akid Documentation

Release 0.1

Shuai Li

January 05, 2017

Contents

1 Get Started 1

2 Introduction 3

3 HOW TO 17

4 Tutorials of akid 21

5 Architecture and Design Principles 23

6 Model Zoo 27

Python Module Index 29

CHAPTER 1

Get Started

1.1 Download and Setup

Currently, `akid` only supports installing from the source.

1.1.1 Dependency

`akid` depends on some regular numerical libraries. If you are using `pip`, you could install them as the following:

```
pip install numpy, scipy, matplotlib, gflags
```

Follow the official installation guide to install tensorflow.

1.1.2 Install from the source

Clone the repository

```
git clone https://github.com/shawnLeeZX/akid
```

1.1.3 Post installation setup

Environment Variables

If you want to use the dataset automatically download feature, an environment variable needs to be set. Add the following line to `.bashrc`, or other configuration files of your favorite shell.

```
AKID_DATA_PATH=  # where you want to store data
```

Also, remember to make `akid` visible by adding the folder that contains `akid` to `PYTHONPATH`.

CHAPTER 2

Introduction

2.1 Why akid

Why another package on neural network?

Neural network, which is broadly named as Deep Learning nowadays, seems to have the potential to lead another technology revolution. It has incurred wide enthusiasm in industry, and serious consideration in public sector and impact evaluation in government. However, though being a remarkable breakthrough in high dimensional perception problems academically and intellectually stimulating and promising, it is still rather an immature technique that is fast moving and in shortage of understanding. Temporarily its true value lies in the capability to solve data analytic problems in industry, e.g. self-driving cars, detection of lung cancer etc. On the other hand, Neural Network is a technique that heavily relies on a large volume of data. It is critical for businesses that use such a technique to leverage on newly available data as soon as possible, which helps form a positive feedback loop that improves the quality of service.

Accordingly, to benefits from the newest development and newly available data, we want the gap between research and production as small as possible. In this package, we explore technology stacks abstraction that enable fast research prototyping and are production ready.

`akid` tries to provide a full stack of softwares that provides abstraction to let researchers focus on research instead of implementation, while at the same time the developed program can also be put into production seamlessly in a distributed environment, and be production ready when orchestrating with containers, a cluster manager, and a distributed network file system.

At the top application stack, it provides out-of-box tools for neural network applications. Lower down, `akid` provides programming paradigm that lets user easily build customized model. The distributed computing stack handles the concurrency and communication, thus letting models be trained or deployed to a single GPU, multiple GPUs, or a distributed environment without affecting how a model is specified in the programming paradigm stack. Lastly, the

distributed deployment stack handles how the distributed computing is deployed, thus decoupling the research prototype environment with the actual production environment, and is able to dynamically allocate computing resources, so developments (Devs) and operations (Ops) could be separated.

2.2 akid stack

Now we discuss each stack provided by akid.

2.2.1 Application stack

At the top of the stack, akid could be used as a part of application without knowing the underlying mechanism of neural networks.

akid provides full machinery from preparing data, augmenting data, specifying computation graph (neural network architecture), choosing optimization algorithms, specifying parallel training scheme (data parallelism etc), logging and visualization.

Neural network training — A holistic example

To create better tools to train neural network has been at the core of the original motivation of akid. Consequently, in this section, we describe how akid can be used to train neural networks. Currently, all the other feature resolves around this.

The snippet below builds a simple neural network, and trains it using MNIST, the digit recognition dataset.

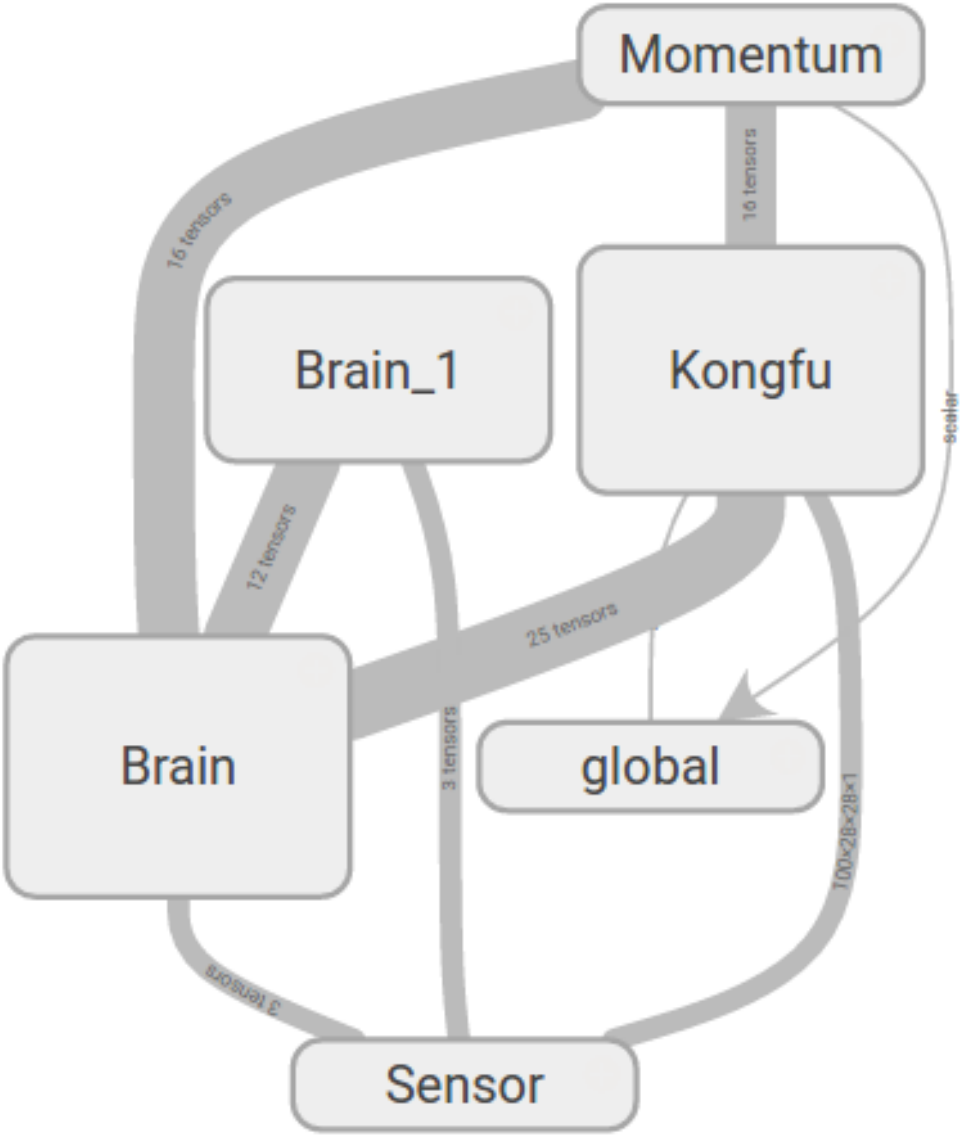
```
from akid import AKID_DATA_PATH
from akid import FeedSensor
from akid import Kid
from akid import MomentumKongFu
from akid import MNISTFeedSource

from akid.models.brains import LeNet

brain = LeNet(name="Brain")
source = MNISTFeedSource(name="Source",
                         url='http://yann.lecun.com/exdb/mnist/',
                         work_dir=AKID_DATA_PATH + '/mnist',
                         center=True,
                         scale=True,
                         num_train=50000,
                         num_val=10000)

sensor = FeedSensor(name='Sensor', source_in=source)
s = Kid(sensor,
        brain,
        MomentumKongFu(name="Kongfu"),
        max_steps=100)
kid.setup()
kid.practice()
```

It builds a computation graph as the following



The story happens underlying are described in the following, which also debriefs the design motivation and vision behind.

`akid` is a kid who has the ability to keep practicing to improve itself. The kid perceives a data `Source` with its `Sensor` and certain learning methods (nicknamed `KongFu`) to improve itself (its `Brain`), to fulfill a certain purpose. The world is timed by a clock. It represents how long the kid has been practicing. Technically, the clock is the conventional training step.

To break things done, `Sensor` takes a `Source` which either provides data in form of tensors from Tensorflow or numpy arrays. Optionally, it can make jokers on the data using `Joker`, meaning doing data augmentation. The data processing engine, which is a deep neural network, is abstracted as a `Brain`. `Brain` is the name we give to the data processing system in living beings. A `Brain` incarnates one of data processing system topology, or in the terminology

of neural network, network structure topology, such as a sequentially linked together layers, to process data. Available topology is defined in module `systems`. The network training methods, which are first order iterative optimization methods, is abstracted as a class `KongFu`. A living being needs to keep practicing Kong Fu to get better at tasks needed to survive.

A living being is abstracted as a `Kid` class, which assemblies all above classes together to play the game. The metaphor means by sensing more examples, with certain genre of Kong Fu(different training algorithms and policies), the data processing engine of the `Kid`, the brain, should get better at doing whatever task it is doing, letting it be image classification or something else.

Visualization

As a library gearing upon research, it also has rich features to visualize various components of a neural network. It has built-in training dynamics visualization, more specifically, distribution visualization on multi-dimensional tensors, e.g., weights, activation, biases, gradients, etc, and line graph visualization on on scalars, e.g., training loss, validation loss, learning rate decay, regularzation loss in each layer, sparsity of neuron activation etc, and filter and feature map visualization for neural networks.

Distribution and scalar visualization are built in for typical parameters and measures, and can be easily extended, and distributedly gathered. Typical visualization are shown below.

`akid` supports visualization of all feature maps and filters with control on the layout through `Observer` class. When having finished creating a `Kid`, pass it to `Observer`, and call visualization as the following.

```
from akid import Observer

o = Observer(kid)
# Visualize filters as the following
o.visualize_filters()
# Or visualize feature maps as the following
o.visualize_activation()
```

Various layouts are provided when drawing the filters. Additional features are also available. The post-preprocessed results are shown below.

2.2.2 Programming Paradigm

We have seen how to use functionality of `akid` without much programming in the previous section. In this section, we would like to introduce the programming paradigm underlying the previous example, and how to use `akid` as a research library with such paradigm. *akid* builds another layer of abstraction on top of *Tensor*: *Block*. Tensor can be taken as the media/formalism signal propagates in digital world, while Block is the data processing entity that processes inputs and emits outputs.

It coincides with a branch of "ideology" called dataism that takes everything in this world is a data processing entity. An interesting one that may come from *A Brief History of Tomorrow* by Yuval Noah Harari.

Best designs mimic nature. *akid* tries to reproduce how signals in nature propagates. Information flow can be abstracted as data propagating through inter-connected blocks, each of which processes inputs and emits outputs. For example, a vision classification system is a block that takes image inputs and gives classification results. Everything is a *Block* in *akid*.

A block could be as simple as a convonlutional neural network layer that merely does convolution on the input data and outputs the results; it also be as complex as an acyclic graph that inter-connects blocks to build a neural network, or sequentially linked block system that does data augmentation.

Compared with pure symbol computation approach, like the one in tensorflow, a block is able to contain states associated with this processing unit. Signals are passed between blocks in form of tensors or list of tensors. Many heavy

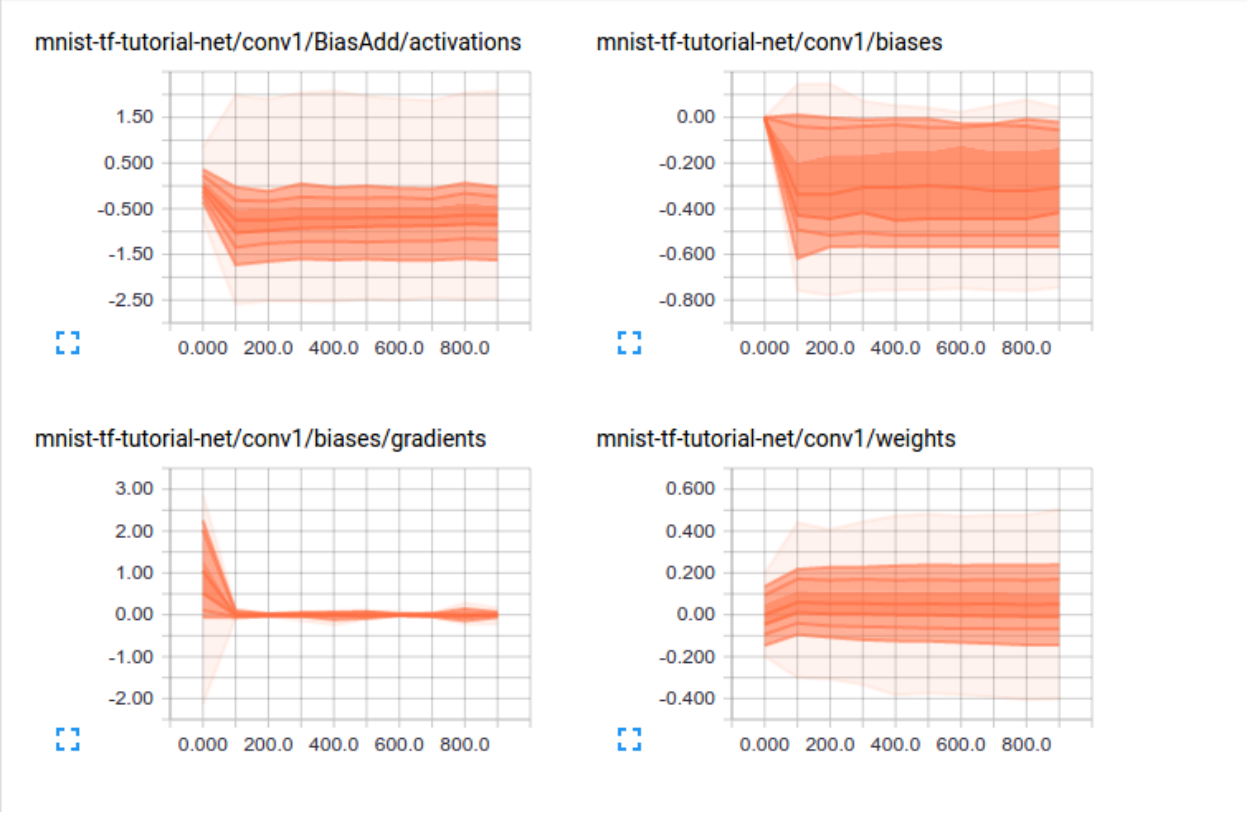


Fig. 2.1: Visualization of how distribution of multi-dimensional tensors change over time. Each line on the chart represents a percentile in the distribution over the data: for example, the bottom line shows how the minimum value has changed over time, and the line in the middle shows how the median has changed. Reading from top to bottom, the lines have the following meaning: [maximum, 93%, 84%, 69%, 50%, 31%, 16%, 7%, minimum] These percentiles can also be viewed as standard deviation boundaries on a normal distribution: [maximum, μ+1.5σ, μ+σ, μ+0.5σ, μ, μ-0.5σ, μ-σ, μ-1.5σ, minimum] so that the colored regions, read from inside to outside, have widths [σ, 2σ, 3σ] respectively.

Fig. 2.2: Visualization of how important scalar measures change over time.

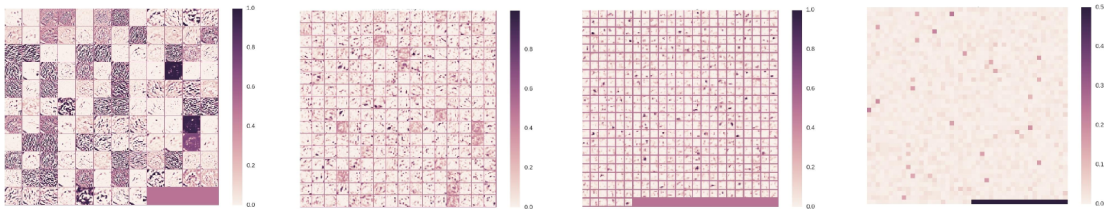


Fig. 2.3: Visualization of feature maps learned.

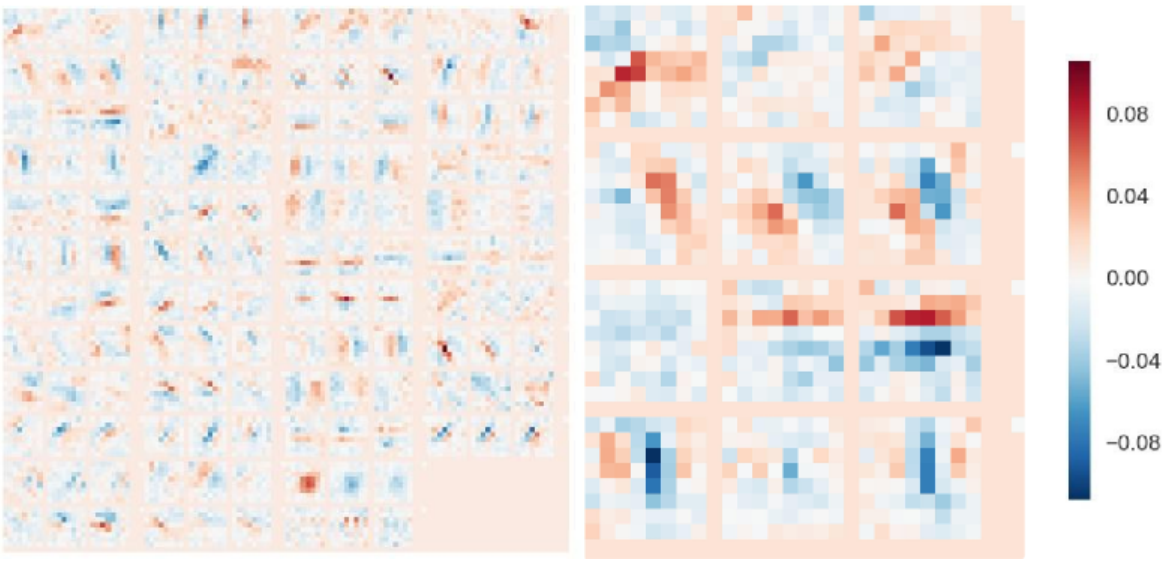


Fig. 2.4: Visualization of filters learned.

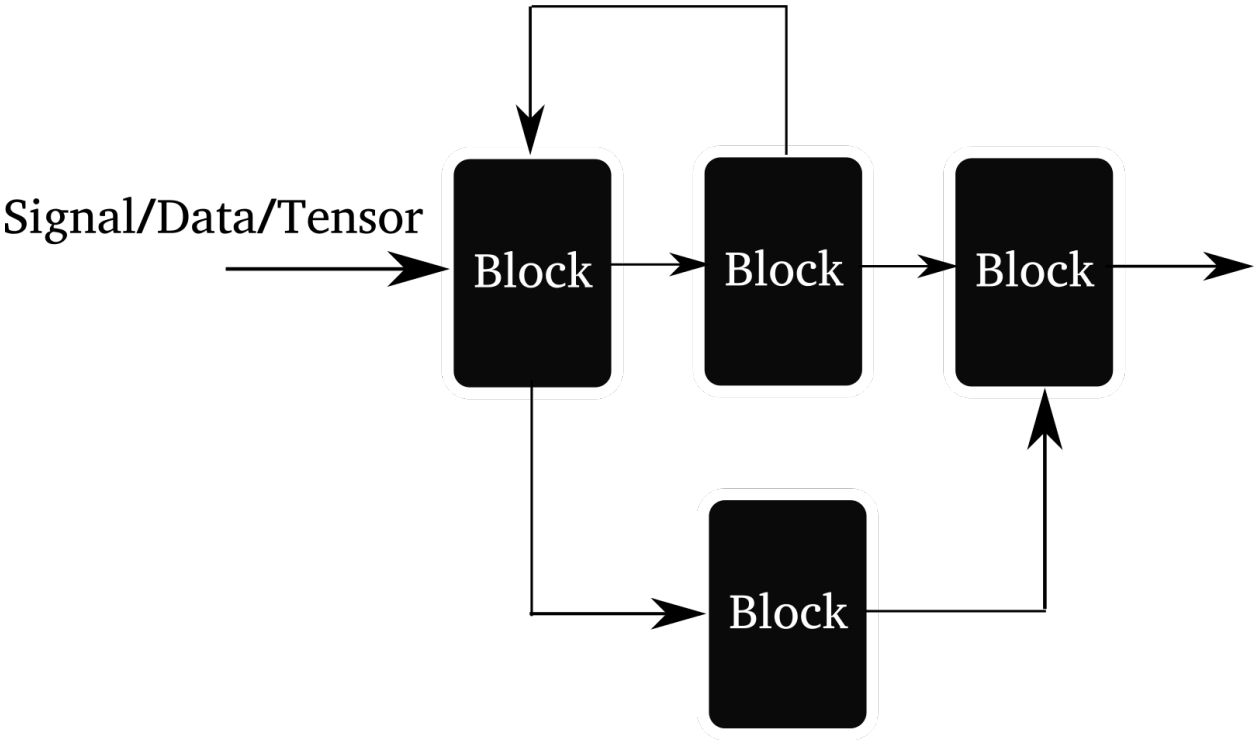


Fig. 2.5: Illustration of the arbitrary connectivity supported by *akid*. Forward connection, branching and mergine, and feedback connection are supported.

lifting has been done in the block (*Block* and its sub-classes), e.g. pre-condition setup, name scope maintenance, copy functionality for validation and copy functionality for distributed replicas, setting up and gathering visualization summaries, centralization of variable allocation, attaching debugging ops now and then etc.

`akid` offers various kinds of blocks that are able to connect to other blocks in an arbitrary way, as illustrated above. It is also easy to build one's own blocks. The `Kid` class is essentially an assembler that assemblies blocks provided by `akid` to mainly fulfill the task to train neural networks. Here we show how to build an arbitrary acyclic graph of blocks, to illustrate how to use blocks in `akid`. A brain is the data processing engine to process data supplied by *Sensor* to fulfill certain tasks. More specifically,

- it builds up blocks to form an arbitrary network
- offers sub-graphs for inference, loss, evaluation, summaries
- provides access to all data and parameters within

To use a brain, feed in data as a list, as how it is done in any other blocks. Some pre-specified brains are available under *akid.models.brains*. An example that sets up a brain using existing brains is:

```
# ... first get a feed sensor
sensor.setup()
brain = OneLayerBrain(name="brain")
input = [sensor.data(), sensor.labels()]
brain.setup(input)
```

Note in this case, *data()* and *labels()* of *sensor* returns tensors. It is not always the case. If it does not, saying return a list of tensors, you need do things like:

```
input = [sensor.data()]
input.extend(sensor.labels())
```

Act accordingly.

Similarly, all blocks work this way.

A brain provides easy ways to connect blocks. For example, a one layer brain can be built through the following:

```
class OneLayerBrain(Brain):
    def __init__(self, **kwargs):
        super(OneLayerBrain, self).__init__(**kwargs)
        self.attach(
            ConvolutionLayer(ksize=[5, 5],
                             strides=[1, 1, 1, 1],
                             padding="SAME",
                             out_channel_num=32,
                             name="conv1")
        )
        self.attach(ReLULayer(name="relu1"))
        self.attach(
            PoolingLayer(ksize=[1, 5, 5, 1],
                         strides=[1, 5, 5, 1],
                         padding="SAME",
                         name="pool1")
        )

        self.attach(InnerProductLayer(out_channel_num=10, name="ip1"))
        self.attach(SoftmaxWithLossLayer(
            class_num=10,
            inputs=[
                {"name": "ip1", "idxs": [0]},
                {"name": "system_in", "idxs": [1]}],
            name="loss"))
```

It assembles a convolution layer, a ReLU Layer, a pooling layer, an inner product layer and a loss layer. To attach a block (layer) that directly takes the outputs of the previous attached layer as inputs, just directly attach the block. If *inputs* exists, the brain will fetch corresponding tensors by name of the block attached and indices of the outputs of that layer. See the loss layer above for an example. Note that even though there are multiple inputs for the brain, the first attached layer of the brain will take the first of these input by default, given the convention that the first tensor is the data, and the remaining tensors are normally labels, which is not used till very late.

As an example to build more complex connectivity scheme, residual units can be built using `Brain` as shown below.

Parameter tuning

`akid` offers automatic parameter tuning through defining template using `tune` function.

`akid.train.tuner.`**`tune`**(*template, opt_paras_list=[{}], net_paras_list=[{}], repeat_times=1, gpu_num_per_instance=1, debug=False*)

A function *tune* that takes a Brain jinja2 template class and a parameters to fill the template in runtime. Parameters provided should complete the remaining network parameters in the template. The tuner is not aware of the content of the list items. It is up to the user to define template right, so parameters will be filled in the right place.

The jinja2 template must be a function named *setup*, and return a set up *Kid*. All necessary module imports should be put in the function instead of module level import usually.

The *tune* function would use all available GPUs to train networks with all given different set of parameters. If available GPUs are not enough, the ones that cannot be trained will wait till some others finish, and get its turn.

Parameter Tuning Usage

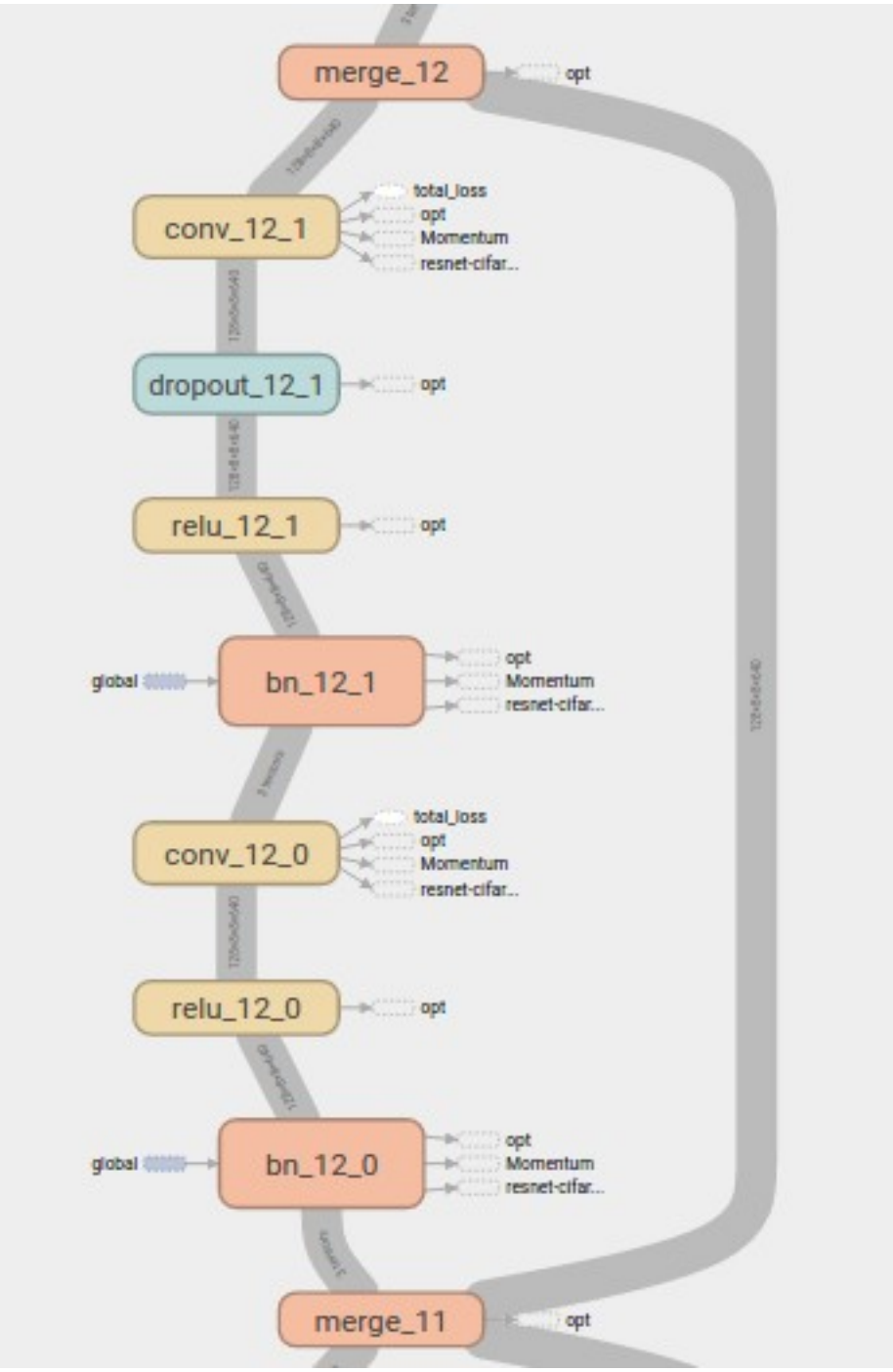


Fig. 2.6: One residual units. On the left is the branch that builds up patterns complexity, and on the right is the stem branch that shorts any layers to any layer. They merge at the at the start and at the end of the branching points.

Tunable parameters are divided into two set, network hyper parameters, *net_paras_list*, and optimization hyper parameters, *opt_paras_list*. Each set is specified by a list whose item is a dictionary that holds the actual value of whatever hyper parameters defined as jinja2 templates. Each item in the list corresponds to a tentative training instance. network paras and optimization paras combine with each other exponentially(or in Cartesian Product way if we could use Math terminology), which is to say if you have two items in network parameter list, and two in optimization parameters, the total number of training instances will be four.

Final training precisions will be returned as a list. Since the final precision normally will not be the optimal one, which normally occurs during training, the returned values are used for testing purpose only now

Run repeated experiment

To run repeated experiment, just leave *opt_paras_list* and *net_paras_list* to their default value.

GPU Resources Allocation

If the *gpu_num_per_instance* is None, a gpu would be allocated to each thread, otherwise, the length of the list should be the same with that of the training instance (aka the #opt_paras_list * #net_paras_list * repeat_times), or an int.

Given the available GPU numbers, a semaphore is created to control access to GPUs. A lock is created to control access to the mask to indicator which GPU is available. After a process has modified the gpu mask, it releases the lock immediately, so other process could access it. But the semaphore is still not release, since it is used to control access to the actual GPU. A training instance will be launched in a subshell using the GPU acquired. The semaphore is only released after the training has finished.

Example

For example, to tune the activation function and learning rates of a network, first we set up network parameters in *net_paras_list*, optimization parameters in *opt_paras_list*, build a network in the *setup* function, then pass all of it to tune:

```
    net_paras_list = []
    net_paras_list.append({
        "activation": [
            {"type": "relu"},
            {"type": "relu"},
            {"type": "relu"},
            {"type": "relu"}],
        "bn": True})
    net_paras_list.append({
        "activation": [
            {"type": "maxout", "group_size": 2},
            {"type": "maxout", "group_size": 2},
            {"type": "maxout", "group_size": 2},
            {"type": "maxout", "group_size": 5}],
        "bn": True})

    opt_paras_list = []
    opt_paras_list.append({"lr": 0.025})
    opt_paras_list.append({"lr": 0.05})

    def setup(graph):

        brain.attach(cnn_block(
            ksize=[8, 8],
            init_para={
                "name": "uniform",
                "range": 0.005},
            wd={"type": "l2", "scale": 0.0005},
```

```
            out_channel_num=384,
            pool_size=[4, 4],
            pool_stride=[2, 2],
            activation={{ net_paras["activation"][1] }},
            keep_prob=0.5,
            bn={{ net_paras["bn"] }}))

    tune(setup, opt_paras_list, net_paras_list)
```

2.2.3 Distributed Computation

The distributed computing stack is responsible to handle concurrency and communication between different computing nodes, so the end user only needs to deal with how to build a power network. All complexity has been hidden in the class *Engine*. The usage of *Engine* is just to pick and use.

More specifically, *akid* offers built-in data parallel scheme in form of class *Engine*. Currently, the engine mainly works with neural network training, which is be used with *Kid* by specifying the engine at the construction of the kid.

As an example, we could do data parallelism on multiple computing towers using:

```
kid = kids.Kid(
    sensor,
    brain,
    MomentumKongFu(lr_scheme={"name": LearningRateScheme.placeholder}),
    engine={"name": "data_parallel", "num_gpu": 2},
    log_dir="log",
    max_epoch=200)
```

The end computational graph constructed is illustrated below

2.2.4 Distributed Deployment

The distributed deployment stack handles the actual production environment, thus decouples the development/prototyping environment and production environment. Mostly, this stack is about how to orchestrate with existing distributed ecosystem. Tutorials will be provided when a production ready setup has been thoroughly investigated. Tentatively, `glusterfs` and `Kubernetes` are powerful candidates.

2.3 Comparison with existing packages

`akid` differs from existing packages from the perspective that it aims to integrate technology stacks to solve both research prototyping and industrial production. Existing packages mostly aim to solve problems in one of the stack. `akid` reduces the friction between different stacks with its unique features. We compare `akid` with existing packages in the following briefly.

Theano, Torch, Caffe, MXNet are packages that aim to provide a friendly front end to complex computation back-end that are written in C++. Theano is a python front end to a computational graph compiler, which has been largely superseded by Tensorflow in the compilation speed, flexibility, portability etc, while `akid` is built on of Tensorflow. MXNet is a competitive competitor to Tensorflow. Torch is similar with theano, but with the front-end language to be Lua, the choice of which is mostly motivated from the fact that it is much easier to interface with C using Lua than Python. It has been widely used before deep learning has reached wide popularity, but is mostly a quick solution to do research in neural networks when the integration with community and general purpose production programming are not pressing. Caffe is written in C++, whose friendly front-end, aka the text network configuration file, loses its affinity when the model goes more than dozens of layer.

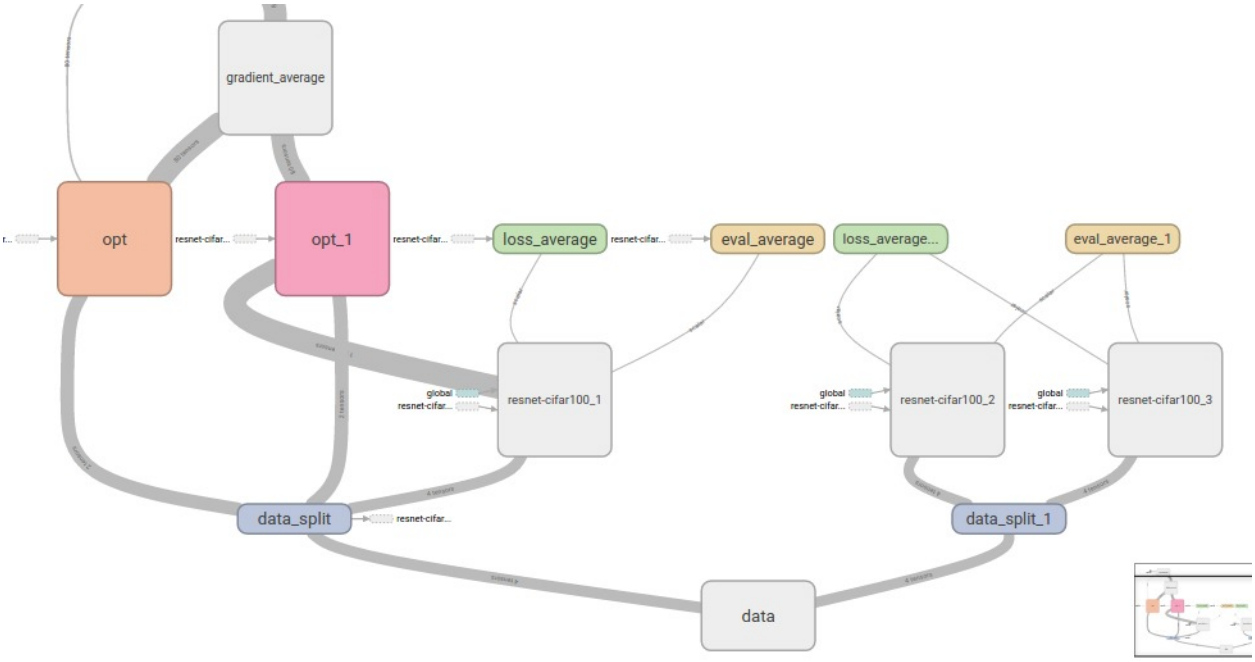


Fig. 2.7: Illustration of computational graph constructed by a data parallel engine. It partitions a mini-batch of data into subsets, as indicated by the *data_split* blue blocks, and passes the subsets to replicates of neural network models at different coputing tower, as indicated by the gray blocks one level above blue blocks, then after the inference results have been computed, the results and the labels (from the splitted data block) will be passed to the optimizers in the same tower, as indicated by red and orange blocks named *opt*, to compute the gradients. Lastly, the gradients will be passed to an tower that computes the average of the gradients, and pass them back to neural networks of each computing towers to update their parameters.

DeepLearning4J is an industrial solution to neural networks written in Java and Scala, and is too heavy weight for research prototyping.

Perhaps the most similar package existing with `akid` is Keras, which both aim to provide a more intuitive interface to relatively low-level library, i.e. Tensorflow. `akid` is different from Keras at least two fundamental aspects. First, `akid` mimics how signals propagates in nature by abstracting everything as a semantic block, which holds many states, thus is able to provide a wide range of functionality in a easily customizable way, while Keras uses a functional API that directly manipulates tensors, which is a lower level of abstraction, e.g. it have to do class attributes traverse to retrieve layer weights with a fixed variable name while in `akid` variable are retrieved by names. Second, Keras mostly only provides an abstraction to build neural network topology, which is roughly the programming paradigm stack of `akid`, while `akid` provides unified abstraction that includes application stack, programming stack, and distributed computing stack. A noticeable improvement is Keras needs the user to handle communication and concurrency, while the distributed computing stack of `akid` hides them.

CHAPTER 3

HOW TO

In this HOW-TO page, available blocks are described.

TODO: finished this section

3.1 Block

akid builds another layer of abstraction on top of *Tensor*: *Block*. Tensor can be taken as the media/formalism signal propagates in digital world, while Block is the data processing entity that processes inputs and emits outputs.

It coincides with a branch of "ideology" called dataism that takes everything in this world is a data processing entity. An interesting one that may come from *A Brief History of Tomorrow* by Yuval Noah Harari.

Best designs mimic nature. *akid* tries to reproduce how signals in nature propagates. Information flow can be abstracted as data propagating through inter-connected blocks, each of which processes inputs and emits outputs. For example, a vision classification system is a block that takes image inputs and gives classification results. Everything is a *Block* in *akid*.

A block could be as simple as a convonlutional neural network layer that merely does convolution on the input data and outputs the results; it also be as complex as an acyclic graph that inter-connects blocks to build a neural network, or sequentially linked block system that does data augmentation.

Compared with pure symbol computation approach, like the one in tensorflow, a block is able to contain states associated with this processing unit. Signals are passed between blocks in form of tensors or list of tensors. Many heavy lifting has been done in the block (*Block* and its sub-classes), e.g. pre-condition setup, name scope maintenance, copy functionality for validation and copy functionality for distributed replicas, setting up and gathering visualization summaries, centralization of variable allocation, attaching debugging ops now and then etc.

3.2 Source

Signals propagated in nature are all abstracted as a source. For instance, an light source (which could be an image source or video source), an audio source, etc.

As an example, saying in supervised setting, a source is a block that takes no inputs (since it is a source), and outputs data. A concrete example could be the source for the MNIST dataset:

```
source = MNISTFeedSource(name="MNIST",
                         url='http://yann.lecun.com/exdb/mnist/',
                         work_dir=AKID_DATA_PATH + '/mnist',
                         center=True,
                         scale=True,
```

```
                            num_train=50000,
                            num_val=10000)
```

The above code creates a source for MNIST. It is supposed to provide data for placeholders of tensorflow through method *get_batch*. Say:

```
source.get_batch(100, get_val=False)
```

would return a tuple of numpy array of *(images, labels)*.

It could be used standalone, or passed to a *Sensor*.

3.2.1 Developer Note

A top level abstract class *Source* implements basic semantics of a natural source. Other abstract classes keep implementing more concrete sources. Abstract *Source* s need to be inherited and abstract methods implemented before it could be used. To create a concrete *Source*, you could use multiple inheritance to compose the *Source* you needs. Available sources are kept under module *sources*.

3.3 Sensor

The interface between nature and an (artificial) signal processing system, saying a brain, is a *Sensor*. It does data augmentation (if needed), and batches datum from *Source*.

Strictly speaking, the functional role of a sensor is to convert the signal in the natural form to a form the data processing engine, which is the brain in this case, could process. It is a Analog/Digital converter. However, the input from *Source* is already in digital form, so this function is not there anymore. But the data batching, augmentation and so on could still be put in preprocessing. Thus we still use the name sensor for concept reuse.

Mathematically, it is a system made up with a series of linked blocks that do data augmentation.

As an example, again saying in supervised setting, a sensor is a block that takes a data source and output sensed (batched and augmented) data. A sensor needs to be used along with a source. A concrete example could be the sensor for the MNIST dataset. Taking a *Source*, we could make a sensor:

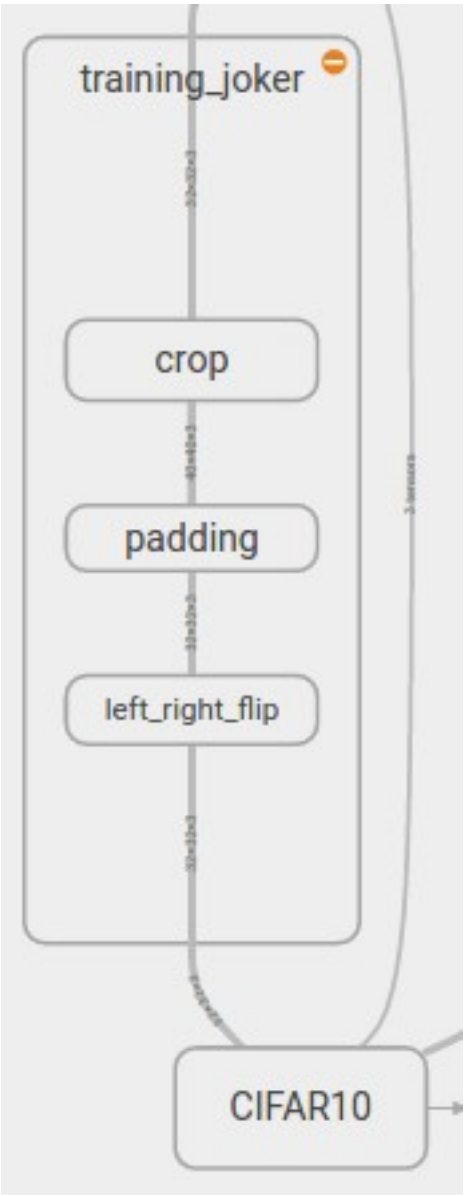
```
sensor = FeedSensor(name='data', source_in=source)
```

The type of a sensor must match that of a source.

For *IntegratedSensor*, it is supported to add *Joker* to augment data. The way to augment data is similar with building blocks using *Brain*, but simpler, since data augmentation is added sequentially, shown in the following:

```
sensor = IntegratedSensor(source_in=cifar_source,
                          batch_size=128,
                          name='data')
sensor.attach(FlipJoker(flip_left_right=True, name="left_right_flip"))
sensor.attach(PaddingLayer(padding=[4, 4]))
sensor.attach(CropJoker(height=32, width=32, name="crop"))
```

The end computational graph is shown as following.



3.4 Brain

A brain is the data processing engine to process data supplied by *Sensor* to fulfill certain tasks. More specifically,

- it builds up blocks to form an arbitrary network
- offers sub-graphs for inference, loss, evaluation, summaries
- provides access to all data and parameters within

To use a brain, feed in data as a list, as how it is done in any other blocks. Some pre-specified brains are available under *akid.models.brains*. An example that sets up a brain using existing brains is:

```
# ... first get a feed sensor
sensor.setup()
brain = OneLayerBrain(name="brain")
```

```
input = [sensor.data(), sensor.labels()]
brain.setup(input)
```

Note in this case, *data()* and *labels()* of *sensor* returns tensors. It is not always the case. If it does not, saying return a list of tensors, you need do things like:

```
input = [sensor.data()]
input.extend(sensor.labels())
```

Act accordingly.

Similarly, all blocks work this way.

A brain provides easy ways to connect blocks. For example, a one layer brain can be built through the following:

```
class OneLayerBrain(Brain):
    def __init__(self, **kwargs):
        super(OneLayerBrain, self).__init__(**kwargs)
        self.attach(
            ConvolutionLayer(ksize=[5, 5],
                             strides=[1, 1, 1, 1],
                             padding="SAME",
                             out_channel_num=32,
                             name="conv1")
        )
        self.attach(ReLULayer(name="relu1"))
        self.attach(
            PoolingLayer(ksize=[1, 5, 5, 1],
                         strides=[1, 5, 5, 1],
                         padding="SAME",
                         name="pool1")
        )

        self.attach(InnerProductLayer(out_channel_num=10, name="ip1"))
        self.attach(SoftmaxWithLossLayer(
            class_num=10,
            inputs=[
                {"name": "ip1", "idxs": [0]},
                {"name": "system_in", "idxs": [1]}],
            name="loss"))
```

It assembles a convolution layer, a ReLU Layer, a pooling layer, an inner product layer and a loss layer. To attach a block (layer) that directly takes the outputs of the previous attached layer as inputs, just directly attach the block. If *inputs* exists, the brain will fetch corresponding tensors by name of the block attached and indices of the outputs of that layer. See the loss layer above for an example. Note that even though there are multiple inputs for the brain, the first attached layer of the brain will take the first of these input by default, given the convention that the first tensor is the data, and the remaining tensors are normally labels, which is not used till very late.

3.5 KongFu

3.6 System

This module provides systems of different topology to compose ‘Block‘s to create more complex blocks. A system does not concern which type of block it holds, but only concerns the mathematical topology how they connect.

CHAPTER 4

Tutorials of akid

4.1 Distributed akid

In this tutorial, we will write a program that does computation distributedly. The full code, `dist_akid` can be found under `example` folder.

OK, now it is just some pointers.

We use a `Source`, a `Sensor`, a `Brain` in this case. Read How To to know what they do. Also read the tensorflow tutorial for distributed computation. This tutorial ports the distributed example provided by tensorflow. The usage of the program is the same as in the tensorflow tutorial.

After successfully running the program, you are supposed to see outputs like:

```
2.45249
2.40535
2.29056
2.2965
2.25567
2.27914
2.26652
2.27446
2.2911
2.26182
2.17706
2.18829
2.23567
2.21965
2.20997
2.14844
2.10352
2.066
2.12029
2.10526
2.10102
2.03739
2.04613
2.05246
2.04463
2.03297
```

which is the training loss.

CHAPTER 5

Architecture and Design Principles

TODO: finish this section

5.1 Architecture

5.1.1 Kick start clock

A centralized clock is available in `akid` to model the elapsing time in the physical world. It is roughly the training step, or iteration step when training a neural network.

If any training is supposed to done with the machinery provided `akid`, the clock needs to be manually started using the following snippets. However, if you are using `Kid` class, it will be called automatically.

```
from akid.common import init
init()
```

5.1.2 Model Abstraction

All core classes a sub class of `Block`.

The philosophy is to bring neural network to its biological origin, the brain, a data processing engine that is tailored to process hierarchical data of the universe. The universe is built block by block, from micro world, modules consisting of atoms, to macro world, buildings made up by bricks and windows, and cosmic world, billions of stars creating galaxies.

Tentatively, there are two kinds of blocks — simple blocks and complex blocks. Simple blocks are traditional processing units such as convolutional layers. While complex blocks holds sub-blocks with certain topology. Complex blocks are `System`. A module `systems` offers different `Systems` to model the mathematical topological structures how data propagates. These two types of blocks build up the recursive scheme to build arbitrarily complex blocks.

5.1.3 Model and Computation

Blocks are responsible for easing building computational graph. Given two-phase procedure that first builds computational graph, then executes that graph, each object (including but not limiting to blocks) who actually needs to do computation has its own computational components, a graph and a session. If no higher level one is given, the block will create a suite of its own; otherwise, it will use what it is given.

Graph

A model's construction is separated from its execution environment.

To use most classes, its `setup` method should be called before anything else. This is tied with Tensorflow's two-stage execution mechanism: first build the computational graph in python, then run the graph in the back end. The `setup` of most classes build and do necessary initialization for the first stage of the computation. The caller is responsible for passing in the right data for `setup`.

`setup` should be called under a `tf.Graph()` umbrella, which is in the simplest case is a context manager that open a default graph:

TODO: build the umbrella within.

```
with self.graph.as_default():
    # Graph building codes here
```

That is to say if you are going to use certain class standalone, a graph context manager is needed.

Each `System` takes a `graph` argument on construction. If no one is given, it will create one internally. So no explicit graph umbrella is needed.

Session

TODO: abstract this within

A graph hold all created blocks. To actually run the computational graph, all computational methods has an `sess` argument to take an opened `tf.Session()` to run within, thus any upper level execution environment could be passed down. The upper level code is responsible to set up a session. In such a way, computational graph construction does not need to take care of execution. However, for certain class, such as a Survivor, if an upper level session does not exist, a default one will be created for the execution for convenience.

This allows a model to be deployed on various execution environment.

5.2 Design Principles

5.2.1 Compactness

The design principle is to make the number of concepts exist at the same time as small as possible.

5.2.2 LEGO Blocks

The coding process is to assembly various smaller blocks to form necessary functional larger blocks.

The top level concept is a survivor. It models how an agent explore the world by learning in order to survive, though the world has not been modeled yet. Up to now, it could be certain virtual reality world that simulate the physical world to provide environment to the survivor.

A `Survivor` assemblies together a `Sensor`, a `Brain` and a `KongFu`. A `Sensor` assemblies together `Jokers` and data `Sources`. A `Brain` assemblies together a number of `ProcessingLayer` to form a neural networking.

5.2.3 Distributed Composition

Every sub-block of a large block should be self-contained. A large block only needs minimum amount of information from a sub block. They communicate through I/O interfaces. So the hierarchical composition scale up in a distributed way and could goes arbitrary deep with manageable complexity.

CHAPTER 6

Model Zoo

Model building source files can be found under example folder, where AlexNet, Maxout Network, VGG style network, Residual Network, are reproduced.

NOTE: since some of the examples are not included in tests, it is possible the examples have some syntax errors, which should be easy to fix. File an issue any time, or make a pull request anytime.

NOTE: Considering the package is not ready for public announcement yet, the more feedback or issues I get, the more motivated I am to keep improving the usibility of the package. Thanks for taking the time!

Fork me at https://github.com/shawnLeeZX/akid !

`akid` is a python package written for doing research in Neural Network. It also aims to be production ready by taking care of concurrency and communication in distributed computing. It is built on Tensorflow. If combining with GlusterFS, Docker and Kubernetes, it is able to provide dynamic and elastic scheduling, auto fault recovery and scalability.

It aims to enable fast prototyping and production ready at the same time. More specifically, it

- supports fast prototyping
  - built-in data pipeline framework that standardizes data preparation and data augmentation.
  - arbitrary connectivity schemes (including multi-input and multi-output training), and easy retrieval of parameters and data in the network
  - meta-syntax to generate neural network structure before training
  - support for visualization of computation graph, weight filters, feature maps, and training dynamics statistics.
- be production ready
  - built-in support for distributed computing
  - compatibility to orchestrate with distributed file systems, docker containers, and distributed operating systems such as Kubernetes. (This feature mainly is a best-practice guide for K8s etc, which is under experimenting and not available yet.)

The name comes from the Kid saved by Neo in *Matrix*, and the metaphor to build a learning agent, which we call *kid* in human culture.

It distinguish itself from an unique design, which is described in the following. *akid* builds another layer of abstraction on top of *Tensor*: *Block*. Tensor can be taken as the media/formalism signal propagates in digital world, while Block is the data processing entity that processes inputs and emits outputs.

It coincides with a branch of "ideology" called dataism that takes everything in this world is a data processing entity. An interesting one that may come from *A Brief History of Tomorrow* by Yuval Noah Harari.

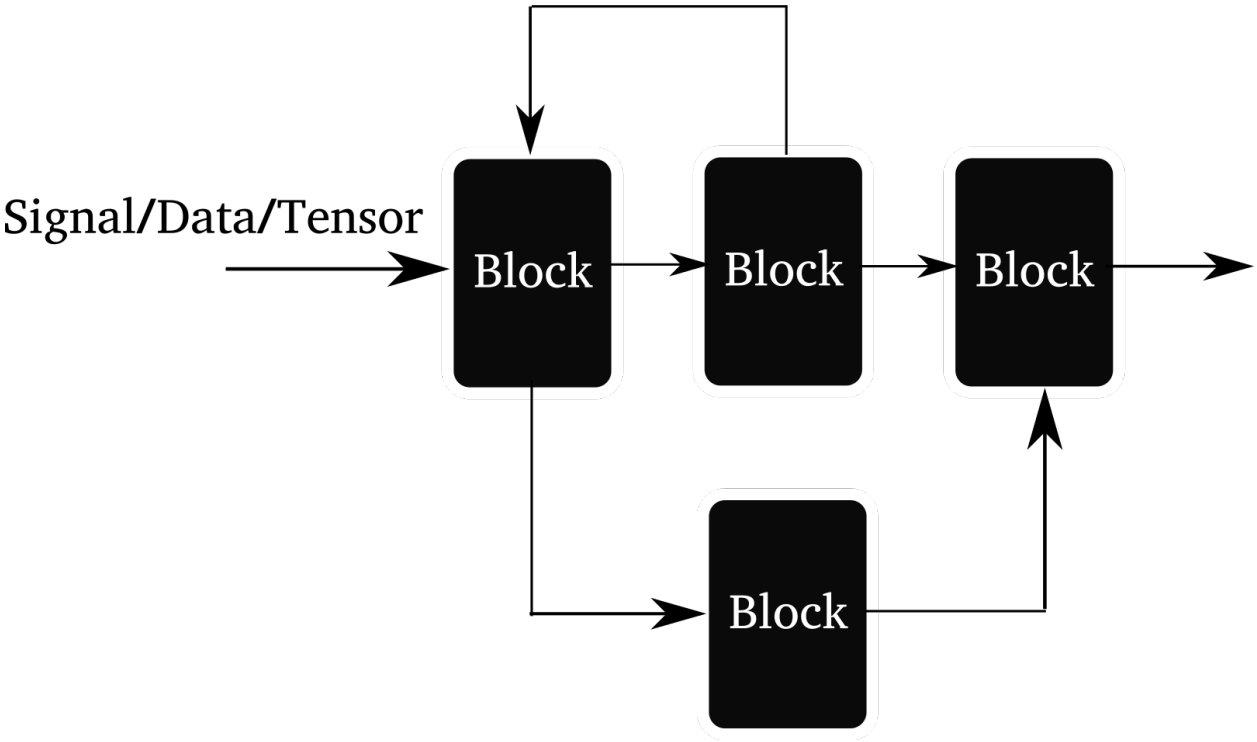


Fig. 6.1: Illustration of the arbitrary connectivity supported by *akid*. Forward connection, branching and mergine, and feedback connection are supported.

Best designs mimic nature. *akid* tries to reproduce how signals in nature propagates. Information flow can be abstracted as data propagating through inter-connected blocks, each of which processes inputs and emits outputs. For example, a vision classification system is a block that takes image inputs and gives classification results. Everything is a *Block* in *akid*.

A block could be as simple as a convonlutional neural network layer that merely does convolution on the input data and outputs the results; it also be as complex as an acyclic graph that inter-connects blocks to build a neural network, or sequentially linked block system that does data augmentation.

Compared with pure symbol computation approach, like the one in tensorflow, a block is able to contain states associated with this processing unit. Signals are passed between blocks in form of tensors or list of tensors. Many heavy lifting has been done in the block (*Block* and its sub-classes), e.g. pre-condition setup, name scope maintenance, copy functionality for validation and copy functionality for distributed replicas, setting up and gathering visualization summaries, centralization of variable allocation, attaching debugging ops now and then etc.

a

akid.core.blocks, 27
akid.core.brains, 9
akid.core.engines, 13
akid.core.sensors, 18
akid.core.sources, 17
akid.core.systems, 20

A

akid.core.blocks (module), 6, 17, 27
akid.core.brains (module), 9, 19
akid.core.engines (module), 13
akid.core.sensors (module), 18
akid.core.sources (module), 17
akid.core.systems (module), 20

T

tune() (in module akid.train.tuner), 10