Guess the way of learning this is through the following:

Going through docs is indeed important. However currently you don’t even have an idea of what docs you should go through.

Guess you can start with this order: loading data, and converting data(includes normalization) before moving on to model construction (as you have enough experience with this part through projects and homework practises)

<https://pytorch.org/tutorials/beginner/basics/intro.html>

however realizing that each coding section has options on top of page indicating whether the code could run on microsoft azune or google colab, which gives you more chance to practise writing them down like you did for javascript.

### PYTORCH

#### Common properties:

##### Tensor related:

Torch.zeros((dimension), dtype=(default)torch.int32)

Torch.ones((dimension), dtype=(default)torch.int32)

Torch.rand((dimension));

Torch.randn((dimension)) -> element-wisely normalized between 0, 1 -> BNN weight generation

Torch.Tensor(custom\_python\_sequential\_data)

Mathematical operations can be applied on tensors directly, including addition, multiplication(scalar), and some mathematical functions:

torch.abs(tensor),

torch.max(tensor, dim),

torch.det(tensor): finding determinant;

torch.std\_mean(tensor, dim=)

torch.reshape(input\_tensor, shape\_array)

torch.range(start, end) # realize start and end are INCLUSIVE!!!

Torch.flatten()

Torch.unflatten

Torch.cat()

Torch.permute

Torch.swapaxis/transpose

Torch.einsum

###### Tensor methods:

# realizing there are too many tensor methods, thus will only record commonly used ones, and for those remaining, will list a category showing what it can do, so can check API with a direction

Properties:

* Tensor.grad: None by default, BECOMES the gradient computed by loss.backward() function after execution.
* Tensor.requires\_grad: False by default.
* Tensor.shape: an array with size “dimension”, each element represents current dimension’s size.

Functions:

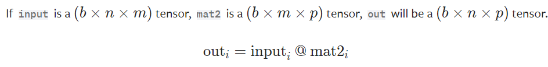
**Bitwise operations;**

* # as this is uncommon, should refer to API for further guidance, and methods will not be listed here.

**Linear algebra functions: (all operations require shape being compatible)**

* Tensor.addr(vec1, vec2, beta=1, alpha=1); an addition with a vector-OUTER-product and input(vector product will be a matrix)

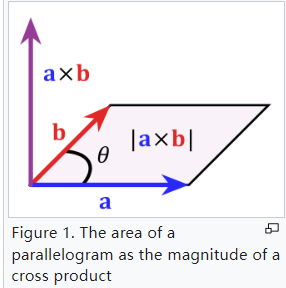
“input” is the tensor itself;

* Tensor.bmm(second-matrix, \*): perform a matrix-matrix product.
* Tensor.add(other): other must have shape boardcastable with Tensor.

Shape boardcasting: other’s shape must be boardcastable with Tensor’s last few dimensions

* Tensor.cross(other, dim=None, )

Returns cross product of vectors from Tensor and other, applied on “dim”;

To avoid problems, need to ensure the vectors size are consistent

* Tensor.det():

Requires tensor to be a square matrix; returns determinant

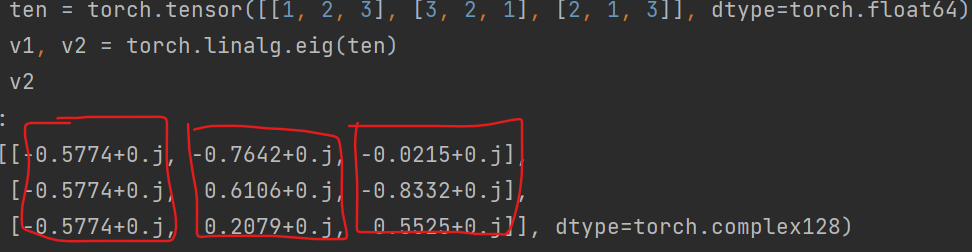
* Tensor.diag(diagonal=0):

Tensor can be either 1D or 2D; if 1D: returns a new square matrix tensor with “Tensor” elements on diagonal; if 2D tensor, will return diagonal elements on “Tensor”

“diagonal”: controls shifting of diagonal line; positive: the diagonal line is drawn above main diagonal;

* Tensor.triu(diagonal=0):

Returns upper triangle of 2D “Tensor”;

* Tensor.dot(other): dot product of two “1D” tensors; (“Tensor” and “other” should be 1D)
* Torch.linalg.eig(A): computes eigenvalue decomposition of “A”, a DIAGONIZABLE square matrix ; returns: tuple(eigenvalues, eigenvectors)

Realize: COLUMNS (dim=1) are corresponding eigenvectors!!!

Final eigenvector matrix is NORMALIZED to 1!!!

* Tensor.matmul(other):

Performs matrix-vector products, requiring size of “Tensor” and “other” to be multipliable;

Order matters: returns Tensor \* other

This function is omniscient, and can automatically determine which operation to adopt based on input vector; more details refer to API;

When dimension is larger than 2, batch\_matrix products are conducted;

* Torch.linalg.norm(A, ord=None, dim=None, ):

Computes vector norm or matrix norm;

Ord: type of matrix norm to choose, if A is a matrix; (refer to API for specific types; )

Common Ord: Frobenius norm

If “dim” is an int, vector norm will be calculated; if 2-tuple, will calculate matrix norm.

* Torch.linalg.inv(A): return inverse of matrix A; requiring squared and invertible.
* Torch.svd(A): A is a batch of square matrices/matrix;

Computes the singular-value-decomposition of matrix A;

Returns a tuple, (U, S, V) where “U” and “V” are square matrices, and “S” is a diagonal matrix;

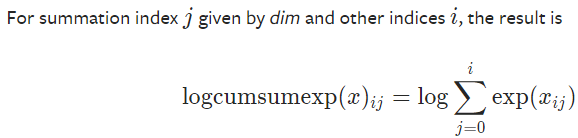
Singular value decomposition can be used:

To determine whether a filter for convolution is separable (see csc420 lecture 2 notes)

**Mathematical functions (includes stable calculation methods, e.g: logsumexp):**

Tensor.asin\_(): with “\_” modification on given tensor is done INSTEAD OF making a copy and returning a new tensor.

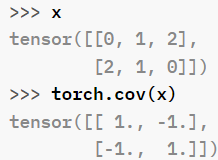
* Tensor.asin()/arcsin()/atan()/arctan()/cos()/cosh()/arccosh()/sin/sinh/arcsinh
* Tensor.ceil()/floor()
* Tensor.logcumsumexp(dim)

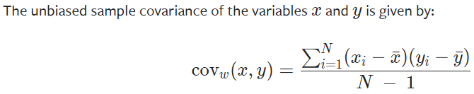
Dimension requires specification;

Realizing the operation is applied on a dimension, to each element of vector on the dimension, and is cumulative w.r.t adding the additional value with previous calculations;

* Tensor.exp(): for each element in tensor, the value will be exponentiated;
* Tensor.logsumexp(dim): logcumsumexp’s variant, with dimension reduced due to only keeping the sum over entire dimension’s result
* Tensor.sum(dim=None)

**Statistical functions:**

* Tensor.amax/amin(dim): take maximum/minimum of Tensor along “dim”; returns VALUE;
* Tensor.argmax/argmin(dim): take maximum/minimum of Tensor along “dim”; returns INDICES;
* Tensor.cov(): gives covariance of given Tensor, by treating:

Tensor as a 2D matrix, where indexing on dim0 gives vector, and indexing on dim1 gives a value in a vector.

Final result: see image on right; realizing that x\_bar and y\_bar are mean value of two vectors respectively.

Refer to API for biased output information; (parameter: correction, fweights, aweights)

NOT COVARIANCE MATRIX!!!

* Tensor.mean(dim=None), Tensor.median(dim=None), Tensor.mode(dim=None), Tensor.sum(dim=None)
* Distributions:

Refer to “probability distributions” in “operation related” section.

**Tensor Operations:**

* Tensor.backward(): compute current tensor’s gradient.

Most commonly applied on final loss, which might be a scalar (“loss.backward()”)

* Tensor.permute()

Your experience shows it could combine with “torch.Unfold” to handle image slicing.

Input parameters are the dimensions expressed by 0, 1, 2…, corresponding to “Tensor”’s

* Tensor.clone(); returns a copy of Tensor;
* Tensor.equal(other): for comparison;
* Tensor.mask\_fill\_(mask, value): “\_” indicates in-place(can be removed to return tensor)

Mask: BOOLEAN tensor, must have shape boardcastable with “Tensor”;

Value: when mask indice is “True”, “value” will replace the value; is a scalar

* Tensor.size(dim=None); returns size of the tensor; when “dim” is specified, a scalar will be returned(the size for the corresponding dimension);
* Tensor.reshape(new\_shape): refer to torch.reshape()
* Tensor.split(split\_size\_or\_sections, dim=0):

Split\_size\_or\_sections: int/list[int], indicating the size of each chunk after splitting;

* Tensor.squeeze(dim=None):

Removes all dimensions in Tensor with size “1”; -> “A x 1 x B” will be changed to “A x B”

Dim: when specified, if the size of “dim” is 1, will be removed; OTHERWISE size remain UNCHANGED

* Tensor.unsqueeze(dim):

Inserts a new dimension with size “1” at given “dim”; “dim” is an int;

* Tensor.expand()
* Tensor.view(dim1\_size, dim2\_size, ….) -> torch.tensor;

The product of each new dimension’s size must match with number of elements in “Tensor”.

* Tensor.to(args) -> NEW TENSOR:

Commonly used for putting a tensor into a device; realizing it returns a NEW tensor!!!

(‘cuda’ if torch.cuda.is\_available() else ‘cpu’)

* Torch.gather()
* Tensor.repeat()

Generating masked array from indices:

scatter

##### Model related:

# note: this note only contains commonly used methods, and only listed common parameters adopted. For detailed usage which might be uncommon, a visit to API is still required.

Torch.nn as nn -- package

Torch.nn.functional as F -- package

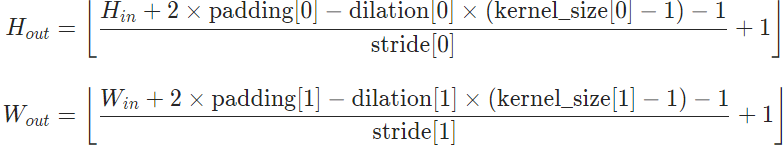
nn.Linear(in\_dim, out\_dim): defines a linear transformation in matrix form

###### Convolutional Layers:

nn.Conv2d(in\_channels, out\_channels, kernel\_size, (stride=1, padding=0, padding\_mode=’zeros’, groups=1))(input) -> (output):

define a convolutional layer

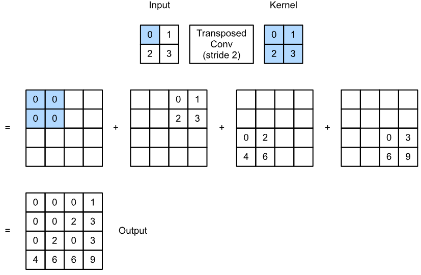
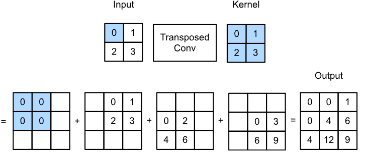
Note that “channel is NOT dimension!!!”; the size of output is determined by the kernel size and input size

Groups: groups input, each grouped input will have independent weights, having same output channel number, but input channel is equivalent to number of input channels in the group. Group must be divisible by input\_channel to perform actions; <https://iksinc.online/2020/05/10/groups-parameter-of-the-convolution-layer/>

Input: has size (NCinHW) or (CinHW); output has size (NCoutHoutWout) or (CoutHoutWout)

nn.ConvTranspose2d(in\_channel, out\_channel, kernel\_size, stride=2, padding=0, (output\_padding=0, padding\_mode=’zeros’, groups=1))(NCHW/CHW):

define a transposed convolution layer; the mechanism is:

for each element (of an input layer), the scalar value will be multiplied with the kernel and the kernel will be located in appropriate position of output, waiting for accumulation of other output results. <https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8>

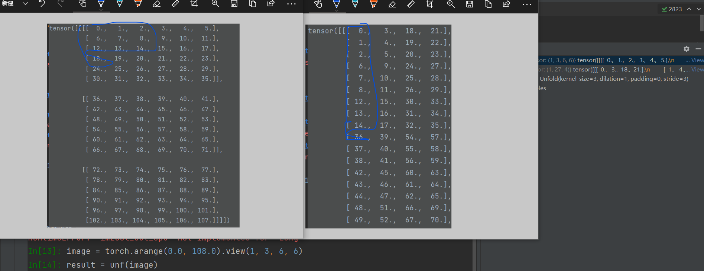
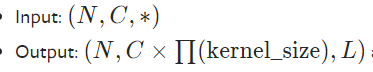
Stride is applied on OUTPUT, padding is applied on INPUT.

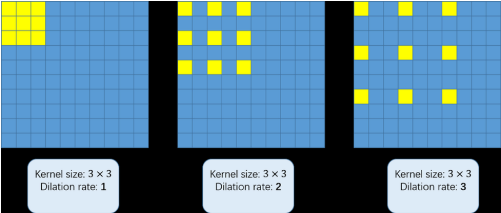
Output\_padding is extra space will be added to output only

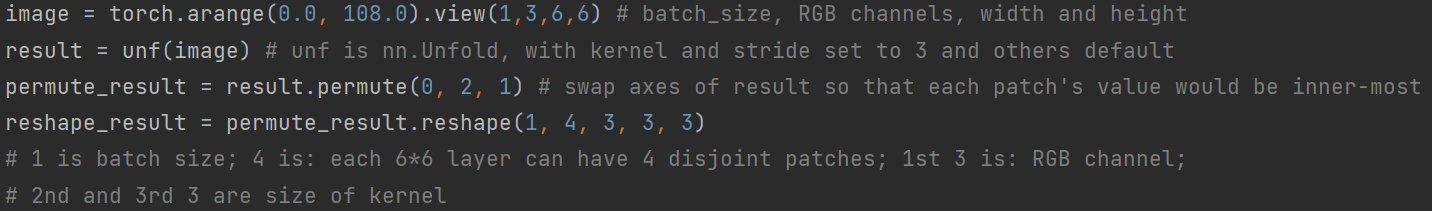
See further reference online.

nn.Unfold(kernel\_size, dilation=1, padding=0, stride=1)

extracts local blocks according to set criterion;

main functioning is for extracting patches from images. Image below shows the result of executing this method.

To demonstrate its true functionality, below is the code segments recommended to use, since it provides a complete procedure for converting images into the form finally required. If feel confused, can run the code in pytorch and see results displayed.



dilation: determines spread of kernel (see graph right)

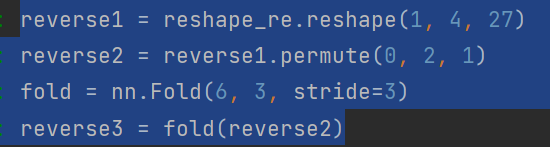
nn.Fold(output\_image\_size(int(for perfect square) or a tuple for each side length),

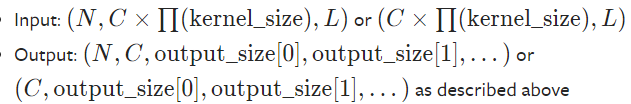
Kernel\_size(be consistent with Unfold’s kernel setting)

Stride(be consistent with Unfold’s kernel setting)

):

perform the exact opposite operations compared with nn.Unfold: it can combine patches into a whole image, which is also necessary under some operations.

The example beside is a continuation of example above. The first step(reverse 1) integrates all patches into one whole array, indicated by the last dimension (27=3\*3\*3, channel \* kernel\_size); then permute the dimension as reversed, and apply Fold; the image size(first argument) is acquired by: (4^0.5) \* 3: the number of kernels for each side of this image (for each channel) times stride size



###### Pooling Layers:

nn.MaxPool2d(Kernel\_size, stride, padding, dilation, return\_indices=False)(NCHW/CHW)

pool the resulting tensor with given configurations;

Kernels can be overlapping if stride is set less than kernel size

Return\_indices=True: indices returned could be useful for MaxUnpool2d below

nn.MaxUnpool2d(kernel\_size, stride, padding)(input\_tensor, max\_indices\_for\_input) (NCHW/CHW)

a PARTIAL reverse of max pooling where non-max values are all set to zero.

Max’s indices for each kernel: can be generated by MaxPool2d; If possible can perform a maxpool on masked array to acquire indices

nn.AvgPool2d(kernel, stride, padding, count\_include\_pad=True, divisor\_override=None) (NCHW/CHW)

instead of extracting max, this one outputs average of values for each kernel

count\_include\_pad: average calculation will add denominators depending on paddings in this kernel

divisor\_override: will change divisor when calculating average (specify a number)

###### Padding Layers:

nn.ReflectionPad2d(padding: int/tuple\_of\_ints)(tensor)

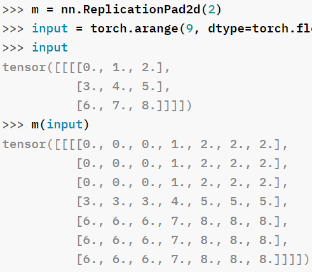
A demonstration of result is shown beside. Pay attention to the reflective axis.

Original input is at the center of resulting padded tensor

“padding”: must have size less than “min(H, W)” of input “tensor”!!!

Tensor: has shape: (NCHW)/(CHW)

nn.ReplicationPad2d(padding)(tensor:NCHW/CHW)

 See demo on right; Replication repeats values CLOSEST to it

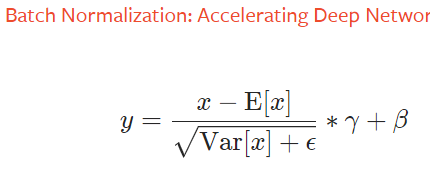
nn.ZeroPad2d(padding)(tensor:NCHW/CHW)

pads extra spaces using zero only

nn.ConstantPad2d(padding, value)(tensor:NCHW/CHW)

Similar to zero padding, but the value can also be other constants not zero.

###### Normalization Layers:



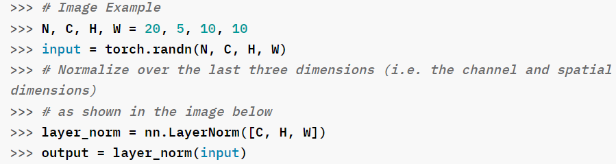
nn.BatchNorm2d(num\_features, eps=1e-05, affine=True, track\_running\_stats=True)(input: has size NCHW, 4D tensor)

num\_features: “C” in above graph(don’t confuse with width/height of input data!!!)

eps: epsilon when calculating normalization configurations (in denominator, to avoid denominator divide by zero error)

affine: the parameter “gamma” and “beta” in normalization formula would be learnable

track\_running\_stats;

nn.LayerNorm(normalized\_shape, eps, elementwise\_affine=True)(input\_tensor)

normalized\_shape: a tuple of integers; length of tuple determine how many dimensions COUNTED FROM LAST would be normalized

Example shown left shows how normalized\_shape shall be given (usually just the shape of last few dimensions of input tensor)

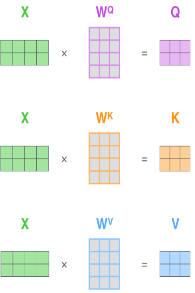
nn.InstanceNorm2d(num\_features, eps=1e-5, affine=True, track\_running\_stats=True)(input: 3D/4D tensor having size CHW/NCHW)

mechanism see above graph illustration, and parameter explanation see BatchNorm2d

###### Non-Linear Activations:

# as the functions are just directly applied, will only list them here for a quick scanning lookup.

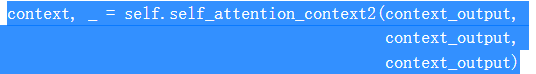
# specific calculations of these activations require online studying of mechanisms.

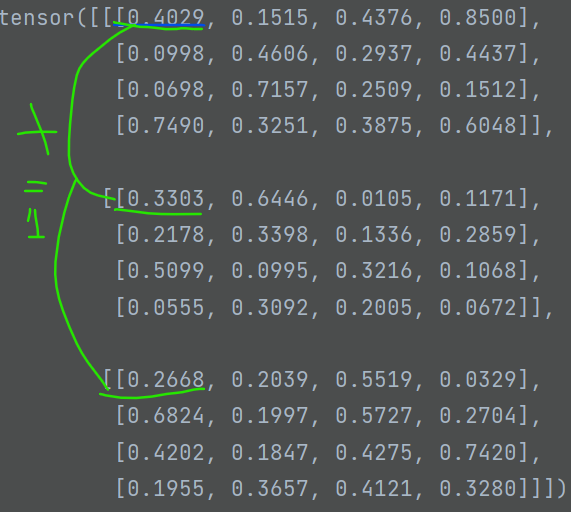
# all activation functions are applied element-wisely

# all listed

nn.LeakyReLu(), nn.ReLu(), nn.LogSigmoid(dim), nn.Sigmoid(), nn.Tanh(), nn.Softmax(dim=None/integer),

nn.MultiheadAttention(embedding\_dimension, number\_heads, dropout=0.0)(query, key, value)

according to the code usage shown right, query, key and values are those inputs for generating query, key and value, thus example code simply used output to be all 3 inputs.



nn.Softmax2d()(NCHW)

image right shows the effect: adds values from all channels at each coordinate and do softmax on that -> channel1+channel2+channel3 for each coordinate

###### RNN:

One thing worth noticing is that, cell weights are actually shared; this helps reducing the number of parameters to train.

RNN:

weights and biases are initialized along with creation of RNNCell. “x”: input; “h”: previous layer’s output;

realizing when # of layers is not large, using RNN provides computational efficiency, while the effect of gradient vanishing is not too significant.

nn.RNNCell(input\_size, hidden\_size, bias=True, nonlinearity=’tanh’)(input\_tensor, hidden\_tensor)

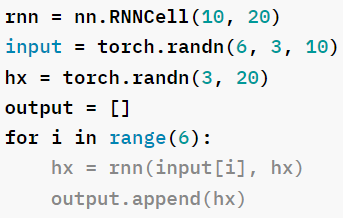
create an RNN cell based on equation described above.

Input\_size: the size of input\_tensor;

Hidden\_size: the size of hidden\_tensor as input;

Bias: if false, no bias will be taken into account;

Nonlinearity: if ‘relu’, will change activation function to relu instead of tanh as default

 Input\_tensor has shape (N, input\_size)

Hidden\_tensor has shape(N, hidden\_size)

Outputs h’, has shape(N, hidden\_size)

An example run shown right gives idea of how previous output along with input shall be used.

nn.RNN(input\_size, hidden\_size, num\_layers, nonlinearity=’tanh’, bias=True, dropout=0, bidirectional=False)(input\_tensor, previous\_h\_tensor)

constructs an RNN where each element is an RNNCell, and the connection method follows from description of RNN

num\_layers: how many recurrent layers will present in RNN

dropout: a ratio between 0 and 1; will have a Dropout layer (see later descriptions) for each output with probability set as “dropout”

input\_tensor: has shape (L, N, input\_size); L is sequence length, input\_size is # of features; N is batches.

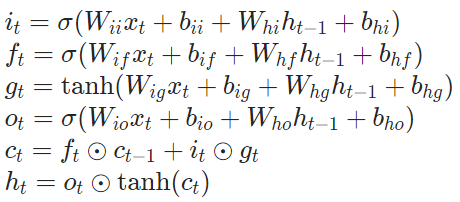
previous\_h\_tensor(OPTIONAL): in case a complex structure has multiple RNN each with different configuration, then previous RNN’s h\_value could be fed as input here.

If not provided, will default as zero.

Has shape (D\*num\_layers, N, hidden-size); D is 2 if bidirectional is True

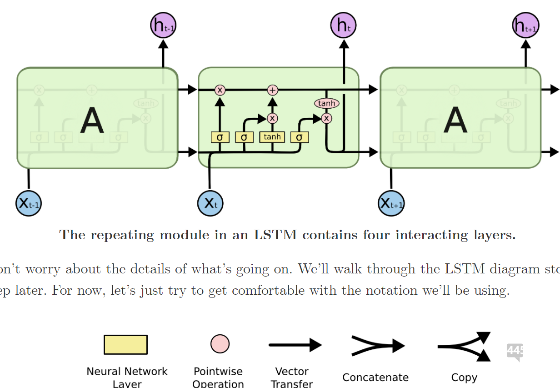
Output:

(Output\_tensor, final\_hidden\_tensor); has shape [(L, N, hidden\_size), (D\*num\_layers, N, hidden\_size)]

LSTM:

h: hidden state; c: cell state; x: input; ifgo: input/forget/cell/output gates;

circle with a dot is element-wise matrix product.



Mechanism: when inputs and previous hidden states are given: first check forget gates (“f”) and see whether previous hidden states shall be kept or ignored.

Then “i” and “g” will be calculated, to respectively determine which values to update(by “i”), and how much (by “g”)

Then “c” is updated based on forget gate and update gate’s contents, which gives current cell’s state.

Finally “o” is computed, along with next hidden state. New hidden state will take current cell state “c” into account.

In general, LSTM allows more manipulation on previous states, so that when analyzing long sequences, only important info would be kept; results in less noise towards understanding current state. Also this resolves “gradient vanishing problems”.

However without attention, when sequence is long enough LSTM could fail as well

To elaborate, the structure of each LSTM cells shown in diagram could be easily modified to have various extra functionings.

nn.LSTMCell(input\_size, hidden\_size, bias=True)(input, (hidden, cell)) -> (late\_hidden, new\_cell)

input: has size (N, input\_size)

hidden: has size (N, hidden\_size)

cell: has size (N, hidden\_size): the initial cell’s state;

If (hidden, cell) is not provided, will both default to zero.

Late\_hidden: has size (N, hidden\_size); tensor containing next hidden state

New\_cell: has size (N, hidden\_size); tensor containing next cell state

nn.LSTM(input\_size, hidden\_size, num\_layers, bias=True, dropout=0, bidirectional=False, proj\_size=0)(input, (h0, c0)) -> output, (hn, cn)

proj\_size: if >0, final output will be linearly projected to this dimension.

input: has size (L, N, input\_size)

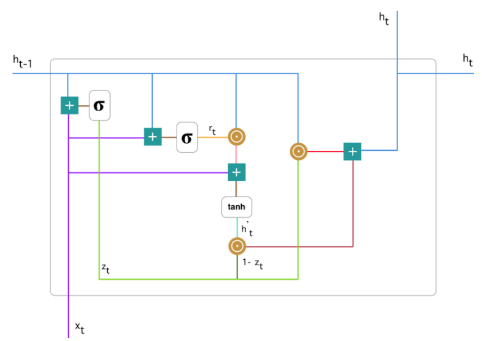
h0: has size (D\*num\_layers, N, hidden\_size/proj\_size if proj\_size>0); D=2 if bidirectional

c0: has size (D\*num\_layers, N, hidden\_size)

output: has size (L, N, D\*hidden\_size/proj\_size if proj\_size>0)

hn: (D\*num\_layers, N, hidden\_size/proj\_size if proj\_size>0)

cn: has size (D\*num\_layers, N, hidden\_size)

GRU:

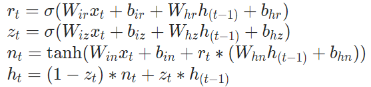
consists of reset gate(r), update gate(z) and new gate(n);

mechanism:

first calculate update gate(z); this determines how much previous gate’s info shall be passed on;

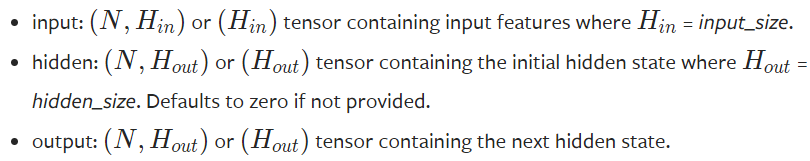
Then calculate reset gate(r); this determines how much info shall be forgotten; (in diagram reset gates and update gates are parallel)

Then new state is calculated; it represents current cell’s status, containing current input, along with previous hidden states, with a reset ratio.

Finally new hidden state is calculated based on previous hiddens and current cell status.

nn.GRUCell(input\_size, hidden\_size, bias=True)(input, hidden) -> new\_hidden

shape:



nn.GRU(input\_size, hidden\_size, num\_layers, bias=True, dropout=0, bidirectional=False)(input, h\_0) -> output, h\_n

shape info:

input: (L, N, input\_size); h\_0: (D\*num\_layers, N, hidden\_size)

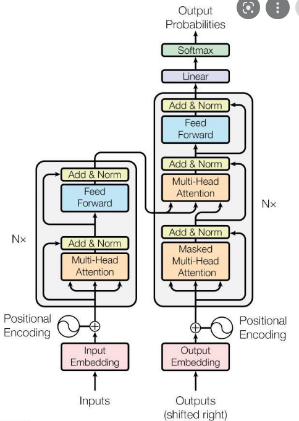
output: (L, N, D\*hidden\_size); h\_n: (D\*num\_layers, N, hidden\_size)

difference between GRU and LSTM:

* Forgetting and updating: GRU’s forget/reset gate only handles previous hidden;

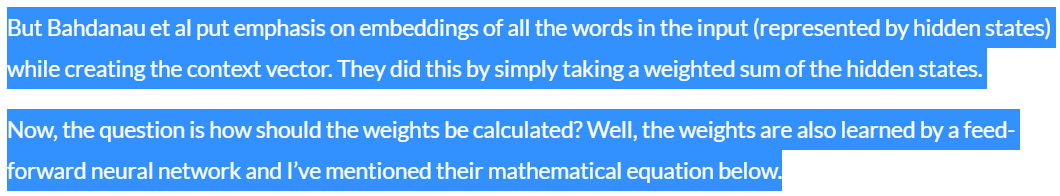
But LSTM forgets both input and hidden by a certain rate;

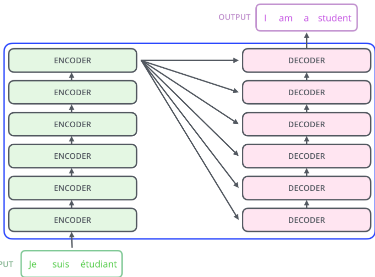
same with update gate; in general: LSTM’s updating and forgetting is applied on COMBINED result of current input and previous hidden state; however GRU deals with previous hidden state separately from current input. (see formula of how forget gate and reset gate are calculated for better understanding)

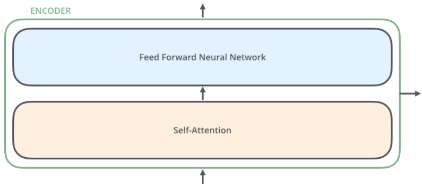
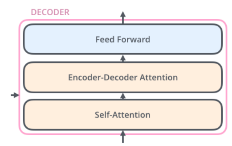


###### Transformer Layers:

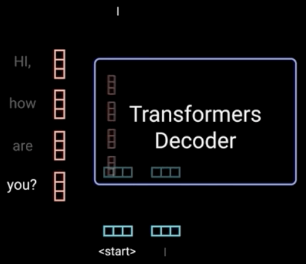
Intuition behind attention mechanism:



Transformer Structure:



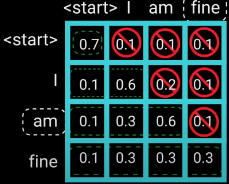
Each encoder’s output will become part of each decoder’s input.

Transformer decoder’s input:

Two parts: one is the whole encoder sequence’s output, the other is the previously generated output of whole transformer model (at beginning of translation, a “start” signal is given instead)

transformer uses previously generated outputs and encoder outputs to predict what to generate next one by one, and the new word generated also become part of decoder’s next input.

<https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0>

Functionality of masking in decoder:

As shown in figure, when predicting next word, the attention score required should only rely on previously generated output’s score. Thus the score for word “fine” is masked out when word “am” is predicted.

nn.Transformer(feature\_dim=512, num\_heads=8, num\_encoder\_layer=6, num\_decoder\_layer=6, dim\_feedforward=2048, dropout=0.1, activation=”relu”, layer\_norm\_eps=1e-05)

feature\_dim: expected input features for encoder and decoder.

Num\_heads: number of heads in multihead attention layer

Dim\_feedforward: the dimension of feedforward neural network inside each encoder and decoder.

----------------

(src, tgt, \*src\_mask, \*tgt\_mask, \*memory\_mask) -> output

Src: the sequence of encoder input, has shape (encoder\_sequence\_length, N, embedding\_dim)

Tgt: the sequence of decoder extra input, has shape (decoder\_sequence\_length, N, embedding\_dim);

For natural language processing task, the extra input is usually a signal for start as the first element, and all remaining parts are zero vectors, awaiting for filling in;

Src\_mask: has shape (encoder\_sequence\_length, encoder\_sequence\_length): checks when each input is attended at, which key words shall be ignored from other src input.

Tgt\_mask: has shape (decoder\_sequence\_length, decoder\_sequence\_length)

Memory\_mask: has shape (DEcoder\_sequence\_length, ENcoder\_sequence\_length)

Ensures when decoding, which encoder output shall be masked when determining each position of decoder respectively

Also indicates dimention 0 is the position to evaluate, and dimension 1 is the words to mask when evaluating dimension 0’s position.

--------------

Output: has shape (decoder\_sequence\_length, N, embedding\_dim) if “batch\_first=False”

nn.TransformerEncoderLayer(input\_feature, nhead, dim-feedforward=2048, dropout=0.1, activation=”relu”, layer\_norm\_eps=1e-5)

(src, src\_mask=None)

create an instance of transformer encoder layer as illustrated in figure above.

nn.TransformerEncoder(encoder-layer, num-layers, norm=None)

(src, mask=None)

creates a stack of “num-layers” many transformer encoder layers.

Looks like all stacked encoders are the same, provided by the input parameter.

encoder-layer: an instance of “nn.TransformerEncoderLayer()”.

nn.TransformerDecoderLayer(input\_feature, nhead, dim-feedforward=2048, dropout=0.1, activation=”relu”, layer\_norm\_eps=1e-5)

(tgt, memory, tgt\_mask=None, memory\_mask=None)

Create a decoder layer for transformer as described in previous diagrams

memory: the result from FINAL encoder layer ONLY;

tgt: realizing the decoder layer might not be the first layer; thus apart from beginning of “tgt sequence”, other elements in the “tgt sequence” might also be non-zero;

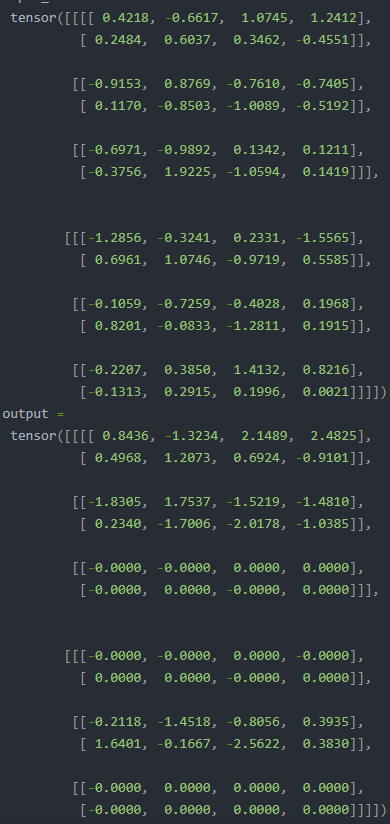
nn.TransformerDecoder(decoder-layer, num\_layers, norm=None)

(tgt, memory, tgt\_mask=None, memory\_mask=None)

Creates a stack of transformer decoder layers.

decoder-layer: an instance of the “TransformerDecoderLayer()”

###### Dropout & Sparse/Embedding & CosineSimilarity Layers:

Dropout:

Dropout layers are used to help improving independence between input layers, which can prevent model from overfitting.

<https://blog.csdn.net/weixin_42419002/article/details/116204674>

nn.Dropout2d(p=0.5, inplace=False)(input)

mask the **entire channel** given a input with shape (NCHW)/(CHW):

When the channel “C” is chosen, input[:C:…] will be filled by 0.

p: the probability of masking the given channel, following a Bernoulli distribution

input: has shape (NCHW)/(CHW) depending on input shape

output has same shape as input

nn.Dropout(p=0.5, inplace=False)(input)

mask the input entries randomly by a probability of “p”.

compared with Dropout2d, the masking is irregular, and not restricted to channel

input: a tensor with no restrictions on shape

output has same shape as input

nn.AlphaDropout(p=0.5, inplace=False)(input)

randomly mask a value with probability “p”, and MAINTAIN mean and std of data. (specific details require reading papers)

input and output must have the same shape.

Embedding:

nn.Embedding(num\_embeddings, embedding\_dim, padding\_idx=None, max\_norm=None, norm\_type=2.0, scale\_grad\_by\_freq=False, sparse=False)(input\_tensor)

num\_embeddings: the number of elements to be indexed by embedding;

embedding\_dim: the # of features of each embedded vector;

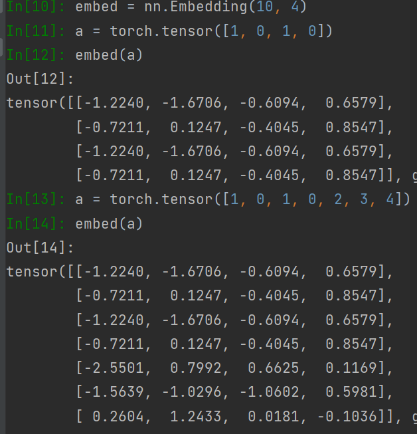
padding\_idx: the integer for padding the remaining space of input -> ensure input length consistency for those short inputs.

Max\_norm: a restriction on embedding vector’s norm, to ensure consistency of vector scale.

Norm\_type: the p-scale (specific restriction on the norm of embedding vector)

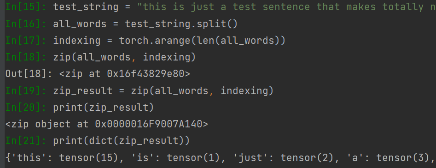
Scale\_grad\_by\_freq: if True, will scale gradients (in backward pass) in INVERSE of word’s frequency

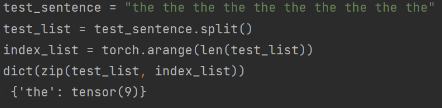
Input\_tensor: can have any shape;

Output: have shape (input\_tensor\_shape, embedding\_dim) -> the set of embedded vectors for all input components

**Tips on using Embedding:**

Refer to code shown left;

* realizing that embedding tries to embed EACH ELEMENT in the input tensor; thus, for example, taking in a number of words/strings, the first step is to iteratively assign each string a number(), and then input the numbers into embedding function to acquire a translated result for training models.
* the number can be acquired using a dictionary, mapping each string to a number; the efficient way to create such dictionary is:

using “zip()” function in python; the code shown right uses the function to combine two length-consistent lists into a pair-wise dictionary.

When “dict()”ing the zipped result, the duplicated words will be handled automatically;

nn.EmbeddingBag(num\_embeddings, embedding\_dim, max\_norm=None, norm\_type=2.0, scale\_grad\_by\_freq=False, padding\_idx=None, mode=’mean’)(input, [offset])

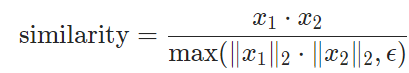
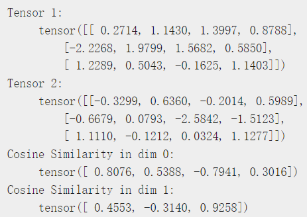
a combination of Embedding and a “mode” operation;

mode: “mean”, “max” or “sum”, a function applied to resulting output tensor AT DIMENSION: dim=1;

input: tensor of shape (B, N)/(N)-> “B(atch)” sequences each with length “N”

offset: when input is of shape (N), offset is resopnsible for providing indexes to “cut” the sequence into “B” many sequences; zero-length sequence will be a zero-vector.

Cosine Similarity:

nn.CosineSimilarity(dim=1, eps=1e-8)(x1, x2)

returns cosine similarity between x1 and x2, computed along “dim”

x1 and x2 must have same number of dimensions, and must agree on length at “dim” dimension.

output: has shape consistent of all dimensions as x1 and x2 with the dimention calculating similarity removed.

Example shown upper right provides how the dimension is applied: the vectors will be extracted in the dimension for each element.

###### Loss Functions:

nn.L1Loss(), nn.CrossEntropyLoss(), … better refer to API for choosing appropriate loss function.

nn.MSELoss()

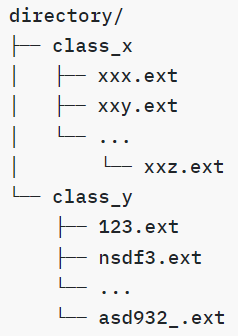
choosing loss function:

#### Operation related:

##### Torchvision:

This part will focus particularly on how to load a set of data and provide a quick look-up for basic image pre-processing methods, which might also include code-snippets for better understanding, just like the demonstration for method “torch.Unfold”.

###### Loading data/custom dataset:

**Directory:**

Based on CSC413’s project, dataset loading directory could be a file directory, with sub-directories where samples under the same category are being grouped into one folder, and the name of the folder would be the class-name (for supervised learning datasets)(see image left for dataset organization)

**Torchvision.datasets.ImageFolder**(root, transform=None, target\_transform=None, loader=<function default\_loader>):

Root: root directory’s file path

Transform: of type “torchvision.transforms”, see documentation below for specifics

As the transformer is sspecified for a folder, the transform must take PIL image as input!!!

Target\_transform: of type “callable” (a function header) which takes in target and transform it according to the mechanism of the function

Torchvision.datasets.DatasetFolder(root, loader, extensions=None, transform=None, target\_transform=None, )

Loader: of type “function” which loads a sample given path as input

Extensions: of type “tuple[str, str…]” containing extensions of files available to be read: jpg, png, …

Torchvision.datasets.VisionDataset(root, transforms=None, transform=None, target\_transform=None)

Transforms: NOT “TRANSFORM”!!!: takes in both a PIL image and a target, return transformed version of both.

* this is an abstract class, all classes inheriting it requires overwritting \_\_getitem\_\_ and \_\_len\_\_ method, or raise “not implemented error”.

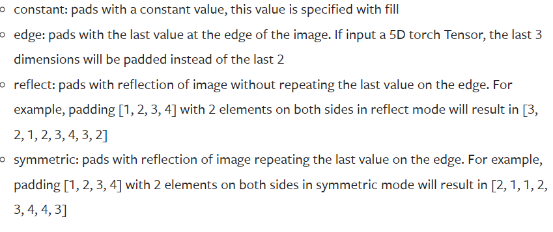
###### Processing images:

**This part also contains function not belong to torchvision, but also commonly used in image processing;**

* Operations include cropping, resizing, color, grayscale, random-operations, padding
* Input PIL image/tensor usually requires the following shape: […, channel/RGB, width, height]
* Random operations are usually used for creating IMAGE AUGMENTATION;
* Interpolation techniques (not in torchvision): used for approximating details when enlarging images, to reduce loss of details.

Torchvision.transforms.Compose([transforms, …]): returns a new transform object

Performs the transforms sequence being composed together.



Torchvision.transforms.Pad(padding, fill=0, padding\_mode=’constant’)(input\_PIL/tensor)

Padding: the size of padding, if ‘int’ then same ‘padding’ on all 4 sides;

If size is 2/4: padding on each side with different value;

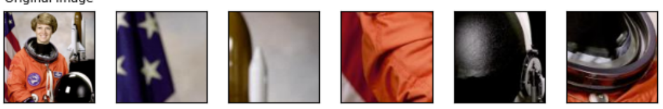
Padding\_mode: ‘constant’, ‘edge’, ‘reflect’, ‘symmetric’

Specific details see image left.

Torchvision.transforms.CenterCrop(size)(input\_PIL/tensor):

Extract the center of “input\_PIL” image by given “size”.

Torchvision.transforms.FiveCrop(size)(input\_PIL/tensor):

 Returns a tuple of length 5, each element is a cropped image by “size”

Torchvision.transforms.RandomCrop(size, padding=None, pad\_if\_needed=False, fill=0, padding\_mode=’constant’)(input\_PIL/tensor):

* “fill” is the value for padding filling.
* “padding\_mode” can be the following: ‘constant’, ‘edge’, ‘reflect’, ‘symmetric’

Specifics can refer to API; only ‘constant’ requires ‘fill’;

Torchvision.transforms.Grayscale(num\_output\_channels=1)(input\_PIL/tensor) -> new\_image:

* Num\_output\_channels: as images usually consist of 3 channels, if output channel is 1: then returned image would only be a grey-scale image; if output channel is 3: then each RGB channel will be propotionally “grey-scaled”;
* Input\_PIL has shape […, 3, H, W]

Torchvision.transforms.RandomGrayscale(p=0.1)(input\_PIL/tensor)

“p”: the probability of applying grayscale transformation on current set of sample images.

* Channel: the output image’s number of channels is consistent with input image’s channels.

Torchvision.transforms.RandomHorizontalFlip(p=0.5)(input\_PIL/tensor)

Perform horizontal flip on some images from the set of images, with probability “p”.

Torchvision.transforms.RandomVerticalFlip(p=0.5)(input\_PIL/tensor)

Perform vertical flip on some images from the set of images, with probability “p”

Torchvision.transforms.RandomInvert(p=0.5)(input\_PIL/tensor)

Perform inversion on color with given probability “p”

Can refer to image shown right for example

Torchvision.transforms.RandomRotation(degrees, interpolation=<interpolation mode, default: nearest>, expand=False, center=None, fill=0, )(input\_PIL/tensor)

Degrees: usually takes the form of: (min, max) or (-degrees, degrees)

Interpolation: requires checking API when using this parameter.

Expand: if True, the output image will be large enough to hold the entire image;

Center: the rotation center; default is center of input; (origin “(0, 0)” is upper-left)

Fill: the value to fill in missing pieces during rotation.

Torchvision.transforms.ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)(input\_PIL/tensor):

* All parameters can be either a float or a tuple of two floats (range of min, max)

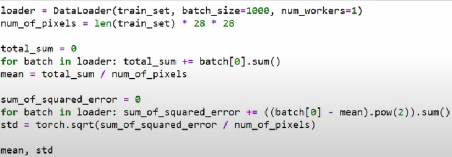
Torchvision.transforms.Resize(size, …)(input\_PIL/tensor)

**Below are methods for Tensor transforms only!**

Torchvision.transforms.Normalize(mean, std, inplace=False)(tensor):  
 Tensor has input shape […, C, H, W];

“mean” and “std” are sequences having “C” many elements, one for each channel.

Inplace: whether to modify on input or return a new tensor.

 Result of normalization:

* Idea of finding mean and std: follow the example shown right;

Explanation: this is for grayscaled images, but the same idea can be applied to a RGB channel-wise. The normalization is done across the channel, thus the sum should be all pixels’ value of all images in current batch, under the channel.

Torchvision.transforms.LinearTransformation(transformation\_matrix, mean\_vector)(tensor):

Perform transformation on input TENSOR. Procedure:

1. Flatten the input tensor into shape (…, D, -1) where D=C x H x W;
2. Subtract from mean\_vector and then multiply with transformation\_matrix;
3. Reshape into original image’s shape: […, C, H, W]

Transformation\_matrix has shape [D, D]; mean\_vector has shape [D]

**Below are image-tensor type conversion methods:**

Torchvision.transforms.ToPILImage(mode=None)(tensor)

Tensor shape info can refer to API.

Typically when input tensor has 3 channels, those three channels are respectively treated as: C, H, W

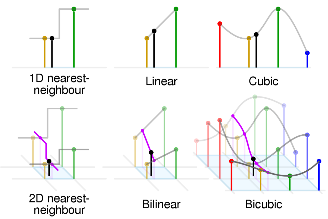
Torchvision.transforms.ToTensor()(input\_PIL)

Specific shape treatements shall refer to API.

Typically the output tensor has shape [C, H, W] and all values are within 0-1

**Below are non-torchvision methods used when converting images:**

Torch.nn.Unfold(), Fold(): see references in “Model\_related/Convolutional Layers”

Torch.nn.functional.interpolate(input, size=None, mode=”nearest”, )

When enlarging images, each pixel is also enlarged makes image not smooth and detailed enough. Thus require enterpolation methods to reduce the effect.

Basically interpolation takes two discrete pixels from original image, and construct a function (can be discrete/continuous, can refer to right image), to determine which value should newly added intermediate pixel take.

Input: has size 3D, 4D or 5D, respectively interpreted as: NCW, NCHW, NCDHW

Size: a sequence of integers representing OUTPUT dimension:

Should be 1D, 2D, 3D respectively, as: W, HW, DHW

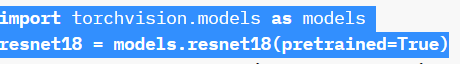
Mode: can refer to image shown right for mechanisms;

Modes are related to input dimension size, e.g “Bilinear” cannot be applied to 1D/3D inputs; more details must refer to API before choosing the right interpolation method;

###### Pre-defined models:

**Loading a model**:

Requires importing model package, initializing model and specifying pretrained parameter;



**Below is an incomplete list of models available in “torchvision.models”;**

**Should refer to API and specific papers to determine whether to use one model over the other**

**\\\\**

AlexNet, VGG, ResNet, SqueezeNet, DenseNet, Inception v3, GoogLeNet, ShuffleNet v2, MobileNetV2, MobileNetV3, ResNeXt, Wide ResNet, MNASNet, EfficientNet, RegNet, VisionTransformer, ConvNeXt

##### Torchtext:

This package provides necessary functionalities for NLP tasks, including loading data, pre-processing data and a list of available models.

Pay attention to the version of this package: below references to 0.12.0 torchtext version only!

###### Preliminary: NLP text pre-processing & featur(e)-ization

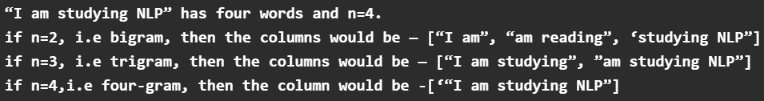
Pre-processing/standardization: eliminate unnecessary symbols, uncaptilize all characters, cut sentences into segments of words, and make each word distinct -> a “dictionary” is formed

**BagOfWords**: creating vectors based on each word’s frequency in each input;

****Example is shown left: for each input sentence/training samples, the corresponding (word, sample) position will indicate the number of occurrence for “word” in the “sample”;

However this method doesn’t take order of words into account, and not suitable for advanced NLP tasks.

**N-grams Vectorization**: an extension of Count Vectorizer: represent combination of words to take word order into account;

Final feature representation is still represented by a sparse matrix, as shown above.

Choice of N: HEURISTICALLY this might depend on the length of input samples: for short sentences, a smaller N is preferred, while for paragraph-predictions a whole sentence might be better.

N-Gram can have too many features to learn, and may result in poor performance of “N” is not suitable for input.

**TF-IDF Vectorization**:

- TF: term frequency: the percentage of one word’s occurrence in ONE document

- IDF: inverse document frequency: the logarithmic ratio of “total # of doc” OVER “total # of docs with ‘word’ in it”.

- The weight of TF-IDF is found by the product of TF and IDF.

- The final feature vectorization result has similar formulation as Bag of Words, except at coordinate (word, doc) the value would be the weight calculated from last step.

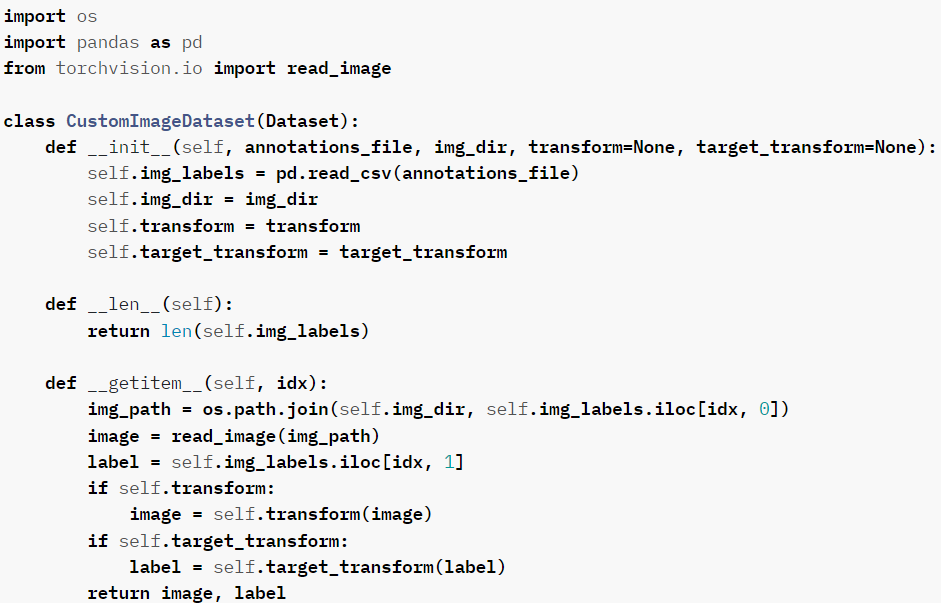
- This vectorization would assign lower weights to high-frequency words (such as “I”, “the”) and higher weight to unusual words which might be important.

###### Loading data:

Class Torch.utils.data.Dataset()

Abstract dataset class being inherited by almost all datasets.

All subclasses of this should overwrite “\_\_init\_\_()” and “\_\_getitem\_\_()” method, and “\_\_len\_\_()” method optionally

Below is an example of creating a custom dataset CLASS;

Explanation:

1. Path of data: in “init” usually a dataset ‘DIRECTORY\_path’ is required, and in “getitem” method, the “os” package’s method is used for finding the file given its file\_path.

Path.join: this helps joining the data directory with the “specific file name”;

1. Transforms would be initialized in “init”, and when trying to return the data, can consider applying the transformation on it.
2. Return value for “\_\_getitem()\_\_” is usually a tuple of format: (data\_instance, label)

For unsupervised\_learning, label is usually not requried.

CLASS Torch.utils.data.DataLoader(dataset, batch\_size=1, shuffle=False, num\_workers=0, …)

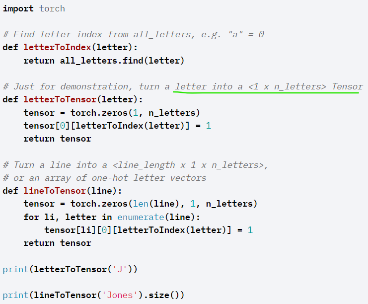
Dataset: of type “CLASS torch.utils.data.dataset”

Shuffle: the order of input batch data will be in random order; suggested to set to True

num\_workers: how many subprocesses to use for loading the dataset. If zero, will be loaded in main process only. (might be useful for multiprocessing)



Apart from using DataLoader, it’s also possible to read a file, and simply convert the data into tensors, following procedures shown in the image left.

the first step is to read the lines from the file, and using string operations to split lines, and eliminate white spaces.

The second image shows one method of converting text into tensor data, by using one-hot key for letters.

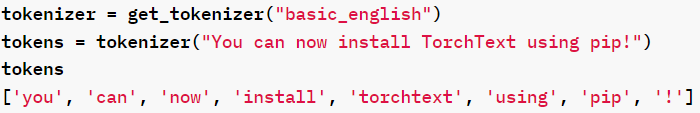
###### Converting text to tokens/numeric tensors:

torchtext.data.utils.get\_tokenizer(tokenizer, language=’en’)(input\_str)

tokenizer can have following options:

1. None: str.split() function;
2. “basic\_english”: \_basic\_english\_normalize(): normalize the sentence and then split.

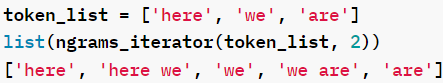
<https://text-docs.readthedocs.io/en/latest/_modules/torchtext/data/utils.html>

above website is source code, where the doc provides all operations to normalize a sentence: lower-casing, remove “!?...”, add space, etc.

1. Callable functions or tokenization library

Return: Returns a list of tokens after splitting on whitespace;

A python list containing possibly all English words (in “str” format)

Trochtext.data.utils.ngrams\_iterator(token\_list, ngram\_factor):

Token\_list: a list of tokens

Ngram\_factor: referring to preliminary part: the number of words for each n-gram-piece

**TOKEN\_VECTORIZATION\_EMBEDDING:**

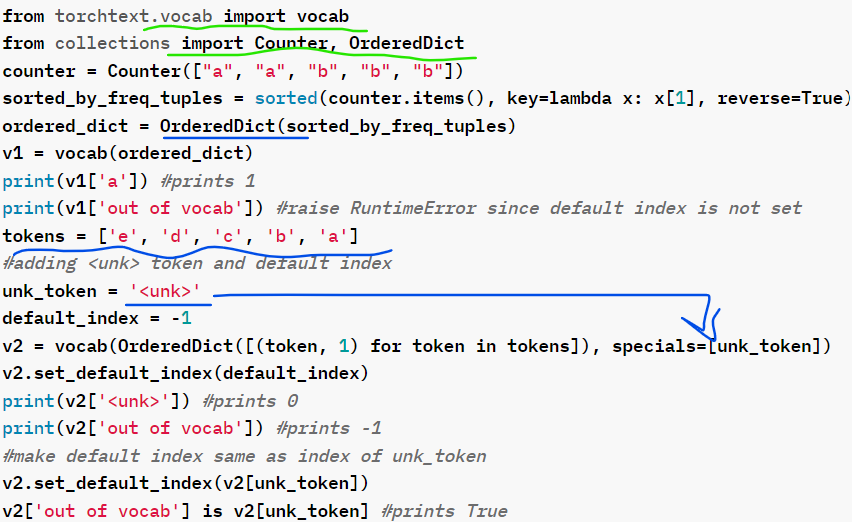
Class torchtext.vocab.Vocab:

The class represents an embedding from a (text) token into a numeric vector as representation of the token element;

Common methods include:

\_\_getitem(token:str), contains(token), len()\_\_\_;

append\_token(token:str); get\_default\_index(); get\_stoi()->Dict[str:token, indices:int]

insert\_token(token, index), lookup\_indices(tokens: List[str])->List[str], lookup\_tokens(indices: List[int])->List[str]

<https://pytorch.org/text/stable/_modules/torchtext/vocab/vocab.html#Vocab>

torchtext.vocab.vocab(ordered\_dict, min\_freq=1, specials=None, special\_first=True)

This is a convenient “VOCAB” initializatino function.

---------------------------------------------------------------------------

ordered\_dict: of type “OrderedDict” in python, where key order is preserved;

min\_freq: the minimum frequency for including a token in vocabulary;

specials: special symbols (SOE, EOF, …) which could be added into token

special\_first: whether symbols are inserted at beginning or end of input token;

Output: “torchtext.vocab.Vocab” type;

CLASS torchtext.vocab.Vectors(name, cache=None, url=None, max\_vectors=None):

This is the parent class of all pre-trained word embeddings, such as GloVe;

----------------------------------------------------------------------------------------

Name: name of file containing vectors;

Cache: directory for cached vectors (not commonly used;)

url: url for download if not vector not found in cache;

max\_vectors: an integer for restricting number of vectors loaded (for memory saving)

-----------------------

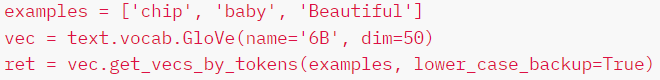
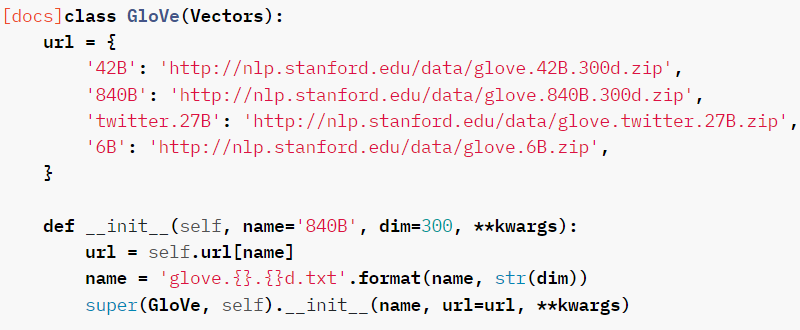
method:

get\_vecs\_by\_tokens(tokens, lower\_case\_backup=False):

token: a string or a list of string;

lower\_case\_backup: whether to look for tokens in lower-case; if not, all letters must match with exactly the same letter as in “tokens”

CLASS torchtext.Vocab.GloVe(Vectors)

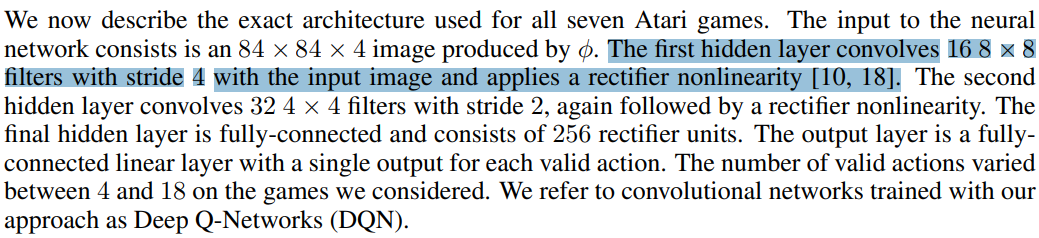
 

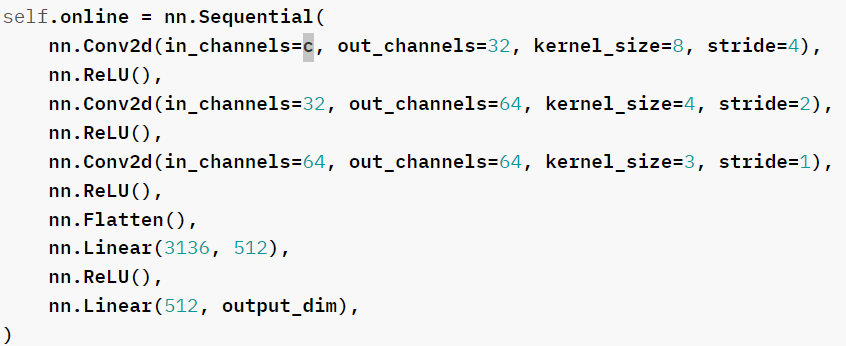
Realizing that “ret” in above code is: a tensor with size (3, 50) where “3” indicates “3” words from “examples” list, and “50” means “dim=50”, the vector for representing each word has dimension 50. The vectors would be already “embedded”, and can be used directly.

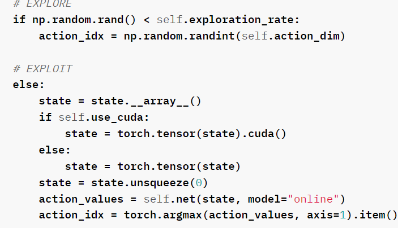
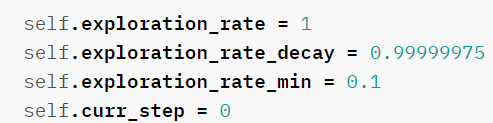
##### Reinforcement learning:

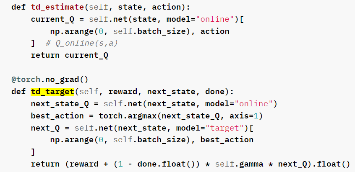
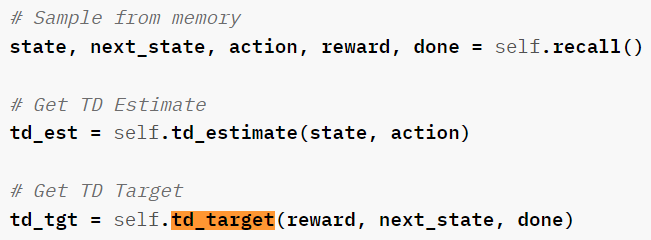
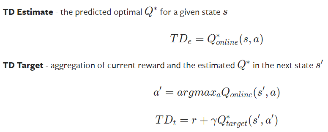
One intuition: the policy is stored in the format of a matrix, where indexing on rows[i] gives the ith state, and indexing on a column of a row (row[i][j]) gives the info of action “j” at state “i”: the expected reward;

The network is created with first layer’s input dimension same as # of states, and last layer’s output dim same as # of actions (assumed the action could take in any state).

The intermediate layers of the network consists of convolutional layers and linear layers;

* Linear layers are used for projection; convolutional layers are, according to the paper discussing DQN, “allows the model to directly act on RGB images…”, which helps identifying the appropriate state expression.
* The specific implementation can refer to this page (as an effective example): <https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html>
* several points worth noticing:

1: idea of exploration-exploitation trade-off and choosing action:

2: idea of updating model based on loss function; “estimate” gives current state’s actions info, “target” gives next state’s action info, followed from Bellman equation

##### Probability Distributions

<https://pytorch.org/docs/stable/distributions.html>

* Should follow instructions in package: “torch.distributions”
* One point to realize:

if input parameters are tensors (for mean and std), then the shape of mean and std should be the same, and each corresponding position’s mean and std value will form an independent distribution.

###### Common methods:

Class torch.distributions.distribution.Distribution()

Parent class of all distributions