

Feed Forward Neural Networks for NLP

CITS4012 Natural Language Processing

A/Prof. Wei Liu

`wei.liu@uwa.edu.au`

Computer Science and Software Engineering
The University of Western Australia

April 16, 2025



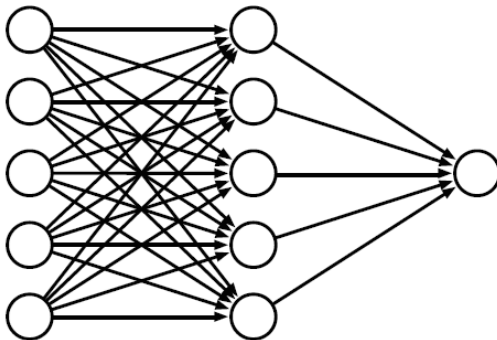
THE UNIVERSITY OF
**WESTERN
AUSTRALIA**

Neural Network Building Blocks



Hidden Layers

Neural networks modify linear models in two important ways. The first is the use of multiple layers. Instead of computing the output value directly from the input values, a hidden layer is introduced.



A Neural Network with One Hidden Layer



Mathematical Notations of A Neural Network

A neural network with a hidden layer consists of

- a vector of input nodes with values $\vec{x} = (x_1; x_2; x_3; \dots; x_n)^T$,
- a vector of hidden nodes with values $\vec{h} = (h_1; h_2; h_3; \dots; h_m)^T$
- a vector of output nodes with values $\vec{y} = (y_1; y_2; y_3; \dots; y_l)^T$
- a matrix of weights connecting input nodes with hidden nodes
 $U = \{u_{ji}\}$
- a matrix of weights connecting hidden nodes with output nodes
 $W = \{w_{kj}\}$

$$h_j = \sum_i x_i u_{ji}$$

$$y_k = \sum_j h_j w_{kj}$$



Non-Linearity through Activation Functions

The introduction of multiple hidden layers still models the output as a linear weighted sum of input.

Activation Function

A salient element of neural networks is the use of a non-linear activation function.

$$\begin{aligned} y_k &= \sum_j h_j w_{kj} \\ &= \sum_j \sum_i x_i u_{ji} w_{kj} \\ &= \sum_i x_i \left(\sum_j u_{ji} w_{kj} \right) \end{aligned}$$

An intuitive way to think about these activation functions is that they segment the range of values for the linear combination s_j into

- a segment where the node is turned off (values close to 0 for sigmoid, or -1 for tanh)
- a transition segment where the node is partly turned on
- a segment where the node is turned on (values close to 1)

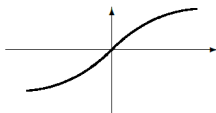


Popular Activation Functions

Hyperbolic tangent

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

output ranges
from -1 to +1



Logistic function

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

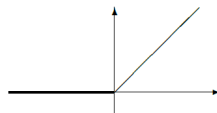
output ranges
from 0 to +1



Rectified linear unit

$$\text{relu}(x) = \max(0, x)$$

output ranges
from 0 to ∞

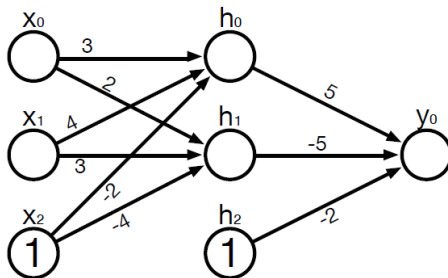


Popular Activation Functions

ReLU and Leaky ReLU

The rectified linear unit (ReLU) is one of the most popular choices, which does not allow for negative values and floors them at 0, but does not alter the value of positive values. It is simpler and faster to compute than $\tanh(x)$ or $\text{sigmoid}(x)$. You could view each hidden node as a feature detector.

Inference using Neural Network



A simple one hidden layer neural network

Layer	Node	Summation	Sigmoid
hidden	h_0	$1 \times 3 + 0 \times 4 + 1 \times (-2) = 1$	0.731
hidden	h_1	$1 \times 2 + 0 \times 3 + 1 \times (-4) = -2$	0.119
output	y_0	$0.731 \times 5 + 0.119 \times (-5) + 1 \times (-2) = 1.060$	0.743

Calculation for input (1,0)



XOR in Neural Network

Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.119	0.018	0.183 $\rightarrow$ 0
0	1	0.881	0.269	0.743 $\rightarrow$ 1
1	0	0.731	0.119	0.743 $\rightarrow$ 1
1	1	0.993	0.731	0.334 $\rightarrow$ 0

Our neural network computes XOR

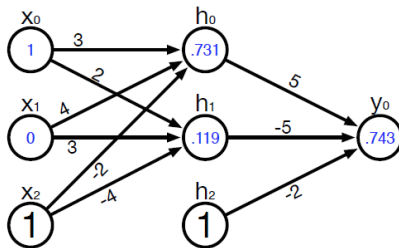
- h_0 acts like the Boolean OR: Its value is high if at least of the two input values is 1.
- h_1 acts like the Boolean AND — it only has a high value if both inputs are 1.
- XOR is effectively implemented as the subtraction of the AND from the OR hidden node.



Back-Propagation

Back-propagation

The most common training method for neural networks is called back-propagation, since it first updates the weights to the output layer, and propagates back error information to earlier layers.



A Neural Network for XOR

Whenever a training example is processed, then for each node in the network, an error term is computed which is the basis for updating the values for incoming weights.

Back-propogation example

Node	Error term	Weight updates
	$\delta = (y_0 - t_0)\sigma'(s_0)$	$\Delta w_{0 \leftarrow j} = \mu \delta h_j$
y_0	$\delta = (1 - 0.743) \times 0.191 = 0.049$	$\Delta w_{0 \leftarrow 0} = \mu \times 0.049 \times 0.731 = 0.036$
		$\Delta w_{0 \leftarrow 1} = \mu \times 0.049 \times 0.119 = 0.006$
		$\Delta w_{0 \leftarrow 2} = \mu \times 0.049 \times 1 = 0.049$
	$\delta_j = \delta w_{i \leftarrow j} \sigma'(z_j)$	$\Delta u_{j \leftarrow i} = \mu \delta_j x_i$
h_0	$\delta_0 = 0.049 \times 5 \times 0.197 = 0.048$	$\Delta u_{0 \leftarrow 0} = \mu \times 0.048 \times 1 = 0.048$
		$\Delta u_{0 \leftarrow 1} = \mu \times 0.048 \times 0 = 0$
		$\Delta u_{0 \leftarrow 2} = \mu \times 0.048 \times 1 = 0.048$
h_1	$\delta_0 = 0.049 \times (-5) \times 0.105 = -0.026$	$\Delta u_{1 \leftarrow 0} = \mu \times (-0.026) \times 1 = -0.026$
		$\Delta u_{1 \leftarrow 1} = \mu \times (-0.026) \times 0 = 0$
		$\Delta u_{1 \leftarrow 2} = \mu \times (-0.026) \times 1 = -0.026$

Weight updates (with unspecified learning rate μ) for the neural network when the training example $(1, 0) \rightarrow 1$ is presented



Loss Functions

Mean Squared Error Loss

For regression problems, the network output ($\hat{y}$) and the ground truth or target (y) are both continuous values, you can choose between:

- $L_{MSE}(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ (`torch.nn.MSELoss()`)
- $L_{MAE}(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$ (`torch.nn.L1Loss()`)
- $L_{RMSE}(\hat{y}, y) = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$

```

1 import torch
2 import torch.nn as nn
3
4 mse_loss = nn.MSELoss()
5 torch.manual_seed(42)
6 outputs = torch.randn(3, 5, requires_grad=True)
7 targets = torch.randn(3, 5)
8 loss = mse_loss(outputs, targets)
9 loss.backward()
10 print(loss)

```

code/MSE.py

tensor(1.0192, grad_fn=<MseLossBackward>)



Root Mean Squared Error Loss

There is no RMSE default implementation of RMSE, let's create one.

```
1 class RMSELoss(nn.Module):  
2     def __init__(self):  
3         super().__init__()  
4         self.mse = nn.MSELoss()  
5  
6     def forward(self, yhat, y):  
7         eps = 1e-7  
8         # In case when mse=0, back propagation will  
9         # return nan  
10        return torch.sqrt(self.mse(yhat, y) + eps)  
11  
12 criterion = RMSELoss()  
13  
14 yhat = torch.randn(3, 5, requires_grad=True)  
15 y = torch.randn(3, 5)  
16 loss = criterion(yhat, y)  
17 print(loss)
```

code/RMSE.py

tensor(1.0096, grad_fn=<SqrtBackward>)



Loss for Binary Classification

A binary classification problem has two class labels, a data point belongs to either

- positive class ($y_i = 1$), or
- negative class ($y_i = 0$)

Note we want to calculate errors to minimize loss.

$$error_i = \begin{cases} \log(P(y_i = 1)) & \text{when } y_i = 1 \\ \log(P(y_i = 0)) = \log(1 - P(y_i = 1)) & \text{when } y_i = 0 \end{cases}$$

Why log? Because when we have a perfect prediction, $\log 1=0$!



Binary Cross-Entropy Loss

For binary classification loss, we often use **binary cross-entropy loss**, which is originated from information theory.

Binary Cross-Entropy (BCE) Loss

$$\begin{aligned} L_{BCE}(\hat{y}, y) &= -\frac{1}{n_{pos} + n_{neg}} \left(\sum_{i=1}^{n_{pos}} \log(P(y_i = 1)) + \sum_{i=1}^{n_{neg}} \log(1 - P(y_i = 1)) \right) \\ &= -\frac{1}{n} \sum_{i=1}^n (y_i \log P(y_i = 1) + (1 - y_i) \log(1 - P(y_i = 1))) \\ &= -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)) \end{aligned}$$



BCE Manual Calculation - Example

```
1 labels = torch.tensor([1.0, 0.0])
2 predictions = torch.tensor([.9, .2])
3 # Positive class (labels == 1)
4 positive_pred = predictions[labels == 1]
5 first_summation = torch.log(positive_pred).sum()
6 # Negative class (labels == 0)
7 negative_pred = predictions[labels == 0]
8 second_summation = torch.log(1 - negative_pred).sum()
9 # n_total = n_pos + n_neg
10 n_total = labels.size(0)
11 loss = -(first_summation + second_summation) / n_total
12 loss
```

code/BCE-pos-neg.py

tensor(0.1643)

```
1 summation = torch.sum(
2     labels * torch.log(predictions) +
3     (1 - labels) * torch.log(1 - predictions))
4 loss = -summation / n_total
5 loss
```

code/BCE-formula.py



BCE Calculation in PyTorch - Example

PyTorch implements the binary cross-entropy loss, `nn.BCELoss`.

```
1 loss_fn = nn.BCELoss(reduction='mean')
2 labels = torch.tensor([1.0, 0.0])
3 predictions = torch.tensor([.9, .2])
4 # RIGHT
5 right_loss = loss_fn(predictions, labels)
6 # WRONG
7 wrong_loss = loss_fn(labels, predictions)
8 print(right_loss, wrong_loss)
```

code/BCE-pytorch.py

tensor(0.1643) tensor(15.0000)

In PyTorch, the order of the arguments for the loss function is important as BCELoss assumes the first argument is the prediction and applies log to it. So remember that convention is $(\hat{y}, y)$.



BCEWithLogitsLoss

BCEWithLogitsLoss takes **logits** as an argument, not probability.

Logit is the inverse of sigmoid (probability)

$$p = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$z = \text{logit}(p) = \log \frac{p}{1 - p}$$

We need to be careful in choosing the right combination of model and loss function.

Option 1 `nn.Sigmoid` as the last layer, meaning your model is producing probabilities, combined with `nn.BCELoss` function

Option 2 no sigmoid in the last layer, meaning your model is producing logits, combined with `nn.BCEWithLogitsLoss` function.



BCEWithLogitsLoss - Example

```
1 import numpy as np
2 import torch.nn as nn
3
4 def log_odds_ratio(prob):
5     return np.log(prob/(1-prob))
6
7 # Work out logit from probability
8 logit1 = log_odds_ratio(.9)
9 logit2 = log_odds_ratio(.2)
10 # BCEWithLogitsLoss() takes logits
11 labels = torch.tensor([1.0, 0.0])
12 logits = torch.tensor([logit1, logit2])
13 print(logits)
14 criterion = nn.BCEWithLogitsLoss()
15 loss = criterion(logits, labels)
16 loss
```

code/BCEWithLogitsLoss.py

```
tensor([ 2.1972, -1.3863], dtype=torch.float64)
tensor(0.1643)
```



From Binary to Multi-class Classification

Recall our BCE loss:

$$L_{BCE}(\hat{y}, y) = -\frac{1}{n_{pos} + n_{neg}} \left(\sum_{i=1}^{n_{pos}} \log(P(y_i = 1)) + \sum_{i=1}^{n_{neg}} \log(1 - P(y_i = 1)) \right)$$

For a three class classification:

$$L_{NNL}(\hat{y}, y) = -\frac{1}{n_0 + n_1 + n_2} \left(\sum_{i=1}^{n_0} \log(P(y_i = 0)) + \sum_{i=1}^{n_1} \log(P(y_i = 1)) + \sum_{i=1}^{n_2} \log(P(y_i = 2)) \right)$$

For a multi-class classification with C classes:

$$L_{NNL}(\hat{y}, y) = -\frac{1}{n_0 + \dots + n_2} \left(\sum_{c=0}^{C-1} \sum_{i=1}^{n_c} \log(P(y_i = c)) \right)$$



Softmax

The softmax function returns, for each class, the contribution that a given class had to the sum of odds ratios.

The softmax is computed using odds ratios instead of log odds ratios (logits), so we need to exponentiate the logits!

$$z = \text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

$$e^z = \text{odd_ratio} = \frac{p}{1-p}$$

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{C-1} e^{z_j}}$$

We have three classes, so our model needs to output three logits (z_0, z_1, z_2). Applying softmax to these logits, we would get:

$$\text{softmax}(z) = \left[\frac{e^{z_0}}{e^{z_0} + e^{z_1} + e^{z_2}}, \frac{e^{z_1}}{e^{z_0} + e^{z_1} + e^{z_2}}, \frac{e^{z_2}}{e^{z_0} + e^{z_1} + e^{z_{C-1}}} \right]$$

LogSoftmax

The logsoftmax function returns the logarithm of the softmax function.

```
1 import torch.nn as nn
2
3 logits = torch.tensor([ 1.3863, 0.0000, -0.6931])
4
5 # Manual softmax,  $e^z$  is odds ratios
6 odds_ratios = torch.exp(logits)
7 softmaxed = odds_ratios/odds_ratios.sum()
8 # PyTorch softmax
9 softmaxed = nn.Softmax(dim=-1)(logits)
10
11 # Manual log_softmax
12 torch.log(softmaxed)
13 # Pytorch log_softmax
14 nn.LogSoftmax(dim=-1)(logits)
```

code/log_softmax.py

Instead of manually taking the logarithm, PyTorch provides `torch.nn.functional.log_softmax` and `torch.nn.LogSoftmax` out of the box. These functions are faster and also have better numerical properties

Negative Log-Likelihood Loss Manual Computation

Recall that softmax $\rightarrow$ probabilities, LogSoftmax $\rightarrow$ log-probabilities.

Negative Log-Likelihood Loss

NLLLoss is the average of the negated log probability of each data point relevant to the class it belongs to.

```
1 torch.manual_seed(11)
2 logits = torch.randn((5, 3))
3 labels = torch.tensor([0, 0, 1, 2, 1])
4 log_probs = F.log_softmax(logits, dim=-1)
5 log_probs
6 # tensor([[-1.5229, -0.3146, -2.9600],
7 #         [-1.7934, -1.0044, -0.7607],
8 #         [-1.2513, -1.0136, -1.0471],
9 #         [-2.6799, -0.2219, -2.0367],
10 #         [-1.0728, -1.9098, -0.6737]])
11 # for each data point, select the log_probs of its class
12 indices = torch.tensor([[idx] for idx in labels])
13 log_probs_cls = log_probs.gather(1, indices)
14 NLLLoss = -log_probs_cls.mean()
```

code/NLLLoss_manual.py



Negative Log-Likelihood Loss in PyTorch

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 torch.manual_seed(11)
6 logits = torch.randn((5, 3))
7 labels = torch.tensor([0, 0, 1, 2, 1])
8 log_probs = F.log_softmax(logits, dim=-1)
9
10 loss_fn = nn.NLLLoss()
11 loss_fn(log_probs, labels)
12 # Dealing with unbalanced dataset
13 loss_fn = nn.NLLLoss(weight=torch.tensor([1., 1., 2.]))
14 loss_fn(log_probs, labels)
15 # Ignoring a class in the loss calculation
16 loss_fn = nn.NLLLoss(ignore_index=2)
17 loss_fn(log_probs, labels)
```

code/NLLLoss_Pytorch.py



Cross Entropy Loss

- The NLLLoss function took log-probabilities as an argument (together with the labels).
- Cross Entropy Loss takes Logits directly. It is the multiclass version of `nn.BCEWithLogitsLoss`.

What does it mean, in practical terms?

It means you should NOT add a logsoftmax as the last layer of your model when using Cross Entropy Loss function. This loss combines the logsoftmax layer with the negative log-likelihood loss into one.

```
1 torch.manual_seed(11)
2 logits = torch.randn((5, 3))
3 labels = torch.tensor([0, 0, 1, 2, 1])
4 loss_fn = nn.CrossEntropyLoss()
5 loss_fn(logits, labels)
```

[code/cross-entropy.py](#)



Classification Loss Function Summary

	BCE	BCE Logits	NLL	Cross-Entropy
Classification	binary	binary	multiclass	multiclass
Input (each data point)	probability	logit	array of log probabilities	array of logits
Label (each data point)	float (0.0 or 1.0)	float (0.0 or 1.0)	long (class index)	long (class index)
Model's last layer	Sigmoid		LogSoft-max	
weight argument	not class weights	not class weights	class weights	class weights
pos_weight argument	n/a	"weighted" loss	n/a	n/a



Computation Graphs



Why computation graphs?

We have seen that in PyTorch, `.backward()` function magically keeps track of the gradients of all parameters and any intermediate variables that the parameters are involved in.

This is because that Neural Networks are viewed as Computation Graphs.

Computation Graphs

Computation graphs are any arbitrary connected operations between an input and any number of parameters. The graph does not have to have a nice tree structure, but may be any **acyclical directed graph**, i.e., anything goes as long there is a straightforward processing direction and no cycles.



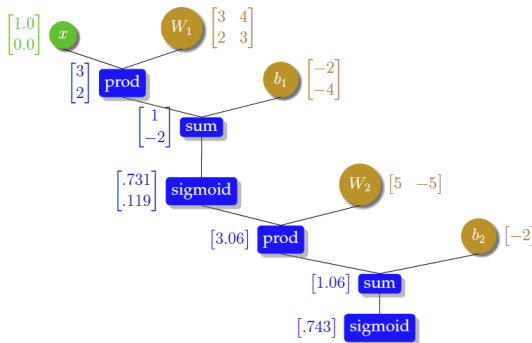
Computation Graph - Forward Pass

For a simple neural network that we have used for XOR, we have

$$h = \text{sigmoid}(W_1x + b_1)$$

$$y = \text{sigmoid}(W_2h + b_2)$$

The computation graph is therefore:



Computation Graph - The chain rule

$$e = L2(y; t)$$

$$y = \text{sigmoid}(s)$$

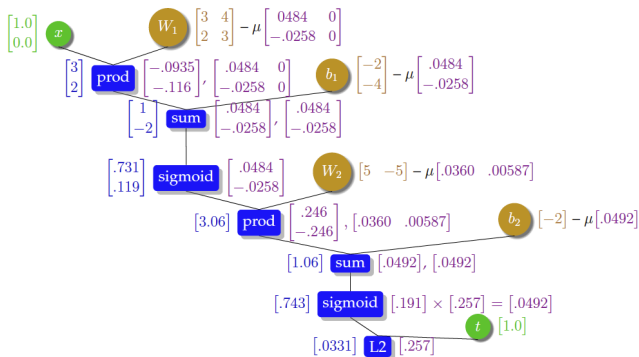
$$s = \text{sum}(p; b_2)$$

$$p = \text{prod}(h; W_2)$$

$$\begin{aligned} \frac{\partial L2(W_2)}{\partial W_2} &= \frac{\partial L2(\text{sigmoid}(\text{sum}(\text{prod}(h; W_2); b_2)); t)}{\partial W_2} \\ &= \frac{\partial e}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial p} \cdot \frac{\partial p}{\partial W_2} \\ &= \frac{\partial L2(y; t)}{\partial y} \cdot \frac{\partial \text{sigmoid}(s)}{\partial s} \cdot \frac{\partial \text{sum}(p; b_2)}{\partial p} \cdot \frac{\partial \text{prod}(h; W_2)}{\partial W_2} \end{aligned}$$



Computation Graph - Backward Pass



Computation graph with gradients computed in the backward pass for the training example $(0; 1)^T \rightarrow 1.0$. Gradients are computed with respect to the input of the nodes, so some nodes that have two inputs also have two gradients.



Dynamic Computation Graphs in PyTorch

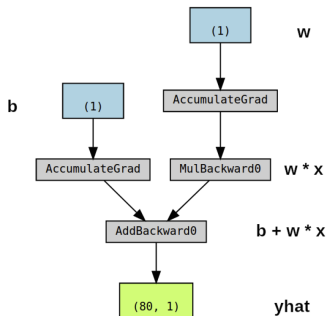
PyTorch uses Dynamic Computation Graphs to keep track of the gradient calculation in backpropagation.

```
1 from torchviz import make_dot
2 # Step 0 - Initializes parameters "b" and "w" randomly
3 torch.manual_seed(42)
4 b = torch.randn(1, requires_grad=True, \
5                 dtype=torch.float, device=device)
6 w = torch.randn(1, requires_grad=True, \
7                 dtype=torch.float, device=device)
8 # Step 1 - Computes our model's predicted output
9 #         - forward pass
10 yhat = b + w * x_train_tensor
11 # Step 2 - Computes the loss
12 error = (yhat - y_train_tensor)
13 loss = (error ** 2).mean()
14 # We can try plotting the graph for any python variable:
15 # yhat, error, loss...
16 make_dot(yhat)
```

code/computation-graph.py



Computational Graph Visualisation in TorchViz



- **blue boxes** $((1)s)$: correspond to the tensors we use as parameters, the ones we're asking PyTorch to compute gradients for.
- **gray boxes** (`MulBackward0` and `AddBackward0`): a Python operation that involves a gradient-computing tensor or its dependencies
- **green box** $((80, 1))$: the tensor used as the starting point for the computation of gradients (assuming the `backward()` method is called from the variable used to visualize the graph) — they are computed from the bottom-up in a graph.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs)

Like feedforward networks, RNNs are a family of neural models that can be trained to represent a dataset through learning weights.

Recurrent neural networks (RNNs) are perfectly suited for sequence problems because:

- RNN take advantage of the underlying structure of the sequential data, namely, the order of the data points.
- The input data points, sequentially presented to a RNN, modify the RNNs internal (hidden) state.
- After training, the hidden state will ultimately be a representation of the full sequence.

Recurrent neural networks are all about producing a hidden state that best represents a sequence.

What is a Hidden State

A **hidden state** is simply a vector.

- You need to specify the size of the vector, i.e. the number of hidden dimensions.

Let's create a two dimensional hidden state just for illustration:

```
1 hidden_state = torch.zeros(2)
2 hidden_state
```

code/hidden_state.py

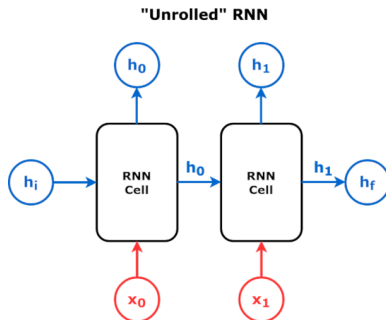
tensor([0., 0.])

Despite its simplicity, this is actually a fine example of an initial hidden state, as we can see in the next slide.

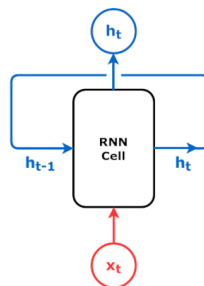


Top Level Representation of RNN

There are several different members in the RNN family, but the most basic form is called the **Elman RNN**, sometimes referred to informally as the “vanilla” RNN.



Recurrent Neural Network



Elman RNN Data Flow

The “unrolled” RNN shows a sequence of two data points being fed to an RNN.

- 1 an initial hidden state (h_i) represent the state of the empty sequence, usually initialized with zeros (like the one we created before).
- 2 an RNN cell takes two inputs: the **hidden state representing the state of the sequence so far**, and a **data point** from the sequence (like the embedding of a word).
- 3 the two inputs are used to produce a new hidden state (h_0 for the first data point), representing the updated state of the sequence now that a new data point was presented to it.
- 4 the new hidden state is both the **output** of the current step and **one of the inputs** to the next step.
- 5 if there is yet another data point in the sequence, it goes back to Step 2; if not, the last hidden state (h_1 in the figure above) is also the final hidden state (h_f) of the whole RNN.



RNN vs. CNN

Similar to CNN, RNN is a featurizer

We run a sequence of data points through RNN cells and use the final hidden state (also a vector) as features for a classifier.

Recall that in CNN, we run the words or pixels through

- multiple convolutional blocks (convolutional layer + pooling + activation) and
- flatten them into a vector at the end to use them as features for a classifier.

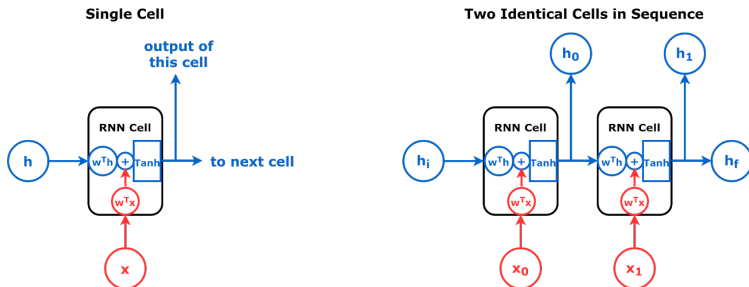
There is a fundamental difference between CNNs and RNNs

CNN has several different convolutional layers, each learning its own filters. The RNN cell is **one and the same**.

The “unrolled” representation is misleading: it looks like each input is being fed to a different RNN cell, but that’s not the case.



Internals of an RNN Cell



- a linear layer to transform the hidden state (in blue) using **hidden-to-hidden weights** W_{hh} ;
- a linear layer to transform the data point from the sequence (in red) using **input-to-hidden weights** W_{ih} ;
- an activation function, usually the hyperbolic tangent ($\tanh$), applied to the sum of the **transformed hidden state** (t_h) and **transformed input vector** (t_i).



RNN - ONE cell over and over again!

The **not “unrolled”** (“rolled” doesn’t sound right!) representation is a better characterization of the internal structure.

There is no other cell, only one set of parameters to learn!

RNN Math Equations

$$t_h = W_{hh}h_{t-1} + b_{hh}$$

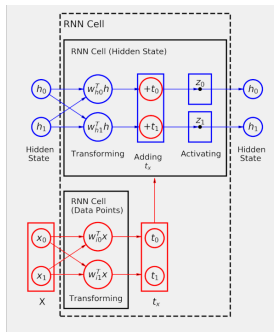
$$t_x = W_{ih}x_t + b_{ih}$$

$$h_t = \tanh(t_h + t_x)$$

There is no restrictions on the sequence size!



RNN Cell at Neuron Level



- The number of red neurons needs to be the same as the chosen number of hidden dimensions since both transformed outputs need to be added together.
- But the data points (x_t) does not need to have the same number of dimensions as the hidden state.

RNN Cell in PyTorch

```
1 n_features = 2
2 hidden_dim = 2
3
4 torch.manual_seed(19)
5 rnn_cell = nn.RNNCell(input_size=n_features, hidden_size=
    hidden_dim)
6 rnn_cell.state_dict()
```

code/RNNCell.py



Why hyperbolic tangent? Isn't ReLU a better activation function?

The hyperbolic tangent has a “competitive advantage” here

- It maps the feature space to clearly defined boundaries: the interval $(-1, 1)$.
- This guarantees that, at every step of the sequence, the hidden state is always within these boundaries.
- Given that we have only one linear layer to transform the hidden state, regardless of which step of the sequence it is being used in, it is definitely convenient to have its values within a predictable range.

RNN Layer

In PyTorch, the `nn.RNN` layer takes care of the hidden state handling, no matter how long the input sequence is.

- This is the layer we'll be actually using in the model.
- The fully-fledged RNN offers many more options (stacked and/or bidirectional layers, for instance).

PyTorch RNN's main arguments are the same as RNN Cell:

- `input_size`: the number of features in each data point of the sequence, e.g. word embedding size
- `hidden_size`: the number of hidden dimensions you want to use
- `bias`: just like any other layer, it includes the bias in the equations
- `nonlinearity`: by default, it uses the hyperbolic tangent, but you can change it to ReLU or ELU if preferred.



RNN Cell vs. RNN Layer

```

1 n_features = 2
2 hidden_dim = 2
3
4 torch.manual_seed(19)
5 rnn_cell = nn.RNNCell(input_size
    =n_features, hidden_size=
    hidden_dim)
6 rnn_cell.state_dict()

```

code/RNNCell.py

```

1 n_features = 2
2 hidden_dim = 2
3
4 torch.manual_seed(19)
5 rnn = nn.RNN(input_size=
    n_features, hidden_size=
    hidden_dim)
6 rnn.state_dict()

```

code/RNN.py

```

OrderedDict([('weight_ih',
    tensor([[ 0.6627, -0.4245],
            [ 0.5373,  0.2294]])),
('weight_hh',
    tensor([[ -0.4015, -0.5385],
            [-0.1956, -0.6835]])),
('bias_ih', tensor([0.4954, 0.6533])),
('bias_hh', tensor([-0.3565, -0.2904]))])

```

```

OrderedDict([('weight_ih_10',
    tensor([[ 0.6627, -0.4245],
            [ 0.5373,  0.2294]])),
('weight_hh_10',
    tensor([[ -0.4015, -0.5385],
            [-0.1956, -0.6835]])),
('bias_ih_10', tensor([0.4954, 0.6533])),
('bias_hh_10', tensor([-0.3565, -0.2904]))])

```



Sequence to Sequence Learning with Attention

CITS4012 Natural Language Processing

A/Prof. Wei Liu

`wei.liu@uwa.edu.au`

Computer Science and Software Engineering
The University of Western Australia

April 16, 2025



THE UNIVERSITY OF
**WESTERN
AUSTRALIA**

What we are going to cover today

1 Attention

- Attention Mechanism
- Attention Score Calculation

2 Multi-headed Attention

3 Self-Attention

- Encoder with Self-Attention
- Encoder with self- and cross-attentions
- Decoder with Masked Self-Attention
- Encoder Decoder with Self Attention

[see Deep Learning with Pytorch Step by Step, Chapter 9]



Attention



Attention Mechanism in A Nutshell

The decoder's role is to generate the translated words.

Paying **attention** to different elements in the source sequence

If the decoder is allowed to choose which hidden states from the encoder it will use to generate each output, it means it can choose which English words it will use to generate each translated word.

English: the European economic zone

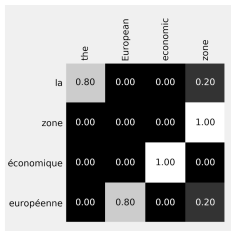
French: la zone économique européenne

- From the 1a to 1a, the decoder needs to "pay attention" to the last English word zone, to determine if it is a **singular feminine noun**.
- For the second French word zone, the decoder should again "pay more attention" to the last English word zone.



Attention Score Matrix

These weights, or **alphas**, are the **attention scores**.



*The numbers above are made-up and for illustration purpose only.

Since the translated "la" is based on "the" and "zone", we can guess that a sensible decoder would

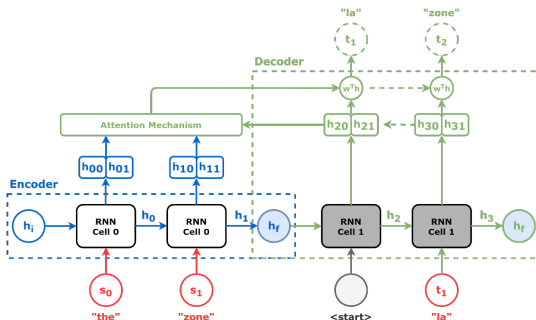
- assigned 80% of its attention to the definite article, and
- the remaining 20% of its attention to the corresponding noun to determine its gender.

The more relevant to the decoder a hidden state from the encoder is, the higher the score.

Encoder-Decoder with Attention

The special token indicating the start of the target sequence

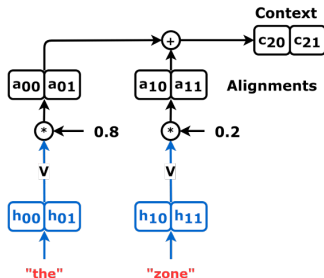
In this translation example, the source and target sequences are **independent**. The first input of the decoder is **not the last element of the source sequence**, but the special token that indicates the start of a new sequence.



Encoder-Decoder with Attention for Translation



Instead of generating predictions solely based on its own hidden states, the decoder will recruit the attention mechanism to help it decide which parts of the source sequence it must pay attention to.



The attention mechanism informed the decoder it should pay 80% of its attention to the encoder's hidden state corresponding to the word "the", and the remaining 20%, to the word "zone".

Let's get the terminology right first

Values

"Values" (V) refer to the encoder's hidden states (or their *affine transformations*).

Alignment Vector

The resulting multiplication of a "value" by its corresponding attention score is called an alignment vector.

$$context\ vector = \underbrace{\alpha_0 * h_0}_{alignment\ vector_0} + \underbrace{\alpha_1 * h_1}_{alignment\ vector_1} = 0.8 * value_{the} + 0.2 * value_{zone}$$

Context Vector

The sum of all alignment vectors (that is, the weighted average of the hidden states) is called a context vector.

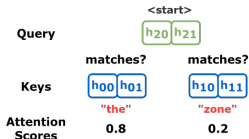
Where do the attention scores come from

The attention scores are based on matching each hidden state of the decoder (h2) to every hidden state of the encoder (h0 and h1).

- Some of them will be good matches (higher attention scores)
- Some others will be poor matches (low attention scores)

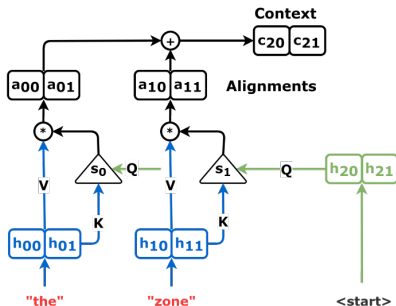
"Keys" and "Queries"

- The encoder's hidden states are called "keys" (K)
- The decoder's hidden state is called a "query" (Q)



An analogical idea is: the **encoder** works like a *key-value store* that the **decoder** can **query**.

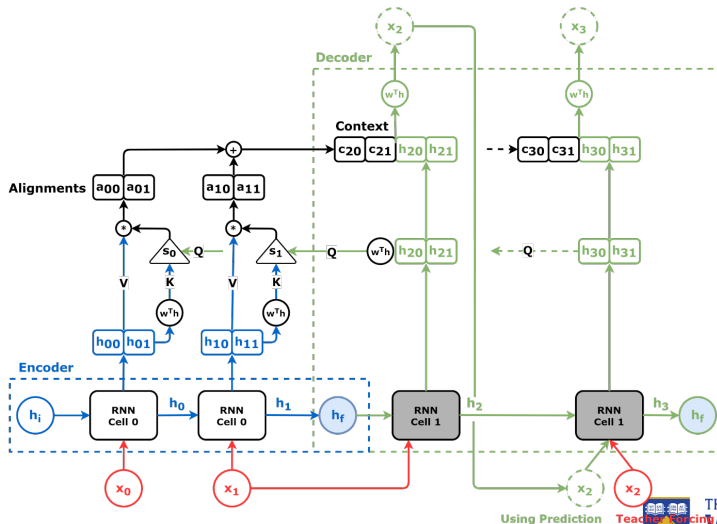
How to match a given "query" (Q) to the "keys" (K)



- The "query" (Q) is matched to both "keys" (K) to compute the attention scores (s) used to compute the context vector (C)
- The context vector (C) is simply the weighted average of the "values" (V).



Encoder + Decoder + Attention



Computing the Context Vector

- 1 The source sequence is the input of the encoder, and the hidden states it outputs become
 - "values" (V), and
 - affine transformed into and "keys" (K)
- 2 The encoder-decoder dynamics stay exactly the same: despite that we are sending the entire sequence of hidden states to the decoder, we still use the encoder's final hidden state as the decoder's initial hidden state.
- 3 In the case of a continuous sequence prediction problem, we still use the last element of the source sequence as input to the first step of the decoder.
- 4 The first "query" (Q) is the decoder's hidden state, affine transformed.
- 5 The attention score (s) is worked out from keys (K) and query (Q).
- 6 The "values" is then scaled to obtain the alignment vector (α).
- 7 The context vector (C) is then concatenated with the hidden state vector from the Decoder to obtain the final prediction after affine transformation.



The Scoring Method

The role of the scoring method

The scoring method needs to determine if two vectors are a good match or not, or, phrased differently, it needs to determine if two vectors (K and Q) are similar or not.

As both K and Q are vectors of the same dimension (i.e. the size of the hidden state), cosine similarity is the most natural way measuring their similarity.

$$\cos \theta = \frac{\sum_i q_i k_i}{\sqrt{\sum_j q_j^2} \sqrt{\sum_j k_j^2}}$$

$$\cos \theta \sqrt{\sum_j q_j^2} \sqrt{\sum_j k_j^2} = \sum_i q_i k_i$$

Since cosine similarity does not consider the norm (magnitude/length) of the vectors, we use the dot product.

$$\cos \theta \|Q\| \|K\| = Q \cdot K$$



Many choices of scoring functions

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \text{cosine}[\mathbf{s}_t, \mathbf{h}_i]$	Graves2014
Additive(*)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_t; \mathbf{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

The dot product is one of the most common ways to compute alignment (and attention) scores, but it is not the only one.

For more information on different mechanisms, and attention in general, please refer to Lilian Weng's amazing blog post



Scaled Scoring Function

- Dot product is the sum of element-wise multiplication of two vectors. The longer the vectors, the bigger the variance.
- We standardize by scaling the dot product by the inverse of its standard deviation, given by the square root of the dimension of the source hidden state.
- On average, the variance equals the number of dimensions.

$$scaled\ dot\ product = \frac{Q \cdot K}{\sqrt{dim_k}}$$

The softmax function ensures that scores add up to 1.

```
1 def calc_alphas(ks, q):
2     dims = q.size(-1)
3     # N, 1, H x N, H, L -> N, 1, L
4     products = torch.bmm(q, ks.permute(0, 2, 1))
5     scaled_products = products / np.sqrt(dims)
6     alphas = F.softmax(scaled_products, dim=-1)
7     return alphas
```

code/score_function.py



Source Mask

When we pad our sequences to have equal length, we want the attention mechanism to ignore those padded data points.

The **source mask** should be **False** for every padded data point, and its shape should be (N, 1, L) where L is the length of the source sequence.

```
1 source_seq = torch.tensor([[-1., 1.], [0., 0.]])
2 # pretend there's an encoder here...
3 keys = torch.tensor([[-.38, .44], [.85, -.05]])
4 query = torch.tensor([[-1., 1.]])
5
6 source_mask = (source_seq != 0).all(axis=2).unsqueeze(1)
7 source_mask # N, 1, L
```

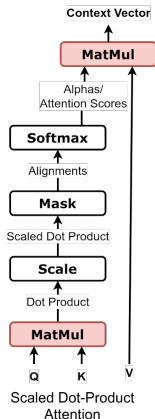
code/mask.py

```
tensor([[[ True, False]]])
```



Attention in PyTorch code I

The job of the attention layer is to calculate context vector as shown below:



```
1 class Attention(nn.Module):
2     def __init__(self, hidden_dim, input_dim=None,
3                 proj_values=False):
4         super().__init__()
5         self.d_k = hidden_dim
6         self.input_dim = hidden_dim
7             if input_dim is None
8             else input_dim
9         self.proj_values = proj_values
10        # Affine transformations for Q, K, and V
11        self.linear_query = nn.Linear(self.input_dim,
12                                     hidden_dim)
13        self.linear_key = nn.Linear(self.input_dim,
14                                   hidden_dim)
15        self.linear_value = nn.Linear(self.input_dim,
16                                    hidden_dim)
17        self.alphas = None
```



```

43 def forward(self, query, mask=None):
44     # Query is batch-first N, 1, H
45     scores = self.score_function(query) # N, 1, L
46     if mask is not None:
47         scores = scores.masked_fill(mask == 0, -1e9)
48     alphas = F.softmax(scores, dim=-1) # N, 1, L
49     self.alphas = alphas.detach()
50
51
52     # N, 1, L x N, L, H -> N, 1, H
53     context = torch.bmm(alphas, self.values)
54     return context

```

code/attention.py

Decoder with Attention in PyTorch code I

```

1 class DecoderAttn(nn.Module):
2     def __init__(self, n_features, hidden_dim):
3         super().__init__()
4         self.hidden_dim = hidden_dim
5         self.n_features = n_features
6         self.hidden = None
7         self.basic_rnn = nn.GRU(self.n_features,
8                                 self.hidden_dim, batch_first=True)
9         self.attn = Attention(self.hidden_dim)
10        self.regression = nn.Linear(2 * self.hidden_dim,
11                                    self.n_features)
12
13    def init_hidden(self, hidden_seq):
14        # the output of the encoder is N, L, H
15        # and init_keys expects batch-first as well
16        self.attn.init_keys(hidden_seq)
17        hidden_final = hidden_seq[:, -1:]
18        self.hidden = hidden_final.permute(1, 0, 2) # L, N, H

```

Decoder with Attention in PyTorch code II

```

22  def forward(self, X, mask=None):
23      # X is N, 1, F
24      batch_first_output, self.hidden =
25          self.basic_rnn(X, self.hidden)
26
27      query = batch_first_output[:, -1:]
28      # Attention
29      context = self.attn(query, mask=mask)
30      concatenated = torch.cat([context, query], axis=-1)
31      out = self.regression(concatenated)
32
33      # N, 1, F
34      return out.view(-1, 1, self.n_features)

```

code/decoder_attn.py



Visualising Attention Scores I

Our former `EncoderDecoder` class works seamlessly with an instance of `DecoderAttn`. We can however modify it to visualise the attention scores.

The easiest way is to create a new class that inherits from `EncoderDecoder` and to override the `init_outputs` and `store_outputs` methods

```

1 class EncoderDecoderAttn(EncoderDecoder):
2     def __init__(self, encoder, decoder,
3                 input_len, target_len,
4                 teacher_forcing_prob=0.5):
5         super().__init__(encoder, decoder, input_len,
6                         target_len, teacher_forcing_prob)
7         self.alphas = None
8
9     def init_outputs(self, batch_size):
10        device = next(self.parameters()).device
11        # N, L (target), F
12        self.outputs = torch.zeros(batch_size,
```

Visualising Attention Scores II

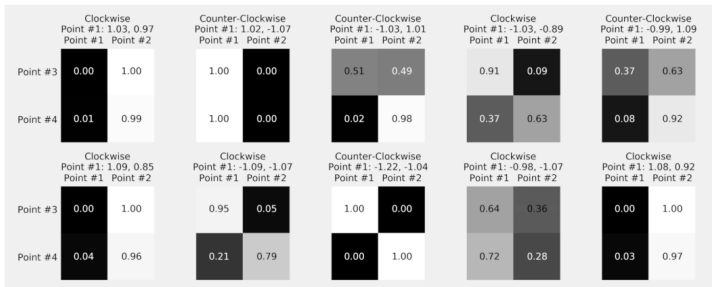
```
13         self.target_len,  
14         self.encoder.n_features)  
15         .to(device)  
16     # N, L (target), L (source)  
17     self.alphas = torch.zeros(batch_size,  
18                               self.target_len,  
19                               self.input_len)  
20                               .to(device)  
21  
22     def store_output(self, i, out):  
23         # Stores the output  
24         self.outputs[:, i:i+1, :] = out  
25         self.alphas[:, i:i+1, :] = self.decoder.attn.alphas
```

code/EncoderDecoderAttn.py



Visualising Attention Scores III

<i>target</i>	<i>source</i>		$\Rightarrow$	<i>target</i>	<i>source</i>	
	x_0	x_1			x_0	x_1
x_2	α_{20}	α_{21}		x_2	0.0011	0.9989
x_3	α_{30}	α_{31}		x_3	0.0146	0.9854



Multi-headed Attention

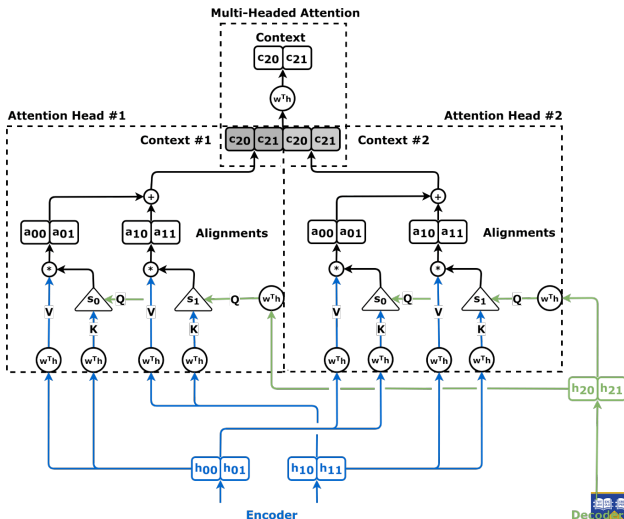
Multi-headed Attention

We can use several attention mechanisms at once, each one being referred to as an **attention head**.

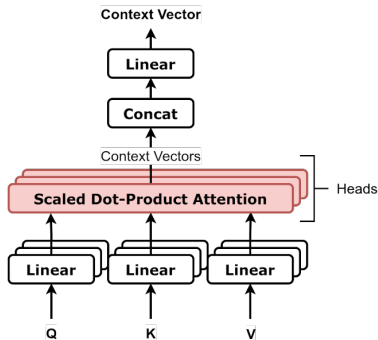
- 1 Each attention head will output its own context vector,
- 2 The context vectors will all get concatenated together and combined using a linear layer.
- 3 In the end, the multi-headed attention mechanism will still output a single context vector.



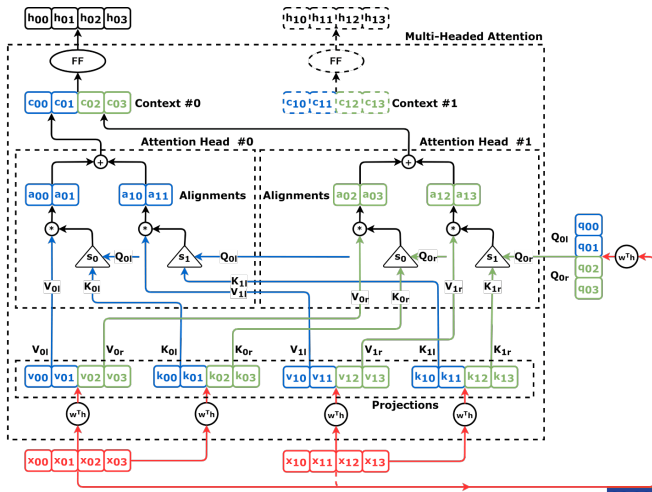
Multi-headed attention flow



A simplified illustration of Multi-headed Attention



Narrow Attention with Two Heads



Wide vs. Narrow Attention

Wide Attention

Each attention head gets the full "hidden state" and produces a context vector of the same size.

- **Pros:** likely yield better models compared to using narrow attention on the same number of dimensions
- **Cons:** cannot deal with large dimensions

Narrow Attention

Each attention head will get a chunk of the affine transformation of the "hidden state" to work with. Note it is not a chunk of the original "hidden state", but of its transformation.

- Pros: can deal with large dimensions
- Cons: may not be as good as Wide Attention, but Transformer uses narrow attention.



Multi-headed Attention in Code I

```
1 class MultiHeadAttention(nn.Module):
2     def __init__(self, n_heads, d_model,
3                   input_dim=None, proj_values=True):
4         super().__init__()
5         self.linear_out = nn.Linear(n_heads*d_model, d_model)
6         self.attn_heads = nn.ModuleList(
7             [Attention(d_model, input_dim=input_dim,
8                       proj_values=proj_values)
9              for _ in range(n_heads)])
10
11     def init_keys(self, key):
12         for attn in self.attn_heads:
13             attn.init_keys(key)
14
15     @property
16     def alphas(self):
17         # Shape: n_heads, N, 1, L (source)
18         return torch.stack([attn.alphas
19                             for attn in self.attn_heads], dim=0)
20
21     def output_function(self, contexts):
```

Multi-headed Attention in Code II

```
22     # N, 1, n_heads * D
23     concatenated = torch.cat(contexts, axis=-1)
24     # Linear transf. to go back to original dimension
25     out = self.linear_out(concatenated) # N, 1, D
26     return out
27
28     def forward(self, query, mask=None):
29         contexts = [attn(query, mask=mask)
30                     for attn in self.attn_heads]
31         out = self.output_function(contexts)
32         return out
```

code/multi-headed.py



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Attention Is All You Need

Some Radical Notion!

what about replacing the recurrent layer with an attention mechanism?

- This is the main proposition of the famous “Attention Is All You Need” from Google Brain [2].
- The 2017 paper introduced the **Transformer** architecture, based on a self attention mechanism, that has completely dominate the NLP landscape and now take over Computer Vision.

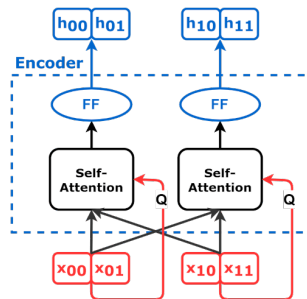
With self-attention, We can use another, separate, attention mechanism to replace the encoder and the decoder too!)

“I pity the fool using recurrent layers.” 😊

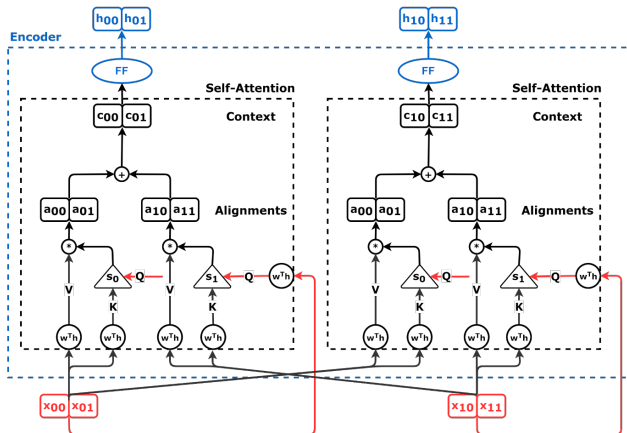


Encoder with Self-Attention

- the source sequence (in red) works as “keys” (K), “values” (V), and “queries” (Q)” as well.
- Now we have “queries”, plural, instead of “query”.
 - In the encoder, each data point is a “query” (the red arrow entering the self-attention mechanism from the side), and
 - each produces its own context vector using alignment vectors for every data point in the source sequence, including itself.
 - This means it is possible for a data point to generate a context vector that is only paying attention to itself.



Encoder with Self-Attention Details



Encoder with Self-Attention Details Explained

- 1 The transformed coordinates of the first data point (x_{00}, x_{01}) are used as “query” (Q).
- 2 This “query” (Q) is paired, independently, with each one of the two “keys” (K).
- 3 One of the keys (K) corresponding to a different transformation of the same input data point (x_{00}, x_{01}) , the other a transformation of the second data point (x_{10}, x_{11}) .
- 4 The pairing will result in two attention scores (alphas) that, multiplied by their corresponding “values” (V), are added up to become the context vector.

$$\alpha_{00}, \alpha_{01} = \text{softmax}\left(\frac{Q_0 \cdot K_0}{\sqrt{2}}, \frac{Q_0 \cdot K_1}{\sqrt{2}}\right)$$

$$\text{context vector}_0 = \alpha_{00} V_0 + \alpha_{01} V_1$$

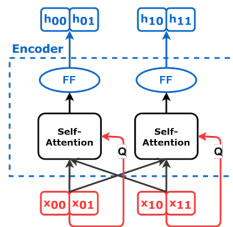
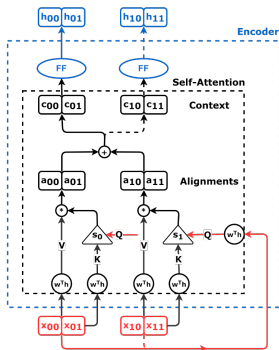
- 5 Next, the context vector goes through the feed-forward network, and the first hidden state is born

A Simplified Illustration

Context Vector - A function of Q

The context vector (and thus the “hidden state”) associated with a data point is basically a function of the corresponding “query” (Q).

This is because everything else (“keys” (K), “values” (V), and the parameters of the self-attention mechanism) is held constant for all queries.

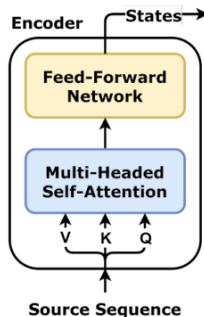


Simplified further by hiding internals



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Even more Simplified Illustration for Self-Attention



One great news is that the process is not sequential!

These operations can be parallelized to generate all “hidden states” at once, which is much more efficient than using a recurrent layer that is sequential in nature.

Encoder with Self Attention in code I

```
1 class EncoderSelfAttn(nn.Module):
2
3
4     def __init__(self, n_heads, d_model, ff_units,
5                     n_features=None):
6
7         super().__init__()
8         self.n_heads = n_heads
9         self.d_model = d_model
10        self.ff_units = ff_units
11        self.n_features = n_features
12        self.self_attn_heads = MultiHeadAttention(
13            n_heads, d_model,
14            input_dim=n_features)
15
16        self.ffn = nn.Sequential(
17            nn.Linear(d_model, ff_units),
18            nn.ReLU(),
19            nn.Linear(ff_units, d_model),
20        )
21
```

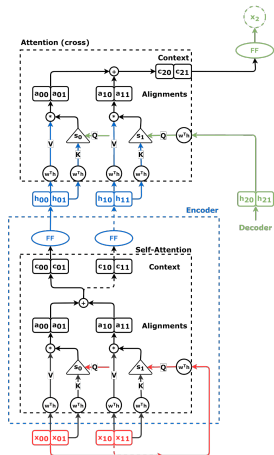
Encoder with Self Attention in code II

```
22
23     def forward(self, query, mask=None):
24         self.self_attn_heads.init_keys(query)
25         att = self.self_attn_heads(query, mask)
26         out = self.ffn(att)
27         return out
```

code/encoder_self_attn.py



Encoder with self- and cross-attentions

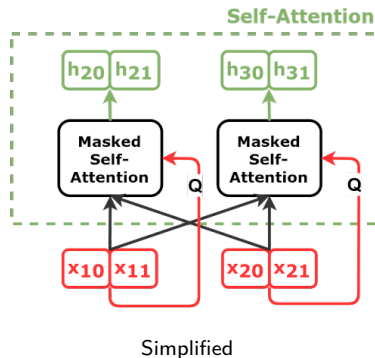
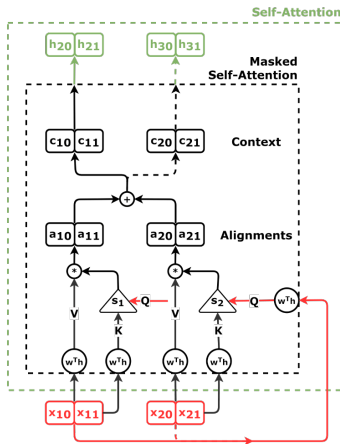


Cross-Attention

The attention mechanism that we have seen before self-attention, where the the decoder provided a “query” (Q) which served not only as input but also got concatenated to the resulting context vector, is called **cross-attention**.

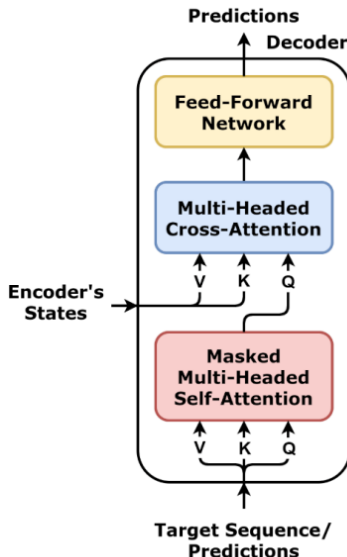
Encoder with self- and cross-attention

Decoder with Self-Attention

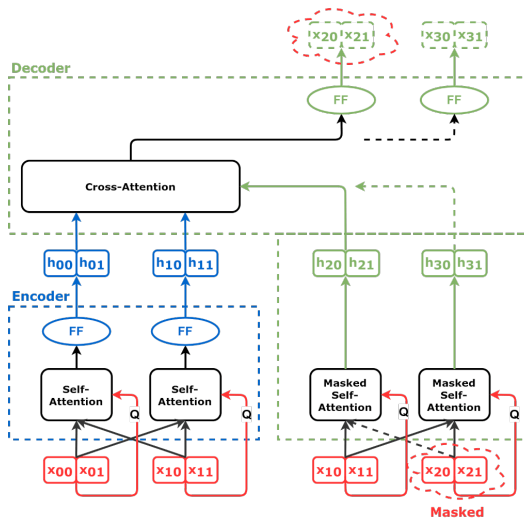


Decoder with Self-Attention Schematic Diagram

- There is one small difference in the self-attention architecture between encoder and decoder: the feed-forward network sits atop the cross-attention mechanism (not depicted in the figure above) instead of the self-attention mechanism.
- The feed-forward network also maps the decoder's output from the dimensionality of the model (d_{model}) back to the number of features, thus yielding predictions.



Decoder with Self Attention in the Encoder-Decoder Framework



How Decoder with Self-Attention Works

- **First input** is (x_{10}, x_{11}) , the last known element of the source sequence, as usual.
- **Source mask** is the same mask used to ignore padded data points in the encoder.

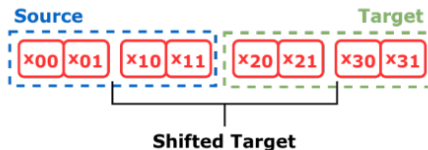
What we also need are:

- Subsequent Inputs and Teacher Forcing
- Target Mask

Subsequent Inputs and Teacher Forcing

The shifted target sequence

The shifted target sequence is defined as a sequence that includes the last known element of the source sequence and all elements in the target sequence **but the last one**.



The **shifted target sequence** was already used when we discussed *teacher forcing*.

- In teacher forcing, at every step (after the first one), it randomly chose as the input to the subsequent step either an actual element from that sequence or a prediction.
- It worked very well with recurrent layers that were sequential in nature.



Problem with Attention Scores in Decoder

We are using the whole *shifted target sequence* at once as the “query” argument of the decoder.

$$\alpha_{21}, \alpha_{22} = \text{softmax}\left(\frac{Q_1 \cdot K_1}{\sqrt{2}}, \frac{Q_1 \cdot K_2}{\sqrt{2}}\right)$$

The problem is that it is using a “key” (K_2) and a “value” (V_2) that are transformations of the data point it is trying to predict. We should not allow the model to **cheat** by peeking into the future.

$$\text{context vector}_2 = \alpha_{21} V_1 + \alpha_{22} V_2$$

$$\alpha_{31}, \alpha_{32} = \text{softmax}\left(\frac{Q_2 \cdot K_1}{\sqrt{2}}, \frac{Q_2 \cdot K_2}{\sqrt{2}}\right)$$

$$\text{context vector}_3 = \alpha_{31} V_1 + \alpha_{32} V_2$$



Decoder Attention Scores and Target Mask

- For the decoder, the shape of the alphas attribute is given by $(N, L_{target}, L_{target})$ since it is looking at itself.
- Any alphas above the diagonal are, literally, cheating codes.

target	source	
	x_1	x_2
h_2	α_{21}	α_{22}
h_3	α_{31}	α_{32}

We need to force the self-attention mechanism to ignore them.

Target Mask

The purpose of the target mask is to zero attention scores for “future” data points.

target	source	
	x_1	x_2
h_2	α_{21}	0
h_3	α_{31}	α_{32}



Subsequent Mask in Code

```

1 def subsequent_mask(size):
2     attn_shape = (1, size, size)
3     # torch.triu returns the upper triangular part of a matrix
4     subsequent_mask = (1 - torch.triu(torch.ones(attn_shape),
5                                         diagonal=1)).bool()
6     return subsequent_mask

```

code/subsequent_mask.py

```

tensor([[[ True, False],
         [ True,  True]]])

```

- We must use this mask while querying the decoder to prevent it from cheating.
- You can choose to use an additional mask to “hide” more data from the decoder if you wish, but the subsequent mask is a strong requirement of the self-attention decoder.



Decoder with Self-Attention in Code I

There is one main difference (in the code) between the encoder and the decoder:

The decoder includes a cross-attention mechanism

```

1  class DecoderSelfAttn(nn.Module):
2
3
4      def __init__(self, n_heads, d_model, ff_units,
5                      n_features=None):
6          super().__init__()
7          self.n_heads = n_heads
8          self.d_model = d_model
9          self.ff_units = ff_units
10         self.n_features = d_model
11         if n_features is None else n_features
12         self.self_attn_heads = MultiHeadAttention(
13                                     n_heads, d_model,
14                                     input_dim=self.n_features)
15         self.cross_attn_heads = MultiHeadAttention(
16                                     n_heads, d_model)

```

Decoder with Self-Attention in Code II

```

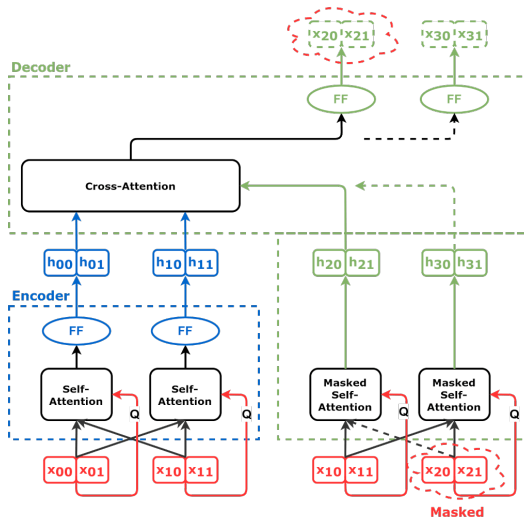
17         self.ffn = nn.Sequential(
18             nn.Linear(d_model, ff_units),
19             nn.ReLU(),
20             nn.Linear(ff_units, self.n_features),
21         )
22
23     def init_keys(self, states):
24         self.cross_attn_heads.init_keys(states)
25
26     def forward(self, query, source_mask=None, target_mask=None)
27     :
28         self.self_attn_heads.init_keys(query)
29         att1 = self.self_attn_heads(query, target_mask)
30         att2 = self.cross_attn_heads(att1, source_mask)
31         out = self.ffn(att2)
32         return out

```

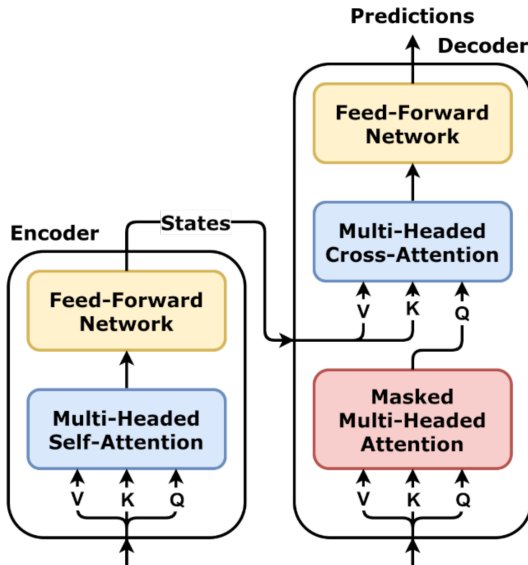
code/decoder_self_attn.py



Encoder Decoder with Self Attention



Encoder Decoder Self-Attention Schematic Diagram



Encoder Decoder Self-Attention Model in Code I

```
1 class EncoderDecoderSelfAttn(nn.Module):
2     def __init__(self, encoder, decoder, input_len, target_len):
3         super().__init__()
4         self.encoder = encoder
5         self.decoder = decoder
6         self.input_len = input_len
7         self.target_len = target_len
8         self.trg_masks = self.subsequent_mask(self.target_len)
9
10    @staticmethod
11    def subsequent_mask(size):
12        attn_shape = (1, size, size)
13        subsequent_mask = (1-torch.triu(torch.ones(attn_shape),
14                                                diagonal=1))
15        return subsequent_mask
```

Encoder Decoder Self-Attention Model in Code II

```
22
23     def encode(self, source_seq, source_mask):
24         # Encodes the source sequence and uses the result
25         # to initialize the decoder
26         encoder_states = self.encoder(source_seq, source_mask)
27         self.decoder.init_keys(encoder_states)
28
29     def decode(self, shifted_target_seq, source_mask=None,
30                target_mask=None):
31         # Decodes/generates a sequence using the shifted
32         # target sequence (masked) - used in TRAIN mode
33         outputs = self.decoder(shifted_target_seq,
34                                source_mask=source_mask,
35                                target_mask=target_mask)
36
37         return outputs
38
39
40
41
42
```



Encoder Decoder Self-Attention Model in Code III

```
43 def predict(self, source_seq, source_mask):
44     # Decodes/generates a sequence using one input
45     # at a time - used in EVAL mode
46     inputs = source_seq[:, -1:]
47     for i in range(self.target_len):
48         out = self.decode(inputs, source_mask, self.
49         trg_masks[:, :i+1, :i+1])
50         out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
51         inputs = out.detach()
52         outputs = inputs[:, 1:, :]
53     return outputs
54
55 def forward(self, X, source_mask=None):
56     # Sends the mask to the same device as the inputs
57     self.trg_masks = self.trg_masks.type_as(X).bool()
58     # Slices the input to get source sequence
59     source_seq = X[:, :self.input_len, :]
60     # Encodes source sequence AND initializes decoder
61     self.encode(source_seq, source_mask)
62     if self.training:
63         # Slices the input to get the shifted target seq
```



Encoder Decoder Self-Attention Model in Code IV

```
63         shifted_target_seq = X[:, self.input_len-1:-1, :]  
64         # Decodes using the mask to prevent cheating  
65         outputs = self.decode(shifted_target_seq,  
66                               source_mask, self.trg_masks)  
67     else:  
68         # Decodes using its own predictions  
69         outputs = self.predict(source_seq, source_mask)  
70  
71     return outputs
```

code/EncoderDecoderSelfAttn.py



Code Explained

- **encode**: takes the source sequence and mask and encodes it into a sequence of states that is immediately used to initialize the “keys” (and “values”) in the decoder
- **decode**: takes the shifted target sequence and both source and target masks to generate a target sequence - it is used for training only
- **predict**: takes the source sequence, the source mask, and uses a subset of the target mask to actually predict an unknown target sequence - it is used for evaluation/prediction only
- **forward**: it splits the input into the source and shifted target sequences (if available), encodes the source sequence, and calls either decode or predict according to the model's mode (train or eval).



Take Aways

- Understand the attention mechanism
- Understand how context vectors are calculated
- Cross-Attention vs. Self-Attention

Transformers

CITS4012 Natural Language Processing

A/Prof. Wei Liu

`wei.liu@uwa.edu.au`

Computer Science and Software Engineering
The University of Western Australia

April 16, 2025



THE UNIVERSITY OF
**WESTERN
AUSTRALIA**

What we are going to cover today

1 Positional Encoding

- Self-Attention: Sequence No More
- How to obtain unique meaningful Positional Encoding?
- Positional Encoding in Code

2 Transform and Roll Out

- Narrow Attention
- Stacking Encoders and Decoders
- Layer Normalization
- Transformers

3 PyTorch's Transformer Class

[see Deep Learning with Pytorch Step by Step, Chapter 9 & 10]



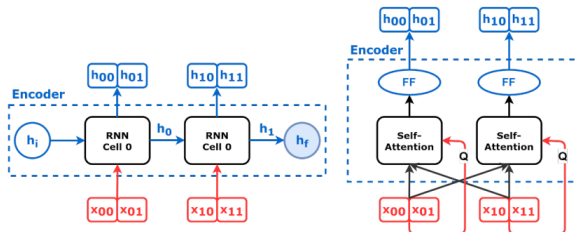
Positional Encoding



RNN vs. Self-Attention

The recurrent neural network ensured that the output at each step is fed to the next, compare it to the encoder using self-attention, however:

- every step is independent of the others.
- if flip the order of the data points in the sequence, the “hidden states” are produced in a different order, but the same values.



Self-Attention is highly parallelizable, but **both a blessing and a curse**

- on the one hand, it makes computation very efficient;
- on the other hand, it is throwing away valuable information.

Positional Encoding

We need to find a way to inform the model about the position of every data point such that it knows the order of the data points.

We need to generate a **unique value** for each position in the input.

Why not just use **index of the position**?

The indices of the positions are unique but not bounded, thus not suitable for neural networks.



Options and Problems

- Problem: not unique anymore

No one said we must use only one vector!

How about we build three vectors instead, using three hypothetical sequence lengths (four, five, and seven)

Position	0	1	2	3	4	5	6	7
(Pos mod 4)/4	0.00	0.25	0.50	0.75	0.00	0.25	0.50	0.75
(Pos mod 5)/5	0.00	0.20	0.40	0.60	0.80	0.00	0.20	0.40
(Pos mod 7)/7	0.00	0.14	0.29	0.43	0.57	0.71	0.86	0.00

- Good news:** The positional encoding above is unique up to the 140th position and we can easily extend that by adding more vectors.
- Problem:** The distance between positions three and two (given by the norm of the difference vector) is not equal to the distance between positions three and four.

3	2	Diff
0.75	0.50	0.25
0.60	0.40	0.20
0.43	0.29	0.14

$$\text{Distance} = \|\text{Diff}\| = 0.35$$

4	3	Diff
0.00	0.75	-0.75
0.80	0.60	0.20
0.57	0.43	0.14

$$\text{Distance} = \|\text{Diff}\| = 0.79$$

A Smoother Cycle is Needed

This inconsistency arises from the fact that our encoding is resetting to zero every time the modulus kicks in.

- The distance between positions three and four got much larger because, at position four, the first vector goes back to zero.

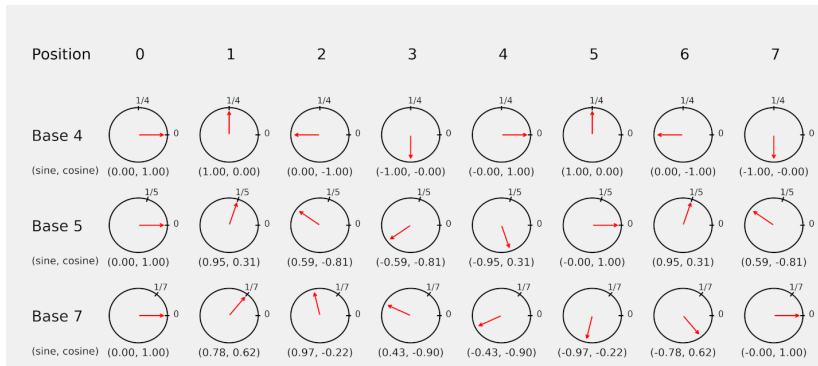
We need some other function that has a smoother cycle.

Let's take our encodings and multiply them by 360° to turn each of our encoding into a degree that moves along a circle.

Position	0	1	2	3	4	5	6	7
Base 4	0	90	180	270	0	90	180	270
Base 5	0	72	144	216	288	0	72	144
Base 7	0	51	103	154	206	257	309	0



Moving around a circle



The circles above show the sine and cosine values corresponding to the coordinates of the tip of each red arrow (assuming a circle with a radius of one).

The Encoding Vectors are Sine/Cosine Values

The sine and cosine values, that is, the coordinates of the red arrow, are the actual positional encoding of a given position

Position	0	1	2	3	4	5	6	7
sine (base 4)	0.00	1.00	0.00	-1.00	0.00	1.00	0.00	-1.00
cosine (base 4)	1.00	0.00	-1.00	0.00	1.00	0.00	-1.00	0.00
sine (base 5)	0.00	0.95	0.59	-0.59	-0.95	0.00	0.95	0.59
cosine (base 5)	1.00	0.31	-0.81	-0.81	0.31	1.00	0.31	-0.81
sine (base 7)	0.00	0.78	0.97	0.43	-0.43	-0.97	-0.78	0.00
cosine (base 7)	1.00	0.62	-0.22	-0.90	-0.90	-0.22	0.62	1.00

Figure: Each column vector is an encoding for that position

Distance between Position Encodings

Figure 1 illustrates two examples of distance calculation between two sets of data. Each example shows a subtraction of a reference vector from a target vector to produce a difference vector, which is then used to calculate a distance.

Example 1 (Left):

3	2	Diff
-1.00	0.00	-1.00
0.00	-1.00	1.00
-0.59	0.59	-1.18
-0.81	-0.81	0.00
0.43	0.97	-0.54
-0.90	-0.22	-0.68

Distance = ||Diff|| = 2.03

Example 2 (Right):

4	3	Diff
0.00	-1.00	1.00
1.00	0.00	1.00
-0.95	-0.59	-0.36
0.31	-0.81	1.12
-0.43	0.43	-0.87
-0.90	-0.90	0.00

Distance = ||Diff|| = 2.03

The encoded distance between any two positions T steps apart is constant now

- 1 Choose three bases (e.g. 4,5,7) then take the modulus;
- 2 Normalise the remainders by the bases;
- 3 Multiple by 360° ;
- 4 Apply the sine and cosine to each of the three vectors (in degrees $^\circ$), to get six vectors.

How to choose the “bases” for the encoding?

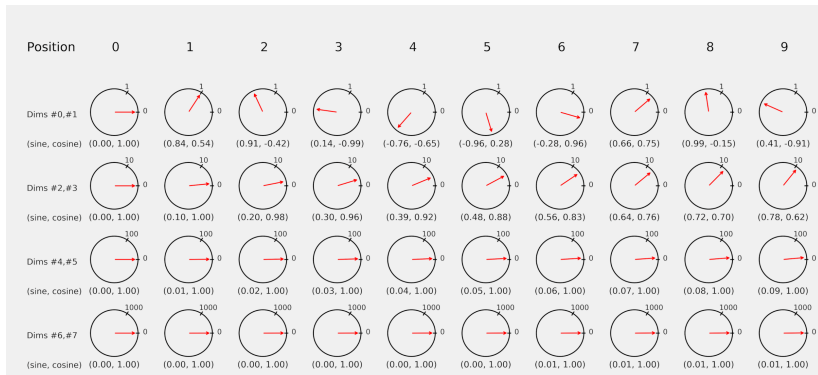
- 1 For the first vector, we can simply move along the circle as many radians as the index of the position (one radian is approximately 57.3°).
- 2 Then, for each new vector we add to the encoding, we move along the circle with exponentially slower angular speeds. For example, in the second vector, we would move only one-tenth of a radian (approximately 5.73°) for each new position.
- 3 In the third vector we would move only one-hundredth of a radian, and so on and so forth.

In practice, we choose the number of dimensions first, and then compute the corresponding speeds.

- For example, for encoding with eight dimensions, we have four angular speeds.



Angular Speed - Exponentially Decreasing



PositionalEncoding in Code

```

1 class PositionalEncoding(nn.Module):
2     def __init__(self, max_len, d_model):
3         super().__init__()
4         self.d_model = d_model
5         pe = torch.zeros(max_len, d_model)
6         position = torch.arange(0, max_len).float()
7                                     .unsqueeze(1)
8         angular_speed = torch.exp(
9             torch.arange(0, d_model, 2).float() *
10             (-np.log(10000.0) / d_model))
11         # even dimensions
12         pe[:, 0::2] = torch.sin(position * angular_speed)
13         # odd dimensions
14         pe[:, 1::2] = torch.cos(position * angular_speed)
15         self.register_buffer('pe', pe.unsqueeze(0))
16
17     def forward(self, x):
18         # x is N, L, D
19         # pe is 1, maxlen, D
20         scaled_x = x * np.sqrt(self.d_model)
21         encoded = scaled_x + self.pe[:, :x.size(1), :]
22         return encoded

```

Register a no gradient attribute

- In the constructor, it uses `register_buffer` to define an attribute of the module
- In the forward method, it is scaling the input before adding the positional encoding

The `register_buffer` is used to define an attribute that is part of the module's state, yet not a parameter.

Buffers

If you have parameters in your model, which should be saved and restored in the `state_dict`, but not trained by the optimizer, you should register them as buffers.

The positional encoding is a good example:

- Its values are computed according to the dimension and length used by the model; and
- Even though these values are going to be used during training, they should not be updated by gradient descent.



Encoder with Positional Encoding

```

1  class EncoderPe(nn.Module):
2      def __init__(self, n_heads, d_model, ff_units,
3                  n_features=None, max_len=100):
4          super().__init__()
5          pe_dim = d_model if n_features is None else n_features
6          self.pe = PositionalEncoding(max_len, pe_dim)
7          self.layer = EncoderSelfAttn(n_heads, d_model, ff_units,
8                                      n_features)
9
10     def forward(self, query, mask=None):
11         query_pe = self.pe(query)
12         out = self.layer(query_pe, mask)
13         return out

```

code/EncoderPe.py



Decoder with Positional Encoding

```

1  class DecoderPe(nn.Module):
2      def __init__(self, n_heads, d_model, ff_units,
3                  n_features=None, max_len=100):
4          super().__init__()
5          pe_dim = d_model if n_features is None else n_features
6          self.pe = PositionalEncoding(max_len, pe_dim)
7          self.layer = DecoderSelfAttn(n_heads, d_model, ff_units,
8                                      n_features)
9
10     def init_keys(self, states):
11         self.layer.init_keys(states)
12
13     def forward(self, query, source_mask=None,
14                target_mask=None):
15         query_pe = self.pe(query)
16         out = self.layer(query_pe, source_mask, target_mask)
17         return out

```

code/DecoderPe.py



Transform and Roll Out



Getting Closer to Transformers!

We are fairly close to developing our own version of the famous Transformer model! We already have:

- Self-attention
- Multi-headed attention
- Positional Encoding

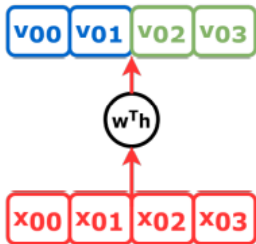
Before we can effectively “transform and roll out”, a few more details are needed.

- We need to revisit the **multi-headed attention** mechanism to make it less computationally expensive by using **narrow attention**.
- To incorporate positional encoding, we have reverse the standardization of input, this “bad” effect can be compensated by **layer normalisation**.
- We need to add **dropout**, **residual connections**, and more “**layers**” to get to the deep Transformer model.

What's Narrow Attention?

Narrow Attention

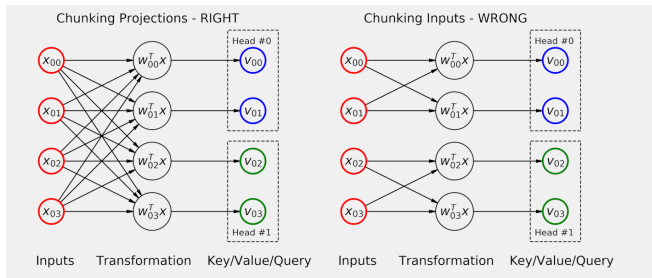
Compare to *full* attention head that does wide attention, in narrow attention each attention head will get a chunk of the transformed data points (projections) to work with.



The attention heads do not use chunks of the original data points, but of their projections.

It may look like we may get the same result, either split the inputs into chunks or the projections into chunks. But that is definitely **not** the case.

Chunking: the wrong and the right way



Chunking input (the wrong approach) is a simpler model but with bad consequences:

- it has only eight weights while the correct one has sixteen, so its learning capacity is limited.
- since each head can only look at a subset of the features, they simply cannot learn about long-range dependencies in the inputs.



What does Multi-headed Attention do in NLP

Since each head is working with a subset of the projected dimensions, these projected dimensions may end up representing a different aspect of the underlying data.

Stanford Chris Manning's group has a paper on WHAT DOES BERT LOOK AT? AN ANALYSIS OF BERT'S ATTENTION [2] found that BERT's attention heads exhibit patterns such as:

- attending to delimiter tokens, specific positional offsets, or broadly attending over the whole sentence;
- heads in the same layer often exhibiting similar behaviors;
- certain attention heads correspond well to linguistic notions of syntax and coreference. For example, they find heads that attend to
 - the direct objects of verbs,
 - determiners of nouns,
 - objects of prepositions,
 - coreferent mentions

The diagram illustrates the Multi-Head Attention mechanism. It shows the flow of information from input tokens through various layers to produce output tokens.

- Input Tokens:** The input consists of two sets of tokens: $x_{00} x_{01} x_{02} x_{03}$ (red) and $x_{10} x_{11} x_{12} x_{13}$ (red).
- Projections:** Each input token is projected using a weight matrix $w^T h$ to produce value (V), key (K), and alignment (s) vectors.
 - For the first set: $V_{0l}, V_{0r}, K_{0l}, K_{0r}, s_0, s_1$.
 - For the second set: $V_{1l}, V_{1r}, K_{1l}, K_{1r}, s_0, s_1$.
- Alignments:** The alignment vectors s_0 and s_1 are used to weight the value vectors V_{0l}, V_{0r} and V_{1l}, V_{1r} respectively, resulting in aligned value vectors a_{00}, a_{01} and a_{10}, a_{11} (blue) for the first set, and a_{02}, a_{03} and a_{12}, a_{13} (green) for the second set.
- Attention Heads:** The aligned value vectors are combined in two attention heads (0 and 1) to produce context vectors $c_{00}, c_{01}, c_{02}, c_{03}$ (blue) and $c_{10}, c_{11}, c_{12}, c_{13}$ (green).
- Feed-Forward (FF):** The context vectors are processed by a Feed-Forward network to produce the final output tokens $h_{00}, h_{01}, h_{02}, h_{03}$ (black) and $h_{10}, h_{11}, h_{12}, h_{13}$ (black).



MultiHeaded Narrow Attention in Code I

As a combination of both Attention and MultiHeadAttention classes from before, the MultiHeadedAttention (narrow) implements

- the chunking of the projections, and
- introduces dropout for attention scores

```

1 class MultiHeadedAttention(nn.Module):
2     def __init__(self, n_heads, d_model, dropout=0.1):
3         super(MultiHeadedAttention, self).__init__()
4         self.n_heads = n_heads
5         self.d_model = d_model
6         self.d_k = int(d_model / n_heads)
7         self.linear_query = nn.Linear(d_model, d_model)
8         self.linear_key = nn.Linear(d_model, d_model)
9         self.linear_value = nn.Linear(d_model, d_model)
10        self.linear_out = nn.Linear(d_model, d_model)
11        self.dropout = nn.Dropout(p=dropout)
12        self.alphas = None
13
14        def make_chunks(self, x):
15            batch_size, seq_len = x.size(0), x.size(1)

```



oooooooooooooooooooo

oooooooo●●●●oooooooooooooooooooooooooooo

oooooooo

MultiHeaded Narrow Attention in Code II

```

16     # N, L, D -> N, L, n_heads * d_k
17     x = x.view(batch_size, seq_len, self.n_heads, self.d_k)
18     # N, n_heads, L, d_k
19     x = x.transpose(1, 2)
20     return x
21
22     def init_keys(self, key):
23         # N, n_heads, L, d_k
24         self.proj_key = self.make_chunks(self.linear_key(key))
25         self.proj_value = self.make_chunks(
26             self.linear_value(key))
27
28     def score_function(self, query):
29         # scaled dot product
30         # N, n_heads, L, d_k x # N, n_heads, d_k,
31         # -> N, n_heads, L, L
32         proj_query = self.make_chunks(self.linear_query(query))
33         dot_products = torch.matmul(proj_query,
34                                     self.proj_key.transpose(-2, -1))
35         scores = dot_products / np.sqrt(self.d_k)
36         return scores

```



○○○○○○○○○○○○○○○○○○○○

○○○○○○○○●●●●○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

○○○○○○○○

MultiHeaded Narrow Attention in Code III

```

37
38 def attn(self, query, mask=None):
39     # Query is batch-first: N, L, D
40     # Score function will generate scores for each head
41     scores = self.score_function(query) # N, n_heads, L, L
42     if mask is not None:
43         scores = scores.masked_fill(mask == 0, -1e9)
44     alphas = F.softmax(scores, dim=-1) # N, n_heads, L, L
45     alphas = self.dropout(alphas)
46     self.alphas = alphas.detach()
47
48     # N, n_heads, L, L x N, n_heads, L, d_k
49     #     -> N, n_heads, L, d_k
50     context = torch.matmul(alphas, self.proj_value)
51     return context
52
53 def output_function(self, contexts):
54     # N, L, D
55     out = self.linear_out(contexts) # N, L, D
56     return out
57

```



MultiHeaded Narrow Attention in Code IV

```

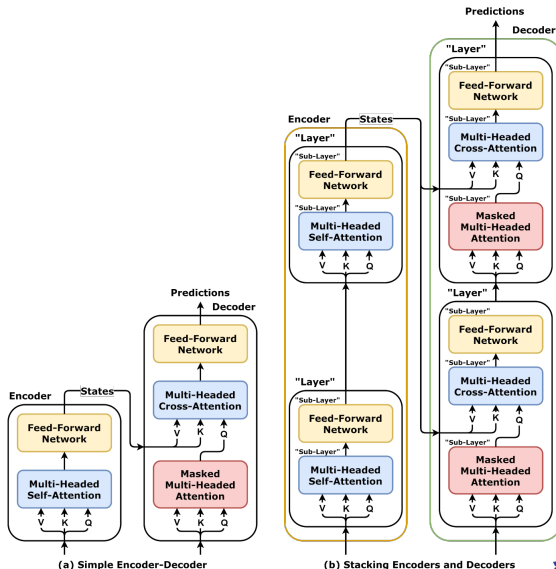
58     def forward(self, query, mask=None):
59         if mask is not None:
60             # N, 1, L, L - every head uses the same mask
61             mask = mask.unsqueeze(1)
62
63         # N, n_heads, L, d_k
64         context = self.attn(query, mask=mask)
65         # N, L, n_heads, d_k
66         context = context.transpose(1, 2).contiguous()
67         # N, L, n_heads * d_k = N, L, d_model
68         context = context.view(query.size(0), -1, self.d_model)
69         # N, L, d_model
70         out = self.output_function(context)
71         return out

```

code/MultiHeadedAttention.py



Stacking two Encoders and two Decoders

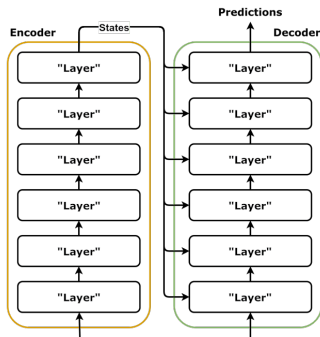


How Layers are Stacked?

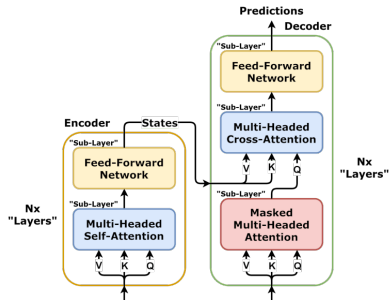
- The output of one encoder feeds the next, and the last encoder outputs states as usual.
- These states will feed the cross-attention mechanism of all stacked decoders.
- The output of one decoder feeds the next, and the last decoder outputs predictions as usual.
- The former encoder/decoder is now a so-called “layer”, and
- a stack of “layers” compose the new, deeper, encoder/decoder.
- Each operation (multi-headed self- and cross-attention mechanisms and feed-forward networks) inside a “layer” is now a “sub-layer”.



Stacking Layers for Deep Learning



(a) Stacked "Layers"



(b) Stacked "Layers" in detail

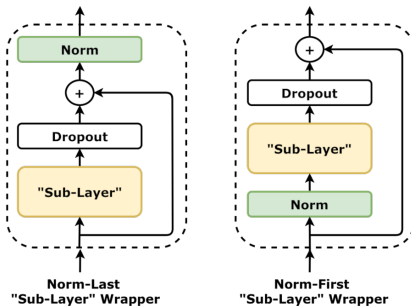
Stacking six "layers"



Wrapping Sub-Layers

Going Deeper → Vanishing Gradients + Overfitting

We can address these by *batch normalization*, *residual connections*, and *dropouts*.



Norm-Last vs. Norm-First

The norm-last wrapper follows the ATTENTION IS ALL YOU NEED paper [4].

$$outputs_{norm-last} = \text{norm}(inputs + \text{dropout}(\text{sublayer}(inputs)))$$

$$outputs_{norm-first} = inputs + \text{dropout}(\text{sublayer}(\text{norm}(inputs)))$$

We will normalize the final outputs, that is, the output of the last “layer” (which is the output of its last, not normalized, “sub-layer”).

- Any intermediate output is simply the input of the subsequent “sub-layer”, and
- Each “sub-layer” normalizes its own inputs

When using positional encoding, we want to normalize inputs, so norm-first is more convenient.



Layer Normalization

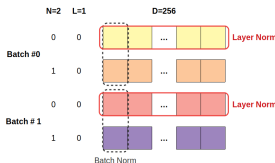
Layer normalization was first introduced by Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton in their 2016 paper LAYER NORMALIZATION [1].

In a tabular dataset, Batch Normalization were standardizing the columns.

Before, each feature, either in the whole training set (using **Scikit-learn's** StandardScaler) or in a mini-batch (using batch norm), was standardized to have zero mean and unit standard deviation.

Layer normalization, in a tabular dataset, standardizes the rows.

Each data point will have the average of its features equals zero, and the standard deviation of its features equals one.



Layer Normalization - Math Notation

Layer normalization standardizes individual data points, the rows in the tensor above, so we need to compute statistics over the corresponding dimension (D).

$$\bar{X}_{n,l} = \frac{1}{D} \sum_{d=1}^D x_{n,l,d}$$

$$\sigma_{n,l}(X) = \sqrt{\frac{1}{D} \sum_{d=1}^D (x_{n,l,d} - \bar{X}_{n,l})^2}$$

$$\text{standardized } x_{n,l,d} = \frac{x_{n,l,d} - \bar{X}_{n,l}}{\sigma_{n,l}(X) + \epsilon}$$



LayerNorm in Code

```

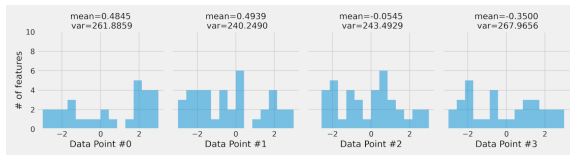
1 d_model = 4
2 seq_len = 2
3 n_points = 3
4
5 torch.manual_seed(34)
6 data = torch.randn(n_points, seq_len, d_model)
7 pe = PositionalEncoding(seq_len, d_model)
8 inputs = pe(data)
9
10 inputs_mean = inputs.mean(axis=2).unsqueeze(2)
11 inputs_var = inputs.var(axis=2, unbiased=False).unsqueeze(2)
12 (inputs - inputs_mean)/torch.sqrt(inputs_var+1e-5)
13
14 # Or simply using PyTorch's nn.LayerNorm
15 layer_norm = nn.LayerNorm(d_model)
16 normalized = layer_norm(inputs)
17 normalized[0][0].mean(), normalized[0][0].std(unbiased=False)

```

code/LayerNorm.py



Data Points Before and After Layer Normalization



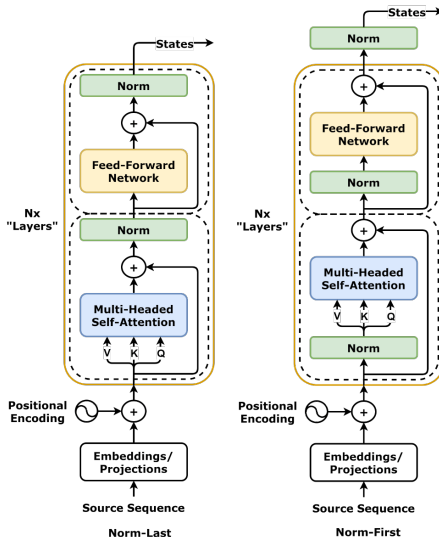
Before Layer Normalisation



After Layer Normalisation



Transformer Encoder with Layer Norm



Transformer Encoder - norm-last vs norm-first



Transformer Encoder with Layer Norm in Math Notations

$$\begin{aligned}
 outputs_{norm-last} = & \underbrace{norm(inputs + att(inputs))}_{\text{Output of SubLayer}_0} + \\
 & \underbrace{ffn(norm(inputs + att(inputs)))}_{\text{Output of SubLayer}_0}
 \end{aligned}$$

$$\begin{aligned}
 outputs_{norm-first} = & \underbrace{inputs + att(norm(inputs))}_{\text{Output of SubLayer}_0} + \\
 & \underbrace{ffn(norm(inputs + att(norm(inputs))))}_{\text{Output of SubLayer}_0}
 \end{aligned}$$

- The norm-first wrapper allows the inputs to flow unimpeded (the inputs are not normalized) all the way to the top, while adding the results of each “sub-layer” along the way (the last normalization of norm-first happens outside of the “sub-layers” so it is not included in the equation).
- There is no straight answer to which one is best.



EncoderLayer in Code I

```
1 class EncoderLayer(nn.Module):
2     def __init__(self, n_heads, d_model,
3                   ff_units, dropout=0.1):
4         super().__init__()
5         self.n_heads = n_heads
6         self.d_model = d_model
7         self.ff_units = ff_units
8         self.self_attn_heads = MultiHeadedAttention(n_heads,
9                                                       d_model, dropout=dropout)
10        self.ffn = nn.Sequential(
11            nn.Linear(d_model, ff_units),
12            nn.ReLU(),
13            nn.Dropout(dropout),
14            nn.Linear(ff_units, d_model),
15        )
16
17        self.norm1 = nn.LayerNorm(d_model)
18        self.norm2 = nn.LayerNorm(d_model)
19        self.drop1 = nn.Dropout(dropout)
20        self.drop2 = nn.Dropout(dropout)
21
```



EncoderLayer in Code II

```

22     def forward(self, query, mask=None):
23         # Sublayer #0
24         # Norm
25         norm_query = self.norm1(query)
26         # Multi-headed Attention
27         self.self_attn_heads.init_keys(norm_query)
28         states = self.self_attn_heads(
29             norm_query, mask)
30         # Add
31         att = query + self.drop1(states)
32
33         # Sublayer #1
34         # Norm
35         norm_att = self.norm2(att)
36         # Feed Forward
37         out = self.ffn(norm_att)
38         # Add
39         out = att + self.drop2(out)
40         return out

```

code/EncoderLayer.py



Stack Layers into an Transformer Encoder

```

1  class EncoderTransf(nn.Module):
2      def __init__(self, encoder_layer, n_layers=1,
3                  max_len=100):
4          super().__init__()
5          self.d_model = encoder_layer.d_model
6          self.pe = PositionalEncoding(max_len,
7                                      self.d_model)
8          self.norm = nn.LayerNorm(self.d_model)
9          self.layers = nn.ModuleList(
10             [copy.deepcopy(encoder_layer)
11              for _ in range(n_layers)])
12
13     def forward(self, query, mask=None):
14         # Positional Encoding
15         x = self.pe(query)
16         for layer in self.layers:
17             x = layer(x, mask)
18         # Norm
19         return self.norm(x)

```

code/EncoderTransf.py



Transformer Decoder Code Explained

The `EncoderTransf` constructor takes

- an instance of an `EncoderLayer`,
- the number of “layers” to stack on top of one another, and
- a max length of the source sequence to be used for the positional encoding.
- `deepcopy` is used to make sure we create real copies of the encoder layer, and
- `nn.ModuleList` to make sure PyTorch can find the “layers” inside the list.
- The default number of “layers” is only one, but the original Transformer uses six “layers”.



Decoder Layer in Code I

```

1 class DecoderLayer(nn.Module):
2     def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
3         super().__init__()
4         self.n_heads = n_heads
5         self.d_model = d_model
6         self.ff_units = ff_units
7         self.self_attn_heads = MultiHeadedAttention(
8             n_heads, d_model, dropout=dropout)
9         self.cross_attn_heads = MultiHeadedAttention(
10            n_heads, d_model, dropout=dropout)
11        self.ffn = nn.Sequential(
12            nn.Linear(d_model, ff_units),
13            nn.ReLU(),
14            nn.Dropout(dropout),
15            nn.Linear(ff_units, d_model),
16        )
17
18        self.norm1 = nn.LayerNorm(d_model)
19        self.norm2 = nn.LayerNorm(d_model)
20        self.norm3 = nn.LayerNorm(d_model)
21        self.drop1 = nn.Dropout(dropout)

```

Decoder Layer in Code II

```

22         self.drop2 = nn.Dropout(dropout)
23         self.drop3 = nn.Dropout(dropout)
24
25     def init_keys(self, states):
26         self.cross_attn_heads.init_keys(states)
27
28     def forward(self, query, source_mask=None, target_mask=None):
29         # Sublayer #0
30         # Norm
31         norm_query = self.norm1(query)
32         # Masked Multi-head Attention
33         self.self_attn_heads.init_keys(norm_query)
34         states = self.self_attn_heads(norm_query, target_mask)
35         # Add
36         att1 = query + self.drop1(states)
37
38         # Sublayer #1
39         # Norm
40         norm_att1 = self.norm2(att1)
41         # Multi-head Attention
42         encoder_states = self.cross_attn_heads(norm_att1,

```



Decoder Layer in Code III

```

43                                     source_mask)
44     # Add
45     att2 = att1 + self.drop2(encoder_states)
46
47     # Sublayer #2
48     # Norm
49     norm_att2 = self.norm3(att2)
50     # Feed Forward
51     out = self.ffn(norm_att2)
52     # Add
53     out = att2 + self.drop3(out)
54     return out

```

code/DecoderLayer.py



Stack Layers into an Transformer Decoder I

```

1 class DecoderTransf(nn.Module):
2     def __init__(self, decoder_layer, n_layers=1, max_len=100):
3         super(DecoderTransf, self).__init__()
4         self.d_model = decoder_layer.d_model
5         self.pe = PositionalEncoding(max_len, self.d_model)
6         self.norm = nn.LayerNorm(self.d_model)
7         self.layers = nn.ModuleList([
8             copy.deepcopy(decoder_layer)
9             for _ in range(n_layers)])
10
11     def init_keys(self, states):
12         for layer in self.layers:
13             layer.init_keys(states)
14
15
16
17
18
19
20
21

```

Stack Layers into an Transformer Decoder II

```

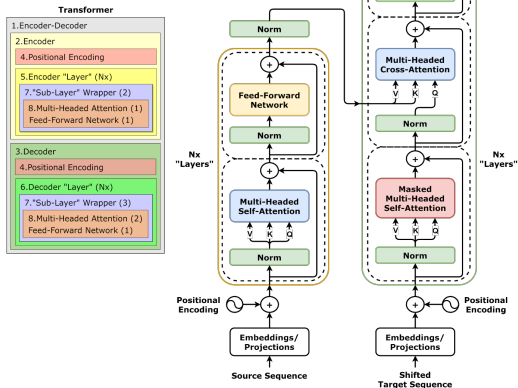
22     def forward(self, query, source_mask=None, target_mask=None):
23         # Positional Encoding
24         x = self.pe(query)
25         for layer in self.layers:
26             x = layer(x, source_mask, target_mask)
27         # Norm
28         return self.norm(x)

```

code/DecoderTransf.py



Full Transformer



Manual Transformer in Code I

```
1 class EncoderDecoderTransf(EncoderDecoderSelfAttn):
2     def __init__(self, encoder, decoder, input_len, target_len,
3                 n_features):
4         super(EncoderDecoderTransf, self).__init__(encoder,
5             decoder, input_len, target_len)
6         self.n_features = n_features
7         self.proj = nn.Linear(n_features, encoder.d_model)
8         self.linear = nn.Linear(encoder.d_model, n_features)
9
10    def encode(self, source_seq, source_mask=None):
11        # Projection
12        source_proj = self.proj(source_seq)
13        encoder_states = self.encoder(source_proj, source_mask)
14        self.decoder.init_keys(encoder_states)
15
16
17
18
19
20
21    def decode(self, shifted_target_seq, source_mask=None,
```

Manual Transformer in Code II

```
22         target_mask=None):
23     # Projection
24     target_proj = self.proj(shifted_target_seq)
25     outputs = self.decoder(target_proj,
26                             source_mask=source_mask,
27                             target_mask=target_mask)
28     # Linear
29     outputs = self.linear(outputs)
30     return outputs
```

code/EncoderDecoderTransf.py



Transformer - An Extended Encoder Decoder Model with Self-Attention

The Transformer still is an encoder-decoder architecture like the one we developed before, so it should be no surprise that we can actually use our former `EncoderDecoderSelfAttn` class as a parent class and add two extra components to it:

- a projection layer to map our original features (`n_features`) to the dimensionality of both encoder and decoder (`d_model`)
- a final linear layer to map the decoder's outputs back to the original feature space (e.g. the coordinates we try to predict).

PyTorch's Transformer Class



PyTorch's Transformer Class

PyTorch implements a full-fledged Transformer class of its own: `nn.Transformer`. However, there are some differences:

- First, and most importantly, PyTorch implements norm-last “sub-layer” wrappers, normalizing the output of each “sub-layer”.
- It does not implement positional encoding, the final linear layer, and the projection layer, so we have to handle those ourselves.
- PyTorch's Transformer uses sequence-first shapes for its inputs (L, N, F), and there is no batch-first option.

The constructor expects many arguments because PyTorch's Transformer actually builds both encoder and decoder by itself.



Transformer Model using nn.Transformer I

```

1 torch.manual_seed(42)
2 transformer = nn.Transformer(d_model=6, nhead=3,
3                               num_encoder_layers=1,
4                               num_decoder_layers=1,
5                               dim_feedforward=20, dropout=0.1)
6 model_transformer = TransformerModel(transformer,
7                                       input_len=2, target_len=2, n_features=2)
8 loss = nn.MSELoss()
9 optimizer = torch.optim.Adam(model_transformer.parameters(),
10                               lr=0.01)
11
12 class TransformerModel(nn.Module):
13     def __init__(self, transformer, input_len,
14                 target_len, n_features):
15         super().__init__()
16         self.transf = transformer
17         self.input_len = input_len
18         self.target_len = target_len
19         self.trg_masks = self.transf.
20             generate_square_subsequent_mask(
21                 self.target_len)

```

Transformer Model using nn.Transformer II

```

22     self.n_features = n_features
23     self.proj = nn.Linear(n_features,
24         self.transf.d_model)
25     self.linear = nn.Linear(self.transf.d_model,
26         n_features)
27
28     max_len = max(self.input_len, self.target_len)
29     self.pe = PositionalEncoding(max_len,
30         self.transf.d_model)
31     self.norm = nn.LayerNorm(self.transf.d_model)
32
33     def preprocess(self, seq):
34         seq_proj = self.proj(seq)
35         seq_enc = self.pe(seq_proj)
36         return self.norm(seq_enc)
37
38     def encode_decode(self, source, target, source_mask=None,
39         target_mask=None):
40         # Projections
41         # PyTorch Transformer expects L, N, F
42         src = self.preprocess(source).permute(1, 0, 2)

```



Transformer Model using nn.Transformer III

```

42     tgt = self.preprocess(target).permute(1, 0, 2)
43
44     out = self.transf(src, tgt,
45                       src_key_padding_mask=source_mask,
46                       tgt_mask=target_mask)
47
48     # Linear
49     # Back to N, L, D
50     out = out.permute(1, 0, 2)
51     out = self.linear(out) # N, L, F
52     return out
53
54     def predict(self, source_seq, source_mask=None):
55         inputs = source_seq[:, -1:]
56         for i in range(self.target_len):
57             out = self.encode_decode(source_seq, inputs,
58                                     source_mask=source_mask,
59                                     target_mask=self.trg_masks[:, i+1, :i+1])
60             out = torch.cat([inputs, out[:, -1:, :]],
61                             dim=-2)
62             inputs = out.detach()

```



Transformer Model using nn.Transformer IV

```

63         outputs = out[:, 1:, :]
64         return outputs
65
66     def forward(self, X, source_mask=None):
67         self.trg_masks = self.trg_masks.type_as(X)
68         source_seq = X[:, :self.input_len, :]
69
70         if self.training:
71             shifted_target_seq = X[:,
72                                     self.input_len-1:-1,
73                                     :]
74             outputs = self.encode_decode(source_seq,
75                                         shifted_target_seq,
76                                         source_mask=source_mask,
77                                         target_mask=self.trg_masks)
78         else:
79             outputs = self.predict(source_seq,
80                                   source_mask)
81
82         return outputs

```

code/TransformerModel.py



UNIVERSITY OF
**WESTERN
AUSTRALIA**



Take Aways

Keys to the famous transformer model:

- Positional Encoding to handle sequences
- Multi-Headed Narrow Attention for both computational efficiency and learning to attend to different aspects of the data
- Layer Normalisation to avoid gradient vanishing and over-fitting
- Deeply stacked encoder layers and decoder layers to handle complexity



References I

- [1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.
- [2] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of BERT's attention. In **Proceedings of the 2019 ACL Workshop Black-boxNLP: Analyzing and Interpreting Neural Networks for NLP**, pages 276–286, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-4828. URL <https://aclanthology.org/W19-4828>.
- [3] Daniel Voigt Godoy. **Deep Learning with PyTorch Step-by-Step: A Beginner's Guide**. Published at: <http://leanpub.com/pytorch>, Github Repo: <https://github.com/dvgodoy/PyTorchStepByStep>, 2021.



References II

- [4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, **Advances in Neural Information Processing Systems**, volume 30. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.

