MinPy Documentation

Release 0.3.3

DMLC

May 13, 2017

Basics

MinPy aims at prototyping a pure NumPy interface above MXNet backend. This package targets two groups of users:

- The beginners who wish to have a firm grasp of the fundamental concepts of deep learning, and
- Researchers who want a quick prototype of advanced and complex algorithms.

It is not intended for those who want to compose with ready-made components, although there are enough (layers and activation functions etc.) to get started.

As much as possible, MinPy strikes to be purely NumPy-compatible. It also abides to a fully imperative programming experience that is familiar to most users. Letting go the popular approach that mixes in symbolic programming sacrifices some runtime optimization opportunities, in favor of algorithmic expressiveness and flexibility. However, MinPy performs reasonably well, especially when computation dominates.

This document describes its main features:

- Auto-differentiation
- Transparent CPU/GPU acceleration
- Visualization using TensorBoard
- Learning deep learning using MinPy

CHAPTER 1

MinPy installation guide

There are generally three steps to follow:

1. Install MXNet
2. Setup Python package and environment
3. Install MinPy

Install MXNet

The full guide of MXNet is here to build and install MXNet. Below, we give the common steps for Linux and OSX.

On Ubuntu/Debian

With CUDA 8.0 and Cudnn 5.0 installed, install the other dependencies and build mxnet by

```
sudo apt-get update
sudo apt-get install -y build-essential git libatlas-base-dev libopencv-dev
git clone --recursive -b engine https://github.com/dmlc/mxnet
cd mxnet;
cp make/config.mk .
echo "USE_CUDA=1" >>config.mk
echo "USE_CUDA_PATH=/usr/local/cuda" >>config.mk
echo "USE_CUDNN=1" >>config.mk
make -j$(nproc)
```

On OSX

Do the following instead.

```
brew update
brew tap homebrew/science
brew info opencv
brew install opencv
brew info openblas
brew install openblas
git clone --recursive -b engine https://github.com/dmlc/mxnet
cd mxnet; cp make/osx.mk ./config.mk; make -j$(sysctl -n hw.ncpu)
```

It turns out that installing `openblas` is necessary, in addition to modify the makefile, to fix one of the build issues.

Setup Python and its environment

Refer to MXNet installation document for Python package installation.. One of the most common problem a beginner runs into is not setting the environment variable to tell Python where to find the library. Suppose you have installed `mxnet` under your home directory and is running bash shell. Put the following line in your `~/.bashrc` (or `~/.bash_profile`)

```
export PYTHONPATH=~/mxnet/python:$PYTHONPATH
```

If the installation meets Numpy or MXNet version conflicts with your other projects, we recommend using virtualenv and virtualenvwrapper to resolve the issue:

```
pip install virtualenv virtualenvwrapper
```

Add two lines to your shell startup file (.bashrc, .profile, etc.) to set the location where the virtual environments should live and the location of the script installed with this package:

```
export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

After editing it, reload the startup file (e.g., run source ~/.bashrc). Then config the virtual python environment:

```
mkvirtualenv minpy_dev
pip install numpy pyyaml
python ~/mxnet/python/setup.py install
```

Note that `~/mxnet/python` should be replaced by the path of the `engine` branch MXNet.

To start the virtual envirment:

```
workon minpy_dev
```

Install MinPy

Minpy prototypes a pure Numpy interface. To make the interface consistent, please make sure Numpy version >= 1.10.0 before install Minpy.

MinPy releases are uploaded to PyPI. Just use `pip` to install after you install MXNet.

```
pip install minpy
```

Don't forget to upgrade once in a while to use the latest features!

For developers

Currently MinPy is going through rapid development (but we do our best to keep stable APIs). So it is adviced to do an editable installation. Change directory into where the Python package lies. If you are in a virtual environment, run `python setup.py develop`. If you are using your system Python packages, then run `python setup.py develop --user`. This will ensure a symbolic link to the project, so you do not have to install a second time when you update this repository.

Docker images for MinPy

Optionally, you may build MinPy/MXNet using docker.

Build images

Build Docker images using the `Dockerfile` found in the `docker` directory.

Just run the following command and it will build MXNet with CUDA support and MinPy in a row:

```
docker build -t dmlc/minpy ./docker/
```

Start a container

To launch the docker, you need to install nvidia-docker first.

Then use `nvidia-docker` to start the container with GPU access. MinPy is ready to use now!

```
$ nvidia-docker run -ti dmlc/minpy python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import minpy as np
>>> ...
```

Train a model on MNIST to check everything works

```
nvidia-docker run dmlc/minpy python dmlc/minpy/examples/basics/logistic.py --gpus 0
```

CHAPTER 2

Logistic regression tutorial

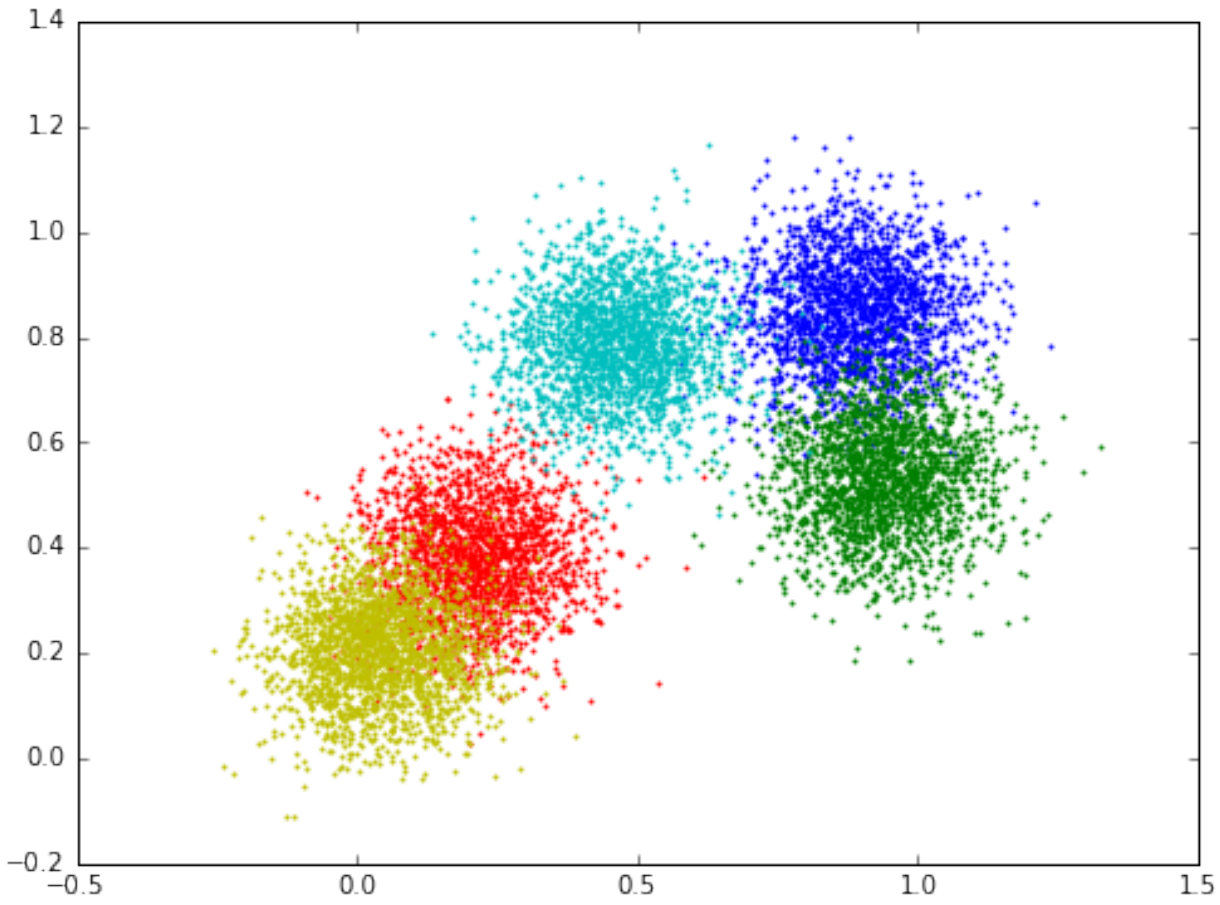
This part of tutorial is derived from its step-by-step notebook version multinomial logistic regression example, the emphasis is to showcase the basic capacity of MinPy.

We will work on a classification problem of a synthetic data set. Each point is a high-dimentional data in one of the five clusters. We will build a one-layer multinomial logistic regression model. The goal is to learn a weight matrix `weight`, such that for a data point `x` the probability that it is assigned to its class (cluster) is the largest.

The data is generated with the following code.

```
import numpy as np

""" Generates several clusters of Gaussian points """
def gaussian_cluster_generator(num_samples=10000, num_features=500, num_classes=5):
    mu = np.random.rand(num_classes, num_features)
    sigma = np.ones((num_classes, num_features)) * 0.1
    num_cls_samples = num_samples / num_classes
    x = np.zeros((num_samples, num_features))
    y = np.zeros((num_samples, num_classes))
    for i in range(num_classes):
        cls_samples = np.random.normal(mu[i,:], sigma[i,:], (num_cls_samples, num_
↪features))
        x[i*num_cls_samples:(i+1)*num_cls_samples] = cls_samples
        y[i*num_cls_samples:(i+1)*num_cls_samples,i] = 1
    return x, y
```

The visualization of the data:



The following is the numpy version. The function `predict` outputs the probability, the `train` function iterates over the data, computes the loss, the gradients, and updates the parameters `w` with a fixed learning rate.

```
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw
```

```
24
25 # Initialize training data.
26 num_samples = 10000
27 num_features = 500
28 num_classes = 5
29 data, label = make_data(num_samples, num_features, num_classes)
30
31 # Initialize training weight and train
32 weight = random.randn(num_features, num_classes)
33 train(weight, data, 100)
```

The minpy version is very similar, except a few lines that are highlighted:

- Among some new imported libraries, `minpy.numpy` replaces `numpy`. This lightweight library is fully numpy compatible, but it allows us to add small instrumentations in the style of autograd
- Defines *loss* explicitly with the function `train_loss`
- MinPy then derives a function to compute gradients automatically (line 24)

```
1  import minpy.numpy as np
2  import minpy.numpy.random as random
3  from minpy.core import grad_and_loss
4  from examples.utils.data_utils import gaussian_cluster_generator as make_data
5  from minpy.context import set_context, gpu
6
7  # Please uncomment following if you have GPU-enabled MXNet installed.
8  # This single line of code will run MXNet operations on GPU 0.
9  # set_context(gpu(0)) # set the global context as gpu(0)
10
11 # Predict the class using multinomial logistic regression (softmax regression).
12 def predict(w, x):
13     a = np.exp(np.dot(x, w))
14     a_sum = np.sum(a, axis=1, keepdims=True)
15     prob = a / a_sum
16     return prob
17
18 def train_loss(w, x):
19     prob = predict(w, x)
20     loss = -np.sum(label * np.log(prob)) / num_samples
21     return loss
22
23 """Use Minpy's auto-grad to derive a gradient function off loss"""
24 grad_function = grad_and_loss(train_loss)
25
26 # Using gradient descent to fit the correct classes.
27 def train(w, x, loops):
28     for i in range(loops):
29         dw, loss = grad_function(w, x)
30         if i % 10 == 0:
31             print('Iter {}, training loss {}'.format(i, loss))
32         # gradient descent
33         w -= 0.1 * dw
34
35 # Initialize training data.
36 num_samples = 10000
37 num_features = 500
38 num_classes = 5
39 data, label = make_data(num_samples, num_features, num_classes)
```

```
40
41 # Initialize training weight and train
42 weight = random.randn(num_features, num_classes)
43 train(weight, data, 100)
```

Now, if you uncomment line 9 to set MXNet context on GPU 0, this one line change (`set_context(gpu(0))`) will enable the same code to run on GPU!

For more functionality of MinPy/MXNet, we invite you to read later sections of this tutorial.

CHAPTER 3

NumPy under MinPy, with GPU

This part of tutorial is also available in step-by-step notebook version on github. Please try it out!

Basic NDArray Operation

MinPy has the same syntax as NumPy, which is the language of choice for numerical computing, and in particular deep learning. The popular Stanford course cs231n uses NumPy as its main coursework. To use NumPy under MinPy, you only need to replace `import numpy as np` with `import minpy.numpy as np` at the header of your NumPy program. if you are not familiar with NumPy, you may want to look up NumPy Quickstart Tutorial for more details.

Using NumPy under MinPy has two simple but important reasons, one for productivity and another for performance: 1) Auto-differentiation, and 2) GPU/CPU co-execution. We will discuss them in this tutorial.

But first, let us review some of the most common usages of NumPy.

Array Creation

An array can be created in multiple ways. For example, we can create an array from a regular Python list or tuple by using the `array` function

```
In [1]: import minpy.numpy as np

        a = np.array([1,2,3])  # create a 1-dimensional array with a python list
        b = np.array([[1,2,3], [2,3,4]])  # create a 2-dimensional array with a nested python list
```

Here are some useful ways to create arrays with initial placeholder content.

```
In [2]: a = np.zeros((2,3))     # create a 2-dimensional array full of zeros with shape (2,3)
        b = np.ones((2,3))      # create a same shape array full of ones
        c = np.full((2,3), 7)   # create a same shape array with all elements set to 7
        d = np.empty((2,3))     # create a same shape whose initial content is random and depends on t
```

Basic Operations

Arithmetic operators on arrays apply *elementwise*, with a new array holding result.

```
In [3]: a = np.ones((2,3))
        b = np.ones((2,3))
        c = a + b  # elementwise plus
        d = - c    # elementwise minus
        print(d)
        e = np.sin(c**2).T  # elementwise pow and sin, and then transpose
        print(e)
        f = np.maximum(a, c)  # elementwise max
        print(f)
```

```
[[-2. -2. -2.]
 [-2. -2. -2.]]
[[-0.7568025 -0.7568025]
 [-0.7568025 -0.7568025]
 [-0.7568025 -0.7568025]]
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
```

Indexing and Slicing

The slice operator `[]` applies on axis 0.

```
In [4]: a = np.arange(6)
        a = np.reshape(a, (3,2))
        print(a[:])
        # assign -1 to the 2nd row
        a[1:2] = -1
        print(a)
```

```
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]]
[[ 0.  1.]
 [-1. -1.]
 [ 4.  5.]]
```

We can also slice a particular axis with the method `slice_axis`

```
In [5]: # slice out the 2nd column
        d = np.slice_axis(a, axis=1, begin=1, end=2)
        print(d)
```

```
[[ 1.]
 [-1.]
 [ 5.]]
```

AutoGrad Feature

If you work in a policy mode called `NumpyOnlyPolicy` (refer here for more details), MinPy is almost compatible with the most of NumPy usages. But what makes MinPy awesome is that it give you the power of autograd, saving you from writing the most tedious and error prone part of deep net implementation:

```
In [6]: from minpy.core import grad
```

```
        # define a function: f(x) = 5*x^2 + 3*x - 2
        def foo(x):
            return 5*(x**2) + 3*x - 2

        # f(4) = 90
        print(foo(4))

        # get the derivative function by `grad`: f'(x) = 10*x + 3
        d_foo = grad(foo)

        # f'(4) = 43.0
        print(d_foo(4))
```

```
90
43.0
```

More details about this part can be found in Autograd Tutorial.

GPU Support

But we do not stop here, we want MinPy not only friendly to use, but also fast. To this end, MinPy leverages GPU's parallel computing ability. The code below shows our GPU support and a set of API to make you freely to change the runnning context (i.e. to run on CPU or GPU). You can refer to Select Context for MXNet for more details.

```
In [7]: import minpy.numpy as np
        import minpy.numpy.random as random
        from minpy.context import cpu, gpu
        import time

        n = 100

        with cpu():
            x_cpu = random.rand(1024, 1024) - 0.5
            y_cpu = random.rand(1024, 1024) - 0.5

            # dry run
            for i in xrange(10):
                z_cpu = np.dot(x_cpu, y_cpu)
            z_cpu.asnumpy()

            # real run
            t0 = time.time()
            for i in xrange(n):
                z_cpu = np.dot(x_cpu, y_cpu)
            z_cpu.asnumpy()
            t1 = time.time()

        with gpu(0):
            x_gpu0 = random.rand(1024, 1024) - 0.5
            y_gpu0 = random.rand(1024, 1024) - 0.5

            # dry run
            for i in xrange(10):
                z_gpu0 = np.dot(x_gpu0, y_gpu0)
            z_gpu0.asnumpy()

            # real run
            t2 = time.time()
```

```
            for i in xrange(n):
                z_gpu0 = np.dot(x_gpu0, y_gpu0)
            z_gpu0.asnumpy()
            t3 = time.time()

        print("run on cpu: %.6f s/iter" % ((t1 - t0) / n))
        print("run on gpu: %.6f s/iter" % ((t3 - t2) / n))
```

```
run on cpu: 0.100039 s/iter
run on gpu: 0.000422 s/iter
```

The `asnumpy()` call is somewhat mysterious, implying `z_cpu` is not NumPy's `ndarray` type. Indeed this is true. For fast execution, MXNet maintains its own datastrcutre `NDArray`. This calls re-synced `z_cpu` into NumPy array.

As you can see, there is a gap between the speeds of matrix multiplication in CPU and GPU. That's why we set default policy mode as `PreferMXNetPolicy`, which means MinPy will dispatch the operator to MXNet as much as possible for you, and achieve transparent fallback while there is no MXNet implementation. MXNet operations run on GPU, whereas the fallbacks run on CPU.

See Transparent Fallback for more details.

Something You Need to Know

With Transparent Fallback, we hope to transparently upgrade the running speed without your changing a line of code. This can be done by expanding the MXNet GPU operators.

However, there are some important pitfalls you should know when you try to use MinPy, we strongly suggest that you should read it next.

CHAPTER 4

Autograd in MinPy

This tutorial is also available in step-by-step notebook version on github. Please try it out!

Writing backprop is often the most tedious and error prone part of a deep net implementation. In fact, the feature of autograd has wide applications and goes beyond the domain of deep learning. MinPy's autograd applies to any NumPy code that is imperatively programmed. Moreover, it is seemlessly integrated with MXNet's symbolic program (see *for example*). By using MXNet's execution engine, all operations can be executed in GPU if available.

A Close Look at Autograd System

MinPy's implementation of autograd is insprired from the Autograd project. It computes a gradient function for any single-output function. For example, we define a simple function `foo`:

```
In [1]: def foo(x):
            return x**2

        foo(4)
```

```
Out[1]: 16
```

Now we want to get its derivative. To do so, simply import `grad` from `minpy.core`.

```
In [2]: import minpy.numpy as np  # currently need import this at the same time
        from minpy.core import grad

        d_foo = grad(foo)
```

```
In [3]: d_foo(4)
```

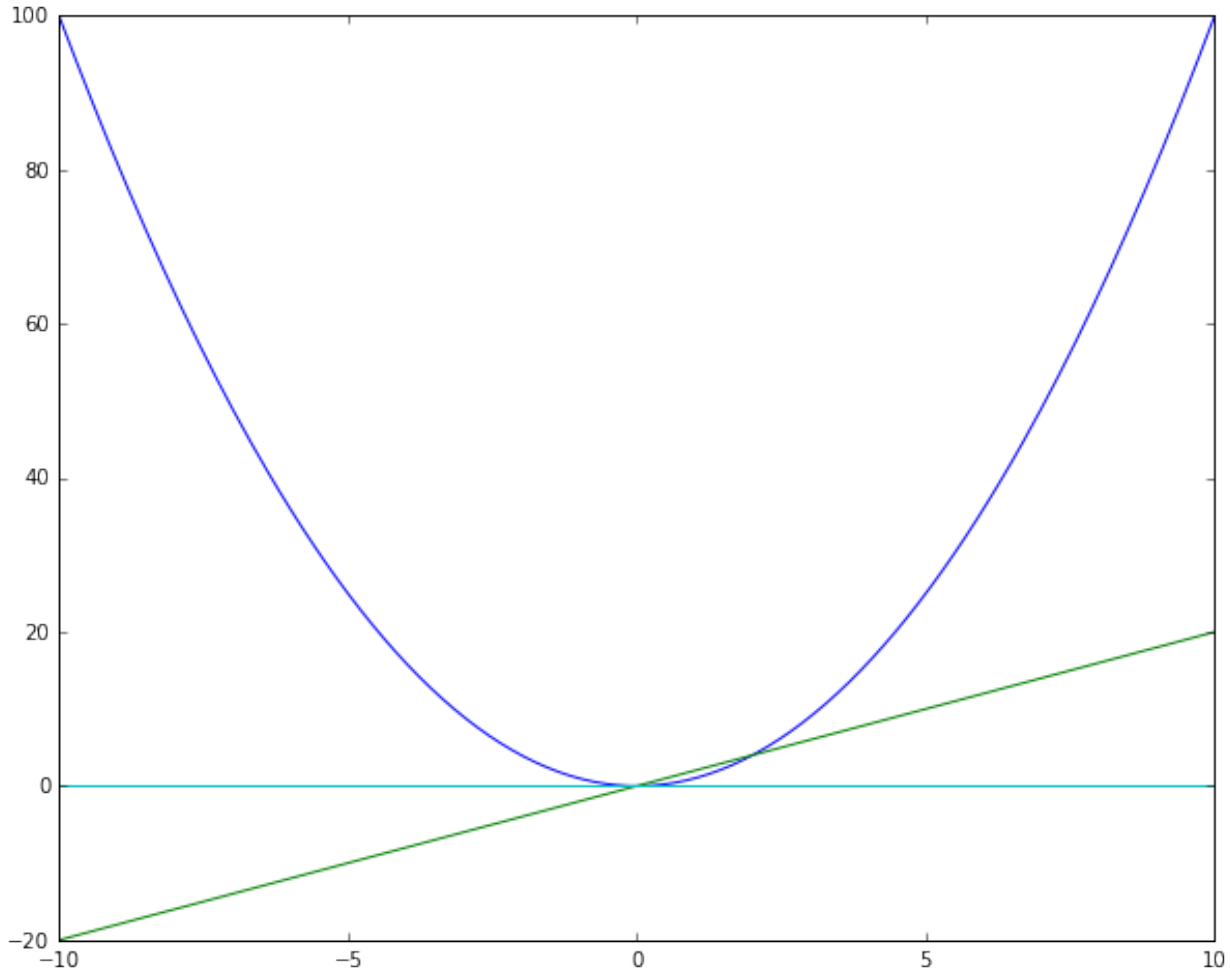
```
Out[3]: 8.0
```

You can also differentiate as many times as you want:

```
In [4]: d_2_foo = grad(d_foo)
        d_3_foo = grad(d_foo)
```

Now import `matplotlib` to visualize the derivatives.

```
In [5]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
        plt.rcParams['image.interpolation'] = 'nearest'
        plt.rcParams['image.cmap'] = 'gray'

        x = np.linspace(-10, 10, 200)
        # plt.plot only takes ndarray as input. Explicitly convert MinPy Array into ndarray.
        plt.plot(x.asnumpy(), foo(x).asnumpy(),
                 x.asnumpy(), d_foo(x).asnumpy(),
                 x.asnumpy(), d_2_foo(x).asnumpy(),
                 x.asnumpy(), d_3_foo(x).asnumpy())
        plt.show()
```



Just as you expected.

Autograd also differentiates vector inputs. For example:

```
In [6]: x = np.array([1, 2, 3, 4])
        d_foo(x)
```

```
Out[6]: [ 2.  4.  6.  8.]
```

Gradient of Multivariate Functions

As for multivariate functions, you also need to specify arguments for derivative calculation. Only the specified argument will be calcualted. Just pass the position of the target argument (of a list of arguments) in `grad`. For example:

```
In [7]: def bar(a, b, c):
            return 3*a + b**2 - c
```

We get their gradients by specifying their argument position.

```
In [8]: gradient = grad(bar, [0, 1, 2])
        grad_array = gradient(2, 3, 4)
        print grad_array
```

```
[3.0, 6.0, -1.0]
```

`grad_array[0]`, `grad_array[1]`, and `grad_array[2]` are gradients of argument `a`, `b`, and `c`.

The following section will introduce a more comprehensive example on matrix calculus.

Autograd for Loss Function

Since in world of machine learning we optimize a scalar loss, Autograd is particular useful to obtain the gradient of input parameters for next updates. For example, we define an affine layer, relu layer, and a softmax loss. Before dive into this section, please see *Logistic regression tutorial* first for a simpler application of Autograd.

```
In [9]: def affine(x, w, b):
            """
            Computes the forward pass for an affine (fully-connected) layer.
            The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
            examples, where each example x[i] has shape (d_1, ..., d_k). We will
            reshape each input into a vector of dimension D = d_1 * ... * d_k, and
            then transform it to an output vector of dimension M.
            Inputs:
            - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
            - w: A numpy array of weights, of shape (D, M)
            - b: A numpy array of biases, of shape (M,)
            Returns a tuple of:
            - out: output, of shape (N, M)
            """
            out = np.dot(x, w) + b
            return out

        def relu(x):
            """
            Computes the forward pass for a layer of rectified linear units (ReLUs).
            Input:
            - x: Inputs, of any shape
            Returns a tuple of:
            - out: Output, of the same shape as x
            """
            out = np.maximum(0, x)
            return out

        def softmax_loss(x, y):
            """
            Computes the loss for softmax classification.
            Inputs:
```

```
        - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
        for the ith input.
        - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
        0 <= y[i] < C
        Returns a tuple of:
        - loss: Scalar giving the loss
        """
        N = x.shape[0]
        probs = np.exp(x - np.max(x, axis=1, keepdims=True))
        probs = probs / np.sum(probs, axis=1, keepdims=True)
        loss = -np.sum(np.log(probs) * y) / N
        return loss
```

Then we use these layers to define a single layer fully-connected network, with a softmax output.

```
In [10]: class SimpleNet(object):
             def __init__(self, input_size=100, num_class=3):
                 # Define model parameters.
                 self.params = {}
                 self.params['w'] = np.random.randn(input_size, num_class) * 0.01
                 self.params['b'] = np.zeros((1, 1))  # don't use int(1) (int cannot track gradient

             def forward(self, X):
                 # First affine layer (fully-connected layer).
                 y1 = affine(X, self.params['w'], self.params['b'])
                 # ReLU activation.
                 y2 = relu(y1)
                 return y2

             def loss(self, X, y):
                 # Compute softmax loss between the output and the label.
                 return softmax_loss(self.forward(X), y)
```

We define some hyperparameters.

```
In [11]: batch_size = 100
         input_size = 50
         num_class = 3
```

Here is the net and data.

```
In [12]: net = SimpleNet(input_size, num_class)
         x = np.random.randn(batch_size, hidden_size)
         idx = np.random.randint(0, 3, size=batch_size)
         y = np.zeros((batch_size, num_class))
         y[np.arange(batch_size), idx] = 1
```

Now get gradients.

```
In [13]: gradient = grad(net.loss)
```

Then we can get gradient by simply call `gradient(X, y)`.

```
In [14]: d_x = gradient(x, y)
```

Ok, Ok, I know you are not interested in `x`'s gradient. I will show you how to get the gradient of the parameters. First, you need to define a function with the parameters as the arguments for Autograd to process. Autograd can only track the gradients **in the parameter list**.

```
In [15]: def loss_func(w, b, X, y):
             net.params['w'] = w
             net.params['b'] = b
             return net.loss(X, y)
```

Yes, you just need to provide an entry in the new function's parameter list for `w` and `b` and that's it! Now let's try to derive its gradient.

```
In [16]: # 0, 1 are the positions of w, b in the paramter list.
         gradient = grad(loss_func, [0, 1])
```

Note that you need to specify a list for the parameters that you want their gradients.

Now we have

```
In [17]: d_w, d_b = gradient(net.params['w'], net.params['b'], x, y)
```

With `d_w` and `d_b` in hand, training `net` is just a piece of cake.

Less Calculation: Get Forward Pass and Backward Pass Simultaneously

Since gradient calculation in MinPy needs forward pass information, if you need the forward result and the gradient calculation at the same time, please use `grad_and_loss` to get them simultaneously. In fact, `grad` is just a wrapper of `grad_and_loss`. For example, we can get

```
In [18]: from minpy.core import grad_and_loss
         forward_backward = grad_and_loss(bar, [0, 1, 2])
         grad_array, result = forward_backward(2, 3, 4)
```

`grad_array` and `result` are result of gradient and forward pass respectively.

CHAPTER 5

Transparent Fallback

This part of tutorial is also available in step-by-step notebook version on github. Please try it out!

Concept of transparent fallback

Since MinPy fully integrates MXNet, it allows you to use GPU to speed up your algorithm with only minor change, while keeping the familia NumPy syntax.

However, NumPy is a giant library with many of operators, each may have different calling conventions with different parameters. MXNet's GPU operators are only a subset of them. Therefore, it is inevitable that you may use some functions that are currently missing on the GPU side.

To solve this problem, MinPy designed a policy system to determine which implementation shoud be applied, consisted of build-in policies in `minpy.dispatch.policy` (also aliased in minpy root):

- `PreferMXNetPolicy()` [**Default**]: Prefer MXNet. Use NumPy as a transparent fallback, which wil be discussed below.
- `OnlyNumPyPolicy()`: Only use NumPy operations.
- `OnlyMXNetPolicy()`: Only use MXNet operations.
- `BlacklistPolicy()`: Discussed below.

The default policy `PreferMXNetPolicy` gracefully adopts the NumPy implementation once the operator is missing on GPU side, and handles the memory copies among GPU and CPU for you, illustrated with the following chart:

The code below will prove this for you.

```
In [1]: import minpy.numpy as np
        # First turn on the logging to know what happens under the hood.
        import logging
        logging.getLogger('minpy.array').setLevel(logging.DEBUG)

        # x is created as a MXNet array
        x = np.zeros((10, 20))
```

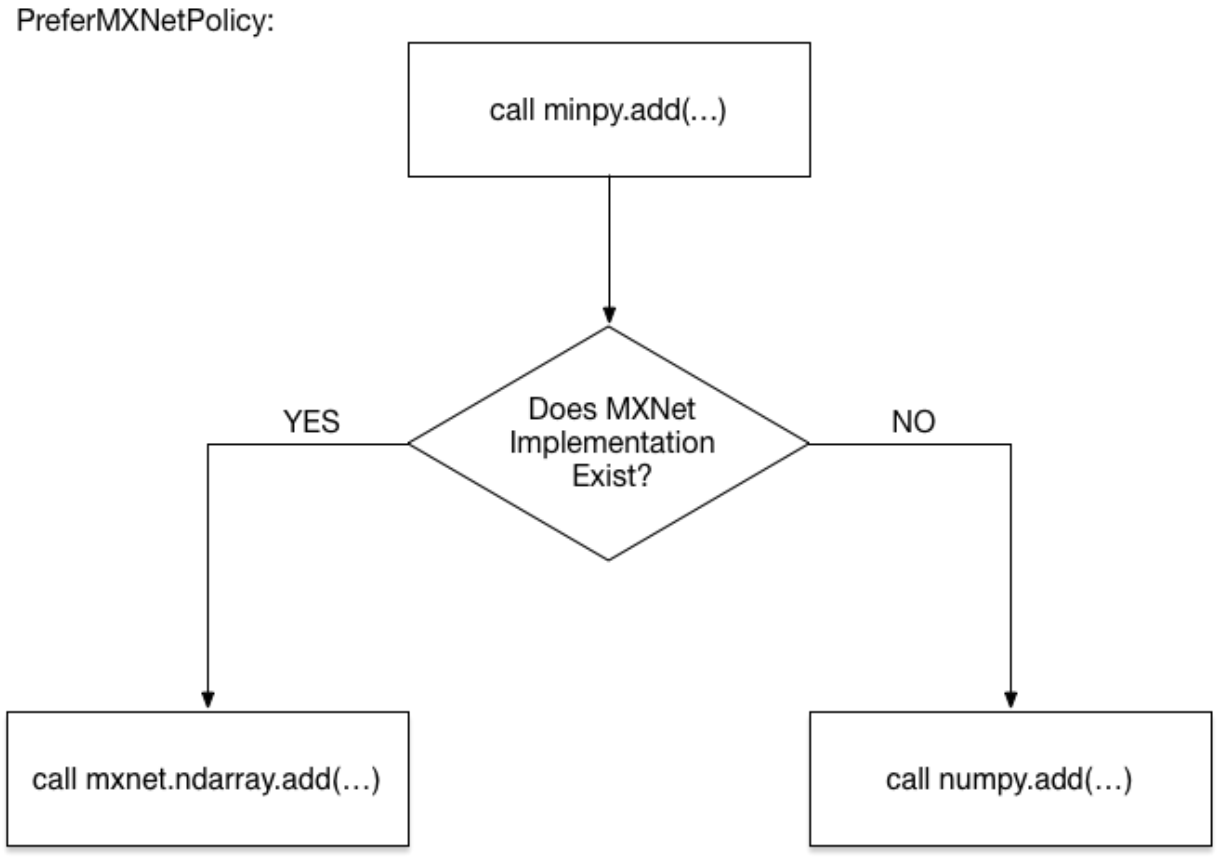


Fig. 5.1: PreferMXNetPolicy

```
        # `cosh` is currently missing in MXNet's GPU implementation.
        # So `x` will fallback to a NumPy array, so you will see a
        # logging like "Copy from MXNet array to NumPy array...", then
        # NumPy's implementation of `cosh` will be called to get the
        # result `y` as a NumPy array. But you don't need to worry
        # about the memory copy from GPU -> CPU
        y = np.cosh(x)


        # `log` has MXNet's GPU implementation, so it will copy the
        # array `y` from NumPy array to MXNet array and you will see
        # a logging like "Copy from NumPy array to MXNet array..."
        # Once again, you don't need to worry about it. It is transparent.
        z = np.log(y)


        # Turn off the logging.
        logging.getLogger('minpy.array').setLevel(logging.WARN)
```

```
I1110 11:11:21 12022 minpy.array:_synchronize_data:423] Copy from MXNet array to NumPy array for Arra
I1110 11:11:21 12022 minpy.array:_synchronize_data:429] Copy from NumPy array to MXNet array for Arra
```

However, there are a few of NumPy functions cannot work properly even in the `PreferMXNetPolicy`, due to the difference between NumPy and MXNet interface. Here is one example with different parameter types:

```
In [2]: # Uner PreferMXNetPolicy, np.random.normal will redirect to MXNet's implementation
        # but it does not support mu and sigma to be arrays (only scalar
        # is supported right now).
        import minpy.numpy as np
        def gaussian_cluster_generator(num_samples=10000, num_features=500, num_classes=5):
            mu = np.random.rand(num_classes, num_features)
            sigma = np.ones((num_classes, num_features)) * 0.1
            num_cls_samples = num_samples / num_classes
            x = np.zeros((num_samples, num_features))
            y = np.zeros((num_samples, num_classes))
            for i in range(num_classes):
                # this line will occur an error
                cls_samples = np.random.normal(mu[i,:], sigma[i,:], (num_cls_samples, num_features))
                x[i*num_cls_samples:(i+1)*num_cls_samples] = cls_samples
                y[i*num_cls_samples:(i+1)*num_cls_samples,i] = 1
            return x, y

        gaussian_cluster_generator(10000, 500, 5)
```

```
---------------------------------------------------------------------------
MXNetError                                Traceback (most recent call last)
<ipython-input-2-3e8f056001e5> in <module>()
     16     return x, y
     17
---> 18 gaussian_cluster_generator(10000, 500, 5)

...

/Users/ATlaS/Library/PyEnvs/minpy/lib/python2.7/site-packages/mxnet-0.7.0-py2.7.egg/mxnet/base.pyc in
     75     """
     76     if ret != 0:
---> 77         raise MXNetError(py_str(_LIB.MXGetLastError()))
     78
     79 if sys.version_info[0] < 3:
```

```
MXNetError: Invalid Parameter format for loc expect float but value='<mxnet.ndarray.NDArray object at
```

What that means is we must control dispatch at a finer granularity. We design another blacklist machinism for you. The operator in the blacklist will fallback to its numpy implementaiton and the content of blacklist will be prepared when you install MinPy automatically. This will solve most of these problems.

The procedure of function call under `PerferMXNetPolicy` will become:

The default blacklist is generated by testing the calls in this file. The test may not be complete, therefore you can run your code iteratively and generate a customized blacklist under `AutoBlacklistPolicy`:

```
In [ ]: import minpy
        p = minpy.AutoBlacklistPolicy(gen_rule=True, append_rule=True)
        set_global_policy(p)

        # under AutoBlacklistPolicy, operators throwing exception will be
        # added into the blacklist, then MinPy will call the NumPy
        # implementation next time to avoid this kind of exception.
        with p:
            gaussian_cluster_generator(10000, 500, 5)

        # this will not occur error afterwards
        gaussian_cluster_generator(10000, 500, 5)
```

Do check "Pitfalls when working together with NumPy" for known issues. If you encounter another, please raise an issue in our github!

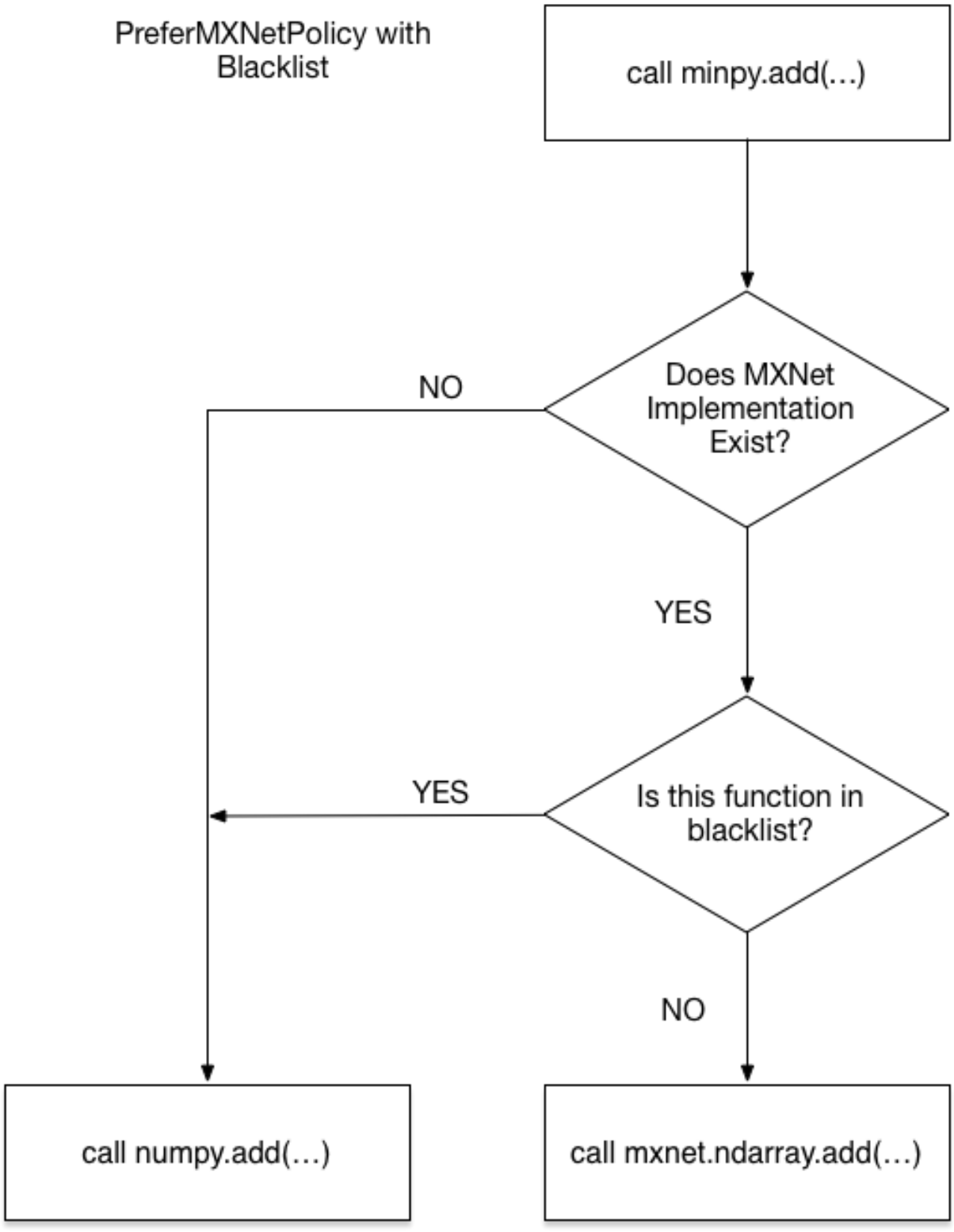


Fig. 5.2: Blacklist

CHAPTER 6

Complete solver and optimizer guide

This tutorial explains the "pipeline" of a typical research project. The idea is to get a quick prototype with the most flexible and familiar package (NumPy), then move the codebase to a more efficient paradigm (MXNet). Typically, one might need to go back and forth iteratively to refine the model. The choice of performance and flexibility depends on the project stage, and is best left to user to decide. Importantly, we have made it as straightforward as possible. For example:

- There is only one codebase to work with. NumPy and MXNet programming idioms mingle together rather easily.
- Neither style, however, requires the user to explicitly write tedious and (often) error-prone backprop path.
- Switching between GPU and CPU is straightforward, same code runs in either environment with only one line of change.

We will begin with a simple neural network using MinPy/NumPy and its `Solver` architecture. Then we will morph it gradually into a fully MXNet implementation, and then add NumPy statements as we see fit.

We do suggest you start with the simpler logistic regression example here.

Stage 0: Setup

All the codes covered in this tutorial could be found in this folder. All the codes in this folder are self-contained and ready-to-run. Before running, please make sure that you:

- Correctly install MXNet and MinPy. For guidance, refer to installation guide.
- Follow the instruction in the *README.md* to download the data.
- Run the example you want.

Stage 1: Pure MinPy

(*This section is also available in iPython Notebook* here)

In general, we advocate following the common coding style with the following modular partition:

- *Model*: your main job!
- *Layers*: building block of your model.
- *Solver* and *optimizer*: takes your model, training and testing data, train it.

The following MinPy code should be self-explainable; it is a simple two-layer feed-forward network. The model is defined in the `TwoLayerNet` class, where the `init`, `forward` and `loss` functions specify the parameters to be learnt, how the network computes all the way till the loss, and the computation of the loss itself, respectively. The crucial thing to note is the **absense of back-prop**, MinPy did it automatically for you.

```
""" Simple multi-layer perception neural network using Minpy """
import minpy
import minpy.numpy as np
from minpy.nn import layers
from minpy.nn.model import ModelBase
from minpy.nn.solver import Solver
from minpy.nn.io import NDArrayIter
from examples.utils.data_utils import get_CIFAR10_data

batch_size=128
input_size=(3, 32, 32)
flattened_input_size=3 * 32 * 32
hidden_size=512
num_classes=10

class TwoLayerNet(ModelBase):
    def __init__(self):
        super(TwoLayerNet, self).__init__()
        # Define model parameters.
        self.add_param(name='w1', shape=(flattened_input_size, hidden_size)) \
            .add_param(name='b1', shape=(hidden_size,)) \
            .add_param(name='w2', shape=(hidden_size, num_classes)) \
            .add_param(name='b2', shape=(num_classes,))

    def forward(self, X, mode):
        # Flatten the input data to matrix.
        X = np.reshape(X, (batch_size, 3 * 32 * 32))
        # First affine layer (fully-connected layer).
        y1 = layers.affine(X, self.params['w1'], self.params['b1'])
        # ReLU activation.
        y2 = layers.relu(y1)
        # Second affine layer.
        y3 = layers.affine(y2, self.params['w2'], self.params['b2'])
        return y3

    def loss(self, predict, y):
        # Compute softmax loss between the output and the label.
        return layers.softmax_loss(predict, y)

def main(args):
    # Create model.
    model = TwoLayerNet()
    # Create data iterators for training and testing sets.
    data = get_CIFAR10_data(args.data_dir)
    train_dataiter = NDArrayIter(data=data['X_train'],
                                 label=data['y_train'],
                                 batch_size=batch_size,
```

```
                                  shuffle=True)
    test_dataiter = NDArrayIter(data=data['X_test'],
                                label=data['y_test'],
                                batch_size=batch_size,
                                shuffle=False)
    # Create solver.
    solver = Solver(model,
                    train_dataiter,
                    test_dataiter,
                    num_epochs=10,
                    init_rule='gaussian',
                    init_config={
                        'stdvar': 0.001
                    },
                    update_rule='sgd_momentum',
                    optim_config={
                        'learning_rate': 1e-4,
                        'momentum': 0.9
                    },
                    verbose=True,
                    print_every=20)
    # Initialize model parameters.
    solver.init()
    # Train!
    solver.train()
```

This simple network takes several common layers from layers file. The same file contains a few other useful layers, such as batch normalization and dropout. Here is how a new model incorporates them.

```
batch_size=128
input_size=(3, 32, 32)
flattened_input_size=3 * 32 * 32
hidden_size=512
num_classes=10

class TwoLayerNet(ModelBase):
    def __init__(self):
        super(TwoLayerNet, self).__init__()
        # Define model parameters.
        self.add_param(name='w1', shape=(flattened_input_size, hidden_size)) \
            .add_param(name='b1', shape=(hidden_size,)) \
            .add_param(name='w2', shape=(hidden_size, num_classes)) \
            .add_param(name='b2', shape=(num_classes,)) \
            .add_param(name='gamma', shape=(hidden_size,),
                       init_rule='constant', init_config={'value': 1.0}) \
            .add_param(name='beta', shape=(hidden_size,), init_rule='constant') \
            .add_aux_param(name='running_mean', value=None) \
            .add_aux_param(name='running_var', value=None)

    def forward(self, X, mode):
        # Flatten the input data to matrix.
        X = np.reshape(X, (batch_size, 3 * 32 * 32))
        # First affine layer (fully-connected layer).
        y1 = layers.affine(X, self.params['w1'], self.params['b1'])
        # ReLU activation.
        y2 = layers.relu(y1)
        # Batch normalization
        y3, self.aux_params['running_mean'], self.aux_params['running_var'] = layers.
↪batchnorm(
```

```
            y2, self.params['gamma'], self.params['beta'],
            running_mean=self.aux_params['running_mean'],
            running_var=self.aux_params['running_var'])
        # Second affine layer.
        y4 = layers.affine(y3, self.params['w2'], self.params['b2'])
        # Dropout
        y5 = layers.dropout(y4, 0.5, mode=mode)
        return y5

    def loss(self, predict, y):
        # ... Same as above
```

Note that `running_mean` and `running_var` are defined as auxiliary parameters (`aux_param`). These parameters will not be updated by backpropagation.

The above code looks like *fully NumPy*, and yet it can run on GPU, and without explicit backprop needs. At this point, you might feel a little mysterious of what's happening under the hood. For advanced readers, here are the essential bits:

- The solver file takes the training and test dataset, and fits the model.
- At the end of the `_step` function, the `loss` function is auto-differentiated, deriving gradients to update the parameters.

Stage 2: MinPy + MXNet

While these features are great, it is by no means complete. For example, it is possible to write nested loops to perform convolution in NumPy, and the code will not break. However, much better implementations exist, especially when running on GPU.

MinPy leverages and integrates seemlessly with MXNet's symbolic programming (see MXNet Python Symbolic API). In a nutshell, MXNet's symbolic programming interface allows one to write a sub-DAG with symbolic expression. MXNet's convolutional kernel runs on both CPU and GPU, and its GPU version is highly optimized.

The following code shows how we add one convolutional layer and one pooling layer, using MXNet. Only the model is shown. You can get ready-to-run code for convolutional network.

```
import mxnet as mx

batch_size=128
input_size=(3, 32, 32)
flattened_input_size=3 * 32 * 32
hidden_size=512
num_classes=10

class ConvolutionNet(ModelBase):
    def __init__(self):
        super(ConvolutionNet, self).__init__()
        # Define symbols that using convolution and max pooling to extract better␣
↪features
        # from input image.
        net = mx.sym.Variable(name='X')
        net = mx.sym.Convolution(
                data=net, name='conv', kernel=(7, 7), num_filter=32)
        net = mx.sym.Activation(
                data=net, act_type='relu')
        net = mx.sym.Pooling(
```

```
                data=net, name='pool', pool_type='max', kernel=(2, 2),
                stride=(2, 2))
        net = mx.sym.Flatten(data=net)
        # Create forward function and add parameters to this model.
        self.conv = Function(
                net, input_shapes={'X': (batch_size,) + input_size},
                name='conv')
        self.add_params(self.conv.get_params())
        # Define ndarray parameters used for classification part.
        output_shape = self.conv.get_one_output_shape()
        conv_out_size = output_shape[1]
        self.add_param(name='w1', shape=(conv_out_size, hidden_size)) \
            .add_param(name='b1', shape=(hidden_size,)) \
            .add_param(name='w2', shape=(hidden_size, num_classes)) \
            .add_param(name='b2', shape=(num_classes,))

    def forward(self, X, mode):
        out = self.conv(X=X, **self.params)
        out = layers.affine(out, self.params['w1'], self.params['b1'])
        out = layers.relu(out)
        out = layers.affine(out, self.params['w2'], self.params['b2'])
        return out

    def loss(self, predict, y):
        return layers.softmax_loss(predict, y)
```

Stage 3: Pure MXNet

Of course, in this example, we can program it in fully MXNet symbolic way. You can get the full file with only MXNet symbols. Model is as the followings.

```
class ConvolutionNet(ModelBase):
    def __init__(self):
        super(ConvolutionNet, self).__init__()
        # Define symbols that using convolution and max pooling to extract better␣
↪features
        # from input image.
        net = mx.sym.Variable(name='X')
        net = mx.sym.Convolution(
                data=net, name='conv', kernel=(7, 7), num_filter=32)
        net = mx.sym.Activation(
                data=net, act_type='relu')
        net = mx.sym.Pooling(
                data=net, name='pool', pool_type='max', kernel=(2, 2),
                stride=(2, 2))
        net = mx.sym.Flatten(data=net)
        net = mx.sym.FullyConnected(
                data=net, name='fc1', num_hidden=hidden_size)
        net = mx.sym.Activation(
                data=net, act_type='relu')
        net = mx.sym.FullyConnected(
                data=net, name='fc2', num_hidden=num_classes)
        net = mx.sym.SoftmaxOutput(
                data=net, name='output')
        # Create forward function and add parameters to this model.
```

```
        self.cnn = Function(
                net, input_shapes={'X': (batch_size,) + input_size},
                name='cnn')
        self.add_params(self.cnn.get_params())

    def forward(self, X, mode):
        out = self.cnn(X=X, **self.params)
        return out

    def loss(self, predict, y):
        return layers.softmax_cross_entropy(predict, y)
```

Stage 3: MXNet + MinPy

However, the advantage of MinPy is that it brings in additional flexibility when needed, this is especially useful for quick prototyping to validate new ideas. Say we want to add a regularization in the loss term, this is done as the followings. Note that we only changed the `loss` function. Full code is available with regularization.

```
weight_decay = 0.001

class ConvolutionNet(ModelBase):
    def __init__(self):
        # ... Same as above.

    def forward(self, X, mode):
        # ... Same as above.

    def loss(self, predict, y):
        # Add L2 regularization for all the weights.
        reg_loss = 0.0
        for name, weight in self.params.items():
            reg_loss += np.sum(weight ** 2) * 0.5
        return layers.softmax_cross_entropy(predict, y) + weight_decay * reg_loss
```

CHAPTER 7

CNN Tutorial

This tutorial describes how to implement convolutional neural network (CNN) on MinPy. CNN is surprisingly effective on computer vision and natural language processing tasks, it is widely used in real world applications.

However, these tasks are also extremely computationally demanding. Therefore, training CNN models effectively calls for GPU acceleration. This tutorial explains how to use MinPy's ability to run on GPU transparently for the same model you developped for CPU.

This is also a gentle introduction on how to use `module builder` to specific an otherwise complex network.

We do suggest you start with *Complete solver and optimizer guide* for MinPy's conventional solver architecture.

Dataset: CIFAR-10

We use CIFAR-10 dataset for our CNN model.

CNN on MinPy

In *Complete solver and optimizer guide*, we introduced a simple model/solver architecture. Implementing CNN in MinPy is straightforward following the convention. The only difference is the model part. As for the performance critical CNN layers, it is important to use MXNet symbol, which has been carefully optimized for better performance on GPU. The following MinPy code defines a classical CNN to classify CIFAR-10 dataset.

If you are running on a server with GPU, **uncommenting line 16** to get the training going on GPU!

```
1  """Convolution Neural Network example using only MXNet symbol."""
2  import sys
3  import argparse
4
5  from minpy.nn.io import NDArrayIter
6  # Can also use MXNet IO here
7  # from mxnet.io import NDArrayIter
```

```
from minpy.core import Function
from minpy.nn import layers
from minpy.nn.model import ModelBase
from minpy.nn.solver import Solver
from examples.utils.data_utils import get_CIFAR10_data

# Please uncomment following if you have GPU-enabled MXNet installed.
#from minpy.context import set_context, gpu
#set_context(gpu(0)) # set the global context as gpu(0)

import mxnet as mx

batch_size=128
input_size=(3, 32, 32)
flattened_input_size=3 * 32 * 32
hidden_size=512
num_classes=10

class ConvolutionNet(ModelBase):
    def __init__(self):
        super(ConvolutionNet, self).__init__()
        # Define symbols that using convolution and max pooling to extract better␣
↪features
        # from input image.
        net = mx.sym.Variable(name='X')
        net = mx.sym.Convolution(
                data=net, name='conv', kernel=(7, 7), num_filter=32)
        net = mx.sym.Activation(
                data=net, act_type='relu')
        net = mx.sym.Pooling(
                data=net, name='pool', pool_type='max', kernel=(2, 2),
                stride=(2, 2))
        net = mx.sym.Flatten(data=net)
        net = mx.sym.FullyConnected(
                data=net, name='fc1', num_hidden=hidden_size)
        net = mx.sym.Activation(
                data=net, act_type='relu')
        net = mx.sym.FullyConnected(
                data=net, name='fc2', num_hidden=num_classes)
        net = mx.sym.SoftmaxOutput(data=net, name='softmax', normalization='batch')
        # Create forward function and add parameters to this model.
        input_shapes = {'X': (batch_size,) + input_size, 'softmax_label': (batch_size,
↪)}
        self.cnn = Function(net, input_shapes=input_shapes, name='cnn')
        self.add_params(self.cnn.get_params())

    def forward_batch(self, batch, mode):
        out = self.cnn(X=batch.data[0],
                       softmax_label=batch.label[0],
                       **self.params)
        return out

    def loss(self, predict, y):
        return layers.softmax_cross_entropy(predict, y)

def main(args):
    # Create model.
    model = ConvolutionNet()
```

```
    # Create data iterators for training and testing sets.
    data = get_CIFAR10_data(args.data_dir)
    train_dataiter = NDArrayIter(data=data['X_train'],
                                 label=data['y_train'],
                                 batch_size=batch_size,
                                 shuffle=True)
    test_dataiter = NDArrayIter(data=data['X_test'],
                                label=data['y_test'],
                                batch_size=batch_size,
                                shuffle=False)
    # Create solver.
    solver = Solver(model,
                    train_dataiter,
                    test_dataiter,
                    num_epochs=10,
                    init_rule='gaussian',
                    init_config={
                        'stdvar': 0.001
                    },
                    update_rule='sgd_momentum',
                    optim_config={
                        'learning_rate': 1e-3,
                        'momentum': 0.9
                    },
                    verbose=True,
                    print_every=20)
    # Initialize model parameters.
    solver.init()
    # Train!
    solver.train()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Multi-layer perceptron example
↪using minpy operators")
    parser.add_argument('--data_dir',
                        type=str,
                        required=True,
                        help='Directory that contains cifar10 data')
    main(parser.parse_args())
```

Build Your Network with `minpy.model_builder`

`minpy.model_builder` provides an interface helping you implement models more efficiently. Model builder generates models compatible with Minpy's solver. You only need to specify basic layer configurations of your model and model builder is going to handle the rest. Below is a model builder implementation of CNN. Please refer to *Complete model builder guide* for details.

Uncommenting line #20 to train on GPU.

```
'''
  This example demonstrates how to use minpy model builder to construct neural
↪networks.

  For details about how to train a model with solver, please refer to:
    http://minpy.readthedocs.io/en/latest/tutorial/complete.html
```

```
6
7     More models are available in minpy.nn.model_gallery.
8   '''
9
10  import sys
11  import argparse
12
13  import minpy.nn.model_builder as builder
14  from minpy.nn.solver import Solver
15  from minpy.nn.io import NDArrayIter
16  from examples.utils.data_utils import get_CIFAR10_data
17
18  # Please uncomment following if you have GPU-enabled MXNet installed.
19  #from minpy.context import set_context, gpu
20  #set_context(gpu(0)) # set the global context as gpu(0)
21
22  batch_size = 128
23  hidden_size = 512
24  num_classes = 10
25
26  def main(args):
27      # Define a convolutional neural network the same as above
28      net = builder.Sequential(
29          builder.Convolution((7, 7), 32),
30          builder.ReLU(),
31          builder.Pooling('max', (2, 2), (2, 2)),
32          builder.Flatten(),
33          builder.Affine(hidden_size),
34          builder.Affine(num_classes),
35      )
36
37      # Cast the definition to a model compatible with minpy solver
38      model = builder.Model(net, 'softmax', (3 * 32 * 32,))
39
40      data = get_CIFAR10_data(args.data_dir)
41
42      train_dataiter = NDArrayIter(data['X_train'],
43                           data['y_train'],
44                           batch_size=batch_size,
45                           shuffle=True)
46
47      test_dataiter = NDArrayIter(data['X_test'],
48                           data['y_test'],
49                           batch_size=batch_size,
50                           shuffle=False)
51
52      solver = Solver(model,
53                      train_dataiter,
54                      test_dataiter,
55                      num_epochs=10,
56                      init_rule='gaussian',
57                      init_config={
58                          'stdvar': 0.001
59                      },
60                      update_rule='sgd_momentum',
61                      optim_config={
62                          'learning_rate': 1e-3,
63                          'momentum': 0.9
```

```
                    },
                    verbose=True,
                    print_every=20)
    solver.init()
    solver.train()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Multi-layer perceptron example
↪using minpy operators")
    parser.add_argument('--data_dir',
                        type=str,
                        required=True,
                        help='Directory that contains cifar10 data')
    main(parser.parse_args())
```

CHAPTER 8

RNN Tutorial

This tutorial describes how to implement recurrent neural network (RNN) on MinPy. RNN has different architecture, the backprop-through-time (BPTT) coupled with various gating mechanisms can make implementation challenging. MinPy focuses on imperative programming and simplifies reasoning logics. This tutorial explains how, with a simple toy data set and three RNNs (vanilla RNN, LSTM and GRU).

We do suggest you start with *Complete solver and optimizer guide* for MinPy's conventional solver architecture.

Toy Dataset: the Adding Problem

The adding problem is a toy dataset for RNN used by many researchers. The input data of the dataset consists of two rows. The first row contains random float numbers between 0 and 1; the second row are all zeros, expect two randomly chosen locations being marked as 1. The corresponding output label is a float number summing up two numbers in the first row of the input data where marked as 1 in the second row. The length of the row `T` is the length of the input sequence. The paper[1] indicates that a less than 0.1767 Mean Square Error (MSE) proves the effectiveness of the

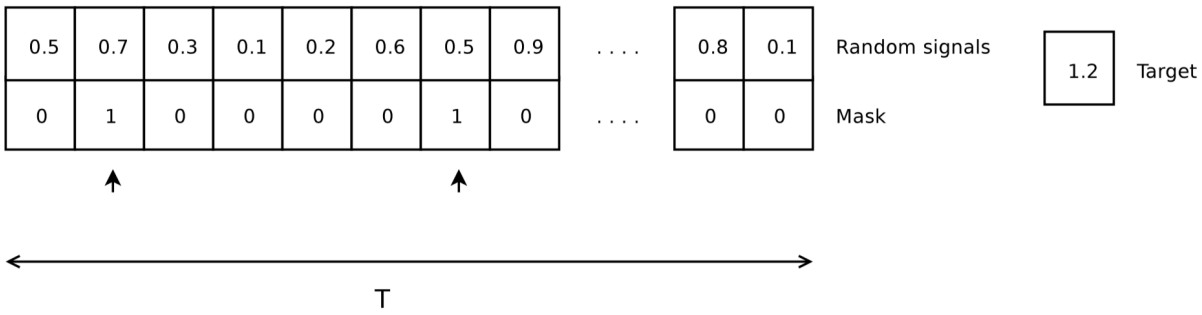


Fig. 8.1: Figure: An example of the adding problem. 0.7 and 0.5 are chosen from the input data on the left and sum up as 1.2, the label on the right[1].

network compared to the random guess. We set 0.1767 as the MSE baseline of our experiment.

We prepared an adding problem generator at `examples.utils.data_utils.adding_problem_generator` (here). We append its implementation as follows:

```
def adding_problem_generator(N, seq_len=6, high=1):
    """ A data generator for adding problem.

    The data definition strictly follows Quoc V. Le, Navdeep Jaitly, Geoffrey E.
    Hintan's paper, A Simple Way to Initialize Recurrent Networks of Rectified
    Linear Units.

    The single datum entry is a 2D vector with two rows with same length.
    The first row is a list of random data; the second row is a list of binary
    mask with all ones, except two positions sampled by uniform distribution.
    The corresponding label entry is the sum of the masked data. For
    example:

     input          label
     -----          -----
    1 4 5 3  ----->   9 (4 + 5)
    0 1 1 0

    :param N: the number of the entries.
    :param seq_len: the length of a single sequence.
    :param p: the probability of 1 in generated mask
    :param high: the random data is sampled from a [0, high] uniform distribution.
    :return: (X, Y), X the data, Y the label.
    """
    X_num = np.random.uniform(low=0, high=high, size=(N, seq_len, 1))
    X_mask = np.zeros((N, seq_len, 1))
    Y = np.ones((N, 1))
    for i in xrange(N):
        # Default uniform distribution on position sampling
        positions = np.random.choice(seq_len, size=2, replace=False)
        X_mask[i, positions] = 1
        Y[i, 0] = np.sum(X_num[i, positions])
    X = np.append(X_num, X_mask, axis=2)
    return X, Y
```

Vanilla RNN

In *Complete solver and optimizer guide*, we introduced a simple model/solver architecture. Implementing RNN in MinPy is very straightforward following the convention. The only difference is the model part. The following MinPy code defines the vanilla RNN in `RNNNet` class. We also include solver code for completeness. (You can find it in this folder.)

```
import minpy.numpy as np
from minpy.nn import layers
from minpy.nn.model import ModelBase
from minpy.nn.solver import Solver
from minpy.nn.io import NDArrayIter
from examples.utils.data_utils import adding_problem_generator as data_gen

```

[1] Q. V. Le, N. Jaitly, and G. E. Hinton, "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units," arXiv.org, vol. cs.NE. 04-Apr-2015.

```

class RNNNet(ModelBase):
    def __init__(self,
                 batch_size=100,
                 input_size=2,  # input dimension
                 hidden_size=64,
                 num_classes=1):
        super(RNNNet, self).__init__()
        self.add_param(name='Wx', shape=(input_size, hidden_size))\
            .add_param(name='Wh', shape=(hidden_size, hidden_size))\
            .add_param(name='b', shape=(hidden_size,))\
            .add_param(name='Wa', shape=(hidden_size, num_classes))\
            .add_param(name='ba', shape=(num_classes,))

    def forward(self, X, mode):
        seq_len = X.shape[1]
        batch_size = X.shape[0]
        hidden_size = self.params['Wh'].shape[0]
        h = np.zeros((batch_size, hidden_size))
        for t in xrange(seq_len):
            h = layers.rnn_step(X[:, t, :], h, self.params['Wx'],
                                self.params['Wh'], self.params['b'])
        y = layers.affine(h, self.params['Wa'], self.params['ba'])
        return y

    def loss(self, predict, y):
        return layers.l2_loss(predict, y)


def main():
    model = RNNNet()
    x_train, y_train = data_gen(10000)
    x_test, y_test = data_gen(1000)

    train_dataiter = NDArrayIter(x_train,
                                 y_train,
                                 batch_size=100,
                                 shuffle=True)

    test_dataiter = NDArrayIter(x_test,
                                y_test,
                                batch_size=100,
                                shuffle=False)

    solver = Solver(model,
                    train_dataiter,
                    test_dataiter,
                    num_epochs=10,
                    init_rule='xavier',
                    update_rule='adam',
                    task_type='regression',
                    verbose=True,
                    print_every=20)
    solver.init()
    solver.train()


if __name__ == '__main__':
```

```
    main()
```

Layers are defined in `minpy.nn.layers` (here). Note that `data_gen` is simply `adding_problem_generator`.

The key layer of vanilla RNN is also shown as follows:

```
def rnn_step(x, prev_h, Wx, Wh, b):
    """
    Run the forward pass for a single timestep of a vanilla RNN that uses a tanh
    activation function.

    The input data has dimension D, the hidden state has dimension H, and we use
    a minibatch size of N.

    Inputs:
    - x: Input data for this timestep, of shape (N, D).
    - prev_h: Hidden state from previous timestep, of shape (N, H)
    - Wx: Weight matrix for input-to-hidden connections, of shape (D, H)
    - Wh: Weight matrix for hidden-to-hidden connections, of shape (H, H)
    - b: Biases of shape (H,)

    Returns a tuple of:
    - next_h: Next hidden state, of shape (N, H)
    """
    next_h = np.tanh(x.dot(Wx) + prev_h.dot(Wh) + b)
    return next_h
```

We see building rnn through imperative programming is both convenient and intuitive.

LSTM

LSTM was introduced by Hochreiter and Schmidhuber[2]. It adds more gates to control the process of forgetting and remembering. LSTM is also quite easy to implement in MinPy like vanilla RNN:

```
class LSTMNet(ModelBase):
    def __init__(self,
                 batch_size=100,
                 input_size=2,  # input dimension
                 hidden_size=100,
                 num_classes=1):
        super(LSTMNet, self).__init__()
        self.add_param(name='Wx', shape=(input_size, 4*hidden_size))\
            .add_param(name='Wh', shape=(hidden_size, 4*hidden_size))\
            .add_param(name='b', shape=(4*hidden_size,))\
            .add_param(name='Wa', shape=(hidden_size, num_classes))\
            .add_param(name='ba', shape=(num_classes,))

    def forward(self, X, mode):
        seq_len = X.shape[1]
        batch_size = X.shape[0]
        hidden_size = self.params['Wh'].shape[0]
        h = np.zeros((batch_size, hidden_size))
        c = np.zeros((batch_size, hidden_size))
```

[2] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

```
        for t in xrange(seq_len):
            h, c = layers.lstm_step(X[:, t, :], h, c,
                                    self.params['Wx'],
                                    self.params['Wh'],
                                    self.params['b'])
        y = layers.affine(h, self.params['Wa'], self.params['ba'])
        return y

    def loss(self, predict, y):
        return layers.l2_loss(predict, y)
```

Layers are defined in `minpy.nn.layers` (here).

The key layer of LSTM is also shown as follows:

```
def lstm_step(x, prev_h, prev_c, Wx, Wh, b):
    """
    Forward pass for a single timestep of an LSTM.

    The input data has dimension D, the hidden state has dimension H, and we use
    a minibatch size of N.

    Inputs:
    - x: Input data, of shape (N, D)
    - prev_h: Previous hidden state, of shape (N, H)
    - prev_c: previous cell state, of shape (N, H)
    - Wx: Input-to-hidden weights, of shape (D, 4H)
    - Wh: Hidden-to-hidden weights, of shape (H, 4H)
    - b: Biases, of shape (4H,)

    Returns a tuple of:
    - next_h: Next hidden state, of shape (N, H)
    - next_c: Next cell state, of shape (N, H)
    """
    N, H = prev_c.shape
    # 1. activation vector
    a = np.dot(x, Wx) + np.dot(prev_h, Wh) + b
    # 2. gate fuctions
    i = sigmoid(a[:, 0:H])
    f = sigmoid(a[:, H:2*H])
    o = sigmoid(a[:, 2*H:3*H])
    g = np.tanh(a[:, 3*H:4*H])
    # 3. next cell state
    next_c = f * prev_c + i * g
    next_h = o * np.tanh(next_c)
    return next_h, next_c
```

The implementation of `lstm_step` is quite straightforward in MinPy.

GRU

GRU was proposed by Cho et al.[3]. It simplifies LSTM by using fewer gates and states. MinPy can also model GRU in an intuitive way:

[3] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," arXiv.org, vol. cs.CL. 04-Jun-2014.

```
class GRUNet(ModelBase):
    def __init__(self,
                 batch_size=100,
                 input_size=2,  # input dimension
                 hidden_size=100,
                 num_classes=1):
        super(GRUNet, self).__init__()
        self.add_param(name='Wx', shape=(input_size, 2*hidden_size))\
            .add_param(name='Wh', shape=(hidden_size, 2*hidden_size))\
            .add_param(name='b', shape=(2*hidden_size,))\
            .add_param(name='Wxh', shape=(input_size, hidden_size))\
            .add_param(name='Whh', shape=(hidden_size, hidden_size))\
            .add_param(name='bh', shape=(hidden_size,))\
            .add_param(name='Wa', shape=(hidden_size, num_classes))\
            .add_param(name='ba', shape=(num_classes,))

    def forward(self, X, mode):
        seq_len = X.shape[1]
        batch_size = X.shape[0]
        hidden_size = self.params['Wh'].shape[0]
        h = np.zeros((batch_size, hidden_size))
        for t in xrange(seq_len):
            h = layers.gru_step(X[:, t, :], h, self.params['Wx'],
                                self.params['Wh'], self.params['b'],
                                self.params['Wxh'], self.params['Whh'],
                                self.params['bh'])
        y = layers.affine(h, self.params['Wa'], self.params['ba'])
        return y

    def loss(self, predict, y):
        return layers.l2_loss(predict, y)
```

Layers are defined in `minpy.nn.layers` (here).

The key layer of GRU is also shown as follows:

```
def gru_step(x, prev_h, Wx, Wh, b, Wxh, Whh, bh):
    """
    Forward pass for a single timestep of an GRU.

    The input data has dimentsion D, the hidden state has dimension H, and we
    use a minibatch size of N.

    Parameters
    ----------
    x : Input data, of shape (N, D)
    prev_h : Previous hidden state, of shape (N, H)
    prev_c : Previous hidden state, of shape (N, H)
    Wx : Input-to-hidden weights for r and z gates, of shape (D, 2H)
    Wh : Hidden-to-hidden weights for r and z gates, of shape (H, 2H)
    b : Biases for r an z gates, of shape (2H,)
    Wxh : Input-to-hidden weights for h', of shape (D, H)
    Whh : Hidden-to-hidden weights for h', of shape (H, H)
    bh : Biases, of shape (H,)

    Returns
    -------
    next_h : Next hidden state, of shape (N, H)
```

```
23
24      Notes
25      -----
26      Implementation follows
27      http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf
28      """
29      N, H = prev_h.shape
30      a = sigmoid(np.dot(x, Wx) + np.dot(prev_h, Wh) + b)
31      r = a[:, 0:H]
32      z = a[:, H:2 * H]
33      h_m = np.tanh(np.dot(x, Wxh) + np.dot(r * prev_h, Whh) + bh)
34      next_h = z * prev_h + (1 - z) * h_m
35      return next_h
```

`gru_step` stays close with `lstm_step` as expected.

Training Result

We trained recurrent networks shown early in this tutorial and use `solver.loss_history` to retrieve the MSE history in the training process. The result is as follows.

rm e We observe that LSTM and GRU are more effective than vanilla RNN due to LSTM and GRU's memory gates. In this particular case, GRU converges much faster than LSTM, probabuly due to fewer parameters.

Reference

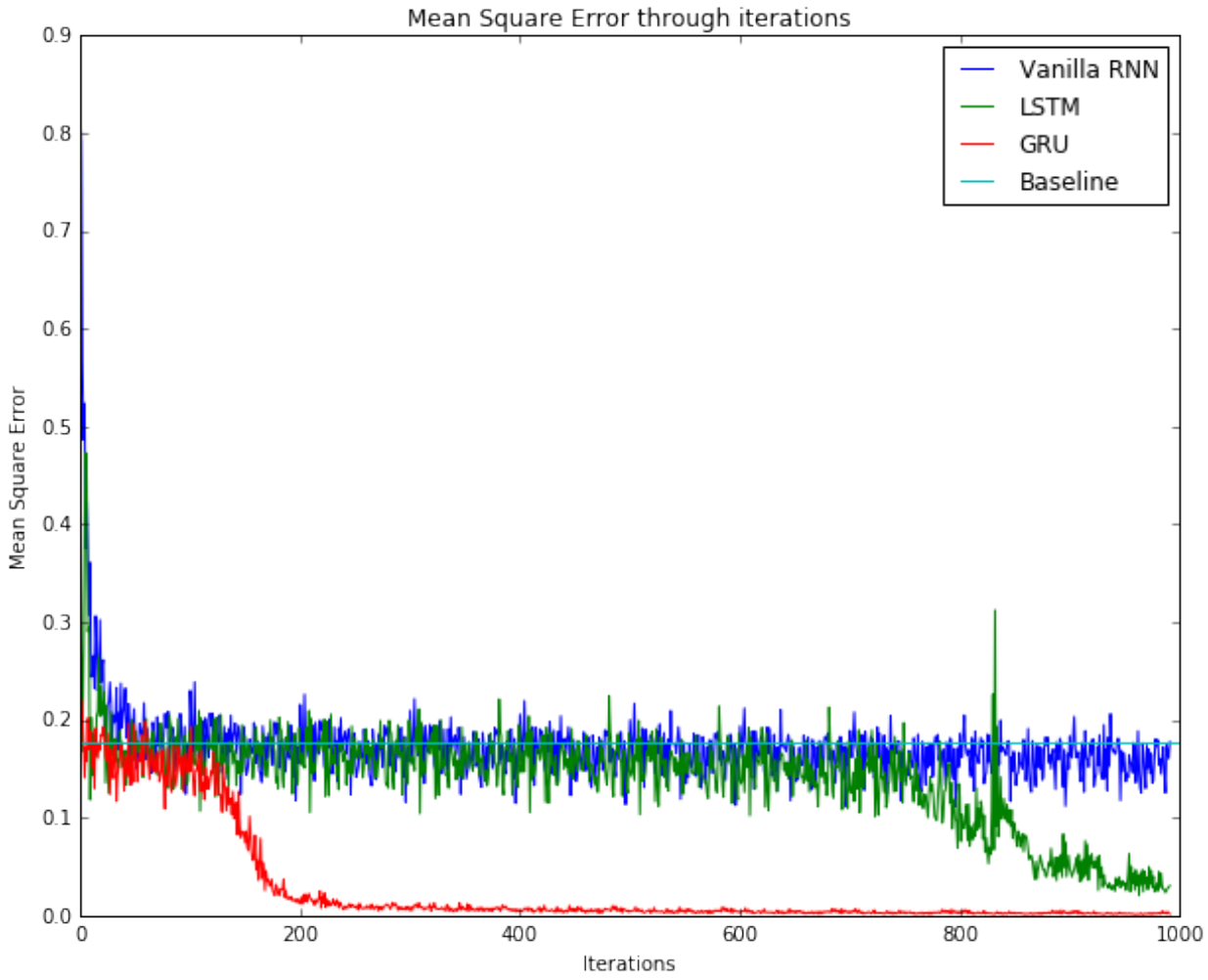


Fig. 8.2: Figure: Result of training vanilla RNN, LSTM, GRU. A baseline of 0.1767 is also included.

CHAPTER 9

RNN on MNIST

This tutorial is also available in step-by-step notebook version on github. Please try it out!

This tutorial is contributed by Kerui Min, CTO of BosonData (links are only available in Chinese).

The previous Adding Problem example demostrated that RNN can handle (sparse) inputs with long-term dependencies. In this example, we apply RNN on the MNIST handwritten digits dataset to further show its effectiveness.

- You can download the data from here.

Fig. 9.1: MNIST Dataset

First, you can define a RNN Network as in the previous section with slight modification:

1. Since this is a classification task, instead of using l2_loss, we employ softmax_loss as our loss function.
2. We initialize 'Wh' as an identity matrix, and 'b' as a zero vector. Therefore, the signal of 'h' flows easily at the begining of the optimization.
3. We set the parameter 'h0' to zero before each forward step, to make sure it doesn't memorize information from previous samples.

```
In [1]: import mxnet
        from minpy.nn.model import ModelBase
        import minpy.nn.layers as layers

        class RNNNet(ModelBase):
            def __init__(self,
                         batch_size=100,
                         input_size=1,
                         hidden_size=64,
                         num_classes=10):
                super(RNNNet, self).__init__()
                self.add_param(name='h0', shape=(batch_size, hidden_size))\
                    .add_param(name='Wx', shape=(input_size, hidden_size))\
                    .add_param(name='Wh', shape=(hidden_size, hidden_size),
                               init_rule='constant',
```

```
                    init_config={'value': np.identity(hidden_size)})\
            .add_param(name='b', shape=(hidden_size,),
                    init_rule='constant',
                    init_config={'value': np.zeros(hidden_size)})\
            .add_param(name='Wa', shape=(hidden_size, num_classes))\
            .add_param(name='ba', shape=(num_classes,))

    def forward(self, X, mode):
        seq_len = X.shape[1]
        self.params['h0'][:] = 0
        h = self.params['h0']
        for t in xrange(seq_len):
            h = layers.rnn_step(X[:, t, :], h, self.params['Wx'],
                        self.params['Wh'], self.params['b'])
        y = layers.affine(h, self.params['Wa'], self.params['ba'])

        return y

    def loss(self, predict, y):
        return layers.softmax_loss(predict, y)
```

The training data consists of 60000 samples, each of which is a 784-dimensional uint8 vector, representing a 28*28 grey image. Usually, people treat each image as a 784-d vector, and build classifiers based on this representation. In this case, however, we treat each 784-d vector as a sequence.

Imagine that instead of reading the whole image, at each step, we are only allowed to read few pixels (a patch) of the given image to determine which class it belongs to at the end. This is much more difficult, as the final decision cannot be made with one or two patches.

To make the dataset easiler to learn, we need to normalize the data before training:

```
In [2]: import joblib
        import numpy as np
        data = joblib.load("data/mnist.dat")

        mean = np.mean(data["train_data"], axis=0)
        std = np.std(data["train_data"] - mean, axis=0)
        data["train_data"] = (data["train_data"][:] - mean)/(std+1e-5)
        data["test_data"] = (data["test_data"][:] - mean)/(std+1e-5)
```

As an example, we set the size of each patch to 7. Hence, the length of each sample is 112 (784/7). RNN needs to classify each sample after reading the whole 112 patches. Notice that we only use 5000 samples for training, 1000 for testing, for faster demonstration.

```
In [ ]: from minpy.nn.io import NDArrayIter
        from minpy.nn.solver import Solver

        BATCH = 50
        INPUT_DIM = 7
        HIDDEN_DIM = 128

        _, dim = data["train_data"].shape
        seq_len = dim / INPUT_DIM

        train_iter = NDArrayIter(data["train_data"][:5000].reshape(5000, seq_len, INPUT_DIM),
                                 data["train_label"][:5000],
                                 batch_size=BATCH,
                                 shuffle=True)

        test_iter = NDArrayIter(data["test_data"][:1000].reshape(1000, seq_len, INPUT_DIM),
                                data["test_label"][:1000],
```

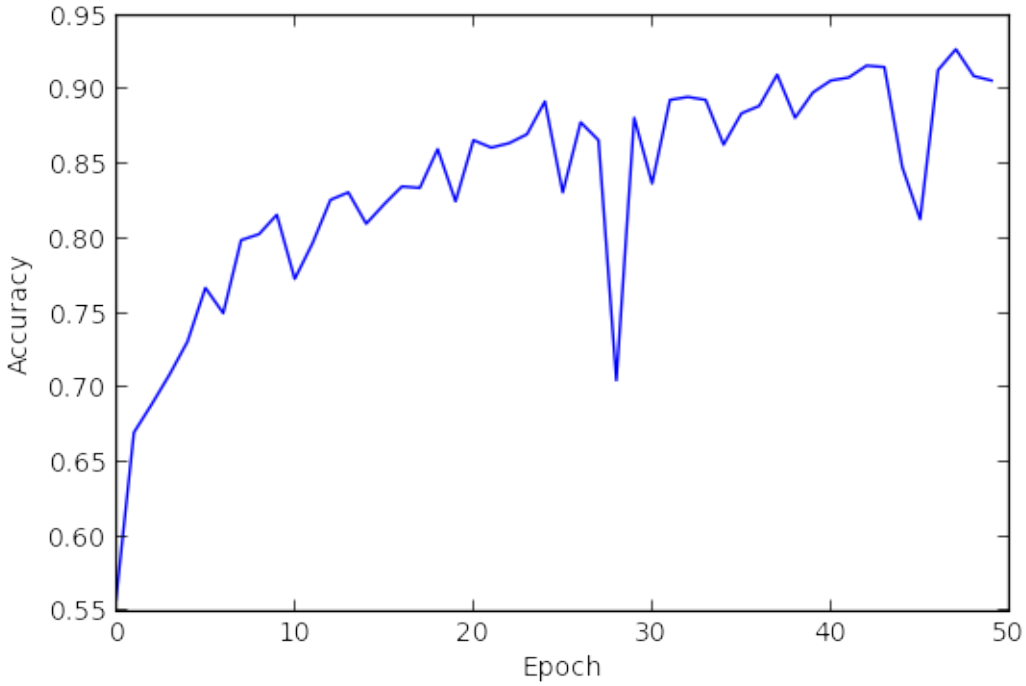
```
                                batch_size=BATCH,
                                shuffle=False)

        model = RNNNet(batch_size=BATCH, input_size=INPUT_DIM, hidden_size=HIDDEN_DIM)

        solver = Solver(model,
                        train_iter,
                        test_iter,
                        num_epochs=100,
                        init_rule='xavier',
                        update_rule='rmsprop',
                        optim_config={
                                'learning_rate': 0.0002,
                        },
                        verbose=True,
                        print_every=10)

        solver.init()
        solver.train()
```

```
(Iteration 1 / 10000) loss: 2.817845
(Iteration 11 / 10000) loss: 1.965365
(Iteration 21 / 10000) loss: 1.868933
(Iteration 31 / 10000) loss: 1.466141
(Iteration 41 / 10000) loss: 1.434501
(Iteration 51 / 10000) loss: 1.485497
(Iteration 61 / 10000) loss: 1.249973
(Iteration 71 / 10000) loss: 1.580822
(Iteration 81 / 10000) loss: 1.350305
(Iteration 91 / 10000) loss: 1.369664
(Epoch 1 / 100) train acc: 0.509000; val_acc: 0.470000
(Iteration 101 / 10000) loss: 1.350214
(Iteration 111 / 10000) loss: 1.396372
(Iteration 121 / 10000) loss: 1.294581
(Iteration 131 / 10000) loss: 1.278551
(Iteration 141 / 10000) loss: 1.132157
(Iteration 151 / 10000) loss: 1.147118
(Iteration 161 / 10000) loss: 0.856366
(Iteration 171 / 10000) loss: 1.439825
(Iteration 181 / 10000) loss: 1.113218
(Iteration 191 / 10000) loss: 1.132181
(Epoch 2 / 100) train acc: 0.609000; val_acc: 0.555000
(Iteration 201 / 10000) loss: 1.147544
(Iteration 211 / 10000) loss: 1.187811
(Iteration 221 / 10000) loss: 1.008041
(Iteration 231 / 10000) loss: 1.207148
(Iteration 241 / 10000) loss: 1.075240
(Iteration 251 / 10000) loss: 1.074992
(Iteration 261 / 10000) loss: 0.694210
(Iteration 271 / 10000) loss: 1.378169
(Iteration 281 / 10000) loss: 0.997993
(Iteration 291 / 10000) loss: 1.032932
(Epoch 3 / 100) train acc: 0.614000; val_acc: 0.595000
(Iteration 301 / 10000) loss: 1.046198
(Iteration 311 / 10000) loss: 1.099993
(Iteration 321 / 10000) loss: 0.825924
(Iteration 331 / 10000) loss: 1.125248
(Iteration 341 / 10000) loss: 0.977916
(Iteration 351 / 10000) loss: 0.967498
```

```
(Iteration 361 / 10000) loss: 0.586793
(Iteration 371 / 10000) loss: 1.243312
(Iteration 381 / 10000) loss: 0.900940
(Iteration 391 / 10000) loss: 1.037108
(Epoch 4 / 100) train acc: 0.662000; val_acc: 0.630000
(Iteration 401 / 10000) loss: 0.989269
(Iteration 411 / 10000) loss: 0.970790
(Iteration 421 / 10000) loss: 0.732133
(Iteration 431 / 10000) loss: 1.103309
(Iteration 441 / 10000) loss: 0.793366
(Iteration 451 / 10000) loss: 0.851175
(Iteration 461 / 10000) loss: 0.745199
(Iteration 471 / 10000) loss: 1.086999
(Iteration 481 / 10000) loss: 0.754697
(Iteration 491 / 10000) loss: 0.927628
(Epoch 5 / 100) train acc: 0.719000; val_acc: 0.674000
(Iteration 501 / 10000) loss: 0.899225
(Iteration 511 / 10000) loss: 0.835000
(Iteration 521 / 10000) loss: 0.632679
(Iteration 531 / 10000) loss: 0.956496
(Iteration 541 / 10000) loss: 0.699549
(Iteration 551 / 10000) loss: 0.734873
(Iteration 561 / 10000) loss: 0.666981
(Iteration 571 / 10000) loss: 0.838450
(Iteration 581 / 10000) loss: 0.668199
(Iteration 591 / 10000) loss: 0.885664
(Epoch 6 / 100) train acc: 0.749000; val_acc: 0.680000
(Iteration 601 / 10000) loss: 0.853983
(Iteration 611 / 10000) loss: 0.760786
(Iteration 621 / 10000) loss: 0.512938
(Iteration 631 / 10000) loss: 0.808366
(Iteration 641 / 10000) loss: 0.572112
```

A typical learning curve for this problem look like the following figure.

```
In [2]: %matplotlib inline
        import matplotlib.pyplot as plt

        hist = solver.val_acc_history

        """
        hist = [0.558, 0.67, 0.689, 0.709, 0.731, 0.767, 0.75, 0.799, 0.803, 0.816, \
                0.773, 0.797, 0.826, 0.831, 0.81, 0.823, 0.835, 0.834, 0.86, 0.825, \
                0.866, 0.861, 0.864, 0.87, 0.892, 0.831, 0.878, 0.866, 0.705, 0.881, \
                0.837, 0.893, 0.895, 0.893, 0.863, 0.884, 0.889, 0.91, 0.881, 0.898, \
                0.906, 0.908, 0.916, 0.915, 0.848, 0.813, 0.913, 0.927, 0.909, 0.906]
        """
        plt.plot(hist)
        plt.xlabel("Epoch")
        plt.ylabel("Accuracy")
        plt.show()
```



Possible assigments

1. Instead of using vanilla RNN, try LSTM and GRU.
2. Verify the effectiveness of data normalization.
3. We can you conclude from the above learning curve figure, can to change learning rate to improve it?
4. Add l_2 regularization term to the RNN model.

CHAPTER 10

Reinforcement learning with policy gradient

Deep Reinforcement Learning (RL) is another area where deep models are used. In this example, we implement an agent that learns to play Pong, trained using policy gradients. Since we are using MinPy, we avoid the need to manually derive gradient computations, and can easily train on a GPU.

The example is based on the problem and model described in Deep Reinforcement Learning: Pong from Pixels , which contains an introduction and background to the ideas discussed here.

The training setups in reinforcement learning often differ based on the approach used, making MinPy's flexibility a great choice for prototyping RL models. Unlike the standard supervised setting, with the policy gradient approach the training data is generated during training by the environment and the actions that the agent chooses, and stochasticity is introduced in the model.

Specifically, we implement a `PolicyNetwork` that learns to map states (i.e. visual frames of the Pong game) to actions (i.e. 'move up' or 'move down'). The network is trained using the `RLPolicyGradientSolver`.

See here for the full implementation.

PolicyNetwork

The forward pass of the network is separated into two steps:

1. Compute a probability distribution over actions, given a state.
2. Choose an action, given the distribution from (1).

These steps are implemented in the `forward` and `choose_action` functions, respectively:

```
class PolicyNetwork(ModelBase):
    def __init__(self,
                 preprocessor,
                 input_size=80*80,
                 hidden_size=200,
                 gamma=0.99):  # Reward discounting factor
        super(PolicyNetwork, self).__init__()
        self.preprocessor = preprocessor
```

```
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.gamma = gamma
        self.add_param('w1', (hidden_size, input_size))
        self.add_param('w2', (1, hidden_size))

    def forward(self, X):
        """Forward pass to obtain the action probabilities for each observation in
↪`X`."""
        a = np.dot(self.params['w1'], X.T)
        h = np.maximum(0, a)
        logits = np.dot(h.T, self.params['w2'].T)
        p = 1.0 / (1.0 + np.exp(-logits))
        return p

    def choose_action(self, p):
        """Return an action `a` and corresponding label `y` using the probability
↪float `p`."""
        a = 2 if numpy.random.uniform() < p else 3
        y = 1 if a == 2 else 0
        return a, y

```

In the `forward` function, we see that the model used is a simple feed-forward network that takes in an observation `X` (a preprocessed image frame) and outputs the probability `p` of taking the 'move up' action.

The `choose_action` function then draws a random number and selects an action according to the probability from `forward`.

For the `loss` function, we use cross-entropy but multiply each observation's loss by the associated discounted reward, which is the key step in forming the policy gradient:

```
    def loss(self, ps, ys, rs):
        step_losses = ys * np.log(ps) + (1.0 - ys) * np.log(1.0 - ps)
        return -np.sum(step_losses * rs)
```

Note that by merely defining the `forward` and `loss` functions in this way, MinPy will be able to automatically compute the proper gradients.

Lastly, we define the reward discounting approach in `discount_rewards`, and define the preprocessing of raw input frames in the `PongPreprocessor` class:

```
    def discount_rewards(self, rs):
        drs = np.zeros_like(rs).asnumpy()
        s = 0
        for t in reversed(xrange(0, len(rs))):
            # Reset the running sum at a game boundary.
            if rs[t] != 0:
                s = 0
            s = s * self.gamma + rs[t]
            drs[t] = s
        drs -= np.mean(drs)
        drs /= np.std(drs)
        return drs

class PongPreprocessor(object):
    ... initialization ...
    def preprocess(self, img):
```

```
        ... preprocess and flatten the input image...
        return diff
```

See here for the full implementation.

RLPolicyGradientSolver

The `RLPolicyGradientSolver` is a custom `Solver` that can train a model that has the functions discussed above. In short, its training approach is:

1. Using the current model, play an episode to generate observations, action labels, and rewards.
2. Perform a forward and backward pass using the observations, action labels, and rewards from (1).
3. Update the model using the gradients found in (2).

Under the covers, for step (1) the `RLPolicyGradientSolver` repeatedly calls the model's `forward` and `choose_action` functions at each step, then uses `forward` and `loss` functions for step 2.

See here for the full implementation.

Training

Run pong_model.py to train the model. The average running reward is printed, and should generally increase over time. Here is a video showing the agent (on the right) playing after 18,000 episodes of training:

While this example uses a very simple feed-forward network, MinPy makes it easy to experiment with more complex architectures, alternative pre-processing, weight updates, or initializations!

Dependencies

Note that Open AI Gym is used for the environment.

CHAPTER 11

Improved RL with Parallel Advantage Actor-Critic

The example above shows a basic model trained using the simplest policy gradient estimator, known as REINFORCE[1]. This gradient estimator is known to have high variance, resulting in slow training. Moreover, with the model and training process above we are limited to obtaining trajectories from a single environment at a time.

In this tutorial, we will address these issues by using an alternative gradient estimator with lower variance and introducing parallelism. In the end, we will be able to train a model to solve Pong in around an hour on a MacBook Pro*[0]. All code can be found here.

We use ideas from the Asynchronous Actor-Critic (A3C)[2] algorithm. Broadly speaking A3C has two key features: asynchronous training that simultaneously uses trajectories from multiple environments, and the use of an estimated advantage function to reduce the variance of the policy gradient estimator.

Parallel training

A3C consists of a global model, with separate worker processes that have a local copy of the model. Each worker collects trajectories from its own environment, makes updates to the global model, then syncs its local model copy. Thus with *n* worker processes, updates are made using trajectories from *n* environments. Another feature of A3C is that updates are made every *t_max* (e.g. 20) steps instead of performing a single update per episode.

One insight is that we can achieve similar behavior as A3C without using multiple processes or copies of the model, when interacting with the environment is not expensive.

Instead, we can maintain *n* copies of the environment and at each step sequentially obtain *n* observations, then predict *n* different actions with the model using a single forward pass:

[1] Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 8(3):229–256, 1992.

[0] Consistent with the Open AI A3C implementation[4], we use the PongDeterministic-V3 environment, which uses a frame-skip of 4. For this experiment, we define 'solved' as achieving a running average score of 20 out of 21 (computed using the previous 100 episodes). Note that for this version of Pong, humans were only able to achieve a score of -11[5].

[2] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Proceedings of the 33rd International Conference on Machine Learning (ICML), pp. 1928–1937, 2016.

```
while not all_done:
    # Stack all the observations from the current time step.
    step_xs = np.vstack([o.ravel() for o in observations])

    # Get actions and values for all environments in a single forward pass.
    step_ps, step_vs = agent.forward(step_xs)
    step_as = agent.act(step_ps)

    # Step each environment whose episode has not completed.
    for i, env in enumerate(envs):
        if not done[i]:
            obs, r, done[i], _ = env.step(step_as[i])

            # Record the observation, action, value, and reward in the buffers.
            env_xs[i].append(step_xs[i])
            env_as[i].append(step_as[i])
            env_vs[i].append(step_vs[i])
            env_rs[i].append(r)

            # Store the observation to be used on the next iteration.
            observations[i] = preprocessors[i].preprocess(obs)

    ...
```

Then we can assemble the trajectories obtained from the *n* environments into a batch and update the model using a single backward pass:

```
# Perform update and clear buffers. Occurs after `t_max` steps
# or when all episodes are complete.
agent.train_step(env_xs, env_as, env_rs, env_vs)
env_xs, env_as = _2d_list(num_envs), _2d_list(num_envs)
env_rs, env_vs = _2d_list(num_envs), _2d_list(num_envs)
```

Thus the training is parallel in the sense that the trajectories are processed in the forward and backward passes for all environments in parallel. See train.py for the full implementation.

Since updates occur every *t_max* steps, the memory requirement for a forward and backward pass only increases marginally as we add additional environments, allowing many parallel environments. The number of environments is also not limited by the number of CPU cores.

One difference with A3C is that since the updates are not performed asynchronously, the most recent model parameters are always used to obtain trajectories from each environment. However, since the environments are independent the policy still explores different trajectories in each environment. Intuitively, this means that the overall set of samples used for training becomes more 'diverse' as we introduce more environments.

Improved gradient estimator

The 'Advantage Actor-Critic' in A3C refers to using an estimated advantage function to reduce the variance of the policy gradient estimator. In the REINFORCE policy gradient, the discounted return is used to decide which direction to update the parameters:

$$\nabla_\theta \log \pi_\theta(a_t|s_t) R_t$$

Where R_t is the total discounted return from time step t and $\pi_\theta(a_t|s_t)$ is the probability of taking action a_t from state s_t, given by the policy π_θ.

Alternatively, with an advantage function the direction of parameter updates is based on whether taking a certain action leads to higher expected return than average, as measured by the advantage function $A(s_t, a_t)$:

$$\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

Where $Q(s, a)$ is the action-value function that gives the expected return of taking action a_t from state s_t, and $V(s_t)$ is the state-value function, which gives the expected return from state s_t.

Although the function $Q(s, a)$ is unknown, we can approximate it using the discounted returns R, resulting in an estimated advantage function:

$$\tilde{A}(s_t, a_t) = R_t - V(s_t)$$

An alternative estimator of the advantage function known as the Generalized Advantage Estimator (GAE)[3] has been developed by Schulman et al., which we implement here. The Generalized Advantage Estimator is as follows:

$$\tilde{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Where r_t is the reward obtained at timestep t, and $V(s_t)$ is the state-value function, which we approximate in practice. γ and λ are fixed hyper-parameters in $(0, 1)$.

To implement GAE, we therefore need to (a) implement the state-value function, (b) compute the advantages according to the equation above, and (c) modify the loss so that it uses the advantages and trains the state-value function.

(a) State-Value Function

We approximate the state-value function by modifying the policy network to produce an additional output, namely the value of the input state. Here, the state-value function is simply a linear transformation of the same features used to compute the policy logits:

```
def forward(self, X):
    a = np.dot(self.params['fc1'], X.T)
    h = np.maximum(0, a)

    # Compute the policy's distribution over actions.
    logits = np.dot(h.T, self.params['policy_fc_last'].T)
    ps = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    ps /= np.sum(ps, axis=1, keepdims=True)

    # Compute the value estimates.
    vs = np.dot(h.T, self.params['vf_fc_last'].T) + self.params['vf_fc_last_bias']
    return ps, vs
```

(b) Compute Advantages

After assembling a batch of trajectories, we compute the discounted rewards used to train the value function, and the δ terms and the advantage estimates for the trajectories from each environment.

[3] Schulman, John, Moritz, Philipp, Levine, Sergey, Jordan, Michael, and Abbeel, Pieter. High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438, 2015b.

```
for i in range(num_envs):
    # Compute discounted rewards with a 'bootstrapped' final value.
    rs_bootstrap = [] if env_rs[i] == [] else env_rs[i] + [env_vs[i][-1]]
    discounted_rewards.extend(self.discount(rs_bootstrap, gamma)[:-1])

    # Compute advantages for each environment using Generalized Advantage Estimation;
    # see eqn. (16) of [Schulman 2016].
    delta_t = env_rs[i] + gamma*env_vs[i][1:] - env_vs[i][:-1]
    advantages.extend(self.discount(delta_t, gamma*lambda_))
```

Notice the subtlety in the way we discount the rewards that are used to train the state-value function. If a trajectory ended mid-episode, we won't know the actual future return of the trajectory's last state. Therefore we 'bootstrap' the future return for the last state by using the value function's last estimate.

(c) Loss Function

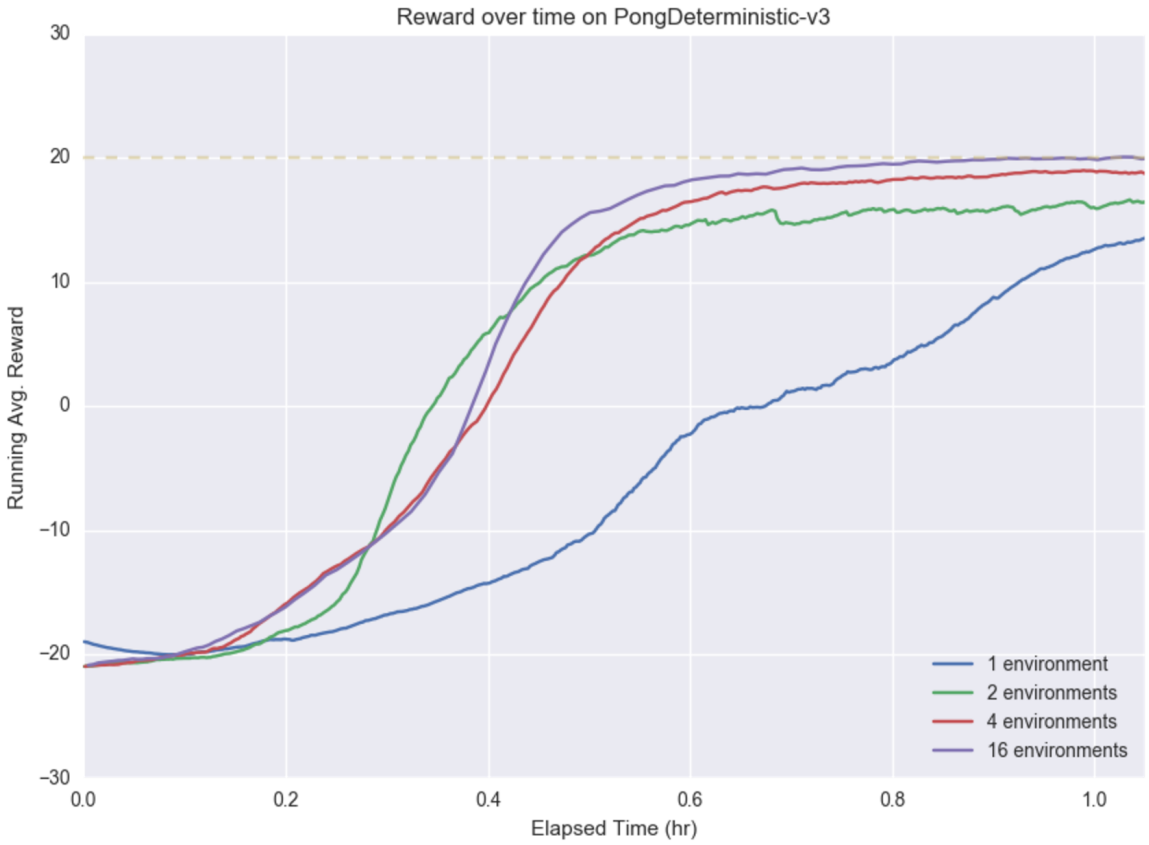
The loss function then uses the advantage estimates. A term is added to train the value function's weights, using L2-loss with the observed discounted rewards as the ground truth. Finally, as noted in the A3C paper, an entropy term can be added that discourages the policy from collapsing to a single action for a given state, which improves exploration.

```
def loss(self, ps, actions_one_hot, vs, rs, advs):
    # Distribution over actions, prevent log of zero.
    ps = np.maximum(1.0e-5, np.minimum(1.0 - 1e-5, ps))

    # Policy gradient loss.
    policy_grad_loss = -np.sum(np.log(ps) * actions_one_hot * advs)

    # Value function loss.
    vf_loss = 0.5 * np.sum((vs - rs) ** 2)

    # Entropy regularizer.
    entropy = -np.sum(ps * np.log(ps))

    # Weight value function loss and entropy, and combine into the final loss.
    loss_ = policy_grad_loss + self.config.vf_wt * vf_loss - self.config.entropy_wt *␣
↪entropy
    return loss_
```

See model.py for the full implementation.

Experiments

To see the effects of advantage estimation and parallel training, we train the model with 1, 2, 4, and 16 environments.



The plot shows the running average reward over time for various numbers of environments during the first hour of training. We can see that the improved gradient estimator allows all models to train quickly. Using 16 environments, the game is solved in around an hour.

We see that increasing the number of environments increases convergence speed in terms of wall-clock time; using 4 or 16 environments results in approaching the faster than using 2 environments or a single environment. This indicates that the increased number of samples per update outweighs the cost of processing the samples.

Moreover, training stability improves as more environments are used, as shown by the smooth, monotonic improvement with 16 environments versus the jagged, noisy improvement with 1 and 2 environments. One possible reason is that with more parallel environments, the samples used to update the policy are more diverse.

While a detailed analysis is outside the scope of this tutorial, we can see that the parallel Advantage Actor-Critic approach can train a model on the Pong environment in a fast and robust manner.

Note that we have kept the structure of the policy network the same as before; it is still a simple feed-forward network with 1 hidden layer.

Running

To train the model with default parameters, run:

```
python train.py
```

Here is a video showing the model after around an hour of training:

Feel free to experiment with other environments (which may require additional tuning), network architectures, or to use the code as a starting point for further research. For instance, we can also solve the CartPole environment:

```
python train.py --env-type CartPole-v0
```

Reference

CHAPTER 12

Complete model builder guide

MinPy's model builder simplifies the declaration of networks, in styles similar to Keras. Networks expressed with model builder is compatible with examples listed elsewehre in other parts of the tutorial.

Get started

MinPy's model builder consists of classes describing various neural network architectures.

- `Sequence` class enables users to create feedforward networks.
- Other classes representing layers of neural networks, such as `Affine` and `Convolution`.

The following code snippet demonstrates how a few lines of specification with model builder removes the need of defining a complete network from scratch. The full code is here

```
'''
  Minpy model builder example.

  For details about how to train a model with solver, please refer to:
    http://minpy.readthedocs.io/en/latest/tutorial/complete.html

  More models are available in minpy.nn.model_gallery.
'''

#... other import modules
import minpy.nn.model_builder as builder

class TwoLayerNet(ModelBase):
    def __init__(self):
        super(TwoLayerNet, self).__init__()
        # Define model parameters.
        self.add_param(name='w1', shape=(flattened_input_size, hidden_size)) \
            .add_param(name='b1', shape=(hidden_size,)) \
            .add_param(name='w2', shape=(hidden_size, num_classes)) \
            .add_param(name='b2', shape=(num_classes,))
```

```
21
22      def forward(self, X, mode):
23          # Flatten the input data to matrix.
24          X = np.reshape(X, (batch_size, 3 * 32 * 32))
25          # First affine layer (fully-connected layer).
26          y1 = layers.affine(X, self.params['w1'], self.params['b1'])
27          # ReLU activation.
28          y2 = layers.relu(y1)
29          # Second affine layer.
30          y3 = layers.affine(y2, self.params['w2'], self.params['b2'])
31          return y3
32
33      def loss(self, predict, y):
34          # Compute softmax loss between the output and the label.
35          return layers.softmax_loss(predict, y)
36
37  def main(args):
38      # Define a 2-layer perceptron
39      MLP = builder.Sequential(
40          builder.Affine(512),
41          builder.ReLU(),
42          builder.Affine(10)
43      )
44
45      # Cast the definition to a model compatible with minpy solver
46      model = builder.Model(MLP, 'softmax', (3 * 32 * 32,))
47
48      # ....
49      # the above is equivalent to use the TwoLayerNet, as below (uncomment )
50      # model = TwoLayerNet()
51
52      solver = Solver(model,
53      # ...
```

Arbitrarily complex networks could be constructed by combining these classes in a nested way. Please refer to the model gallery to discover how to easily declare complex networks such as ResNet with model builder.

Customize model builder layers

MinPy's model builder is designed to be extended easily. To create customized layers, one only needs to inherit classes from `minpy.nn.model_builder.Module` class and implement several inherited functions. These functions are `forward`, `output_shape`, `parameter_shape` and `parameter_settings`.

- `forward` receives input data and a dictionary containing the parameters of the network, and generates output data. It can be implemented by Numpy syntax, layers provided in `minpy.nn.layers`, MXNet symbols or their combination as described in *Complete solver and optimizer guide*.
- `output_shape` returns the layer's output shape given the input shape. Please be aware that the shapes should be tuples specifying shape of one training sample, i.e. the shape of CIFAR-10 data is either (3072,) or (3, 32, 32).
- `parameter_shape` returns a dictionary that includes the shapes of all parameters of the layer. One could pass more information to MinPy's solver in `parameter_settings`.

It is optional to implement `parameter_settings`. The model builder script should be self-explanatory.

CHAPTER 13

Learning Deep Learning with MinPy

We believe there is no easy shortcut if one wishes to master the concepts of deep learning and make novel contributions. A very nice introduction to deep learning is CS231n from Stanford, which contains self-paced lectures and notes. This is a great course, particularly because of its hands-on homeworks.

We tailor-made a modified version of the coursework using MinPy, for assignment 2 and 3, fixing exactly the two problems that the original NumPy coureware lacks: the support to run on GPU and auto-gradient. The new homework can be found here: CS231n Homework links

This new courseware has been pilot in ShanghaiTech and Shanghai JiaoTong universities. We want to thank Karpathy for giving the permission, and the instructors for trying it out. Please send us feedbacks if you run into issues.

CHAPTER 14

Select Policy for Operations

MinPy integrates MXNet NDArray and NumPy into a seamless system. For a single operation, it may have MXNet implementation, pure NumPy CPU implementation, or both of them. MinPy utilizes a policy system to determine which implementation will be applied. MinPy currently has three build-in policies:

1. `prefer_mxnet` **[Default]**: Prefer MXNet. Use NumPy as a transparent fallback.
2. `only_numpy`: Only use NumPy.
3. `only_mxnet`: Only use MXNet.

The policy is global. To change the policy, use `minpy.set_global_policy`. For example:

```
import minpy.numpy as np
import minpy
minpy.set_global_policy('only_numpy')
```

It is worth mentioning that `minpy.set_global_policy` only accepts strings of policy names.

`@minpy.wrap_policy`: Wrap a Function under Specific Policy

`@minpy.wrap_policy` is a wrapper that wraps a function under specific policy. It only accpets policy objects. For example

```
import minpy.numpy as np
import minpy
import minpy.dispatch.policy as policy
minpy.set_global_policy('prefer_mxnet')


@minpy.wrap_policy(policy.OnlyNumPyPolicy())
def foo(a, b):
    return np.log(a + b)

a = np.ones((2, 2))
```

```
b = np.zeros((2, 2))

# a + b runs under PreferMXNetPolicy
c = a + b

# foo runs under OnlyNumPyPolicy.
c = foo(np.ones((2, 2)), np.zeros((2, 2)))

# a + b runs under PreferMXNetPolicy again
c = a + b
```

CHAPTER 15

Show Operation Dispatch Statistics

In the default `PreferMXNetPolicy()`, the operation prefers to MXNet implementation and trasparently falls back to NumPy if MXNet hasn't defined it. However, given a network, usually with dozens of operations, it's hard to know which operation is not supported in MXNet and running in NumPy. In some cases, the NumPy running operation is the bottleneck of the program and leads to slow training/inferring speed, especially when MXNet operations are running in GPU. When that happens, a useful speed-up approach is to replace NumPy operation to a MXNet defined one.

To better locate those operations, method `show_op_stat()` is provided to show the dispatch statistics information, i.e. which operation and how many operations are executed in MXNet and the same for NumPy. For example, the following network is trained in the default `PreferMXNetPolicy()`. `show_op_stat()` is called after training is finished

```
from minpy.core import grad
import minpy.numpy as np
import minpy.numpy.random as random
import minpy.dispatch.policy as policy

def test_op_statistics():

    def sigmoid(x):
        return 0.5 * (np.tanh(x / 2) + 1)


    def predict(weights, inputs):
        return sigmoid(np.dot(inputs, weights))


    def training_loss(weights, inputs):
        preds = predict(weights, inputs)
        label_probabilities = preds * targets + (1 - preds) * (1 - targets)
        l = -np.sum(np.log(label_probabilities))
        return l


    def training_accuracy(weights, inputs):
        preds = predict(weights, inputs)
```

```
        error = np.count_nonzero(
            np.argmax(
                preds, axis=1) - np.argmax(
                    targets, axis=1))
        return (256 - error) * 100 / 256.0


    xshape = (256, 500)
    wshape = (500, 250)
    tshape = (256, 250)
    inputs = random.rand(*xshape) - 0.5
    targets = np.zeros(tshape)
    truth = random.randint(0, 250, 256)
    targets[np.arange(256), truth] = 1
    weights = random.rand(*wshape) - 0.5

    training_gradient_fun = grad(training_loss)

    for i in range(30):
        print('Trained accuracy #{}: {}%'.format(i, training_accuracy(weights,
                                                                      inputs)))
        gr = training_gradient_fun(weights, inputs)
        weights -= gr * 0.01

    # Print Op Statistics Info
    np.policy.show_op_stat()

if __name__ == "__main__":
    test_op_statistics()
```

Here's the statistics result:

```
MXNET op called times:
  divide : 90
  sum : 30
  negative : 30
  add : 120
  zeros : 1
  multiply : 180
  subtract : 152
  dot : 60
  log : 30

NUMPY op called times:
  rand : 2
  tanh : 60
  argmax : 60
  randint : 1
  arange : 1
  count_nonzero : 30

Total Dispatch Proportion: 81.8% in MXNet, 18.2% in NumPy
```

CHAPTER 16

Select Context for MXNet

MinPy as a system fully integrates MXNet, enjoys MXNet's flexibility to run operations on CPU and different GPUs. The `Context` in MinPy determines where MXNet operations run. MinPy has two built-in `Context` in `minpy.context`:

1. `minpy.context.cpu()` [**Default**]: runs on CPU. No `device_id` needed for CPU context.
2. `minpy.context.gpu(device_id)`: runs on GPU specified by `device_id`. Usually `gpu(0)` is the first GPU in the system. Note that GPU context is only available with MXNet complied with GPU support.

There are two functions to set context:

1. use `minpy.context.set_context` to set global context. we encourage you to use it at the header of program. For example:

```
from minpy.context import set_context, cpu, gpu
set_context(gpu(0))  # set the global context as gpu(0)
```

It is worth mentioning that `minpy.context.set_context` only accepts instances of context classes.

The context is active in the lifetime of the current imported MinPy module, which is usually the scope of the current file.

2. use `with` statement to set local context. For example:

```
with gpu(0):
    x_gpu0 = random.rand(32, 64) - 0.5
    y_gpu0 = random.rand(64, 32) - 0.5
    z_gpu0 = np.dot(x_gpu0, y_gpu0)
with gpu(1):
    x_gpu1 = random.rand(32, 64) - 0.5
    y_gpu1 = random.rand(64, 32) - 0.5
    z_gpu1 = np.dot(x_gpu1, x_gpu1)
```

The code snippet will run on `gpu0` or `gpu1` decided by the device information in the with statement. With this feature, you can achive distributing computation on multi-device.

CHAPTER 17

Customized Operator

Sometimes it is useful to define a customized operator with its own derivatives. Here is a comprehensive example of customized operator: Example.

The following code is taken from our example.

```
@customop('numpy')
def my_softmax(x, y):
    probs = numpy.exp(x - numpy.max(x, axis=1, keepdims=True))
    probs /= numpy.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    loss = -numpy.sum(numpy.log(probs[numpy.arange(N), y])) / N
    return loss


def my_softmax_grad(ans, x, y):
    def grad(g):
        N = x.shape[0]
        probs = numpy.exp(x - numpy.max(x, axis=1, keepdims=True))
        probs /= numpy.sum(probs, axis=1, keepdims=True)
        probs[numpy.arange(N), y] -= 1
        probs /= N
        return probs

    return grad

my_softmax.def_grad(my_softmax_grad)
```

As in the example, the forward pass of the operator is defined in a normal python function. The only discrepancy is the decorator `@customop('numpy')`. The decorator will change the function into a class instance with the same name as the function.

The decorator `customop` has two options:

- `@customop('numpy')`: It assumes the arrays in the input and the output of the user-defined function are both NumPy arrays.

- `@customop('mxnet')`: It assumes the arrays in the input and the output of the user-defined function are both MXNet NDArrays.

Register derivatives for customized operator

To register a derivative, you first need to define a function that takes output and inputs as parameters and returns a function, just as the example above. The returned function takes upstream gradient as input, and outputs downstream gradient. Basically, the returned function describes how to modify the gradient in the backpropagation process on this specific customized operator w.r.t. a certain variable.

After derivative function is defined, simply register the function by `def_grad` as shown in the example above. In fact, `my_softmax.def_grad(my_softmax_grad)` is the shorthand of `my_softmax.def_grad(my_softmax_grad, argnum=0)`. Use `argnum` to specify which variable to bind with the given derivative.

CHAPTER 18

MinPy IO

This tutorial introduces IO part of MinPy. We will describe the concepts of Dataset, DataIter and how to use MXNet IO module in MinPy code.

Dataset

Dataset is a collection of samples. Each sample may have multiple entries, each representing a particular input variable for a certain learning task. Taking image classification as an example, each sample may contain one entry for the image, and another for the label. This is in the context of supervised learning. Of course, this is not the only game in town. For example, Policy gradient reinforcement learning is an interesting departure, where label is generated during computing, thus there's no label entry in the dataset.

The source of a dataset can vary: a list of images for vision task, rows of text for NLP task, etc. The task of the IO module is to turn the source dataset into the data structure that can be used by the learning system. In MinPy, the data structure for learning is `minpy.NDArray`, whereas `numpy.ndarray`, `mxnet.NDArray` and `minpy.NDArray` can all serve as source dataset. These three kinds of source are easy to produce in pure Python code, thus there's no black box in preparing the dataset. Please refer to data_utils.py to see how to prepare raw data for MinPy IO.

If you want to utilize more complex IO schemas like prefetching, or handle raw data with decoding and augmentations, you can use the MXNet IO to achieve that, as we will discuss it later.

DataIter

Usually the optimization method for deep learning traverses data in a round-robin fashion. This perfectly matches the pattern of Python iterator. Therefore, MinPy/MXNet both choose to implement IO logic with iterater.

Generally, to create a data iterator, you need to provide five kinds of parameters:

- **Dataset Param** gives the basic information for the dataset, e.g. MinPy/NumPy/MXNet NDArray, file path, input shape, etc.
- **Batch Param** gives the information to form a batch, e.g. batch size.

- **Augmentation Param** tells which augmentation operations(e.g. crop, mirror) should be taken on an input image.
- **Backend Param** controls the behavior of the backend threads in order to hide data loading cost.
- **Auxiliary Param** provides options to help checking and debugging.

Usually, **Dataset Param** and **Batch Param** MUST be given, otherwise data batch cannot be created. Other parameters can be given according to algorithm and performance need. Suppose the raw source dataset has been prepared into a dictionary‘‘data‘‘, with two entries, containing training input and label, then the following code materialize them into an `NDArray` iterator:

```
train_dataiter = NDArrayIter(data=data['X_train'],
                             label=data['y_train'],
                             batch_size=batch_size,
                             shuffle=True)
```

We can now control the logic of each epoch by using:

```
for each_batch in self.train_dataiter:
```

Refer to the `_step` function in solver.py to see how data are accessed.

Using MXNet IO in MinPy

IO is a crucial part for deep learning. Raw data may need to go through a complex pipeline before feeding into solver. Obviously, poor IO implementation can become performance bottleneck. MXNet has a high performing and mature IO subsystem, we recommend MXNet IO when you move on to complex task and/or need better performance.

Minpy has automatically imported all the available MXNet DataIters into minpy.io namespace. To use MXNet IO, we just need to `import minpy.io` then using the DataIters. For example:

```
from minpy.io import MNISTIter
train           = MNISTIter(
    image       = data_dir + "train-images-idx3-ubyte",
    label       = data_dir + "train-labels-idx1-ubyte",
    input_shape = data_shape,
    batch_size  = args.batch_size,
    shuffle     = True,
    flat        = flat,
    num_parts   = kv.num_workers,
    part_index  = kv.rank)
```

Current available MXNet DataIters include `MNISTIter`, `ImageRecordIter`, `CSVIter`, `PrefetchingIter`, `ResizeIter`. They can directly take raw data like images, sequences as source and do decoding, augmenting and layout on the fly. To get more information about MXNet IO, please visit io.md and io.py.

CHAPTER 19

Limitation and Pitfalls

Not support in-place array operations

In-place array operations lead to confusion in gradient definition and therefore the team decides to exclude the support for all in-place array operations. For example, using the following mutable array operation is not allowed in MinPy.

```
In [1]: import minpy.numpy as np
        a = np.zeros((2,3))
        a.transpose()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-c6777f33a28a> in <module>()
      1 import minpy.numpy as np
      2 a = np.zeros((2,3))
----> 3 a.transpose()

AttributeError: 'Array' object has no attribute 'transpose'
```

But you can use immutable operation instead:

```
In [2]: a = np.transpose(a)
        # instead of a.transpose(), which is feasible in NumPy.
        # In MinPy, it will occur an error, since we can't calculate
        # its gradient.
```

A more common example which is not supported is:

```
In [3]: a[0, 1] = 12
```

The system will still allow you to perform such operations, but keep in mind that the autograd will fail in such cases.

Use MinPy consistantly

If you try to put NumPy array into MinPy operation, there are some cases that the computation will still happen in NumPy's namespace instead of MinPy's. For example

```
In [4]: import minpy.numpy as np
        def simple_add(a, b):
            return a + b

        # Now we declare two NumPy arrays:
        import numpy as npp
        a = npp.ones((10, 10))
        b = npp.zeros((10, 10))
```

If we pass `a` and `b` into function `simple_add`. the add operation will happen in NumPy's namespace. This is not the expected behavior. So we recommend you to use MinPy array consistently.

On the other hand, if you want to recover NumPy array from MinPy array for other packages like `matplotlib`, you can use `asnumpy`, which will return the corresponding NumPy array:

```
In [5]: a = np.zeros((2,3))
        print(type(a)) # this is a MinPy array
        print(type(a.asnumpy())) # convert to NumPy array
```

```
<class 'minpy.array.Array'>
<type 'numpy.ndarray'>
```

Not support all submodules

Since NumPy package is distributed as multiple submodules, currently not all submodules are supported. If you find any submodules (such as `numpy.random`) without support, please raise an issue on GitHub. The dev team will add support as soon as possible.

Not support multiple executions of the same MXNet symbol before BP

Unlike MinPy's primitives, MXNet symbol has internal stage to record gradient information. Thus applying same symbol to different data will fail BP in the later stage. You can create duplicate symbol to fulfil the same goal and designate same parameter name for parameter sharing.

CHAPTER 20

Supported GPU operators

MinPy integrates MXNet operators to enable computation on GPUs. Technically all MXNet GPU operators are supported.

As a reference, following MXNet operators exist.

- Elementwise unary operators
  - Abs
  - Sign
  - Round
  - Ceil
  - Floor
  - Square
  - Square root
  - Exponential
  - Logarithm
  - Cosine
  - Sine
- Elementwise binary operators
  - Plus
  - Minus
  - Multiplication
  - Division
  - Power
  - Maximum

  - Minimum
- Broadcast
  - Norm
  - Maximum
  - Minimum
  - Sum
  - Max axis
  - Min axis
  - Sum axis
  - Argmax channel
- Elementwise binary broadcast
  - Plus
  - Minus
  - Multiplication
  - Division
  - Power
- Matrix
  - Transpose
  - Expand dims
  - Crop
  - Slice axis
  - Flip
  - Dot
  - Batch dot
- Deconvolution
- Sequence mask
- Concatenation
- Cast
- Swap axis
- Block grad
- Leaky relu
- RNN
- Softmax
- Pooling
- Softmax cross entropy
- Sample uniform

- Sample normal
- Smooth L1

But not all MXNet operators are gradable. You can still use them in computation, but trying to `grad` them will result in an error.

Following MXNet operators have gradient implemented.

- Dot
- Exponential
- Logarithm
- Sum
- Plus
- Minus
- Multiplication
- Division
- True division
- Maximum
- Negation
- Transpose
- Abs
- Sign
- Round
- Ceil
- Floor
- Sqrt
- Sine
- Cosine
- Power
- Reshape
- Expand dims

As for NumPy operators, all preceding operators, plus the following, have gradient defined.

- Broadcast to
- Mod
- Minimum
- Append
- Sigmoid

The following table summarizes elementwise unary operators.

GPU operator	Gradient defined for MXNet	Gradient defined for NumPy
Abs	Y	Y
Sign	Y	Y
Round	Y	Y
Ceil	Y	Y
Floor	Y	Y
Square	Y	Y
Square root	Y	Y
Exponential	Y	Y
Logarithm	Y	Y
Cosine	Y	Y
Sine	Y	Y

The following table summarizes elementwise binary operators (broadcast included).

GPU operator	Gradient defined for MXNet	Gradient defined for NumPy
Plus	Y	Y
Subtract	Y	Y
Multiply	Y	Y
Divide	Y	Y
Power	Y	Y
Maximum	Y	Y
Minimum	N	Y

The following table summarizes broadcast reduce operators.

GPU operator	Gradient defined for MXNet	Gradient defined for NumPy
Norm	N	N
Maximum	Y	Y
Minimum	N	Y
Sum	Y	Y
Arg max	N	N

The following table summarizes basic matrix operators.

GPU operator	Gradient defined for MXNet	Gradient defined for NumPy
Transpose	Y	Y
Expand dims	Y	Y
Crop	Y	Y
Slice axis	Y	Y
Flip	N	N
Dot	Y	Y

CHAPTER 21

Visualization with TensorBoard

Visualization is a very intuitive way to inspect what is going on in a network. For that purpose, we integrate TensorBoard with MinPy. This tutorial begins with the *Logistic Regression Tutorial*, then moves on to a more advanced case with CNN.

Before trying this tutorial, make sure you have installed *MinPy* and TensorFlow.

Logistic regression

Set up as in the original tutorial.

```
In [1]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        import minpy.numpy as np
        import minpy.numpy.random as random
        from minpy.core import grad_and_loss
        from examples.utils.data_utils import gaussian_cluster_generator as make_data
        from minpy.context import set_context, gpu
```

Set up for Visualization. The related code is largely based on the source code from TensorFlow.

```
In [2]: from minpy.visualization.writer import LegacySummaryWriter as SummaryWriter
        import minpy.visualization.summaryOps as summaryOps
```

Declare the directory for log files which will be used for storing data during the training and later uploaded to TensorBoard later. The directory does not necessarily need to exist. The directory should start with `/private` where TensorBoard looks for log files by default.

```
In [3]: summaries_dir = '/private/tmp/LR_log'
```

```
In [4]: # Predict the class using multinomial logistic regression (softmax regression).
        def predict(w, x):
            a = np.exp(np.dot(x, w))
            a_sum = np.sum(a, axis=1, keepdims=True)
```

```
            prob = a / a_sum
            return prob

        def train_loss(w, x):
            prob = predict(w, x)
            loss = -np.sum(label * np.log(prob)) / num_samples
            return loss

        """Use Minpy's auto-grad to derive a gradient function off loss"""
        grad_function = grad_and_loss(train_loss)
```

Create the writer for the trainning. You may replace `/train` with `/test`, `/validation`, etc. as you like.

```
In [5]: train_writer = SummaryWriter(summaries_dir + '/train')
```

`summaryOps.scalarSummary` accepts a tag and a scalar as arguments and crates corresponding summary protos with scalars. `train_writer` then adds the summary proto to the log file. At the end of the training, close the writer.

The same trick works for all kinds of scalars. To clarify, it works for python scalars `float`, `int`, `long`, as well as one-element `minpy.array.Array` and `numpy.ndarray`.

Currently, Minpy only supports scalar summaries.

```
In [6]: # Using gradient descent to fit the correct classes.
        def train(w, x, loops):
            for i in range(loops):
                dw, loss = grad_function(w, x)
                # gradient descent
                w -= 0.1 * dw
                if i % 10 == 0:
                    print('Iter {}, training loss {}'.format(i, loss))
                summary1 = summaryOps.scalarSummary('loss', loss)
                train_writer.add_summary(summary1, i)
            train_writer.close()
```

```
In [7]: # Initialize training data.
        num_samples = 10000
        num_features = 500
        num_classes = 5
        data, label = make_data(num_samples, num_features, num_classes)
```

```
In [8]: # Initialize training weight and train
        weight = random.randn(num_features, num_classes)
        train(weight, data, 100)
```

```
Iter 0, training loss [ 14.2357111]
Iter 10, training loss [ 1.60548949]
Iter 20, training loss [ 0.25217342]
Iter 30, training loss [ 0.10623146]
Iter 40, training loss [ 0.06312769]
Iter 50, training loss [ 0.0435785]
Iter 60, training loss [ 0.03269563]
Iter 70, training loss [ 0.02586649]
Iter 80, training loss [ 0.02122972]
Iter 90, training loss [ 0.01790031]
```

Open the terminal, and call the following command:

```
tensorboard --logdir=summaries_dir
```

Note you don't need to include `/private` for the `summaries_dir`, so in this case the `summaries_dir` will be `/tmp/LR_log`.

Once you start TensorBoard, you should see the visualization of scalars in the EVENTS section as below. When you move your mouse along the curve, you should see the value at each step. You may change the size of the graph by clicking the button in the bottom-left corner.

Fig. 21.1: Loss History Interface

On the left hand side, you may decide to which extent you want to smooth the curve. You may choose one of the three choices as the horizontal axis of the graph. By checking **Data download links**, you may download the data in the format of `.csv` or `.json`. In the `.csv` or `.json` files, the data will be displayed in the form of [wall time - step - value].

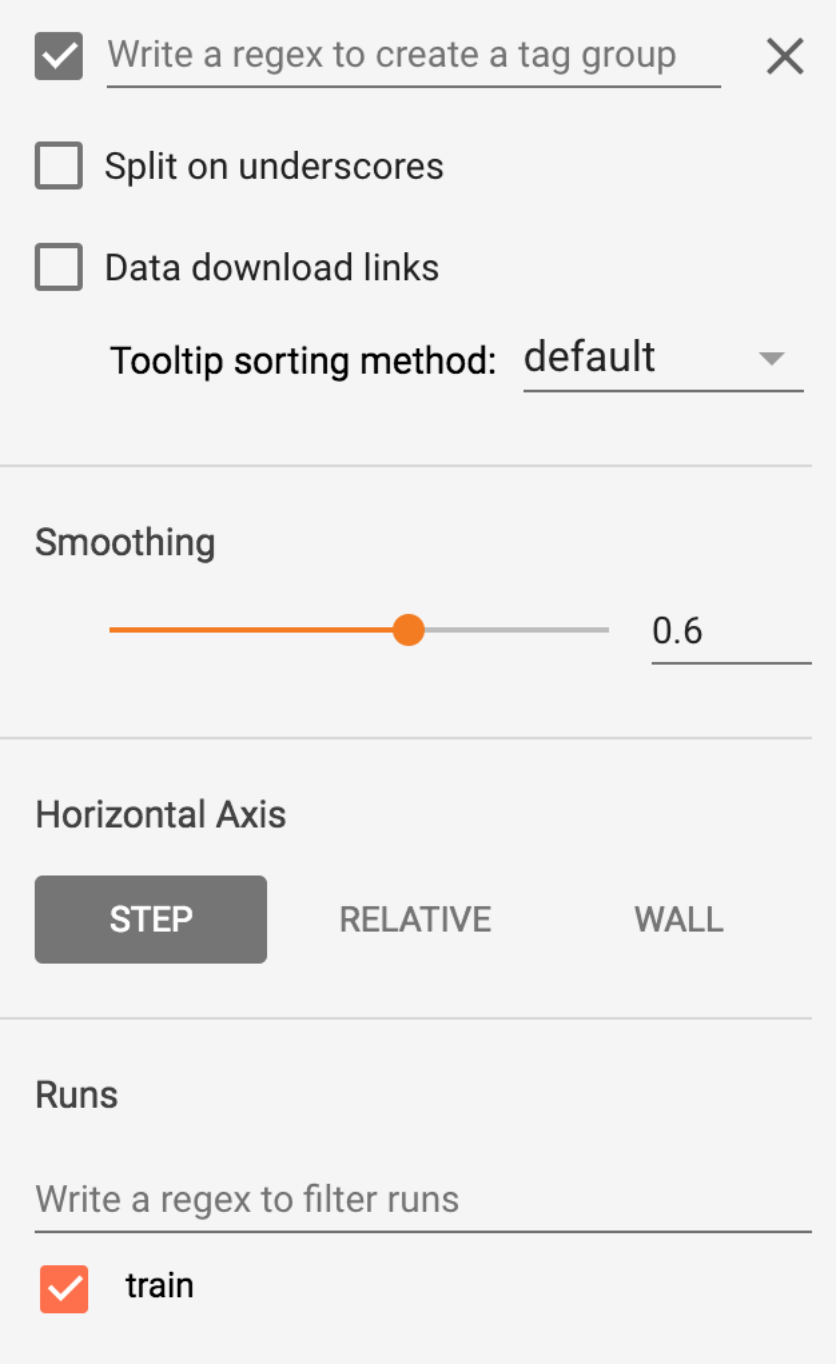


Fig. 21.2: Control Interface

CHAPTER 22

Visualizing Solvers with TensorBoard

This tutorial may assume knowledge from the tutorial on *Visualization with TensorBoard*. It is based on MinPy's *CNN Tutorial*.

Equip the CNN Tutorial with Visualization Functions

Set up as in the original tutorial.

```
In [1]: """Convolution Neural Network example using only MXNet symbol."""
        import sys

        from minpy.nn.io import NDArrayIter
        # Can also use MXNet IO here
        # from mxnet.io import NDArrayIter
        from minpy.core import Function
        from minpy.nn import layers
        from minpy.nn.model import ModelBase
        from minpy.nn.solver import Solver
        from examples.utils.data_utils import get_CIFAR10_data

        # Please uncomment following if you have GPU-enabled MXNet installed.
        #from minpy.context import set_context, gpu
        #set_context(gpu(0)) # set the global context as gpu(0)

        import mxnet as mx

        batch_size=128
        input_size=(3, 32, 32)
        flattened_input_size=3 * 32 * 32
        hidden_size=512
        num_classes=10
```

Design a template for CNN.

```
In [2]: class ConvolutionNet(ModelBase):
            def __init__(self):
                super(ConvolutionNet, self).__init__()
                # Define symbols that using convolution and max pooling to extract better features
                # from input image.
                net = mx.sym.Variable(name='X')
                net = mx.sym.Convolution(
                        data=net, name='conv', kernel=(7, 7), num_filter=32)
                net = mx.sym.Activation(
                        data=net, act_type='relu')
                net = mx.sym.Pooling(
                        data=net, name='pool', pool_type='max', kernel=(2, 2),
                        stride=(2, 2))
                net = mx.sym.Flatten(data=net)
                net = mx.sym.FullyConnected(
                        data=net, name='fc1', num_hidden=hidden_size)
                net = mx.sym.Activation(
                        data=net, act_type='relu')
                net = mx.sym.FullyConnected(
                        data=net, name='fc2', num_hidden=num_classes)
                net = mx.sym.SoftmaxOutput(data=net, name='softmax', normalization='batch')
                # Create forward function and add parameters to this model.
                input_shapes = {'X': (batch_size,) + input_size, 'softmax_label': (batch_size,)}
                self.cnn = Function(net, input_shapes=input_shapes, name='cnn')
                self.add_params(self.cnn.get_params())

            def forward_batch(self, batch, mode):
                out = self.cnn(X=batch.data[0],
                               softmax_label=batch.label[0],
                               **self.params)
                return out

            def loss(self, predict, y):
                return layers.softmax_cross_entropy(predict, y)
```

Set `get_CIFAR10_data`'s argument to the data file location for cifar-10 dataset. The original tutorial applied an argparse to read the directory directly in the terminal. For the convenience of using a Jupyter notebook, this is not used here.

Declare the directory for storing log files, which will be used for viusalization later.

`visualize` is an optional argument of Solver and is set to be `False` by default. Set `visualize` to be `True` and pass the `summaries_dir` argument as well. We will touch the details of implementing visualization functions in Solver later.

```
In [3]: def main():
            # Create model.
            model = ConvolutionNet()
            # Create data iterators for training and testing sets.
            data = get_CIFAR10_data('cifar-10-batches-py')
            train_dataiter = NDArrayIter(data=data['X_train'],
                                         label=data['y_train'],
                                         batch_size=batch_size,
                                         shuffle=True)
            test_dataiter = NDArrayIter(data=data['X_test'],
                                        label=data['y_test'],
                                        batch_size=batch_size,
                                        shuffle=False)

            # Declare the directory for storing data, which will be used for visualization with tens
```

```
            summaries_dir = '/private/tmp/cnn_log'

            # Create solver.
            solver = Solver(model,
                            train_dataiter,
                            test_dataiter,
                            num_epochs=10,
                            init_rule='gaussian',
                            init_config={
                                'stdvar': 0.001
                            },
                            update_rule='sgd_momentum',
                            optim_config={
                                'learning_rate': 1e-3,
                                'momentum': 0.9
                            },
                            verbose=True,
                            print_every=20,
                            visualize=True,
                            summaries_dir=summaries_dir)
            # Initialize model parameters.
            solver.init()
            # Train!
            solver.train()
```

```
In [4]: if __name__ == '__main__':
            main()
```

```
(Iteration 1 / 3828) loss: 2.302535
(Iteration 21 / 3828) loss: 2.302051
(Iteration 41 / 3828) loss: 2.291640
(Iteration 61 / 3828) loss: 2.133044
(Iteration 81 / 3828) loss: 2.033680
(Iteration 101 / 3828) loss: 1.995795
(Iteration 121 / 3828) loss: 1.796180
(Iteration 141 / 3828) loss: 1.884282
(Iteration 161 / 3828) loss: 1.702727
(Iteration 181 / 3828) loss: 1.745341
(Iteration 201 / 3828) loss: 1.550407
(Iteration 221 / 3828) loss: 1.405793
(Iteration 241 / 3828) loss: 1.529175
(Iteration 261 / 3828) loss: 1.440347
(Iteration 281 / 3828) loss: 1.859766
(Iteration 301 / 3828) loss: 1.416149
(Iteration 321 / 3828) loss: 1.481019
(Iteration 341 / 3828) loss: 1.501948
(Iteration 361 / 3828) loss: 1.508027
(Iteration 381 / 3828) loss: 1.516997
(Epoch 1 / 10) train acc: 0.501953125, val_acc: 0.4931640625, time: 1253.37731194.
(Iteration 401 / 3828) loss: 1.296929
(Iteration 421 / 3828) loss: 1.496588
(Iteration 441 / 3828) loss: 1.330925
(Iteration 461 / 3828) loss: 1.450040
(Iteration 481 / 3828) loss: 1.393043
(Iteration 501 / 3828) loss: 1.239604
(Iteration 521 / 3828) loss: 1.210205
(Iteration 541 / 3828) loss: 1.295574
(Iteration 561 / 3828) loss: 1.372109
(Iteration 581 / 3828) loss: 1.231615
(Iteration 601 / 3828) loss: 1.243544
```

```
(Iteration 621 / 3828) loss: 1.313342
(Iteration 641 / 3828) loss: 1.510346
(Iteration 661 / 3828) loss: 1.155001
(Iteration 681 / 3828) loss: 1.241223
(Iteration 701 / 3828) loss: 1.305725
(Iteration 721 / 3828) loss: 1.218895
(Iteration 741 / 3828) loss: 1.208463
(Iteration 761 / 3828) loss: 1.319934
(Epoch 2 / 10) train acc: 0.5751953125, val_acc: 0.5517578125, time: 1238.14002705.
(Iteration 781 / 3828) loss: 1.204560
(Iteration 801 / 3828) loss: 1.388396
(Iteration 821 / 3828) loss: 1.208335
(Iteration 841 / 3828) loss: 1.197055
(Iteration 861 / 3828) loss: 1.225983
(Iteration 881 / 3828) loss: 1.007661
(Iteration 901 / 3828) loss: 1.083537
(Iteration 921 / 3828) loss: 1.170273
(Iteration 941 / 3828) loss: 1.079046
(Iteration 961 / 3828) loss: 1.060466
(Iteration 981 / 3828) loss: 1.186217
(Iteration 1001 / 3828) loss: 1.176932
(Iteration 1021 / 3828) loss: 1.049240
(Iteration 1041 / 3828) loss: 1.084303
(Iteration 1061 / 3828) loss: 1.137581
(Iteration 1081 / 3828) loss: 1.201812
(Iteration 1101 / 3828) loss: 0.991179
(Iteration 1121 / 3828) loss: 1.053682
(Iteration 1141 / 3828) loss: 1.033876
(Epoch 3 / 10) train acc: 0.5771484375, val_acc: 0.5859375, time: 1111.29330206.
(Iteration 1161 / 3828) loss: 0.945752
(Iteration 1181 / 3828) loss: 0.900214
(Iteration 1201 / 3828) loss: 0.996316
(Iteration 1221 / 3828) loss: 0.725004
(Iteration 1241 / 3828) loss: 1.053474
(Iteration 1261 / 3828) loss: 0.956877
(Iteration 1281 / 3828) loss: 1.118823
(Iteration 1301 / 3828) loss: 1.032918
(Iteration 1321 / 3828) loss: 1.078873
(Iteration 1341 / 3828) loss: 0.964023
(Iteration 1361 / 3828) loss: 1.081211
(Iteration 1381 / 3828) loss: 0.975109
(Iteration 1401 / 3828) loss: 0.887941
(Iteration 1421 / 3828) loss: 0.812622
(Iteration 1441 / 3828) loss: 0.781776
(Iteration 1461 / 3828) loss: 0.839401
(Iteration 1481 / 3828) loss: 1.083514
(Iteration 1501 / 3828) loss: 0.916411
(Iteration 1521 / 3828) loss: 0.820561
(Epoch 4 / 10) train acc: 0.658203125, val_acc: 0.599609375, time: 1107.30718303.
(Iteration 1541 / 3828) loss: 0.956412
(Iteration 1561 / 3828) loss: 0.835572
(Iteration 1581 / 3828) loss: 0.791931
(Iteration 1601 / 3828) loss: 0.892034
(Iteration 1621 / 3828) loss: 0.846968
(Iteration 1641 / 3828) loss: 0.790181
(Iteration 1661 / 3828) loss: 1.008565
(Iteration 1681 / 3828) loss: 0.971547
(Iteration 1701 / 3828) loss: 0.904101
(Iteration 1721 / 3828) loss: 0.764249
```

```
(Iteration 1741 / 3828) loss: 0.839634
(Iteration 1761 / 3828) loss: 0.667381
(Iteration 1781 / 3828) loss: 0.892126
(Iteration 1801 / 3828) loss: 0.790432
(Iteration 1821 / 3828) loss: 0.915785
(Iteration 1841 / 3828) loss: 0.701808
(Iteration 1861 / 3828) loss: 0.713519
(Iteration 1881 / 3828) loss: 0.939402
(Iteration 1901 / 3828) loss: 0.728612
(Epoch 5 / 10) train acc: 0.6630859375, val_acc: 0.5966796875, time: 1127.98228502.
(Iteration 1921 / 3828) loss: 0.898663
(Iteration 1941 / 3828) loss: 1.081481
(Iteration 1961 / 3828) loss: 0.956133
(Iteration 1981 / 3828) loss: 0.664632
(Iteration 2001 / 3828) loss: 0.986162
(Iteration 2021 / 3828) loss: 0.921607
(Iteration 2041 / 3828) loss: 0.855872
(Iteration 2061 / 3828) loss: 0.785384
(Iteration 2081 / 3828) loss: 0.985731
(Iteration 2101 / 3828) loss: 0.693248
(Iteration 2121 / 3828) loss: 1.032196
(Iteration 2141 / 3828) loss: 0.918029
(Iteration 2161 / 3828) loss: 0.809714
(Iteration 2181 / 3828) loss: 0.876201
(Iteration 2201 / 3828) loss: 0.714913
(Iteration 2221 / 3828) loss: 0.964526
(Iteration 2241 / 3828) loss: 0.795892
(Iteration 2261 / 3828) loss: 0.756644
(Iteration 2281 / 3828) loss: 0.571955
(Epoch 6 / 10) train acc: 0.720703125, val_acc: 0.6044921875, time: 1100.48066902.
(Iteration 2301 / 3828) loss: 0.584125
(Iteration 2321 / 3828) loss: 0.818221
(Iteration 2341 / 3828) loss: 0.647816
(Iteration 2361 / 3828) loss: 0.807244
(Iteration 2381 / 3828) loss: 0.663801
(Iteration 2401 / 3828) loss: 0.710950
(Iteration 2421 / 3828) loss: 0.869763
(Iteration 2441 / 3828) loss: 0.659388
(Iteration 2461 / 3828) loss: 0.884262
(Iteration 2481 / 3828) loss: 0.892994
(Iteration 2501 / 3828) loss: 0.696201
(Iteration 2521 / 3828) loss: 0.792361
(Iteration 2541 / 3828) loss: 0.583030
(Iteration 2561 / 3828) loss: 0.987736
(Iteration 2581 / 3828) loss: 0.812939
(Iteration 2601 / 3828) loss: 0.686343
(Iteration 2621 / 3828) loss: 0.696793
(Iteration 2641 / 3828) loss: 0.730227
(Iteration 2661 / 3828) loss: 0.717481
(Iteration 2681 / 3828) loss: 0.717061
(Epoch 7 / 10) train acc: 0.6875, val_acc: 0.5849609375, time: 1019.18220496.
(Iteration 2701 / 3828) loss: 0.960259
(Iteration 2721 / 3828) loss: 0.851661
(Iteration 2741 / 3828) loss: 0.547349
(Iteration 2761 / 3828) loss: 0.629300
(Iteration 2781 / 3828) loss: 0.794492
(Iteration 2801 / 3828) loss: 0.674677
(Iteration 2821 / 3828) loss: 0.547635
(Iteration 2841 / 3828) loss: 0.633213
```

```
(Iteration 2861 / 3828) loss: 0.817622
(Iteration 2881 / 3828) loss: 0.759713
(Iteration 2901 / 3828) loss: 0.746527
(Iteration 2921 / 3828) loss: 0.809928
(Iteration 2941 / 3828) loss: 0.804247
(Iteration 2961 / 3828) loss: 0.593531
(Iteration 2981 / 3828) loss: 0.884193
(Iteration 3001 / 3828) loss: 0.645554
(Iteration 3021 / 3828) loss: 0.568051
(Iteration 3041 / 3828) loss: 0.523802
(Iteration 3061 / 3828) loss: 0.691015
(Epoch 8 / 10) train acc: 0.6953125, val_acc: 0.5947265625, time: 962.428817034.
(Iteration 3081 / 3828) loss: 0.646333
(Iteration 3101 / 3828) loss: 0.893681
(Iteration 3121 / 3828) loss: 0.822102
(Iteration 3141 / 3828) loss: 0.619557
(Iteration 3161 / 3828) loss: 0.787171
(Iteration 3181 / 3828) loss: 0.725924
(Iteration 3201 / 3828) loss: 0.559321
(Iteration 3221 / 3828) loss: 0.654796
(Iteration 3241 / 3828) loss: 0.646047
(Iteration 3261 / 3828) loss: 0.789430
(Iteration 3281 / 3828) loss: 0.639559
(Iteration 3301 / 3828) loss: 0.798087
(Iteration 3321 / 3828) loss: 0.669927
(Iteration 3341 / 3828) loss: 0.706900
(Iteration 3361 / 3828) loss: 0.560583
(Iteration 3381 / 3828) loss: 0.630658
(Iteration 3401 / 3828) loss: 0.804180
(Iteration 3421 / 3828) loss: 0.727579
(Iteration 3441 / 3828) loss: 0.547852
(Epoch 9 / 10) train acc: 0.6982421875, val_acc: 0.5888671875, time: 958.027697086.
(Iteration 3461 / 3828) loss: 0.599252
(Iteration 3481 / 3828) loss: 0.485362
(Iteration 3501 / 3828) loss: 0.741121
(Iteration 3521 / 3828) loss: 0.636478
(Iteration 3541 / 3828) loss: 0.711437
(Iteration 3561 / 3828) loss: 0.655215
(Iteration 3581 / 3828) loss: 0.651631
(Iteration 3601 / 3828) loss: 0.762882
(Iteration 3621 / 3828) loss: 0.817763
(Iteration 3641 / 3828) loss: 0.768698
(Iteration 3661 / 3828) loss: 0.742337
(Iteration 3681 / 3828) loss: 0.569759
(Iteration 3701 / 3828) loss: 0.610525
(Iteration 3721 / 3828) loss: 0.623297
(Iteration 3741 / 3828) loss: 0.733673
(Iteration 3761 / 3828) loss: 0.573780
(Iteration 3781 / 3828) loss: 0.606257
(Iteration 3801 / 3828) loss: 0.800820
(Iteration 3821 / 3828) loss: 0.639535
(Epoch 10 / 10) train acc: 0.7216796875, val_acc: 0.5810546875, time: 703.184649944.
```

Open the terminal, and call the following command:

```
tensorboard --logdir=summaries_dir
```

Note you don't need to include `/private` for `summaries_dir`, so in this case `summaries_dir` will be `/tmp/cnn_log`.

Once you start TensorBoard, you should see the visualization of scalars in the EVENTS section as below. The training accuracy, validation accuracy, training loss and the squared L2-norm of the gradient are implemented by default in the Solver.

Note: If you have more than one `SummaryWriter`(2 in this case), the data of some `SummaryWriters` might not be written into the log files immediately. But you should get whatever you want by the end of the training.

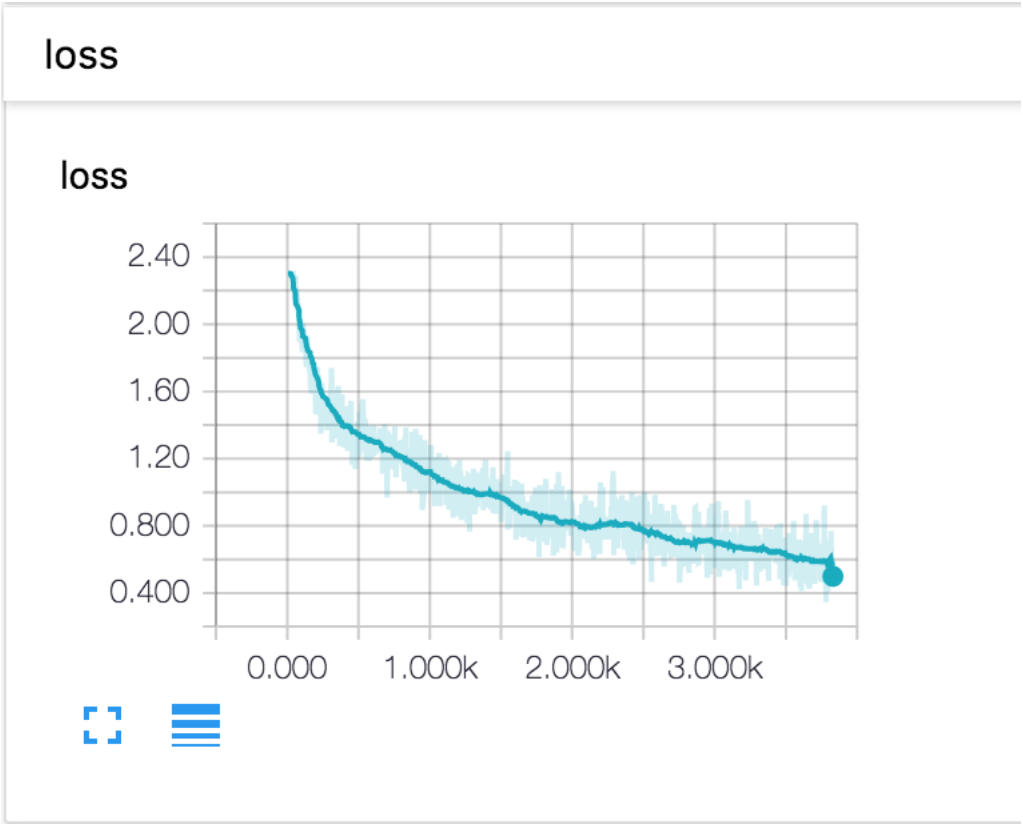


Fig. 22.1: CNN Loss Curve

Implementation Details of the Solver

Now we touch the details of the implementation of visualization in Solver. This will not show a complete implementation for the Solver class.

Step1: Generate SummaryWriters

If `self.visualize == True`, two `SummaryWriters` will be generated by default, one for training and one for testing.

```
In [ ]: class Solver(object):
            ...
            def __init__(self, model, train_dataiter, test_dataiter, **kwargs):
                ...
                self.visualize = kwargs.pop('visualize', False)
```

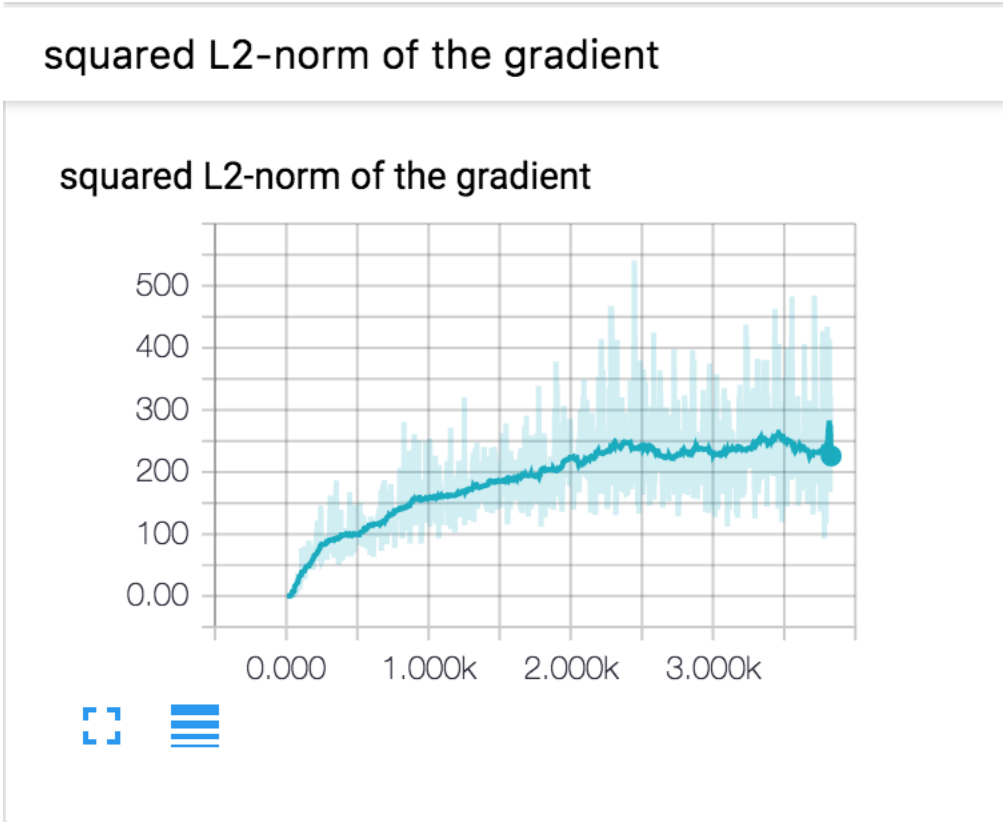


Fig. 22.2: Curve of Squared L2-Norm

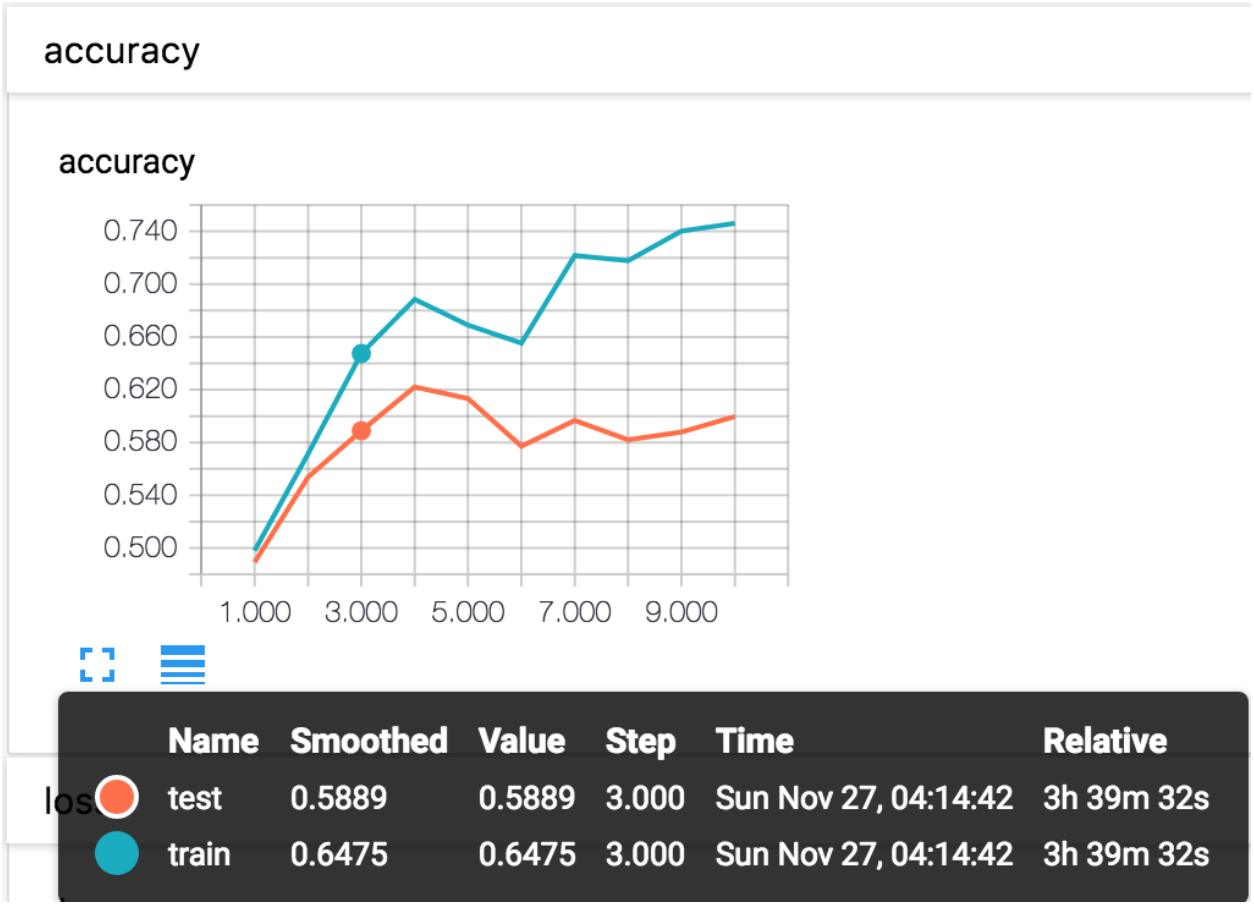


Fig. 22.3: Training accuracy

```
        if self.visualize:
            # Retrieve the summary directory. Create summary writers for training and test.
            self.summaries_dir = kwargs.pop('summaries_dir', '/private/tmp/newlog')
            self.train_writer = SummaryWriter(self.summaries_dir + '/train')
            self.test_writer = SummaryWriter(self.summaries_dir + '/test')
```

Step2: Set a Scalar Summary for Squared L2-norm of the Gradient

```
In [ ]: def _step(self, batch, iteration):
            ...
            if self.visualize:
                    Grad_norm = 0

            # Perform a parameter update
            for p, w in self.model.params.items():
                dw = grads[p]
                if self.visualize:
                    norm = dw ** 2
                    while not isinstance(norm, minpy.array.Number):
                        norm = sum(norm)
                    Grad_norm += norm
                config = self.optim_configs[p]
                next_w, next_config = self.update_rule(w, dw, config)
                self.model.params[p] = next_w
                self.optim_configs[p] = next_config

            if self.visualize:
                grad_norm_summary = summaryOps.scalarSummary('squared L2-norm of the gradient', G
                self.train_writer.add_summary(grad_norm_summary, iteration)
            ...
```

Step3: Set a Scalar Summary for Training Loss

```
In [ ]: def train(self):
            """
            Run optimization to train the model.
            """
            num_iterations = self.train_dataiter.getnumiterations(
            ) * self.num_epochs
            t = 0
            for epoch in range(self.num_epochs):
                start = time.time()
                self.epoch = epoch + 1
                for each_batch in self.train_dataiter:
                    self._step(each_batch, t + 1)
                    # Maybe print training loss
                    if self.verbose and t % self.print_every == 0:
                        print('(Iteration %d / %d) loss: %f' %
                              (t + 1, num_iterations, self.loss_history[-1]))
                    if self.visualize:
                        # Add scalar summaries of training loss.
                        loss_summary = summaryOps.scalarSummary('loss', self.loss_history[-1])
                        self.train_writer.add_summary(loss_summary, t + 1)

                    t += 1
```

Step4: Set a Scalar Summary for Training/Validation Accuracy

```
In [ ]: def train(self):
            ...
            for epoch in range(self.num_epochs):
                start = time.time()
                self.epoch = epoch + 1
                ...
                # evaluate after each epoch
                train_acc = self.check_accuracy(self.train_dataiter, num_samples=self.train_acc_num_s
                val_acc = self.check_accuracy(self.test_dataiter)
                self.train_acc_history.append(train_acc)
                self.val_acc_history.append(val_acc)
                ...
                if self.visualize:
                    val_acc_summary = summaryOps.scalarSummary('accuracy', val_acc)
                    self.test_writer.add_summary(val_acc_summary, self.epoch)
                    train_acc_summary = summaryOps.scalarSummary('accuracy', train_acc)
                    self.train_writer.add_summary(train_acc_summary, self.epoch)
                ...
```

You could do whatever you want like cross entropy, dropout_keep_probability, mean, etc. This is a result from the TensorFlow's tutorial on constructing a deep convolutional MNIST classifier: link.

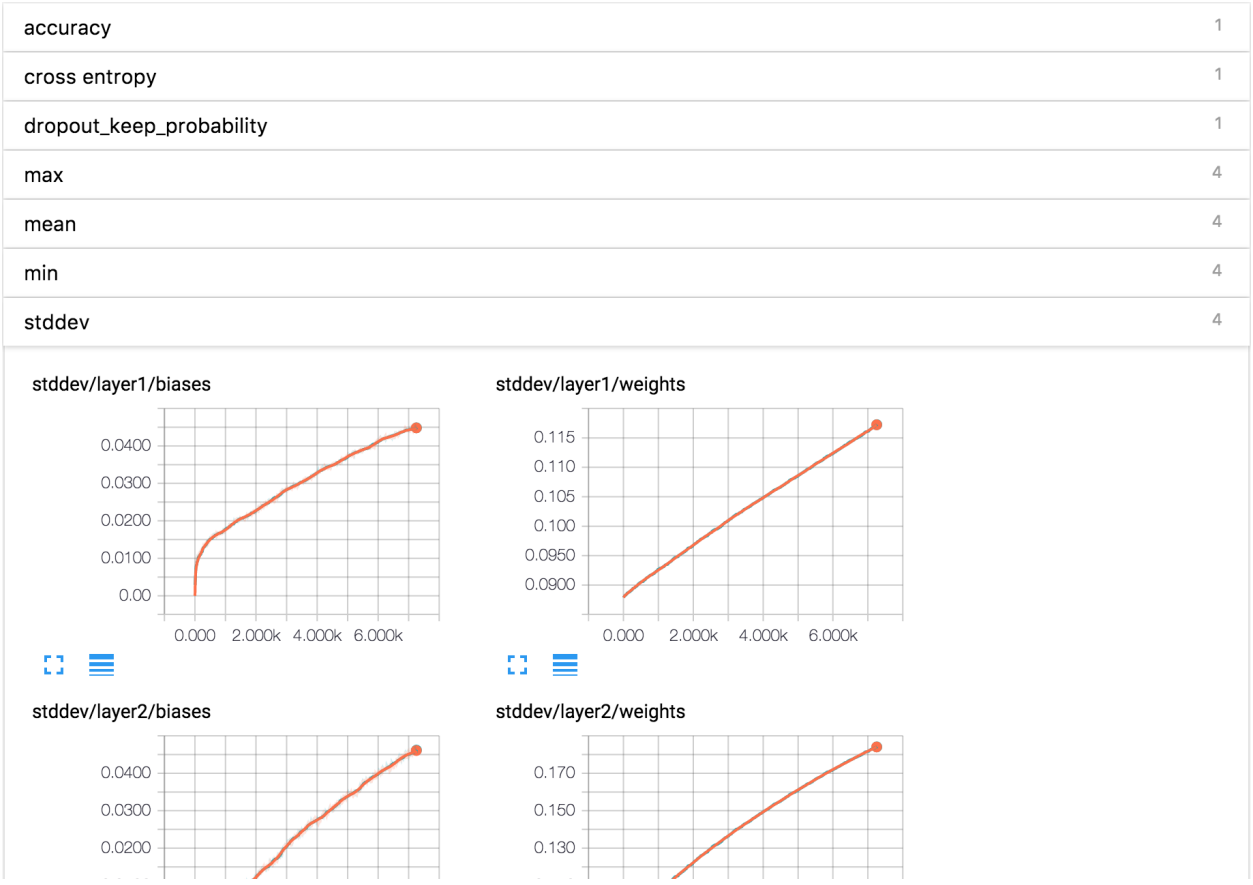


Fig. 22.4: MNIST Result

CHAPTER 23

How to build and release

Working on a development version of MinPy is most convenient when installed in editable mode. Refer to *MinPy installation guide* for instructions. This way changes will appear directly in the installed package.

MinPy also needs MXNet to run its GPU operators. There is also a regular and a development installtion for MXNet. Please refer to MXNet's document for details. MXNet has some C++ parts, so it requires compilation. But MinPy is pure Python, so a `pip` installation is enough.

When contributing, make sure coding convention is consistent. Run `yapf -i <python-file>` to auto format source files. Google Python Style Guide is a good stop for advice.

MinPy version numbers conform to Semver rules. To ensure consistency between version numbers on PyPI and git tags, there are a few utility scripts at the root of the repositry.

First run `./install_hooks.sh` once to install some Git hooks. This will automatically tag your commits upon version change, and push them if necessary. Now after you make some changes, run `./bump_version (major | minor | patch)` to increment the version number. Then make a commit and push to upstream. If your Git version is not too old, it will push the tags along the commit. Otherwise you would have to push the tags manually.

Travis CI will test and build the commit, and if there is a tag, release new version to PyPI.

CHAPTER 24

Add to MinPy's operators

Since MinPy inherits nearly all operators from NumPy, you could expect to use a NumPy operator without additional instructions. The MinPy operator might actually run the NumPy implementation with the same name or corresponding MXNet implementation, depending on the policy.

Apart from the NumPy experience, all other automatic switching details should not worry users. But if you want to define a new operator, or implement some missing gradient for operators from NumPy, there are a few steps to follow.

1. Define the gradient of the NumPy version of the operator in `minpy/array_varians/numpy/numpy_core.py`. There are already a bunch of operators with patterns to follow. In essence you need to define a second order function that takes all inputs to the original function, and the gradient passed down as well. Your job is to have the function pass the gradient through the operator.

For example, in the `np.dot` case, two gradient functions are defined. One is for the first argument, same for the second. It is provided as `argnum` to the `def_grad` method. The gradient itself is a second order function which takes first the result of the computation and orginal inputs, then the gradient. So it is `lambda ans, a, b: lambda g: np.dot(g, b.T)` in this case.

2. Follow the same steps for the MXNet version in `minpy/array_variants/mxnet/mxnet_core.py`. Sometimes the MXNet operator has a slightly different name, then you should `register` it under function `register_primitives`.

You could technically define only NumPy version or MXNet version. The policy is smart enought to fall back. But there will be performance penalty copying data back and forth.

CHAPTER 25

MinPy API

minpy package

Subpackages

minpy.array_variants package

Subpackages

minpy.array_variants.mxnet package

Submodules

minpy.array_variants.mxnet.mxnet_core module

minpy.array_variants.mxnet.mxnet_wrapper module

minpy.array_variants.mxnet.random module

minpy.array_variants.numpy package

Submodules

minpy.array_variants.numpy.numpy_core module

minpy.array_variants.numpy.numpy_wrapper module

CHAPTER 26

History and Credits

MinPy is intended to be the NumPy frontend of the MXNet project. Its core members contributed to MXNet's execution engine. Having completed that part, the team decided to take a step back and rehthink the user experience, before moving to yet another stage of performance optimizations.

The key goal is *innovation without compromising performance and usability*. This philosophy is shared by the community, but the difference is we see sticking to NumPy interface and imperative programming as higher priority.

Technical inspiration

We have also observed a great deal of innovations from the community. Therefore, we innovate when necessary, and otherwise draw inspiration (and sometimes direct implementations) from the followings:

- Auto-differentiation, i.e. Autograd project.
- Transparent CPU/GPU acceleration
- Visualization, i.e. TensorBoard.
- Learning deep learning using NumPy, e.g. Stanford's CS231n course.

People

A great number of people have contributed to the project. Most of them are students from around the world, and work with their spare time. People listed below without specified contributions are *generalists* who have wider work span. We want also to thank the MXNet development community for timely technical support.

- Minjie Wang; NYU, project lead (GitHub)
- Yutian Li: Face++/Stanford (GitHub)
- Larry Tang: NYU Shanghai/Michigan (GitHub)
- Haoran Wang: NYU Shanghai/CMU (GitHub)

- Tianjun Xiao: Microsoft/Tesla (GitHub)
- Ziheng Jiang: Fudan/NYU Shanghai (GitHub)
- Alex Gai: NYU Shanghai [Model builder] (GitHub)
- Sean Welleck: NYU/NYU Shanghai [Reinforcement learning] (GitHub)
- Xu Zhou: ShanghaiTech [CS231 courseware] (GitHub)
- Kerui Min: BosonData [RNN example on MNIST] (Linkedin)
- Murphy Li: NYU Shangai [Tensorboard integration] (GitHub)
- Professor Zheng Zhang: NYU Shanghai [general advising] (GitHub)

CHAPTER 27

Indices and tables

- genindex
- modindex
- search