Assignment #3

RNNs and Attention

1.1.2 Effect of Activation - Single Nueron RNN

$$z_i = Relu(w_i.(z_{i-1})) = \begin{cases} w_i \cdot z_{i-1} & \text{if } w_i \cdot z_{i-1} > 0 \\ 0 & \text{else} \end{cases} \qquad \frac{\partial z_i}{\partial z_{i-1}} = \begin{cases} w_i & \text{if } w_i \cdot z_{i-1} > 0 \\ 0 & \text{else} \end{cases} \qquad \frac{\partial f(x)}{\partial x} = \prod_{i=1}^{N} \frac{\partial z_i}{\partial z_{i-1}}$$

- if $w_i \cdot z_{i-1} \leq 0$ for any nueron then $\frac{\partial z_i}{\partial z_{i-1}}$ for that nueron is 0 hence $\frac{\partial f(x)}{\partial x} = 0$

- if $w_i \cdot z_{i-1} > 0$ all nuerons then $\left|\frac{\partial f(x)}{\partial x}\right| = \prod_{i=1}^{N} |w_i|$

Hence $0 \leq \left|\frac{\partial f(x)}{\partial x}\right| \leq \prod_{i=1}^{N} |w_i|$ The gradient vanishing or exploding depends on the the number of recurrent units (N) and the value of the weights.

1.2.1 Gradient through RNN - Matrices and RNN

$$x_{t+1} = \text{Sigmoid}(Wx_t) \qquad \frac{\partial x_n}{\partial x_1} = \prod_{i=1}^{n} \frac{\partial x_i}{\partial x_{i-1}} \qquad \sigma'(z_t) = \sigma(z_t)(1-\sigma(z_t))$$

$$\frac{\partial x_n}{\partial x_{n-1}} = \frac{\partial \sigma(Wx_{n-1})}{\partial x_{n-1}} = \begin{bmatrix} \sigma'(Wx_{n-1}) & 0 & 0 & 0 \\ 0 & \sigma'(Wx_{n-1}) & & 0 \\ 0 & 0 & \ddots & \sigma'(Wx_{n-1}) \end{bmatrix} W$$

using $C = AB$ $\quad \sigma_{max}(C) \leq \sigma_{max}(A)\,\sigma_{max}(B)$

$$\sigma_{max}\left(\frac{\partial x_n}{\partial x_{n-1}}\right) \leq \sigma_{max}\left(Diag\left(\sigma'(Wx_{n-1})\right)\right) \times \cancel{\sigma_{max}}(W) \; \tfrac{1}{4}$$

$\sigma_{max}\left(Diag\left(\sigma'(Wx_{n-1})\right)\right) = 1/4$ This value is maximized when $z = Wx_{n-1} = 0 \longrightarrow \sigma'(z) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) \rightarrow \sigma'(0) = \frac{1}{2}\left(1-\frac{1}{2}\right) = 1/4$

$$\sigma_{max}\left(\frac{\partial x_n}{\partial x_{n-1}}\right) \leq \overbrace{1/4 \times 1/4}^{1/16} \xrightarrow{\text{since } \frac{\partial x_n}{\partial x_1} = \prod_{i=1}^{n} \frac{\partial x_i}{\partial x_{i-1}}} \sigma_{max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(\frac{1}{16}\right)^{n-1} \longrightarrow n-1 \text{ bc/ of } Wx_{n-1}$$

Since Singular Values are non negative the lower bound is 0 $\quad 0 \leq \sigma_{max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(1/16\right)^{n-1}$

1.3.1 Implement a 1D Convolution - Relative Attention

$W = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$ $\quad \text{Conv1D}(x;w)_i = \sum_{j=-1}^{1} x_{i+j}\, w_{j+2} = 2x_{i-1} + 0x_i + 1x_{i+1}$ $\quad \text{Softmax}(a_{ij}) = z_{ij}$

z values need to sum to 1 bc of softmax

$y_i = \sum_{i=1}^{n} z_{ij} V_j \xrightarrow{\text{we want}} y_i = z_{i,i-1} V_{i-1} + z_{i,i+1} V_{i+1}$ $\quad z_{i,i-1} = 2/3$ $\quad z_{i,i+1} = 1/3$ $\quad$ anything else $z_{ij} = 0$

$$a_{ij}(Q,k,p) = \left(\frac{Q_i k_j}{\sqrt{dk}} + p_{i-j}\right) = \left(\frac{W_Q x_i W_k x_j}{\sqrt{dk}} + p_{i-j}\right)$$

Since the coefficient of x_i is 0 and we don't want any x_i terms set $W_Q = 0$ which eliminates the $\frac{Q_i k_j}{\sqrt{dk}}$ term hence W_k from k_j can be anything. In this case we set it to 0 so $W_k = 0$.

So $z_{ij} = \text{softmax}(a_{ij}) = \text{softmax}(p_{i-j})$

if $j \neq i-1$ or $i+1$ then z_{ij} needs to be zero $\longrightarrow$ hence in these cases $p_{i-j} = -\infty$

for $z_{i,i-1}$ and $z_{i,i+1}$:

$$\left.\begin{aligned} z_{i,i-1} = \tfrac{2}{3} &\longrightarrow \frac{e^{p_{i-(i-1)}}}{e^{p_{i-(i-1)}} + e^{p_{i-(i+1)}}} = \tfrac{2}{3} \\ z_{i,i+1} = \tfrac{1}{3} &\longrightarrow \frac{e^{p_{i-(i+1)}}}{e^{p_{i-(i-1)}} + e^{p_{i-(i+1)}}} = \tfrac{1}{3} \end{aligned}\right\} \quad \begin{aligned} p_{i-(i-1)} &= p_1 = \ln(2) \\ p_{i-(i+1)} &= p_{-1} = \ln(1) = 0 \end{aligned}$$

$$y_i = z_{i,i-1} V_{i-1} + z_{i,i+1} V_{i+1} = \tfrac{2}{3} W_V x_{i-1} + \tfrac{1}{3} W_V x_{i+1} \xrightarrow{\text{needs to be equivalent}} y_i = 2x_{i-1} + 1x_{i+1} \longrightarrow W_V = 3I$$

Parameters:

$W_Q = 0 \quad W_k = 0 \quad W_V = 3I \quad p_1 = \ln(2) \quad p_{-1} = 0 \quad$ any other $p_{ij} = -\infty$

1.3.2 Implement Max Pooling - Relative Attention

$a_{ij}(Q,k,p) = \left(\frac{Q_i k_j}{\sqrt{dk}} + p_{i-j}\right) \qquad z_{ij} = \text{Softmax}(a_{ij}) \qquad y_i = \sum_{j=1}^{n} z_{ij} V x_j$

Strategy: if we have softmax(x) the value outputted will be representative of the how large the value is compared to other x_{ij} values

z_{ij} needs to represent how large x_{ij} is. Also need to 0 out the z_{ij} values not in the convolution window.

Eliminate what's not in the convolution window.

With window of $-k \leq m \leq k$ $\quad$ $p_{i-j} = 0$ if $|i-j| \leq k$ $\quad$ only $i\text{-}j$ values between $-k$ and k will have a $z_{i,j}$ value

$p_{i-j} = -\infty$ if $|i-j| > k$ $\quad$ This would make the softmax 0

Make $\alpha_{i,j}$ as large as possible for max x_j:

if x_j is multiplied by a large constant then the max value dominates all the other terms in softmax because in $\frac{e^{x_j}}{\sum e^{x_j}}$ with x_j being max the denominator is dominated by x_j hence the value of the softmax will be nearly 1. So the output of the softmax would be a one hot encoding of the max x_j.

$$\alpha_{i,j}(Q,k,p) = \left(\frac{Q_i k_j}{\sqrt{d_k}}\right) = \left(\frac{W_Q x_i W_T x_j}{\sqrt{d_k}}\right)$$

To make x_j as large as possible make it's coefficient as large as possible so W_Q and W_k need to be large constants $W_Q = C$ $\quad W_k = C$

$$a_{i,j} = \left(\frac{C^2 x_i x_j}{\sqrt{d_k}}\right) \text{ for } |i-j| \leq k = \text{one hot encoding of max } x_j \text{ in rang } -k \text{ to } k$$

$$y_i = \sum_j^N \text{one hot encoding of max } x_j \cdot W_V x_j \xrightarrow{\text{Set } W_V = I} \max_i = (\text{one hot encoding of max } x_j)_i \cdot x_j = z_{i,j} \cdot x_j \approx \max_{-k \leq m \leq k} x_{j+m}$$

Parameters

W_Q = large Constant (C) $\quad$ W_k = large Constant (C) $\quad$ $W_V = I$ $\quad$ $p_{i-j} = 0$ if $|i-j| \leq k$ $\quad$ $p_{i-j} = -\infty$ if $|i-j| > k$

Programming Assignment

Neural Machine Translation (NMT)

2.1

1. Scaled Dot Attention

```python
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=2)
        self.scaling_factor = torch.rsqrt(
            torch.tensor(self.hidden_size, dtype=torch.float)
        )

    def forward(self, queries, keys, values):
        """The forward pass of the scaled dot attention mechanism.

        Arguments:
            queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_size)
            keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
            values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

        Returns:
            context: weighted average of the values (batch_size x k x hidden_size)
            attention_weights: Normalized attention weights for each encoder hidden state. (batch_size x k x seq_len)

            The output must be a softmax weighting over the seq_len annotations.
        """

        #MY CODE
        batch_size = queries.size(0)
        q = self.Q(queries) # [batch_size, k, hidden_size]
        k = self.K(keys)    # [batch_size, seq_len, hidden_size]
        v = self.V(values)  # [batch_size, seq_len, hidden_size]

        unnormalized_attention = torch.bmm(q, k.transpose(1, 2)) * self.scaling_factor    # [batch_size, k, seq_len]
        attention_weights = self.softmax(unnormalized_attention)                          # [batch_size, k, seq_len]
        context = torch.bmm(attention_weights, v)                                         # [batch_size, k, hidden_size]

        return context, attention_weights
```

2. Causal Scaled Dot Attention

```python
class CausalScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(CausalScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size
        self.neg_inf = torch.tensor(-1e7)

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=2)
        self.scaling_factor = torch.rsqrt(
            torch.tensor(self.hidden_size, dtype=torch.float)
        )

    def forward(self, queries, keys, values):
        """The forward pass of the scaled dot attention mechanism.

        Arguments:
            queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_size)
            keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
            values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

        Returns:
            context: weighted average of the values (batch_size x k x hidden_size)
            attention_weights: Normalized attention weights for each encoder hidden state. (batch_size x seq_len x k)

            The output must be a softmax weighting over the seq_len annotations.
        """

        #MY CODE
        batch_size = queries.size(0)
        q = self.Q(queries) # [batch_size, k, hidden_size]
        k = self.K(keys)    # [batch_size, seq_len, hidden_size]
        v = self.V(values)  # [batch_size, seq_len, hidden_size]

        unnormalized_attention = torch.bmm(q, k.transpose(1, 2)) * self.scaling_factor   # [batch_size, k, seq_len]
        k = queries.size(1)
        seq_length = keys.size(1)
        mask = torch.triu(torch.ones(k, seq_length, device = queries.device), diagonal=1).bool().unsqueeze(0).expand(batch_size, -1, -1) # [batch_size, k, seq_len]
        unnormalized_attention = unnormalized_attention.masked_fill(mask, self.neg_inf)              # [batch_size, k, seq_len]
        attention_weights = self.softmax(unnormalized_attention)                                     # [batch_size, k, seq_len]
        context = torch.bmm(attention_weights, v)                                                    # [batch_size, k, hidden_size]
        attention_weights = attention_weights.transpose(1, 2)                                        # [batch_size, seq_len, k]

        return context, attention_weights
```

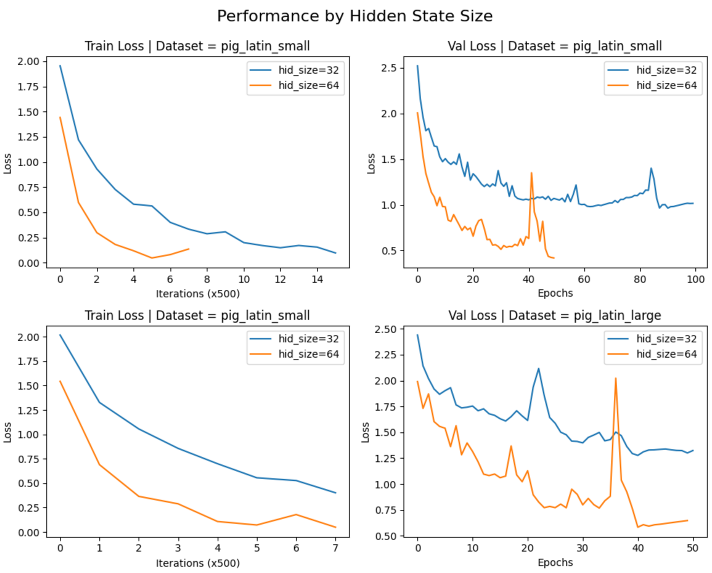
3. Why positional encoding?

Since transformers process all the tokens in parallel they would not be able to know in what order the tokens apear. This information is very important for sequential data such as text, hence positional Encodings are used to give the model information about the order of the inputs.

Why sinusoidal?

Sinusoidal pos encodings can be applied to a varying length of inputs (unlike one hot encoding) and still repesent the distance between the tokens.

4.



Lowest Validation Loss :

	32 hidden layers	64 hidden layers
Big Data	1.277	0.583
Small Data	0.963	0.417

- When trained on the small dataset, the large model learned faster due to its higher representational capacity but began overfitting, as indicated by early stopping after no improvement in validation loss for 10 epochs.
- In contrast, the small model generalized better and avoided overfitting but trained more slowly and never achieved a lower loss than the large model.
- Both models performed better on the smaller dataset than on the larger one in terms of validation loss.
- Notably, model size had a greater impact on performance than dataset size. Training loss trends were more consistent across different dataset sizes with the same model than across different model sizes with the same dataset.

2.2 Decoder only NMT

1.

```
[ ] def generate_tensors_for_training_decoder_nmt(src_EOP, tgt_EOS, start_token, cuda):
        # ------------
        # FILL THIS IN
        # ------------
        # Step1: concatenate input_EOP, and target_EOS vectors to form a target tensor.
        # src_EOP_tgt_EOS =
        # Step2: make a sos vector
        # sos_vector =
        # sos_vector = to_var(sos_vector, cuda)
        # Step3: make a concatenated input tensor to the decoder-only NMT (format: Start-of-token source end-of-prompt target)
        # SOS_src_EOP_tgt =
        src_EOP_tgt_EOS = torch.cat([src_EOP,tgt_EOS], dim=1)
        batch_size = src_EOP.size(0)
        sos_vector = torch.full((batch_size, 1), start_token, dtype=src_EOP.dtype)
        sos_vector = to_var(sos_vector, cuda)
        SOS_src_EOP_tgt = torch.cat([sos_vector,src_EOP, tgt_EOS[:,:-1]], dim=1)

        return SOS_src_EOP_tgt, src_EOP_tgt_EOS
```

2.

```
def forward(self, inputs):
    """Forward pass of the attention-based decoder RNN.

    Arguments:
        inputs: Input token indexes across a batch for all the time step. (batch_size x decoder_seq_len)
    Returns:
        output: Un-normalized scores for each token in the vocabulary, across a batch for all the decoding time steps. (batch_size x decoder_seq_len x vocab_size)
        attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_len x decoder_seq_len)
    """
    # ------------
    x = self.embedding(inputs)                                              # [batch_size, decoder_seq_len, hidden_size]
    x = x + self.positional_encodings[:x.size(1), :].unsqueeze(0).to(x.device)    # [batch_size, decoder_seq_len, hidden_size]

    self_attention_weights = []
    for i in range(self.num_layers):
        context, self_attention_weights_i = self.self_attentions[i](x, x, x)
        self_attention_weights.append(self_attention_weights_i)

        x = x + context

        x = x + self.attention_mlps[i](x)

    output = self.out(x)
    # ------------
    return output, self_attention_weights
```

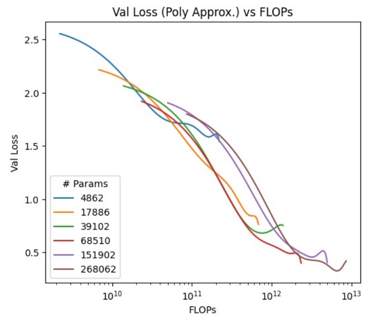
3. Lowest validation Loss: 0.428

One advantage of the encoder-only architecture compared to the encoder decoder architecture is its simplicity. However, it also has a reduced representational capacity. For example, encoder decoder models can incorporate mechanisms like cross attention layers, which enhance their ability to capture complex relationships in the data.

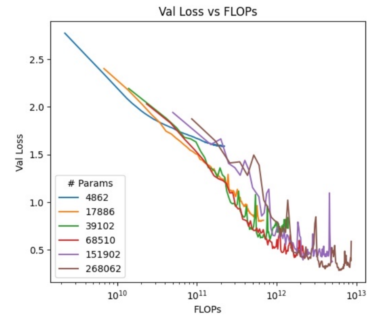
When comparing a decoder-only model trained on a smaller dataset with 128 hidden layers to an encoder decoder model trained on the same dataset with 64 hidden layers, the lowest validation losses achieved are quite similar 0.428 and 0.417, respectively. As expected, the encoder decoder model performs slightly better due to its higher representational capacity. Still, the performance gap is relatively small.

Overall, we can conclude that for simpler tasks, a decoder only model can perform nearly as well as an encoder decoder model. However, for more task the additional representational power of the encoder decoder architecture can offer meaningful benefits.

2.3 Scaling law and IsoFlop profiles

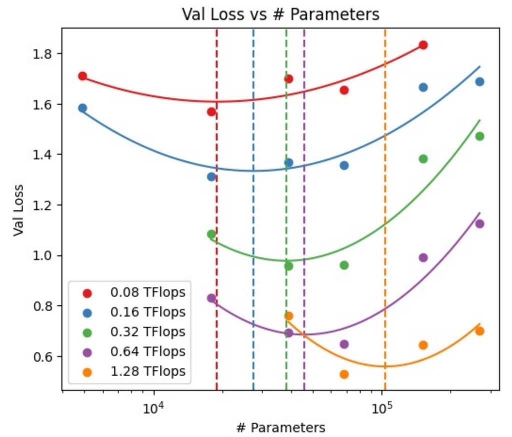


1. • The graphs show that as model size increases, validation loss generally decreases, accompanied by a rise in FLOPs. Smaller models initially perform better at lower FLOP counts, but larger models eventually achieve lower losses as the FLOPs count increases. However, diminishing returns are can be seen that beyond a certain point (bottom right of the graph), increased FLOPs and model size result in minimal to no improvements in validation loss.



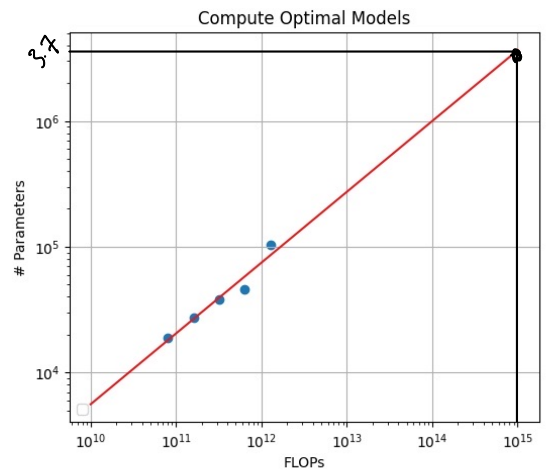
2.

```python
def find_optimal_params(x, y):
    # ------------
    # FILL THIS IN
    # ------------
    p = np.polyfit(np.log10(x), y, 2)
    log_optimal_params = -p[1]/(2*p[0]) #vertex
    optimal_params = 10 ** log_optimal_params
    return p, optimal_params
```



3.

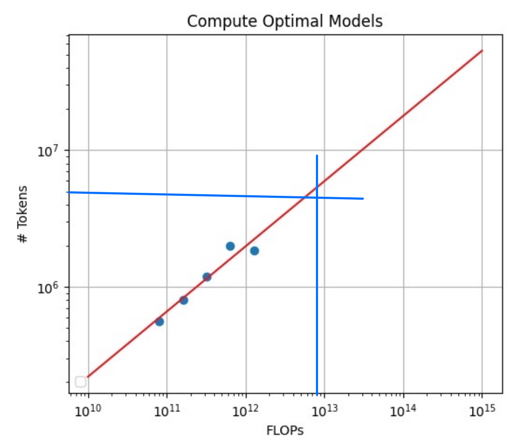
```python
[ ] def fit_linear_log(x, y):
        # ------------
        # FILL THIS IN
        # ------------
        m, c = np.polyfit(np.log10(x), np.log10(y), 1)
        return m, c
```



The optimal number of parameters for 1e15 flop : 3.7×10^6

4.

The Total number of FlOPS is 8.6T total number of Tokens is 6.6 M based by looking at the Token and flops graph, it can be deduced that the model's training is not optimized at it would benefit from an increase in the size of the input data or a decrease in the size of the model.



3. Fine tuning Pretrained LM

1.

```python
from transformers import BertModel
import torch.nn as nn

class BertForSentenceClassification(BertModel):
    def __init__(self, config):
        super().__init__(config)

        ##### START YOUR CODE HERE #####
        # Add a linear classifier that map BERTs [CLS] token representation to the unnormalized
        # output probabilities for each class (logits).
        # Notes:
        #  * See the documentation for torch.nn.Linear
        #  * You do not need to add a softmax, as this is included in the loss function
        #  * The size of BERTs token representation can be accessed at config.hidden_size
        #  * The number of output classes can be accessed at config.num_labels
        self.classifier = torch.nn.Linear(config.hidden_size, config.num_labels)
        ##### END YOUR CODE HERE #####
        self.loss = torch.nn.CrossEntropyLoss()

    def forward(self, labels=None, **kwargs):
        outputs = super().forward(**kwargs)
        ##### START YOUR CODE HERE #####
        # Pass BERTs [CLS] token representation to this new classifier to produce the logits.
        # Notes:
        #  * The [CLS] token representation can be accessed at outputs.pooler_output
        cls_token_repr = outputs.pooler_output
        logits = self.classifier(cls_token_repr)
        ##### END YOUR CODE HERE #####
        if labels is not None:
            outputs = (logits, self.loss(logits, labels))
        else:
            outputs = (logits,)
        return outputs
```

3. Training Time:

When BERT's weights were frozen, the training time was significantly reduced compared to fine tuning. This is because fewer parameters are being updated, resulting in less FLOPs and faster computation per epoch.

Validation Accuracy:

BERT with frozen weights achieved a validation accuracy of 74.5%, while fine-tuning all weights led to an accuracy above 92%. This is because frozen weights reduces the model's capacity to learn form new data

4.

The fine tuned BERTweet model achieves lower validation accuracy of 71.9% compared to MathBERT. This is likely because MathBERT was trained on math related data, which aligns more closely with the purpose of our model, allowing it to better understand and represent the math language. However, BERTweet was trained on social media data, making its pretrained weights less suitable for the math related task.

4 Connecting Text and Image with CLIP

my caption = "3 Clown fish infront of a Coral"

Finding the caption was easy as I got it on my first try. I used the hint given in the notebook. "Be short and descriptive".