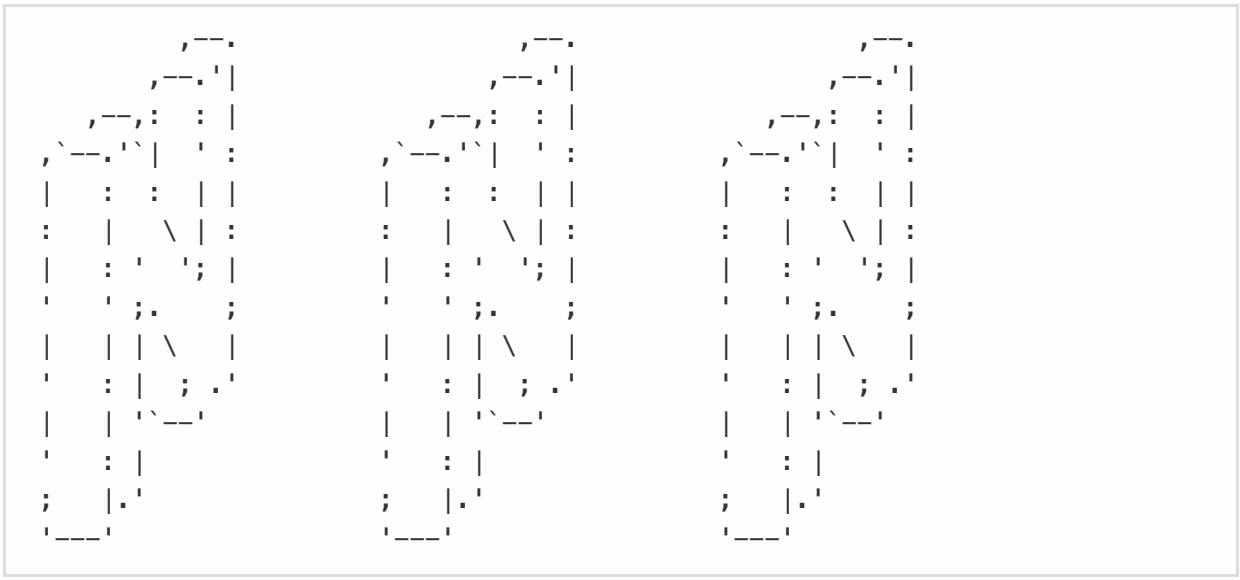
Numpy Neural Network



Numpy Neural Network(NNN) is neural network framework based on only numpy which is fundamental package for scientific computing with Python.

NNN support *activations, metrics, layers, optimizers* and *loss functions*. You can build network with high-level expressions like keras on tensorflow or PyTorch.

Below code, create two-layer network using *sigmoid* as activation, solving problem given as assignment. You can use the same interface for *numpy* and *tensorflow*, depending on whether you define `nn` as `tf.keras` or `nnn`.

```
model = nn.Sequentail([
    nn.layers.Dense(2, input_dim=2, activation='sigmoid'),
    nn.layers.Dense(1, input_dim=2, activation='sigmoid'),
    # more layers if you need like
    # nn.layers.Dense(2, input_dim=2, activation='relu'),
])
```

After compile you model, use `fit` and `evaluate` to train and test model.

```
model.fit(train.X, train.Y, epochs=100)
loss, acc = model.evaluate(test.X, test.Y)
```

Run!

There are several arguments for define task. You can run *numpy* and *tensorflow* implementations using the `--mode` flag as `np` or `tf`. Also you can set `epoch`, `size`, `lr`, `optimizer`, `loss`, `layer`, `repeat` and `seed` flags.

```
python main.py
--mode tf
--lr .1
--optimizer SGD
--loss binary_corssentropy
```

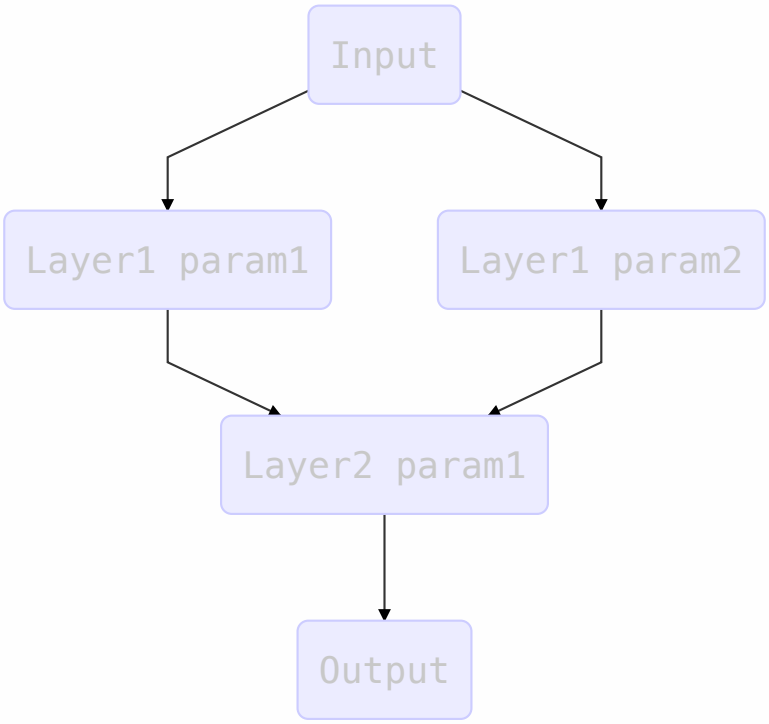
Also, You can find help and possible options by typing

```
python main.py --help
```

For dependencies, install pcakge list on `requirements.txt`

```
pip install -r requirements.txt
```

Base Network Architecture



In this assignment, use `SGD` as optimizer, `BCE` as loss function and `sigmoid` as activation function. Each test performed on 128 dataset with 5000 epoch, learning rate 0.1. The random seed is fixed at 2.

Data

`utils.data.Dataset` generate a random sample from *standard normal* distribution using `np.random.randn`. Set normalized dataset with `--normal` flag.

`utils.data.Dataset` generate labels from `X`, `Y` where $X^2 > Y$. If you want custom dataset, override `utils.data.Dataset` and define `X` and `Y`.

Comparison Results

%	Base	LR[.5]	LR[.01]	Loss[MSE]	Data[Normal]	Data[256]	Data[2048]
TF	96.95	97.97	96.88	97.11	97.27	97.27	99.28
NNN	97.27	96.88	69.92	97.27	95.94	95.86	96.45

Implementation

The **Layer** must determine the size of *input dimension* and the size of the *output dimension* and also specify activation function. `forward` and `backward` is default method for network for forward propagation and backward propagation. Also **Layer** calls are replaced with `forward`. `after_forward` is method for keep *last output* and pass the output to *activation* if exists. `update` call optimizer to update parameters. `paraemters` is getter and setter, which returns all *parameters* belongs to layer and update it. You can make custom layer by inheriting the **Layer**.

In **Dense**(Layer), `forward` just dot product the *input value* by the *parameters* and add *bias*. `backward` calculate gradient of *parameters* and *bias* using last saved *output value* and passed *gradient*.

Activation works like a **Layer**, but basically it does not need an *initial parameters* such as *input dimension* or *output dimension*, so it can be called like a function even though it still performs `forward` and `backward`. `Sigmoid` and `ReLU` are implemented. You can make custom activation function by inheriting the **Activation**.

Optimizer update parmaters by *learning rate* or other *hyper paramters*. `SGD` and `Adam` are implemented.

NNN support optimzers (from now, only support `SGD` and `Adam`) Loss (support `MSE` and `BCE`) and Metric(only `accuracy` now). You can easily define it as you do in *tensorflow*.

```
optimizer = nn.optimizers.sgd(lr=.1)
loss_function = nn.losses.MSE()
metric = nn.metrics.accuracy()
```

Save parameter retrival

Numpy support *np.ndarray* save method. `parameters.npz` is saved *paramters*.

```
parameters = np.load('parameters.npz')

for i, layer in enumerate(model.layers):
    layer.parameters = parameters[str(i)]
```