# CS323 Project 3: Sequence Modeling

**Author:** Modar Alfadly, Guocheng Qian, and Shuming Liu

In [ ]:
```
# Student Name:
# KAUST ID:
# Degree:
# Major:
```

To setup a conda environment for this project, just run the following commands:

```
source $(conda info --base)/etc/profile.d/conda.sh
conda create -n cs323 python=3.9.2 -y
conda activate cs323

conda install pytorch=1.8.0 torchvision=0.9.0 torchaudio=0.8.0
cudatoolkit=11.1 -c pytorch -c conda-forge -y
conda install av=8.0.3 ffmpeg=4.3.1 -c conda-forge -y  # PyAV for
video processing
conda install h5py=2.10.0 -y  # H5Py for processing HDF5 files
conda install pandas=1.2.3 -y  # Pandas for tabular data
manipulation
conda install jupyter=1.0.0 -y  # to edit this file
conda install matplotlib=3.3.4 -y  # for plotting
conda install tqdm=4.59.0 -y  # for a nice progress bar

pip install jupyter_http_over_ws  # for Google Colab
jupyter serverextension enable --py jupyter_http_over_ws  # Google
Colab
```

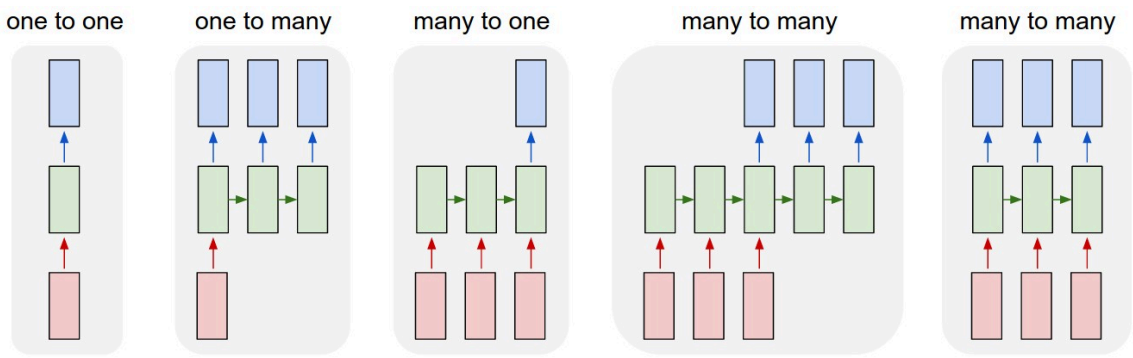
In the previous projects, you learned the basics of building, training, and evaluating deep learning models. In this project, we will pay some attention to handling data. Specifically, sequential data such as natural language (sequences of characters) and videos (sequences of images). The main challenge with this type of data is that input and output sequence lengths are variable. Previously, we more or less expected the input and output to always have fixed shapes. However, we did tackle one type of variability which was on the spatial dimensions. With fully convolutional networks, we are able to process images with virtually any size. In videos, one can think of extrapolating the idea of 2D convolutions to 3D where the kernels move along the temporal dimension as well. Although that works in theory, training such monstrosity can easily become infeasible. This was a special case and we need a better tool to efficiently handle similar tasks with sequential nature.

Take for example the language translation task; the input is a sentence in language $A$ and the output is a sentence in language $B$. We notice that the number of words in the translated sentence doesn't need to be the same as the original sentence since languages

have different characteristics. Moreover, not all sentences have a fixed length even in the same language. If we want to use what we learned so far, we will assume a maximum sequence length and appropriately apply padding and masking as we did with bounding boxes in object detection.

*Note:* you might be wondering how can we pass words to a neural network that accepts vectors of numbers not strings? A simple approach is to define a vocabulary (list of words) and represent each word with one-hot encoding or an embedding vector that has nice properties (e.g., close words semantically should be close in the embedding space). Such embedding can be learned using large corpus of text.

In [ ]:
```
import math
import datetime
from pathlib import Path

from IPython import display
from tqdm.notebook import tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, IterableDataset

import torchvision
import torchvision.transforms as T
from torchvision import models
```




In sequential data, we will face various types of tasks. So far, we are familiar with one-to-one tasks. For example, image classification; the input (the bottom red block in the above figure) is a single image that is fed to the network (middle green block) and it spits a single output for the logits (top blue block). Although the logits are for multiple classes, we still consider them as one because it is the same for all possible inputs.

Examples of sequential data types:

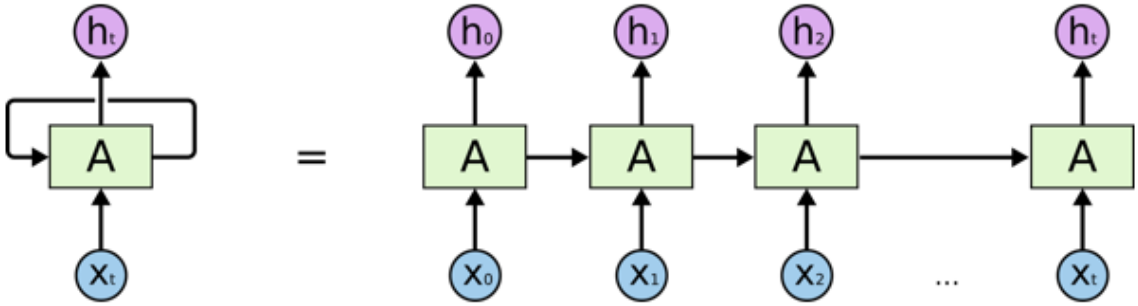
- One-to-many: image captioning where the input is a single image and the output is a sentence describing the image

- Many-to-one: text-to-image generation where the input is a sentence describing a scene and the output is an image
- Many-to-many: language translation where the input and output sequnces don't necessarily have the same length
- Many-to-many (same length): noise cancelling in audio where the input and output have the same length

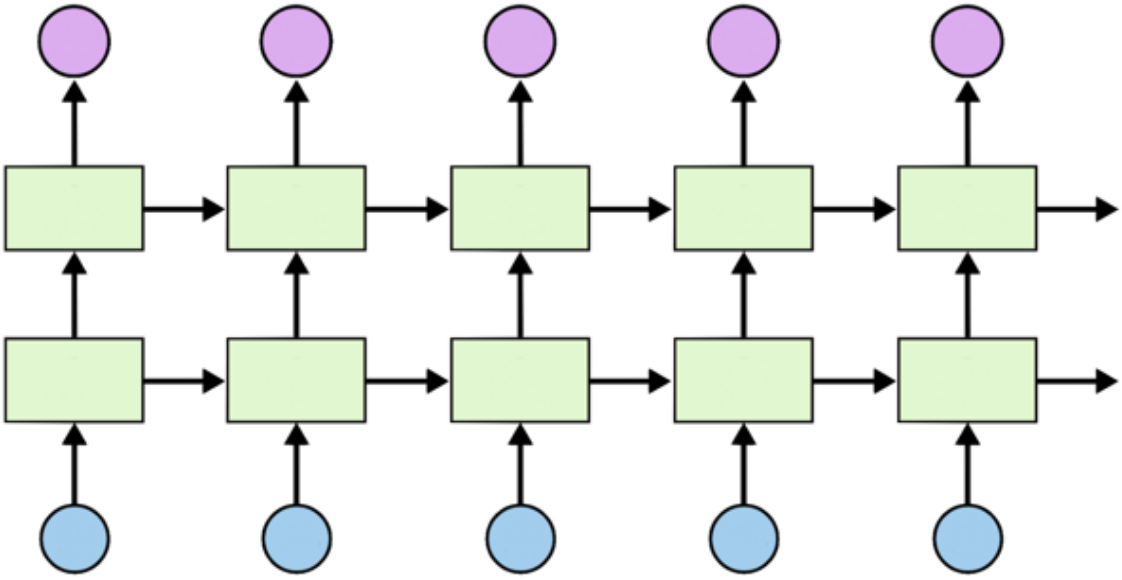
Before you start this project, you are highly encouraged to watch this introductory video and go through this notebook.

# Part 1: Recurrent Neural Networks (4 points)

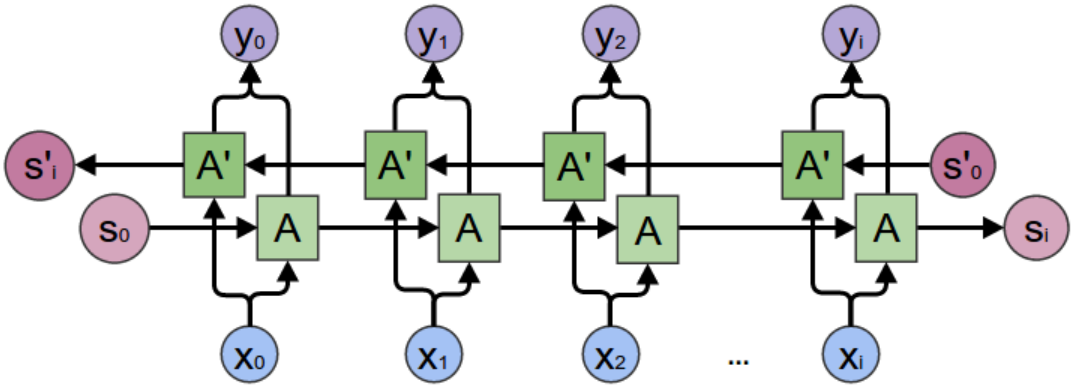
We cannot just simply train models for the maximum sequence length as they can be quite prohibitive. Instead, we will train a single recurrent cell; a single network that we apply on every element of the sequence. However, to incorporate the sequence, we will force the cell to accept a second input (other than sequence elements) we will call it the hidden state (the "memory" of the network if you will) and outputs an updated state. The hidden state acts as a feedback loop which is represented by the side arrows between the unrolled recurrent cells (green blocks). The first hidden state is initialized with zeros. Then, it gets updated by running through the sequence elements.




This seamingly simple idea is quite effective for such tasks. We can easily make this network deep by stacking RNNs cells.



In some use cases, it might be beneficial to use information from future elements to get the desired output for current elements in the sequence. This manifests a lot in natural language were sometimes you need to wait until the end to understand the sentence. For example, to translate `spider man` to spanish `el hombre araña`, man `hombre` comes before spider `araña` and the netwok cannot translate spider first. Bidirectional RNNs try to alleviate this by stacking a second RNN cell working on the reverse direction.

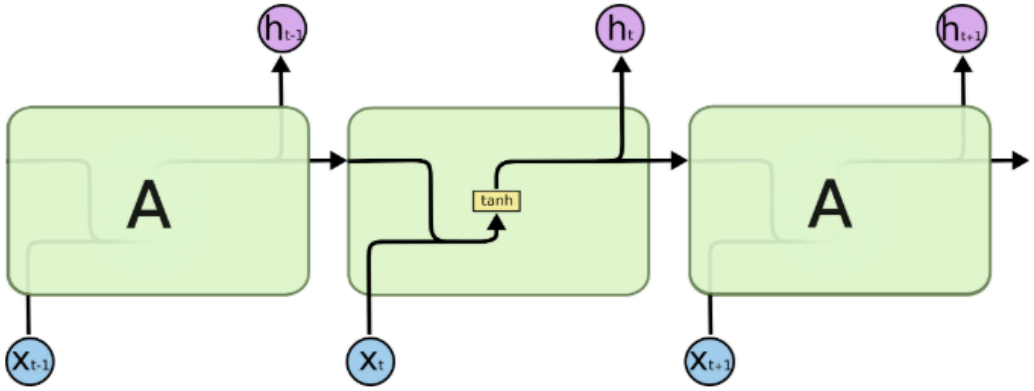



## RNN

The only remaining thing that we didn't discuss is what goes inside this recurrent cell. The most basic version is the Elman RNN. It applies a linear layer on the input and another linear layer on the hidden state and sum their results then apply an activation function; $\tanh$ in this case.

```
input_to_hidden = nn.Linear(input_size, hidden_size)
hidden_to_hidden = nn.Linear(hidden_size, hidden_size)

new_hidden = torch.tanh(input_to_hidden(inputs) +
hidden_to_hidden(hidden))
```

$$h_t = \tanh((\mathbf{W}_{ih}\mathbf{x}_t + \mathbf{b}_{ih}) + (\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_{hh})) \qquad \text{(hidden state)} \qquad (1)$$




## LSTM

The community came up with better recurrent cells since then. Most notably, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). For LSTMs, the intuition was to add "memory" features; forgetting irrelevant information $\rightarrow$ storing new information $\rightarrow$ updating the state.

$$i_t = \sigma\left((\mathbf{W}_{ii}\mathbf{x}_t + \mathbf{b}_{ii}) + (\mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_{hi})\right) \qquad \text{(input gate)} \qquad (2)$$

$$f_t = \sigma\left((\mathbf{W}_{if}\mathbf{x}_t + \mathbf{b}_{if}) + (\mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_{hf})\right) \qquad \text{(forget gate)} \qquad (3)$$

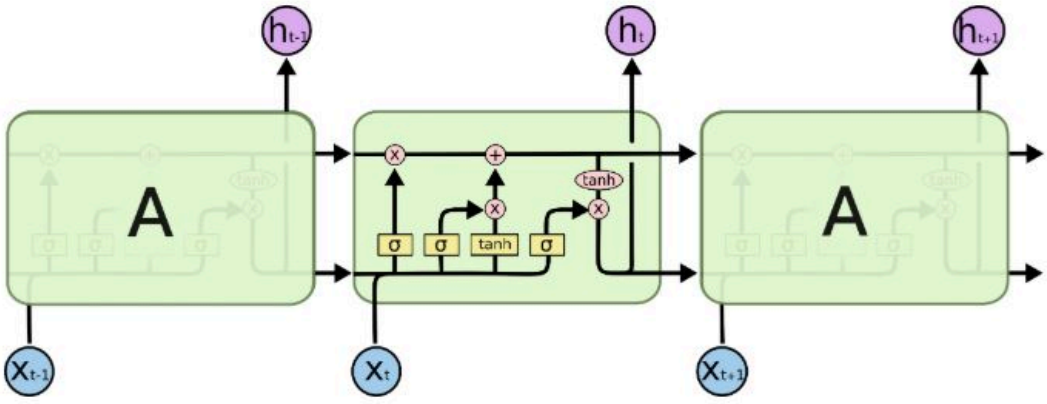
$$g_t = \tanh\left((\mathbf{W}_{ig}\mathbf{x}_t + \mathbf{b}_{ig}) + (\mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_{hg})\right) \qquad \text{(cell gate)} \qquad (4)$$

$$o_t = \sigma\left((\mathbf{W}_{io}\mathbf{x}_t + \mathbf{b}_{io}) + (\mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_{ho})\right) \qquad \text{(output gate)} \qquad (5)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \qquad \text{(cell state)} \qquad (6)$$

$$h_t = o_t \odot \tanh(c_t) \qquad \text{(hidden state)} \qquad (7)$$

Watch this video or read this blog for a detailed explanation of these components.

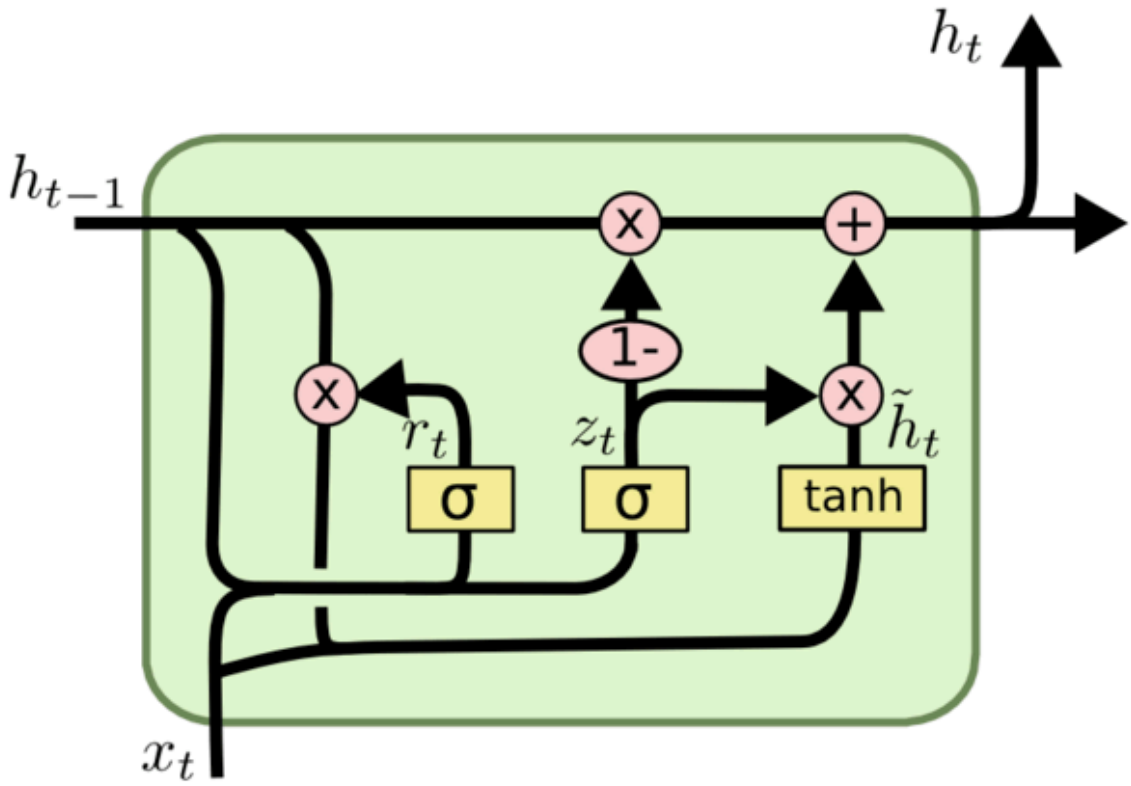



The repeating module in an LSTM contains four interacting layers.

## GRU

Notice how LSTMs add another state (in addition to the hidden state) called the cell state (top side link in the above figure). It helps the backpropagation step to handle long sequences as it offers a nice flow of gradients. GRUs tried to improve on this using simpler gates.

$$r_t = \sigma\left((\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir}) + (\mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr})\right) \quad \text{(reset gate)} \quad (8)$$
$$z_t = \sigma\left((\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz}) + (\mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz})\right) \quad \text{(update gate)} \quad (9)$$
$$n_t = \tanh((\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in}) + r_t \odot (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})) \quad \text{(new gate)} \quad (10)$$
$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{t-1} \quad \text{(hidden state)} \quad (11)$$




GRUs are computationally more efficient and they give similar performance to LSTMs. This is why we will primarily use them in this project. Luckily, all these models are implemented in PyTorch and you can use them directly. It also supports stacking and bidirectional RNNs. Here is an example that uses a GRU for a many-to-one task (i.e., given an input sequence, predict the class).

In [ ]:
```
# many-to-one GRU model
class GRUClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes, num_layers=1, b
        super().__init__()
        self.gru = nn.GRU(input_size, hidden_size, num_layers, bidirectional
        self.output = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # last_hidden_state is given as output for convenience
        all_hidden_states, last_hidden_state = self.gru(x)
```

```
        return self.output(last_hidden_state[-1])


def test():
    batch_size = 2
    max_sequence_length = 10
    input_size = 3
    hidden_size = 50
    num_classes = 7

    inputs = torch.randn(batch_size, max_sequence_length, input_size)
    net = GRUClassifier(input_size, hidden_size, num_classes)
    print('Output shape:', net(inputs).shape)


test()
```

# Task 1.1: Many-to-Many Sequential Task (1 points)

In [ ]:

```
# TODO: many-to-many GRU model
class GRUTranslation(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes, num_layers=1, b
        super().__init__()
        self.gru = nn.GRU(input_size, hidden_size, num_layers, bidirectional
        self.output = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # TODO: how can we implement a many-to-many sequential task (same in
        ...
        return


def test():
    batch_size = 2
    max_sequence_length = 10
    input_size = 3
    hidden_size = 50
    num_classes = 7

    inputs = torch.randn(batch_size, max_sequence_length, input_size)
    net = GRUTranslation(input_size, hidden_size, num_classes)
    print('Output shape:', net(inputs).shape)

test()
```

Notice, how now the input has an extra dimension which is the sequence length `(seq_length, batch, features)`. Also, the batch dimension comes second after the sequence length. This is inconsistent to the convention that we are used to but it is more efficient under the hood. Nevertheless, the other option is also available using the `batch_first` argument. Another thing you should notice here is that `nn.GRU` uses a number of `nn.GRUCell`'s according to the `num_layers` and the `bidirectional` arguments. Let's reimplement a cell to check our understanding.

## Task 1.2: Forward Pass (3 points)

In [ ]:
```
batch_size = 2
net = nn.GRUCell(7, 5)
net.eval()
x = torch.randn(batch_size, net.input_size)
h = torch.randn(batch_size, net.hidden_size)

w_ih, b_ih = net.weight_ih, net.bias_ih
w_hh, b_hh = net.weight_hh, net.bias_hh
print('w_ih:', w_ih.shape)
print('w_hh:', w_hh.shape)

# TODO: write the forward pass by your own using the weights from net (2 poi
# hint 1:
# internally, PyTorch combines all three linear layers of GRU
# that act on the input as a single linear layer and does the same
# for the hidden state layers given by w_ih and w_hh, respectively.

# hint 2:
# do the forward pass using F.linear() then split the output in the
# correct order and make sure that the output of net matches your h_t
r_i, z_i, n_i = ...
r_h, z_h, n_h = ...

# hint 3: follow the equations of GRUCell
r_t = ...
z_t = ...
n_t = ...
h_t = (1 - z_t) * n_t + z_t * h

# Make sure returns True
print(torch.allclose(h_t, net(x, h)))
```

# Part 2: Revisit CIFAR-10 Image Classification (2.5 points)

We have worked on CIFAR-10 in our first project, where we classify images using Conv2D. In this project, we revist CIFAR-10, this time we view an image as $16 \times 16$ patches and embed each patch to a vector $\mathbb{R}^C$. These embedded patches are called tokens. In this way, the

classical image classification problem can be converted to a Many-to-One sequence modeling problem, where the input is the 256 tokens (257 tokens if we consider the class token) and the output should be the predicted logits.

In [ ]:
```
from torch.utils.data import random_split
from torchvision import datasets


root_dir = Path(torch.hub.get_dir()) / f'datasets/CIFAR10'
normalize = T.Normalize(
    mean=(0.4915, 0.4823, 0.4468),
    std=(0.2470, 0.2435, 0.2616),
)

train_set = datasets.CIFAR10(
    root_dir,
    train=True,
    download=True,
    transform=T.Compose([
        T.ToTensor(),
        normalize,
    ]),
)
test_set = datasets.CIFAR10(
    root_dir,
    train=False,
    download=True,
    transform=T.Compose([
        T.ToTensor(),
        normalize,
    ]),
)

print(train_set)
print(test_set)

batch_size = 32
trainloader = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

In [ ]:
```
# understand the data
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # simple unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

```
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
# print shape one image
# understand the data first.
img, lable = train_set.__getitem__(0)
print(img.shape, lable)
```

# Task 2.1: Patch Embedding (1 points)

In [ ]:
```
# TODO: Patch embedding.
# Given CIFAR-10 image, convert it into 16x16 non-overlap patches (tokens) u
# hint 1: The CIFAR-10 images are with shape 32x32
# hint 2: Reference, the ViT implementation here: https://github.com/rwightm
class PatchEmbed(nn.Module):
  def __init__(self, in_channels, embed_dim):
    super().__init__()
    self.proj = nn.Sequential(
      ... , # todo: replace ... with 1 conv2d layer
      nn.Flatten(2),
      )
    self.norm = nn.LayerNorm(embed_dim)

  def forward(self, x):
    return self.norm(self.proj(x).transpose(1, 2))
# test.
embed_dim = 64
patch_embed = PatchEmbed(3, embed_dim)
tokens = patch_embed(images)
print(images.shape, tokens.shape)
```

# Task 2.2: Training a GRU-based Classifier (1.5 points)

## TODOs:

- Copy the training and evaluation code from Project 2.
- Print the confusion matrix for training and testing at every epoch (0.5 points)
- Write a GRU-based image classifier. Hint: GRUClassifier + PatchEmbed (0.5 points)
- Reach over 80% validation accuracy (0.5 points)

In [ ]:
```
# TODO: training.
# Hint: we have worked on Image Classification before, you can simply copy t
```

```
# Do evaluation after training. The expected accuracy on validation set shou
```

# Part 3: Transformers (3.5 points)

RNNs introduced the idea of a hidden state that is zero-initialized in the beginning and it gets updated as we process each element of the sequence in order. The problem with this, is that we need to wait for all previous elements to be processed first at every time step. An alternative approach was proposed in 2017 and was named the Transformer model. It outperformed all recurrent models with impressive results at a memory and computational cost. Furthermore, new transformers were adapted to tackle different tasks such as image (ViT) and video (VTN) classification, object detection (DETR), semantic segmentation (SETR), and language generation (GPT-3). It has definitly become a main stream deep learning model showing a great potential not as Capsule Networks. To alleviate the computational cost of transfomers, many models were proposed such as Linformer and Longformer. It has a growing research community and many imporvements are yet to come.

Before you continue, you are highly encouraged to watch this video and/or read these notes explaining the first transformer model.

## Task 3.1: Get familar with Transformers (1.5 points)

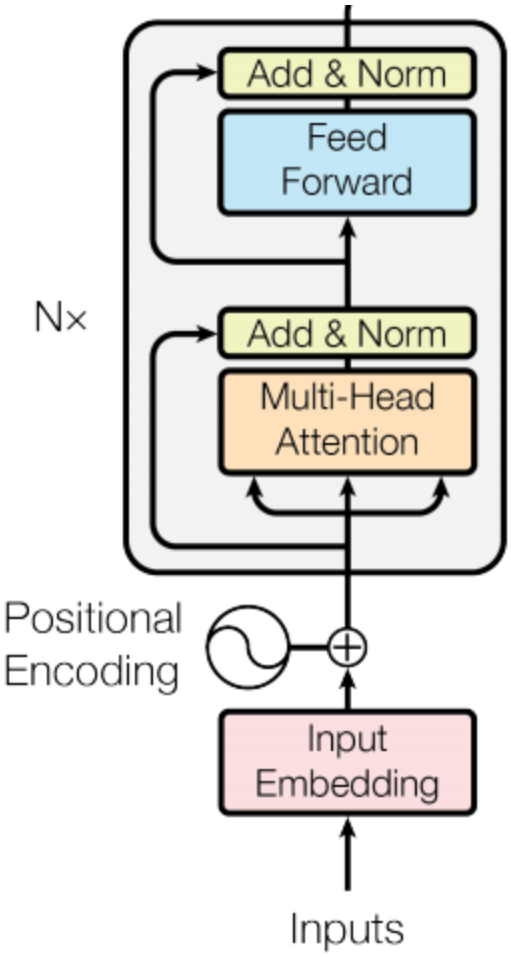
In [ ]:
```
batch_size = 4
max_seq_len = 100
feature_size = 64
x = torch.randn(batch_size, max_seq_len, feature_size)

# directly import transformer layer
net = nn.TransformerEncoderLayer(
    d_model=feature_size,
    nhead=4,  # must divides feature_size
    dim_feedforward=3,  # hidden size,
    batch_first=True
)
rnn = nn.GRU(feature_size, hidden_size=feature_size, batch_first=True)

print('Input shape :', x.shape)
```

```
print('Output shape:', net(x).shape)
print('RNN\'s shape :', rnn(x)[0].shape)  # all_states
```

As you can see from the above code, the basic building block of transformers is the nn.TransformerEncoderLayer. You can think of it as an equivalent to a recurrent cell (e.g., GRU) unrolling on the input sequence. The main difference is that the transformer layer is processing the entire sequence in parallel while the recurrent cell is doing it sequentially. The gotcha here is that the transformer layer doesn't know anything about the positions of the sequence elements. If you shuffle the input sequence order, the corresponding output will still be the same. This property is called permutation equivariance. This is definitely not the case with RNNs.

In [ ]:
```
shuffle = torch.randperm(max_seq_len)
reorder = torch.empty_like(shuffle)
reorder[shuffle] = torch.arange(max_seq_len)
print(torch.allclose(x[:, shuffle][:, reorder], x))  # True
print(shuffle.shape)

# to avoid the possible numerical issue
x = x.double()
net = net.double()
rnn = rnn.double()
net.eval()
rnn.eval()

# TODO: vvvvvvvvvvv (0.5 points)
# verify that transformer is permutation equivariant
# also, check that it does matter for RNNs (use the defined net and rnn)
# take the above assertion as a hint
print(torch.allclose(..., ...))  # expect True for transformer
print(torch.allclose(..., ...))  # expect False for RNN
# ^^^^^^^^^^^^^^^^^
```

To remedy this caveat, we can apply a possitional encoding (i.e., add information about the position to every sequence element). A naive way of doing this is by adding the position index value to the features (i.e., add $i$ to all elements at position $i$ in the sequence). A more clever approach was proposed in the original paper which uses sine and cosine functions with different frequencies. See an implementation here.




The above figure shows what is inside our transformer layer. It has a multi-head self-attention layer (the most important part) and two residual connections with layer norm in between. The "Nx" on the left simply represents stacking multiple transformer layers (nn.TransformerEncoder). Nothing is new so far to us except for the attention block. In its simplest forms it looks like this (without any loss of generality and for ease of notation, assuming the input $\mathbf{x} \in \mathbb{R}^{N \times 1}$ is a sequence of length $N$ with a feature size of $1$):

$$\text{Attention}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \text{softmax}\left(\frac{\mathbf{q}\mathbf{k}^\top}{\sqrt{N}}\right)\mathbf{v} \tag{12}$$

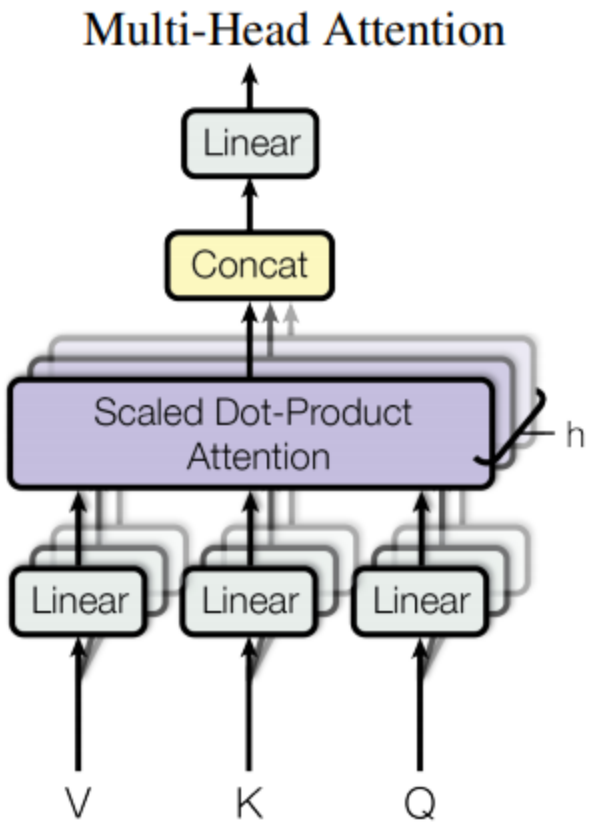
$$\text{Self-Attention}(\mathbf{x}) = \text{Attention}(\mathbf{W}_q\mathbf{x} + \mathbf{b}_q, \mathbf{W}_k\mathbf{x} + \mathbf{b}_k, \mathbf{W}_v\mathbf{x} + \mathbf{b}_v) \tag{13}$$

*Note*: the attention weights matrix ($\text{softmax}\left(\frac{\mathbf{q}\mathbf{k}^\top}{\sqrt{N}}\right) \in \mathbb{R}^{N \times N}$) is quadratic in size with respect to the sequence length.

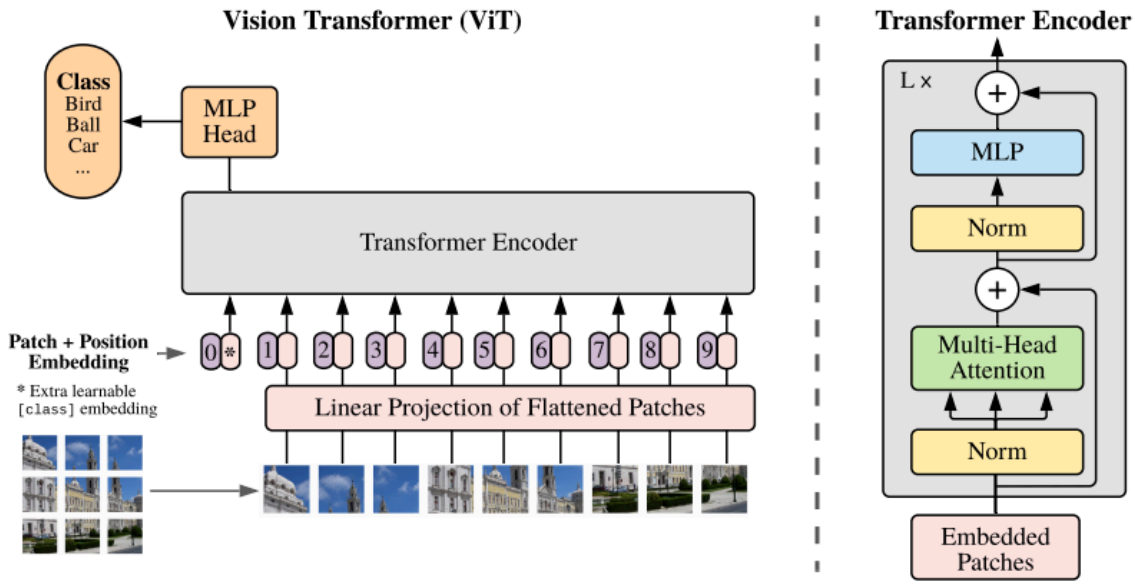
In [ ]:
```
x = torch.randn(batch_size, max_seq_len, feature_size)

# initialize the query, key, and value mapping
query = nn.Linear(feature_size, feature_size)
key = nn.Linear(feature_size, feature_size)
value = nn.Linear(feature_size, feature_size)
```

```
# TODO: implement the scaled dot-product attention as described in the equat
attn = ... # calculate attn weight
out = ... # perform self-attention

print('Input shape :', x.shape)
print('Output shape:', out.shape)
```

A multi-head attention is just a multiple attention models applied in parallel and their results are concatenated and followed by a linear layer. Purely for efficiency, the authors decided to reduce the number of output features for `query`, `key`, and `value` to be exactly the feature size divided by the number of heads. Therefore, the number of heads ($8$ in the paper) must divide the feature size ($512$ in the paper).




In a nutshell, attention is just a way to weigh the effect of every sequence element on every other sequence element. In vision transfomer (ViT) for example, the input image is cropped into 16x16 pixel patches that are passed to the network as a sequence (every patch is a sequence element with the patch position in the original image as the positional encoding). With this, every patch can attend to every other patch starting from the first layer. The receptive field is the entire image and the attention weight matrix intuitively represents how important every patch is to every other patch to classify the image (recall the size is of the attention matrix is $N \times N$).




# Task 3.2: Build A ViT Model for Image Classification (2 points)

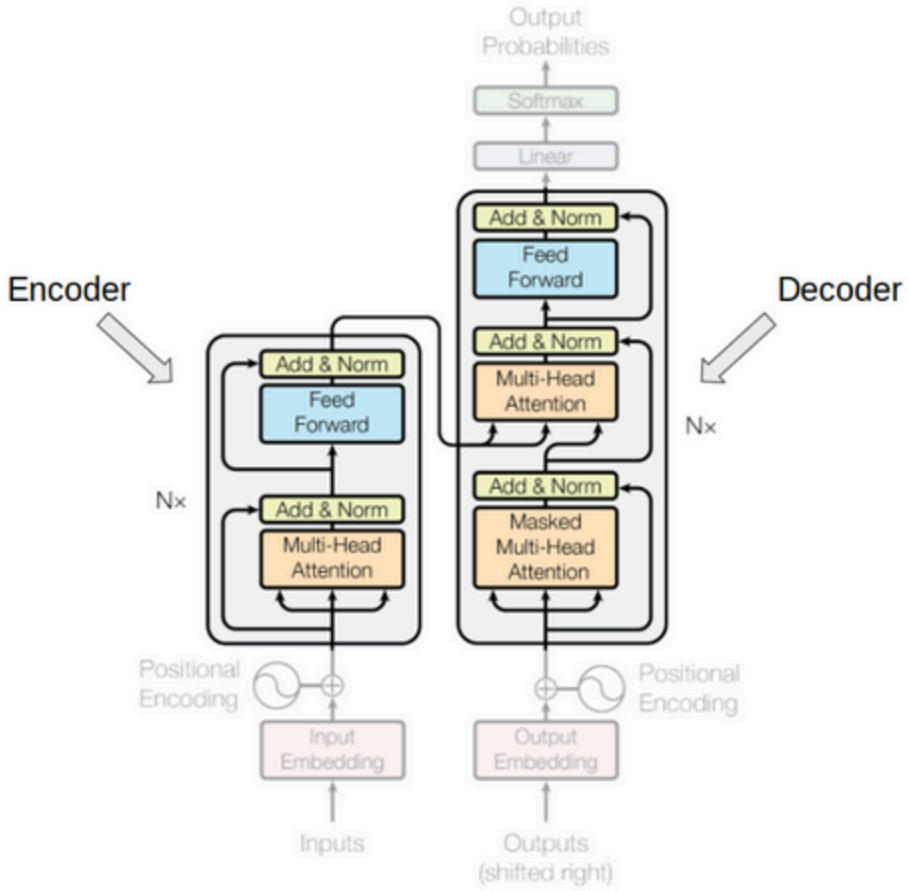
Similarly, we now can train an image classifier using Transformers. In this project, we focus on the ViT architecture from paper An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. We only need to stack one patch embedding layer to embed the image as 256 tokens, followed by multiple transformer layers with a positional encoding on the input tokens in the middle, and a classifier head at the end. We can train it exactly as we did with RNNs. We do **NOT require training the ViT** at this time, since we want the students to put more effort into understanding the essences of RNNs and Transformers, and get a touch of the recent hot topic, vision transformers.

In [ ]:
```
# TODO: vvvvvvvvvvv (2 points)
# build ViT

class ViT(nn.Module):
  def __init__(self, in_channels=3, num_classes=10, embed_dim=64, num_patche
    super().__init__()

    # step 1, build a patch embedding layer
    self.patch_embed = ...
    self.cls_token = nn.Parameter(torch.randn(1, 1, embed_dim))

    # step 2, positional encoding layer. We use a simple learnable positiona
    self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim)

    # step 3, build a three-layer four-head TransformerEncoder using nn.Tran
    encoder_layer = ...
    self.encoder = ...

    # step 4, build a linear classifier
```

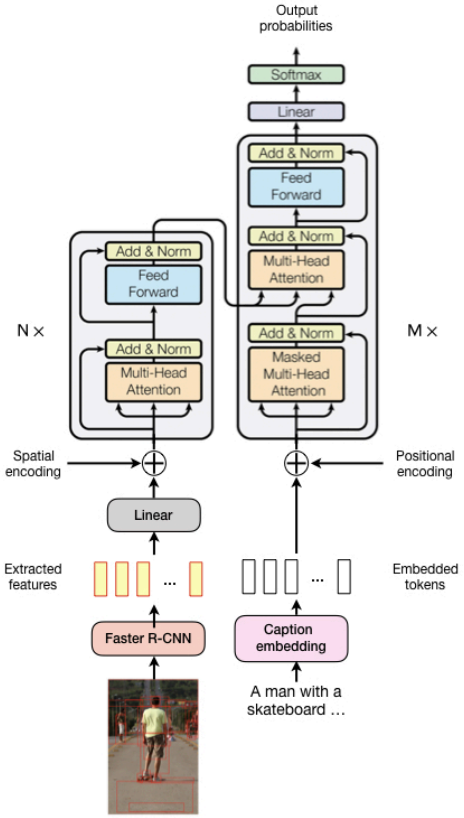
```
    self.head = ...

    # final normalization layer
    self.norm = nn.LayerNorm(embed_dim)

  def forward(self, x):
    # forward pass here.
    x = self.patch_embed(x)
    x = torch.cat((self.cls_token.expand(x.shape[0], -1, -1), x), dim=1)
    # TODO: pass x into transformers
    ...
    return self.head(x[:, 0, :]) # why only use x[:, 0, :]

# test code.
x = torch.randn(batch_size, 3, 32, 32).cuda()
print('Input: ', x.shape)
vit = ViT().cuda()
print('Output:', vit(x).shape)
# ^^^^^^^^^^^^^^^^^
```

# Part 4: Further Reading (Optional, 0 Point)

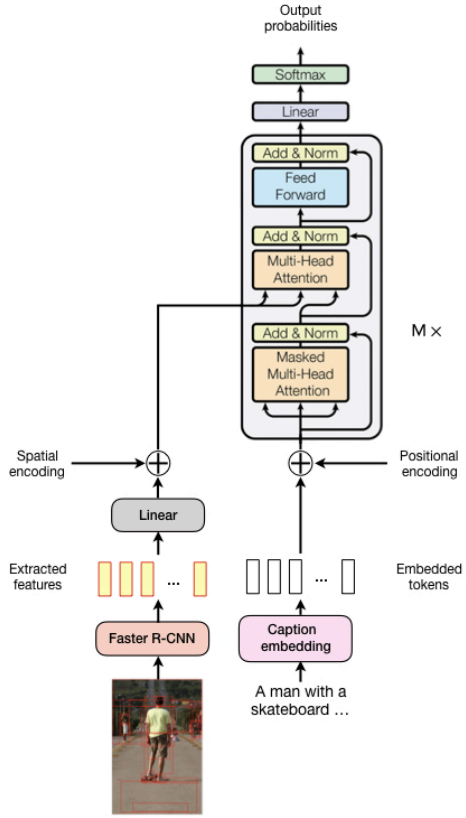
Transformers work with any type of sequential data tasks. The exception is many-to-many with different sequence lengths between the input and the output as it needs extra care, e.g. image semantic **segmentation**. To do this, the full transformer model implements an encoder and a **decoder**.




The final output of the encoder is considered as the memory which gets passed to every decoder block as `key` and `value` while `query` comes from the output. The last difference in the decoder that we need to point out is the masking of the attention weight matrix. Since the order doesn't matter in transformers and we should not attend to future output elements, a square subsequent mask needs to be applied. The entire transformer model with all its layers is encapsulated in nn.Transformer. It can be readily instantiated and trained.




Default baseline architecture




Simplistic baseline architecture

If you are interested to learn more about transformers, read this survey paper and checkout these applications:

- Language: BERT, GPT-3, DeBERTa
- Images: IT, ViT, IPT, BoTNet, PVT, iGPT, DETR, SETR
- Video: VTN, TimeSformer
- Point Clouds: CT, PT, PCT