Lab 5

Chaojie Zhang

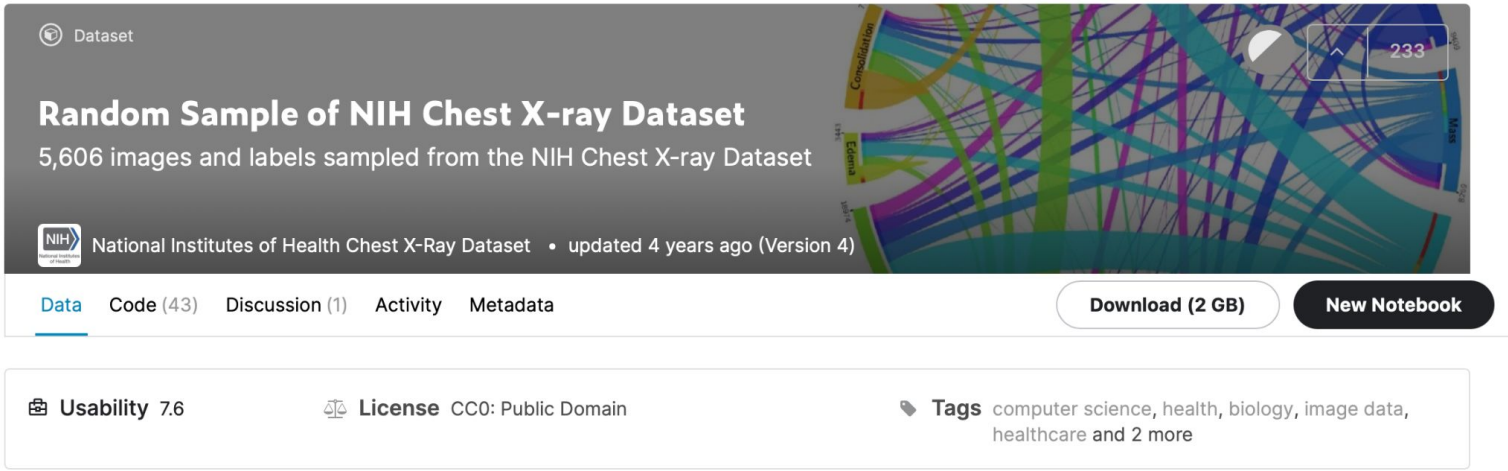
- HPC workshop 2 next week
- Start HW2 early! I will be in the Q&A session next Tuesday.
- Modification of HW2 (check campuswire)
- Additional reading (not for HW)
- Lab5 can be ran on Kaggle, I will post the instructions for running lab5 on Greene

Lab5 Jupyter Notebook

You can run the notebook in Kaggle:
https://www.kaggle.com/nih-chest-xrays/sample

Change the paths from the notebook, and use cpu to train.

Metadata path: /kaggle/input/sample/sample_labels.csv



Dataset

```
label_df = pd.read_csv('/kaggle/input/sample/sample_labels.csv').iloc[:, :2]
label_df
```

	Image Index	Finding Labels
0	00000013_005.png	Emphysema|Infiltration|Pleural_Thickening|Pneu...
1	00000013_026.png	Cardiomegaly|Emphysema
2	00000017_001.png	No Finding
3	00000030_001.png	Atelectasis
4	00000032_001.png	Cardiomegaly|Edema|Effusion
...	...	...
5601	00030712_000.png	No Finding
5602	00030786_005.png	Cardiomegaly|Effusion|Emphysema
5603	00030789_000.png	Infiltration
5604	00030792_000.png	No Finding
5605	00030797_000.png	No Finding

5606 rows × 2 columns

Remove ‘No Finding’ samples

Use only part of the samples (600 train, 200 val, 200 test)

```
label_df['Disease']=(label_df['Finding Labels'] != 'No Finding').astype(int)
print(label_df.head())
num_rows = 1000
label_df = label_df.iloc[:num_rows,:]

# define train, val and test idx
idx = np.arange(num_rows)
np.random.shuffle(idx)
train_size = 600
val_size = 200
test_size = 200
train_idx = idx[:train_size]
val_idx = idx[train_size:train_size+val_size]
test_idx = idx[train_size+val_size:]

# get train, val and test dataframes
train_df = label_df.iloc[train_idx,:]
val_df = label_df.iloc[val_idx,:]
test_df = label_df.iloc[test_idx,:]

# save the dataframes
train_df.to_csv('train.csv', index = False)
val_df.to_csv('val.csv', index = False)
test_df.to_csv('test.csv', index = False)
```

Dataloader

```
class Xray_dataset(Dataset):
    '''X-ray Dataset'''
    def __init__(self, df_path, train = False):
        self.df = pd.read_csv(df_path)
        self.train = train

    def __len__(self):
        return len(self.df)

    def __getitem__(self,idx):
        file_name = self.df.iloc[idx,0]
        label = self.df.iloc[idx,-1]
        img = Image.open('/kaggle/input/sample/sample/images/'+file_name)
        img = img.resize((512, 512)) # resize the image, the original size is 1024x1024
        if self.train:
            # rotate the image (data augementation)
            rand_num = np.random.random()
            if rand_num > 0.7:
                rot_angle = np.random.uniform(low = -10, high = 10)
                img = img.rotate(rot_angle)
        # Normalization
        img = np.asarray(img)
        min_image = np.min(img)
        max_image = np.max(img)
        img = (img - min_image)/(max_image - min_image + 1e-4)
        img = torch.tensor(img).unsqueeze(0).float()
        label = torch.tensor(label).long()
        if img.dim() != 3:
            img = img[:,:,:,0]
        return img, label
```

Load samples from the dataloader

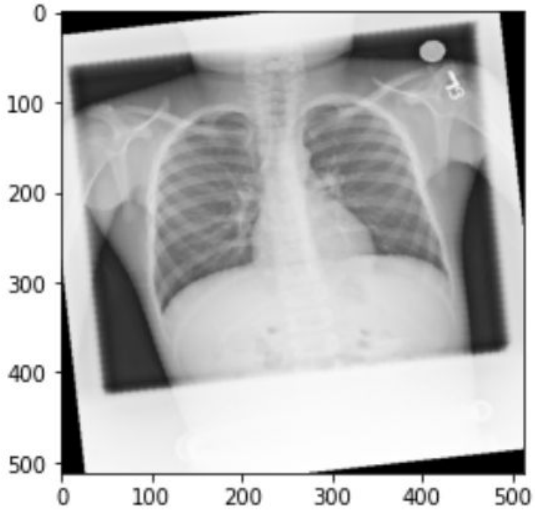
```
train_df_path = 'train.csv'
val_df_path = 'val.csv'
test_df_path = 'test.csv'
transformed_dataset = {'train': Xray_dataset(train_df_path, train = True),
                       'validate':Xray_dataset(val_df_path),
                       'test':Xray_dataset(test_df_path),
                                           }
bs = 4
dataloader = {x: DataLoader(transformed_dataset[x], batch_size=bs,
                        shuffle=True, num_workers=0) for x in ['train', 'validate','test']}
data_sizes ={x: len(transformed_dataset[x]) for x in ['train', 'validate','test']}
```

```
iterator = iter(dataloader['train']) # create an iterator
sample = next(iterator) # load samples from the iterator
print(len(sample))
print(sample[0].size()) # images (batch size, gray scale image, height, weight)
print(sample[1]) # label
```

```
2
torch.Size([4, 1, 512, 512])
tensor([0, 1, 0, 0])
```

Sample

```
sample_img = sample[0][1].squeeze().numpy()
plt.imshow(sample_img, cmap = 'gray')
plt.show()
```

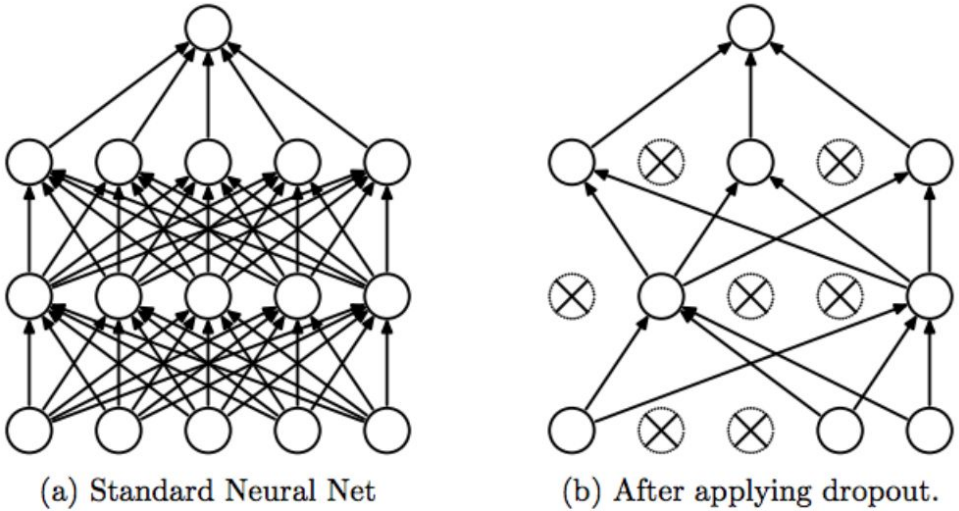


Training Function

```
if p == 'train':
    model.train()
else:
    model.eval()
```

model.train() tells your model that you are training the model. So effectively layers like dropout, batchnorm etc. which behave different on the train and test procedures know what is going on and hence can behave accordingly.
https://stackoverflow.com/questions/51433378/what-does-model-train-do-in-pytorch

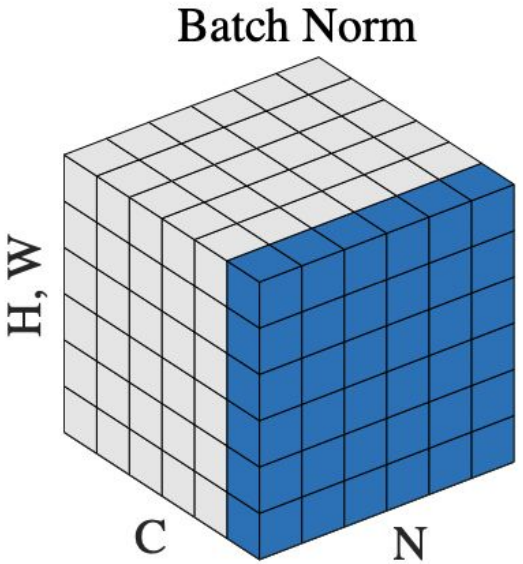
Dropout: https://towardsdatascience.com/machine-learning-part-20-dropout-keras-layers-explained-8c9f6dc4c9ab



(a) Standard Neural Net

(b) After applying dropout.

Batch Normalization: Normalize among N (batch), H, W, not including C (channel)



Training Function

```
for data in dataloader[p]:
    optimizer.zero_grad()
```

For every batch during the training phase, we typically want to explicitly set the gradients to zero before starting to do backpropragation because PyTorch accumulates the gradients on subsequent backward passes.
(Gradients accumulation works in RNN.)

```
if p== 'train':
    loss.backward()
    optimizer.step()
```

`loss.backward()` computes `dloss/dx` for every parameter `x` which has `requires_grad=True`. These are accumulated into `x.grad` for every parameter `x`. In pseudo-code:

```
x.grad += dloss/dx
```

`optimizer.step` updates the value of `x` using the gradient `x.grad`. For example, the SGD optimizer performs:

```
x += -lr * x.grad
```

`optimizer.zero_grad()` clears `x.grad` for every parameter `x` in the optimizer. It's important to call this before `loss.backward()`, otherwise you'll accumulate the gradients from multiple passes.

If you have multiple losses (loss1, loss2) you can sum them and then call backwards once:

```
loss3 = loss1 + loss2
loss3.backward()
```

optimizer.step() makes the optimizer iterate over all parameters (tensors) it is supposed to update and use their internally stored grad to update their values.

https://discuss.pytorch.org/t/what-does-the-backward-function-do/9944/2

Model

```
class Conv_model(nn.Module):
    def __init__(self, kernel_size = 3):
        super(Conv_model,self).__init__()
        self.conv1 = nn.Conv2d(1,16,kernel_size, stride = 2)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(16,32,kernel_size, stride = 2)
        self.relu2 = nn.ReLU()

        self.conv3 = nn.Conv2d(32,64,kernel_size, padding = 1, stride = 3)
        self.relu3 = nn.ReLU()

        self.conv4 = nn.Conv2d(64,128,kernel_size, padding = 1, stride = 3)
        self.relu4 = nn.ReLU()

        self.conv5 = nn.Conv2d(128,256,kernel_size)
        self.relu5 = nn.ReLU()

        self.avg = nn.AdaptiveAvgPool2d(1)
        self.linear = nn.Linear(256, 2)

    def forward(self,x):
        x = self.relu1(self.conv1(x))
        x = self.relu2(self.conv2(x))
        x = self.relu3(self.conv3(x))
        x = self.relu4(self.conv4(x))
        x = self.relu5(self.conv5(x))
        x = self.avg(x)
        x = self.linear(x.view(-1,256))

        return x
```

Adding Batch Norms

```
class Conv_model_bn(nn.Module):
    def __init__(self, kernel_size = 3):
        super(Conv_model_bn,self).__init__()
        self.conv1 = nn.Conv2d(1,16,kernel_size, stride = 2)
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16,32,kernel_size, stride = 2)
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32,64,kernel_size, padding = 1, stride = 3)
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64,128,kernel_size, padding = 1, stride = 3)
        self.relu4 = nn.ReLU()
        self.bn4 = nn.BatchNorm2d(128)
        self.conv5 = nn.Conv2d(128,256,kernel_size)
        self.relu5 = nn.ReLU()
        self.bn5 = nn.BatchNorm2d(256)
        self.avg = nn.AdaptiveAvgPool2d(1)
        self.linear = nn.Linear(256, 2)

    def forward(self,x):
        x = self.relu1(self.bn1(self.conv1(x)))
        x = self.relu2(self.bn2(self.conv2(x)))
        x = self.relu3(self.bn3(self.conv3(x)))
        x = self.relu4(self.bn4(self.conv4(x)))
        x = self.relu5(self.bn5(self.conv5(x)))
        x = self.avg(x)
        x = self.linear(x.view(-1,256))

        return x
```

Without BN: acc = 0.525
With BN: acc =0.55

BN reduced the effect of overfitting.

Projects

Random Seed (reproducible)

```
def setup_seed(seed):
   torch.manual_seed(seed)
   torch.cuda.manual_seed_all(seed)
   np.random.seed(seed)
   random.seed(seed)
   torch.backends.cudnn.deterministic = True
setup_seed(0)
device = torch.device('cuda')
```

Loss

nn.BCELoss: Binary Cross Entropy Loss, you need to add a sigmoid function

nn.BCEWithLogitsLoss: loss combines a Sigmoid layer and the BCELoss

nn.CrossEntropyLoss: Softmax + Cross Entropy Loss, you don’t need to add a softmax function

nn.NLLLoss: Negative Log Likelihood Loss (multiclass cross-entropy), you need to add a softmax function

Training

```
----------Epoch10/100----------
Train set | Loss: 0.5563 | Accuracy: 72.37% | AUC: 0.7861
Test set  | Loss: 0.5099 | Accuracy: 75.76% | AUC: 0.8840 | Best AUC: 0.8840 | time elapse:  00:41:24
```

Print the loss, acc, auc in each epoch. Choose the best model (best acc/auc/loss) from the validation set.

```
state = {'best_model_wts':best_model_wts, 'model':model, \
         'train_loss':train_loss, 'train_acc':train_acc,'train_auc':train_auc,\
         'val_loss':val_loss, 'val_acc':val_acc,'val_auc':val_auc}
torch.save(state, file_name+'.pt')
```

Save your model in each epoch, including best weights and current weights.

Reading (not for the HW but important!)

A Recipe for Training Neural Networks

http://karpathy.github.io/2019/04/25/recipe/

Must Know Tips/Tricks in Deep Neural Networks

http://210.28.132.67/weixs/project/CNNTricks/CNNTricks.html

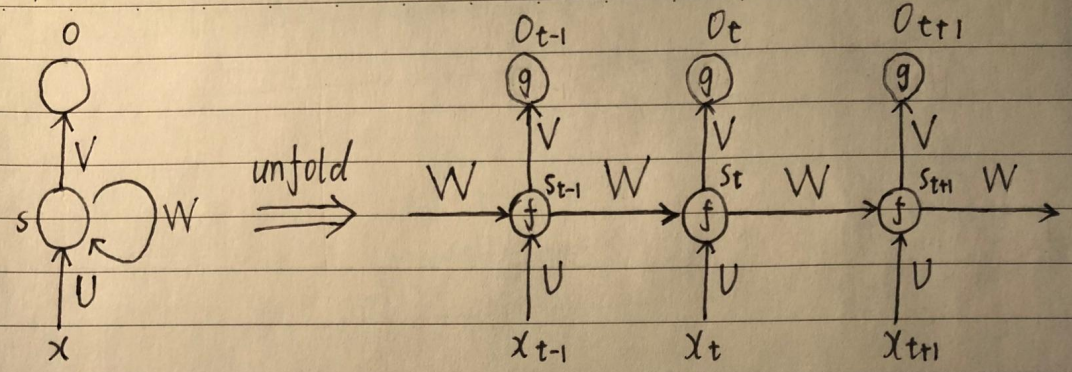
ResNet Structure

http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006

From Attention to Vision Transformer

Reference: https://github.com/wangshusen/DeepLearning

RNN



x, s, o are vectors

U, W, V are matrixes

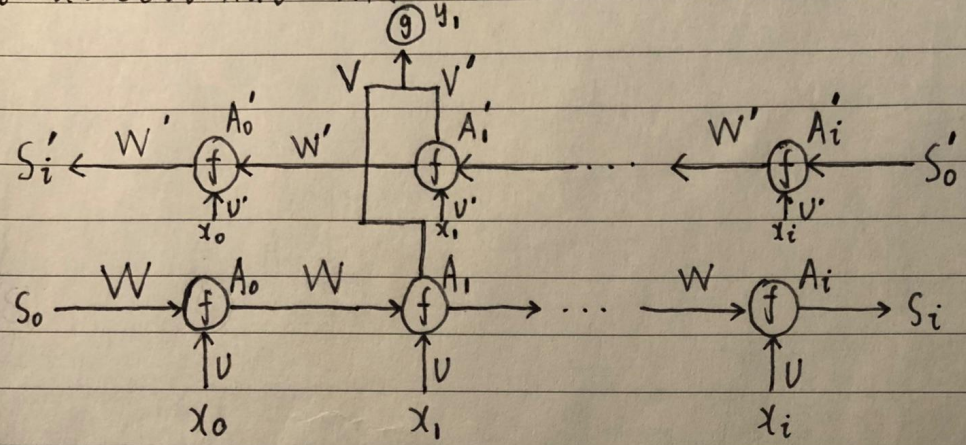
f, g are activation functions

f can be relu / sigmoid , g can be softmax

$o_t = g(V s_t)$ $\quad s_t = f(U x_t + W s_{t-1})$

$$
\begin{aligned}
o_t &= g(V s_t) \\
&= g[V f(U x_t + W s_{t-1})] \\
&= g[V f(U x_t + W f(U x_{t-1} + W s_{t-2}))] \\
&= g[V f(U x_t + W f(U x_{t-1} + W f(U x_{t-2} + \cdots)))]
\end{aligned}
$$

Bi-directional RNN

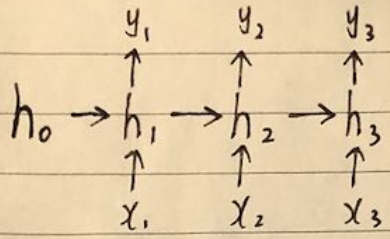


$$y_1 = g(VA_1 + V'A_1')$$
$$= g(Vf(WA_0 + Ux_1) + V'f(W'A_2' + U'x_2))$$
$$= \cdots$$

$$\begin{cases} O_t = g(Vs_t + V's_t') \\ s_t = f(Ux_t + Ws_{t-1}) \\ s_t' = f(U'x_t + W's_{t+1}') \end{cases}$$

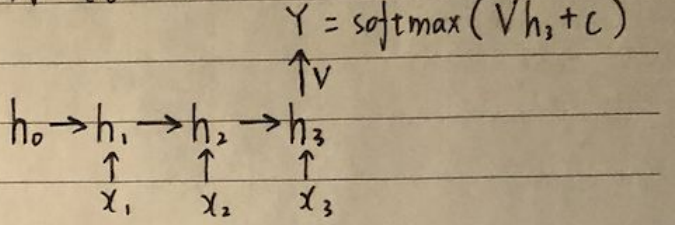
s_i and A_i are similar representation

① N to N



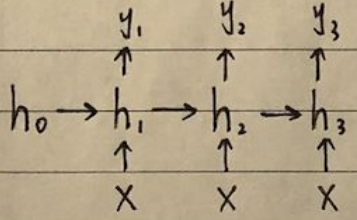
ex. predict next word of an unfinished sentence

② N to 1



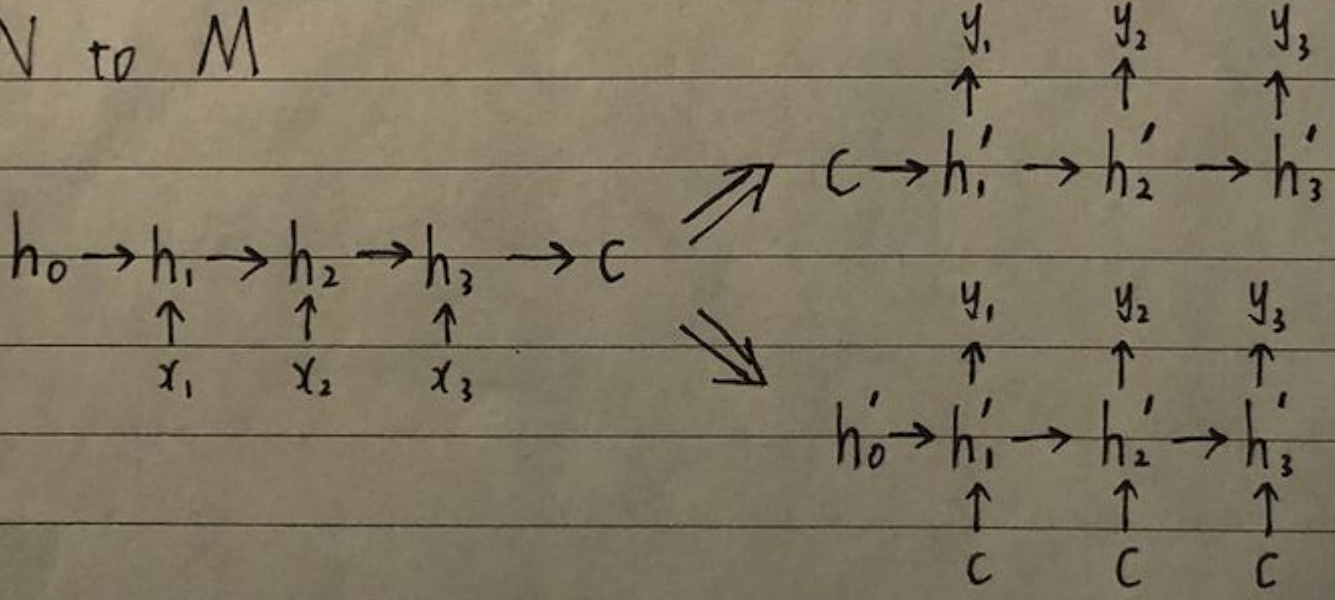
ex. sentiment prediction of a sentence

③ 1 to N



ex. image caption

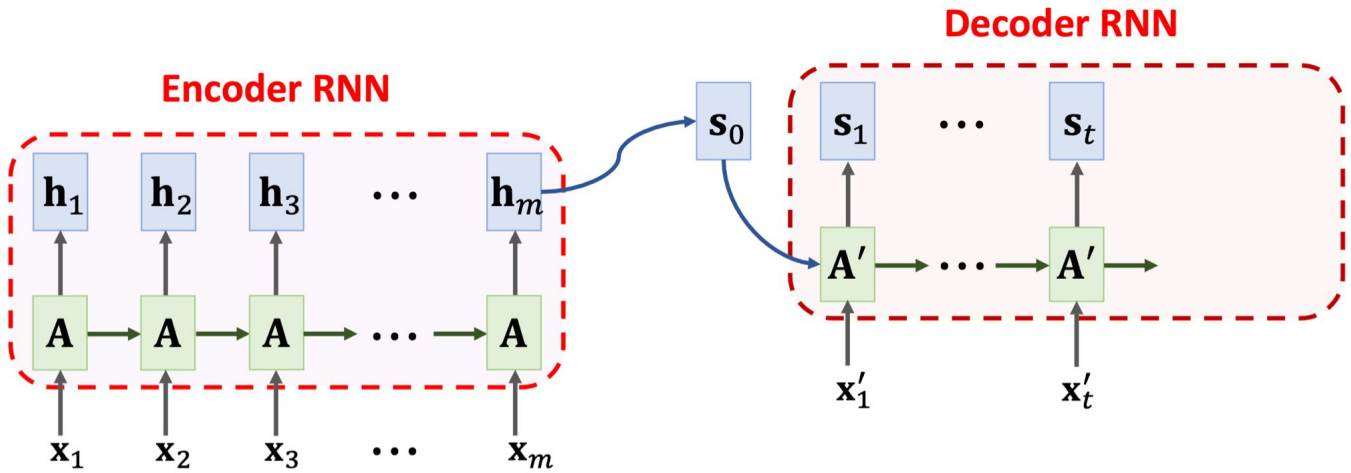
④ N to M

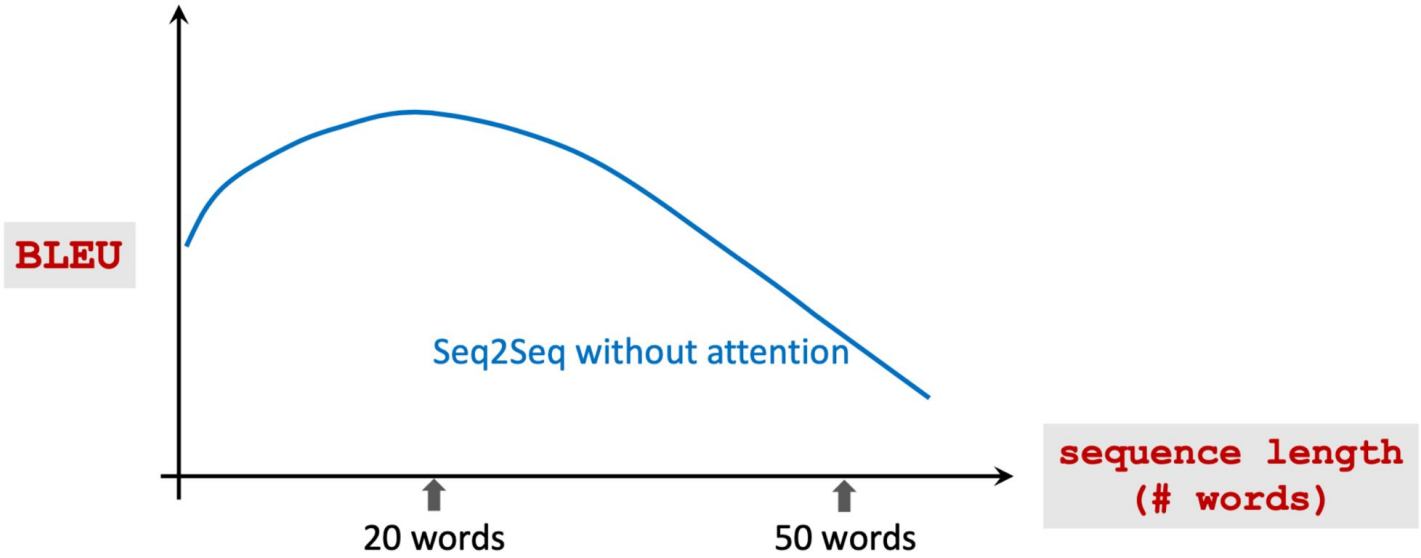


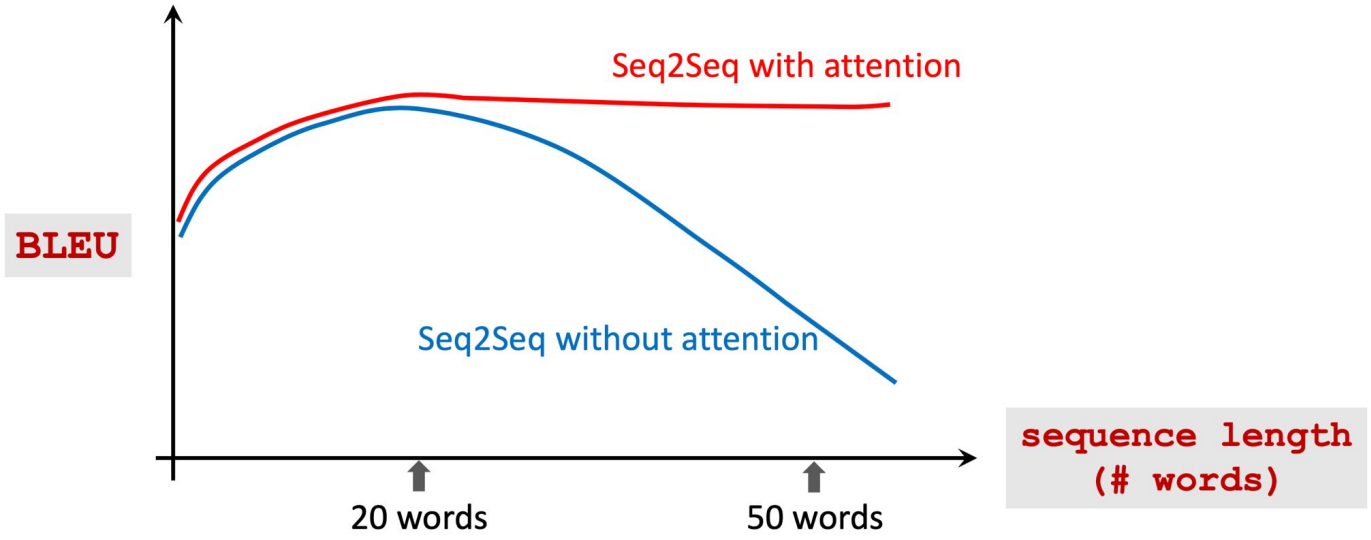
ex. machine translation , speech recognition

Seq2Seq Model

- The final state is incapable of remembering a long sequence.

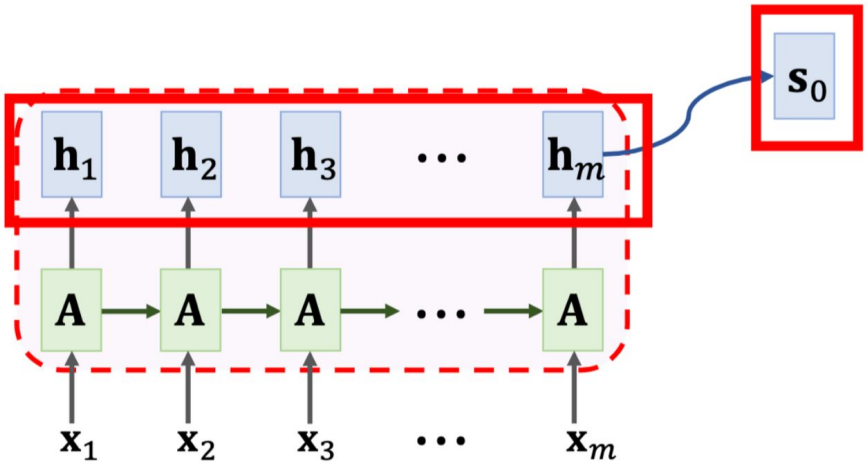



BLEU
Seq2Seq without attention
sequence length
(# words)
20 words
50 words


BLEU
Seq2Seq with attention
Seq2Seq without attention
sequence length
(# words)
20 words
50 words

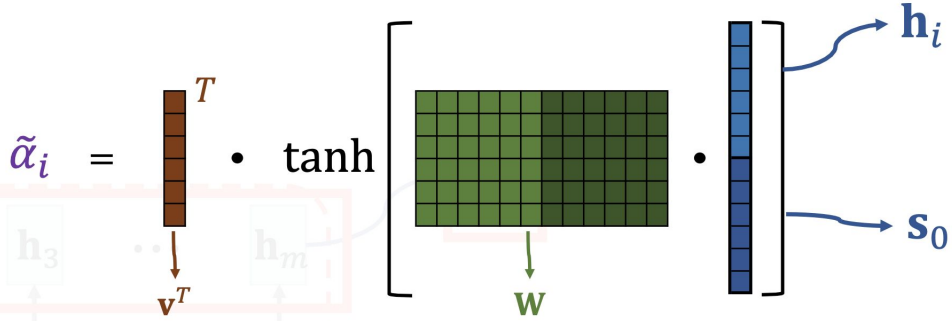
SimpleRNN + Attention

Weight: $\alpha_i = \text{align}(\mathbf{h}_i, \mathbf{s}_0)$.



align()

Option 1 (used in the attention paper)



Then **normalize** $\tilde{\alpha}_1, \cdots, \tilde{\alpha}_m$ (so that they sum to 1):

$$[\alpha_1, \cdots, \alpha_m] = \text{Softmax}\,([\tilde{\alpha}_1, \cdots, \tilde{\alpha}_m]).$$

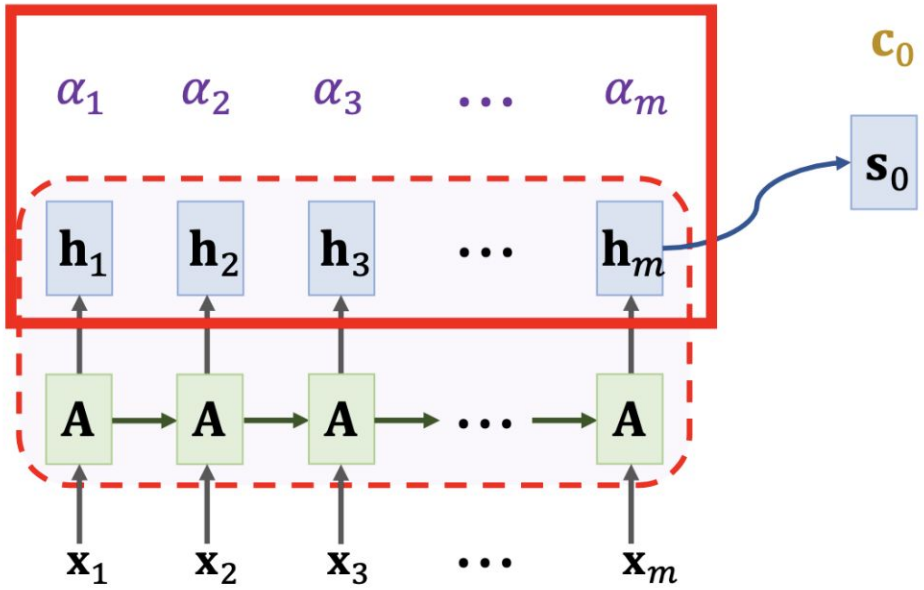
Option 2 (used in the Transformer)

1. Linear maps:
   - $\mathbf{k}_i = \mathbf{W}_K \cdot \mathbf{h}_i$, for $i = 1$ to m.
   - $\mathbf{q}_0 = \mathbf{W}_Q \cdot \mathbf{s}_0$.
2. Inner product:
   - $\tilde{\alpha}_i = \mathbf{k}_i^T \mathbf{q}_0$, for $i = 1$ to m.
3. Normalization:
   - $[\alpha_1, \cdots, \alpha_m] = \text{Softmax}\,([\tilde{\alpha}_1, \cdots, \tilde{\alpha}_m])$.

SimpleRNN + Attention

Weight: $\alpha_i = \text{align}(\mathbf{h}_i, \mathbf{s}_0)$.

Context vector: $\mathbf{c}_0 = \alpha_1 \mathbf{h}_1 + \cdots + \alpha_m \mathbf{h}_m$.



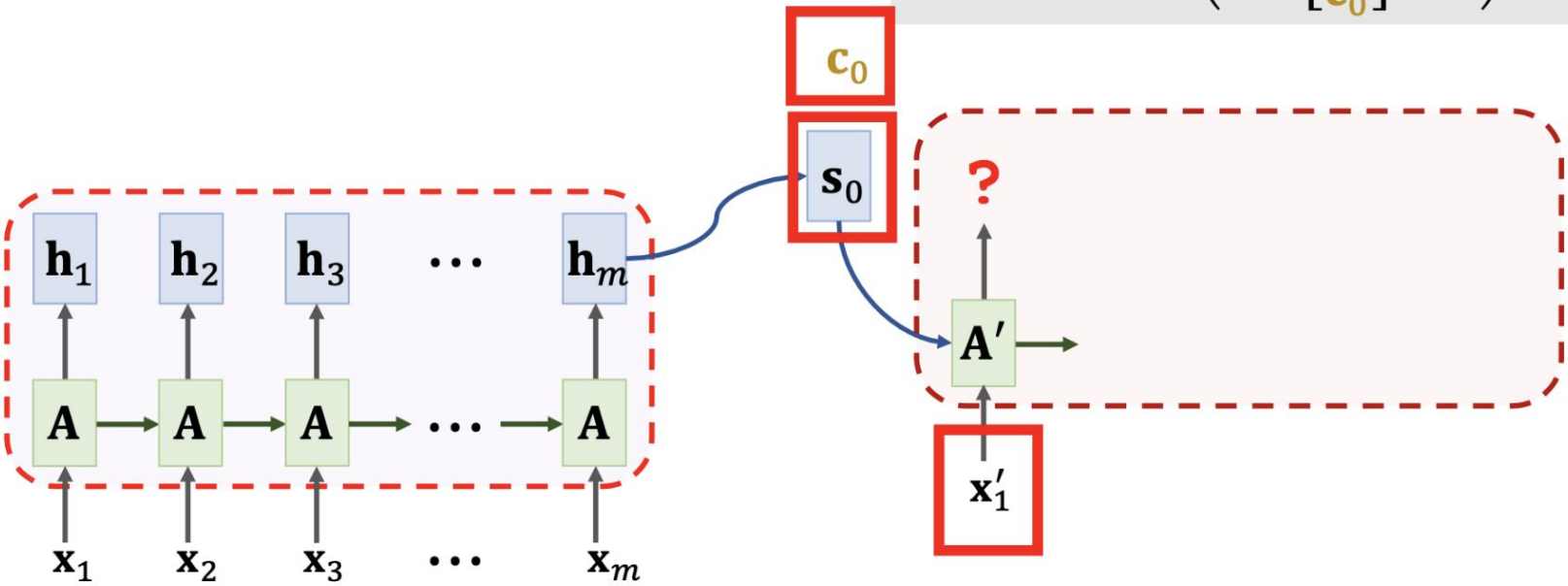
SimpleRNN + Attention

SimpleRNN:

$$s_1 = \tanh\left(A' \cdot \begin{bmatrix} x'_1 \\ s_0 \end{bmatrix} + b\right)$$

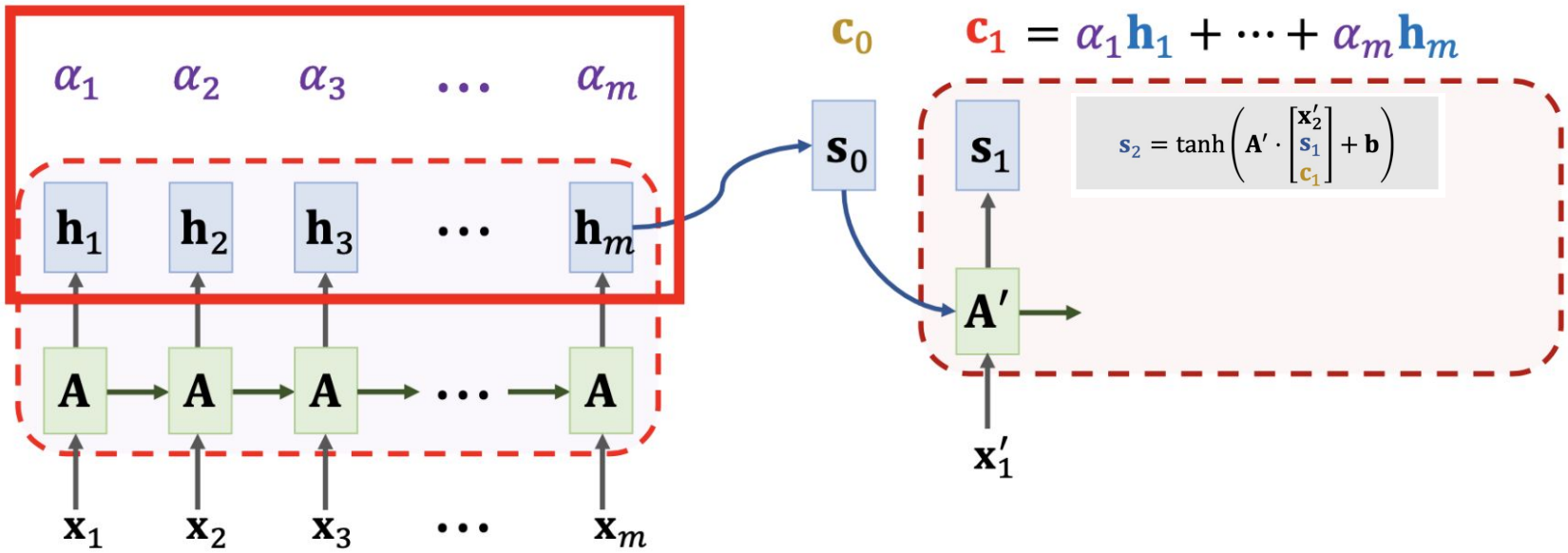
SimpleRNN + Attention:

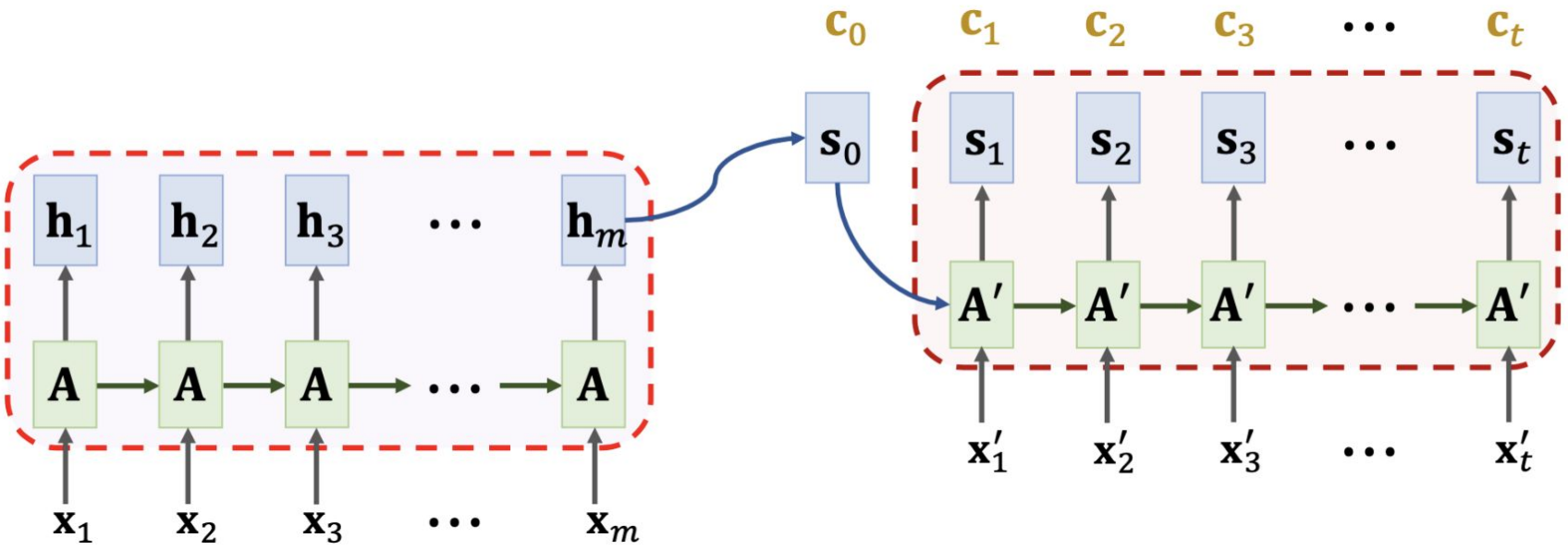
$$s_1 = \tanh\left(A' \cdot \begin{bmatrix} x'_1 \\ s_0 \\ c_0 \end{bmatrix} + b\right)$$



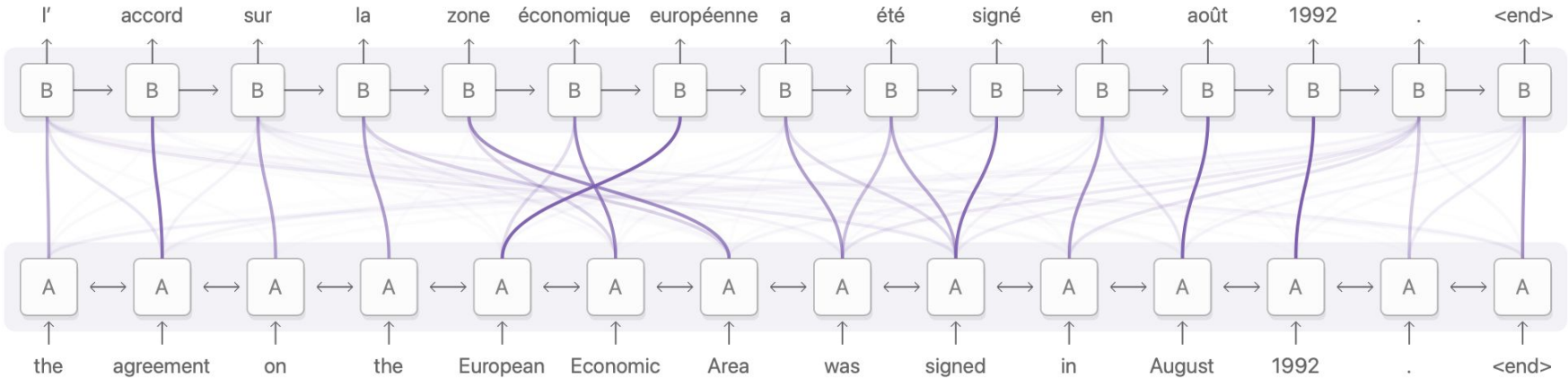
SimpleRNN + Attention

Weight: $\alpha_i = \text{align}(\mathbf{h}_i, \mathbf{s}_1)$.




$\mathbf{c}_0$ $\mathbf{c}_1$ $\mathbf{c}_2$ $\mathbf{c}_3$ ⋯ $\mathbf{c}_t$
$\mathbf{s}_0$ $\mathbf{s}_1$ $\mathbf{s}_2$ $\mathbf{s}_3$ ⋯ $\mathbf{s}_t$
$\mathbf{h}_1$ $\mathbf{h}_2$ $\mathbf{h}_3$ ⋯ $\mathbf{h}_m$
$\mathbf{A}$ $\mathbf{A}$ $\mathbf{A}$ ⋯ $\mathbf{A}$
$\mathbf{A}'$ $\mathbf{A}'$ $\mathbf{A}'$ ⋯ $\mathbf{A}'$
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ ⋯ $\mathbf{x}_m$
$\mathbf{x}'_1$ $\mathbf{x}'_2$ $\mathbf{x}'_3$ ⋯ $\mathbf{x}'_t$

Visualization of Machine Translation

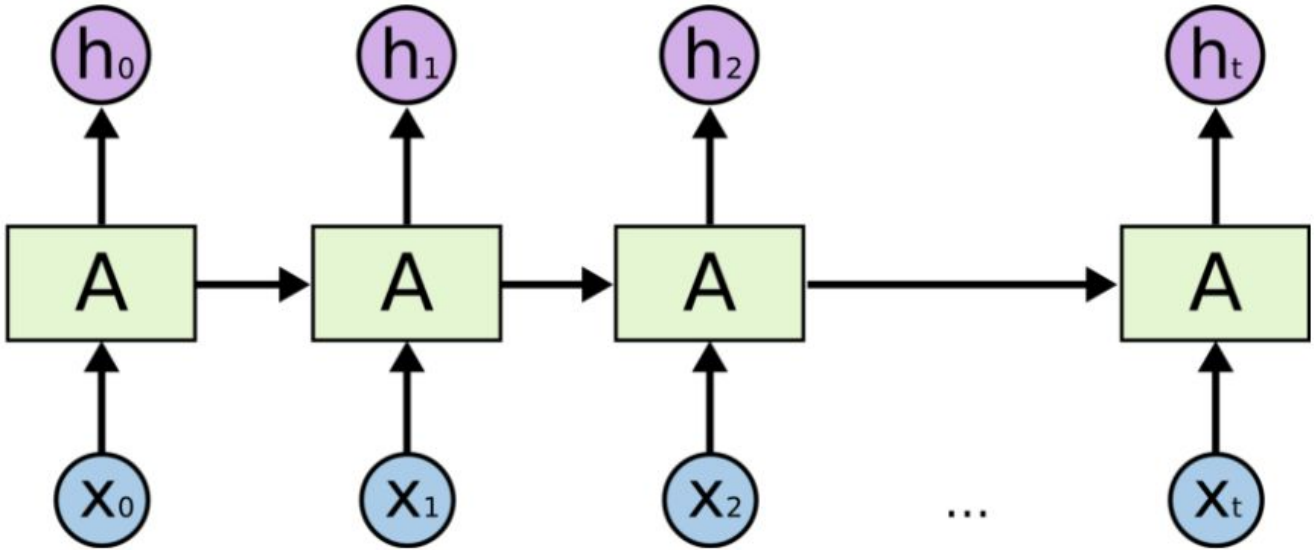


Weight: $\alpha_i = \text{align}(\mathbf{h}_i, \mathbf{s}_0)$.

Context vector: $\mathbf{c}_0 = \alpha_1 \mathbf{h}_1 + \cdots + \alpha_m \mathbf{h}_m$.

https://distill.pub/2016/augmented-rnns/

RNN



SimpleRNN + Self-Attention

SimpleRNN:

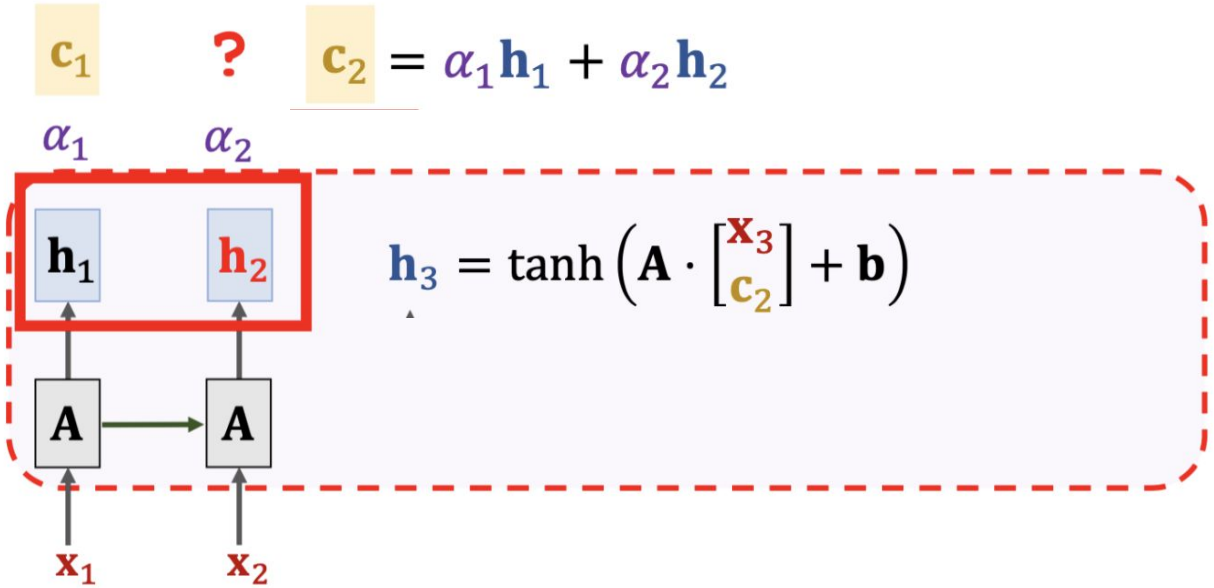
$$\mathbf{h}_1 = \tanh\left(\mathbf{A} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{h}_0 \end{bmatrix} + \mathbf{b}\right)$$

SimpleRNN + Self-Attention:

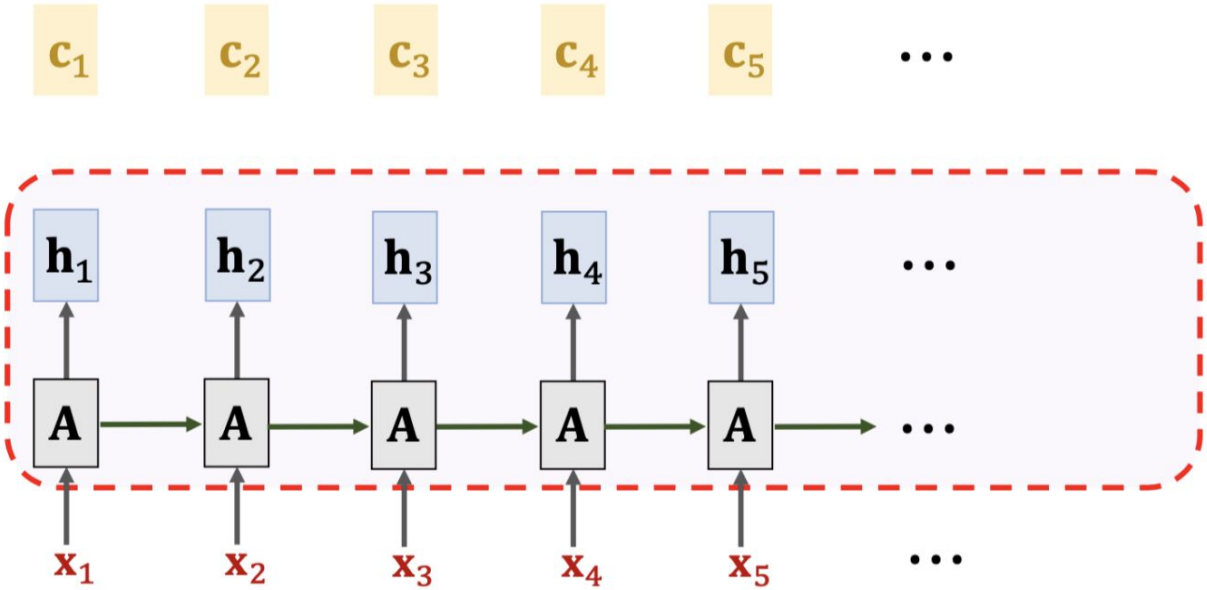
$$\mathbf{h}_1 = \tanh\left(\mathbf{A} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{c}_0 \end{bmatrix} + \mathbf{b}\right)$$

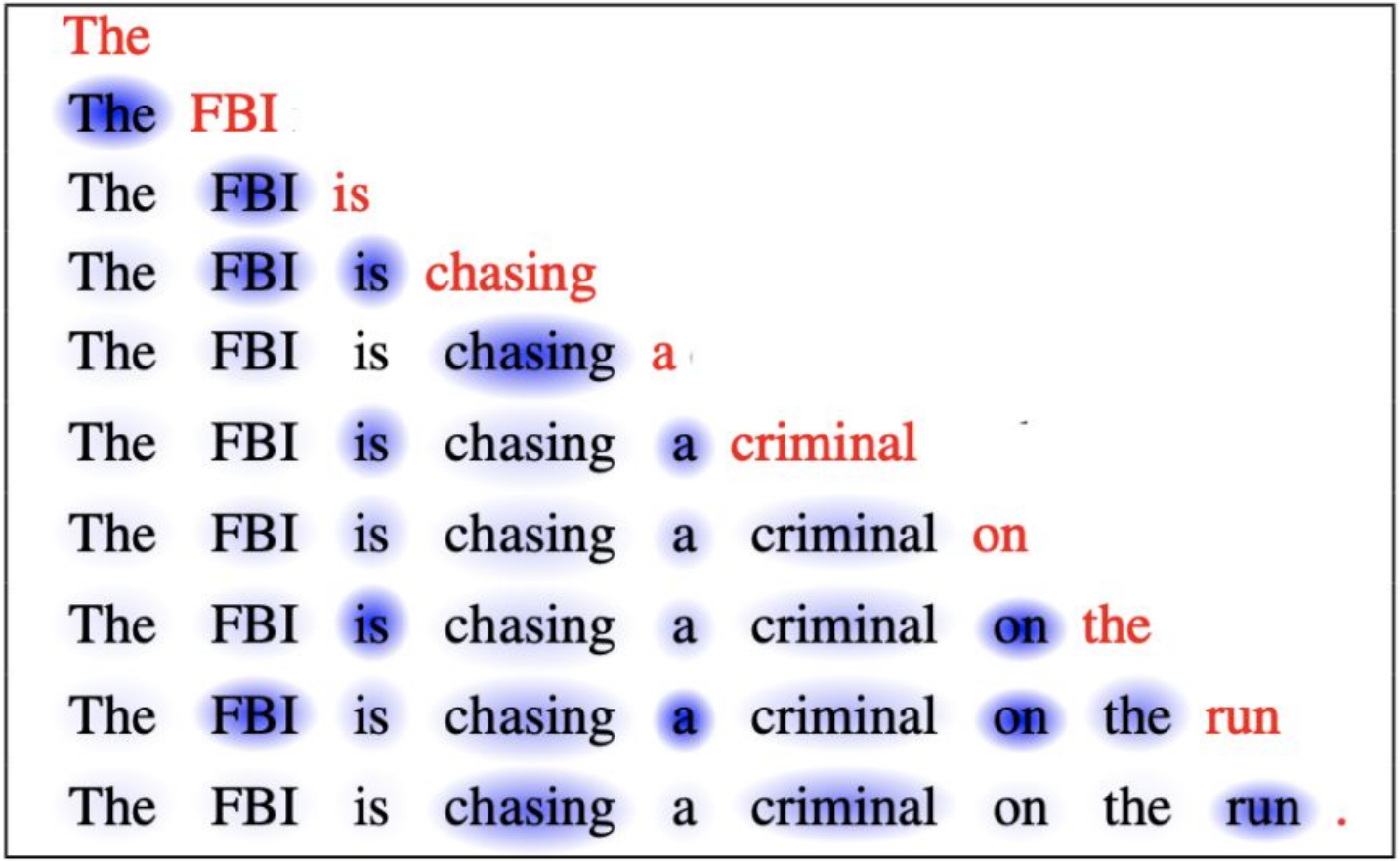
SimpleRNN + Self-Attention

Weights: $\alpha_i = \text{align}(\mathbf{h}_i, \mathbf{h}_2)$.

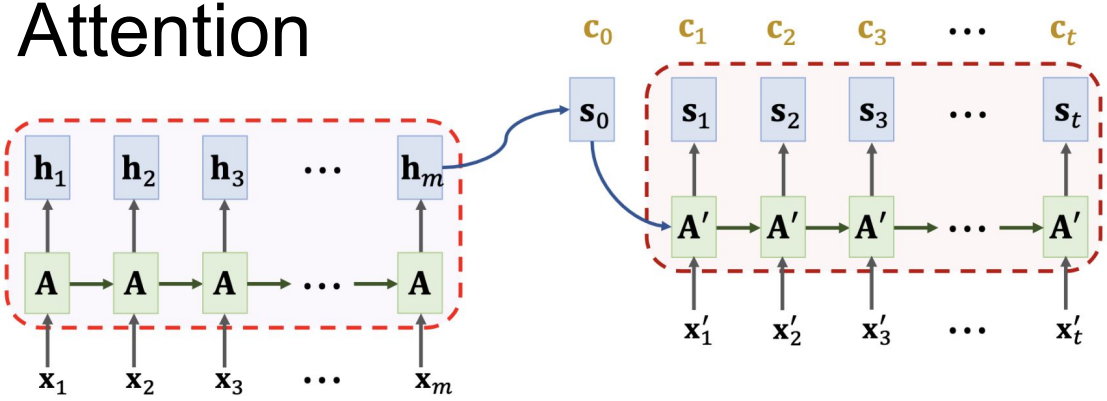


SimpleRNN + Self-Attention

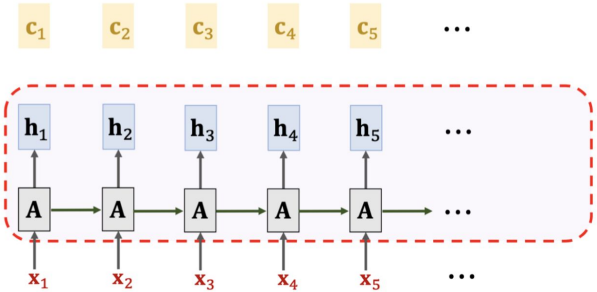



The
The FBI
The FBI is
The FBI is chasing
The FBI is chasing a
The FBI is chasing a criminal
The FBI is chasing a criminal on
The FBI is chasing a criminal on the
The FBI is chasing a criminal on the run
The FBI is chasing a criminal on the run .

Attention

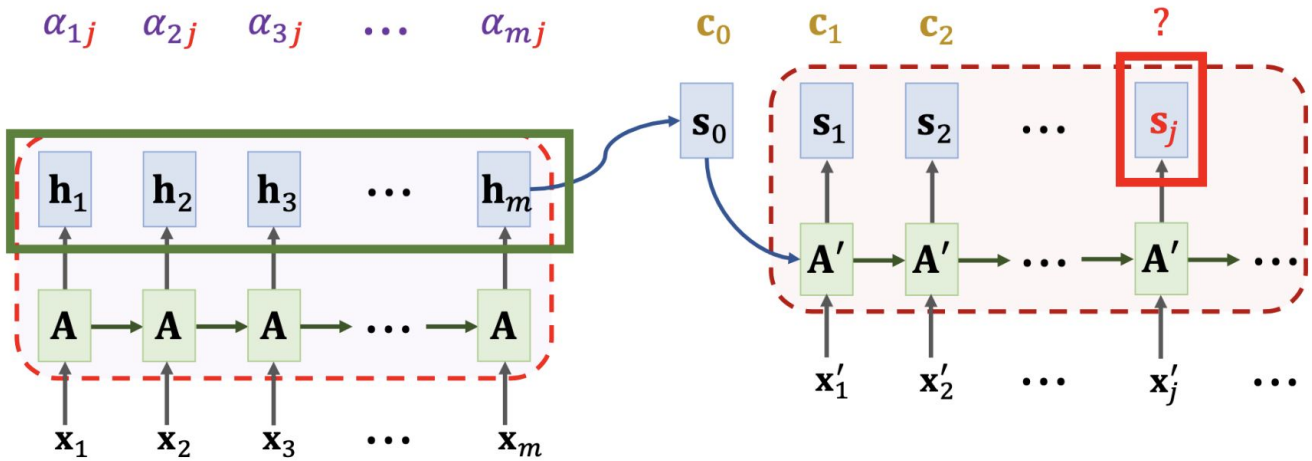


Self-Attention



Attention for Seq2Seq Model

Weights: $\alpha_{ij} = \text{align}(\mathbf{h}_i, \mathbf{s}_j)$.



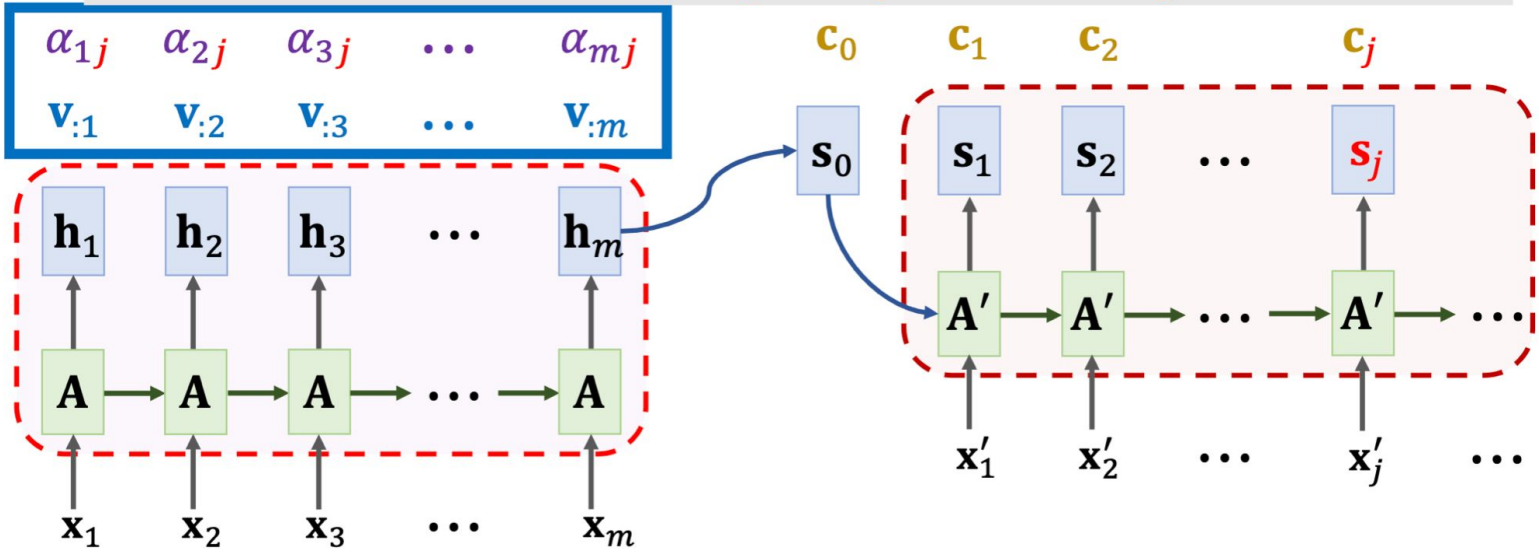
align()

1. Linear maps:
   - $\mathbf{k}_i = \mathbf{W}_K \cdot \mathbf{h}_i$, for $i = 1$ to m.
   - $\mathbf{q}_0 = \mathbf{W}_Q \cdot \mathbf{s}_0$.
2. Inner product:
   - $\tilde{\alpha}_i = \mathbf{k}_i^T \mathbf{q}_0$, for $i = 1$ to m.
3. Normalization:
   - $[\alpha_1, \cdots, \alpha_m] = \text{Softmax}\,([\tilde{\alpha}_1, \cdots, \tilde{\alpha}_m])$.

Query: $\mathbf{q}_{:j} = \mathbf{W}_Q \mathbf{s}_j$, Key: $\mathbf{k}_{:i} = \mathbf{W}_K \mathbf{h}_i$, Value: $\mathbf{v}_{:i} = \mathbf{W}_V \mathbf{h}_i$.

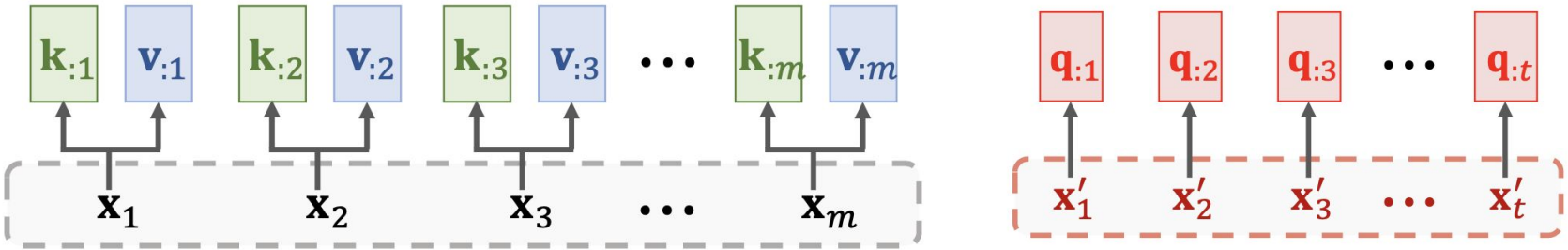
Weights: $\boldsymbol{\alpha}_{:j} = \text{Softmax}\left(\mathbf{K}^T \mathbf{q}_{:j}\right) \in \mathbb{R}^m$.

Context vector: $\mathbf{c}_j = \alpha_{1j}\mathbf{v}_{:1} + \cdots + \alpha_{mj}\mathbf{v}_{:m}$.

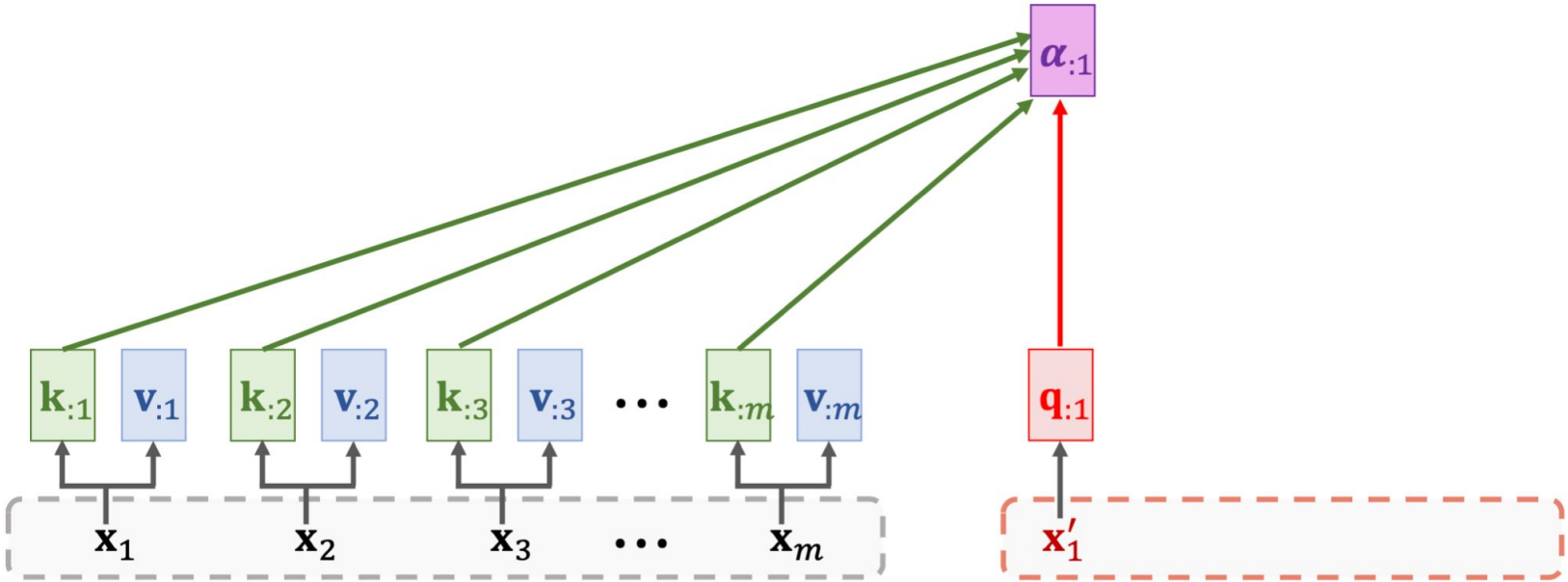


Attention without RNN

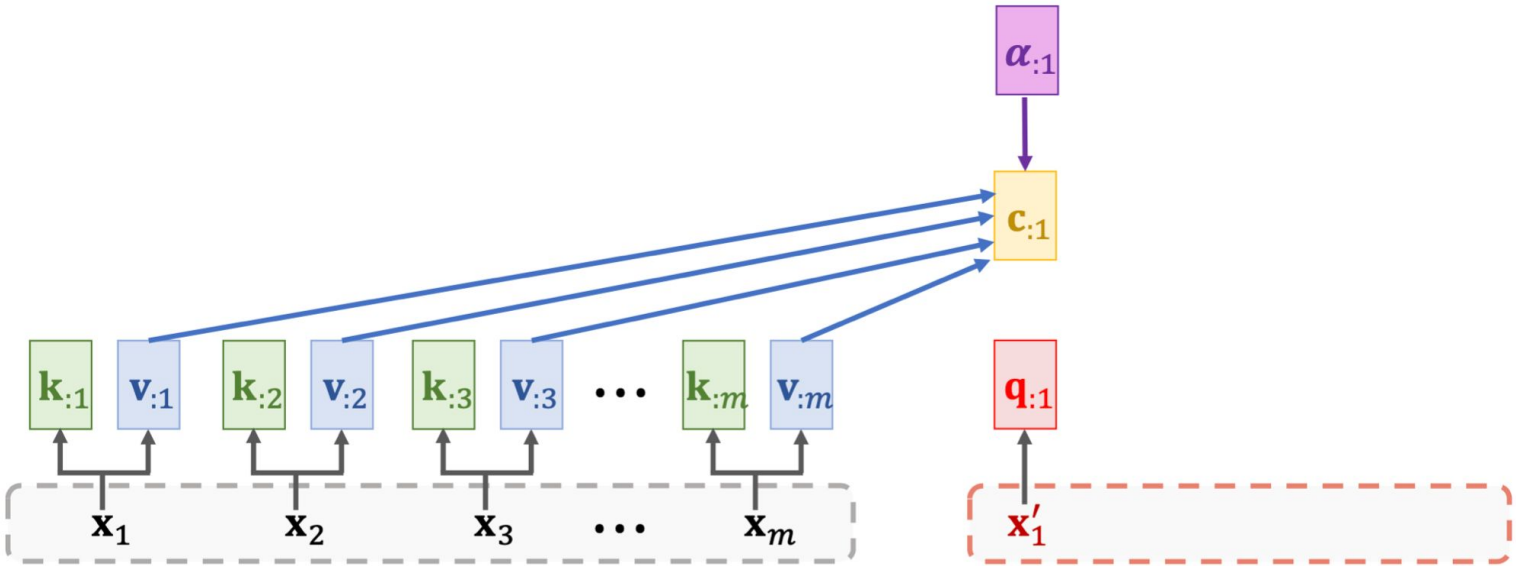
- Keys and values are based on encoder's inputs $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m$.
- Key: $\mathbf{k}_{:i} = \mathbf{W}_K \mathbf{x}_i$.
- Value: $\mathbf{v}_{:i} = \mathbf{W}_V \mathbf{x}_i$.
- Queries are based on decoder's inputs $\mathbf{x}'_1, \mathbf{x}'_2, \cdots, \mathbf{x}'_t$.
- Query: $\mathbf{q}_{:j} = \mathbf{W}_Q \mathbf{x}'_j$.



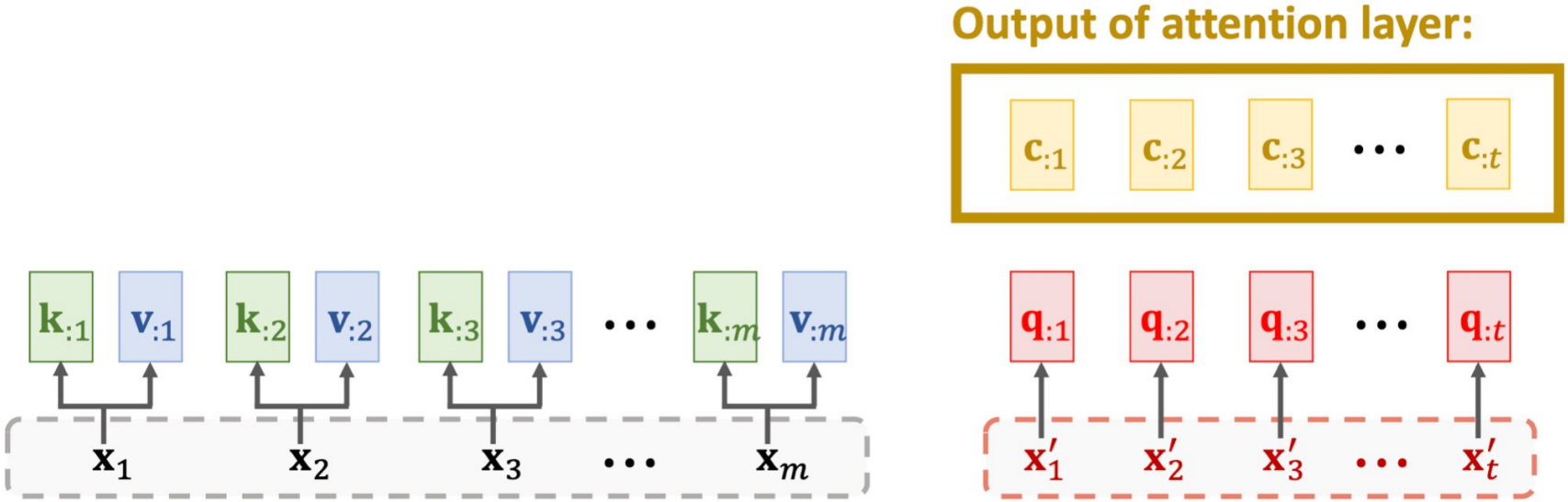
- Compute weights: $\boldsymbol{\alpha}_{:1} = \text{Softmax}(\mathbf{K}^T \mathbf{q}_{:1}) \in \mathbb{R}^m$.



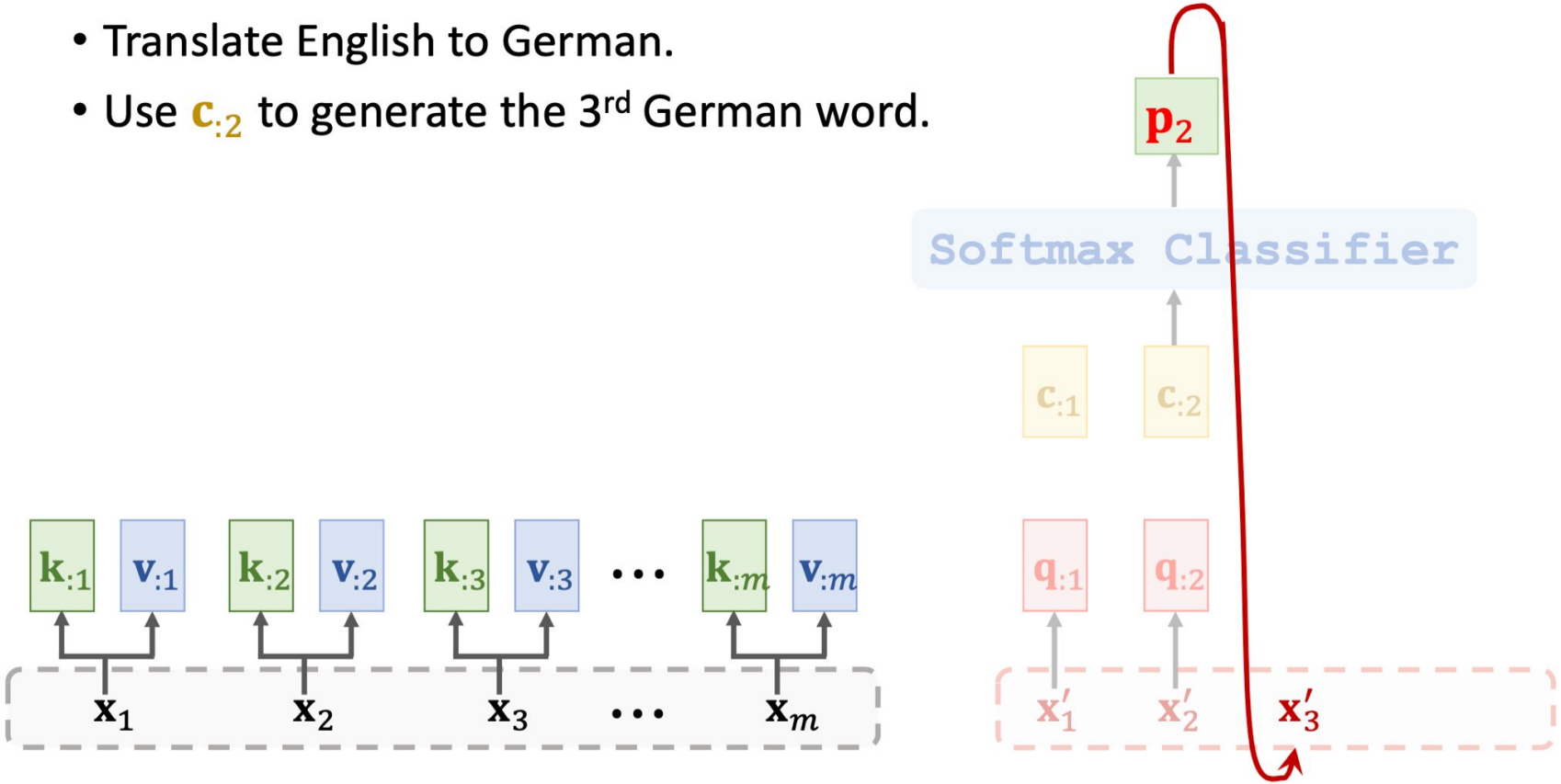
- Compute context vector: $\mathbf{c}_{:1} = \alpha_{11}\mathbf{v}_{:1} + \cdots + \alpha_{m1}\mathbf{v}_{:m} = \mathbf{V}\boldsymbol{\alpha}_{:1}$.



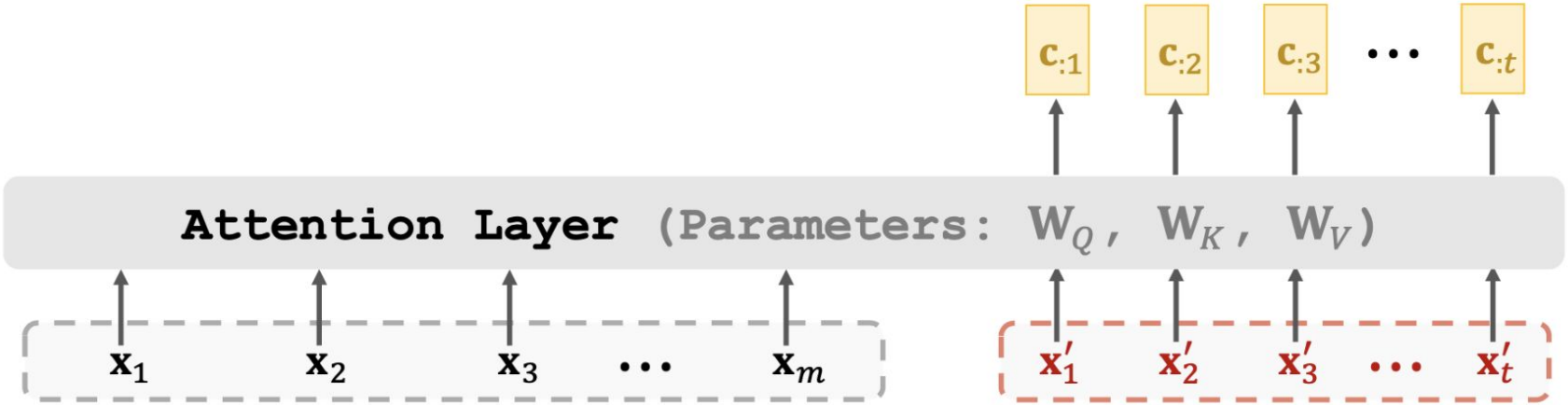
- Output of attention layer: $\mathbf{C} = [\mathbf{c}_{:1}, \mathbf{c}_{:2}, \mathbf{c}_{:3}, \cdots, \mathbf{c}_{:t}]$.
- Here, $\mathbf{c}_{:j} = \mathbf{V} \cdot \mathrm{Softmax}(\mathbf{K}^T \mathbf{q}_{:j})$.
- Thus, $\mathbf{c}_{:j}$ is a function of $\mathbf{x}'_j$ and $[\mathbf{x}_1, \cdots, \mathbf{x}_m]$.



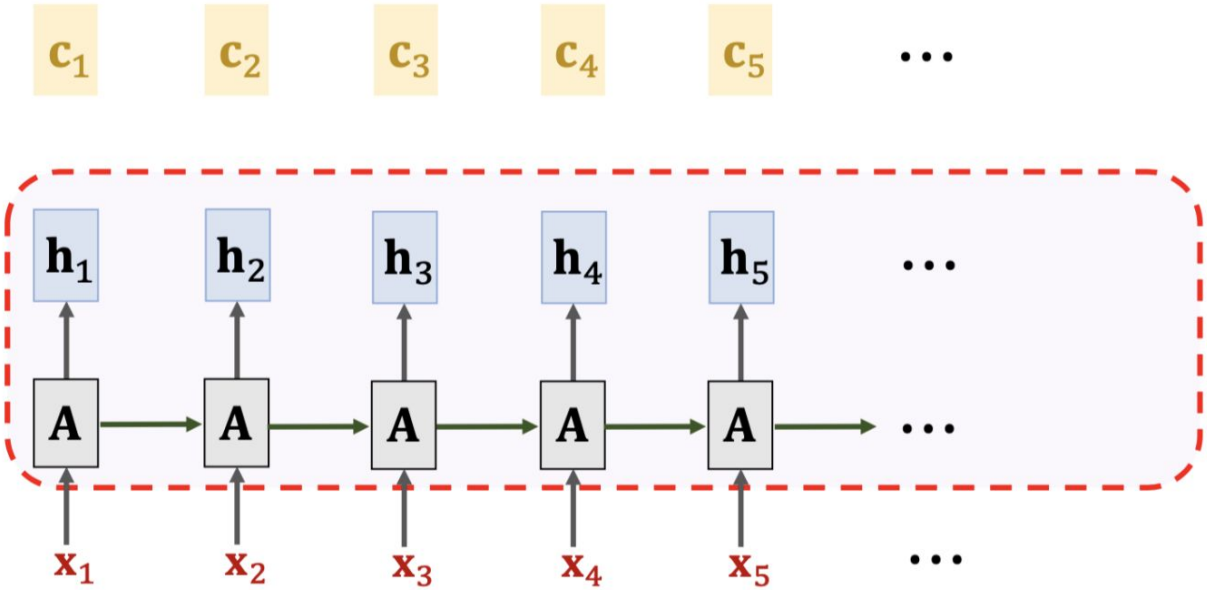
- Translate English to German.
- Use $\mathbf{c}_{:2}$ to generate the 3rd German word.



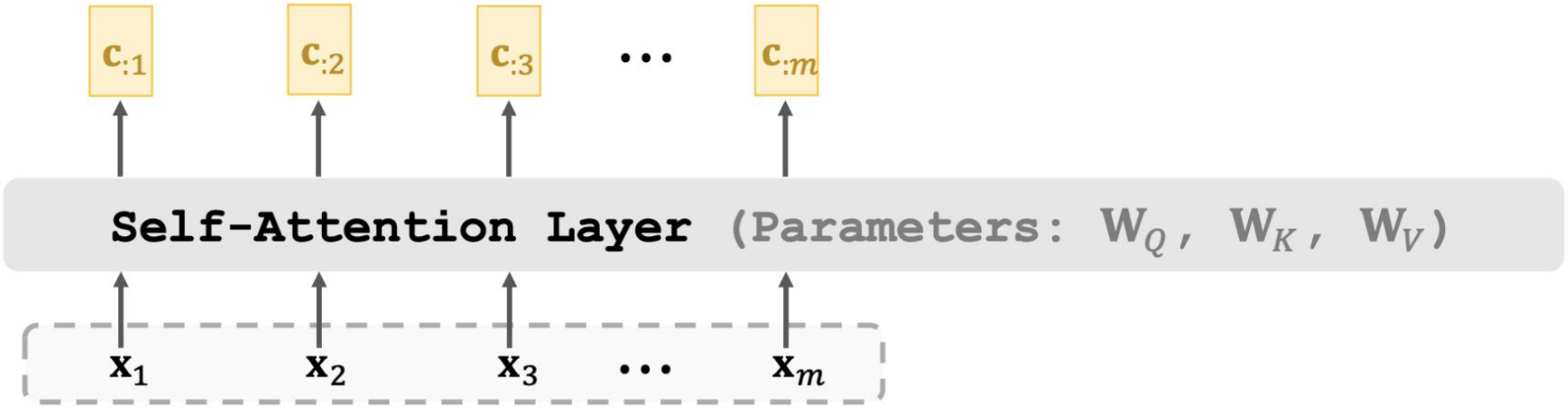
Attention without RNN



SimpleRNN + Self-Attention

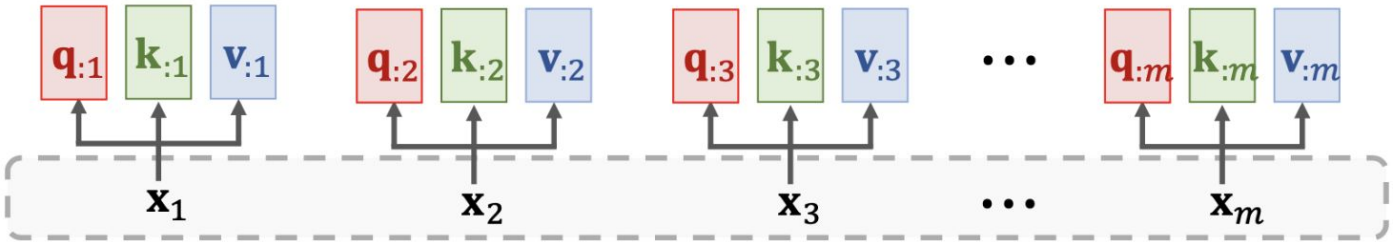


Self-Attention without RNN

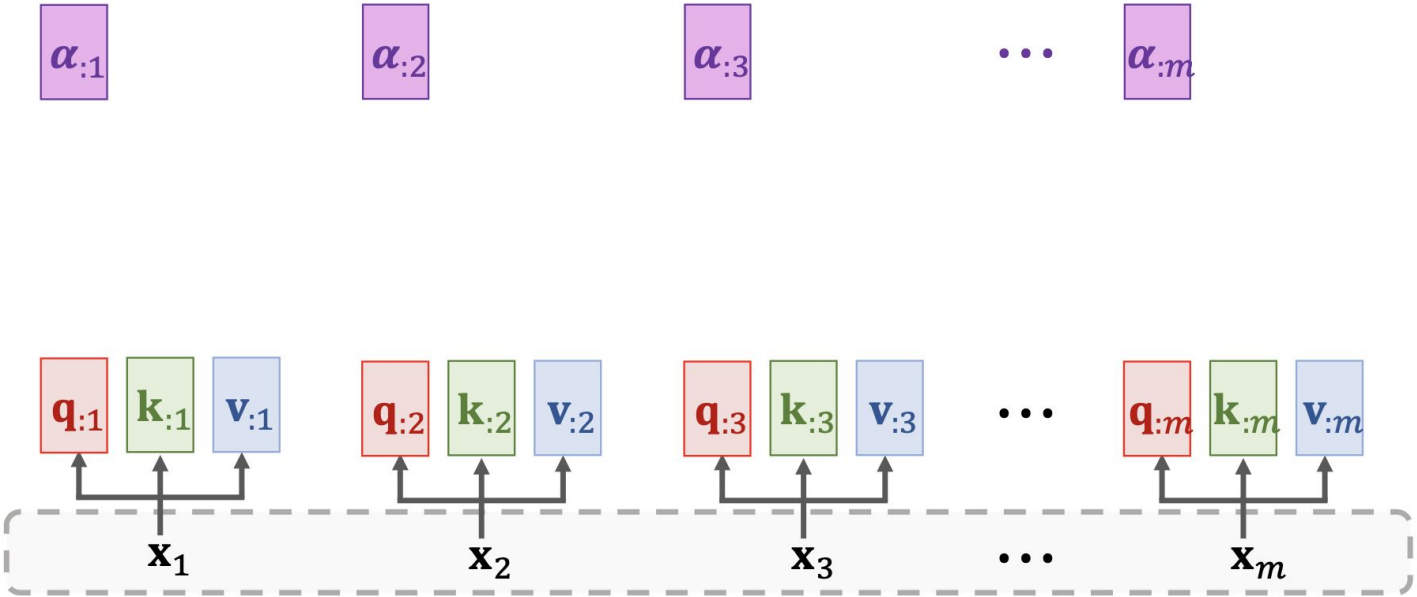


Self-Attention Layer

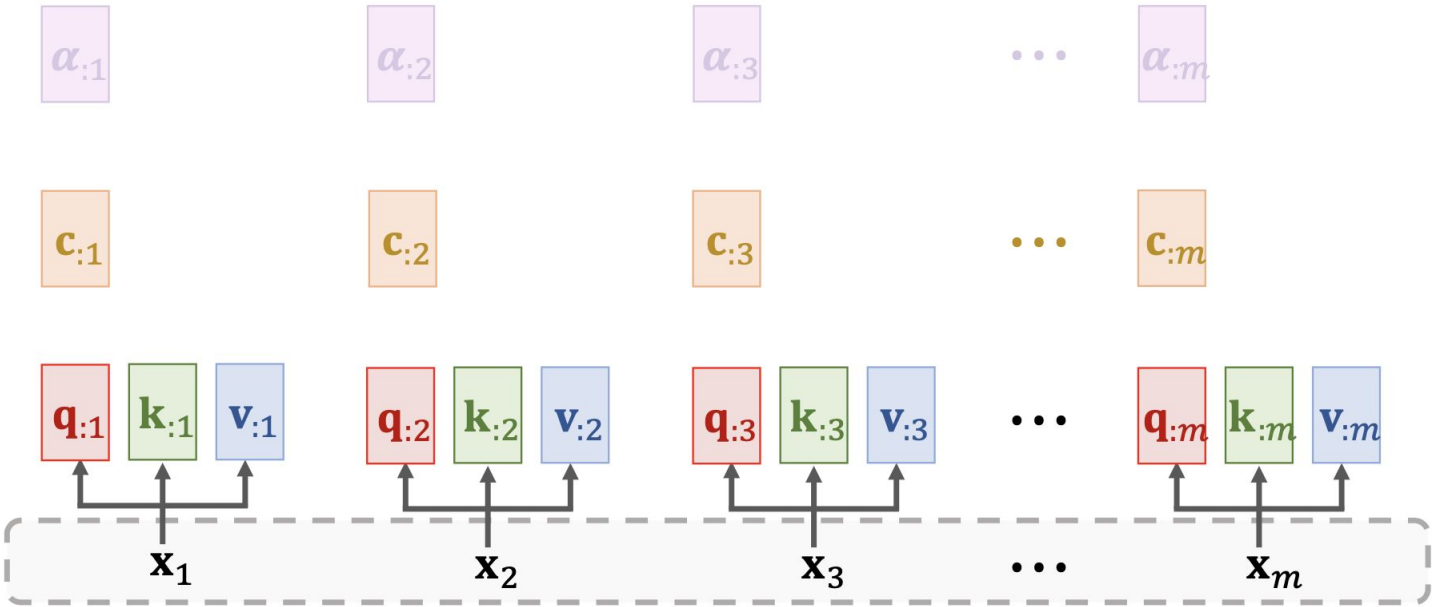
Query: $\mathbf{q}_{:i} = \mathbf{W}_Q \mathbf{x}_i$, Key: $\mathbf{k}_{:i} = \mathbf{W}_K \mathbf{x}_i$, Value: $\mathbf{v}_{:i} = \mathbf{W}_V \mathbf{x}_i$.



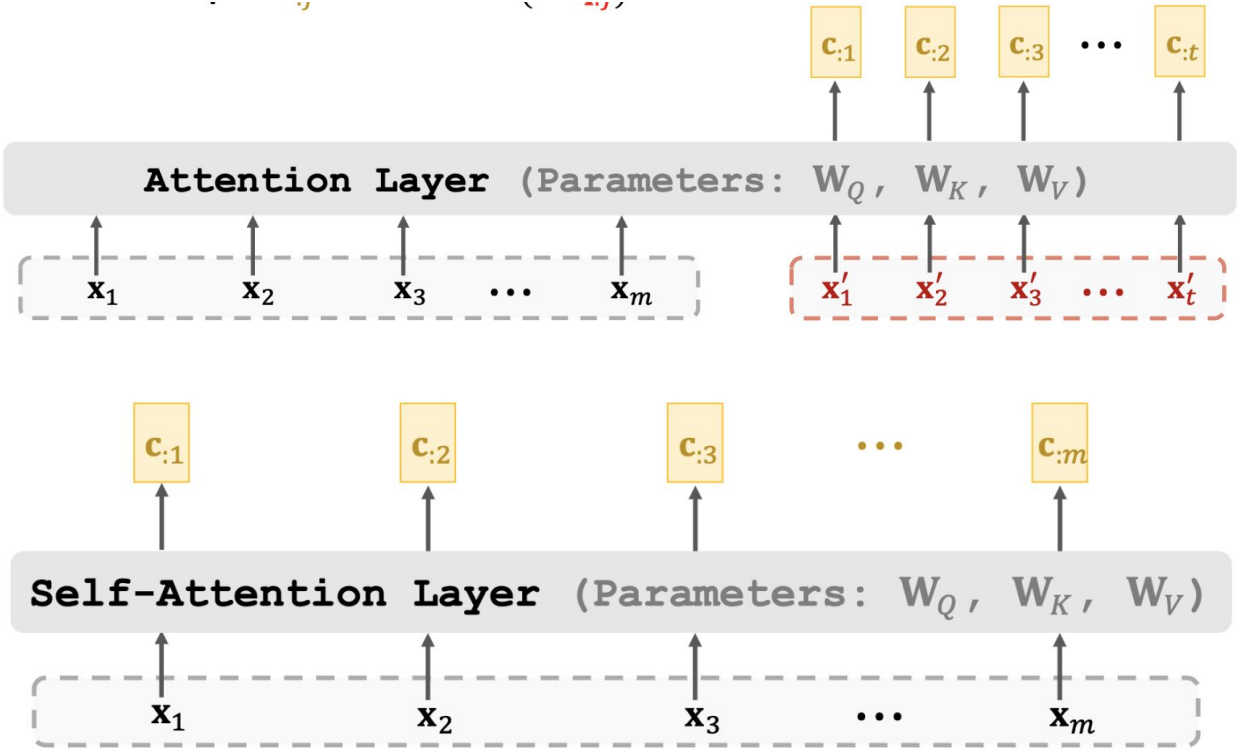
Weights: $\alpha_{:j} = \text{Softmax}\left(\mathbf{K}^T \mathbf{q}_{:j}\right) \in \mathbb{R}^m.$



Context vector: $\mathbf{c}_{:j} = \alpha_{1j}\mathbf{v}_{:1} + \cdots + \alpha_{mj}\mathbf{v}_{:m} = \mathbf{V}\boldsymbol{\alpha}_{:j}.$

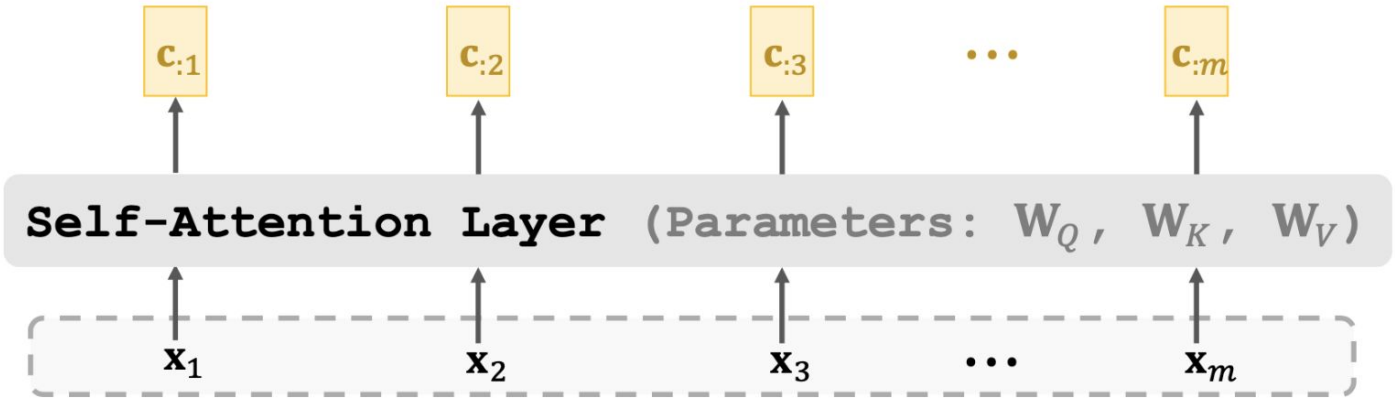


Attention and Self-Attention



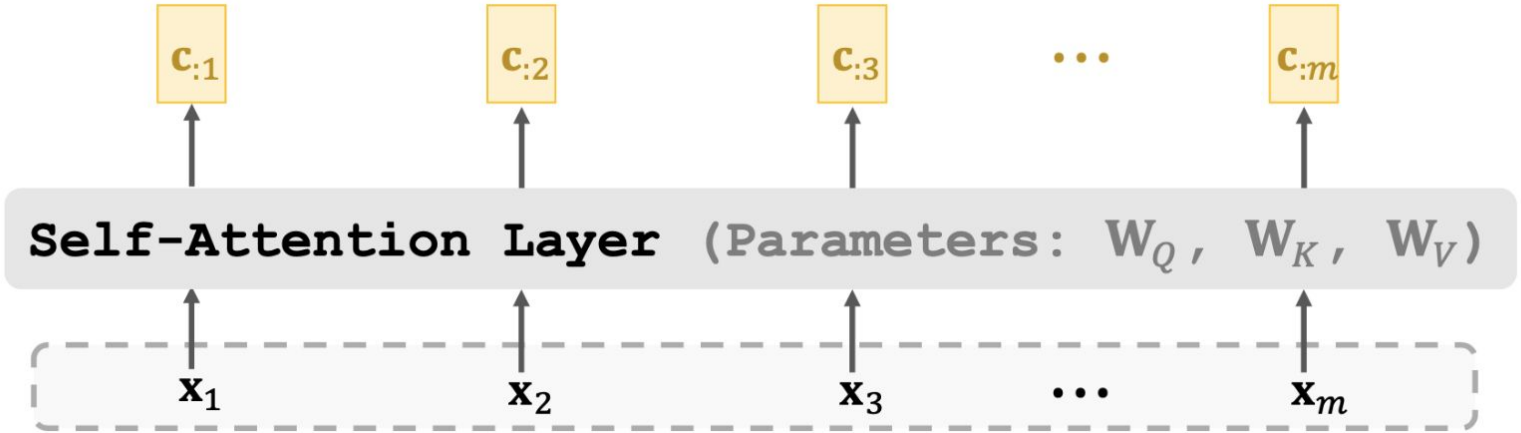
Single-Head Self-Attention

- Self-attention layer: $\mathbf{C} = \mathrm{Attn}(\mathbf{X}, \mathbf{X})$.
- This is called "single-head self-attention".



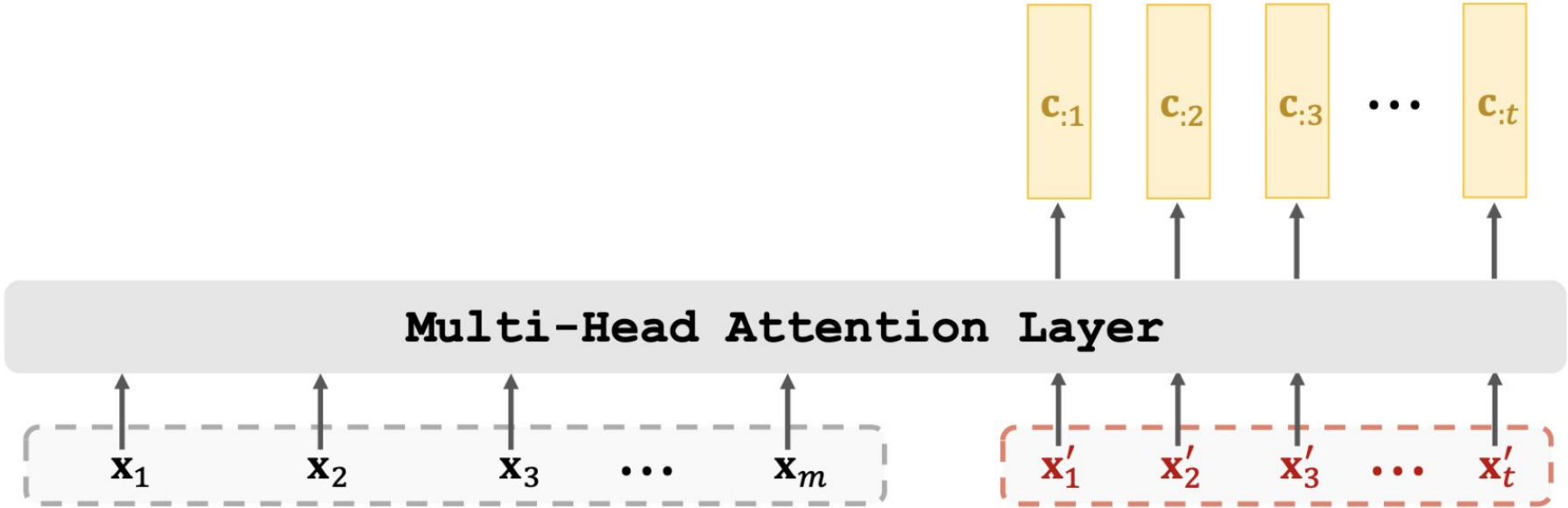
Multi-Head Self-Attention

- Using l single-head self-attentions (which do not share parameters.)
  - A single-head self-attention has 3 parameter matrices: $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$.
  - Totally $3l$ parameters matrices.
- Concatenating outputs of single-head self-attentions.
  - Suppose single-head self-attentions' outputs are $d \times m$ matrices.
  - Multi-head's output shape: $(ld) \times m$.

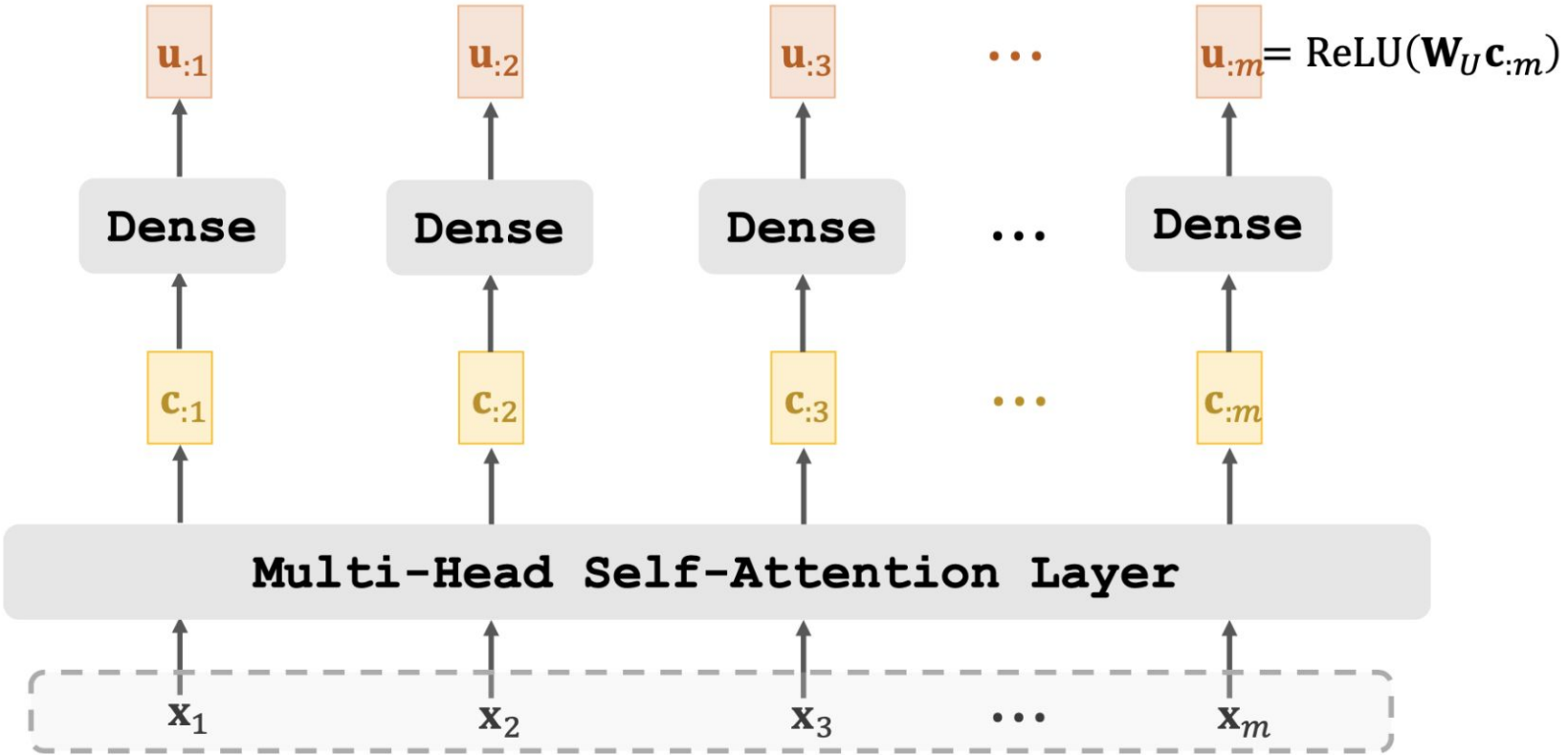


Multi-Head Attention

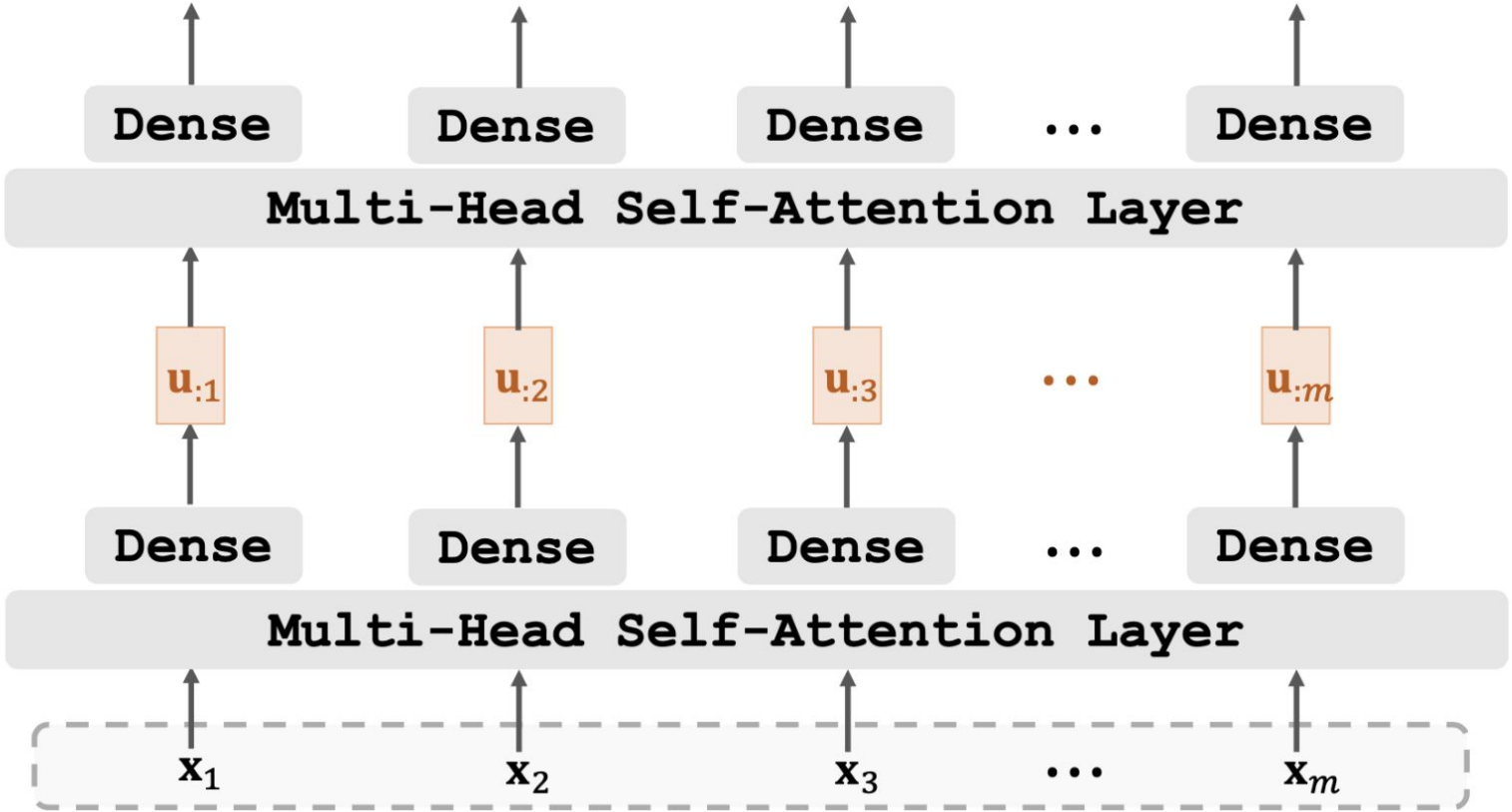
- Using l single-head attentions (which do not share parameters.)
- Concatenating single-head attentions' outputs.



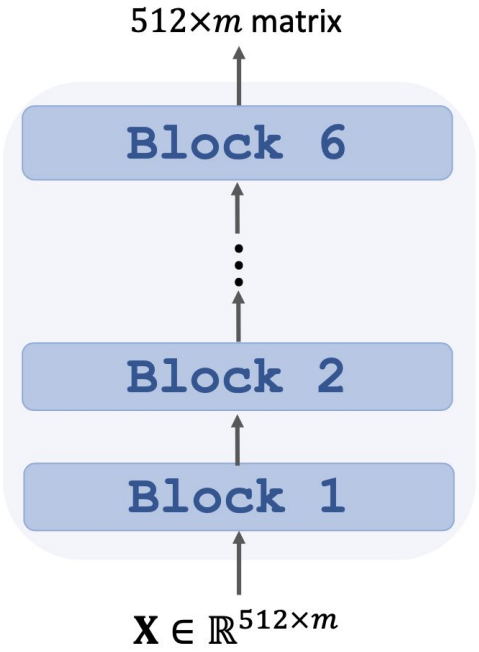
Self-Attention Layer + Dense Layer

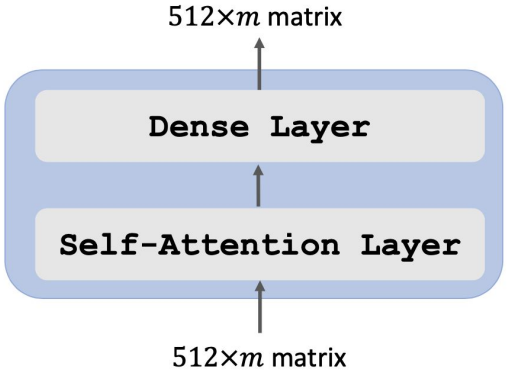


Stacked Self-Attention Layers



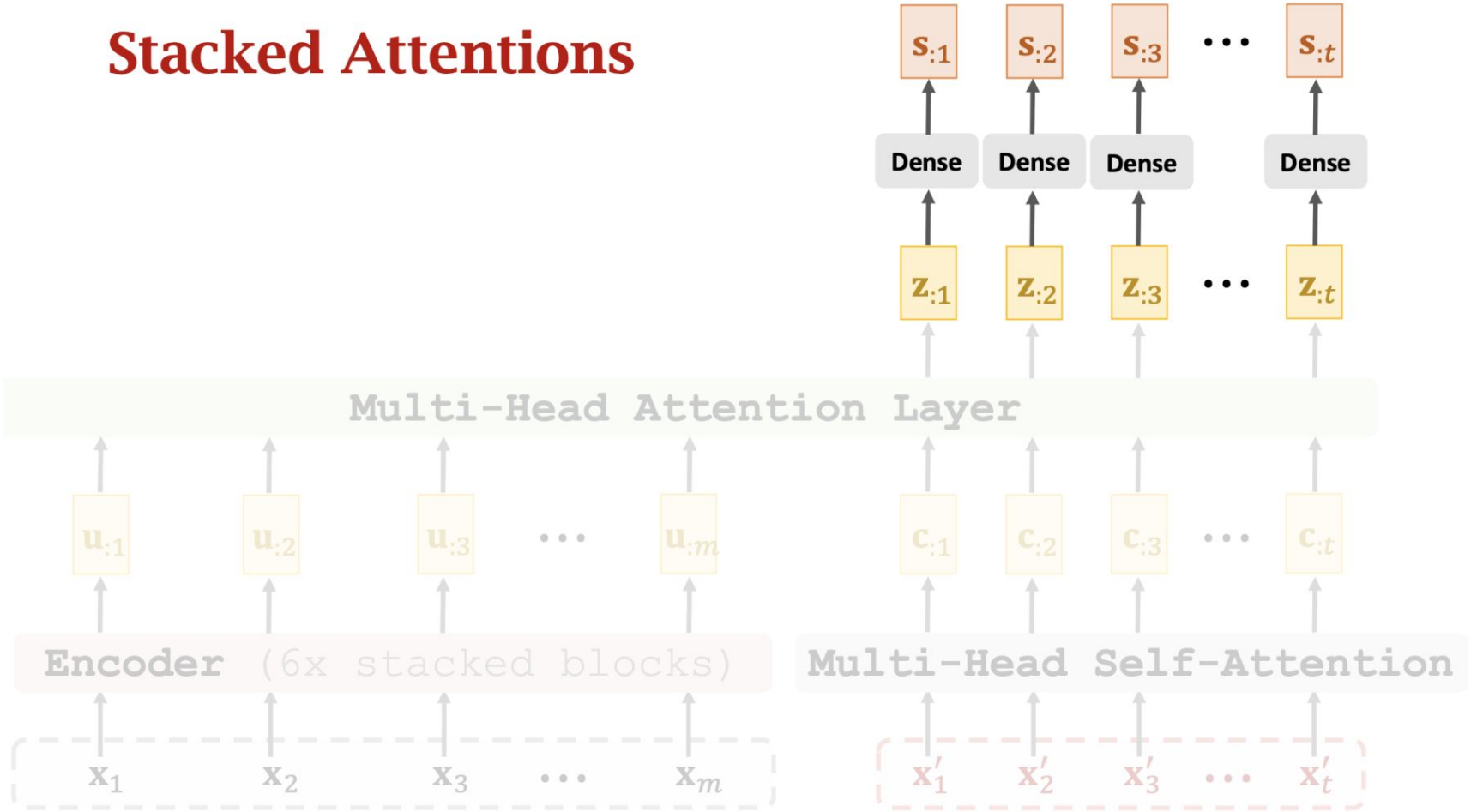
Transformer's Encoder



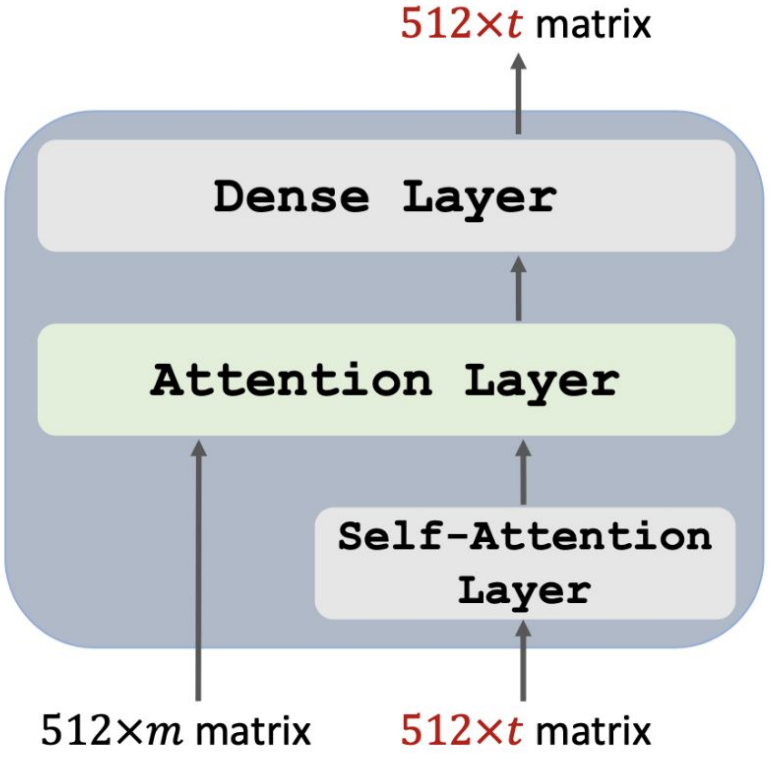


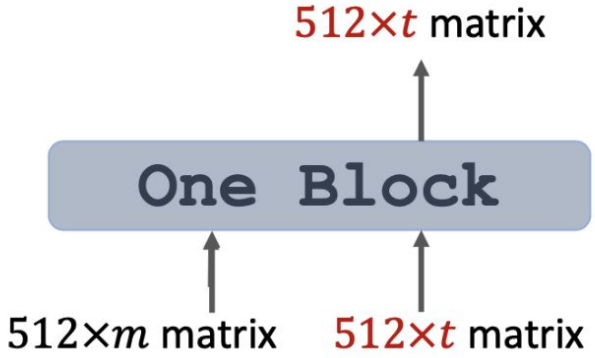
One Block of Encoder

Stacked Attentions

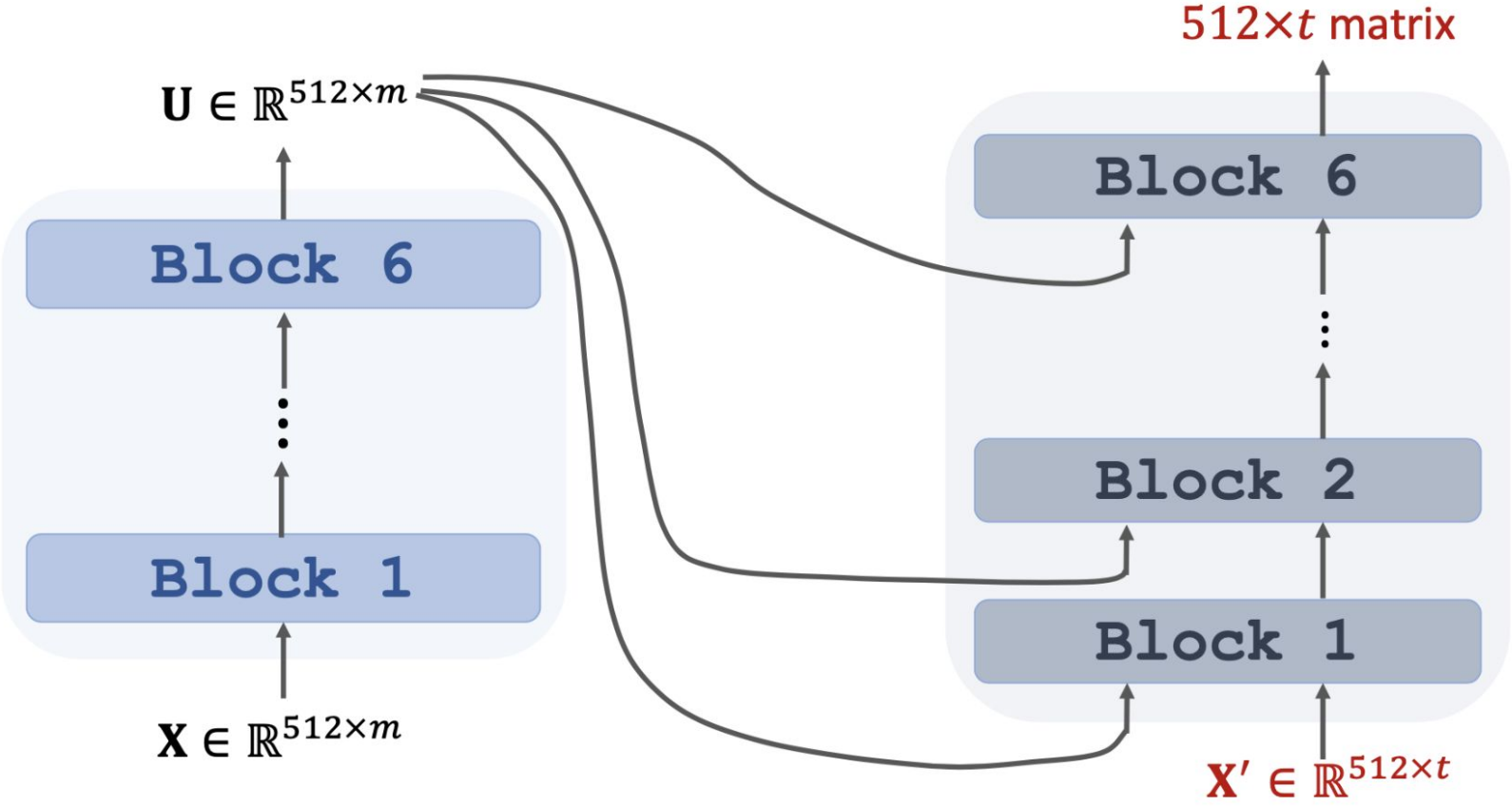


Transformer's Decoder : One Block





Transformer

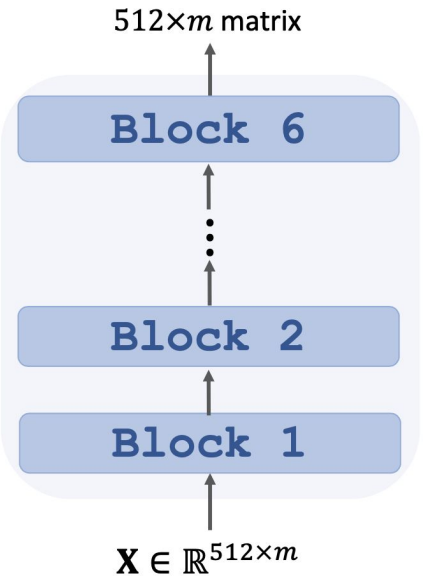


Transformer

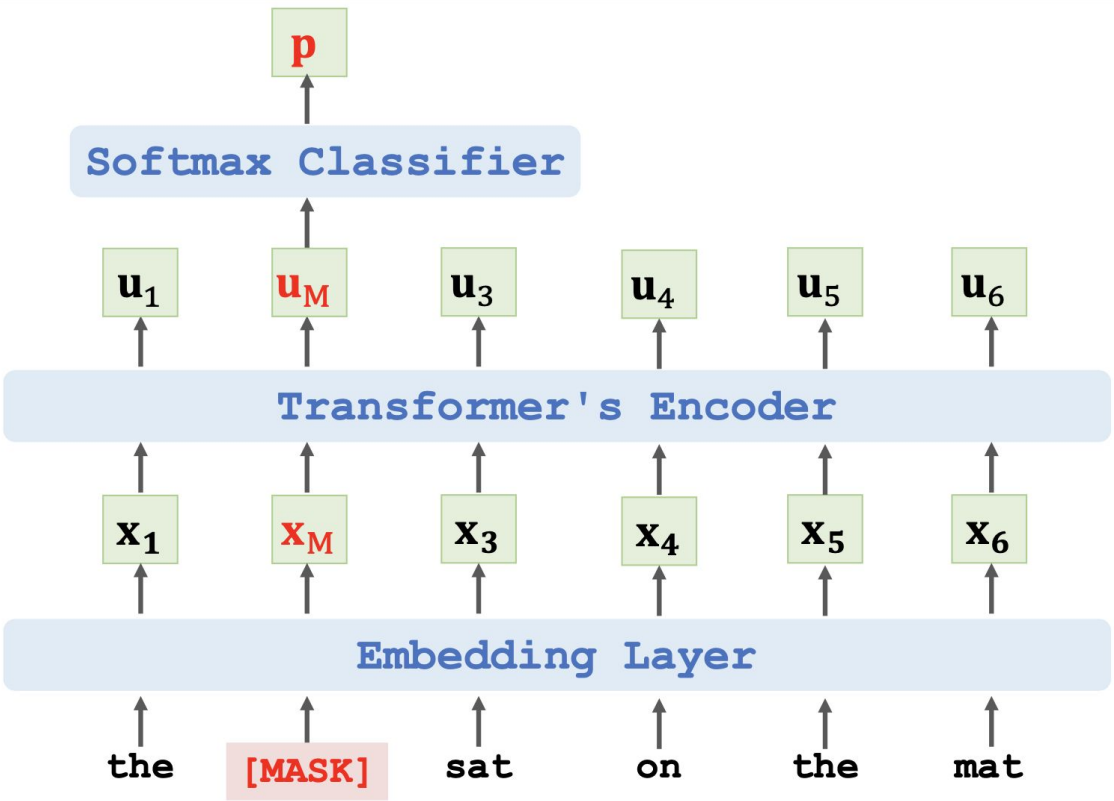
- Seq2Seq
- Attention and Self-Attention
- Attention and Self-Attention without RNN
- Multi-head Attention and Multi-head Self-Attention

Bidirectional Encoder Representations from Transformers (BERT)

- BERT is for pre-training Transformer's encoder.



Predict masked words



Predict the next sentence

- Given the sentence:

  "calculus is a branch of math".

- Is this the next sentence?

  "it was developed by newton and leibniz"

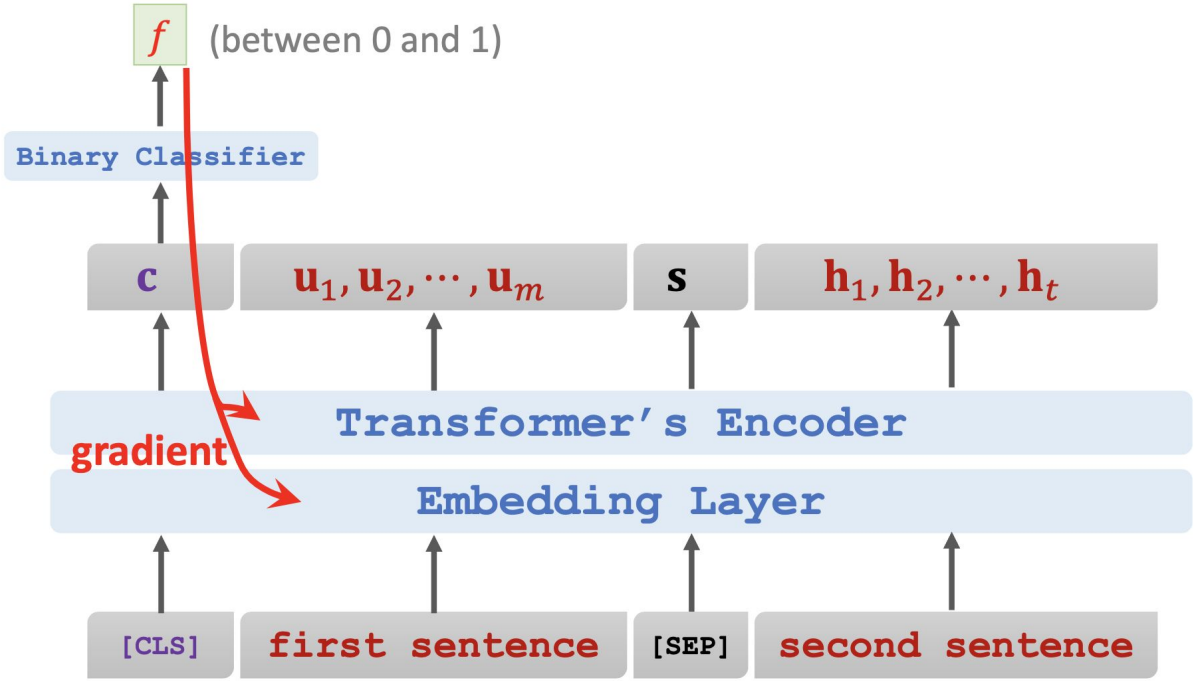
- **Input:**

  [CLS] "calculus is a branch of math"
  [SEP] "it was developed by newton and leibniz"

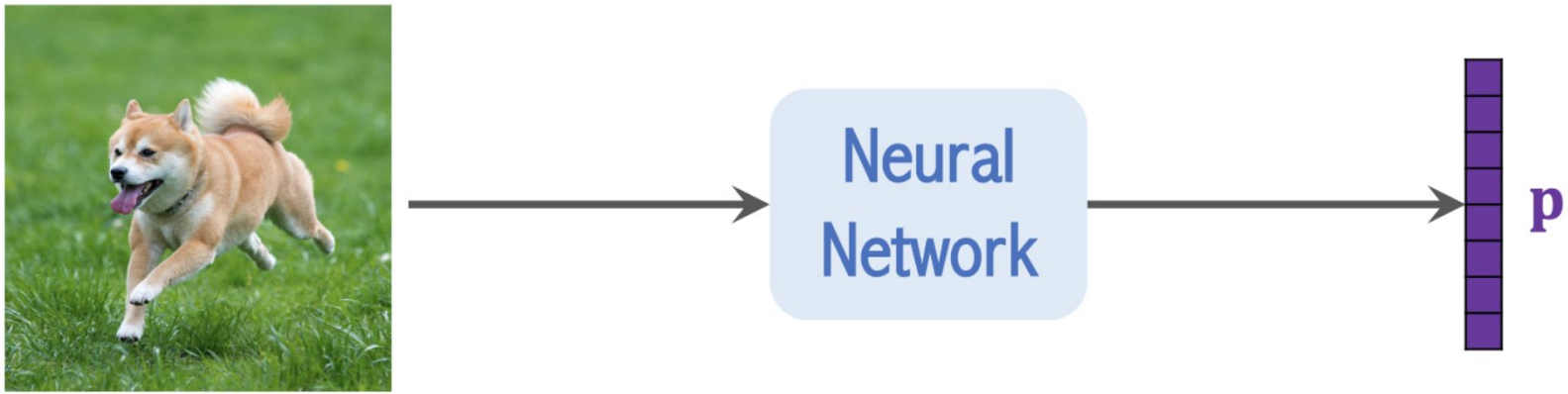
- **[CLS]** is a token for classification.
- **[SEP]** is for separating sentences.

- **Target:** true

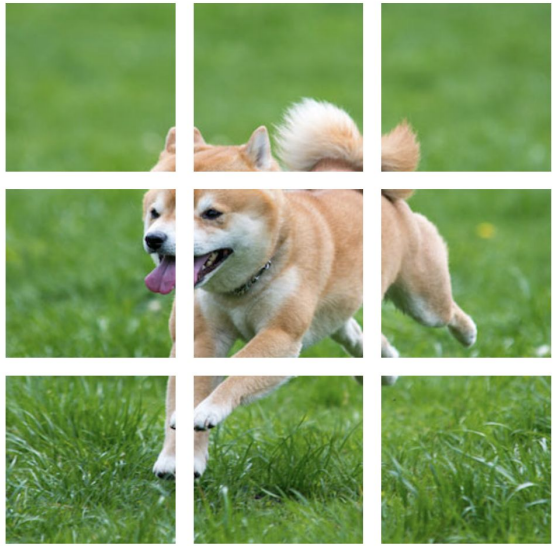
Predict the next sentence



Vision Transformer



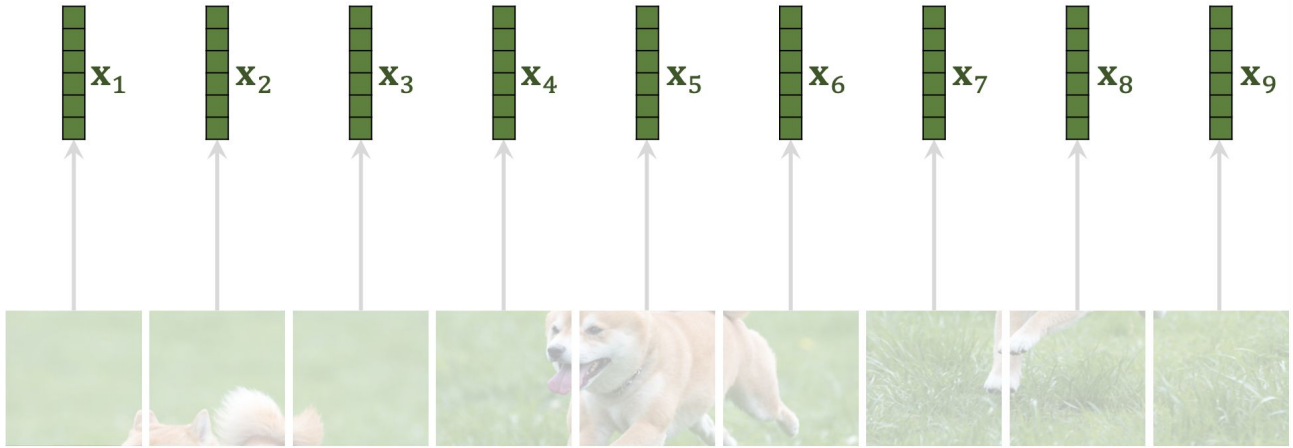
Split image into patches

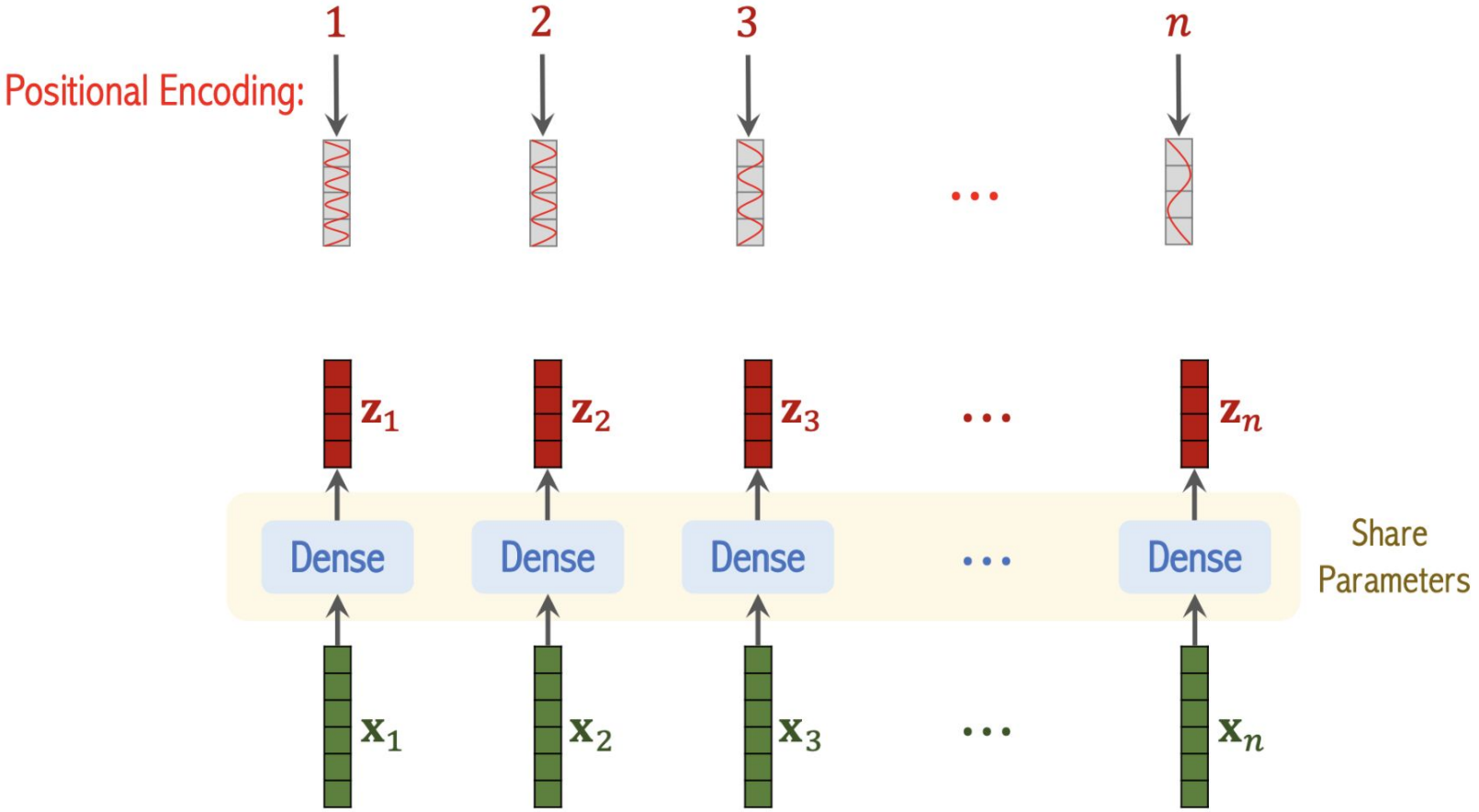


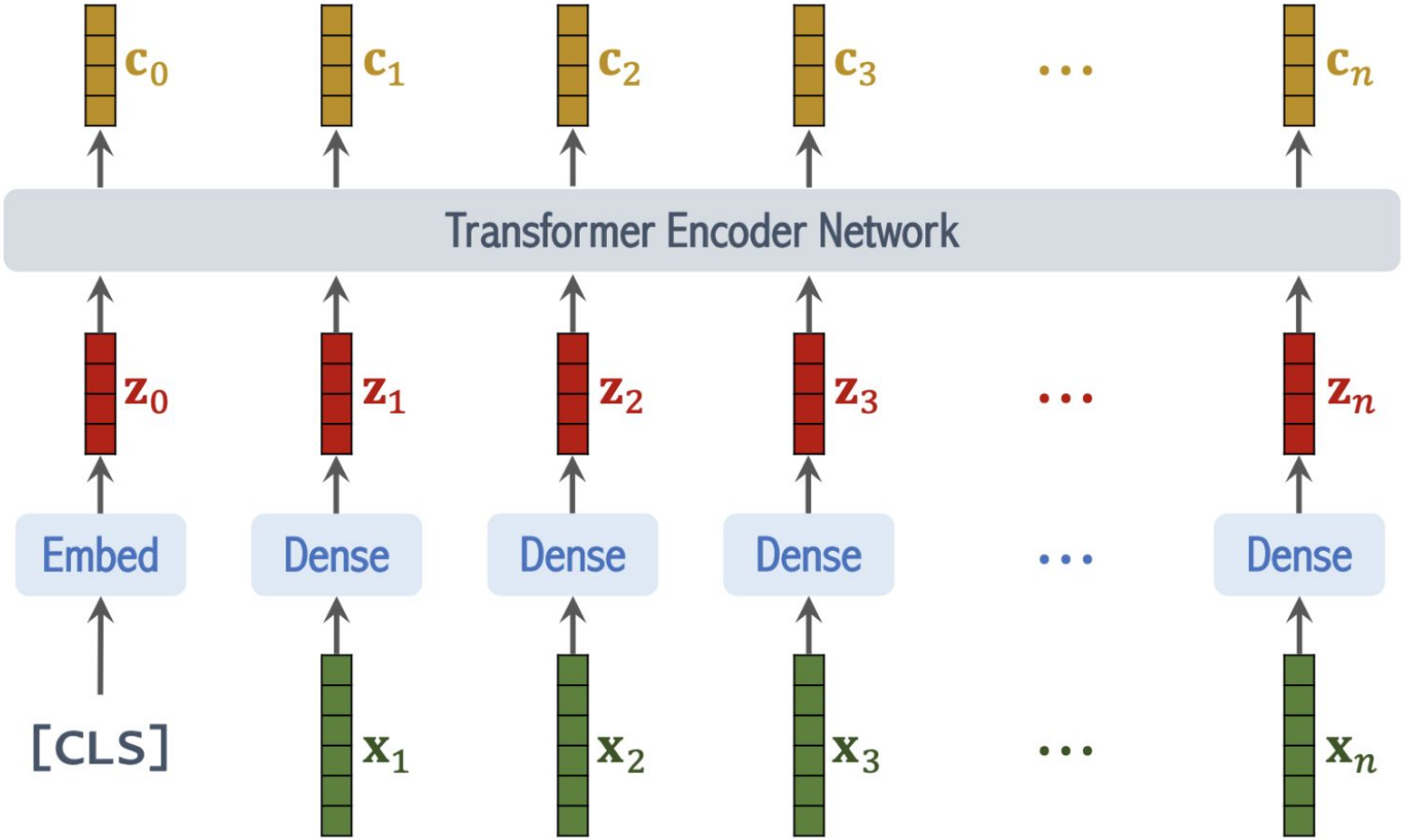
- Here, the patches do not overlap.
- The patches can overlap.
- User specifies:
  - patch size, e.g., 16×16;
  - stride, e.g., 16×16.

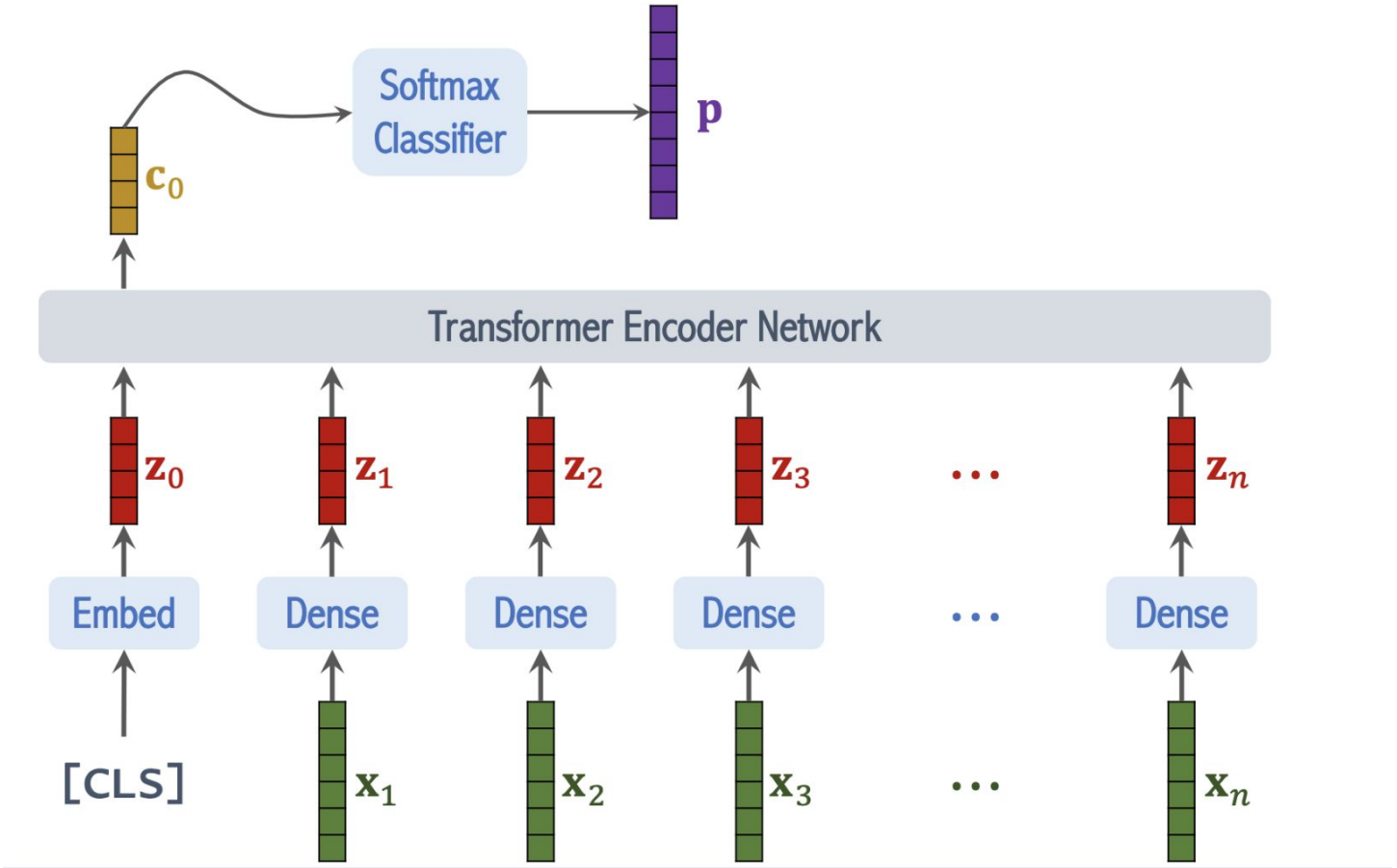
Vectorization

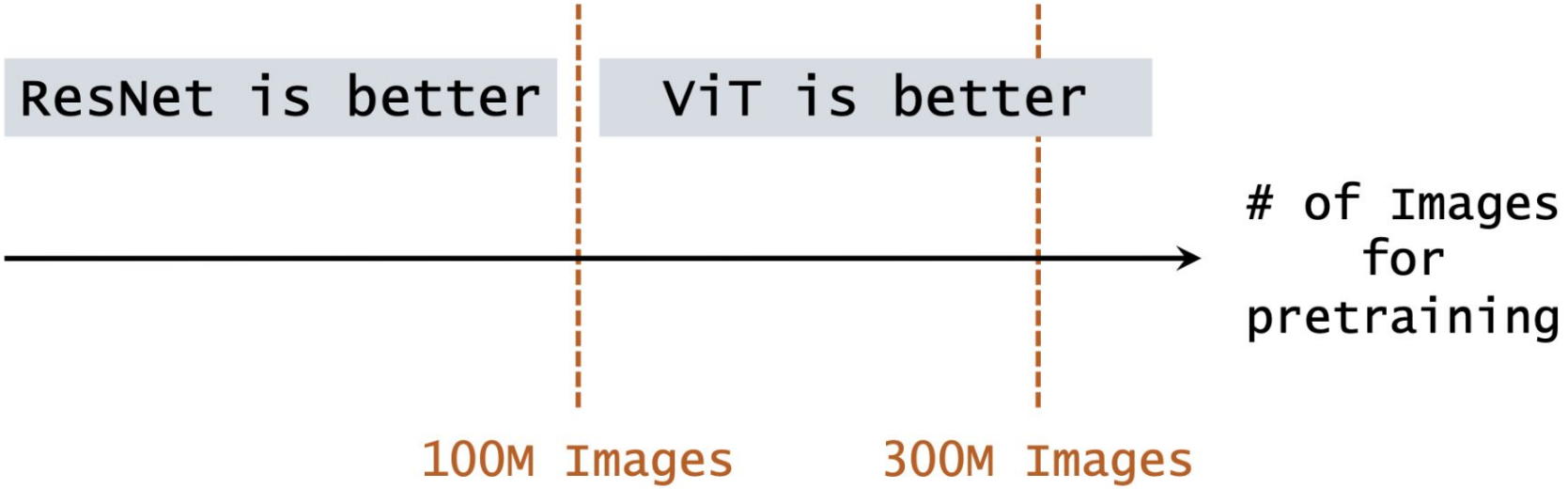
If the patches are $d_1 \times d_2 \times d_3$ tensors, then the vectors are $d_1 d_2 d_3 \times 1$.




Positional Encoding:
1
2
3
n
z1
z2
z3
zn
Dense
Dense
Dense
Dense
Share Parameters
x1
x2
x3
xn


c_0
c_1
c_2
c_3
...
c_n
Transformer Encoder Network
z_0
z_1
z_2
z_3
...
z_n
Embed
Dense
Dense
Dense
...
Dense
[CLS]
x_1
x_2
x_3
...
x_n


Softmax Classifier
p
c_0
Transformer Encoder Network
z_0
z_1
z_2
z_3
z_n
Embed
Dense
Dense
Dense
Dense
[CLS]
x_1
x_2
x_3
x_n


ResNet is better
ViT is better
# of Images for pretraining
100M Images
300M Images

Thank you!