TP03-LogisticRegression

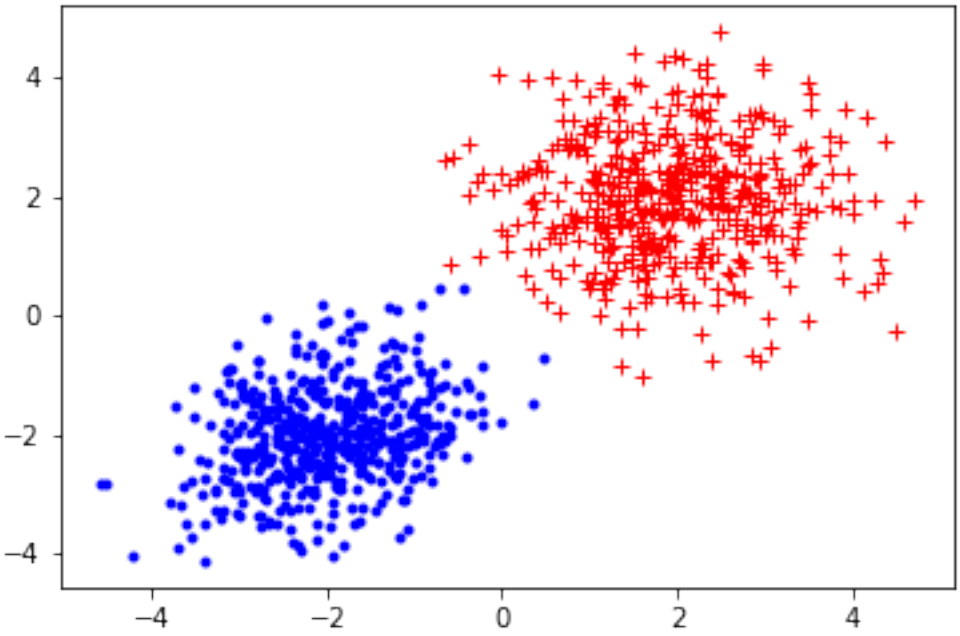
September 18, 2019

1 Logistic Regression

The following script generate a fake classification dataset.

[1]:
```
import numpy as np
import tensorflow as tf

import matplotlib.pyplot as plt
%matplotlib inline
import datetime as dt

def generate_all_dataset():
    # --- Fake dataset ---

    np.random.seed(0)

    ntrain = 1000
    nvalid = 100
    ntest = 100

    mupos = np.array([2., 2.])
    sigmapos = np.array([[1., 0.], [0., 1.]])
    muneg = np.array([-2., -2.])
    sigmaneg = np.array([[.7, .2], [.2, .7]])

    def generate_a_dataset(mupos, sigmapos, muneg, sigmaneg, n):
        npos = int(n/2)
        nneg = n - npos

        Xpos = np.random.multivariate_normal(mupos, sigmapos, npos)
        Ypos = np.stack((np.ones((npos,)), np.zeros((npos,))), axis=1)

        Xneg = np.random.multivariate_normal(muneg, sigmaneg, nneg)
        Yneg = np.stack((np.zeros((nneg,)), np.ones((nneg,))), axis=1)

        X, Y = np.concatenate((Xpos, Xneg)), np.concatenate((Ypos, Yneg))
```

```
        idx = np.arange(n)
        np.random.shuffle(idx)
        X, Y = X[idx], Y[idx]

        return np.array(X, dtype='float32'), np.array(Y, dtype='float32')

    Xtrain, Ytrain = generate_a_dataset(
        mupos, sigmapos, muneg, sigmaneg, ntrain)
    Xvalid, Yvalid = generate_a_dataset(
        mupos, sigmapos, muneg, sigmaneg, nvalid)
    Xtest, Ytest = generate_a_dataset(mupos, sigmapos, muneg, sigmaneg, ntest)

    return (Xtrain, Ytrain), (Xvalid, Yvalid), (Xtest, Ytest)


def plot_dataset(X, Y):
    plt.figure()
    idpos, = np.nonzero(Y[:, 0] == 1.)
    idneg, = np.nonzero(Y[:, 1] == 1.)
    plt.plot(X[idpos, 0], X[idpos, 1], 'r+')
    plt.plot(X[idneg, 0], X[idneg, 1], 'b.')
    plt.show()


def demo(trainset, validset, testset):
    plot_dataset(*trainset)


demo(*generate_all_dataset())
```



1.1 Exercise

Inspired by the precedent object oriented verision of the linear regression, write a logistic regression model:

$$\hat{y} = softmax(< x, A > +b)$$

It can be learnt by the cross-entropy loss:

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

In fact, that's a **one layer perceptron** !

You will have to use the following operations: - `tf.matmul` - `tf.nn.softmax` - `tf.losses.softmax_cross_entropy`

Please note that the `softmax_cross_entropy` loss takes as input the logits, i.e. the output of the linear model before the softmax, and that both `softmax` and `softmax_cross_entropy` need a two column input (one per class).

Guidelines:

1) Copy the code of `LinearRegressionV3` class and rename the class to `LogisticRegressionV1`
2) Modify `_build_network` method to have a logistic regression model and a cross-entropy loss
3) Add a `_compute_pred_loss(self, set_iterator_init, nbatches)` method that return the prediction and the loss
4) Modify `train` method by using `_compute_pred_loss` on the test set, and make `train` return the prediction of the test set

[2]:
```
# object oriented verision of the linear regression
class LinearRegression:

    def __init__(self, dtype, ninputs, noutputs,
                 learning_rate=5e-3, training_epochs=100, batchsize=50):

        self.nc = ninputs
        self.no = noutputs

        self.dt_shapes = (tf.TensorShape((None, ninputs)),
                          tf.TensorShape((None, noutputs)))
        self.dt_types = (dtype, dtype)

        self.learning_rate = learning_rate
        self.training_epochs = training_epochs
        self.batchsize = batchsize

        self._build_network()

        # Global variable initializer
        self.init_global = tf.initializers.global_variables()

    def _build_network(self):
        # Create ONLY ONE iterator base on types and shapes of one of the␣
↪dataset
        # both dataset should have the same types and shapes...
        self.dataset_iterator = tf.data.Iterator.from_structure(
            self.dt_types, self.dt_shapes)

        # Placeholders from the dataset iterator
        x, y = self.dataset_iterator.get_next()

        # Linear regression parameters
        self.A = tf.Variable(tf.zeros([self.nc, self.no]))
        self.b = tf.Variable(tf.zeros([self.no]))
        # All parameters are gathered into var_list
        var_list = [self.A, self.b]

        # Actual linear regression
        self.pred = tf.matmul(x, self.A) + self.b

        # Model loss
        self.loss = tf.losses.mean_squared_error(y, self.pred)

        self.optimizer = tf.train.GradientDescentOptimizer(
            self.learning_rate).minimize(self.loss, var_list=var_list)
```

```
    def _prepareset(self, dataset, shuffle=True):
        if shuffle:
            dataset = dataset.shuffle(buffer_size=1000)
        dataset = dataset.batch(self.batchsize)
        return dataset

    def _compute_loss(self, set_iterator_init, nbatches):
        self.session.run(set_iterator_init)
        lossval = 0.
        for b in range(nbatches):
            batch_lossval, = self.session.run([self.loss])
            lossval += batch_lossval
        lossval /= nbatches
        return lossval

    def _compute_gradient_step(self, set_iterator_init, nbatches):
        self.session.run(set_iterator_init)
        lossval = 0.
        for b in range(nbatches):
            _, batch_lossval, = self.session.run([self.optimizer, self.loss])
            lossval += batch_lossval
        lossval /= nbatches
        return lossval

    def train(self, trainset, validset, testset):

        (Xtrain, Ytrain) = trainset
        (Xvalid, Yvalid) = validset
        (Xtest, Ytest) = testset

        # --- Linear Regression ---

        ntrain, _ = Xtrain.shape
        ntest, _ = Xtest.shape
        nvalid, _ = Xvalid.shape

        trainset = self._prepareset(
            tf.data.Dataset.from_tensor_slices((Xtrain, Ytrain)))
        testset = self._prepareset(
            tf.data.Dataset.from_tensor_slices((Xtest, Ytest)), shuffle=False)
        validset = self._prepareset(
            tf.data.Dataset.from_tensor_slices((Xvalid, Yvalid)), shuffle=False)

        ntrainbatches = int(np.ceil(ntrain/self.batchsize))
        ntestbatches = int(np.ceil(ntest/self.batchsize))
        nvalidbatches = int(np.ceil(nvalid/self.batchsize))
```

```
        # Create one initializer per dataset
        training_init_op = self.dataset_iterator.make_initializer(trainset)
        test_init_op = self.dataset_iterator.make_initializer(testset)
        validation_init_op = self.dataset_iterator .make_initializer(validset)


        with tf.Session() as self.session:

            # We call the initialization of A and b
            self.session.run(self.init_global)

            # We compute the train and validation loss
            # Note that you just have to change the feed_dict to change the set

            trainloss = self._compute_loss(training_init_op, ntrainbatches)
            validationloss = self._compute_loss(
                validation_init_op, nvalidbatches)
            print("Init\t\t train loss %f\t valid loss %f" %
                  (trainloss, validationloss))

            # We cycle on epochs
            for epoch in range(self.training_epochs):

                trainloss = self._compute_gradient_step(
                    training_init_op, ntrainbatches)
                validationloss = self._compute_loss(
                    validation_init_op,  nvalidbatches)
                print("Epoch %03d\t train loss %f\t valid loss %f" %
                      (epoch+1, trainloss, validationloss))

            # We compute the test loss
            testloss = self._compute_loss(test_init_op, ntestbatches)
            print("Test loss %f" % (testloss,))

            # Found parameters
            Aval, bval = self.session.run([self.A, self.b])
            print("Estimated A\n", Aval)
            print('Estimated b\n', bval)
        # Here session is closed automatically
```

1.2 Metrics

In the precedent script, each time you compute the loss of batch, the result is copied from the memory of the computation engine back to the computer memory. Moreover, it is actually the computer which is doing the averaging.

You can faster the loss loop/computation by using `metrics`: that's a trick to do all the work inside the memory of the computation engine.

A metric contain an internal state that will be update for each batch. When all the examples have been seen , the last internal state is used to compute the actual metrics.

For example, a mean metrics consists in a `total` and `count` internal states. At each update, the targeted tensor (for example the loss) is added to `total` and `count` is incremented by one. At the end, `total` is divided by `count` to give the actual mean metric.

1.2.1 How to use a metric ?

1) Create a metric on targeted tensor(s) (like the loss), you will be provide by two tensors in return `actualmetric`, `metricupdate`
2) Initialize the internal state of the metric
3) Update the internal state of the metric running `metricupdate` for each batch inside a loop.
4) Get the metric value by running `actualmetric`.

1.2.2 Example

Here is how you can use `tf.metrics.mean` for averaging the loss over all the batches to compute a loss

```
class LogisticRegressionV2(LogisticRegressionV1):

    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

        # Initializers of metrics,
        # should be instantiate after the network is built
        self.init_local = tf.initializers.local_variables()

    def _build_network(self):
        super()._build_network()

        # Metrics
        self.lossmetric, self.lossmetric_update = tf.metrics.mean(self.loss)

    def _compute_loss(self, set_iterator_init, nbatches):
        # Initialize all the metrics
        self.session.run(self.init_local)
        # Initialize the iterator
        self.session.run(set_iterator_init)
        # Loop
        for b in range(nbatches):
            self.session.run(self.lossmetric_update)
        lossval = self.session.run(self.lossmetric)
        return lossval
```

1.2.3 Exercise

Complete the class `LogisticRegressionV2` by methods `_compute_gradient_step` and `_compute_pred_loss` that compute the loss through a metric.

1.2.4 Tracking the accuracy

Metrics are also useful to track indicators other than the loss. For example in our classification task we can look at the accuracy of the model, i.e. the matches between prediction and labels.

Here is a code for computing the loss and the accuracy at the same time:

```
class LogisticRegressionV3(LogisticRegressionV2):

    def _build_network(self):
        super()._build_network()

        self.accuracymetric,  self.accuracymetric_update = tf.metrics.accuracy(
            self.y, self.pred)

    def _compute_loss(self, set_iterator_init, nbatches):
        # Initialize all the metrics
        self.session.run(self.init_local)
        # Initialize the iterator
        self.session.run(set_iterator_init)
        # Loop
        for b in range(nbatches):
            self.session.run(
                [self.lossmetric_update, self.accuracymetric_update])
        return self.session.run([self.lossmetric, self.accuracymetric])
```

Please refer to this page to see other possible metrics: https://www.tensorflow.org/api_docs/python/tf/metrics

1.2.5 Excercice

Complete the class `LogisticRegressionV3` in order to have a method `_compute_pred_loss` that compute prediction, loss and accuracy, and a `train` method that displays the accuracy. Use a low learning rate (1e-10) if you want to see the evolution of accuracy as it is an easy dataset.

1.3 Tensorboard

Tensorboard is an easy way to monitor the training of your model. It is a dashboard in a web browser displaying metrics and possibly parameters of your model.

In order to do so you first need to export your metrics to log files. Tensorboard will then explore the log files and show you the different graphs in your browser.

1.3.1 Summaries

Summaries are object that can be evaluated and written to tensorboard log files. You need to wrap your metrics around summaries.

After defining a metric just do:

```
trainlosssummary = tf.summary.scalar("train_loss",lossmetric)
```

You can merge multiple summaries into one :

```
trainlosssummary = tf.summary.scalar("train_loss",lossmetric)
trainaccuracysummary = tf.summary.scalar("train_accuracy",accuracymetric)
trainsummaries = tf.summary.merge([trainlosssummary,trainaccuracysummary ])
```

Now `trainsummaries` contains all the summaries of the trainset.

If you have multiple set (e.g. train and valid), you should create one summary per set, even if they are based on the same metric:

```
trainlosssummary = tf.summary.scalar("train_loss",lossmetric)
trainaccuracysummary = tf.summary.scalar("train_accuracy",accuracymetric)
trainsummaries = tf.summary.merge([trainlosssummary,trainaccuracysummary ])

validationlosssummary = tf.summary.scalar("validation_loss",lossmetric)
validationaccuracysummary = tf.summary.scalar("validation_accuracy",accuracymetric)
validationsummaries = tf.summary.merge([validationlosssummary,validationaccuracysummary ])
```

1.3.2 Writer

A Writer is an object that represent a recorder to a folder.

To instanciate a recorder do inside a session:

```
with tf.Session() as session:
    #[...]
    logdir = "logs/somename/" + dt.datetime.now().strftime("%Y%m%d-%H%M%S")
    writer = tf.summary.FileWriter(logdir)
    #[...]
```

Now at each epoch, you can evaluate the summaries and add them to the recorder:

```
with tf.Session() as session:

    # We call the global initialization
    session.run(init_global)
    # Create the log writer
    logdir="logs/logisticregression/" + dt.datetime.now().strftime("%Y%m%d-%H%M%S")
    writer = tf.summary.FileWriter(logdir)
    #[...]
    for epoch in range(training_epochs):
        # Work on the training set
        #[...]
        # And then
        trainsummariesval = session.run(trainsummaries) # evaluate the summary
        writer.add_summary(trainsummariesval, epoch+1) # record the summary
```

```
        # Work on the validation set
        #[...]
        # And then
        validationsummariesval = session.run(validationsummaries) # evaluate the summary
        writer.add_summary(validationsummariesval,epoch+1) # record the summary
```

1.3.3 Log analyzing

A `logisticregressionlogs` folder will be create where the script is run. It contains log file describing the evolution of the training.

To analyze them, we can actually launch tensorboard with the command inside a bash terminal:

```
tensorboard --logdir=logisticregressionlogs
```

It will automatically launch a web browser where you can monitor the metrics of your model.

Caution ! as summaries and log files cumulate it is highly recommended to restart the python kernel and to erase the log folder before each execution of the script.

1.3.4 Your turn !

1) Create a `LogisticRegressionV4` class derivated from `LogisticRegressionV3`
2) Modify `_build_network`and `train` methods to support file logging.
3) Analyze the log in Tensorboard
4) Create a `LogisticRegression` class (no subclassing) merging all the logistic regression versions.