Neural Network Design and Integration into online, interactive website platform and responsive database administration

Prerequisites: Installed Python newest version

First Steps

1. Install TensorFlow with pip:

   What you need to do in order for all of us to work together is download TensorFlow 2 on this website:

   https://www.tensorflow.org/install/pip#windows-wsl2

For Windows users, the code you will need to enter in your terminal should look something like this:

```
conda install -c conda-forge cudatoolkit=11.8.0
python3 -m pip install nvidia-cudnn-cu11==8.6.0.163 tensorflow==2.13.*
mkdir -p $CONDA_PREFIX/etc/conda/activate.d
echo 'CUDNN_PATH=$(dirname $(python -c "import
nvidia.cudnn;print(nvidia.cudnn.__file__)"))' >>
$CONDA_PREFIX/etc/conda/activate.d/env_vars.sh
echo 'export
LD_LIBRARY_PATH=$CUDNN_PATH/lib:$CONDA_PREFIX/lib/:$LD_LIBRARY_PATH' >>
$CONDA_PREFIX/etc/conda/activate.d/env_vars.sh
source $CONDA_PREFIX/etc/conda/activate.d/env_vars.sh
python3 -c "import tensorflow as tf;
print(tf.config.list_physical_devices('GPU'))"
```

For OSX, the code is as follows:

```
python3 -m pip install tensorflow
python3 -c "import tensorflow as tf;
print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

2. Enter the following code to upgrade to the latest pip:

```
# Requires the latest pip
pip install --upgrade pip

# Current stable release for CPU and GPU
pip install tensorflow

# Or try the preview build (unstable)
pip install tf-nightly
```

3. Run a TensorFlow container – This is important as it sets up a virtual environment for GPU support.

```
 docker pull tensorflow/tensorflow:latest  # Download latest stable image
 docker run -it -p 8888:8888 tensorflow/tensorflow:latest-jupyter  # Start
Jupyter server
```

4. Now, install Keras with the simple command:

```
from tensorflow import keras
```

5. Data processing

Keep in mind this is a basic, suggested version of the Neural Network which we will upgrade as times moves on.

We'll start with a simple neural network-based classifier. It has an input dimension of 2 and a binary output. We'll have a hidden layer with a dimension of 2 and the ReLU activation function. The output layer will have a two-dimensional output with a softmax function in the end. The steps we'll follow in implementing the network will be an SQL-based version of a Python example.

Model

The model has the following parameters:

Input-to-hidden

- W: 2x2 weight matrix (elements: w_00, w_01, w_10, w_11)
- B: 2x1 bias vector (elements: b_0, b_1)

Hidden-to-output

- W2: 2x2 weight matrix (elements: w2_00, w2_01, w2_10, w2_11)
- B2: 2x1 bias vector (elements: b2_0, b2_1)

The training data is stored in a BigQuery table with the columns x1 and x2 having the input and y having the output as shown below (table name: example_project.example_dataset.example_table).

Row	x1	x2	y
1	-1.5241897353240659	0.49462454372528281	0
2	-1.6651448938224989	1.0632741331823559	0
3	-0.55670811597348113	-0.41908549316663984	0
4	-1.6579230061612551	0.50002705734451558	0
5	-1.3267022282765013	-0.85302462596180773	0

Table JSON First < Prev Rows 1 - 5 of 100000

As mentioned earlier, we'll implement the entire training as a single SQL query. The query will return the values of the parameters after the training is completed. As you might have guessed, this will be a heavily nested query. We'll do step-by-step construction to prepare this query. We'll start with the innermost sub-query and then we'll add the nested outer layers one by one.

Forward Pass

Initially, we'll assign random normal values for the weight parameters W and W2 and zero values to the bias parameters B and B2. The random values for W and W2 can be generated within SQL itself. For simplicity, we'll generate these values externally and use within the SQL query. The inner sub-query for initializing the parameters is:

```
SELECT *,
 -0.00569693 AS w_00,
 0.00186517 AS w_01,
 0.00414431 AS w_10,
 0.0105101 AS w_11,
 0.0 AS b_0,
 0.0 AS b_1,
 -0.01312284 AS w2_00,
 -0.01269512 AS w2_01,
 0.00379152 AS w2_10,
 -0.01218354 AS w2_11,
 0.0 AS b2_0,
 0.0 AS b2_1
 FROM `example_project.example_dataset.example_table`
```

Please note that the table `example_project.example_dataset.example_table` already contains the columns x1, x2 and y. The model parameters will be added as additional columns in the result of the above query.

Next, we'll compute the hidden layer. We'll denote the hidden layer with a vector `D` having elements `d0` and `d1`. We'll need to perform the matrix operation: `D = np.maximum(0, np.dot(X, W) + B)` where `X` denotes the input vector (elements `x1` and `x2`). This matrix operation involves firstly multiplying X with the weights in W and then adding the bias vector B. Then the result is passed through the non-linear ReLu activation function which just sets the negative values to 0. The equivalent query in SQL is:

```
SELECT *,
(CASE
WHEN ((x1*w_00 + x2*w_10) + b_0) > 0.0 THEN ((x1*w_00 +
x2*w_10) + b_0)
ELSE 0.0
END) AS d0,
(CASE
```

```
WHEN ((x1*w_01 + x2*w_11) + b_0) > 0.0 THEN ((x1*w_01 +
x2*w_11) + b_1)
ELSE 0.0
END) AS d1
FROM {inner subquery}
```

The above query adds two new columns d0 and d1 to the results of the previous inner sub-query. The output of the above query is shown below.

Row	x1	x2	y	w_00	b2_1	d0	d1
1	-1.5241897353240659	0.4946245437252828	0	-0.005696	0.0	0.010733079671665858	0.0023556804483727068
2	-1.6651448938224989	1.0632741331823559	0	-0.005696	0.0	0.01389275152285318	0.008069339165548969
3	-1.657923006161255	0.5000270573445156	0	-0.005696	0.0	0.011517338445513689	0.0021630261219948046
4	-0.6850105052476166	0.46652714093752384	0	-0.005696	0.0	0.005835889993119094	0.0036255858598947723
5	-1.3053576078644122	0.4088344680433513	0	-0.005696	0.0	0.009130867691227748	0.0018621772931219607

Table JSON First < Prev ≥ Last

This completes the input-to-hidden layer transformation. Now we'll perform the hidden layer-to-output layer transformation.

First, we'll compute the scores for the output layer. The formula is: scores = np.dot(D, W2) + B2. Then we'll apply a softmax function on the scores to obtain the predicted probability of each class. The equivalent inner sub-query in SQL is:

```
SELECT *,
EXP(scores_0)/(EXP(scores_0) + EXP(scores_1)) AS probs_0,
EXP(scores_1)/(EXP(scores_0) + EXP(scores_1)) AS probs_1
FROM
( SELECT *,
((d0*w2_00 + d1*w2_10) + b2_0) AS scores_0,
```

```
((d0*w2_01 + d1*w2_11) + b2_1) AS scores_1
FROM {INNER sub-query})
```

This completes the forward pass of the neural network. Next, we'll do back-propagation to adjust the model parameters based on the comparison of the predicted output (probs) with the expected output (Y).

First we'll compute the aggregate loss resulting from the current prediction. We'll use the cross-entropy loss function to compute the loss. We'll first compute the negative log of the predicted probabilities of the correct class in each example. Cross-entropy loss is nothing but the average of these values across all the instances in X and Y. The natural log is an increasing function. Hence, it is intuitive to define the loss as the negative of the log of the predicted probability of the correct class. If the predicted probability of the correct class is high, loss will be low. Conversely, if the predicted probability of the correct class is low, loss will be high.

To reduce the chance of over-fitting, we'll add L2 regularization too. In the overall loss, we'll include 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2) where reg is a hyper-parameter. Including this function in the loss will penalize high magnitude values in the weight vectors.

In the query, we'll also count the number of training examples (num_examples). This will be useful later when computing averages. The query in SQL to compute the overall loss is:

```
SELECT *,
(sum_correct_logprobs/num_examples) + 1e-3*(0.5*(w_00*w_00 +
w_01*w_01 + w_10*w_10 + w_11*w_11) + 0.5*(w2_00*w2_00 +
w2_01*w2_01 + w2_10*w2_10 + w2_11*w2_11)) AS loss
```

```
FROM
(SELECT *,
SUM(correct_logprobs) OVER () sum_correct_logprobs,
COUNT(1) OVER () num_examples
FROM
(SELECT *,
(CASE
WHEN y = 0 THEN -1*LOG(probs_0)
ELSE -1*LOG(probs_1)
END) AS correct_logprobs
FROM {inner subquery}))
```

Back-propagation

Next, for back-propagation, we'll compute the partial derivatives of each of the parameters w.r.t the loss. We'll use the chain rule to compute this layer by layer starting with the last layer. First, we'll compute the gradients of the scores by using the derivatives of the cross entropy and softmax functions. The query corresponding to this is:

```
SELECT *,
(CASE
WHEN y = 0 THEN (probs_0-1)/num_examples
ELSE probs_0/num_examples
END) AS dscores_0,
(CASE
WHEN y = 1 THEN (probs_1-1)/num_examples
ELSE probs_1/num_examples
```

```
 END) AS dscores_1
FROM {inner subquery}
```

Recall that we computed the scores using scores = np.dot(D, W2) + B2. Hence, based on the derivatives of the scores (termed dscores), we can compute the gradients of the hidden layer D and the model parameters W2 and B2. The corresponding query is: [next page]

```
SELECT *,
SUM(d0*dscores_0) OVER () AS dw2_00,
SUM(d0*dscores_1) OVER () AS dw2_01,
SUM(d1*dscores_0) OVER () AS dw2_10,
SUM(d1*dscores_1) OVER () AS dw2_11,

SUM(dscores_0) OVER () AS db2_0,
SUM(dscores_1) OVER () AS db2_1,

CASE
WHEN (d0) <= 0.0 THEN 0.0
ELSE (dscores_0*w2_00 + dscores_1*w2_01)
END AS dhidden_0,
CASE
WHEN (d1) <= 0.0 THEN 0.0
ELSE (dscores_0*w2_10 + dscores_1*w2_11)
END AS dhidden_1
FROM {inner subquery}
```

Proceeding in a similar way, we know that D = np.maximum(0, np.dot(X, W) + B). Thus by using the derivative of D, we can compute the derivatives of W and B. No point in computing the derivative of X as it is not a model parameter or computed using any model parameter. The query to compute the derivatives of W and B are:

```
SELECT *,
SUM(x1*dhidden_0) OVER () AS dw_00,
SUM(x1*dhidden_1) OVER () AS dw_01,
SUM(x2*dhidden_0) OVER () AS dw_10,
SUM(x2*dhidden_1) OVER () AS dw_11,

SUM(dhidden_0) OVER () AS db_0,
SUM(dhidden_1) OVER () AS db_1
```

```
FROM {inner subquery}
```

Finally, we'll update the model parameters W, B, W2 and B2 using their respective gradients. This can be computed by param -= learning_rate * d_param where learning_rate is a parameter. An additional factor of reg*weight will also be added in dW and dW2 to incorporate L2 regularization in the gradient computation. We'll also remove the temporary columns like dw_00, correct_logprobs etc. which we created in the inner sub-queries and retain only the training data (columns x1, x2 and y) and the model parameters (weights and biases). The corresponding query is:

```
SELECT x1,
x2,
y,
w_00 - (2.0)*(dw_00+(1e-3)*w_00) AS w_00,
w_01 - (2.0)*(dw_01+(1e-3)*w_01) AS w_01,
w_10 - (2.0)*(dw_10+(1e-3)*w_10) AS w_10,
w_11 - (2.0)*(dw_11+(1e-3)*w_11) AS w_11,
b_0 - (2.0)*db_0 AS b_0,
b_1 - (2.0)*db_1 AS b_1,
w2_00 - (2.0)*(dw2_00+(1e-3)*w2_00) AS w2_00,
w2_01 - (2.0)*(dw2_01+(1e-3)*w2_01) AS w2_01,
w2_10 - (2.0)*(dw2_10+(1e-3)*w2_10) AS w2_10,
w2_11 - (2.0)*(dw2_11+(1e-3)*w2_11) AS w2_11,
b2_0 - (2.0)*db2_0 AS b2_0,
b2_1 - (2.0)*db2_1 AS b2_1
FROM {inner subquery}
```

This completes one iteration of forward pass and back-propagation. The above query will provide the updated values of the weights and biases. Sample result is shown below:

Row	x1	x2	y	w_00	w_01
1	-1.5241897353240659	0.4946245437252828	0	-0.005563885461154239	-0.0026534374027
2	-1.6651448938224989	1.0632741331823559	0	-0.005563885461154239	-0.0026534374027
3	-0.5567081159734811	-0.41908549316663984	0	-0.005563885461154239	-0.0026534374027
4	-1.657923006161255	0.5000270573445156	0	-0.005563885461154239	-0.0026534374027
5	-1.3267022282765013	-0.8530246259618077	0	-0.005563885461154239	-0.0026534374027

Table JSON First < Prev Rows 1 - 5 of 100000 Next > Last

To do more training iterations, we'll perform all the above steps recursively. We can do this using a simple Python function. The code is available in link. As we add more iterations, the query gets heavily nested. The resulting query for performing 10 training iterations is available in link.

Because of the heavy nesting and the complexity of the query, I ran into multiple resource limits while trying to execute it in BigQuery. Bigquery's Standard SQL dialect is scaling better compared to its legacy SQL dialect. Even with Standard SQL, for a dataset with 100k instances, it is tough to perform more than 10 iterations. Because of the resource limits, we'll evaluate this model on a simple decision boundary, so that we'll get a decent accuracy with a small number of iterations.

We'll use a simple dataset with inputs x1 and x2 which are sampled from a normal distribution with mean 0 and variance 1. The binary output y simply checks whether x1 + x2 is greater than zero or not. To train faster within 10 iterations, we'll use a high learning rate of 2.0 (Note: such high value is not

recommended in practice as the learning could diverge). Applying the above query with 10 iterations gives the learned model parameters as shown below.

Row	x1	x2	y	w_00	w_01	
1	-1.5241897353240659	0.4946245437252828	0	-0.03415807520631206	-0.02114403423967911	-0
2	-1.6651448938224989	1.0632741331823559	0	-0.03415807520631206	-0.02114403423967911	-0
3	-0.5567081159734811	-0.41908549316663984	0	-0.03415807520631206	-0.02114403423967911	-0
4	-1.657923006161255	0.5000270573445156	0	-0.03415807520631206	-0.02114403423967911	-0
5	-1.3267022282765013	-0.8530246259618077	0	-0.03415807520631206	-0.02114403423967911	-0

Table JSON First < Prev Rows 1 - 5 of 100000 Next > Last

We'll store the result into a new table using Bigquery's 'save to table' functionality. We can now check the accuracy on the training data by performing only the forward pass and then comparing the predicted and expected results. The query snippet for this is in link. We're able to get an accuracy of 93% with 10 iterations (accuracy is similar on a separate test dataset).

```
19      (CASE
20          WHEN ((x1*w_00 + x2*w_10) + b_0) > 0.0 THEN ((x1*w_00 + x2*w_10) + b_0)
21          ELSE 0.0 END) AS d0,
22      (CASE
23          WHEN ((x1*w_01 + x2*w_11) + b_0) > 0.0 THEN ((x1*w_01 + x2*w_11) + b_1)
24          ELSE 0.0 END) AS d1
```

Standard SQL Dialect × Ctrl + Enter:

RUN QUERY | Save Query | Save View | Format Query | Show Options | Query complete (2.2s

Results Details Download as CSV Download as JSON

Row	accuracy_perc
1	93.23

Table JSON

If we can go till ~100 iterations, we'll get an accuracy of more than 99%.

This concludes the project of implementing a deep neural network using pure SQL in BigQuery.

6. That was an introductory creation and implementation of a neural network system. At this moment I would like to point to Amjad and Oscar to ask if they have any suggestions and if they think that this is the model we should start using or if they have another vision of a Neural Network that we can implement.

I think that we should build our own neural network as I have shown above, only using student data and not these sample statistics that I used. Thank you for your time.