TP04-MultiLayerPerceptron

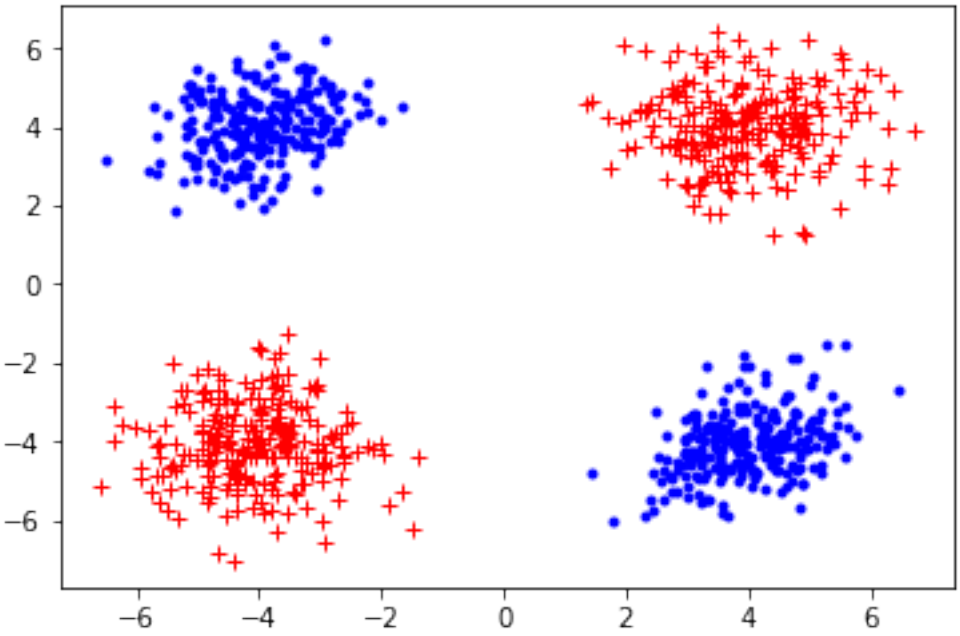
September 18, 2019

0.0.1 Multi Layer Perceptron

The following script generate a *Xor* dataset that can not be well separated by a logistic regression.

[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([4., 4.])
    sigmapos = np.array([[1., 0.], [0., 1.]])
    muneg = np.array([4., -4.])
    sigmaneg = np.array([[.7, .2], [.2, .7]])

    def generate_a_dataset(mupos, sigmapos, muneg, sigmaneg, n):
        nelem = int(n/4)
        npos1 = n-nelem*3
        npos2, nneg1, nneg2 = nelem, nelem, nelem

        Xpos1 = np.random.multivariate_normal(mupos, sigmapos, npos1)
        Ypos1 = np.stack((np.ones((npos1,)), np.zeros((npos1,))), axis=1)
        Xpos2 = np.random.multivariate_normal(-mupos, sigmapos, npos2)
        Ypos2 = np.stack((np.ones((npos2,)), np.zeros((npos2,))), axis=1)

        Xneg1 = np.random.multivariate_normal(muneg, sigmaneg, nneg1)
        Yneg1 = np.stack((np.zeros((nneg1,)), np.ones((nneg1,))), axis=1)
```

```
        Xneg2 = np.random.multivariate_normal(-muneg, sigmaneg, nneg2)
        Yneg2 = np.stack((np.zeros((nneg2,)), np.ones((nneg2,))), axis=1)

        X = np.concatenate((Xpos1, Xpos2, Xneg1, Xneg2))
        Y = np.concatenate((Ypos1, Ypos2, Yneg1, Yneg2))

        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())
```



0.0.2 Model

In order to solve this problem you will need to add a second layer to the logistic regression model :

$$h = sigmoid(< x, A_1 > + b_1)$$
$$\hat{y} = softmax(< h, A_2 > + b_2)$$

where $h \in \mathbb{R}^n$ with n the number of hidden units.
It can still 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 **Multi layer perceptron** !

0.0.3 Tensorboard histograms, parameters and gradients

Usually you don't want to track in the log file all the component of a parameter as it represents many scalars. That is why Tensorflow provides a facility to record histogram and distribrution of parameters:

For example, if you want to track the weights of a layer you can do:

```
asummary = tf.summary.histogram("A_parameter", self.A)
```

In order to also track gradients in Tensorboard, we need tensors representing them. Right now they are hidden inside the optimizer.

They can be obtain by the `compute_gradients` method on an optimizer:

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

`self.grads` is a list of tuple, each tuple consisting in (gradient,parameter).
Then summaries for parameters and gradients can be create by

```
        parametersummaries = []
        gradsummaries = []
        for gradient, parameter in self.grads:
            parametersummaries.append(
                tf.summary.histogram("parameters/" + parameter.name, parameter))
            gradsummaries.append(
                tf.summary.histogram("gradients/" + parameter.name, gradient))
        self.parametersummaries = tf.summary.merge(parametersummaries)
        self.gradsummaries = tf.summary.merge(gradsummaries)
```

Warnings

1) `parameter.name` relies on the `name` argument you give at the creation of parameters:

```
toto = tf.Variable(tf.zeros([5,5]), name="toto")
```

2) `self.gradsummaries` must be executed at the same *run* as `self.optimizer`, and only once per epoch:

```
 _,_,gradsummaryval = self.session.run([self.optimizer, self.lossmetric_update, self.gradsumma
```

0.0.4 Exercise

1) Copy the code of `LogisticRegression` class and rename the class to `MLPV1`
2) Modify the `__init__` method to take a number of hidden units as argument.
3) Modify `_build_network` method in order to have a 2-layer model. You will need `tf.nn.sigmoid`
4) Try your network with the following parameters:

```
nhiddens = 10
learning_rate = 1e-1
training_epochs = 100
batchsize = 5
```

5) Add parameter and gradient summaries
6) Analyze the training trough Tensorboard

What can be said about the evolution of the parameters (look at the distribution and histogram tabs in Tensorboard) ?

[2]:
```
class LogisticRegression:

    def __init__(self, dtype, ninputs, noutputs,
                 learning_rate=1e-1, 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()

        # local and global variable initializers
        self.init_global = tf.initializers.global_variables()
        self.init_local = tf.initializers.local_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
        self.x, self.y = self.dataset_iterator.get_next()

        # Logistic 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 logistic regression
        self.logit = tf.matmul(self.x, self.A) + self.b
        self.output = tf.nn.softmax(self.logit)
        self.pred = self.output > .5

        # Model loss
        self.loss = tf.losses.softmax_cross_entropy(self.y, self.logit)

        self.optimizer = tf.train.GradientDescentOptimizer(
```

```
            self.learning_rate).minimize(self.loss, var_list=var_list)

        # Metrics
        self.lossmetric, self.lossmetric_update = tf.metrics.mean(self.loss)
        self.accuracymetric,  self.accuracymetric_update = tf.metrics.accuracy(
            self.y, self.pred)

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

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

    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):
        # 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])

    def _compute_gradient_step(self, set_iterator_init, nbatches):
        self.session.run(self.init_local)
        self.session.run(set_iterator_init)
        for b in range(nbatches):
            self.session.run([self.optimizer, self.lossmetric_update])
        lossval = self.session.run(self.lossmetric)
        return lossval

    def _compute_pred_loss(self, set_iterator_init, nbatches):
```

```
        self.session.run(self.init_local)
        self.session.run(set_iterator_init)
        predval = None
        for b in range(nbatches):
            batch_pred, _, _ = self.session.run(
                [self.pred, self.lossmetric_update, self.accuracymetric_update])
            if predval is None:
                predval = batch_pred
            else:
                predval = np.concatenate((predval, batch_pred))
        lossval, accval = self.session.run(
            [self.lossmetric, self.accuracymetric])
        return predval, lossval, accval

    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 global initialization
            self.session.run(self.init_global)
```

```
            # Create the log writer
            logdir = "logs/logisticregression/" + dt.datetime.now().
↪strftime("%Y%m%d-%H%M%S")
            writer = tf.summary.FileWriter(logdir)

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

            trainloss, trainacc = self._compute_loss(
                training_init_op, ntrainbatches)
            validationloss, validacc = self._compute_loss(
                validation_init_op, nvalidbatches)

            trainsummariesval = self.session.run(self.trainsummaries)
            writer.add_summary(trainsummariesval, 0)
            validationsummariesval = self.session.run(self.validationsummaries)
            writer.add_summary(validationsummariesval, 0)

            print("Init\t\t train loss %f\t valid loss %f\t valid acc %f" %
                  (trainloss, validationloss, validacc))

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

                trainloss = self._compute_gradient_step(
                    training_init_op, ntrainbatches)
                validationloss, validacc = self._compute_loss(
                    validation_init_op,  nvalidbatches)

                trainsummariesval = self.session.run(self.trainsummaries)
                writer.add_summary(trainsummariesval, epoch+1)
                validationsummariesval = self.session.run(
                    self.validationsummaries)
                writer.add_summary(validationsummariesval, epoch+1)

                print("Epoch %03d\t train loss %f\t valid loss %f\t valid acc␣
↪%f" %
                      (epoch+1, trainloss, validationloss, validacc))

            # We compute the prediction and the test loss
            ytestpred, testloss, testacc = self._compute_pred_loss(
                test_init_op, ntestbatches)
            print("Test loss %f\t test acc %f" % (testloss, testacc))

        # Here session is closed automatically
        return ytestpred
```

0.1 Weight initialization and regularization

A good start to speedup the training is to have a better weight initialization than 0 init.

Here is an example to get a variable initialize with a random normal distribution:

```
toto = tf.get_variable("toto", (5, 5), initializer=tf.random_normal_initializer())
```

Initializer available for Tensorflow are listed here: https://www.tensorflow.org/api_docs/python/tf/initializ

Moreover, weight regularization can prevent the network to be stuck into a local minimum at training start. Usually a L2 norm is used.

In Tensorflow, you can directly write the standard Tikhonov regularization scheme can expressed by adding a regularization term to the loss:

```
# losses
self.regularizers = tf.nn.l2_loss(self.A1) + tf.nn.l2_loss(self.A2)
self.model_loss = tf.losses.softmax_cross_entropy(self.y, self.logit)
beta = 5e-2
self.loss = self.model_loss + beta*self.regularizers
```

0.2 Variable scope

In order to structure your network, you may want to use `variable_scope` environment. It will automatically prepend the name of the variable by it's own name:

```
# Parameters
with tf.variable_scope("mlp"):
    with tf.variable_scope("layer1"):
        self.A1 = tf.get_variable("A", (self.nc, self.nh),
            initializer=tf.random_normal_initializer())

        self.b1 = tf.Variable(tf.zeros([self.nh]), name="b")

    with tf.variable_scope("layer2"):
        self.A2 = tf.get_variable("A", (self.nh, self.no),
            initializer=tf.random_normal_initializer())
        self.b2 = tf.Variable(tf.zeros([self.no]), name="b")
```

Here the first A variable will be named (approximatively) *mlp/layer1/A* and the second *mlp/layer2/A*

0.3 Exercise

Modify your code to add weight initialization/regularization and variable scope.