documentation. In the second part, you will practise training a Deep Learning model.

1.1 How to submit

Submit by uploading this notebook to Canvas. It should include **plots**, **results** and **code** showing how the results were genereated. Remember to name your file(s) appropriately. **Deadline** is on 17:30 on November 14, 2019.

1.2 Installation

Instructions can be found here: * Keras * Tensorflow

Since Tensorflow 2.0, Keras is included in Tensorflow and will be automatically installed with Tensorflow. It can be accessed as `tensorflow.keras`

I recommend using `pip`. For Tensorflow is it sufficient to install the CPU version. The GPU version requires a good workstation with high-end Nvidia GPU(s), and it is not necessary for this tutorial.

If you're using a virtualenv:

```
pip3 install tensorflow
```

Add `sudo` for a systemwide installation (i.e. no `virtualenv`).

```
sudo pip3 install tensorflow
```

Make sure that you have `sklearn`, `matplotlib` and `numpy` installed, too.

1.3 Part 1 - understand a model

1.3.1 Optimizers

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater than zero. The goal of training a model is to find a set of weights and biases (i.e. parameters) that have, on average, a low loss across all examples. The term cost is used interchangably with loss. See the loss section in the Keras documentation for a list and descriptions of what is available.

Figure. Left: high loss and right: low loss.

The optimizer is the algorithm used to minimize the loss/cost. Optimizers in neural networks work by finding the gradient/derivative of the loss with respect to the parameters (i.e. the weights). "Gradient" is the correct term since a we are looking at multi-dimensional systems (i.e. many parameters), however, the terms are often used interchangably. For those who didn't take multivariate calculus, just think of the gradient as a derivative. The derivative of the loss with respect to a parameters tells us how much the loss changes when we nudge a weight up or down. So, by knowing how a given parameter affects the loss the optimizer can change it so as to decrease the loss. The various optimizers differ in how they change the weights.

Mini-overview over popular optimizers

- **Stochastic Gradient Descent (SGD)**. This is the most basic and easy to understand optimizer. It updates the weights in the negative direction of the gradient by taking the average gradient of mini-batch of data (e.g. 20-50 exemplars) in each step. Vanilla SGD only has one hyper-parameter, the learning rate.
- **Momentum**. This optimizer "gains speed" when the gradient has pointed in the same direction for several consecutive updates. That is, it gains momentum. It does this by accumulating an exponentially decaying moving average of past gradients. The step size depends on how large and aligned the sequence of gradients are. The most important hyper-parameter is alpha and common values are 0.5 and 0.9.
- **Nesterov Momentum**. This is a modification of the standard momentum optimizer.
- **AdaGrad**. This optimizer Ada-ptively sets the learning rate depending on the steepness/magnitude of the Grad-ients. This is done so that weights with big gradients get a smaller effective learning rate, and weights with small gradients will get a greater effective learning rate. The result is quicker progress in the more gently sloped directions of the weight space and a slowdown in stepp regions.
- **RMSProp**. This is modification of AdaGrad, where the accumulated gradient decays, that is, the influence of previous gradients gradually decreases.
- **Adam**. The name comes from "adaptive moments", and it is a combination of RMSProp and momentum. It has several hyper-parameters.

The above list just gives a quick overview of some of the most common. However, old optimizers are constantly improved and new are developed. SGD and momentum are most basic and easiest to understand and implement. They are still in use, but the more advanced optimizers tend to be better for practical use. Which one to use is generally an emperical question depending on both the data and the model.

For a more complete overview of optimization algorithms see this comparison, and to see what is available in Keras, see the optimizer section of the documentation.

See the images below for a comparison of optimizers in a 2D space (NAG: Nesterov accelerated gradient, Adadelta: an extension of AdaGrad).

[ ]:
```
# imports
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
# for the random seed
import tensorflow as tf

# set the random seeds to get reproducible results
np.random.seed(1)
tf.set_random_seed(2)

# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
X, y = X[:1000], y[:1000]
X = X.reshape(X.shape[0], 28, 28, 1)
# Normalize
X = X / 255.
# number of unique classes
num_classes = len(np.unique(y))
y = y.astype(int)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2,␣
 ↪random_state=1)

num_tot = y.shape[0]
num_train = y_train.shape[0]
num_test = y_test.shape[0]

y_oh = np.zeros((num_tot, num_classes))
y_oh[range(num_tot), y] = 1

y_oh_train = np.zeros((num_train, num_classes))
y_oh_train[range(num_train), y_train] = 1

y_oh_test = np.zeros((num_test, num_classes))
y_oh_test[range(num_test), y_test] = 1
```

1.3.2 Question 1

The data set

Plot a three examples from the data set. * What type of data are in the data set?

`<*answer here*>`

- What does the line `X = X.reshape(X.shape[0], 28, 28, 1)` do?

Look at how the encoding of the targets (i.e. `y`) is changed. E.g. the lines

```
y_oh = np.zeros((num_tot, num_classes))
y_oh[range(num_tot), y] = 1
```

Print out a few rows of `y` next to `y_oh`. * What is the relationship between `y` and `y_oh`?

`<*answer here*>`

- What is the type of encoding in `y_oh` called and why is it used?

  <answer here>

- Plot three data examples in the same figure and set the correct label as title.
  - It should be possible to see what the data represent.

1.3.3 Question 2

The model

Below is some code for bulding and training a model with Keras. * What type of network is implemented below? I.e. a normal MLP, RNN, CNN, Logistic Regression…?

`<*answer here*>`

- What does `Dropout()` do?

  <answer here>

- Which type of activation function is used for the hidden layers?

  <answer here>

- Which type of activation function is used for the output layer?

  <answer here>

- Why are two different activation functions used?

  <answer here>

- What optimizer is used in the model below?

  <answer here>

- How often are the weights updated (i.e. after how many data examples)?

  <answer here>

- What loss function is used?

  <answer here>

- How many parameters (i.e. weights and biases, NOT hyper-parameters) does the model have?

  <answer here>

[ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.optimizers import SGD


model = Sequential()

model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.))

model.add(Conv2D(32, (3, 3), activation='relu'))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.))

model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=sgd)

# Train the model
model.fit(X_train, y_oh_train, batch_size=32, epochs=60)

# Evaluate performance
test_loss = model.evaluate(X_test, y_oh_test, batch_size=32)

predictions = model.predict(X_test, batch_size=32)
predictions = np.argmax(predictions, axis=1) # change encoding again
print('Accuracy:', (predictions == y_test).sum() / predictions.shape[0])
```

1.4 Part 2 - train a model

A model's performance depends on many factors apart from the model architecture (e.g. type and number of layers) and the dataset. Here you will get to explore some of the factors that affect model performance. Much of the skill in training deep learning models lies in quickly finding good

values/options for these choises.

In order to observe the learning process it is best to compare the training set loss with the loss on the test set. How to visualize these variables with Keras is described under Training history visualization in the documentation.

You will explore the effect of 1) optimizer, 2) training duration, and 3) dropout (see the question above).

When training, an **epoch** is one pass through the full training set.

1.4.1 Question 3

- **Vizualize the training**. Use the model above to observe the training process. Train it for 150 epochs and then plot both "loss" and "val_loss" (i.e. loss on the valiadtion set, here the terms "validation set" and "test set" are used interchangably, but this is not always true). What is the optimal number of epochs for minimizing the test set loss?
  - Remember to first reset the weights (`model.reset_states()`), otherwise the training just continues from where it was stopped earlier.
- **Optimizer**. Select three different optimizers and for each find the close-to-optimal hyper-parameter(s). In your answer, include a) your three choises, b) best hyper-parameters for each of the three optimizers and, c) the code that produced the results.
  - *NOTE* that how long the training takes varies with optimizer. I.e., make sure that the model is trained for long enough to reach optimal performance.
- **Dropout**. Use the best optimizer and do hyper-parameter seach and find the best value for `Dropout()`.
- **Best model**. Combine the what you learned from the above three questions to build the best model. How much better is it than the worst and average models?

  <answer here>
- **Results on the test set**. When doing this search for good model configuration/hyper-parameter values, the data set was split into *two* parts: a training set and a test set (the term "validation" was used interchangably wiht "test"). For your final model, is the performance (i.e. accuracy) on the test set representative for the performance one would expect on a previously unseen data set (drawn from the same distribution)? Why?

  <answer here>

HINT: For ideas about hyper-parameter tuning, take a look at the strategies described in the sklearn documentation under model selection

Good luck!

[ ]: