
Load essential libraries

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('dark_background')
%matplotlib inline

import tensorflow as tf
```

Check TensorFlow version

In [ ]:
```
tf.__version__
```

Out[ ]:
```
'2.14.0'
```

Answer the following questions inline using the documentation from:

- Introduction to tensors https://www.tensorflow.org/guide/tensor
- Introduction to variables https://www.tensorflow.org/guide/variable
- Introduction to gradients and automatic differentiation https://www.tensorflow.org/guide/autodiff

---

1. A scalar is a rank $\underbrace{0/1/2}_{\text{choose one}}$ tensor.
2. *True/false*: a scalar has no axes.
3. A matrix is a rank $\underbrace{0/1/2}_{\text{choose one}}$ tensor and has $\underbrace{1/2}_{\text{choose one}}$ axes.
4. What does the function call `tf.reshape(A, [-1]` does for a given tensor `A`?

5. *True/false*: `tf.reshape()` can be used to swap axes of a tensor such as `(patients, timestamps, features)` to `(timestamps, patients, features)`,
6. `tf.keras` uses ______________ to store model parameters.
7. *True/false*: calling `assign` reuses a tensor's exisiting memory to assign the values.
8. *True/false*: creating a new tensor `b` based on the value of another tensor `a` as `b = tf.Variable(a)` will have the tensors allocated different memory.
9. *True/false*: two tensor variables can have the same name.
10. An example of a variable that would not need gradients is a ___________________.
11. Tensor variables are typically placed in $\underbrace{\text{CPU/GPU}}_{\text{choose one}}$.
12. *True/false*: a tensor variable is trainable by default.
13. A gradient tape tape will automatically watch a $\underbrace{\texttt{tf.Variable/tf.constant}}_{\text{choose one}}$ but not a $\underbrace{\texttt{tf.Variable/tf.constant}}_{\text{choose one}}$.
14. What attribute can be used to calculate a layer's gradient w.r.t. all its trainable variables?
15. The option `persistent = True` for a gradient tape $\underbrace{\texttt{stores/discards}}_{\text{choose one}}$ all intermediate results during the forward pass.
16. Executing the statement `print(type(x).__name__)` when `x` is a constant and when `x` is a variable results in what?
17. Which one among `tf.Tensor` and `tf.Variable` is immutable? Which one has no state but only value? Which one has a state which is actually its value?

Answers:

1. Scalar is a rank **0** tensor.
2. True.
3. A matrix is a rank **2** tensor and has 2 axes.
4. Fits the tensor in "whatever good shape" that is possible.
5. False. We use tf..transpose for this operation.
6. tf.keras use tf.Variable to store model parameters.
7. True.

8. True. Two variables will not share the same memory.
9. True.
10. Constant.
11. GPU. For better performance, tensorflow will appempt to place a variable on a faster running device compatible with its dtype. So most variables are placed on a GPU, if available. However, we can override this, and place a float tensor and a variable on a CPU, even if a GPU is available.
12. True. We can turn it off during creation by setting trainable=False.
13. A gradient tape will automatically watch a tf.Variable but not a tf.constant.
14. Module.trainable_variables.
15. stores.
16. If x is a constant, the output will be the data type of x.
    If x is a variable, the output will be the data type of the value stored in that variable at the time of execution.
17. Immutable: tf.Tensor.
    No state but only one value: tf.Tensor.
    Has a state which is actually its value: tf.Variable.

Consider a 1-layer neural network for a sample with 3 features: heart rate, blood pressure, and temperature and 2 possible output categories: diabetic and non-diabetic.

An individual who is diabetic has heart rate = 76 BPM, BP = 120 mm Hg, and temperature = 37.5 $^\circ\mathrm{C}$.

Here is the forward propagation:

$$\mathbf{x} \longrightarrow \mathbf{x}_B = \begin{bmatrix}\mathbf{x}\\ 1\end{bmatrix} \longrightarrow \mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}_B \longrightarrow \underbrace{\mathbf{a}^{[1]}}_{=\hat{\mathbf{y}}} = \mathrm{softmax}\left(\mathbf{a}^{[1]}\right) \longrightarrow L = \sum_{k=0}^{1} -y_k \log(\hat{y}_k).$$

- Fill in the missing entries of the code below to calculate the gradients there-in.
- Explain why some gradient shapes do not seem to match with the usual $\text{input shape} \times \text{output shape}$ rule for gradient shapes using the documentation on Gradients of non-scalar target as resource.
- Try `persistent = true` and `persistent = false` in the gradient tape and observe what happens in each case.

In [ ]:
```
x = tf.constant([76., 120., 37.5], dtype = float)
y = tf.constant([0, 1])
xB = tf.concat([x, 1.0*tf.ones(1)], axis = 0)
W = tf.Variable(0.01*(tf.random.normal((2, 4))))

with tf.GradientTape(persistent = True) as g:
  z = tf.linalg.matvec(W, xB)
  a = tf.nn.softmax(z)
  yhat = a
  L = tf.reduce_sum(-tf.one_hot(indices=y, depth=2))*tf.math.log(yhat)
  L.numpy()

print('Loss = ', L.numpy())
print('Gradient of L w.r.t. yhat')
gradL_yhat = g.gradient(L, yhat)
print(gradL_yhat)
print('-----')
print('Gradient of a w.r.t. z')
grada_z = g.gradient(a, z)
print(grada_z)
print('-----')
print('Gradient of z w.r.t. W')
gradz_W = g.gradient(z, W)
print(gradz_W)
print('-----')
print('Gradient of L w.r.t. W')
gradL_W = g.gradient(L, W)
print(gradL_W)

# Delete gradient tape and release memory
del g
```

```
Loss =  [1.1960657 1.596537 ]
Gradient of L w.r.t. yhat
tf.Tensor([-3.637076  -4.4433813], shape=(2,), dtype=float32)
-----
Gradient of a w.r.t. z
tf.Tensor([0. 0.], shape=(2,), dtype=float32)
-----
Gradient of z w.r.t. W
tf.Tensor(
[[ 76.   120.    37.5    1. ]
 [ 76.   120.    37.5    1. ]], shape=(2, 4), dtype=float32)
-----
Gradient of L w.r.t. W
tf.Tensor(
[[ 15.167256    23.9483       7.483844     0.19956917]
 [-15.167269   -23.94832     -7.48385     -0.19956933]], shape=(2, 4), dtype=float32)
```

In [ ]:
```
x = tf.constant([76., 120., 37.5], dtype = float)
y = tf.constant([0, 1])
xB = tf.concat([x, 1.0*tf.ones(1)], axis = 0)
W = tf.Variable(0.01*(tf.random.normal((2, 4))))

with tf.GradientTape(persistent = False) as g:
  z = tf.linalg.matvec(W, xB)
  a = tf.nn.softmax(z)
  yhat = a
  L = tf.reduce_sum(-tf.one_hot(indices=y, depth=2))*tf.math.log(yhat)
  L.numpy()

print('Loss = ', L.numpy())
print('Gradient of L w.r.t. yhat')
gradL_yhat = g.gradient(L, yhat)
print(gradL_yhat)
print('-----')
print('Gradient of a w.r.t. z')
grada_z = g.gradient(a, z)
print(grada_z)
print('-----')
print('Gradient of z w.r.t. W')
gradz_W = g.gradient(z, W)
print(gradz_W)
```

```
print('-----')
print('Gradient of L w.r.t. W')
gradL_W = g.gradient(L, W)
print(gradL_W)

# Delete gradient tape and release memory
del g
```

```
Loss =  [3.4732852  0.38743818]
Gradient of L w.r.t. yhat
tf.Tensor([-11.356494   -2.4275105], shape=(2,), dtype=float32)
-----
Gradient of a w.r.t. z
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[5], line 19
     17 print('-----')
     18 print('Gradient of a w.r.t. z')
---> 19 grada_z = g.gradient(a, z)
     20 print(grada_z)
     21 print('-----')

File c:\Users\ATISHAY SG\anaconda3\envs\AIMLSem1\lib\site-packages\tensorflow\python\eager\backprop.py:1004, in GradientTape.gradient(self, target, sources, output_gradients, unconnected_gradients)
    964 """Computes the gradient using operations recorded in context of this tape.
    965 
    966 Note: Unless you set `persistent=True` a GradientTape can only be used to
   (...)
   1001   called with an unknown value.
   1002 """
   1003 if self._tape is None:
-> 1004   raise RuntimeError("A non-persistent GradientTape can only be used to "
   1005                      "compute one set of gradients (or jacobians)")
   1006 if self._recording:
   1007   if not self._persistent:

RuntimeError: A non-persistent GradientTape can only be used to compute one set of gradients (or jacobians)
```

From the documentation of Gradient Tape

Recalculate gradients pen-and-paper-way with the same weights from above using numpy. Compare the gradient results here with the ones that you had from the previous cell. Why are some gradients different? In both approaches (in this cell and in the one above), is the gradient of interest $\nabla_{\mathbf{W}^{[1]}}(L)$ the same? Note that this is the only gradient we need to update the weights matrix $\mathbf{W}^{[1]}$.

In [ ]:
```
xB_np = xB.numpy().reshape(-1, 1) # bias-feature added sample vector for numpy
y = np.array([0, 1]) #
z = np.dot(W.numpy(), xB_np) # note we use the same weights from the previous cell here
a = tf.nn.softmax(z, axis = 0).numpy().flatten()
yhat = a
L = tf.reduce_sum(-y*yhat)

print('Loss = %f'%(L))
print('-----')
print('Gradient of L w.r.t. yhat')
gradL_yhat = (-y / yhat)
print(gradL_yhat)
print('-----')
print('Gradient of a w.r.t. z')
grada_z = (np.identity(np.size(z))-z.reshape(-1, z.shape[0]).T) * a.reshape(a.shape[0], 1)
print(grada_z)
print('-----')
print('Gradient of z w.r.t. W')
gradz_W = np.zeros((W.shape[0], W.shape[1], a.shape[0]))
gradz_W[range(W.shape[0]), :, range(a.shape[0])] = xB_np.flatten()
print(gradz_W)
print('-----')
print('Gradient of L w.r.t. W')
gradL_W = np.dot(gradL_yhat, np.dot(gradz_W.transpose(0, 2, 1), grada_z.reshape(1, -1).T)).squeeze()
print(gradL_W)
```

```
Loss = -0.823889
-----
Gradient of L w.r.t. yhat
[ 0.         -1.21375527]
-----
Gradient of a w.r.t. z
[[0.53347806 0.35736741]
 [0.40065506 1.22454437]]
-----
Gradient of z w.r.t. W
[[[ 76.    0. ]
  [120.    0. ]
  [ 37.5   0. ]
  [  1.    0. ]]

 [[  0.   76. ]
  [  0.  120. ]
  [  0.   37.5]
  [  0.    1. ]]]
-----
Gradient of L w.r.t. W
[   0.         -120.98412853]
```

For each activation function below,

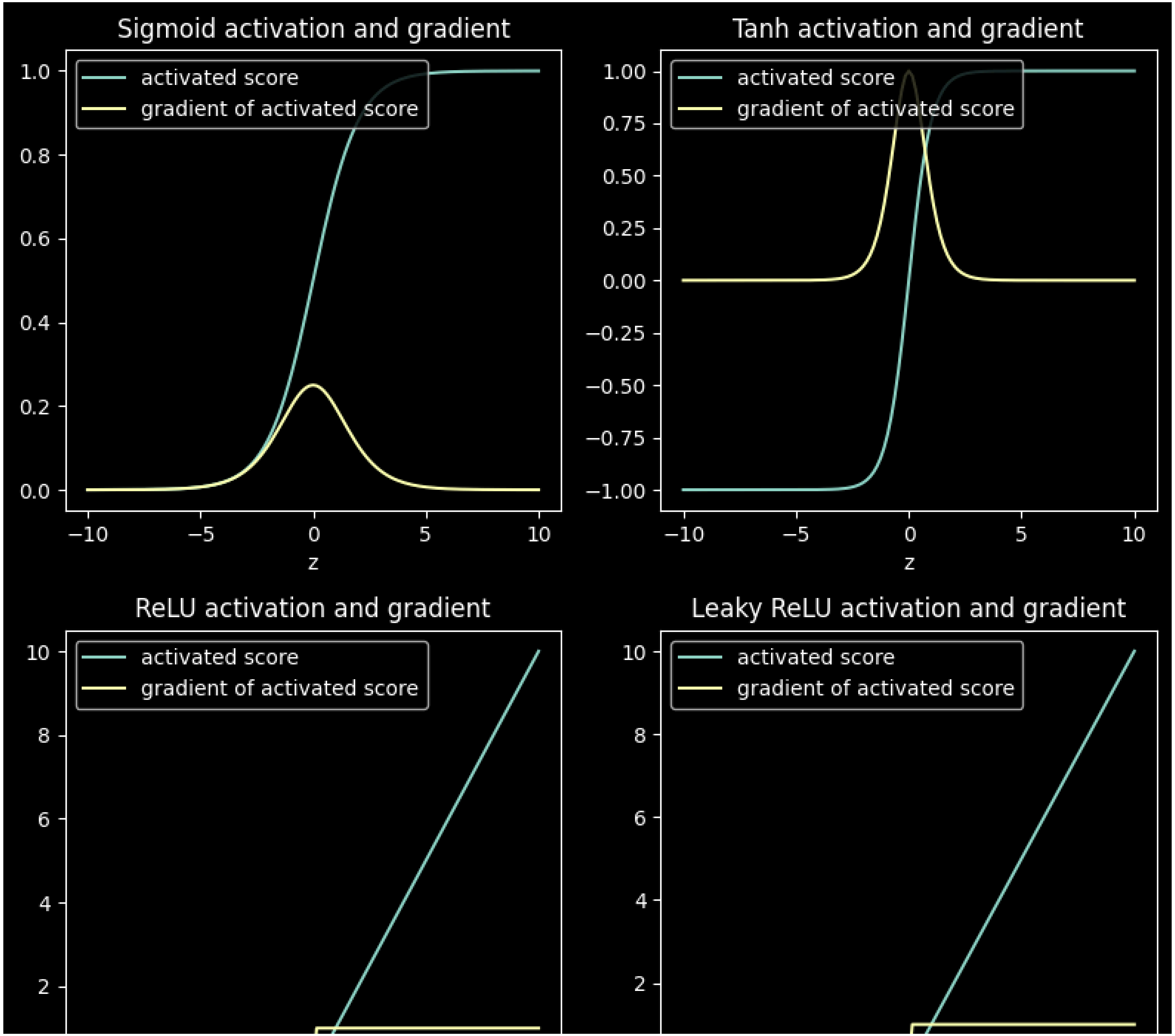
1. Sigmoid $\sigma(z)$
2. Hyperbolic tangent $\tanh(z)$
3. Rectified Linear Unit $\mathrm{ReLU}(z)$
4. Leaky rectified linear unit $\mathrm{LReLU}(z)$

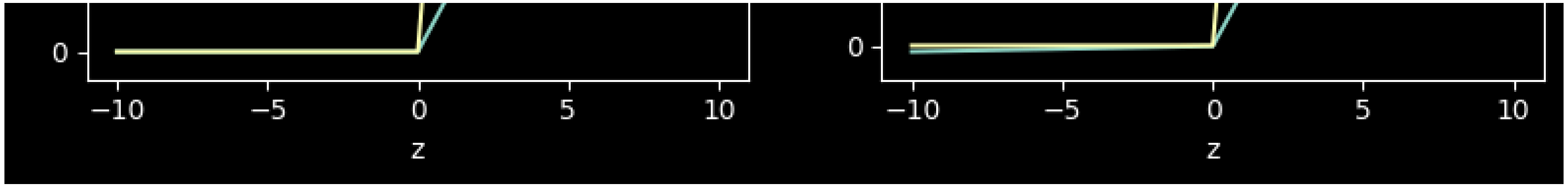
- plot the activation and its gradient in the same figure for raw score values z ranging between -10 and 10;
- comment on whether the backward flowing gradient on the input side of the activation layer will have a smaller or bigger magnitude compared to the backward flowing gradient on the output side of the activation layer. Recall that what connects these two gradients is the local gradient of the activation layer which you may have just plotted.

In [ ]:
```
z = tf.linspace(-10, 10, 129) # A tf.Tensor, not a tf.Variable

with tf.GradientTape(persistent = True) as g:
    g.watch(z)
    g.gradient(L, W)
    a_sigmoid = tf.math.sigmoid(z)
    a_tanh = tf.math.tanh(z)
    # a_ReLU = z * tf.cast((z > 0), tf.float64)
    # a_LReLU = 0 + 0.01*z*tf.cast((z <= 0), tf.float64)
    a_ReLU = tf.maximum(0.0, z)
    a_LReLU = tf.maximum(0.01*z, z)
    # a_ReLU = tf.cast(z, tf.float64) * tf.cast((z > 0), tf.float64)
    # a_LReLU = tf.cast(z, tf.float64) * tf.cast((z > 0.0), tf.float64) + 0.01 * tf.cast(z, tf.float64) * tf.cast((z <= 0.0),



grada_sigmoid_z = g.gradient(a_sigmoid, z)
grada_tanh_z = g.gradient(a_tanh, z)
grada_ReLU_z = g.gradient(a_ReLU, z)
grada_LReLU_z = g.gradient(a_LReLU, z)


fig, axs = plt.subplots(2, 2, figsize = (8, 8))
axs[0, 0].plot(z, a_sigmoid, label = 'activated score')
axs[0, 0].plot(z, grada_sigmoid_z, label='gradient of activated score')
axs[0, 0].legend(loc = 'upper left')
axs[0, 0].set_xlabel('z')
```

```
axs[0, 0].set_title('Sigmoid activation and gradient');

axs[0, 1].plot(z, a_tanh, label = 'activated score')
axs[0, 1].plot(z, grada_tanh_z, label='gradient of activated score')
axs[0, 1].legend(loc = 'upper left')
axs[0, 1].set_xlabel('z')
axs[0, 1].set_title('Tanh activation and gradient');

axs[1, 0].plot(z, a_ReLU, label = 'activated score')
axs[1, 0].plot(z, grada_ReLU_z, label='gradient of activated score')
axs[1, 0].legend(loc = 'upper left')
axs[1, 0].set_xlabel('z')
axs[1, 0].set_title('ReLU activation and gradient');

axs[1, 1].plot(z, a_LReLU, label = 'activated score')
axs[1, 1].plot(z, grada_LReLU_z, label='gradient of activated score')
axs[1, 1].legend(loc = 'upper left')
axs[1, 1].set_xlabel('z')
axs[1, 1].set_title('Leaky ReLU activation and gradient');

plt.tight_layout()
plt.show()
```

```
WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context is significantly less efficient than c
alling it outside the context (it causes the gradient ops to be recorded on the tape, leading to increased CPU and memory usag
e). Only call GradientTape.gradient inside the context if you actually want to trace the gradient in order to compute higher or
der derivatives.
```


Sigmoid activation and gradient
activated score
gradient of activated score
1.0
0.8
0.6
0.4
0.2
0.0
−10
−5
0
5
10
z
Tanh activation and gradient
activated score
gradient of activated score
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−10
−5
0
5
10
z
ReLU activation and gradient
activated score
gradient of activated score
10
8
6
4
2
Leaky ReLU activation and gradient
activated score
gradient of activated score
10
8
6
4
2


0
−10
−5
0
5
10
z
0
−10
−5
0
5
10
z