# gNet (v0.1)

Mehmet Gökçay KABATAŞ - MGokcayK

github.com/MGokcayK

mgokcaykdev@gmail.com

July 9, 2020

# Contents

# Chapter 1: What is gNet?

gNet is a mini Deep Learning(DL) library. It is written to understand how DL works. It is running on CPU. It is written on Python language and used :

- Numpy for linear algebra calculations
- Matplotlib for plottings
- Texttable for proper printing of model summary in cmd
- wget for download MNIST data
- idx2numpy for load MNIST data

some 3rd party libraries.

During devolopment, Tensorflow, Keras, Pytorch and some other libraries examined. Keras end-user approach is used. Therefore, if you are familiar with Keras, you can use gNet easily.

gNet has not a lot functions and methods for now, because subject is written when they needed to learn. Also, gNet is personal project. Thus, its development process depends on author learning process.

## 1.1 Installation

Installation can be done with pip or clone the git and use in local file of your workspace.

To install with [pip][https://pypi.org]

```
pip install gNet
```

Code 1.1: Install with pip

# Chapter 2: Usage with Examples

gNet is also mini computation library based on numpy. Details can be find in next chapters. This chapter is abstract of other chapters and examples of them. One of the ability of gNet is taking gradient (derivative) of function. It is used in gNet; yet, user can calculate gradient of custom functions. Let's look at it.

## 2.1 How calculate gradient?

Gradient calculation has two rules. First rule is tensor which inputs of function should have_grad = True. Second rule is call backward methods from result. To explain it, give example. There is a function $f(x, y) = 3x^2 + 2y$, user want to find $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ where $x = 2, y = 5$.

Analytically, $\frac{\partial f}{\partial x} = 6x$, $\frac{\partial f(2,5)}{\partial x} = 12$ and $\frac{\partial f}{\partial y} = 2$, $\frac{\partial f(2,5)}{\partial y} = 2$.

```
import gNet.tensor as T

x = T.Tensor(2., have_grad=True)
y = T.Tensor(5., have_grad=True)

f = 3 * T.power(x, 2) + 2 * y  # also 3 * x**2 + 2 * y can be used.
f.backward() # calculate derivatives.

print(f, '\n', x.grad, '\n', y.grad)

-----------------------------------------------
Tensor(22.0,shape=(()), have\_grad=True)
Tensor(12.0,shape=(()), have\_grad=False)
Tensor(2.0,shape=(()), have\_grad=False)
```

Code 2.1: Calculation of gradient.

## 2.2 How load MNIST Dataset?

To load MNIST Dataset, gNet has class for it. In ‘utils‘ module, ‘MNIST_Downloader‘ class does the job. Usage is easy. Let’s look at it.

```
from gNet import utils

mnist = utils.MNIST_Downloader()
x_train, y_train = mnist.load_train()
x_test, y_test = mnist.load_test()

# to normalize data
x_train, x_test = x_train / 255.0, x_test /255.0

print('SHAPES :', x_train.shape, y_train.shape, x_test.shape, y_test.
    shape)
-----------------------------------------------
SHAPES : (60000, 28, 28) (60000,) (10000, 28, 28) (10000,)
```

Code 2.2: Load MNIST Dataset.

## 2.3 How make one-hot vector of label of MNIST Dataset?

gNet has work on one-hot vector for label values. If dataset is not one-hot vector, gNet has function for it. In ‘utils‘ module, ‘make_one_hot‘ function does the job. Usage is easy. Let’s look at it.

```
from gNet import utils

num_classes = 10 # for MNIST dataset
y_train = utils.make_one_hot(y_train, num_classes)
y_test = utils.make_one_hot(y_test, num_classes)

print('SHAPES :', x_train.shape, y_train.shape, x_test.shape, y_test.
    shape)
-----------------------------------------------
SHAPES : (60000, 28, 28) (60000, 10) (10000, 28, 28) (10000, 10)
```

Code 2.3: Making one-hot vector of label of dataset.

## 2.4 How create MLP model?

MLP model contain only Dense layers. Let's create model which has one hidden layer and one output layer. **Note that before first Dense layer, flatten layer should be used and in this structure, flatten layer is input layer of model. Therefore, it has 'input_shape' parameter.**

```
from gNet import model as gModel
from gNet import layer

model = gModel.Model()

# add first layer as flatten layer with input_shape
model.add(layer.Flatten(input_shape=x_train[0].shape))

# add hidden layer with ReLU activation function
model.add(layer.Dense(128,'relu'))

# add output layer with Softmax activaion function
model.add(layer.Dense(10, 'softmax' ))
```

Code 2.4: Create MLP model.

## 2.5 How create CNN model?

CNN model contain Conv2D, Activation, Pooling (Max or Average), Flatten and Dense layers. Let's create model which has one Conv2D layer. **Note that before first Dense layer, flatten layer should be used and in this structure but not with 'input_shape' as MLP model. Conv2D layer is input layer of model. Therefore, it has 'input_shape' parameter.**

**Note about Conv2D is input should have 3 dimensional and have 'channel first'**. To do it for MNIST Dataset, 3rd dimension should be added.

```
from gNet import model as gModel
from gNet import layer

# make it channel first 3D data.
x_test = x_test[:, None, :, :]
x_train = x_train[:, None, :, :]

model = gModel.Model()
```

```
# add first layer as Conv2D layer with input_shape
model.add(layer.Conv2D(filter=5, kernel=(9,9),stride=(1,1),padding=0,
    input_shape=x_train[0].shape, use_bias=True))

# activate output
model.add(layer.Activation('relu'))

# pool output
model.add(layer.MaxPool2D())

# flat output for dense layer
model.add(layer.Flatten())

# add hidden layer with ReLU activation function
model.add(layer.Dense(128,'relu'))

# add output layer with Softmax activaion function
model.add(layer.Dense(10, 'softmax' ))
```

Code 2.5: Create CNN model.

## 2.6 How create model with Dropout layer?

Dropout layer can be added anywhere except as input layer. General usage of Dropout layer after hidden Dense layers.

```
from gNet import model as gModel
from gNet import layer

model = gModel.Model()

# add first layer as flatten layer with input_shape
model.add(layer.Flatten(input_shape=x_train[0].shape))

# add hidden layer with ReLU activation function
model.add(layer.Dense(128,'relu'))

# add dropout with 0.5 probability
model.add(layer.Dropout(0.5))

# add output layer with Softmax activaion function
model.add(layer.Dense(10, 'softmax' ))
```

Code 2.6: Create model with Dropout.

## 2.7 How create model with Batch Normalization layer?

Batch Normalization layer can be after Conv2D, Activation (sometimes previous from it) and Dense layer. General suggestion of using BN layer in CNN is after Activation Layer.

```
from gNet import model as gModel
from gNet import layer

model = gModel.Model()

# add first layer as Conv2D layer with input_shape
model.add(layer.Conv2D(filter=5, kernel=(9,9),stride=(1,1),padding=0,
    input_shape=x_train.shape[1:], use_bias=True))

# activate output
model.add(layer.Activation('relu'))

# add Batch Normalization
model.add(layer.BatchNormalization())

# pool output
model.add(layer.MaxPool2D())

# flat output for dense layer
model.add(layer.Flatten())

# add hidden layer with ReLU activation function
model.add(layer.Dense(128,'relu'))

# add output layer with Softmax activaion function
model.add(layer.Dense(10, 'softmax' ))
```

Code 2.7: Create model with Batch Normalization.

## 2.8 How create supervised learning structure and setup?

After model creation, structure should be created then train it. Before training, loss function and optimizer should be setted. Let's do it.

```
from gNet import neuralnetwork as NN

# create structure and put created model into structure
net = NN.NeuralNetwork(model)

# setup structure
net.setup(loss_function='cce', optimizer='adam')
```

Code 2.8: Set loss function and optimizer.

## 2.9 How create supervised learning structure and setup with custom parameters?

If user want to use built-in loss function, optimizer or both of them with the custom parameters, user need to create needed classes from modules. Creation of own loss function class explained in Chapter 8. Creation of own optimizer class explained in Chapter 11.

```
from gNet import neuralnetwork as NN
from gNet import loss_functions
from gNet import optimizer

# create structure and put created moden into structure
net = NN.NeuralNetwork(model)

# create loss function
loss = loss_functions.CategoricalCrossEntropy()

# create optimizer
opt = optimizer.Adam(lr=0.0001)

# setup structure
net.setup(loss_function=loss, optimizer=opt)
```

Code 2.9: Set loss function and optimizer with custom parameters.

## 2.10 How train and evaluate model?

After setup the structure, training of model and evaluating is easy.

```
# train model
```

```
net.train(x_train, y_train, batch_size=32, epoch=10, printing=['loss',
    'accuracy'])

# evaluate model
net.evaluate(x_test, y_test)
```

Code 2.10: Train and evaluate model.

## 2.11 How train and evaluate model with validation?

If model training with validation, there is two way to do it. First way assigning validation rate. To print validation loss and accuracy done by editting printing list.

```
# train model with 20% validation.
net.train(x_train, y_train, batch_size=32, epoch=10, val_rate = 0.2,
    printing=['loss', 'accuracy', 'val_loss', 'val_acc'])

# evaluate model
net.evaluate(x_test, y_test)
```

Code 2.11: Train and evaluate model with validation rate.

Second way is assigning validation data.

```
# train model with validation data where valid_x is sample validation
    data and valid_y is labels of them.
net.train(x_train, y_train, batch_size=32, epoch=10, val_x= valid_x,
    val_y=valid_y, printing=['loss', 'accuracy', 'val_loss', 'val_acc'])

# evaluate model
net.evaluate(x_test, y_test)
```

Code 2.12: Train and evaluate model with validation data.

More details can be found in Chapter 4.

## 2.12 How train and evaluate model one batch?

After setup the structure, training batch by batch is easy. This approach is suggested when data is to big to load memory. Usage is same as ‘train‘ method.

```

batch_size = 32
```

```
epoch = 10

for e in range(epoch):

  # get start index of data for each epoch.
  _starts = np.arange(0, x_train.shape[0], batch_size)
  # if data willing to shuffled, shuffle it by shuffling index for each
    epoch.
  np.random.shuffle(_starts)

  # run batchs
  print("\nEpoch : ", e + 1)
  for _start in _starts:
    # find last index of batch and iterate other parameters.
    _end = _start + batch_size
    _x_batch = x_train[_start:_end]
    _y_batch = y_train[_start:_end]
    net.train_one_batch(_x_batch, _y_batch, printing=['loss', 'accuracy
   '], single_batch=False)

  # set new epoch
  net.new_epoch()


# get start index of data for evaluation
_starts_test = np.arange(0, x_test.shape[0], batch_size)
# if data willing to shuffled, shuffle it by shuffling index for
   evaluation
np.random.shuffle(_starts_test)

# run evaluation
print("\nEvaluate:")

for _start_t in _starts_test:
  # find last index of batch and iterate other parameters.
  _end_t = _start_t + batch_size
  _x_batch_t = x_test[_start_t:_end_t]
  _y_batch_t = y_test[_start_t:_end_t]
  net.evaluate_one_batch(_x_batch_t, _y_batch_t, single_batch=False)
```

Code 2.13: Train and evaluate model on batch.

## 2.13 How train and evaluate model with validation one batch?

If model training one batch with validation, there is two way to do it. First way assigning validation rate. To print validation loss and accuracy done by editting printing list.

```
batch_size = 32
epoch = 10

for e in range(epoch):

  # get start index of data for each epoch.
  _starts = np.arange(0, x_train.shape[0], batch_size)
  # if data willing to shuffled, shuffle it by shuffling index for each
    epoch.
  np.random.shuffle(_starts)

  # run batchs
  print("\nEpoch : ", e + 1)
  for _start in _starts:
    # find last index of batch and iterate other parameters.
    _end = _start + batch_size
    _x_batch = x_train[_start:_end]
    _y_batch = y_train[_start:_end]
    # train batch with 20% validation
    net.train_one_batch(_x_batch, _y_batch, val_rate = 0.2, printing=['
   loss', 'accuracy', 'val_loss', 'val_acc'], single_batch=False)

  # set new epoch
  net.new_epoch()


# get start index of data for evaluation
_starts_test = np.arange(0, x_test.shape[0], batch_size)
# if data willing to shuffled, shuffle it by shuffling index for
   evaluation
np.random.shuffle(_starts_test)

# run evaluation
print("\nEvaluate:")

for _start_t in _starts_test:
  # find last index of batch and iterate other parameters.
```

```
  _end_t = _start_t + batch_size
  _x_batch_t = x_test[_start_t:_end_t]
  _y_batch_t = y_test[_start_t:_end_t]
  net.evaluate_one_batch(_x_batch_t, _y_batch_t, single_batch=False)
```

Code 2.14: Train and evaluate batch with validation rate.

Second way is assigning validation data.

```
batch_size = 32
epoch = 10

for e in range(epoch):

  # get start index of data for each epoch.
  _starts = np.arange(0, x_train.shape[0], batch_size)
  # if data willing to shuffled, shuffle it by shuffling index for each
    epoch.
  np.random.shuffle(_starts)

  # run batchs
  print("\nEpoch : ", e + 1)
  for _start in _starts:
    # find last index of batch and iterate other parameters.
    _end = _start + batch_size
    _x_batch = x_train[_start:_end]
    _y_batch = y_train[_start:_end]
    # train model with validation data where valid_x is sample
   validation data and valid_y is labels of them.
    net.train_one_batch(_x_batch, _y_batch, val_x= valid_x, val_y=
   valid_y, printing=['loss', 'accuracy', 'val_loss', 'val_acc'],
   single_batch=False)

  # set new epoch
  net.new_epoch()


# get start index of data for evaluation
_starts_test = np.arange(0, x_test.shape[0], batch_size)
# if data willing to shuffled, shuffle it by shuffling index for
   evaluation
np.random.shuffle(_starts_test)

# run evaluation
print("\nEvaluate:")

for _start_t in _starts_test:
```

```
  # find last index of batch and iterate other parameters.
  _end_t = _start_t + batch_size
  _x_batch_t = x_test[_start_t:_end_t]
  _y_batch_t = y_test[_start_t:_end_t]
  net.evaluate_one_batch(_x_batch_t, _y_batch_t, single_batch=False)
```

Code 2.15: Train and evaluate batch with validation data.

More details can be found in Chapter 4.

## 2.14 How predict of model?

After training, prediction of unseen data can be run with ‘predict‘ method.

```
# prediction of x which is unseed data.
net.predict(x)
```

Code 2.16: Predict data on trained model.

## 2.15 How save model?

After training, save trainable parameters with name ’gNet_usage_save_model’.

```
# save model with name 'gNet_usage_save_model'.
net.save_model('gNet_usage_save_model')
```

Code 2.17: Save model with name’gNet_usage_save_model’.

## 2.16 How load model?

After training, saved trainable parameters with name ’gNet_usage_save_model’ can be load by ‘load_model‘ method.

```
# load model with name 'gNet_usage_save_model'.
net.load_model('gNet_usage_save_model')
```

Code 2.18: Load model with name’gNet_usage_save_model’.

## 2.17 How get loss and accuracy plots of traning?

After training, plots of loss and accuracy of training can be show and saved with custom names.

```
# show loss plot of training and saved as 'gNet_usage_loss.png'
net.get_loss_plot(show=True, save=True, figure_name='gNet_usage_loss.
    png')

# show accuracy plot of training and saved as 'gNet_usage_acc.png'
net.get_accuracy_plot(show=True, save=True, figure_name='gNet_usage_acc
    .png')
```

Code 2.19: Plots of training.

# Chapter 3: tensor, tensor_ops modules and Tensor class

Tensor is basically n-dimensional array. gNet uses numpy as linear algebra library. Thus, tensor is basically numpy's n-dimensional array which is ndarray. Yet, there is a reason for creating Tensor class instead of using ndarray directly. The reason is calculation of gradient. Gradient is derivative of anything.

In DL, gradient is need for backpropagation (BP). BP is finding gradients of trainable parameters such as weights and biases and updating trainable parameters with them. To calculate gradient, gNet uses Automatic Differentiation (AD) [3] like other libraries (Tensorflow, Pytorch etc.). Implementation of AD is taken by Joel Grus's autograd videos series and codes [1]. It is similar to Pytorch. If you familiar with Pytorch, you get used to it easily.

AD will not explained it details, but it is basically using chain rule. AD has two modes which are forward and reverse AD. gNet uses reverse mode AD. AD is based on operator overloading. Therefore, Tensor class is created. It has 4 instances which are value, have_grad, depends_on and grad. **value** is value of Tensor. **have_grad** is boolean value of Tensor which show it has gradient or not. If Tensor have_grad = True, it means that Tensor has gradient. **depends_on** is stores depending derivative functions of Tensor. It store derivative function of tensor operations. It will explained later. Lastly, **grad** is gradient of Tensor.

## 3.1 tensor module

In tensor module, there is Tensor class and tensor operations. Tensor operations' description written in tensor module. These operations implementation in

tensor_ops module. When **calling tensor operations, use them from tensor module**.

## 3.2 tensor_ops module

In tensor_ops module, a lot of basic operation implementation can be found. Each operation should have 4 variables which are value, have_grad, ops_name and depends_on. These variables are same as Tensor instanses because there operations return new calculated Tensor. Difference is ops_name and it is just a string of operation name for debugging BP in development stage.

To explain tensor operation, let's explain exp operation. exp operation is calculation of exponential of tensor.

```
def exp(t: 'Tensor') -> 'Tensor':
  '''
  Exponent calculation of tensor. Also it is calculate its gradient of
   operation
  if tensor have_grad = True.
  '''
  value = np.exp(t._value)
  have_grad = t.have_grad
  ops_name = '_exp'

  if have_grad:
    def grad_fn_exp(grad: np.ndarray) -> np.ndarray:
      grad = value * grad
      return grad

    depends_on = [T.Dependency(t, grad_fn_exp, ops_name)]
  else:
    depends_on = []

  return T.Tensor(value, have_grad, depends_on)
```

Code 3.1: exp operation.

In 3.1, 't' is input of opetaration which is also Tensor. value is calculation of Tensor. Be aware that it is not directly use np.exp(t), it is calculated by **t._value** (or t.value also works) . It means that Tensor object is not used directly in numpy, because it is not just an array.

have_grad is boolean of Tensor which shows whether Tensor ‘t‘ has differentiable (gradient can be calculated) or not. It is needed because if Tensor ‘t‘ is not differentiable, gradient will not be calculated and it will has zero gradients. If inputs of basic operation are more than one like multiplication which has ‘t1‘ and ‘t2‘, and one of the inputs have_grad, return Tensor has also have_grad.

ops_name is string of operation name which used during debug of BP. depends_on is another class of which stores dependent Tensor, gradient function and ops_name. depends_on is important because when calculation of gradient, depends_on called recursively until whole dependencies calculated.

There is two important things. Firstly, if Tensor have_grad, depends_on will be store, else it will be empty. Therefore, if operations have multiple Tensor inputs like multiplication which has ‘t1‘ and ‘t2‘, depends_on created for each Tensor which has gradient.

```
def mul(t1: 'Tensor', t2: 'Tensor') -> 'Tensor':
  '''
  Element wise multiplication of two `Tensor`. Also it is calculate its
  gradient of operation if tensor have_grad = True.
  '''
  value = t1._value * t2._value
  have_grad = t1.have_grad or t2.have_grad
  ops_name = '_mul'
  depends_on: List[Dependency] = []

  if t1.have_grad:
    def grad_fn_mul1(grad: np.ndarray) -> np.ndarray:

      grad = grad * t2._value

      # to handle broadcast, add dimension
      ndims_added = grad.ndim - t1._value.ndim
      for _ in range(ndims_added):
        grad = grad.sum(axis=0)

      for i, dim in enumerate(t1.shape):
        if dim == 1:
          grad = grad.sum(axis=i, keepdims=True)

      return grad

    depends_on.append(T.Dependency(t1, grad_fn_mul1, ops_name))
```

```
if t2.have_grad:
  def grad_fn_mul2(grad: np.ndarray) -> np.ndarray:

    grad = grad * t1._value

    ndims_added = grad.ndim - t2._value.ndim
    for _ in range(ndims_added):
      grad = grad.sum(axis=0)

    for i, dim in enumerate(t2.shape):
      if dim == 1:
        grad = grad.sum(axis=i, keepdims=True)

    return grad

  depends_on.append(T.Dependency(t2, grad_fn_mul2, ops_name))

return T.Tensor(value, have_grad, depends_on)
```

Code 3.2: mul operation.

Lastly, the most important properties of tensor operations is gradient functions of operation. To create proper gradient, shape of gradient should assign carefully and thinks reversely. It can be explain in this way. When take analytically gradient of some functions, gradient shape will be calculated in forward way. For example, in mean operation of 3x3 tensor (or can call matrix but it is also Tensor class), result will be 1x1 tensor. Yet, because of the reverse AD which starts from result to calculate gradient of function, return of the gradient function of mean operation should be 3x3 tensor. This is the reversely thinking. If operation is elementwise, it is just same as analytical gradient calculation. On the other hand, if the operation is non-elementwise, reversely thinking should take consider into calculation.

With these notes, user can use create custom tensor operation. Make sure that inputs of operations will be Tensor. To sure that, user can call ‘make_tensor‘ method in tensor module. The structure same for all operations in gNet.

Note that, Tensor class support most of magic methods.

## 3.3 How calculate gradient?

Gradient calculation has two rules. First rule is tensor which inputs of function should have_grad = True. Second rule is call backward methods from result. To explain it, give example. There is a function $f(x, y) = 3x^2 + 2y$, user want to find $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ where $x = 2, y = 5$.

Analytically, $\frac{\partial f}{\partial x} = 6x$, $\frac{\partial f(2,5)}{\partial x} = 12$ and $\frac{\partial f}{\partial y} = 2$, $\frac{\partial f(2,5)}{\partial y} = 2$.

```
import gNet.tensor as T

x = T.Tensor(2., have_grad=True)
y = T.Tensor(5., have_grad=True)

f = 3 * T.power(x, 2) + 2 * y  # also 3 * x**2 + 2 * y can be used.
f.backward() # calculate derivatives.

print(f, '\n', x.grad, '\n', y.grad)

-----------------------------------------------
Tensor(22.0,shape=(()), have\_grad=True)
Tensor(12.0,shape=(()), have\_grad=False)
Tensor(2.0,shape=(()), have\_grad=False)
```

Code 3.3: Calculation of gradient.

As seen in 3.3, variables x and y have have_grad = True, it means that if f.backward() called, $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ will be calculated. Their results will be as grad instance. If tensor x has have_grad = False, x.grad will be zero tensor. So, differentiable tensors (x and y in this example) should be assign carefully.

# Chapter 4: neuralnetwork module

neuralnetwork module is module of Neural Network (NN) structure. gNet can create supervised learning structure for now. Other structure like unsupervised or reinforcement learning not implemented. This module has one class which is NeuralNetwork. NeuralNetwork class has one input which is model. Model can be created from model module which will be explained.

NeuralNetwork class 15 methods where '___init___' not included. Some of these methods are hidden for end-user. Most of the methods for end-user. Let's explain each methods and some details about them.

**___init___** Initialization of NeuralNetwork. It assigning model from input, set layer from model and set optimizer and loss function caller. Caller is dictionary of related modules' class. It stores calling strings of classes. By using caller, developer can call classes with strings. 4.1 is example of how to use caller.

**_feedForward** This method is hidden (if _ is first letter of function name shows that it is hidden) for end-user. This method calculates each layers which will be explained in Chapter 7. It takes two inputs. First input is 'inp' which is input of first layer of model which is tensor. Second input is 'train' which is boolean and indicate whether '_feedForward' method called in trainin model or not. This boolean is needed because some of layers which are Dropout and Batch Normalization have different calculation during train. To make this difference, layer should know whether feed forward method is calling on training or not.

**_regularizer** This method is also hidden for end-user. It calculates regularizer of each layer. What is regularizer will be explain in related Chapter 12. How implemented them in layer will be explained in related chapter.

**__train__print** This method is also hidden for end-user. Method helps for printing different paramters during training. It is called by related methods. It used two flags which are boolean values for showing printing on some conditions or not. For example, 'TRAIN_ONE_BATCH_FLAG' shows whether the method is called from 'train_one_batch' method or not. This knowladge makes different printing.

Printing parameters can be listed as :

- Loss as 'loss',
- Epoch loss as 'epoch_loss',
- Accuracy as 'accuracy',
- Validation Loss as 'val_loss',
- Validation Accuracy as 'val_acc',
- ETA as 'ETA'.

There is a note that printing loss is average loss of training. This works in this way. Epoch loss is summation of loss of batch during epoch. Loss is calculated by $\frac{epoch\_loss}{passed\_number\_of\_batch}$. This calculates average loss of training which is similar in Keras and Tensorflow. Accuracy calculated by metric class which will be explained in Chapter 10.

**train** This method is heart of supervised learning. It trains the model. Input arguments will be found in function description or called help function. The method has two loop for epochs and batches. Some notes about this method is for each batches, before calculate gradient of required parameter, make zero grad of required parameters. If not maked zero grad, gradient will be wrong. The reason of them is structure of Tensor class. To make zero grad for trainable parameters, model has 'zero_grad' function. One of notes that when epoch changes, some parameters should be reset as grad. Accuracy is also one of them.

Also, another note that when creating convolutional NN, training data should have 'have_grad=True'. If model has only Multi-Layer Perceptron (MLP) structure, it is not needed. To handle it, there is a condition which checks whether first layer is 'Conv2D' or not. If remove the condition or 'have_grad' always True, speed of training will be decrease. To handle it, there is a inline condition.

**train_one_batch** This method does same job as train. Difference is how does the job. This method train only one batch of data. If user want to calculate just one batch, user should call the method. Also, there is reason for adding the method. If there is memory problem (using lots of data or bigger data) for training, user can use the method.

Important point of the method is 'single_batch' boolean variable. If model is single_batch, it means that only calculate one batch parameters. If model is not single_batch model calculate parameters upto that time. Example of difference between single_batch=True and single_batch=False is loss. When single_batch=True, loss will be equal to that batch's loss. Yet, when single_batch=False, loss will be equal to average loss upto that time.

'train' and 'train_one_batch' methods can calculate **validation**. There is two way to do it. First way is assign validation rate parameters 'val_rate' between 0 to 1. It split data into validation and traning seperately. Second way is assign 'val_x' and 'val_y' data directly. Important point for second way is validation data should not in train data also.

Another notes about validation is when running 'train_one_batch' with 'val_rate', it will print different result w.r.t running train with 'val_rate'. Reason is when use 'val_rate' for 'train_one_batch', it will split validation data from batch. On the other hand, when use 'val_rate' for train, it will split validation data from whole data. This makes the difference for validation results. Yet, if user input 'val_x' and 'val_y' into train and 'train_one_batch', it will give same results because data is same for two methods. Also 'val_rate' will not used with 'val_x' and 'val_y' at the same time.

If there is memory problem, instead of using 'train', using 'train_one_batch' is suggested.

**new_epoch** When 'train_one_batch' methods is used for training for more than one epoch, method should be called to reset some parameters and calculate validation if there is validation case. Important point of method is usage. Call the method at the end of loop of epoch. Example can be found in 2.13

**setup** Setup is one of base function of NeuralNetwork class. It is assign loss function of model and optimizer of model. Built-in functions or methods can be called by string of them. To handle this, 4.1 is used. This approach is used a lot in gNet. 4.1 is one of them which assign optimizer. Firstly, it checks whether input is string or not, if string call it by optimizer caller. Caller is dictionary and created in some modules and it stores strings of related class. For example, in optimizer module there is hidden dictionary called '___optimizerDecleration'. This dictionary store strings for related class such as ''adam' : Adam'. When user input 'optimizer='adam'' in setup function, it calls from built-in adam optimizer from optimizer module with default parameters. Usage and creation of custom optimizer class explained in Chapter 11.

```
if isinstance(optimizer, str):
  _opt = optimizer.lower()
  self._optimizer = self._optimizerCaller[_opt]()
else:
  self._optimizer = optimizer
```

Code 4.1: Assigning optimizer.

**predict** Predict is calling '_feedForward' for its input. This method created for end-user. After training, user can predict their unseen example by 'predict' method.

**evaluate** Evaluation of model. After training, user can call evaluate method to test the model with unseen test data. To handle it, evaluate method created.

**evaluate_one_batch** Does same job as evaluate. Difference is evaluating only one batch. One batch approach same as 'train_one_batch' method. If model is single_batch, it means that only evaluate one batch parameters. If model is not single_batch model evaluate parameters upto that time. Example of difference between single_batch=True and single_batch=False is loss. When single_batch=True, loss will be equal to that batch's loss. Yet, when single_batch=False, loss will be equal to average loss upto that time.

**save_model** This method save trainable variables with numpy save method w.r.t 'file_name' input. Generally, this methods creates list to save layer trainable variables. For now, only Batch Normalization layer has exception. This layer has two another parameters which are running mean and running variance should save to. To handle it, there is a condition and it is expanding list to save these parameters to. To understand which layer is Batch Normalization layer, 'layer_name' parameters

stores all layer names in model class. Therefore, gNet know that which class is Batch Normalization.

**load_model** This method is load trainable variables with numpy load method w.r.t 'file_name' input. It is opposite to 'save_model' method. It is also take care of Batch Normalization layer to handle additional stored parameters.

**get_loss_plot and get_accuracy_plot** These methods plotting loss and accuracy w.r.t iterations. Matplotlibs library used for them. Usage can be found in their descriptions.

**get_model_summary** This method write model summary. Texttable library used for it. If developer want to adjust it, be careful for row addition. First row should be list of one variable which is list of column names. Usage can be found in its description.

# Chapter 5: model module

model module has one class which called as Model. Model class has ‘_params‘ variable which is dictionary and stores some parameters about model of NN. To understand what is done in this class, let’s look at methods.

**___init___** Initialization of Model class. It creates ‘_params‘ dict. ‘_params‘ has 7 values. These values are :

- ‘layer_number‘ : number of layer. It increases when adding layer to model.
- ’layer_name’ : list of name of layer. It stores layer names.
- ’activation’ : lisf of activation function of layer.
- ’model_neuron’ : list of neuron number of layer. This number changes depends on layer.
- ’layers’ : list of layers. It stores layers’ adress (which is handled by python). By this list, each layer of model can be called.
- ’layer_output_shape’ : list of output shape of layers. It needed for some layers such as ‘Conv2D‘ which use to calculate its output shape. Layer output shape has not contain batch size.
- ’#parameter’ : list of number of parameters of layers. It is used in model summary.

**add** ‘add‘ method used for add layer to model. It call layer’s ‘___call___‘ functions, add layer to ’layers‘ list and increase ‘layer_number‘.

**get_layers and get_params** These methods returns layers of model and params of model.

**zero_grad** It makes zero calculated in each layer in ‘layers‘ parameter.

# Chapter 6: initializer module

initializer module contains built-in initializer classes. These classes based on base class which called ‘Initializer‘. Built-in initializer classes listed with calling strings as :

- Ones init (’ones_init’)
- Zeros init (’zeros_init’)
- He’s normal (’he_normal’)
- He’s uniform (’he_uniform’)
- Normal init (’normal_init’)
- Uniform init (’uniform_init’)
- Xavier’s normal (’xavier_normal’)
- Xavier’s uniform (’xavier_uniform’).

Calling strings used by callers. Initializers have also caller in layers. If user want to use built-in initializer, just set proper input argument with calling strings. For example, default initialize_method of Dense layer is ’xavier_uniform’.

If user want to use built-in initializer with different parameters, user needs to create built-in class with custom parameters then input as the created class. 6.1 show initializer a Dense layer with normal initializer method with mean as 0.2 and standart deviation as 0.8 . In defaults, mean equal 0.0 and standart deviation equal 1.0.

```
from gNet import initializer
...
init2 = initializer.Normal_init(mean=0.2, stdDev=0.8)
net = NeuralNetwork(...)
...
net.add(Dense(100, initialize_method=init2))
...
```

Code 6.1: Built-in initializer with custom parameters.

initializer module has declaretor dictionary (called ‘caller‘ in other modules) which is ‘___initializeDeclaretion‘. It stores built-in class adresses with calling string. For example, He_normal stored as ‘’he_normal’: He_normal‘.

## 6.1 Creating Custom Initializer Class

gNet supports creating custom initializer class. First of all, **class should inherited from Initializer class which is base class, and should have ‘get_init‘ method**. Without these, class can not be called by gNet. ‘get_init‘ method should have ‘shape‘ argument as input argument. ‘get_init‘ method returns to initialized array (not tensor) w.r.t shape inputs. Also, base class have ‘_get_fans‘ method for calculate fan paremeters which are ‘fan_in‘ and ‘fan_out‘.

‘fan_in‘ and ‘fan_out‘ parameters different w.r.t dimension of ‘shape‘ parameter. If length of shape is 2, it means that shape is 2D, ‘fan_in‘ will be ‘shape[0]‘ and ‘fan_out‘ will be ‘shape[1]‘. If length of shape is not 2, ‘fan_in‘ will be ‘production of shape[1:]‘ and ‘fan_out‘ will be ‘shape[0]‘. The reason is that kind of implementation is structure of im2col algorithms which use channel first approach and details will be in Chapter 7. **‘_get_fans‘ methods is not must method to use**. Therefore, if user create custom class, user can use also custom class’ methods.

To give an example of custom initializer class, let’s create it.

```
class myInitializer(initializer.Initializer):

  def __init__(self, scale=0.05, **kwargs):
    self._scale = scale

  def get_init(self, shape=None, **kwargs) -> np.ndarray:
```

```
    return np.random.uniform(-1.0, 1.0, size=shape) * self._scale
```

Code 6.2: Custom initializer class.

In 6.2, custom initalizer class created which is uniform distribution from -1 to 1 with scale factor.

To call ‘myInitializer‘ by layers, there is two way to do it. First way is put class into input arguments directly. Second way is adding into ‘__initializeDeclaretion‘, then called by calling string. 6.3 show two way of calling custom initializer class.

```
# FIRST WAY
...
net = NeuralNetwork(...)
...
net.add(Dense(100, 'relu', initialize_method=myInitializer))
...

# SECOND WAY

...
initializer.__initializeDeclaretion['myinit'] =  myInitializer
...
net = NeuralNetwork(...)
...
net.add(Dense(100, 'relu', initialize_method=myinit))
...
```

Code 6.3: Calling custom initializer class.

**Important note for way two is calling string of custom class should be all lower case letters.**

# Chapter 7: layer module

layer module contains built-in layer classes. These classes based on base class which called ‘Layer‘. Built-in layer classes listed as :

- Dense
- Flatten
- Activation
- Conv2D
- MaxPool2D
- AveragePool2D
- Dropout
- Batch Normalization

Layer classes should be used by model‘s ‘add‘ method. User also can create custom layer w.r.t some rules.

## 7.1 Creating Custom Layer Class

gNet supports creating custom layer class. First of all, **class should inherited from Layer class which is base class, and should have some methods**. Without these, class can not be called by gNet. There is several methods for user should implement to create custom layer class. These are ‘___call___‘, ‘_init_trainable‘, ‘compute‘ and ‘regularize‘ methods.

**___call___** This method is called when layer added to model. It is hidden Python's method and called when class is created. It generally computes model parameters. Updated parameters of model can be changed by layer. Thus, be carefull for updated parameters.

**_init_trainable** This method is called at the end of '___call___' method. It initialize proper parameters such as trainables. Even layer has no trainable parameter, class should has _init_trainable method and just 'pass'. Initialization parameters depends on layer. Thus, implementation should be carefully selected and method should be called in '___call___' method.

**compute** This method is called by '_feedForward' method. compute method is base of computation of layer. This is the core of computation. Implementation should be carefully done. Without compute method, layer cannot be called by NN structure. Return should be also tensor and input arguments are 'inputs' and 'train'. 'inputs' is input of layer, and 'train' is boolean variable which shows whether the compute is called during training or not.

**regularize** This method is called by '_regularizer' method. regularize method is base of computation of regularization of layer. Each layer can have different regularization with this implementation. If regularization is not need in that layer like dropout or flatten, return zero. Implementation should be carefully done. Without regularize method, layer cannot be called by NN structure. Return should be also tensor and it has no input arguments because all regularization done by trainable parameters of layer.

Base Layer class has also other methods. If layer has only weights and bias as trainable variables, '_set_initalizer' and '_get_inits' methods are usefull. Set initializer methods for layer parameters. If one of the parameters of layer have initializer, this function should be called at the end of layer's '___init___' method. This method also have 'bias initializer' which can initialize bias separately. '_get_inits' method called after initializer setted. Method have 2 argumest which pass shape of parameters. Method returns initialized W and B respectively.

Also, base Layer class has 'zero_grad' method which zeroing trainable parameters' grad values. For trainable parameters, base Layer class has getter and setter which used in optimization module a lot.

**___init___** Init method of base Layer class has two variables which are activation function caller '_actFuncCaller' and trainable variables list '_trainable'. Activation function caller used when activation functions need to call. Yet, trainable list is used a lot. Trainable parameters of layer should be append to '_trainable' list because these parameters will be updated during optimization step. Therefore; base layer have getter and setter for trainable parameter to get and set it easily. To understand general structure, Dense layer implementation will be in 7.1.

```
class Dense(Layer):

  def __init__(self,
        neuron_number = None,
        activation_function = None,
        initialize_method = 'xavier_uniform',
        bias_initializer = 'zeros_init',
        kernel_regularizer = None,
        bias_regularizer = None,
        **kwargs):
    super(Dense, self).__init__(**kwargs)
    if activation_function == None:
      activation_function = 'none'
    self._activation = activation_function
    self._neuronNumber = neuron_number
    self._initialize_method = initialize_method
    self._bias_initialize_method = bias_initializer
    self._set_initializer()
    self._kernel_regularizer = kernel_regularizer
    self._bias_regularizer = bias_regularizer

  def __call__(self, params) -> None:
    self._thisLayer = params['layer_number']
    params['layer_name'].append('Dense : ' + self._activation)
    params['activation'].append(self._activation)
    params['model_neuron'].append(self._neuronNumber)
    params['layer_output_shape'].append(self._neuronNumber)
    self._init_trainable(params)

  def _init_trainable(self, params):
    if self._thisLayer == 0:
      row = 1
      col = 1
    else:
      row = params['model_neuron'][self._thisLayer-1]
      col = params['model_neuron'][self._thisLayer]
```

```
    _w_shape = (row, col)
    _b_shape = [col]

    params['#parameters'].append(row*col+col)

    # get initialized values of weight and biases
    _w, _b = self._get_inits(_w_shape, _b_shape)
    # append weight and biases into trainable as `tensor`.
    self._trainable.append(T.Tensor(_w, have_grad=True))
    self._trainable.append(T.Tensor(_b, have_grad=True))

  def compute(self, inputs: T.Tensor, train: bool, **kwargs) -> T.
   Tensor:
    _z_layer = inputs @ self._trainable[0] + self._trainable[1]
    return self._actFuncCaller[self._activation].activate(_z_layer)

  def regularize(self) -> T.Tensor:
    _res = T.Tensor(0.)
    if self._kernel_regularizer:
      _res += self._kernel_regularizer.compute(self._trainable[0])
    if self._bias_regularizer:
      _res += self._bias_regularizer.compute(self._trainable[1])
    return _res
```

Code 7.1: Dense Layer implementation.

## 7.2 Dense Layer

Dense layer one of the basic and important layer of DL. It makes MLP structure. It has two trainable parameters which are weights and biases. These parameters change during training.

Dense layer's input should be 2D and first dimension is batch size. If previous layer output is not 2D, flatten layer should be used. Dense layer can not used as first layer of model. Even if data is 1D, flatten layer should be added because Dense layer needed previous layer output shape.

Dense layer's append model's some parameters as :

- 'layer_name' as 'Dense : 'activation function'' of layer,

- ‘activation‘ as ‘activation function‘ of layer,
- ‘model_neuron‘ as ‘neuron_number‘ of layer,
- ‘layer_output_shape‘ as ‘neuron_number‘ of layer,
- ‘#parameters‘ as $\#w_{row} * \#w_{col} + \#w_{col}$ where w is weight of layer.

Dense layer’s ‘model_neuron‘ and ‘layer_output_shape‘ are same because output of Dense layer is 1D (without batch dimension). Yet, in some other layers, these parameters are different.

## 7.3 Flatten Layer

Flatten layer flat input data to make it 1D. Flatten layer calls flatten operation. Flatten operation implemented as tensor operations because gradient implementation need to be done. For example, when created model with convolution layer, flatten layer should be added to model to add dense layer. To calculate convolution layer’s trainable parameters gradient, flatten operations gradient should be calculated as well because of reverse AD structure.

Flatten operation has second input arguments which is batching. If flatten operation is called in training with batch, flatten operations should be done for each batch’s data. To handle it, there is a input argument. If ‘batching=False‘ flatten operation flat all data as 1D. If ‘batching=True‘, flatten operation flat data with corresponding batch size; therefore, flatten data will be 2D. In layer, ‘batching=True‘.

Flatten layer’s append model’s some parameters as :

- ‘layer_name’ as ’flatten’,
- ‘activation‘ as ’none’,
- ‘model_neuron‘ as output dimension of layer (without batch dimension),
- ‘layer_output_shape‘ as output dimension of layer (without batch dimension),
- ‘#parameters‘ as ’0’.

## 7.4 Conv2D Layer

Conv2D layer is convolution of 2D data. Implementation done from Stanford lecture notes [2]. Im2col approach is used in gNet. Therefore; channel first is used in gNet. It is different than Tensorflow. Its compute method calculation based on flatten local space of input and kernels then stored as 2D array. After making 2D array, by using dot product, calculation of all convolution can be done. Then, reshaping result to proper size.

Conv2D layer's append model's some parameters as :

- 'layer_name' as 'Conv2D',
- 'activation' as 'none',
- 'model_neuron' as '#filter',
- 'layer_output_shape' as '(filter, H_out, W_out)',
- '#parameters' as '#filter * H_kernel * W_kernel + #filter'.

## 7.5 Activation Layer

Activate the previous layer output. It just use activation function to 'inputs' of compute method. At the activation layer, there is no regularization like flatten layer.

Activation layer's append model's some parameters as :

- 'layer_name' as 'Activation Layer : 'activation of layer'',
- 'activation' as 'activation of layer',
- 'model_neuron' as 'previous layer neuron number',
- 'layer_output_shape' as 'previous layer output shape',
- '#parameters' as '0'.

## 7.6 MaxPool2D Layer

MaxPool2D layer is maximum pooling of 2D data. Implementation done from Stanford lecture notes [2]. Its compute method calculation based on flatten local space of input's max values and store values and their indexes. Creating output as proper shaped tensor. There is no regularization like flatten layer.

MaxPool2D layer's append model's some parameters as :

- 'layer_name' as 'MaxPool2D',
- 'activation' as 'none',
- 'model_neuron' as '0',
- 'layer_output_shape' as '(C, H_out, W_out)',
- '#parameters' as '0'.

## 7.7 AveragePool2D Layer

AveragePool2D layer is average pooling of 2D data. Implementation done from Stanford lecture notes [2]. Its compute method calculation based on flatten local space of input's average values. Creating output as proper shaped tensor. There is no regularization like flatten layer.

AveragePool2D layer's append model's some parameters as :

- 'layer_name' as 'AveragePool2D',
- 'activation' as 'none',
- 'model_neuron' as '0',
- 'layer_output_shape' as '(C, H_out, W_out)',
- '#parameters' as '0'.

## 7.8 Dropout Layer

Dropout is one of regularization mechanism. It killing/deactivate neuron temporary for reduce of possibility of overfitting. Implementation done from Stanford lecture notes [2]. One of the reason of ‘compute‘ method has an input argument which called train is dropout layer becuase it acts different between training and evaluating.

During training, dropout layer kill some neurons (it means that multiply with 0 at that moment). On the other hand, during evaluation, layer do not kill some neurons. Therefore, calling ‘_feedForward‘ method is different during training. To make this difference, there is a boolean value which is called ‘train‘. There is no regularization like flatten layer.

Dropout layer’s append model’s some parameters as :

- ‘layer_name’ as ’Dropout : ‘dropout probability of layer‘’,
- ‘activation‘ as ’none’,
- ‘model_neuron‘ as ’previous layer neuron number’,
- ‘layer_output_shape‘ as ’previous layer neuron output shape’,
- ‘#parameters‘ as ’0’.

## 7.9 Batch Normalization Layer

Batch Normalization (BN) Layer has some difference from other layers. First of all it has two trainable values; yet, it has four saved values. Reason is difference calculation of BN during training and evaluating is not same like dropout layer.

During training, running mean and variance parameters are not used, they are just updated. Yet, during evaluation, running mean and variance parameters are used. Therefore, when saving trainable parameters, running mean and variance are also saved. This is why, ‘save_model‘ and ‘load_model‘ have conditions for BN.

Two trainable parameters and two running parameters have each initializer method. Thus, in ‘_init_trainable‘ method has four initializer caller. Also, previous layer is effecting these four parameters dimension. To handle it, there is a condition.

Batch Normalization layer’s append model’s some parameters as :

- ‘layer_name’ as ’Batch Normalization’,
- ‘activation‘ as ’none’,
- ‘model_neuron‘ as ’previous layer neuron number’,
- ‘layer_output_shape‘ as ’previous layer neuron output shape’,
- ‘#parameters‘ as depends on which parameter used.

# Chapter 8: loss_functions module

loss_functions module contains built-in loss function classes. These classes based on base class which called ‘Loss‘. Built-in loss functions classes listed with calling strings as :

- Categorical Cross Entropy (’categoricalcrossentropy’, ’cce’)
- Binary Cross Entropy (’binarycrossentropy’, ’bce’)
- Mean Square Error (’meansquareerror’, ’mse’)

Calling strings used by callers. If user want to use built-in loss functions, just set proper input argument with calling strings in setup method.

If user want to use built-in loss functions with different parameters, user needs to create built-in class with custom parameters then input as the created class. 8.1 show loss functions with logits. From logits means that before calculate loss, model need to normalize output w.r.t proper method such as softmax. If last layer is not softmax, from logits make it softmax. Even if last layer is not softmax and ‘from_logits=False‘, output will be normalized without softmax. From logits is depends on loss function. CCE has softmax, BCE has sigmoid as logits.

```
from gNet import loss_functions
...
loss = loss_functions.CategoricalCrossEntropy(from_logits=True)
net = NeuralNetwork(...)
...
net.setup(loss_function=loss, optimizer='adam')
...
```

Code 8.1: Built-in loss function with custom parameters.

loss_functions module has declaretor dictionary (called ‘caller‘ in other modules) which is ‘__lossFunctionsDecleration‘. It stores built-in class adresses with calling string. For example, Categorical Cross Entropy stored as ‘’categoricalcrossentropy’ : CategoricalCrossEntropy‘ and ‘’cce’ : CategoricalCrossEntropy‘.

## 8.1 Creating Custom Loss Class

gNet supports creating custom loss functions class. First of all, **class should inherited from Loss class which is base class, and should have ‘loss‘ and ‘get_metric‘ methods**. Without these, class can not be called by gNet.

Calculation of loss function implemented in ‘loss‘ method. ‘loss‘ method should have ‘y_true‘, ‘y_pred‘ and ‘model_params‘ as input arguments. ‘y_true‘ is true value of label (which is input as y in training). ‘y_pred‘ is prediction of model and it is calculated from ‘_feedForward‘ method. ‘model_params‘ is model parameters which can be get by model’s ‘get_params‘ method. ‘loss‘ method should be return one value tensor. One value means that shape of tensor should be 1x1.

‘get_metric‘ method has no input argument. It should returned proper metric calculate from ‘metric‘ module. It is needed for accuracy calculation and explain in Chapter 10.

To give an example of custom loss function class, let’s create mean square error.

```
import gNet.metric as mt

class myMSE(loss_functions.Loss):

  def loss(self, y_true, y_pred, model_params):
    y_true = T.make_tensor(y_true)
    y_pred = T.make_tensor(y_pred)
    error = (y_pred - y_true) ** 2
    return T.mean(T.tensor_sum(error, axis=-1), axis=-1)

  def get_metric(self) -> mt.Metric:
    return mt.CategoricalAccuracy()
```

Code 8.2: Custom loss function class.

In 8.2, custom loss function class created which is mean square error.

To call ‘myMSE‘ by gNet, there is two way to do it. First way is put class into input arguments directly. Second way is adding into ‘__lossFunctionsDecleration‘, then called by calling string. 8.3 show two way of calling custom initializer class.

```
# FIRST WAY
...
net = NeuralNetwork(...)
...
net.setup(loss_function=myMSE, optimizer='adam')
...

# SECOND WAY

...
loss_functions.__lossFunctionsDecleration['mymse'] =  myMSE
...
net = NeuralNetwork(...)
...
net.setup(loss_function='mymse', optimizer='adam')
...
```

Code 8.3: Calling custom loss function class.

**Important note for way two is calling string of custom class should be all lower case letters.**

# Chapter 9: activation_functions module

activation_functions module contains built-in activation function classes. These classes based on base class which called ‘ActivationFunction‘. Built-in activation functions classes listed with calling strings as :

- Rectified Linear Unit Function, Relu (’relu’)
- Leaky Rectified Linear Unit Function, LRelu (’lrelu’)
- Sigmoid, (’sigmoid’)
- Softmax, (’softmax’)
- Softplus (’softplus’)
- Tanh, (’tanh’)

Calling strings used by callers. If user want to use built-in activation functions, just set proper input argument with calling strings in layer such as Dense layer.

```
...
net = NeuralNetwork(...)
...
net.add(Dense(100, activation_function='relu')
...
```

Code 9.1: Built-in activation function.

activation_functions module has declaretor dictionary (called ‘caller‘ in other modules) which is ‘___activationFunctionsDecleration‘. It stores built-in class adresses with calling string. For example, ReLU stored as ‘’relu’ : Relu‘.

## 9.1 Creating Custom ActivationFunction Class

gNet supports creating custom activation functions class. First of all, **class should inherited from ActivationFunction class which is base class, and should have *static* 'activate' methods**. Without these, class can not be called by gNet.

An important note about custom activation function is **static** 'activate' method. Reason behind adding '@staticmethod' is activation function class is not created in gNet. To make it more structural, activation function class is created; yet, 'activate' method can be called directly.

Calculation of activation function implemented in 'activate' method. 'activate' method should have 'x' as input argument. 'x' is tensor and return of 'activate' method is also tensor.

To give an example of custom loss function class, let's create ReLU and Sigmoid activation functions.

```
class myRelu(ActivationFunction):

  @staticmethod
  def activate(x):
    x = T.where(x, x.value > 0, x, 0)
    return x

class mySigmoid(ActivationFunction):

  @staticmethod
  def activate(x):
    return 1.0 / (1.0 + T.exp(-x))
```

Code 9.2: Custom activation function classes.

To call 'myRelu' or 'mySigmoid' by layers, there is two way to do it. First way is put class into input arguments directly. Second way is adding into '___activationFunctionsDecleration', then called by calling string. 9.3 show two way of calling custom initializer class.

```
from gNet import activation_functions
# FIRST WAY
...
net = NeuralNetwork(...)
```

```
...
net.add(Dense(100, activation_function= myRelu))
...

# SECOND WAY

...
activation_functions.__activationFunctionsDecleration['mysigmoid'] =
    mySigmoid
...
net = NeuralNetwork(...)
...
net.add(Dense(100, activation_function= 'mysigmoid'))
...
```

Code 9.3: Calling custom activation function class.

**Important note for way two is calling string of custom class should be all lower case letters.**

# Chapter 10: metric module

metric module contains built-in metric classes. These classes based on base class which called ‘Metric‘. Built-in metric classes listed as :

- Categorical Accuracy
- Binary Accuracy

Metric class is not called with calling strings. Metric class created in loss function class’s ‘get_metric‘ method in gNet. Therefore; calling strings are not needed.

## 10.1 Creating Custom Metric Class

gNet supports creating custom metric class. First of all, **class should inherited from Metric class which is base class, and should have ‘accuracy‘ method**. Without this, class can not be called by gNet.

Calculation of accuracy implemented in ‘accuracy‘ method. ‘accuracy‘ method should have ‘y_true‘ and ‘y_pred‘ as input arguments. These are true and predicted label respectively.

An important note about custom metric is ‘_count‘ and ‘_total‘ variables. These variables are integers to calculate accuracy and they are created in base Metric class. When ‘y_true‘ and ‘y_pred‘ are in intended condition, ‘_count‘ should be increased by number of intended condition and ‘_total‘ is increased by number of batch. Also, base Metric class has reset function which make ‘_count‘ and ‘_total‘ to 0. This method called when new epoch started.

To give an example of custom metric class, let's create binary accuracy which has threshold.

```
class myBinaryAccuracy(Metric):

  def __init__(self, threshold=0.5) -> None:
    self._threshold = threshold

  def accuracy(self, y_true, y_pred) -> float:
    # set values which are bigger than threshold is 1.
    argmax_pred = np.where(y_pred.value > self._threshold, 1., 0.)
    # find maximum values indexes
    argmax_true = np.argmax(y_true.value, axis=-1).reshape(-1,1)
    argmax_pred = np.argmax(argmax_pred, axis=-1).reshape(-1,1)
    # check whether max indexes are equal.
    # if equal add to count
    self._count += np.equal(argmax_true, argmax_pred).sum()
    # add how many item does validate
    self._total += argmax_pred.shape[0]
    return self._count / self._total
```

Code 10.1: Custom metric classes.

To call 'myBinaryAccuracy' custom loss function class should be created or somehow built-in loss function class's 'get_metric' method changed. Let's changed our 'myMSE' class's metric with 'myBinaryAccuracy' which has threshold 0.7.

```
class myMSE(loss_functions.Loss):
  ...
  def get_metric(self) -> mt.Metric:
    return myBinaryAccuracy(threshold=0.7)
  ...
```

Code 10.2: Calling custom metric class.

# Chapter 11: optimizer module

optimizer module contains built-in optimizer classes. These classes based on base class which called ‘Optimizer‘. Built-in optimizer classes listed with calling strings as :

- SGD (’sgd’)
- Adagrad (’adagrad’)
- RMSprop (’rmsprop’)
- AdaDelta (’adadelta’)
- Adam (’adam’)

Calling strings used by callers. If user want to use built-in optimizer, just set proper input argument with calling strings in setup method.

If user want to use built-in optimizer with different parameters, user needs to create built-in class with custom parameters then input as the created class. 11.1 show ‘Adam‘ optimizer with custom learning rate where default learning rate equal 0.001.

```
from gNet import optimizer
...
opt = optimizer.Adam(lr=0.0001)
net = NeuralNetwork(...)
...
net.setup(loss_function='cce', optimizer=opt)
...
```

Code 11.1: Built-in optimizer with custom parameters.

optimizer module has declaretor dictionary (called ‘caller‘ in other modules) which is ‘___optimizerDecleration‘. It stores built-in class adresses with calling string. For example, ‘Adam‘ optimizer stored as ‘’adam’: Adam‘.

## 11.1 Creating Custom Optimizer Class

gNet supports creating custom optimizer class. First of all, **class should inherited from Optimizer class which is base class, and should have ‘step‘ method**. Without these, class can not be called by gNet.

Calculation of optimization step is implemented in ‘step‘ method. ‘step‘ method should have ‘layers‘ as input argument. ‘layers‘ is layers of model which can be get by model’s ‘get_layers‘ method. ‘step‘ method will not be return because it is optimizing trainable values of layers.

There is two point of implementing ‘step‘ method. Firstly, each layer has own trainable parameters. Thus; update these parameters seperately. To do that there is two for loops. First loop on layers and second loop on trainable parameters of looped layer. Lastly, if there is a variable which is not trainable and updated during training, it should be a list of arrays for each layer. To initialize this parameter, ‘step‘ method should have ‘init‘ condition.

To give an example of custom Adam optimizer class.

```
from gNet import optimizer

class myAdam(optimizer.Optimizer)

    def __init__(self,) -> None:
    self.lr = 0.0001
    self.beta1 = 0.9
    self.beta2 = 0.999
    self.eps = 1e-7 # small values to get rid of division zero error.
    self.t = 1
    self.init = True
    self.m = []
    self.v = []
    self.mhat = []
    self.vhat = []

  def step(self, layers)->None:
```

```
    # for first time call the step function, initialize parameter w.r.t
     layer size and trainable variable size.
    if self.init:
      for layer in layers:
      self.m.append(np.zeros_like(layer.trainable))
      self.v.append(np.zeros_like(layer.trainable))
      self.mhat.append(np.zeros_like(layer.trainable))
      self.vhat.append(np.zeros_like(layer.trainable))
      self.init = False

    # loop for layers
    for ind, layer in enumerate(layers):
      # loop for trainables
      for ind_tra, trainable in enumerate(layer.trainable):
        # update momentum
        self.m[ind][ind_tra] = self.beta1 * self.m[ind][ind_tra] + (1 -
     self.beta1) * trainable.grad.value
        # update velocity
        self.v[ind][ind_tra] = self.beta2 * self.v[ind][ind_tra] + (1 -
     self.beta2) * (trainable.grad.value ** 2)
        # update momentum hat
        self.mhat[ind][ind_tra] = self.m[ind][ind_tra] / (1 - self.
    beta1 ** self.t)
        # update velocity hat
        self.vhat[ind][ind_tra] = self.v[ind][ind_tra] / (1 - self.
    beta2 ** self.t)
        # update weights and biases
        trainable.value -= self.lr * self.mhat[ind][ind_tra] / (np.sqrt
    (self.vhat[ind][ind_tra]) + self.eps)

     self.t += 1
```

Code 11.2: Custom optimizer class.

In 11.2, custom Adam optimizer class. There is 'self.init' condition to initialize some optimization parameters. Reason behind the structure is dynamic structure of gNet. Layer can be changed dynamically, thus; in '__init__' method, initialize these optimization parameters will not be fit with the layers. To handle it there is a boolean value called 'self.init'. It setted True in '__init__' method, then when called first step function, it will assign as False. With this approach, each layers' trainable parameters has own optimization parameters.

To call 'myAdam' by gNet, there is two way to do it. First way is put class into input arguments directly. Second way is adding into '__optimizerDecleration', then called by calling string. 11.3 show two way of calling custom optimizer class.

```
# FIRST WAY
...
net = NeuralNetwork(...)
...
net.setup(loss_function='cce', optimizer=myAdam)

# SECOND WAY

...
optimizer.__optimizerDecleration['myadam'] =  myAdam
...
net = NeuralNetwork(...)
...
net.setup(loss_function='cce', optimizer='myadam')
...
```

Code 11.3: Calling custom optimizer class.

**Important note for way two is calling string of custom class should be all lower case letters.**

# Chapter 12: regularizer module

regularizer module contains built-in regularizer classes. These classes based on base class which called ‘Regularizer‘. Built-in regularizer classes as :

- L1
- L2
- L1L2 (Containing two of them at the same time)

If user want to use built-in regularizer, just set proper input argument with calling strings in proper layer.

If user want to use built-in regularizer with different parameters, user needs to create built-in class with custom parameters then input as the created class. 12.1 show ‘L2‘ regularizer with custom $\lambda$ value.

```
from gNet import regularizer
...
regu2 = regularizer.L2(Lmb=0.001)
net = NeuralNetwork(...)
...
net.Dense(100, 'relu', weight_regularizer=regu2)
...

# or
...
net = NeuralNetwork(...)
...
net.Dense(100, 'relu', weight_regularizer=regularizer.L2(Lmb=0.001))
...
```

Code 12.1: Built-in L2 regularizer with custom parameters.

optimizer module has not declaretor dictionary.

## 12.1 Creating Custom Regularizer Class

gNet supports creating custom regularizer class. First of all, **class should inherited from Regularizer class which is base class, and should have 'compute' method like layer**. Without these, class can not be called by gNet.

Calculation of regularization is implemented in 'compute' method. 'compute' method should have 'parameters' as input argument. 'parameters' is tensor. Thus; regularization can be calculated for any tensor of gNet.

To give an example of custom L2 regularizer class.

```
class myL2(Regularizer):

  def __init__(self):
    self._lmb = 0.001

  def compute(self, parameter: 'Tensor') -> 'Tensor':
    self._regularizer = self._lmb * T.power(T.make_tensor(parameter),
   2).sum()
    return self._regularizer
```

Code 12.2: Custom regularizer class.

In 12.2, custom L2 regularizer class which $\lambda$ value is 0.001.

To call 'myL2' by gNet, there is one way to do it. This way is put class into input arguments directly like 12.1.

Also note that gNet has layerwise regularizer. It means that every layer has its own regularization if user want it. Therefore; there can be different combination of regularization.

# Chapter 13: conv_utils module and utils module

These modules are used some utility function of gNet. conv_utils has utility functions for convolution operations.

utils has ‘make_one_hot‘ function which makes sparse label values into one_hot vector label. Also, utils has ‘MNIST_Downloader‘ which helps to download and load MNIST data [4].

# Bibliography

[1] J. Grus. Autograd, 2019. URL `https://github.com/joelgrus/autograd`.

[2] F.-F. Li. Cs231n: Convolutional neural networks for visual recognition lecture notes, 2020. URL `https://cs231n.github.io`.

[3] A. Savine. *Modern Computational Finance: AAD and Parallel Simulations*. Wiley Publishing, 1st edition, 2018. ISBN 1119539455.

[4] L. Yann, B. Leon, B. Yoshua, and H. Patrick. The mnist dataset, 1998. URL `http://yann.lecun.com/exdb/mnist/`.