Intro

1. The perceptron: forward propagation

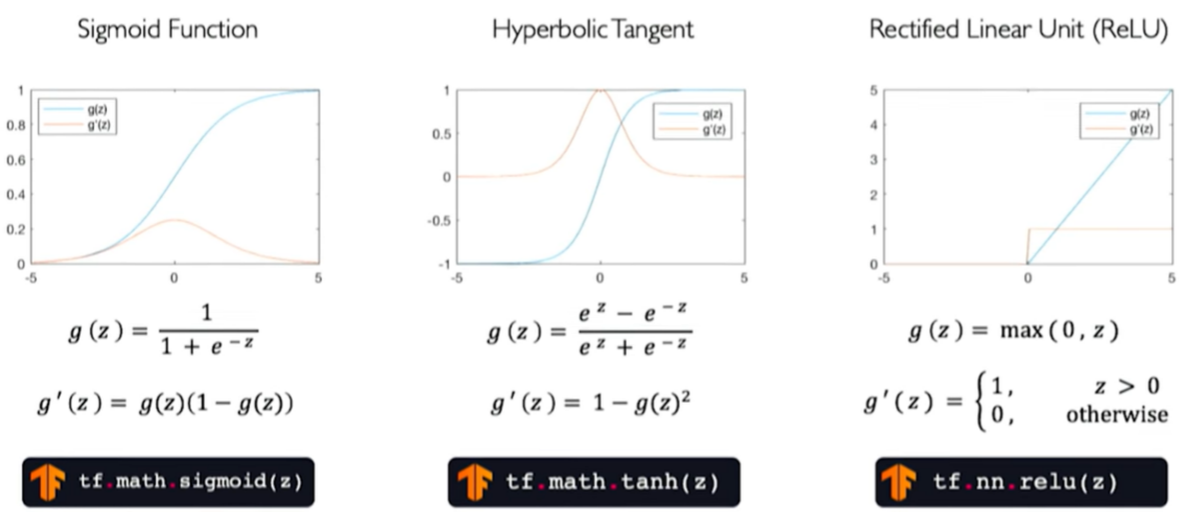
- Structural building blocks
- Nonlinear activation functions

2. Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpagation

3. Training in Practice

- Adaptive learning
- Batching
- Regularization




```python # Dense layer from scratch class MyDenseLayer(tf.keras.layers.Layer): def _init_(self, input_dim, output_dim): super(MyDenseLayer, self)._int_()
```

```python
# Initialize weights and bias
self.W = self.add_weight([input_dim, output_dim])
self.b = self.add_weight([1, output_dim])

def call(self, inputs):
  # Forward propagate the inputs
  z = tf.matmul(inputs, self.W) + self.b

  # Feed through a non-linear activation
  output = tf.math.sigmoid(z)

  return output
```

~~~
-  <font big color=green>Multi Output Perceptron: </font>
```python
import tensorflow as tf
model = tf.keras.Sequential([tf.keras.layers.Dense(n), tf.keras.layers.Dense(2)])
~~~

- Binary Cross Entropy Loss

```python
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + \big(1 - \underbrace{y^{(i)}}_{\text{Actual}}\big) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

- Mean Squared Error Loss

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}(y^(i) - f(x^(i); W))^2$$

```python
loss = tf.reduce_mean(tf.square(tf.subtract(y, predicted)))
loss = tf.keras.losses.MSE( y, predicted )
```

Gradient Descent

```python
import tensor flow as tf

weights = tf.Variable([tf.random.normal()])

while True:     # loop forever
  with tf.GradientTape( ) as g:
    loss = compute_loss(weights)
    gradient = g.gradient(loss, weights)

  weights = wights -lr *gradient
```

Gradient Descent Algorithm:

- SGD `tf.keras.optimizers.SGD`
- Adam `tf.keras.optimizers.Adam`
- Adadelta `tf.keras.optimizers.Adadelta`
- Adagrad `tf.keras.optimizers.Adagrad`
- RMSProp `tf.keras.optimizers.RMSProp`

  References:

  [1] Stochastic Estimation of the Maximum of a Regression Function

  [2] Adam: A Method for Stochastic Optimization

  [3] ADADELTA: An Adaptive Learning Rate Method

  [4] Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

```
# Putting it all together
import tensorflow as tf
model = tf.keras.Sequential([...])
optimizer = tf.keras.optimizer.SGD()

while True:  # loop forever
  # forward pass through the network
  prediction = model(x)
  with tf.GradientTape() as tape:
    # compute the loss
    loss = compute_loss(y, prediction)
  # update the weights using the Gradient
  grads = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Regularization 1: Dropout

`tf.keras.layer.Dropout(p=05)`

Recurrent Neural Networks and Transformers

`tf.keras.layers.SimpleRNN(run_units)`

```python
class MyRNNCell(tf.keras.layers.Layer):
  def _init_(self, rnn_units, input_dim, output_dim):
    super(MyRNNCell, self)._init_()

    # Initialize weight matrices
    self.W_xh = self.add_weight([rnn_units, input_dim])
    self.W_hh = self.add_weight([rnn_units, runn_units])
    self.W_hy = self.add_weight([output_dim, rnn_units])

    # Initialize hidden state to zeros
    self.h = tf.zeros([rnn_units, 1])

  def call(self, x):
    # Update the hidden state
    self.h = tf.math.tanh( self.W_hh + self.h + self.W_xh * x )

    # Compute the output
    output = self.W_hy * self.h

    # Return the current output and hidden state
    return output, self.h
```

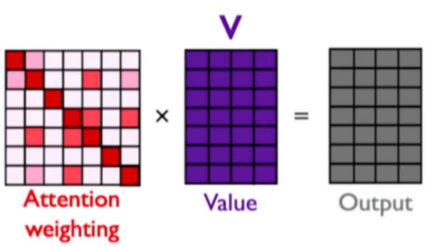
Long Short Term Memory (LSTMs)

1) Forget   2) Store   3) Update   4) Output

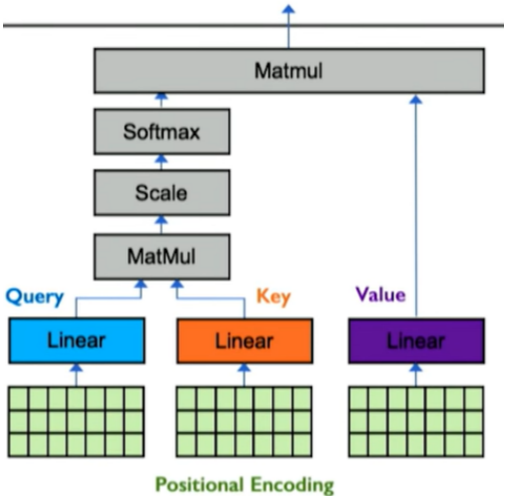
```python
tf.keras.layers.LSTM(num_units)
```

```python
loss = tf.nn.softmax_cross_entropy_with_logits(y, predicted)
```

Self-Attention: (query, key, value)



$$softmax\left(\frac{Q \cdot K^T}{scaling}\right) \cdot V = A(Q,K,V)$$



Language Processing BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Language Models are Few-Shot Learners

Biological Sequences Highly accurate protein structure prediction with AlphaFold

Computer Vision An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Attention is All you Need *

Model Architecture

3.1 Encoder and Decoder Stacks

3.2 Attention

3.2.1 Scaled Dot-Product Attention

3.2.2 Multi-Head Attention

3.2.3 Applications of Attention in our Model

3.3 Position-wise Feed-Forward Networks

3.4 Embeddings and Softmax

3.5 Positional Encoding

Training

5.1 Training Data and Batching

5.2 Hardware and Schedule

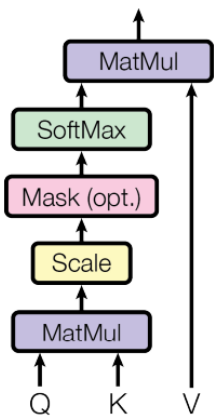
5.3 Optimizer

5.4 Regularization

Residual Dropout

Label Smoothing

Scaled Dot-Product Attention



> From other Vision Transformer(VIT) lectures:
>
> 1.Encoding is important but no need for 2 more dimensions unless they have specific meanings.
>
> 2.One more token for output.
>
> 3.TNT: Transformer in Transformer. -Patch position encoding, -Pixel position encoding.

pytorch implementation - Very detailed explanation of the code in Chinese, the author is still in school. 😄

PDE *