CSE 676 Deeep Learning

Recurrent Neural Networks and Transformer

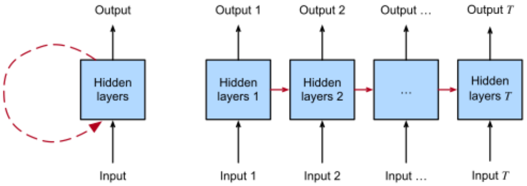
Kaiyi Ji

University at Buffalo

Course Content

Recurrent Neural Network

- Sequence of images, as in videos → sequentially structured prediction
- Countless learning tasks require dealing with sequential data.
  - Image Captioning
  - Speech Synthesis
  - Music Generation
- **Recurrent neural networks (RNNs)** are deep learning models that capture the dynamics of sequences via recurrent connections, which can be thought of as cycles in the network of nodes.



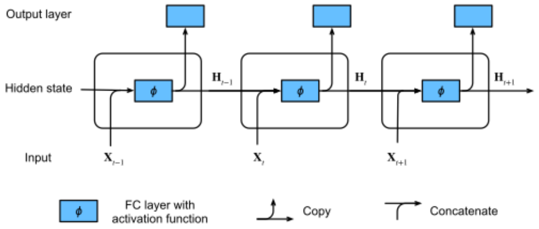
Dive into Recurrent Neural Network I

Recurrent Neural Network are neural networks with hidden states.

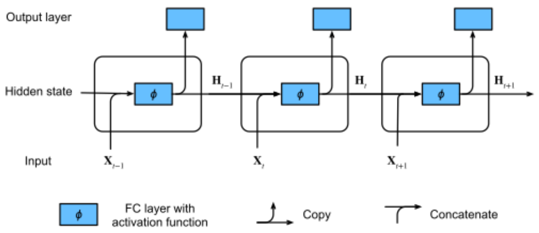
$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h)$$

For time step t, the output of the output layer is similar to the computation in the MLP:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$



Dive into Recurrent Neural Network II



- concatenating the input $\mathbf{X}_t$ at the current time step t and the hidden state $\mathbf{H}_{t-1}$ at the previous time step $t-1$;
- feeding the concatenation result into a fully connected layer with the activation function ϕ

Dive into Recurrent Neural Network III

Going more indepth, the calulation:

$$\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$$

```
X, W_xh = torch.randn(3, 1), torch.randn(1, 4)
H, W_hh = torch.randn(3, 4), torch.randn(4, 4)
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

```
tensor([[-1.6464, -8.4141,  1.5096,  3.9953],
[-1.2590, -0.2353,  2.5025,  0.2107],
[-2.5954,  0.8102, -1.3280, -1.1265]])
```

Dive into Recurrent Neural Network IV

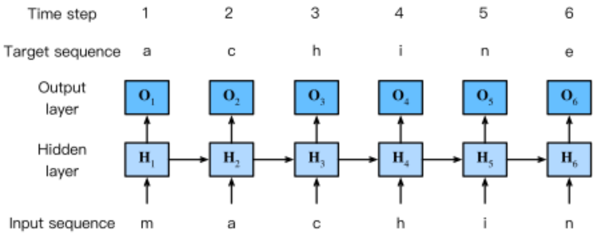
$$\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh}$$

is equivalent to matrix multiplication of concatenation of $\mathbf{X}_t$ and $\mathbf{H}_{t-1}$ and concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$

```
torch.matmul(torch.cat((X, H), 1), torch.cat((W_xh, W_hh), 0))
```

```
tensor([[-1.6464, -8.4141,  1.5096,  3.9953],
[-1.2590, -0.2353,  2.5025,  0.2107],
[-2.5954,  0.8102, -1.3280, -1.1265]])
```

RNN-based Character-Level Language Model



RNN Model

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h)$$

```
class RNNScratch():  #@save
"""The RNN model implemented from scratch."""
def __init__(self, num_inputs, num_hiddens, sigma=0.01):
    super().__init__()
    self.save_hyperparameters()
    self.W_xh = nn.Parameter(
        torch.randn(num_inputs, num_hiddens) * sigma)
    self.W_hh = nn.Parameter(
        torch.randn(num_hiddens, num_hiddens) * sigma)
    self.b_h = nn.Parameter(torch.zeros(num_hiddens))
```

RNN Model II

The forward methods defines how to compute the output andd hidden state at any time step, given the current input and the state of the model at the previous time step.

output:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$

hidden state:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h)$$

```python
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        state = torch.zeros((inputs.shape[1], self.num_hiddens),
                            device=inputs.device)
    else:
        state, = state
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = torch.tanh(torch.matmul(X, self.W_xh) +
                            torch.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
return outputs, state
```

Gradient Clipping I

RNN faces same issues as Multi-layer Perceptrons of exploding and vanishing gradient problem.

- pass through a chain of T layers

Vanishing Gradident Problem

- specialized architectures that were designed in hopes of mitigating the vanishing gradient problem.

Exploding Gradident Problem

- **Gradient Clipping** - Clip the gradients forcing the "clipped" gradients to take smaller values

Gradient Clipping II

Generally speaking, when optimizing some objective by gradient descent, we iteratively update the parameter of interest, $\mathbf{x}$

- we push $\mathbf{x}$ in the direction of the negative gradient $\mathbf{g}$
- with learning rate $\eta > 0$, each update takes the form $\mathbf{x} \leftarrow \mathbf{x} - \eta \mathbf{g}$

Let's assume that the objective function f is sufficiently smooth. (*Lipschitz continuou*s with constant L)

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|$$

Update the parameter vector by substracting $\eta \mathbf{g}$

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta \mathbf{g})| \leq L\eta\|\mathbf{g}\|$$

In other words, the objective cannot change by more than $L\eta\|\mathbf{g}\|$.

Gradient Clipping III

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|$$

When we say the gradient explode, we mean that $\|\mathbf{g}\|$ becomes excessively large.
When gradients can be so large, neural network training often diverges, failing to reduce the value of the objective.

- shrink the learning rate η to tiny values, slow down the training progress.
- popular alternative:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}$$

  This ensures that the gradient norm never exceed θ and that the updated gradient is entirely aligned with original direction of $\mathbf{g}$

Gradient Clipping IV

Equation

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}$$

Code Implementation

```
def clip_gradients(self, grad_clip_val, model):
    params = [p for p in model.parameters() if p.requires_grad]
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > grad_clip_val:
        for param in params:
            param.grad[:] *= grad_clip_val / norm
```

- The code calculates the L2-norm of the gradients ($\|\mathbf{g}\|$).
- If the L2-norm of the gradients is greater than the gradient clipping threshold (θ), the gradients are scaled by the factor $\frac{\theta}{\|\mathbf{g}\|}$.
- If the L2-norm of the gradients is less than or equal to the gradient clipping threshold, the gradients remain unchanged (i.e., they are multiplied by 1), which corresponds to the $\min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)$ part of the equation.

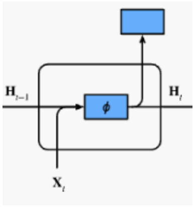
Modern Recurrent Neural Networks

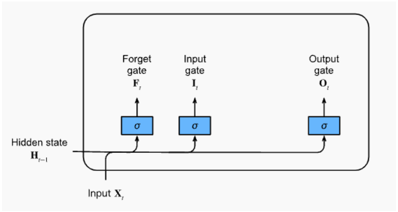
- Limitation of Recurrent Neural Network
  - Vanishing Gradients
- **Solution**:
  - Long Short-Term Memory (1997)
  - Bidirectional Recurrent Neural Network (1997)
- What information to keep and what not to keep? Locating key information.

Long Short-Term Memory (LSTM)

While gradient clipping helps with exploding gradients, handling vanishing gradients appears to require a more elaborate solution.

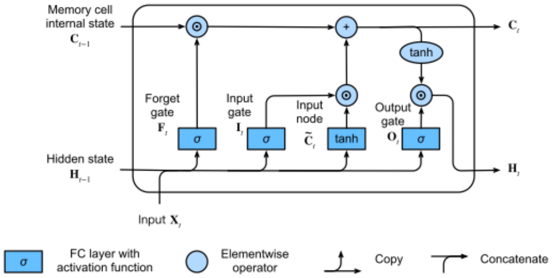
- LSTMs resemble standard recurrent neural networks but here each ordernary recurrent node is replaced by a memory cell.




Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
σ
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$

An Overview

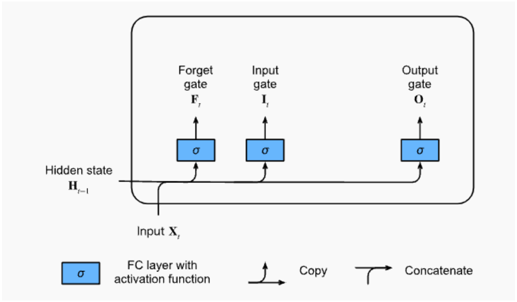
$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$



Gated Memory Cell

Each meory cell is equipped with an internal state and a number of multiplicative gates that determines whether

- a given input should impact the internal state (the *input gate*)
- the internal state should be flushed to 0 (the *forget gate*)
- the internal state of a given neuron should be allowed to impact the cell's output(the *output* gate).

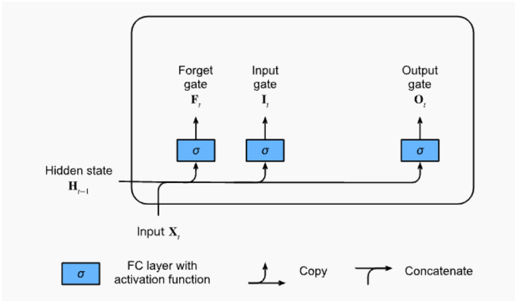

Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
σ
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation function
Copy
Concatenate

Gated Hidden State

- The key distinction between vanilla RNNs and LSTMs is that the latter support gating of hidden state.
- Dedicated mechanisms for when a hidden state should be *updated* and also when it should be *reset*.

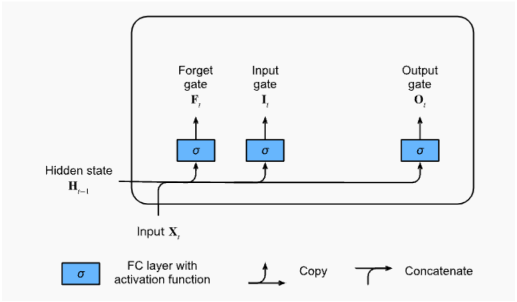
Input Gate, Forget Gate, and Ouput Gate I

The data feeding into the LSTM gates are the input at the current time step($\mathbf{X}_t$) and the hidden state of the previous time step($\mathbf{H}_{t-1}$)


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
σ
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation function
Copy
Concatenate

Three fully connected layers with sigmoid activation functions compute the values of the input, forget and output gates.

Input Gate, Forget Gate, and Ouput Gate II


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
FC layer with activation function
Copy
Concatenate

As a result of sigmoid activation, all values of the three gates are in the range of $(0, 1)$

- **input gate** determines how much of the **input node**'s value should be added to the current memory cell internal state.
- **forget gate** determines whether to keep the current value of the memory or flush it.
- **output gate** determines whether the memory cell should influence the output at the current time step.

Input Gate, Forget Gate, and Output Gate II

How do we represent these gates mathematically?

$$\begin{aligned}
\mathbf{I}_t &= \sigma\left(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i\right) \\
\mathbf{F}_t &= \sigma\left(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f\right) \\
\mathbf{O}_t &= \sigma\left(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o\right)
\end{aligned}$$

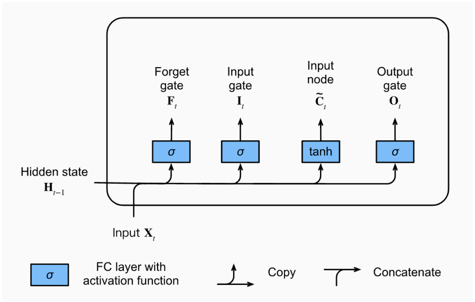
Note that the weight and bias parameters are **not the same** for each gate. Each gate has its own set of weight matrices and bias vectors, which allows the LSTM to learn different patterns and dependencies for each gate during the training process.

Why Sigmoid Activation?

- **Interpretability as probabilities**: The sigmoid function's output can be interpreted as probabilities, which is useful for gating mechanisms. For example, in the forget gate, a value close to 0 indicates "forgetting" the information, while a value close to 1 indicates "retaining" the information.
- **Element-wise multiplication**: Gates in an LSTM are used to modulate the information flow through element-wise multiplication with the memory cell's state or the input. The sigmoid function's range (0, 1) ensures that the resulting values are scaled between complete suppression (0) and no change (1), effectively controlling the information flow.
- **Non-linear activation**: Similar to the tanh function, the sigmoid function is also a non-linear activation function, allowing LSTMs to capture and model complex non-linear relationships in the input data.
- **Smooth differentiable function**: The sigmoid function has a smooth, differentiable curve, which is important for gradient-based optimization techniques like backpropagation.

Input Node I

Now let's progressively build the memory cell:


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation function
Copy
Concatenate

Similarly as before,

$$\tilde{\mathbf{C}}_t = \tanh\left(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c\right)$$

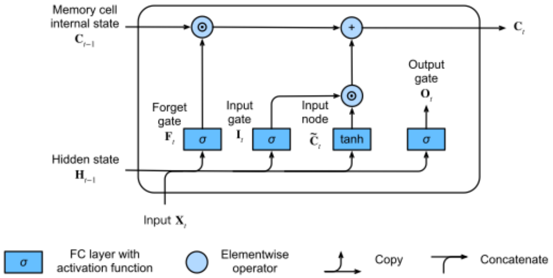
Input Node II: Why tanh?

$$\tilde{\mathbf{C}}_t = \tanh\left(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c\right)$$

- **Bounded output range**: The tanh function maps the input values to a range between -1 and 1. This helps in controlling the magnitude of the values in the memory cell and prevents potential issues with exploding or vanishing gradients during training.
- **Non-linear activation**: The tanh function is a non-linear activation function, which is essential for enabling the LSTM to model complex non-linear relationships in the input data.
- **Zero-centered**: Unlike the sigmoid function, which has an output range of (0, 1), the tanh function is zero-centered, meaning its output ranges from -1 to 1. This property helps in mitigating the effects of bias during training and provides more balanced updates to the weights.
- **Smooth differentiable function**: The tanh function has a smooth, differentiable curve, which is essential for gradient-based optimization techniques like backpropagation.

Memory Cell Internal State

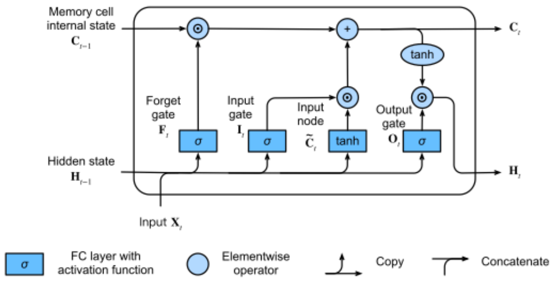
- In LSTMs,
  - the input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$
  - and the forget gate $\mathbf{F}_t$ addresses how much of the old cell internal state
- **Update Equation**: $\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$
  - If the forget gate is always 1 and the input gate is always 0, the memory cell internal state $\mathbf{C}_{t-1}$ will remain constant forever.
  - Input gates and forget gates give the model the flexibility to learn when to keep this value unchanged and when to perturb it in response to subsequent inputs.


Memory cell internal state $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
Output gate $\mathbf{O}_t$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

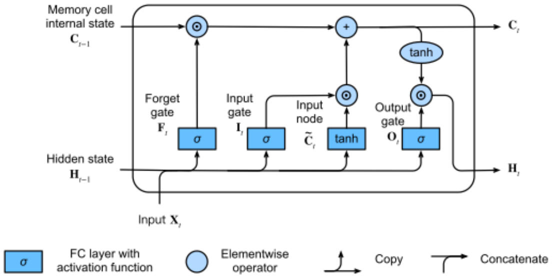
Hidden State I

Now, we start to finalize the memory cell,

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh\left(\mathbf{C}_t\right)$$


Memory cell internal state $\mathbf{C}_{t-1}$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
$\mathbf{C}_t$
$\mathbf{H}_t$
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

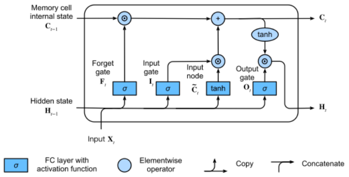
Hidden State II


Memory cell internal state $\mathbf{C}_{t-1}$
Hidden state $\mathbf{H}_{t-1}$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
$\mathbf{C}_t$
$\mathbf{H}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Why the tanh?

- This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$
- Whenever the output gate is close to 1, we allow the memory cell internal state to impact the subsequent layers uninhibited, vice versa

Dive into Long Short-Term Memory (LSTM) I



$$\mathbf{I}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i\right)$$
$$\mathbf{F}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f\right)$$
$$\mathbf{O}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o\right)$$
$$\tilde{\mathbf{C}}_t = \tanh\left(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c\right)$$

Dive into Long Short-Term Memory (LSTM) II

pseudocode (prepare the parameters)

```
        class LSTMScratch():
            def __init__():
                super().__init__()
                init_weight = lambda *shape: nn.Parameter(torch.randn(*shape) *
                ↪  sigma)
                triple = lambda: (init_weight(num_inputs, num_hiddens),
                                init_weight(num_hiddens, num_hiddens),
                                nn.Parameter(torch.zeros(num_hiddens)))
                self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
                self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
                self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
                self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

Dive into Long Short-Term Memory (LSTM) III

```
    def forward(self, inputs, H_C=None):
        if H_C is None:
            # Initial state with shape: (batch_size, num_hiddens)
            H = torch.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
            C = torch.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        else:
            H, C = H_C
        outputs = []
        for X in inputs:
            I = torch.sigmoid(torch.matmul(X, self.W_xi) +
                            torch.matmul(H, self.W_hi) + self.b_i)
            F = torch.sigmoid(torch.matmul(X, self.W_xf) +
                            torch.matmul(H, self.W_hf) + self.b_f)
            O = torch.sigmoid(torch.matmul(X, self.W_xo) +
                            torch.matmul(H, self.W_ho) + self.b_o)
            C_tilde = torch.tanh(torch.matmul(X, self.W_xc) +
                            torch.matmul(H, self.W_hc) + self.b_c)
            C = F * C + I * C_tilde
            H = O * torch.tanh(C)
            outputs.append(H)
        return outputs, (H, C)
```

Gated Recurrent Units (GRU)

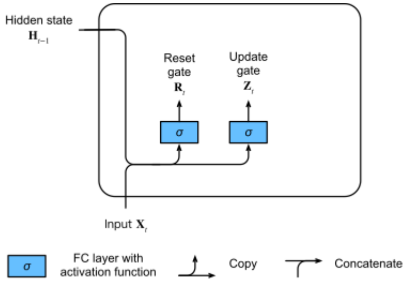
- As RNNs and particularly the LSTM architectures rapidly gained popularity during the 2010s.
- Papers began to experiment with simplified architectures in hopes of retaining the key idea of incorporating an internal state and multiplicative gating mechanisms but with the aim of speeding up computation.
- In 2014, gated recurrent unit (GRU) was born and offered a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of **being faster to compute**.

Reset Gate and Update Gate I

Here, the LSTM's three gates are replaced by two: the *reset gate* and the *update gate*.

- the reset gate controls how much of the previous state we might still want to remember
- an update gate would allow us to control how much of the new state is just a copy of the old state

Reset Gate and Update Gate II

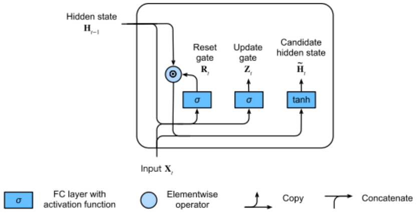

Hidden state $\mathbf{H}_{t-1}$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
σ
σ
Input $\mathbf{X}_t$
σ
FC layer with activation function
Copy
Concatenate

$$\mathbf{R}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r\right)$$
$$\mathbf{Z}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z\right)$$

Candidate Hidden State

$$\tilde{\mathbf{H}}_t = \tanh\left(\mathbf{X}_t\mathbf{W}_{xh} + \left(\mathbf{R}_t \odot \mathbf{H}_{t-1}\right)\mathbf{W}_{hh} + \mathbf{b}_h\right)$$

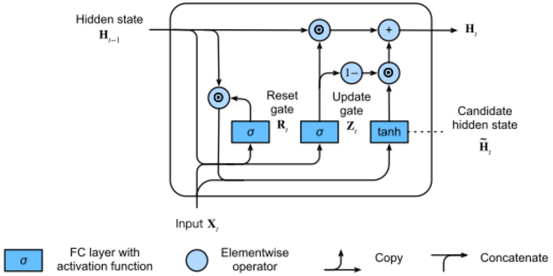
- The result is candidate, since we still need to incorporate the action of the update gate.
- Whenever the entries in the reset gate $\mathbf{R}_t$ are close to 1, we recover a vanilla RNN.
- For all entries of the reset gate $\mathbf{R}_t$ that are close to 0, the candidate hidden state is the result of an MLP with $\mathbf{X}_t$ as input



Hidden State I

Finally, we need to incorporate the effect of the update gate $\mathbf{Z}_t$.

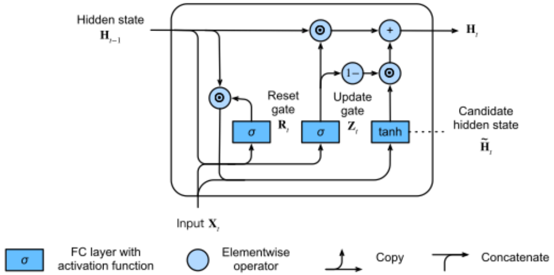
- This determines the extent to which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ matches the old state $\mathbf{H}_{t-1}$ versus how much it resembles the new candicate state $\tilde{\mathbf{H}}_t$.
- **Update Equation**: $\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$


Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
σ
σ
tanh
1−
+
Candidate hidden state $\tilde{\mathbf{H}}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Hidden State II

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

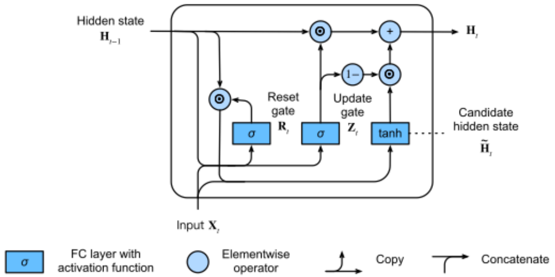
- Whenever the update gate $\mathbf{Z}_t$ is close to 1, we simply retain the old state. In this case the infromation from $\mathbf{X}_t$ is ignored, effectively skipping time step t in the dependency chain.
- Whenever $\mathbf{Z}_t$ is close to 0, the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$.


Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
σ
σ
tanh
1−
+
Candidate hidden state $\tilde{\mathbf{H}}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Dive into Gated Recurrent Unit (GRU) I

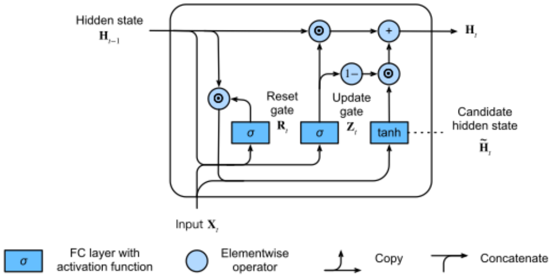
In summary, GRUs have the following two distinguish features:

- Reset gates help capture short-term dependencies in sequences.
  - The reset gate helps the model capture short-term dependencies by deciding which parts of the **previous hidden** state to "forget" or "reset" based on the **current input**.
- Update gates help capture long-term dependencies in sequences.
  - The update gate, on the other hand, helps the model capture long-term dependencies by deciding how much of the **current input and previous hidden state** to "keep" or "update" in the **current hidden state**.



Dive into Gated Recurrent Unit (GRU) II

$$\mathbf{R}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r\right)$$
$$\mathbf{Z}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z\right)$$
$$\tilde{\mathbf{H}}_t = \tanh\left(\mathbf{X}_t\mathbf{W}_{xh} + \left(\mathbf{R}_t \odot \mathbf{H}_{t-1}\right)\mathbf{W}_{hh} + \mathbf{b}_h\right)$$
$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$



Dive into Gated Recurrent Unit (GRU) III

$$\mathbf{R}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r\right)$$
$$\mathbf{Z}_t = \sigma\left(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z\right)$$
$$\tilde{\mathbf{H}}_t = \tanh\left(\mathbf{X}_t\mathbf{W}_{xh} + \left(\mathbf{R}_t \odot \mathbf{H}_{t-1}\right)\mathbf{W}_{hh} + \mathbf{b}_h\right)$$
$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

```
def forward(self, inputs, H=None):
    if H is None:
        # Initial state with shape: (batch_size, num_hiddens)
        H = torch.zeros((inputs.shape[1], self.num_hiddens),
                      device=inputs.device)
    outputs = []
    for X in inputs:
        Z = torch.sigmoid(torch.matmul(X, self.W_xz) +
                        torch.matmul(H, self.W_hz) + self.b_z)
        R = torch.sigmoid(torch.matmul(X, self.W_xr) +
                        torch.matmul(H, self.W_hr) + self.b_r)
        H_tilde = torch.tanh(torch.matmul(X, self.W_xh) +
                        torch.matmul(R * H, self.W_hh) + self.b_h)
        H = Z * H + (1 - Z) * H_tilde
        outputs.append(H)
    return outputs, H
```

Deep Recurrent Neural Networks I

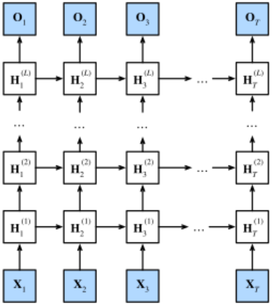
Up until now, we have focused on defining networks consisting of a sequence input, a single hidden RNN layer, and an output layer.

- In a sense, these networks are deep: inputs from the first time step can influence the outputs at the final time step T
- However, we often construct RNNs that are deep not only in the time direction but also in the input-to-output direction.

Deep Recurrent Neural Networks II

How to achieve this?

- we stack the RNNs on top of each other. Given a sequence of length T, the first RNN produces a sequence of outputs, also of length T. These, in turn, constitute the inputs to the next RNN layer.



Bidirectional Recurrent Neural Network I

So far,

- Language Modeling, where we aim to predict the next token given all previous tokens in a sequence.
- We wish only to condition upon the leftward context, unidirectional chaining.

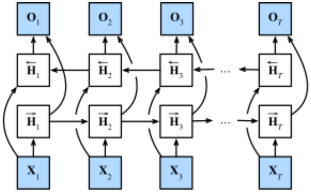
Now,

- It is perfectly fine to condition the prediction at every time step on both the leftward and the rightward context.
- Why shouldn't we take the context in both directions into account when assessing the part of speech associated with a given word?

Bidirectional Recurrent Neural Network II

A simple technique transforms any unidirectional RNN into a Bidirectional RNN

- Implement two undirectional RNN into a bidirectional RNN.
- Simply implement two unidirectional RNN layers chained together in opposite directions and acting on the same input.



Bidirectional Recurrent Neural Network III

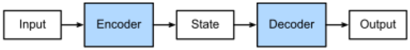
```
    def forward(self, inputs, Hs=None):
        f_H, b_H = Hs if Hs is not None else (None, None)
        f_outputs, f_H = self.f_rnn(inputs, f_H)
        b_outputs, b_H = self.b_rnn(reversed(inputs), b_H)
        outputs = [torch.cat((f, b), -1) for f, b in zip(
            f_outputs, reversed(b_outputs))]
        return outputs, (f_H, b_H)
```

- In bidirectional RNNs,
    - the hidden state for each time step is simultaneously determined by the data prior to and after the current time step.
    - Bidirectional RNNs are mostly useful for sequence encoding and the estimation of observations given bidirectional context.
    - Bidirectional RNNs are very costly to train due to long gradient chains.

The Encoder-Decoder Architecture

In general seq2seq problems like machine translation, inputs and outputs are of varying lengths that are unaligned.

- The standard approach to handling this sort of data is to design an encoder-decoder architecture consisiting of two major components:
  - *Encoder* takes a variable-length sequence as input
  - *Decoder* that acts as a conditional language model
- Machine translation from English to French as an example.
  - Given an input sequence in English: "They", "are", "watching", ".", this encoder-decoder architecture first encodes the variable-length input into a state, then decodes the state to generate the translated sequence, token by token, as output: "Ils", "regardent", "."



Attention Mechanisms and Transformers

Now let's talk about present and near future,

- While plenty of new methodological innovations made their way into most practitioner's toolkits—ReLU activations, residual layers, batch normalization, dropout, and adaptive learning rate schedules come to mind—the core underlying architectures were clearly recognizable as **scaled-up implementations of classic ideas**.
- At the present moment, the dominant models for nearly all natural language processing tasks are based on the Transformer architecture.
  - BERT, GPT and vision Transformer
  - "Attention is all you need"

Attention Is All You Need (NIPS 2017)

Abstract

- The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder.
- The best performing models connect encoder and decoder through an attention mechanism.
- The author propose a **a new simple network architecture**, the *transformer*.
  - based solely on attention mechanism
  - with no recurrence and convolution

What does the author achieve?

- replacing the recurrent layers most commonly used in encoder-decoder architecture with **multi-headed self-attention**.
- trained significantly faster than architectures based on recurrent or convolutional layers.
    - **State of the art** on **WMT 2014 English-to-German** and **WMT 2014 English-to-French translation** tasks.
    - Future work may extend Transformer to **images, audio and video.**

Introduction I

- Recurrent neural networks, LSTM, GRU been SOTA in sequence modeling and transduction problem (language modeling and machine translation).
- Pros and Cons of Recurrent Models:
  - **Pro**: When handling time series information, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t.
  - **Cons**: The inherently sequential nature precludes parallelization within training examples.
- **Cons**
  - When you are caluclating h_t, you have to make sure that h_{t-1} is done. If you have 100 words, you have to calculate 100 steps. (parallelization is very low)
  - Similar issue arise as you have to save every step up until h_{t-1}, the RAM consumption is also really high.

Introduction II

- Attention Mechanism
  - allowing modeling of dependencies without regard to their distance in the input or output sequence
  - attention mechanisms are used in conjunction with a recurrent network
- **Innovation**:
  - Transformer, a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output.
  - The Transformer allows for **significantly more parallelization** and can reach a new state of the art in translation quality after being trained for as little as twelve hours on eight P100 GPUs.

Background I

The author mentioned some of the previous attempts in reducing sequential computation:

- Replace recurrent neural network with convolution neural network
- however, more difficult to learn dependencies between distant positions.

The Transformer model addresses this issue by reducing the number of operations required to relate signals from two arbitrary input or output positions to a constant number, independent of their distance.

- This is achieved by using self-attention mechanisms, which allow the model to consider all positions simultaneously.
- However, there is a trade-off with this approach, as the effective resolution is reduced due to the averaging of attention-weighted positions. (*Multi-Head Attention*)

Multi-Head Attention

- improves the model's ability to learn long-range dependencies and maintain a higher effective resolution, while still benefiting from the reduced sequential computation enabled by the self-attention mechanism.

Background II

Self-attention

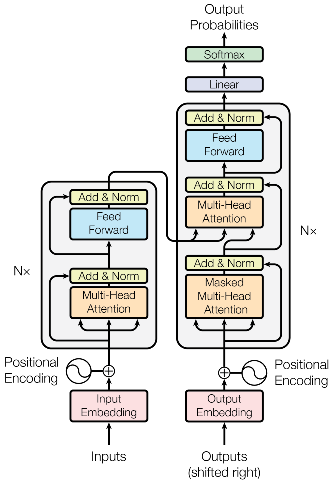
- called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.
- it has been used in other works before, it is **not the innovation** of this work

"To the best of our knowledge, however, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequencealigned RNNs or convolution."

Model Architecture I

- Most competitive neural sequence transduction models have an **encoder-decoder** structure.
  - **encoder** maps an input sequence of symbol representations $(x_1, \dots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \dots, z_n)$.
  - Given $\mathbf{z}$, the **decoder** then generates an output sequence $(y_1, \dots, y_m)$ of symbols one element at a time.
- At each step, the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

Model Architecture II



Encoder and Decoder Stacks I

Encoder

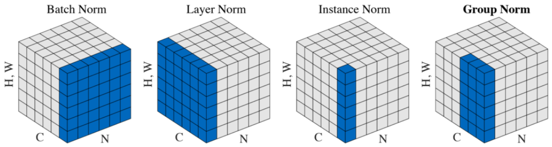
- $N = 6$ identical layers
  - Multi-head self-attention mechanism
  - A simple, position-wise fully connected feed-forward network
  - Residual Connection, output of dimension $d_{\text{model}} = 512$

Decoder

- $N = 6$ identical layers
  - A third layer Masked Multi-Head Attention Attention (Not able to see the future)

Encoder and Decoder Stacks II

Different Normalization Technique



Additional Reading: Deep Learning normalization methods

Attention

An attention function can be described

- as mapping a query and a set of key-value pairs to an output
- the output is computed as a weighted sum of the values
- the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Queries, Keys, and Values I

- So far all the network we reviewed crucially relied on the input being of a well-defined size.
- The images in ImageNet are of size 224×224 pixels and CNNs are specifically tuned to this size.
- In natural language processing the input size for RNNs is well defined and fixed.
  - Variable size is addressed by sequentially processing one token at a time, or by specially designed convolutioan kernels.
  - This approach can lead to significant problems when the input is truly of varying size with varying information content.
  - In particular, for long sequences it become quite difficult to keep track of everything that has already been generated or even viewed by the network

Queries, Keys, and Values II

Compare this to the databases.

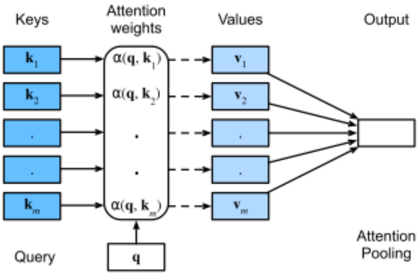
- In their simplest form they are collections of keys (k) and values (v).
- $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \dots (\mathbf{k}_m, \mathbf{v}_m)\}$ a database of m tuples of keys and values.

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\, \mathbf{v}_i$$

Attention: An Introduction I

- Typically referred to as *attention pooling*.

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \overset{\text{def}}{=} \sum_{i=1}^{m} \alpha\left(\mathbf{q}, \mathbf{k}_i\right) \mathbf{v}_i$$



Attention: An Introduction II

- Common strategy to ensure that the weights sum up to 1 is to normalize them via

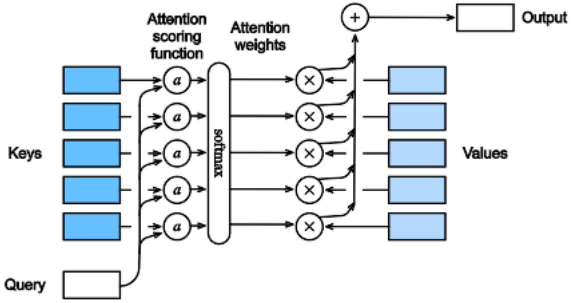
$$\alpha\left(\mathbf{q}, \mathbf{k}_i\right) = \frac{\alpha\left(\mathbf{q}, \mathbf{k}_i\right)}{\sum_j \alpha\left(\mathbf{q}, \mathbf{k}_j\right)}$$

- In particular, to ensure that the weights are also nonnegative, one can resort to exponentiation.

$$\alpha\left(\mathbf{q}, \mathbf{k}_i\right) = \frac{\exp\left(a\left(\mathbf{q}, \mathbf{k}_i\right)\right)}{\sum_j \exp\left(a\left(\mathbf{q}, \mathbf{k}_j\right)\right)}$$

Attention Scoring Functions

With softmax operation to ensure nonnegative attention weights, much of the work has gone into *attention scoring function a*



Attention Scoring Functions: Dot Product Attention I

Attention function from the Gaussian kernel:

$$\begin{aligned} a(\mathbf{q}, \mathbf{k}_i) &= -\frac{1}{2}\|\mathbf{q} - \mathbf{k}_i\|^2 \\ &= \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2}\|\mathbf{k}_i\|^2 - \frac{1}{2}\|\mathbf{q}\|^2. \end{aligned}$$

- Note the last term depends on **q**, normalizing the attention weights to 1 ensures this term disappear entirely.
- If the keys $\mathbf{k}_i$ are generated by a layer norm, their norms can be considered as approximately constant, and the middle term can be dropped.
- We need to keep the order of magnitude in the exponential function under control.

Attention Scoring Functions: Dot Product Attention II

- We need to keep the order of magnitude in the exponential function under control.

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}$$

- Note that attention α still need normalizing.

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \operatorname{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp\left(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d}\right)}{\sum_{j=1} \exp\left(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d}\right)}$$

Attention Scoring Functions: Convenience Functions

- Need a few functions to make the attention mechanism efficient to deploy.
- Tools to deal with strings of variable length (common for natural language processing).
- Tools for efficient evaluation on minibatches (batch matrix multiplication).

Convenience Functions: Masked Softmax Operation I

- One of the most popular applications of the attention mechanism is to sequence models.
- We need to deal with sequences of different lengths.
  - such sequences may end up in the same minibatch, necessitating padding with dummy tokens for shorter sequences.
  - these special tokens do not carry meaning
- For instance, assume we have the following three sentences:

```
Dive  into  Deep    Learning
Learn to    code    <blank>
Hello world <blank> <blank>
```

Convience Functions: Masked Softmax Operation II

For instance, assume we have the following three sentences:

Dive	into	Deep	Learning
Learn	to	code	<blank>
Hello	world	<blank>	<blank>

- We do not want the model to attend on the special padding tokens.
- **Masked Softmax Operation**
  - $\sum_{i=1}^{n} \alpha\left(\mathbf{q}, \mathbf{k}_i\right) \mathbf{v}_i \rightarrow \sum_{i=1}^{l} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ for however long the actual sentence is.

Convenience Functions: Batch Matrix Multiplication

- Another commonly used operation is to multiply batches of matrices wioth another. This comes in handy when we have minibatches of queries, keys, and values.

$$\mathbf{Q} = [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n] \in \mathbb{R}^{n \times a \times b}$$

$$\mathbf{K} = [\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_n] \in \mathbb{R}^{n \times b \times c}$$

- Then the batch matrix multiplication (BMM) computes the element-wise product:

$$\mathrm{BMM}(\mathbf{Q}, \mathbf{K}) = [\mathbf{Q}_1\mathbf{K}_1, \mathbf{Q}_2\mathbf{K}_2, \ldots, \mathbf{Q}_n\mathbf{K}_n] \in \mathbb{R}^{n \times a \times c}$$

Scaled Dot-Product Attention: Introduction

- In sequence models, the attention mechanism is used to assign different levels of importance to different inputs.
- In the dot-product attention mechanism, the importance of an input (the "key") is determined by the dot product of the key with a "query".
- Both the query and the key are vectors of the same length, d.
  - If the dimensions of the query and key do not match, we can use a transformation matrix $\mathbf{M}$ to adjust the key vector to the same space as the query vector.
  - For simplicity, we will assume in this presentation that the dimensions of the query and key vectors match.

Minibatches and Dimensionality

- Minibatches are subsets of the training data used for one iteration of model updates.
    - The use of minibatches can improve computational efficiency and balance the quality of the gradient estimate.
- In the context of attention mechanisms, we might compute attention for n queries and m key-value pairs in one minibatch.
    - The queries and keys are vectors of length d, while values are vectors of length v.
    - These dimensions are important as they determine the shapes of the matrices involved in the attention computation.

Scaled Dot-Product Attention Equation

$$\mathrm{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times v}. \tag{1}$$

- The scaled dot-product attention equation represents a weighted sum of the values.
- Weights are determined by the scaled dot-product of the queries and keys.
- The scaling factor of $\sqrt{d}$ helps to prevent the dot-product values from growing too large.
- The softmax function provides a probability distribution over the keys for each query.
- The final output is a matrix of size $n \times v$, where n is the number of queries and v is the length of the value vectors.

Additive Attention: Introduction

- **Recall** in Scaled Dot-Product Attention, we assumed query and key as vectors of same length d.
- Queries and keys can also be vectors of different dimensions.
- There are two ways to handle this mismatch:
  - Use a matrix $\mathbf{q}^\top \mathbf{M} \mathbf{k}$.
  - Use **Additive Attention** as the scoring function.

Additive Attention Equation

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^T \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}$$

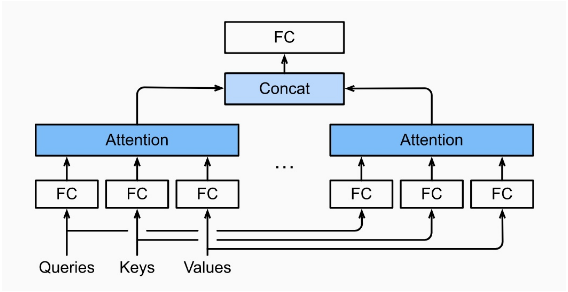
- $\mathbf{W}_q \in \mathbb{R}^{h \times q}$, $\mathbf{W}_k \in \mathbb{R}^{h \times k}$, and $\mathbf{w}_v \in \mathbb{R}^h$ are learnable parameters.
- To ensure non-negativity and normalization, we apply softmax to this equation.

Multi-Head Attention

- In processing queries, keys, and values, leveraging different behaviors of the attention mechanism can be advantageous.
- Different attention mechanism behaviors capture dependencies of varying ranges within a sequence.
  - Some behaviors may be better at capturing shorter-range dependencies.
  - Some may be better at capturing longer-range dependencies.
- Multi-head attention combines knowledge of the same attention pooling via different representation subspaces of queries, keys, and values.

Multi-Head Attention: Introduction

- Instead of a single attention pooling step, the queries, keys, and values undergo separate linear projections, each learned independently.
- These projected queries, keys, and values are then simultaneously processed through attention pooling.
- The resulting outputs from the attention pooling are concatenated and transformed through another learned linear projection, yielding the final output.
- Each output from the attention pooling is referred to as a head in the multi-head attention design.



Multi-Head Attention Equation

- Each attention head $\mathbf{h}_i (i = 1, \ldots, h)$ can be computed as:

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v},$$

  - where learnable parameters $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}, \mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ and $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$, and f is attention pooling
- The multi-head attention output is another linear transformation via learnable parameters $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$ of the concatenation of h heads:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}.$$

Self-Attention

- With attention mechanism, when calculating the representation of a token in the next layer, a token can attend to every other token by utilizing its query vector and matching against their key vectors.
- With the full set of query-key compatibility scores, we can compute the representation of each token by performing a weighted sum over the other tokens.
- Since tokens attend to each other, such architectures are described as *self-attention* models or *intra-attention* models.

Self-Attention Equation

$$\mathbf{y_i} = \mathbf{f}(\mathbf{x_i}, (\mathbf{x_1}, \mathbf{x_1}), ..., (\mathbf{x_n}, \mathbf{x_n})) \in \mathbb{R}^d$$

- where x_1 to x_n are input tokens and $x_1 \in \mathbb{R}^d (1 <= i <= n)$
- y_1 to y_n are the outputs of the self attention having the same length.

Positional Encoding

- Self-attention replaces sequential operations with parallel computation for more efficient processing.
- Self-attention by itself does not preserve the order of the sequence.
- The dominant method for retaining the order of tokens is to introduce an extra input associated with each token and present it to the model.
- These inputs are called positional encodings. and they can either be learned or fixed a priori.
- Suppose the input representation $\mathbf{X} \in \mathbb{R}^{n \times d}$ contains d-dimensional embeddings for n tokens of a sequence.
- The positional encoding outputs $\mathbf{X} + \mathbf{P}$ using a positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$ f the same shape.
- The element of the i^{th} and the $(2j)^{\text{th}}$ or the $(2j+1)^{\text{th}}$ column is

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$$
$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$$