NNs

http://bit.ly/2lFyeYZ

- Create algorithms from blocks (Caffe)

- Write a program:
  - Variable level
  - Layers or Blocks level
  - Change model dynamically

Top libraries by Github forks		
#1:	10282	tensorflow/tensorflow
#2:	6443	BVLC/caffe
#3:	1859	fchollet/keras
#4:	1461	dmlc/mxnet
#5:	1427	Theano/Theano
#6:	1336	torch/torch7
#7:	1151	Microsoft/CNTK
#8:	1080	deeplearning4j/deeplearning4j
#9:	960	karpathy/convnetjs
#10:	613	Lasagne/Lasagne
#11:	390	NervanaSystems/neon
#12:	355	NVIDIA/DIGITS
#13:	314	pfnet/chainer
#14:	249	mila-udem/blocks
#15:	204	tflearn/tflearn
#16:	101	IDSIA/brainstorm

Top libraries by Github contributors		
#1:	264	tensorflow/tensorflow
#2:	233	Theano/Theano
#3:	196	BVLC/caffe
#4:	195	fchollet/keras
#5:	147	dmlc/mxnet
#6:	100	torch/torch7
#7:	76	deeplearning4j/deeplearning4j
#8:	67	Microsoft/CNTK
#9:	58	pfnet/chainer
#10:	48	mila-udem/blocks
#11:	47	Lasagne/Lasagne
#12:	38	NervanaSystems/neon
#13:	28	NVIDIA/DIGITS
#14:	19	tflearn/tflearn
#15:	14	karpathy/convnetjs
#16:	14	IDSIA/brainstorm

Top libraries by Github issues opened		
#1:	2706	BVLC/caffe
#2:	2155	fchollet/keras
#3:	1855	tensorflow/tensorflow
#4:	1592	Theano/Theano
#5:	1373	dmlc/mxnet
#6:	905	deeplearning4j/deeplearning4j
#7:	533	Microsoft/CNTK
#8:	496	mila-udem/blocks
#9:	432	pfnet/chainer
#10:	424	NVIDIA/DIGITS
#11:	371	Lasagne/Lasagne
#12:	315	torch/torch7
#13:	208	NervanaSystems/neon
#14:	97	tflearn/tflearn
#15:	82	IDSIA/brainstorm
#16:	41	karpathy/convnetjs

Aggregate popularity (30•contrib + 10•issues + 5•forks)•10e-3		
#1:	77.88	tensorflow/tensorflow
#2:	65.16	BVLC/caffe
#3:	36.70	fchollet/keras
#4:	30.05	Theano/Theano
#5:	25.45	dmlc/mxnet
#6:	16.73	deeplearning4j/deeplearning4j
#7:	13.10	Microsoft/CNTK
#8:	12.83	torch/torch7
#9:	8.19	Lasagne/Lasagne
#10:	7.65	mila-udem/blocks
#11:	7.63	pfnet/chainer
#12:	6.86	NVIDIA/DIGITS
#13:	5.63	karpathy/convnetjs
#14:	5.17	NervanaSystems/neon
#15:	2.56	tflearn/tflearn
#16:	1.75	IDSIA/brainstorm

How to write a perceptron

Tensorflow

```python
# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])


# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with RELU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Hidden layer with RELU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
# Construct model
pred = multilayer_perceptron(x, weights, biases)
# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
# Initializing the variables
init = tf.global_variables_initializer()
```

```python
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            # Run optimization op (backprop) and cost op (to get loss value)
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_x,
                                                          y: batch_y})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(avg_cost))
    print("Optimization Finished!")

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print("Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))
```

Keras

```
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])

history = model.fit(X_train, Y_train,
                    batch_size=batch_size, nb_epoch=nb_epoch,
                    verbose=1, validation_data=(X_test, Y_test))
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

Chainer

```python
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import training
from chainer.training import extensions


# Network definition
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__(
            # the size of the inputs to each layer will be inferred
            l1=L.Linear(None, n_units),  # n_in -> n_units
            l2=L.Linear(None, n_units),  # n_units -> n_units
            l3=L.Linear(None, n_out),  # n_units -> n_out
        )

    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

```python
# Set up a neural network to train
# Classifier reports softmax cross entropy loss and accuracy at every
# iteration, which will be used by the PrintReport extension below.
model = L.Classifier(MLP(args.unit, 10))
if args.gpu >= 0:
    chainer.cuda.get_device(args.gpu).use()  # Make a specified GPU current
    model.to_gpu()  # Copy the model to the GPU
# Setup an optimizer
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)
# Load the MNIST dataset
train, test = chainer.datasets.get_mnist()
train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
test_iter = chainer.iterators.SerialIterator(test, args.batchsize,
                                             repeat=False, shuffle=False)
# Set up a trainer
updater = training.StandardUpdater(train_iter, optimizer, device=args.gpu)
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)
# Evaluate the model with the test dataset for each epoch
trainer.extend(extensions.Evaluator(test_iter, model, device=args.gpu))
# Dump a computational graph from 'loss' variable at the first iteration
# The "main" refers to the target link of the "main" optimizer.
trainer.extend(extensions.dump_graph('main/loss'))
# Take a snapshot at each epoch
trainer.extend(extensions.snapshot(), trigger=(args.epoch, 'epoch'))
# Write a log of evaluation statistics for each epoch
trainer.extend(extensions.LogReport())
# Run the training
trainer.run()
```

Theano

```python
# start-snippet-1
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,
                 activation=T.tanh):
        self.input = input
        if W is None:
            W_values = numpy.asarray(
                rng.uniform(
                    low=-numpy.sqrt(6. / (n_in + n_out)),
                    high=numpy.sqrt(6. / (n_in + n_out)),
                    size=(n_in, n_out)
                ),
                dtype=theano.config.floatX
            )
            if activation == theano.tensor.nnet.sigmoid:
                W_values *= 4

            W = theano.shared(value=W_values, name='W', borrow=True)

        if b is None:
            b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
            b = theano.shared(value=b_values, name='b', borrow=True)

        self.W = W
        self.b = b

        lin_output = T.dot(input, self.W) + self.b
        self.output = (
            lin_output if activation is None
            else activation(lin_output)
        )
        # parameters of the model
        self.params = [self.W, self.b]
```

```python
# start-snippet-2
class MLP(object):
    def __init__(self, rng, input, n_in, n_hidden, n_out):
        self.hiddenLayer = HiddenLayer(
            rng=rng,
            input=input,
            n_in=n_in,
            n_out=n_hidden,
            activation=T.tanh
        )
        self.logRegressionLayer = LogisticRegression(
            input=self.hiddenLayer.output,
            n_in=n_hidden,
            n_out=n_out
        )
        self.L1 = (
            abs(self.hiddenLayer.W).sum()
            + abs(self.logRegressionLayer.W).sum()
        )
        self.L2_sqr = (
            (self.hiddenLayer.W ** 2).sum()
            + (self.logRegressionLayer.W ** 2).sum()
        )
        self.negative_log_likelihood = (
            self.logRegressionLayer.negative_log_likelihood
        )
        # same holds for the function computing the number of errors
        self.errors = self.logRegressionLayer.errors
        # the parameters of the model are the parameters of the two layer it is
        # made out of
        self.params = self.hiddenLayer.params + self.logRegressionLayer.params
        # end-snippet-3
        # keep track of model input
        self.input = input
```

class `sklearn.neural_network.` **`MLPClassifier`** (*hidden_layer_sizes=(100,), activation='relu', solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08*) [source]

hidden_layer_sizes : tuple, length = n_layers - 2, default (100,) -The ith element represents the number of neurons in the ith hidden layer.

activation : {'identity', 'logistic', 'tanh', 'relu'}, default 'relu'

solver : {'lbfgs', 'sgd', 'adam'}, default 'adam'

alpha : float, optional, default 0.0001 - L2 penalty (regularization term) parameter.

batch_size : int, optional, default 'auto' - Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", batch_size=min(200, n_samples)

learning_rate : {'constant', 'invscaling', 'adaptive'}, default 'constant'

max_iter : int, optional, default 200 - Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations.

random_state : int or RandomState, optional, default None - State or seed for random number generator.

momentum : float, default 0.9 - Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum : boolean, default True - Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

early_stopping : bool, default False