PyNet API Documentations

Xiao Zhang, Haoyuan Zhang

November 28, 2017

1 Installation

Please go inside the PyNet folder directory and run the command in the terminal (either Windows or Linux) to compile the C code.

python setup.py build_ext - -inplace

Also, you need to install *cython* and *pickle* package in order to execute PyNet properly.

2 Layer

- ***class Linear***(input_channel, output_channel, name=*None*, bias=*False*)
  Applies a linear transformation to the incoming data: $y = A \times x + b$

  Parameters

  – input_channel(*int*): number of input channel
  – output_channel(*int*): numbers of channels produced by the linear layer
  – name(*string*): the name of layer (default is **None**)
  – bias: the label whether to introduce bias in this linear layer (default is **True**)

  Shape

  – Input: $(N, inChannel)$ where N represents the batch size and inChannel is the number of input feature dimension.
  – Output: $(N, outChannel)$ where N represents the batch size and outChannel is the number of output feature dimension.

  Variables

  – w: the learnable weights has shape (inChannel $\times$ outChannel)
  – b: the learnable bias has shape (outChannel)
  – grad_w: the gradient of weight, which has shape (inChannel $\times$ outChannel)
  – grad_b: the gradient with respect to bias, which has shape (outChannel)

- ***class Upsample***(size=*None*, scale=*None*, name=*None*)
  Upsample a given multi-channel spatial data, the algorithm is available for upsampling is nearest neighbor and bilinear for 4D input data.

  Parameters

  – size(*tuple*, *optional*): a tuple of ints(Height_out, Width_out) output sizes
  – scale(*int/tuple of python:ints*, *optional*): the multiplier for the image height / width
  – name(*string*): the name of the Upsample layer

  Shape

- Input: $(N,\ C,\ H_{in},\ W_{in})$ where C represents the channel number of input data
- Output: $(N,\ C,\ H_{out},\ W_{out})$

- ***class Relu***(name=*None*)
  Applies the rectified unit function element-wise $ReLU(x) = max(0, x)$.

  Parameters

  - name(*string*): the name of the Relu layer

  Shape

  - Input: $(N,\ *)$ where * means any number of additional dimensions
  - Output: $(N,\ *)$ same shape as the input data

- ***class Sigmoid***(name=*None*)
  Applies the element-wise function element-wise $f(x) = 1/(1 + exp(-x))$.

  Parameters

  - name(*string*): the name of the Sigmoid layer

  Shape

  - Input: $(N,\ *)$ where * means any number of additional dimensions
  - Output: $(N,\ *)$ same shape as the input data

- ***class Flatten***(name=*None*)
  Flattens the input data while maintaining the batch size.

  Parameters

  - name(*string*): the name of the Flatten layer

  Shape

  - Input: $(N,\ C,\ H_{in},\ W_{in})$
  - Output: $(N,\ C \times H_{in} \times W_{in})$

- ***class Softmax***(name=*None*)
  Applies the Softmax function to an n-dimensional input data rescaling then so that the elements of the n-dimensional output data lie in the range (0, 1) and sum to 1. The Softmax function is defined as

  $$f_i(x) = exp(x_i)/\sum_j exp(x_j)$$

  Parameters

  - name(*string*): the name of the Softmax layer

  Shape

  - Input: $(N,\ K)$ where N and K denotes the batch size and dimension of data respectively
  - Output: $(N,\ K)$ same shape as the input

- ***class L2_loss***(average=*True*, name=*None*)

  The criterion that measures the mean squared error between n elements in the input x and target y. The function is defined as

  $$l2\_loss(x, y) = (1/n) \times \sum_i |x_i - y_i|^2$$

  x and y arbitrary shapes with a total of n elements each. The sum operation still operates over all elements, and divides by n.

  Parameters

  – average(*bool*, *optional*): by default, the losses are averaged over observations for each minibatch. However, if the average is set to *False*, the losses are instead summed for each minibatch.

  – name(*string*): the name of the L2_loss layer

  Shape

  – Input x: (N, $*$) where $*$ denotes any number of additional dimensions

  – Target y: (N, $*$) same shape as x

- ***class Binary_cross_entropy_loss***(average=*True*, name=*None*)

  The criterion that measures the Binary Cross Entropy between the target and the prediction. The function is defined as

  $$loss(p, t) = -(1/n) \times \sum_{i=1}^{N} \{t[i] \times log(p[i]) + (1 - t[i]) \times log(1 - p[i])\}$$

  This is used for measuring the error of reconstruction in for example an auto-encoder. Note that the target *t[i]* should be numbers between 0 and 1.

  Parameters

  – average(*bool*, *optional*): by default, the losses are averaged over observations for each minibatch. However, if the average is set to *False*, the losses are instead summed for each minibatch.

  – name(*string*): the name of the Binary_cross_entropy_loss layer

  Shape

  – Input p: (N, $*$) where $*$ denotes any number of additional dimensions

  – Target t: (N, $*$) same shape as p

  – Output: scalar

- ***class Cross_entropy_loss***(average=*True*, name=*None*)

  This criterion measures the negative log likelihood loss in one single class. It is useful when training a classification problem with C classes. The input is expected to contain scores for each class, which has to be a 2D matrix of size (batch_size, C).

  This criterion expects a class index (0 to C-1) as the target for each value of a 1D matrix of size *batch_size*. The loss function is defined as

  $$loss(p, t) = -(1/n) \times \sum_{i=1}^{N} \{log(p[i, t[i]])\}$$

  This is used for measuring the error of reconstruction in for example an auto-encoder. Note that the target *t[i]* should be numbers between 0 and 1.

  Parameters

  – average(*bool*, *optional*): by default, the losses are averaged over observations for each minibatch. However, if the average is set to *False*, the losses are instead summed for each minibatch.

– name(*string*): the name of the Binary_cross_entropy_loss layer

Shape

– Input x: (N, $*$) where $*$ denotes any number of additional dimensions

– Target y: (N, $*$) same shape as x

- ***class Conv2d***(output_channel, kernel_size, padding = 0, stride = 1, name=*None*, bias=*True*)
  Applies a 2D convolution over an input feature map. This layer is doing cross-correlation instead of convolution. The equation to compute output shape should be

$$S_{out} = \frac{S_{in} - 2 * padding + kernel\_size}{stride} + 1$$

Where S_{out}, S_{in} represents the output size and input size respectively.

Parameters

– output_channel(*int*): Number of channels produced by the convolution layer.

– kernel_size (*int or tuple*): Size of convolution kernel

– padding (*int or tuple*): zero-padding added to both sides of the input

– stride (*int or tuple*): stride of convolution

– name (*string*): the name of layer

– bias (*boolean*): if True, adding learnable bias to the output

Shape

– Input: $(N, C_{in}, H_{in}, W_{in})$

– Output: $(N, C_{out}, H_{out}, W_{out})$

Variables

– w: the learnable weight has shape $(C_{out}, C_{int}, kernel\_size\_h, kernel\_size\_w)$

– b: the learnable bias has shape $(1, C_{out}, 1, 1)$

– grad_w: the gradient of weight, which has shape $(C_{out}, C_{int}, kernel\_size\_h, kernel\_size\_w)$

– grad_b: the gradient with respect to bias, which has shape $(1, C_{out}, 1, 1)$

- ***class MaxPool2d***(kernel_size, padding = 0, stride = 1, name=*None*)
  Applies a 2D Maxpooling over an input feature map. The equation to compute output shape should be

$$S_{out} = \frac{S_{in} - 2 * padding + kernel\_size}{stride} + 1$$

Where S_{out}, S_{in} represents the output size and input size respectively.

Parameters

– kernel_size (*int or tuple*): Size of convolution kernel

– padding (*int or tuple*): zero-padding added to both sides of the input

– stride (*int or tuple*): stride of convolution

– name (*string*): the name of layer

Shape

– Input: $(N, C_{in}, H_{in}, W_{in})$

– Output: $(N, C_{out}, H_{out}, W_{out})$

- ***class BatchNorm1D***(momentum = 0.9, name=*None*)
  Applies a 1D batchnormalization over input feature. The mean and standard-deviation are calculated per-channel over mini-batch. This layer perform the algorithm:

  $$Y = \frac{x - mean(x)}{\sqrt{var(x) + eps}} \times gamma + beta$$

  The *eps* is a small value added to the denominator for numerical stability, which is set $1e^{-5}$

  Parameters

  – momentum (*float*): The momentum used for *running_mean* and *running_val*

  Shape

  – Input: $(N,\ C)$ where C represents the channel number

  – Output: $(N,\ C)$

  Variables

  – beta: the learnable parameter has shape (C)

  – gamma: the learnable parameter has shape (C)

  – r_mean: the mean value used for testing, which has shape of (C)

  – r_var: the variance used for testing, which has shape of (C)

  – grad_beta: the gradient of beta, which has shape of (C)

  – grad_gamma: the gradient of gamma, which has shape of (C)

- ***class BatchNorm2D***(momentum = 0.9, name=*None*)
  Applies a 2D(spatial) batchnormalization over input feature. The mean and standard-deviation are calculated per-channel over mini-batch. This layer perform the algorithm:

  $$Y = \frac{x - mean(x)}{\sqrt{var(x) + eps}} \times gamma + beta$$

  The *eps* is a small value added to the denominator for numerical stability, which is set $1e^{-5}$

  Parameters

  – momentum (*float*): The momentum used for *running_mean* and *running_val*

  Shape

  – Input:(N,C,H_{in},W_{in})

  – Output: (N,C,H_{out},W_{out})

  Variables

  – beta: the learnable parameter has shape (C)

  – gamma: the learnable parameter has shape (C)

  – r_mean: the mean value used for testing, which has shape of (C)

  – r_var: the variance used for testing, which has shape of (C)

  – grad_beta: the gradient of beta, which has shape of (C)

  – grad_gamma: the gradient of gamma, which has shape of (C)

3 Model

The model to store the defined layer list and its parameter, connecting the layer and then performoing forward, backward and parameter updating.

- ***method __init__***(input_layers, loss_layer, optimizer = *None*, lr_decay=*None*)
  Input defined network layers and loss layers. Initilizing the model.

  Parameters

  – input_layers(*list*): the list of defined network structure
  – loss_layers(*layer*): the loss layer to compute the loss
  – optimizer(*optimizer*): the optimizer to update the parameter based on the computed gradient. You can ignore this parameter for testing for not updating parameters.
  – lr_decay(*lr_decay*): Decaying the learning rate for each step. Setting to None means the constant learning rate

  Return

  – *None*

- ***method set_input_channel***(dim)
  Set the input channel (dimension) number for the network to initlize layer's weight

  Parameters

  – dim(*int*): the dimension number of input data

  Return *None*

- ***class show_layer_name***()
  Display the layer name and network structure

  Parameters

  – None

  Return

  – None

- ***class forward***(input, label = *None*)
  Do forward computation and compute the loss if the label is given

  Parameters

  – input(*numpy array*): the input data
  – label(*numpy array*): the data label. If the label is None, it will only output the prediction.

  Return

  – loss, prediction (if the label is provided)
  – prediction (if the label is None)

- ***method backward***(loss)
  Do backward computation and compute the gradient

  Parameters

– loss(*float*): the loss obtained through forward computation

Return

– None

- ***method update_param***()
  Updating the model parameter based on the computed gradient. To update the parameter, you need to initlize the model with optimizer.

  Parameters

  – None

  Return

  – None

- ***method get_layer_output***(layer_name)
  Extract the output for specific layer.

  Parameters

  – layer_name(*string*): Denote the layer that the output will be extracted.

  Return

  – output(*numpy array*): The layer output

- ***method get_layer_grad***(layer_name)
  Extract the output gradient for a specific layer. You can use this function only when you use *model.backward*()
  The gradient is the layer output gradient, i.e. the input gradient of its previous layer

  Parameters

  – layer_name(*string*): Denote the layer that the output gradient will be extracted.

  Return

  – output(*numpy array*): The layer output gradient

- ***method train***(is_train)
  Changing the model mode, since the model tend to perform differently during training and testing. For example, batchnorm layer. By default the model mode is training.

  Parameters

  – is_train(*boolean*): Indicate the current model mode, True for training, False for testing.

  Return

  – None

- ***method save_model***(path)
  Saved current model layer, parameter and optimizer history if the optimizer if provided.

  Parameters

  – path(*string*): The saved model path

  Return

– None

- ***method load_model***(path)
Restore the model from the saved model file

  Parameters

  – path(*string*): The saved model path

  Return

  – None

- ***method layer_init***()
Initializing the model with the provided layer. You don't need this method since it's automatically used within the method $model.set\_input\_channel(dim)$

  Parameters

  – None

  Return

  – None

4 Optimizer

The optimizer to update parameter

- ***class SGD_Optimizer***(lr_rate, weight_decay, momentum = 0.99)
The optimizer performing Stochastic Gradient Descent algorithm to update the parameter

$$P = P - \text{lr\_rate} \times (\text{grad} + P * \text{weight\_decay} + \text{momentum} * \text{grad\_history})$$

  Parameters

  – lr_rate(*float*): The learning rate of the optimizer

  – weight_decay(*float*): The weight_decay rate of the optimizer

  – momentum(*float*): The momentum rate of the optimizer

5 Learning Rate Decay

Decaying the learning rate for during training for certain condition.

- ***class Decay_learning_rate***(decay_step = 500, base = 0.96, staircase = True)
This performed exponentially learning rate decay.

$$\text{new\_lr\_rate} = \text{base\_lr\_rate} \times (\text{base}^{\frac{step}{\text{decay\_step}}})$$

  Parameters

  – decay_step(*int*): period of learning rate decay

  – base(*float*): The base to do exponential learning rate decay

  – staircase(*boolean*): Whether to employ stairecase decaying strategy. If set to True, the learning rate will decay only when decay_step is reached. Otherwise, it will decay every step.

6 utils

The utils file stores two helper functions.

- ***method upsample2d***(input, output_size)
  Upsample matrix into the output size.

  Parameters

  - input(*4D ndarray*): 4D matrix with shape (N, C_{in}, H_{in}, W_{in})
  - output_size(*tuple*): define the output size (H_{out}, W_{out}).

  Return

  - *output*: same data type with input but has shape (N, C_{in}, H_{out}, W_{out})

- ***method get_gt_map***(get_label, h, w)
  Convert the ground truth label into matrix format, same dimension as training data.

  Parameters

  - gt_label(*2D ndarray*): a 2D array with shape (batch_size, 10) stores five landmarks position information for each instance. The 10 landmark coordinates should have the order (x1,x2,x3,x4,x5,y1,y2,y3,y4,y5)
  - h(*int*): the height of image.
  - w(*int*): the width of image.

  Return

  - *label*: a 4D matrix representing the converted ground truth data with shape (N, C, h, w)